```
**********************************************************
    2410 Tue Aug 18 16:13:43 2015
new/usr/src/cmd/devfsadm/fssnap_link.c
6136 sysmacros.h unnecessarily polutes the namespace
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2000-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

#include <regex.h>
#include <devfsadm.h>
#include <stdio.h>
#include <strings.h>
#include <stdlib.h>
#include <limits.h>
#include <sys/fssnap_if.h>
#include <sys/mkdev.h>
#endif /* ! codereview */


static int fssnap(di_minor_t minor, di_node_t node);

static devfsadm_create_t fssnap_cbt[] = {
        { "pseudo", "ddi_pseudo", SNAP_NAME,
            TYPE_EXACT | DRV_EXACT, ILEVEL_0, fssnap,
        },
};

DEVFSADM_CREATE_INIT_V0(fssnap_cbt);

/*
 * For the master device:
 *      /dev/fssnapctl -> /devices/pseudo/fssnap@0:ctl
 * For each other device
 *      /dev/fssnap/1 -> /devices/pseudo/fssnap@0:1
 *      /dev/rfssnap/1 -> /devices/pseudo/fssnap@0:1,raw
 */
static int
fssnap(di_minor_t minor, di_node_t node)
{
        dev_t   dev;
        char mn[MAXNAMELEN + 1];
```



```
        char blkname[MAXNAMELEN + 1];
        char rawname[MAXNAMELEN + 1];
        char path[PATH_MAX + 1];

        (void) strcpy(mn, di_minor_name(minor));

        if (strcmp(mn, "ctl") == 0) {
                (void) devfsadm_mklink(SNAP_CTL_NAME, node, minor, 0);
        } else {
                dev = di_minor_devt(minor);
                (void) snprintf(blkname, sizeof (blkname), "%d",
                    (int)minor(dev));
                (void) snprintf(rawname, sizeof (rawname), "%d,raw",
                    (int)minor(dev));

                if (strcmp(mn, blkname) == 0) {
                        (void) snprintf(path, sizeof (path), "%s/%s",
                            SNAP_BLOCK_NAME, blkname);
                } else if (strcmp(mn, rawname) == 0) {
                        (void) snprintf(path, sizeof (path), "%s/%s",
                            SNAP_CHAR_NAME, blkname);
                } else {
                        return (DEVFSADM_CONTINUE);
                }

                (void) devfsadm_mklink(path, node, minor, 0);
        }
        return (DEVFSADM_CONTINUE);
}
```

```
**********************************************************
  117237 Tue Aug 18 16:13:43 2015
new/usr/src/cmd/fs.d/ufs/fsdb/fsdb.c
6136 sysmacros.h unnecessarily polutes the namespace
**********************************************************
```

```
/*
 * Copyright 2015 Gary Mills
 * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
 */

/*
 * Copyright (c) 1988 Regents of the University of California.
 * All rights reserved.
 *
 * This code is derived from software contributed to Berkeley by
 * Computer Consoles Inc.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that: (1) source distributions retain this entire copyright
 * notice and comment, and (2) distributions including binaries display
 * the following acknowledgement:  ''This product includes software
 * developed by the University of California, Berkeley and its contributors''
 * in the documentation or other materials provided with the distribution
 * and in all advertising materials mentioning features or use of this
 * software. Neither the name of the University nor the names of its
 * contributors may be used to endorse or promote products derived
 * from this software without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ''AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

#ifndef lint
char copyright[] =
"@(#) Copyright(c) 1988 Regents of the University of California.\n\
All rights reserved.\n";
#endif /* not lint */

#ifndef lint
static char sccsid[] = "@(#)fsdb.c      5.8 (Berkeley) 6/1/90";
#endif /* not lint */

/*
 *  fsdb - file system debugger
 *
 *  usage: fsdb [-o suboptions] special
 *  options/suboptions:
 *      -o
 *              ?               display usage
 *              o               override some error conditions
 *              p="string"      set prompt to string
 *              w               open for write
 */

#include <sys/param.h>
#include <sys/signal.h>
#include <sys/file.h>
#include <inttypes.h>
#include <sys/sysmacros.h>
#include <sys/mkdev.h>
#endif /* ! codereview */

#ifdef sun
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
```



```
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/vnode.h>
#include <sys/mntent.h>
#include <sys/wait.h>
#include <sys/fs/ufs_fsdir.h>
#include <sys/fs/ufs_fs.h>
#include <sys/fs/ufs_inode.h>
#include <sys/fs/ufs_acl.h>
#include <sys/fs/ufs_log.h>
#else
#include <sys/dir.h>
#include <ufs/fs.h>
#include <ufs/dinode.h>
#include <paths.h>
#endif /* sun */

#include <stdio.h>
#include <setjmp.h>

#define OLD_FSDB_COMPATIBILITY  /* To support the obsoleted "-z" option */

#ifndef _PATH_BSHELL
#define _PATH_BSHELL    "/bin/sh"
#endif /* _PATH_BSHELL */
/*
 * Defines from the 4.3-tahoe file system, for systems with the 4.2 or 4.3
 * file system.
 */
#ifndef FS_42POSTBLFMT
#define cg_blktot(cgp) (((cgp))->cg_btot)
#define cg_blks(fs, cgp, cylno) (((cgp))->cg_b[cylno])
#define cg_inosused(cgp) (((cgp))->cg_iused)
#define cg_blksfree(cgp) (((cgp))->cg_free)
#define cg_chkmagic(cgp) ((cgp)->cg_magic == CG_MAGIC)
#endif

/*
 * Never changing defines.
 */
#define OCTAL           8               /* octal base */
#define DECIMAL         10              /* decimal base */
#define HEX             16              /* hexadecimal base */

/*
 * Adjustable defines.
 */
#define NBUF            10              /* number of cache buffers */
#define PROMPTSIZE      80              /* size of user definable prompt */
#define MAXFILES        40000           /* max number of files ls can handle */
#define FIRST_DEPTH     10              /* default depth for find and ls */
#define SECOND_DEPTH    100             /* second try at depth (maximum) */
#define INPUTBUFFER     1040            /* size of input buffer */
#define BYTESPERLINE    16              /* bytes per line of /dxo output */
#define NREG            36              /* number of save registers */

#define DEVPREFIX       "/dev/"         /* Uninteresting part of "special" */

#if defined(OLD_FSDB_COMPATIBILITY)
#define FSDB_OPTIONS    "o:wp:z:"
#else
#define FSDB_OPTIONS    "o:wp:"
#endif /* OLD_FSDB_COMPATIBILITY */
```

```
128 /*
129  * Values dependent on sizes of structs and such.
130  */
131 #define NUMB            3                       /* these three are arbitrary, */
132 #define BLOCK           5                       /* but must be different from */
133 #define FRAGMENT        7                       /* the rest (hence odd). */
134 #define BITSPERCHAR     8                       /* couldn't find it anywhere  */
135 #define CHAR            (sizeof (char))
136 #define SHORT           (sizeof (short))
137 #define LONG            (sizeof (long))
138 #define U_OFFSET_T      (sizeof (u_offset_t))   /* essentially "long long" */
139 #define INODE           (sizeof (struct dinode))
140 #define DIRECTORY       (sizeof (struct direct))
141 #define CGRP            (sizeof (struct cg))
142 #define SB              (sizeof (struct fs))
143 #define BLKSIZE         (fs->fs_bsize)          /* for clarity */
144 #define FRGSIZE         (fs->fs_fsize)
145 #define BLKSHIFT        (fs->fs_bshift)
146 #define FRGSHIFT        (fs->fs_fshift)
147 #define SHADOW_DATA     (sizeof (struct ufs_fsd))

149 /*
150  * Messy macros that would otherwise clutter up such glamorous code.
151  */
152 #define itob(i)         (((u_offset_t)itod(fs, (i)) << \
153         (u_offset_t)FRGSHIFT) + (u_offset_t)itoo(fs, (i)) * (u_offset_t)INODE)
154 #define min(x, y)       ((x) < (y) ? (x) : (y))
155 #define STRINGSIZE(d)   ((long)d->d_reclen - \
156                                 ((long)&d->d_name[0] - (long)&d->d_ino))
157 #define letter(c)       ((((c) >= 'a')&&((c) <= 'z')) ||\
158                                 (((c) >= 'A')&&((c) <= 'Z')))
159 #define digit(c)        (((c) >= '0') && ((c) <= '9'))
160 #define HEXLETTER(c)    (((c) >= 'A') && ((c) <= 'F'))
161 #define hexletter(c)    (((c) >= 'a') && ((c) <= 'f'))
162 #define octaldigit(c)   (((c) >= '0') && ((c) <= '7'))
163 #define uppertolower(c) ((c) - 'A' + 'a')
164 #define hextodigit(c)   ((c) - 'a' + 10)
165 #define numtodigit(c)   ((c) - '0')

167 #if !defined(loword)
168 #define loword(X)       (((ushort_t *)&X)[1])
169 #endif /* loword */

171 #if !defined(lobyte)
172 #define lobyte(X)       (((unsigned char *)&X)[1])
173 #endif /* lobyte */

175 /*
176  * buffer cache structure.
177  */
178 static struct lbuf {
179         struct  lbuf  *fwd;
180         struct  lbuf  *back;
181         char    *blkaddr;
182         short   valid;
183         u_offset_t      blkno;
184 } lbuf[NBUF], bhdr;

186 /*
187  * used to hold save registers (see '<' and '>').
188  */
189 struct  save_registers {
190         u_offset_t      sv_addr;
191         u_offset_t      sv_value;
192         long            sv_objsz;
193 } regs[NREG];
```



```
195 /*
196  * cd, find, and ls use this to hold filenames.  Each filename is broken
197  * up by a slash.  In other words, /usr/src/adm would have a len field
198  * of 2 (starting from 0), and filenames->fname[0-2] would hold usr,
199  * src, and adm components of the pathname.
200  */
201 static struct filenames {
202         ino_t   ino;            /* inode */
203         long    len;            /* number of components */
204         char    flag;           /* flag if using SECOND_DEPTH allocator */
205         char    find;           /* flag if found by find */
206         char    **fname;        /* hold components of pathname */
207 } *filenames, *top;

209 enum log_enum { LOG_NDELTAS, LOG_ALLDELTAS, LOG_CHECKSCAN };
210 #ifdef sun
211 struct fs       *fs;
212 static union {
213         struct fs       un_filesystem;
214         char            un_sbsize[SBSIZE];
215 } fs_un;
216 #define filesystem      fs_un.un_filesystem
217 #else
218 struct fs filesystem, *fs;      /* super block */
219 #endif /* sun */

221 /*
222  * Global data.
223  */
224 static char             *input_path[MAXPATHLEN];
225 static char             *stack_path[MAXPATHLEN];
226 static char             *current_path[MAXPATHLEN];
227 static char             input_buffer[INPUTBUFFER];
228 static char             *prompt;
229 static char             *buffers;
230 static char             scratch[64];
231 static char             BASE[] = "o u     x";
232 static char             PROMPT[PROMPTSIZE];
233 static char             laststyle = '/';
234 static char             lastpo = 'x';
235 static short            input_pointer;
236 static short            current_pathp;
237 static short            stack_pathp;
238 static short            input_pathp;
239 static short            cmp_level;
240 static int              nfiles;
241 static short            type = NUMB;
242 static short            dirslot;
243 static short            fd;
244 static short            c_count;
245 static short            error;
246 static short            paren;
247 static short            trapped;
248 static short            doing_cd;
249 static short            doing_find;
250 static short            find_by_name;
251 static short            find_by_inode;
252 static short            long_list;
253 static short            recursive;
254 static short            objsz = SHORT;
255 static short            override = 0;
256 static short            wrtflag = O_RDONLY;
257 static short            base = HEX;
258 static short            acting_on_inode;
259 static short            acting_on_directory;
```

```
260 static short            should_print = 1;
261 static short            clear;
262 static short            star;
263 static u_offset_t       addr;
264 static u_offset_t       bod_addr;
265 static u_offset_t       value;
266 static u_offset_t       erraddr;
267 static long             errcur_bytes;
268 static u_offset_t       errino;
269 static long             errinum;
270 static long             cur_cgrp;
271 static u_offset_t       cur_ino;
272 static long             cur_inum;
273 static u_offset_t       cur_dir;
274 static long             cur_block;
275 static long             cur_bytes;
276 static long             find_ino;
277 static u_offset_t       filesize;
278 static u_offset_t       blocksize;
279 static long             stringsize;
280 static long             count = 1;
281 static long             commands;
282 static long             read_requests;
283 static long             actual_disk_reads;
284 static jmp_buf          env;
285 static long             maxfiles;
286 static long             cur_shad;

288 #ifndef sun
289 extern char     *malloc(), *calloc();
290 #endif
291 static char             getachar();
292 static char             *getblk(), *fmtentry();

294 static offset_t         get(short);
295 static long             bmap();
296 static long             expr();
297 static long             term();
298 static long             getnumb();
299 static u_offset_t       getdirslot();
300 static unsigned long    *print_check(unsigned long *, long *, short, int);

302 static void             usage(char *);
303 static void             ungetachar(char);
304 static void             getnextinput();
305 static void             eat_spaces();
306 static void             restore_inode(ino_t);
307 static void             find();
308 static void             ls(struct filenames *, struct filenames *, short);
309 static void             formatf(struct filenames *, struct filenames *);
310 static void             parse();
311 static void             follow_path(long, long);
312 static void             getname();
313 static void             freemem(struct filenames *, int);
314 static void             print_path(char **, int);
315 static void             fill();
316 static void             put(u_offset_t, short);
317 static void             insert(struct lbuf *);
318 static void             puta();
319 static void             fprnt(char, char);
320 static void             index();
321 #ifdef _LARGEFILE64_SOURCE
322 static void             printll
323         (u_offset_t value, int fieldsz, int digits, int lead);
324 #define print(value, fieldsz, digits, lead) \
325         printll((u_offset_t)value, fieldsz, digits, lead)
```



```
326 #else /* !_LARGEFILE64_SOURCE */
327 static void             print(long value, int fieldsz, int digits, int lead);
328 #endif /* _LARGEFILE64_SOURCE */
329 static void             printsb(struct fs *);
330 static void             printcg(struct cg *);
331 static void             pbits(unsigned char *, int);
332 static void             old_fsdb(int, char *);  /* For old fsdb functionality */

334 static int              isnumber(char *);
335 static int              icheck(u_offset_t);
336 static int              cgrp_check(long);
337 static int              valid_addr();
338 static int              match(char *, int);
339 static int              devcheck(short);
340 static int              bcomp();
341 static int              compare(char *, char *, short);
342 static int              check_addr(short, short *, short *, short);
343 static int              fcmp();
344 static int              ffcmp();

346 static int              getshadowslot(long);
347 static void             getshadowdata(long *, int);
348 static void             syncshadowscan(int);
349 static void             log_display_header(void);
350 static void             log_show(enum log_enum);

352 #ifdef sun
353 static void             err();
354 #else
355 static int              err();
356 #endif /* sun */

358 /* Suboption vector */
359 static char *subopt_v[] = {
360 #define OVERRIDE        0
361         "o",
362 #define NEW_PROMPT      1
363         "p",
364 #define WRITE_ENABLED   2
365         "w",
366 #define ALT_PROMPT      3
367         "prompt",
368         NULL
369 };

371 /*
372  * main - lines are read up to the unprotected ('\') newline and
373  *      held in an input buffer.  Characters may be read from the
374  *      input buffer using getachar() and unread using ungetachar().
375  *      Reading the whole line ahead allows the use of debuggers
376  *      which would otherwise be impossible since the debugger
377  *      and fsdb could not share stdin.
378  */

380 int
381 main(int argc, char *argv[])
382 {

384         char            c, *cptr;
385         short           i;
386         struct direct   *dirp;
387         struct lbuf     *bp;
388         char            *progname;
389         volatile short  colon;
390         short           mode;
391         long            temp;
```

```
393         /* Options/Suboptions processing */
394         int     opt;
395         char    *subopts;
396         char    *optval;
397
398         /*
399          * The following are used to support the old fsdb functionality
400          * of clearing an inode. It's better to use 'clri'.
401          */
402         int                     inum;   /* Inode number to clear */
403         char                    *special;
404
405         setbuf(stdin, NULL);
406         progname = argv[0];
407         prompt = &PROMPT[0];
408         /*
409          * Parse options.
410          */
411         while ((opt = getopt(argc, argv, FSDB_OPTIONS)) != EOF) {
412                 switch (opt) {
413 #if defined(OLD_FSDB_COMPATIBILITY)
414                 case 'z':       /* Hack - Better to use clri */
415                         (void) fprintf(stderr, "%s\n%s\n%s\n%s\n",
416 "Warning: The '-z' option of 'fsdb_ufs' has been declared obsolete",
417 "and may not be supported in a future version of Solaris.",
418 "While this functionality is currently still supported, the",
419 "recommended procedure to clear an inode is to use clri(1M).");
420                         if (isnumber(optarg)) {
421                                 inum = atoi(optarg);
422                                 special = argv[optind];
423                                 /* Doesn't return */
424                                 old_fsdb(inum, special);
425                         } else {
426                                 usage(progname);
427                                 exit(31+1);
428                         }
429                         /* Should exit() before here */
430                         /*NOTREACHED*/
431 #endif /* OLD_FSDB_COMPATIBILITY */
432                 case 'o':
433                         /* UFS Specific Options */
434                         subopts = optarg;
435                         while (*subopts != '\0') {
436                                 switch (getsubopt(&subopts, subopt_v,
437                                                                 &optval)) {
438                                 case OVERRIDE:
439                                         printf("error checking off\n");
440                                         override = 1;
441                                         break;
442
443                                 /*
444                                  * Change the "-o prompt=foo" option to
445                                  * "-o p=foo" to match documentation.
446                                  * ALT_PROMPT continues support for the
447                                  * undocumented "-o prompt=foo" option so
448                                  * that we don't break anyone.
449                                  */
450                                 case NEW_PROMPT:
451                                 case ALT_PROMPT:
452                                         if (optval == NULL) {
453                                                 (void) fprintf(stderr,
454                                                         "No prompt string\n");
455                                                 usage(progname);
456                                         }
457                                         (void) strncpy(PROMPT, optval,
```



```
458                                                                 PROMPTSIZE);
459                                         break;
460
461                                 case WRITE_ENABLED:
462                                         /* suitable for open */
463                                         wrtflag = O_RDWR;
464                                         break;
465
466                                 default:
467                                         usage(progname);
468                                         /* Should exit here */
469                                 }
470                         }
471                         break;
472
473                 default:
474                         usage(progname);
475                 }
476         }
477
478         if ((argc - optind) != 1) {     /* Should just have "special" left */
479                 usage(progname);
480         }
481         special = argv[optind];
482
483         /*
484          * Unless it's already been set, the default prompt includes the
485          * name of the special device.
486          */
487         if (*prompt == NULL)
488                 (void) sprintf(prompt, "%s > ", special);
489
490         /*
491          * Attempt to open the special file.
492          */
493         if ((fd = open(special, wrtflag)) < 0) {
494                 perror(special);
495                 exit(1);
496         }
497         /*
498          * Read in the super block and validate (not too picky).
499          */
500         if (llseek(fd, (offset_t)(SBLOCK * DEV_BSIZE), 0) == -1) {
501                 perror(special);
502                 exit(1);
503         }
504
505 #ifdef sun
506         if (read(fd, &filesystem, SBSIZE) != SBSIZE) {
507                 printf("%s: cannot read superblock\n", special);
508                 exit(1);
509         }
510 #else
511         if (read(fd, &filesystem, sizeof (filesystem)) != sizeof (filesystem)) {
512                 printf("%s: cannot read superblock\n", special);
513                 exit(1);
514         }
515 #endif /* sun */
516
517         fs = &filesystem;
518         if ((fs->fs_magic != FS_MAGIC) && (fs->fs_magic != MTB_UFS_MAGIC)) {
519                 if (!override) {
520                         printf("%s: Bad magic number in file system\n",
521                                                                 special);
522                         exit(1);
523                 }
```

```
525                 printf("WARNING: Bad magic number in file system. ");
526                 printf("Continue? (y/n): ");
527                 (void) fflush(stdout);
528                 if (gets(input_buffer) == NULL) {
529                         exit(1);
530                 }

532                 if (*input_buffer != 'y' && *input_buffer != 'Y') {
533                         exit(1);
534                 }
535         }

537         if ((fs->fs_magic == FS_MAGIC &&
538             (fs->fs_version != UFS_EFISTYLE4NONEFI_VERSION_2 &&
539             fs->fs_version != UFS_VERSION_MIN)) ||
540             (fs->fs_magic == MTB_UFS_MAGIC &&
541             (fs->fs_version > MTB_UFS_VERSION_1 ||
542             fs->fs_version < MTB_UFS_VERSION_MIN))) {
543                 if (!override) {
544                         printf("%s: Unrecognized UFS version number: %d\n",
545                             special, fs->fs_version);
546                         exit(1);
547                 }

549                 printf("WARNING: Unrecognized UFS version number. ");
550                 printf("Continue? (y/n): ");
551                 (void) fflush(stdout);
552                 if (gets(input_buffer) == NULL) {
553                         exit(1);
554                 }

556                 if (*input_buffer != 'y' && *input_buffer != 'Y') {
557                         exit(1);
558                 }
559         }
560 #ifdef FS_42POSTBLFMT
561         if (fs->fs_postblformat == FS_42POSTBLFMT)
562                 fs->fs_nrpos = 8;
563 #endif
564         printf("fsdb of %s %s -- last mounted on %s\n",
565                 special,
566                 (wrtflag == O_RDWR) ? "(Opened for write)" : "(Read only)",
567                 &fs->fs_fsmnt[0]);
568 #ifdef sun
569         printf("fs_clean is currently set to ");
570         switch (fs->fs_clean) {

572         case FSACTIVE:
573                 printf("FSACTIVE\n");
574                 break;
575         case FSCLEAN:
576                 printf("FSCLEAN\n");
577                 break;
578         case FSSTABLE:
579                 printf("FSSTABLE\n");
580                 break;
581         case FSBAD:
582                 printf("FSBAD\n");
583                 break;
584         case FSSUSPEND:
585                 printf("FSSUSPEND\n");
586                 break;
587         case FSLOG:
588                 printf("FSLOG\n");
589                 break;
```



```
590         case FSFIX:
591                 printf("FSFIX\n");
592                 if (!override) {
593                         printf("%s: fsck may be running on this file system\n",
594                                                                 special);
595                         exit(1);
596                 }

598                 printf("WARNING: fsck may be running on this file system. ");
599                 printf("Continue? (y/n): ");
600                 (void) fflush(stdout);
601                 if (gets(input_buffer) == NULL) {
602                         exit(1);
603                 }

605                 if (*input_buffer != 'y' && *input_buffer != 'Y') {
606                         exit(1);
607                 }
608                 break;
609         default:
610                 printf("an unknown value (0x%x)\n", fs->fs_clean);
611                 break;
612         }

614         if (fs->fs_state == (FSOKAY - fs->fs_time)) {
615                 printf("fs_state consistent (fs_clean CAN be trusted)\n");
616         } else {
617                 printf("fs_state inconsistent (fs_clean CAN'T trusted)\n");
618         }
619 #endif /* sun */
620         /*
621          * Malloc buffers and set up cache.
622          */
623         buffers = malloc(NBUF * BLKSIZE);
624         bhdr.fwd = bhdr.back = &bhdr;
625         for (i = 0; i < NBUF; i++) {
626                 bp = &lbuf[i];
627                 bp->blkaddr = buffers + (i * BLKSIZE);
628                 bp->valid = 0;
629                 insert(bp);
630         }
631         /*
632          * Malloc filenames structure.  The space for the actual filenames
633          * is allocated as it needs it. We estimate the size based on the
634          * number of inodes(objects) in the filesystem and the number of
635          * directories.  The number of directories are padded by 3 because
636          * each directory traversed during a "find" or "ls -R" needs 3
637          * entries.
638          */
639         maxfiles = (long)((((u_offset_t)fs->fs_ncg * (u_offset_t)fs->fs_ipg) -
640             (u_offset_t)fs->fs_cstotal.cs_nifree) +
641             ((u_offset_t)fs->fs_cstotal.cs_ndir * (u_offset_t)3));

643         filenames = (struct filenames *)calloc(maxfiles,
644             sizeof (struct filenames));
645         if (filenames == NULL) {
646                 /*
647                  * If we could not allocate memory for all of files
648                  * in the filesystem then, back off to the old fixed
649                  * value.
650                  */
651                 maxfiles = MAXFILES;
652                 filenames = (struct filenames *)calloc(maxfiles,
653                     sizeof (struct filenames));
654                 if (filenames == NULL) {
655                         printf("out of memory\n");
```

```
656                         exit(1);
657                 }
658         }

660         restore_inode(2);
661         /*
662          * Malloc a few filenames (needed by pwd for example).
663          */
664         for (i = 0; i < MAXPATHLEN; i++) {
665                 input_path[i] = calloc(1, MAXNAMLEN);
666                 stack_path[i] = calloc(1, MAXNAMLEN);
667                 current_path[i] = calloc(1, MAXNAMLEN);
668                 if (current_path[i] == NULL) {
669                         printf("out of memory\n");
670                         exit(1);
671                 }
672         }
673         current_pathp = -1;

675         (void) signal(2, err);
676         (void) setjmp(env);

678         getnextinput();
679         /*
680          * Main loop and case statement.  If an error condition occurs
681          * initialization and recovery is attempted.
682          */
683         for (;;) {
684                 if (error) {
685                         freemem(filenames, nfiles);
686                         nfiles = 0;
687                         c_count = 0;
688                         count = 1;
689                         star = 0;
690                         error = 0;
691                         paren = 0;
692                         acting_on_inode = 0;
693                         acting_on_directory = 0;
694                         should_print = 1;
695                         addr = erraddr;
696                         cur_ino = errino;
697                         cur_inum = errinum;
698                         cur_bytes = errcur_bytes;
699                         printf("?\n");
700                         getnextinput();
701                         if (error)
702                                 continue;
703                 }
704                 c_count++;

706                 switch (c = getachar()) {

708                 case '\n': /* command end */
709                         freemem(filenames, nfiles);
710                         nfiles = 0;
711                         if (should_print && laststyle == '=') {
712                                 ungetachar(c);
713                                 goto calc;
714                         }
715                         if (c_count == 1) {
716                                 clear = 0;
717                                 should_print = 1;
718                                 erraddr = addr;
719                                 errino = cur_ino;
720                                 errinum = cur_inum;
721                                 errcur_bytes = cur_bytes;
```



```
722                         switch (objsz) {
723                         case DIRECTORY:
724                                 if ((addr = getdirslot(
725                                                 (long)dirslot+1)) == 0)
726                                         should_print = 0;
727                                 if (error) {
728                                         ungetachar(c);
729                                         continue;
730                                 }
731                                 break;
732                         case INODE:
733                                 cur_inum++;
734                                 addr = itob(cur_inum);
735                                 if (!icheck(addr)) {
736                                         cur_inum--;
737                                         should_print = 0;
738                                 }
739                                 break;
740                         case CGRP:
741                         case SB:
742                                 cur_cgrp++;
743                                 addr = cgrp_check(cur_cgrp);
744                                 if (addr == 0) {
745                                         cur_cgrp--;
746                                         continue;
747                                 }
748                                 break;
749                         case SHADOW_DATA:
750                                 if ((addr = getshadowslot(
751                                     (long)cur_shad + 1)) == 0)
752                                         should_print = 0;
753                                 if (error) {
754                                         ungetachar(c);
755                                         continue;
756                                 }
757                                 break;
758                         default:
759                                 addr += objsz;
760                                 cur_bytes += objsz;
761                                 if (valid_addr() == 0)
762                                         continue;
763                         }
764                 }
765                 if (type == NUMB)
766                         trapped = 0;
767                 if (should_print)
768                         switch (objsz) {
769                         case DIRECTORY:
770                                 fprnt('?', 'd');
771                                 break;
772                         case INODE:
773                                 fprnt('?', 'i');
774                                 if (!error)
775                                         cur_ino = addr;
776                                 break;
777                         case CGRP:
778                                 fprnt('?', 'c');
779                                 break;
780                         case SB:
781                                 fprnt('?', 's');
782                                 break;
783                         case SHADOW_DATA:
784                                 fprnt('?', 'S');
785                                 break;
786                         case CHAR:
787                         case SHORT:
```

```
788                             case LONG:
789                                     fprnt(laststyle, lastpo);
790                             }
791                     if (error) {
792                             ungetachar(c);
793                             continue;
794                     }
795                     c_count = colon = acting_on_inode = 0;
796                     acting_on_directory = 0;
797                     should_print = 1;
798                     getnextinput();
799                     if (error)
800                             continue;
801                     erraddr = addr;
802                     errino = cur_ino;
803                     errinum = cur_inum;
804                     errcur_bytes = cur_bytes;
805                     continue;

807             case '(': /* numeric expression or unknown command */
808             default:
809                     colon = 0;
810                     if (digit(c) || c == '(') {
811                             ungetachar(c);
812                             addr = expr();
813                             type = NUMB;
814                             value = addr;
815                             continue;
816                     }
817                     printf("unknown command or bad syntax\n");
818                     error++;
819                     continue;

821             case '?': /* general print facilities */
822             case '/':
823                     fprnt(c, getachar());
824                     continue;

826             case ';': /* command separator and . */
827             case '\t':
828             case ' ':
829             case '.':
830                     continue;

832             case ':': /* command indicator */
833                     colon++;
834                     commands++;
835                     should_print = 0;
836                     stringsize = 0;
837                     trapped = 0;
838                     continue;

840             case ',': /* count indicator */
841                     colon = star = 0;
842                     if ((c = getachar()) == '*') {
843                             star = 1;
844                             count = BLKSIZE;
845                     } else {
846                             ungetachar(c);
847                             count = expr();
848                             if (error)
849                                     continue;
850                             if (!count)
851                                     count = 1;
852                     }
853                     clear = 0;
```



```
854                     continue;

856             case '+': /* address addition */
857                     colon = 0;
858                     c = getachar();
859                     ungetachar(c);
860                     if (c == '\n')
861                             temp = 1;
862                     else {
863                             temp = expr();
864                             if (error)
865                                     continue;
866                     }
867                     erraddr = addr;
868                     errcur_bytes = cur_bytes;
869                     switch (objsz) {
870                     case DIRECTORY:
871                             addr = getdirslot((long)(dirslot + temp));
872                             if (error)
873                                     continue;
874                             break;
875                     case INODE:
876                             cur_inum += temp;
877                             addr = itob(cur_inum);
878                             if (!icheck(addr)) {
879                                     cur_inum -= temp;
880                                     continue;
881                             }
882                             break;
883                     case CGRP:
884                     case SB:
885                             cur_cgrp += temp;
886                             if ((addr = cgrp_check(cur_cgrp)) == 0) {
887                                     cur_cgrp -= temp;
888                                     continue;
889                             }
890                             break;
891                     case SHADOW_DATA:
892                             addr = getshadowslot((long)(cur_shad + temp));
893                             if (error)
894                                 continue;
895                             break;

897                     default:
898                             laststyle = '/';
899                             addr += temp * objsz;
900                             cur_bytes += temp * objsz;
901                             if (valid_addr() == 0)
902                                     continue;
903                     }
904                     value = get(objsz);
905                     continue;

907             case '-': /* address subtraction */
908                     colon = 0;
909                     c = getachar();
910                     ungetachar(c);
911                     if (c == '\n')
912                             temp = 1;
913                     else {
914                             temp = expr();
915                             if (error)
916                                     continue;
917                     }
918                     erraddr = addr;
919                     errcur_bytes = cur_bytes;
```

```
920                     switch (objsz) {
921                     case DIRECTORY:
922                             addr = getdirslot((long)(dirslot - temp));
923                             if (error)
924                                     continue;
925                             break;
926                     case INODE:
927                             cur_inum -= temp;
928                             addr = itob(cur_inum);
929                             if (!icheck(addr)) {
930                                     cur_inum += temp;
931                                     continue;
932                             }
933                             break;
934                     case CGRP:
935                     case SB:
936                             cur_cgrp -= temp;
937                             if ((addr = cgrp_check(cur_cgrp)) == 0) {
938                                     cur_cgrp += temp;
939                                     continue;
940                             }
941                             break;
942                     case SHADOW_DATA:
943                             addr = getshadowslot((long)(cur_shad - temp));
944                             if (error)
945                                     continue;
946                             break;
947                     default:
948                             laststyle = '/';
949                             addr -= temp * objsz;
950                             cur_bytes -= temp * objsz;
951                             if (valid_addr() == 0)
952                                     continue;
953                     }
954                     value = get(objsz);
955                     continue;

957             case '*': /* address multiplication */
958                     colon = 0;
959                     temp = expr();
960                     if (error)
961                             continue;
962                     if (objsz != INODE && objsz != DIRECTORY)
963                             laststyle = '/';
964                     addr *= temp;
965                     value = get(objsz);
966                     continue;

968             case '%': /* address division */
969                     colon = 0;
970                     temp = expr();
971                     if (error)
972                             continue;
973                     if (!temp) {
974                             printf("divide by zero\n");
975                             error++;
976                             continue;
977                     }
978                     if (objsz != INODE && objsz != DIRECTORY)
979                             laststyle = '/';
980                     addr /= temp;
981                     value = get(objsz);
982                     continue;

984             case '=': { /* assignment operation */
985                     short tbase;
```



```
986 calc:
987                     tbase = base;

989                     c = getachar();
990                     if (c == '\n') {
991                             ungetachar(c);
992                             c = lastpo;
993                             if (acting_on_inode == 1) {
994                                     if (c != 'o' && c != 'd' && c != 'x' &&
995                                         c != 'O' && c != 'D' && c != 'X') {
996                                             switch (objsz) {
997                                             case LONG:
998                                                     c = lastpo = 'X';
999                                                     break;
1000                                            case SHORT:
1001                                                    c = lastpo = 'x';
1002                                                    break;
1003                                            case CHAR:
1004                                                    c = lastpo = 'c';
1005                                            }
1006                                    }
1007                            } else {
1008                                    if (acting_on_inode == 2)
1009                                            c = lastpo = 't';
1010                            }
1011                    } else if (acting_on_inode)
1012                            lastpo = c;
1013                    should_print = star = 0;
1014                    count = 1;
1015                    erraddr = addr;
1016                    errcur_bytes = cur_bytes;
1017                    switch (c) {
1018                    case '"': /* character string */
1019                            if (type == NUMB) {
1020                                    blocksize = BLKSIZE;
1021                                    filesize = BLKSIZE * 2;
1022                                    cur_bytes = blkoff(fs, addr);
1023                                    if (objsz == DIRECTORY ||
1024                                                            objsz == INODE)
1025                                            lastpo = 'X';
1026                            }
1027                            puta();
1028                            continue;
1029                    case '+': /* =+ operator */
1030                            temp = expr();
1031                            value = get(objsz);
1032                            if (!error)
1033                                    put(value+temp, objsz);
1034                            continue;
1035                    case '-': /* =- operator */
1036                            temp = expr();
1037                            value = get(objsz);
1038                            if (!error)
1039                                    put(value-temp, objsz);
1040                            continue;
1041                    case 'b':
1042                    case 'c':
1043                            if (objsz == CGRP)
1044                                    fprnt('?', c);
1045                            else
1046                                    fprnt('/', c);
1047                            continue;
1048                    case 'i':
1049                            addr = cur_ino;
1050                            fprnt('?', 'i');
1051                            continue;
```

```
1052                        case 's':
1053                                fprnt('?', 's');
1054                                continue;
1055                        case 't':
1056                        case 'T':
1057                                laststyle = '=';
1058                                printf("\t\t");
1059                                {
1060                                        /*
1061                                         * Truncation is intentional so
1062                                         * ctime is happy.
1063                                         */
1064                                        time_t tvalue = (time_t)value;
1065                                        printf("%s", ctime(&tvalue));
1066                                }
1067                                continue;
1068                        case 'o':
1069                                base = OCTAL;
1070                                goto otx;
1071                        case 'd':
1072                                if (objsz == DIRECTORY) {
1073                                        addr = cur_dir;
1074                                        fprnt('?', 'd');
1075                                        continue;
1076                                }
1077                                base = DECIMAL;
1078                                goto otx;
1079                        case 'x':
1080                                base = HEX;
1081 otx:
1082                                laststyle = '=';
1083                                printf("\t\t");
1084                                if (acting_on_inode)
1085                                        print(value & 0177777L, 12, -8, 0);
1086                                else
1087                                        print(addr & 0177777L, 12, -8, 0);
1088                                printf("\n");
1089                                base = tbase;
1090                                continue;
1091                        case 'O':
1092                                base = OCTAL;
1093                                goto OTX;
1094                        case 'D':
1095                                base = DECIMAL;
1096                                goto OTX;
1097                        case 'X':
1098                                base = HEX;
1099 OTX:
1100                                laststyle = '=';
1101                                printf("\t\t");
1102                                if (acting_on_inode)
1103                                        print(value, 12, -8, 0);
1104                                else
1105                                        print(addr, 12, -8, 0);
1106                                printf("\n");
1107                                base = tbase;
1108                                continue;
1109                        default: /* regular assignment */
1110                                ungetachar(c);
1111                                value = expr();
1112                                if (error)
1113                                        printf("syntax error\n");
1114                                else
1115                                        put(value, objsz);
1116                                continue;
1117                        }
```



```
1118                }

1120                case '>': /* save current address */
1121                        colon = 0;
1122                        should_print = 0;
1123                        c = getachar();
1124                        if (!letter(c) && !digit(c)) {
1125                                printf("invalid register specification, ");
1126                                printf("must be letter or digit\n");
1127                                error++;
1128                                continue;
1129                        }
1130                        if (letter(c)) {
1131                                if (c < 'a')
1132                                        c = uppertolower(c);
1133                                c = hextodigit(c);
1134                        } else
1135                                c = numtodigit(c);
1136                        regs[c].sv_addr = addr;
1137                        regs[c].sv_value = value;
1138                        regs[c].sv_objsz = objsz;
1139                        continue;

1141                case '<': /* restore saved address */
1142                        colon = 0;
1143                        should_print = 0;
1144                        c = getachar();
1145                        if (!letter(c) && !digit(c)) {
1146                                printf("invalid register specification, ");
1147                                printf("must be letter or digit\n");
1148                                error++;
1149                                continue;
1150                        }
1151                        if (letter(c)) {
1152                                if (c < 'a')
1153                                        c = uppertolower(c);
1154                                c = hextodigit(c);
1155                        } else
1156                                c = numtodigit(c);
1157                        addr = regs[c].sv_addr;
1158                        value = regs[c].sv_value;
1159                        objsz = regs[c].sv_objsz;
1160                        continue;

1162                case 'a':
1163                        if (colon)
1164                                colon = 0;
1165                        else
1166                                goto no_colon;
1167                        if (match("at", 2)) {           /* access time */
1168                                acting_on_inode = 2;
1169                                should_print = 1;
1170                                addr = (long)&((struct dinode *)
1171                                                (uintptr_t)cur_ino)->di_atime;
1172                                value = get(LONG);
1173                                type = NULL;
1174                                continue;
1175                        }
1176                        goto bad_syntax;

1178                case 'b':
1179                        if (colon)
1180                                colon = 0;
1181                        else
1182                                goto no_colon;
1183                        if (match("block", 2)) {        /* block conversion */
```

```
1184                                if (type == NUMB) {
1185                                        value = addr;
1186                                        cur_bytes = 0;
1187                                        blocksize = BLKSIZE;
1188                                        filesize = BLKSIZE * 2;
1189                                }
1190                                addr = value << FRGSHIFT;
1191                                bod_addr = addr;
1192                                value = get(LONG);
1193                                type = BLOCK;
1194                                dirslot = 0;
1195                                trapped++;
1196                                continue;
1197                        }
1198                        if (match("bs", 2)) {           /* block size */
1199                                acting_on_inode = 1;
1200                                should_print = 1;
1201                                if (icheck(cur_ino) == 0)
1202                                        continue;
1203                                addr = (long)&((struct dinode *)
1204                                        (uintptr_t)cur_ino)->di_blocks;
1205                                value = get(LONG);
1206                                type = NULL;
1207                                continue;
1208                        }
1209                        if (match("base", 2)) {         /* change/show base */
1210 showbase:
1211                                if ((c = getachar()) == '\n') {
1212                                        ungetachar(c);
1213                                        printf("base =\t\t");
1214                                        switch (base) {
1215                                        case OCTAL:
1216                                                printf("OCTAL\n");
1217                                                continue;
1218                                        case DECIMAL:
1219                                                printf("DECIMAL\n");
1220                                                continue;
1221                                        case HEX:
1222                                                printf("HEX\n");
1223                                                continue;
1224                                        }
1225                                }
1226                                if (c != '=') {
1227                                        printf("missing '='\n");
1228                                        error++;
1229                                        continue;
1230                                }
1231                                value = expr();
1232                                switch (value) {
1233                                default:
1234                                        printf("invalid base\n");
1235                                        error++;
1236                                        break;
1237                                case OCTAL:
1238                                case DECIMAL:
1239                                case HEX:
1240                                        base = (short)value;
1241                                }
1242                                goto showbase;
1243                        }
1244                        goto bad_syntax;

1246                case 'c':
1247                        if (colon)
1248                                colon = 0;
1249                        else
```



```
1250                                goto no_colon;
1251                        if (match("cd", 2)) {           /* change directory */
1252                                top = filenames - 1;
1253                                eat_spaces();
1254                                if ((c = getachar()) == '\n') {
1255                                        ungetachar(c);
1256                                        current_pathp = -1;
1257                                        restore_inode(2);
1258                                        continue;
1259                                }
1260                                ungetachar(c);
1261                                temp = cur_inum;
1262                                doing_cd = 1;
1263                                parse();
1264                                doing_cd = 0;
1265                                if (nfiles != 1) {
1266                                        restore_inode((ino_t)temp);
1267                                        if (!error) {
1268                                                print_path(input_path,
1269                                                        (int)input_pathp);
1270                                                if (nfiles == 0)
1271                                                        printf(" not found\n");
1272                                                else
1273                                                        printf(" ambiguous\n");
1274                                                error++;
1275                                        }
1276                                        continue;
1277                                }
1278                                restore_inode(filenames->ino);
1279                                if ((mode = icheck(addr)) == 0)
1280                                        continue;
1281                                if ((mode & IFMT) != IFDIR) {
1282                                        restore_inode((ino_t)temp);
1283                                        print_path(input_path,
1284                                                        (int)input_pathp);
1285                                        printf(" not a directory\n");
1286                                        error++;
1287                                        continue;
1288                                }
1289                                for (i = 0; i <= top->len; i++)
1290                                        (void) strcpy(current_path[i],
1291                                                top->fname[i]);
1292                                current_pathp = top->len;
1293                                continue;
1294                        }
1295                        if (match("cg", 2)) {           /* cylinder group */
1296                                if (type == NUMB)
1297                                        value = addr;
1298                                if (value > fs->fs_ncg - 1) {
1299                                        printf("maximum cylinder group is ");
1300                                        print(fs->fs_ncg - 1, 8, -8, 0);
1301                                        printf("\n");
1302                                        error++;
1303                                        continue;
1304                                }
1305                                type = objsz = CGRP;
1306                                cur_cgrp = (long)value;
1307                                addr = cgtod(fs, cur_cgrp) << FRGSHIFT;
1308                                continue;
1309                        }
1310                        if (match("ct", 2)) {           /* creation time */
1311                                acting_on_inode = 2;
1312                                should_print = 1;
1313                                addr = (long)&((struct dinode *)
1314                                        (uintptr_t)cur_ino)->di_ctime;
1315                                value = get(LONG);
```

```
1316                                    type = NULL;
1317                                    continue;
1318                            }
1319                            goto bad_syntax;

1321                    case 'd':
1322                            if (colon)
1323                                    colon = 0;
1324                            else
1325                                    goto no_colon;
1326                            if (match("directory", 2)) {    /* directory offsets */
1327                                    if (type == NUMB)
1328                                            value = addr;
1329                                    objsz = DIRECTORY;
1330                                    type = DIRECTORY;
1331                                    addr = (u_offset_t)getdirslot((long)value);
1332                                    continue;
1333                            }
1334                            if (match("db", 2)) {           /* direct block */
1335                                    acting_on_inode = 1;
1336                                    should_print = 1;
1337                                    if (type == NUMB)
1338                                            value = addr;
1339                                    if (value >= NDADDR) {
1340                                            printf("direct blocks are 0 to ");
1341                                            print(NDADDR - 1, 0, 0, 0);
1342                                            printf("\n");
1343                                            error++;
1344                                            continue;
1345                                    }
1346                                    addr = cur_ino;
1347                                    if (!icheck(addr))
1348                                            continue;
1349                                    addr = (long)
1350                                            &((struct dinode *)(uintptr_t)cur_ino)->
1351                                                                    di_db[value];
1352                                    bod_addr = addr;
1353                                    cur_bytes = (value) * BLKSIZE;
1354                                    cur_block = (long)value;
1355                                    type = BLOCK;
1356                                    dirslot = 0;
1357                                    value = get(LONG);
1358                                    if (!value && !override) {
1359                                            printf("non existent block\n");
1360                                            error++;
1361                                    }
1362                                    continue;
1363                            }
1364                            goto bad_syntax;

1366                    case 'f':
1367                            if (colon)
1368                                    colon = 0;
1369                            else
1370                                    goto no_colon;
1371                            if (match("find", 3)) {         /* find command */
1372                                    find();
1373                                    continue;
1374                            }
1375                            if (match("fragment", 2)) {     /* fragment conv. */
1376                                    if (type == NUMB) {
1377                                            value = addr;
1378                                            cur_bytes = 0;
1379                                            blocksize = FRGSIZE;
1380                                            filesize = FRGSIZE * 2;
1381                                    }
```



```
1382                                    if (min(blocksize, filesize) - cur_bytes >
1383                                                            FRGSIZE) {
1384                                            blocksize = cur_bytes + FRGSIZE;
1385                                            filesize = blocksize * 2;
1386                                    }
1387                                    addr = value << FRGSHIFT;
1388                                    bod_addr = addr;
1389                                    value = get(LONG);
1390                                    type = FRAGMENT;
1391                                    dirslot = 0;
1392                                    trapped++;
1393                                    continue;
1394                            }
1395                            if (match("file", 4)) {         /* access as file */
1396                                    acting_on_inode = 1;
1397                                    should_print = 1;
1398                                    if (type == NUMB)
1399                                            value = addr;
1400                                    addr = cur_ino;
1401                                    if ((mode = icheck(addr)) == 0)
1402                                            continue;
1403                                    if (!override) {
1404                                            switch (mode & IFMT) {
1405                                            case IFCHR:
1406                                            case IFBLK:
1407                                                printf("special device\n");
1408                                                error++;
1409                                                continue;
1410                                            }
1411                                    }
1412                                    if ((addr = (u_offset_t)
1413                                        (bmap((long)value) << FRGSHIFT)) == 0)
1414                                            continue;
1415                                    cur_block = (long)value;
1416                                    bod_addr = addr;
1417                                    type = BLOCK;
1418                                    dirslot = 0;
1419                                    continue;
1420                            }
1421                            if (match("fill", 4)) {         /* fill */
1422                                    if (getachar() != '=') {
1423                                            printf("missing '='\n");
1424                                            error++;
1425                                            continue;
1426                                    }
1427                                    if (objsz == INODE || objsz == DIRECTORY ||
1428                                        objsz == SHADOW_DATA) {
1429                                            printf(
1430                                                "can't fill inode or directory\n");
1431                                            error++;
1432                                            continue;
1433                                    }
1434                                    fill();
1435                                    continue;
1436                            }
1437                            goto bad_syntax;

1439                    case 'g':
1440                            if (colon)
1441                                    colon = 0;
1442                            else
1443                                    goto no_colon;
1444                            if (match("gid", 1)) {          /* group id */
1445                                    acting_on_inode = 1;
1446                                    should_print = 1;
1447                                    addr = (long)&((struct dinode *)
```

```
1448                                            (uintptr_t)cur_ino)->di_gid;
1449                                    value = get(SHORT);
1450                                    type = NULL;
1451                                    continue;
1452                            }
1453                            goto bad_syntax;

1455                    case 'i':
1456                            if (colon)
1457                                    colon = 0;
1458                            else
1459                                    goto no_colon;
1460                            if (match("inode", 2)) { /* i# to inode conversion */
1461                                    if (c_count == 2) {
1462                                            addr = cur_ino;
1463                                            value = get(INODE);
1464                                            type = NULL;
1465                                            laststyle = '=';
1466                                            lastpo = 'i';
1467                                            should_print = 1;
1468                                            continue;
1469                                    }
1470                                    if (type == NUMB)
1471                                            value = addr;
1472                                    addr = itob(value);
1473                                    if (!icheck(addr))
1474                                            continue;
1475                                    cur_ino = addr;
1476                                    cur_inum = (long)value;
1477                                    value = get(INODE);
1478                                    type = NULL;
1479                                    continue;
1480                            }
1481                            if (match("ib", 2)) {   /* indirect block */
1482                                    acting_on_inode = 1;
1483                                    should_print = 1;
1484                                    if (type == NUMB)
1485                                            value = addr;
1486                                    if (value >= NIADDR) {
1487                                            printf("indirect blocks are 0 to ");
1488                                            print(NIADDR - 1, 0, 0, 0);
1489                                            printf("\n");
1490                                            error++;
1491                                            continue;
1492                                    }
1493                                    addr = (long)&((struct dinode *)(uintptr_t)
1494                                                    cur_ino)->di_ib[value];
1495                                    cur_bytes = (NDADDR - 1) * BLKSIZE;
1496                                    temp = 1;
1497                                    for (i = 0; i < value; i++) {
1498                                            temp *= NINDIR(fs) * BLKSIZE;
1499                                            cur_bytes += temp;
1500                                    }
1501                                    type = BLOCK;
1502                                    dirslot = 0;
1503                                    value = get(LONG);
1504                                    if (!value && !override) {
1505                                            printf("non existent block\n");
1506                                            error++;
1507                                    }
1508                                    continue;
1509                            }
1510                            goto bad_syntax;

1512                    case 'l':
1513                            if (colon)
```



```
1514                                    colon = 0;
1515                            else
1516                                    goto no_colon;
1517                            if (match("log_head", 8)) {
1518                                    log_display_header();
1519                                    should_print = 0;
1520                                    continue;
1521                            }
1522                            if (match("log_delta", 9)) {
1523                                    log_show(LOG_NDELTAS);
1524                                    should_print = 0;
1525                                    continue;
1526                            }
1527                            if (match("log_show", 8)) {
1528                                    log_show(LOG_ALLDELTAS);
1529                                    should_print = 0;
1530                                    continue;
1531                            }
1532                            if (match("log_chk", 7)) {
1533                                    log_show(LOG_CHECKSCAN);
1534                                    should_print = 0;
1535                                    continue;
1536                            }
1537                            if (match("log_otodb", 9)) {
1538                                    if (log_lodb((u_offset_t)addr, &temp)) {
1539                                            addr = temp;
1540                                            should_print = 1;
1541                                            laststyle = '=';
1542                                    } else
1543                                            error++;
1544                                    continue;
1545                            }
1546                            if (match("ls", 2)) {           /* ls command */
1547                                    temp = cur_inum;
1548                                    recursive = long_list = 0;
1549                                    top = filenames - 1;
1550                                    for (;;) {
1551                                            eat_spaces();
1552                                            if ((c = getachar()) == '-') {
1553                                                    if ((c = getachar()) == 'R') {
1554                                                            recursive = 1;
1555                                                            continue;
1556                                                    } else if (c == 'l') {
1557                                                            long_list = 1;
1558                                                    } else {
1559                                                            printf(
1560                                                                "unknown option ");
1561                                                            printf("'%c'\n", c);
1562                                                            error++;
1563                                                            break;
1564                                                    }
1565                                            } else
1566                                                    ungetachar(c);
1567                                            if ((c = getachar()) == '\n') {
1568                                                    if (c_count != 2) {
1569                                                            ungetachar(c);
1570                                                            break;
1571                                                    }
1572                                            }
1573                                            c_count++;
1574                                            ungetachar(c);
1575                                            parse();
1576                                            restore_inode((ino_t)temp);
1577                                            if (error)
1578                                                    break;
1579                                    }
```

```
1580                                    recursive = 0;
1581                                    if (error || nfiles == 0) {
1582                                            if (!error) {
1583                                                    print_path(input_path,
1584                                                            (int)input_pathp);
1585                                                    printf(" not found\n");
1586                                            }
1587                                            continue;
1588                                    }
1589                                    if (nfiles) {
1590                                        cmp_level = 0;
1591                                        qsort((char *)filenames, nfiles,
1592                                            sizeof (struct filenames), ffcmp);
1593                                        ls(filenames, filenames + (nfiles - 1), 0);
1594                                    } else {
1595                                        printf("no match\n");
1596                                        error++;
1597                                    }
1598                                    restore_inode((ino_t)temp);
1599                                    continue;
1600                            }
1601                            if (match("ln", 2)) {           /* link count */
1602                                    acting_on_inode = 1;
1603                                    should_print = 1;
1604                                    addr = (long)&((struct dinode *)
1605                                                    (uintptr_t)cur_ino)->di_nlink;
1606                                    value = get(SHORT);
1607                                    type = NULL;
1608                                    continue;
1609                            }
1610                            goto bad_syntax;

1612                    case 'm':
1613                            if (colon)
1614                                    colon = 0;
1615                            else
1616                                    goto no_colon;
1617                            addr = cur_ino;
1618                            if ((mode = icheck(addr)) == 0)
1619                                    continue;
1620                            if (match("mt", 2)) {           /* modification time */
1621                                    acting_on_inode = 2;
1622                                    should_print = 1;
1623                                    addr = (long)&((struct dinode *)
1624                                                    (uintptr_t)cur_ino)->di_mtime;
1625                                    value = get(LONG);
1626                                    type = NULL;
1627                                    continue;
1628                            }
1629                            if (match("md", 2)) {           /* mode */
1630                                    acting_on_inode = 1;
1631                                    should_print = 1;
1632                                    addr = (long)&((struct dinode *)
1633                                                    (uintptr_t)cur_ino)->di_mode;
1634                                    value = get(SHORT);
1635                                    type = NULL;
1636                                    continue;
1637                            }
1638                            if (match("maj", 2)) {  /* major device number */
1639                                    acting_on_inode = 1;
1640                                    should_print = 1;
1641                                    if (devcheck(mode))
1642                                            continue;
1643                                    addr = (uintptr_t)&((struct dinode *)(uintptr_t)
1644                                                    cur_ino)->di_ordev;
1645                                    {
```



```
1646                                            long    dvalue;
1647                                            dvalue = get(LONG);
1648                                            value = major(dvalue);
1649                                    }
1650                                    type = NULL;
1651                                    continue;
1652                            }
1653                            if (match("min", 2)) {  /* minor device number */
1654                                    acting_on_inode = 1;
1655                                    should_print = 1;
1656                                    if (devcheck(mode))
1657                                            continue;
1658                                    addr = (uintptr_t)&((struct dinode *)(uintptr_t)
1659                                                            cur_ino)->di_ordev;
1660                                    {
1661                                            long    dvalue;
1662                                            dvalue = (long)get(LONG);
1663                                            value = minor(dvalue);
1664                                    }
1665                                    type = NULL;
1666                                    continue;
1667                            }
1668                            goto bad_syntax;

1670                    case 'n':
1671                            if (colon)
1672                                    colon = 0;
1673                            else
1674                                    goto no_colon;
1675                            if (match("nm", 1)) {           /* directory name */
1676                                    objsz = DIRECTORY;
1677                                    acting_on_directory = 1;
1678                                    cur_dir = addr;
1679                                    if ((cptr = getblk(addr)) == 0)
1680                                            continue;
1681                                    /*LINTED*/
1682                                    dirp = (struct direct *)(cptr+blkoff(fs, addr));
1683                                    stringsize = (long)dirp->d_reclen -
1684                                            ((long)&dirp->d_name[0] -
1685                                                    (long)&dirp->d_ino);
1686                                    addr = (long)&((struct direct *)
1687                                                    (uintptr_t)addr)->d_name[0];
1688                                    type = NULL;
1689                                    continue;
1690                            }
1691                            goto bad_syntax;

1693                    case 'o':
1694                            if (colon)
1695                                    colon = 0;
1696                            else
1697                                    goto no_colon;
1698                            if (match("override", 1)) {     /* override flip flop */
1699                                    override = !override;
1700                                    if (override)
1701                                            printf("error checking off\n");
1702                                    else
1703                                            printf("error checking on\n");
1704                                    continue;
1705                            }
1706                            goto bad_syntax;

1708                    case 'p':
1709                            if (colon)
1710                                    colon = 0;
1711                            else
```

```
1712                            goto no_colon;
1713                    if (match("pwd", 2)) {          /* print working dir */
1714                            print_path(current_path, (int)current_pathp);
1715                            printf("\n");
1716                            continue;
1717                    }
1718                    if (match("prompt", 2)) {       /* change prompt */
1719                            if ((c = getachar()) != '=') {
1720                                    printf("missing '='\n");
1721                                    error++;
1722                                    continue;
1723                            }
1724                            if ((c = getachar()) != '"') {
1725                                    printf("missing '\"'\n");
1726                                    error++;
1727                                    continue;
1728                            }
1729                            i = 0;
1730                            prompt = &prompt[0];
1731                            while ((c = getachar()) != '"' && c != '\n') {
1732                                    prompt[i++] = c;
1733                                    if (i >= PROMPTSIZE) {
1734                                            printf("string too long\n");
1735                                            error++;
1736                                            break;
1737                                    }
1738                            }
1739                            prompt[i] = '\0';
1740                            continue;
1741                    }
1742                    goto bad_syntax;

1744            case 'q':
1745                    if (!colon)
1746                            goto no_colon;
1747                    if (match("quit", 1)) {         /* quit */
1748                            if ((c = getachar()) != '\n') {
1749                                    error++;
1750                                    continue;
1751                            }
1752                            exit(0);
1753                    }
1754                    goto bad_syntax;

1756            case 's':
1757                    if (colon)
1758                            colon = 0;
1759                    else
1760                            goto no_colon;
1761                    if (match("sb", 2)) {           /* super block */
1762                            if (c_count == 2) {
1763                                    cur_cgrp = -1;
1764                                    type = objsz = SB;
1765                                    laststyle = '=';
1766                                    lastpo = 's';
1767                                    should_print = 1;
1768                                    continue;
1769                            }
1770                            if (type == NUMB)
1771                                    value = addr;
1772                            if (value > fs->fs_ncg - 1) {
1773                                    printf("maximum super block is ");
1774                                    print(fs->fs_ncg - 1, 8, -8, 0);
1775                                    printf("\n");
1776                                    error++;
1777                                    continue;
```



```
1778                            }
1779                            type = objsz = SB;
1780                            cur_cgrp = (long)value;
1781                            addr = cgsblock(fs, cur_cgrp) << FRGSHIFT;
1782                            continue;
1783                    }
1784                    if (match("shadow", 2)) {       /* shadow inode data */
1785                            if (type == NUMB)
1786                                    value = addr;
1787                            objsz = SHADOW_DATA;
1788                            type = SHADOW_DATA;
1789                            addr = getshadowslot(value);
1790                            continue;
1791                    }
1792                    if (match("si", 2)) {   /* shadow inode field */
1793                            acting_on_inode = 1;
1794                            should_print = 1;
1795                            addr = (long)&((struct dinode *)
1796                                    (uintptr_t)cur_ino)->di_shadow;
1797                            value = get(LONG);
1798                            type = NULL;
1799                            continue;
1800                    }

1802                    if (match("sz", 2)) {           /* file size */
1803                            acting_on_inode = 1;
1804                            should_print = 1;
1805                            addr = (long)&((struct dinode *)
1806                                    (uintptr_t)cur_ino)->di_size;
1807                            value = get(U_OFFSET_T);
1808                            type = NULL;
1809                            objsz = U_OFFSET_T;
1810                            laststyle = '=';
1811                            lastpo = 'X';
1812                            continue;
1813                    }
1814                    goto bad_syntax;

1816            case 'u':
1817                    if (colon)
1818                            colon = 0;
1819                    else
1820                            goto no_colon;
1821                    if (match("uid", 1)) {          /* user id */
1822                            acting_on_inode = 1;
1823                            should_print = 1;
1824                            addr = (long)&((struct dinode *)
1825                                    (uintptr_t)cur_ino)->di_uid;
1826                            value = get(SHORT);
1827                            type = NULL;
1828                            continue;
1829                    }
1830                    goto bad_syntax;

1832            case 'F': /* buffer status (internal use only) */
1833                    if (colon)
1834                            colon = 0;
1835                    else
1836                            goto no_colon;
1837                    for (bp = bhdr.fwd; bp != &bhdr; bp = bp->fwd)
1838                            printf("%8" PRIx64 " %d\n",
1839                                bp->blkno, bp->valid);
1840                    printf("\n");
1841                    printf("# commands\t\t%ld\n", commands);
1842                    printf("# read requests\t\t%ld\n", read_requests);
1843                    printf("# actual disk reads\t%ld\n", actual_disk_reads);
```

```
1844                        continue;
1845 no_colon:
1846                printf("a colon should precede a command\n");
1847                error++;
1848                continue;
1849 bad_syntax:
1850                printf("more letters needed to distinguish command\n");
1851                error++;
1852                continue;
1853                }
1854        }
1855 }

1857 /*
1858  * usage - print usage and exit
1859  */
1860 static void
1861 usage(char *progname)
1862 {
1863        printf("usage:   %s [options] special\n", progname);
1864        printf("options:\n");
1865        printf("\t-o             Specify ufs filesystem sepcific options\n");
1866        printf("                Available suboptions are:\n");
1867        printf("\t\t?           display usage\n");
1868        printf("\t\to           override some error conditions\n");
1869        printf("\t\tp=\"string\"        set prompt to string\n");
1870        printf("\t\tw           open for write\n");
1871        exit(1);
1872 }

1874 /*
1875  * getachar - get next character from input buffer.
1876  */
1877 static char
1878 getachar()
1879 {
1880        return (input_buffer[input_pointer++]);
1881 }

1883 /*
1884  * ungetachar - return character to input buffer.
1885  */
1886 static void
1887 ungetachar(char c)
1888 {
1889        if (input_pointer == 0) {
1890                printf("internal problem maintaining input buffer\n");
1891                error++;
1892                return;
1893        }
1894        input_buffer[--input_pointer] = c;
1895 }

1897 /*
1898  * getnextinput - display the prompt and read an input line.
1899  *      An input line is up to 128 characters terminated by the newline
1900  *      character.  Handle overflow, shell escape, and eof.
1901  */
1902 static void
1903 getnextinput()
1904 {
1905        int     i;
1906        char    c;
1907        short   pid, rpid;
1908        int     retcode;
```



```
1910 newline:
1911        i = 0;
1912        printf("%s", prompt);
1913 ignore_eol:
1914        while ((c = getc(stdin)) != '\n' && !(c == '!' && i == 0) &&
1915                                        !feof(stdin) && i <= INPUTBUFFER - 2)
1916                input_buffer[i++] = c;
1917        if (i > 0 && input_buffer[i - 1] == '\\') {
1918                input_buffer[i++] = c;
1919                goto ignore_eol;
1920        }
1921        if (feof(stdin)) {
1922                printf("\n");
1923                exit(0);
1924        }
1925        if (c == '!') {
1926                if ((pid = fork()) == 0) {
1927                        (void) execl(_PATH_BSHELL, "sh", "-t", 0);
1928                        error++;
1929                        return;
1930                }
1931                while ((rpid = wait(&retcode)) != pid && rpid != -1)
1932                        ;
1933                printf("!\n");
1934                goto newline;
1935        }
1936        if (c != '\n')
1937                printf("input truncated to 128 characters\n");
1938        input_buffer[i] = '\n';
1939        input_pointer = 0;
1940 }

1942 /*
1943  * eat_spaces - read extraneous spaces.
1944  */
1945 static void
1946 eat_spaces()
1947 {
1948        char    c;

1950        while ((c = getachar()) == ' ')
1951                ;
1952        ungetachar(c);
1953 }

1955 /*
1956  * restore_inode - set up all inode indicators so inum is now
1957  *      the current inode.
1958  */
1959 static void
1960 restore_inode(ino_t inum)
1961 {
1962        errinum = cur_inum = inum;
1963        addr = errino = cur_ino = itob(inum);
1964 }

1966 /*
1967  * match - return false if the input does not match string up to
1968  *      upto letters.   Then proceed to chew up extraneous letters.
1969  */
1970 static int
1971 match(char *string, int upto)
1972 {
1973        int     i, length = strlen(string) - 1;
1974        char    c;
1975        int     save_upto = upto;
```

```
1977         while (--upto) {
1978                 string++;
1979                 if ((c = getachar()) != *string) {
1980                         for (i = save_upto - upto; i; i--) {
1981                                 ungetachar(c);
1982                                 c = *--string;
1983                         }
1984                         return (0);
1985                 }
1986                 length--;
1987         }
1988         while (length--) {
1989                 string++;
1990                 if ((c = getachar()) != *string) {
1991                         ungetachar(c);
1992                         return (1);
1993                 }
1994         }
1995         return (1);
1996 }

1998 /*
1999  * expr - expression evaluator.  Will evaluate expressions from
2000  *      left to right with no operator precedence.  Parentheses may
2001  *      be used.
2002  */
2003 static long
2004 expr()
2005 {
2006         long    numb = 0, temp;
2007         char    c;

2009         numb = term();
2010         for (;;) {
2011                 if (error)
2012                         return (~0);    /* error is set so value is ignored */
2013                 c = getachar();
2014                 switch (c) {

2016                 case '+':
2017                         numb += term();
2018                         continue;

2020                 case '-':
2021                         numb -= term();
2022                         continue;

2024                 case '*':
2025                         numb *= term();
2026                         continue;

2028                 case '%':
2029                         temp = term();
2030                         if (!temp) {
2031                                 printf("divide by zero\n");
2032                                 error++;
2033                                 return (~0);
2034                         }
2035                         numb /= temp;
2036                         continue;

2038                 case ')':
2039                         paren--;
2040                         return (numb);
```



```
2042                 default:
2043                         ungetachar(c);
2044                         if (paren && !error) {
2045                                 printf("missing ')'\n");
2046                                 error++;
2047                         }
2048                         return (numb);
2049                 }
2050         }
2051 }

2053 /*
2054  * term - used by expression evaluator to get an operand.
2055  */
2056 static long
2057 term()
2058 {
2059         char    c;

2061         switch (c = getachar()) {

2063         default:
2064                 ungetachar(c);
2065                 /*FALLTHRU*/
2066         case '+':
2067                 return (getnumb());

2069         case '-':
2070                 return (-getnumb());

2072         case '(':
2073                 paren++;
2074                 return (expr());
2075         }
2076 }

2078 /*
2079  * getnumb - read a number from the input stream.  A leading
2080  *      zero signifies octal interpretation, a leading '0x'
2081  *      signifies hexadecimal, and a leading '0t' signifies
2082  *      decimal.  If the first character is a character,
2083  *      return an error.
2084  */
2085 static long
2086 getnumb()
2087 {

2089         char            c, savec;
2090         long            number = 0, tbase, num;
2091         extern short    error;

2093         c = getachar();
2094         if (!digit(c)) {
2095                 error++;
2096                 ungetachar(c);
2097                 return (-1);
2098         }
2099         if (c == '0') {
2100                 tbase = OCTAL;
2101                 if ((c = getachar()) == 'x')
2102                         tbase = HEX;
2103                 else if (c == 't')
2104                         tbase = DECIMAL;
2105                 else ungetachar(c);
2106         } else {
2107                 tbase = base;
```

```
2108                    ungetachar(c);
2109            }
2110            for (;;) {
2111                    num = tbase;
2112                    c = savec = getachar();
2113                    if (HEXLETTER(c))
2114                            c = uppertolower(c);
2115                    switch (tbase) {
2116                    case HEX:
2117                            if (hexletter(c)) {
2118                                    num = hextodigit(c);
2119                                    break;
2120                            }
2121                            /*FALLTHRU*/
2122                    case DECIMAL:
2123                            if (digit(c))
2124                                    num = numtodigit(c);
2125                            break;
2126                    case OCTAL:
2127                            if (octaldigit(c))
2128                                    num = numtodigit(c);
2129                    }
2130                    if (num == tbase)
2131                            break;
2132                    number = number * tbase + num;
2133            }
2134            ungetachar(savec);
2135            return (number);
2136    }

2138    /*
2139     * find - the syntax is almost identical to the unix command.
2140     *              find dir [-name pattern] [-inum number]
2141     *      Note:  only one of -name or -inum may be used at a time.
2142     *             Also, the -print is not needed (implied).
2143     */
2144    static void
2145    find()
2146    {
2147            struct filenames        *fn;
2148            char                    c;
2149            long                    temp;
2150            short                   mode;

2152            eat_spaces();
2153            temp = cur_inum;
2154            top = filenames - 1;
2155            doing_cd = 1;
2156            parse();
2157            doing_cd = 0;
2158            if (nfiles != 1) {
2159                    restore_inode((ino_t)temp);
2160                    if (!error) {
2161                            print_path(input_path, (int)input_pathp);
2162                            if (nfiles == 0)
2163                                    printf(" not found\n");
2164                            else
2165                                    printf(" ambiguous\n");
2166                            error++;
2167                            return;
2168                    }
2169            }
2170            restore_inode(filenames->ino);
2171            freemem(filenames, nfiles);
2172            nfiles = 0;
2173            top = filenames - 1;
```



```
2174            if ((mode = icheck(addr)) == 0)
2175                    return;
2176            if ((mode & IFMT) != IFDIR) {
2177                    print_path(input_path, (int)input_pathp);
2178                    printf(" not a directory\n");
2179                    error++;
2180                    return;
2181            }
2182            eat_spaces();
2183            if ((c = getachar()) != '-') {
2184                    restore_inode((ino_t)temp);
2185                    printf("missing '-'\n");
2186                    error++;
2187                    return;
2188            }
2189            find_by_name = find_by_inode = 0;
2190            c = getachar();
2191            if (match("name", 4)) {
2192                    eat_spaces();
2193                    find_by_name = 1;
2194            } else if (match("inum", 4)) {
2195                    eat_spaces();
2196                    find_ino = expr();
2197                    if (error) {
2198                            restore_inode((ino_t)temp);
2199                            return;
2200                    }
2201                    while ((c = getachar()) != '\n')
2202                            ;
2203                    ungetachar(c);
2204                    find_by_inode = 1;
2205            } else {
2206                    restore_inode((ino_t)temp);
2207                    printf("use -name or -inum with find\n");
2208                    error++;
2209                    return;
2210            }
2211            doing_find = 1;
2212            parse();
2213            doing_find = 0;
2214            if (error) {
2215                    restore_inode((ino_t)temp);
2216                    return;
2217            }
2218            for (fn = filenames; fn <= top; fn++) {
2219                    if (fn->find == 0)
2220                            continue;
2221                    printf("i#: ");
2222                    print(fn->ino, 12, -8, 0);
2223                    print_path(fn->fname, (int)fn->len);
2224                    printf("\n");
2225            }
2226            restore_inode((ino_t)temp);
2227    }

2229    /*
2230     * ls - do an ls.  Should behave exactly as ls(1).
2231     *      Only -R and -l is supported and -l gives different results.
2232     */
2233    static void
2234    ls(struct filenames *fn0, struct filenames *fnlast, short level)
2235    {
2236            struct filenames        *fn, *fnn;

2238            fn = fn0;
2239            for (;;) {
```

```
2240                fn0 = fn;
2241                if (fn0->len) {
2242                        cmp_level = level;
2243                        qsort((char *)fn0, fnlast - fn0 + 1,
2244                                sizeof (struct filenames), fcmp);
2245                }
2246                for (fnn = fn, fn++; fn <= fnlast; fnn = fn, fn++) {
2247                        if (fnn->len != fn->len && level == fnn->len - 1)
2248                                break;
2249                        if (fnn->len == 0)
2250                                continue;
2251                        if (strcmp(fn->fname[level], fnn->fname[level]))
2252                                break;
2253                }
2254                if (fn0->len && level != fn0->len - 1)
2255                        ls(fn0, fnn, level + 1);
2256                else {
2257                        if (fn0 != filenames)
2258                                printf("\n");
2259                        print_path(fn0->fname, (int)(fn0->len - 1));
2260                        printf(":\n");
2261                        if (fn0->len == 0)
2262                                cmp_level = level;
2263                        else
2264                                cmp_level = level + 1;
2265                        qsort((char *)fn0, fnn - fn0 + 1,
2266                                sizeof (struct filenames), fcmp);
2267                        formatf(fn0, fnn);
2268                        nfiles -= fnn - fn0 + 1;
2269                }
2270                if (fn > fnlast)
2271                        return;
2272        }
2273 }

2275 /*
2276  * formatf - code lifted from ls.
2277  */
2278 static void
2279 formatf(struct filenames *fn0, struct filenames *fnlast)
2280 {
2281        struct filenames        *fn;
2282        int                     width = 0, w, nentry = fnlast - fn0 + 1;
2283        int                     i, j, columns, lines;
2284        char                    *cp;

2286        if (long_list) {
2287                columns = 1;
2288        } else {
2289                for (fn = fn0; fn <= fnlast; fn++) {
2290                        int len = strlen(fn->fname[cmp_level]) + 2;

2292                        if (len > width)
2293                                width = len;
2294                }
2295                width = (width + 8) &~ 7;
2296                columns = 80 / width;
2297                if (columns == 0)
2298                        columns = 1;
2299        }
2300        lines = (nentry + columns - 1) / columns;
2301        for (i = 0; i < lines; i++) {
2302                for (j = 0; j < columns; j++) {
2303                        fn = fn0 + j * lines + i;
2304                        if (long_list) {
2305                                printf("i#: ");
```



```
2306                                print(fn->ino, 12, -8, 0);
2307                        }
2308                        if ((cp = fmtentry(fn)) == NULL) {
2309                                printf("cannot read inode %ld\n", fn->ino);
2310                                return;
2311                        }
2312                        printf("%s", cp);
2313                        if (fn + lines > fnlast) {
2314                                printf("\n");
2315                                break;
2316                        }
2317                        w = strlen(cp);
2318                        while (w < width) {
2319                                w = (w + 8) &~ 7;
2320                                (void) putchar('\t');
2321                        }
2322                }
2323        }
2324 }

2326 /*
2327  * fmtentry - code lifted from ls.
2328  */
2329 static char *
2330 fmtentry(struct filenames *fn)
2331 {
2332        static char     fmtres[BUFSIZ];
2333        struct dinode   *ip;
2334        char            *cptr, *cp, *dp;

2336        dp = &fmtres[0];
2337        for (cp = fn->fname[cmp_level]; *cp; cp++) {
2338                if (*cp < ' ' || *cp >= 0177)
2339                        *dp++ = '?';
2340                else
2341                        *dp++ = *cp;
2342        }
2343        addr = itob(fn->ino);
2344        if ((cptr = getblk(addr)) == 0)
2345                return (NULL);
2346        cptr += blkoff(fs, addr);
2347        /*LINTED*/
2348        ip = (struct dinode *)cptr;
2349        switch (ip->di_mode & IFMT) {
2350        case IFDIR:
2351                *dp++ = '/';
2352                break;
2353        case IFLNK:
2354                *dp++ = '@';
2355                break;
2356        case IFSOCK:
2357                *dp++ = '=';
2358                break;
2359 #ifdef IFIFO
2360        case IFIFO:
2361                *dp++ = 'p';
2362                break;
2363 #endif
2364        case IFCHR:
2365        case IFBLK:
2366        case IFREG:
2367                if (ip->di_mode & 0111)
2368                        *dp++ = '*';
2369                else
2370                        *dp++ = ' ';
2371                break;
```

```
2372         default:
2373                 *dp++ = '?';

2375         }
2376         *dp++ = 0;
2377         return (fmtres);
2378 }

2380 /*
2381  * fcmp - routine used by qsort.  Will sort first by name, then
2382  *      then by pathname length if names are equal.  Uses global
2383  *      cmp_level to tell what component of the path name we are comparing.
2384  */
2385 static int
2386 fcmp(struct filenames *f1, struct filenames *f2)
2387 {
2388         int value;

2390         if ((value = strcmp(f1->fname[cmp_level], f2->fname[cmp_level])))
2391                 return (value);
2392         return (f1->len - f2->len);
2393 }

2395 /*
2396  * ffcmp - routine used by qsort.  Sort only by pathname length.
2397  */
2398 static int
2399 ffcmp(struct filenames *f1, struct filenames *f2)
2400 {
2401         return (f1->len - f2->len);
2402 }

2404 /*
2405  * parse - set up the call to follow_path.
2406  */
2407 static void
2408 parse()
2409 {
2410         int     i;
2411         char    c;

2413         stack_pathp = input_pathp = -1;
2414         if ((c = getachar()) == '/') {
2415                 while ((c = getachar()) == '/')
2416                         ;
2417                 ungetachar(c);
2418                 cur_inum = 2;
2419                 c = getachar();
2420                 if ((c == '\n') || ((doing_cd) && (c == ' '))) {
2421                         ungetachar(c);
2422                         if (doing_cd) {
2423                                 top++;
2424                                 top->ino = 2;
2425                                 top->len = -1;
2426                                 nfiles = 1;
2427                                 return;
2428                         }
2429                 } else
2430                         ungetachar(c);
2431         } else {
2432                 ungetachar(c);
2433                 stack_pathp = current_pathp;
2434                 if (!doing_find)
2435                         input_pathp = current_pathp;
2436                 for (i = 0; i <= current_pathp; i++) {
2437                         if (!doing_find)
```



```
2438                                 (void) strcpy(input_path[i], current_path[i]);
2439                         (void) strcpy(stack_path[i], current_path[i]);
2440                 }
2441         }
2442         getname();
2443         follow_path((long)(stack_pathp + 1), cur_inum);
2444 }

2446 /*
2447  * follow_path - called by cd, find, and ls.
2448  *      input_path holds the name typed by the user.
2449  *      stack_path holds the name at the current depth.
2450  */
2451 static void
2452 follow_path(long level, long inum)
2453 {
2454         struct direct           *dirp;
2455         char                    **ccptr, *cptr;
2456         int                     i;
2457         struct filenames        *tos, *bos, *fn, *fnn, *fnnn;
2458         long                    block;
2459         short                   mode;

2461         tos = top + 1;
2462         restore_inode((ino_t)inum);
2463         if ((mode = icheck(addr)) == 0)
2464                 return;
2465         if ((mode & IFMT) != IFDIR)
2466             return;
2467         block = cur_bytes = 0;
2468         while (cur_bytes < filesize) {
2469             if (block == 0 || bcomp(addr)) {
2470                 error = 0;
2471                 if ((addr = ((u_offset_t)bmap(block++) <<
2472                                 (u_offset_t)FRGSHIFT)) == 0)
2473                     break;
2474                 if ((cptr = getblk(addr)) == 0)
2475                     break;
2476                 cptr += blkoff(fs, addr);
2477             }
2478                 /*LINTED*/
2479             dirp = (struct direct *)cptr;
2480             if (dirp->d_ino) {
2481                 if (level > input_pathp || doing_find ||
2482                         compare(input_path[level], &dirp->d_name[0], 1)) {
2483                     if ((doing_find) &&
2484                         ((strcmp(dirp->d_name, ".") == 0 ||
2485                                 strcmp(dirp->d_name, "..") == 0)))
2486                         goto duplicate;
2487                     if (++top - filenames >= maxfiles) {
2488                         printf("too many files\n");
2489                         error++;
2490                         return;
2491                     }
2492                     top->fname = (char **)calloc(FIRST_DEPTH, sizeof (char **));
2493                     top->flag = 0;
2494                     if (top->fname == 0) {
2495                         printf("out of memory\n");
2496                         error++;
2497                         return;
2498                     }
2499                     nfiles++;
2500                     top->ino = dirp->d_ino;
2501                     top->len = stack_pathp;
2502                     top->find = 0;
2503                     if (doing_find) {
```

```
2504                        if (find_by_name) {
2505                            if (compare(input_path[0], &dirp->d_name[0], 1))
2506                                top->find = 1;
2507                        } else if (find_by_inode)
2508                            if (find_ino == dirp->d_ino)
2509                                top->find = 1;
2510                    }
2511                    if (top->len + 1 >= FIRST_DEPTH && top->flag == 0) {
2512                        ccptr = (char **)calloc(SECOND_DEPTH, sizeof (char **));
2513                        if (ccptr == 0) {
2514                            printf("out of memory\n");
2515                            error++;
2516                            return;
2517                        }
2518                        for (i = 0; i < FIRST_DEPTH; i++)
2519                                ccptr[i] = top->fname[i];
2520                        free((char *)top->fname);
2521                        top->fname = ccptr;
2522                        top->flag = 1;
2523                    }
2524                    if (top->len >= SECOND_DEPTH) {
2525                        printf("maximum depth exceeded, try to cd lower\n");
2526                        error++;
2527                        return;
2528                    }
2529                        /*
2530                         * Copy current depth.
2531                         */
2532                    for (i = 0; i <= stack_pathp; i++) {
2533                        top->fname[i] = calloc(1, strlen(stack_path[i])+1);
2534                        if (top->fname[i] == 0) {
2535                            printf("out of memory\n");
2536                            error++;
2537                            return;
2538                        }
2539                        (void) strcpy(top->fname[i], stack_path[i]);
2540                    }
2541                        /*
2542                         * Check for '.' or '..' typed.
2543                         */
2544                    if ((level <= input_pathp) &&
2545                            (strcmp(input_path[level], ".") == 0 ||
2546                                    strcmp(input_path[level], "..") == 0)) {
2547                        if (strcmp(input_path[level], "..") == 0 &&
2548                                                        top->len >= 0) {
2549                            free(top->fname[top->len]);
2550                            top->len -= 1;
2551                        }
2552                    } else {
2553                        /*
2554                         * Check for duplicates.
2555                         */
2556                        if (!doing_cd && !doing_find) {
2557                            for (fn = filenames; fn < top; fn++) {
2558                                if (fn->ino == dirp->d_ino &&
2559                                        fn->len == stack_pathp + 1) {
2560                                    for (i = 0; i < fn->len; i++)
2561                                        if (strcmp(fn->fname[i], stack_path[i]))
2562                                            break;
2563                                    if (i != fn->len ||
2564                                            strcmp(fn->fname[i], dirp->d_name))
2565                                        continue;
2566                                    freemem(top, 1);
2567                                    if (top == filenames)
2568                                        top = NULL;
2569                                    else
```



```
2570                                        top--;
2571                                        nfiles--;
2572                                        goto duplicate;
2573                                    }
2574                                }
2575                            }
2576                            top->len += 1;
2577                            top->fname[top->len] = calloc(1,
2578                                                strlen(&dirp->d_name[0])+1);
2579                            if (top->fname[top->len] == 0) {
2580                                printf("out of memory\n");
2581                                error++;
2582                                return;
2583                            }
2584                            (void) strcpy(top->fname[top->len], &dirp->d_name[0]);
2585                        }
2586                    }
2587                }
2588            duplicate:
2589                    addr += dirp->d_reclen;
2590                    cptr += dirp->d_reclen;
2591                    cur_bytes += dirp->d_reclen;
2592                }
2593                if (top < filenames)
2594                    return;
2595                if ((doing_cd && level == input_pathp) ||
2596                        (!recursive && !doing_find && level > input_pathp))
2597                    return;
2598                bos = top;
2599                /*
2600                 * Check newly added entries to determine if further expansion
2601                 * is required.
2602                 */
2603                for (fn = tos; fn <= bos; fn++) {
2604                        /*
2605                         * Avoid '.' and '..' if beyond input.
2606                         */
2607                    if ((recursive || doing_find) && (level > input_pathp) &&
2608                        (strcmp(fn->fname[fn->len], ".") == 0 ||
2609                                strcmp(fn->fname[fn->len], "..") == 0))
2610                        continue;
2611                    restore_inode(fn->ino);
2612                    if ((mode = icheck(cur_ino)) == 0)
2613                        return;
2614                    if ((mode & IFMT) == IFDIR || level < input_pathp) {
2615                        /*
2616                         * Set up current depth, remove current entry and
2617                         * continue recursion.
2618                         */
2619                        for (i = 0; i <= fn->len; i++)
2620                            (void) strcpy(stack_path[i], fn->fname[i]);
2621                        stack_pathp = fn->len;
2622                        if (!doing_find &&
2623                                (!recursive || (recursive && level <= input_pathp))) {
2624                                /*
2625                                 * Remove current entry by moving others up.
2626                                 */
2627                            freemem(fn, 1);
2628                            fnn = fn;
2629                            for (fnnn = fnn, fnn++; fnn <= top; fnnn = fnn, fnn++) {
2630                                fnnn->ino = fnn->ino;
2631                                fnnn->len = fnn->len;
2632                                if (fnnn->len + 1 < FIRST_DEPTH) {
2633                                    fnnn->fname = (char **)calloc(FIRST_DEPTH,
2634                                                                sizeof (char **));
2635                                    fnnn->flag = 0;
```

```
2636                        } else if (fnnn->len < SECOND_DEPTH) {
2637                            fnnn->fname = (char **)calloc(SECOND_DEPTH,
2638                                                sizeof (char **));
2639                            fnnn->flag = 1;
2640                        } else {
2641                            printf("maximum depth exceeded, ");
2642                            printf("try to cd lower\n");
2643                            error++;
2644                            return;
2645                        }
2646                        for (i = 0; i <= fnn->len; i++)
2647                            fnnn->fname[i] = fnn->fname[i];
2648                    }
2649                    if (fn == tos)
2650                        fn--;
2651                    top--;
2652                    bos--;
2653                    nfiles--;
2654                }
2655                follow_path(level + 1, cur_inum);
2656                if (error)
2657                        return;
2658            }
2659        }
2660 }

2662 /*
2663  * getname - break up the pathname entered by the user into components.
2664  */
2665 static void
2666 getname()
2667 {
2668         int     i;
2669         char    c;

2671         if ((c = getachar()) == '\n') {
2672             ungetachar(c);
2673             return;
2674         }
2675         ungetachar(c);
2676         input_pathp++;
2677 clear:
2678         for (i = 0; i < MAXNAMLEN; i++)
2679             input_path[input_pathp][i] = '\0';
2680         for (;;) {
2681             c = getachar();
2682             if (c == '\\') {
2683                 if ((int)strlen(input_path[input_pathp]) + 1 >= MAXNAMLEN) {
2684                     printf("maximum name length exceeded, ");
2685                     printf("truncating\n");
2686                     return;
2687                 }
2688                 input_path[input_pathp][strlen(input_path[input_pathp])] = c;
2689                 input_path[input_pathp][strlen(input_path[input_pathp])] =
2690                                                 getachar();
2691                 continue;
2692             }
2693             if (c == ' ' || c == '\n') {
2694                 ungetachar(c);
2695                 return;
2696             }
2697             if (!doing_find && c == '/') {
2698                 if (++input_pathp >= MAXPATHLEN) {
2699                     printf("maximum path length exceeded, ");
2700                     printf("truncating\n");
2701                     input_pathp--;
```



```
2702                        return;
2703                }
2704                goto clear;
2705            }
2706            if ((int)strlen(input_path[input_pathp]) >= MAXNAMLEN) {
2707                printf("maximum name length exceeded, truncating\n");
2708                return;
2709            }
2710            input_path[input_pathp][strlen(input_path[input_pathp])] = c;
2711        }
2712 }

2714 /*
2715  * compare - check if a filename matches the pattern entered by the user.
2716  *      Handles '*', '?', and '[]'.
2717  */
2718 static int
2719 compare(char *s1, char *s2, short at_start)
2720 {
2721         char    c, *s;

2723         s = s2;
2724         while ((c = *s1) != NULL) {
2725                 if (c == '*') {
2726                         if (at_start && s == s2 && !letter(*s2) && !digit(*s2))
2727                                 return (0);
2728                         if (*++s1 == 0)
2729                                 return (1);
2730                         while (*s2) {
2731                                 if (compare(s1, s2, 0))
2732                                         return (1);
2733                                 if (error)
2734                                         return (0);
2735                                 s2++;
2736                         }
2737                 }
2738                 if (*s2 == 0)
2739                         return (0);
2740                 if (c == '\\') {
2741                         s1++;
2742                         goto compare_chars;
2743                 }
2744                 if (c == '?') {
2745                         if (at_start && s == s2 && !letter(*s2) && !digit(*s2))
2746                                 return (0);
2747                         s1++;
2748                         s2++;
2749                         continue;
2750                 }
2751                 if (c == '[') {
2752                         s1++;
2753                         if (*s2 >= *s1++) {
2754                                 if (*s1++ != '-') {
2755                                         printf("missing '-'\n");
2756                                         error++;
2757                                         return (0);
2758                                 }
2759                                 if (*s2 <= *s1++) {
2760                                         if (*s1++ != ']') {
2761                                                 printf("missing ']'");
2762                                                 error++;
2763                                                 return (0);
2764                                         }
2765                                         s2++;
2766                                         continue;
2767                                 }
```

```
2768                        }
2769                }
2770 compare_chars:
2771                if (*s1++ == *s2++)
2772                        continue;
2773                else
2774                        return (0);
2775        }
2776        if (*s1 == *s2)
2777                return (1);
2778        return (0);
2779 }

2781 /*
2782  * freemem - free the memory allocated to the filenames structure.
2783  */
2784 static void
2785 freemem(struct filenames *p, int numb)
2786 {
2787        int     i, j;

2789        if (numb == 0)
2790                return;
2791        for (i = 0; i < numb; i++, p++) {
2792                for (j = 0; j <= p->len; j++)
2793                        free(p->fname[j]);
2794                free((char *)p->fname);
2795        }
2796 }

2798 /*
2799  * print_path - print the pathname held in p.
2800  */
2801 static void
2802 print_path(char *p[], int pntr)
2803 {
2804        int     i;

2806        printf("/");
2807        if (pntr >= 0) {
2808                for (i = 0; i < pntr; i++)
2809                        printf("%s/", p[i]);
2810                printf("%s", p[pntr]);
2811        }
2812 }

2814 /*
2815  * fill - fill a section with a value or string.
2816  *      addr,count:fill=[value, "string"].
2817  */
2818 static void
2819 fill()
2820 {
2821        char            *cptr;
2822        int             i;
2823        short           eof_flag, end = 0, eof = 0;
2824        long            temp, tcount;
2825        u_offset_t      taddr;

2827        if (wrtflag == O_RDONLY) {
2828                printf("not opened for write '-w'\n");
2829                error++;
2830                return;
2831        }
2832        temp = expr();
2833        if (error)
```



```
2834                return;
2835        if ((cptr = getblk(addr)) == 0)
2836                return;
2837        if (type == NUMB)
2838                eof_flag = 0;
2839        else
2840                eof_flag = 1;
2841        taddr = addr;
2842        switch (objsz) {
2843        case LONG:
2844                addr &= ~(LONG - 1);
2845                break;
2846        case SHORT:
2847                addr &= ~(SHORT - 1);
2848                temp &= 0177777L;
2849                break;
2850        case CHAR:
2851                temp &= 0377;
2852        }
2853        cur_bytes -= taddr - addr;
2854        cptr += blkoff(fs, addr);
2855        tcount = check_addr(eof_flag, &end, &eof, 0);
2856        for (i = 0; i < tcount; i++) {
2857                switch (objsz) {
2858                case LONG:
2859                        /*LINTED*/
2860                        *(long *)cptr = temp;
2861                        break;
2862                case SHORT:
2863                        /*LINTED*/
2864                        *(short *)cptr = temp;
2865                        break;
2866                case CHAR:
2867                        *cptr = temp;
2868                }
2869                cptr += objsz;
2870        }
2871        addr += (tcount - 1) * objsz;
2872        cur_bytes += (tcount - 1) * objsz;
2873        put((u_offset_t)temp, objsz);
2874        if (eof) {
2875                printf("end of file\n");
2876                error++;
2877        } else if (end) {
2878                printf("end of block\n");
2879                error++;
2880        }
2881 }

2883 /*
2884  * get - read a byte, short or long from the file system.
2885  *      The entire block containing the desired item is read
2886  *      and the appropriate data is extracted and returned.
2887  */
2888 static offset_t
2889 get(short lngth)
2890 {

2892        char            *bptr;
2893        u_offset_t      temp = addr;

2895        objsz = lngth;
2896        if (objsz == INODE || objsz == SHORT)
2897                temp &= ~(SHORT - 1);
2898        else if (objsz == DIRECTORY || objsz == LONG || objsz == SHADOW_DATA)
2899                temp &= ~(LONG - 1);
```

```
2900         if ((bptr = getblk(temp)) == 0)
2901                 return (-1);
2902         bptr += blkoff(fs, temp);
2903         switch (objsz) {
2904         case CHAR:
2905                 return ((offset_t)*bptr);
2906         case SHORT:
2907         case INODE:
2908                 /*LINTED*/
2909                 return ((offset_t)(*(short *)bptr));
2910         case LONG:
2911         case DIRECTORY:
2912         case SHADOW_DATA:
2913                 /*LINTED*/
2914                 return ((offset_t)(*(long *)bptr));
2915         case U_OFFSET_T:
2916                 /*LINTED*/
2917                 return (*(offset_t *)bptr);
2918         }
2919         return (0);
2920 }

2922 /*
2923  * cgrp_check - make sure that we don't bump the cylinder group
2924  *      beyond the total number of cylinder groups or before the start.
2925  */
2926 static int
2927 cgrp_check(long cgrp)
2928 {
2929         if (cgrp < 0) {
2930                 if (objsz == CGRP)
2931                         printf("beginning of cylinder groups\n");
2932                 else
2933                         printf("beginning of super blocks\n");
2934                 error++;
2935                 return (0);
2936         }
2937         if (cgrp >= fs->fs_ncg) {
2938                 if (objsz == CGRP)
2939                         printf("end of cylinder groups\n");
2940                 else
2941                         printf("end of super blocks\n");
2942                 error++;
2943                 return (0);
2944         }
2945         if (objsz == CGRP)
2946                 return (cgtod(fs, cgrp) << FRGSHIFT);
2947         else
2948                 return (cgsblock(fs, cgrp) << FRGSHIFT);
2949 }

2951 /*
2952  * icheck -  make sure we can read the block containing the inode
2953  *      and determine the filesize (0 if inode not allocated).  Return
2954  *      0 if error otherwise return the mode.
2955  */
2956 int
2957 icheck(u_offset_t address)
2958 {
2959         char            *cptr;
2960         struct dinode   *ip;

2962         if ((cptr = getblk(address)) == 0)
2963                 return (0);
2964         cptr += blkoff(fs, address);
2965         /*LINTED*/
```



```
2966         ip = (struct dinode *)cptr;
2967         if ((ip->di_mode & IFMT) == 0) {
2968                 if (!override) {
2969                         printf("inode not allocated\n");
2970                         error++;
2971                         return (0);
2972                 }
2973                 blocksize = filesize = 0;
2974         } else {
2975                 trapped++;
2976                 filesize = ip->di_size;
2977                 blocksize = filesize * 2;
2978         }
2979         return (ip->di_mode);
2980 }

2982 /*
2983  * getdirslot - get the address of the directory slot desired.
2984  */
2985 static u_offset_t
2986 getdirslot(long slot)
2987 {
2988         char            *cptr;
2989         struct direct   *dirp;
2990         short           i;
2991         char            *string = &scratch[0];
2992         short           bod = 0, mode, temp;

2994         if (slot < 0) {
2995                 slot = 0;
2996                 bod++;
2997         }
2998         if (type != DIRECTORY) {
2999                 if (type == BLOCK)
3000                         string = "block";
3001                 else
3002                         string = "fragment";
3003                 addr = bod_addr;
3004                 if ((cptr = getblk(addr)) == 0)
3005                         return (0);
3006                 cptr += blkoff(fs, addr);
3007                 cur_bytes = 0;
3008                 /*LINTED*/
3009                 dirp = (struct direct *)cptr;
3010                 for (dirslot = 0; dirslot < slot; dirslot++) {
3011                         /*LINTED*/
3012                         dirp = (struct direct *)cptr;
3013                         if (blocksize > filesize) {
3014                                 if (cur_bytes + (long)dirp->d_reclen >=
3015                                                                 filesize) {
3016                                         printf("end of file\n");
3017                                         erraddr = addr;
3018                                         errcur_bytes = cur_bytes;
3019                                         stringsize = STRINGSIZE(dirp);
3020                                         error++;
3021                                         return (addr);
3022                                 }
3023                         } else {
3024                                 if (cur_bytes + (long)dirp->d_reclen >=
3025                                                                 blocksize) {
3026                                         printf("end of %s\n", string);
3027                                         erraddr = addr;
3028                                         errcur_bytes = cur_bytes;
3029                                         stringsize = STRINGSIZE(dirp);
3030                                         error++;
3031                                         return (addr);
```

```
3032                                    }
3033                            }
3034                            cptr += dirp->d_reclen;
3035                            addr += dirp->d_reclen;
3036                            cur_bytes += dirp->d_reclen;
3037                    }
3038                    if (bod) {
3039                            if (blocksize > filesize)
3040                                    printf("beginning of file\n");
3041                            else
3042                                    printf("beginning of %s\n", string);
3043                            erraddr = addr;
3044                            errcur_bytes = cur_bytes;
3045                            error++;
3046                    }
3047                    stringsize = STRINGSIZE(dirp);
3048                    return (addr);
3049            } else {
3050                    addr = cur_ino;
3051                    if ((mode = icheck(addr)) == 0)
3052                            return (0);
3053                    if (!override && (mode & IFDIR) == 0) {
3054                            printf("inode is not a directory\n");
3055                            error++;
3056                            return (0);
3057                    }
3058                    temp = slot;
3059                    i = cur_bytes = 0;
3060                    for (;;) {
3061                            if (i == 0 || bcomp(addr)) {
3062                                    error = 0;
3063                                    if ((addr = (bmap((long)i++) << FRGSHIFT)) == 0)
3064                                            break;
3065                                    if ((cptr = getblk(addr)) == 0)
3066                                            break;
3067                                    cptr += blkoff(fs, addr);
3068                            }
3069                            /*LINTED*/
3070                            dirp = (struct direct *)cptr;
3071                            value = dirp->d_ino;
3072                            if (!temp--)
3073                                    break;
3074                            if (cur_bytes + (long)dirp->d_reclen >= filesize) {
3075                                    printf("end of file\n");
3076                                    dirslot = slot - temp - 1;
3077                                    objsz = DIRECTORY;
3078                                    erraddr = addr;
3079                                    errcur_bytes = cur_bytes;
3080                                    stringsize = STRINGSIZE(dirp);
3081                                    error++;
3082                                    return (addr);
3083                            }
3084                            addr += dirp->d_reclen;
3085                            cptr += dirp->d_reclen;
3086                            cur_bytes += dirp->d_reclen;
3087                    }
3088                    dirslot = slot;
3089                    objsz = DIRECTORY;
3090                    if (bod) {
3091                            printf("beginning of file\n");
3092                            erraddr = addr;
3093                            errcur_bytes = cur_bytes;
3094                            error++;
3095                    }
3096                    stringsize = STRINGSIZE(dirp);
3097                    return (addr);
```



```
3098            }
3099    }


3102    /*
3103     * getshadowslot - get the address of the shadow data desired
3104     */
3105    static int
3106    getshadowslot(long shadow)
3107    {
3108            struct ufs_fsd          fsd;
3109            short                   bod = 0, mode;
3110            long                    taddr, tcurbytes;

3112            if (shadow < 0) {
3113                    shadow = 0;
3114                    bod++;
3115            }
3116            if (type != SHADOW_DATA) {
3117                    if (shadow < cur_shad) {
3118                            printf("can't scan shadow data in reverse\n");
3119                            error++;
3120                            return (0);
3121                    }
3122            } else {
3123                    addr = cur_ino;
3124                    if ((mode = icheck(addr)) == 0)
3125                            return (0);
3126                    if (!override && (mode & IFMT) != IFSHAD) {
3127                            printf("inode is not a shadow\n");
3128                            error++;
3129                            return (0);
3130                    }
3131                    cur_bytes = 0;
3132                    cur_shad = 0;
3133                    syncshadowscan(1);      /* force synchronization */
3134            }

3136            for (; cur_shad < shadow; cur_shad++) {
3137                    taddr = addr;
3138                    tcurbytes = cur_bytes;
3139                    getshadowdata((long *)&fsd, LONG + LONG);
3140                    addr = taddr;
3141                    cur_bytes = tcurbytes;
3142                    if (cur_bytes + (long)fsd.fsd_size > filesize) {
3143                            syncshadowscan(0);
3144                            printf("end of file\n");
3145                            erraddr = addr;
3146                            errcur_bytes = cur_bytes;
3147                            error++;
3148                            return (addr);
3149                    }
3150                    addr += fsd.fsd_size;
3151                    cur_bytes += fsd.fsd_size;
3152                    syncshadowscan(0);
3153            }
3154            if (type == SHADOW_DATA)
3155                    objsz = SHADOW_DATA;
3156            if (bod) {
3157                    printf("beginning of file\n");
3158                    erraddr = addr;
3159                    errcur_bytes = cur_bytes;
3160                    error++;
3161            }
3162            return (addr);
3163    }
```

```
3165 static void
3166 getshadowdata(long *buf, int len)
3167 {
3168         long    tfsd;

3170         len /= LONG;
3171         for (tfsd = 0; tfsd < len; tfsd++) {
3172                 buf[tfsd] = get(SHADOW_DATA);
3173                 addr += LONG;
3174                 cur_bytes += LONG;
3175                 syncshadowscan(0);
3176         }
3177 }

3179 static void
3180 syncshadowscan(int force)
3181 {
3182         long    curblkoff;
3183         if (type == SHADOW_DATA && (force ||
3184             lblkno(fs, addr) != (bhdr.fwd)->blkno)) {
3185                 curblkoff = blkoff(fs, cur_bytes);
3186                 addr = bmap(lblkno(fs, cur_bytes)) << FRGSHIFT;
3187                 addr += curblkoff;
3188                 cur_bytes += curblkoff;
3189                 (void) getblk(addr);
3190                 objsz = SHADOW_DATA;
3191         }
3192 }



3196 /*
3197  * putf - print a byte as an ascii character if possible.
3198  *      The exceptions are tabs, newlines, backslashes
3199  *      and nulls which are printed as the standard C
3200  *      language escapes. Characters which are not
3201  *      recognized are printed as \?.
3202  */
3203 static void
3204 putf(char c)
3205 {

3207         if (c <= 037 || c >= 0177 || c == '\\') {
3208                 printf("\\");
3209                 switch (c) {
3210                 case '\\':
3211                         printf("\\");
3212                         break;
3213                 case '\t':
3214                         printf("t");
3215                         break;
3216                 case '\n':
3217                         printf("n");
3218                         break;
3219                 case '\0':
3220                         printf("0");
3221                         break;
3222                 default:
3223                         printf("?");
3224                 }
3225         } else {
3226                 printf("%c", c);
3227                 printf(" ");
3228         }
3229 }
```



```
3231 /*
3232  * put - write an item into the buffer for the current address
3233  *      block.  The value is checked to make sure that it will
3234  *      fit in the size given without truncation.  If successful,
3235  *      the entire block is written back to the file system.
3236  */
3237 static void
3238 put(u_offset_t item, short lngth)
3239 {

3241         char    *bptr, *sbptr;
3242         long    s_err, nbytes;
3243         long    olditem;

3245         if (wrtflag == O_RDONLY) {
3246                 printf("not opened for write '-w'\n");
3247                 error++;
3248                 return;
3249         }
3250         objsz = lngth;
3251         if ((sbptr = getblk(addr)) == 0)
3252                 return;
3253         bptr = sbptr + blkoff(fs, addr);
3254         switch (objsz) {
3255         case LONG:
3256         case DIRECTORY:
3257                 /*LINTED*/
3258                 olditem = *(long *)bptr;
3259                 /*LINTED*/
3260                 *(long *)bptr = item;
3261                 break;
3262         case SHORT:
3263         case INODE:
3264                 /*LINTED*/
3265                 olditem = (long)*(short *)bptr;
3266                 item &= 0177777L;
3267                 /*LINTED*/
3268                 *(short *)bptr = item;
3269                 break;
3270         case CHAR:
3271                 olditem = (long)*bptr;
3272                 item &= 0377;
3273                 *bptr = lobyte(loword(item));
3274                 break;
3275         default:
3276                 error++;
3277                 return;
3278         }
3279         if ((s_err = llseek(fd, (offset_t)(addr & fs->fs_bmask), 0)) == -1) {
3280                 error++;
3281                 printf("seek error : %" PRIx64 "\n", addr);
3282                 return;
3283         }
3284         if ((nbytes = write(fd, sbptr, BLKSIZE)) != BLKSIZE) {
3285                 error++;
3286                 printf("write error : addr   = %" PRIx64 "\n", addr);
3287                 printf("            : s_err  = %lx\n", s_err);
3288                 printf("            : nbytes = %lx\n", nbytes);
3289                 return;
3290         }
3291         if (!acting_on_inode && objsz != INODE && objsz != DIRECTORY) {
3292                 index(base);
3293                 print(olditem, 8, -8, 0);
3294                 printf("\t=\t");
3295                 print(item, 8, -8, 0);
```

```
3296                printf("\n");
3297        } else {
3298                if (objsz == DIRECTORY) {
3299                        addr = cur_dir;
3300                        fprnt('?', 'd');
3301                } else {
3302                        addr = cur_ino;
3303                        objsz = INODE;
3304                        fprnt('?', 'i');
3305                }
3306        }
3307 }

3309 /*
3310  * getblk - check if the desired block is in the file system.
3311  *      Search the incore buffers to see if the block is already
3312  *      available. If successful, unlink the buffer control block
3313  *      from its position in the buffer list and re-insert it at
3314  *      the head of the list.  If failure, use the last buffer
3315  *      in the list for the desired block. Again, this control
3316  *      block is placed at the head of the list. This process
3317  *      will leave commonly requested blocks in the in-core buffers.
3318  *      Finally, a pointer to the buffer is returned.
3319  */
3320 static char *
3321 getblk(u_offset_t address)
3322 {

3324        struct lbuf     *bp;
3325        long            s_err, nbytes;
3326        unsigned long   block;

3328        read_requests++;
3329        block = lblkno(fs, address);
3330        if (block >= fragstoblks(fs, fs->fs_size)) {
3331                printf("cannot read block %lu\n", block);
3332                error++;
3333                return (0);
3334        }
3335        for (bp = bhdr.fwd; bp != &bhdr; bp = bp->fwd)
3336                if (bp->valid && bp->blkno == block)
3337                        goto xit;
3338        actual_disk_reads++;
3339        bp = bhdr.back;
3340        bp->blkno = block;
3341        bp->valid = 0;
3342        if ((s_err = llseek(fd, (offset_t)(address & fs->fs_bmask), 0)) == -1) {
3343                error++;
3344                printf("seek error : %" PRIx64 "\n", address);
3345                return (0);
3346        }
3347        if ((nbytes = read(fd, bp->blkaddr, BLKSIZE)) != BLKSIZE) {
3348                error++;
3349                printf("read error : addr   = %" PRIx64 "\n", address);
3350                printf("           : s_err  = %lx\n", s_err);
3351                printf("           : nbytes = %lx\n", nbytes);
3352                return (0);
3353        }
3354        bp->valid++;
3355 xit:   bp->back->fwd = bp->fwd;
3356        bp->fwd->back = bp->back;
3357        insert(bp);
3358        return (bp->blkaddr);
3359 }

3361 /*
```



```
3362  * insert - place the designated buffer control block
3363  *      at the head of the linked list of buffers.
3364  */
3365 static void
3366 insert(struct lbuf *bp)
3367 {

3369        bp->back = &bhdr;
3370        bp->fwd = bhdr.fwd;
3371        bhdr.fwd->back = bp;
3372        bhdr.fwd = bp;
3373 }

3375 /*
3376  * err - called on interrupts.  Set the current address
3377  *      back to the last address stored in erraddr. Reset all
3378  *      appropriate flags.  A reset call is made to return
3379  *      to the main loop;
3380  */
3381 #ifdef sun
3382 /*ARGSUSED*/
3383 static void
3384 err(int sig)
3385 #else
3386 err()
3387 #endif /* sun */
3388 {
3389        freemem(filenames, nfiles);
3390        nfiles = 0;
3391        (void) signal(2, err);
3392        addr = erraddr;
3393        cur_ino = errino;
3394        cur_inum = errinum;
3395        cur_bytes = errcur_bytes;
3396        error = 0;
3397        c_count = 0;
3398        printf("\n?\n");
3399        (void) fseek(stdin, 0L, 2);
3400        longjmp(env, 0);
3401 }

3403 /*
3404  * devcheck - check that the given mode represents a
3405  *      special device. The IFCHR bit is on for both
3406  *      character and block devices.
3407  */
3408 static int
3409 devcheck(short md)
3410 {
3411        if (override)
3412                return (0);
3413        switch (md & IFMT) {
3414        case IFCHR:
3415        case IFBLK:
3416                return (0);
3417        }

3419        printf("not character or block device\n");
3420        error++;
3421        return (1);
3422 }

3424 /*
3425  * nullblk - return error if address is zero.  This is done
3426  *      to prevent block 0 from being used as an indirect block
3427  *      for a large file or as a data block for a small file.
```

```
3428  */
3429 static int
3430 nullblk(long bn)
3431 {
3432         if (bn != 0)
3433                 return (0);
3434         printf("non existent block\n");
3435         error++;
3436         return (1);
3437 }

3439 /*
3440  * puta - put ascii characters into a buffer.  The string
3441  *      terminates with a quote or newline.  The leading quote,
3442  *      which is optional for directory names, was stripped off
3443  *      by the assignment case in the main loop.
3444  */
3445 static void
3446 puta()
3447 {
3448         char            *cptr, c;
3449         int             i;
3450         char            *sbptr;
3451         short           terror = 0;
3452         long            maxchars, s_err, nbytes, temp;
3453         u_offset_t      taddr = addr;
3454         long            tcount = 0, item, olditem = 0;

3456         if (wrtflag == O_RDONLY) {
3457                 printf("not opened for write '-w'\n");
3458                 error++;
3459                 return;
3460         }
3461         if ((sbptr = getblk(addr)) == 0)
3462                 return;
3463         cptr = sbptr + blkoff(fs, addr);
3464         if (objsz == DIRECTORY) {
3465                 if (acting_on_directory)
3466                         maxchars = stringsize - 1;
3467                 else
3468                         maxchars = LONG;
3469         } else if (objsz == INODE)
3470                 maxchars = objsz - (addr - cur_ino);
3471         else
3472                 maxchars = min(blocksize - cur_bytes, filesize - cur_bytes);
3473         while ((c = getachar()) != '"') {
3474                 if (tcount >= maxchars) {
3475                         printf("string too long\n");
3476                         if (objsz == DIRECTORY)
3477                                 addr = cur_dir;
3478                         else if (acting_on_inode || objsz == INODE)
3479                                 addr = cur_ino;
3480                         else
3481                                 addr = taddr;
3482                         erraddr = addr;
3483                         errcur_bytes = cur_bytes;
3484                         terror++;
3485                         break;
3486                 }
3487                 tcount++;
3488                 if (c == '\n') {
3489                         ungetachar(c);
3490                         break;
3491                 }
3492                 temp = (long)*cptr;
3493                 olditem <<= BITSPERCHAR;
```



```
3494                 olditem += temp & 0xff;
3495                 if (c == '\\') {
3496                         switch (c = getachar()) {
3497                         case 't':
3498                                 *cptr++ = '\t';
3499                                 break;
3500                         case 'n':
3501                                 *cptr++ = '\n';
3502                                 break;
3503                         case '0':
3504                                 *cptr++ = '\0';
3505                                 break;
3506                         default:
3507                                 *cptr++ = c;
3508                                 break;
3509                         }
3510                 }
3511                 else
3512                         *cptr++ = c;
3513         }
3514         if (objsz == DIRECTORY && acting_on_directory)
3515                 for (i = tcount; i <= maxchars; i++)
3516                         *cptr++ = '\0';
3517         if ((s_err = llseek(fd, (offset_t)(addr & fs->fs_bmask), 0)) == -1) {
3518                 error++;
3519                 printf("seek error : %" PRIx64 "\n", addr);
3520                 return;
3521         }
3522         if ((nbytes = write(fd, sbptr, BLKSIZE)) != BLKSIZE) {
3523                 error++;
3524                 printf("write error : addr   = %" PRIx64 "\n", addr);
3525                 printf("            : s_err  = %lx\n", s_err);
3526                 printf("            : nbytes = %lx\n", nbytes);
3527                 return;
3528         }
3529         if (!acting_on_inode && objsz != INODE && objsz != DIRECTORY) {
3530                 addr += tcount;
3531                 cur_bytes += tcount;
3532                 taddr = addr;
3533                 if (objsz != CHAR) {
3534                         addr &= ~(objsz - 1);
3535                         cur_bytes -= taddr - addr;
3536                 }
3537                 if (addr == taddr) {
3538                         addr -= objsz;
3539                         taddr = addr;
3540                 }
3541                 tcount = LONG - (taddr - addr);
3542                 index(base);
3543                 if ((cptr = getblk(addr)) == 0)
3544                         return;
3545                 cptr += blkoff(fs, addr);
3546                 switch (objsz) {
3547                 case LONG:
3548                         /*LINTED*/
3549                         item = *(long *)cptr;
3550                         if (tcount < LONG) {
3551                                 olditem <<= tcount * BITSPERCHAR;
3552                                 temp = 1;
3553                                 for (i = 0; i < (tcount*BITSPERCHAR); i++)
3554                                         temp <<= 1;
3555                                 olditem += item & (temp - 1);
3556                         }
3557                         break;
3558                 case SHORT:
3559                         /*LINTED*/
```

```
3560                        item = (long)*(short *)cptr;
3561                        if (tcount < SHORT) {
3562                                olditem <<= tcount * BITSPERCHAR;
3563                                temp = 1;
3564                                for (i = 0; i < (tcount * BITSPERCHAR); i++)
3565                                        temp <<= 1;
3566                                olditem += item & (temp - 1);
3567                        }
3568                        olditem &= 0177777L;
3569                        break;
3570                case CHAR:
3571                        item = (long)*cptr;
3572                        olditem &= 0377;
3573                }
3574                print(olditem, 8, -8, 0);
3575                printf("\t=\t");
3576                print(item, 8, -8, 0);
3577                printf("\n");
3578        } else {
3579                if (objsz == DIRECTORY) {
3580                        addr = cur_dir;
3581                        fprnt('?', 'd');
3582                } else {
3583                        addr = cur_ino;
3584                        objsz = INODE;
3585                        fprnt('?', 'i');
3586                }
3587        }
3588        if (terror)
3589                error++;
3590 }

3592 /*
3593  * fprnt - print data.  'count' elements are printed where '*' will
3594  *      print an entire blocks worth or up to the eof, whichever
3595  *      occurs first.  An error will occur if crossing a block boundary
3596  *      is attempted since consecutive blocks don't usually have
3597  *      meaning.  Current print types:
3598  *              /               b   - print as bytes (base sensitive)
3599  *                              c   - print as characters
3600  *                              o O - print as octal shorts (longs)
3601  *                              d D - print as decimal shorts (longs)
3602  *                              x X - print as hexadecimal shorts (longs)
3603  *              ?               c   - print as cylinder groups
3604  *                              d   - print as directories
3605  *                              i   - print as inodes
3606  *                              s   - print as super blocks
3607  *                              S   - print as shadow data
3608  */
3609 static void
3610 fprnt(char style, char po)
3611 {
3612        int             i;
3613        struct fs       *sb;
3614        struct cg       *cg;
3615        struct direct   *dirp;
3616        struct dinode   *ip;
3617        int             tbase;
3618        char            c, *cptr, *p;
3619        long            tinode, tcount, temp;
3620        u_offset_t      taddr;
3621        short           offset, mode, end = 0, eof = 0, eof_flag;
3622        unsigned short  *sptr;
3623        unsigned long   *lptr;
3624        offset_t        curoff, curioff;
```



```
3626        laststyle = style;
3627        lastpo = po;
3628        should_print = 0;
3629        if (count != 1) {
3630                if (clear) {
3631                        count = 1;
3632                        star = 0;
3633                        clear = 0;
3634                } else
3635                        clear = 1;
3636        }
3637        tcount = count;
3638        offset = blkoff(fs, addr);

3640        if (style == '/') {
3641                if (type == NUMB)
3642                        eof_flag = 0;
3643                else
3644                        eof_flag = 1;
3645                switch (po) {

3647                case 'c': /* print as characters */
3648                case 'b': /* or bytes */
3649                        if ((cptr = getblk(addr)) == 0)
3650                                return;
3651                        cptr += offset;
3652                        objsz = CHAR;
3653                        tcount = check_addr(eof_flag, &end, &eof, 0);
3654                        if (tcount) {
3655                                for (i = 0; tcount--; i++) {
3656                                        if (i % 16 == 0) {
3657                                                if (i)
3658                                                        printf("\n");
3659                                                index(base);
3660                                        }
3661                                        if (po == 'c') {
3662                                                putf(*cptr++);
3663                                                if ((i + 1) % 16)
3664                                                        printf("  ");
3665                                        } else {
3666                                                if ((i + 1) % 16 == 0)
3667                                                        print(*cptr++ & 0377L,
3668                                                                2, -2, 0);
3669                                                else
3670                                                        print(*cptr++ & 0377L,
3671                                                                4, -2, 0);
3672                                        }
3673                                        addr += CHAR;
3674                                        cur_bytes += CHAR;
3675                                }
3676                                printf("\n");
3677                        }
3678                        addr -= CHAR;
3679                        erraddr = addr;
3680                        cur_bytes -= CHAR;
3681                        errcur_bytes = cur_bytes;
3682                        if (eof) {
3683                                printf("end of file\n");
3684                                error++;
3685                        } else if (end) {
3686                                if (type == BLOCK)
3687                                        printf("end of block\n");
3688                                else
3689                                        printf("end of fragment\n");
3690                                error++;
3691                        }
```

```
3692                        return;
3693
3694                case 'o': /* print as octal shorts */
3695                        tbase = OCTAL;
3696                        goto otx;
3697                case 'd': /* print as decimal shorts */
3698                        tbase = DECIMAL;
3699                        goto otx;
3700                case 'x': /* print as hex shorts */
3701                        tbase = HEX;
3702 otx:
3703                        if ((cptr = getblk(addr)) == 0)
3704                                return;
3705                        taddr = addr;
3706                        addr &= ~(SHORT - 1);
3707                        cur_bytes -= taddr - addr;
3708                        cptr += blkoff(fs, addr);
3709                        /*LINTED*/
3710                        sptr = (unsigned short *)cptr;
3711                        objsz = SHORT;
3712                        tcount = check_addr(eof_flag, &end, &eof, 0);
3713                        if (tcount) {
3714                                for (i = 0; tcount--; i++) {
3715                                        sptr = (unsigned short *)print_check(
3716                                                        /*LINTED*/
3717                                                        (unsigned long *)sptr,
3718                                                        &tcount, tbase, i);
3719                                        switch (po) {
3720                                        case 'o':
3721                                                printf("%06o ", *sptr++);
3722                                                break;
3723                                        case 'd':
3724                                                printf("%05d  ", *sptr++);
3725                                                break;
3726                                        case 'x':
3727                                                printf("%04x   ", *sptr++);
3728                                        }
3729                                        addr += SHORT;
3730                                        cur_bytes += SHORT;
3731                                }
3732                                printf("\n");
3733                        }
3734                        addr -= SHORT;
3735                        erraddr = addr;
3736                        cur_bytes -= SHORT;
3737                        errcur_bytes = cur_bytes;
3738                        if (eof) {
3739                                printf("end of file\n");
3740                                error++;
3741                        } else if (end) {
3742                                if (type == BLOCK)
3743                                        printf("end of block\n");
3744                                else
3745                                        printf("end of fragment\n");
3746                                error++;
3747                        }
3748                        return;
3749
3750                case 'O': /* print as octal longs */
3751                        tbase = OCTAL;
3752                        goto OTX;
3753                case 'D': /* print as decimal longs */
3754                        tbase = DECIMAL;
3755                        goto OTX;
3756                case 'X': /* print as hex longs */
3757                        tbase = HEX;
```



```
3758 OTX:
3759                        if ((cptr = getblk(addr)) == 0)
3760                                return;
3761                        taddr = addr;
3762                        addr &= ~(LONG - 1);
3763                        cur_bytes -= taddr - addr;
3764                        cptr += blkoff(fs, addr);
3765                        /*LINTED*/
3766                        lptr = (unsigned long *)cptr;
3767                        objsz = LONG;
3768                        tcount = check_addr(eof_flag, &end, &eof, 0);
3769                        if (tcount) {
3770                                for (i = 0; tcount--; i++) {
3771                                        lptr = print_check(lptr, &tcount,
3772                                                                tbase, i);
3773                                        switch (po) {
3774                                        case 'O':
3775                                                printf("%011lo    ", *lptr++);
3776                                                break;
3777                                        case 'D':
3778                                                printf("%010lu     ", *lptr++);
3779                                                break;
3780                                        case 'X':
3781                                                printf("%08lx       ", *lptr++);
3782                                        }
3783                                        addr += LONG;
3784                                        cur_bytes += LONG;
3785                                }
3786                                printf("\n");
3787                        }
3788                        addr -= LONG;
3789                        erraddr = addr;
3790                        cur_bytes -= LONG;
3791                        errcur_bytes = cur_bytes;
3792                        if (eof) {
3793                                printf("end of file\n");
3794                                error++;
3795                        } else if (end) {
3796                                if (type == BLOCK)
3797                                        printf("end of block\n");
3798                                else
3799                                        printf("end of fragment\n");
3800                                error++;
3801                        }
3802                        return;
3803
3804                default:
3805                        error++;
3806                        printf("no such print option\n");
3807                        return;
3808                }
3809        } else
3810                switch (po) {
3811
3812                case 'c': /* print as cylinder group */
3813                        if (type != NUMB)
3814                                if (cur_cgrp + count > fs->fs_ncg) {
3815                                        tcount = fs->fs_ncg - cur_cgrp;
3816                                        if (!star)
3817                                                end++;
3818                                }
3819                        addr &= ~(LONG - 1);
3820                        for (/* void */; tcount--; /* void */) {
3821                                erraddr = addr;
3822                                errcur_bytes = cur_bytes;
3823                                if (type != NUMB) {
```

```
3824                                    addr = cgtod(fs, cur_cgrp)
3825                                            << FRGSHIFT;
3826                                    cur_cgrp++;
3827                            }
3828                            if ((cptr = getblk(addr)) == 0) {
3829                                    if (cur_cgrp)
3830                                            cur_cgrp--;
3831                                    return;
3832                            }
3833                            cptr += blkoff(fs, addr);
3834                            /*LINTED*/
3835                            cg = (struct cg *)cptr;
3836                            if (type == NUMB) {
3837                                    cur_cgrp = cg->cg_cgx + 1;
3838                                    type = objsz = CGRP;
3839                                    if (cur_cgrp + count - 1 > fs->fs_ncg) {
3840                                            tcount = fs->fs_ncg - cur_cgrp;
3841                                            if (!star)
3842                                                    end++;
3843                                    }
3844                            }
3845                            if (! override && !cg_chkmagic(cg)) {
3846                                    printf("invalid cylinder group ");
3847                                    printf("magic word\n");
3848                                    if (cur_cgrp)
3849                                            cur_cgrp--;
3850                                    error++;
3851                                    return;
3852                            }
3853                            printcg(cg);
3854                            if (tcount)
3855                                    printf("\n");
3856                    }
3857                    cur_cgrp--;
3858                    if (end) {
3859                            printf("end of cylinder groups\n");
3860                            error++;
3861                    }
3862                    return;

3864            case 'd': /* print as directories */
3865                    if ((cptr = getblk(addr)) == 0)
3866                            return;
3867                    if (type == NUMB) {
3868                            if (fragoff(fs, addr)) {
3869                                    printf("address must be at the ");
3870                                    printf("beginning of a fragment\n");
3871                                    error++;
3872                                    return;
3873                            }
3874                            bod_addr = addr;
3875                            type = FRAGMENT;
3876                            dirslot = 0;
3877                            cur_bytes = 0;
3878                            blocksize = FRGSIZE;
3879                            filesize = FRGSIZE * 2;
3880                    }
3881                    cptr += offset;
3882                    objsz = DIRECTORY;
3883                    while (tcount-- && cur_bytes < filesize &&
3884                            cur_bytes < blocksize && !bcomp(addr)) {
3885                            /*LINTED*/
3886                            dirp = (struct direct *)cptr;
3887                            tinode = dirp->d_ino;
3888                            printf("i#: ");
3889                            if (tinode == 0)
```



```
3890                                    printf("free\t");
3891                            else
3892                                    print(tinode, 12, -8, 0);
3893                            printf("%s\n", &dirp->d_name[0]);
3894                            erraddr = addr;
3895                            errcur_bytes = cur_bytes;
3896                            addr += dirp->d_reclen;
3897                            cptr += dirp->d_reclen;
3898                            cur_bytes += dirp->d_reclen;
3899                            dirslot++;
3900                            stringsize = STRINGSIZE(dirp);
3901                    }
3902                    addr = erraddr;
3903                    cur_dir = addr;
3904                    cur_bytes = errcur_bytes;
3905                    dirslot--;
3906                    if (tcount >= 0 && !star) {
3907                            switch (type) {
3908                            case FRAGMENT:
3909                                    printf("end of fragment\n");
3910                                    break;
3911                            case BLOCK:
3912                                    printf("end of block\n");
3913                                    break;
3914                            default:
3915                                    printf("end of directory\n");
3916                            }
3917                            error++;
3918                    } else
3919                            error = 0;
3920                    return;

3922            case 'i': /* print as inodes */
3923                    /*LINTED*/
3924                    if ((ip = (struct dinode *)getblk(addr)) == 0)
3925                            return;
3926                    for (i = 1; i < fs->fs_ncg; i++)
3927                            if (addr < (cgimin(fs, i) << FRGSHIFT))
3928                                    break;
3929                    i--;
3930                    offset /= INODE;
3931                    temp = (addr - (cgimin(fs, i) << FRGSHIFT)) >> FRGSHIFT;
3932                    temp = (i * fs->fs_ipg) + fragstoblks(fs, temp) *
3933                                                    INOPB(fs) + offset;
3934                    if (count + offset > INOPB(fs)) {
3935                            tcount = INOPB(fs) - offset;
3936                            if (!star)
3937                                    end++;
3938                    }
3939                    objsz = INODE;
3940                    ip += offset;
3941                    for (i = 0; tcount--; ip++, temp++) {
3942                            if ((mode = icheck(addr)) == 0)
3943                                    if (!override)
3944                                            continue;
3945                            p = " ugtrwxrwxrwx";

3947                            switch (mode & IFMT) {
3948                            case IFDIR:
3949                                    c = 'd';
3950                                    break;
3951                            case IFCHR:
3952                                    c = 'c';
3953                                    break;
3954                            case IFBLK:
3955                                    c = 'b';
```

```
3956                                    break;
3957                            case IFREG:
3958                                    c = '-';
3959                                    break;
3960                            case IFLNK:
3961                                    c = 'l';
3962                                    break;
3963                            case IFSOCK:
3964                                    c = 's';
3965                                    break;
3966                            case IFSHAD:
3967                                    c = 'S';
3968                                    break;
3969                            case IFATTRDIR:
3970                                    c = 'A';
3971                                    break;
3972                            default:
3973                                    c = '?';
3974                                    if (!override)
3975                                            goto empty;

3977                            }
3978                            printf("i#: ");
3979                            print(temp, 12, -8, 0);
3980                            printf("   md: ");
3981                            printf("%c", c);
3982                            for (mode = mode << 4; *++p; mode = mode << 1) {
3983                                    if (mode & IFREG)
3984                                            printf("%c", *p);
3985                                    else
3986                                            printf("-");
3987                            }
3988                            printf("  uid: ");
3989                            print(ip->di_uid, 8, -4, 0);
3990                            printf("      gid: ");
3991                            print(ip->di_gid, 8, -4, 0);
3992                            printf("\n");
3993                            printf("ln: ");
3994                            print((long)ip->di_nlink, 8, -4, 0);
3995                            printf("       bs: ");
3996                            print(ip->di_blocks, 12, -8, 0);
3997                            printf("c_flags : ");
3998                            print(ip->di_cflags, 12, -8, 0);
3999                            printf("   sz : ");
4000 #ifdef _LARGEFILE64_SOURCE
4001                            printll(ip->di_size, 20, -16, 0);
4002 #else /* !_LARGEFILE64_SOURCE */
4003                            print(ip->di_size, 12, -8, 0);
4004 #endif /* _LARGEFILE64_SOURCE */
4005                            if (ip->di_shadow) {
4006                                    printf("   si: ");
4007                                    print(ip->di_shadow, 12, -8, 0);
4008                            }
4009                            printf("\n");
4010                            if (ip->di_oeftflag) {
4011                                    printf("ai: ");
4012                                    print(ip->di_oeftflag, 12, -8, 0);
4013                                    printf("\n");
4014                            }
4015                            printf("\n");
4016                            switch (ip->di_mode & IFMT) {
4017                            case IFBLK:
4018                            case IFCHR:
4019                                    printf("maj: ");
4020                                    print(major(ip->di_ordev), 4, -2, 0);
4021                                    printf("  min: ");
```



```
4022                                    print(minor(ip->di_ordev), 4, -2, 0);
4023                                    printf("\n");
4024                                    break;
4025                            default:
4026                                    /*
4027                                     * only display blocks below the
4028                                     * current file size
4029                                     */
4030                                    curoff = 0LL;
4031                                    for (i = 0; i < NDADDR; ) {
4032                                            if (ip->di_size <= curoff)
4033                                                    break;
4034                                            printf("db#%x: ", i);
4035                                            print(ip->di_db[i], 11, -8, 0);

4037                                            if (++i % 4 == 0)
4038                                                    printf("\n");
4039                                            else
4040                                                    printf("  ");
4041                                            curoff += fs->fs_bsize;
4042                                    }
4043                                    if (i % 4)
4044                                            printf("\n");

4046                                    /*
4047                                     * curioff keeps track of the number
4048                                     * of bytes covered by each indirect
4049                                     * pointer in the inode, and is added
4050                                     * to curoff each time to get the
4051                                     * actual offset into the file.
4052                                     */
4053                                    curioff = fs->fs_bsize *
4054                                        (fs->fs_bsize / sizeof (daddr_t));
4055                                    for (i = 0; i < NIADDR; i++) {
4056                                            if (ip->di_size <= curoff)
4057                                                    break;
4058                                            printf("ib#%x: ", i);
4059                                            print(ip->di_ib[i], 11, -8, 0);
4060                                            printf("  ");
4061                                            curoff += curioff;
4062                                            curioff *= (fs->fs_bsize /
4063                                                sizeof (daddr_t));
4064                                    }
4065                                    if (i)
4066                                            printf("\n");
4067                                    break;
4068                            }
4069                            if (count == 1) {
4070                                    time_t t;

4072                                    t = ip->di_atime;
4073                                    printf("\taccessed: %s", ctime(&t));
4074                                    t = ip->di_mtime;
4075                                    printf("\tmodified: %s", ctime(&t));
4076                                    t = ip->di_ctime;
4077                                    printf("\tcreated : %s", ctime(&t));
4078                            }
4079                            if (tcount)
4080                                    printf("\n");
4081 empty:
4082                            if (c == '?' && !override) {
4083                                    printf("i#: ");
4084                                    print(temp, 12, -8, 0);
4085                                    printf("  is unallocated\n");
4086                                    if (count != 1)
4087                                            printf("\n");
```

```
4088                            }
4089                            cur_ino = erraddr = addr;
4090                            errcur_bytes = cur_bytes;
4091                            cur_inum++;
4092                            addr = addr + INODE;
4093                    }
4094                    addr = erraddr;
4095                    cur_bytes = errcur_bytes;
4096                    cur_inum--;
4097                    if (end) {
4098                            printf("end of block\n");
4099                            error++;
4100                    }
4101                    return;

4103            case 's': /* print as super block */
4104                    if (cur_cgrp == -1) {
4105                            addr = SBLOCK * DEV_BSIZE;
4106                            type = NUMB;
4107                    }
4108                    addr &= ~(LONG - 1);
4109                    if (type != NUMB)
4110                            if (cur_cgrp + count > fs->fs_ncg) {
4111                                    tcount = fs->fs_ncg - cur_cgrp;
4112                                    if (!star)
4113                                            end++;
4114                            }
4115                    for (/* void */; tcount--; /* void */) {
4116                            erraddr = addr;
4117                            cur_bytes = errcur_bytes;
4118                            if (type != NUMB) {
4119                                    addr = cgsblock(fs, cur_cgrp)
4120                                                    << FRGSHIFT;
4121                                    cur_cgrp++;
4122                            }
4123                            if ((cptr = getblk(addr)) == 0) {
4124                                    if (cur_cgrp)
4125                                            cur_cgrp--;
4126                                    return;
4127                            }
4128                            cptr += blkoff(fs, addr);
4129                            /*LINTED*/
4130                            sb = (struct fs *)cptr;
4131                            if (type == NUMB) {
4132                                    for (i = 0; i < fs->fs_ncg; i++)
4133                                            if (addr == cgsblock(fs, i) <<
4134                                                            FRGSHIFT)
4135                                                    break;
4136                                    if (i == fs->fs_ncg)
4137                                            cur_cgrp = 0;
4138                                    else
4139                                            cur_cgrp = i + 1;
4140                                    type = objsz = SB;
4141                                    if (cur_cgrp + count - 1 > fs->fs_ncg) {
4142                                            tcount = fs->fs_ncg - cur_cgrp;
4143                                            if (!star)
4144                                                    end++;
4145                                    }
4146                            }
4147                            if ((sb->fs_magic != FS_MAGIC) &&
4148                                (sb->fs_magic != MTB_UFS_MAGIC)) {
4149                                    cur_cgrp = 0;
4150                                    if (!override) {
4151                                            printf("invalid super block ");
4152                                            printf("magic word\n");
4153                                            cur_cgrp--;
```



```
4154                                            error++;
4155                                            return;
4156                                    }
4157                            }
4158                            if (sb->fs_magic == FS_MAGIC &&
4159                                (sb->fs_version !=
4160                                    UFS_EFISTYLE4NONEFI_VERSION_2 &&
4161                                sb->fs_version != UFS_VERSION_MIN)) {
4162                                    cur_cgrp = 0;
4163                                    if (!override) {
4164                                            printf("invalid super block ");
4165                                            printf("version number\n");
4166                                            cur_cgrp--;
4167                                            error++;
4168                                            return;
4169                                    }
4170                            }
4171                            if (sb->fs_magic == MTB_UFS_MAGIC &&
4172                                (sb->fs_version > MTB_UFS_VERSION_1 ||
4173                                sb->fs_version < MTB_UFS_VERSION_MIN)) {
4174                                    cur_cgrp = 0;
4175                                    if (!override) {
4176                                            printf("invalid super block ");
4177                                            printf("version number\n");
4178                                            cur_cgrp--;
4179                                            error++;
4180                                            return;
4181                                    }
4182                            }
4183                            if (cur_cgrp == 0)
4184                                    printf("\tsuper block:\n");
4185                            else {
4186                                    printf("\tsuper block in cylinder ");
4187                                    printf("group ");
4188                                    print(cur_cgrp - 1, 0, 0, 0);
4189                                    printf(":\n");
4190                            }
4191                            printsb(sb);
4192                            if (tcount)
4193                                    printf("\n");
4194                    }
4195                    cur_cgrp--;
4196                    if (end) {
4197                            printf("end of super blocks\n");
4198                            error++;
4199                    }
4200                    return;

4202            case 'S': /* print as shadow data */
4203                    if (type == NUMB) {
4204                            type = FRAGMENT;
4205                            cur_shad = 0;
4206                            cur_bytes = fragoff(fs, addr);
4207                            bod_addr = addr - cur_bytes;
4208                            /* no more than two fragments */
4209                            filesize = fragroundup(fs,
4210                                bod_addr + FRGSIZE + 1);
4211                    }
4212                    objsz = SHADOW_DATA;
4213                    while (tcount-- &&
4214                        (cur_bytes + SHADOW_DATA) <= filesize &&
4215                        (type != SHADOW_DATA ||
4216                        (cur_bytes + SHADOW_DATA)) <= blocksize) {
4217                            /*LINTED*/
4218                            struct ufs_fsd fsd;
4219                            long tcur_bytes;
```

```
4221                                    taddr = addr;
4222                                    tcur_bytes = cur_bytes;
4223                                    index(base);
4224                                    getshadowdata((long *)&fsd, LONG + LONG);
4225                                    printf("  type: ");
4226                                    print((long)fsd.fsd_type, 8, -8, 0);
4227                                    printf("  size: ");
4228                                    print((long)fsd.fsd_size, 8, -8, 0);
4229                                    tbase = fsd.fsd_size - LONG - LONG;
4230                                    if (tbase > 256)
4231                                            tbase = 256;
4232                                    for (i = 0; i < tbase; i++) {
4233                                            if (i % LONG == 0) {
4234                                                    if (i % 16 == 0) {
4235                                                            printf("\n");
4236                                                            index(base);
4237                                                    } else
4238                                                            printf("  ");
4239                                                    getshadowdata(&temp, LONG);
4240                                                    p = (char *)&temp;
4241                                            } else
4242                                                    printf(" ");
4243                                            printf("%02x", (int)(*p++ & 0377L));
4244                                    }
4245                                    printf("\n");
4246                                    addr = taddr;
4247                                    cur_bytes = tcur_bytes;
4248                                    erraddr = addr;
4249                                    errcur_bytes = cur_bytes;
4250                                    addr += FSD_RECSZ((&fsd), fsd.fsd_size);
4251                                    cur_bytes += FSD_RECSZ((&fsd), fsd.fsd_size);
4252                                    cur_shad++;
4253                                    syncshadowscan(0);
4254                            }
4255                            addr = erraddr;
4256                            cur_bytes = errcur_bytes;
4257                            cur_shad--;
4258                            if (tcount >= 0 && !star) {
4259                                    switch (type) {
4260                                    case FRAGMENT:
4261                                            printf("end of fragment\n");
4262                                            break;
4263                                    default:
4264                                            printf("end of shadow data\n");
4265                                    }
4266                                    error++;
4267                            } else
4268                                    error = 0;
4269                            return;
4270                    default:
4271                            error++;
4272                            printf("no such print option\n");
4273                            return;
4274                    }
4275 }

4277 /*
4278  * valid_addr - call check_addr to validate the current address.
4279  */
4280 static int
4281 valid_addr()
4282 {
4283         short   end = 0, eof = 0;
4284         long    tcount = count;
```



```
4286         if (!trapped)
4287                 return (1);
4288         if (cur_bytes < 0) {
4289                 cur_bytes = 0;
4290                 if (blocksize > filesize) {
4291                         printf("beginning of file\n");
4292                 } else {
4293                         if (type == BLOCK)
4294                                 printf("beginning of block\n");
4295                         else
4296                                 printf("beginning of fragment\n");
4297                 }
4298                 error++;
4299                 return (0);
4300         }
4301         count = 1;
4302         (void) check_addr(1, &end, &eof, (filesize < blocksize));
4303         count = tcount;
4304         if (eof) {
4305                 printf("end of file\n");
4306                 error++;
4307                 return (0);
4308         }
4309         if (end == 2) {
4310                 if (erraddr > addr) {
4311                         if (type == BLOCK)
4312                                 printf("beginning of block\n");
4313                         else
4314                                 printf("beginning of fragment\n");
4315                         error++;
4316                         return (0);
4317                 }
4318         }
4319         if (end) {
4320                 if (type == BLOCK)
4321                         printf("end of block\n");
4322                 else
4323                         printf("end of fragment\n");
4324                 error++;
4325                 return (0);
4326         }
4327         return (1);
4328 }

4330 /*
4331  * check_addr - check if the address crosses the end of block or
4332  *      end of file.  Return the proper count.
4333  */
4334 static int
4335 check_addr(short eof_flag, short *end, short *eof, short keep_on)
4336 {
4337         long    temp, tcount = count, tcur_bytes = cur_bytes;
4338         u_offset_t      taddr = addr;

4340         if (bcomp(addr + count * objsz - 1) ||
4341             (keep_on && taddr < (bmap(cur_block) << FRGSHIFT))) {
4342                 error = 0;
4343                 addr = taddr;
4344                 cur_bytes = tcur_bytes;
4345                 if (keep_on) {
4346                         if (addr < erraddr) {
4347                                 if (cur_bytes < 0) {
4348                                         (*end) = 2;
4349                                         return (0);     /* Value ignored */
4350                                 }
4351                                 temp = cur_block - lblkno(fs, cur_bytes);
```

```
4352                                cur_block -= temp;
4353                                if ((addr = bmap(cur_block) << FRGSHIFT) == 0) {
4354                                        cur_block += temp;
4355                                        return (0);     /* Value ignored */
4356                                }
4357                                temp = tcur_bytes - cur_bytes;
4358                                addr += temp;
4359                                cur_bytes += temp;
4360                                return (0);     /* Value ignored */
4361                        } else {
4362                                if (cur_bytes >= filesize) {
4363                                        (*eof)++;
4364                                        return (0);     /* Value ignored */
4365                                }
4366                                temp = lblkno(fs, cur_bytes) - cur_block;
4367                                cur_block += temp;
4368                                if ((addr = bmap(cur_block) << FRGSHIFT) == 0) {
4369                                        cur_block -= temp;
4370                                        return (0);     /* Value ignored */
4371                                }
4372                                temp = tcur_bytes - cur_bytes;
4373                                addr += temp;
4374                                cur_bytes += temp;
4375                                return (0);     /* Value ignored */
4376                        }
4377                }
4378                tcount = (blkroundup(fs, addr+1)-addr) / objsz;
4379                if (!star)
4380                        (*end) = 2;
4381        }
4382        addr = taddr;
4383        cur_bytes = tcur_bytes;
4384        if (eof_flag) {
4385                if (blocksize > filesize) {
4386                        if (cur_bytes >= filesize) {
4387                                tcount = 0;
4388                                (*eof)++;
4389                        } else if (tcount > (filesize - cur_bytes) / objsz) {
4390                                tcount = (filesize - cur_bytes) / objsz;
4391                                if (!star || tcount == 0)
4392                                        (*eof)++;
4393                        }
4394                } else {
4395                        if (cur_bytes >= blocksize) {
4396                                tcount = 0;
4397                                (*end)++;
4398                        } else if (tcount > (blocksize - cur_bytes) / objsz) {
4399                                tcount = (blocksize - cur_bytes) / objsz;
4400                                if (!star || tcount == 0)
4401                                        (*end)++;
4402                        }
4403                }
4404        }
4405        return (tcount);
4406 }

4408 /*
4409  * print_check - check if the index needs to be printed and delete
4410  *      rows of zeros from the output.
4411  */
4412 unsigned long *
4413 print_check(unsigned long *lptr, long *tcount, short tbase, int i)
4414 {
4415        int             j, k, temp = BYTESPERLINE / objsz;
4416        short           first_time = 0;
4417        unsigned long   *tlptr;
```



```
4418        unsigned short  *tsptr, *sptr;

4420        sptr = (unsigned short *)lptr;
4421        if (i == 0)
4422                first_time = 1;
4423        if (i % temp == 0) {
4424                if (*tcount >= temp - 1) {
4425                        if (objsz == SHORT)
4426                                tsptr = sptr;
4427                        else
4428                                tlptr = lptr;
4429                        k = *tcount - 1;
4430                        for (j = i; k--; j++)
4431                                if (objsz == SHORT) {
4432                                        if (*tsptr++ != 0)
4433                                                break;
4434                                } else {
4435                                        if (*tlptr++ != 0)
4436                                                break;
4437                                }
4438                        if (j > (i + temp - 1)) {
4439                                j = (j - i) / temp;
4440                                while (j-- > 0) {
4441                                        if (objsz == SHORT)
4442                                                sptr += temp;
4443                                        else
4444                                                lptr += temp;
4445                                        *tcount -= temp;
4446                                        i += temp;
4447                                        addr += BYTESPERLINE;
4448                                        cur_bytes += BYTESPERLINE;
4449                                }
4450                                if (first_time)
4451                                        printf("*");
4452                                else
4453                                        printf("\n*");
4454                        }
4455                        if (i)
4456                                printf("\n");
4457                        index(tbase);
4458                } else {
4459                        if (i)
4460                                printf("\n");
4461                        index(tbase);
4462                }
4463        }
4464        if (objsz == SHORT)
4465                /*LINTED*/
4466                return ((unsigned long *)sptr);
4467        else
4468                return (lptr);
4469 }

4471 /*
4472  * index - print a byte index for the printout in base b
4473  *      with leading zeros.
4474  */
4475 static void
4476 index(int b)
4477 {
4478        int     tbase = base;

4480        base = b;
4481        print(addr, 8, 8, 1);
4482        printf(":\t");
4483        base = tbase;
```

```
}

/*
 * print - print out the value to digits places with/without
 *      leading zeros and right/left justified in the current base.
 */
static void
#ifdef _LARGEFILE64_SOURCE
printll(u_offset_t value, int fieldsz, int digits, int lead)
#else /* !_LARGEFILE64_SOURCE */
print(long value, int fieldsz, int digits, int lead)
#endif /* _LARGEFILE64_SOURCE */
{
        int     i, left = 0;
        char    mode = BASE[base - OCTAL];
        char    *string = &scratch[0];

        if (digits < 0) {
                left = 1;
                digits *= -1;
        }
        if (base != HEX)
                if (digits)
                        digits = digits + (digits - 1)/((base >> 1) - 1) + 1;
                else
                        digits = 1;
        if (lead) {
                if (left)
                        (void) sprintf(string, "%%%c%d%d.%d"
#ifdef _LARGEFILE64_SOURCE
                                "ll"
#endif /* _LARGEFILE64_SOURCE */
                                "%c", '-', 0, digits, lead, mode);
                else
                        (void) sprintf(string, "%%%d%d.%d"
#ifdef _LARGEFILE64_SOURCE
                                "ll"
#endif /* _LARGEFILE64_SOURCE */
                                "%c", 0, digits, lead, mode);
        } else {
                if (left)
                        (void) sprintf(string, "%%%c%d"
#ifdef _LARGEFILE64_SOURCE
                                "ll"
#endif /* _LARGEFILE64_SOURCE */
                                "%c", '-', digits, mode);
                else
                        (void) sprintf(string, "%%%d"
#ifdef _LARGEFILE64_SOURCE
                                "ll"
#endif /* _LARGEFILE64_SOURCE */
                                "%c", digits, mode);
        }
        printf(string, value);
        for (i = 0; i < fieldsz - digits; i++)
                printf(" ");
}

/*
 * Print out the contents of a superblock.
 */
static void
printsb(struct fs *fs)
{
        int c, i, j, k, size;
        caddr_t sip;
```



```
        time_t t;

        t = fs->fs_time;
#ifdef FS_42POSTBLFMT
        if (fs->fs_postblformat == FS_42POSTBLFMT)
                fs->fs_nrpos = 8;
        printf("magic\t%lx\tformat\t%s\ttime\t%s", fs->fs_magic,
            fs->fs_postblformat == FS_42POSTBLFMT ? "static" : "dynamic",
            ctime(&t));
#else
        printf("magic\t%x\ttime\t%s",
            fs->fs_magic, ctime(&t));
#endif
        printf("version\t%x\n", fs->fs_version);
        printf("nbfree\t%ld\tndir\t%ld\tnifree\t%ld\tnffree\t%ld\n",
            fs->fs_cstotal.cs_nbfree, fs->fs_cstotal.cs_ndir,
            fs->fs_cstotal.cs_nifree, fs->fs_cstotal.cs_nffree);
        printf("ncg\t%ld\tncyl\t%ld\tsize\t%ld\tblocks\t%ld\n",
            fs->fs_ncg, fs->fs_ncyl, fs->fs_size, fs->fs_dsize);
        printf("bsize\t%ld\tshift\t%ld\tmask\t0x%08lx\n",
            fs->fs_bsize, fs->fs_bshift, fs->fs_bmask);
        printf("fsize\t%ld\tshift\t%ld\tmask\t0x%08lx\n",
            fs->fs_fsize, fs->fs_fshift, fs->fs_fmask);
        printf("frag\t%ld\tshift\t%ld\tfsbtodb\t%ld\n",
            fs->fs_frag, fs->fs_fragshift, fs->fs_fsbtodb);
        printf("cpg\t%ld\tbpg\t%ld\tfpg\t%ld\tipg\t%ld\n",
            fs->fs_cpg, fs->fs_fpg / fs->fs_frag, fs->fs_fpg, fs->fs_ipg);
        printf("minfree\t%ld%%\toptim\t%s\tmaxcontig %ld\tmaxbpg\t%ld\n",
            fs->fs_minfree, fs->fs_optim == FS_OPTSPACE ? "space" : "time",
            fs->fs_maxcontig, fs->fs_maxbpg);
#ifdef FS_42POSTBLFMT
#ifdef sun
        printf("rotdelay %ldms\tfs_id[0] 0x%lx\tfs_id[1] 0x%lx\trps\t%ld\n",
            fs->fs_rotdelay, fs->fs_id[0], fs->fs_id[1], fs->fs_rps);
#else
        printf("rotdelay %dms\theadswitch %dus\ttrackseek %dus\trps\t%d\n",
            fs->fs_rotdelay, fs->fs_headswitch, fs->fs_trkseek, fs->fs_rps);
#endif /* sun */
        printf("ntrak\t%ld\tnsect\t%ld\tnpsect\t%ld\tspc\t%ld\n",
            fs->fs_ntrak, fs->fs_nsect, fs->fs_npsect, fs->fs_spc);
        printf("trackskew %ld\n", fs->fs_trackskew);
#else
        printf("rotdelay %ldms\trps\t%ld\n",
            fs->fs_rotdelay, fs->fs_rps);
        printf("ntrak\t%ld\tnsect\t%ld\tspc\t%ld\n",
            fs->fs_ntrak, fs->fs_nsect, fs->fs_spc);
#endif
        printf("si %ld\n", fs->fs_si);
        printf("nindir\t%ld\tinopb\t%ld\tnspf\t%ld\n",
            fs->fs_nindir, fs->fs_inopb, fs->fs_nspf);
        printf("sblkno\t%ld\tcblkno\t%ld\tiblkno\t%ld\tdblkno\t%ld\n",
            fs->fs_sblkno, fs->fs_cblkno, fs->fs_iblkno, fs->fs_dblkno);
        printf("sbsize\t%ld\tcgsize\t%ld\tcgoffset %ld\tcgmask\t0x%08lx\n",
            fs->fs_sbsize, fs->fs_cgsize, fs->fs_cgoffset, fs->fs_cgmask);
        printf("csaddr\t%ld\tcssize\t%ld\tshift\t%ld\tmask\t0x%08lx\n",
            fs->fs_csaddr, fs->fs_cssize, fs->fs_csshift, fs->fs_csmask);
        printf("cgrotor\t%ld\tfmod\t%d\tronly\t%d\n",
            fs->fs_cgrotor, fs->fs_fmod, fs->fs_ronly);
#ifdef FS_42POSTBLFMT
        if (fs->fs_cpc != 0)
                printf("blocks available in each of %ld rotational positions",
                        fs->fs_nrpos);
        else
                printf("insufficient space to maintain rotational tables\n");
#endif
        for (c = 0; c < fs->fs_cpc; c++) {
```

```
4616                printf("\ncylinder number %d:", c);
4617 #ifdef FS_42POSTBLFMT
4618                for (i = 0; i < fs->fs_nrpos; i++) {
4619                        /*LINTED*/
4620                        if (fs_postbl(fs, c)[i] == -1)
4621                                continue;
4622                        printf("\n   position %d:\t", i);
4623                        /*LINTED*/
4624                        for (j = fs_postbl(fs, c)[i], k = 1; /* void */;
4625                                                j += fs_rotbl(fs)[j], k++) {
4626                                printf("%5d", j);
4627                                if (k % 12 == 0)
4628                                        printf("\n\t\t");
4629                                if (fs_rotbl(fs)[j] == 0)
4630                                        break;
4631                        }
4632                }
4633 #else
4634                for (i = 0; i < NRPOS; i++) {
4635                        if (fs->fs_postbl[c][i] == -1)
4636                                continue;
4637                        printf("\n   position %d:\t", i);
4638                        for (j = fs->fs_postbl[c][i], k = 1; /* void */;
4639                                                j += fs->fs_rotbl[j], k++) {
4640                                printf("%5d", j);
4641                                if (k % 12 == 0)
4642                                        printf("\n\t\t");
4643                                if (fs->fs_rotbl[j] == 0)
4644                                        break;
4645                        }
4646                }
4647 #endif
4648        }
4649        printf("\ncs[].cs_(nbfree, ndir, nifree, nffree):");
4650        sip = calloc(1, fs->fs_cssize);
4651        fs->fs_u.fs_csp = (struct csum *)sip;
4652        for (i = 0, j = 0; i < fs->fs_cssize; i += fs->fs_bsize, j++) {
4653                size = fs->fs_cssize - i < fs->fs_bsize ?
4654                    fs->fs_cssize - i : fs->fs_bsize;
4655                (void) llseek(fd,
4656                        (offset_t)fsbtodb(fs, (fs->fs_csaddr + j * fs->fs_frag))
4657                            * fs->fs_fsize / fsbtodb(fs, 1), 0);
4658                if (read(fd, sip, size) != size) {
4659                        free(fs->fs_u.fs_csp);
4660                        return;
4661                }
4662                sip += size;
4663        }
4664        for (i = 0; i < fs->fs_ncg; i++) {
4665                struct csum *cs = &fs->fs_cs(fs, i);
4666                if (i % 4 == 0)
4667                        printf("\n     ");
4668                printf("%d:(%ld,%ld,%ld,%ld) ", i, cs->cs_nbfree, cs->cs_ndir,
4669                                                cs->cs_nifree, cs->cs_nffree);
4670        }
4671        free(fs->fs_u.fs_csp);
4672        printf("\n");
4673        if (fs->fs_ncyl % fs->fs_cpg) {
4674                printf("cylinders in last group %d\n",
4675                    i = fs->fs_ncyl % fs->fs_cpg);
4676                printf("blocks in last group %ld\n",
4677                    i * fs->fs_spc / NSPB(fs));
4678        }
4679 }

4681 /*
```



```
4682  * Print out the contents of a cylinder group.
4683  */
4684 static void
4685 printcg(struct cg *cg)
4686 {
4687        int i, j;
4688        time_t t;

4690        printf("\ncg %ld:\n", cg->cg_cgx);
4691        t = cg->cg_time;
4692 #ifdef FS_42POSTBLFMT
4693        printf("magic\t%lx\ttell\t%llx\ttime\t%s",
4694            fs->fs_postblformat == FS_42POSTBLFMT ?
4695            ((struct ocg *)cg)->cg_magic : cg->cg_magic,
4696            fsbtodb(fs, cgtod(fs, cg->cg_cgx)) * fs->fs_fsize / fsbtodb(fs, 1),
4697            ctime(&t));
4698 #else
4699        printf("magic\t%x\ttell\t%llx\ttime\t%s",
4700            cg->cg_magic,
4701            fsbtodb(fs, cgtod(fs, cg->cg_cgx)) * fs->fs_fsize / fsbtodb(fs, 1),
4702            ctime(&t));
4703 #endif
4704        printf("cgx\t%ld\tncyl\t%d\tniblk\t%d\tndblk\t%ld\n",
4705            cg->cg_cgx, cg->cg_ncyl, cg->cg_niblk, cg->cg_ndblk);
4706        printf("nbfree\t%ld\tndir\t%ld\tnifree\t%ld\tnffree\t%ld\n",
4707            cg->cg_cs.cs_nbfree, cg->cg_cs.cs_ndir,
4708            cg->cg_cs.cs_nifree, cg->cg_cs.cs_nffree);
4709        printf("rotor\t%ld\tirotor\t%ld\tfrotor\t%ld\nfrsum",
4710            cg->cg_rotor, cg->cg_irotor, cg->cg_frotor);
4711        for (i = 1, j = 0; i < fs->fs_frag; i++) {
4712                printf("\t%ld", cg->cg_frsum[i]);
4713                j += i * cg->cg_frsum[i];
4714        }
4715        printf("\nsum of frsum: %d\niused:\t", j);
4716        pbits((unsigned char *)cg_inosused(cg), fs->fs_ipg);
4717        printf("free:\t");
4718        pbits(cg_blksfree(cg), fs->fs_fpg);
4719        printf("b:\n");
4720        for (i = 0; i < fs->fs_cpg; i++) {
4721                /*LINTED*/
4722                if (cg_blktot(cg)[i] == 0)
4723                        continue;
4724                /*LINTED*/
4725                printf("   c%d:\t(%ld)\t", i, cg_blktot(cg)[i]);
4726 #ifdef FS_42POSTBLFMT
4727                for (j = 0; j < fs->fs_nrpos; j++) {
4728                        if (fs->fs_cpc == 0 ||
4729                                /*LINTED*/
4730                            fs_postbl(fs, i % fs->fs_cpc)[j] == -1)
4731                                continue;
4732                        /*LINTED*/
4733                        printf(" %d", cg_blks(fs, cg, i)[j]);
4734                }
4735 #else
4736                for (j = 0; j < NRPOS; j++) {
4737                        if (fs->fs_cpc == 0 ||
4738                            fs->fs_postbl[i % fs->fs_cpc][j] == -1)
4739                                continue;
4740                        printf(" %d", cg->cg_b[i][j]);
4741                }
4742 #endif
4743                printf("\n");
4744        }
4745 }

4747 /*
```

```
4748  * Print out the contents of a bit array.
4749  */
4750 static void
4751 pbits(unsigned char *cp, int max)
4752 {
4753         int i;
4754         int count = 0, j;

4756         for (i = 0; i < max; i++)
4757                 if (isset(cp, i)) {
4758                         if (count)
4759                                 printf(",%s", count % 6 ? " " : "\n\t");
4760                         count++;
4761                         printf("%d", i);
4762                         j = i;
4763                         while ((i+1) < max && isset(cp, i+1))
4764                                 i++;
4765                         if (i != j)
4766                                 printf("-%d", i);
4767                 }
4768         printf("\n");
4769 }

4771 /*
4772  * bcomp - used to check for block over/under flows when stepping through
4773  *      a file system.
4774  */
4775 static int
4776 bcomp(addr)
4777         u_offset_t      addr;
4778 {
4779         if (override)
4780                 return (0);

4782         if (lblkno(fs, addr) == (bhdr.fwd)->blkno)
4783                 return (0);
4784         error++;
4785         return (1);
4786 }

4788 /*
4789  * bmap - maps the logical block number of a file into
4790  *      the corresponding physical block on the file
4791  *      system.
4792  */
4793 static long
4794 bmap(long bn)
4795 {
4796         int             j;
4797         struct dinode   *ip;
4798         int             sh;
4799         long            nb;
4800         char            *cptr;

4802         if ((cptr = getblk(cur_ino)) == 0)
4803                 return (0);

4805         cptr += blkoff(fs, cur_ino);

4807         /*LINTED*/
4808         ip = (struct dinode *)cptr;

4810         if (bn < NDADDR) {
4811                 nb = ip->di_db[bn];
4812                 return (nullblk(nb) ? 0L : nb);
4813         }
```



```
4815         sh = 1;
4816         bn -= NDADDR;
4817         for (j = NIADDR; j > 0; j--) {
4818                 sh *= NINDIR(fs);
4819                 if (bn < sh)
4820                         break;
4821                 bn -= sh;
4822         }
4823         if (j == 0) {
4824                 printf("file too big\n");
4825                 error++;
4826                 return (0L);
4827         }
4828         addr = (uintptr_t)&ip->di_ib[NIADDR - j];
4829         nb = get(LONG);
4830         if (nb == 0)
4831                 return (0L);
4832         for (; j <= NIADDR; j++) {
4833                 sh /= NINDIR(fs);
4834                 addr = (nb << FRGSHIFT) + ((bn / sh) % NINDIR(fs)) * LONG;
4835                 if (nullblk(nb = get(LONG)))
4836                         return (0L);
4837         }
4838         return (nb);
4839 }

4841 #if defined(OLD_FSDB_COMPATIBILITY)

4843 /*
4844  * The following are "tacked on" to support the old fsdb functionality
4845  * of clearing an inode. (All together now...) "It's better to use clri".
4846  */

4848 #define ISIZE   (sizeof (struct dinode))
4849 #define NI      (MAXBSIZE/ISIZE)


4852 static struct   dinode  di_buf[NI];

4854 static union {
4855         char            dummy[SBSIZE];
4856         struct fs       sblk;
4857 } sb_un;

4859 #define sblock sb_un.sblk

4861 static void
4862 old_fsdb(int inum, char *special)
4863 {
4864         int             f;      /* File descriptor for "special" */
4865         int             j;
4866         int             status = 0;
4867         u_offset_t      off;
4868         long            gen;
4869         time_t          t;

4871         f = open(special, 2);
4872         if (f < 0) {
4873                 perror("open");
4874                 printf("cannot open %s\n", special);
4875                 exit(31+4);
4876         }
4877         (void) llseek(f, (offset_t)SBLOCK * DEV_BSIZE, 0);
4878         if (read(f, &sblock, SBSIZE) != SBSIZE) {
4879                 printf("cannot read %s\n", special);
```

```
4880                    exit(31+4);
4881            }
4882            if (sblock.fs_magic != FS_MAGIC) {
4883                    printf("bad super block magic number\n");
4884                    exit(31+4);
4885            }
4886            if (inum == 0) {
4887                    printf("%d: is zero\n", inum);
4888                    exit(31+1);
4889            }
4890            off = (u_offset_t)fsbtodb(&sblock, itod(&sblock, inum)) * DEV_BSIZE;
4891            (void) llseek(f, off, 0);
4892            if (read(f, (char *)di_buf, sblock.fs_bsize) != sblock.fs_bsize) {
4893                    printf("%s: read error\n", special);
4894                    status = 1;
4895            }
4896            if (status)
4897                    exit(31+status);

4899            /*
4900             * Update the time in superblock, so fsck will check this filesystem.
4901             */
4902            (void) llseek(f, (offset_t)(SBLOCK * DEV_BSIZE), 0);
4903            (void) time(&t);
4904            sblock.fs_time = (time32_t)t;
4905            if (write(f, &sblock, SBSIZE) != SBSIZE) {
4906                    printf("cannot update %s\n", special);
4907                    exit(35);
4908            }

4910            printf("clearing %u\n", inum);
4911            off = (u_offset_t)fsbtodb(&sblock, itod(&sblock, inum)) * DEV_BSIZE;
4912            (void) llseek(f, off, 0);
4913            read(f, (char *)di_buf, sblock.fs_bsize);
4914            j = itoo(&sblock, inum);
4915            gen = di_buf[j].di_gen;
4916            (void) memset((caddr_t)&di_buf[j], 0, ISIZE);
4917            di_buf[j].di_gen = gen + 1;
4918            (void) llseek(f, off, 0);
4919            write(f, (char *)di_buf, sblock.fs_bsize);
4920            exit(31+status);
4921    }

4923    static int
4924    isnumber(char *s)
4925    {
4926            register int    c;

4928            if (s == NULL)
4929                    return (0);
4930            while ((c = *s++) != NULL)
4931                    if (c < '0' || c > '9')
4932                            return (0);
4933            return (1);
4934    }
4935    #endif /* OLD_FSDB_COMPATIBILITY */

4937    enum boolean { True, False };
4938    extent_block_t  *log_eb;
4939    ml_odunit_t     *log_odi;
4940    int             lufs_tid;       /* last valid TID seen */

4942    /*
4943     * no single value is safe to use to indicate
4944     * lufs_tid being invalid so we need a
4945     * seperate variable.
```



```
4946     */
4947    enum boolean    lufs_tid_valid;

4949    /*
4950     * log_get_header_info - get the basic info of the logging filesystem
4951     */
4952    int
4953    log_get_header_info(void)
4954    {
4955            char            *b;
4956            int             nb;

4958            /*
4959             * Mark the global tid as invalid everytime we're called to
4960             * prevent any false positive responses.
4961             */
4962            lufs_tid_valid = False;

4964            /*
4965             * See if we've already set up the header areas. The only problem
4966             * with this approach is we don't reread the on disk data though
4967             * it shouldn't matter since we don't operate on a live disk.
4968             */
4969            if ((log_eb != NULL) && (log_odi != NULL))
4970                    return (1);

4972            /*
4973             * Either logging is disabled or we've not running 2.7.
4974             */
4975            if (fs->fs_logbno == 0) {
4976                    printf("Logging doesn't appear to be enabled on this disk\n");
4977                    return (0);
4978            }

4980            /*
4981             * To find the log we need to first pick up the block allocation
4982             * data. The block number for that data is fs_logbno in the
4983             * super block.
4984             */
4985            if ((b = getblk((u_offset_t)ldbtob(logbtodb(fs, fs->fs_logbno))))
4986                == 0) {
4987                    printf("getblk() indicates an error with logging block\n");
4988                    return (0);
4989            }

4991            /*
4992             * Next we need to figure out how big the extent data structure
4993             * really is. It can't be more then fs_bsize and you could just
4994             * allocate that but, why get sloppy.
4995             * 1 is subtracted from nextents because extent_block_t contains
4996             * a single extent_t itself.
4997             */
4998            log_eb = (extent_block_t *)b;
4999            if (log_eb->type != LUFS_EXTENTS) {
5000                    printf("Extents block has invalid type (0x%x)\n",
5001                        log_eb->type);
5002                    return (0);
5003            }
5004            nb = sizeof (extent_block_t) +
5005                (sizeof (extent_t) * (log_eb->nextents - 1));

5007            log_eb = (extent_block_t *)malloc(nb);
5008            if (log_eb == NULL) {
5009                    printf("Failed to allocate memory for extent block log\n");
5010                    return (0);
5011            }
```

```
5012         memcpy(log_eb, b, nb);

5014         if (log_eb->nextbno != 0)
5015                 /*
5016                  * Currently, as of 11-Dec-1997 the field nextbno isn't
5017                  * implemented. If someone starts using this sucker we'd
5018                  * better warn somebody.
5019                  */
5020                 printf("WARNING: extent block field nextbno is non-zero!\n");

5022         /*
5023          * Now read in the on disk log structure. This is always in the
5024          * first block of the first extent.
5025          */
5026         b = getblk((u_offset_t)ldbtob(logbtodb(fs, log_eb->extents[0].pbno)));
5027         log_odi = (ml_odunit_t *)malloc(sizeof (ml_odunit_t));
5028         if (log_odi == NULL) {
5029                 free(log_eb);
5030                 log_eb = NULL;
5031                 printf("Failed to allocate memory for ondisk structure\n");
5032                 return (0);
5033         }
5034         memcpy(log_odi, b, sizeof (ml_odunit_t));

5036         /*
5037          * Consistency checks.
5038          */
5039         if (log_odi->od_version != LUFS_VERSION_LATEST) {
5040                 free(log_eb);
5041                 log_eb = NULL;
5042                 free(log_odi);
5043                 log_odi = NULL;
5044                 printf("Version mismatch in on-disk version of log data\n");
5045                 return (0);
5046         } else if (log_odi->od_badlog) {
5047                 printf("WARNING: Log was marked as bad\n");
5048         }

5050         return (1);
5051 }

5053 static void
5054 log_display_header(void)
5055 {
5056         int x;
5057         if (!log_get_header_info())
5058                 /*
5059                  * No need to display anything here. The previous routine
5060                  * has already done so.
5061                  */
5062                 return;

5064         if (fs->fs_magic == FS_MAGIC)
5065                 printf("Log block number: 0x%x\n------------------\n",
5066                     fs->fs_logbno);
5067         else
5068                 printf("Log frag number: 0x%x\n------------------\n",
5069                     fs->fs_logbno);
5070         printf("Extent Info\n\t# Extents  : %d\n\t# Bytes    : 0x%x\n",
5071             log_eb->nextents, log_eb->nbytes);
5072         printf("\tNext Block : 0x%x\n\tExtent List\n\t--------\n",
5073             log_eb->nextbno);
5074         for (x = 0; x < log_eb->nextents; x++)
5075                 printf("\t  [%d] lbno 0x%08x pbno 0x%08x nbno 0x%08x\n",
5076                     x, log_eb->extents[x].lbno, log_eb->extents[x].pbno,
5077                     log_eb->extents[x].nbno);
```



```
5078         printf("\nOn Disk Info\n\tbol_lof    : 0x%08x\n\teol_lof    : 0x%08x\n",
5079             log_odi->od_bol_lof, log_odi->od_eol_lof);
5080         printf("\tlog_size   : 0x%08x\n",
5081             log_odi->od_logsize);
5082         printf("\thead_lof   : 0x%08x\tident : 0x%x\n",
5083             log_odi->od_head_lof, log_odi->od_head_ident);
5084         printf("\ttail_lof   : 0x%08x\tident : 0x%x\n\thead_tid   : 0x%08x\n",
5085             log_odi->od_tail_lof, log_odi->od_tail_ident, log_odi->od_head_tid);
5086         printf("\tcheck sum  : 0x%08x\n", log_odi->od_chksum);
5087         if (log_odi->od_chksum !=
5088             (log_odi->od_head_ident + log_odi->od_tail_ident))
5089                 printf("bad checksum: found 0x%08x, should be 0x%08x\n",
5090                     log_odi->od_chksum,
5091                     log_odi->od_head_ident + log_odi->od_tail_ident);
5092         if (log_odi->od_head_lof == log_odi->od_tail_lof)
5093                 printf("\t --- Log is empty ---\n");
5094 }

5096 /*
5097  * log_lodb -- logical log offset to disk block number
5098  */
5099 int
5100 log_lodb(u_offset_t off, diskaddr_t *pblk)
5101 {
5102         uint32_t        lblk = (uint32_t)btodb(off);
5103         int     x;

5105         if (!log_get_header_info())
5106                 /*
5107                  * No need to display anything here. The previous routine
5108                  * has already done so.
5109                  */
5110                 return (0);

5112         for (x = 0; x < log_eb->nextents; x++)
5113                 if ((lblk >= log_eb->extents[x].lbno) &&
5114                     (lblk < (log_eb->extents[x].lbno +
5115                         log_eb->extents[x].nbno))) {
5116                         *pblk = (diskaddr_t)lblk - log_eb->extents[x].lbno +
5117                                 logbtodb(fs, log_eb->extents[x].pbno);
5118                         return (1);
5119                 }
5120         return (0);
5121 }

5123 /*
5124  * String names for the enumerated types. These are only used
5125  * for display purposes.
5126  */
5127 char *dt_str[] = {
5128         "DT_NONE", "DT_SB", "DT_CG", "DT_SI", "DT_AB",
5129         "DT_ABZERO", "DT_DIR", "DT_INODE", "DT_FBI",
5130         "DT_QR", "DT_COMMIT", "DT_CANCEL", "DT_BOT",
5131         "DT_EOT", "DT_UD", "DT_SUD", "DT_SHAD", "DT_MAX"
5132 };

5134 /*
5135  * log_read_log -- transfer information from the log and adjust offset
5136  */
5137 int
5138 log_read_log(u_offset_t *addr, caddr_t va, int nb, uint32_t *chk)
5139 {
5140         int             xfer;
5141         caddr_t         bp;
5142         diskaddr_t      pblk;
5143         sect_trailer_t  *st;
```

```
5145         while (nb) {
5146                 if (!log_lodb(*addr, &pblk)) {
5147                         printf("Invalid log offset\n");
5148                         return (0);
5149                 }

5151                 /*
5152                  * fsdb getblk() expects offsets not block number.
5153                  */
5154                 if ((bp = getblk((u_offset_t)dbtob(pblk))) == NULL)
5155                         return (0);

5157                 xfer = MIN(NB_LEFT_IN_SECTOR(*addr), nb);
5158                 if (va != NULL) {
5159                         memcpy(va, bp + blkoff(fs, *addr), xfer);
5160                         va += xfer;
5161                 }
5162                 nb -= xfer;
5163                 *addr += xfer;

5165                 /*
5166                  * If the log offset is now at a sector trailer
5167                  * run the checks if requested.
5168                  */
5169                 if (NB_LEFT_IN_SECTOR(*addr) == 0) {
5170                         if (chk != NULL) {
5171                                 st = (sect_trailer_t *)
5172                                     (bp + blkoff(fs, *addr));
5173                                 if (*chk != st->st_ident) {
5174                                         printf(
5175                 "Expected sector trailer id 0x%08x, but saw 0x%08x\n",
5176                                                 *chk, st->st_ident);
5177                                         return (0);
5178                                 } else {
5179                                         *chk = st->st_ident + 1;
5180                                         /*
5181                                          * We update the on disk structure
5182                                          * transaction ID each time we see
5183                                          * one. By comparing this value
5184                                          * to the last valid DT_COMMIT record
5185                                          * we can determine if our log is
5186                                          * completely valid.
5187                                          */
5188                                         log_odi->od_head_tid = st->st_tid;
5189                                 }
5190                         }
5191                         *addr += sizeof (sect_trailer_t);
5192                 }
5193                 if ((int32_t)*addr == log_odi->od_eol_lof)
5194                         *addr = log_odi->od_bol_lof;
5195         }
5196         return (1);
5197 }

5199 u_offset_t
5200 log_nbcommit(u_offset_t a)
5201 {
5202         /*
5203          * Comments are straight from ufs_log.c
5204          *
5205          * log is the offset following the commit header. However,
5206          * if the commit header fell on the end-of-sector, then lof
5207          * has already been advanced to the beginning of the next
5208          * sector. So do nothgin. Otherwise, return the remaining
5209          * bytes in the sector.
```



```
5210          */
5211         if ((a & (DEV_BSIZE - 1)) == 0)
5212                 return (0);
5213         else
5214                 return (NB_LEFT_IN_SECTOR(a));
5215 }

5217 /*
5218  * log_show --  pretty print the deltas. The number of which is determined
5219  *              by the log_enum arg. If LOG_ALLDELTAS the routine, as the
5220  *              name implies dumps everything. If LOG_NDELTAS, the routine
5221  *              will print out "count" deltas starting at "addr". If
5222  *              LOG_CHECKSCAN then run through the log checking the st_ident
5223  *              for valid data.
5224  */
5225 static void
5226 log_show(enum log_enum l)
5227 {
5228         struct delta    d;
5229         int32_t         bol, eol;
5230         int             x = 0;
5231         uint32_t        chk;

5233         if (!log_get_header_info())
5234                 /*
5235                  * No need to display any error messages here. The previous
5236                  * routine has already done so.
5237                  */
5238                 return;

5240         bol = log_odi->od_head_lof;
5241         eol = log_odi->od_tail_lof;
5242         chk = log_odi->od_head_ident;

5244         if (bol == eol) {
5245                 if ((l == LOG_ALLDELTAS) || (l == LOG_CHECKSCAN)) {
5246                         printf("Empty log.\n");
5247                         return;
5248                 } else
5249                         printf("WARNING: empty log. addr may generate bogus"
5250                             " information");
5251         }

5253         /*
5254          * Only reset the "addr" if we've been requested to show all
5255          * deltas in the log.
5256          */
5257         if ((l == LOG_ALLDELTAS) || (l == LOG_CHECKSCAN))
5258                 addr = (u_offset_t)bol;

5260         if (l != LOG_CHECKSCAN) {
5261                 printf("       Log Offset       Delta       Count     Type\n");
5262                 printf("----------------------------------------"
5263                         "----------------\n");
5264         }

5266         while ((bol != eol) && ((l == LOG_ALLDELTAS) ||
5267             (l == LOG_CHECKSCAN) || count--)) {
5268                 if (!log_read_log(&addr, (caddr_t)&d, sizeof (d),
5269                     ((l == LOG_ALLDELTAS) || (l == LOG_CHECKSCAN)) ?
5270                     &chk : NULL))
5271                         /*
5272                          * Two failures are possible. One from getblk()
5273                          * which prints out a message or when we've hit
5274                          * an invalid block which may or may not indicate
5275                          * an error
```

```
5276                         */
5277                        goto end_scan;

5279                if ((uint32_t)d.d_nb > log_odi->od_logsize) {
5280                        printf("Bad delta entry. size out of bounds\n");
5281                        return;
5282                }
5283                if (l != LOG_CHECKSCAN)
5284                        printf("[%04d]  %08x  %08x.%08x %08x  %s\n", x++, bol,
5285                            d.d_mof, d.d_nb,
5286                            dt_str[d.d_typ >= DT_MAX ? DT_MAX : d.d_typ]);

5288                switch (d.d_typ) {
5289                case DT_CANCEL:
5290                case DT_ABZERO:
5291                        /*
5292                         * These two deltas don't have log space
5293                         * associated with the entry even though
5294                         * d_nb is non-zero.
5295                         */
5296                        break;

5298                case DT_COMMIT:
5299                        /*
5300                         * Commit records have zero size yet, the
5301                         * rest of the current disk block is avoided.
5302                         */
5303                        addr += log_nbcommit(addr);
5304                        lufs_tid = log_odi->od_head_tid;
5305                        lufs_tid_valid = True;
5306                        break;

5308                default:
5309                        if (!log_read_log(&addr, NULL, d.d_nb,
5310                            ((l == LOG_ALLDELTAS) ||
5311                            (l == LOG_CHECKSCAN)) ? &chk : NULL))
5312                                goto end_scan;
5313                        break;
5314                }
5315                bol = (int32_t)addr;
5316        }

5318 end_scan:
5319        if (lufs_tid_valid == True) {
5320                if (lufs_tid == log_odi->od_head_tid)
5321                        printf("scan -- okay\n");
5322                else
5323                        printf("scan -- some transactions have been lost\n");
5324        } else {
5325                printf("scan -- failed to find a single valid transaction\n");
5326                printf("        (possibly due to an empty log)\n");
5327        }
5328 }
```

```
**********************************************************
   15250 Tue Aug 18 16:13:44 2015
new/usr/src/cmd/fuser/fuser.c
6136 sysmacros.h unnecessarily polutes the namespace
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License, Version 1.0 only
 * (the "License").  You may not use this file except in compliance
 * with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


#include <errno.h>
#include <fcntl.h>
#include <kstat.h>
#include <libdevinfo.h>
#include <locale.h>
#include <pwd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mnttab.h>
#include <sys/modctl.h>
#include <sys/stat.h>
#include <sys/sysmacros.h>
#include <sys/types.h>
#include <sys/utssys.h>
#include <sys/var.h>
#include <sys/mkdev.h>
#endif /* ! codereview */

/*
 * Command line options for fuser command. Mutually exclusive.
 */
#define OPT_FILE_ONLY           0x0001          /* -f */
#define OPT_CONTAINED           0x0002          /* -c */

/*
 * Command line option modifiers for fuser command.
```



```
 */
#define OPT_SIGNAL              0x0100          /* -k, -s */
#define OPT_USERID              0x0200          /* -u */
#define OPT_NBMANDLIST          0x0400          /* -n */
#define OPT_DEVINFO             0x0800          /* -d */

#define NELEM(a)                (sizeof (a) / sizeof ((a)[0]))

/*
 * System call prototype
 */
extern int utssys(void *buf, int arg, int type, void *outbp);

/*
 * Option flavors or types of options fuser command takes. Exclusive
 * options (EXCL_OPT) are mutually exclusive key options, while
 * modifier options (MOD_OPT) add to the key option. Examples are -f
 * for EXCL_OPT and -u for MOD_OPT.
 */
typedef enum {EXCL_OPT, MOD_OPT} opt_flavor_t;

struct co_tab {
        int     c_flag;
        char    c_char;
};

static struct co_tab code_tab[] = {
        {F_CDIR,        'c'},   /* current directory */
        {F_RDIR,        'r'},   /* root directory (via chroot) */
        {F_TEXT,        't'},   /* textfile */
        {F_OPEN,        'o'},   /* open (creat, etc.) file */
        {F_MAP,         'm'},   /* mapped file */
        {F_TTY,         'y'},   /* controlling tty */
        {F_TRACE,       'a'},   /* trace file */
        {F_NBM,         'n'}    /* nbmand lock/share reservation on file */
};

/*
 * Return a pointer to the mount point matching the given special name, if
 * possible, otherwise, exit with 1 if mnttab corruption is detected, else
 * return NULL.
 *
 * NOTE:  the underlying storage for mget and mref is defined static by
 * libos.  Repeated calls to getmntany() overwrite it; to save mnttab
 * structures would require copying the member strings elsewhere.
 */
static char *
spec_to_mount(char *specname)
{
        struct mnttab   mref, mget;
        struct stat     st;
        FILE            *frp;
        int             ret;

        /* get mount-point */
        if ((frp = fopen(MNTTAB, "r")) == NULL)
                return (NULL);

        mntnull(&mref);
        mref.mnt_special = specname;
        ret = getmntany(frp, &mget, &mref);
        (void) fclose(frp);

        if (ret == 0) {
                if ((stat(specname, &st) == 0) && S_ISBLK(st.st_mode))
                        return (mget.mnt_mountp);
```

```
126         } else if (ret > 0) {
127                 (void) fprintf(stderr, gettext("mnttab is corrupted\n"));
128                 exit(1);
129         }
130         return (NULL);
131 }

133 /*
134  * The main objective of this routine is to allocate an array of f_user_t's.
135  * In order for it to know how large an array to allocate, it must know
136  * the value of v.v_proc in the kernel.  To get this, we do a kstat
137  * lookup to get the var structure from the kernel.
138  */
139 static fu_data_t *
140 get_f_user_buf()
141 {
142         fu_data_t       fu_header, *fu_data;
143         kstat_ctl_t     *kc;
144         struct var      v;
145         kstat_t         *ksp;
146         int             count;

148         if ((kc = kstat_open()) == NULL ||
149             (ksp = kstat_lookup(kc, "unix", 0, "var")) == NULL ||
150             kstat_read(kc, ksp, &v) == -1) {
151                 perror(gettext("kstat_read() of struct var failed"));
152                 exit(1);
153         }
154         (void) kstat_close(kc);

156         /*
157          * get a count of the current number of kernel file consumers
158          *
159          * the number of kernel file consumers can change between
160          * the time when we get this count of all kernel file
161          * consumers and when we get the actual file usage
162          * information back from the kernel.
163          *
164          * we use the current count as a maximum because we assume
165          * that not all kernel file consumers are accessing the
166          * file we're interested in.  this assumption should make
167          * the current number of kernel file consumers a valid
168          * upper limit of possible file consumers.
169          *
170          * this call should never fail
171          */
172         fu_header.fud_user_max = 0;
173         fu_header.fud_user_count = 0;
174         (void) utssys(NULL, F_KINFO_COUNT, UTS_FUSERS, &fu_header);

176         count = v.v_proc + fu_header.fud_user_count;

178         fu_data = (fu_data_t *)malloc(fu_data_size(count));
179         if (fu_data == NULL) {
180                 (void) fprintf(stderr,
181                     gettext("fuser: could not allocate buffer\n"));
182                 exit(1);
183         }
184         fu_data->fud_user_max = count;
185         fu_data->fud_user_count = 0;
186         return (fu_data);
187 }

189 /*
190  * display the fuser usage message and exit
191  */
```



```
192 static void
193 usage()
194 {
195         (void) fprintf(stderr,
196             gettext("Usage:  fuser [-[k|s sig]un[c|f|d]] files"
197             " [-[[k|s sig]un[c|f|d]] files]..\n"));
198         exit(1);
199 }

201 static int
202 report_process(f_user_t *f_user, int options, int sig)
203 {
204         struct passwd   *pwdp;
205         int             i;

207         (void) fprintf(stdout, " %7d", (int)f_user->fu_pid);
208         (void) fflush(stdout);

210         /* print out any character codes for the process */
211         for (i = 0; i < NELEM(code_tab); i++) {
212                 if (f_user->fu_flags & code_tab[i].c_flag)
213                         (void) fprintf(stderr, "%c", code_tab[i].c_char);
214         }

216         /* optionally print the login name for the process */
217         if ((options & OPT_USERID) &&
218             ((pwdp = getpwuid(f_user->fu_uid)) != NULL))
219                 (void) fprintf(stderr, "(%s)", pwdp->pw_name);

221         /* optionally send a signal to the process */
222         if (options & OPT_SIGNAL)
223                 (void) kill(f_user->fu_pid, sig);

225         return (0);
226 }

228 static char *
229 i_get_dev_path(f_user_t *f_user, char *drv_name, int major, di_node_t *di_root)
230 {
231         di_minor_t      di_minor;
232         di_node_t       di_node;
233         dev_t           dev;
234         char            *path;

236         /*
237          * if we don't have a snapshot of the device tree yet, then
238          * take one so we can try to look up the device node and
239          * some kind of path to it.
240          */
241         if (*di_root == DI_NODE_NIL) {
242                 *di_root = di_init("/", DINFOSUBTREE | DINFOMINOR);
243                 if (*di_root == DI_NODE_NIL) {
244                         perror(gettext("devinfo snapshot failed"));
245                         return ((char *)-1);
246                 }
247         }

249         /* find device nodes that are bound to this driver */
250         di_node = di_drv_first_node(drv_name, *di_root);
251         if (di_node == DI_NODE_NIL)
252                 return (NULL);

254         /* try to get a dev_t for the device node we want to look up */
255         if (f_user->fu_minor == -1)
256                 dev = DDI_DEV_T_NONE;
257         else
```

```
258                 dev = makedev(major, f_user->fu_minor);

260         /* walk all the device nodes bound to this driver */
261         do {

263                 /* see if we can get a path to the minor node */
264                 if (dev != DDI_DEV_T_NONE) {
265                         di_minor = DI_MINOR_NIL;
266                         while (di_minor = di_minor_next(di_node, di_minor)) {
267                                 if (dev != di_minor_devt(di_minor))
268                                         continue;
269                                 path = di_devfs_minor_path(di_minor);
270                                 if (path == NULL) {
271                                         perror(gettext(
272                                             "unable to get device path"));
273                                         return ((char *)-1);
274                                 }
275                                 return (path);
276                         }
277                 }

279                 /* see if we can get a path to the device instance */
280                 if ((f_user->fu_instance != -1) &&
281                     (f_user->fu_instance == di_instance(di_node))) {
282                         path = di_devfs_path(di_node);
283                         if (path == NULL) {
284                                 perror(gettext("unable to get device path"));
285                                 return ((char *)-1);
286                         }
287                         return (path);
288                 }
289         } while (di_node = di_drv_next_node(di_node));

291         return (NULL);
292 }

294 static int
295 report_kernel(f_user_t *f_user, di_node_t *di_root)
296 {
297         struct modinfo  modinfo;
298         char            *path;
299         int             major = -1;

301         /* get the module name */
302         modinfo.mi_info = MI_INFO_ONE | MI_INFO_CNT | MI_INFO_NOBASE;
303         modinfo.mi_id = modinfo.mi_nextid = f_user->fu_modid;
304         if (modctl(MODINFO, f_user->fu_modid, &modinfo) < 0) {
305                 perror(gettext("unable to get kernel module information"));
306                 return (-1);
307         }

309         /*
310          * if we don't have any device info then just
311          * print the module name
312          */
313         if ((f_user->fu_instance == -1) && (f_user->fu_minor == -1)) {
314                 (void) fprintf(stderr, " [%s]", modinfo.mi_name);
315                 return (0);
316         }

318         /* get the driver major number */
319         if (modctl(MODGETMAJBIND,
320             modinfo.mi_name, strlen(modinfo.mi_name) + 1, &major) < 0) {
321                 perror(gettext("unable to get driver major number"));
322                 return (-1);
323         }
```



```
325         path = i_get_dev_path(f_user, modinfo.mi_name, major, di_root);
326         if (path == (char *)-1)
327                 return (-1);

329         /* check if we couldn't get any device pathing info */
330         if (path == NULL) {
331                 if (f_user->fu_minor == -1) {
332                         /*
333                          * we don't really have any more info on the device
334                          * so display the driver name in the same format
335                          * that we would for a plain module
336                          */
337                         (void) fprintf(stderr, " [%s]", modinfo.mi_name);
338                         return (0);
339                 } else {
340                         /*
341                          * if we only have dev_t information, then display
342                          * the driver name and the dev_t info
343                          */
344                         (void) fprintf(stderr, " [%s,dev=(%d,%d)]",
345                             modinfo.mi_name, major, f_user->fu_minor);
346                         return (0);
347                 }
348         }

350         /* display device pathing information */
351         if (f_user->fu_minor == -1) {
352                 /*
353                  * display the driver name and a path to the device
354                  * instance.
355                  */
356                 (void) fprintf(stderr, " [%s,dev_path=%s]",
357                     modinfo.mi_name, path);
358         } else {
359                 /*
360                  * here we have lot's of info.  the driver name, the minor
361                  * node dev_t, and a path to the device.  display it all.
362                  */
363                 (void) fprintf(stderr, " [%s,dev=(%d,%d),dev_path=%s]",
364                     modinfo.mi_name, major, f_user->fu_minor, path);
365         }

367         di_devfs_path_free(path);
368         return (0);
369 }

371 /*
372  * Show pids and usage indicators for the nusers processes in the users list.
373  * When OPT_USERID is set, give associated login names.  When OPT_SIGNAL is
374  * set, issue the specified signal to those processes.
375  */
376 static void
377 report(fu_data_t *fu_data, int options, int sig)
378 {
379         di_node_t       di_root = DI_NODE_NIL;
380         f_user_t        *f_user;
381         int             err, i;

383         for (err = i = 0; (err == 0) && (i <  fu_data->fud_user_count); i++) {

385                 f_user = &(fu_data->fud_user[i]);
386                 if (f_user->fu_flags & F_KERNEL) {
387                         /* a kernel module is using the file */
388                         err = report_kernel(f_user, &di_root);
389                 } else {
```

```
390                         /* a userland process using the file */
391                         err = report_process(f_user, options, sig);
392                 }
393         }

395         if (di_root != DI_NODE_NIL)
396                 di_fini(di_root);
397 }

399 /*
400  * Sanity check the option "nextopt" and OR it into *options.
401  */
402 static void
403 set_option(int *options, int nextopt, opt_flavor_t type)
404 {
405         static const char       *excl_opts[] = {"-c", "-f", "-d"};
406         int                     i;

408         /*
409          * Disallow repeating options
410          */
411         if (*options & nextopt)
412                 usage();

414         /*
415          * If EXCL_OPT, allow only one option to be set
416          */
417         if ((type == EXCL_OPT) && (*options)) {
418                 (void) fprintf(stderr,
419                     gettext("Use only one of the following options :"));
420                 for (i = 0; i < NELEM(excl_opts); i++) {
421                         if (i == 0) {
422                                 (void) fprintf(stderr, gettext(" %s"),
423                                     excl_opts[i]);
424                         } else {
425                                 (void) fprintf(stderr, gettext(", %s"),
426                                     excl_opts[i]);
427                         }
428                 }
429                 (void) fprintf(stderr, "\n"),
430                 usage();
431         }
432         *options |= nextopt;
433 }

435 /*
436  * Determine which processes are using a named file or file system.
437  * On stdout, show the pid of each process using each command line file
438  * with indication(s) of its use(s).  Optionally display the login
439  * name with each process.  Also optionally, issue the specified signal to
440  * each process.
441  *
442  * X/Open Commands and Utilites, Issue 5 requires fuser to process
443  * the complete list of names it is given, so if an error is encountered
444  * it will continue through the list, and then exit with a non-zero
445  * value. This is a change from earlier behavior where the command
446  * would exit immediately upon an error.
447  *
448  * The preferred use of the command is with a single file or file system.
449  */

451 int
452 main(int argc, char **argv)
453 {
454         fu_data_t       *fu_data;
455         char            *mntname, c;
```



```
456         int             newfile = 0, errors = 0, opts = 0, flags = 0;
457         int             uts_flags, sig, okay, err;

459         (void) setlocale(LC_ALL, "");
460         (void) textdomain(TEXT_DOMAIN);

462         if (argc < 2)
463                 usage();

465         do {
466                 while ((c = getopt(argc, argv, "cdfkns:u")) != EOF) {
467                         if (newfile) {
468                                 /*
469                                  * Starting a new group of files.
470                                  * Clear out options currently in
471                                  * force.
472                                  */
473                                 flags = opts = newfile = 0;
474                         }
475                         switch (c) {
476                         case 'd':
477                                 set_option(&opts, OPT_DEVINFO, EXCL_OPT);
478                                 break;
479                         case 'k':
480                                 set_option(&flags, OPT_SIGNAL, MOD_OPT);
481                                 sig = SIGKILL;
482                                 break;
483                         case 's':
484                                 set_option(&flags, OPT_SIGNAL, MOD_OPT);
485                                 if (str2sig(optarg, &sig) != 0) {
486                                         (void) fprintf(stderr,
487                                             gettext("Invalid signal %s\n"),
488                                             optarg);
489                                         usage();
490                                 }
491                                 break;
492                         case 'u':
493                                 set_option(&flags, OPT_USERID, MOD_OPT);
494                                 break;
495                         case 'n':
496                                 /*
497                                  * Report only users with NBMAND locks
498                                  */
499                                 set_option(&flags, OPT_NBMANDLIST, MOD_OPT);
500                                 break;
501                         case 'c':
502                                 set_option(&opts, OPT_CONTAINED, EXCL_OPT);
503                                 break;
504                         case 'f':
505                                 set_option(&opts, OPT_FILE_ONLY, EXCL_OPT);
506                                 break;
507                         default:
508                                 (void) fprintf(stderr,
509                                     gettext("Illegal option %c.\n"), c);
510                                 usage();
511                         }
512                 }

514                 if ((optind < argc) && (newfile)) {
515                         /*
516                          * Cancel the options currently in
517                          * force if a lone dash is specified.
518                          */
519                         if (strcmp(argv[optind], "-") == 0) {
520                                 flags = opts = newfile = 0;
521                                 optind++;
```

```
522                     }
523             }

525             /*
526              * newfile is set when a new group of files is found.  If all
527              * arguments are processed and newfile isn't set here, then
528              * the user did not use the correct syntax
529              */
530             if (optind > argc - 1) {
531                     if (!newfile) {
532                             (void) fprintf(stderr,
533                                 gettext("fuser: missing file name\n"));
534                             usage();
535                     }
536             } else {
537                     if (argv[optind][0] == '-') {
538                             (void) fprintf(stderr,
539                                 gettext("fuser: incorrect use of -\n"));
540                             usage();
541                     } else {
542                             newfile = 1;
543                     }
544             }

546             /* allocate a buffer to hold usage data */
547             fu_data = get_f_user_buf();

549             /*
550              * First print file name on stderr
551              * (so stdout (pids) can be piped to kill)
552              */
553             (void) fflush(stdout);
554             (void) fprintf(stderr, "%s: ", argv[optind]);

556             /*
557              * if not OPT_FILE_ONLY, OPT_DEVINFO, or OPT_CONTAINED,
558              * attempt to translate the target file name to a mount
559              * point via /etc/mnttab.
560              */
561             okay = 0;
562             if (!opts &&
563                 (mntname = spec_to_mount(argv[optind])) != NULL) {

565                     uts_flags = F_CONTAINED |
566                         ((flags & OPT_NBMANDLIST) ? F_NBMANDLIST : 0);

568                     err = utssys(mntname, uts_flags, UTS_FUSERS, fu_data);
569                     if (err == 0) {
570                             report(fu_data, flags, sig);
571                             okay = 1;
572                     }
573             }

575             uts_flags = \
576                 ((opts & OPT_CONTAINED) ? F_CONTAINED : 0) |
577                 ((opts & OPT_DEVINFO) ? F_DEVINFO : 0) |
578                 ((flags & OPT_NBMANDLIST) ? F_NBMANDLIST : 0);

580             err = utssys(argv[optind], uts_flags, UTS_FUSERS, fu_data);
581             if (err == 0) {
582                     report(fu_data, flags, sig);
583             } else if (!okay) {
584                     perror("fuser");
585                     errors = 1;
586                     free(fu_data);
587                     continue;
```



```
588                     }

590                     (void) fprintf(stderr, "\n");
591                     free(fu_data);
592             } while (++optind < argc);

594             return (errors);
595     }
```

```
**********
  206931 Tue Aug 18 16:13:44 2015
new/usr/src/cmd/sendmail/src/queue.c
6136 sysmacros.h unnecessarily polutes the namespace
**********
```

```
/*
 * Copyright (c) 1998-2009 Sendmail, Inc. and its suppliers.
 *      All rights reserved.
 * Copyright (c) 1983, 1995-1997 Eric P. Allman.  All rights reserved.
 * Copyright (c) 1988, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * By using this file, you agree to the terms and conditions set
 * forth in the LICENSE file which can be found at the top level of
 * the sendmail distribution.
 *
 */

#include <sendmail.h>
#include <sm/sem.h>

SM_RCSID("@(#)$Id: queue.c,v 8.987 2009/12/18 17:08:01 ca Exp $")

#include <sys/types.h>
#include <sys/mkdev.h>
#endif /* ! codereview */
#include <dirent.h>

# define RELEASE_QUEUE  (void) 0
# define ST_INODE(st)   (st).st_ino

#  define sm_file_exists(errno) ((errno) == EEXIST)

# if HASFLOCK && defined(O_EXLOCK)
#   define SM_OPEN_EXLOCK 1
#   define TF_OPEN_FLAGS (O_CREAT|O_WRONLY|O_EXCL|O_EXLOCK)
# else /* HASFLOCK && defined(O_EXLOCK) */
#  define TF_OPEN_FLAGS (O_CREAT|O_WRONLY|O_EXCL)
# endif /* HASFLOCK && defined(O_EXLOCK) */

#ifndef SM_OPEN_EXLOCK
# define SM_OPEN_EXLOCK 0
#endif /* ! SM_OPEN_EXLOCK */

/*
**  Historical notes:
**      QF_VERSION == 4 was sendmail 8.10/8.11 without _FFR_QUEUEDELAY
**      QF_VERSION == 5 was sendmail 8.10/8.11 with    _FFR_QUEUEDELAY
**      QF_VERSION == 6 was sendmail 8.12      without _FFR_QUEUEDELAY
**      QF_VERSION == 7 was sendmail 8.12      with    _FFR_QUEUEDELAY
**      QF_VERSION == 8 is  sendmail 8.13
*/

#define QF_VERSION      8       /* version number of this queue format */

static char     queue_letter __P((ENVELOPE *, int));
static bool     quarantine_queue_item __P((int, int, ENVELOPE *, char *));

/* Naming convention: qgrp: index of queue group, qg: QUEUEGROUP */

/*
**  Work queue.
*/

struct work
{
```



```
        char            *w_name;        /* name of control file */
        char            *w_host;        /* name of recipient host */
        bool            w_lock;         /* is message locked? */
        bool            w_tooyoung;     /* is it too young to run? */
        long            w_pri;          /* priority of message, see below */
        time_t          w_ctime;        /* creation time */
        time_t          w_mtime;        /* modification time */
        int             w_qgrp;         /* queue group located in */
        int             w_qdir;         /* queue directory located in */
        struct work     *w_next;        /* next in queue */
};

typedef struct work     WORK;

static WORK     *WorkQ;         /* queue of things to be done */
static int      NumWorkGroups;  /* number of work groups */
static time_t   Current_LA_time = 0;

/* Get new load average every 30 seconds. */
#define GET_NEW_LA_TIME 30

#define SM_GET_LA(now)  \
        do                                                      \
        {                                                       \
                now = curtime();                                \
                if (Current_LA_time < now - GET_NEW_LA_TIME)    \
                {                                               \
                        sm_getla();                             \
                        Current_LA_time = now;                  \
                }                                               \
        } while (0)

/*
**  DoQueueRun indicates that a queue run is needed.
**      Notice: DoQueueRun is modified in a signal handler!
*/

static bool     volatile DoQueueRun; /* non-interrupt time queue run needed */

/*
**  Work group definition structure.
**      Each work group contains one or more queue groups. This is done
**      to manage the number of queue group runners active at the same time
**      to be within the constraints of MaxQueueChildren (if it is set).
**      The number of queue groups that can be run on the next work run
**      is kept track of. The queue groups are run in a round robin.
*/

struct workgrp
{
        int             wg_numqgrp;     /* number of queue groups in work grp */
        int             wg_runners;     /* total runners */
        int             wg_curqgrp;     /* current queue group */
        QUEUEGRP        **wg_qgs;       /* array of queue groups */
        int             wg_maxact;      /* max # of active runners */
        time_t          wg_lowqintvl;   /* lowest queue interval */
        int             wg_restart;     /* needs restarting? */
        int             wg_restartcnt;  /* count of times restarted */
};

typedef struct workgrp WORKGRP;

static WORKGRP  volatile WorkGrp[MAXWORKGROUPS + 1];    /* work groups */

#if SM_HEAP_CHECK
static SM_DEBUG_T DebugLeakQ = SM_DEBUG_INITIALIZER("leak_q",
```

```
        "@(#)$Debug: leak_q - trace memory leaks during queue processing $");
#endif /* SM_HEAP_CHECK */

/*
**  We use EmptyString instead of "" to avoid
**  'zero-length format string' warnings from gcc
*/

static const char EmptyString[] = "";

static void     grow_wlist __P((int, int));
static int      multiqueue_cache __P((char *, int, QUEUEGRP *, int, unsigned int
static int      gatherq __P((int, int, bool, bool *, bool *, int *));
static int      sortq __P((int));
static void     printctladdr __P((ADDRESS *, SM_FILE_T *));
static bool     readqf __P((ENVELOPE *, bool));
static void     restart_work_group __P((int));
static void     runner_work __P((ENVELOPE *, int, bool, int, int));
static void     schedule_queue_runs __P((bool, int, bool));
static char     *strrev __P((char *));
static ADDRESS  *setctluser __P((char *, int, ENVELOPE *));
#if _FFR_RHS
static int      sm_strshufflecmp __P((char *, char *));
static void     init_shuffle_alphabet __P(());
#endif /* _FFR_RHS */

/*
**  Note: workcmpf?() don't use a prototype because it will cause a conflict
**  with the qsort() call (which expects something like
**  int (*compar)(const void *, const void *), not (WORK *, WORK *))
*/

static int      workcmpf0();
static int      workcmpf1();
static int      workcmpf2();
static int      workcmpf3();
static int      workcmpf4();
static int      randi = 3;      /* index for workcmpf5() */
static int      workcmpf5();
static int      workcmpf6();
#if _FFR_RHS
static int      workcmpf7();
#endif /* _FFR_RHS */

#if RANDOMSHIFT
# define get_rand_mod(m)        ((get_random() >> RANDOMSHIFT) % (m))
#else /* RANDOMSHIFT */
# define get_rand_mod(m)        (get_random() % (m))
#endif /* RANDOMSHIFT */

/*
**  File system definition.
**      Used to keep track of how much free space is available
**      on a file system in which one or more queue directories reside.
*/

typedef struct filesys_shared   FILESYS;

struct filesys_shared
{
        dev_t   fs_dev;         /* unique device id */
        long    fs_avail;       /* number of free blocks available */
        long    fs_blksize;     /* block size, in bytes */
};

/* probably kept in shared memory */
```



```
static FILESYS  FileSys[MAXFILESYS];    /* queue file systems */
static const char *FSPath[MAXFILESYS];  /* pathnames for file systems */

#if SM_CONF_SHM

/*
**  Shared memory data
**
**  Current layout:
**      size -- size of shared memory segment
**      pid -- pid of owner, should be a unique id to avoid misinterpretations
**             by other processes.
**      tag -- should be a unique id to avoid misinterpretations by others.
**             idea: hash over configuration data that will be stored here.
**      NumFileSys -- number of file systems.
**      FileSys -- (arrary of) structure for used file systems.
**      RSATmpCnt -- counter for number of uses of ephemeral RSA key.
**      QShm -- (array of) structure for information about queue directories.
*/

/*
**  Queue data in shared memory
*/

typedef struct queue_shared     QUEUE_SHM_T;

struct queue_shared
{
        int     qs_entries;     /* number of entries */
        /* XXX more to follow? */
};

static void     *Pshm;          /* pointer to shared memory */
static FILESYS  *PtrFileSys;    /* pointer to queue file system array */
int             ShmId = SM_SHM_NO_ID;   /* shared memory id */
static QUEUE_SHM_T      *QShm;          /* pointer to shared queue data */
static size_t shms;

# define SHM_OFF_PID(p) (((char *) (p)) + sizeof(int))
# define SHM_OFF_TAG(p) (((char *) (p)) + sizeof(pid_t) + sizeof(int))
# define SHM_OFF_HEAD   (sizeof(pid_t) + sizeof(int) * 2)

/* how to access FileSys */
# define FILE_SYS(i)    (PtrFileSys[i])

/* first entry is a tag, for now just the size */
# define OFF_FILE_SYS(p)        (((char *) (p)) + SHM_OFF_HEAD)

/* offset for PNumFileSys */
# define OFF_NUM_FILE_SYS(p)    (((char *) (p)) + SHM_OFF_HEAD + sizeof(FileSys)

/* offset for PRSATmpCnt */
# define OFF_RSA_TMP_CNT(p) (((char *) (p)) + SHM_OFF_HEAD + sizeof(FileSys) + s
int     *PRSATmpCnt;

/* offset for queue_shm */
# define OFF_QUEUE_SHM(p) (((char *) (p)) + SHM_OFF_HEAD + sizeof(FileSys) + siz

# define QSHM_ENTRIES(i)        QShm[i].qs_entries

/* basic size of shared memory segment */
# define SM_T_SIZE      (SHM_OFF_HEAD + sizeof(FileSys) + sizeof(int) * 2)

static unsigned int     hash_q __P((char *, unsigned int));

/*
```

```
**  HASH_Q -- simple hash function
**
**      Parameters:
**              p -- string to hash.
**              h -- hash start value (from previous run).
**
**      Returns:
**              hash value.
*/

static unsigned int
hash_q(p, h)
        char *p;
        unsigned int h;
{
        int c, d;

        while (*p != '\0')
        {
                d = *p++;
                c = d;
                c ^= c<<6;
                h += (c<<11) ^ (c>>1);
                h ^= (d<<14) + (d<<7) + (d<<4) + d;
        }
        return h;
}


#else /* SM_CONF_SHM */
# define FILE_SYS(i)    FileSys[i]
#endif /* SM_CONF_SHM */

/* access to the various components of file system data */
#define FILE_SYS_NAME(i)        FSPath[i]
#define FILE_SYS_AVAIL(i)       FILE_SYS(i).fs_avail
#define FILE_SYS_BLKSIZE(i)     FILE_SYS(i).fs_blksize
#define FILE_SYS_DEV(i) FILE_SYS(i).fs_dev


/*
**  Current qf file field assignments:
**
**      A       AUTH= parameter
**      B       body type
**      C       controlling user
**      D       data file name
**      d       data file directory name (added in 8.12)
**      E       error recipient
**      F       flag bits
**      G       free (was: queue delay algorithm if _FFR_QUEUEDELAY)
**      H       header
**      I       data file's inode number
**      K       time of last delivery attempt
**      L       Solaris Content-Length: header (obsolete)
**      M       message
**      N       number of delivery attempts
**      P       message priority
**      q       quarantine reason
**      Q       original recipient (ORCPT=)
**      r       final recipient (Final-Recipient: DSN field)
**      R       recipient
**      S       sender
**      T       init time
**      V       queue file version
**      X       free (was: character set if _FFR_SAVE_CHARSET)
```



```
**      Y       free (was: current delay if _FFR_QUEUEDELAY)
**      Z       original envelope id from ESMTP
**      !       deliver by (added in 8.12)
**      $       define macro
**      .       terminate file
*/

/*
**  QUEUEUP -- queue a message up for future transmission.
**
**      Parameters:
**              e -- the envelope to queue up.
**              announce -- if true, tell when you are queueing up.
**              msync -- if true, then fsync() if SuperSafe interactive mode.
**
**      Returns:
**              none.
**
**      Side Effects:
**              The current request is saved in a control file.
**              The queue file is left locked.
*/

void
queueup(e, announce, msync)
        register ENVELOPE *e;
        bool announce;
        bool msync;
{
        register SM_FILE_T *tfp;
        register HDR *h;
        register ADDRESS *q;
        int tfd = -1;
        int i;
        bool newid;
        register char *p;
        MAILER nullmailer;
        MCI mcibuf;
        char qf[MAXPATHLEN];
        char tf[MAXPATHLEN];
        char df[MAXPATHLEN];
        char buf[MAXLINE];

        /*
        **  Create control file.
        */

#define OPEN_TF do                                                      \
                {                                                       \
                        MODE_T oldumask = 0;                            \
                                                                        \
                        if (bitset(S_IWGRP, QueueFileMode))             \
                                oldumask = umask(002);                  \
                        tfd = open(tf, TF_OPEN_FLAGS, QueueFileMode);   \
                        if (bitset(S_IWGRP, QueueFileMode))             \
                                (void) umask(oldumask);                 \
                } while (0)


        newid = (e->e_id == NULL) || !bitset(EF_INQUEUE, e->e_flags);
        (void) sm_strlcpy(tf, queuename(e, NEWQFL_LETTER), sizeof(tf));
        tfp = e->e_lockfp;
        if (tfp == NULL && newid)
        {
                /*
                **  open qf file directly: this will give an error if the file
```

```
392                 **  already exists and hence prevent problems if a queue-id
393                 **  is reused (e.g., because the clock is set back).
394                 */

396                 (void) sm_strlcpy(tf, queuename(e, ANYQFL_LETTER), sizeof(tf));
397                 OPEN_TF;
398                 if (tfd < 0 ||
399 #if !SM_OPEN_EXLOCK
400                     !lockfile(tfd, tf, NULL, LOCK_EX|LOCK_NB) ||
401 #endif /* !SM_OPEN_EXLOCK */
402                     (tfp = sm_io_open(SmFtStdiofd, SM_TIME_DEFAULT,
403                                       (void *) &tfd, SM_IO_WRONLY,
404                                       NULL)) == NULL)
405                 {
406                         int save_errno = errno;

408                         printopenfds(true);
409                         errno = save_errno;
410                         syserr("!queueup: cannot create queue file %s, euid=%d,
411                                tf, (int) geteuid(), tfd, tfp);
412                         /* NOTREACHED */
413                 }
414                 e->e_lockfp = tfp;
415                 upd_qs(e, 1, 0, "queueup");
416         }

418         /* if newid, write the queue file directly (instead of temp file) */
419         if (!newid)
420         {
421                 /* get a locked tf file */
422                 for (i = 0; i < 128; i++)
423                 {
424                         if (tfd < 0)
425                         {
426                                 OPEN_TF;
427                                 if (tfd < 0)
428                                 {
429                                         if (errno != EEXIST)
430                                                 break;
431                                         if (LogLevel > 0 && (i % 32) == 0)
432                                                 sm_syslog(LOG_ALERT, e->e_id,
433                                                           "queueup: cannot creat
434                                                           tf, (int) geteuid(),
435                                                           sm_errstring(errno));
436                                 }
437 #if SM_OPEN_EXLOCK
438                                 else
439                                         break;
440 #endif /* SM_OPEN_EXLOCK */
441                         }
442                         if (tfd >= 0)
443                         {
444 #if SM_OPEN_EXLOCK
445                                 /* file is locked by open() */
446                                 break;
447 #else /* SM_OPEN_EXLOCK */
448                                 if (lockfile(tfd, tf, NULL, LOCK_EX|LOCK_NB))
449                                         break;
450                                 else
451 #endif /* SM_OPEN_EXLOCK */
452                                 if (LogLevel > 0 && (i % 32) == 0)
453                                         sm_syslog(LOG_ALERT, e->e_id,
454                                                   "queueup: cannot lock %s: %s",
455                                                   tf, sm_errstring(errno));
456                                 if ((i % 32) == 31)
457                                 {
```



```
458                                         (void) close(tfd);
459                                         tfd = -1;
460                                 }
461                         }

463                         if ((i % 32) == 31)
464                         {
465                                 /* save the old temp file away */
466                                 (void) rename(tf, queuename(e, TEMPQF_LETTER));
467                         }
468                         else
469                                 (void) sleep(i % 32);
470                 }
471                 if (tfd < 0 || (tfp = sm_io_open(SmFtStdiofd, SM_TIME_DEFAULT,
472                                                  (void *) &tfd, SM_IO_WRONLY_B,
473                                                  NULL)) == NULL)
474                 {
475                         int save_errno = errno;

477                         printopenfds(true);
478                         errno = save_errno;
479                         syserr("!queueup: cannot create queue temp file %s, uid=
480                                tf, (int) geteuid());
481                 }
482         }

484         if (tTd(40, 1))
485                 sm_dprintf("\n>>>>> queueing %s/%s%s >>>>>\n",
486                            qid_printqueue(e->e_qgrp, e->e_qdir),
487                            queuename(e, ANYQFL_LETTER),
488                            newid ? " (new id)" : "");
489         if (tTd(40, 3))
490         {
491                 sm_dprintf("  e_flags=");
492                 printenvflags(e);
493         }
494         if (tTd(40, 32))
495         {
496                 sm_dprintf("  sendq=");
497                 printaddr(sm_debug_file(), e->e_sendqueue, true);
498         }
499         if (tTd(40, 9))
500         {
501                 sm_dprintf("  tfp=");
502                 dumpfd(sm_io_getinfo(tfp, SM_IO_WHAT_FD, NULL), true, false);
503                 sm_dprintf("  lockfp=");
504                 if (e->e_lockfp == NULL)
505                         sm_dprintf("NULL\n");
506                 else
507                         dumpfd(sm_io_getinfo(e->e_lockfp, SM_IO_WHAT_FD, NULL),
508                                true, false);
509         }

511         /*
512         **  If there is no data file yet, create one.
513         */

515         (void) sm_strlcpy(df, queuename(e, DATAFL_LETTER), sizeof(df));
516         if (bitset(EF_HAS_DF, e->e_flags))
517         {
518                 if (e->e_dfp != NULL &&
519                     SuperSafe != SAFE_REALLY &&
520                     SuperSafe != SAFE_REALLY_POSTMILTER &&
521                     sm_io_setinfo(e->e_dfp, SM_BF_COMMIT, NULL) < 0 &&
522                     errno != EINVAL)
523                 {
```

```
524                         syserr("!queueup: cannot commit data file %s, uid=%d",
525                                queuename(e, DATAFL_LETTER), (int) geteuid());
526                 }
527                 if (e->e_dfp != NULL &&
528                     SuperSafe == SAFE_INTERACTIVE && msync)
529                 {
530                         if (tTd(40,32))
531                                 sm_syslog(LOG_INFO, e->e_id,
532                                           "queueup: fsync(e->e_dfp)");

534                         if (fsync(sm_io_getinfo(e->e_dfp, SM_IO_WHAT_FD,
535                                                 NULL)) < 0)
536                         {
537                                 if (newid)
538                                         syserr("!552 Error writing data file %s"
539                                                df);
540                                 else
541                                         syserr("!452 Error writing data file %s"
542                                                df);
543                         }
544                 }
545         }
546         else
547         {
548                 int dfd;
549                 MODE_T oldumask = 0;
550                 register SM_FILE_T *dfp = NULL;
551                 struct stat stbuf;

553                 if (e->e_dfp != NULL &&
554                     sm_io_getinfo(e->e_dfp, SM_IO_WHAT_ISTYPE, BF_FILE_TYPE))
555                         syserr("committing over bf file");

557                 if (bitset(S_IWGRP, QueueFileMode))
558                         oldumask = umask(002);
559                 dfd = open(df, O_WRONLY|O_CREAT|O_TRUNC|QF_O_EXTRA,
560                            QueueFileMode);
561                 if (bitset(S_IWGRP, QueueFileMode))
562                         (void) umask(oldumask);
563                 if (dfd < 0 || (dfp = sm_io_open(SmFtStdiofd, SM_TIME_DEFAULT,
564                                                  (void *) &dfd, SM_IO_WRONLY_B,
565                                                  NULL)) == NULL)
566                         syserr("!queueup: cannot create data temp file %s, uid=%
567                                df, (int) geteuid());
568                 if (fstat(dfd, &stbuf) < 0)
569                         e->e_dfino = -1;
570                 else
571                 {
572                         e->e_dfdev = stbuf.st_dev;
573                         e->e_dfino = ST_INODE(stbuf);
574                 }
575                 e->e_flags |= EF_HAS_DF;
576                 memset(&mcibuf, '\0', sizeof(mcibuf));
577                 mcibuf.mci_out = dfp;
578                 mcibuf.mci_mailer = FileMailer;
579                 (*e->e_putbody)(&mcibuf, e, NULL);

581                 if (SuperSafe == SAFE_REALLY ||
582                     SuperSafe == SAFE_REALLY_POSTMILTER ||
583                     (SuperSafe == SAFE_INTERACTIVE && msync))
584                 {
585                         if (tTd(40,32))
586                                 sm_syslog(LOG_INFO, e->e_id,
587                                           "queueup: fsync(dfp)");

589                         if (fsync(sm_io_getinfo(dfp, SM_IO_WHAT_FD, NULL)) < 0)
```



```
590                         {
591                                 if (newid)
592                                         syserr("!552 Error writing data file %s"
593                                                df);
594                                 else
595                                         syserr("!452 Error writing data file %s"
596                                                df);
597                         }
598                 }

600                 if (sm_io_close(dfp, SM_TIME_DEFAULT) < 0)
601                         syserr("!queueup: cannot save data temp file %s, uid=%d"
602                                df, (int) geteuid());
603                 e->e_putbody = putbody;
604         }

606         /*
607         **  Output future work requests.
608         **      Priority and creation time should be first, since
609         **      they are required by gatherq.
610         */

612         /* output queue version number (must be first!) */
613         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "V%d\n", QF_VERSION);

615         /* output creation time */
616         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "T%ld\n", (long) e->e_ctime);

618         /* output last delivery time */
619         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "K%ld\n", (long) e->e_dtime);

621         /* output number of delivery attempts */
622         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "N%d\n", e->e_ntries);

624         /* output message priority */
625         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "P%ld\n", e->e_msgpriority);

627         /*
628         **  If data file is in a different directory than the queue file,
629         **  output a "d" record naming the directory of the data file.
630         */

632         if (e->e_dfqgrp != e->e_qgrp)
633         {
634                 (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "d%s\n",
635                                      Queue[e->e_dfqgrp]->qg_qpaths[e->e_dfqdir].qp_name);
636         }

638         /* output inode number of data file */
639         /* XXX should probably include device major/minor too */
640         if (e->e_dfino != -1)
641         {
642                 (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "I%ld/%ld/%llu\n",
643                                      (long) major(e->e_dfdev),
644                                      (long) minor(e->e_dfdev),
645                                      (ULONGLONG_T) e->e_dfino);
646         }

648         /* output body type */
649         if (e->e_bodytype != NULL)
650                 (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "B%s\n",
651                                      denlstring(e->e_bodytype, true, false));

653         /* quarantine reason */
654         if (e->e_quarmsg != NULL)
655                 (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "q%s\n",
```

```
656                                     denlstring(e->e_quarmsg, true, false));
657
658             /* message from envelope, if it exists */
659             if (e->e_message != NULL)
660                     (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "M%s\n",
661                                     denlstring(e->e_message, true, false));
662
663             /* send various flag bits through */
664             p = buf;
665             if (bitset(EF_WARNING, e->e_flags))
666                     *p++ = 'w';
667             if (bitset(EF_RESPONSE, e->e_flags))
668                     *p++ = 'r';
669             if (bitset(EF_HAS8BIT, e->e_flags))
670                     *p++ = '8';
671             if (bitset(EF_DELETE_BCC, e->e_flags))
672                     *p++ = 'b';
673             if (bitset(EF_RET_PARAM, e->e_flags))
674                     *p++ = 'd';
675             if (bitset(EF_NO_BODY_RETN, e->e_flags))
676                     *p++ = 'n';
677             if (bitset(EF_SPLIT, e->e_flags))
678                     *p++ = 's';
679             *p++ = '\0';
680             if (buf[0] != '\0')
681                     (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "F%s\n", buf);
682
683             /* save $={persistentMacros} macro values */
684             queueup_macros(macid("{persistentMacros}"), tfp, e);
685
686             /* output name of sender */
687             if (bitnset(M_UDBENVELOPE, e->e_from.q_mailer->m_flags))
688                     p = e->e_sender;
689             else
690                     p = e->e_from.q_paddr;
691             (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "S%s\n",
692                                  denlstring(p, true, false));
693
694             /* output ESMTP-supplied "original" information */
695             if (e->e_envid != NULL)
696                     (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "Z%s\n",
697                                     denlstring(e->e_envid, true, false));
698
699             /* output AUTH= parameter */
700             if (e->e_auth_param != NULL)
701                     (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "A%s\n",
702                                     denlstring(e->e_auth_param, true, false));
703             if (e->e_dlvr_flag != 0)
704                     (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "!%c %ld\n",
705                                     (char) e->e_dlvr_flag, e->e_deliver_by);
706
707             /* output list of recipient addresses */
708             printctladdr(NULL, NULL);
709             for (q = e->e_sendqueue; q != NULL; q = q->q_next)
710             {
711                     if (!QS_IS_UNDELIVERED(q->q_state))
712                             continue;
713
714                     /* message for this recipient, if it exists */
715                     if (q->q_message != NULL)
716                             (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "M%s\n",
717                                             denlstring(q->q_message, true,
718                                                        false));
719
720                     printctladdr(q, tfp);
721                     if (q->q_orcpt != NULL)
```



```
722                             (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "Q%s\n",
723                                             denlstring(q->q_orcpt, true,
724                                                        false));
725                     if (q->q_finalrcpt != NULL)
726                             (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "r%s\n",
727                                             denlstring(q->q_finalrcpt, true,
728                                                        false));
729                     (void) sm_io_putc(tfp, SM_TIME_DEFAULT, 'R');
730                     if (bitset(QPRIMARY, q->q_flags))
731                             (void) sm_io_putc(tfp, SM_TIME_DEFAULT, 'P');
732                     if (bitset(QHASNOTIFY, q->q_flags))
733                             (void) sm_io_putc(tfp, SM_TIME_DEFAULT, 'N');
734                     if (bitset(QPINGONSUCCESS, q->q_flags))
735                             (void) sm_io_putc(tfp, SM_TIME_DEFAULT, 'S');
736                     if (bitset(QPINGONFAILURE, q->q_flags))
737                             (void) sm_io_putc(tfp, SM_TIME_DEFAULT, 'F');
738                     if (bitset(QPINGONDELAY, q->q_flags))
739                             (void) sm_io_putc(tfp, SM_TIME_DEFAULT, 'D');
740                     if (q->q_alias != NULL &&
741                         bitset(QALIAS, q->q_alias->q_flags))
742                             (void) sm_io_putc(tfp, SM_TIME_DEFAULT, 'A');
743                     (void) sm_io_putc(tfp, SM_TIME_DEFAULT, ':');
744                     (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "%s\n",
745                                          denlstring(q->q_paddr, true, false));
746                     if (announce)
747                     {
748                             char *tag = "queued";
749
750                             if (e->e_quarmsg != NULL)
751                                     tag = "quarantined";
752
753                             e->e_to = q->q_paddr;
754                             message(tag);
755                             if (LogLevel > 8)
756                                     logdelivery(q->q_mailer, NULL, q->q_status,
757                                                 tag, NULL, (time_t) 0, e);
758                             e->e_to = NULL;
759                     }
760                     if (tTd(40, 1))
761                     {
762                             sm_dprintf("queueing ");
763                             printaddr(sm_debug_file(), q, false);
764                     }
765             }
766
767             /*
768             **  Output headers for this message.
769             **      Expand macros completely here.  Queue run will deal with
770             **      everything as absolute headers.
771             **              All headers that must be relative to the recipient
772             **              can be cracked later.
773             **      We set up a "null mailer" -- i.e., a mailer that will have
774             **      no effect on the addresses as they are output.
775             */
776
777             memset((char *) &nullmailer, '\0', sizeof(nullmailer));
778             nullmailer.m_re_rwset = nullmailer.m_rh_rwset =
779                             nullmailer.m_se_rwset = nullmailer.m_sh_rwset = -1;
780             nullmailer.m_eol = "\n";
781             memset(&mcibuf, '\0', sizeof(mcibuf));
782             mcibuf.mci_mailer = &nullmailer;
783             mcibuf.mci_out = tfp;
784
785             macdefine(&e->e_macro, A_PERM, 'g', "\201f");
786             for (h = e->e_header; h != NULL; h = h->h_link)
787             {
```

```
788                 if (h->h_value == NULL)
789                         continue;
790
791                 /* don't output resent headers on non-resent messages */
792                 if (bitset(H_RESENT, h->h_flags) &&
793                     !bitset(EF_RESENT, e->e_flags))
794                         continue;
795
796                 /* expand macros; if null, don't output header at all */
797                 if (bitset(H_DEFAULT, h->h_flags))
798                 {
799                         (void) expand(h->h_value, buf, sizeof(buf), e);
800                         if (buf[0] == '\0')
801                                 continue;
802                         if (buf[0] == ' ' && buf[1] == '\0')
803                                 continue;
804                 }
805
806                 /* output this header */
807                 (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "H?");
808
809                 /* output conditional macro if present */
810                 if (h->h_macro != '\0')
811                 {
812                         if (bitset(0200, h->h_macro))
813                                 (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT,
814                                                      "${%s}",
815                                                      macname(bitidx(h->h_macro)
816                         else
817                                 (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT,
818                                                      "$%c", h->h_macro);
819                 }
820                 else if (!bitzerop(h->h_mflags) &&
821                          bitset(H_CHECK|H_ACHECK, h->h_flags))
822                 {
823                         int j;
824
825                         /* if conditional, output the set of conditions */
826                         for (j = '\0'; j <= '\177'; j++)
827                                 if (bitnset(j, h->h_mflags))
828                                         (void) sm_io_putc(tfp, SM_TIME_DEFAULT,
829                                                           j);
830                 }
831                 (void) sm_io_putc(tfp, SM_TIME_DEFAULT, '?');
832
833                 /* output the header: expand macros, convert addresses */
834                 if (bitset(H_DEFAULT, h->h_flags) &&
835                     !bitset(H_BINDLATE, h->h_flags))
836                 {
837                         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "%s:%s\n",
838                                              h->h_field,
839                                              denlstring(buf, false, true));
840                 }
841                 else if (bitset(H_FROM|H_RCPT, h->h_flags) &&
842                          !bitset(H_BINDLATE, h->h_flags))
843                 {
844                         bool oldstyle = bitset(EF_OLDSTYLE, e->e_flags);
845                         SM_FILE_T *savetrace = TrafficLogFile;
846
847                         TrafficLogFile = NULL;
848
849                         if (bitset(H_FROM, h->h_flags))
850                                 oldstyle = false;
851                         commaize(h, h->h_value, oldstyle, &mcibuf, e,
852                                  PXLF_HEADER);
```



```
854                         TrafficLogFile = savetrace;
855                 }
856                 else
857                 {
858                         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "%s:%s\n",
859                                              h->h_field,
860                                              denlstring(h->h_value, false,
861                                                         true));
862                 }
863         }
864
865         /*
866         **  Clean up.
867         **
868         **      Write a terminator record -- this is to prevent
869         **      scurrilous crackers from appending any data.
870         */
871
872         (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, ".\n");
873
874         if (sm_io_flush(tfp, SM_TIME_DEFAULT) != 0 ||
875             ((SuperSafe == SAFE_REALLY ||
876               SuperSafe == SAFE_REALLY_POSTMILTER ||
877               (SuperSafe == SAFE_INTERACTIVE && msync)) &&
878              fsync(sm_io_getinfo(tfp, SM_IO_WHAT_FD, NULL)) < 0) ||
879             sm_io_error(tfp))
880         {
881                 if (newid)
882                         syserr("!552 Error writing control file %s", tf);
883                 else
884                         syserr("!452 Error writing control file %s", tf);
885         }
886
887         if (!newid)
888         {
889                 char new = queue_letter(e, ANYQFL_LETTER);
890
891                 /* rename (locked) tf to be (locked) [qh]f */
892                 (void) sm_strlcpy(qf, queuename(e, ANYQFL_LETTER),
893                                   sizeof(qf));
894                 if (rename(tf, qf) < 0)
895                         syserr("cannot rename(%s, %s), uid=%d",
896                                tf, qf, (int) geteuid());
897                 else
898                 {
899                         /*
900                         **  Check if type has changed and only
901                         **  remove the old item if the rename above
902                         **  succeeded.
903                         */
904
905                         if (e->e_qfletter != '\0' &&
906                             e->e_qfletter != new)
907                         {
908                                 if (tTd(40, 5))
909                                 {
910                                         sm_dprintf("type changed from %c to %c\n
911                                                    e->e_qfletter, new);
912                                 }
913
914                                 if (unlink(queuename(e, e->e_qfletter)) < 0)
915                                 {
916                                         /* XXX: something more drastic? */
917                                         if (LogLevel > 0)
918                                                 sm_syslog(LOG_ERR, e->e_id,
919                                                           "queueup: unlink(%s) f
```

```
                                                        queuename(e, e->e_qfle
                                                        sm_errstring(errno));
                                }
                        }
                }
                e->e_qfletter = new;

                /*
                **  fsync() after renaming to make sure metadata is
                **  written to disk on filesystems in which renames are
                **  not guaranteed.
                */

                if (SuperSafe != SAFE_NO)
                {
                        /* for softupdates */
                        if (tfd >= 0 && fsync(tfd) < 0)
                        {
                                syserr("!queueup: cannot fsync queue temp file %
                                        tf);
                        }
                        SYNC_DIR(qf, true);
                }

                /* close and unlock old (locked) queue file */
                if (e->e_lockfp != NULL)
                        (void) sm_io_close(e->e_lockfp, SM_TIME_DEFAULT);
                e->e_lockfp = tfp;

                /* save log info */
                if (LogLevel > 79)
                        sm_syslog(LOG_DEBUG, e->e_id, "queueup %s", qf);
        }
        else
        {
                /* save log info */
                if (LogLevel > 79)
                        sm_syslog(LOG_DEBUG, e->e_id, "queueup %s", tf);

                e->e_qfletter = queue_letter(e, ANYQFL_LETTER);
        }

        errno = 0;
        e->e_flags |= EF_INQUEUE;

        if (tTd(40, 1))
                sm_dprintf("<<<<< done queueing %s <<<<<\n\n", e->e_id);
        return;
}

/*
**  PRINTCTLADDR -- print control address to file.
**
**      Parameters:
**              a -- address.
**              tfp -- file pointer.
**
**      Returns:
**              none.
**
**      Side Effects:
**              The control address (if changed) is printed to the file.
**              The last control address and uid are saved.
*/

static void
```



```
printctladdr(a, tfp)
        register ADDRESS *a;
        SM_FILE_T *tfp;
{
        char *user;
        register ADDRESS *q;
        uid_t uid;
        gid_t gid;
        static ADDRESS *lastctladdr = NULL;
        static uid_t lastuid;

        /* initialization */
        if (a == NULL || a->q_alias == NULL || tfp == NULL)
        {
                if (lastctladdr != NULL && tfp != NULL)
                        (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "C\n");
                lastctladdr = NULL;
                lastuid = 0;
                return;
        }

        /* find the active uid */
        q = getctladdr(a);
        if (q == NULL)
        {
                user = NULL;
                uid = 0;
                gid = 0;
        }
        else
        {
                user = q->q_ruser != NULL ? q->q_ruser : q->q_user;
                uid = q->q_uid;
                gid = q->q_gid;
        }
        a = a->q_alias;

        /* check to see if this is the same as last time */
        if (lastctladdr != NULL && uid == lastuid &&
            strcmp(lastctladdr->q_paddr, a->q_paddr) == 0)
                return;
        lastuid = uid;
        lastctladdr = a;

        if (uid == 0 || user == NULL || user[0] == '\0')
                (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "C");
        else
                (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, "C%s:%ld:%ld",
                                     denlstring(user, true, false), (long) uid,
                                     (long) gid);
        (void) sm_io_fprintf(tfp, SM_TIME_DEFAULT, ":%s\n",
                             denlstring(a->q_paddr, true, false));
}

/*
**  RUNNERS_SIGTERM -- propagate a SIGTERM to queue runner process
**
**      This propagates the signal to the child processes that are queue
**      runners. This is for a queue runner "cleanup". After all of the
**      child queue runner processes are signaled (it should be SIGTERM
**      being the sig) then the old signal handler (Oldsh) is called
**      to handle any cleanup set for this process (provided it is not
**      SIG_DFL or SIG_IGN). The signal may not be handled immediately
**      if the BlockOldsh flag is set. If the current process doesn't
**      have a parent then handle the signal immediately, regardless of
**      BlockOldsh.
```

```
1052 **
1053 **      Parameters:
1054 **              sig -- the signal number being sent
1055 **
1056 **      Returns:
1057 **              none.
1058 **
1059 **      Side Effects:
1060 **              Sets the NoMoreRunners boolean to true to stop more runners
1061 **              from being started in runqueue().
1062 **
1063 **      NOTE:   THIS CAN BE CALLED FROM A SIGNAL HANDLER.  DO NOT ADD
1064 **              ANYTHING TO THIS ROUTINE UNLESS YOU KNOW WHAT YOU ARE
1065 **              DOING.
1066 */

1068 static bool             volatile NoMoreRunners = false;
1069 static sigfunc_t        Oldsh_term = SIG_DFL;
1070 static sigfunc_t        Oldsh_hup = SIG_DFL;
1071 static sigfunc_t        volatile Oldsh = SIG_DFL;
1072 static bool             BlockOldsh = false;
1073 static int              volatile Oldsig = 0;
1074 static SIGFUNC_DECL     runners_sigterm __P((int));
1075 static SIGFUNC_DECL     runners_sighup __P((int));

1077 static SIGFUNC_DECL
1078 runners_sigterm(sig)
1079         int sig;
1080 {
1081         int save_errno = errno;

1083         FIX_SYSV_SIGNAL(sig, runners_sigterm);
1084         errno = save_errno;
1085         CHECK_CRITICAL(sig);
1086         NoMoreRunners = true;
1087         Oldsh = Oldsh_term;
1088         Oldsig = sig;
1089         proc_list_signal(PROC_QUEUE, sig);

1091         if (!BlockOldsh || getppid() <= 1)
1092         {
1093                 /* Check that a valid 'old signal handler' is callable */
1094                 if (Oldsh_term != SIG_DFL && Oldsh_term != SIG_IGN &&
1095                     Oldsh_term != runners_sigterm)
1096                         (*Oldsh_term)(sig);
1097         }
1098         errno = save_errno;
1099         return SIGFUNC_RETURN;
1100 }
1101 /*
1102 **  RUNNERS_SIGHUP -- propagate a SIGHUP to queue runner process
1103 **
1104 **      This propagates the signal to the child processes that are queue
1105 **      runners. This is for a queue runner "cleanup". After all of the
1106 **      child queue runner processes are signaled (it should be SIGHUP
1107 **      being the sig) then the old signal handler (Oldsh) is called to
1108 **      handle any cleanup set for this process (provided it is not SIG_DFL
1109 **      or SIG_IGN). The signal may not be handled immediately if the
1110 **      BlockOldsh flag is set. If the current process doesn't have
1111 **      a parent then handle the signal immediately, regardless of
1112 **      BlockOldsh.
1113 **
1114 **      Parameters:
1115 **              sig -- the signal number being sent
1116 **
1117 **      Returns:
```



```
1118 **              none.
1119 **
1120 **      Side Effects:
1121 **              Sets the NoMoreRunners boolean to true to stop more runners
1122 **              from being started in runqueue().
1123 **
1124 **      NOTE:   THIS CAN BE CALLED FROM A SIGNAL HANDLER.  DO NOT ADD
1125 **              ANYTHING TO THIS ROUTINE UNLESS YOU KNOW WHAT YOU ARE
1126 **              DOING.
1127 */

1129 static SIGFUNC_DECL
1130 runners_sighup(sig)
1131         int sig;
1132 {
1133         int save_errno = errno;

1135         FIX_SYSV_SIGNAL(sig, runners_sighup);
1136         errno = save_errno;
1137         CHECK_CRITICAL(sig);
1138         NoMoreRunners = true;
1139         Oldsh = Oldsh_hup;
1140         Oldsig = sig;
1141         proc_list_signal(PROC_QUEUE, sig);

1143         if (!BlockOldsh || getppid() <= 1)
1144         {
1145                 /* Check that a valid 'old signal handler' is callable */
1146                 if (Oldsh_hup != SIG_DFL && Oldsh_hup != SIG_IGN &&
1147                     Oldsh_hup != runners_sighup)
1148                         (*Oldsh_hup)(sig);
1149         }
1150         errno = save_errno;
1151         return SIGFUNC_RETURN;
1152 }
1153 /*
1154 **  MARK_WORK_GROUP_RESTART -- mark a work group as needing a restart
1155 **
1156 **  Sets a workgroup for restarting.
1157 **
1158 **      Parameters:
1159 **              wgrp -- the work group id to restart.
1160 **              reason -- why (signal?), -1 to turn off restart
1161 **
1162 **      Returns:
1163 **              none.
1164 **
1165 **      Side effects:
1166 **              May set global RestartWorkGroup to true.
1167 **
1168 **      NOTE:   THIS CAN BE CALLED FROM A SIGNAL HANDLER.  DO NOT ADD
1169 **              ANYTHING TO THIS ROUTINE UNLESS YOU KNOW WHAT YOU ARE
1170 **              DOING.
1171 */

1173 void
1174 mark_work_group_restart(wgrp, reason)
1175         int wgrp;
1176         int reason;
1177 {
1178         if (wgrp < 0 || wgrp > NumWorkGroups)
1179                 return;

1181         WorkGrp[wgrp].wg_restart = reason;
1182         if (reason >= 0)
1183                 RestartWorkGroup = true;
```

```
}
/*
**  RESTART_MARKED_WORK_GROUPS -- restart work groups marked as needing restart
**
**  Restart any workgroup marked as needing a restart provided more
**  runners are allowed.
**
**      Parameters:
**              none.
**
**      Returns:
**              none.
**
**      Side effects:
**              Sets global RestartWorkGroup to false.
*/

void
restart_marked_work_groups()
{
        int i;
        int wasblocked;

        if (NoMoreRunners)
                return;

        /* Block SIGCHLD so reapchild() doesn't mess with us */
        wasblocked = sm_blocksignal(SIGCHLD);

        for (i = 0; i < NumWorkGroups; i++)
        {
                if (WorkGrp[i].wg_restart >= 0)
                {
                        if (LogLevel > 8)
                                sm_syslog(LOG_ERR, NOQID,
                                          "restart queue runner=%d due to signal
                                          i, WorkGrp[i].wg_restart);
                        restart_work_group(i);
                }
        }
        RestartWorkGroup = false;

        if (wasblocked == 0)
                (void) sm_releasesignal(SIGCHLD);
}
/*
**  RESTART_WORK_GROUP -- restart a specific work group
**
**  Restart a specific workgroup provided more runners are allowed.
**  If the requested work group has been restarted too many times log
**  this and refuse to restart.
**
**      Parameters:
**              wgrp -- the work group id to restart
**
**      Returns:
**              none.
**
**      Side Effects:
**              starts another process doing the work of wgrp
*/

#define MAX_PERSIST_RESTART     10      /* max allowed number of restarts */

static void
restart_work_group(wgrp)
```



```
        int wgrp;
{
        if (NoMoreRunners ||
            wgrp < 0 || wgrp > NumWorkGroups)
                return;

        WorkGrp[wgrp].wg_restart = -1;
        if (WorkGrp[wgrp].wg_restartcnt < MAX_PERSIST_RESTART)
        {
                /* avoid overflow; increment here */
                WorkGrp[wgrp].wg_restartcnt++;
                (void) run_work_group(wgrp, RWG_FORK|RWG_PERSISTENT|RWG_RUNALL);
        }
        else
        {
                sm_syslog(LOG_ERR, NOQID,
                          "ERROR: persistent queue runner=%d restarted too many
                          wgrp);
        }
}
/*
**  SCHEDULE_QUEUE_RUNS -- schedule the next queue run for a work group.
**
**      Parameters:
**              runall -- schedule even if individual bit is not set.
**              wgrp -- the work group id to schedule.
**              didit -- the queue run was performed for this work group.
**
**      Returns:
**              nothing
*/

#define INCR_MOD(v, m)  if (++v >= m)   \
                                v = 0;  \
                        else

static void
schedule_queue_runs(runall, wgrp, didit)
        bool runall;
        int wgrp;
        bool didit;
{
        int qgrp, cgrp, endgrp;
#if _FFR_QUEUE_SCHED_DBG
        time_t lastsched;
        bool sched;
#endif /* _FFR_QUEUE_SCHED_DBG */
        time_t now;
        time_t minqintvl;

        /*
        **  This is a bit ugly since we have to duplicate the
        **  code that "walks" through a work queue group.
        */

        now = curtime();
        minqintvl = 0;
        cgrp = endgrp = WorkGrp[wgrp].wg_curqgrp;
        do
        {
                time_t qintvl;

#if _FFR_QUEUE_SCHED_DBG
                lastsched = 0;
                sched = false;
#endif /* _FFR_QUEUE_SCHED_DBG */
```

```
1316                 qgrp = WorkGrp[wgrp].wg_qgs[cgrp]->qg_index;
1317                 if (Queue[qgrp]->qg_queueintvl > 0)
1318                         qintvl = Queue[qgrp]->qg_queueintvl;
1319                 else if (QueueIntvl > 0)
1320                         qintvl = QueueIntvl;
1321                 else
1322                         qintvl = (time_t) 0;
1323 #if _FFR_QUEUE_SCHED_DBG
1324                 lastsched = Queue[qgrp]->qg_nextrun;
1325 #endif /* _FFR_QUEUE_SCHED_DBG */
1326                 if ((runall || Queue[qgrp]->qg_nextrun <= now) && qintvl > 0)
1327                 {
1328 #if _FFR_QUEUE_SCHED_DBG
1329                         sched = true;
1330 #endif /* _FFR_QUEUE_SCHED_DBG */
1331                         if (minqintvl == 0 || qintvl < minqintvl)
1332                                 minqintvl = qintvl;

1334                         /*
1335                         **  Only set a new time if a queue run was performed
1336                         **  for this queue group.  If the queue was not run,
1337                         **  we could starve it by setting a new time on each
1338                         **  call.
1339                         */

1341                         if (didit)
1342                                 Queue[qgrp]->qg_nextrun += qintvl;
1343                 }
1344 #if _FFR_QUEUE_SCHED_DBG
1345                 if (tTd(69, 10))
1346                         sm_syslog(LOG_INFO, NOQID,
1347                                 "sqr: wgrp=%d, cgrp=%d, qgrp=%d, intvl=%ld, QI=%
1348                                 wgrp, cgrp, qgrp, Queue[qgrp]->qg_queueintvl,
1349                                 QueueIntvl, runall, lastsched,
1350                                 Queue[qgrp]->qg_nextrun, sched);
1351 #endif /* _FFR_QUEUE_SCHED_DBG */
1352                 INCR_MOD(cgrp, WorkGrp[wgrp].wg_numqgrp);
1353         } while (endgrp != cgrp);
1354         if (minqintvl > 0)
1355                 (void) sm_setevent(minqintvl, runqueueevent, 0);
1356 }

1358 #if _FFR_QUEUE_RUN_PARANOIA
1359 /*
1360 **  CHECKQUEUERUNNER -- check whether a queue group hasn't been run.
1361 **
1362 **      Use this if events may get lost and hence queue runners may not
1363 **      be started and mail will pile up in a queue.
1364 **
1365 **      Parameters:
1366 **              none.
1367 **
1368 **      Returns:
1369 **              true if a queue run is necessary.
1370 **
1371 **      Side Effects:
1372 **              may schedule a queue run.
1373 */

1375 bool
1376 checkqueuerunner()
1377 {
1378         int qgrp;
1379         time_t now, minqintvl;

1381         now = curtime();
```



```
1382         minqintvl = 0;
1383         for (qgrp = 0; qgrp < NumQueue && Queue[qgrp] != NULL; qgrp++)
1384         {
1385                 time_t qintvl;

1387                 if (Queue[qgrp]->qg_queueintvl > 0)
1388                         qintvl = Queue[qgrp]->qg_queueintvl;
1389                 else if (QueueIntvl > 0)
1390                         qintvl = QueueIntvl;
1391                 else
1392                         qintvl = (time_t) 0;
1393                 if (Queue[qgrp]->qg_nextrun <= now - qintvl)
1394                 {
1395                         if (minqintvl == 0 || qintvl < minqintvl)
1396                                 minqintvl = qintvl;
1397                         if (LogLevel > 1)
1398                                 sm_syslog(LOG_WARNING, NOQID,
1399                                         "checkqueuerunner: queue %d should have
1400                                         qgrp,
1401                                         arpadate(ctime(&Queue[qgrp]->qg_nextrun)
1402                                         qintvl);
1403                 }
1404         }
1405         if (minqintvl > 0)
1406         {
1407                 (void) sm_setevent(minqintvl, runqueueevent, 0);
1408                 return true;
1409         }
1410         return false;
1411 }
1412 #endif /* _FFR_QUEUE_RUN_PARANOIA */

1414 /*
1415 **  RUNQUEUE -- run the jobs in the queue.
1416 **
1417 **      Gets the stuff out of the queue in some presumably logical
1418 **      order and processes them.
1419 **
1420 **      Parameters:
1421 **              forkflag -- true if the queue scanning should be done in
1422 **                      a child process.  We double-fork so it is not our
1423 **                      child and we don't have to clean up after it.
1424 **                      false can be ignored if we have multiple queues.
1425 **              verbose -- if true, print out status information.
1426 **              persistent -- persistent queue runner?
1427 **              runall -- run all groups or only a subset (DoQueueRun)?
1428 **
1429 **      Returns:
1430 **              true if the queue run successfully began.
1431 **
1432 **      Side Effects:
1433 **              runs things in the mail queue using run_work_group().
1434 **              maybe schedules next queue run.
1435 */

1437 static ENVELOPE QueueEnvelope;          /* the queue run envelope */
1438 static time_t   LastQueueTime = 0;      /* last time a queue ID assigned */
1439 static pid_t    LastQueuePid = -1;      /* last PID which had a queue ID */

1441 /* values for qp_supdirs */
1442 #define QP_NOSUB        0x0000  /* No subdirectories */
1443 #define QP_SUBDF        0x0001  /* "df" subdirectory */
1444 #define QP_SUBQF        0x0002  /* "qf" subdirectory */
1445 #define QP_SUBXF        0x0004  /* "xf" subdirectory */

1447 bool
```

```
runqueue(forkflag, verbose, persistent, runall)
        bool forkflag;
        bool verbose;
        bool persistent;
        bool runall;
{
        int i;
        bool ret = true;
        static int curnum = 0;
        sigfunc_t cursh;
#if SM_HEAP_CHECK
        SM_NONVOLATILE int oldgroup = 0;

        if (sm_debug_active(&DebugLeakQ, 1))
        {
                oldgroup = sm_heap_group();
                sm_heap_newgroup();
                sm_dprintf("runqueue() heap group #%d\n", sm_heap_group());
        }
#endif /* SM_HEAP_CHECK */

        /* queue run has been started, don't do any more this time */
        DoQueueRun = false;

        /* more than one queue or more than one directory per queue */
        if (!forkflag && !verbose &&
            (WorkGrp[0].wg_qgs[0]->qg_numqueues > 1 || NumWorkGroups > 1 ||
             WorkGrp[0].wg_numqgrp > 1))
                forkflag = true;

        /*
        **  For controlling queue runners via signals sent to this process.
        **  Oldsh* will get called too by runners_sig* (if it is not SIG_IGN
        **  or SIG_DFL) to preserve cleanup behavior. Now that this process
        **  will have children (and perhaps grandchildren) this handler will
        **  be left in place. This is because this process, once it has
        **  finished spinning off queue runners, may go back to doing something
        **  else (like being a daemon). And we still want on a SIG{TERM,HUP} to
        **  clean up the child queue runners. Only install 'runners_sig*' once
        **  else we'll get stuck looping forever.
        */

        cursh = sm_signal(SIGTERM, runners_sigterm);
        if (cursh != runners_sigterm)
                Oldsh_term = cursh;
        cursh = sm_signal(SIGHUP, runners_sighup);
        if (cursh != runners_sighup)
                Oldsh_hup = cursh;

        for (i = 0; i < NumWorkGroups && !NoMoreRunners; i++)
        {
                int rwgflags = RWG_NONE;

                /*
                **  If MaxQueueChildren active then test whether the start
                **  of the next queue group's additional queue runners (maximum)
                **  will result in MaxQueueChildren being exceeded.
                **
                **  Note: do not use continue; even though another workgroup
                **      may have fewer queue runners, this would be "unfair",
                **      i.e., this work group might "starve" then.
                */

#if _FFR_QUEUE_SCHED_DBG
                if (tTd(69, 10))
                        sm_syslog(LOG_INFO, NOQID,
```



```
                                "rq: curnum=%d, MaxQueueChildren=%d, CurRunners=
                                curnum, MaxQueueChildren, CurRunners,
                                WorkGrp[curnum].wg_maxact);
#endif /* _FFR_QUEUE_SCHED_DBG */
                if (MaxQueueChildren > 0 &&
                    CurRunners + WorkGrp[curnum].wg_maxact > MaxQueueChildren)
                        break;

                /*
                **  Pick up where we left off (curnum), in case we
                **  used up all the children last time without finishing.
                **  This give a round-robin fairness to queue runs.
                **
                **  Increment CurRunners before calling run_work_group()
                **  to avoid a "race condition" with proc_list_drop() which
                **  decrements CurRunners if the queue runners terminate.
                **  Notice: CurRunners is an upper limit, in some cases
                **  (too few jobs in the queue) this value is larger than
                **  the actual number of queue runners. The discrepancy can
                **  increase if some queue runners "hang" for a long time.
                */

                CurRunners += WorkGrp[curnum].wg_maxact;
                if (forkflag)
                        rwgflags |= RWG_FORK;
                if (verbose)
                        rwgflags |= RWG_VERBOSE;
                if (persistent)
                        rwgflags |= RWG_PERSISTENT;
                if (runall)
                        rwgflags |= RWG_RUNALL;
                ret = run_work_group(curnum, rwgflags);

                /*
                **  Failure means a message was printed for ETRN
                **  and subsequent queues are likely to fail as well.
                **  Decrement CurRunners in that case because
                **  none have been started.
                */

                if (!ret)
                {
                        CurRunners -= WorkGrp[curnum].wg_maxact;
                        break;
                }

                if (!persistent)
                        schedule_queue_runs(runall, curnum, true);
                INCR_MOD(curnum, NumWorkGroups);
        }

        /* schedule left over queue runs */
        if (i < NumWorkGroups && !NoMoreRunners && !persistent)
        {
                int h;

                for (h = curnum; i < NumWorkGroups; i++)
                {
                        schedule_queue_runs(runall, h, false);
                        INCR_MOD(h, NumWorkGroups);
                }
        }


#if SM_HEAP_CHECK
        if (sm_debug_active(&DebugLeakQ, 1))
```

```
1580                 sm_heap_setgroup(oldgroup);
1581 #endif /* SM_HEAP_CHECK */
1582         return ret;
1583 }

1585 #if _FFR_SKIP_DOMAINS
1586 /*
1587 **  SKIP_DOMAINS -- Skip 'skip' number of domains in the WorkQ.
1588 **
1589 **  Added by Stephen Frost <sfrost@snowman.net> to support
1590 **  having each runner process every N'th domain instead of
1591 **  every N'th message.
1592 **
1593 **      Parameters:
1594 **              skip -- number of domains in WorkQ to skip.
1595 **
1596 **      Returns:
1597 **              total number of messages skipped.
1598 **
1599 **      Side Effects:
1600 **              may change WorkQ
1601 */

1603 static int
1604 skip_domains(skip)
1605         int skip;
1606 {
1607         int n, seqjump;

1609         for (n = 0, seqjump = 0; n < skip && WorkQ != NULL; seqjump++)
1610         {
1611                 if (WorkQ->w_next != NULL)
1612                 {
1613                         if (WorkQ->w_host != NULL &&
1614                             WorkQ->w_next->w_host != NULL)
1615                         {
1616                                 if (sm_strcasecmp(WorkQ->w_host,
1617                                                   WorkQ->w_next->w_host) != 0)
1618                                         n++;
1619                         }
1620                         else
1621                         {
1622                                 if ((WorkQ->w_host != NULL &&
1623                                      WorkQ->w_next->w_host == NULL) ||
1624                                     (WorkQ->w_host == NULL &&
1625                                      WorkQ->w_next->w_host != NULL))
1626                                         n++;
1627                         }
1628                 }
1629                 WorkQ = WorkQ->w_next;
1630         }
1631         return seqjump;
1632 }
1633 #endif /* _FFR_SKIP_DOMAINS */

1635 /*
1636 **  RUNNER_WORK -- have a queue runner do its work
1637 **
1638 **  Have a queue runner do its work a list of entries.
1639 **  When work isn't directly being done then this process can take a signal
1640 **  and terminate immediately (in a clean fashion of course).
1641 **  When work is directly being done, it's not to be interrupted
1642 **  immediately: the work should be allowed to finish at a clean point
1643 **  before termination (in a clean fashion of course).
1644 **
1645 **      Parameters:
```



```
1646 **              e -- envelope.
1647 **              sequenceno -- 'th process to run WorkQ.
1648 **              didfork -- did the calling process fork()?
1649 **              skip -- process only each skip'th item.
1650 **              njobs -- number of jobs in WorkQ.
1651 **
1652 **      Returns:
1653 **              none.
1654 **
1655 **      Side Effects:
1656 **              runs things in the mail queue.
1657 */

1659 static void
1660 runner_work(e, sequenceno, didfork, skip, njobs)
1661         register ENVELOPE *e;
1662         int sequenceno;
1663         bool didfork;
1664         int skip;
1665         int njobs;
1666 {
1667         int n, seqjump;
1668         WORK *w;
1669         time_t now;

1671         SM_GET_LA(now);

1673         /*
1674         **  Here we temporarily block the second calling of the handlers.
1675         **  This allows us to handle the signal without terminating in the
1676         **  middle of direct work. If a signal does come, the test for
1677         **  NoMoreRunners will find it.
1678         */

1680         BlockOldsh = true;
1681         seqjump = skip;

1683         /* process them once at a time */
1684         while (WorkQ != NULL)
1685         {
1686 #if SM_HEAP_CHECK
1687                 SM_NONVOLATILE int oldgroup = 0;

1689                 if (sm_debug_active(&DebugLeakQ, 1))
1690                 {
1691                         oldgroup = sm_heap_group();
1692                         sm_heap_newgroup();
1693                         sm_dprintf("run_queue_group() heap group #%d\n",
1694                                 sm_heap_group());
1695                 }
1696 #endif /* SM_HEAP_CHECK */

1698                 /* do no more work */
1699                 if (NoMoreRunners)
1700                 {
1701                         /* Check that a valid signal handler is callable */
1702                         if (Oldsh != SIG_DFL && Oldsh != SIG_IGN &&
1703                             Oldsh != runners_sighup &&
1704                             Oldsh != runners_sigterm)
1705                                 (*Oldsh)(Oldsig);
1706                         break;
1707                 }

1709                 w = WorkQ; /* assign current work item */

1711                 /*
```

```
1712                 **  Set the head of the WorkQ to the next work item.
1713                 **  It is set 'skip' ahead (the number of parallel queue
1714                 **  runners working on WorkQ together) since each runner
1715                 **  works on every 'skip'th (N-th) item.
1716 #if _FFR_SKIP_DOMAINS
1717                 **  In the case of the BYHOST Queue Sort Order, the 'item'
1718                 **  is a domain, so we work on every 'skip'th (N-th) domain.
1719 #endif * _FFR_SKIP_DOMAINS *
1720                 */

1722 #if _FFR_SKIP_DOMAINS
1723                 if (QueueSortOrder == QSO_BYHOST)
1724                 {
1725                         seqjump = 1;
1726                         if (WorkQ->w_next != NULL)
1727                         {
1728                                 if (WorkQ->w_host != NULL &&
1729                                     WorkQ->w_next->w_host != NULL)
1730                                 {
1731                                         if (sm_strcasecmp(WorkQ->w_host,
1732                                                           WorkQ->w_next->w_host)
1733                                                                 != 0)
1734                                                 seqjump = skip_domains(skip);
1735                                         else
1736                                                 WorkQ = WorkQ->w_next;
1737                                 }
1738                                 else
1739                                 {
1740                                         if ((WorkQ->w_host != NULL &&
1741                                              WorkQ->w_next->w_host == NULL) ||
1742                                             (WorkQ->w_host == NULL &&
1743                                              WorkQ->w_next->w_host != NULL))
1744                                                 seqjump = skip_domains(skip);
1745                                         else
1746                                                 WorkQ = WorkQ->w_next;
1747                                 }
1748                         }
1749                         else
1750                                 WorkQ = WorkQ->w_next;
1751                 }
1752                 else
1753 #endif /* _FFR_SKIP_DOMAINS */
1754                 {
1755                         for (n = 0; n < skip && WorkQ != NULL; n++)
1756                                 WorkQ = WorkQ->w_next;
1757                 }

1759                 e->e_to = NULL;

1761                 /*
1762                 **  Ignore jobs that are too expensive for the moment.
1763                 **
1764                 **      Get new load average every GET_NEW_LA_TIME seconds.
1765                 */

1767                 SM_GET_LA(now);
1768                 if (shouldqueue(WkRecipFact, Current_LA_time))
1769                 {
1770                         char *msg = "Aborting queue run: load average too high";

1772                         if (Verbose)
1773                                 message("%s", msg);
1774                         if (LogLevel > 8)
1775                                 sm_syslog(LOG_INFO, NOQID, "runqueue: %s", msg);
1776                         break;
1777                 }
```



```
1778                 if (shouldqueue(w->w_pri, w->w_ctime))
1779                 {
1780                         if (Verbose)
1781                                 message(EmptyString);
1782                         if (QueueSortOrder == QSO_BYPRIORITY)
1783                         {
1784                                 if (Verbose)
1785                                         message("Skipping %s/%s (sequence %d of
1786                                                 qid_printqueue(w->w_qgrp,
1787                                                                w->w_qdir),
1788                                                 w->w_name + 2, sequenceno,
1789                                                 njobs);
1790                                 if (LogLevel > 8)
1791                                         sm_syslog(LOG_INFO, NOQID,
1792                                                   "runqueue: Flushing queue from
1793                                                   qid_printqueue(w->w_qgrp,
1794                                                                  w->w_qdir),
1795                                                   w->w_name + 2, w->w_pri,
1796                                                   CurrentLA, sequenceno,
1797                                                   njobs);
1798                                 break;
1799                         }
1800                         else if (Verbose)
1801                                 message("Skipping %s/%s (sequence %d of %d)",
1802                                         qid_printqueue(w->w_qgrp, w->w_qdir),
1803                                         w->w_name + 2, sequenceno, njobs);
1804                 }
1805                 else
1806                 {
1807                         if (Verbose)
1808                         {
1809                                 message(EmptyString);
1810                                 message("Running %s/%s (sequence %d of %d)",
1811                                         qid_printqueue(w->w_qgrp, w->w_qdir),
1812                                         w->w_name + 2, sequenceno, njobs);
1813                         }
1814                         if (didfork && MaxQueueChildren > 0)
1815                         {
1816                                 sm_blocksignal(SIGCHLD);
1817                                 (void) sm_signal(SIGCHLD, reapchild);
1818                         }
1819                         if (tTd(63, 100))
1820                                 sm_syslog(LOG_DEBUG, NOQID,
1821                                           "runqueue %s dowork(%s)",
1822                                           qid_printqueue(w->w_qgrp, w->w_qdir),
1823                                           w->w_name + 2);

1825                         (void) dowork(w->w_qgrp, w->w_qdir, w->w_name + 2,
1826                                       ForkQueueRuns, false, e);
1827                         errno = 0;
1828                 }
1829                 sm_free(w->w_name); /* XXX */
1830                 if (w->w_host != NULL)
1831                         sm_free(w->w_host); /* XXX */
1832                 sm_free((char *) w); /* XXX */
1833                 sequenceno += seqjump; /* next sequence number */
1834 #if SM_HEAP_CHECK
1835                 if (sm_debug_active(&DebugLeakQ, 1))
1836                         sm_heap_setgroup(oldgroup);
1837 #endif /* SM_HEAP_CHECK */
1838         }

1840         BlockOldsh = false;

1842         /* check the signals didn't happen during the revert */
1843         if (NoMoreRunners)
```

```
1844         {
1845                 /* Check that a valid signal handler is callable */
1846                 if (Oldsh != SIG_DFL && Oldsh != SIG_IGN &&
1847                     Oldsh != runners_sighup && Oldsh != runners_sigterm)
1848                         (*Oldsh)(Oldsig);
1849         }

1851         Oldsh = SIG_DFL; /* after the NoMoreRunners check */
1852 }
1853 /*
1854 **  RUN_WORK_GROUP -- run the jobs in a queue group from a work group.
1855 **
1856 **      Gets the stuff out of the queue in some presumably logical
1857 **      order and processes them.
1858 **
1859 **      Parameters:
1860 **              wgrp -- work group to process.
1861 **              flags -- RWG_* flags
1862 **
1863 **      Returns:
1864 **              true if the queue run successfully began.
1865 **
1866 **      Side Effects:
1867 **              runs things in the mail queue.
1868 */

1870 /* Minimum sleep time for persistent queue runners */
1871 #define MIN_SLEEP_TIME  5

1873 bool
1874 run_work_group(wgrp, flags)
1875         int wgrp;
1876         int flags;
1877 {
1878         register ENVELOPE *e;
1879         int njobs, qdir;
1880         int sequenceno = 1;
1881         int qgrp, endgrp, h, i;
1882         time_t now;
1883         bool full, more;
1884         SM_RPOOL_T *rpool;
1885         extern ENVELOPE BlankEnvelope;
1886         extern SIGFUNC_DECL reapchild __P((int));

1888         if (wgrp < 0)
1889                 return false;

1891         /*
1892         **  If no work will ever be selected, don't even bother reading
1893         **  the queue.
1894         */

1896         SM_GET_LA(now);

1898         if (!bitset(RWG_PERSISTENT, flags) &&
1899             shouldqueue(WkRecipFact, Current_LA_time))
1900         {
1901                 char *msg = "Skipping queue run -- load average too high";

1903                 if (bitset(RWG_VERBOSE, flags))
1904                         message("458 %s\n", msg);
1905                 if (LogLevel > 8)
1906                         sm_syslog(LOG_INFO, NOQID, "runqueue: %s", msg);
1907                 return false;
1908         }
```



```
1910         /*
1911         **  See if we already have too many children.
1912         */

1914         if (bitset(RWG_FORK, flags) &&
1915             WorkGrp[wgrp].wg_lowqintvl > 0 &&
1916             !bitset(RWG_PERSISTENT, flags) &&
1917             MaxChildren > 0 && CurChildren >= MaxChildren)
1918         {
1919                 char *msg = "Skipping queue run -- too many children";

1921                 if (bitset(RWG_VERBOSE, flags))
1922                         message("458 %s (%d)\n", msg, CurChildren);
1923                 if (LogLevel > 8)
1924                         sm_syslog(LOG_INFO, NOQID, "runqueue: %s (%d)",
1925                                   msg, CurChildren);
1926                 return false;
1927         }

1929         /*
1930         **  See if we want to go off and do other useful work.
1931         */

1933         if (bitset(RWG_FORK, flags))
1934         {
1935                 pid_t pid;

1937                 (void) sm_blocksignal(SIGCHLD);
1938                 (void) sm_signal(SIGCHLD, reapchild);

1940                 pid = dofork();
1941                 if (pid == -1)
1942                 {
1943                         const char *msg = "Skipping queue run -- fork() failed";
1944                         const char *err = sm_errstring(errno);

1946                         if (bitset(RWG_VERBOSE, flags))
1947                                 message("458 %s: %s\n", msg, err);
1948                         if (LogLevel > 8)
1949                                 sm_syslog(LOG_INFO, NOQID, "runqueue: %s: %s",
1950                                           msg, err);
1951                         (void) sm_releasesignal(SIGCHLD);
1952                         return false;
1953                 }
1954                 if (pid != 0)
1955                 {
1956                         /* parent -- pick up intermediate zombie */
1957                         (void) sm_blocksignal(SIGALRM);

1959                         /* wgrp only used when queue runners are persistent */
1960                         proc_list_add(pid, "Queue runner", PROC_QUEUE,
1961                                       WorkGrp[wgrp].wg_maxact,
1962                                       bitset(RWG_PERSISTENT, flags) ? wgrp : -1,
1963                                       NULL);
1964                         (void) sm_releasesignal(SIGALRM);
1965                         (void) sm_releasesignal(SIGCHLD);
1966                         return true;
1967                 }

1969                 /* child -- clean up signals */

1971                 /* Reset global flags */
1972                 RestartRequest = NULL;
1973                 RestartWorkGroup = false;
1974                 ShutdownRequest = NULL;
1975                 PendingSignal = 0;
```

```
1976                CurrentPid = getpid();
1977                close_sendmail_pid();

1979                /*
1980                **  Initialize exception stack and default exception
1981                **  handler for child process.
1982                */

1984                sm_exc_newthread(fatal_error);
1985                clrcontrol();
1986                proc_list_clear();

1988                /* Add parent process as first child item */
1989                proc_list_add(CurrentPid, "Queue runner child process",
1990                              PROC_QUEUE_CHILD, 0, -1, NULL);
1991                (void) sm_releasesignal(SIGCHLD);
1992                (void) sm_signal(SIGCHLD, SIG_DFL);
1993                (void) sm_signal(SIGHUP, SIG_DFL);
1994                (void) sm_signal(SIGTERM, intsig);
1995        }

1997        /*
1998        **  Release any resources used by the daemon code.
1999        */

2001        clrdaemon();

2003        /* force it to run expensive jobs */
2004        NoConnect = false;

2006        /* drop privileges */
2007        if (geteuid() == (uid_t) 0)
2008                (void) drop_privileges(false);

2010        /*
2011        **  Create ourselves an envelope
2012        */

2014        CurEnv = &QueueEnvelope;
2015        rpool = sm_rpool_new_x(NULL);
2016        e = newenvelope(&QueueEnvelope, CurEnv, rpool);
2017        e->e_flags = BlankEnvelope.e_flags;
2018        e->e_parent = NULL;

2020        /* make sure we have disconnected from parent */
2021        if (bitset(RWG_FORK, flags))
2022        {
2023                disconnect(1, e);
2024                QuickAbort = false;
2025        }

2027        /*
2028        **  If we are running part of the queue, always ignore stored
2029        **  host status.
2030        */

2032        if (QueueLimitId != NULL || QueueLimitSender != NULL ||
2033            QueueLimitQuarantine != NULL ||
2034            QueueLimitRecipient != NULL)
2035        {
2036                IgnoreHostStatus = true;
2037                MinQueueAge = 0;
2038        }

2040        /*
2041        **  Here is where we choose the queue group from the work group.
```



```
2042        **  The caller of the "domorework" label must setup a new envelope.
2043        */

2045        endgrp = WorkGrp[wgrp].wg_curqgrp; /* to not spin endlessly */

2047    domorework:

2049        /*
2050        **  Run a queue group if:
2051        **  RWG_RUNALL bit is set or the bit for this group is set.
2052        */

2054        now = curtime();
2055        for (;;)
2056        {
2057                /*
2058                **  Find the next queue group within the work group that
2059                **  has been marked as needing a run.
2060                */

2062                qgrp = WorkGrp[wgrp].wg_qgs[WorkGrp[wgrp].wg_curqgrp]->qg_index;
2063                WorkGrp[wgrp].wg_curqgrp++; /* advance */
2064                WorkGrp[wgrp].wg_curqgrp %= WorkGrp[wgrp].wg_numqgrp; /* wrap */
2065                if (bitset(RWG_RUNALL, flags) ||
2066                    (Queue[qgrp]->qg_nextrun <= now &&
2067                     Queue[qgrp]->qg_nextrun != (time_t) -1))
2068                        break;
2069                if (endgrp == WorkGrp[wgrp].wg_curqgrp)
2070                {
2071                        e->e_id = NULL;
2072                        if (bitset(RWG_FORK, flags))
2073                                finis(true, true, ExitStat);
2074                        return true; /* we're done */
2075                }
2076        }

2078        qdir = Queue[qgrp]->qg_curnum; /* round-robin init of queue position */
2079 #if _FFR_QUEUE_SCHED_DBG
2080        if (tTd(69, 12))
2081                sm_syslog(LOG_INFO, NOQID,
2082                        "rwg: wgrp=%d, qgrp=%d, qdir=%d, name=%s, curqgrp=%d, nu
2083                        wgrp, qgrp, qdir, qid_printqueue(qgrp, qdir),
2084                        WorkGrp[wgrp].wg_curqgrp, WorkGrp[wgrp].wg_numqgrp);
2085 #endif /* _FFR_QUEUE_SCHED_DBG */

2087 #if HASNICE
2088        /* tweak niceness of queue runs */
2089        if (Queue[qgrp]->qg_nice > 0)
2090                (void) nice(Queue[qgrp]->qg_nice);
2091 #endif /* HASNICE */

2093        /* XXX running queue group... */
2094        sm_setproctitle(true, CurEnv, "running queue: %s",
2095                        qid_printqueue(qgrp, qdir));

2097        if (LogLevel > 69 || tTd(63, 99))
2098                sm_syslog(LOG_DEBUG, NOQID,
2099                          "runqueue %s, pid=%d, forkflag=%d",
2100                          qid_printqueue(qgrp, qdir), (int) CurrentPid,
2101                          bitset(RWG_FORK, flags));

2103        /*
2104        **  Start making passes through the queue.
2105        **      First, read and sort the entire queue.
2106        **      Then, process the work in that order.
2107        **              But if you take too long, start over.
```

```
2108         */

2110         for (i = 0; i < Queue[qgrp]->qg_numqueues; i++)
2111         {
2112                 (void) gatherq(qgrp, qdir, false, &full, &more, &h);
2113 #if SM_CONF_SHM
2114                 if (ShmId != SM_SHM_NO_ID)
2115                         QSHM_ENTRIES(Queue[qgrp]->qg_qpaths[qdir].qp_idx) = h;
2116 #endif /* SM_CONF_SHM */
2117                 /* If there are no more items in this queue advance */
2118                 if (!more)
2119                 {
2120                         /* A round-robin advance */
2121                         qdir++;
2122                         qdir %= Queue[qgrp]->qg_numqueues;
2123                 }

2125                 /* Has the WorkList reached the limit? */
2126                 if (full)
2127                         break; /* don't try to gather more */
2128         }

2130         /* order the existing work requests */
2131         njobs = sortq(Queue[qgrp]->qg_maxlist);
2132         Queue[qgrp]->qg_curnum = qdir; /* update */


2135         if (!Verbose && bitnset(QD_FORK, Queue[qgrp]->qg_flags))
2136         {
2137                 int loop, maxrunners;
2138                 pid_t pid;

2140                 /*
2141                 **  For this WorkQ we want to fork off N children (maxrunners)
2142                 **  at this point. Each child has a copy of WorkQ. Each child
2143                 **  will process every N-th item. The parent will wait for all
2144                 **  of the children to finish before moving on to the next
2145                 **  queue group within the work group. This saves us forking
2146                 **  a new runner-child for each work item.
2147                 **  It's valid for qg_maxqrun == 0 since this may be an
2148                 **  explicit "don't run this queue" setting.
2149                 */

2151                 maxrunners = Queue[qgrp]->qg_maxqrun;

2153                 /*
2154                 **  If no runners are configured for this group but
2155                 **  the queue is "forced" then lets use 1 runner.
2156                 */

2158                 if (maxrunners == 0 && bitset(RWG_FORCE, flags))
2159                         maxrunners = 1;

2161                 /* No need to have more runners then there are jobs */
2162                 if (maxrunners > njobs)
2163                         maxrunners = njobs;
2164                 for (loop = 0; loop < maxrunners; loop++)
2165                 {
2166                         /*
2167                         **  Since the delivery may happen in a child and the
2168                         **  parent does not wait, the parent may close the
2169                         **  maps thereby removing any shared memory used by
2170                         **  the map.  Therefore, close the maps now so the
2171                         **  child will dynamically open them if necessary.
2172                         */
```



```
2174                         closemaps(false);

2176                         pid = fork();
2177                         if (pid < 0)
2178                         {
2179                                 syserr("run_work_group: cannot fork");
2180                                 return false;
2181                         }
2182                         else if (pid > 0)
2183                         {
2184                                 /* parent -- clean out connection cache */
2185                                 mci_flush(false, NULL);
2186 #if _FFR_SKIP_DOMAINS
2187                                 if (QueueSortOrder == QSO_BYHOST)
2188                                 {
2189                                         sequenceno += skip_domains(1);
2190                                 }
2191                                 else
2192 #endif /* _FFR_SKIP_DOMAINS */
2193                                 {
2194                                         /* for the skip */
2195                                         WorkQ = WorkQ->w_next;
2196                                         sequenceno++;
2197                                 }
2198                                 proc_list_add(pid, "Queue child runner process",
2199                                               PROC_QUEUE_CHILD, 0, -1, NULL);

2201                                 /* No additional work, no additional runners */
2202                                 if (WorkQ == NULL)
2203                                         break;
2204                         }
2205                         else
2206                         {
2207                                 /* child -- Reset global flags */
2208                                 RestartRequest = NULL;
2209                                 RestartWorkGroup = false;
2210                                 ShutdownRequest = NULL;
2211                                 PendingSignal = 0;
2212                                 CurrentPid = getpid();
2213                                 close_sendmail_pid();

2215                                 /*
2216                                 **  Initialize exception stack and default
2217                                 **  exception handler for child process.
2218                                 **  When fork()'d the child now has a private
2219                                 **  copy of WorkQ at its current position.
2220                                 */

2222                                 sm_exc_newthread(fatal_error);

2224                                 /*
2225                                 **  SMTP processes (whether -bd or -bs) set
2226                                 **  SIGCHLD to reapchild to collect
2227                                 **  children status.  However, at delivery
2228                                 **  time, that status must be collected
2229                                 **  by sm_wait() to be dealt with properly
2230                                 **  (check success of delivery based
2231                                 **  on status code, etc).  Therefore, if we
2232                                 **  are an SMTP process, reset SIGCHLD
2233                                 **  back to the default so reapchild
2234                                 **  doesn't collect status before
2235                                 **  sm_wait().
2236                                 */

2238                                 if (OpMode == MD_SMTP ||
2239                                     OpMode == MD_DAEMON ||
```

```
2240                                    MaxQueueChildren > 0)
2241                                {
2242                                        proc_list_clear();
2243                                        sm_releasesignal(SIGCHLD);
2244                                        (void) sm_signal(SIGCHLD, SIG_DFL);
2245                                }

2247                                /* child -- error messages to the transcript */
2248                                QuickAbort = OnlyOneError = false;
2249                                runner_work(e, sequenceno, true,
2250                                            maxrunners, njobs);

2252                                /* This child is done */
2253                                finis(true, true, ExitStat);
2254                                /* NOTREACHED */
2255                        }
2256                }

2258                sm_releasesignal(SIGCHLD);

2260                /*
2261                **  Wait until all of the runners have completed before
2262                **  seeing if there is another queue group in the
2263                **  work group to process.
2264                **  XXX Future enhancement: don't wait() for all children
2265                **  here, just go ahead and make sure that overall the number
2266                **  of children is not exceeded.
2267                */

2269                while (CurChildren > 0)
2270                {
2271                        int status;
2272                        pid_t ret;

2274                        while ((ret = sm_wait(&status)) <= 0)
2275                                continue;
2276                        proc_list_drop(ret, status, NULL);
2277                }
2278        }
2279        else if (Queue[qgrp]->qg_maxqrun > 0 || bitset(RWG_FORCE, flags))
2280        {
2281                /*
2282                **  When current process will not fork children to do the work,
2283                **  it will do the work itself. The 'skip' will be 1 since
2284                **  there are no child runners to divide the work across.
2285                */

2287                runner_work(e, sequenceno, false, 1, njobs);
2288        }

2290        /* free memory allocated by newenvelope() above */
2291        sm_rpool_free(rpool);
2292        QueueEnvelope.e_rpool = NULL;

2294        /* Are there still more queues in the work group to process? */
2295        if (endgrp != WorkGrp[wgrp].wg_curqgrp)
2296        {
2297                rpool = sm_rpool_new_x(NULL);
2298                e = newenvelope(&QueueEnvelope, CurEnv, rpool);
2299                e->e_flags = BlankEnvelope.e_flags;
2300                goto domorework;
2301        }

2303        /* No more queues in work group to process. Now check persistent. */
2304        if (bitset(RWG_PERSISTENT, flags))
2305        {
```



```
2306                sequenceno = 1;
2307                sm_setproctitle(true, CurEnv, "running queue: %s",
2308                                qid_printqueue(qgrp, qdir));

2310                /*
2311                **  close bogus maps, i.e., maps which caused a tempfail,
2312                **      so we get fresh map connections on the next lookup.
2313                **  closemaps() is also called when children are started.
2314                */

2316                closemaps(true);

2318                /* Close any cached connections. */
2319                mci_flush(true, NULL);

2321                /* Clean out expired related entries. */
2322                rmexpstab();

2324 #if NAMED_BIND
2325                /* Update MX records for FallbackMX. */
2326                if (FallbackMX != NULL)
2327                        (void) getfallbackmxrr(FallbackMX);
2328 #endif /* NAMED_BIND */

2330 #if USERDB
2331                /* close UserDatabase */
2332                _udbx_close();
2333 #endif /* USERDB */

2335 #if SM_HEAP_CHECK
2336                if (sm_debug_active(&SmHeapCheck, 2)
2337                    && access("memdump", F_OK) == 0
2338                   )
2339                {
2340                        SM_FILE_T *out;

2342                        remove("memdump");
2343                        out = sm_io_open(SmFtStdio, SM_TIME_DEFAULT,
2344                                         "memdump.out", SM_IO_APPEND, NULL);
2345                        if (out != NULL)
2346                        {
2347                                (void) sm_io_fprintf(out, SM_TIME_DEFAULT, "----
2348                                sm_heap_report(out,
2349                                               sm_debug_level(&SmHeapCheck) - 1);
2350                                (void) sm_io_close(out, SM_TIME_DEFAULT);
2351                        }
2352                }
2353 #endif /* SM_HEAP_CHECK */

2355                /* let me rest for a second to catch my breath */
2356                if (njobs == 0 && WorkGrp[wgrp].wg_lowqintvl < MIN_SLEEP_TIME)
2357                        sleep(MIN_SLEEP_TIME);
2358                else if (WorkGrp[wgrp].wg_lowqintvl <= 0)
2359                        sleep(QueueIntvl > 0 ? QueueIntvl : MIN_SLEEP_TIME);
2360                else
2361                        sleep(WorkGrp[wgrp].wg_lowqintvl);

2363                /*
2364                **  Get the LA outside the WorkQ loop if necessary.
2365                **  In a persistent queue runner the code is repeated over
2366                **  and over but gatherq() may ignore entries due to
2367                **  shouldqueue() (do we really have to do this twice?).
2368                **  Hence the queue runners would just idle around when once
2369                **  CurrentLA caused all entries in a queue to be ignored.
2370                */
```

```
2372                 if (njobs == 0)
2373                         SM_GET_LA(now);
2374                 rpool = sm_rpool_new_x(NULL);
2375                 e = newenvelope(&QueueEnvelope, CurEnv, rpool);
2376                 e->e_flags = BlankEnvelope.e_flags;
2377                 goto domorework;
2378         }

2380         /* exit without the usual cleanup */
2381         e->e_id = NULL;
2382         if (bitset(RWG_FORK, flags))
2383                 finis(true, true, ExitStat);
2384         /* NOTREACHED */
2385         return true;
2386 }

2388 /*
2389 **  DOQUEUERUN -- do a queue run?
2390 */

2392 bool
2393 doqueuerun()
2394 {
2395         return DoQueueRun;
2396 }

2398 /*
2399 **  RUNQUEUEEVENT -- Sets a flag to indicate that a queue run should be done.
2400 **
2401 **      Parameters:
2402 **              none.
2403 **
2404 **      Returns:
2405 **              none.
2406 **
2407 **      Side Effects:
2408 **              The invocation of this function via an alarm may interrupt
2409 **              a set of actions. Thus errno may be set in that context.
2410 **              We need to restore errno at the end of this function to ensure
2411 **              that any work done here that sets errno doesn't return a
2412 **              misleading/false errno value. Errno may be EINTR upon entry to
2413 **              this function because of non-restartable/continuable system
2414 **              API was active. Iff this is true we will override errno as
2415 **              a timeout (as a more accurate error message).
2416 **
2417 **      NOTE:   THIS CAN BE CALLED FROM A SIGNAL HANDLER.  DO NOT ADD
2418 **              ANYTHING TO THIS ROUTINE UNLESS YOU KNOW WHAT YOU ARE
2419 **              DOING.
2420 */

2422 void
2423 runqueueevent(ignore)
2424         int ignore;
2425 {
2426         int save_errno = errno;

2428         /*
2429         **  Set the general bit that we want a queue run,
2430         **  tested in doqueuerun()
2431         */

2433         DoQueueRun = true;
2434 #if _FFR_QUEUE_SCHED_DBG
2435         if (tTd(69, 10))
2436                 sm_syslog(LOG_INFO, NOQID, "rqe: done");
2437 #endif /* _FFR_QUEUE_SCHED_DBG */
```



```
2439         errno = save_errno;
2440         if (errno == EINTR)
2441                 errno = ETIMEDOUT;
2442 }
2443 /*
2444 **  GATHERQ -- gather messages from the message queue(s) the work queue.
2445 **
2446 **      Parameters:
2447 **              qgrp -- the index of the queue group.
2448 **              qdir -- the index of the queue directory.
2449 **              doall -- if set, include everything in the queue (even
2450 **                      the jobs that cannot be run because the load
2451 **                      average is too high, or MaxQueueRun is reached).
2452 **                      Otherwise, exclude those jobs.
2453 **              full -- (optional) to be set 'true' if WorkList is full
2454 **              more -- (optional) to be set 'true' if there are still more
2455 **                      messages in this queue not added to WorkList
2456 **              pnentries -- (optional) total nuber of entries in queue
2457 **
2458 **      Returns:
2459 **              The number of request in the queue (not necessarily
2460 **              the number of requests in WorkList however).
2461 **
2462 **      Side Effects:
2463 **              prepares available work into WorkList
2464 */

2466 #define NEED_P          0001    /* 'P': priority */
2467 #define NEED_T          0002    /* 'T': time */
2468 #define NEED_R          0004    /* 'R': recipient */
2469 #define NEED_S          0010    /* 'S': sender */
2470 #define NEED_H          0020    /* host */
2471 #define HAS_QUARANTINE  0040    /* has an unexpected 'q' line */
2472 #define NEED_QUARANTINE 0100    /* 'q': reason */

2474 static WORK     *WorkList = NULL;       /* list of unsort work */
2475 static int      WorkListSize = 0;       /* current max size of WorkList */
2476 static int      WorkListCount = 0;      /* # of work items in WorkList */

2478 static int
2479 gatherq(qgrp, qdir, doall, full, more, pnentries)
2480         int qgrp;
2481         int qdir;
2482         bool doall;
2483         bool *full;
2484         bool *more;
2485         int *pnentries;
2486 {
2487         register struct dirent *d;
2488         register WORK *w;
2489         register char *p;
2490         DIR *f;
2491         int i, num_ent, wn, nentries;
2492         QUEUE_CHAR *check;
2493         char qd[MAXPATHLEN];
2494         char qf[MAXPATHLEN];

2496         wn = WorkListCount - 1;
2497         num_ent = 0;
2498         nentries = 0;
2499         if (qdir == NOQDIR)
2500                 (void) sm_strlcpy(qd, ".", sizeof(qd));
2501         else
2502                 (void) sm_strlcpyn(qd, sizeof(qd), 2,
2503                         Queue[qgrp]->qg_qpaths[qdir].qp_name,
```

```
2504                        (bitset(QP_SUBQF,
2505                                Queue[qgrp]->qg_qpaths[qdir].qp_subdirs)
2506                                        ? "/qf" : ""));

2508        if (tTd(41, 1))
2509        {
2510                sm_dprintf("gatherq:\n");

2512                check = QueueLimitId;
2513                while (check != NULL)
2514                {
2515                        sm_dprintf("\tQueueLimitId = %s%s\n",
2516                                check->queue_negate ? "!" : "",
2517                                check->queue_match);
2518                        check = check->queue_next;
2519                }

2521                check = QueueLimitSender;
2522                while (check != NULL)
2523                {
2524                        sm_dprintf("\tQueueLimitSender = %s%s\n",
2525                                check->queue_negate ? "!" : "",
2526                                check->queue_match);
2527                        check = check->queue_next;
2528                }

2530                check = QueueLimitRecipient;
2531                while (check != NULL)
2532                {
2533                        sm_dprintf("\tQueueLimitRecipient = %s%s\n",
2534                                check->queue_negate ? "!" : "",
2535                                check->queue_match);
2536                        check = check->queue_next;
2537                }

2539                if (QueueMode == QM_QUARANTINE)
2540                {
2541                        check = QueueLimitQuarantine;
2542                        while (check != NULL)
2543                        {
2544                                sm_dprintf("\tQueueLimitQuarantine = %s%s\n",
2545                                           check->queue_negate ? "!" : "",
2546                                           check->queue_match);
2547                                check = check->queue_next;
2548                        }
2549                }
2550        }

2552        /* open the queue directory */
2553        f = opendir(qd);
2554        if (f == NULL)
2555        {
2556                syserr("gatherq: cannot open \"%s\"",
2557                        qid_printqueue(qgrp, qdir));
2558                if (full != NULL)
2559                        *full = WorkListCount >= MaxQueueRun && MaxQueueRun > 0;
2560                if (more != NULL)
2561                        *more = false;
2562                return 0;
2563        }

2565        /*
2566        **  Read the work directory.
2567        */

2569        while ((d = readdir(f)) != NULL)
```



```
2570        {
2571                SM_FILE_T *cf;
2572                int qfver = 0;
2573                char lbuf[MAXNAME + 1];
2574                struct stat sbuf;

2576                if (tTd(41, 50))
2577                        sm_dprintf("gatherq: checking %s..", d->d_name);

2579                /* is this an interesting entry? */
2580                if (!(((QueueMode == QM_NORMAL &&
2581                        d->d_name[0] == NORMQF_LETTER) ||
2582                       (QueueMode == QM_QUARANTINE &&
2583                        d->d_name[0] == QUARQF_LETTER) ||
2584                       (QueueMode == QM_LOST &&
2585                        d->d_name[0] == LOSEQF_LETTER)) &&
2586                      d->d_name[1] == 'f'))
2587                {
2588                        if (tTd(41, 50))
2589                                sm_dprintf("  skipping\n");
2590                        continue;
2591                }
2592                if (tTd(41, 50))
2593                        sm_dprintf("\n");

2595                if (strlen(d->d_name) >= MAXQFNAME)
2596                {
2597                        if (Verbose)
2598                                (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
2599                                                     "gatherq: %s too long, %d m
2600                                                     d->d_name, MAXQFNAME);
2601                        if (LogLevel > 0)
2602                                sm_syslog(LOG_ALERT, NOQID,
2603                                          "gatherq: %s too long, %d max characte
2604                                          d->d_name, MAXQFNAME);
2605                        continue;
2606                }

2608                ++nentries;
2609                check = QueueLimitId;
2610                while (check != NULL)
2611                {
2612                        if (strcontainedin(false, check->queue_match,
2613                                           d->d_name) != check->queue_negate)
2614                                break;
2615                        else
2616                                check = check->queue_next;
2617                }
2618                if (QueueLimitId != NULL && check == NULL)
2619                        continue;

2621                /* grow work list if necessary */
2622                if (++wn >= MaxQueueRun && MaxQueueRun > 0)
2623                {
2624                        if (wn == MaxQueueRun && LogLevel > 0)
2625                                sm_syslog(LOG_WARNING, NOQID,
2626                                          "WorkList for %s maxed out at %d",
2627                                          qid_printqueue(qgrp, qdir),
2628                                          MaxQueueRun);
2629                        if (doall)
2630                                continue;       /* just count entries */
2631                        break;
2632                }
2633                if (wn >= WorkListSize)
2634                {
2635                        grow_wlist(qgrp, qdir);
```

```
2636                    if (wn >= WorkListSize)
2637                            continue;
2638            }
2639            SM_ASSERT(wn >= 0);
2640            w = &WorkList[wn];

2642            (void) sm_strlcpyn(qf, sizeof(qf), 3, qd, "/", d->d_name);
2643            if (stat(qf, &sbuf) < 0)
2644            {
2645                    if (errno != ENOENT)
2646                            sm_syslog(LOG_INFO, NOQID,
2647                                      "gatherq: can't stat %s/%s",
2648                                      qid_printqueue(qgrp, qdir),
2649                                      d->d_name);
2650                    wn--;
2651                    continue;
2652            }
2653            if (!bitset(S_IFREG, sbuf.st_mode))
2654            {
2655                    /* Yikes!  Skip it or we will hang on open! */
2656                    if (!((d->d_name[0] == DATAFL_LETTER ||
2657                           d->d_name[0] == NORMQF_LETTER ||
2658                           d->d_name[0] == QUARQF_LETTER ||
2659                           d->d_name[0] == LOSEQF_LETTER ||
2660                           d->d_name[0] == XSCRPT_LETTER) &&
2661                          d->d_name[1] == 'f' && d->d_name[2] == '\0'))
2662                            syserr("gatherq: %s/%s is not a regular file",
2663                                   qid_printqueue(qgrp, qdir), d->d_name);
2664                    wn--;
2665                    continue;
2666            }

2668            /* avoid work if possible */
2669            if ((QueueSortOrder == QSO_BYFILENAME ||
2670                 QueueSortOrder == QSO_BYMODTIME ||
2671                 QueueSortOrder == QSO_NONE ||
2672                 QueueSortOrder == QSO_RANDOM) &&
2673                QueueLimitQuarantine == NULL &&
2674                QueueLimitSender == NULL &&
2675                QueueLimitRecipient == NULL)
2676            {
2677                    w->w_qgrp = qgrp;
2678                    w->w_qdir = qdir;
2679                    w->w_name = newstr(d->d_name);
2680                    w->w_host = NULL;
2681                    w->w_lock = w->w_tooyoung = false;
2682                    w->w_pri = 0;
2683                    w->w_ctime = 0;
2684                    w->w_mtime = sbuf.st_mtime;
2685                    ++num_ent;
2686                    continue;
2687            }

2689            /* open control file */
2690            cf = sm_io_open(SmFtStdio, SM_TIME_DEFAULT, qf, SM_IO_RDONLY_B,
2691                            NULL);
2692            if (cf == NULL && OpMode != MD_PRINT)
2693            {
2694                    /* this may be some random person sending hir msgs */
2695                    if (tTd(41, 2))
2696                            sm_dprintf("gatherq: cannot open %s: %s\n",
2697                                       d->d_name, sm_errstring(errno));
2698                    errno = 0;
2699                    wn--;
2700                    continue;
2701            }
```



```
2702            w->w_qgrp = qgrp;
2703            w->w_qdir = qdir;
2704            w->w_name = newstr(d->d_name);
2705            w->w_host = NULL;
2706            if (cf != NULL)
2707            {
2708                    w->w_lock = !lockfile(sm_io_getinfo(cf, SM_IO_WHAT_FD,
2709                                                        NULL),
2710                                          w->w_name, NULL,
2711                                          LOCK_SH|LOCK_NB);
2712            }
2713            w->w_tooyoung = false;

2715            /* make sure jobs in creation don't clog queue */
2716            w->w_pri = 0x7fffffff;
2717            w->w_ctime = 0;
2718            w->w_mtime = sbuf.st_mtime;

2720            /* extract useful information */
2721            i = NEED_P|NEED_T;
2722            if (QueueSortOrder == QSO_BYHOST
2723 #if _FFR_RHS
2724                || QueueSortOrder == QSO_BYSHUFFLE
2725 #endif /* _FFR_RHS */
2726                )
2727            {
2728                    /* need w_host set for host sort order */
2729                    i |= NEED_H;
2730            }
2731            if (QueueLimitSender != NULL)
2732                    i |= NEED_S;
2733            if (QueueLimitRecipient != NULL)
2734                    i |= NEED_R;
2735            if (QueueLimitQuarantine != NULL)
2736                    i |= NEED_QUARANTINE;
2737            while (cf != NULL && i != 0 &&
2738                   sm_io_fgets(cf, SM_TIME_DEFAULT, lbuf,
2739                               sizeof(lbuf)) != NULL)
2740            {
2741                    int c;
2742                    time_t age;

2744                    p = strchr(lbuf, '\n');
2745                    if (p != NULL)
2746                            *p = '\0';
2747                    else
2748                    {
2749                            /* flush rest of overly long line */
2750                            while ((c = sm_io_getc(cf, SM_TIME_DEFAULT))
2751                                   != SM_IO_EOF && c != '\n')
2752                                    continue;
2753                    }

2755                    switch (lbuf[0])
2756                    {
2757                      case 'V':
2758                            qfver = atoi(&lbuf[1]);
2759                            break;

2761                      case 'P':
2762                            w->w_pri = atol(&lbuf[1]);
2763                            i &= ~NEED_P;
2764                            break;

2766                      case 'T':
2767                            w->w_ctime = atol(&lbuf[1]);
```

```
2768                                    i &= ~NEED_T;
2769                                    break;

2771                            case 'q':
2772                                    if (QueueMode != QM_QUARANTINE &&
2773                                        QueueMode != QM_LOST)
2774                                    {
2775                                            if (tTd(41, 49))
2776                                                    sm_dprintf("%s not marked as qua
2777                                                               w->w_name);
2778                                            i |= HAS_QUARANTINE;
2779                                    }
2780                                    else if (QueueMode == QM_QUARANTINE)
2781                                    {
2782                                            if (QueueLimitQuarantine == NULL)
2783                                            {
2784                                                    i &= ~NEED_QUARANTINE;
2785                                                    break;
2786                                            }
2787                                            p = &lbuf[1];
2788                                            check = QueueLimitQuarantine;
2789                                            while (check != NULL)
2790                                            {
2791                                                    if (strcontainedin(false,
2792                                                                       check->queue_
2793                                                                       p) !=
2794                                                        check->queue_negate)
2795                                                            break;
2796                                                    else
2797                                                            check = check->queue_nex
2798                                            }
2799                                            if (check != NULL)
2800                                                    i &= ~NEED_QUARANTINE;
2801                                    }
2802                                    break;

2804                            case 'R':
2805                                    if (w->w_host == NULL &&
2806                                        (p = strrchr(&lbuf[1], '@')) != NULL)
2807                                    {
2808 #if _FFR_RHS
2809                                            if (QueueSortOrder == QSO_BYSHUFFLE)
2810                                                    w->w_host = newstr(&p[1]);
2811                                            else
2812 #endif /* _FFR_RHS */
2813                                                    w->w_host = strrev(&p[1]);
2814                                            makelower(w->w_host);
2815                                            i &= ~NEED_H;
2816                                    }
2817                                    if (QueueLimitRecipient == NULL)
2818                                    {
2819                                            i &= ~NEED_R;
2820                                            break;
2821                                    }
2822                                    if (qfver > 0)
2823                                    {
2824                                            p = strchr(&lbuf[1], ':');
2825                                            if (p == NULL)
2826                                                    p = &lbuf[1];
2827                                            else
2828                                                    ++p; /* skip over ':' */
2829                                    }
2830                                    else
2831                                            p = &lbuf[1];
2832                                    check = QueueLimitRecipient;
2833                                    while (check != NULL)
```



```
2834                                    {
2835                                            if (strcontainedin(true,
2836                                                               check->queue_match,
2837                                                               p) !=
2838                                                check->queue_negate)
2839                                                    break;
2840                                            else
2841                                                    check = check->queue_next;
2842                                    }
2843                                    if (check != NULL)
2844                                            i &= ~NEED_R;
2845                                    break;

2847                            case 'S':
2848                                    check = QueueLimitSender;
2849                                    while (check != NULL)
2850                                    {
2851                                            if (strcontainedin(true,
2852                                                               check->queue_match,
2853                                                               &lbuf[1]) !=
2854                                                check->queue_negate)
2855                                                    break;
2856                                            else
2857                                                    check = check->queue_next;
2858                                    }
2859                                    if (check != NULL)
2860                                            i &= ~NEED_S;
2861                                    break;

2863                            case 'K':
2864 #if _FFR_EXPDELAY
2865                                    if (MaxQueueAge > 0)
2866                                    {
2867                                            time_t lasttry, delay;

2869                                            lasttry = (time_t) atol(&lbuf[1]);
2870                                            delay = MIN(lasttry - w->w_ctime,
2871                                                        MaxQueueAge);
2872                                            age = curtime() - lasttry;
2873                                            if (age < delay)
2874                                                    w->w_tooyoung = true;
2875                                            break;
2876                                    }
2877 #endif /* _FFR_EXPDELAY */

2879                                    age = curtime() - (time_t) atol(&lbuf[1]);
2880                                    if (age >= 0 && MinQueueAge > 0 &&
2881                                        age < MinQueueAge)
2882                                            w->w_tooyoung = true;
2883                                    break;

2885                            case 'N':
2886                                    if (atol(&lbuf[1]) == 0)
2887                                            w->w_tooyoung = false;
2888                                    break;
2889                            }
2890                    }
2891                    if (cf != NULL)
2892                            (void) sm_io_close(cf, SM_TIME_DEFAULT);

2894                    if ((!doall && (shouldqueue(w->w_pri, w->w_ctime) ||
2895                        w->w_tooyoung)) ||
2896                        bitset(HAS_QUARANTINE, i) ||
2897                        bitset(NEED_QUARANTINE, i) ||
2898                        bitset(NEED_R|NEED_S, i))
2899                    {
```

```
2900                        /* don't even bother sorting this job in */
2901                        if (tTd(41, 49))
2902                                sm_dprintf("skipping %s (%x)\n", w->w_name, i);
2903                        sm_free(w->w_name); /* XXX */
2904                        if (w->w_host != NULL)
2905                                sm_free(w->w_host); /* XXX */
2906                        wn--;
2907                }
2908                else
2909                        ++num_ent;
2910        }
2911        (void) closedir(f);
2912        wn++;

2914        i = wn - WorkListCount;
2915        WorkListCount += SM_MIN(num_ent, WorkListSize);

2917        if (more != NULL)
2918                *more = WorkListCount < wn;

2920        if (full != NULL)
2921                *full = (wn >= MaxQueueRun && MaxQueueRun > 0) ||
2922                        (WorkList == NULL && wn > 0);

2924        if (pnentries != NULL)
2925                *pnentries = nentries;
2926        return i;
2927 }
2928 /*
2929 **  SORTQ -- sort the work list
2930 **
2931 **      First the old WorkQ is cleared away. Then the WorkList is sorted
2932 **      for all items so that important (higher sorting value) items are not
2933 **      trunctated off. Then the most important items are moved from
2934 **      WorkList to WorkQ. The lower count of 'max' or MaxListCount items
2935 **      are moved.
2936 **
2937 **      Parameters:
2938 **              max -- maximum number of items to be placed in WorkQ
2939 **
2940 **      Returns:
2941 **              the number of items in WorkQ
2942 **
2943 **      Side Effects:
2944 **              WorkQ gets released and filled with new work. WorkList
2945 **              gets released. Work items get sorted in order.
2946 */

2948 static int
2949 sortq(max)
2950        int max;
2951 {
2952        register int i;                 /* local counter */
2953        register WORK *w;               /* tmp item pointer */
2954        int wc = WorkListCount;         /* trim size for WorkQ */

2956        if (WorkQ != NULL)
2957        {
2958                WORK *nw;

2960                /* Clear out old WorkQ. */
2961                for (w = WorkQ; w != NULL; w = nw)
2962                {
2963                        nw = w->w_next;
2964                        sm_free(w->w_name); /* XXX */
2965                        if (w->w_host != NULL)
```



```
2966                                sm_free(w->w_host); /* XXX */
2967                        sm_free((char *) w); /* XXX */
2968                }
2969                WorkQ = NULL;
2970        }

2972        if (WorkList == NULL || wc <= 0)
2973                return 0;

2975        /*
2976        **  The sort now takes place using all of the items in WorkList.
2977        **  The list gets trimmed to the most important items after the sort.
2978        **  If the trim were to happen before the sort then one or more
2979        **  important items might get truncated off -- not what we want.
2980        */

2982        if (QueueSortOrder == QSO_BYHOST)
2983        {
2984                /*
2985                **  Sort the work directory for the first time,
2986                **  based on host name, lock status, and priority.
2987                */

2989                qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf1);

2991                /*
2992                **  If one message to host is locked, "lock" all messages
2993                **  to that host.
2994                */

2996                i = 0;
2997                while (i < wc)
2998                {
2999                        if (!WorkList[i].w_lock)
3000                        {
3001                                i++;
3002                                continue;
3003                        }
3004                        w = &WorkList[i];
3005                        while (++i < wc)
3006                        {
3007                                if (WorkList[i].w_host == NULL &&
3008                                    w->w_host == NULL)
3009                                        WorkList[i].w_lock = true;
3010                                else if (WorkList[i].w_host != NULL &&
3011                                         w->w_host != NULL &&
3012                                         sm_strcasecmp(WorkList[i].w_host,
3013                                                       w->w_host) == 0)
3014                                        WorkList[i].w_lock = true;
3015                                else
3016                                        break;
3017                        }
3018                }

3020                /*
3021                **  Sort the work directory for the second time,
3022                **  based on lock status, host name, and priority.
3023                */

3025                qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf2);
3026        }
3027        else if (QueueSortOrder == QSO_BYTIME)
3028        {
3029                /*
3030                **  Simple sort based on submission time only.
3031                */
```

```
3033                    qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf3);
3034            }
3035            else if (QueueSortOrder == QSO_BYFILENAME)
3036            {
3037                    /*
3038                    **  Sort based on queue filename.
3039                    */

3041                    qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf4);
3042            }
3043            else if (QueueSortOrder == QSO_RANDOM)
3044            {
3045                    /*
3046                    **  Sort randomly.  To avoid problems with an instable sort,
3047                    **  use a random index into the queue file name to start
3048                    **  comparison.
3049                    */

3051                    randi = get_rand_mod(MAXQFNAME);
3052                    if (randi < 2)
3053                            randi = 3;
3054                    qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf5);
3055            }
3056            else if (QueueSortOrder == QSO_BYMODTIME)
3057            {
3058                    /*
3059                    **  Simple sort based on modification time of queue file.
3060                    **  This puts the oldest items first.
3061                    */

3063                    qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf6);
3064            }
3065 #if _FFR_RHS
3066            else if (QueueSortOrder == QSO_BYSHUFFLE)
3067            {
3068                    /*
3069                    **  Simple sort based on shuffled host name.
3070                    */

3072                    init_shuffle_alphabet();
3073                    qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf7);
3074            }
3075 #endif /* _FFR_RHS */
3076            else if (QueueSortOrder == QSO_BYPRIORITY)
3077            {
3078                    /*
3079                    **  Simple sort based on queue priority only.
3080                    */

3082                    qsort((char *) WorkList, wc, sizeof(*WorkList), workcmpf0);
3083            }
3084            /* else don't sort at all */

3086            /* Check if the per queue group item limit will be exceeded */
3087            if (wc > max && max > 0)
3088                    wc = max;

3090            /*
3091            **  Convert the work list into canonical form.
3092            **      Should be turning it into a list of envelopes here perhaps.
3093            **  Only take the most important items up to the per queue group
3094            **  maximum.
3095            */

3097            for (i = wc; --i >= 0; )
```



```
3098            {
3099                    w = (WORK *) xalloc(sizeof(*w));
3100                    w->w_qgrp = WorkList[i].w_qgrp;
3101                    w->w_qdir = WorkList[i].w_qdir;
3102                    w->w_name = WorkList[i].w_name;
3103                    w->w_host = WorkList[i].w_host;
3104                    w->w_lock = WorkList[i].w_lock;
3105                    w->w_tooyoung = WorkList[i].w_tooyoung;
3106                    w->w_pri = WorkList[i].w_pri;
3107                    w->w_ctime = WorkList[i].w_ctime;
3108                    w->w_mtime = WorkList[i].w_mtime;
3109                    w->w_next = WorkQ;
3110                    WorkQ = w;
3111            }

3113            /* free the rest of the list */
3114            for (i = WorkListCount; --i >= wc; )
3115            {
3116                    sm_free(WorkList[i].w_name);
3117                    if (WorkList[i].w_host != NULL)
3118                            sm_free(WorkList[i].w_host);
3119            }

3121            if (WorkList != NULL)
3122                    sm_free(WorkList); /* XXX */
3123            WorkList = NULL;
3124            WorkListSize = 0;
3125            WorkListCount = 0;

3127            if (tTd(40, 1))
3128            {
3129                    for (w = WorkQ; w != NULL; w = w->w_next)
3130                    {
3131                            if (w->w_host != NULL)
3132                                    sm_dprintf("%22s: pri=%ld %s\n",
3133                                            w->w_name, w->w_pri, w->w_host);
3134                            else
3135                                    sm_dprintf("%32s: pri=%ld\n",
3136                                            w->w_name, w->w_pri);
3137                    }
3138            }

3140            return wc; /* return number of WorkQ items */
3141 }
3142 /*
3143 **  GROW_WLIST -- make the work list larger
3144 **
3145 **      Parameters:
3146 **              qgrp -- the index for the queue group.
3147 **              qdir -- the index for the queue directory.
3148 **
3149 **      Returns:
3150 **              none.
3151 **
3152 **      Side Effects:
3153 **              Adds another QUEUESEGSIZE entries to WorkList if possible.
3154 **              It can fail if there isn't enough memory, so WorkListSize
3155 **              should be checked again upon return.
3156 */

3158 static void
3159 grow_wlist(qgrp, qdir)
3160         int qgrp;
3161         int qdir;
3162 {
3163         if (tTd(41, 1))
```

```
3164                 sm_dprintf("grow_wlist: WorkListSize=%d\n", WorkListSize);
3165         if (WorkList == NULL)
3166         {
3167                 WorkList = (WORK *) xalloc((sizeof(*WorkList)) *
3168                                            (QUEUESEGSIZE + 1));
3169                 WorkListSize = QUEUESEGSIZE;
3170         }
3171         else
3172         {
3173                 int newsize = WorkListSize + QUEUESEGSIZE;
3174                 WORK *newlist = (WORK *) sm_realloc((char *) WorkList,
3175                                            (unsigned) sizeof(WORK) * (newsize + 1

3177                 if (newlist != NULL)
3178                 {
3179                         WorkListSize = newsize;
3180                         WorkList = newlist;
3181                         if (LogLevel > 1)
3182                         {
3183                                 sm_syslog(LOG_INFO, NOQID,
3184                                           "grew WorkList for %s to %d",
3185                                           qid_printqueue(qgrp, qdir),
3186                                           WorkListSize);
3187                         }
3188                 }
3189                 else if (LogLevel > 0)
3190                 {
3191                         sm_syslog(LOG_ALERT, NOQID,
3192                                   "FAILED to grow WorkList for %s to %d",
3193                                   qid_printqueue(qgrp, qdir), newsize);
3194                 }
3195         }
3196         if (tTd(41, 1))
3197                 sm_dprintf("grow_wlist: WorkListSize now %d\n", WorkListSize);
3198 }
3199 /*
3200 **  WORKCMPF0 -- simple priority-only compare function.
3201 **
3202 **      Parameters:
3203 **              a -- the first argument.
3204 **              b -- the second argument.
3205 **
3206 **      Returns:
3207 **              -1 if a < b
3208 **               0 if a == b
3209 **              +1 if a > b
3210 **
3211 */

3213 static int
3214 workcmpf0(a, b)
3215         register WORK *a;
3216         register WORK *b;
3217 {
3218         long pa = a->w_pri;
3219         long pb = b->w_pri;

3221         if (pa == pb)
3222                 return 0;
3223         else if (pa > pb)
3224                 return 1;
3225         else
3226                 return -1;
3227 }
3228 /*
3229 **  WORKCMPF1 -- first compare function for ordering work based on host name.
```



```
3230 **
3231 **      Sorts on host name, lock status, and priority in that order.
3232 **
3233 **      Parameters:
3234 **              a -- the first argument.
3235 **              b -- the second argument.
3236 **
3237 **      Returns:
3238 **              <0 if a < b
3239 **               0 if a == b
3240 **              >0 if a > b
3241 **
3242 */

3244 static int
3245 workcmpf1(a, b)
3246         register WORK *a;
3247         register WORK *b;
3248 {
3249         int i;

3251         /* host name */
3252         if (a->w_host != NULL && b->w_host == NULL)
3253                 return 1;
3254         else if (a->w_host == NULL && b->w_host != NULL)
3255                 return -1;
3256         if (a->w_host != NULL && b->w_host != NULL &&
3257             (i = sm_strcasecmp(a->w_host, b->w_host)) != 0)
3258                 return i;

3260         /* lock status */
3261         if (a->w_lock != b->w_lock)
3262                 return b->w_lock - a->w_lock;

3264         /* job priority */
3265         return workcmpf0(a, b);
3266 }
3267 /*
3268 **  WORKCMPF2 -- second compare function for ordering work based on host name.
3269 **
3270 **      Sorts on lock status, host name, and priority in that order.
3271 **
3272 **      Parameters:
3273 **              a -- the first argument.
3274 **              b -- the second argument.
3275 **
3276 **      Returns:
3277 **              <0 if a < b
3278 **               0 if a == b
3279 **              >0 if a > b
3280 **
3281 */

3283 static int
3284 workcmpf2(a, b)
3285         register WORK *a;
3286         register WORK *b;
3287 {
3288         int i;

3290         /* lock status */
3291         if (a->w_lock != b->w_lock)
3292                 return a->w_lock - b->w_lock;

3294         /* host name */
3295         if (a->w_host != NULL && b->w_host == NULL)
```

```
3296                return 1;
3297        else if (a->w_host == NULL && b->w_host != NULL)
3298                return -1;
3299        if (a->w_host != NULL && b->w_host != NULL &&
3300            (i = sm_strcasecmp(a->w_host, b->w_host)) != 0)
3301                return i;

3303        /* job priority */
3304        return workcmpf0(a, b);
3305 }
3306 /*
3307 **  WORKCMPF3 -- simple submission-time-only compare function.
3308 **
3309 **      Parameters:
3310 **              a -- the first argument.
3311 **              b -- the second argument.
3312 **
3313 **      Returns:
3314 **              -1 if a < b
3315 **               0 if a == b
3316 **              +1 if a > b
3317 **
3318 */

3320 static int
3321 workcmpf3(a, b)
3322         register WORK *a;
3323         register WORK *b;
3324 {
3325         if (a->w_ctime > b->w_ctime)
3326                 return 1;
3327         else if (a->w_ctime < b->w_ctime)
3328                 return -1;
3329         else
3330                 return 0;
3331 }
3332 /*
3333 **  WORKCMPF4 -- compare based on file name
3334 **
3335 **      Parameters:
3336 **              a -- the first argument.
3337 **              b -- the second argument.
3338 **
3339 **      Returns:
3340 **              -1 if a < b
3341 **               0 if a == b
3342 **              +1 if a > b
3343 **
3344 */

3346 static int
3347 workcmpf4(a, b)
3348         register WORK *a;
3349         register WORK *b;
3350 {
3351         return strcmp(a->w_name, b->w_name);
3352 }
3353 /*
3354 **  WORKCMPF5 -- compare based on assigned random number
3355 **
3356 **      Parameters:
3357 **              a -- the first argument.
3358 **              b -- the second argument.
3359 **
3360 **      Returns:
3361 **              randomly 1/-1
```



```
3362 */

3364 /* ARGSUSED0 */
3365 static int
3366 workcmpf5(a, b)
3367         register WORK *a;
3368         register WORK *b;
3369 {
3370         if (strlen(a->w_name) < randi || strlen(b->w_name) < randi)
3371                 return -1;
3372         return a->w_name[randi] - b->w_name[randi];
3373 }
3374 /*
3375 **  WORKCMPF6 -- simple modification-time-only compare function.
3376 **
3377 **      Parameters:
3378 **              a -- the first argument.
3379 **              b -- the second argument.
3380 **
3381 **      Returns:
3382 **              -1 if a < b
3383 **               0 if a == b
3384 **              +1 if a > b
3385 **
3386 */

3388 static int
3389 workcmpf6(a, b)
3390         register WORK *a;
3391         register WORK *b;
3392 {
3393         if (a->w_mtime > b->w_mtime)
3394                 return 1;
3395         else if (a->w_mtime < b->w_mtime)
3396                 return -1;
3397         else
3398                 return 0;
3399 }
3400 #if _FFR_RHS
3401 /*
3402 **  WORKCMPF7 -- compare function for ordering work based on shuffled host name.
3403 **
3404 **      Sorts on lock status, host name, and priority in that order.
3405 **
3406 **      Parameters:
3407 **              a -- the first argument.
3408 **              b -- the second argument.
3409 **
3410 **      Returns:
3411 **              <0 if a < b
3412 **               0 if a == b
3413 **              >0 if a > b
3414 **
3415 */

3417 static int
3418 workcmpf7(a, b)
3419         register WORK *a;
3420         register WORK *b;
3421 {
3422         int i;

3424         /* lock status */
3425         if (a->w_lock != b->w_lock)
3426                 return a->w_lock - b->w_lock;
```

```
3428         /* host name */
3429         if (a->w_host != NULL && b->w_host == NULL)
3430                 return 1;
3431         else if (a->w_host == NULL && b->w_host != NULL)
3432                 return -1;
3433         if (a->w_host != NULL && b->w_host != NULL &&
3434             (i = sm_strshufflecmp(a->w_host, b->w_host)) != 0)
3435                 return i;

3437         /* job priority */
3438         return workcmpf0(a, b);
3439 }
3440 #endif /* _FFR_RHS */
3441 /*
3442 **  STRREV -- reverse string
3443 **
3444 **      Returns a pointer to a new string that is the reverse of
3445 **      the string pointed to by fwd.  The space for the new
3446 **      string is obtained using xalloc().
3447 **
3448 **      Parameters:
3449 **              fwd -- the string to reverse.
3450 **
3451 **      Returns:
3452 **              the reversed string.
3453 */

3455 static char *
3456 strrev(fwd)
3457         char *fwd;
3458 {
3459         char *rev = NULL;
3460         int len, cnt;

3462         len = strlen(fwd);
3463         rev = xalloc(len + 1);
3464         for (cnt = 0; cnt < len; ++cnt)
3465                 rev[cnt] = fwd[len - cnt - 1];
3466         rev[len] = '\0';
3467         return rev;
3468 }

3470 #if _FFR_RHS

3472 # define NASCII 128
3473 # define NCHAR  256

3475 static unsigned char ShuffledAlphabet[NCHAR];

3477 void
3478 init_shuffle_alphabet()
3479 {
3480         static bool init = false;
3481         int i;

3483         if (init)
3484                 return;

3486         /* fill the ShuffledAlphabet */
3487         for (i = 0; i < NASCII; i++)
3488                 ShuffledAlphabet[i] = i;

3490         /* mix it */
3491         for (i = 1; i < NASCII; i++)
3492         {
3493                 register int j = get_random() % NASCII;
```



```
3494                 register int tmp;

3496                 tmp = ShuffledAlphabet[j];
3497                 ShuffledAlphabet[j] = ShuffledAlphabet[i];
3498                 ShuffledAlphabet[i] = tmp;
3499         }

3501         /* make it case insensitive */
3502         for (i = 'A'; i <= 'Z'; i++)
3503                 ShuffledAlphabet[i] = ShuffledAlphabet[i + 'a' - 'A'];

3505         /* fill the upper part */
3506         for (i = 0; i < NASCII; i++)
3507                 ShuffledAlphabet[i + NASCII] = ShuffledAlphabet[i];
3508         init = true;
3509 }

3511 static int
3512 sm_strshufflecmp(a, b)
3513         char *a;
3514         char *b;
3515 {
3516         const unsigned char *us1 = (const unsigned char *) a;
3517         const unsigned char *us2 = (const unsigned char *) b;

3519         while (ShuffledAlphabet[*us1] == ShuffledAlphabet[*us2++])
3520         {
3521                 if (*us1++ == '\0')
3522                         return 0;
3523         }
3524         return (ShuffledAlphabet[*us1] - ShuffledAlphabet[*--us2]);
3525 }
3526 #endif /* _FFR_RHS */

3528 /*
3529 **  DOWORK -- do a work request.
3530 **
3531 **      Parameters:
3532 **              qgrp -- the index of the queue group for the job.
3533 **              qdir -- the index of the queue directory for the job.
3534 **              id -- the ID of the job to run.
3535 **              forkflag -- if set, run this in background.
3536 **              requeueflag -- if set, reinstantiate the queue quickly.
3537 **                      This is used when expanding aliases in the queue.
3538 **                      If forkflag is also set, it doesn't wait for the
3539 **                      child.
3540 **              e - the envelope in which to run it.
3541 **
3542 **      Returns:
3543 **              process id of process that is running the queue job.
3544 **
3545 **      Side Effects:
3546 **              The work request is satisfied if possible.
3547 */

3549 pid_t
3550 dowork(qgrp, qdir, id, forkflag, requeueflag, e)
3551         int qgrp;
3552         int qdir;
3553         char *id;
3554         bool forkflag;
3555         bool requeueflag;
3556         register ENVELOPE *e;
3557 {
3558         register pid_t pid;
3559         SM_RPOOL_T *rpool;
```

```
3561        if (tTd(40, 1))
3562                sm_dprintf("dowork(%s/%s)\n", qid_printqueue(qgrp, qdir), id);
3563
3564        /*
3565        **  Fork for work.
3566        */
3567
3568        if (forkflag)
3569        {
3570                /*
3571                **  Since the delivery may happen in a child and the
3572                **  parent does not wait, the parent may close the
3573                **  maps thereby removing any shared memory used by
3574                **  the map.  Therefore, close the maps now so the
3575                **  child will dynamically open them if necessary.
3576                */
3577
3578                closemaps(false);
3579
3580                pid = fork();
3581                if (pid < 0)
3582                {
3583                        syserr("dowork: cannot fork");
3584                        return 0;
3585                }
3586                else if (pid > 0)
3587                {
3588                        /* parent -- clean out connection cache */
3589                        mci_flush(false, NULL);
3590                }
3591                else
3592                {
3593                        /*
3594                        **  Initialize exception stack and default exception
3595                        **  handler for child process.
3596                        */
3597
3598                        /* Reset global flags */
3599                        RestartRequest = NULL;
3600                        RestartWorkGroup = false;
3601                        ShutdownRequest = NULL;
3602                        PendingSignal = 0;
3603                        CurrentPid = getpid();
3604                        sm_exc_newthread(fatal_error);
3605
3606                        /*
3607                        **  See note above about SMTP processes and SIGCHLD.
3608                        */
3609
3610                        if (OpMode == MD_SMTP ||
3611                            OpMode == MD_DAEMON ||
3612                            MaxQueueChildren > 0)
3613                        {
3614                                proc_list_clear();
3615                                sm_releasesignal(SIGCHLD);
3616                                (void) sm_signal(SIGCHLD, SIG_DFL);
3617                        }
3618
3619                        /* child -- error messages to the transcript */
3620                        QuickAbort = OnlyOneError = false;
3621                }
3622        }
3623        else
3624        {
3625                pid = 0;
```



```
3626        }
3627
3628        if (pid == 0)
3629        {
3630                /*
3631                **  CHILD
3632                **      Lock the control file to avoid duplicate deliveries.
3633                **              Then run the file as though we had just read it.
3634                **      We save an idea of the temporary name so we
3635                **              can recover on interrupt.
3636                */
3637
3638                if (forkflag)
3639                {
3640                        /* Reset global flags */
3641                        RestartRequest = NULL;
3642                        RestartWorkGroup = false;
3643                        ShutdownRequest = NULL;
3644                        PendingSignal = 0;
3645                }
3646
3647                /* set basic modes, etc. */
3648                sm_clear_events();
3649                clearstats();
3650                rpool = sm_rpool_new_x(NULL);
3651                clearenvelope(e, false, rpool);
3652                e->e_flags |= EF_QUEUERUN|EF_GLOBALERRS;
3653                set_delivery_mode(SM_DELIVER, e);
3654                e->e_errormode = EM_MAIL;
3655                e->e_id = id;
3656                e->e_qgrp = qgrp;
3657                e->e_qdir = qdir;
3658                GrabTo = UseErrorsTo = false;
3659                ExitStat = EX_OK;
3660                if (forkflag)
3661                {
3662                        disconnect(1, e);
3663                        set_op_mode(MD_QUEUERUN);
3664                }
3665                sm_setproctitle(true, e, "%s from queue", qid_printname(e));
3666                if (LogLevel > 76)
3667                        sm_syslog(LOG_DEBUG, e->e_id, "dowork, pid=%d",
3668                                  (int) CurrentPid);
3669
3670                /* don't use the headers from sendmail.cf... */
3671                e->e_header = NULL;
3672
3673                /* read the queue control file -- return if locked */
3674                if (!readqf(e, false))
3675                {
3676                        if (tTd(40, 4) && e->e_id != NULL)
3677                                sm_dprintf("readqf(%s) failed\n",
3678                                           qid_printname(e));
3679                        e->e_id = NULL;
3680                        if (forkflag)
3681                                finis(false, true, EX_OK);
3682                        else
3683                        {
3684                                /* adding this frees 8 bytes */
3685                                clearenvelope(e, false, rpool);
3686
3687                                /* adding this frees 12 bytes */
3688                                sm_rpool_free(rpool);
3689                                e->e_rpool = NULL;
3690                                return 0;
3691                        }
```

```
3692                }

3694                e->e_flags |= EF_INQUEUE;
3695                eatheader(e, requeueflag, true);

3697                if (requeueflag)
3698                        queueup(e, false, false);

3700                /* do the delivery */
3701                sendall(e, SM_DELIVER);

3703                /* finish up and exit */
3704                if (forkflag)
3705                        finis(true, true, ExitStat);
3706                else
3707                {
3708                        (void) dropenvelope(e, true, false);
3709                        sm_rpool_free(rpool);
3710                        e->e_rpool = NULL;
3711                }
3712        }
3713        e->e_id = NULL;
3714        return pid;
3715 }

3717 /*
3718 **  DOWORKLIST -- process a list of envelopes as work requests
3719 **
3720 **      Similar to dowork(), except that after forking, it processes an
3721 **      envelope and its siblings, treating each envelope as a work request.
3722 **
3723 **      Parameters:
3724 **              el -- envelope to be processed including its siblings.
3725 **              forkflag -- if set, run this in background.
3726 **              requeueflag -- if set, reinstantiate the queue quickly.
3727 **                      This is used when expanding aliases in the queue.
3728 **                      If forkflag is also set, it doesn't wait for the
3729 **                      child.
3730 **
3731 **      Returns:
3732 **              process id of process that is running the queue job.
3733 **
3734 **      Side Effects:
3735 **              The work request is satisfied if possible.
3736 */

3738 pid_t
3739 doworklist(el, forkflag, requeueflag)
3740         ENVELOPE *el;
3741         bool forkflag;
3742         bool requeueflag;
3743 {
3744         register pid_t pid;
3745         ENVELOPE *ei;

3747         if (tTd(40, 1))
3748                 sm_dprintf("doworklist()\n");

3750         /*
3751         **  Fork for work.
3752         */

3754         if (forkflag)
3755         {
3756                 /*
3757                 **  Since the delivery may happen in a child and the
```



```
3758                 **  parent does not wait, the parent may close the
3759                 **  maps thereby removing any shared memory used by
3760                 **  the map.  Therefore, close the maps now so the
3761                 **  child will dynamically open them if necessary.
3762                 */

3764                 closemaps(false);

3766                 pid = fork();
3767                 if (pid < 0)
3768                 {
3769                         syserr("doworklist: cannot fork");
3770                         return 0;
3771                 }
3772                 else if (pid > 0)
3773                 {
3774                         /* parent -- clean out connection cache */
3775                         mci_flush(false, NULL);
3776                 }
3777                 else
3778                 {
3779                         /*
3780                         **  Initialize exception stack and default exception
3781                         **  handler for child process.
3782                         */

3784                         /* Reset global flags */
3785                         RestartRequest = NULL;
3786                         RestartWorkGroup = false;
3787                         ShutdownRequest = NULL;
3788                         PendingSignal = 0;
3789                         CurrentPid = getpid();
3790                         sm_exc_newthread(fatal_error);

3792                         /*
3793                         **  See note above about SMTP processes and SIGCHLD.
3794                         */

3796                         if (OpMode == MD_SMTP ||
3797                             OpMode == MD_DAEMON ||
3798                             MaxQueueChildren > 0)
3799                         {
3800                                 proc_list_clear();
3801                                 sm_releasesignal(SIGCHLD);
3802                                 (void) sm_signal(SIGCHLD, SIG_DFL);
3803                         }

3805                         /* child -- error messages to the transcript */
3806                         QuickAbort = OnlyOneError = false;
3807                 }
3808         }
3809         else
3810         {
3811                 pid = 0;
3812         }

3814         if (pid != 0)
3815                 return pid;

3817         /*
3818         **  IN CHILD
3819         **      Lock the control file to avoid duplicate deliveries.
3820         **              Then run the file as though we had just read it.
3821         **      We save an idea of the temporary name so we
3822         **              can recover on interrupt.
3823         */
```

```
3825         if (forkflag)
3826         {
3827                 /* Reset global flags */
3828                 RestartRequest = NULL;
3829                 RestartWorkGroup = false;
3830                 ShutdownRequest = NULL;
3831                 PendingSignal = 0;
3832         }

3834         /* set basic modes, etc. */
3835         sm_clear_events();
3836         clearstats();
3837         GrabTo = UseErrorsTo = false;
3838         ExitStat = EX_OK;
3839         if (forkflag)
3840         {
3841                 disconnect(1, el);
3842                 set_op_mode(MD_QUEUERUN);
3843         }
3844         if (LogLevel > 76)
3845                 sm_syslog(LOG_DEBUG, el->e_id, "doworklist, pid=%d",
3846                         (int) CurrentPid);

3848         for (ei = el; ei != NULL; ei = ei->e_sibling)
3849         {
3850                 ENVELOPE e;
3851                 SM_RPOOL_T *rpool;

3853                 if (WILL_BE_QUEUED(ei->e_sendmode))
3854                         continue;
3855                 else if (QueueMode != QM_QUARANTINE &&
3856                          ei->e_quarmsg != NULL)
3857                         continue;

3859                 rpool = sm_rpool_new_x(NULL);
3860                 clearenvelope(&e, true, rpool);
3861                 e.e_flags |= EF_QUEUERUN|EF_GLOBALERRS;
3862                 set_delivery_mode(SM_DELIVER, &e);
3863                 e.e_errormode = EM_MAIL;
3864                 e.e_id = ei->e_id;
3865                 e.e_qgrp = ei->e_qgrp;
3866                 e.e_qdir = ei->e_qdir;
3867                 openxscript(&e);
3868                 sm_setproctitle(true, &e, "%s from queue", qid_printname(&e));

3870                 /* don't use the headers from sendmail.cf... */
3871                 e.e_header = NULL;
3872                 CurEnv = &e;

3874                 /* read the queue control file -- return if locked */
3875                 if (readqf(&e, false))
3876                 {
3877                         e.e_flags |= EF_INQUEUE;
3878                         eatheader(&e, requeueflag, true);

3880                         if (requeueflag)
3881                                 queueup(&e, false, false);

3883                         /* do the delivery */
3884                         sendall(&e, SM_DELIVER);
3885                         (void) dropenvelope(&e, true, false);
3886                 }
3887                 else
3888                 {
3889                         if (tTd(40, 4) && e.e_id != NULL)
```



```
3890                                 sm_dprintf("readqf(%s) failed\n",
3891                                         qid_printname(&e));
3892                 }
3893                 sm_rpool_free(rpool);
3894                 ei->e_id = NULL;
3895         }

3897         /* restore CurEnv */
3898         CurEnv = el;

3900         /* finish up and exit */
3901         if (forkflag)
3902                 finis(true, true, ExitStat);
3903         return 0;
3904 }
3905 /*
3906 **  READQF -- read queue file and set up environment.
3907 **
3908 **      Parameters:
3909 **              e -- the envelope of the job to run.
3910 **              openonly -- only open the qf (returned as e_lockfp)
3911 **
3912 **      Returns:
3913 **              true if it successfully read the queue file.
3914 **              false otherwise.
3915 **
3916 **      Side Effects:
3917 **              The queue file is returned locked.
3918 */

3920 static bool
3921 readqf(e, openonly)
3922         register ENVELOPE *e;
3923         bool openonly;
3924 {
3925         register SM_FILE_T *qfp;
3926         ADDRESS *ctladdr;
3927         struct stat st, stf;
3928         char *bp;
3929         int qfver = 0;
3930         long hdrsize = 0;
3931         register char *p;
3932         char *frcpt = NULL;
3933         char *orcpt = NULL;
3934         bool nomore = false;
3935         bool bogus = false;
3936         MODE_T qsafe;
3937         char *err;
3938         char qf[MAXPATHLEN];
3939         char buf[MAXLINE];
3940         int bufsize;

3942         /*
3943         **  Read and process the file.
3944         */

3946         SM_REQUIRE(e != NULL);
3947         bp = NULL;
3948         (void) sm_strlcpy(qf, queuename(e, ANYQFL_LETTER), sizeof(qf));
3949         qfp = sm_io_open(SmFtStdio, SM_TIME_DEFAULT, qf, SM_IO_RDWR_B, NULL);
3950         if (qfp == NULL)
3951         {
3952                 int save_errno = errno;

3954                 if (tTd(40, 8))
3955                         sm_dprintf("readqf(%s): sm_io_open failure (%s)\n",
```

```
3956                                qf, sm_errstring(errno));
3957                errno = save_errno;
3958                if (errno != ENOENT
3959                    )
3960                        syserr("readqf: no control file %s", qf);
3961                RELEASE_QUEUE;
3962                return false;
3963        }
3964
3965        if (!lockfile(sm_io_getinfo(qfp, SM_IO_WHAT_FD, NULL), qf, NULL,
3966                      LOCK_EX|LOCK_NB))
3967        {
3968                /* being processed by another queuer */
3969                if (Verbose)
3970                        (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
3971                                             "%s: locked\n", e->e_id);
3972                if (tTd(40, 8))
3973                        sm_dprintf("%s: locked\n", e->e_id);
3974                if (LogLevel > 19)
3975                        sm_syslog(LOG_DEBUG, e->e_id, "locked");
3976                (void) sm_io_close(qfp, SM_TIME_DEFAULT);
3977                RELEASE_QUEUE;
3978                return false;
3979        }
3980
3981        RELEASE_QUEUE;
3982
3983        /*
3984        **  Prevent locking race condition.
3985        **
3986        **  Process A: readqf(): qfp = fopen(qffile)
3987        **  Process B: queueup(): rename(tf, qf)
3988        **  Process B: unlocks(tf)
3989        **  Process A: lockfile(qf);
3990        **
3991        **  Process A (us) has the old qf file (before the rename deleted
3992        **  the directory entry) and will be delivering based on old data.
3993        **  This can lead to multiple deliveries of the same recipients.
3994        **
3995        **  Catch this by checking if the underlying qf file has changed
3996        **  *after* acquiring our lock and if so, act as though the file
3997        **  was still locked (i.e., just return like the lockfile() case
3998        **  above.
3999        */
4000
4001        if (stat(qf, &stf) < 0 ||
4002            fstat(sm_io_getinfo(qfp, SM_IO_WHAT_FD, NULL), &st) < 0)
4003        {
4004                /* must have been being processed by someone else */
4005                if (tTd(40, 8))
4006                        sm_dprintf("readqf(%s): [f]stat failure (%s)\n",
4007                                   qf, sm_errstring(errno));
4008                (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4009                return false;
4010        }
4011
4012        if (st.st_nlink != stf.st_nlink ||
4013            st.st_dev != stf.st_dev ||
4014            ST_INODE(st) != ST_INODE(stf) ||
4015 #if HAS_ST_GEN && 0             /* AFS returns garbage in st_gen */
4016            st.st_gen != stf.st_gen ||
4017 #endif /* HAS_ST_GEN && 0 */
4018            st.st_uid != stf.st_uid ||
4019            st.st_gid != stf.st_gid ||
4020            st.st_size != stf.st_size)
4021        {
```



```
4022                /* changed after opened */
4023                if (Verbose)
4024                        (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4025                                             "%s: changed\n", e->e_id);
4026                if (tTd(40, 8))
4027                        sm_dprintf("%s: changed\n", e->e_id);
4028                if (LogLevel > 19)
4029                        sm_syslog(LOG_DEBUG, e->e_id, "changed");
4030                (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4031                return false;
4032        }
4033
4034        /*
4035        **  Check the queue file for plausibility to avoid attacks.
4036        */
4037
4038        qsafe = S_IWOTH|S_IWGRP;
4039        if (bitset(S_IWGRP, QueueFileMode))
4040                qsafe &= ~S_IWGRP;
4041
4042        bogus = st.st_uid != geteuid() &&
4043                st.st_uid != TrustedUid &&
4044                geteuid() != RealUid;
4045
4046        /*
4047        **  If this qf file results from a set-group-ID binary, then
4048        **  we check whether the directory is group-writable,
4049        **  the queue file mode contains the group-writable bit, and
4050        **  the groups are the same.
4051        **  Notice: this requires that the set-group-ID binary is used to
4052        **  run the queue!
4053        */
4054
4055        if (bogus && st.st_gid == getegid() && UseMSP)
4056        {
4057                char delim;
4058                struct stat dst;
4059
4060                bp = SM_LAST_DIR_DELIM(qf);
4061                if (bp == NULL)
4062                        delim = '\0';
4063                else
4064                {
4065                        delim = *bp;
4066                        *bp = '\0';
4067                }
4068                if (stat(delim == '\0' ? "." : qf, &dst) < 0)
4069                        syserr("readqf: cannot stat directory %s",
4070                               delim == '\0' ? "." : qf);
4071                else
4072                {
4073                        bogus = !(bitset(S_IWGRP, QueueFileMode) &&
4074                                  bitset(S_IWGRP, dst.st_mode) &&
4075                                  dst.st_gid == st.st_gid);
4076                }
4077                if (delim != '\0')
4078                        *bp = delim;
4079                bp = NULL;
4080        }
4081        if (!bogus)
4082                bogus = bitset(qsafe, st.st_mode);
4083        if (bogus)
4084        {
4085                if (LogLevel > 0)
4086                {
4087                        sm_syslog(LOG_ALERT, e->e_id,
```

```
4088                                    "bogus queue file, uid=%d, gid=%d, mode=%o",
4089                                    st.st_uid, st.st_gid, st.st_mode);
4090                    }
4091                    if (tTd(40, 8))
4092                            sm_dprintf("readqf(%s): bogus file\n", qf);
4093                    e->e_flags |= EF_INQUEUE;
4094                    if (!openonly)
4095                            loseqfile(e, "bogus file uid/gid in mqueue");
4096                    (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4097                    return false;
4098            }

4100            if (st.st_size == 0)
4101            {
4102                    /* must be a bogus file -- if also old, just remove it */
4103                    if (!openonly && st.st_ctime + 10 * 60 < curtime())
4104                    {
4105                            (void) xunlink(queuename(e, DATAFL_LETTER));
4106                            (void) xunlink(queuename(e, ANYQFL_LETTER));
4107                    }
4108                    (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4109                    return false;
4110            }

4112            if (st.st_nlink == 0)
4113            {
4114                    /*
4115                    **  Race condition -- we got a file just as it was being
4116                    **  unlinked.  Just assume it is zero length.
4117                    */

4119                    (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4120                    return false;
4121            }

4123 #if _FFR_TRUSTED_QF
4124            /*
4125            **  If we don't own the file mark it as unsafe.
4126            **  However, allow TrustedUser to own it as well
4127            **  in case TrustedUser manipulates the queue.
4128            */

4130            if (st.st_uid != geteuid() && st.st_uid != TrustedUid)
4131                    e->e_flags |= EF_UNSAFE;
4132 #else /* _FFR_TRUSTED_QF */
4133            /* If we don't own the file mark it as unsafe */
4134            if (st.st_uid != geteuid())
4135                    e->e_flags |= EF_UNSAFE;
4136 #endif /* _FFR_TRUSTED_QF */

4138            /* good file -- save this lock */
4139            e->e_lockfp = qfp;

4141            /* Just wanted the open file */
4142            if (openonly)
4143                    return true;

4145            /* do basic system initialization */
4146            initsys(e);
4147            macdefine(&e->e_macro, A_PERM, 'i', e->e_id);

4149            LineNumber = 0;
4150            e->e_flags |= EF_GLOBALERRS;
4151            set_op_mode(MD_QUEUERUN);
4152            ctladdr = NULL;
4153            e->e_qfletter = queue_letter(e, ANYQFL_LETTER);
```



```
4154            e->e_dfqgrp = e->e_qgrp;
4155            e->e_dfqdir = e->e_qdir;
4156 #if _FFR_QUEUE_MACRO
4157            macdefine(&e->e_macro, A_TEMP, macid("{queue}"),
4158                      qid_printqueue(e->e_qgrp, e->e_qdir));
4159 #endif /* _FFR_QUEUE_MACRO */
4160            e->e_dfino = -1;
4161            e->e_msgsize = -1;
4162            while (bufsize = sizeof(buf),
4163                   (bp = fgetfolded(buf, &bufsize, qfp)) != NULL)
4164            {
4165                    unsigned long qflags;
4166                    ADDRESS *q;
4167                    int r;
4168                    time_t now;
4169                    auto char *ep;

4171                    if (tTd(40, 4))
4172                            sm_dprintf("+++++ %s\n", bp);
4173                    if (nomore)
4174                    {
4175                            /* hack attack */
4176   hackattack:
4177                            syserr("SECURITY ALERT: extra or bogus data in queue fil
4178                                   bp);
4179                            err = "bogus queue line";
4180                            goto fail;
4181                    }
4182                    switch (bp[0])
4183                    {
4184                      case 'A':             /* AUTH= parameter */
4185                            if (!xtextok(&bp[1]))
4186                                    goto hackattack;
4187                            e->e_auth_param = sm_rpool_strdup_x(e->e_rpool, &bp[1]);
4188                            break;

4190                      case 'B':             /* body type */
4191                            r = check_bodytype(&bp[1]);
4192                            if (!BODYTYPE_VALID(r))
4193                                    goto hackattack;
4194                            e->e_bodytype = sm_rpool_strdup_x(e->e_rpool, &bp[1]);
4195                            break;

4197                      case 'C':             /* specify controlling user */
4198                            ctladdr = setctluser(&bp[1], qfver, e);
4199                            break;

4201                      case 'D':             /* data file name */
4202                            /* obsolete -- ignore */
4203                            break;

4205                      case 'd':             /* data file directory name */
4206                            {
4207                                    int qgrp, qdir;

4209 #if _FFR_MSP_PARANOIA
4210                                    /* forbid queue groups in MSP? */
4211                                    if (UseMSP)
4212                                            goto hackattack;
4213 #endif /* _FFR_MSP_PARANOIA */
4214                                    for (qgrp = 0;
4215                                         qgrp < NumQueue && Queue[qgrp] != NULL;
4216                                         ++qgrp)
4217                                    {
4218                                            for (qdir = 0;
4219                                                 qdir < Queue[qgrp]->qg_numqueues;
```

```
4220                                        ++qdir)
4221                                    {
4222                                            if (strcmp(&bp[1],
4223                                                       Queue[qgrp]->qg_qpath
4224                                                == 0)
4225                                            {
4226                                                    e->e_dfqgrp = qgrp;
4227                                                    e->e_dfqdir = qdir;
4228                                                    goto done;
4229                                            }
4230                                    }
4231                            }
4232                            err = "bogus queue file directory";
4233                            goto fail;
4234                      done:
4235                            break;
4236                    }

4238                case 'E':             /* specify error recipient */
4239                    /* no longer used */
4240                    break;

4242                case 'F':             /* flag bits */
4243                    if (strncmp(bp, "From ", 5) == 0)
4244                    {
4245                            /* we are being spoofed! */
4246                            syserr("SECURITY ALERT: bogus qf line %s", bp);
4247                            err = "bogus queue line";
4248                            goto fail;
4249                    }
4250                    for (p = &bp[1]; *p != '\0'; p++)
4251                    {
4252                            switch (*p)
4253                            {
4254                              case '8':     /* has 8 bit data */
4255                                    e->e_flags |= EF_HAS8BIT;
4256                                    break;

4258                              case 'b':     /* delete Bcc: header */
4259                                    e->e_flags |= EF_DELETE_BCC;
4260                                    break;

4262                              case 'd':     /* envelope has DSN RET= */
4263                                    e->e_flags |= EF_RET_PARAM;
4264                                    break;

4266                              case 'n':     /* don't return body */
4267                                    e->e_flags |= EF_NO_BODY_RETN;
4268                                    break;

4270                              case 'r':     /* response */
4271                                    e->e_flags |= EF_RESPONSE;
4272                                    break;

4274                              case 's':     /* split */
4275                                    e->e_flags |= EF_SPLIT;
4276                                    break;

4278                              case 'w':     /* warning sent */
4279                                    e->e_flags |= EF_WARNING;
4280                                    break;
4281                            }
4282                    }
4283                    break;

4285                case 'q':             /* quarantine reason */
```



```
4286                            e->e_quarmsg = sm_rpool_strdup_x(e->e_rpool, &bp[1]);
4287                            macdefine(&e->e_macro, A_PERM,
4288                                    macid("{quarantine}"), e->e_quarmsg);
4289                            break;

4291                case 'H':             /* header */

4293                            /*
4294                            **  count size before chompheader() destroys the line.
4295                            **  this isn't accurate due to macro expansion, but
4296                            **  better than before. "-3" to skip H?? at least.
4297                            */

4299                            hdrsize += strlen(bp) - 3;
4300                            (void) chompheader(&bp[1], CHHDR_QUEUE, NULL, e);
4301                            break;

4303                case 'I':             /* data file's inode number */
4304                            /* regenerated below */
4305                            break;

4307                case 'K':             /* time of last delivery attempt */
4308                            e->e_dtime = atol(&buf[1]);
4309                            break;

4311                case 'L':             /* Solaris Content-Length: */
4312                case 'M':             /* message */
4313                            /* ignore this; we want a new message next time */
4314                            break;

4316                case 'N':             /* number of delivery attempts */
4317                            e->e_ntries = atoi(&buf[1]);

4319                            /* if this has been tried recently, let it be */
4320                            now = curtime();
4321                            if (e->e_ntries > 0 && e->e_dtime <= now &&
4322                                now < e->e_dtime + MinQueueAge)
4323                            {
4324                                    char *howlong;

4326                                    howlong = pintvl(now - e->e_dtime, true);
4327                                    if (Verbose)
4328                                            (void) sm_io_fprintf(smioout,
4329                                                                 SM_TIME_DEFAULT,
4330                                                                 "%s: too young (%s)
4331                                                                 e->e_id, howlong);
4332                                    if (tTd(40, 8))
4333                                            sm_dprintf("%s: too young (%s)\n",
4334                                                       e->e_id, howlong);
4335                                    if (LogLevel > 19)
4336                                            sm_syslog(LOG_DEBUG, e->e_id,
4337                                                      "too young (%s)",
4338                                                      howlong);
4339                                    e->e_id = NULL;
4340                                    unlockqueue(e);
4341                                    if (bp != buf)
4342                                            sm_free(bp);
4343                                    return false;
4344                            }
4345                            macdefine(&e->e_macro, A_TEMP,
4346                                    macid("{ntries}"), &buf[1]);

4348 #if NAMED_BIND
4349                            /* adjust BIND parameters immediately */
4350                            if (e->e_ntries == 0)
4351                            {
```

```
4352                                _res.retry = TimeOuts.res_retry[RES_TO_FIRST];
4353                                _res.retrans = TimeOuts.res_retrans[RES_TO_FIRST
4354                        }
4355                        else
4356                        {
4357                                _res.retry = TimeOuts.res_retry[RES_TO_NORMAL];
4358                                _res.retrans = TimeOuts.res_retrans[RES_TO_NORMA
4359                        }
4360 #endif /* NAMED_BIND */
4361                        break;

4363                case 'P':               /* message priority */
4364                        e->e_msgpriority = atol(&bp[1]) + WkTimeFact;
4365                        break;

4367                case 'Q':               /* original recipient */
4368                        orcpt = sm_rpool_strdup_x(e->e_rpool, &bp[1]);
4369                        break;

4371                case 'r':               /* final recipient */
4372                        frcpt = sm_rpool_strdup_x(e->e_rpool, &bp[1]);
4373                        break;

4375                case 'R':               /* specify recipient */
4376                        p = bp;
4377                        qflags = 0;
4378                        if (qfver >= 1)
4379                        {
4380                                /* get flag bits */
4381                                while (*++p != '\0' && *p != ':')
4382                                {
4383                                        switch (*p)
4384                                        {
4385                                          case 'N':
4386                                                qflags |= QHASNOTIFY;
4387                                                break;

4389                                          case 'S':
4390                                                qflags |= QPINGONSUCCESS;
4391                                                break;

4393                                          case 'F':
4394                                                qflags |= QPINGONFAILURE;
4395                                                break;

4397                                          case 'D':
4398                                                qflags |= QPINGONDELAY;
4399                                                break;

4401                                          case 'P':
4402                                                qflags |= QPRIMARY;
4403                                                break;

4405                                          case 'A':
4406                                                if (ctladdr != NULL)
4407                                                        ctladdr->q_flags |= QALI
4408                                                break;

4410                                          default: /* ignore or complain? */
4411                                                break;
4412                                        }
4413                                }
4414                        }
4415                        else
4416                                qflags |= QPRIMARY;
4417                        macdefine(&e->e_macro, A_PERM, macid("{addr_type}"),
```



```
4418                                "e r");
4419                        if (*p != '\0')
4420                                q = parseaddr(++p, NULLADDR, RF_COPYALL, '\0',
4421                                              NULL, e, true);
4422                        else
4423                                q = NULL;
4424                        if (q != NULL)
4425                        {
4426                                /* make sure we keep the current qgrp */
4427                                if (ISVALIDQGRP(e->e_qgrp))
4428                                        q->q_qgrp = e->e_qgrp;
4429                                q->q_alias = ctladdr;
4430                                if (qfver >= 1)
4431                                        q->q_flags &= ~Q_PINGFLAGS;
4432                                q->q_flags |= qflags;
4433                                q->q_finalrcpt = frcpt;
4434                                q->q_orcpt = orcpt;
4435                                (void) recipient(q, &e->e_sendqueue, 0, e);
4436                        }
4437                        frcpt = NULL;
4438                        orcpt = NULL;
4439                        macdefine(&e->e_macro, A_PERM, macid("{addr_type}"),
4440                                NULL);
4441                        break;

4443                case 'S':               /* sender */
4444                        setsender(sm_rpool_strdup_x(e->e_rpool, &bp[1]),
4445                                  e, NULL, '\0', true);
4446                        break;

4448                case 'T':               /* init time */
4449                        e->e_ctime = atol(&bp[1]);
4450                        break;

4452                case 'V':               /* queue file version number */
4453                        qfver = atoi(&bp[1]);
4454                        if (qfver <= QF_VERSION)
4455                                break;
4456                        syserr("Version number in queue file (%d) greater than m
4457                                qfver, QF_VERSION);
4458                        err = "unsupported queue file version";
4459                        goto fail;
4460                        /* NOTREACHED */
4461                        break;

4463                case 'Z':               /* original envelope id from ESMTP */
4464                        e->e_envid = sm_rpool_strdup_x(e->e_rpool, &bp[1]);
4465                        macdefine(&e->e_macro, A_PERM,
4466                                macid("{dsn_envid}"), e->e_envid);
4467                        break;

4469                case '!':               /* deliver by */

4471                        /* format: flag (1 char) space long-integer */
4472                        e->e_dlvr_flag = buf[1];
4473                        e->e_deliver_by = strtol(&buf[3], NULL, 10);

4475                case '$':               /* define macro */
4476                        {
4477                                char *p;

4479                                /* XXX elimate p? */
4480                                r = macid_parse(&bp[1], &ep);
4481                                if (r == 0)
4482                                        break;
4483                                p = sm_rpool_strdup_x(e->e_rpool, ep);
```

```
4484                                    macdefine(&e->e_macro, A_PERM, r, p);
4485                            }
4486                            break;

4488                    case '.':               /* terminate file */
4489                            nomore = true;
4490                            break;

4492 #if _FFR_QUEUEDELAY
4493                    case 'G':
4494                    case 'Y':

4496                            /*
4497                            **  Maintain backward compatibility for
4498                            **  users who defined _FFR_QUEUEDELAY in
4499                            **  previous releases.  Remove this
4500                            **  code in 8.14 or 8.15.
4501                            */

4503                            if (qfver == 5 || qfver == 7)
4504                                    break;

4506                            /* If not qfver 5 or 7, then 'G' or 'Y' is invalid */
4507                            /* FALLTHROUGH */
4508 #endif /* _FFR_QUEUEDELAY */

4510                    default:
4511                            syserr("readqf: %s: line %d: bad line \"%s\"",
4512                                    qf, LineNumber, shortenstring(bp, MAXSHORTSTR));
4513                            err = "unrecognized line";
4514                            goto fail;
4515                    }

4517                    if (bp != buf)
4518                            SM_FREE(bp);
4519            }

4521            /*
4522            **  If we haven't read any lines, this queue file is empty.
4523            **  Arrange to remove it without referencing any null pointers.
4524            */

4526            if (LineNumber == 0)
4527            {
4528                    errno = 0;
4529                    e->e_flags |= EF_CLRQUEUE|EF_FATALERRS|EF_RESPONSE;
4530                    return true;
4531            }

4533            /* Check to make sure we have a complete queue file read */
4534            if (!nomore)
4535            {
4536                    syserr("readqf: %s: incomplete queue file read", qf);
4537                    (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4538                    return false;
4539            }

4541 #if _FFR_QF_PARANOIA
4542            /* Check to make sure key fields were read */
4543            if (e->e_from.q_mailer == NULL)
4544            {
4545                    syserr("readqf: %s: sender not specified in queue file", qf);
4546                    (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4547                    return false;
4548            }
4549            /* other checks? */
```



```
4550 #endif /* _FFR_QF_PARANOIA */

4552            /* possibly set ${dsn_ret} macro */
4553            if (bitset(EF_RET_PARAM, e->e_flags))
4554            {
4555                    if (bitset(EF_NO_BODY_RETN, e->e_flags))
4556                            macdefine(&e->e_macro, A_PERM,
4557                                    macid("{dsn_ret}"), "hdrs");
4558                    else
4559                            macdefine(&e->e_macro, A_PERM,
4560                                    macid("{dsn_ret}"), "full");
4561            }

4563            /*
4564            **  Arrange to read the data file.
4565            */

4567            p = queuename(e, DATAFL_LETTER);
4568            e->e_dfp = sm_io_open(SmFtStdio, SM_TIME_DEFAULT, p, SM_IO_RDONLY_B,
4569                                  NULL);
4570            if (e->e_dfp == NULL)
4571            {
4572                    syserr("readqf: cannot open %s", p);
4573            }
4574            else
4575            {
4576                    e->e_flags |= EF_HAS_DF;
4577                    if (fstat(sm_io_getinfo(e->e_dfp, SM_IO_WHAT_FD, NULL), &st)
4578                        >= 0)
4579                    {
4580                            e->e_msgsize = st.st_size + hdrsize;
4581                            e->e_dfdev = st.st_dev;
4582                            e->e_dfino = ST_INODE(st);
4583                            (void) sm_snprintf(buf, sizeof(buf), "%ld",
4584                                               e->e_msgsize);
4585                            macdefine(&e->e_macro, A_TEMP, macid("{msg_size}"),
4586                                      buf);
4587                    }
4588            }

4590            return true;

4592    fail:
4593            /*
4594            **  There was some error reading the qf file (reason is in err var.)
4595            **  Cleanup:
4596            **      close file; clear e_lockfp since it is the same as qfp,
4597            **      hence it is invalid (as file) after qfp is closed;
4598            **      the qf file is on disk, so set the flag to avoid calling
4599            **      queueup() with bogus data.
4600            */

4602            if (bp != buf)
4603                    SM_FREE(bp);
4604            if (qfp != NULL)
4605                    (void) sm_io_close(qfp, SM_TIME_DEFAULT);
4606            e->e_lockfp = NULL;
4607            e->e_flags |= EF_INQUEUE;
4608            loseqfile(e, err);
4609            return false;
4610 }
4611 /*
4612 **  PRTSTR -- print a string, "unprintable" characters are shown as \oct
4613 **
4614 **      Parameters:
4615 **              s -- string to print
```

```
4616 **              ml -- maximum length of output
4617 **
4618 **      Returns:
4619 **              number of entries
4620 **
4621 **      Side Effects:
4622 **              Prints a string on stdout.
4623 */

4625 static void prtstr __P((char *, int));

4627 static void
4628 prtstr(s, ml)
4629         char *s;
4630         int ml;
4631 {
4632         int c;

4634         if (s == NULL)
4635                 return;
4636         while (ml-- > 0 && ((c = *s++) != '\0'))
4637         {
4638                 if (c == '\\')
4639                 {
4640                         if (ml-- > 0)
4641                         {
4642                                 (void) sm_io_putc(smioout, SM_TIME_DEFAULT, c);
4643                                 (void) sm_io_putc(smioout, SM_TIME_DEFAULT, c);
4644                         }
4645                 }
4646                 else if (isascii(c) && isprint(c))
4647                         (void) sm_io_putc(smioout, SM_TIME_DEFAULT, c);
4648                 else
4649                 {
4650                         if ((ml -= 3) > 0)
4651                                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4652                                                      "\\%03o", c & 0xFF);
4653                 }
4654         }
4655 }
4656 /*
4657 **  PRINTNQE -- print out number of entries in the mail queue
4658 **
4659 **      Parameters:
4660 **              out -- output file pointer.
4661 **              prefix -- string to output in front of each line.
4662 **
4663 **      Returns:
4664 **              none.
4665 */

4667 void
4668 printnqe(out, prefix)
4669         SM_FILE_T *out;
4670         char *prefix;
4671 {
4672 #if SM_CONF_SHM
4673         int i, k = 0, nrequests = 0;
4674         bool unknown = false;

4676         if (ShmId == SM_SHM_NO_ID)
4677         {
4678                 if (prefix == NULL)
4679                         (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4680                                              "Data unavailable: shared memory not upd
4681                 else
```



```
4682                         (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4683                                              "%sNOTCONFIGURED:-1\r\n", prefix);
4684                 return;
4685         }
4686         for (i = 0; i < NumQueue && Queue[i] != NULL; i++)
4687         {
4688                 int j;

4690                 k++;
4691                 for (j = 0; j < Queue[i]->qg_numqueues; j++)
4692                 {
4693                         int n;

4695                         if (StopRequest)
4696                                 stop_sendmail();

4698                         n = QSHM_ENTRIES(Queue[i]->qg_qpaths[j].qp_idx);
4699                         if (prefix != NULL)
4700                                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4701                                                      "%s%s:%d\r\n",
4702                                                      prefix, qid_printqueue(i, j), n);
4703                         else if (n < 0)
4704                         {
4705                                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4706                                                      "%s: unknown number of entries\n",
4707                                                      qid_printqueue(i, j));
4708                                 unknown = true;
4709                         }
4710                         else if (n == 0)
4711                         {
4712                                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4713                                                      "%s is empty\n",
4714                                                      qid_printqueue(i, j));
4715                         }
4716                         else if (n > 0)
4717                         {
4718                                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4719                                                      "%s: entries=%d\n",
4720                                                      qid_printqueue(i, j), n);
4721                                 nrequests += n;
4722                                 k++;
4723                         }
4724                 }
4725         }
4726         if (prefix == NULL && k > 1)
4727                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4728                                      "\t\tTotal requests: %d%s\n",
4729                                      nrequests, unknown ? " (about)" : "");
4730 #else /* SM_CONF_SHM */
4731         if (prefix == NULL)
4732                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4733                                      "Data unavailable without shared memory support\n")
4734         else
4735                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
4736                                      "%sNOTAVAILABLE:-1\r\n", prefix);
4737 #endif /* SM_CONF_SHM */
4738 }
4739 /*
4740 **  PRINTQUEUE -- print out a representation of the mail queue
4741 **
4742 **      Parameters:
4743 **              none.
4744 **
4745 **      Returns:
4746 **              none.
4747 **
```

```
4748 **      Side Effects:
4749 **              Prints a listing of the mail queue on the standard output.
4750 */

4752 void
4753 printqueue()
4754 {
4755         int i, k = 0, nrequests = 0;

4757         for (i = 0; i < NumQueue && Queue[i] != NULL; i++)
4758         {
4759                 int j;

4761                 k++;
4762                 for (j = 0; j < Queue[i]->qg_numqueues; j++)
4763                 {
4764                         if (StopRequest)
4765                                 stop_sendmail();
4766                         nrequests += print_single_queue(i, j);
4767                         k++;
4768                 }
4769         }
4770         if (k > 1)
4771                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4772                                      "\t\tTotal requests: %d\n",
4773                                      nrequests);
4774 }
4775 /*
4776 **  PRINT_SINGLE_QUEUE -- print out a representation of a single mail queue
4777 **
4778 **      Parameters:
4779 **              qgrp -- the index of the queue group.
4780 **              qdir -- the queue directory.
4781 **
4782 **      Returns:
4783 **              number of requests in mail queue.
4784 **
4785 **      Side Effects:
4786 **              Prints a listing of the mail queue on the standard output.
4787 */

4789 int
4790 print_single_queue(qgrp, qdir)
4791         int qgrp;
4792         int qdir;
4793 {
4794         register WORK *w;
4795         SM_FILE_T *f;
4796         int nrequests;
4797         char qd[MAXPATHLEN];
4798         char qddf[MAXPATHLEN];
4799         char buf[MAXLINE];

4801         if (qdir == NOQDIR)
4802         {
4803                 (void) sm_strlcpy(qd, ".", sizeof(qd));
4804                 (void) sm_strlcpy(qddf, ".", sizeof(qddf));
4805         }
4806         else
4807         {
4808                 (void) sm_strlcpyn(qd, sizeof(qd), 2,
4809                         Queue[qgrp]->qg_qpaths[qdir].qp_name,
4810                         (bitset(QP_SUBQF,
4811                                 Queue[qgrp]->qg_qpaths[qdir].qp_subdirs)
4812                                         ? "/qf" : ""));
4813                 (void) sm_strlcpyn(qddf, sizeof(qddf), 2,
```



```
4814                         Queue[qgrp]->qg_qpaths[qdir].qp_name,
4815                         (bitset(QP_SUBDF,
4816                                 Queue[qgrp]->qg_qpaths[qdir].qp_subdirs)
4817                                         ? "/df" : ""));
4818         }

4820         /*
4821         **  Check for permission to print the queue
4822         */

4824         if (bitset(PRIV_RESTRICTMAILQ, PrivacyFlags) && RealUid != 0)
4825         {
4826                 struct stat st;
4827 #ifdef NGROUPS_MAX
4828                 int n;
4829                 extern GIDSET_T InitialGidSet[NGROUPS_MAX];
4830 #endif /* NGROUPS_MAX */

4832                 if (stat(qd, &st) < 0)
4833                 {
4834                         syserr("Cannot stat %s",
4835                                 qid_printqueue(qgrp, qdir));
4836                         return 0;
4837                 }
4838 #ifdef NGROUPS_MAX
4839                 n = NGROUPS_MAX;
4840                 while (--n >= 0)
4841                 {
4842                         if (InitialGidSet[n] == st.st_gid)
4843                                 break;
4844                 }
4845                 if (n < 0 && RealGid != st.st_gid)
4846 #else /* NGROUPS_MAX */
4847                 if (RealGid != st.st_gid)
4848 #endif /* NGROUPS_MAX */
4849                 {
4850                         usrerr("510 You are not permitted to see the queue");
4851                         setstat(EX_NOPERM);
4852                         return 0;
4853                 }
4854         }

4856         /*
4857         **  Read and order the queue.
4858         */

4860         nrequests = gatherq(qgrp, qdir, true, NULL, NULL, NULL);
4861         (void) sortq(Queue[qgrp]->qg_maxlist);

4863         /*
4864         **  Print the work list that we have read.
4865         */

4867         /* first see if there is anything */
4868         if (nrequests <= 0)
4869         {
4870                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "%s is empty\n",
4871                                      qid_printqueue(qgrp, qdir));
4872                 return 0;
4873         }

4875         sm_getla();     /* get load average */

4877         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "\t\t%s (%d request%s",
4878                              qid_printqueue(qgrp, qdir),
4879                              nrequests, nrequests == 1 ? "" : "s");
```

```
4880         if (MaxQueueRun > 0 && nrequests > MaxQueueRun)
4881                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4882                                      ", only %d printed", MaxQueueRun);
4883         if (Verbose)
4884                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4885                                      ")\n-----Q-ID----- --Size-- -Priority- ---Q-Time--- ----
4886         else
4887                 (void) sm_io_fprintf(smioout,  SM_TIME_DEFAULT,
4888                                      ")\n-----Q-ID----- --Size-- -----Q-Time----- ----------
4889         for (w = WorkQ; w != NULL; w = w->w_next)
4890         {
4891                 struct stat st;
4892                 auto time_t submittime = 0;
4893                 long dfsize;
4894                 int flags = 0;
4895                 int qfver;
4896                 char quarmsg[MAXLINE];
4897                 char statmsg[MAXLINE];
4898                 char bodytype[MAXNAME + 1];
4899                 char qf[MAXPATHLEN];

4901                 if (StopRequest)
4902                         stop_sendmail();

4904                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "%13s",
4905                                      w->w_name + 2);
4906                 (void) sm_strlcpyn(qf, sizeof(qf), 3, qd, "/", w->w_name);
4907                 f = sm_io_open(SmFtStdio, SM_TIME_DEFAULT, qf, SM_IO_RDONLY_B,
4908                                NULL);
4909                 if (f == NULL)
4910                 {
4911                         if (errno == EPERM)
4912                                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4913                                                      " (permission denied)\n");
4914                         else if (errno == ENOENT)
4915                                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4916                                                      " (job completed)\n");
4917                         else
4918                                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
4919                                                      " (%s)\n",
4920                                                      sm_errstring(errno));
4921                         errno = 0;
4922                         continue;
4923                 }
4924                 w->w_name[0] = DATAFL_LETTER;
4925                 (void) sm_strlcpyn(qf, sizeof(qf), 3, qddf, "/", w->w_name);
4926                 if (stat(qf, &st) >= 0)
4927                         dfsize = st.st_size;
4928                 else
4929                 {
4930                         ENVELOPE e;

4932                         /*
4933                         **  Maybe the df file can't be statted because
4934                         **  it is in a different directory than the qf file.
4935                         **  In order to find out, we must read the qf file.
4936                         */

4938                         newenvelope(&e, &BlankEnvelope, sm_rpool_new_x(NULL));
4939                         e.e_id = w->w_name + 2;
4940                         e.e_qgrp = qgrp;
4941                         e.e_qdir = qdir;
4942                         dfsize = -1;
4943                         if (readqf(&e, false))
4944                         {
4945                                 char *df = queuename(&e, DATAFL_LETTER);
```



```
4946                                 if (stat(df, &st) >= 0)
4947                                         dfsize = st.st_size;
4948                         }
4949                         if (e.e_lockfp != NULL)
4950                         {
4951                                 (void) sm_io_close(e.e_lockfp, SM_TIME_DEFAULT);
4952                                 e.e_lockfp = NULL;
4953                         }
4954                         clearenvelope(&e, false, e.e_rpool);
4955                         sm_rpool_free(e.e_rpool);
4956                 }
4957                 if (w->w_lock)
4958                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "*");
4959                 else if (QueueMode == QM_LOST)
4960                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "?");
4961                 else if (w->w_tooyoung)
4962                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "-");
4963                 else if (shouldqueue(w->w_pri, w->w_ctime))
4964                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "X");
4965                 else
4966                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, " ");

4968                 errno = 0;

4970                 quarmsg[0] = '\0';
4971                 statmsg[0] = bodytype[0] = '\0';
4972                 qfver = 0;
4973                 while (sm_io_fgets(f, SM_TIME_DEFAULT, buf, sizeof(buf)) != NULL
4974                 {
4975                         register int i;
4976                         register char *p;

4978                         if (StopRequest)
4979                                 stop_sendmail();

4981                         fixcrlf(buf, true);
4982                         switch (buf[0])
4983                         {
4984                           case 'V':     /* queue file version */
4985                                 qfver = atoi(&buf[1]);
4986                                 break;

4988                           case 'M':     /* error message */
4989                                 if ((i = strlen(&buf[1])) >= sizeof(statmsg))
4990                                         i = sizeof(statmsg) - 1;
4991                                 memmove(statmsg, &buf[1], i);
4992                                 statmsg[i] = '\0';
4993                                 break;

4995                           case 'q':     /* quarantine reason */
4996                                 if ((i = strlen(&buf[1])) >= sizeof(quarmsg))
4997                                         i = sizeof(quarmsg) - 1;
4998                                 memmove(quarmsg, &buf[1], i);
4999                                 quarmsg[i] = '\0';
5000                                 break;

5002                           case 'B':     /* body type */
5003                                 if ((i = strlen(&buf[1])) >= sizeof(bodytype))
5004                                         i = sizeof(bodytype) - 1;
5005                                 memmove(bodytype, &buf[1], i);
5006                                 bodytype[i] = '\0';
5007                                 break;

5009                           case 'S':     /* sender name */
5010                                 if (Verbose)
5011                                 {
```

```
5012                                    (void) sm_io_fprintf(smioout,
5013                                            SM_TIME_DEFAULT,
5014                                            "%8ld %10ld%c%.12s ",
5015                                            dfsize,
5016                                            w->w_pri,
5017                                            bitset(EF_WARNING, flags)
5018                                                    ? '+' : ' ',
5019                                            ctime(&submittime) + 4);
5020                                    prtstr(&buf[1], 78);
5021                            }
5022                            else
5023                            {
5024                                    (void) sm_io_fprintf(smioout,
5025                                            SM_TIME_DEFAULT,
5026                                            "%8ld %.16s ",
5027                                            dfsize,
5028                                            ctime(&submittime));
5029                                    prtstr(&buf[1], 39);
5030                            }

5032                            if (quarmsg[0] != '\0')
5033                            {
5034                                    (void) sm_io_fprintf(smioout,
5035                                                    SM_TIME_DEFAULT,
5036                                                    "\n     QUARANTINE:
5037                                                    Verbose ? 100 : 60,
5038                                                    quarmsg);
5039                                    quarmsg[0] = '\0';
5040                            }

5042                            if (statmsg[0] != '\0' || bodytype[0] != '\0')
5043                            {
5044                                    (void) sm_io_fprintf(smioout,
5045                                            SM_TIME_DEFAULT,
5046                                            "\n    %10.10s",
5047                                            bodytype);
5048                                    if (statmsg[0] != '\0')
5049                                            (void) sm_io_fprintf(smioout,
5050                                                    SM_TIME_DEFAULT,
5051                                                    "   (%.*s)",
5052                                                    Verbose ? 100 : 60,
5053                                                    statmsg);
5054                                    statmsg[0] = '\0';
5055                            }
5056                            break;

5058                    case 'C':       /* controlling user */
5059                            if (Verbose)
5060                                    (void) sm_io_fprintf(smioout,
5061                                            SM_TIME_DEFAULT,
5062                                            "\n\t\t\t\t\t\t(---%.64s---)",
5063                                            &buf[1]);
5064                            break;

5066                    case 'R':       /* recipient name */
5067                            p = &buf[1];
5068                            if (qfver >= 1)
5069                            {
5070                                    p = strchr(p, ':');
5071                                    if (p == NULL)
5072                                            break;
5073                                    p++;
5074                            }
5075                            if (Verbose)
5076                            {
5077                                    (void) sm_io_fprintf(smioout,
```



```
5078                                                    SM_TIME_DEFAULT,
5079                                                    "\n\t\t\t\t\t\t");
5080                                    prtstr(p, 71);
5081                            }
5082                            else
5083                            {
5084                                    (void) sm_io_fprintf(smioout,
5085                                                    SM_TIME_DEFAULT,
5086                                                    "\n\t\t\t\t\t ");
5087                                    prtstr(p, 38);
5088                            }
5089                            if (Verbose && statmsg[0] != '\0')
5090                            {
5091                                    (void) sm_io_fprintf(smioout,
5092                                                    SM_TIME_DEFAULT,
5093                                                    "\n\t\t (%.100s)",
5094                                                    statmsg);
5095                                    statmsg[0] = '\0';
5096                            }
5097                            break;

5099                    case 'T':       /* creation time */
5100                            submittime = atol(&buf[1]);
5101                            break;

5103                    case 'F':       /* flag bits */
5104                            for (p = &buf[1]; *p != '\0'; p++)
5105                            {
5106                                    switch (*p)
5107                                    {
5108                                      case 'w':
5109                                            flags |= EF_WARNING;
5110                                            break;
5111                                    }
5112                            }
5113                    }
5114            }
5115            if (submittime == (time_t) 0)
5116                    (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
5117                                                  " (no control file)");
5118            (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT, "\n");
5119            (void) sm_io_close(f, SM_TIME_DEFAULT);
5120    }
5121    return nrequests;
5122 }

5124 /*
5125 **  QUEUE_LETTER -- get the proper queue letter for the current QueueMode.
5126 **
5127 **      Parameters:
5128 **              e -- envelope to build it in/from.
5129 **              type -- the file type, used as the first character
5130 **                      of the file name.
5131 **
5132 **      Returns:
5133 **              the letter to use
5134 */

5136 static char
5137 queue_letter(e, type)
5138         ENVELOPE *e;
5139         int type;
5140 {
5141         /* Change type according to QueueMode */
5142         if (type == ANYQFL_LETTER)
5143         {
```

```
5144                if (e->e_quarmsg != NULL)
5145                        type = QUARQF_LETTER;
5146                else
5147                {
5148                        switch (QueueMode)
5149                        {
5150                          case QM_NORMAL:
5151                                type = NORMQF_LETTER;
5152                                break;

5154                          case QM_QUARANTINE:
5155                                type = QUARQF_LETTER;
5156                                break;

5158                          case QM_LOST:
5159                                type = LOSEQF_LETTER;
5160                                break;

5162                          default:
5163                                /* should never happen */
5164                                abort();
5165                                /* NOTREACHED */
5166                        }
5167                }
5168        }
5169        return type;
5170 }

5172 /*
5173 **  QUEUENAME -- build a file name in the queue directory for this envelope.
5174 **
5175 **      Parameters:
5176 **              e -- envelope to build it in/from.
5177 **              type -- the file type, used as the first character
5178 **                      of the file name.
5179 **
5180 **      Returns:
5181 **              a pointer to the queue name (in a static buffer).
5182 **
5183 **      Side Effects:
5184 **              If no id code is already assigned, queuename() will
5185 **              assign an id code with assign_queueid().  If no queue
5186 **              directory is assigned, one will be set with setnewqueue().
5187 */

5189 char *
5190 queuename(e, type)
5191         register ENVELOPE *e;
5192         int type;
5193 {
5194         int qd, qg;
5195         char *sub = "/";
5196         char pref[3];
5197         static char buf[MAXPATHLEN];

5199         /* Assign an ID if needed */
5200         if (e->e_id == NULL)
5201                 assign_queueid(e);
5202         type = queue_letter(e, type);

5204         /* begin of filename */
5205         pref[0] = (char) type;
5206         pref[1] = 'f';
5207         pref[2] = '\0';

5209         /* Assign a queue group/directory if needed */
```



```
5210         if (type == XSCRPT_LETTER)
5211         {
5212                 /*
5213                 **  We don't want to call setnewqueue() if we are fetching
5214                 **  the pathname of the transcript file, because setnewqueue
5215                 **  chooses a queue, and sometimes we need to write to the
5216                 **  transcript file before we have gathered enough information
5217                 **  to choose a queue.
5218                 */

5220                 if (e->e_xfqgrp == NOQGRP || e->e_xfqdir == NOQDIR)
5221                 {
5222                         if (e->e_qgrp != NOQGRP && e->e_qdir != NOQDIR)
5223                         {
5224                                 e->e_xfqgrp = e->e_qgrp;
5225                                 e->e_xfqdir = e->e_qdir;
5226                         }
5227                         else
5228                         {
5229                                 e->e_xfqgrp = 0;
5230                                 if (Queue[e->e_xfqgrp]->qg_numqueues <= 1)
5231                                         e->e_xfqdir = 0;
5232                                 else
5233                                 {
5234                                         e->e_xfqdir = get_rand_mod(
5235                                                 Queue[e->e_xfqgrp]->qg_numqueues);
5236                                 }
5237                         }
5238                 }
5239                 qd = e->e_xfqdir;
5240                 qg = e->e_xfqgrp;
5241         }
5242         else
5243         {
5244                 if (e->e_qgrp == NOQGRP || e->e_qdir == NOQDIR)
5245                         (void) setnewqueue(e);
5246                 if (type ==  DATAFL_LETTER)
5247                 {
5248                         qd = e->e_dfqdir;
5249                         qg = e->e_dfqgrp;
5250                 }
5251                 else
5252                 {
5253                         qd = e->e_qdir;
5254                         qg = e->e_qgrp;
5255                 }
5256         }

5258         /* xf files always have a valid qd and qg picked above */
5259         if ((qd == NOQDIR || qg == NOQGRP) && type != XSCRPT_LETTER)
5260                 (void) sm_strlcpyn(buf, sizeof(buf), 2, pref, e->e_id);
5261         else
5262         {
5263                 switch (type)
5264                 {
5265                   case DATAFL_LETTER:
5266                         if (bitset(QP_SUBDF, Queue[qg]->qg_qpaths[qd].qp_subdirs
5267                                 sub = "/df/";
5268                         break;

5270                   case QUARQF_LETTER:
5271                   case TEMPQF_LETTER:
5272                   case NEWQFL_LETTER:
5273                   case LOSEQF_LETTER:
5274                   case NORMQF_LETTER:
5275                         if (bitset(QP_SUBQF, Queue[qg]->qg_qpaths[qd].qp_subdirs
```

```
5276                                sub = "/qf/";
5277                        break;

5279                  case XSCRPT_LETTER:
5280                        if (bitset(QP_SUBXF, Queue[qg]->qg_qpaths[qd].qp_subdirs
5281                                sub = "/xf/";
5282                        break;

5284                  default:
5285                        sm_abort("queuename: bad queue file type %d", type);
5286                }

5288                (void) sm_strlcpyn(buf, sizeof(buf), 4,
5289                                Queue[qg]->qg_qpaths[qd].qp_name,
5290                                sub, pref, e->e_id);
5291        }

5293        if (tTd(7, 2))
5294                sm_dprintf("queuename: %s\n", buf);
5295        return buf;
5296 }

5298 /*
5299 **  INIT_QID_ALG -- Initialize the (static) parameters that are used to
5300 **      generate a queue ID.
5301 **
5302 **      This function is called by the daemon to reset
5303 **      LastQueueTime and LastQueuePid which are used by assign_queueid().
5304 **      Otherwise the algorithm may cause problems because
5305 **      LastQueueTime and LastQueuePid are set indirectly by main()
5306 **      before the daemon process is started, hence LastQueuePid is not
5307 **      the pid of the daemon and therefore a child of the daemon can
5308 **      actually have the same pid as LastQueuePid which means the section
5309 **      in  assign_queueid():
5310 **      * see if we need to get a new base time/pid *
5311 **      is NOT triggered which will cause the same queue id to be generated.
5312 **
5313 **      Parameters:
5314 **              none
5315 **
5316 **      Returns:
5317 **              none.
5318 */

5320 void
5321 init_qid_alg()
5322 {
5323        LastQueueTime = 0;
5324        LastQueuePid = -1;
5325 }

5327 /*
5328 **  ASSIGN_QUEUEID -- assign a queue ID for this envelope.
5329 **
5330 **      Assigns an id code if one does not already exist.
5331 **      This code assumes that nothing will remain in the queue for
5332 **      longer than 60 years.  It is critical that files with the given
5333 **      name do not already exist in the queue.
5334 **      [No longer initializes e_qdir to NOQDIR.]
5335 **
5336 **      Parameters:
5337 **              e -- envelope to set it in.
5338 **
5339 **      Returns:
5340 **              none.
5341 */
```



```
5343 static const char QueueIdChars[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefgh
5344 # define QIC_LEN        60
5345 # define QIC_LEN_R      62

5347 /*
5348 **  Note: the length is "officially" 60 because minutes and seconds are
5349 **      usually only 0-59.  However (Linux):
5350 **       tm_sec The number of seconds after the minute, normally in
5351 **              the range 0 to 59, but can be up to 61 to allow for
5352 **              leap seconds.
5353 **      Hence the real length of the string is 62 to take this into account.
5354 **      Alternatively % QIC_LEN can (should) be used for access everywhere.
5355 */

5357 # define queuenextid() CurrentPid
5358 #define QIC_LEN_SQR     (QIC_LEN * QIC_LEN)

5360 void
5361 assign_queueid(e)
5362        register ENVELOPE *e;
5363 {
5364        pid_t pid = queuenextid();
5365        static unsigned int cX = 0;
5366        static unsigned int random_offset;
5367        struct tm *tm;
5368        char idbuf[MAXQFNAME - 2];
5369        unsigned int seq;

5371        if (e->e_id != NULL)
5372                return;

5374        /* see if we need to get a new base time/pid */
5375        if (cX >= QIC_LEN_SQR || LastQueueTime == 0 || LastQueuePid != pid)
5376        {
5377                time_t then = LastQueueTime;

5379                /* if the first time through, pick a random offset */
5380                if (LastQueueTime == 0)
5381                        random_offset = ((unsigned int)get_random())
5382                                        % QIC_LEN_SQR;

5384                while ((LastQueueTime = curtime()) == then &&
5385                       LastQueuePid == pid)
5386                {
5387                        (void) sleep(1);
5388                }
5389                LastQueuePid = queuenextid();
5390                cX = 0;
5391        }

5393        /*
5394        **  Generate a new sequence number between 0 and QIC_LEN_SQR-1.
5395        **  This lets us generate up to QIC_LEN_SQR unique queue ids
5396        **  per second, per process.  With envelope splitting,
5397        **  a single message can consume many queue ids.
5398        */

5400        seq = (cX + random_offset) % QIC_LEN_SQR;
5401        ++cX;
5402        if (tTd(7, 50))
5403                sm_dprintf("assign_queueid: random_offset=%u (%u)\n",
5404                        random_offset, seq);

5406        tm = gmtime(&LastQueueTime);
5407        idbuf[0] = QueueIdChars[tm->tm_year % QIC_LEN];
```

```
5408         idbuf[1] = QueueIdChars[tm->tm_mon];
5409         idbuf[2] = QueueIdChars[tm->tm_mday];
5410         idbuf[3] = QueueIdChars[tm->tm_hour];
5411         idbuf[4] = QueueIdChars[tm->tm_min % QIC_LEN_R];
5412         idbuf[5] = QueueIdChars[tm->tm_sec % QIC_LEN_R];
5413         idbuf[6] = QueueIdChars[seq / QIC_LEN];
5414         idbuf[7] = QueueIdChars[seq % QIC_LEN];
5415         (void) sm_snprintf(&idbuf[8], sizeof(idbuf) - 8, "%06d",
5416                            (int) LastQueuePid);
5417         e->e_id = sm_rpool_strdup_x(e->e_rpool, idbuf);
5418         macdefine(&e->e_macro, A_PERM, 'i', e->e_id);
5419 #if 0
5420         /* XXX: inherited from MainEnvelope */
5421         e->e_qgrp = NOQGRP;  /* too early to do anything else */
5422         e->e_qdir = NOQDIR;
5423         e->e_xfqgrp = NOQGRP;
5424 #endif /* 0 */

5426         /* New ID means it's not on disk yet */
5427         e->e_qfletter = '\0';

5429         if (tTd(7, 1))
5430                 sm_dprintf("assign_queueid: assigned id %s, e=%p\n",
5431                         e->e_id, e);
5432         if (LogLevel > 93)
5433                 sm_syslog(LOG_DEBUG, e->e_id, "assigned id");
5434 }
5435 /*
5436 **  SYNC_QUEUE_TIME -- Assure exclusive PID in any given second
5437 **
5438 **      Make sure one PID can't be used by two processes in any one second.
5439 **
5440 **              If the system rotates PIDs fast enough, may get the
5441 **              same pid in the same second for two distinct processes.
5442 **              This will interfere with the queue file naming system.
5443 **
5444 **      Parameters:
5445 **              none
5446 **
5447 **      Returns:
5448 **              none
5449 */

5451 void
5452 sync_queue_time()
5453 {
5454 #if FAST_PID_RECYCLE
5455         if (OpMode != MD_TEST &&
5456             OpMode != MD_CHECKCONFIG &&
5457             OpMode != MD_VERIFY &&
5458             LastQueueTime > 0 &&
5459             LastQueuePid == CurrentPid &&
5460             curtime() == LastQueueTime)
5461                 (void) sleep(1);
5462 #endif /* FAST_PID_RECYCLE */
5463 }
5464 /*
5465 **  UNLOCKQUEUE -- unlock the queue entry for a specified envelope
5466 **
5467 **      Parameters:
5468 **              e -- the envelope to unlock.
5469 **
5470 **      Returns:
5471 **              none
5472 **
5473 **      Side Effects:
```



```
5474 **              unlocks the queue for 'e'.
5475 */

5477 void
5478 unlockqueue(e)
5479         ENVELOPE *e;
5480 {
5481         if (tTd(51, 4))
5482                 sm_dprintf("unlockqueue(%s)\n",
5483                         e->e_id == NULL ? "NOQUEUE" : e->e_id);


5486         /* if there is a lock file in the envelope, close it */
5487         if (e->e_lockfp != NULL)
5488                 (void) sm_io_close(e->e_lockfp, SM_TIME_DEFAULT);
5489         e->e_lockfp = NULL;

5491         /* don't create a queue id if we don't already have one */
5492         if (e->e_id == NULL)
5493                 return;

5495         /* remove the transcript */
5496         if (LogLevel > 87)
5497                 sm_syslog(LOG_DEBUG, e->e_id, "unlock");
5498         if (!tTd(51, 104))
5499                 (void) xunlink(queuename(e, XSCRPT_LETTER));
5500 }
5501 /*
5502 **  SETCTLUSER -- create a controlling address
5503 **
5504 **      Create a fake "address" given only a local login name; this is
5505 **      used as a "controlling user" for future recipient addresses.
5506 **
5507 **      Parameters:
5508 **              user -- the user name of the controlling user.
5509 **              qfver -- the version stamp of this queue file.
5510 **              e -- envelope
5511 **
5512 **      Returns:
5513 **              An address descriptor for the controlling user,
5514 **              using storage allocated from e->e_rpool.
5515 **
5516 */

5518 static ADDRESS *
5519 setctluser(user, qfver, e)
5520         char *user;
5521         int qfver;
5522         ENVELOPE *e;
5523 {
5524         register ADDRESS *a;
5525         struct passwd *pw;
5526         char *p;

5528         /*
5529         **  See if this clears our concept of controlling user.
5530         */

5532         if (user == NULL || *user == '\0')
5533                 return NULL;

5535         /*
5536         **  Set up addr fields for controlling user.
5537         */

5539         a = (ADDRESS *) sm_rpool_malloc_x(e->e_rpool, sizeof(*a));
```

```
5540         memset((char *) a, '\0', sizeof(*a));

5542         if (*user == ':')
5543         {
5544                 p = &user[1];
5545                 a->q_user = sm_rpool_strdup_x(e->e_rpool, p);
5546         }
5547         else
5548         {
5549                 p = strtok(user, ":");
5550                 a->q_user = sm_rpool_strdup_x(e->e_rpool, user);
5551                 if (qfver >= 2)
5552                 {
5553                         if ((p = strtok(NULL, ":")) != NULL)
5554                                 a->q_uid = atoi(p);
5555                         if ((p = strtok(NULL, ":")) != NULL)
5556                                 a->q_gid = atoi(p);
5557                         if ((p = strtok(NULL, ":")) != NULL)
5558                         {
5559                                 char *o;

5561                                 a->q_flags |= QGOODUID;

5563                                 /* if there is another ':': restore it */
5564                                 if ((o = strtok(NULL, ":")) != NULL && o > p)
5565                                         o[-1] = ':';
5566                         }
5567                 }
5568                 else if ((pw = sm_getpwnam(user)) != NULL)
5569                 {
5570                         if (*pw->pw_dir == '\0')
5571                                 a->q_home = NULL;
5572                         else if (strcmp(pw->pw_dir, "/") == 0)
5573                                 a->q_home = "";
5574                         else
5575                                 a->q_home = sm_rpool_strdup_x(e->e_rpool, pw->pw
5576                         a->q_uid = pw->pw_uid;
5577                         a->q_gid = pw->pw_gid;
5578                         a->q_flags |= QGOODUID;
5579                 }
5580         }

5582         a->q_flags |= QPRIMARY;         /* flag as a "ctladdr" */
5583         a->q_mailer = LocalMailer;
5584         if (p == NULL)
5585                 a->q_paddr = sm_rpool_strdup_x(e->e_rpool, a->q_user);
5586         else
5587                 a->q_paddr = sm_rpool_strdup_x(e->e_rpool, p);
5588         return a;
5589 }
5590 /*
5591 **  LOSEQFILE -- rename queue file with LOSEQF_LETTER & try to let someone know
5592 **
5593 **      Parameters:
5594 **              e -- the envelope (e->e_id will be used).
5595 **              why -- reported to whomever can hear.
5596 **
5597 **      Returns:
5598 **              none.
5599 */

5601 void
5602 loseqfile(e, why)
5603         register ENVELOPE *e;
5604         char *why;
5605 {
```



```
5606         bool loseit = true;
5607         char *p;
5608         char buf[MAXPATHLEN];

5610         if (e == NULL || e->e_id == NULL)
5611                 return;
5612         p = queuename(e, ANYQFL_LETTER);
5613         if (sm_strlcpy(buf, p, sizeof(buf)) >= sizeof(buf))
5614                 return;
5615         if (!bitset(EF_INQUEUE, e->e_flags))
5616                 queueup(e, false, true);
5617         else if (QueueMode == QM_LOST)
5618                 loseit = false;

5620         /* if already lost, no need to re-lose */
5621         if (loseit)
5622         {
5623                 p = queuename(e, LOSEQF_LETTER);
5624                 if (rename(buf, p) < 0)
5625                         syserr("cannot rename(%s, %s), uid=%d",
5626                                 buf, p, (int) geteuid());
5627                 else if (LogLevel > 0)
5628                         sm_syslog(LOG_ALERT, e->e_id,
5629                                 "Losing %s: %s", buf, why);
5630         }
5631         if (e->e_dfp != NULL)
5632         {
5633                 (void) sm_io_close(e->e_dfp, SM_TIME_DEFAULT);
5634                 e->e_dfp = NULL;
5635         }
5636         e->e_flags &= ~EF_HAS_DF;
5637 }
5638 /*
5639 **  NAME2QID -- translate a queue group name to a queue group id
5640 **
5641 **      Parameters:
5642 **              queuename -- name of queue group.
5643 **
5644 **      Returns:
5645 **              queue group id if found.
5646 **              NOQGRP otherwise.
5647 */

5649 int
5650 name2qid(queuename)
5651         char *queuename;
5652 {
5653         register STAB *s;

5655         s = stab(queuename, ST_QUEUE, ST_FIND);
5656         if (s == NULL)
5657                 return NOQGRP;
5658         return s->s_quegrp->qg_index;
5659 }
5660 /*
5661 **  QID_PRINTNAME -- create externally printable version of queue id
5662 **
5663 **      Parameters:
5664 **              e -- the envelope.
5665 **
5666 **      Returns:
5667 **              a printable version
5668 */

5670 char *
5671 qid_printname(e)
```

```
5672         ENVELOPE *e;
5673 {
5674         char *id;
5675         static char idbuf[MAXQFNAME + 34];

5677         if (e == NULL)
5678                 return "";

5680         if (e->e_id == NULL)
5681                 id = "";
5682         else
5683                 id = e->e_id;

5685         if (e->e_qdir == NOQDIR)
5686                 return id;

5688         (void) sm_snprintf(idbuf, sizeof(idbuf), "%.32s/%s",
5689                            Queue[e->e_qgrp]->qg_qpaths[e->e_qdir].qp_name,
5690                            id);
5691         return idbuf;
5692 }
5693 /*
5694 **  QID_PRINTQUEUE -- create full version of queue directory for data files
5695 **
5696 **      Parameters:
5697 **              qgrp -- index in queue group.
5698 **              qdir -- the short version of the queue directory
5699 **
5700 **      Returns:
5701 **              the full pathname to the queue (might point to a static var)
5702 */

5704 char *
5705 qid_printqueue(qgrp, qdir)
5706         int qgrp;
5707         int qdir;
5708 {
5709         char *subdir;
5710         static char dir[MAXPATHLEN];

5712         if (qdir == NOQDIR)
5713                 return Queue[qgrp]->qg_qdir;

5715         if (strcmp(Queue[qgrp]->qg_qpaths[qdir].qp_name, ".") == 0)
5716                 subdir = NULL;
5717         else
5718                 subdir = Queue[qgrp]->qg_qpaths[qdir].qp_name;

5720         (void) sm_strlcpyn(dir, sizeof(dir), 4,
5721                            Queue[qgrp]->qg_qdir,
5722                            subdir == NULL ? "" : "/",
5723                            subdir == NULL ? "" : subdir,
5724                            (bitset(QP_SUBDF,
5725                                    Queue[qgrp]->qg_qpaths[qdir].qp_subdirs)
5726                                            ? "/df" : ""));
5727         return dir;
5728 }

5730 /*
5731 **  PICKQDIR -- Pick a queue directory from a queue group
5732 **
5733 **      Parameters:
5734 **              qg -- queue group
5735 **              fsize -- file size in bytes
5736 **              e -- envelope, or NULL
5737 **
```



```
5738 **      Result:
5739 **              NOQDIR if no queue directory in qg has enough free space to
5740 **              hold a file of size 'fsize', otherwise the index of
5741 **              a randomly selected queue directory which resides on a
5742 **              file system with enough disk space.
5743 **              XXX This could be extended to select a queuedir with
5744 **                      a few (the fewest?) number of entries. That data
5745 **                      is available if shared memory is used.
5746 **
5747 **      Side Effects:
5748 **              If the request fails and e != NULL then sm_syslog is called.
5749 */

5751 int
5752 pickqdir(qg, fsize, e)
5753         QUEUEGRP *qg;
5754         long fsize;
5755         ENVELOPE *e;
5756 {
5757         int qdir;
5758         int i;
5759         long avail = 0;

5761         /* Pick a random directory, as a starting point. */
5762         if (qg->qg_numqueues <= 1)
5763                 qdir = 0;
5764         else
5765                 qdir = get_rand_mod(qg->qg_numqueues);

5767 #if _FFR_TESTS
5768         if (tTd(4, 101))
5769                 return NOQDIR;
5770 #endif /* _FFR_TESTS */
5771         if (MinBlocksFree <= 0 && fsize <= 0)
5772                 return qdir;

5774         /*
5775         **  Now iterate over the queue directories,
5776         **  looking for a directory with enough space for this message.
5777         */

5779         i = qdir;
5780         do
5781         {
5782                 QPATHS *qp = &qg->qg_qpaths[i];
5783                 long needed = 0;
5784                 long fsavail = 0;

5786                 if (fsize > 0)
5787                         needed += fsize / FILE_SYS_BLKSIZE(qp->qp_fsysidx)
5788                                   + ((fsize % FILE_SYS_BLKSIZE(qp->qp_fsysidx)
5789                                       > 0) ? 1 : 0);
5790                 if (MinBlocksFree > 0)
5791                         needed += MinBlocksFree;
5792                 fsavail = FILE_SYS_AVAIL(qp->qp_fsysidx);
5793 #if SM_CONF_SHM
5794                 if (fsavail <= 0)
5795                 {
5796                         long blksize;

5798                         /*
5799                         **  might be not correctly updated,
5800                         **  let's try to get the info directly.
5801                         */

5803                         fsavail = freediskspace(FILE_SYS_NAME(qp->qp_fsysidx),
```

```
                                                &blksize);
                        if (fsavail < 0)
                                fsavail = 0;
                }
#endif /* SM_CONF_SHM */
                if (needed <= fsavail)
                        return i;
                if (avail < fsavail)
                        avail = fsavail;

                if (qg->qg_numqueues > 0)
                        i = (i + 1) % qg->qg_numqueues;
        } while (i != qdir);

        if (e != NULL && LogLevel > 0)
                sm_syslog(LOG_ALERT, e->e_id,
                        "low on space (%s needs %ld bytes + %ld blocks in %s), m
                        CurHostName == NULL ? "SMTP-DAEMON" : CurHostName,
                        fsize, MinBlocksFree,
                        qg->qg_qdir, avail);
        return NOQDIR;
}
/*
**  SETNEWQUEUE -- Sets a new queue group and directory
**
**      Assign a queue group and directory to an envelope and store the
**      directory in e->e_qdir.
**
**      Parameters:
**              e -- envelope to assign a queue for.
**
**      Returns:
**              true if successful
**              false otherwise
**
**      Side Effects:
**              On success, e->e_qgrp and e->e_qdir are non-negative.
**              On failure (not enough disk space),
**              e->qgrp = NOQGRP, e->e_qdir = NOQDIR
**              and usrerr() is invoked (which could raise an exception).
*/

bool
setnewqueue(e)
        ENVELOPE *e;
{
        if (tTd(41, 20))
                sm_dprintf("setnewqueue: called\n");

        /* not set somewhere else */
        if (e->e_qgrp == NOQGRP)
        {
                ADDRESS *q;

                /*
                **  Use the queue group of the "first" recipient, as set by
                **  the "queuegroup" rule set.  If that is not defined, then
                **  use the queue group of the mailer of the first recipient.
                **  If that is not defined either, then use the default
                **  queue group.
                **  Notice: "first" depends on the sorting of sendqueue
                **  in recipient().
                **  To avoid problems with "bad" recipients look
                **  for a valid address first.
                */
```



```
                q = e->e_sendqueue;
                while (q != NULL &&
                       (QS_IS_BADADDR(q->q_state) || QS_IS_DEAD(q->q_state)))
                {
                        q = q->q_next;
                }
                if (q == NULL)
                        e->e_qgrp = 0;
                else if (q->q_qgrp >= 0)
                        e->e_qgrp = q->q_qgrp;
                else if (q->q_mailer != NULL &&
                         ISVALIDQGRP(q->q_mailer->m_qgrp))
                        e->e_qgrp = q->q_mailer->m_qgrp;
                else
                        e->e_qgrp = 0;
                e->e_dfqgrp = e->e_qgrp;
        }

        if (ISVALIDQDIR(e->e_qdir) && ISVALIDQDIR(e->e_dfqdir))
        {
                if (tTd(41, 20))
                        sm_dprintf("setnewqueue: e_qdir already assigned (%s)\n"
                                qid_printqueue(e->e_qgrp, e->e_qdir));
                return true;
        }

        filesys_update();
        e->e_qdir = pickqdir(Queue[e->e_qgrp], e->e_msgsize, e);
        if (e->e_qdir == NOQDIR)
        {
                e->e_qgrp = NOQGRP;
                if (!bitset(EF_FATALERRS, e->e_flags))
                        usrerr("452 4.4.5 Insufficient disk space; try again lat
                e->e_flags |= EF_FATALERRS;
                return false;
        }

        if (tTd(41, 3))
                sm_dprintf("setnewqueue: Assigned queue directory %s\n",
                        qid_printqueue(e->e_qgrp, e->e_qdir));

        if (e->e_xfqgrp == NOQGRP || e->e_xfqdir == NOQDIR)
        {
                e->e_xfqgrp = e->e_qgrp;
                e->e_xfqdir = e->e_qdir;
        }
        e->e_dfqdir = e->e_qdir;
        return true;
}
/*
**  CHKQDIR -- check a queue directory
**
**      Parameters:
**              name -- name of queue directory
**              sff -- flags for safefile()
**
**      Returns:
**              is it a queue directory?
*/

static bool chkqdir __P((char *, long));

static bool
chkqdir(name, sff)
        char *name;
        long sff;
```

```
{
        struct stat statb;
        int i;

        /* skip over . and .. directories */
        if (name[0] == '.' &&
            (name[1] == '\0' || (name[1] == '.' && name[2] == '\0')))
                return false;
#if HASLSTAT
        if (lstat(name, &statb) < 0)
#else /* HASLSTAT */
        if (stat(name, &statb) < 0)
#endif /* HASLSTAT */
        {
                if (tTd(41, 2))
                        sm_dprintf("chkqdir: stat(\"%s\"): %s\n",
                                   name, sm_errstring(errno));
                return false;
        }
#if HASLSTAT
        if (S_ISLNK(statb.st_mode))
        {
                /*
                **  For a symlink we need to make sure the
                **  target is a directory
                */

                if (stat(name, &statb) < 0)
                {
                        if (tTd(41, 2))
                                sm_dprintf("chkqdir: stat(\"%s\"): %s\n",
                                           name, sm_errstring(errno));
                        return false;
                }
        }
#endif /* HASLSTAT */

        if (!S_ISDIR(statb.st_mode))
        {
                if (tTd(41, 2))
                        sm_dprintf("chkqdir: \"%s\": Not a directory\n",
                                   name);
                return false;
        }

        /* Print a warning if unsafe (but still use it) */
        /* XXX do this only if we want the warning? */
        i = safedirpath(name, RunAsUid, RunAsGid, NULL, sff, 0, 0);
        if (i != 0)
        {
                if (tTd(41, 2))
                        sm_dprintf("chkqdir: \"%s\": Not safe: %s\n",
                                   name, sm_errstring(i));
#if _FFR_CHK_QUEUE
                if (LogLevel > 8)
                        sm_syslog(LOG_WARNING, NOQID,
                                  "queue directory \"%s\": Not safe: %s",
                                  name, sm_errstring(i));
#endif /* _FFR_CHK_QUEUE */
        }
        return true;
}
/*
**  MULTIQUEUE_CACHE -- cache a list of paths to queues.
**
**      Each potential queue is checked as the cache is built.
```



```
**      Thereafter, each is blindly trusted.
**      Note that we can be called again after a timeout to rebuild
**      (although code for that is not ready yet).
**
**      Parameters:
**              basedir -- base of all queue directories.
**              blen -- strlen(basedir).
**              qg -- queue group.
**              qn -- number of queue directories already cached.
**              phash -- pointer to hash value over queue dirs.
#if SM_CONF_SHM
**                      only used if shared memory is active.
#endif * SM_CONF_SHM *
**
**      Returns:
**              new number of queue directories.
*/

#define INITIAL_SLOTS   20
#define ADD_SLOTS       10

static int
multiqueue_cache(basedir, blen, qg, qn, phash)
        char *basedir;
        int blen;
        QUEUEGRP *qg;
        int qn;
        unsigned int *phash;
{
        char *cp;
        int i, len;
        int slotsleft = 0;
        long sff = SFF_ANYFILE;
        char qpath[MAXPATHLEN];
        char subdir[MAXPATHLEN];
        char prefix[MAXPATHLEN];        /* dir relative to basedir */

        if (tTd(41, 20))
                sm_dprintf("multiqueue_cache: called\n");

        /* Initialize to current directory */
        prefix[0] = '.';
        prefix[1] = '\0';
        if (qg->qg_numqueues != 0 && qg->qg_qpaths != NULL)
        {
                for (i = 0; i < qg->qg_numqueues; i++)
                {
                        if (qg->qg_qpaths[i].qp_name != NULL)
                                (void) sm_free(qg->qg_qpaths[i].qp_name); /* XXX
                }
                (void) sm_free((char *) qg->qg_qpaths); /* XXX */
                qg->qg_qpaths = NULL;
                qg->qg_numqueues = 0;
        }

        /* If running as root, allow safedirpath() checks to use privs */
        if (RunAsUid == 0)
                sff |= SFF_ROOTOK;
#if _FFR_CHK_QUEUE
        sff |= SFF_SAFEDIRPATH|SFF_NOWWFILES;
        if (!UseMSP)
                sff |= SFF_NOGWFILES;
#endif /* _FFR_CHK_QUEUE */

        if (!SM_IS_DIR_START(qg->qg_qdir))
        {
```

```
6068                /*
6069                **  XXX we could add basedir, but then we have to realloc()
6070                **  the string... Maybe another time.
6071                */

6073                syserr("QueuePath %s not absolute", qg->qg_qdir);
6074                ExitStat = EX_CONFIG;
6075                return qn;
6076        }

6078        /* qpath: directory of current workgroup */
6079        len = sm_strlcpy(qpath, qg->qg_qdir, sizeof(qpath));
6080        if (len >= sizeof(qpath))
6081        {
6082                syserr("QueuePath %.256s too long (%d max)",
6083                       qg->qg_qdir, (int) sizeof(qpath));
6084                ExitStat = EX_CONFIG;
6085                return qn;
6086        }

6088        /* begin of qpath must be same as basedir */
6089        if (strncmp(basedir, qpath, blen) != 0 &&
6090            (strncmp(basedir, qpath, blen - 1) != 0 || len != blen - 1))
6091        {
6092                syserr("QueuePath %s not subpath of QueueDirectory %s",
6093                       qpath, basedir);
6094                ExitStat = EX_CONFIG;
6095                return qn;
6096        }

6098        /* Do we have a nested subdirectory? */
6099        if (blen < len && SM_FIRST_DIR_DELIM(qg->qg_qdir + blen) != NULL)
6100        {

6102                /* Copy subdirectory into prefix for later use */
6103                if (sm_strlcpy(prefix, qg->qg_qdir + blen, sizeof(prefix)) >=
6104                    sizeof(prefix))
6105                {
6106                        syserr("QueuePath %.256s too long (%d max)",
6107                               qg->qg_qdir, (int) sizeof(qpath));
6108                        ExitStat = EX_CONFIG;
6109                        return qn;
6110                }
6111                cp = SM_LAST_DIR_DELIM(prefix);
6112                SM_ASSERT(cp != NULL);
6113                *cp = '\0';     /* cut off trailing / */
6114        }

6116        /* This is guaranteed by the basedir check above */
6117        SM_ASSERT(len >= blen - 1);
6118        cp = &qpath[len - 1];
6119        if (*cp == '*')
6120        {
6121                register DIR *dp;
6122                register struct dirent *d;
6123                int off;
6124                char *delim;
6125                char relpath[MAXPATHLEN];

6127                *cp = '\0';     /* Overwrite wildcard */
6128                if ((cp = SM_LAST_DIR_DELIM(qpath)) == NULL)
6129                {
6130                        syserr("QueueDirectory: can not wildcard relative path")
6131                        if (tTd(41, 2))
6132                                sm_dprintf("multiqueue_cache: \"%s*\": Can not w
6133                                           qpath);
```



```
6134                        ExitStat = EX_CONFIG;
6135                        return qn;
6136                }
6137                if (cp == qpath)
6138                {
6139                        /*
6140                        **  Special case of top level wildcard, like /foo*
6141                        **      Change to //foo*
6142                        */

6144                        (void) sm_strlcpy(qpath + 1, qpath, sizeof(qpath) - 1);
6145                        ++cp;
6146                }
6147                delim = cp;
6148                *(cp++) = '\0';         /* Replace / with \0 */
6149                len = strlen(cp);       /* Last component of queue directory */

6151                /*
6152                **  Path relative to basedir, with trailing /
6153                **  It will be modified below to specify the subdirectories
6154                **  so they can be opened without chdir().
6155                */

6157                off = sm_strlcpyn(relpath, sizeof(relpath), 2, prefix, "/");
6158                SM_ASSERT(off < sizeof(relpath));

6160                if (tTd(41, 2))
6161                        sm_dprintf("multiqueue_cache: prefix=\"%s%s\"\n",
6162                                   relpath, cp);

6164                /* It is always basedir: we don't need to store it per group */
6165                /* XXX: optimize this! -> one more global? */
6166                qg->qg_qdir = newstr(basedir);
6167                qg->qg_qdir[blen - 1] = '\0';   /* cut off trailing / */

6169                /*
6170                **  XXX Should probably wrap this whole loop in a timeout
6171                **  in case some wag decides to NFS mount the queues.
6172                */

6174                /* Test path to get warning messages. */
6175                if (qn == 0)
6176                {
6177                        /*  XXX qg_runasuid and qg_runasgid for specials? */
6178                        i = safedirpath(basedir, RunAsUid, RunAsGid, NULL,
6179                                        sff, 0, 0);
6180                        if (i != 0 && tTd(41, 2))
6181                                sm_dprintf("multiqueue_cache: \"%s\": Not safe:
6182                                           basedir, sm_errstring(i));
6183                }

6185                if ((dp = opendir(prefix)) == NULL)
6186                {
6187                        syserr("can not opendir(%s/%s)", qg->qg_qdir, prefix);
6188                        if (tTd(41, 2))
6189                                sm_dprintf("multiqueue_cache: opendir(\"%s/%s\")
6190                                           qg->qg_qdir, prefix,
6191                                           sm_errstring(errno));
6192                        ExitStat = EX_CONFIG;
6193                        return qn;
6194                }
6195                while ((d = readdir(dp)) != NULL)
6196                {
6197                        /* Skip . and .. directories */
6198                        if (strcmp(d->d_name, ".") == 0 ||
6199                            strcmp(d->d_name, "..") == 0)
```

```
6200                                    continue;

6202                            i = strlen(d->d_name);
6203                            if (i < len || strncmp(d->d_name, cp, len) != 0)
6204                            {
6205                                    if (tTd(41, 5))
6206                                            sm_dprintf("multiqueue_cache: \"%s\", sk
6207                                                    d->d_name);
6208                                    continue;
6209                            }

6211                            /* Create relative pathname: prefix + local directory */
6212                            i = sizeof(relpath) - off;
6213                            if (sm_strlcpy(relpath + off, d->d_name, i) >= i)
6214                                    continue;       /* way too long */

6216                            if (!chkqdir(relpath, sff))
6217                                    continue;

6219                            if (qg->qg_qpaths == NULL)
6220                            {
6221                                    slotsleft = INITIAL_SLOTS;
6222                                    qg->qg_qpaths = (QPATHS *)xalloc((sizeof(*qg->qg
6223                                                                    slotsleft);
6224                                    qg->qg_numqueues = 0;
6225                            }
6226                            else if (slotsleft < 1)
6227                            {
6228                                    qg->qg_qpaths = (QPATHS *)sm_realloc((char *)qg-
6229                                                    (sizeof(*qg->qg_qpaths
6230                                                    (qg->qg_numqueues +
6231                                                     ADD_SLOTS));
6232                                    if (qg->qg_qpaths == NULL)
6233                                    {
6234                                            (void) closedir(dp);
6235                                            return qn;
6236                                    }
6237                                    slotsleft += ADD_SLOTS;
6238                            }

6240                            /* check subdirs */
6241                            qg->qg_qpaths[qg->qg_numqueues].qp_subdirs = QP_NOSUB;

6243 #define CHKRSUBDIR(name, flag)  \
6244                 (void) sm_strlcpyn(subdir, sizeof(subdir), 3, relpath, "/", name); \
6245                 if (chkqdir(subdir, sff))       \
6246                         qg->qg_qpaths[qg->qg_numqueues].qp_subdirs |= flag;     \
6247                 else



6250                            CHKRSUBDIR("qf", QP_SUBQF);
6251                            CHKRSUBDIR("df", QP_SUBDF);
6252                            CHKRSUBDIR("xf", QP_SUBXF);

6254                            /* assert(strlen(d->d_name) < MAXPATHLEN - 14) */
6255                            /* maybe even - 17 (subdirs) */

6257                            if (prefix[0] != '.')
6258                                    qg->qg_qpaths[qg->qg_numqueues].qp_name =
6259                                            newstr(relpath);
6260                            else
6261                                    qg->qg_qpaths[qg->qg_numqueues].qp_name =
6262                                            newstr(d->d_name);

6264                            if (tTd(41, 2))
6265                                    sm_dprintf("multiqueue_cache: %d: \"%s\" cached
```



```
6266                                            qg->qg_numqueues, relpath,
6267                                            qg->qg_qpaths[qg->qg_numqueues].qp_subdi
6268 #if SM_CONF_SHM
6269                             qg->qg_qpaths[qg->qg_numqueues].qp_idx = qn;
6270                             *phash = hash_q(relpath, *phash);
6271 #endif /* SM_CONF_SHM */
6272                             qg->qg_numqueues++;
6273                             ++qn;
6274                             slotsleft--;
6275                     }
6276                     (void) closedir(dp);

6278                     /* undo damage */
6279                     *delim = '/';
6280             }
6281             if (qg->qg_numqueues == 0)
6282             {
6283                     qg->qg_qpaths = (QPATHS *) xalloc(sizeof(*qg->qg_qpaths));

6285                     /* test path to get warning messages */
6286                     i = safedirpath(qpath, RunAsUid, RunAsGid, NULL, sff, 0, 0);
6287                     if (i == ENOENT)
6288                     {
6289                             syserr("can not opendir(%s)", qpath);
6290                             if (tTd(41, 2))
6291                                     sm_dprintf("multiqueue_cache: opendir(\"%s\"): %
6292                                                     qpath, sm_errstring(i));
6293                             ExitStat = EX_CONFIG;
6294                             return qn;
6295                     }

6297                     qg->qg_qpaths[0].qp_subdirs = QP_NOSUB;
6298                     qg->qg_numqueues = 1;

6300                     /* check subdirs */
6301 #define CHKSUBDIR(name, flag)   \
6302             (void) sm_strlcpyn(subdir, sizeof(subdir), 3, qg->qg_qdir, "/", name); \
6303             if (chkqdir(subdir, sff))       \
6304                     qg->qg_qpaths[0].qp_subdirs |= flag;    \
6305             else

6307                     CHKSUBDIR("qf", QP_SUBQF);
6308                     CHKSUBDIR("df", QP_SUBDF);
6309                     CHKSUBDIR("xf", QP_SUBXF);

6311                     if (qg->qg_qdir[blen - 1] != '\0' &&
6312                         qg->qg_qdir[blen] != '\0')
6313                     {
6314                             /*
6315                             **  Copy the last component into qpaths and
6316                             **  cut off qdir
6317                             */

6319                             qg->qg_qpaths[0].qp_name = newstr(qg->qg_qdir + blen);
6320                             qg->qg_qdir[blen - 1] = '\0';
6321                     }
6322                     else
6323                             qg->qg_qpaths[0].qp_name = newstr(".");

6325 #if SM_CONF_SHM
6326                     qg->qg_qpaths[0].qp_idx = qn;
6327                     *phash = hash_q(qg->qg_qpaths[0].qp_name, *phash);
6328 #endif /* SM_CONF_SHM */
6329                     ++qn;
6330             }
6331             return qn;
```

```
}

/*
**  FILESYS_FIND -- find entry in FileSys table, or add new one
**
**      Given the pathname of a directory, determine the file system
**      in which that directory resides, and return a pointer to the
**      entry in the FileSys table that describes the file system.
**      A new entry is added if necessary (and requested).
**      If the directory does not exist, -1 is returned.
**
**      Parameters:
**              name -- name of directory (must be persistent!)
**              path -- pathname of directory (name plus maybe "/df")
**              add -- add to structure if not found.
**
**      Returns:
**              >=0: found: index in file system table
**              <0: some error, i.e.,
**              FSF_TOO_MANY: too many filesystems (-> syserr())
**              FSF_STAT_FAIL: can't stat() filesystem (-> syserr())
**              FSF_NOT_FOUND: not in list
*/

static short filesys_find __P((const char *, const char *, bool));

#define FSF_NOT_FOUND   (-1)
#define FSF_STAT_FAIL   (-2)
#define FSF_TOO_MANY    (-3)

static short
filesys_find(name, path, add)
        const char *name;
        const char *path;
        bool add;
{
        struct stat st;
        short i;

        if (stat(path, &st) < 0)
        {
                syserr("cannot stat queue directory %s", path);
                return FSF_STAT_FAIL;
        }
        for (i = 0; i < NumFileSys; ++i)
        {
                if (FILE_SYS_DEV(i) == st.st_dev)
                {
                        /*
                        **  Make sure the file system (FS) name is set:
                        **  even though the source code indicates that
                        **  FILE_SYS_DEV() is only set below, it could be
                        **  set via shared memory, hence we need to perform
                        **  this check/assignment here.
                        */

                        if (NULL == FILE_SYS_NAME(i))
                                FILE_SYS_NAME(i) = name;
                        return i;
                }
        }
        if (i >= MAXFILESYS)
        {
                syserr("too many queue file systems (%d max)", MAXFILESYS);
                return FSF_TOO_MANY;
        }
```



```
        if (!add)
                return FSF_NOT_FOUND;

        ++NumFileSys;
        FILE_SYS_NAME(i) = name;
        FILE_SYS_DEV(i) = st.st_dev;
        FILE_SYS_AVAIL(i) = 0;
        FILE_SYS_BLKSIZE(i) = 1024; /* avoid divide by zero */
        return i;
}

/*
**  FILESYS_SETUP -- set up mapping from queue directories to file systems
**
**      This data structure is used to efficiently check the amount of
**      free space available in a set of queue directories.
**
**      Parameters:
**              add -- initialize structure if necessary.
**
**      Returns:
**              0: success
**              <0: some error, i.e.,
**              FSF_NOT_FOUND: not in list
**              FSF_STAT_FAIL: can't stat() filesystem (-> syserr())
**              FSF_TOO_MANY: too many filesystems (-> syserr())
*/

static int filesys_setup __P((bool));

static int
filesys_setup(add)
        bool add;
{
        int i, j;
        short fs;
        int ret;

        ret = 0;
        for (i = 0; i < NumQueue && Queue[i] != NULL; i++)
        {
                for (j = 0; j < Queue[i]->qg_numqueues; ++j)
                {
                        QPATHS *qp = &Queue[i]->qg_qpaths[j];
                        char qddf[MAXPATHLEN];

                        (void) sm_strlcpyn(qddf, sizeof(qddf), 2, qp->qp_name,
                                           (bitset(QP_SUBDF, qp->qp_subdirs)
                                                   ? "/df" : ""));
                        fs = filesys_find(qp->qp_name, qddf, add);
                        if (fs >= 0)
                                qp->qp_fsysidx = fs;
                        else
                                qp->qp_fsysidx = 0;
                        if (fs < ret)
                                ret = fs;
                }
        }
        return ret;
}

/*
**  FILESYS_UPDATE -- update amount of free space on all file systems
**
**      The FileSys table is used to cache the amount of free space
**      available on all queue directory file systems.
```

```
**      This function updates the cached information if it has expired.
**
**      Parameters:
**              none.
**
**      Returns:
**              none.
**
**      Side Effects:
**              Updates FileSys table.
*/

void
filesys_update()
{
        int i;
        long avail, blksize;
        time_t now;
        static time_t nextupdate = 0;

#if SM_CONF_SHM
        /*
        **  Only the daemon updates the shared memory, i.e.,
        **  if shared memory is available but the pid is not the
        **  one of the daemon, then don't do anything.
        */

        if (ShmId != SM_SHM_NO_ID && DaemonPid != CurrentPid)
                return;
#endif /* SM_CONF_SHM */
        now = curtime();
        if (now < nextupdate)
                return;
        nextupdate = now + FILESYS_UPDATE_INTERVAL;
        for (i = 0; i < NumFileSys; ++i)
        {
                FILESYS *fs = &FILE_SYS(i);

                avail = freediskspace(FILE_SYS_NAME(i), &blksize);
                if (avail < 0 || blksize <= 0)
                {
                        if (LogLevel > 5)
                                sm_syslog(LOG_ERR, NOQID,
                                        "filesys_update failed: %s, fs=%s, avail
                                        sm_errstring(errno),
                                        FILE_SYS_NAME(i), avail, blksize);
                        fs->fs_avail = 0;
                        fs->fs_blksize = 1024; /* avoid divide by zero */
                        nextupdate = now + 2; /* let's do this soon again */
                }
                else
                {
                        fs->fs_avail = avail;
                        fs->fs_blksize = blksize;
                }
        }
}

#if _FFR_ANY_FREE_FS
/*
**  FILESYS_FREE -- check whether there is at least one fs with enough space.
**
**      Parameters:
**              fsize -- file size in bytes
**
**      Returns:
```



```
**              true iff there is one fs with more than fsize bytes free.
*/

bool
filesys_free(fsize)
        long fsize;
{
        int i;

        if (fsize <= 0)
                return true;
        for (i = 0; i < NumFileSys; ++i)
        {
                long needed = 0;

                if (FILE_SYS_AVAIL(i) < 0 || FILE_SYS_BLKSIZE(i) <= 0)
                        continue;
                needed += fsize / FILE_SYS_BLKSIZE(i)
                          + ((fsize % FILE_SYS_BLKSIZE(i)
                              > 0) ? 1 : 0)
                          + MinBlocksFree;
                if (needed <= FILE_SYS_AVAIL(i))
                        return true;
        }
        return false;
}
#endif /* _FFR_ANY_FREE_FS */

/*
**  DISK_STATUS -- show amount of free space in queue directories
**
**      Parameters:
**              out -- output file pointer.
**              prefix -- string to output in front of each line.
**
**      Returns:
**              none.
*/

void
disk_status(out, prefix)
        SM_FILE_T *out;
        char *prefix;
{
        int i;
        long avail, blksize;
        long free;

        for (i = 0; i < NumFileSys; ++i)
        {
                avail = freediskspace(FILE_SYS_NAME(i), &blksize);
                if (avail >= 0 && blksize > 0)
                {
                        free = (long)((double) avail *
                                ((double) blksize / 1024));
                }
                else
                        free = -1;
                (void) sm_io_fprintf(out, SM_TIME_DEFAULT,
                                "%s%d/%s/%ld\r\n",
                                prefix, i,
                                FILE_SYS_NAME(i),
                                        free);
        }
}
```

```
#if SM_CONF_SHM

/*
**  INIT_SEM -- initialize semaphore system
**
**      Parameters:
**              owner -- is this the owner of semaphores?
**
**      Returns:
**              none.
*/

#if _FFR_USE_SEM_LOCKING
#if SM_CONF_SEM
static int SemId = -1;          /* Semaphore Id */
int SemKey = SM_SEM_KEY;
#endif /* SM_CONF_SEM */
#endif /* _FFR_USE_SEM_LOCKING */

static void init_sem __P((bool));

static void
init_sem(owner)
        bool owner;
{
#if _FFR_USE_SEM_LOCKING
#if SM_CONF_SEM
        SemId = sm_sem_start(SemKey, 1, 0, owner);
        if (SemId < 0)
        {
                sm_syslog(LOG_ERR, NOQID,
                        "func=init_sem, sem_key=%ld, sm_sem_start=%d, error=%s",
                        (long) SemKey, SemId, sm_errstring(-SemId));
                return;
        }
        if (owner && RunAsUid != 0)
        {
                int r;

                r = sm_semsetowner(SemId, RunAsUid, RunAsGid, 0660);
                if (r != 0)
                        sm_syslog(LOG_ERR, NOQID,
                                "key=%ld, sm_semsetowner=%d, RunAsUid=%d, RunAsG
                                (long) SemKey, r, RunAsUid, RunAsGid);
        }
#endif /* SM_CONF_SEM */
#endif /* _FFR_USE_SEM_LOCKING */
        return;
}

/*
**  STOP_SEM -- stop semaphore system
**
**      Parameters:
**              owner -- is this the owner of semaphores?
**
**      Returns:
**              none.
*/

static void stop_sem __P((bool));

static void
stop_sem(owner)
        bool owner;
{
```



```
#if _FFR_USE_SEM_LOCKING
#if SM_CONF_SEM
        if (owner && SemId >= 0)
                sm_sem_stop(SemId);
#endif /* SM_CONF_SEM */
#endif /* _FFR_USE_SEM_LOCKING */
        return;
}

/*
**  UPD_QS -- update information about queue when adding/deleting an entry
**
**      Parameters:
**              e -- envelope.
**              count -- add/remove entry (+1/0/-1: add/no change/remove)
**              space -- update the space available as well.
**                      (>0/0/<0: add/no change/remove)
**              where -- caller (for logging)
**
**      Returns:
**              none.
**
**      Side Effects:
**              Modifies available space in filesystem.
**              Changes number of entries in queue directory.
*/

void
upd_qs(e, count, space, where)
        ENVELOPE *e;
        int count;
        int space;
        char *where;
{
        short fidx;
        int idx;
# if _FFR_USE_SEM_LOCKING
        int r;
# endif /* _FFR_USE_SEM_LOCKING */
        long s;

        if (ShmId == SM_SHM_NO_ID || e == NULL)
                return;
        if (e->e_qgrp == NOQGRP || e->e_qdir == NOQDIR)
                return;
        idx = Queue[e->e_qgrp]->qg_qpaths[e->e_qdir].qp_idx;
        if (tTd(73,2))
                sm_dprintf("func=upd_qs, count=%d, space=%d, where=%s, idx=%d, e
                        count, space, where, idx, QSHM_ENTRIES(idx));

        /* XXX in theory this needs to be protected with a mutex */
        if (QSHM_ENTRIES(idx) >= 0 && count != 0)
        {
# if _FFR_USE_SEM_LOCKING
                r = sm_sem_acq(SemId, 0, 1);
# endif /* _FFR_USE_SEM_LOCKING */
                QSHM_ENTRIES(idx) += count;
# if _FFR_USE_SEM_LOCKING
                if (r >= 0)
                        r = sm_sem_rel(SemId, 0, 1);
# endif /* _FFR_USE_SEM_LOCKING */
        }

        fidx = Queue[e->e_qgrp]->qg_qpaths[e->e_qdir].qp_fsysidx;
        if (fidx < 0)
                return;
```

```
6729         /* update available space also?  (might be loseqfile) */
6730         if (space == 0)
6731                 return;

6733         /* convert size to blocks; this causes rounding errors */
6734         s = e->e_msgsize / FILE_SYS_BLKSIZE(fidx);
6735         if (s == 0)
6736                 return;

6738         /* XXX in theory this needs to be protected with a mutex */
6739         if (space > 0)
6740                 FILE_SYS_AVAIL(fidx) += s;
6741         else
6742                 FILE_SYS_AVAIL(fidx) -= s;

6744 }

6746 static bool write_key_file __P((char *, long));
6747 static long read_key_file __P((char *, long));

6749 /*
6750 **  WRITE_KEY_FILE -- record some key into a file.
6751 **
6752 **      Parameters:
6753 **              keypath -- file name.
6754 **              key -- key to write.
6755 **
6756 **      Returns:
6757 **              true iff file could be written.
6758 **
6759 **      Side Effects:
6760 **              writes file.
6761 */

6763 static bool
6764 write_key_file(keypath, key)
6765         char *keypath;
6766         long key;
6767 {
6768         bool ok;
6769         long sff;
6770         SM_FILE_T *keyf;

6772         ok = false;
6773         if (keypath == NULL || *keypath == '\0')
6774                 return ok;
6775         sff = SFF_NOLINK|SFF_ROOTOK|SFF_REGONLY|SFF_CREAT;
6776         if (TrustedUid != 0 && RealUid == TrustedUid)
6777                 sff |= SFF_OPENASROOT;
6778         keyf = safefopen(keypath, O_WRONLY|O_TRUNC, FileMode, sff);
6779         if (keyf == NULL)
6780         {
6781                 sm_syslog(LOG_ERR, NOQID, "unable to write %s: %s",
6782                           keypath, sm_errstring(errno));
6783         }
6784         else
6785         {
6786                 if (geteuid() == 0 && RunAsUid != 0)
6787                 {
6788 #  if HASFCHOWN
6789                         int fd;

6791                         fd = keyf->f_file;
6792                         if (fd >= 0 && fchown(fd, RunAsUid, -1) < 0)
6793                         {
```



```
6794                                 int err = errno;

6796                                 sm_syslog(LOG_ALERT, NOQID,
6797                                           "ownership change on %s to %d failed:
6798                                           keypath, RunAsUid, sm_errstring(err));
6799                         }
6800 #  endif /* HASFCHOWN */
6801                 }
6802                 ok = sm_io_fprintf(keyf, SM_TIME_DEFAULT, "%ld\n", key) !=
6803                     SM_IO_EOF;
6804                 ok = (sm_io_close(keyf, SM_TIME_DEFAULT) != SM_IO_EOF) && ok;
6805         }
6806         return ok;
6807 }

6809 /*
6810 **  READ_KEY_FILE -- read a key from a file.
6811 **
6812 **      Parameters:
6813 **              keypath -- file name.
6814 **              key -- default key.
6815 **
6816 **      Returns:
6817 **              key.
6818 */

6820 static long
6821 read_key_file(keypath, key)
6822         char *keypath;
6823         long key;
6824 {
6825         int r;
6826         long sff, n;
6827         SM_FILE_T *keyf;

6829         if (keypath == NULL || *keypath == '\0')
6830                 return key;
6831         sff = SFF_NOLINK|SFF_ROOTOK|SFF_REGONLY;
6832         if (RealUid == 0 || (TrustedUid != 0 && RealUid == TrustedUid))
6833                 sff |= SFF_OPENASROOT;
6834         keyf = safefopen(keypath, O_RDONLY, FileMode, sff);
6835         if (keyf == NULL)
6836         {
6837                 sm_syslog(LOG_ERR, NOQID, "unable to read %s: %s",
6838                           keypath, sm_errstring(errno));
6839         }
6840         else
6841         {
6842                 r = sm_io_fscanf(keyf, SM_TIME_DEFAULT, "%ld", &n);
6843                 if (r == 1)
6844                         key = n;
6845                 (void) sm_io_close(keyf, SM_TIME_DEFAULT);
6846         }
6847         return key;
6848 }

6850 /*
6851 **  INIT_SHM -- initialize shared memory structure
6852 **
6853 **      Initialize or attach to shared memory segment.
6854 **      Currently it is not a fatal error if this doesn't work.
6855 **      However, it causes us to have a "fallback" storage location
6856 **      for everything that is supposed to be in the shared memory,
6857 **      which makes the code slightly ugly.
6858 **
6859 **      Parameters:
```

```
6860 **              qn -- number of queue directories.
6861 **              owner -- owner of shared memory.
6862 **              hash -- identifies data that is stored in shared memory.
6863 **
6864 **      Returns:
6865 **              none.
6866 */

6868 static void init_shm __P((int, bool, unsigned int));

6870 static void
6871 init_shm(qn, owner, hash)
6872         int qn;
6873         bool owner;
6874         unsigned int hash;
6875 {
6876         int i;
6877         int count;
6878         int save_errno;
6879         bool keyselect;

6881         PtrFileSys = &FileSys[0];
6882         PNumFileSys = &Numfilesys;
6883 /* if this "key" is specified: select one yourself */
6884 #define SEL_SHM_KEY     ((key_t) -1)
6885 #define FIRST_SHM_KEY   25

6887         /* This allows us to disable shared memory at runtime. */
6888         if (ShmKey == 0)
6889                 return;

6891         count = 0;
6892         shms = SM_T_SIZE + qn * sizeof(QUEUE_SHM_T);
6893         keyselect = ShmKey == SEL_SHM_KEY;
6894         if (keyselect)
6895         {
6896                 if (owner)
6897                         ShmKey = FIRST_SHM_KEY;
6898                 else
6899                 {
6900                         errno = 0;
6901                         ShmKey = read_key_file(ShmKeyFile, ShmKey);
6902                         keyselect = false;
6903                         if (ShmKey == SEL_SHM_KEY)
6904                         {
6905                                 save_errno = (errno != 0) ? errno : EINVAL;
6906                                 goto error;
6907                         }
6908                 }
6909         }
6910         for (;;)
6911         {
6912                 /* allow read/write access for group? */
6913                 Pshm = sm_shmstart(ShmKey, shms,
6914                                 SHM_R|SHM_W|(SHM_R>>3)|(SHM_W>>3),
6915                                 &ShmId, owner);
6916                 save_errno = errno;
6917                 if (Pshm != NULL || !sm_file_exists(save_errno))
6918                         break;
6919                 if (++count >= 3)
6920                 {
6921                         if (keyselect)
6922                         {
6923                                 ++ShmKey;

6925                                 /* back where we started? */
```



```
6926                                 if (ShmKey == SEL_SHM_KEY)
6927                                         break;
6928                                 continue;
6929                         }
6930                         break;
6931                 }

6933                 /* only sleep if we are at the first key */
6934                 if (!keyselect || ShmKey == SEL_SHM_KEY)
6935                         sleep(count);
6936         }
6937         if (Pshm != NULL)
6938         {
6939                 int *p;

6941                 if (keyselect)
6942                         (void) write_key_file(ShmKeyFile, (long) ShmKey);
6943                 if (owner && RunAsUid != 0)
6944                 {
6945                         i = sm_shmsetowner(ShmId, RunAsUid, RunAsGid, 0660);
6946                         if (i != 0)
6947                                 sm_syslog(LOG_ERR, NOQID,
6948                                         "key=%ld, sm_shmsetowner=%d, RunAsUid=%d
6949                                         (long) ShmKey, i, RunAsUid, RunAsGid);
6950                 }
6951                 p = (int *) Pshm;
6952                 if (owner)
6953                 {
6954                         *p = (int) shms;
6955                         *((pid_t *) SHM_OFF_PID(Pshm)) = CurrentPid;
6956                         p = (int *) SHM_OFF_TAG(Pshm);
6957                         *p = hash;
6958                 }
6959                 else
6960                 {
6961                         if (*p != (int) shms)
6962                         {
6963                                 save_errno = EINVAL;
6964                                 cleanup_shm(false);
6965                                 goto error;
6966                         }
6967                         p = (int *) SHM_OFF_TAG(Pshm);
6968                         if (*p != (int) hash)
6969                         {
6970                                 save_errno = EINVAL;
6971                                 cleanup_shm(false);
6972                                 goto error;
6973                         }

6975                         /*
6976                         **  XXX how to check the pid?
6977                         **  Read it from the pid-file? That does
6978                         **  not need to exist.
6979                         **  We could disable shm if we can't confirm
6980                         **  that it is the right one.
6981                         */
6982                 }

6984                 PtrFileSys = (FILESYS *) OFF_FILE_SYS(Pshm);
6985                 PNumFileSys = (int *) OFF_NUM_FILE_SYS(Pshm);
6986                 QShm = (QUEUE_SHM_T *) OFF_QUEUE_SHM(Pshm);
6987                 PRSATmpCnt = (int *) OFF_RSA_TMP_CNT(Pshm);
6988                 *PRSATmpCnt = 0;
6989                 if (owner)
6990                 {
6991                         /* initialize values in shared memory */
```

```
6992                        NumFileSys = 0;
6993                        for (i = 0; i < qn; i++)
6994                                QShm[i].qs_entries = -1;
6995                }
6996                init_sem(owner);
6997                return;
6998        }
6999   error:
7000        if (LogLevel > (owner ? 8 : 11))
7001        {
7002                sm_syslog(owner ? LOG_ERR : LOG_NOTICE, NOQID,
7003                          "can't %s shared memory, key=%ld: %s",
7004                          owner ? "initialize" : "attach to",
7005                          (long) ShmKey, sm_errstring(save_errno));
7006        }
7007 }
7008 #endif /* SM_CONF_SHM */


7011 /*
7012 **  SETUP_QUEUES -- set up all queue groups
7013 **
7014 **      Parameters:
7015 **              owner -- owner of shared memory?
7016 **
7017 **      Returns:
7018 **              none.
7019 **
7020 #if SM_CONF_SHM
7021 **      Side Effects:
7022 **              attaches shared memory.
7023 #endif * SM_CONF_SHM *
7024 */

7026 void
7027 setup_queues(owner)
7028        bool owner;
7029 {
7030        int i, qn, len;
7031        unsigned int hashval;
7032        time_t now;
7033        char basedir[MAXPATHLEN];
7034        struct stat st;

7036        /*
7037        **  Determine basedir for all queue directories.
7038        **  All queue directories must be (first level) subdirectories
7039        **  of the basedir.  The basedir is the QueueDir
7040        **  without wildcards, but with trailing /
7041        */

7043        hashval = 0;
7044        errno = 0;
7045        len = sm_strlcpy(basedir, QueueDir, sizeof(basedir));

7047        /* Provide space for trailing '/' */
7048        if (len >= sizeof(basedir) - 1)
7049        {
7050                syserr("QueueDirectory: path too long: %d,  max %d",
7051                       len, (int) sizeof(basedir) - 1);
7052                ExitStat = EX_CONFIG;
7053                return;
7054        }
7055        SM_ASSERT(len > 0);
7056        if (basedir[len - 1] == '*')
7057        {
```



```
7058                char *cp;

7060                cp = SM_LAST_DIR_DELIM(basedir);
7061                if (cp == NULL)
7062                {
7063                        syserr("QueueDirectory: can not wildcard relative path \
7064                                QueueDir);
7065                        if (tTd(41, 2))
7066                                sm_dprintf("setup_queues: \"%s\": Can not wildca
7067                                        QueueDir);
7068                        ExitStat = EX_CONFIG;
7069                        return;
7070                }

7072                /* cut off wildcard pattern */
7073                *++cp = '\0';
7074                len = cp - basedir;
7075        }
7076        else if (!SM_IS_DIR_DELIM(basedir[len - 1]))
7077        {
7078                /* append trailing slash since it is a directory */
7079                basedir[len] = '/';
7080                basedir[++len] = '\0';
7081        }

7083        /* len counts up to the last directory delimiter */
7084        SM_ASSERT(basedir[len - 1] == '/');

7086        if (chdir(basedir) < 0)
7087        {
7088                int save_errno = errno;

7090                syserr("can not chdir(%s)", basedir);
7091                if (save_errno == EACCES)
7092                        (void) sm_io_fprintf(smioerr, SM_TIME_DEFAULT,
7093                                "Program mode requires special privileges, e.g.,
7094                if (tTd(41, 2))
7095                        sm_dprintf("setup_queues: \"%s\": %s\n",
7096                                basedir, sm_errstring(errno));
7097                ExitStat = EX_CONFIG;
7098                return;
7099        }
7100 #if SM_CONF_SHM
7101        hashval = hash_q(basedir, hashval);
7102 #endif /* SM_CONF_SHM */

7104        /* initialize for queue runs */
7105        DoQueueRun = false;
7106        now = curtime();
7107        for (i = 0; i < NumQueue && Queue[i] != NULL; i++)
7108                Queue[i]->qg_nextrun = now;


7111        if (UseMSP && OpMode != MD_TEST)
7112        {
7113                long sff = SFF_CREAT;

7115                if (stat(".", &st) < 0)
7116                {
7117                        syserr("can not stat(%s)", basedir);
7118                        if (tTd(41, 2))
7119                                sm_dprintf("setup_queues: \"%s\": %s\n",
7120                                        basedir, sm_errstring(errno));
7121                        ExitStat = EX_CONFIG;
7122                        return;
7123                }
```

```
7124                 if (RunAsUid == 0)
7125                         sff |= SFF_ROOTOK;
7126
7127                 /*
7128                 **  Check queue directory permissions.
7129                 **      Can we write to a group writable queue directory?
7130                 */
7131
7132                 if (bitset(S_IWGRP, QueueFileMode) &&
7133                     bitset(S_IWGRP, st.st_mode) &&
7134                     safefile(" ", RunAsUid, RunAsGid, RunAsUserName, sff,
7135                              QueueFileMode, NULL) != 0)
7136                 {
7137                         syserr("can not write to queue directory %s (RunAsGid=%d
7138                                basedir, (int) RunAsGid, (int) st.st_gid);
7139                 }
7140                 if (bitset(S_IWOTH|S_IXOTH, st.st_mode))
7141                 {
7142 #if _FFR_MSP_PARANOIA
7143                         syserr("dangerous permissions=%o on queue directory %s",
7144                                (int) st.st_mode, basedir);
7145 #else /* _FFR_MSP_PARANOIA */
7146                         if (LogLevel > 0)
7147                                 sm_syslog(LOG_ERR, NOQID,
7148                                           "dangerous permissions=%o on queue dir
7149                                           (int) st.st_mode, basedir);
7150 #endif /* _FFR_MSP_PARANOIA */
7151                 }
7152 #if _FFR_MSP_PARANOIA
7153                 if (NumQueue > 1)
7154                         syserr("can not use multiple queues for MSP");
7155 #endif /* _FFR_MSP_PARANOIA */
7156         }
7157
7158         /* initial number of queue directories */
7159         qn = 0;
7160         for (i = 0; i < NumQueue && Queue[i] != NULL; i++)
7161                 qn = multiqueue_cache(basedir, len, Queue[i], qn, &hashval);
7162
7163 #if SM_CONF_SHM
7164         init_shm(qn, owner, hashval);
7165         i = filesys_setup(owner || ShmId == SM_SHM_NO_ID);
7166         if (i == FSF_NOT_FOUND)
7167         {
7168                 /*
7169                 **  We didn't get the right filesystem data
7170                 **  This may happen if we don't have the right shared memory.
7171                 **  So let's do this without shared memory.
7172                 */
7173
7174                 SM_ASSERT(!owner);
7175                 cleanup_shm(false);     /* release shared memory */
7176                 i = filesys_setup(false);
7177                 if (i < 0)
7178                         syserr("filesys_setup failed twice, result=%d", i);
7179                 else if (LogLevel > 8)
7180                         sm_syslog(LOG_WARNING, NOQID,
7181                                   "shared memory does not contain expected data,
7182         }
7183 #else /* SM_CONF_SHM */
7184         i = filesys_setup(true);
7185 #endif /* SM_CONF_SHM */
7186         if (i < 0)
7187                 ExitStat = EX_CONFIG;
7188 }
```



```
7190 #if SM_CONF_SHM
7191 /*
7192 **  CLEANUP_SHM -- do some cleanup work for shared memory etc
7193 **
7194 **      Parameters:
7195 **              owner -- owner of shared memory?
7196 **
7197 **      Returns:
7198 **              none.
7199 **
7200 **      Side Effects:
7201 **              detaches shared memory.
7202 */
7203
7204 void
7205 cleanup_shm(owner)
7206         bool owner;
7207 {
7208         if (ShmId != SM_SHM_NO_ID)
7209         {
7210                 if (sm_shmstop(Pshm, ShmId, owner) < 0 && LogLevel > 8)
7211                         sm_syslog(LOG_INFO, NOQID, "sm_shmstop failed=%s",
7212                                   sm_errstring(errno));
7213                 Pshm = NULL;
7214                 ShmId = SM_SHM_NO_ID;
7215         }
7216         stop_sem(owner);
7217 }
7218 #endif /* SM_CONF_SHM */
7219
7220 /*
7221 **  CLEANUP_QUEUES -- do some cleanup work for queues
7222 **
7223 **      Parameters:
7224 **              none.
7225 **
7226 **      Returns:
7227 **              none.
7228 **
7229 */
7230
7231 void
7232 cleanup_queues()
7233 {
7234         sync_queue_time();
7235 }
7236 /*
7237 **  SET_DEF_QUEUEVAL -- set default values for a queue group.
7238 **
7239 **      Parameters:
7240 **              qg -- queue group
7241 **              all -- set all values (true for default group)?
7242 **
7243 **      Returns:
7244 **              none.
7245 **
7246 **      Side Effects:
7247 **              sets default values for the queue group.
7248 */
7249
7250 void
7251 set_def_queueval(qg, all)
7252         QUEUEGRP *qg;
7253         bool all;
7254 {
7255         if (bitnset(QD_DEFINED, qg->qg_flags))
```

```
7256                return;
7257        if (all)
7258                qg->qg_qdir = QueueDir;
7259 #if _FFR_QUEUE_GROUP_SORTORDER
7260        qg->qg_sortorder = QueueSortOrder;
7261 #endif /* _FFR_QUEUE_GROUP_SORTORDER */
7262        qg->qg_maxqrun = all ? MaxRunnersPerQueue : -1;
7263        qg->qg_nice = NiceQueueRun;
7264 }
7265 /*
7266 **  MAKEQUEUE -- define a new queue.
7267 **
7268 **      Parameters:
7269 **              line -- description of queue.  This is in labeled fields.
7270 **                      The fields are:
7271 **                         F -- the flags associated with the queue
7272 **                         I -- the interval between running the queue
7273 **                         J -- the maximum # of jobs in work list
7274 **                         [M -- the maximum # of jobs in a queue run]
7275 **                         N -- the niceness at which to run
7276 **                         P -- the path to the queue
7277 **                         S -- the queue sorting order
7278 **                         R -- number of parallel queue runners
7279 **                         r -- max recipients per envelope
7280 **                      The first word is the canonical name of the queue.
7281 **              qdef -- this is a 'Q' definition from .cf
7282 **
7283 **      Returns:
7284 **              none.
7285 **
7286 **      Side Effects:
7287 **              enters the queue into the queue table.
7288 */

7290 void
7291 makequeue(line, qdef)
7292        char *line;
7293        bool qdef;
7294 {
7295        register char *p;
7296        register QUEUEGRP *qg;
7297        register STAB *s;
7298        int i;
7299        char fcode;

7301        /* allocate a queue and set up defaults */
7302        qg = (QUEUEGRP *) xalloc(sizeof(*qg));
7303        memset((char *) qg, '\0', sizeof(*qg));

7305        if (line[0] == '\0')
7306        {
7307                syserr("name required for queue");
7308                return;
7309        }

7311        /* collect the queue name */
7312        for (p = line;
7313             *p != '\0' && *p != ',' && !(isascii(*p) && isspace(*p));
7314             p++)
7315                continue;
7316        if (*p != '\0')
7317                *p++ = '\0';
7318        qg->qg_name = newstr(line);

7320        /* set default values, can be overridden below */
7321        set_def_queueval(qg, false);
```



```
7323        /* now scan through and assign info from the fields */
7324        while (*p != '\0')
7325        {
7326                auto char *delimptr;

7328                while (*p != '\0' &&
7329                       (*p == ',' || (isascii(*p) && isspace(*p))))
7330                        p++;

7332                /* p now points to field code */
7333                fcode = *p;
7334                while (*p != '\0' && *p != '=' && *p != ',')
7335                        p++;
7336                if (*p++ != '=')
7337                {
7338                        syserr("queue %s: `=' expected", qg->qg_name);
7339                        return;
7340                }
7341                while (isascii(*p) && isspace(*p))
7342                        p++;

7344                /* p now points to the field body */
7345                p = munchstring(p, &delimptr, ',');

7347                /* install the field into the queue struct */
7348                switch (fcode)
7349                {
7350                  case 'P':             /* pathname */
7351                        if (*p == '\0')
7352                                syserr("queue %s: empty path name",
7353                                        qg->qg_name);
7354                        else
7355                                qg->qg_qdir = newstr(p);
7356                        break;

7358                  case 'F':             /* flags */
7359                        for (; *p != '\0'; p++)
7360                                if (!(isascii(*p) && isspace(*p)))
7361                                        setbitn(*p, qg->qg_flags);
7362                        break;

7364                        /*
7365                        **  Do we need two intervals here:
7366                        **  One for persistent queue runners,
7367                        **  one for "normal" queue runs?
7368                        */

7370                  case 'I':     /* interval between running the queue */
7371                        qg->qg_queueintvl = convtime(p, 'm');
7372                        break;

7374                  case 'N':             /* run niceness */
7375                        qg->qg_nice = atoi(p);
7376                        break;

7378                  case 'R':             /* maximum # of runners for the group */
7379                        i = atoi(p);

7381                        /* can't have more runners than allowed total */
7382                        if (MaxQueueChildren > 0 && i > MaxQueueChildren)
7383                        {
7384                                qg->qg_maxqrun = MaxQueueChildren;
7385                                (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
7386                                                     "Q=%s: R=%d exceeds MaxQueu
7387                                                     qg->qg_name, i,
```

```
7388                                                    MaxQueueChildren);
7389                          }
7390                          else
7391                                qg->qg_maxqrun = i;
7392                          break;

7394                    case 'J':             /* maximum # of jobs in work list */
7395                          qg->qg_maxlist = atoi(p);
7396                          break;

7398                    case 'r':             /* max recipients per envelope */
7399                          qg->qg_maxrcpt = atoi(p);
7400                          break;

7402 #if _FFR_QUEUE_GROUP_SORTORDER
7403                    case 'S':             /* queue sorting order */
7404                          switch (*p)
7405                          {
7406                            case 'h':     /* Host first */
7407                            case 'H':
7408                                  qg->qg_sortorder = QSO_BYHOST;
7409                                  break;

7411                            case 'p':     /* Priority order */
7412                            case 'P':
7413                                  qg->qg_sortorder = QSO_BYPRIORITY;
7414                                  break;

7416                            case 't':     /* Submission time */
7417                            case 'T':
7418                                  qg->qg_sortorder = QSO_BYTIME;
7419                                  break;

7421                            case 'f':     /* File name */
7422                            case 'F':
7423                                  qg->qg_sortorder = QSO_BYFILENAME;
7424                                  break;

7426                            case 'm':     /* Modification time */
7427                            case 'M':
7428                                  qg->qg_sortorder = QSO_BYMODTIME;
7429                                  break;

7431                            case 'r':     /* Random */
7432                            case 'R':
7433                                  qg->qg_sortorder = QSO_RANDOM;
7434                                  break;

7436 # if _FFR_RHS
7437                            case 's':     /* Shuffled host name */
7438                            case 'S':
7439                                  qg->qg_sortorder = QSO_BYSHUFFLE;
7440                                  break;
7441 # endif /* _FFR_RHS */

7443                            case 'n':     /* none */
7444                            case 'N':
7445                                  qg->qg_sortorder = QSO_NONE;
7446                                  break;

7448                            default:
7449                                  syserr("Invalid queue sort order \"%s\"", p);
7450                          }
7451                          break;
7452 #endif /* _FFR_QUEUE_GROUP_SORTORDER */
```



```
7454                    default:
7455                          syserr("Q%s: unknown queue equate %c=",
7456                                 qg->qg_name, fcode);
7457                          break;
7458                  }

7460                  p = delimptr;
7461          }

7463 #if !HASNICE
7464          if (qg->qg_nice != NiceQueueRun)
7465          {
7466                  (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
7467                                       "Q%s: Warning: N= set on system that doesn'
7468                                       qg->qg_name);
7469          }
7470 #endif /* !HASNICE */

7472          /* do some rationality checking */
7473          if (NumQueue >= MAXQUEUEGROUPS)
7474          {
7475                  syserr("too many queue groups defined (%d max)",
7476                          MAXQUEUEGROUPS);
7477                  return;
7478          }

7480          if (qg->qg_qdir == NULL)
7481          {
7482                  if (QueueDir == NULL || *QueueDir == '\0')
7483                  {
7484                          syserr("QueueDir must be defined before queue groups");
7485                          return;
7486                  }
7487                  qg->qg_qdir = newstr(QueueDir);
7488          }

7490          if (qg->qg_maxqrun > 1 && !bitnset(QD_FORK, qg->qg_flags))
7491          {
7492                  (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
7493                                       "Warning: Q=%s: R=%d: multiple queue runner
7494                                       qg->qg_name, qg->qg_maxqrun, QD_FORK);
7495          }

7497          /* enter the queue into the symbol table */
7498          if (tTd(37, 8))
7499                  sm_syslog(LOG_INFO, NOQID,
7500                            "Adding %s to stab, path: %s", qg->qg_name,
7501                            qg->qg_qdir);
7502          s = stab(qg->qg_name, ST_QUEUE, ST_ENTER);
7503          if (s->s_quegrp != NULL)
7504          {
7505                  i = s->s_quegrp->qg_index;

7507                  /* XXX what about the pointers inside this struct? */
7508                  sm_free(s->s_quegrp); /* XXX */
7509          }
7510          else
7511                  i = NumQueue++;
7512          Queue[i] = s->s_quegrp = qg;
7513          qg->qg_index = i;

7515          /* set default value for max queue runners */
7516          if (qg->qg_maxqrun < 0)
7517          {
7518                  if (MaxRunnersPerQueue > 0)
7519                          qg->qg_maxqrun = MaxRunnersPerQueue;
```

```
7520                 else
7521                         qg->qg_maxqrun = 1;
7522         }
7523         if (qdef)
7524                 setbitn(QD_DEFINED, qg->qg_flags);
7525 }
7526 #if 0
7527 /*
7528 **  HASHFQN -- calculate a hash value for a fully qualified host name
7529 **
7530 **      Arguments:
7531 **              fqn -- an all lower-case host.domain string
7532 **              buckets -- the number of buckets (queue directories)
7533 **
7534 **      Returns:
7535 **              a bucket number (signed integer)
7536 **              -1 on error
7537 **
7538 **      Contributed by Exactis.com, Inc.
7539 */

7541 int
7542 hashfqn(fqn, buckets)
7543         register char *fqn;
7544         int buckets;
7545 {
7546         register char *p;
7547         register int h = 0, hash, cnt;

7549         if (fqn == NULL)
7550                 return -1;

7552         /*
7553         **  A variation on the gdb hash
7554         **  This is the best as of Feb 19, 1996 --bcx
7555         */

7557         p = fqn;
7558         h = 0x238F13AF * strlen(p);
7559         for (cnt = 0; *p != 0; ++p, cnt++)
7560         {
7561                 h = (h + (*p << (cnt * 5 % 24))) & 0x7FFFFFFF;
7562         }
7563         h = (1103515243 * h + 12345) & 0x7FFFFFFF;
7564         if (buckets < 2)
7565                 hash = 0;
7566         else
7567                 hash = (h % buckets);

7569         return hash;
7570 }
7571 #endif /* 0 */

7573 /*
7574 **  A structure for sorting Queue according to maxqrun without
7575 **      screwing up Queue itself.
7576 */

7578 struct sortqgrp
7579 {
7580         int sg_idx;             /* original index */
7581         int sg_maxqrun;         /* max queue runners */
7582 };
7583 typedef struct sortqgrp SORTQGRP_T;
7584 static int cmpidx __P((const void *, const void *));
```



```
7586 static int
7587 cmpidx(a, b)
7588         const void *a;
7589         const void *b;
7590 {
7591         /* The sort is highest to lowest, so the comparison is reversed */
7592         if (((SORTQGRP_T *)a)->sg_maxqrun < ((SORTQGRP_T *)b)->sg_maxqrun)
7593                 return 1;
7594         else if (((SORTQGRP_T *)a)->sg_maxqrun > ((SORTQGRP_T *)b)->sg_maxqrun)
7595                 return -1;
7596         else
7597                 return 0;
7598 }

7600 /*
7601 **  MAKEWORKGROUP -- balance queue groups into work groups per MaxQueueChildren
7602 **
7603 **  Take the now defined queue groups and assign them to work groups.
7604 **  This is done to balance out the number of concurrently active
7605 **  queue runners such that MaxQueueChildren is not exceeded. This may
7606 **  result in more than one queue group per work group. In such a case
7607 **  the number of running queue groups in that work group will have no
7608 **  more than the work group maximum number of runners (a "fair" portion
7609 **  of MaxQueueRunners). All queue groups within a work group will get a
7610 **  chance at running.
7611 **
7612 **      Parameters:
7613 **              none.
7614 **
7615 **      Returns:
7616 **              nothing.
7617 **
7618 **      Side Effects:
7619 **              Sets up WorkGrp structure.
7620 */

7622 void
7623 makeworkgroups()
7624 {
7625         int i, j, total_runners, dir, h;
7626         SORTQGRP_T si[MAXQUEUEGROUPS + 1];

7628         total_runners = 0;
7629         if (NumQueue == 1 && strcmp(Queue[0]->qg_name, "mqueue") == 0)
7630         {
7631                 /*
7632                 **  There is only the "mqueue" queue group (a default)
7633                 **  containing all of the queues. We want to provide to
7634                 **  this queue group the maximum allowable queue runners.
7635                 **  To match older behavior (8.10/8.11) we'll try for
7636                 **  1 runner per queue capping it at MaxQueueChildren.
7637                 **  So if there are N queues, then there will be N runners
7638                 **  for the "mqueue" queue group (where N is kept less than
7639                 **  MaxQueueChildren).
7640                 */

7642                 NumWorkGroups = 1;
7643                 WorkGrp[0].wg_numqgrp = 1;
7644                 WorkGrp[0].wg_qgs = (QUEUEGRP **) xalloc(sizeof(QUEUEGRP *));
7645                 WorkGrp[0].wg_qgs[0] = Queue[0];
7646                 if (MaxQueueChildren > 0 &&
7647                     Queue[0]->qg_numqueues > MaxQueueChildren)
7648                         WorkGrp[0].wg_runners = MaxQueueChildren;
7649                 else
7650                         WorkGrp[0].wg_runners = Queue[0]->qg_numqueues;
```

```
7652                Queue[0]->qg_wgrp = 0;

7654                /* can't have more runners than allowed total */
7655                if (MaxQueueChildren > 0 &&
7656                    Queue[0]->qg_maxqrun > MaxQueueChildren)
7657                        Queue[0]->qg_maxqrun = MaxQueueChildren;
7658                WorkGrp[0].wg_maxact = Queue[0]->qg_maxqrun;
7659                WorkGrp[0].wg_lowqintvl = Queue[0]->qg_queueintvl;
7660                return;
7661        }

7663        for (i = 0; i < NumQueue; i++)
7664        {
7665                si[i].sg_maxqrun = Queue[i]->qg_maxqrun;
7666                si[i].sg_idx = i;
7667        }
7668        qsort(si, NumQueue, sizeof(si[0]), cmpidx);

7670        NumWorkGroups = 0;
7671        for (i = 0; i < NumQueue; i++)
7672        {
7673                total_runners += si[i].sg_maxqrun;
7674                if (MaxQueueChildren <= 0 || total_runners <= MaxQueueChildren)
7675                        NumWorkGroups++;
7676                else
7677                        break;
7678        }

7680        if (NumWorkGroups < 1)
7681                NumWorkGroups = 1; /* gotta have one at least */
7682        else if (NumWorkGroups > MAXWORKGROUPS)
7683                NumWorkGroups = MAXWORKGROUPS; /* the limit */

7685        /*
7686        **  We now know the number of work groups to pack the queue groups
7687        **  into. The queue groups in 'Queue' are sorted from highest
7688        **  to lowest for the number of runners per queue group.
7689        **  We put the queue groups with the largest number of runners
7690        **  into work groups first. Then the smaller ones are fitted in
7691        **  where it looks best.
7692        */

7694        j = 0;
7695        dir = 1;
7696        for (i = 0; i < NumQueue; i++)
7697        {
7698                /* a to-and-fro packing scheme, continue from last position */
7699                if (j >= NumWorkGroups)
7700                {
7701                        dir = -1;
7702                        j = NumWorkGroups - 1;
7703                }
7704                else if (j < 0)
7705                {
7706                        j = 0;
7707                        dir = 1;
7708                }

7710                if (WorkGrp[j].wg_qgs == NULL)
7711                        WorkGrp[j].wg_qgs = (QUEUEGRP **)sm_malloc(sizeof(QUEUEG
7712                                                           (WorkGrp[j].wg_numqgrp +
7713                else
7714                        WorkGrp[j].wg_qgs = (QUEUEGRP **)sm_realloc(WorkGrp[j].w
7715                                                           sizeof(QUEUEGRP *) *
7716                                                           (WorkGrp[j].wg_numqgrp +
7717                if (WorkGrp[j].wg_qgs == NULL)
```



```
7718                {
7719                        syserr("!cannot allocate memory for work queues, need %d
7720                               (int) (sizeof(QUEUEGRP *) *
7721                                      (WorkGrp[j].wg_numqgrp + 1)));
7722                }

7724                h = si[i].sg_idx;
7725                WorkGrp[j].wg_qgs[WorkGrp[j].wg_numqgrp] = Queue[h];
7726                WorkGrp[j].wg_numqgrp++;
7727                WorkGrp[j].wg_runners += Queue[h]->qg_maxqrun;
7728                Queue[h]->qg_wgrp = j;

7730                if (WorkGrp[j].wg_maxact == 0)
7731                {
7732                        /* can't have more runners than allowed total */
7733                        if (MaxQueueChildren > 0 &&
7734                            Queue[h]->qg_maxqrun > MaxQueueChildren)
7735                                Queue[h]->qg_maxqrun = MaxQueueChildren;
7736                        WorkGrp[j].wg_maxact = Queue[h]->qg_maxqrun;
7737                }

7739                /*
7740                **  XXX: must wg_lowqintvl be the GCD?
7741                **  qg1: 2m, qg2: 3m, minimum: 2m, when do queue runs for
7742                **  qg2 occur?
7743                */

7745                /* keep track of the lowest interval for a persistent runner */
7746                if (Queue[h]->qg_queueintvl > 0 &&
7747                    WorkGrp[j].wg_lowqintvl < Queue[h]->qg_queueintvl)
7748                        WorkGrp[j].wg_lowqintvl = Queue[h]->qg_queueintvl;
7749                j += dir;
7750        }
7751        if (tTd(41, 9))
7752        {
7753                for (i = 0; i < NumWorkGroups; i++)
7754                {
7755                        sm_dprintf("Workgroup[%d]=", i);
7756                        for (j = 0; j < WorkGrp[i].wg_numqgrp; j++)
7757                        {
7758                                sm_dprintf("%s, ",
7759                                        WorkGrp[i].wg_qgs[j]->qg_name);
7760                        }
7761                        sm_dprintf("\n");
7762                }
7763        }
7764 }

7766 /*
7767 **  DUP_DF -- duplicate envelope data file
7768 **
7769 **      Copy the data file from the 'old' envelope to the 'new' envelope
7770 **      in the most efficient way possible.
7771 **
7772 **      Create a hard link from the 'old' data file to the 'new' data file.
7773 **      If the old and new queue directories are on different file systems,
7774 **      then the new data file link is created in the old queue directory,
7775 **      and the new queue file will contain a 'd' record pointing to the
7776 **      directory containing the new data file.
7777 **
7778 **      Parameters:
7779 **              old -- old envelope.
7780 **              new -- new envelope.
7781 **
7782 **      Results:
7783 **              Returns true on success, false on failure.
```

```
7784 **
7785 **      Side Effects:
7786 **              On success, the new data file is created.
7787 **              On fatal failure, EF_FATALERRS is set in old->e_flags.
7788 */

7790 static bool     dup_df __P((ENVELOPE *, ENVELOPE *));

7792 static bool
7793 dup_df(old, new)
7794         ENVELOPE *old;
7795         ENVELOPE *new;
7796 {
7797         int ofs, nfs, r;
7798         char opath[MAXPATHLEN];
7799         char npath[MAXPATHLEN];

7801         if (!bitset(EF_HAS_DF, old->e_flags))
7802         {
7803                 /*
7804                 **  this can happen if: SuperSafe != True
7805                 **  and a bounce mail is sent that is split.
7806                 */

7808                 queueup(old, false, true);
7809         }
7810         SM_REQUIRE(ISVALIDQGRP(old->e_qgrp) && ISVALIDQDIR(old->e_qdir));
7811         SM_REQUIRE(ISVALIDQGRP(new->e_qgrp) && ISVALIDQDIR(new->e_qdir));

7813         (void) sm_strlcpy(opath, queuename(old, DATAFL_LETTER), sizeof(opath));
7814         (void) sm_strlcpy(npath, queuename(new, DATAFL_LETTER), sizeof(npath));

7816         if (old->e_dfp != NULL)
7817         {
7818                 r = sm_io_setinfo(old->e_dfp, SM_BF_COMMIT, NULL);
7819                 if (r < 0 && errno != EINVAL)
7820                 {
7821                         syserr("@can't commit %s", opath);
7822                         old->e_flags |= EF_FATALERRS;
7823                         return false;
7824                 }
7825         }

7827         /*
7828         **  Attempt to create a hard link, if we think both old and new
7829         **  are on the same file system, otherwise copy the file.
7830         **
7831         **  Don't waste time attempting a hard link unless old and new
7832         **  are on the same file system.
7833         */

7835         SM_REQUIRE(ISVALIDQGRP(old->e_dfqgrp) && ISVALIDQDIR(old->e_dfqdir));
7836         SM_REQUIRE(ISVALIDQGRP(new->e_dfqgrp) && ISVALIDQDIR(new->e_dfqdir));

7838         ofs = Queue[old->e_dfqgrp]->qg_qpaths[old->e_dfqdir].qp_fsysidx;
7839         nfs = Queue[new->e_dfqgrp]->qg_qpaths[new->e_dfqdir].qp_fsysidx;
7840         if (FILE_SYS_DEV(ofs) == FILE_SYS_DEV(nfs))
7841         {
7842                 if (link(opath, npath) == 0)
7843                 {
7844                         new->e_flags |= EF_HAS_DF;
7845                         SYNC_DIR(npath, true);
7846                         return true;
7847                 }
7848                 goto error;
7849         }
```



```
7851         /*
7852         **  Can't link across queue directories, so try to create a hard
7853         **  link in the same queue directory as the old df file.
7854         **  The qf file will refer to the new df file using a 'd' record.
7855         */

7857         new->e_dfqgrp = old->e_dfqgrp;
7858         new->e_dfqdir = old->e_dfqdir;
7859         (void) sm_strlcpy(npath, queuename(new, DATAFL_LETTER), sizeof(npath));
7860         if (link(opath, npath) == 0)
7861         {
7862                 new->e_flags |= EF_HAS_DF;
7863                 SYNC_DIR(npath, true);
7864                 return true;
7865         }

7867   error:
7868         if (LogLevel > 0)
7869                 sm_syslog(LOG_ERR, old->e_id,
7870                           "dup_df: can't link %s to %s, error=%s, envelope split
7871                           opath, npath, sm_errstring(errno));
7872         return false;
7873 }

7875 /*
7876 **  SPLIT_ENV -- Allocate a new envelope based on a given envelope.
7877 **
7878 **      Parameters:
7879 **              e -- envelope.
7880 **              sendqueue -- sendqueue for new envelope.
7881 **              qgrp -- index of queue group.
7882 **              qdir -- queue directory.
7883 **
7884 **      Results:
7885 **              new envelope.
7886 **
7887 */

7889 static ENVELOPE *split_env __P((ENVELOPE *, ADDRESS *, int, int));

7891 static ENVELOPE *
7892 split_env(e, sendqueue, qgrp, qdir)
7893         ENVELOPE *e;
7894         ADDRESS *sendqueue;
7895         int qgrp;
7896         int qdir;
7897 {
7898         ENVELOPE *ee;

7900         ee = (ENVELOPE *) sm_rpool_malloc_x(e->e_rpool, sizeof(*ee));
7901         STRUCTCOPY(*e, *ee);
7902         ee->e_message = NULL;   /* XXX use original message? */
7903         ee->e_id = NULL;
7904         assign_queueid(ee);
7905         ee->e_sendqueue = sendqueue;
7906         ee->e_flags &= ~(EF_INQUEUE|EF_CLRQUEUE|EF_FATALERRS
7907                          |EF_SENDRECEIPT|EF_RET_PARAM|EF_HAS_DF);
7908         ee->e_flags |= EF_NORECEIPT;    /* XXX really? */
7909         ee->e_from.q_state = QS_SENDER;
7910         ee->e_dfp = NULL;
7911         ee->e_lockfp = NULL;
7912         if (e->e_xfp != NULL)
7913                 ee->e_xfp = sm_io_dup(e->e_xfp);

7915         /* failed to dup e->e_xfp, start a new transcript */
```

```
7916         if (ee->e_xfp == NULL)
7917                 openxscript(ee);

7919         ee->e_qgrp = ee->e_dfqgrp = qgrp;
7920         ee->e_qdir = ee->e_dfqdir = qdir;
7921         ee->e_errormode = EM_MAIL;
7922         ee->e_statmsg = NULL;
7923         if (e->e_quarmsg != NULL)
7924                 ee->e_quarmsg = sm_rpool_strdup_x(ee->e_rpool,
7925                                                   e->e_quarmsg);

7927         /*
7928         **  XXX Not sure if this copying is necessary.
7929         **  sendall() does this copying, but I (dm) don't know if that is
7930         **  because of the storage management discipline we were using
7931         **  before rpools were introduced, or if it is because these lists
7932         **  can be modified later.
7933         */

7935         ee->e_header = copyheader(e->e_header, ee->e_rpool);
7936         ee->e_errorqueue = copyqueue(e->e_errorqueue, ee->e_rpool);

7938         return ee;
7939 }

7941 /* return values from split functions, check also below! */
7942 #define SM_SPLIT_FAIL   (0)
7943 #define SM_SPLIT_NONE   (1)
7944 #define SM_SPLIT_NEW(n) (1 + (n))

7946 /*
7947 **  SPLIT_ACROSS_QUEUE_GROUPS
7948 **
7949 **      This function splits an envelope across multiple queue groups
7950 **      based on the queue group of each recipient.
7951 **
7952 **      Parameters:
7953 **              e -- envelope.
7954 **
7955 **      Results:
7956 **              SM_SPLIT_FAIL on failure
7957 **              SM_SPLIT_NONE if no splitting occurred,
7958 **              or 1 + the number of additional envelopes created.
7959 **
7960 **      Side Effects:
7961 **              On success, e->e_sibling points to a list of zero or more
7962 **              additional envelopes, and the associated data files exist
7963 **              on disk.  But the queue files are not created.
7964 **
7965 **              On failure, e->e_sibling is not changed.
7966 **              The order of recipients in e->e_sendqueue is permuted.
7967 **              Abandoned data files for additional envelopes that failed
7968 **              to be created may exist on disk.
7969 */

7971 static int      q_qgrp_compare __P((const void *, const void *));
7972 static int      e_filesys_compare __P((const void *, const void *));

7974 static int
7975 q_qgrp_compare(p1, p2)
7976         const void *p1;
7977         const void *p2;
7978 {
7979         ADDRESS **pq1 = (ADDRESS **) p1;
7980         ADDRESS **pq2 = (ADDRESS **) p2;
```



```
7982         return (*pq1)->q_qgrp - (*pq2)->q_qgrp;
7983 }

7985 static int
7986 e_filesys_compare(p1, p2)
7987         const void *p1;
7988         const void *p2;
7989 {
7990         ENVELOPE **pe1 = (ENVELOPE **) p1;
7991         ENVELOPE **pe2 = (ENVELOPE **) p2;
7992         int fs1, fs2;

7994         fs1 = Queue[(*pe1)->e_qgrp]->qg_qpaths[(*pe1)->e_qdir].qp_fsysidx;
7995         fs2 = Queue[(*pe2)->e_qgrp]->qg_qpaths[(*pe2)->e_qdir].qp_fsysidx;
7996         if (FILE_SYS_DEV(fs1) < FILE_SYS_DEV(fs2))
7997                 return -1;
7998         if (FILE_SYS_DEV(fs1) > FILE_SYS_DEV(fs2))
7999                 return 1;
8000         return 0;
8001 }

8003 static int split_across_queue_groups __P((ENVELOPE *));
8004 static int
8005 split_across_queue_groups(e)
8006         ENVELOPE *e;
8007 {
8008         int naddrs, nsplits, i;
8009         bool changed;
8010         char **pvp;
8011         ADDRESS *q, **addrs;
8012         ENVELOPE *ee, *es;
8013         ENVELOPE *splits[MAXQUEUEGROUPS];
8014         char pvpbuf[PSBUFSIZE];

8016         SM_REQUIRE(ISVALIDQGRP(e->e_qgrp));

8018         /* Count addresses and assign queue groups. */
8019         naddrs = 0;
8020         changed = false;
8021         for (q = e->e_sendqueue; q != NULL; q = q->q_next)
8022         {
8023                 if (QS_IS_DEAD(q->q_state))
8024                         continue;
8025                 ++naddrs;

8027                 /* bad addresses and those already sent stay put */
8028                 if (QS_IS_BADADDR(q->q_state) ||
8029                     QS_IS_SENT(q->q_state))
8030                         q->q_qgrp = e->e_qgrp;
8031                 else if (!ISVALIDQGRP(q->q_qgrp))
8032                 {
8033                         /* call ruleset which should return a queue group */
8034                         i = rscap(RS_QUEUEGROUP, q->q_user, NULL, e, &pvp,
8035                                   pvpbuf, sizeof(pvpbuf));
8036                         if (i == EX_OK &&
8037                             pvp != NULL && pvp[0] != NULL &&
8038                             (pvp[0][0] & 0377) == CANONNET &&
8039                             pvp[1] != NULL && pvp[1][0] != '\0')
8040                         {
8041                                 i = name2qid(pvp[1]);
8042                                 if (ISVALIDQGRP(i))
8043                                 {
8044                                         q->q_qgrp = i;
8045                                         changed = true;
8046                                         if (tTd(20, 4))
8047                                                 sm_syslog(LOG_INFO, NOQID,
```

```
8048                                                    "queue group name %s ->
8049                                                    pvp[1], i);
8050                                    continue;
8051                            }
8052                            else if (LogLevel > 10)
8053                                    sm_syslog(LOG_INFO, NOQID,
8054                                            "can't find queue group name %s,
8055                                            pvp[1]);
8056                    }
8057                    if (q->q_mailer != NULL &&
8058                        ISVALIDQGRP(q->q_mailer->m_qgrp))
8059                    {
8060                            changed = true;
8061                            q->q_qgrp = q->q_mailer->m_qgrp;
8062                    }
8063                    else if (ISVALIDQGRP(e->e_qgrp))
8064                            q->q_qgrp = e->e_qgrp;
8065                    else
8066                            q->q_qgrp = 0;
8067            }
8068    }

8070    /* only one address? nothing to split. */
8071    if (naddrs <= 1 && !changed)
8072            return SM_SPLIT_NONE;

8074    /* sort the addresses by queue group */
8075    addrs = sm_rpool_malloc_x(e->e_rpool, naddrs * sizeof(ADDRESS *));
8076    for (i = 0, q = e->e_sendqueue; q != NULL; q = q->q_next)
8077    {
8078            if (QS_IS_DEAD(q->q_state))
8079                    continue;
8080            addrs[i++] = q;
8081    }
8082    qsort(addrs, naddrs, sizeof(ADDRESS *), q_qgrp_compare);

8084    /* split into multiple envelopes, by queue group */
8085    nsplits = 0;
8086    es = NULL;
8087    e->e_sendqueue = NULL;
8088    for (i = 0; i < naddrs; ++i)
8089    {
8090            if (i == naddrs - 1 || addrs[i]->q_qgrp != addrs[i + 1]->q_qgrp)
8091                    addrs[i]->q_next = NULL;
8092            else
8093                    addrs[i]->q_next = addrs[i + 1];

8095            /* same queue group as original envelope? */
8096            if (addrs[i]->q_qgrp == e->e_qgrp)
8097            {
8098                    if (e->e_sendqueue == NULL)
8099                            e->e_sendqueue = addrs[i];
8100                    continue;
8101            }

8103            /* different queue group than original envelope */
8104            if (es == NULL || addrs[i]->q_qgrp != es->e_qgrp)
8105            {
8106                    ee = split_env(e, addrs[i], addrs[i]->q_qgrp, NOQDIR);
8107                    es = ee;
8108                    splits[nsplits++] = ee;
8109            }
8110    }

8112    /* no splits? return right now. */
8113    if (nsplits <= 0)
```



```
8114                    return SM_SPLIT_NONE;

8116            /* assign a queue directory to each additional envelope */
8117            for (i = 0; i < nsplits; ++i)
8118            {
8119                    es = splits[i];
8120    #if 0
8121                    es->e_qdir = pickqdir(Queue[es->e_qgrp], es->e_msgsize, es);
8122    #endif /* 0 */
8123                    if (!setnewqueue(es))
8124                            goto failure;
8125            }

8127            /* sort the additional envelopes by queue file system */
8128            qsort(splits, nsplits, sizeof(ENVELOPE *), e_filesys_compare);

8130            /* create data files for each additional envelope */
8131            if (!dup_df(e, splits[0]))
8132            {
8133                    i = 0;
8134                    goto failure;
8135            }
8136            for (i = 1; i < nsplits; ++i)
8137            {
8138                    /* copy or link to the previous data file */
8139                    if (!dup_df(splits[i - 1], splits[i]))
8140                            goto failure;
8141            }

8143            /* success: prepend the new envelopes to the e->e_sibling list */
8144            for (i = 0; i < nsplits; ++i)
8145            {
8146                    es = splits[i];
8147                    es->e_sibling = e->e_sibling;
8148                    e->e_sibling = es;
8149            }
8150            return SM_SPLIT_NEW(nsplits);

8152            /* failure: clean up */
8153      failure:
8154            if (i > 0)
8155            {
8156                    int j;

8158                    for (j = 0; j < i; j++)
8159                            (void) unlink(queuename(splits[j], DATAFL_LETTER));
8160            }
8161            e->e_sendqueue = addrs[0];
8162            for (i = 0; i < naddrs - 1; ++i)
8163                    addrs[i]->q_next = addrs[i + 1];
8164            addrs[naddrs - 1]->q_next = NULL;
8165            return SM_SPLIT_FAIL;
8166    }

8168    /*
8169    **  SPLIT_WITHIN_QUEUE
8170    **
8171    **      Split an envelope with multiple recipients into several
8172    **      envelopes within the same queue directory, if the number of
8173    **      recipients exceeds the limit for the queue group.
8174    **
8175    **      Parameters:
8176    **              e -- envelope.
8177    **
8178    **      Results:
8179    **              SM_SPLIT_FAIL on failure
```

```
8180 **              SM_SPLIT_NONE if no splitting occurred,
8181 **              or 1 + the number of additional envelopes created.
8182 */

8184 #define SPLIT_LOG_LEVEL 8

8186 static int      split_within_queue __P((ENVELOPE *));

8188 static int
8189 split_within_queue(e)
8190         ENVELOPE *e;
8191 {
8192         int maxrcpt, nrcpt, ndead, nsplit, i;
8193         int j, l;
8194         char *lsplits;
8195         ADDRESS *q, **addrs;
8196         ENVELOPE *ee, *firstsibling;

8198         if (!ISVALIDQGRP(e->e_qgrp) || bitset(EF_SPLIT, e->e_flags))
8199                 return SM_SPLIT_NONE;

8201         /* don't bother if there is no recipient limit */
8202         maxrcpt = Queue[e->e_qgrp]->qg_maxrcpt;
8203         if (maxrcpt <= 0)
8204                 return SM_SPLIT_NONE;

8206         /* count recipients */
8207         nrcpt = 0;
8208         for (q = e->e_sendqueue; q != NULL; q = q->q_next)
8209         {
8210                 if (QS_IS_DEAD(q->q_state))
8211                         continue;
8212                 ++nrcpt;
8213         }
8214         if (nrcpt <= maxrcpt)
8215                 return SM_SPLIT_NONE;

8217         /*
8218         **  Preserve the recipient list
8219         **  so that we can restore it in case of error.
8220         **  (But we discard dead addresses.)
8221         */

8223         addrs = sm_rpool_malloc_x(e->e_rpool, nrcpt * sizeof(ADDRESS *));
8224         for (i = 0, q = e->e_sendqueue; q != NULL; q = q->q_next)
8225         {
8226                 if (QS_IS_DEAD(q->q_state))
8227                         continue;
8228                 addrs[i++] = q;
8229         }

8231         /*
8232         **  Partition the recipient list so that bad and sent addresses
8233         **  come first. These will go with the original envelope, and
8234         **  do not count towards the maxrcpt limit.
8235         **  addrs[] does not contain QS_IS_DEAD() addresses.
8236         */

8238         ndead = 0;
8239         for (i = 0; i < nrcpt; ++i)
8240         {
8241                 if (QS_IS_BADADDR(addrs[i]->q_state) ||
8242                     QS_IS_SENT(addrs[i]->q_state) ||
8243                     QS_IS_DEAD(addrs[i]->q_state)) /* for paranoia's sake */
8244                 {
8245                         if (i > ndead)
```



```
8246                         {
8247                                 ADDRESS *tmp = addrs[i];

8249                                 addrs[i] = addrs[ndead];
8250                                 addrs[ndead] = tmp;
8251                         }
8252                         ++ndead;
8253                 }
8254         }

8256         /* Check if no splitting required. */
8257         if (nrcpt - ndead <= maxrcpt)
8258                 return SM_SPLIT_NONE;

8260         /* fix links */
8261         for (i = 0; i < nrcpt - 1; ++i)
8262                 addrs[i]->q_next = addrs[i + 1];
8263         addrs[nrcpt - 1]->q_next = NULL;
8264         e->e_sendqueue = addrs[0];

8266         /* prepare buffer for logging */
8267         if (LogLevel > SPLIT_LOG_LEVEL)
8268         {
8269                 l = MAXLINE;
8270                 lsplits = sm_malloc(l);
8271                 if (lsplits != NULL)
8272                         *lsplits = '\0';
8273                 j = 0;
8274         }
8275         else
8276         {
8277                 /* get rid of stupid compiler warnings */
8278                 lsplits = NULL;
8279                 j = l = 0;
8280         }

8282         /* split the envelope */
8283         firstsibling = e->e_sibling;
8284         i = maxrcpt + ndead;
8285         nsplit = 0;
8286         for (;;)
8287         {
8288                 addrs[i - 1]->q_next = NULL;
8289                 ee = split_env(e, addrs[i], e->e_qgrp, e->e_qdir);
8290                 if (!dup_df(e, ee))
8291                 {

8293                         ee = firstsibling;
8294                         while (ee != NULL)
8295                         {
8296                                 (void) unlink(queuename(ee, DATAFL_LETTER));
8297                                 ee = ee->e_sibling;
8298                         }

8300                         /* Error.  Restore e's sibling & recipient lists. */
8301                         e->e_sibling = firstsibling;
8302                         for (i = 0; i < nrcpt - 1; ++i)
8303                                 addrs[i]->q_next = addrs[i + 1];
8304                         if (lsplits != NULL)
8305                                 sm_free(lsplits);
8306                         return SM_SPLIT_FAIL;
8307                 }

8309                 /* prepend the new envelope to e->e_sibling */
8310                 ee->e_sibling = e->e_sibling;
8311                 e->e_sibling = ee;
```

```
8312                ++nsplit;
8313                if (LogLevel > SPLIT_LOG_LEVEL && lsplits != NULL)
8314                {
8315                        if (j >= l - strlen(ee->e_id) - 3)
8316                        {
8317                                char *p;

8319                                l += MAXLINE;
8320                                p = sm_realloc(lsplits, l);
8321                                if (p == NULL)
8322                                {
8323                                        /* let's try to get this done */
8324                                        sm_free(lsplits);
8325                                        lsplits = NULL;
8326                                }
8327                                else
8328                                        lsplits = p;
8329                        }
8330                        if (lsplits != NULL)
8331                        {
8332                                if (j == 0)
8333                                        j += sm_strlcat(lsplits + j,
8334                                                        ee->e_id,
8335                                                        l - j);
8336                                else
8337                                        j += sm_strlcat2(lsplits + j,
8338                                                         "; ",
8339                                                         ee->e_id,
8340                                                         l - j);
8341                                SM_ASSERT(j < l);
8342                        }
8343                }
8344                if (nrcpt - i <= maxrcpt)
8345                        break;
8346                i += maxrcpt;
8347        }
8348        if (LogLevel > SPLIT_LOG_LEVEL && lsplits != NULL)
8349        {
8350                if (nsplit > 0)
8351                {
8352                        sm_syslog(LOG_NOTICE, e->e_id,
8353                                  "split: maxrcpts=%d, rcpts=%d, count=%d, id%s=
8354                                  maxrcpt, nrcpt - ndead, nsplit,
8355                                  nsplit > 1 ? "s" : "", lsplits);
8356                }
8357                sm_free(lsplits);
8358        }
8359        return SM_SPLIT_NEW(nsplit);
8360 }
8361 /*
8362 **  SPLIT_BY_RECIPIENT
8363 **
8364 **      Split an envelope with multiple recipients into multiple
8365 **      envelopes as required by the sendmail configuration.
8366 **
8367 **      Parameters:
8368 **              e -- envelope.
8369 **
8370 **      Results:
8371 **              Returns true on success, false on failure.
8372 **
8373 **      Side Effects:
8374 **              see split_across_queue_groups(), split_within_queue(e)
8375 */

8377 bool
```



```
8378 split_by_recipient(e)
8379         ENVELOPE *e;
8380 {
8381         int split, n, i, j, l;
8382         char *lsplits;
8383         ENVELOPE *ee, *next, *firstsibling;

8385         if (OpMode == SM_VERIFY || !ISVALIDQGRP(e->e_qgrp) ||
8386             bitset(EF_SPLIT, e->e_flags))
8387                 return true;
8388         n = split_across_queue_groups(e);
8389         if (n == SM_SPLIT_FAIL)
8390                 return false;
8391         firstsibling = ee = e->e_sibling;
8392         if (n > 1 && LogLevel > SPLIT_LOG_LEVEL)
8393         {
8394                 l = MAXLINE;
8395                 lsplits = sm_malloc(l);
8396                 if (lsplits != NULL)
8397                         *lsplits = '\0';
8398                 j = 0;
8399         }
8400         else
8401         {
8402                 /* get rid of stupid compiler warnings */
8403                 lsplits = NULL;
8404                 j = l = 0;
8405         }
8406         for (i = 1; i < n; ++i)
8407         {
8408                 next = ee->e_sibling;
8409                 if (split_within_queue(ee) == SM_SPLIT_FAIL)
8410                 {
8411                         e->e_sibling = firstsibling;
8412                         return false;
8413                 }
8414                 ee->e_flags |= EF_SPLIT;
8415                 if (LogLevel > SPLIT_LOG_LEVEL && lsplits != NULL)
8416                 {
8417                         if (j >= l - strlen(ee->e_id) - 3)
8418                         {
8419                                 char *p;

8421                                 l += MAXLINE;
8422                                 p = sm_realloc(lsplits, l);
8423                                 if (p == NULL)
8424                                 {
8425                                         /* let's try to get this done */
8426                                         sm_free(lsplits);
8427                                         lsplits = NULL;
8428                                 }
8429                                 else
8430                                         lsplits = p;
8431                         }
8432                         if (lsplits != NULL)
8433                         {
8434                                 if (j == 0)
8435                                         j += sm_strlcat(lsplits + j,
8436                                                         ee->e_id, l - j);
8437                                 else
8438                                         j += sm_strlcat2(lsplits + j, "; ",
8439                                                          ee->e_id, l - j);
8440                                 SM_ASSERT(j < l);
8441                         }
8442                 }
8443                 ee = next;
```

```
8444         }
8445         if (LogLevel > SPLIT_LOG_LEVEL && lsplits != NULL && n > 1)
8446         {
8447                 sm_syslog(LOG_NOTICE, e->e_id, "split: count=%d, id%s=%s",
8448                           n - 1, n > 2 ? "s" : "", lsplits);
8449                 sm_free(lsplits);
8450         }
8451         split = split_within_queue(e) != SM_SPLIT_FAIL;
8452         if (split)
8453                 e->e_flags |= EF_SPLIT;
8454         return split;
8455 }

8457 /*
8458 **  QUARANTINE_QUEUE_ITEM -- {un,}quarantine a single envelope
8459 **
8460 **      Add/remove quarantine reason and requeue appropriately.
8461 **
8462 **      Parameters:
8463 **              qgrp -- queue group for the item
8464 **              qdir -- queue directory in the given queue group
8465 **              e -- envelope information for the item
8466 **              reason -- quarantine reason, NULL means unquarantine.
8467 **
8468 **      Results:
8469 **              true if item changed, false otherwise
8470 **
8471 **      Side Effects:
8472 **              Changes quarantine tag in queue file and renames it.
8473 */

8475 static bool
8476 quarantine_queue_item(qgrp, qdir, e, reason)
8477         int qgrp;
8478         int qdir;
8479         ENVELOPE *e;
8480         char *reason;
8481 {
8482         bool dirty = false;
8483         bool failing = false;
8484         bool foundq = false;
8485         bool finished = false;
8486         int fd;
8487         int flags;
8488         int oldtype;
8489         int newtype;
8490         int save_errno;
8491         MODE_T oldumask = 0;
8492         SM_FILE_T *oldqfp, *tempqfp;
8493         char *bp;
8494         int bufsize;
8495         char oldqf[MAXPATHLEN];
8496         char tempqf[MAXPATHLEN];
8497         char newqf[MAXPATHLEN];
8498         char buf[MAXLINE];

8500         oldtype = queue_letter(e, ANYQFL_LETTER);
8501         (void) sm_strlcpy(oldqf, queuename(e, ANYQFL_LETTER), sizeof(oldqf));
8502         (void) sm_strlcpy(tempqf, queuename(e, NEWQFL_LETTER), sizeof(tempqf));

8504         /*
8505         **  Instead of duplicating all the open
8506         **  and lock code here, tell readqf() to
8507         **  do that work and return the open
8508         **  file pointer in e_lockfp.  Note that
8509         **  we must release the locks properly when
```



```
8510         **  we are done.
8511         */

8513         if (!readqf(e, true))
8514         {
8515                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8516                                      "Skipping %s\n", qid_printname(e));
8517                 return false;
8518         }
8519         oldqfp = e->e_lockfp;

8521         /* open the new queue file */
8522         flags = O_CREAT|O_WRONLY|O_EXCL;
8523         if (bitset(S_IWGRP, QueueFileMode))
8524                 oldumask = umask(002);
8525         fd = open(tempqf, flags, QueueFileMode);
8526         if (bitset(S_IWGRP, QueueFileMode))
8527                 (void) umask(oldumask);
8528         RELEASE_QUEUE;

8530         if (fd < 0)
8531         {
8532                 save_errno = errno;
8533                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8534                                      "Skipping %s: Could not open %s: %s\n",
8535                                      qid_printname(e), tempqf,
8536                                      sm_errstring(save_errno));
8537                 (void) sm_io_close(oldqfp, SM_TIME_DEFAULT);
8538                 return false;
8539         }
8540         if (!lockfile(fd, tempqf, NULL, LOCK_EX|LOCK_NB))
8541         {
8542                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8543                                      "Skipping %s: Could not lock %s\n",
8544                                      qid_printname(e), tempqf);
8545                 (void) close(fd);
8546                 (void) sm_io_close(oldqfp, SM_TIME_DEFAULT);
8547                 return false;
8548         }

8550         tempqfp = sm_io_open(SmFtStdiofd, SM_TIME_DEFAULT, (void *) &fd,
8551                              SM_IO_WRONLY_B, NULL);
8552         if (tempqfp == NULL)
8553         {
8554                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8555                                      "Skipping %s: Could not lock %s\n",
8556                                      qid_printname(e), tempqf);
8557                 (void) close(fd);
8558                 (void) sm_io_close(oldqfp, SM_TIME_DEFAULT);
8559                 return false;
8560         }

8562         /* Copy the data over, changing the quarantine reason */
8563         while (bufsize = sizeof(buf),
8564                (bp = fgetfolded(buf, &bufsize, oldqfp)) != NULL)
8565         {
8566                 if (tTd(40, 4))
8567                         sm_dprintf("+++++ %s\n", bp);
8568                 switch (bp[0])
8569                 {
8570                   case 'q':             /* quarantine reason */
8571                         foundq = true;
8572                         if (reason == NULL)
8573                         {
8574                                 if (Verbose)
8575                                 {
```

```
8576                                    (void) sm_io_fprintf(smioout,
8577                                                         SM_TIME_DEFAULT,
8578                                                         "%s: Removed quaran
8579                                                         e->e_id, &bp[1]);
8580                            }
8581                            sm_syslog(LOG_INFO, e->e_id, "unquarantine");
8582                            dirty = true;
8583                    }
8584                    else if (strcmp(reason, &bp[1]) == 0)
8585                    {
8586                            if (Verbose)
8587                            {
8588                                    (void) sm_io_fprintf(smioout,
8589                                                         SM_TIME_DEFAULT,
8590                                                         "%s: Already quaran
8591                                                         e->e_id, reason);
8592                            }
8593                            (void) sm_io_fprintf(tempqfp, SM_TIME_DEFAULT,
8594                                                 "q%s\n", reason);
8595                    }
8596                    else
8597                    {
8598                            if (Verbose)
8599                            {
8600                                    (void) sm_io_fprintf(smioout,
8601                                                         SM_TIME_DEFAULT,
8602                                                         "%s: Quarantine cha
8603                                                         e->e_id, &bp[1],
8604                                                         reason);
8605                            }
8606                            (void) sm_io_fprintf(tempqfp, SM_TIME_DEFAULT,
8607                                                 "q%s\n", reason);
8608                            sm_syslog(LOG_INFO, e->e_id, "quarantine=%s",
8609                                      reason);
8610                            dirty = true;
8611                    }
8612                    break;

8614            case 'S':
8615                    /*
8616                    **  If we are quarantining an unquarantined item,
8617                    **  need to put in a new 'q' line before it's
8618                    **  too late.
8619                    */

8621                    if (!foundq && reason != NULL)
8622                    {
8623                            if (Verbose)
8624                            {
8625                                    (void) sm_io_fprintf(smioout,
8626                                                         SM_TIME_DEFAULT,
8627                                                         "%s: Quarantined wi
8628                                                         e->e_id, reason);
8629                            }
8630                            (void) sm_io_fprintf(tempqfp, SM_TIME_DEFAULT,
8631                                                 "q%s\n", reason);
8632                            sm_syslog(LOG_INFO, e->e_id, "quarantine=%s",
8633                                      reason);
8634                            foundq = true;
8635                            dirty = true;
8636                    }

8638                    /* Copy the line to the new file */
8639                    (void) sm_io_fprintf(tempqfp, SM_TIME_DEFAULT,
8640                                         "%s\n", bp);
8641                    break;
```



```
8643            case '.':
8644                    finished = true;
8645                    /* FALLTHROUGH */

8647            default:
8648                    /* Copy the line to the new file */
8649                    (void) sm_io_fprintf(tempqfp, SM_TIME_DEFAULT,
8650                                         "%s\n", bp);
8651                    break;
8652            }
8653            if (bp != buf)
8654                    sm_free(bp);
8655    }

8657    /* Make sure we read the whole old file */
8658    errno = sm_io_error(tempqfp);
8659    if (errno != 0 && errno != SM_IO_EOF)
8660    {
8661            save_errno = errno;
8662            (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8663                                 "Skipping %s: Error reading %s: %s\n",
8664                                 qid_printname(e), oldqf,
8665                                 sm_errstring(save_errno));
8666            failing = true;
8667    }

8669    if (!failing && !finished)
8670    {
8671            (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8672                                 "Skipping %s: Incomplete file: %s\n",
8673                                 qid_printname(e), oldqf);
8674            failing = true;
8675    }

8677    /* Check if we actually changed anything or we can just bail now */
8678    if (!dirty)
8679    {
8680            /* pretend we failed, even though we technically didn't */
8681            failing = true;
8682    }

8684    /* Make sure we wrote things out safely */
8685    if (!failing &&
8686        (sm_io_flush(tempqfp, SM_TIME_DEFAULT) != 0 ||
8687         ((SuperSafe == SAFE_REALLY ||
8688           SuperSafe == SAFE_REALLY_POSTMILTER ||
8689           SuperSafe == SAFE_INTERACTIVE) &&
8690          fsync(sm_io_getinfo(tempqfp, SM_IO_WHAT_FD, NULL)) < 0) ||
8691         ((errno = sm_io_error(tempqfp)) != 0)))
8692    {
8693            save_errno = errno;
8694            (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8695                                 "Skipping %s: Error writing %s: %s\n",
8696                                 qid_printname(e), tempqf,
8697                                 sm_errstring(save_errno));
8698            failing = true;
8699    }


8702    /* Figure out the new filename */
8703    newtype = (reason == NULL ? NORMQF_LETTER : QUARQF_LETTER);
8704    if (oldtype == newtype)
8705    {
8706            /* going to rename tempqf to oldqf */
8707            (void) sm_strlcpy(newqf, oldqf, sizeof(newqf));
```

```
8708         }
8709         else
8710         {
8711                 /* going to rename tempqf to new name based on newtype */
8712                 (void) sm_strlcpy(newqf, queuename(e, newtype), sizeof(newqf));
8713         }

8715         save_errno = 0;

8717         /* rename tempqf to newqf */
8718         if (!failing &&
8719             rename(tempqf, newqf) < 0)
8720                 save_errno = (errno == 0) ? EINVAL : errno;

8722         /* Check rename() success */
8723         if (!failing && save_errno != 0)
8724         {
8725                 sm_syslog(LOG_DEBUG, e->e_id,
8726                           "quarantine_queue_item: rename(%s, %s): %s",
8727                           tempqf, newqf, sm_errstring(save_errno));

8729                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8730                                      "Error renaming %s to %s: %s\n",
8731                                      tempqf, newqf,
8732                                      sm_errstring(save_errno));
8733                 if (oldtype == newtype)
8734                 {
8735                         /*
8736                         **  Bail here since we don't know the state of
8737                         **  the filesystem and may need to keep tempqf
8738                         **  for the user to rescue us.
8739                         */

8741                         RELEASE_QUEUE;
8742                         errno = save_errno;
8743                         syserr("!452 Error renaming control file %s", tempqf);
8744                         /* NOTREACHED */
8745                 }
8746                 else
8747                 {
8748                         /* remove new file (if rename() half completed) */
8749                         if (xunlink(newqf) < 0)
8750                         {
8751                                 save_errno = errno;
8752                                 (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8753                                                      "Error removing %s: %s\n",
8754                                                      newqf,
8755                                                      sm_errstring(save_errno));
8756                         }

8758                         /* tempqf removed below */
8759                         failing = true;
8760                 }

8762         }

8764         /* If changing file types, need to remove old type */
8765         if (!failing && oldtype != newtype)
8766         {
8767                 if (xunlink(oldqf) < 0)
8768                 {
8769                         save_errno = errno;
8770                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8771                                              "Error removing %s: %s\n",
8772                                              oldqf, sm_errstring(save_errno));
8773                 }
```



```
8774         }

8776         /* see if anything above failed */
8777         if (failing)
8778         {
8779                 /* Something failed: remove new file, old file still there */
8780                 (void) xunlink(tempqf);
8781         }

8783         /*
8784         **  fsync() after file operations to make sure metadata is
8785         **  written to disk on filesystems in which renames are
8786         **  not guaranteed.  It's ok if they fail, mail won't be lost.
8787         */

8789         if (SuperSafe != SAFE_NO)
8790         {
8791                 /* for soft-updates */
8792                 (void) fsync(sm_io_getinfo(tempqfp,
8793                                            SM_IO_WHAT_FD, NULL));

8795                 if (!failing)
8796                 {
8797                         /* for soft-updates */
8798                         (void) fsync(sm_io_getinfo(oldqfp,
8799                                                    SM_IO_WHAT_FD, NULL));
8800                 }

8802                 /* for other odd filesystems */
8803                 SYNC_DIR(tempqf, false);
8804         }

8806         /* Close up shop */
8807         RELEASE_QUEUE;
8808         if (tempqfp != NULL)
8809                 (void) sm_io_close(tempqfp, SM_TIME_DEFAULT);
8810         if (oldqfp != NULL)
8811                 (void) sm_io_close(oldqfp, SM_TIME_DEFAULT);

8813         /* All went well */
8814         return !failing;
8815 }

8817 /*
8818 **  QUARANTINE_QUEUE -- {un,}quarantine matching items in the queue
8819 **
8820 **      Read all matching queue items, add/remove quarantine
8821 **      reason, and requeue appropriately.
8822 **
8823 **      Parameters:
8824 **              reason -- quarantine reason, "." means unquarantine.
8825 **              qgrplimit -- limit to single queue group unless NOQGRP
8826 **
8827 **      Results:
8828 **              none.
8829 **
8830 **      Side Effects:
8831 **              Lots of changes to the queue.
8832 */

8834 void
8835 quarantine_queue(reason, qgrplimit)
8836         char *reason;
8837         int qgrplimit;
8838 {
8839         int changed = 0;
```

```
8840         int qgrp;
8841
8842         /* Convert internal representation of unquarantine */
8843         if (reason != NULL && reason[0] == '.' && reason[1] == '\0')
8844                 reason = NULL;
8845
8846         if (reason != NULL)
8847         {
8848                 /* clean it */
8849                 reason = newstr(denlstring(reason, true, true));
8850         }
8851
8852         for (qgrp = 0; qgrp < NumQueue && Queue[qgrp] != NULL; qgrp++)
8853         {
8854                 int qdir;
8855
8856                 if (qgrplimit != NOQGRP && qgrplimit != qgrp)
8857                         continue;
8858
8859                 for (qdir = 0; qdir < Queue[qgrp]->qg_numqueues; qdir++)
8860                 {
8861                         int i;
8862                         int nrequests;
8863
8864                         if (StopRequest)
8865                                 stop_sendmail();
8866
8867                         nrequests = gatherq(qgrp, qdir, true, NULL, NULL, NULL);
8868
8869                         /* first see if there is anything */
8870                         if (nrequests <= 0)
8871                         {
8872                                 if (Verbose)
8873                                 {
8874                                         (void) sm_io_fprintf(smioout,
8875                                                              SM_TIME_DEFAULT, "%
8876                                                              qid_printqueue(qgrp
8877                                 }
8878                                 continue;
8879                         }
8880
8881                         if (Verbose)
8882                         {
8883                                 (void) sm_io_fprintf(smioout,
8884                                                      SM_TIME_DEFAULT, "Processin
8885                                                      qid_printqueue(qgrp, qdir))
8886                         }
8887
8888                         for (i = 0; i < WorkListCount; i++)
8889                         {
8890                                 ENVELOPE e;
8891
8892                                 if (StopRequest)
8893                                         stop_sendmail();
8894
8895                                 /* setup envelope */
8896                                 clearenvelope(&e, true, sm_rpool_new_x(NULL));
8897                                 e.e_id = WorkList[i].w_name + 2;
8898                                 e.e_qgrp = qgrp;
8899                                 e.e_qdir = qdir;
8900
8901                                 if (tTd(70, 101))
8902                                 {
8903                                         sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8904                                                       "Would do %s\n", e.e_id);
8905                                         changed++;
```



```
8906                                 }
8907                                 else if (quarantine_queue_item(qgrp, qdir,
8908                                                                &e, reason))
8909                                         changed++;
8910
8911                                 /* clean up */
8912                                 sm_rpool_free(e.e_rpool);
8913                                 e.e_rpool = NULL;
8914                         }
8915                         if (WorkList != NULL)
8916                                 sm_free(WorkList); /* XXX */
8917                         WorkList = NULL;
8918                         WorkListSize = 0;
8919                         WorkListCount = 0;
8920                 }
8921         }
8922         if (Verbose)
8923         {
8924                 if (changed == 0)
8925                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8926                                              "No changes\n");
8927                 else
8928                         (void) sm_io_fprintf(smioout, SM_TIME_DEFAULT,
8929                                              "%d change%s\n",
8930                                              changed,
8931                                              changed == 1 ? "" : "s");
8932         }
8933 }
```

```
**********
   54731 Tue Aug 18 16:13:44 2015
new/usr/src/cmd/sendmail/src/util.c
6136 sysmacros.h unnecessarily polutes the namespace
**********
```

```
/*
 * Copyright (c) 1998-2007, 2009 Sendmail, Inc. and its suppliers.
 *      All rights reserved.
 * Copyright (c) 1983, 1995-1997 Eric P. Allman.  All rights reserved.
 * Copyright (c) 1988, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * By using this file, you agree to the terms and conditions set
 * forth in the LICENSE file which can be found at the top level of
 * the sendmail distribution.
 *
 */

#include <sendmail.h>

SM_RCSID("@(#)$Id: util.c,v 8.416 2009/12/18 17:05:26 ca Exp $")

#include <sm/sendmail.h>
#include <sysexits.h>
#include <sm/xtrap.h>
#include <sys/types.h>
#include <sys/mkdev.h>
#endif /* ! codereview */

/*
**  NEWSTR -- Create a copy of a C string
**
**      Parameters:
**              s -- the string to copy.
**
**      Returns:
**              pointer to newly allocated string.
*/

char *
newstr(s)
        const char *s;
{
        size_t l;
        char *n;

        l = strlen(s);
        SM_ASSERT(l + 1 > l);
        n = xalloc(l + 1);
        sm_strlcpy(n, s, l + 1);
        return n;
}

/*
**  ADDQUOTES -- Adds quotes & quote bits to a string.
**
**      Runs through a string and adds backslashes and quote bits.
**
**      Parameters:
**              s -- the string to modify.
**              rpool -- resource pool from which to allocate result
**
**      Returns:
**              pointer to quoted string.
*/
```



```
char *
addquotes(s, rpool)
        char *s;
        SM_RPOOL_T *rpool;
{
        int len = 0;
        char c;
        char *p = s, *q, *r;

        if (s == NULL)
                return NULL;

        /* Find length of quoted string */
        while ((c = *p++) != '\0')
        {
                len++;
                if (c == '\\' || c == '"')
                        len++;
        }

        q = r = sm_rpool_malloc_x(rpool, len + 3);
        p = s;

        /* add leading quote */
        *q++ = '"';
        while ((c = *p++) != '\0')
        {
                /* quote \ or " */
                if (c == '\\' || c == '"')
                        *q++ = '\\';
                *q++ = c;
        }
        *q++ = '"';
        *q = '\0';
        return r;
}

/*
**  STRIPBACKSLASH -- Strip all leading backslashes from a string, provided
**      the following character is alpha-numerical.
**
**      This is done in place.
**
**      Parameters:
**              s -- the string to strip.
**
**      Returns:
**              none.
*/

void
stripbackslash(s)
        char *s;
{
        char *p, *q, c;

        if (s == NULL || *s == '\0')
                return;
        p = q = s;
        while (*p == '\\' && (p[1] == '\\' || (isascii(p[1]) && isalnum(p[1]))))
                p++;
        do
        {
                c = *q++ = *p++;
        } while (c != '\0');
}
```

```
/*
**  RFC822_STRING -- Checks string for proper RFC822 string quoting.
**
**      Runs through a string and verifies RFC822 special characters
**      are only found inside comments, quoted strings, or backslash
**      escaped.  Also verified balanced quotes and parenthesis.
**
**      Parameters:
**              s -- the string to modify.
**
**      Returns:
**              true iff the string is RFC822 compliant, false otherwise.
*/

bool
rfc822_string(s)
        char *s;
{
        bool quoted = false;
        int commentlev = 0;
        char *c = s;

        if (s == NULL)
                return false;

        while (*c != '\0')
        {
                /* escaped character */
                if (*c == '\\')
                {
                        c++;
                        if (*c == '\0')
                                return false;
                }
                else if (commentlev == 0 && *c == '"')
                        quoted = !quoted;
                else if (!quoted)
                {
                        if (*c == ')')
                        {
                                /* unbalanced ')' */
                                if (commentlev == 0)
                                        return false;
                                else
                                        commentlev--;
                        }
                        else if (*c == '(')
                                commentlev++;
                        else if (commentlev == 0 &&
                                 strchr(MustQuoteChars, *c) != NULL)
                                return false;
                }
                c++;
        }

        /* unbalanced '"' or '(' */
        return !quoted && commentlev == 0;
}

/*
**  SHORTEN_RFC822_STRING -- Truncate and rebalance an RFC822 string
**
**      Arbitrarily shorten (in place) an RFC822 string and rebalance
**      comments and quotes.
**
```



```
**      Parameters:
**              string -- the string to shorten
**              length -- the maximum size, 0 if no maximum
**
**      Returns:
**              true if string is changed, false otherwise
**
**      Side Effects:
**              Changes string in place, possibly resulting
**              in a shorter string.
*/

bool
shorten_rfc822_string(string, length)
        char *string;
        size_t length;
{
        bool backslash = false;
        bool modified = false;
        bool quoted = false;
        size_t slen;
        int parencount = 0;
        char *ptr = string;

        /*
        **  If have to rebalance an already short enough string,
        **  need to do it within allocated space.
        */

        slen = strlen(string);
        if (length == 0 || slen < length)
                length = slen;

        while (*ptr != '\0')
        {
                if (backslash)
                {
                        backslash = false;
                        goto increment;
                }

                if (*ptr == '\\')
                        backslash = true;
                else if (*ptr == '(')
                {
                        if (!quoted)
                                parencount++;
                }
                else if (*ptr == ')')
                {
                        if (--parencount < 0)
                                parencount = 0;
                }

                /* Inside a comment, quotes don't matter */
                if (parencount <= 0 && *ptr == '"')
                        quoted = !quoted;

increment:
                /* Check for sufficient space for next character */
                if (length - (ptr - string) <= (size_t) ((backslash ? 1 : 0) +
                                                         parencount +
                                                         (quoted ? 1 : 0)))
                {
                        /* Not enough, backtrack */
                        if (*ptr == '\\')
```

```
260                             backslash = false;
261                     else if (*ptr == '(' && !quoted)
262                             parencount--;
263                     else if (*ptr == '"' && parencount == 0)
264                             quoted = false;
265                     break;
266             }
267             ptr++;
268     }

270     /* Rebalance */
271     while (parencount-- > 0)
272     {
273             if (*ptr != ')')
274             {
275                     modified = true;
276                     *ptr = ')';
277             }
278             ptr++;
279     }
280     if (quoted)
281     {
282             if (*ptr != '"')
283             {
284                     modified = true;
285                     *ptr = '"';
286             }
287             ptr++;
288     }
289     if (*ptr != '\0')
290     {
291             modified = true;
292             *ptr = '\0';
293     }
294     return modified;
295 }

297 /*
298 **  FIND_CHARACTER -- find an unquoted character in an RFC822 string
299 **
300 **      Find an unquoted, non-commented character in an RFC822
301 **      string and return a pointer to its location in the
302 **      string.
303 **
304 **      Parameters:
305 **              string -- the string to search
306 **              character -- the character to find
307 **
308 **      Returns:
309 **              pointer to the character, or
310 **              a pointer to the end of the line if character is not found
311 */

313 char *
314 find_character(string, character)
315     char *string;
316     int character;
317 {
318     bool backslash = false;
319     bool quoted = false;
320     int parencount = 0;

322     while (string != NULL && *string != '\0')
323     {
324             if (backslash)
325             {
```



```
326                     backslash = false;
327                     if (!quoted && character == '\\' && *string == '\\')
328                             break;
329                     string++;
330                     continue;
331             }
332             switch (*string)
333             {
334               case '\\':
335                     backslash = true;
336                     break;

338               case '(':
339                     if (!quoted)
340                             parencount++;
341                     break;

343               case ')':
344                     if (--parencount < 0)
345                             parencount = 0;
346                     break;
347             }

349             /* Inside a comment, nothing matters */
350             if (parencount > 0)
351             {
352                     string++;
353                     continue;
354             }

356             if (*string == '"')
357                     quoted = !quoted;
358             else if (*string == character && !quoted)
359                     break;
360             string++;
361     }

363     /* Return pointer to the character */
364     return string;
365 }

367 /*
368 **  CHECK_BODYTYPE -- check bodytype parameter
369 **
370 **      Parameters:
371 **              bodytype -- bodytype parameter
372 **
373 **      Returns:
374 **              BODYTYPE_* according to parameter
375 **
376 */

378 int
379 check_bodytype(bodytype)
380     char *bodytype;
381 {
382     /* check body type for legality */
383     if (bodytype == NULL)
384             return BODYTYPE_NONE;
385     if (sm_strcasecmp(bodytype, "7BIT") == 0)
386             return BODYTYPE_7BIT;
387     if (sm_strcasecmp(bodytype, "8BITMIME") == 0)
388             return BODYTYPE_8BITMIME;
389     return BODYTYPE_ILLEGAL;
390 }
```

```
/*
**  TRUNCATE_AT_DELIM -- truncate string at a delimiter and append "..."
**
**      Parameters:
**              str -- string to truncate
**              len -- maximum length (including '\0') (0 for unlimited)
**              delim -- delimiter character
**
**      Returns:
**              None.
*/

void
truncate_at_delim(str, len, delim)
        char *str;
        size_t len;
        int delim;
{
        char *p;

        if (str == NULL || len == 0 || strlen(str) < len)
                return;

        *(str + len - 1) = '\0';
        while ((p = strrchr(str, delim)) != NULL)
        {
                *p = '\0';
                if (p - str + 4 < len)
                {
                        *p++ = (char) delim;
                        *p = '\0';
                        (void) sm_strlcat(str, "...", len);
                        return;
                }
        }

        /* Couldn't find a place to append "..." */
        if (len > 3)
                (void) sm_strlcpy(str, "...", len);
        else
                str[0] = '\0';
}

/*
**  XALLOC -- Allocate memory, raise an exception on error
**
**      Parameters:
**              sz -- size of area to allocate.
**
**      Returns:
**              pointer to data region.
**
**      Exceptions:
**              SmHeapOutOfMemory (F:sm.heap) -- cannot allocate memory
**
**      Side Effects:
**              Memory is allocated.
*/

char *
#if SM_HEAP_CHECK
xalloc_tagged(sz, file, line)
        register int sz;
        char *file;
        int line;
#else /* SM_HEAP_CHECK */
```



```
xalloc(sz)
        register int sz;
#endif /* SM_HEAP_CHECK */
{
        register char *p;

        SM_REQUIRE(sz >= 0);

        /* some systems can't handle size zero mallocs */
        if (sz <= 0)
                sz = 1;

        /* scaffolding for testing error handling code */
        sm_xtrap_raise_x(&SmHeapOutOfMemory);

        p = sm_malloc_tagged((unsigned) sz, file, line, sm_heap_group());
        if (p == NULL)
        {
                sm_exc_raise_x(&SmHeapOutOfMemory);
        }
        return p;
}

/*
**  COPYPLIST -- copy list of pointers.
**
**      This routine is the equivalent of strdup for lists of
**      pointers.
**
**      Parameters:
**              list -- list of pointers to copy.
**                      Must be NULL terminated.
**              copycont -- if true, copy the contents of the vector
**                      (which must be a string) also.
**              rpool -- resource pool from which to allocate storage,
**                      or NULL
**
**      Returns:
**              a copy of 'list'.
*/

char **
copyplist(list, copycont, rpool)
        char **list;
        bool copycont;
        SM_RPOOL_T *rpool;
{
        register char **vp;
        register char **newvp;

        for (vp = list; *vp != NULL; vp++)
                continue;

        vp++;

        newvp = (char **) sm_rpool_malloc_x(rpool, (vp - list) * sizeof(*vp));
        memmove((char *) newvp, (char *) list, (int) (vp - list) * sizeof(*vp));

        if (copycont)
        {
                for (vp = newvp; *vp != NULL; vp++)
                        *vp = sm_rpool_strdup_x(rpool, *vp);
        }

        return newvp;
}
```

```
/*
**  COPYQUEUE -- copy address queue.
**
**      This routine is the equivalent of strdup for address queues;
**      addresses marked as QS_IS_DEAD() aren't copied
**
**      Parameters:
**              addr -- list of address structures to copy.
**              rpool -- resource pool from which to allocate storage
**
**      Returns:
**              a copy of 'addr'.
*/

ADDRESS *
copyqueue(addr, rpool)
        ADDRESS *addr;
        SM_RPOOL_T *rpool;
{
        register ADDRESS *newaddr;
        ADDRESS *ret;
        register ADDRESS **tail = &ret;

        while (addr != NULL)
        {
                if (!QS_IS_DEAD(addr->q_state))
                {
                        newaddr = (ADDRESS *) sm_rpool_malloc_x(rpool,
                                                        sizeof(*newaddr));
                        STRUCTCOPY(*addr, *newaddr);
                        *tail = newaddr;
                        tail = &newaddr->q_next;
                }
                addr = addr->q_next;
        }
        *tail = NULL;

        return ret;
}

/*
**  LOG_SENDMAIL_PID -- record sendmail pid and command line.
**
**      Parameters:
**              e -- the current envelope.
**
**      Returns:
**              none.
**
**      Side Effects:
**              writes pidfile, logs command line.
**              keeps file open and locked to prevent overwrite of active file
*/

static SM_FILE_T        *Pidf = NULL;

void
log_sendmail_pid(e)
        ENVELOPE *e;
{
        long sff;
        char pidpath[MAXPATHLEN];
        extern char *CommandLineArgs;

        /* write the pid to the log file for posterity */
```



```
        sff = SFF_NOLINK|SFF_ROOTOK|SFF_REGONLY|SFF_CREAT|SFF_NBLOCK;
        if (TrustedUid != 0 && RealUid == TrustedUid)
                sff |= SFF_OPENASROOT;
        expand(PidFile, pidpath, sizeof(pidpath), e);
        Pidf = safefopen(pidpath, O_WRONLY|O_TRUNC, FileMode, sff);
        if (Pidf == NULL)
        {
                if (errno == EWOULDBLOCK)
                        sm_syslog(LOG_ERR, NOQID,
                                  "unable to write pid to %s: file in use by ano
                                  pidpath);
                else
                        sm_syslog(LOG_ERR, NOQID,
                                  "unable to write pid to %s: %s",
                                  pidpath, sm_errstring(errno));
        }
        else
        {
                PidFilePid = getpid();

                /* write the process id on line 1 */
                (void) sm_io_fprintf(Pidf, SM_TIME_DEFAULT, "%ld\n",
                                     (long) PidFilePid);

                /* line 2 contains all command line flags */
                (void) sm_io_fprintf(Pidf, SM_TIME_DEFAULT, "%s\n",
                                     CommandLineArgs);

                /* flush */
                (void) sm_io_flush(Pidf, SM_TIME_DEFAULT);

                /*
                **  Leave pid file open until process ends
                **  so it's not overwritten by another
                **  process.
                */
        }
        if (LogLevel > 9)
                sm_syslog(LOG_INFO, NOQID, "started as: %s", CommandLineArgs);
}

/*
**  CLOSE_SENDMAIL_PID -- close sendmail pid file
**
**      Parameters:
**              none.
**
**      Returns:
**              none.
*/

void
close_sendmail_pid()
{
        if (Pidf == NULL)
                return;

        (void) sm_io_close(Pidf, SM_TIME_DEFAULT);
        Pidf = NULL;
}

/*
**  SET_DELIVERY_MODE -- set and record the delivery mode
**
**      Parameters:
**              mode -- delivery mode
```

```
656 **              e -- the current envelope.
657 **
658 **      Returns:
659 **              none.
660 **
661 **      Side Effects:
662 **              sets {deliveryMode} macro
663 */

665 void
666 set_delivery_mode(mode, e)
667         int mode;
668         ENVELOPE *e;
669 {
670         char buf[2];

672         e->e_sendmode = (char) mode;
673         buf[0] = (char) mode;
674         buf[1] = '\0';
675         macdefine(&e->e_macro, A_TEMP, macid("{deliveryMode}"), buf);
676 }

678 /*
679 **  SET_OP_MODE -- set and record the op mode
680 **
681 **      Parameters:
682 **              mode -- op mode
683 **              e -- the current envelope.
684 **
685 **      Returns:
686 **              none.
687 **
688 **      Side Effects:
689 **              sets {opMode} macro
690 */

692 void
693 set_op_mode(mode)
694         int mode;
695 {
696         char buf[2];
697         extern ENVELOPE BlankEnvelope;

699         OpMode = (char) mode;
700         buf[0] = (char) mode;
701         buf[1] = '\0';
702         macdefine(&BlankEnvelope.e_macro, A_TEMP, MID_OPMODE, buf);
703 }

705 /*
706 **  PRINTAV -- print argument vector.
707 **
708 **      Parameters:
709 **              fp -- output file pointer.
710 **              av -- argument vector.
711 **
712 **      Returns:
713 **              none.
714 **
715 **      Side Effects:
716 **              prints av.
717 */

719 void
720 printav(fp, av)
721         SM_FILE_T *fp;
```



```
722         char **av;
723 {
724         while (*av != NULL)
725         {
726                 if (tTd(0, 44))
727                         sm_dprintf("\n\t%08lx=", (unsigned long) *av);
728                 else
729                         (void) sm_io_putc(fp, SM_TIME_DEFAULT, ' ');
730                 if (tTd(0, 99))
731                         sm_dprintf("%s", str2prt(*av++));
732                 else
733                         xputs(fp, *av++);
734         }
735         (void) sm_io_putc(fp, SM_TIME_DEFAULT, '\n');
736 }

738 /*
739 **  XPUTS -- put string doing control escapes.
740 **
741 **      Parameters:
742 **              fp -- output file pointer.
743 **              s -- string to put.
744 **
745 **      Returns:
746 **              none.
747 **
748 **      Side Effects:
749 **              output to stdout
750 */

752 void
753 xputs(fp, s)
754         SM_FILE_T *fp;
755         const char *s;
756 {
757         int c;
758         struct metamac *mp;
759         bool shiftout = false;
760         extern struct metamac MetaMacros[];
761         static SM_DEBUG_T DebugANSI = SM_DEBUG_INITIALIZER("ANSI",
762                 "@(#)$Debug: ANSI - enable reverse video in debug output $");

764         /*
765         **  TermEscape is set here, rather than in main(),
766         **  because ANSI mode can be turned on or off at any time
767         **  if we are in -bt rule testing mode.
768         */

770         if (sm_debug_unknown(&DebugANSI))
771         {
772                 if (sm_debug_active(&DebugANSI, 1))
773                 {
774                         TermEscape.te_rv_on = "\033[7m";
775                         TermEscape.te_normal = "\033[0m";
776                 }
777                 else
778                 {
779                         TermEscape.te_rv_on = "";
780                         TermEscape.te_normal = "";
781                 }
782         }

784         if (s == NULL)
785         {
786                 (void) sm_io_fprintf(fp, SM_TIME_DEFAULT, "%s<null>%s",
787                                      TermEscape.te_rv_on, TermEscape.te_normal);
```

```
788                 return;
789         }
790         while ((c = (*s++ & 0377)) != '\0')
791         {
792                 if (shiftout)
793                 {
794                         (void) sm_io_fprintf(fp, SM_TIME_DEFAULT, "%s",
795                                              TermEscape.te_normal);
796                         shiftout = false;
797                 }
798                 if (!isascii(c) && !tTd(84, 1))
799                 {
800                         if (c == MATCHREPL)
801                         {
802                                 (void) sm_io_fprintf(fp, SM_TIME_DEFAULT,
803                                                      "%s$",
804                                                      TermEscape.te_rv_on);
805                                 shiftout = true;
806                                 if (*s == '\0')
807                                         continue;
808                                 c = *s++ & 0377;
809                                 goto printchar;
810                         }
811                         if (c == MACROEXPAND || c == MACRODEXPAND)
812                         {
813                                 (void) sm_io_fprintf(fp, SM_TIME_DEFAULT,
814                                                      "%s$",
815                                                      TermEscape.te_rv_on);
816                                 if (c == MACRODEXPAND)
817                                         (void) sm_io_putc(fp,
818                                                           SM_TIME_DEFAULT, '&');
819                                 shiftout = true;
820                                 if (*s == '\0')
821                                         continue;
822                                 if (strchr("=~&?", *s) != NULL)
823                                         (void) sm_io_putc(fp,
824                                                           SM_TIME_DEFAULT,
825                                                           *s++);
826                                 if (bitset(0200, *s))
827                                         (void) sm_io_fprintf(fp,
828                                                              SM_TIME_DEFAULT,
829                                                              "{%s}",
830                                                              macname(bitidx(*s++
831                                 else
832                                         (void) sm_io_fprintf(fp,
833                                                              SM_TIME_DEFAULT,
834                                                              "%c",
835                                                              *s++);
836                                 continue;
837                         }
838                         for (mp = MetaMacros; mp->metaname != '\0'; mp++)
839                         {
840                                 if (bitidx(mp->metaval) == c)
841                                 {
842                                         (void) sm_io_fprintf(fp,
843                                                              SM_TIME_DEFAULT,
844                                                              "%s$%c",
845                                                              TermEscape.te_rv_on
846                                                              mp->metaname);
847                                         shiftout = true;
848                                         break;
849                                 }
850                         }
851                         if (c == MATCHCLASS || c == MATCHNCLASS)
852                         {
853                                 if (bitset(0200, *s))
```



```
854                                         (void) sm_io_fprintf(fp,
855                                                              SM_TIME_DEFAULT,
856                                                              "{%s}",
857                                                              macname(bitidx(*s++
858                                 else if (*s != '\0')
859                                         (void) sm_io_fprintf(fp,
860                                                              SM_TIME_DEFAULT,
861                                                              "%c",
862                                                              *s++);
863                                 }
864                                 if (mp->metaname != '\0')
865                                         continue;
866
867                                 /* unrecognized meta character */
868                                 (void) sm_io_fprintf(fp, SM_TIME_DEFAULT, "%sM-",
869                                                      TermEscape.te_rv_on);
870                                 shiftout = true;
871                                 c &= 0177;
872                         }
873         printchar:
874                         if (isascii(c) && isprint(c))
875                         {
876                                 (void) sm_io_putc(fp, SM_TIME_DEFAULT, c);
877                                 continue;
878                         }
879
880                         /* wasn't a meta-macro -- find another way to print it */
881                         switch (c)
882                         {
883                           case '\n':
884                                 c = 'n';
885                                 break;
886
887                           case '\r':
888                                 c = 'r';
889                                 break;
890
891                           case '\t':
892                                 c = 't';
893                                 break;
894                         }
895                         if (!shiftout)
896                         {
897                                 (void) sm_io_fprintf(fp, SM_TIME_DEFAULT, "%s",
898                                                      TermEscape.te_rv_on);
899                                 shiftout = true;
900                         }
901                         if (isascii(c) && isprint(c))
902                         {
903                                 (void) sm_io_putc(fp, SM_TIME_DEFAULT, '\\');
904                                 (void) sm_io_putc(fp, SM_TIME_DEFAULT, c);
905                         }
906                         else if (tTd(84, 2))
907                                 (void) sm_io_fprintf(fp, SM_TIME_DEFAULT, " %o ", c);
908                         else if (tTd(84, 1))
909                                 (void) sm_io_fprintf(fp, SM_TIME_DEFAULT, " %#x ", c);
910                         else if (!isascii(c) && !tTd(84, 1))
911                         {
912                                 (void) sm_io_putc(fp, SM_TIME_DEFAULT, '^');
913                                 (void) sm_io_putc(fp, SM_TIME_DEFAULT, c ^ 0100);
914                         }
915                 }
916                 if (shiftout)
917                         (void) sm_io_fprintf(fp, SM_TIME_DEFAULT, "%s",
918                                              TermEscape.te_normal);
919                 (void) sm_io_flush(fp, SM_TIME_DEFAULT);
```

```
}

/*
**  MAKELOWER -- Translate a line into lower case
**
**      Parameters:
**              p -- the string to translate.  If NULL, return is
**                      immediate.
**
**      Returns:
**              none.
**
**      Side Effects:
**              String pointed to by p is translated to lower case.
*/

void
makelower(p)
        register char *p;
{
        register char c;

        if (p == NULL)
                return;
        for (; (c = *p) != '\0'; p++)
                if (isascii(c) && isupper(c))
                        *p = tolower(c);
}

/*
**  FIXCRLF -- fix <CR><LF> in line.
**
**      Looks for the <CR><LF> combination and turns it into the
**      UNIX canonical <NL> character.  It only takes one line,
**      i.e., it is assumed that the first <NL> found is the end
**      of the line.
**
**      Parameters:
**              line -- the line to fix.
**              stripnl -- if true, strip the newline also.
**
**      Returns:
**              none.
**
**      Side Effects:
**              line is changed in place.
*/

void
fixcrlf(line, stripnl)
        char *line;
        bool stripnl;
{
        register char *p;

        p = strchr(line, '\n');
        if (p == NULL)
                return;
        if (p > line && p[-1] == '\r')
                p--;
        if (!stripnl)
                *p++ = '\n';
        *p = '\0';
}

/*
```



```
**  PUTLINE -- put a line like fputs obeying SMTP conventions
**
**      This routine always guarantees outputing a newline (or CRLF,
**      as appropriate) at the end of the string.
**
**      Parameters:
**              l -- line to put.
**              mci -- the mailer connection information.
**
**      Returns:
**              true iff line was written successfully
**
**      Side Effects:
**              output of l to mci->mci_out.
*/

bool
putline(l, mci)
        register char *l;
        register MCI *mci;
{
        return putxline(l, strlen(l), mci, PXLF_MAPFROM);
}

/*
**  PUTXLINE -- putline with flags bits.
**
**      This routine always guarantees outputing a newline (or CRLF,
**      as appropriate) at the end of the string.
**
**      Parameters:
**              l -- line to put.
**              len -- the length of the line.
**              mci -- the mailer connection information.
**              pxflags -- flag bits:
**                  PXLF_MAPFROM -- map From_ to >From_.
**                  PXLF_STRIP8BIT -- strip 8th bit.
**                  PXLF_HEADER -- map bare newline in header to newline space.
**                  PXLF_NOADDEOL -- don't add an EOL if one wasn't present.
**                  PXLF_STRIPMQUOTE -- strip METAQUOTE bytes.
**
**      Returns:
**              true iff line was written successfully
**
**      Side Effects:
**              output of l to mci->mci_out.
*/


#define PUTX(limit)                                                     \
        do                                                              \
        {                                                               \
                quotenext = false;                                      \
                while (l < limit)                                       \
                {                                                       \
                        unsigned char c = (unsigned char) *l++;         \
                                                                        \
                        if (bitset(PXLF_STRIPMQUOTE, pxflags) &&        \
                            !quotenext && c == METAQUOTE)               \
                        {                                               \
                                quotenext = true;                       \
                                continue;                               \
                        }                                               \
                        quotenext = false;                              \
                        if (strip8bit)                                  \
                                c &= 0177;                              \
```

```
1052                        if (sm_io_putc(mci->mci_out, SM_TIME_DEFAULT,   \
1053                                       c) == SM_IO_EOF)                 \
1054                        {                                               \
1055                                dead = true;                            \
1056                                break;                                  \
1057                        }                                               \
1058                        if (TrafficLogFile != NULL)                     \
1059                                (void) sm_io_putc(TrafficLogFile,       \
1060                                                  SM_TIME_DEFAULT,      \
1061                                                  c);                   \
1062                }                                                       \
1063        } while (0)

1065 bool
1066 putxline(l, len, mci, pxflags)
1067         register char *l;
1068         size_t len;
1069         register MCI *mci;
1070         int pxflags;
1071 {
1072         register char *p, *end;
1073         int slop;
1074         bool dead, quotenext, strip8bit;

1076         /* strip out 0200 bits -- these can look like TELNET protocol */
1077         strip8bit = bitset(MCIF_7BIT, mci->mci_flags) ||
1078                     bitset(PXLF_STRIP8BIT, pxflags);
1079         dead = false;
1080         slop = 0;

1082         end = l + len;
1083         do
1084         {
1085                 bool noeol = false;

1087                 /* find the end of the line */
1088                 p = memchr(l, '\n', end - l);
1089                 if (p == NULL)
1090                 {
1091                         p = end;
1092                         noeol = true;
1093                 }

1095                 if (TrafficLogFile != NULL)
1096                         (void) sm_io_fprintf(TrafficLogFile, SM_TIME_DEFAULT,
1097                                              "%05d >>> ", (int) CurrentPid);

1099                 /* check for line overflow */
1100                 while (mci->mci_mailer->m_linelimit > 0 &&
1101                        (p - l + slop) > mci->mci_mailer->m_linelimit)
1102                 {
1103                         register char *q = &l[mci->mci_mailer->m_linelimit - slo

1105                         if (l[0] == '.' && slop == 0 &&
1106                             bitset(M_XDOT, mci->mci_mailer->m_flags))
1107                         {
1108                                 if (sm_io_putc(mci->mci_out, SM_TIME_DEFAULT,
1109                                                '.') == SM_IO_EOF)
1110                                         dead = true;
1111                                 if (TrafficLogFile != NULL)
1112                                         (void) sm_io_putc(TrafficLogFile,
1113                                                           SM_TIME_DEFAULT, '.');
1114                         }
1115                         else if (l[0] == 'F' && slop == 0 &&
1116                                  bitset(PXLF_MAPFROM, pxflags) &&
1117                                  strncmp(l, "From ", 5) == 0 &&
```



```
1118                                  bitset(M_ESCFROM, mci->mci_mailer->m_flags))
1119                         {
1120                                 if (sm_io_putc(mci->mci_out, SM_TIME_DEFAULT,
1121                                                '>') == SM_IO_EOF)
1122                                         dead = true;
1123                                 if (TrafficLogFile != NULL)
1124                                         (void) sm_io_putc(TrafficLogFile,
1125                                                           SM_TIME_DEFAULT,
1126                                                           '>');
1127                         }
1128                         if (dead)
1129                                 break;

1131                         PUTX(q);
1132                         if (dead)
1133                                 break;

1135                         if (sm_io_putc(mci->mci_out, SM_TIME_DEFAULT,
1136                                        '!') == SM_IO_EOF ||
1137                             sm_io_fputs(mci->mci_out, SM_TIME_DEFAULT,
1138                                         mci->mci_mailer->m_eol) == SM_IO_EOF ||
1139                             sm_io_putc(mci->mci_out, SM_TIME_DEFAULT,
1140                                        ' ') == SM_IO_EOF)
1141                         {
1142                                 dead = true;
1143                                 break;
1144                         }
1145                         if (TrafficLogFile != NULL)
1146                         {
1147                                 (void) sm_io_fprintf(TrafficLogFile,
1148                                                      SM_TIME_DEFAULT,
1149                                                      "!\n%05d >>>  ",
1150                                                      (int) CurrentPid);
1151                         }
1152                         slop = 1;
1153                 }

1155                 if (dead)
1156                         break;

1158                 /* output last part */
1159                 if (l[0] == '.' && slop == 0 &&
1160                     bitset(M_XDOT, mci->mci_mailer->m_flags) &&
1161                     !bitset(MCIF_INLONGLINE, mci->mci_flags))
1162                 {
1163                         if (sm_io_putc(mci->mci_out, SM_TIME_DEFAULT, '.') ==
1164                             SM_IO_EOF)
1165                         {
1166                                 dead = true;
1167                                 break;
1168                         }
1169                         if (TrafficLogFile != NULL)
1170                                 (void) sm_io_putc(TrafficLogFile,
1171                                                   SM_TIME_DEFAULT, '.');
1172                 }
1173                 else if (l[0] == 'F' && slop == 0 &&
1174                          bitset(PXLF_MAPFROM, pxflags) &&
1175                          strncmp(l, "From ", 5) == 0 &&
1176                          bitset(M_ESCFROM, mci->mci_mailer->m_flags) &&
1177                          !bitset(MCIF_INLONGLINE, mci->mci_flags))
1178                 {
1179                         if (sm_io_putc(mci->mci_out, SM_TIME_DEFAULT, '>') ==
1180                             SM_IO_EOF)
1181                         {
1182                                 dead = true;
1183                                 break;
```

```
1184                        }
1185                        if (TrafficLogFile != NULL)
1186                                (void) sm_io_putc(TrafficLogFile,
1187                                                  SM_TIME_DEFAULT, '>');
1188                }
1189                PUTX(p);
1190                if (dead)
1191                        break;

1193                if (TrafficLogFile != NULL)
1194                        (void) sm_io_putc(TrafficLogFile, SM_TIME_DEFAULT,
1195                                          '\n');
1196                if ((!bitset(PXLF_NOADDEOL, pxflags) || !noeol))
1197                {
1198                        mci->mci_flags &= ~MCIF_INLONGLINE;
1199                        if (sm_io_fputs(mci->mci_out, SM_TIME_DEFAULT,
1200                                        mci->mci_mailer->m_eol) == SM_IO_EOF)
1201                        {
1202                                dead = true;
1203                                break;
1204                        }
1205                }
1206                else
1207                        mci->mci_flags |= MCIF_INLONGLINE;

1209                if (l < end && *l == '\n')
1210                {
1211                        if (*++l != ' ' && *l != '\t' && *l != '\0' &&
1212                            bitset(PXLF_HEADER, pxflags))
1213                        {
1214                                if (sm_io_putc(mci->mci_out, SM_TIME_DEFAULT,
1215                                               ' ') == SM_IO_EOF)
1216                                {
1217                                        dead = true;
1218                                        break;
1219                                }

1221                                if (TrafficLogFile != NULL)
1222                                        (void) sm_io_putc(TrafficLogFile,
1223                                                          SM_TIME_DEFAULT, ' ');
1224                        }
1225                }

1227        } while (l < end);
1228        return !dead;
1229 }

1231 /*
1232 **  XUNLINK -- unlink a file, doing logging as appropriate.
1233 **
1234 **      Parameters:
1235 **              f -- name of file to unlink.
1236 **
1237 **      Returns:
1238 **              return value of unlink()
1239 **
1240 **      Side Effects:
1241 **              f is unlinked.
1242 */

1244 int
1245 xunlink(f)
1246        char *f;
1247 {
1248        register int i;
1249        int save_errno;
```



```
1251        if (LogLevel > 98)
1252                sm_syslog(LOG_DEBUG, CurEnv->e_id, "unlink %s", f);

1254        i = unlink(f);
1255        save_errno = errno;
1256        if (i < 0 && LogLevel > 97)
1257                sm_syslog(LOG_DEBUG, CurEnv->e_id, "%s: unlink-fail %d",
1258                          f, errno);
1259        if (i >= 0)
1260                SYNC_DIR(f, false);
1261        errno = save_errno;
1262        return i;
1263 }

1265 /*
1266 **  SFGETS -- "safe" fgets -- times out and ignores random interrupts.
1267 **
1268 **      Parameters:
1269 **              buf -- place to put the input line.
1270 **              siz -- size of buf.
1271 **              fp -- file to read from.
1272 **              timeout -- the timeout before error occurs.
1273 **              during -- what we are trying to read (for error messages).
1274 **
1275 **      Returns:
1276 **              NULL on error (including timeout).  This may also leave
1277 **                      buf containing a null string.
1278 **              buf otherwise.
1279 */


1282 char *
1283 sfgets(buf, siz, fp, timeout, during)
1284        char *buf;
1285        int siz;
1286        SM_FILE_T *fp;
1287        time_t timeout;
1288        char *during;
1289 {
1290        register char *p;
1291        int save_errno;
1292        int io_timeout;

1294        SM_REQUIRE(siz > 0);
1295        SM_REQUIRE(buf != NULL);

1297        if (fp == NULL)
1298        {
1299                buf[0] = '\0';
1300                errno = EBADF;
1301                return NULL;
1302        }

1304        /* try to read */
1305        p = NULL;
1306        errno = 0;

1308        /* convert the timeout to sm_io notation */
1309        io_timeout = (timeout <= 0) ? SM_TIME_DEFAULT : timeout * 1000;
1310        while (!sm_io_eof(fp) && !sm_io_error(fp))
1311        {
1312                errno = 0;
1313                p = sm_io_fgets(fp, io_timeout, buf, siz);
1314                if (p == NULL && errno == EAGAIN)
1315                {
```

```
1316                        /* The sm_io_fgets() call timedout */
1317                        if (LogLevel > 1)
1318                                sm_syslog(LOG_NOTICE, CurEnv->e_id,
1319                                          "timeout waiting for input from %.100s
1320                                          CURHOSTNAME,
1321                                          during);
1322                        buf[0] = '\0';
1323 #if XDEBUG
1324                        checkfd012(during);
1325 #endif /* XDEBUG */
1326                        if (TrafficLogFile != NULL)
1327                                (void) sm_io_fprintf(TrafficLogFile,
1328                                                     SM_TIME_DEFAULT,
1329                                                     "%05d <<< [TIMEOUT]\n",
1330                                                     (int) CurrentPid);
1331                        errno = ETIMEDOUT;
1332                        return NULL;
1333                }
1334                if (p != NULL || errno != EINTR)
1335                        break;
1336                (void) sm_io_clearerr(fp);
1337        }
1338        save_errno = errno;

1340        /* clean up the books and exit */
1341        LineNumber++;
1342        if (p == NULL)
1343        {
1344                buf[0] = '\0';
1345                if (TrafficLogFile != NULL)
1346                        (void) sm_io_fprintf(TrafficLogFile, SM_TIME_DEFAULT,
1347                                             "%05d <<< [EOF]\n",
1348                                             (int) CurrentPid);
1349                errno = save_errno;
1350                return NULL;
1351        }
1352        if (TrafficLogFile != NULL)
1353                (void) sm_io_fprintf(TrafficLogFile, SM_TIME_DEFAULT,
1354                                     "%05d <<< %s", (int) CurrentPid, buf);
1355        if (SevenBitInput)
1356        {
1357                for (p = buf; *p != '\0'; p++)
1358                        *p &= ~0200;
1359        }
1360        else if (!HasEightBits)
1361        {
1362                for (p = buf; *p != '\0'; p++)
1363                {
1364                        if (bitset(0200, *p))
1365                        {
1366                                HasEightBits = true;
1367                                break;
1368                        }
1369                }
1370        }
1371        return buf;
1372 }

1374 /*
1375 **  FGETFOLDED -- like fgets, but knows about folded lines.
1376 **
1377 **      Parameters:
1378 **              buf -- place to put result.
1379 **              np -- pointer to bytes available; will be updated with
1380 **                      the actual buffer size (not number of bytes filled)
1381 **                      on return.
```



```
1382 **              f -- file to read from.
1383 **
1384 **      Returns:
1385 **              input line(s) on success, NULL on error or SM_IO_EOF.
1386 **              This will normally be buf -- unless the line is too
1387 **                      long, when it will be sm_malloc_x()ed.
1388 **
1389 **      Side Effects:
1390 **              buf gets lines from f, with continuation lines (lines
1391 **              with leading white space) appended.  CRLF's are mapped
1392 **              into single newlines.  Any trailing NL is stripped.
1393 */

1395 char *
1396 fgetfolded(buf, np, f)
1397        char *buf;
1398        int *np;
1399        SM_FILE_T *f;
1400 {
1401        register char *p = buf;
1402        char *bp = buf;
1403        register int i;
1404        int n;

1406        SM_REQUIRE(np != NULL);
1407        n = *np;
1408        SM_REQUIRE(n > 0);
1409        SM_REQUIRE(buf != NULL);
1410        if (f == NULL)
1411        {
1412                buf[0] = '\0';
1413                errno = EBADF;
1414                return NULL;
1415        }

1417        n--;
1418        while ((i = sm_io_getc(f, SM_TIME_DEFAULT)) != SM_IO_EOF)
1419        {
1420                if (i == '\r')
1421                {
1422                        i = sm_io_getc(f, SM_TIME_DEFAULT);
1423                        if (i != '\n')
1424                        {
1425                                if (i != SM_IO_EOF)
1426                                        (void) sm_io_ungetc(f, SM_TIME_DEFAULT,
1427                                                            i);
1428                                i = '\r';
1429                        }
1430                }
1431                if (--n <= 0)
1432                {
1433                        /* allocate new space */
1434                        char *nbp;
1435                        int nn;

1437                        nn = (p - bp);
1438                        if (nn < MEMCHUNKSIZE)
1439                                nn *= 2;
1440                        else
1441                                nn += MEMCHUNKSIZE;
1442                        nbp = sm_malloc_x(nn);
1443                        memmove(nbp, bp, p - bp);
1444                        p = &nbp[p - bp];
1445                        if (bp != buf)
1446                                sm_free(bp);
1447                        bp = nbp;
```

```
1448                        n = nn - (p - bp);
1449                        *np = nn;
1450                }
1451                *p++ = i;
1452                if (i == '\n')
1453                {
1454                        LineNumber++;
1455                        i = sm_io_getc(f, SM_TIME_DEFAULT);
1456                        if (i != SM_IO_EOF)
1457                                (void) sm_io_ungetc(f, SM_TIME_DEFAULT, i);
1458                        if (i != ' ' && i != '\t')
1459                                break;
1460                }
1461        }
1462        if (p == bp)
1463                return NULL;
1464        if (p[-1] == '\n')
1465                p--;
1466        *p = '\0';
1467        return bp;
1468 }

1470 /*
1471 **  CURTIME -- return current time.
1472 **
1473 **      Parameters:
1474 **              none.
1475 **
1476 **      Returns:
1477 **              the current time.
1478 */

1480 time_t
1481 curtime()
1482 {
1483        auto time_t t;

1485        (void) time(&t);
1486        return t;
1487 }

1489 /*
1490 **  ATOBOOL -- convert a string representation to boolean.
1491 **
1492 **      Defaults to false
1493 **
1494 **      Parameters:
1495 **              s -- string to convert.  Takes "tTyY", empty, and NULL as true,
1496 **                      others as false.
1497 **
1498 **      Returns:
1499 **              A boolean representation of the string.
1500 */

1502 bool
1503 atobool(s)
1504        register char *s;
1505 {
1506        if (s == NULL || *s == '\0' || strchr("tTyY", *s) != NULL)
1507                return true;
1508        return false;
1509 }

1511 /*
1512 **  ATOOCT -- convert a string representation to octal.
1513 **
```



```
1514 **      Parameters:
1515 **              s -- string to convert.
1516 **
1517 **      Returns:
1518 **              An integer representing the string interpreted as an
1519 **              octal number.
1520 */

1522 int
1523 atooct(s)
1524        register char *s;
1525 {
1526        register int i = 0;

1528        while (*s >= '0' && *s <= '7')
1529                i = (i << 3) | (*s++ - '0');
1530        return i;
1531 }

1533 /*
1534 **  BITINTERSECT -- tell if two bitmaps intersect
1535 **
1536 **      Parameters:
1537 **              a, b -- the bitmaps in question
1538 **
1539 **      Returns:
1540 **              true if they have a non-null intersection
1541 **              false otherwise
1542 */

1544 bool
1545 bitintersect(a, b)
1546        BITMAP256 a;
1547        BITMAP256 b;
1548 {
1549        int i;

1551        for (i = BITMAPBYTES / sizeof(int); --i >= 0; )
1552        {
1553                if ((a[i] & b[i]) != 0)
1554                        return true;
1555        }
1556        return false;
1557 }

1559 /*
1560 **  BITZEROP -- tell if a bitmap is all zero
1561 **
1562 **      Parameters:
1563 **              map -- the bit map to check
1564 **
1565 **      Returns:
1566 **              true if map is all zero.
1567 **              false if there are any bits set in map.
1568 */

1570 bool
1571 bitzerop(map)
1572        BITMAP256 map;
1573 {
1574        int i;

1576        for (i = BITMAPBYTES / sizeof(int); --i >= 0; )
1577        {
1578                if (map[i] != 0)
1579                        return false;
```

```
1580         }
1581         return true;
1582 }

1584 /*
1585 **  STRCONTAINEDIN -- tell if one string is contained in another
1586 **
1587 **      Parameters:
1588 **              icase -- ignore case?
1589 **              a -- possible substring.
1590 **              b -- possible superstring.
1591 **
1592 **      Returns:
1593 **              true if a is contained in b (case insensitive).
1594 **              false otherwise.
1595 */

1597 bool
1598 strcontainedin(icase, a, b)
1599         bool icase;
1600         register char *a;
1601         register char *b;
1602 {
1603         int la;
1604         int lb;
1605         int c;

1607         la = strlen(a);
1608         lb = strlen(b);
1609         c = *a;
1610         if (icase && isascii(c) && isupper(c))
1611                 c = tolower(c);
1612         for (; lb-- >= la; b++)
1613         {
1614                 if (icase)
1615                 {
1616                         if (*b != c &&
1617                             isascii(*b) && isupper(*b) && tolower(*b) != c)
1618                                 continue;
1619                         if (sm_strncasecmp(a, b, la) == 0)
1620                                 return true;
1621                 }
1622                 else
1623                 {
1624                         if (*b != c)
1625                                 continue;
1626                         if (strncmp(a, b, la) == 0)
1627                                 return true;
1628                 }
1629         }
1630         return false;
1631 }

1633 /*
1634 **  CHECKFD012 -- check low numbered file descriptors
1635 **
1636 **      File descriptors 0, 1, and 2 should be open at all times.
1637 **      This routine verifies that, and fixes it if not true.
1638 **
1639 **      Parameters:
1640 **              where -- a tag printed if the assertion failed
1641 **
1642 **      Returns:
1643 **              none
1644 */
```



```
1646 void
1647 checkfd012(where)
1648         char *where;
1649 {
1650 #if XDEBUG
1651         register int i;

1653         for (i = 0; i < 3; i++)
1654                 fill_fd(i, where);
1655 #endif /* XDEBUG */
1656 }

1658 /*
1659 **  CHECKFDOPEN -- make sure file descriptor is open -- for extended debugging
1660 **
1661 **      Parameters:
1662 **              fd -- file descriptor to check.
1663 **              where -- tag to print on failure.
1664 **
1665 **      Returns:
1666 **              none.
1667 */

1669 void
1670 checkfdopen(fd, where)
1671         int fd;
1672         char *where;
1673 {
1674 #if XDEBUG
1675         struct stat st;

1677         if (fstat(fd, &st) < 0 && errno == EBADF)
1678         {
1679                 syserr("checkfdopen(%d): %s not open as expected!", fd, where);
1680                 printopenfds(true);
1681         }
1682 #endif /* XDEBUG */
1683 }

1685 /*
1686 **  CHECKFDS -- check for new or missing file descriptors
1687 **
1688 **      Parameters:
1689 **              where -- tag for printing.  If null, take a base line.
1690 **
1691 **      Returns:
1692 **              none
1693 **
1694 **      Side Effects:
1695 **              If where is set, shows changes since the last call.
1696 */

1698 void
1699 checkfds(where)
1700         char *where;
1701 {
1702         int maxfd;
1703         register int fd;
1704         bool printhdr = true;
1705         int save_errno = errno;
1706         static BITMAP256 baseline;
1707         extern int DtableSize;

1709         if (DtableSize > BITMAPBITS)
1710                 maxfd = BITMAPBITS;
1711         else
```

```
1712                maxfd = DtableSize;
1713        if (where == NULL)
1714                clrbitmap(baseline);

1716        for (fd = 0; fd < maxfd; fd++)
1717        {
1718                struct stat stbuf;

1720                if (fstat(fd, &stbuf) < 0 && errno != EOPNOTSUPP)
1721                {
1722                        if (!bitnset(fd, baseline))
1723                                continue;
1724                        clrbitn(fd, baseline);
1725                }
1726                else if (!bitnset(fd, baseline))
1727                        setbitn(fd, baseline);
1728                else
1729                        continue;

1731                /* file state has changed */
1732                if (where == NULL)
1733                        continue;
1734                if (printhdr)
1735                {
1736                        sm_syslog(LOG_DEBUG, CurEnv->e_id,
1737                                  "%s: changed fds:",
1738                                  where);
1739                        printhdr = false;
1740                }
1741                dumpfd(fd, true, true);
1742        }
1743        errno = save_errno;
1744 }

1746 /*
1747 **  PRINTOPENFDS -- print the open file descriptors (for debugging)
1748 **
1749 **      Parameters:
1750 **              logit -- if set, send output to syslog; otherwise
1751 **                      print for debugging.
1752 **
1753 **      Returns:
1754 **              none.
1755 */

1757 #if NETINET || NETINET6
1758 # include <arpa/inet.h>
1759 #endif /* NETINET || NETINET6 */

1761 void
1762 printopenfds(logit)
1763        bool logit;
1764 {
1765        register int fd;
1766        extern int DtableSize;

1768        for (fd = 0; fd < DtableSize; fd++)
1769                dumpfd(fd, false, logit);
1770 }

1772 /*
1773 **  DUMPFD -- dump a file descriptor
1774 **
1775 **      Parameters:
1776 **              fd -- the file descriptor to dump.
1777 **              printclosed -- if set, print a notification even if
```



```
1778 **                      it is closed; otherwise print nothing.
1779 **              logit -- if set, use sm_syslog instead of sm_dprintf()
1780 **
1781 **      Returns:
1782 **              none.
1783 */

1785 void
1786 dumpfd(fd, printclosed, logit)
1787        int fd;
1788        bool printclosed;
1789        bool logit;
1790 {
1791        register char *p;
1792        char *hp;
1793 #ifdef S_IFSOCK
1794        SOCKADDR sa;
1795 #endif /* S_IFSOCK */
1796        auto SOCKADDR_LEN_T slen;
1797        int i;
1798 #if STAT64 > 0
1799        struct stat64 st;
1800 #else /* STAT64 > 0 */
1801        struct stat st;
1802 #endif /* STAT64 > 0 */
1803        char buf[200];

1805        p = buf;
1806        (void) sm_snprintf(p, SPACELEFT(buf, p), "%3d: ", fd);
1807        p += strlen(p);

1809        if (
1810 #if STAT64 > 0
1811            fstat64(fd, &st)
1812 #else /* STAT64 > 0 */
1813            fstat(fd, &st)
1814 #endif /* STAT64 > 0 */
1815            < 0)
1816        {
1817                if (errno != EBADF)
1818                {
1819                        (void) sm_snprintf(p, SPACELEFT(buf, p),
1820                                "CANNOT STAT (%s)",
1821                                sm_errstring(errno));
1822                        goto printit;
1823                }
1824                else if (printclosed)
1825                {
1826                        (void) sm_snprintf(p, SPACELEFT(buf, p), "CLOSED");
1827                        goto printit;
1828                }
1829                return;
1830        }

1832        i = fcntl(fd, F_GETFL, 0);
1833        if (i != -1)
1834        {
1835                (void) sm_snprintf(p, SPACELEFT(buf, p), "fl=0x%x, ", i);
1836                p += strlen(p);
1837        }

1839        (void) sm_snprintf(p, SPACELEFT(buf, p), "mode=%o: ",
1840                           (int) st.st_mode);
1841        p += strlen(p);
1842        switch (st.st_mode & S_IFMT)
1843        {
```

```
#ifdef S_IFSOCK
          case S_IFSOCK:
                (void) sm_snprintf(p, SPACELEFT(buf, p), "SOCK ");
                p += strlen(p);
                memset(&sa, '\0', sizeof(sa));
                slen = sizeof(sa);
                if (getsockname(fd, &sa.sa, &slen) < 0)
                        (void) sm_snprintf(p, SPACELEFT(buf, p), "(%s)",
                                sm_errstring(errno));
                else
                {
                        hp = hostnamebyanyaddr(&sa);
                        if (hp == NULL)
                        {
                                /* EMPTY */
                                /* do nothing */
                        }
# if NETINET
                        else if (sa.sa.sa_family == AF_INET)
                                (void) sm_snprintf(p, SPACELEFT(buf, p),
                                        "%s/%d", hp, ntohs(sa.sin.sin_port));
# endif /* NETINET */
# if NETINET6
                        else if (sa.sa.sa_family == AF_INET6)
                                (void) sm_snprintf(p, SPACELEFT(buf, p),
                                        "%s/%d", hp, ntohs(sa.sin6.sin6_port));
# endif /* NETINET6 */
                        else
                                (void) sm_snprintf(p, SPACELEFT(buf, p),
                                        "%s", hp);
                }
                p += strlen(p);
                (void) sm_snprintf(p, SPACELEFT(buf, p), "->");
                p += strlen(p);
                slen = sizeof(sa);
                if (getpeername(fd, &sa.sa, &slen) < 0)
                        (void) sm_snprintf(p, SPACELEFT(buf, p), "(%s)",
                                        sm_errstring(errno));
                else
                {
                        hp = hostnamebyanyaddr(&sa);
                        if (hp == NULL)
                        {
                                /* EMPTY */
                                /* do nothing */
                        }
# if NETINET
                        else if (sa.sa.sa_family == AF_INET)
                                (void) sm_snprintf(p, SPACELEFT(buf, p),
                                        "%s/%d", hp, ntohs(sa.sin.sin_port));
# endif /* NETINET */
# if NETINET6
                        else if (sa.sa.sa_family == AF_INET6)
                                (void) sm_snprintf(p, SPACELEFT(buf, p),
                                        "%s/%d", hp, ntohs(sa.sin6.sin6_port));
# endif /* NETINET6 */
                        else
                                (void) sm_snprintf(p, SPACELEFT(buf, p),
                                        "%s", hp);
                }
                break;
#endif /* S_IFSOCK */

          case S_IFCHR:
                (void) sm_snprintf(p, SPACELEFT(buf, p), "CHR: ");
                p += strlen(p);
```



```
                goto defprint;

#ifdef S_IFBLK
          case S_IFBLK:
                (void) sm_snprintf(p, SPACELEFT(buf, p), "BLK: ");
                p += strlen(p);
                goto defprint;
#endif /* S_IFBLK */

#if defined(S_IFIFO) && (!defined(S_IFSOCK) || S_IFIFO != S_IFSOCK)
          case S_IFIFO:
                (void) sm_snprintf(p, SPACELEFT(buf, p), "FIFO: ");
                p += strlen(p);
                goto defprint;
#endif /* defined(S_IFIFO) && (!defined(S_IFSOCK) || S_IFIFO != S_IFSOCK) */

#ifdef S_IFDIR
          case S_IFDIR:
                (void) sm_snprintf(p, SPACELEFT(buf, p), "DIR: ");
                p += strlen(p);
                goto defprint;
#endif /* S_IFDIR */

#ifdef S_IFLNK
          case S_IFLNK:
                (void) sm_snprintf(p, SPACELEFT(buf, p), "LNK: ");
                p += strlen(p);
                goto defprint;
#endif /* S_IFLNK */

          default:
defprint:
                (void) sm_snprintf(p, SPACELEFT(buf, p),
                                "dev=%d/%d, ino=%llu, nlink=%d, u/gid=%d/%d, ",
                                major(st.st_dev), minor(st.st_dev),
                                (ULONGLONG_T) st.st_ino,
                                (int) st.st_nlink, (int) st.st_uid,
                                (int) st.st_gid);
                p += strlen(p);
                (void) sm_snprintf(p, SPACELEFT(buf, p), "size=%llu",
                                (ULONGLONG_T) st.st_size);
                break;
        }

printit:
        if (logit)
                sm_syslog(LOG_DEBUG, CurEnv ? CurEnv->e_id : NULL,
                          "%.800s", buf);
        else
                sm_dprintf("%s\n", buf);
}

/*
**  SHORTEN_HOSTNAME -- strip local domain information off of hostname.
**
**      Parameters:
**              host -- the host to shorten (stripped in place).
**
**      Returns:
**              place where string was truncated, NULL if not truncated.
*/

char *
shorten_hostname(host)
        char host[];
{
```

```
1976         register char *p;
1977         char *mydom;
1978         int i;
1979         bool canon = false;

1981         /* strip off final dot */
1982         i = strlen(host);
1983         p = &host[(i == 0) ? 0 : i - 1];
1984         if (*p == '.')
1985         {
1986                 *p = '\0';
1987                 canon = true;
1988         }

1990         /* see if there is any domain at all -- if not, we are done */
1991         p = strchr(host, '.');
1992         if (p == NULL)
1993                 return NULL;

1995         /* yes, we have a domain -- see if it looks like us */
1996         mydom = macvalue('m', CurEnv);
1997         if (mydom == NULL)
1998                 mydom = "";
1999         i = strlen(++p);
2000         if ((canon ? sm_strcasecmp(p, mydom)
2001                    : sm_strncasecmp(p, mydom, i)) == 0 &&
2002                      (mydom[i] == '.' || mydom[i] == '\0'))
2003         {
2004                 *--p = '\0';
2005                 return p;
2006         }
2007         return NULL;
2008 }

2010 /*
2011 **  PROG_OPEN -- open a program for reading
2012 **
2013 **      Parameters:
2014 **              argv -- the argument list.
2015 **              pfd -- pointer to a place to store the file descriptor.
2016 **              e -- the current envelope.
2017 **
2018 **      Returns:
2019 **              pid of the process -- -1 if it failed.
2020 */

2022 pid_t
2023 prog_open(argv, pfd, e)
2024         char **argv;
2025         int *pfd;
2026         ENVELOPE *e;
2027 {
2028         pid_t pid;
2029         int save_errno;
2030         int sff;
2031         int ret;
2032         int fdv[2];
2033         char *p, *q;
2034         char buf[MAXPATHLEN];
2035         extern int DtableSize;

2037         if (pipe(fdv) < 0)
2038         {
2039                 syserr("%s: cannot create pipe for stdout", argv[0]);
2040                 return -1;
2041         }
```



```
2042         pid = fork();
2043         if (pid < 0)
2044         {
2045                 syserr("%s: cannot fork", argv[0]);
2046                 (void) close(fdv[0]);
2047                 (void) close(fdv[1]);
2048                 return -1;
2049         }
2050         if (pid > 0)
2051         {
2052                 /* parent */
2053                 (void) close(fdv[1]);
2054                 *pfd = fdv[0];
2055                 return pid;
2056         }

2058         /* Reset global flags */
2059         RestartRequest = NULL;
2060         RestartWorkGroup = false;
2061         ShutdownRequest = NULL;
2062         PendingSignal = 0;
2063         CurrentPid = getpid();

2065         /*
2066         **  Initialize exception stack and default exception
2067         **  handler for child process.
2068         */

2070         sm_exc_newthread(fatal_error);

2072         /* child -- close stdin */
2073         (void) close(0);

2075         /* stdout goes back to parent */
2076         (void) close(fdv[0]);
2077         if (dup2(fdv[1], 1) < 0)
2078         {
2079                 syserr("%s: cannot dup2 for stdout", argv[0]);
2080                 _exit(EX_OSERR);
2081         }
2082         (void) close(fdv[1]);

2084         /* stderr goes to transcript if available */
2085         if (e->e_xfp != NULL)
2086         {
2087                 int xfd;

2089                 xfd = sm_io_getinfo(e->e_xfp, SM_IO_WHAT_FD, NULL);
2090                 if (xfd >= 0 && dup2(xfd, 2) < 0)
2091                 {
2092                         syserr("%s: cannot dup2 for stderr", argv[0]);
2093                         _exit(EX_OSERR);
2094                 }
2095         }

2097         /* this process has no right to the queue file */
2098         if (e->e_lockfp != NULL)
2099         {
2100                 int fd;

2102                 fd = sm_io_getinfo(e->e_lockfp, SM_IO_WHAT_FD, NULL);
2103                 if (fd >= 0)
2104                         (void) close(fd);
2105                 else
2106                         syserr("%s: lockfp does not have a fd", argv[0]);
2107         }
```

```
2109         /* chroot to the program mailer directory, if defined */
2110         if (ProgMailer != NULL && ProgMailer->m_rootdir != NULL)
2111         {
2112                 expand(ProgMailer->m_rootdir, buf, sizeof(buf), e);
2113                 if (chroot(buf) < 0)
2114                 {
2115                         syserr("prog_open: cannot chroot(%s)", buf);
2116                         exit(EX_TEMPFAIL);
2117                 }
2118                 if (chdir("/") < 0)
2119                 {
2120                         syserr("prog_open: cannot chdir(/)");
2121                         exit(EX_TEMPFAIL);
2122                 }
2123         }

2125         /* run as default user */
2126         endpwent();
2127         sm_mbdb_terminate();
2128 #if _FFR_MEMSTAT
2129         (void) sm_memstat_close();
2130 #endif /* _FFR_MEMSTAT */
2131         if (setgid(DefGid) < 0 && geteuid() == 0)
2132         {
2133                 syserr("prog_open: setgid(%ld) failed", (long) DefGid);
2134                 exit(EX_TEMPFAIL);
2135         }
2136         if (setuid(DefUid) < 0 && geteuid() == 0)
2137         {
2138                 syserr("prog_open: setuid(%ld) failed", (long) DefUid);
2139                 exit(EX_TEMPFAIL);
2140         }

2142         /* run in some directory */
2143         if (ProgMailer != NULL)
2144                 p = ProgMailer->m_execdir;
2145         else
2146                 p = NULL;
2147         for (; p != NULL; p = q)
2148         {
2149                 q = strchr(p, ':');
2150                 if (q != NULL)
2151                         *q = '\0';
2152                 expand(p, buf, sizeof(buf), e);
2153                 if (q != NULL)
2154                         *q++ = ':';
2155                 if (buf[0] != '\0' && chdir(buf) >= 0)
2156                         break;
2157         }
2158         if (p == NULL)
2159         {
2160                 /* backup directories */
2161                 if (chdir("/tmp") < 0)
2162                         (void) chdir("/");
2163         }

2165         /* Check safety of program to be run */
2166         sff = SFF_ROOTOK|SFF_EXECOK;
2167         if (!bitnset(DBS_RUNWRITABLEPROGRAM, DontBlameSendmail))
2168                 sff |= SFF_NOGWFILES|SFF_NOWWFILES;
2169         if (bitnset(DBS_RUNPROGRAMINUNSAFEDIRPATH, DontBlameSendmail))
2170                 sff |= SFF_NOPATHCHECK;
2171         else
2172                 sff |= SFF_SAFEDIRPATH;
2173         ret = safefile(argv[0], DefUid, DefGid, DefUser, sff, 0, NULL);
```



```
2174         if (ret != 0)
2175                 sm_syslog(LOG_INFO, e->e_id,
2176                           "Warning: prog_open: program %s unsafe: %s",
2177                           argv[0], sm_errstring(ret));

2179         /* arrange for all the files to be closed */
2180         sm_close_on_exec(STDERR_FILENO + 1, DtableSize);

2182         /* now exec the process */
2183         (void) execve(argv[0], (ARGV_T) argv, (ARGV_T) UserEnviron);

2185         /* woops!  failed */
2186         save_errno = errno;
2187         syserr("%s: cannot exec", argv[0]);
2188         if (transienterror(save_errno))
2189                 _exit(EX_OSERR);
2190         _exit(EX_CONFIG);
2191         return -1;      /* avoid compiler warning on IRIX */
2192 }

2194 /*
2195 **  GET_COLUMN -- look up a Column in a line buffer
2196 **
2197 **      Parameters:
2198 **              line -- the raw text line to search.
2199 **              col -- the column number to fetch.
2200 **              delim -- the delimiter between columns.  If null,
2201 **                      use white space.
2202 **              buf -- the output buffer.
2203 **              buflen -- the length of buf.
2204 **
2205 **      Returns:
2206 **              buf if successful.
2207 **              NULL otherwise.
2208 */

2210 char *
2211 get_column(line, col, delim, buf, buflen)
2212         char line[];
2213         int col;
2214         int delim;
2215         char buf[];
2216         int buflen;
2217 {
2218         char *p;
2219         char *begin, *end;
2220         int i;
2221         char delimbuf[4];

2223         if ((char) delim == '\0')
2224                 (void) sm_strlcpy(delimbuf, "\n\t ", sizeof(delimbuf));
2225         else
2226         {
2227                 delimbuf[0] = (char) delim;
2228                 delimbuf[1] = '\0';
2229         }

2231         p = line;
2232         if (*p == '\0')
2233                 return NULL;                    /* line empty */
2234         if (*p == (char) delim && col == 0)
2235                 return NULL;                    /* first column empty */

2237         begin = line;

2239         if (col == 0 && (char) delim == '\0')
```

```
2240         {
2241                 while (*begin != '\0' && isascii(*begin) && isspace(*begin))
2242                         begin++;
2243         }
2244
2245         for (i = 0; i < col; i++)
2246         {
2247                 if ((begin = strpbrk(begin, delimbuf)) == NULL)
2248                         return NULL;            /* no such column */
2249                 begin++;
2250                 if ((char) delim == '\0')
2251                 {
2252                         while (*begin != '\0' && isascii(*begin) && isspace(*beg
2253                                 begin++;
2254                 }
2255         }
2256
2257         end = strpbrk(begin, delimbuf);
2258         if (end == NULL)
2259                 i = strlen(begin);
2260         else
2261                 i = end - begin;
2262         if (i >= buflen)
2263                 i = buflen - 1;
2264         (void) sm_strlcpy(buf, begin, i + 1);
2265         return buf;
2266 }
2267
2268 /*
2269 **  CLEANSTRCPY -- copy string keeping out bogus characters
2270 **
2271 **      Parameters:
2272 **              t -- "to" string.
2273 **              f -- "from" string.
2274 **              l -- length of space available in "to" string.
2275 **
2276 **      Returns:
2277 **              none.
2278 */
2279
2280 void
2281 cleanstrcpy(t, f, l)
2282         register char *t;
2283         register char *f;
2284         int l;
2285 {
2286         /* check for newlines and log if necessary */
2287         (void) denlstring(f, true, true);
2288
2289         if (l <= 0)
2290                 syserr("!cleanstrcpy: length == 0");
2291
2292         l--;
2293         while (l > 0 && *f != '\0')
2294         {
2295                 if (isascii(*f) &&
2296                     (isalnum(*f) || strchr("!#$%&'*+-./^_`{|}~", *f) != NULL))
2297                 {
2298                         l--;
2299                         *t++ = *f;
2300                 }
2301                 f++;
2302         }
2303         *t = '\0';
2304 }
```



```
2305
2306 /*
2307 **  DENLSTRING -- convert newlines in a string to spaces
2308 **
2309 **      Parameters:
2310 **              s -- the input string
2311 **              strict -- if set, don't permit continuation lines.
2312 **              logattacks -- if set, log attempted attacks.
2313 **
2314 **      Returns:
2315 **              A pointer to a version of the string with newlines
2316 **              mapped to spaces.  This should be copied.
2317 */
2318
2319 char *
2320 denlstring(s, strict, logattacks)
2321         char *s;
2322         bool strict;
2323         bool logattacks;
2324 {
2325         register char *p;
2326         int l;
2327         static char *bp = NULL;
2328         static int bl = 0;
2329
2330         p = s;
2331         while ((p = strchr(p, '\n')) != NULL)
2332                 if (strict || (*++p != ' ' && *p != '\t'))
2333                         break;
2334         if (p == NULL)
2335                 return s;
2336
2337         l = strlen(s) + 1;
2338         if (bl < l)
2339         {
2340                 /* allocate more space */
2341                 char *nbp = sm_pmalloc_x(l);
2342
2343                 if (bp != NULL)
2344                         sm_free(bp);
2345                 bp = nbp;
2346                 bl = l;
2347         }
2348         (void) sm_strlcpy(bp, s, l);
2349         for (p = bp; (p = strchr(p, '\n')) != NULL; )
2350                 *p++ = ' ';
2351
2352         if (logattacks)
2353         {
2354                 sm_syslog(LOG_NOTICE, CurEnv ? CurEnv->e_id : NULL,
2355                           "POSSIBLE ATTACK from %.100s: newline in string \"%s\"
2356                           RealHostName == NULL ? "[UNKNOWN]" : RealHostName,
2357                           shortenstring(bp, MAXSHORTSTR));
2358         }
2359
2360         return bp;
2361 }
2362
2363 /*
2364 **  STRREPLNONPRT -- replace "unprintable" characters in a string with subst
2365 **
2366 **      Parameters:
2367 **              s -- string to manipulate (in place)
2368 **              subst -- character to use as replacement
2369 **
2370 **      Returns:
2371 **              true iff string did not contain "unprintable" characters
```

```
2372 */

2374 bool
2375 strreplnonprt(s, c)
2376         char *s;
2377         int c;
2378 {
2379         bool ok;

2381         ok = true;
2382         if (s == NULL)
2383                 return ok;
2384         while (*s != '\0')
2385         {
2386                 if (!(isascii(*s) && isprint(*s)))
2387                 {
2388                         *s = c;
2389                         ok = false;
2390                 }
2391                 ++s;
2392         }
2393         return ok;
2394 }

2396 /*
2397 **  PATH_IS_DIR -- check to see if file exists and is a directory.
2398 **
2399 **      There are some additional checks for security violations in
2400 **      here.  This routine is intended to be used for the host status
2401 **      support.
2402 **
2403 **      Parameters:
2404 **              pathname -- pathname to check for directory-ness.
2405 **              createflag -- if set, create directory if needed.
2406 **
2407 **      Returns:
2408 **              true -- if the indicated pathname is a directory
2409 **              false -- otherwise
2410 */

2412 bool
2413 path_is_dir(pathname, createflag)
2414         char *pathname;
2415         bool createflag;
2416 {
2417         struct stat statbuf;

2419 #if HASLSTAT
2420         if (lstat(pathname, &statbuf) < 0)
2421 #else /* HASLSTAT */
2422         if (stat(pathname, &statbuf) < 0)
2423 #endif /* HASLSTAT */
2424         {
2425                 if (errno != ENOENT || !createflag)
2426                         return false;
2427                 if (mkdir(pathname, 0755) < 0)
2428                         return false;
2429                 return true;
2430         }
2431         if (!S_ISDIR(statbuf.st_mode))
2432         {
2433                 errno = ENOTDIR;
2434                 return false;
2435         }

2437         /* security: don't allow writable directories */
```



```
2438         if (bitset(S_IWGRP|S_IWOTH, statbuf.st_mode))
2439         {
2440                 errno = EACCES;
2441                 return false;
2442         }
2443         return true;
2444 }

2446 /*
2447 **  PROC_LIST_ADD -- add process id to list of our children
2448 **
2449 **      Parameters:
2450 **              pid -- pid to add to list.
2451 **              task -- task of pid.
2452 **              type -- type of process.
2453 **              count -- number of processes.
2454 **              other -- other information for this type.
2455 **
2456 **      Returns:
2457 **              none
2458 **
2459 **      Side Effects:
2460 **              May increase CurChildren. May grow ProcList.
2461 */

2463 typedef struct procs    PROCS_T;

2465 struct procs
2466 {
2467         pid_t           proc_pid;
2468         char            *proc_task;
2469         int             proc_type;
2470         int             proc_count;
2471         int             proc_other;
2472         SOCKADDR        proc_hostaddr;
2473 };

2475 static PROCS_T  *volatile ProcListVec = NULL;
2476 static int      ProcListSize = 0;

2478 void
2479 proc_list_add(pid, task, type, count, other, hostaddr)
2480         pid_t pid;
2481         char *task;
2482         int type;
2483         int count;
2484         int other;
2485         SOCKADDR *hostaddr;
2486 {
2487         int i;

2489         for (i = 0; i < ProcListSize; i++)
2490         {
2491                 if (ProcListVec[i].proc_pid == NO_PID)
2492                         break;
2493         }
2494         if (i >= ProcListSize)
2495         {
2496                 /* probe the existing vector to avoid growing infinitely */
2497                 proc_list_probe();

2499                 /* now scan again */
2500                 for (i = 0; i < ProcListSize; i++)
2501                 {
2502                         if (ProcListVec[i].proc_pid == NO_PID)
2503                                 break;
```

```
2504                    }
2505            }
2506            if (i >= ProcListSize)
2507            {
2508                    /* grow process list */
2509                    int chldwasblocked;
2510                    PROCS_T *npv;

2512                    SM_ASSERT(ProcListSize < INT_MAX - PROC_LIST_SEG);
2513                    npv = (PROCS_T *) sm_pmalloc_x((sizeof(*npv)) *
2514                                                   (ProcListSize + PROC_LIST_SEG));

2516                    /* Block SIGCHLD so reapchild() doesn't mess with us */
2517                    chldwasblocked = sm_blocksignal(SIGCHLD);
2518                    if (ProcListSize > 0)
2519                    {
2520                            memmove(npv, ProcListVec,
2521                                    ProcListSize * sizeof(PROCS_T));
2522                            sm_free(ProcListVec);
2523                    }

2525                    /* XXX just use memset() to initialize this part? */
2526                    for (i = ProcListSize; i < ProcListSize + PROC_LIST_SEG; i++)
2527                    {
2528                            npv[i].proc_pid = NO_PID;
2529                            npv[i].proc_task = NULL;
2530                            npv[i].proc_type = PROC_NONE;
2531                    }
2532                    i = ProcListSize;
2533                    ProcListSize += PROC_LIST_SEG;
2534                    ProcListVec = npv;
2535                    if (chldwasblocked == 0)
2536                            (void) sm_releasesignal(SIGCHLD);
2537            }
2538            ProcListVec[i].proc_pid = pid;
2539            PSTRSET(ProcListVec[i].proc_task, task);
2540            ProcListVec[i].proc_type = type;
2541            ProcListVec[i].proc_count = count;
2542            ProcListVec[i].proc_other = other;
2543            if (hostaddr != NULL)
2544                    ProcListVec[i].proc_hostaddr = *hostaddr;
2545            else
2546                    memset(&ProcListVec[i].proc_hostaddr, 0,
2547                            sizeof(ProcListVec[i].proc_hostaddr));

2549            /* if process adding itself, it's not a child */
2550            if (pid != CurrentPid)
2551            {
2552                    SM_ASSERT(CurChildren < INT_MAX);
2553                    CurChildren++;
2554            }
2555    }

2557    /*
2558    **  PROC_LIST_SET -- set pid task in process list
2559    **
2560    **      Parameters:
2561    **              pid -- pid to set
2562    **              task -- task of pid
2563    **
2564    **      Returns:
2565    **              none.
2566    */

2568    void
2569    proc_list_set(pid, task)
```



```
2570            pid_t pid;
2571            char *task;
2572    {
2573            int i;

2575            for (i = 0; i < ProcListSize; i++)
2576            {
2577                    if (ProcListVec[i].proc_pid == pid)
2578                    {
2579                            PSTRSET(ProcListVec[i].proc_task, task);
2580                            break;
2581                    }
2582            }
2583    }

2585    /*
2586    **  PROC_LIST_DROP -- drop pid from process list
2587    **
2588    **      Parameters:
2589    **              pid -- pid to drop
2590    **              st -- process status
2591    **              other -- storage for proc_other (return).
2592    **
2593    **      Returns:
2594    **              none.
2595    **
2596    **      Side Effects:
2597    **              May decrease CurChildren, CurRunners, or
2598    **              set RestartRequest or ShutdownRequest.
2599    **
2600    **      NOTE:   THIS CAN BE CALLED FROM A SIGNAL HANDLER.  DO NOT ADD
2601    **              ANYTHING TO THIS ROUTINE UNLESS YOU KNOW WHAT YOU ARE
2602    **              DOING.
2603    */

2605    void
2606    proc_list_drop(pid, st, other)
2607            pid_t pid;
2608            int st;
2609            int *other;
2610    {
2611            int i;
2612            int type = PROC_NONE;

2614            for (i = 0; i < ProcListSize; i++)
2615            {
2616                    if (ProcListVec[i].proc_pid == pid)
2617                    {
2618                            ProcListVec[i].proc_pid = NO_PID;
2619                            type = ProcListVec[i].proc_type;
2620                            if (other != NULL)
2621                                    *other = ProcListVec[i].proc_other;
2622                            if (CurChildren > 0)
2623                                    CurChildren--;
2624                            break;
2625                    }
2626            }


2629            if (type == PROC_CONTROL && WIFEXITED(st))
2630            {
2631                    /* if so, see if we need to restart or shutdown */
2632                    if (WEXITSTATUS(st) == EX_RESTART)
2633                            RestartRequest = "control socket";
2634                    else if (WEXITSTATUS(st) == EX_SHUTDOWN)
2635                            ShutdownRequest = "control socket";
```

```
2636         }
2637         else if (type == PROC_QUEUE_CHILD && !WIFSTOPPED(st) &&
2638                  ProcListVec[i].proc_other > -1)
2639         {
2640                 /* restart this persistent runner */
2641                 mark_work_group_restart(ProcListVec[i].proc_other, st);
2642         }
2643         else if (type == PROC_QUEUE)
2644                 CurRunners -= ProcListVec[i].proc_count;
2645 }

2647 /*
2648 **  PROC_LIST_CLEAR -- clear the process list
2649 **
2650 **      Parameters:
2651 **              none.
2652 **
2653 **      Returns:
2654 **              none.
2655 **
2656 **      Side Effects:
2657 **              Sets CurChildren to zero.
2658 */

2660 void
2661 proc_list_clear()
2662 {
2663         int i;

2665         /* start from 1 since 0 is the daemon itself */
2666         for (i = 1; i < ProcListSize; i++)
2667                 ProcListVec[i].proc_pid = NO_PID;
2668         CurChildren = 0;
2669 }

2671 /*
2672 **  PROC_LIST_PROBE -- probe processes in the list to see if they still exist
2673 **
2674 **      Parameters:
2675 **              none
2676 **
2677 **      Returns:
2678 **              none
2679 **
2680 **      Side Effects:
2681 **              May decrease CurChildren.
2682 */

2684 void
2685 proc_list_probe()
2686 {
2687         int i, children;
2688         int chldwasblocked;
2689         pid_t pid;

2691         children = 0;
2692         chldwasblocked = sm_blocksignal(SIGCHLD);

2694         /* start from 1 since 0 is the daemon itself */
2695         for (i = 1; i < ProcListSize; i++)
2696         {
2697                 pid = ProcListVec[i].proc_pid;
2698                 if (pid == NO_PID || pid == CurrentPid)
2699                         continue;
2700                 if (kill(pid, 0) < 0)
2701                 {
```



```
2702                         if (LogLevel > 3)
2703                                 sm_syslog(LOG_DEBUG, CurEnv->e_id,
2704                                         "proc_list_probe: lost pid %d",
2705                                         (int) ProcListVec[i].proc_pid);
2706                         ProcListVec[i].proc_pid = NO_PID;
2707                         SM_FREE_CLR(ProcListVec[i].proc_task);
2708                         CurChildren--;
2709                 }
2710                 else
2711                 {
2712                         ++children;
2713                 }
2714         }
2715         if (CurChildren < 0)
2716                 CurChildren = 0;
2717         if (chldwasblocked == 0)
2718                 (void) sm_releasesignal(SIGCHLD);
2719         if (LogLevel > 10 && children != CurChildren && CurrentPid == DaemonPid)
2720         {
2721                 sm_syslog(LOG_ERR, NOQID,
2722                           "proc_list_probe: found %d children, expected %d",
2723                           children, CurChildren);
2724         }
2725 }

2727 /*
2728 **  PROC_LIST_DISPLAY -- display the process list
2729 **
2730 **      Parameters:
2731 **              out -- output file pointer
2732 **              prefix -- string to output in front of each line.
2733 **
2734 **      Returns:
2735 **              none.
2736 */

2738 void
2739 proc_list_display(out, prefix)
2740         SM_FILE_T *out;
2741         char *prefix;
2742 {
2743         int i;

2745         for (i = 0; i < ProcListSize; i++)
2746         {
2747                 if (ProcListVec[i].proc_pid == NO_PID)
2748                         continue;

2750                 (void) sm_io_fprintf(out, SM_TIME_DEFAULT, "%s%d %s%s\n",
2751                                      prefix,
2752                                      (int) ProcListVec[i].proc_pid,
2753                                      ProcListVec[i].proc_task != NULL ?
2754                                      ProcListVec[i].proc_task : "(unknown)",
2755                                      (OpMode == MD_SMTP ||
2756                                       OpMode == MD_DAEMON ||
2757                                       OpMode == MD_ARPAFTP) ? "\r" : "");
2758         }
2759 }

2761 /*
2762 **  PROC_LIST_SIGNAL -- send a signal to a type of process in the list
2763 **
2764 **      Parameters:
2765 **              type -- type of process to signal
2766 **              signal -- the type of signal to send
2767 **
```

```
**      Results:
**              none.
**
**      NOTE:   THIS CAN BE CALLED FROM A SIGNAL HANDLER.  DO NOT ADD
**              ANYTHING TO THIS ROUTINE UNLESS YOU KNOW WHAT YOU ARE
**              DOING.
*/

void
proc_list_signal(type, signal)
        int type;
        int signal;
{
        int chldwasblocked;
        int alrmwasblocked;
        int i;
        pid_t mypid = getpid();

        /* block these signals so that we may signal cleanly */
        chldwasblocked = sm_blocksignal(SIGCHLD);
        alrmwasblocked = sm_blocksignal(SIGALRM);

        /* Find all processes of type and send signal */
        for (i = 0; i < ProcListSize; i++)
        {
                if (ProcListVec[i].proc_pid == NO_PID ||
                    ProcListVec[i].proc_pid == mypid)
                        continue;
                if (ProcListVec[i].proc_type != type)
                        continue;
                (void) kill(ProcListVec[i].proc_pid, signal);
        }

        /* restore the signals */
        if (alrmwasblocked == 0)
                (void) sm_releasesignal(SIGALRM);
        if (chldwasblocked == 0)
                (void) sm_releasesignal(SIGCHLD);
}

/*
**  COUNT_OPEN_CONNECTIONS
**
**      Parameters:
**              hostaddr - ClientAddress
**
**      Returns:
**              the number of open connections for this client
**
*/

int
count_open_connections(hostaddr)
        SOCKADDR *hostaddr;
{
        int i, n;

        if (hostaddr == NULL)
                return 0;

        /*
        **  This code gets called before proc_list_add() gets called,
        **  so we (the daemon child for this connection) have not yet
        **  counted ourselves.  Hence initialize the counter to 1
        **  instead of 0 to compensate.
        */
```



```
        n = 1;
        for (i = 0; i < ProcListSize; i++)
        {
                if (ProcListVec[i].proc_pid == NO_PID)
                        continue;
                if (hostaddr->sa.sa_family !=
                    ProcListVec[i].proc_hostaddr.sa.sa_family)
                        continue;
#if NETINET
                if (hostaddr->sa.sa_family == AF_INET &&
                    (hostaddr->sin.sin_addr.s_addr ==
                     ProcListVec[i].proc_hostaddr.sin.sin_addr.s_addr))
                        n++;
#endif /* NETINET */
#if NETINET6
                if (hostaddr->sa.sa_family == AF_INET6 &&
                    IN6_ARE_ADDR_EQUAL(&(hostaddr->sin6.sin6_addr),
                                       &(ProcListVec[i].proc_hostaddr.sin6.sin6_
                        n++;
#endif /* NETINET6 */
        }
        return n;
}
```

```
**********************************************************
   67397 Tue Aug 18 16:13:44 2015
new/usr/src/lib/brand/solaris10/s10_brand/common/s10_brand.c
6136 sysmacros.h unnecessarily polutes the namespace
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */

  22 /*
  23  * Copyright (c) 2013, OmniTI Computer Consulting, Inc. All rights reserved.
  24  * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
  25  */

  27 #include <errno.h>
  28 #include <fcntl.h>
  29 #include <dirent.h>
  30 #include <stddef.h>
  31 #include <stdio.h>
  32 #include <stdlib.h>
  33 #include <strings.h>
  34 #include <unistd.h>
  35 #include <thread.h>
  36 #include <sys/auxv.h>
  37 #include <sys/brand.h>
  38 #include <sys/inttypes.h>
  39 #include <sys/lwp.h>
  40 #include <sys/syscall.h>
  41 #include <sys/systm.h>
  42 #include <sys/utsname.h>
  43 #include <sys/sysconfig.h>
  44 #include <sys/systeminfo.h>
  45 #include <sys/zone.h>
  46 #include <sys/stat.h>
  47 #include <sys/mntent.h>
  48 #include <sys/ctfs.h>
  49 #include <sys/priv.h>
  50 #include <sys/acctctl.h>
  51 #include <libgen.h>
  52 #include <bsm/audit.h>
  53 #include <sys/crypto/ioctl.h>
  54 #include <sys/fs/zfs.h>
  55 #include <sys/zfs_ioctl.h>
  56 #include <sys/ucontext.h>
  57 #include <sys/mntio.h>
  58 #include <sys/mnttab.h>
  59 #include <sys/attr.h>
  60 #include <sys/lofi.h>
  61 #include <sys/mkdev.h>
```



```
  62 #endif /* ! codereview */
  63 #include <atomic.h>
  64 #include <sys/acl.h>
  65 #include <sys/socket.h>

  67 #include <s10_brand.h>
  68 #include <brand_misc.h>
  69 #include <s10_misc.h>
  70 #include <s10_signal.h>

  72 /*
  73  * See usr/src/lib/brand/shared/brand/common/brand_util.c for general
  74  * emulation notes.
  75  */

  77 static zoneid_t zoneid;
  78 static boolean_t emul_global_zone = B_FALSE;
  79 static s10_emul_bitmap_t emul_bitmap;
  80 pid_t zone_init_pid;

  82 /*
  83  * S10_FEATURE_IS_PRESENT is a macro that helps facilitate conditional
  84  * emulation.  For each constant N defined in the s10_emulated_features
  85  * enumeration in usr/src/uts/common/brand/solaris10/s10_brand.h,
  86  * S10_FEATURE_IS_PRESENT(N) is true iff the feature/backport represented by N
  87  * is present in the Solaris 10 image hosted within the zone.  In other words,
  88  * S10_FEATURE_IS_PRESENT(N) is true iff the file /usr/lib/brand/solaris10/M,
  89  * where M is the enum value of N, was present in the zone when the zone booted.
  90  *
  91  *
  92  * *** Sample Usage
  93  *
  94  * Suppose that you need to backport a fix to Solaris 10 and there is
  95  * emulation in place for the fix.  Suppose further that the emulation won't be
  96  * needed if the fix is backported (i.e., if the fix is present in the hosted
  97  * Solaris 10 environment, then the brand won't need the emulation).  Then if
  98  * you add a constant named "S10_FEATURE_X" to the end of the
  99  * s10_emulated_features enumeration that represents the backported fix and
 100  * S10_FEATURE_X evaluates to four, then you should create a file named
 101  * /usr/lib/brand/solaris10/4 as part of your backport.  Additionally, you
 102  * should retain the aforementioned emulation but modify it so that it's
 103  * performed only when S10_FEATURE_IS_PRESENT(S10_FEATURE_X) is false.  Thus the
 104  * emulation function should look something like the following:
 105  *
 106  *      static int
 107  *      my_emul_function(sysret_t *rv, ...)
 108  *      {
 109  *              if (S10_FEATURE_IS_PRESENT(S10_FEATURE_X)) {
 110  *                      // Don't emulate
 111  *                      return (__systemcall(rv, ...));
 112  *              } else {
 113  *                      // Emulate whatever needs to be emulated when the
 114  *                      // backport isn't present in the Solaris 10 image.
 115  *              }
 116  *      }
 117  */
 118 #define S10_FEATURE_IS_PRESENT(s10_emulated_features_constant)  \
 119         ((emul_bitmap[(s10_emulated_features_constant) >> 3] &  \
 120         (1 << ((s10_emulated_features_constant) & 0x7))) != 0)

 122 brand_sysent_table_t brand_sysent_table[];

 124 #define S10_UTS_RELEASE "5.10"
 125 #define S10_UTS_VERSION "Generic_Virtual"

 127 /*
```

```
 * If the ioctl fd's major doesn't match "major", then pass through the
 * ioctl, since it is not the expected device.  major should be a
 * pointer to a static dev_t initialized to -1, and devname should be
 * the path of the device.
 *
 * Returns 1 if the ioctl was handled (in which case *err contains the
 * error code), or 0 if it still needs handling.
 */
static int
passthru_otherdev_ioctl(dev_t *majordev, const char *devname, int *err,
    sysret_t *rval, int fdes, int cmd, intptr_t arg)
{
        struct stat sbuf;

        if (*majordev == (dev_t)-1) {
                if ((*err = __systemcall(rval, SYS_fstatat + 1024,
                    AT_FDCWD, devname, &sbuf, 0) != 0) != 0)
                        goto doioctl;

                *majordev = major(sbuf.st_rdev);
        }

        if ((*err = __systemcall(rval, SYS_fstatat + 1024, fdes,
            NULL, &sbuf, 0)) != 0)
                goto doioctl;

        if (major(sbuf.st_rdev) == *majordev)
                return (0);

doioctl:
        *err = (__systemcall(rval, SYS_ioctl + 1024, fdes, cmd, arg));
        return (1);
}

/*
 * Figures out the PID of init for the zone.  Also returns a boolean
 * indicating whether this process currently has that pid: if so,
 * then at this moment, we are init.
 */
static boolean_t
get_initpid_info(void)
{
        pid_t pid;
        sysret_t rval;
        int err;

        /*
         * Determine the current process PID and the PID of the zone's init.
         * We use care not to call getpid() here, because we're not supposed
         * to call getpid() until after the program is fully linked-- the
         * first call to getpid() is a signal from the linker to debuggers
         * that linking has been completed.
         */
        if ((err = __systemcall(&rval, SYS_brand,
            B_S10_PIDINFO, &pid, &zone_init_pid)) != 0) {
                brand_abort(err, "Failed to get init's pid");
        }

        /*
         * Note that we need to be cautious with the pid we get back--
         * it should not be stashed and used in place of getpid(), since
         * we might fork(2).  So we keep zone_init_pid and toss the pid
         * we otherwise got.
         */
        if (pid == zone_init_pid)
                return (B_TRUE);
```



```
        return (B_FALSE);
}

/* Free the thread-local storage provided by mntfs_get_mntentbuf(). */
static void
mntfs_free_mntentbuf(void *arg)
{
        struct mntentbuf *embufp = arg;

        if (embufp == NULL)
                return;
        if (embufp->mbuf_emp)
                free(embufp->mbuf_emp);
        if (embufp->mbuf_buf)
                free(embufp->mbuf_buf);
        bzero(embufp, sizeof (struct mntentbuf));
        free(embufp);
}

/* Provide the thread-local storage required by mntfs_ioctl(). */
static struct mntentbuf *
mntfs_get_mntentbuf(size_t size)
{
        static mutex_t keylock;
        static thread_key_t key;
        static int once_per_keyname = 0;
        void *tsd = NULL;
        struct mntentbuf *embufp;

        /* Create the key. */
        if (!once_per_keyname) {
                (void) mutex_lock(&keylock);
                if (!once_per_keyname) {
                        if (thr_keycreate(&key, mntfs_free_mntentbuf)) {
                                (void) mutex_unlock(&keylock);
                                return (NULL);
                        } else {
                                once_per_keyname++;
                        }
                }
                (void) mutex_unlock(&keylock);
        }

        /*
         * The thread-specific datum for this key is the address of a struct
         * mntentbuf. If this is the first time here then we allocate the struct
         * and its contents, and associate its address with the thread; if there
         * are any problems then we abort.
         */
        if (thr_getspecific(key, &tsd))
                return (NULL);
        if (tsd == NULL) {
                if (!(embufp = calloc(1, sizeof (struct mntentbuf))) ||
                    !(embufp->mbuf_emp = malloc(sizeof (struct extmnttab))) ||
                    thr_setspecific(key, embufp)) {
                        mntfs_free_mntentbuf(embufp);
                        return (NULL);
                }
        } else {
                embufp = tsd;
        }

        /* Return the buffer, resizing it if necessary. */
        if (size > embufp->mbuf_bufsize) {
                if (embufp->mbuf_buf)
```

```
260                         free(embufp->mbuf_buf);
261                 if ((embufp->mbuf_buf = malloc(size)) == NULL) {
262                         embufp->mbuf_bufsize = 0;
263                         return (NULL);
264                 } else {
265                         embufp->mbuf_bufsize = size;
266                 }
267         }
268         return (embufp);
269 }

271 /*
272  * The MNTIOC_GETMNTENT command in this release differs from that in early
273  * versions of Solaris 10.
274  *
275  * Previously, the command would copy a pointer to a struct extmnttab to an
276  * address provided as an argument. The pointer would be somewhere within a
277  * mapping already present within the user's address space. In addition, the
278  * text to which the struct's members pointed would also be within a
279  * pre-existing mapping. Now, the user is required to allocate memory for both
280  * the struct and the text buffer, and to pass the address of each within a
281  * struct mntentbuf. In order to conceal these details from a Solaris 10 client
282  * we allocate some thread-local storage in which to create the necessary data
283  * structures; this is static, thread-safe memory that will be cleaned up
284  * without the caller's intervention.
285  *
286  * MNTIOC_GETEXTMNTENT and MNTIOC_GETMNTANY are new in this release; they should
287  * not work for older clients.
288  */
289 int
290 mntfs_ioctl(sysret_t *rval, int fdes, int cmd, intptr_t arg)
291 {
292         int err;
293         struct stat statbuf;
294         struct mntentbuf *embufp;
295         static size_t bufsize = MNT_LINE_MAX;

297         /* Do not emulate mntfs commands from up-to-date clients. */
298         if (S10_FEATURE_IS_PRESENT(S10_FEATURE_ALTERED_MNTFS_IOCTL))
299                 return (__systemcall(rval, SYS_ioctl + 1024, fdes, cmd, arg));

301         /* Do not emulate mntfs commands directed at other file systems. */
302         if ((err = __systemcall(rval, SYS_fstatat + 1024,
303             fdes, NULL, &statbuf, 0)) != 0)
304                 return (err);
305         if (strcmp(statbuf.st_fstype, MNTTYPE_MNTFS) != 0)
306                 return (__systemcall(rval, SYS_ioctl + 1024, fdes, cmd, arg));

308         if (cmd == MNTIOC_GETEXTMNTENT || cmd == MNTIOC_GETMNTANY)
309                 return (EINVAL);

311         if ((embufp = mntfs_get_mntentbuf(bufsize)) == NULL)
312                 return (ENOMEM);

314         /*
315          * MNTIOC_GETEXTMNTENT advances the file pointer once it has
316          * successfully copied out the result to the address provided. We
317          * therefore need to check the user-supplied address now since the
318          * one we'll be providing is guaranteed to work.
319          */
320         if (brand_uucopy(&embufp->mbuf_emp, (void *)arg, sizeof (void *)) != 0)
321                 return (EFAULT);

323         /*
324          * Keep retrying for as long as we fail for want of a large enough
325          * buffer.
```



```
326          */
327         for (;;) {
328                 if ((err = __systemcall(rval, SYS_ioctl + 1024, fdes,
329                     MNTIOC_GETEXTMNTENT, embufp)) != 0)
330                         return (err);

332                 if (rval->sys_rval1 == MNTFS_TOOLONG) {
333                         /* The buffer wasn't large enough. */
334                         (void) atomic_swap_ulong((unsigned long *)&bufsize,
335                             2 * embufp->mbuf_bufsize);
336                         if ((embufp = mntfs_get_mntentbuf(bufsize)) == NULL)
337                                 return (ENOMEM);
338                 } else {
339                         break;
340                 }
341         }

343         if (brand_uucopy(&embufp->mbuf_emp, (void *)arg, sizeof (void *)) != 0)
344                 return (EFAULT);

346         return (0);
347 }

349 /*
350  * Assign the structure member value from the s (source) structure to the
351  * d (dest) structure.
352  */
353 #define struct_assign(d, s, val)        (((d).val) = ((s).val))

355 /*
356  * The CRYPTO_GET_FUNCTION_LIST parameter structure crypto_function_list_t
357  * changed between S10 and Nevada, so we have to emulate the old S10
358  * crypto_function_list_t structure when interposing on the ioctl syscall.
359  */
360 typedef struct s10_crypto_function_list {
361         boolean_t fl_digest_init;
362         boolean_t fl_digest;
363         boolean_t fl_digest_update;
364         boolean_t fl_digest_key;
365         boolean_t fl_digest_final;

367         boolean_t fl_encrypt_init;
368         boolean_t fl_encrypt;
369         boolean_t fl_encrypt_update;
370         boolean_t fl_encrypt_final;

372         boolean_t fl_decrypt_init;
373         boolean_t fl_decrypt;
374         boolean_t fl_decrypt_update;
375         boolean_t fl_decrypt_final;

377         boolean_t fl_mac_init;
378         boolean_t fl_mac;
379         boolean_t fl_mac_update;
380         boolean_t fl_mac_final;

382         boolean_t fl_sign_init;
383         boolean_t fl_sign;
384         boolean_t fl_sign_update;
385         boolean_t fl_sign_final;
386         boolean_t fl_sign_recover_init;
387         boolean_t fl_sign_recover;

389         boolean_t fl_verify_init;
390         boolean_t fl_verify;
391         boolean_t fl_verify_update;
```

```
392         boolean_t fl_verify_final;
393         boolean_t fl_verify_recover_init;
394         boolean_t fl_verify_recover;

396         boolean_t fl_digest_encrypt_update;
397         boolean_t fl_decrypt_digest_update;
398         boolean_t fl_sign_encrypt_update;
399         boolean_t fl_decrypt_verify_update;

401         boolean_t fl_seed_random;
402         boolean_t fl_generate_random;

404         boolean_t fl_session_open;
405         boolean_t fl_session_close;
406         boolean_t fl_session_login;
407         boolean_t fl_session_logout;

409         boolean_t fl_object_create;
410         boolean_t fl_object_copy;
411         boolean_t fl_object_destroy;
412         boolean_t fl_object_get_size;
413         boolean_t fl_object_get_attribute_value;
414         boolean_t fl_object_set_attribute_value;
415         boolean_t fl_object_find_init;
416         boolean_t fl_object_find;
417         boolean_t fl_object_find_final;

419         boolean_t fl_key_generate;
420         boolean_t fl_key_generate_pair;
421         boolean_t fl_key_wrap;
422         boolean_t fl_key_unwrap;
423         boolean_t fl_key_derive;

425         boolean_t fl_init_token;
426         boolean_t fl_init_pin;
427         boolean_t fl_set_pin;

429         boolean_t prov_is_hash_limited;
430         uint32_t prov_hash_threshold;
431         uint32_t prov_hash_limit;
432 } s10_crypto_function_list_t;

434 typedef struct s10_crypto_get_function_list {
435         uint_t                          fl_return_value;
436         crypto_provider_id_t            fl_provider_id;
437         s10_crypto_function_list_t      fl_list;
438 } s10_crypto_get_function_list_t;

440 /*
441  * The structure returned by the CRYPTO_GET_FUNCTION_LIST ioctl on /dev/crypto
442  * increased in size due to:
443  *      6482533 Threshold for HW offload via PKCS11 interface
444  * between S10 and Nevada.  This is a relatively simple process of filling
445  * in the S10 structure fields with the Nevada data.
446  *
447  * We stat the device to make sure that the ioctl is meant for /dev/crypto.
448  *
449  */
450 static int
451 crypto_ioctl(sysret_t *rval, int fdes, int cmd, intptr_t arg)
452 {
453         int                             err;
454         s10_crypto_get_function_list_t  s10_param;
455         crypto_get_function_list_t      native_param;
456         static dev_t                    crypto_dev = (dev_t)-1;
```



```
458         if (passthru_otherdev_ioctl(&crypto_dev, "/dev/crypto", &err,
459             rval, fdes, cmd, arg) == 1)
460                 return (err);

462         if (brand_uucopy((const void *)arg, &s10_param, sizeof (s10_param))
463             != 0)
464                 return (EFAULT);
465         struct_assign(native_param, s10_param, fl_provider_id);
466         if ((err = __systemcall(rval, SYS_ioctl + 1024, fdes, cmd,
467             &native_param)) != 0)
468                 return (err);

470         struct_assign(s10_param, native_param, fl_return_value);
471         struct_assign(s10_param, native_param, fl_provider_id);

473         struct_assign(s10_param, native_param, fl_list.fl_digest_init);
474         struct_assign(s10_param, native_param, fl_list.fl_digest);
475         struct_assign(s10_param, native_param, fl_list.fl_digest_update);
476         struct_assign(s10_param, native_param, fl_list.fl_digest_key);
477         struct_assign(s10_param, native_param, fl_list.fl_digest_final);

479         struct_assign(s10_param, native_param, fl_list.fl_encrypt_init);
480         struct_assign(s10_param, native_param, fl_list.fl_encrypt);
481         struct_assign(s10_param, native_param, fl_list.fl_encrypt_update);
482         struct_assign(s10_param, native_param, fl_list.fl_encrypt_final);

484         struct_assign(s10_param, native_param, fl_list.fl_decrypt_init);
485         struct_assign(s10_param, native_param, fl_list.fl_decrypt);
486         struct_assign(s10_param, native_param, fl_list.fl_decrypt_update);
487         struct_assign(s10_param, native_param, fl_list.fl_decrypt_final);

489         struct_assign(s10_param, native_param, fl_list.fl_mac_init);
490         struct_assign(s10_param, native_param, fl_list.fl_mac);
491         struct_assign(s10_param, native_param, fl_list.fl_mac_update);
492         struct_assign(s10_param, native_param, fl_list.fl_mac_final);

494         struct_assign(s10_param, native_param, fl_list.fl_sign_init);
495         struct_assign(s10_param, native_param, fl_list.fl_sign);
496         struct_assign(s10_param, native_param, fl_list.fl_sign_update);
497         struct_assign(s10_param, native_param, fl_list.fl_sign_final);
498         struct_assign(s10_param, native_param, fl_list.fl_sign_recover_init);
499         struct_assign(s10_param, native_param, fl_list.fl_sign_recover);

501         struct_assign(s10_param, native_param, fl_list.fl_verify_init);
502         struct_assign(s10_param, native_param, fl_list.fl_verify);
503         struct_assign(s10_param, native_param, fl_list.fl_verify_update);
504         struct_assign(s10_param, native_param, fl_list.fl_verify_final);
505         struct_assign(s10_param, native_param, fl_list.fl_verify_recover_init);
506         struct_assign(s10_param, native_param, fl_list.fl_verify_recover);

508         struct_assign(s10_param, native_param,
509             fl_list.fl_digest_encrypt_update);
510         struct_assign(s10_param, native_param,
511             fl_list.fl_decrypt_digest_update);
512         struct_assign(s10_param, native_param, fl_list.fl_sign_encrypt_update);
513         struct_assign(s10_param, native_param,
514             fl_list.fl_decrypt_verify_update);

516         struct_assign(s10_param, native_param, fl_list.fl_seed_random);
517         struct_assign(s10_param, native_param, fl_list.fl_generate_random);

519         struct_assign(s10_param, native_param, fl_list.fl_session_open);
520         struct_assign(s10_param, native_param, fl_list.fl_session_close);
521         struct_assign(s10_param, native_param, fl_list.fl_session_login);
522         struct_assign(s10_param, native_param, fl_list.fl_session_logout);
```

```
524         struct_assign(s10_param, native_param, fl_list.fl_object_create);
525         struct_assign(s10_param, native_param, fl_list.fl_object_copy);
526         struct_assign(s10_param, native_param, fl_list.fl_object_destroy);
527         struct_assign(s10_param, native_param, fl_list.fl_object_get_size);
528         struct_assign(s10_param, native_param,
529             fl_list.fl_object_get_attribute_value);
530         struct_assign(s10_param, native_param,
531             fl_list.fl_object_set_attribute_value);
532         struct_assign(s10_param, native_param, fl_list.fl_object_find_init);
533         struct_assign(s10_param, native_param, fl_list.fl_object_find);
534         struct_assign(s10_param, native_param, fl_list.fl_object_find_final);

536         struct_assign(s10_param, native_param, fl_list.fl_key_generate);
537         struct_assign(s10_param, native_param, fl_list.fl_key_generate_pair);
538         struct_assign(s10_param, native_param, fl_list.fl_key_wrap);
539         struct_assign(s10_param, native_param, fl_list.fl_key_unwrap);
540         struct_assign(s10_param, native_param, fl_list.fl_key_derive);

542         struct_assign(s10_param, native_param, fl_list.fl_init_token);
543         struct_assign(s10_param, native_param, fl_list.fl_init_pin);
544         struct_assign(s10_param, native_param, fl_list.fl_set_pin);

546         struct_assign(s10_param, native_param, fl_list.prov_is_hash_limited);
547         struct_assign(s10_param, native_param, fl_list.prov_hash_threshold);
548         struct_assign(s10_param, native_param, fl_list.prov_hash_limit);

550         return (brand_uucopy(&s10_param, (void *)arg, sizeof (s10_param)));
551 }

553 /*
554  * The process contract CT_TGET and CT_TSET parameter structure ct_param_t
555  * changed between S10 and Nevada, so we have to emulate the old S10
556  * ct_param_t structure when interposing on the ioctl syscall.
557  */
558 typedef struct s10_ct_param {
559         uint32_t ctpm_id;
560         uint32_t ctpm_pad;
561         uint64_t ctpm_value;
562 } s10_ct_param_t;

564 /*
565  * We have to emulate process contract ioctls for init(1M) because the
566  * ioctl parameter structure changed between S10 and Nevada.  This is
567  * a relatively simple process of filling Nevada structure fields,
568  * shuffling values, and initiating a native system call.
569  *
570  * For now, we'll assume that all consumers of CT_TGET and CT_TSET will
571  * need emulation.  We'll issue a stat to make sure that the ioctl
572  * is meant for the contract file system.
573  *
574  */
575 static int
576 ctfs_ioctl(sysret_t *rval, int fdes, int cmd, intptr_t arg)
577 {
578         int err;
579         s10_ct_param_t s10param;
580         ct_param_t param;
581         struct stat statbuf;

583         if ((err = __systemcall(rval, SYS_fstatat + 1024,
584             fdes, NULL, &statbuf, 0)) != 0)
585                 return (err);
586         if (strcmp(statbuf.st_fstype, MNTTYPE_CTFS) != 0)
587                 return (__systemcall(rval, SYS_ioctl + 1024, fdes, cmd, arg));

589         if (brand_uucopy((const void *)arg, &s10param, sizeof (s10param)) != 0)
```



```
590                 return (EFAULT);
591         param.ctpm_id = s10param.ctpm_id;
592         param.ctpm_size = sizeof (uint64_t);
593         param.ctpm_value = &s10param.ctpm_value;
594         if ((err = __systemcall(rval, SYS_ioctl + 1024, fdes, cmd, &param))
595             != 0)
596                 return (err);

598         if (cmd == CT_TGET)
599                 return (brand_uucopy(&s10param, (void *)arg,
600                     sizeof (s10param)));

602         return (0);
603 }

605 /*
606  * ZFS ioctls have changed in each Solaris 10 (S10) release as well as in
607  * Solaris Next.  The brand wraps ZFS commands so that the native commands
608  * are used, but we want to be sure no command sneaks in that uses ZFS
609  * without our knowledge.  We'll abort the process if we see a ZFS ioctl.
610  */
611 static int
612 zfs_ioctl(sysret_t *rval, int fdes, int cmd, intptr_t arg)
613 {
614         static dev_t zfs_dev = (dev_t)-1;
615         int err;

617         if (passthru_otherdev_ioctl(&zfs_dev, ZFS_DEV, &err,
618             rval, fdes, cmd, arg) == 1)
619                 return (err);

621         brand_abort(0, "ZFS ioctl!");
622         /*NOTREACHED*/
623         return (0);
624 }

626 struct s10_lofi_ioctl {
627         uint32_t li_minor;
628         boolean_t li_force;
629         char li_filename[MAXPATHLEN + 1];
630 };

632 static int
633 lofi_ioctl(sysret_t *rval, int fdes, int cmd, intptr_t arg)
634 {
635         static dev_t lofi_dev = (dev_t)-1;
636         struct s10_lofi_ioctl s10_param;
637         struct lofi_ioctl native_param;
638         int err;

640         if (passthru_otherdev_ioctl(&lofi_dev, "/dev/lofictl", &err,
641             rval, fdes, cmd, arg) == 1)
642                 return (err);

644         if (brand_uucopy((const void *)arg, &s10_param,
645             sizeof (s10_param)) != 0)
646                 return (EFAULT);

648         /*
649          * Somewhat weirdly, EIO is what the S10 lofi driver would
650          * return for unrecognised cmds.
651          */
652         if (cmd >= LOFI_CHECK_COMPRESSED)
653                 return (EIO);

655         bzero(&native_param, sizeof (native_param));
```

```
657         struct_assign(native_param, s10_param, li_minor);
658         struct_assign(native_param, s10_param, li_force);

660         /*
661          * Careful here, this has changed from [MAXPATHLEN + 1] to
662          * [MAXPATHLEN].
663          */
664         bcopy(s10_param.li_filename, native_param.li_filename,
665             sizeof (native_param.li_filename));
666         native_param.li_filename[MAXPATHLEN - 1] = '\0';

668         err = __systemcall(rval, SYS_ioctl + 1024, fdes, cmd, &native_param);

670         struct_assign(s10_param, native_param, li_minor);
671         /* li_force is input-only */

673         bcopy(native_param.li_filename, s10_param.li_filename,
674             sizeof (native_param.li_filename));

676         (void) brand_uucopy(&s10_param, (void *)arg, sizeof (s10_param));
677         return (err);
678 }

680 int
681 s10_ioctl(sysret_t *rval, int fdes, int cmd, intptr_t arg)
682 {
683         switch (cmd) {
684         case CRYPTO_GET_FUNCTION_LIST:
685                 return (crypto_ioctl(rval, fdes, cmd, arg));
686         case CT_TGET:
687                 /*FALLTHRU*/
688         case CT_TSET:
689                 return (ctfs_ioctl(rval, fdes, cmd, arg));
690         case MNTIOC_GETMNTENT:
691                 /*FALLTHRU*/
692         case MNTIOC_GETEXTMNTENT:
693                 /*FALLTHRU*/
694         case MNTIOC_GETMNTANY:
695                 return (mntfs_ioctl(rval, fdes, cmd, arg));
696         }

698         switch (cmd & ~0xff) {
699         case ZFS_IOC:
700                 return (zfs_ioctl(rval, fdes, cmd, arg));

702         case LOFI_IOC_BASE:
703                 return (lofi_ioctl(rval, fdes, cmd, arg));

705         default:
706                 break;
707         }

709         return (__systemcall(rval, SYS_ioctl + 1024, fdes, cmd, arg));
710 }

712 /*
713  * Unfortunately, pwrite()'s behavior differs between S10 and Nevada when
714  * applied to files opened with O_APPEND.  The offset argument is ignored and
715  * the buffer is appended to the target file in S10, whereas the current file
716  * position is ignored in Nevada (i.e., pwrite() acts as though the target file
717  * wasn't opened with O_APPEND).  This is a result of the fix for CR 6655660
718  * (pwrite() must ignore the O_APPEND/FAPPEND flag).
719  *
720  * We emulate the old S10 pwrite() behavior by checking whether the target file
721  * was opened with O_APPEND.  If it was, then invoke the write() system call
```



```
722  * instead of pwrite(); otherwise, invoke the pwrite() system call as usual.
723  */
724 static int
725 s10_pwrite(sysret_t *rval, int fd, const void *bufferp, size_t num_bytes,
726     off_t offset)
727 {
728         int err;

730         if ((err = __systemcall(rval, SYS_fcntl + 1024, fd, F_GETFL)) != 0)
731                 return (err);
732         if (rval->sys_rval1 & O_APPEND)
733                 return (__systemcall(rval, SYS_write + 1024, fd, bufferp,
734                     num_bytes));
735         return (__systemcall(rval, SYS_pwrite + 1024, fd, bufferp, num_bytes,
736             offset));
737 }

739 #if !defined(_LP64)
740 /*
741  * This is the large file version of the pwrite() system call for 32-bit
742  * processes.  This exists for the same reason that s10_pwrite() exists; see
743  * the comment above s10_pwrite().
744  */
745 static int
746 s10_pwrite64(sysret_t *rval, int fd, const void *bufferp, size32_t num_bytes,
747     uint32_t offset_1, uint32_t offset_2)
748 {
749         int err;

751         if ((err = __systemcall(rval, SYS_fcntl + 1024, fd, F_GETFL)) != 0)
752                 return (err);
753         if (rval->sys_rval1 & O_APPEND)
754                 return (__systemcall(rval, SYS_write + 1024, fd, bufferp,
755                     num_bytes));
756         return (__systemcall(rval, SYS_pwrite64 + 1024, fd, bufferp,
757             num_bytes, offset_1, offset_2));
758 }
759 #endif  /* !_LP64 */

761 /*
762  * These are convenience macros that s10_getdents_common() uses.  Both treat
763  * their arguments, which should be character pointers, as dirent pointers or
764  * dirent64 pointers and yield their d_name and d_reclen fields.  These
765  * macros shouldn't be used outside of s10_getdents_common().
766  */
767 #define dirent_name(charptr)    ((charptr) + name_offset)
768 #define dirent_reclen(charptr)  \
769         (*(unsigned short *)(uintptr_t)((charptr) + reclen_offset))

771 /*
772  * This function contains code that is common to both s10_getdents() and
773  * s10_getdents64().  See the comment above s10_getdents() for details.
774  *
775  * rval, fd, buf, and nbyte should be passed unmodified from s10_getdents()
776  * and s10_getdents64().  getdents_syscall_id should be either SYS_getdents
777  * or SYS_getdents64.  name_offset should be the the byte offset of
778  * the d_name field in the dirent structures passed to the kernel via the
779  * syscall represented by getdents_syscall_id.  reclen_offset should be
780  * the byte offset of the d_reclen field in the aforementioned dirent
781  * structures.
782  */
783 static int
784 s10_getdents_common(sysret_t *rval, int fd, char *buf, size_t nbyte,
785     int getdents_syscall_id, size_t name_offset, size_t reclen_offset)
786 {
787         int err;
```

```
788         size_t buf_size;
789         char *local_buf;
790         char *buf_current;
791
792         /*
793          * Use a special brand operation, B_S10_ISFDXATTRDIR, to determine
794          * whether the specified file descriptor refers to an extended file
795          * attribute directory.  If it doesn't, then SYS_getdents won't
796          * reveal extended file attributes, in which case we can simply
797          * hand the syscall to the native kernel.
798          */
799         if ((err = __systemcall(rval, SYS_brand + 1024, B_S10_ISFDXATTRDIR,
800             fd)) != 0)
801                 return (err);
802         if (rval->sys_rval1 == 0)
803                 return (__systemcall(rval, getdents_syscall_id + 1024, fd, buf,
804                     nbyte));
805
806         /*
807          * The file descriptor refers to an extended file attributes directory.
808          * We need to create a dirent buffer that's as large as buf into which
809          * the native SYS_getdents will store the special extended file
810          * attribute directory's entries.  We can't dereference buf because
811          * it might be an invalid pointer!
812          */
813         if (nbyte > MAXGETDENTS_SIZE)
814                 nbyte = MAXGETDENTS_SIZE;
815         local_buf = (char *)malloc(nbyte);
816         if (local_buf == NULL) {
817                 /*
818                  * getdents(2) doesn't return an error code indicating a memory
819                  * allocation error and it doesn't make sense to return any of
820                  * its documented error codes for a malloc(3C) failure.  We'll
821                  * use ENOMEM even though getdents(2) doesn't use it because it
822                  * best describes the failure.
823                  */
824                 (void) B_TRUSS_POINT_3(rval, getdents_syscall_id, ENOMEM, fd,
825                     buf, nbyte);
826                 rval->sys_rval1 = -1;
827                 rval->sys_rval2 = 0;
828                 return (EIO);
829         }
830
831         /*
832          * Issue a native SYS_getdents syscall but use our local dirent buffer
833          * instead of buf.  This will allow us to examine the returned dirent
834          * structures immediately and copy them to buf later.  That way the
835          * calling process won't be able to see the dirent structures until
836          * we finish examining them.
837          */
838         if ((err = __systemcall(rval, getdents_syscall_id + 1024, fd, local_buf,
839             nbyte)) != 0) {
840                 free(local_buf);
841                 return (err);
842         }
843         buf_size = rval->sys_rval1;
844         if (buf_size == 0) {
845                 free(local_buf);
846                 return (0);
847         }
848
849         /*
850          * Look for SUNWattr_ro (VIEW_READONLY) and SUNWattr_rw
851          * (VIEW_READWRITE) in the directory entries and remove them
852          * from the dirent buffer.
853          */
```



```
854         for (buf_current = local_buf;
855             (size_t)(buf_current - local_buf) < buf_size; /* cstyle */) {
856                 if (strcmp(dirent_name(buf_current), VIEW_READONLY) != 0 &&
857                     strcmp(dirent_name(buf_current), VIEW_READWRITE) != 0) {
858                         /*
859                          * The dirent refers to an attribute that should
860                          * be visible to Solaris 10 processes.  Keep it
861                          * and examine the next entry in the buffer.
862                          */
863                         buf_current += dirent_reclen(buf_current);
864                 } else {
865                         /*
866                          * We found either SUNWattr_ro (VIEW_READONLY)
867                          * or SUNWattr_rw (VIEW_READWRITE).  Remove it
868                          * from the dirent buffer by decrementing
869                          * buf_size by the size of the entry and
870                          * overwriting the entry with the remaining
871                          * entries.
872                          */
873                         buf_size -= dirent_reclen(buf_current);
874                         (void) memmove(buf_current, buf_current +
875                             dirent_reclen(buf_current), buf_size -
876                             (size_t)(buf_current - local_buf));
877                 }
878         }
879
880         /*
881          * Copy local_buf into buf so that the calling process can see
882          * the results.
883          */
884         if ((err = brand_uucopy(local_buf, buf, buf_size)) != 0) {
885                 free(local_buf);
886                 rval->sys_rval1 = -1;
887                 rval->sys_rval2 = 0;
888                 return (err);
889         }
890         rval->sys_rval1 = buf_size;
891         free(local_buf);
892         return (0);
893 }
894
895 /*
896  * Solaris Next added two special extended file attributes, SUNWattr_ro and
897  * SUNWattr_rw, which are called "extended system attributes".  They have
898  * special semantics (e.g., a process cannot unlink SUNWattr_ro) and should
899  * not appear in solaris10-branded zones because no Solaris 10 applications,
900  * including system commands such as tar(1), are coded to correctly handle these
901  * special attributes.
902  *
903  * This emulation function solves the aforementioned problem by emulating
904  * the getdents(2) syscall and filtering both system attributes out of resulting
905  * directory entry lists.  The emulation function only filters results when
906  * the given file descriptor refers to an extended file attribute directory.
907  * Filtering getdents(2) results is expensive because it requires dynamic
908  * memory allocation; however, the performance cost is tolerable because
909  * we don't expect Solaris 10 processes to frequently examine extended file
910  * attribute directories.
911  *
912  * The brand's emulation library needs two getdents(2) emulation functions
913  * because getdents(2) comes in two flavors: non-largefile-aware getdents(2)
914  * and largefile-aware getdents64(2).  s10_getdents() handles the non-largefile-
915  * aware case for 32-bit processes and all getdents(2) syscalls for 64-bit
916  * processes (64-bit processes use largefile-aware interfaces by default).
917  * See s10_getdents64() below for the largefile-aware getdents64(2) emulation
918  * function for 32-bit processes.
919  */
```

```
static int
s10_getdents(sysret_t *rval, int fd, struct dirent *buf, size_t nbyte)
{
        return (s10_getdents_common(rval, fd, (char *)buf, nbyte, SYS_getdents,
            offsetof(struct dirent, d_name),
            offsetof(struct dirent, d_reclen)));
}

#ifndef _LP64
/*
 * This is the largefile-aware version of getdents(2) for 32-bit processes.
 * This exists for the same reason that s10_getdents() exists.  See the comment
 * above s10_getdents().
 */
static int
s10_getdents64(sysret_t *rval, int fd, struct dirent64 *buf, size_t nbyte)
{
        return (s10_getdents_common(rval, fd, (char *)buf, nbyte,
            SYS_getdents64, offsetof(struct dirent64, d_name),
            offsetof(struct dirent64, d_reclen)));
}
#endif  /* !_LP64 */

#define S10_TRIVIAL_ACL_CNT     6
#define NATIVE_TRIVIAL_ACL_CNT  3

/*
 * Check if the ACL qualifies as a trivial ACL based on the native
 * interpretation.
 */
static boolean_t
has_trivial_native_acl(int cmd, int cnt, const char *fname, int fd)
{
        int i, err;
        sysret_t rval;
        ace_t buf[NATIVE_TRIVIAL_ACL_CNT];

        if (fname != NULL)
                err = __systemcall(&rval, SYS_pathconf + 1024, fname,
                    _PC_ACL_ENABLED);
        else
                err = __systemcall(&rval, SYS_fpathconf + 1024, fd,
                    _PC_ACL_ENABLED);
        if (err != 0 || rval.sys_rval1 != _ACL_ACE_ENABLED)
                return (B_FALSE);

        /*
         * If we just got the ACL cnt, we don't need to get it again, its
         * passed in as the cnt arg.
         */
        if (cmd != ACE_GETACLCNT) {
                if (fname != NULL) {
                        if (__systemcall(&rval, SYS_acl + 1024, fname,
                            ACE_GETACLCNT, 0, NULL) != 0)
                                return (B_FALSE);
                } else {
                        if (__systemcall(&rval, SYS_facl + 1024, fd,
                            ACE_GETACLCNT, 0, NULL) != 0)
                                return (B_FALSE);
                }
                cnt = rval.sys_rval1;
        }

        if (cnt != NATIVE_TRIVIAL_ACL_CNT)
                return (B_FALSE);
```



```
        if (fname != NULL) {
                if (__systemcall(&rval, SYS_acl + 1024, fname, ACE_GETACL, cnt,
                    buf) != 0)
                        return (B_FALSE);
        } else {
                if (__systemcall(&rval, SYS_facl + 1024, fd, ACE_GETACL, cnt,
                    buf) != 0)
                        return (B_FALSE);
        }

        /*
         * The following is based on the logic from the native OS
         * ace_trivial_common() to determine if the native ACL is trivial.
         */
        for (i = 0; i < cnt; i++) {
                switch (buf[i].a_flags & ACE_TYPE_FLAGS) {
                case ACE_OWNER:
                case ACE_GROUP|ACE_IDENTIFIER_GROUP:
                case ACE_EVERYONE:
                        break;
                default:
                        return (B_FALSE);
                }

                if (buf[i].a_flags & (ACE_FILE_INHERIT_ACE|
                    ACE_DIRECTORY_INHERIT_ACE|ACE_NO_PROPAGATE_INHERIT_ACE|
                    ACE_INHERIT_ONLY_ACE))
                        return (B_FALSE);

                /*
                 * Special check for some special bits
                 *
                 * Don't allow anybody to deny reading basic
                 * attributes or a files ACL.
                 */
                if (buf[i].a_access_mask & (ACE_READ_ACL|ACE_READ_ATTRIBUTES) &&
                    buf[i].a_type == ACE_ACCESS_DENIED_ACE_TYPE)
                        return (B_FALSE);

                /*
                 * Delete permissions are never set by default
                 */
                if (buf[i].a_access_mask & (ACE_DELETE|ACE_DELETE_CHILD))
                        return (B_FALSE);
                /*
                 * only allow owner@ to have
                 * write_acl/write_owner/write_attributes/write_xattr/
                 */
                if (buf[i].a_type == ACE_ACCESS_ALLOWED_ACE_TYPE &&
                    (!(buf[i].a_flags & ACE_OWNER) && (buf[i].a_access_mask &
                    (ACE_WRITE_OWNER|ACE_WRITE_ACL| ACE_WRITE_ATTRIBUTES|
                    ACE_WRITE_NAMED_ATTRS))))
                        return (B_FALSE);

        }

        return (B_TRUE);
}

/*
 * The following logic is based on the S10 adjust_ace_pair_common() code.
 */
static void
s10_adjust_ace_mask(void *pair, size_t access_off, size_t pairsize, mode_t mode)
{
        char *datap = (char *)pair;
```

```
1052         uint32_t *amask0 = (uint32_t *)(uintptr_t)(datap + access_off);
1053         uint32_t *amask1 = (uint32_t *)(uintptr_t)(datap + pairsize +
1054             access_off);

1056         if (mode & S_IROTH)
1057                 *amask1 |= ACE_READ_DATA;
1058         else
1059                 *amask0 |= ACE_READ_DATA;
1060         if (mode & S_IWOTH)
1061                 *amask1 |= ACE_WRITE_DATA|ACE_APPEND_DATA;
1062         else
1063                 *amask0 |= ACE_WRITE_DATA|ACE_APPEND_DATA;
1064         if (mode & S_IXOTH)
1065                 *amask1 |= ACE_EXECUTE;
1066         else
1067                 *amask0 |= ACE_EXECUTE;
1068 }

1070 /*
1071  * Construct a trivial S10 style ACL.
1072  */
1073 static int
1074 make_trivial_s10_acl(const char *fname, int fd, ace_t *bp)
1075 {
1076         int err;
1077         sysret_t rval;
1078         struct stat64 buf;
1079         ace_t trivial_s10_acl[] = {
1080                 {(uint_t)-1, 0, ACE_OWNER, ACE_ACCESS_DENIED_ACE_TYPE},
1081                 {(uint_t)-1, ACE_WRITE_ACL|ACE_WRITE_OWNER|ACE_WRITE_ATTRIBUTES|
1082                     ACE_WRITE_NAMED_ATTRS, ACE_OWNER,
1083                     ACE_ACCESS_ALLOWED_ACE_TYPE},
1084                 {(uint_t)-1, 0, ACE_GROUP|ACE_IDENTIFIER_GROUP,
1085                     ACE_ACCESS_DENIED_ACE_TYPE},
1086                 {(uint_t)-1, 0, ACE_GROUP|ACE_IDENTIFIER_GROUP,
1087                     ACE_ACCESS_ALLOWED_ACE_TYPE},
1088                 {(uint_t)-1, ACE_WRITE_ACL|ACE_WRITE_OWNER|ACE_WRITE_ATTRIBUTES|
1089                     ACE_WRITE_NAMED_ATTRS, ACE_EVERYONE,
1090                     ACE_ACCESS_DENIED_ACE_TYPE},
1091                 {(uint_t)-1, ACE_READ_ACL|ACE_READ_ATTRIBUTES|
1092                     ACE_READ_NAMED_ATTRS|ACE_SYNCHRONIZE, ACE_EVERYONE,
1093                     ACE_ACCESS_ALLOWED_ACE_TYPE}
1094         };

1096         if (fname != NULL) {
1097                 if ((err = __systemcall(&rval, SYS_fstatat64 + 1024, AT_FDCWD,
1098                     fname, &buf, 0)) != 0)
1099                         return (err);
1100         } else {
1101                 if ((err = __systemcall(&rval, SYS_fstatat64 + 1024, fd,
1102                     NULL, &buf, 0)) != 0)
1103                         return (err);
1104         }

1106         s10_adjust_ace_mask(&trivial_s10_acl[0], offsetof(ace_t, a_access_mask),
1107             sizeof (ace_t), (buf.st_mode & 0700) >> 6);
1108         s10_adjust_ace_mask(&trivial_s10_acl[2], offsetof(ace_t, a_access_mask),
1109             sizeof (ace_t), (buf.st_mode & 0070) >> 3);
1110         s10_adjust_ace_mask(&trivial_s10_acl[4], offsetof(ace_t, a_access_mask),
1111             sizeof (ace_t), buf.st_mode & 0007);

1113         if (brand_uucopy(&trivial_s10_acl, bp, sizeof (trivial_s10_acl)) != 0)
1114                 return (EFAULT);

1116         return (0);
1117 }
```



```
1119 /*
1120  * The definition of a trivial ace-style ACL (used by ZFS and NFSv4) has been
1121  * simplified since S10.  Instead of 6 entries on a trivial S10 ACE ACL we now
1122  * have 3 streamlined entries.  The new, simpler trivial style confuses S10
1123  * commands such as 'ls -v' or 'cp -p' which don't see the expected S10 trivial
1124  * ACL entries and thus assume that there is a complex ACL on the file.
1125  *
1126  * See: PSARC/2010/029 Improved ACL interoperability
1127  *
1128  * Note that the trival ACL detection code is implemented in acl_trival() in
1129  * lib/libsec/common/aclutils.c.  It always uses the acl() syscall (not the
1130  * facl syscall) to determine if an ACL is trivial.  However, we emulate both
1131  * acl() and facl() so that the two provide consistent results.
1132  *
1133  * We don't currently try to emulate setting of ACLs since the primary
1134  * consumer of this feature is SMB or NFSv4 servers, neither of which are
1135  * supported in solaris10-branded zones.  If ACLs are used they must be set on
1136  * files using the native OS interpretation.
1137  */
1138 int
1139 s10_acl(sysret_t *rval, const char *fname, int cmd, int nentries, void *aclbufp)
1140 {
1141         int res;

1143         res = __systemcall(rval, SYS_acl + 1024, fname, cmd, nentries, aclbufp);

1145         switch (cmd) {
1146         case ACE_GETACLCNT:
1147                 if (res == 0 && has_trivial_native_acl(ACE_GETACLCNT,
1148                     rval->sys_rval1, fname, 0)) {
1149                         rval->sys_rval1 = S10_TRIVIAL_ACL_CNT;
1150                 }
1151                 break;
1152         case ACE_GETACL:
1153                 if (res == 0 &&
1154                     has_trivial_native_acl(ACE_GETACL, 0, fname, 0) &&
1155                     nentries >= S10_TRIVIAL_ACL_CNT) {
1156                         res = make_trivial_s10_acl(fname, 0, aclbufp);
1157                         rval->sys_rval1 = S10_TRIVIAL_ACL_CNT;
1158                 }
1159                 break;
1160         }

1162         return (res);
1163 }

1165 int
1166 s10_facl(sysret_t *rval, int fdes, int cmd, int nentries, void *aclbufp)
1167 {
1168         int res;

1170         res = __systemcall(rval, SYS_facl + 1024, fdes, cmd, nentries, aclbufp);

1172         switch (cmd) {
1173         case ACE_GETACLCNT:
1174                 if (res == 0 && has_trivial_native_acl(ACE_GETACLCNT,
1175                     rval->sys_rval1, NULL, fdes)) {
1176                         rval->sys_rval1 = S10_TRIVIAL_ACL_CNT;
1177                 }
1178                 break;
1179         case ACE_GETACL:
1180                 if (res == 0 &&
1181                     has_trivial_native_acl(ACE_GETACL, 0, NULL, fdes) &&
1182                     nentries >= S10_TRIVIAL_ACL_CNT) {
1183                         res = make_trivial_s10_acl(NULL, fdes, aclbufp);
```

```
                        rval->sys_rval1 = S10_TRIVIAL_ACL_CNT;
                }
                break;
        }

        return (res);
}

#define S10_AC_PROC             (0x1 << 28)
#define S10_AC_TASK             (0x2 << 28)
#define S10_AC_FLOW             (0x4 << 28)
#define S10_AC_MODE(x)          ((x) & 0xf0000000)
#define S10_AC_OPTION(x)        ((x) & 0x0fffffff)

/*
 * The mode shift, mode mask and option mask for acctctl have changed.  The
 * mode is currently the top full byte and the option is the lower 3 full bytes.
 */
int
s10_acctctl(sysret_t *rval, int cmd, void *buf, size_t bufsz)
{
        int mode = S10_AC_MODE(cmd);
        int option = S10_AC_OPTION(cmd);

        switch (mode) {
        case S10_AC_PROC:
                mode = AC_PROC;
                break;
        case S10_AC_TASK:
                mode = AC_TASK;
                break;
        case S10_AC_FLOW:
                mode = AC_FLOW;
                break;
        default:
                return (B_TRUSS_POINT_3(rval, SYS_acctctl, EINVAL, cmd, buf,
                    bufsz));
        }

        return (__systemcall(rval, SYS_acctctl + 1024, mode | option, buf,
            bufsz));
}

/*
 * The Audit Policy parameters have changed due to:
 *    6466722 adituser and AUDIT_USER are defined, unused, undocumented and
 *            should be removed.
 *
 * In S10 we had the following flag:
 *      #define AUDIT_USER 0x0040
 * which doesn't exist in Solaris Next where the subsequent flags are shifted
 * down.  For example, in S10 we had:
 *      #define AUDIT_GROUP     0x0080
 * but on Solaris Next we have:
 *      #define AUDIT_GROUP     0x0040
 * AUDIT_GROUP has the value AUDIT_USER had in S10 and all of the subsequent
 * bits are also shifted one place.
 *
 * When we're getting or setting the Audit Policy parameters we need to
 * shift the outgoing or incoming bits into their proper positions.  Since
 * S10_AUDIT_USER was always unused, we always clear that bit on A_GETPOLICY.
 *
 * The command we care about, BSM_AUDITCTL, passes the most parameters (3),
 * so declare this function to take up to 4 args and just pass them on.
 * The number of parameters for s10_auditsys needs to be equal to the BSM_*
 * subcommand that has the most parameters, since we want to pass all
```



```
 * parameters through, regardless of which subcommands we interpose on.
 *
 * Note that the auditsys system call uses the SYSENT_AP macro wrapper instead
 * of the more common SYSENT_CI macro.  This means the return value is a
 * SE_64RVAL so the syscall table uses RV_64RVAL.
 */

#define S10_AUDIT_HMASK 0xffffffc0
#define S10_AUDIT_LMASK 0x3f
#define S10_AUC_NOSPACE 0x3

int
s10_auditsys(sysret_t *rval, int bsmcmd, intptr_t a0, intptr_t a1, intptr_t a2)
{
        int         err;
        uint32_t    m;

        if (bsmcmd != BSM_AUDITCTL)
                return (__systemcall(rval, SYS_auditsys + 1024, bsmcmd, a0, a1,
                    a2));

        if ((int)a0 == A_GETPOLICY) {
                if ((err = __systemcall(rval, SYS_auditsys + 1024, bsmcmd, a0,
                    &m, a2)) != 0)
                        return (err);
                m = ((m & S10_AUDIT_HMASK) << 1) | (m & S10_AUDIT_LMASK);
                if (brand_uucopy(&m, (void *)a1, sizeof (m)) != 0)
                        return (EFAULT);
                return (0);

        } else if ((int)a0 == A_SETPOLICY) {
                if (brand_uucopy((const void *)a1, &m, sizeof (m)) != 0)
                        return (EFAULT);
                m = ((m >> 1) & S10_AUDIT_HMASK) | (m & S10_AUDIT_LMASK);
                return (__systemcall(rval, SYS_auditsys + 1024, bsmcmd, a0, &m,
                    a2));
        } else if ((int)a0 == A_GETCOND) {
                if ((err = __systemcall(rval, SYS_auditsys + 1024, bsmcmd, a0,
                    &m, a2)) != 0)
                        return (err);
                if (m == AUC_NOSPACE)
                        m = S10_AUC_NOSPACE;
                if (brand_uucopy(&m, (void *)a1, sizeof (m)) != 0)
                        return (EFAULT);
                return (0);
        } else if ((int)a0 == A_SETCOND) {
                if (brand_uucopy((const void *)a1, &m, sizeof (m)) != 0)
                        return (EFAULT);
                if (m == S10_AUC_NOSPACE)
                        m = AUC_NOSPACE;
                return (__systemcall(rval, SYS_auditsys + 1024, bsmcmd, a0, &m,
                    a2));
        }

        return (__systemcall(rval, SYS_auditsys + 1024, bsmcmd, a0, a1, a2));
}

/*
 * Determine whether the executable passed to SYS_exec or SYS_execve is a
 * native executable.  The s10_npreload.so invokes the B_S10_NATIVE brand
 * operation which patches up the processes exec info to eliminate any trace
 * of the wrapper.  That will make pgrep and other commands that examine
 * process' executable names and command-line parameters work properly.
 */
static int
s10_exec_native(sysret_t *rval, const char *fname, const char **argp,
```

```
1316     const char **envp)
1317 {
1318         const char *filename = fname;
1319         char path[64];
1320         int err;

1322         /* Get a copy of the executable we're trying to run */
1323         path[0] = '\0';
1324         (void) brand_uucopystr(filename, path, sizeof (path));

1326         /* Check if we're trying to run a native binary */
1327         if (strncmp(path, "/.SUNWnative/usr/lib/brand/solaris10/s10_native",
1328             sizeof (path)) != 0)
1329                 return (0);

1331         /* Skip the first element in the argv array */
1332         argp++;

1334         /*
1335          * The the path of the dynamic linker is the second parameter
1336          * of s10_native_exec().
1337          */
1338         if (brand_uucopy(argp, &filename, sizeof (char *)) != 0)
1339                 return (EFAULT);

1341         /* If an exec call succeeds, it never returns */
1342         err = __systemcall(rval, SYS_brand + 1024, B_EXEC_NATIVE, filename,
1343             argp, envp, NULL, NULL, NULL);
1344         brand_assert(err != 0);
1345         return (err);
1346 }

1348 /*
1349  * Interpose on the SYS_exec syscall to detect native wrappers.
1350  */
1351 int
1352 s10_exec(sysret_t *rval, const char *fname, const char **argp)
1353 {
1354         int err;

1356         if ((err = s10_exec_native(rval, fname, argp, NULL)) != 0)
1357                 return (err);

1359         /* If an exec call succeeds, it never returns */
1360         err = __systemcall(rval, SYS_execve + 1024, fname, argp, NULL);
1361         brand_assert(err != 0);
1362         return (err);
1363 }

1365 /*
1366  * Interpose on the SYS_execve syscall to detect native wrappers.
1367  */
1368 int
1369 s10_execve(sysret_t *rval, const char *fname, const char **argp,
1370     const char **envp)
1371 {
1372         int err;

1374         if ((err = s10_exec_native(rval, fname, argp, envp)) != 0)
1375                 return (err);

1377         /* If an exec call succeeds, it never returns */
1378         err = __systemcall(rval, SYS_execve + 1024, fname, argp, envp);
1379         brand_assert(err != 0);
1380         return (err);
1381 }
```



```
1383 /*
1384  * S10's issetugid() syscall is now a subcode to privsys().
1385  */
1386 static int
1387 s10_issetugid(sysret_t *rval)
1388 {
1389         return (__systemcall(rval, SYS_privsys + 1024, PRIVSYS_ISSETUGID,
1390             0, 0, 0, 0, 0));
1391 }

1393 /*
1394  * S10's socket() syscall does not split type and flags
1395  */
1396 static int
1397 s10_so_socket(sysret_t *rval, int domain, int type, int protocol,
1398     char *devpath, int version)
1399 {
1400         if ((type & ~SOCK_TYPE_MASK) != 0) {
1401                 errno = EINVAL;
1402                 return (-1);
1403         }
1404         return (__systemcall(rval, SYS_so_socket + 1024, domain, type,
1405             protocol, devpath, version));
1406 }

1408 /*
1409  * S10's pipe() syscall has a different calling convention
1410  */
1411 static int
1412 s10_pipe(sysret_t *rval)
1413 {
1414         int fds[2], err;
1415         if ((err = __systemcall(rval, SYS_pipe + 1024, fds, 0)) != 0)
1416                 return (err);

1418         rval->sys_rval1 = fds[0];
1419         rval->sys_rval2 = fds[1];
1420         return (0);
1421 }

1423 /*
1424  * S10's accept() syscall takes three arguments
1425  */
1426 static int
1427 s10_accept(sysret_t *rval, int sock, struct sockaddr *addr, uint_t *addrlen,
1428     int version)
1429 {
1430         return (__systemcall(rval, SYS_accept + 1024, sock, addr, addrlen,
1431             version, 0));
1432 }

1434 static long
1435 s10_uname(sysret_t *rv, uintptr_t p1)
1436 {
1437         struct utsname un, *unp = (struct utsname *)p1;
1438         int rev, err;

1440         if ((err = __systemcall(rv, SYS_uname + 1024, &un)) != 0)
1441                 return (err);

1443         rev = atoi(&un.release[2]);
1444         brand_assert(rev >= 11);
1445         bzero(un.release, _SYS_NMLN);
1446         (void) strlcpy(un.release, S10_UTS_RELEASE, _SYS_NMLN);
1447         bzero(un.version, _SYS_NMLN);
```

```
1448         (void) strlcpy(un.version, S10_UTS_VERSION, _SYS_NMLN);

1450         /* copy out the modified uname info */
1451         return (brand_uucopy(&un, unp, sizeof (un)));
1452 }

1454 int
1455 s10_sysconfig(sysret_t *rv, int which)
1456 {
1457         long value;

1459         /*
1460          * We must interpose on the sysconfig(2) requests
1461          * that deal with the realtime signal number range.
1462          * All others get passed to the native sysconfig(2).
1463          */
1464         switch (which) {
1465         case _CONFIG_RTSIG_MAX:
1466                 value = S10_SIGRTMAX - S10_SIGRTMIN + 1;
1467                 break;
1468         case _CONFIG_SIGRT_MIN:
1469                 value = S10_SIGRTMIN;
1470                 break;
1471         case _CONFIG_SIGRT_MAX:
1472                 value = S10_SIGRTMAX;
1473                 break;
1474         default:
1475                 return (__systemcall(rv, SYS_sysconfig + 1024, which));
1476         }

1478         (void) B_TRUSS_POINT_1(rv, SYS_sysconfig, 0, which);
1479         rv->sys_rval1 = value;
1480         rv->sys_rval2 = 0;

1482         return (0);
1483 }

1485 int
1486 s10_sysinfo(sysret_t *rv, int command, char *buf, long count)
1487 {
1488         char *value;
1489         int len;

1491         /*
1492          * We must interpose on the sysinfo(2) commands SI_RELEASE and
1493          * SI_VERSION; all others get passed to the native sysinfo(2)
1494          * command.
1495          */
1496         switch (command) {
1497                 case SI_RELEASE:
1498                         value = S10_UTS_RELEASE;
1499                         break;

1501                 case SI_VERSION:
1502                         value = S10_UTS_VERSION;
1503                         break;

1505                 default:
1506                         /*
1507                          * The default action is to pass the command to the
1508                          * native sysinfo(2) syscall.
1509                          */
1510                         return (__systemcall(rv, SYS_systeminfo + 1024,
1511                             command, buf, count));
1512         }
```



```
1514         len = strlen(value) + 1;
1515         if (count > 0) {
1516                 if (brand_uucopystr(value, buf, count) != 0)
1517                         return (EFAULT);

1519                 /*
1520                  * Assure NULL termination of buf as brand_uucopystr() doesn't.
1521                  */
1522                 if (len > count && brand_uucopy("\0", buf + (count - 1), 1)
1523                     != 0)
1524                         return (EFAULT);
1525         }

1527         /*
1528          * On success, sysinfo(2) returns the size of buffer required to hold
1529          * the complete value plus its terminating NULL byte.
1530          */
1531         (void) B_TRUSS_POINT_3(rv, SYS_systeminfo, 0, command, buf, count);
1532         rv->sys_rval1 = len;
1533         rv->sys_rval2 = 0;
1534         return (0);
1535 }

1537 #if defined(__x86)
1538 #if defined(__amd64)
1539 /*
1540  * 64-bit x86 LWPs created by SYS_lwp_create start here if they need to set
1541  * their %fs registers to the legacy Solaris 10 selector value.
1542  *
1543  * This function does three things:
1544  *
1545  *      1.  Trap to the kernel so that it can set %fs to the legacy Solaris 10
1546  *          selector value.
1547  *      2.  Read the LWP's true entry point (the entry point supplied by libc
1548  *          when SYS_lwp_create was invoked) from %r14.
1549  *      3.  Eliminate this function's stack frame and pass control to the LWP's
1550  *          true entry point.
1551  *
1552  * See the comment above s10_lwp_create_correct_fs() (see below) for the reason
1553  * why this function exists.
1554  */
1555 /*ARGSUSED*/
1556 static void
1557 s10_lwp_create_entry_point(void *ulwp_structp)
1558 {
1559         sysret_t rval;

1561         /*
1562          * The new LWP's %fs register is initially zero, but libc won't
1563          * function correctly when %fs is zero.  Change the LWP's %fs register
1564          * via SYS_brand.
1565          */
1566         (void) __systemcall(&rval, SYS_brand + 1024, B_S10_FSREGCORRECTION);

1568         /*
1569          * Jump to the true entry point, which is stored in %r14.
1570          * Remove our stack frame before jumping so that
1571          * s10_lwp_create_entry_point() won't be seen in stack traces.
1572          *
1573          * NOTE: s10_lwp_create_entry_point() pushes %r12 onto its stack frame
1574          * so that it can use it as a temporary register.  We don't restore %r12
1575          * in this assembly block because we don't care about its value (and
1576          * neither does _lwp_start()).  Besides, the System V ABI AMD64
1577          * Actirecture Processor Supplement doesn't specify that %r12 should
1578          * have a special value when LWPs start, so we can ignore its value when
1579          * we jump to the true entry point.  Furthermore, %r12 is a callee-saved
```

```
1580          * register, so the true entry point should push %r12 onto its stack
1581          * before using the register.  We ignore %r14 after we read it for
1582          * similar reasons.
1583          *
1584          * NOTE: The compiler will generate a function epilogue for this
1585          * function despite the fact that the LWP will never execute it.
1586          * We could hand-code this entire function in assembly to eliminate
1587          * the epilogue, but the epilogue is only three or four instructions,
1588          * so we wouldn't save much space.  Besides, why would we want
1589          * to create yet another ugly, hard-to-maintain assembly function when
1590          * we could write most of it in C?
1591          */
1592         __asm__ __volatile__(
1593             "movq %0, %%rdi\n\t"        /* pass ulwp_structp as arg1 */
1594             "movq %%rbp, %%rsp\n\t"     /* eliminate the stack frame */
1595             "popq %%rbp\n\t"
1596             "jmp *%%r14\n\t"            /* jump to the true entry point */
1597             : : "r" (ulwp_structp));
1598         /*NOTREACHED*/
1599 }

1601 /*
1602  * The S10 libc expects that %fs will be nonzero for new 64-bit x86 LWPs but the
1603  * Nevada kernel clears %fs for such LWPs.  Unforunately, new LWPs do not issue
1604  * SYS_lwp_private (see s10_lwp_private() below) after they are created, so
1605  * we must ensure that new LWPs invoke a brand operation that sets %fs to a
1606  * nonzero value immediately after their creation.
1607  *
1608  * The easiest way to do this is to make new LWPs start at a special function,
1609  * s10_lwp_create_entry_point() (see its definition above), that invokes the
1610  * brand operation that corrects %fs.  We'll store the entry points of new LWPs
1611  * in their %r14 registers so that s10_lwp_create_entry_point() can find and
1612  * call them after invoking the special brand operation.  %r14 is a callee-saved
1613  * register; therefore, any functions invoked by s10_lwp_create_entry_point()
1614  * and all functions dealing with signals (e.g., sigacthandler()) will preserve
1615  * %r14 for s10_lwp_create_entry_point().
1616  *
1617  * The Nevada kernel can safely work with nonzero %fs values because the kernel
1618  * configures per-thread %fs segment descriptors so that the legacy %fs selector
1619  * value will still work.  See the comment in lwp_load() regarding %fs and
1620  * %fsbase in 64-bit x86 processes.
1621  *
1622  * This emulation exists thanks to CRs 6467491 and 6501650.
1623  */
1624 static int
1625 s10_lwp_create_correct_fs(sysret_t *rval, ucontext_t *ucp, int flags,
1626     id_t *new_lwp)
1627 {
1628         ucontext_t s10_uc;

1630         /*
1631          * Copy the supplied ucontext_t structure to the local stack
1632          * frame and store the new LWP's entry point (the value of %rip
1633          * stored in the ucontext_t) in the new LWP's %r14 register.
1634          * Then make s10_lwp_create_entry_point() the new LWP's entry
1635          * point.
1636          */
1637         if (brand_uucopy(ucp, &s10_uc, sizeof (s10_uc)) != 0)
1638                 return (EFAULT);

1640         s10_uc.uc_mcontext.gregs[REG_R14] = s10_uc.uc_mcontext.gregs[REG_RIP];
1641         s10_uc.uc_mcontext.gregs[REG_RIP] = (greg_t)s10_lwp_create_entry_point;

1643         /*  fix up the signal mask */
1644         if (s10_uc.uc_flags & UC_SIGMASK)
1645                 (void) s10sigset_to_native(&s10_uc.uc_sigmask,
```



```
1646                     &s10_uc.uc_sigmask);

1648         /*
1649          * Issue SYS_lwp_create to create the new LWP.  We pass the
1650          * modified ucontext_t to make sure that the new LWP starts at
1651          * s10_lwp_create_entry_point().
1652          */
1653         return (__systemcall(rval, SYS_lwp_create + 1024, &s10_uc,
1654             flags, new_lwp));
1655 }
1656 #endif  /* __amd64 */

1658 /*
1659  * SYS_lwp_private is issued by libc_init() to set %fsbase in 64-bit x86
1660  * processes.  The Nevada kernel sets %fs to zero but the S10 libc expects
1661  * %fs to be nonzero.  We'll pass the issued system call to the kernel untouched
1662  * and invoke a brand operation to set %fs to the legacy S10 selector value.
1663  *
1664  * This emulation exists thanks to CRs 6467491 and 6501650.
1665  */
1666 static int
1667 s10_lwp_private(sysret_t *rval, int cmd, int which, uintptr_t base)
1668 {
1669 #if defined(__amd64)
1670         int err;

1672         /*
1673          * The current LWP's %fs register should be zero.  Determine whether the
1674          * Solaris 10 libc with which we're working functions correctly when %fs
1675          * is zero by calling thr_main() after issuing the SYS_lwp_private
1676          * syscall.  If thr_main() barfs (returns -1), then change the LWP's %fs
1677          * register via SYS_brand and patch brand_sysent_table so that issuing
1678          * SYS_lwp_create executes s10_lwp_create_correct_fs() rather than the
1679          * default s10_lwp_create().  s10_lwp_create_correct_fs() will
1680          * guarantee that new LWPs will have correct %fs values.
1681          */
1682         if ((err = __systemcall(rval, SYS_lwp_private + 1024, cmd, which,
1683             base)) != 0)
1684                 return (err);
1685         if (thr_main() == -1) {
1686                 /*
1687                  * SYS_lwp_private is only issued by libc_init(), which is
1688                  * executed when libc is first loaded by ld.so.1.  Thus we
1689                  * are guaranteed to be single-threaded at this point.  Even
1690                  * if we were multithreaded at this point, writing a 64-bit
1691                  * value to the st_callc field of a brand_sysent_table
1692                  * entry is guaranteed to be atomic on 64-bit x86 chips
1693                  * as long as the field is not split across cache lines
1694                  * (It shouldn't be.).  See chapter 8, section 1.1 of
1695                  * "The Intel 64 and IA32 Architectures Software Developer's
1696                  * Manual," Volume 3A for more details.
1697                  */
1698                 brand_sysent_table[SYS_lwp_create].st_callc =
1699                     (sysent_cb_t)s10_lwp_create_correct_fs;
1700                 return (__systemcall(rval, SYS_brand + 1024,
1701                     B_S10_FSREGCORRECTION));
1702         }
1703         return (0);
1704 #else   /* !__amd64 */
1705         return (__systemcall(rval, SYS_lwp_private + 1024, cmd, which, base));
1706 #endif  /* !__amd64 */
1707 }
1708 #endif  /* __x86 */

1710 /*
1711  * The Opensolaris versions of lwp_mutex_timedlock() and lwp_mutex_trylock()
```

```
 * add an extra argument to the interfaces, a uintptr_t value for the mutex's
 * mutex_owner field.  The Solaris 10 libc assigns the mutex_owner field at
 * user-level, so we just make the extra argument be zero in both syscalls.
 */

static int
s10_lwp_mutex_timedlock(sysret_t *rval, lwp_mutex_t *lp, timespec_t *tsp)
{
        return (__systemcall(rval, SYS_lwp_mutex_timedlock + 1024, lp, tsp, 0));
}

static int
s10_lwp_mutex_trylock(sysret_t *rval, lwp_mutex_t *lp)
{
        return (__systemcall(rval, SYS_lwp_mutex_trylock + 1024, lp, 0));
}

/*
 * If the emul_global_zone flag is set then emulate some aspects of the
 * zone system call.  In particular, emulate the global zone ID on the
 * ZONE_LOOKUP subcommand and emulate some of the global zone attributes
 * on the ZONE_GETATTR subcommand.  If the flag is not set or we're performing
 * some other operation, simply pass the calls through.
 */
int
s10_zone(sysret_t *rval, int cmd, void *arg1, void *arg2, void *arg3,
    void *arg4)
{
        char            *aval;
        int             len;
        zoneid_t        zid;
        int             attr;
        char            *buf;
        size_t          bufsize;

        /*
         * We only emulate the zone syscall for a subset of specific commands,
         * otherwise we just pass the call through.
         */
        if (!emul_global_zone)
                return (__systemcall(rval, SYS_zone + 1024, cmd, arg1, arg2,
                    arg3, arg4));

        switch (cmd) {
        case ZONE_LOOKUP:
                (void) B_TRUSS_POINT_1(rval, SYS_zone, 0, cmd);
                rval->sys_rval1 = GLOBAL_ZONEID;
                rval->sys_rval2 = 0;
                return (0);

        case ZONE_GETATTR:
                zid = (zoneid_t)(uintptr_t)arg1;
                attr = (int)(uintptr_t)arg2;
                buf = (char *)arg3;
                bufsize = (size_t)arg4;

                /*
                 * If the request is for the global zone then we're emulating
                 * that, otherwise pass this thru.
                 */
                if (zid != GLOBAL_ZONEID)
                        goto passthru;

                switch (attr) {
                case ZONE_ATTR_NAME:
                        aval = GLOBAL_ZONENAME;
```



```
                        break;

                case ZONE_ATTR_BRAND:
                        aval = NATIVE_BRAND_NAME;
                        break;
                default:
                        /*
                         * We only emulate a subset of the attrs, use the
                         * real zone id to pass thru the rest.
                         */
                        arg1 = (void *)(uintptr_t)zoneid;
                        goto passthru;
                }

                (void) B_TRUSS_POINT_5(rval, SYS_zone, 0, cmd, zid, attr,
                    buf, bufsize);

                len = strlen(aval) + 1;
                if (len > bufsize)
                        return (ENAMETOOLONG);

                if (buf != NULL) {
                        if (len == 1) {
                                if (brand_uucopy("\0", buf, 1) != 0)
                                        return (EFAULT);
                        } else {
                                if (brand_uucopystr(aval, buf, len) != 0)
                                        return (EFAULT);

                                /*
                                 * Assure NULL termination of "buf" as
                                 * brand_uucopystr() does NOT.
                                 */
                                if (brand_uucopy("\0", buf + (len - 1), 1) != 0)
                                        return (EFAULT);
                        }
                }

                rval->sys_rval1 = len;
                rval->sys_rval2 = 0;
                return (0);

        default:
                break;
        }

passthru:
        return (__systemcall(rval, SYS_zone + 1024, cmd, arg1, arg2, arg3,
            arg4));
}

/*ARGSUSED*/
int
brand_init(int argc, char *argv[], char *envp[])
{
        sysret_t                rval;
        ulong_t                 ldentry;
        int                     err;
        char                    *bname;

        brand_pre_init();

        /*
         * Cache the pid of the zone's init process and determine if
         * we're init(1m) for the zone.  Remember: we might be init
         * now, but as soon as we fork(2) we won't be.
```

```
1844          */
1845         (void) get_initpid_info();

1847         /* get the current zoneid */
1848         err = __systemcall(&rval, SYS_zone, ZONE_LOOKUP, NULL);
1849         brand_assert(err == 0);
1850         zoneid = (zoneid_t)rval.sys_rval1;

1852         /* Get the zone's emulation bitmap. */
1853         if ((err = __systemcall(&rval, SYS_zone, ZONE_GETATTR, zoneid,
1854             S10_EMUL_BITMAP, emul_bitmap, sizeof (emul_bitmap))) != 0) {
1855                 brand_abort(err, "The zone's patch level is unsupported");
1856                 /*NOTREACHED*/
1857         }

1859         bname = basename(argv[0]);

1861         /*
1862          * In general we want the S10 commands that are zone-aware to continue
1863          * to behave as they normally do within a zone.  Since these commands
1864          * are zone-aware, they should continue to "do the right thing".
1865          * However, some zone-aware commands aren't going to work the way
1866          * we expect them to inside the branded zone.  In particular, the pkg
1867          * and patch commands will not properly manage all pkgs/patches
1868          * unless the commands think they are running in the global zone.  For
1869          * these commands we want to emulate the global zone.
1870          *
1871          * We don't do any emulation for pkgcond since it is typically used
1872          * in pkg/patch postinstall scripts and we want those scripts to do
1873          * the right thing inside a zone.
1874          *
1875          * One issue is the handling of hollow pkgs.  Since the pkgs are
1876          * hollow, they won't use pkgcond in their postinstall scripts.  These
1877          * pkgs typically are installing drivers so we handle that by
1878          * replacing add_drv and rem_drv in the s10_boot script.
1879          */
1880         if (strcmp("pkgadd", bname) == 0 || strcmp("pkgrm", bname) == 0 ||
1881             strcmp("patchadd", bname) == 0 || strcmp("patchrm", bname) == 0)
1882                 emul_global_zone = B_TRUE;

1884         ldentry = brand_post_init(S10_VERSION, argc, argv, envp);

1886         brand_runexe(argv, ldentry);
1887         /*NOTREACHED*/
1888         brand_abort(0, "brand_runexe() returned");
1889         return (-1);
1890 }

1892 /*
1893  * This table must have at least NSYSCALL entries in it.
1894  *
1895  * The second parameter of each entry in the brand_sysent_table
1896  * contains the number of parameters and flags that describe the
1897  * syscall return value encoding.  See the block comments at the
1898  * top of this file for more information about the syscall return
1899  * value flags and when they should be used.
1900  */
1901 brand_sysent_table_t brand_sysent_table[] = {
1902 #if defined(__sparc) && !defined(__sparcv9)
1903         EMULATE(brand_indir, 9 | RV_64RVAL),    /*  0 */
1904 #else
1905         NOSYS,                                  /*  0 */
1906 #endif
1907         NOSYS,                                  /*   1 */
1908         EMULATE(s10_forkall, 0 | RV_32RVAL2),   /*   2 */
1909         NOSYS,                                  /*   3 */
```



```
1910         NOSYS,                                  /*   4 */
1911         EMULATE(s10_open, 3 | RV_DEFAULT),      /*   5 */
1912         NOSYS,                                  /*   6 */
1913         EMULATE(s10_wait, 0 | RV_32RVAL2),      /*   7 */
1914         EMULATE(s10_creat, 2 | RV_DEFAULT),     /*   8 */
1915         EMULATE(s10_link, 2 | RV_DEFAULT),      /*   9 */
1916         EMULATE(s10_unlink, 1 | RV_DEFAULT),    /*  10 */
1917         EMULATE(s10_exec, 2 | RV_DEFAULT),      /*  11 */
1918         NOSYS,                                  /*  12 */
1919         NOSYS,                                  /*  13 */
1920         EMULATE(s10_mknod, 3 | RV_DEFAULT),     /*  14 */
1921         EMULATE(s10_chmod, 2 | RV_DEFAULT),     /*  15 */
1922         EMULATE(s10_chown, 3 | RV_DEFAULT),     /*  16 */
1923         NOSYS,                                  /*  17 */
1924         EMULATE(s10_stat, 2 | RV_DEFAULT),      /*  18 */
1925         NOSYS,                                  /*  19 */
1926         NOSYS,                                  /*  20 */
1927         NOSYS,                                  /*  21 */
1928         EMULATE(s10_umount, 1 | RV_DEFAULT),    /*  22 */
1929         NOSYS,                                  /*  23 */
1930         NOSYS,                                  /*  24 */
1931         NOSYS,                                  /*  25 */
1932         NOSYS,                                  /*  26 */
1933         NOSYS,                                  /*  27 */
1934         EMULATE(s10_fstat, 2 | RV_DEFAULT),     /*  28 */
1935         NOSYS,                                  /*  29 */
1936         EMULATE(s10_utime, 2 | RV_DEFAULT),     /*  30 */
1937         NOSYS,                                  /*  31 */
1938         NOSYS,                                  /*  32 */
1939         EMULATE(s10_access, 2 | RV_DEFAULT),    /*  33 */
1940         NOSYS,                                  /*  34 */
1941         NOSYS,                                  /*  35 */
1942         NOSYS,                                  /*  36 */
1943         EMULATE(s10_kill, 2 | RV_DEFAULT),      /*  37 */
1944         NOSYS,                                  /*  38 */
1945         NOSYS,                                  /*  39 */
1946         NOSYS,                                  /*  40 */
1947         EMULATE(s10_dup, 1 | RV_DEFAULT),       /*  41 */
1948         EMULATE(s10_pipe, 0 | RV_32RVAL2),      /*  42 */
1949         NOSYS,                                  /*  43 */
1950         NOSYS,                                  /*  44 */
1951         NOSYS,                                  /*  45 */
1952         NOSYS,                                  /*  46 */
1953         NOSYS,                                  /*  47 */
1954         NOSYS,                                  /*  48 */
1955         NOSYS,                                  /*  49 */
1956         NOSYS,                                  /*  50 */
1957         NOSYS,                                  /*  51 */
1958         NOSYS,                                  /*  52 */
1959         NOSYS,                                  /*  53 */
1960         EMULATE(s10_ioctl, 3 | RV_DEFAULT),     /*  54 */
1961         NOSYS,                                  /*  55 */
1962         NOSYS,                                  /*  56 */
1963         NOSYS,                                  /*  57 */
1964         NOSYS,                                  /*  58 */
1965         EMULATE(s10_execve, 3 | RV_DEFAULT),    /*  59 */
1966         NOSYS,                                  /*  60 */
1967         NOSYS,                                  /*  61 */
1968         NOSYS,                                  /*  62 */
1969         NOSYS,                                  /*  63 */
1970         NOSYS,                                  /*  64 */
1971         NOSYS,                                  /*  65 */
1972         NOSYS,                                  /*  66 */
1973         NOSYS,                                  /*  67 */
1974         NOSYS,                                  /*  68 */
1975         NOSYS,                                  /*  69 */
```

```
1976         NOSYS,                                     /*  70 */
1977         EMULATE(s10_acctctl, 3 | RV_DEFAULT),      /*  71 */
1978         NOSYS,                                     /*  72 */
1979         NOSYS,                                     /*  73 */
1980         NOSYS,                                     /*  74 */
1981         EMULATE(s10_issetugid, 0 | RV_DEFAULT),    /*  75 */
1982         EMULATE(s10_fsat, 6 | RV_DEFAULT),         /*  76 */
1983         NOSYS,                                     /*  77 */
1984         NOSYS,                                     /*  78 */
1985         EMULATE(s10_rmdir, 1 | RV_DEFAULT),        /*  79 */
1986         EMULATE(s10_mkdir, 2 | RV_DEFAULT),        /*  80 */
1987         EMULATE(s10_getdents, 3 | RV_DEFAULT),     /*  81 */
1988         NOSYS,                                     /*  82 */
1989         NOSYS,                                     /*  83 */
1990         NOSYS,                                     /*  84 */
1991         NOSYS,                                     /*  85 */
1992         NOSYS,                                     /*  86 */
1993         EMULATE(s10_poll, 3 | RV_DEFAULT),         /*  87 */
1994         EMULATE(s10_lstat, 2 | RV_DEFAULT),        /*  88 */
1995         EMULATE(s10_symlink, 2 | RV_DEFAULT),      /*  89 */
1996         EMULATE(s10_readlink, 3 | RV_DEFAULT),     /*  90 */
1997         NOSYS,                                     /*  91 */
1998         NOSYS,                                     /*  92 */
1999         EMULATE(s10_fchmod, 2 | RV_DEFAULT),       /*  93 */
2000         EMULATE(s10_fchown, 3 | RV_DEFAULT),       /*  94 */
2001         EMULATE(s10_sigprocmask, 3 | RV_DEFAULT), /*  95 */
2002         EMULATE(s10_sigsuspend, 1 | RV_DEFAULT), /*  96 */
2003         NOSYS,                                     /*  97 */
2004         EMULATE(s10_sigaction, 3 | RV_DEFAULT), /*  98 */
2005         EMULATE(s10_sigpending, 2 | RV_DEFAULT), /*  99 */
2006         NOSYS,                                     /* 100 */
2007         NOSYS,                                     /* 101 */
2008         NOSYS,                                     /* 102 */
2009         NOSYS,                                     /* 103 */
2010         NOSYS,                                     /* 104 */
2011         NOSYS,                                     /* 105 */
2012         NOSYS,                                     /* 106 */
2013         EMULATE(s10_waitid, 4 | RV_DEFAULT),       /* 107 */
2014         EMULATE(s10_sigsendsys, 2 | RV_DEFAULT), /* 108 */
2015         NOSYS,                                     /* 109 */
2016         NOSYS,                                     /* 110 */
2017         NOSYS,                                     /* 111 */
2018         NOSYS,                                     /* 112 */
2019         NOSYS,                                     /* 113 */
2020         NOSYS,                                     /* 114 */
2021         NOSYS,                                     /* 115 */
2022         NOSYS,                                     /* 116 */
2023         NOSYS,                                     /* 117 */
2024         NOSYS,                                     /* 118 */
2025         NOSYS,                                     /* 119 */
2026         NOSYS,                                     /* 120 */
2027         NOSYS,                                     /* 121 */
2028         NOSYS,                                     /* 122 */
2029 #if defined(__x86)
2030         EMULATE(s10_xstat, 3 | RV_DEFAULT),        /* 123 */
2031         EMULATE(s10_lxstat, 3 | RV_DEFAULT),       /* 124 */
2032         EMULATE(s10_fxstat, 3 | RV_DEFAULT),       /* 125 */
2033         EMULATE(s10_xmknod, 4 | RV_DEFAULT),       /* 126 */
2034 #else
2035         NOSYS,                                     /* 123 */
2036         NOSYS,                                     /* 124 */
2037         NOSYS,                                     /* 125 */
2038         NOSYS,                                     /* 126 */
2039 #endif
2040         NOSYS,                                     /* 127 */
2041         NOSYS,                                     /* 128 */
```



```
2042         NOSYS,                                     /* 129 */
2043         EMULATE(s10_lchown, 3 | RV_DEFAULT),       /* 130 */
2044         NOSYS,                                     /* 131 */
2045         NOSYS,                                     /* 132 */
2046         NOSYS,                                     /* 133 */
2047         EMULATE(s10_rename, 2 | RV_DEFAULT),       /* 134 */
2048         EMULATE(s10_uname, 1 | RV_DEFAULT),        /* 135 */
2049         NOSYS,                                     /* 136 */
2050         EMULATE(s10_sysconfig, 1 | RV_DEFAULT), /* 137 */
2051         NOSYS,                                     /* 138 */
2052         EMULATE(s10_sysinfo, 3 | RV_DEFAULT),      /* 139 */
2053         NOSYS,                                     /* 140 */
2054         NOSYS,                                     /* 141 */
2055         NOSYS,                                     /* 142 */
2056         EMULATE(s10_fork1, 0 | RV_32RVAL2),        /* 143 */
2057         EMULATE(s10_sigtimedwait, 3 | RV_DEFAULT), /* 144 */
2058         NOSYS,                                     /* 145 */
2059         NOSYS,                                     /* 146 */
2060         EMULATE(s10_lwp_sema_wait, 1 | RV_DEFAULT), /* 147 */
2061         NOSYS,                                     /* 148 */
2062         NOSYS,                                     /* 149 */
2063         NOSYS,                                     /* 150 */
2064         NOSYS,                                     /* 151 */
2065         NOSYS,                                     /* 152 */
2066         NOSYS,                                     /* 153 */
2067         EMULATE(s10_utimes, 2 | RV_DEFAULT),       /* 154 */
2068         NOSYS,                                     /* 155 */
2069         NOSYS,                                     /* 156 */
2070         NOSYS,                                     /* 157 */
2071         NOSYS,                                     /* 158 */
2072         EMULATE(s10_lwp_create, 3 | RV_DEFAULT), /* 159 */
2073         NOSYS,                                     /* 160 */
2074         NOSYS,                                     /* 161 */
2075         NOSYS,                                     /* 162 */
2076         EMULATE(s10_lwp_kill, 2 | RV_DEFAULT),     /* 163 */
2077         NOSYS,                                     /* 164 */
2078         EMULATE(s10_lwp_sigmask, 3 | RV_32RVAL2), /* 165 */
2079 #if defined(__x86)
2080         EMULATE(s10_lwp_private, 3 | RV_DEFAULT), /* 166 */
2081 #else
2082         NOSYS,                                     /* 166 */
2083 #endif
2084         NOSYS,                                     /* 167 */
2085         NOSYS,                                     /* 168 */
2086         EMULATE(s10_lwp_mutex_lock, 1 | RV_DEFAULT), /* 169 */
2087         NOSYS,                                     /* 170 */
2088         NOSYS,                                     /* 171 */
2089         NOSYS,                                     /* 172 */
2090         NOSYS,                                     /* 173 */
2091         EMULATE(s10_pwrite, 4 | RV_DEFAULT),       /* 174 */
2092         NOSYS,                                     /* 175 */
2093         NOSYS,                                     /* 176 */
2094         NOSYS,                                     /* 177 */
2095         NOSYS,                                     /* 178 */
2096         NOSYS,                                     /* 179 */
2097         NOSYS,                                     /* 180 */
2098         NOSYS,                                     /* 181 */
2099         NOSYS,                                     /* 182 */
2100         NOSYS,                                     /* 183 */
2101         NOSYS,                                     /* 184 */
2102         EMULATE(s10_acl, 4 | RV_DEFAULT),          /* 185 */
2103         EMULATE(s10_auditsys, 4 | RV_64RVAL),      /* 186 */
2104         NOSYS,                                     /* 187 */
2105         NOSYS,                                     /* 188 */
2106         NOSYS,                                     /* 189 */
2107         EMULATE(s10_sigqueue, 4 | RV_DEFAULT),     /* 190 */
```

```
2108         NOSYS,                                  /* 191 */
2109         NOSYS,                                  /* 192 */
2110         NOSYS,                                  /* 193 */
2111         NOSYS,                                  /* 194 */
2112         NOSYS,                                  /* 195 */
2113         NOSYS,                                  /* 196 */
2114         NOSYS,                                  /* 197 */
2115         NOSYS,                                  /* 198 */
2116         NOSYS,                                  /* 199 */
2117         EMULATE(s10_facl, 4 | RV_DEFAULT),      /* 200 */
2118         NOSYS,                                  /* 201 */
2119         NOSYS,                                  /* 202 */
2120         NOSYS,                                  /* 203 */
2121         NOSYS,                                  /* 204 */
2122         EMULATE(s10_signotify, 3 | RV_DEFAULT), /* 205 */
2123         NOSYS,                                  /* 206 */
2124         NOSYS,                                  /* 207 */
2125         NOSYS,                                  /* 208 */
2126         NOSYS,                                  /* 209 */
2127         EMULATE(s10_lwp_mutex_timedlock, 2 | RV_DEFAULT), /* 210 */
2128         NOSYS,                                  /* 211 */
2129         NOSYS,                                  /* 212 */
2130 #if defined(_LP64)
2131         NOSYS,                                  /* 213 */
2132 #else
2133         EMULATE(s10_getdents64, 3 | RV_DEFAULT), /* 213 */
2134 #endif
2135         NOSYS,                                  /* 214 */
2136 #if defined(_LP64)
2137         NOSYS,                                  /* 215 */
2138         NOSYS,                                  /* 216 */
2139         NOSYS,                                  /* 217 */
2140 #else
2141         EMULATE(s10_stat64, 2 | RV_DEFAULT),    /* 215 */
2142         EMULATE(s10_lstat64, 2 | RV_DEFAULT),   /* 216 */
2143         EMULATE(s10_fstat64, 2 | RV_DEFAULT),   /* 217 */
2144 #endif
2145         NOSYS,                                  /* 218 */
2146         NOSYS,                                  /* 219 */
2147         NOSYS,                                  /* 220 */
2148         NOSYS,                                  /* 221 */
2149         NOSYS,                                  /* 222 */
2150 #if defined(_LP64)
2151         NOSYS,                                  /* 223 */
2152         NOSYS,                                  /* 224 */
2153         NOSYS,                                  /* 225 */
2154 #else
2155         EMULATE(s10_pwrite64, 5 | RV_DEFAULT),  /* 223 */
2156         EMULATE(s10_creat64, 2 | RV_DEFAULT),   /* 224 */
2157         EMULATE(s10_open64, 3 | RV_DEFAULT),    /* 225 */
2158 #endif
2159         NOSYS,                                  /* 226 */
2160         EMULATE(s10_zone, 5 | RV_DEFAULT),      /* 227 */
2161         NOSYS,                                  /* 228 */
2162         NOSYS,                                  /* 229 */
2163         EMULATE(s10_so_socket, 5 | RV_DEFAULT), /* 230 */
2164         NOSYS,                                  /* 231 */
2165         NOSYS,                                  /* 232 */
2166         NOSYS,                                  /* 233 */
2167         EMULATE(s10_accept, 4 | RV_DEFAULT),    /* 234 */
2168         NOSYS,                                  /* 235 */
2169         NOSYS,                                  /* 236 */
2170         NOSYS,                                  /* 237 */
2171         NOSYS,                                  /* 238 */
2172         NOSYS,                                  /* 239 */
2173         NOSYS,                                  /* 240 */
```



```
2174         NOSYS,                                  /* 241 */
2175         NOSYS,                                  /* 242 */
2176         NOSYS,                                  /* 243 */
2177         NOSYS,                                  /* 244 */
2178         NOSYS,                                  /* 245 */
2179         NOSYS,                                  /* 246 */
2180         NOSYS,                                  /* 247 */
2181         NOSYS,                                  /* 248 */
2182         NOSYS,                                  /* 249 */
2183         NOSYS,                                  /* 250 */
2184         EMULATE(s10_lwp_mutex_trylock, 1 | RV_DEFAULT), /* 251 */
2185         NOSYS,                                  /* 252 */
2186         NOSYS,                                  /* 253 */
2187         NOSYS,                                  /* 254 */
2188         NOSYS                                   /* 255 */
2189 };
```

```
**********************************************************
    2796 Tue Aug 18 16:13:44 2015
new/usr/src/uts/common/sys/mkdev.h
6136 sysmacros.h unnecessarily polutes the namespace
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2014 Garrett D'Amore <garrett@damore.org>
 *
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

#ifndef _SYS_MKDEV_H
#define _SYS_MKDEV_H

#include <sys/types.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * SVR3/Pre-EFT device number constants.
 */
#define ONBITSMAJOR     7       /* # of SVR3 major device bits */
#define ONBITSMINOR     8       /* # of SVR3 minor device bits */
#define OMAXMAJ         0x7f    /* SVR3 max major value */
#define OMAXMIN         0xff    /* SVR3 max major value */

/*
 * 32-bit Solaris device major/minor sizes.
 */
#define NBITSMAJOR32    14
#define NBITSMINOR32    18
#define MAXMAJ32        0x3fffu         /* SVR4 max major value */
#define MAXMIN32        0x3ffffu        /* SVR4 max minor value */

#define NBITSMAJOR64    32      /* # of major device bits in 64-bit Solaris */
#define NBITSMINOR64    32      /* # of minor device bits in 64-bit Solaris */

#ifdef _LP64

#define MAXMAJ64        0xffffffffu     /* max major value */
```



```
#define MAXMIN64        0xffffffffu     /* max minor value */

#define NBITSMAJOR      NBITSMAJOR64
#define NBITSMINOR      NBITSMINOR64
#define MAXMAJ          MAXMAJ64
#define MAXMIN          MAXMIN64

#else /* !_LP64 */

#define NBITSMAJOR      NBITSMAJOR32
#define NBITSMINOR      NBITSMINOR32
#define MAXMAJ          MAXMAJ32
#define MAXMIN          MAXMIN32

#endif /* !_LP64 */

#if !defined(_KERNEL)

/*
 * Undefine sysmacros.h device macros.
 */
#undef makedev
#undef major
#undef minor

extern dev_t makedev(const major_t, const minor_t);
extern major_t major(const dev_t);
extern minor_t minor(const dev_t);
extern dev_t __makedev(const int, const major_t, const minor_t);
extern major_t __major(const int, const dev_t);
extern minor_t __minor(const int, const dev_t);

#define OLDDEV 0        /* old device format */
#define NEWDEV 1        /* new device format */

#define makedev(maj, min)       (__makedev(NEWDEV, maj, min))
#define major(dev)              (__major(NEWDEV, dev))
#define minor(dev)              (__minor(NEWDEV, dev))

#endif  /* !defined(_KERNEL) */

#ifdef  __cplusplus
}
```

_____unchanged_portion_omitted_

```
**********************************************************
   11077 Tue Aug 18 16:13:45 2015
new/usr/src/uts/common/sys/sysmacros.h
6136 sysmacros.h unnecessarily polutes the namespace
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */


/*
 * Copyright 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 *
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */

#ifndef _SYS_SYSMACROS_H
#define _SYS_SYSMACROS_H

#include <sys/param.h>

#ifdef  __cplusplus
extern "C" {
#endif

/*
 * Some macros for units conversion
 */
/*
 * Disk blocks (sectors) and bytes.
 */
#define dtob(DD)        ((DD) << DEV_BSHIFT)
#define btod(BB)        (((BB) + DEV_BSIZE - 1) >> DEV_BSHIFT)
#define btodt(BB)       ((BB) >> DEV_BSHIFT)
#define lbtod(BB)       (((offset_t)(BB) + DEV_BSIZE - 1) >> DEV_BSHIFT)

/* common macros */
#ifndef MIN
#define MIN(a, b)       ((a) < (b) ? (a) : (b))
#endif
#ifndef MAX
#define MAX(a, b)       ((a) < (b) ? (b) : (a))
#endif
#ifndef ABS
#define ABS(a)          ((a) < 0 ? -(a) : (a))
#endif
```



```
#ifndef SIGNOF
#define SIGNOF(a)       ((a) < 0 ? -1 : (a) > 0)
#endif

#ifdef _KERNEL

/*
 * Convert a single byte to/from binary-coded decimal (BCD).
 */
extern unsigned char byte_to_bcd[256];
extern unsigned char bcd_to_byte[256];

#define BYTE_TO_BCD(x)  byte_to_bcd[(x) & 0xff]
#define BCD_TO_BYTE(x)  bcd_to_byte[(x) & 0xff]

#endif  /* _KERNEL */

/*
 * WARNING: The device number macros defined here should not be used by device
 * drivers or user software. Device drivers should use the device functions
 * defined in the DDI/DKI interface (see also ddi.h). Application software
 * should make use of the library routines available in makedev(3). A set of
 * new device macros are provided to operate on the expanded device number
 * format supported in SVR4. Macro versions of the DDI device functions are
 * provided for use by kernel proper routines only.
 * provided for use by kernel proper routines only. Macro routines bmajor(),
 * major(), minor(), emajor(), eminor(), and makedev() will be removed or
 * their definitions changed at the next major release following SVR4.
 */

#define O_BITSMAJOR     7       /* # of SVR3 major device bits */
#define O_BITSMINOR     8       /* # of SVR3 minor device bits */
#define O_MAXMAJ        0x7f    /* SVR3 max major value */
#define O_MAXMIN        0xff    /* SVR3 max minor value */


#define L_BITSMAJOR32   14      /* # of SVR4 major device bits */
#define L_BITSMINOR32   18      /* # of SVR4 minor device bits */
#define L_MAXMAJ32      0x3fff  /* SVR4 max major value */
#define L_MAXMIN32      0x3ffff /* MAX minor for 3b2 software drivers. */
                                /* For 3b2 hardware devices the minor is */
                                /* restricted to 256 (0-255) */

#ifdef _LP64
#define L_BITSMAJOR     32      /* # of major device bits in 64-bit Solaris */
#define L_BITSMINOR     32      /* # of minor device bits in 64-bit Solaris */
#define L_MAXMAJ        0xfffffffful    /* max major value */
#define L_MAXMIN        0xfffffffful    /* max minor value */
#else
#define L_BITSMAJOR     L_BITSMAJOR32
#define L_BITSMINOR     L_BITSMINOR32
#define L_MAXMAJ        L_MAXMAJ32
#define L_MAXMIN        L_MAXMIN32
#endif

#ifdef _KERNEL

/* major part of a device internal to the kernel */

#define major(x)        (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)
#define bmajor(x)       (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)

/* get internal major part of expanded device number */

#define getmajor(x)     (major_t)((((dev_t)(x)) >> L_BITSMINOR) & L_MAXMAJ)
```

```
/* minor part of a device internal to the kernel */

#define minor(x)        (minor_t)((x) & O_MAXMIN)

/* get internal minor part of expanded device number */

#define getminor(x)     (minor_t)((x) & L_MAXMIN)

#else   /* _KERNEL */

/* major part of a device external from the kernel (same as emajor below) */

#define major(x)        (major_t)((((unsigned)(x)) >> O_BITSMINOR) & O_MAXMAJ)

/* minor part of a device external from the kernel  (same as eminor below) */

#define minor(x)        (minor_t)((x) & O_MAXMIN)

#endif  /* _KERNEL */

/* create old device number */

#define makedev(x, y) (unsigned short)(((x) << O_BITSMINOR) | ((y) & O_MAXMIN))

/* make an new device number */

#define makedevice(x, y) (dev_t)(((dev_t)(x) << L_BITSMINOR) | ((y) & L_MAXMIN))


/*
 * emajor() allows kernel/driver code to print external major numbers
 * eminor() allows kernel/driver code to print external minor numbers
 */

#define emajor(x) \
        (major_t)(((unsigned int)(x) >> O_BITSMINOR) > O_MAXMAJ) ? \
            NODEV : (((unsigned int)(x) >> O_BITSMINOR) & O_MAXMAJ)

#define eminor(x) \
        (minor_t)((x) & O_MAXMIN)

/*
 * get external major and minor device
 * components from expanded device number
 */
#define getemajor(x)    (major_t)((((dev_t)(x) >> L_BITSMINOR) > L_MAXMAJ) ? \
                            NODEV : (((dev_t)(x) >> L_BITSMINOR) & L_MAXMAJ))
#define geteminor(x)    (minor_t)((x) & L_MAXMIN)

/*
 * These are versions of the kernel routines for compressing and
 * expanding long device numbers that don't return errors.
 */
#if (L_BITSMAJOR32 == L_BITSMAJOR) && (L_BITSMINOR32 == L_BITSMINOR)

#define DEVCMPL(x)      (x)
#define DEVEXPL(x)      (x)

#else

#define DEVCMPL(x)      \
        (dev32_t)((((x) >> L_BITSMINOR) > L_MAXMAJ32 || \
            ((x) & L_MAXMIN) > L_MAXMIN32) ? NODEV32 : \
            ((((x) >> L_BITSMINOR) << L_BITSMINOR32) | ((x) & L_MAXMIN32)))

#define DEVEXPL(x)      \
```



```
        (((x) == NODEV32) ? NODEV : \
        makedevice(((x) >> L_BITSMINOR32) & L_MAXMAJ32, (x) & L_MAXMIN32))

#endif /* L_BITSMAJOR32 ... */

/* convert to old (SVR3.2) dev format */

#define cmpdev(x) \
        (o_dev_t)((((x) >> L_BITSMINOR) > O_MAXMAJ || \
            ((x) & L_MAXMIN) > O_MAXMIN) ? NODEV : \
            ((((x) >> L_BITSMINOR) << O_BITSMINOR) | ((x) & O_MAXMIN)))

/* convert to new (SVR4) dev format */

#define expdev(x) \
        (dev_t)(((dev_t)(((x) >> O_BITSMINOR) & O_MAXMAJ) << L_BITSMINOR) | \
            ((x) & O_MAXMIN))

/*
 * Macro for checking power of 2 address alignment.
 */
#define IS_P2ALIGNED(v, a) ((((uintptr_t)(v)) & ((uintptr_t)(a) - 1)) == 0)

/*
 * Macros for counting and rounding.
 */
#define howmany(x, y)   (((x)+((y)-1))/(y))
#define roundup(x, y)   ((((x)+((y)-1))/(y))*(y))

/*
 * Macro to determine if value is a power of 2
 */
#define ISP2(x)         (((x) & ((x) - 1)) == 0)

/*
 * Macros for various sorts of alignment and rounding.  The "align" must
 * be a power of 2.  Often times it is a block, sector, or page.
 */

/*
 * return x rounded down to an align boundary
 * eg, P2ALIGN(1200, 1024) == 1024 (1*align)
 * eg, P2ALIGN(1024, 1024) == 1024 (1*align)
 * eg, P2ALIGN(0x1234, 0x100) == 0x1200 (0x12*align)
 * eg, P2ALIGN(0x5600, 0x100) == 0x5600 (0x56*align)
 */
#define P2ALIGN(x, align)               ((x) & -(align))

/*
 * return x % (mod) align
 * eg, P2PHASE(0x1234, 0x100) == 0x34 (x-0x12*align)
 * eg, P2PHASE(0x5600, 0x100) == 0x00 (x-0x56*align)
 */
#define P2PHASE(x, align)               ((x) & ((align) - 1))

/*
 * return how much space is left in this block (but if it's perfectly
 * aligned, return 0).
 * eg, P2NPHASE(0x1234, 0x100) == 0xcc (0x13*align-x)
 * eg, P2NPHASE(0x5600, 0x100) == 0x00 (0x56*align-x)
 */
#define P2NPHASE(x, align)              (-(x) & ((align) - 1))

/*
 * return x rounded up to an align boundary
 * eg, P2ROUNDUP(0x1234, 0x100) == 0x1300 (0x13*align)
```

```
 221  * eg, P2ROUNDUP(0x5600, 0x100) == 0x5600 (0x56*align)
 222  */
 223 #define P2ROUNDUP(x, align)             (-(-(x) & -(align)))

 225 /*
 226  * return the ending address of the block that x is in
 227  * eg, P2END(0x1234, 0x100) == 0x12ff (0x13*align - 1)
 228  * eg, P2END(0x5600, 0x100) == 0x56ff (0x57*align - 1)
 229  */
 230 #define P2END(x, align)                 (-(~(x) & -(align)))

 232 /*
 233  * return x rounded up to the next phase (offset) within align.
 234  * phase should be < align.
 235  * eg, P2PHASEUP(0x1234, 0x100, 0x10) == 0x1310 (0x13*align + phase)
 236  * eg, P2PHASEUP(0x5600, 0x100, 0x10) == 0x5610 (0x56*align + phase)
 237  */
 238 #define P2PHASEUP(x, align, phase)      ((phase) - (((phase) - (x)) & -(align)))

 240 /*
 241  * return TRUE if adding len to off would cause it to cross an align
 242  * boundary.
 243  * eg, P2BOUNDARY(0x1234, 0xe0, 0x100) == TRUE (0x1234 + 0xe0 == 0x1314)
 244  * eg, P2BOUNDARY(0x1234, 0x50, 0x100) == FALSE (0x1234 + 0x50 == 0x1284)
 245  */
 246 #define P2BOUNDARY(off, len, align) \
 247         (((off) ^ ((off) + (len) - 1)) > (align) - 1)

 249 /*
 250  * Return TRUE if they have the same highest bit set.
 251  * eg, P2SAMEHIGHBIT(0x1234, 0x1001) == TRUE (the high bit is 0x1000)
 252  * eg, P2SAMEHIGHBIT(0x1234, 0x3010) == FALSE (high bit of 0x3010 is 0x2000)
 253  */
 254 #define P2SAMEHIGHBIT(x, y)             (((x) ^ (y)) < ((x) & (y)))

 256 /*
 257  * Typed version of the P2* macros.  These macros should be used to ensure
 258  * that the result is correctly calculated based on the data type of (x),
 259  * which is passed in as the last argument, regardless of the data
 260  * type of the alignment.  For example, if (x) is of type uint64_t,
 261  * and we want to round it up to a page boundary using "PAGESIZE" as
 262  * the alignment, we can do either
 263  *      P2ROUNDUP(x, (uint64_t)PAGESIZE)
 264  * or
 265  *      P2ROUNDUP_TYPED(x, PAGESIZE, uint64_t)
 266  */
 267 #define P2ALIGN_TYPED(x, align, type)   \
 268         ((type)(x) & -(type)(align))
 269 #define P2PHASE_TYPED(x, align, type)   \
 270         ((type)(x) & ((type)(align) - 1))
 271 #define P2NPHASE_TYPED(x, align, type)  \
 272         (-(type)(x) & ((type)(align) - 1))
 273 #define P2ROUNDUP_TYPED(x, align, type) \
 274         (-(-(type)(x) & -(type)(align)))
 275 #define P2END_TYPED(x, align, type)     \
 276         (-(~(type)(x) & -(type)(align)))
 277 #define P2PHASEUP_TYPED(x, align, phase, type)  \
 278         ((type)(phase) - (((type)(phase) - (type)(x)) & -(type)(align)))
 279 #define P2CROSS_TYPED(x, y, align, type)        \
 280         (((type)(x) ^ (type)(y)) > (type)(align) - 1)
 281 #define P2SAMEHIGHBIT_TYPED(x, y, type) \
 282         (((type)(x) ^ (type)(y)) < ((type)(x) & (type)(y)))

 284 /*
 285  * Macros to atomically increment/decrement a variable.  mutex and var
 286  * must be pointers.
```



```
 287  */
 288 #define INCR_COUNT(var, mutex) mutex_enter(mutex), (*(var))++, mutex_exit(mutex)
 289 #define DECR_COUNT(var, mutex) mutex_enter(mutex), (*(var))--, mutex_exit(mutex)

 291 /*
 292  * Macros to declare bitfields - the order in the parameter list is
 293  * Low to High - that is, declare bit 0 first.  We only support 8-bit bitfields
 294  * because if a field crosses a byte boundary it's not likely to be meaningful
 295  * without reassembly in its nonnative endianness.
 296  */
 297 #if defined(_BIT_FIELDS_LTOH)
 298 #define DECL_BITFIELD2(_a, _b)                          \
 299         uint8_t _a, _b
 300 #define DECL_BITFIELD3(_a, _b, _c)                      \
 301         uint8_t _a, _b, _c
 302 #define DECL_BITFIELD4(_a, _b, _c, _d)                  \
 303         uint8_t _a, _b, _c, _d
 304 #define DECL_BITFIELD5(_a, _b, _c, _d, _e)              \
 305         uint8_t _a, _b, _c, _d, _e
 306 #define DECL_BITFIELD6(_a, _b, _c, _d, _e, _f)          \
 307         uint8_t _a, _b, _c, _d, _e, _f
 308 #define DECL_BITFIELD7(_a, _b, _c, _d, _e, _f, _g)      \
 309         uint8_t _a, _b, _c, _d, _e, _f, _g
 310 #define DECL_BITFIELD8(_a, _b, _c, _d, _e, _f, _g, _h)  \
 311         uint8_t _a, _b, _c, _d, _e, _f, _g, _h
 312 #elif defined(_BIT_FIELDS_HTOL)
 313 #define DECL_BITFIELD2(_a, _b)                          \
 314         uint8_t _b, _a
 315 #define DECL_BITFIELD3(_a, _b, _c)                      \
 316         uint8_t _c, _b, _a
 317 #define DECL_BITFIELD4(_a, _b, _c, _d)                  \
 318         uint8_t _d, _c, _b, _a
 319 #define DECL_BITFIELD5(_a, _b, _c, _d, _e)              \
 320         uint8_t _e, _d, _c, _b, _a
 321 #define DECL_BITFIELD6(_a, _b, _c, _d, _e, _f)          \
 322         uint8_t _f, _e, _d, _c, _b, _a
 323 #define DECL_BITFIELD7(_a, _b, _c, _d, _e, _f, _g)      \
 324         uint8_t _g, _f, _e, _d, _c, _b, _a
 325 #define DECL_BITFIELD8(_a, _b, _c, _d, _e, _f, _g, _h)  \
 326         uint8_t _h, _g, _f, _e, _d, _c, _b, _a
 327 #else
 328 #error  One of _BIT_FIELDS_LTOH or _BIT_FIELDS_HTOL must be defined
 329 #endif  /* _BIT_FIELDS_LTOH */

 331 /* avoid any possibility of clashing with <stddef.h> version */
 332 #if (defined(_KERNEL) || defined(_FAKE_KERNEL)) && !defined(_KMEMUSER)

 334 #if !defined(offsetof)
 335 #define offsetof(s, m)  ((size_t)(&(((s *)0)->m)))
 336 #endif /* !offsetof */

 338 #define container_of(m, s, name)                        \
 339         (void *)((uintptr_t)(m) - (uintptr_t)offsetof(s, name))

 341 #define ARRAY_SIZE(x)   (sizeof (x) / sizeof (x[0]))
 342 #endif /* _KERNEL, !_KMEMUSER */

 344 #ifdef  __cplusplus
 345 }
```

_____***unchanged_portion_omitted_***