```
**********************************************************
   38907 Tue Jan  7 20:44:10 2014
new/usr/src/tools/scripts/wsdiff.py
4445 wsdiff results file timestamp format is strange
**********************************************************
```

```
#!/usr/bin/python2.6
#
# 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) 2010, Oracle and/or its affiliates. All rights reserved.
#

#
# wsdiff(1) is a tool that can be used to determine which compiled objects
# have changed as a result of a given source change. Developers backporting
# new features, RFEs and bug fixes need to be able to identify the set of
# patch deliverables necessary for feature/fix realization on a patched system.
#
# The tool works by comparing objects in two trees/proto areas (one build with,
# and without the source changes.
#
# Using wsdiff(1) is fairly simple:
#       - Bringover to a fresh workspace
#       - Perform a full non-debug build (clobber if workspace isn't fresh)
#       - Move the proto area aside, call it proto.old, or something.
#       - Integrate your changes to the workspace
#       - Perform another full non-debug clobber build.
#       - Use wsdiff(1) to see what changed:
#               $ wsdiff proto.old proto
#
# By default, wsdiff will print the list of changed objects / deliverables to
# stdout. If a results file is specified via -r, the list of differing objects,
# and details about why wsdiff(1) thinks they are different will be logged to
# the results file.
#
# By invoking nightly(1) with the -w option to NIGHTLY_FLAGS, nightly(1) will us
# wsdiff(1) to report on what objects changed since the last build.
#
# For patch deliverable purposes, it's advised to have nightly do a clobber,
# non-debug build.
#
# Think about the results. Was something flagged that you don't expect? Go look
# at the results file to see details about the differences.
#
# Use the -i option in conjunction with -v and -V to dive deeper and have wsdiff
# report with more verbosity.
#
# Usage: wsdiff [-vVt] [-r results ] [-i filelist ] old new
#
```



```
# Where "old" is the path to the proto area build without the changes, and
# "new" is the path to the proto area built with the changes. The following
# options are supported:
#
#        -v      Do not truncate observed diffs in results
#        -V      Log *all* ELF sect diffs vs. logging the first diff found
#        -t      Use onbld tools in $SRC/tools
#        -r      Log results and observed differences
#        -i      Tell wsdiff which objects to compare via an input file list

import datetime, fnmatch, getopt, os, profile, commands
import re, resource, select, shutil, signal, string, struct, sys, tempfile
import time, threading
from stat import *

# Human readable diffs truncated by default if longer than this
# Specifying -v on the command line will override
diffs_sz_thresh = 4096

# Lock name      Provides exclusive access to
# -------------+-------------------------------------------------
# output_lock    standard output or temporary file (difference())
# log_lock       the results file (log_difference())
# wset_lock      changedFiles list (workerThread())
output_lock = threading.Lock()
log_lock = threading.Lock()
wset_lock = threading.Lock()

# Variable for thread control
keep_processing = True

# Default search path for wsdiff
wsdiff_path = [ "/usr/bin",
                "/usr/ccs/bin",
                "/lib/svc/bin",
                "/opt/onbld/bin" ]

# These are objects that wsdiff will notice look different, but will not report.
# Existence of an exceptions list, and adding things here is *dangerous*,
# and therefore the *only* reasons why anything would be listed here is because
# the objects do not build deterministically, yet we *cannot* fix this.
#
# These perl libraries use __DATE__ and therefore always look different.
# Ideally, we would purge use the use of __DATE__ from the source, but because
# this is source we wish to distribute with Solaris "unchanged", we cannot modif
#
wsdiff_exceptions = [ "usr/perl5/5.8.4/lib/sun4-solaris-64int/CORE/libperl.so.1"
                      "usr/perl5/5.6.1/lib/sun4-solaris-64int/CORE/libperl.so.1"
                      "usr/perl5/5.8.4/lib/i86pc-solaris-64int/CORE/libperl.so.1
                      "usr/perl5/5.6.1/lib/i86pc-solaris-64int/CORE/libperl.so.1
                      ]

#####
# Logging routines
#

# Debug message to be printed to the screen, and the log file
def debug(msg) :

        # Add prefix to highlight debugging message
        msg = "## " + msg
        if debugon :
                output_lock.acquire()
                print >> sys.stdout, msg
                sys.stdout.flush()
                output_lock.release()
```

```
128                 if logging :
129                         log_lock.acquire()
130                         print >> log, msg
131                         log.flush()
132                         log_lock.release()

134 # Informational message to be printed to the screen, and the log file
135 def info(msg) :

137         output_lock.acquire()
138         print >> sys.stdout, msg
139         sys.stdout.flush()
140         output_lock.release()
141         if logging :
142                 log_lock.acquire()
143                 print >> log, msg
144                 log.flush()
145                 log_lock.release()

147 # Error message to be printed to the screen, and the log file
148 def error(msg) :
149
150         output_lock.acquire()
151         print >> sys.stderr, "ERROR:", msg
152         sys.stderr.flush()
153         output_lock.release()
154         if logging :
155                 log_lock.acquire()
156                 print >> log, "ERROR:", msg
157                 log.flush()
158                 log_lock.release()

160 # Informational message to be printed only to the log, if there is one.
161 def v_info(msg) :

163         if logging :
164                 log_lock.acquire()
165                 print >> log, msg
166                 log.flush()
167                 log_lock.release()
168
169 #
170 # Flag a detected file difference
171 # Display the fileName to stdout, and log the difference
172 #
173 def difference(f, dtype, diffs) :

175         if f in wsdiff_exceptions :
176                 return

178         output_lock.acquire()
179         if sorted :
180                 differentFiles.append(f)
181         else:
182                 print >> sys.stdout, f
183                 sys.stdout.flush()
184         output_lock.release()

186         log_difference(f, dtype, diffs)

188 #
189 # Do the actual logging of the difference to the results file
190 #
191 def log_difference(f, dtype, diffs) :

193         if logging :
```



```
194                 log_lock.acquire()
195                 print >> log, f
196                 print >> log, "NOTE:", dtype, "difference detected."

198                 difflen = len(diffs)
199                 if difflen > 0 :
200                         print >> log

202                         if not vdiffs and difflen > diffs_sz_thresh :
203                                 print >> log, diffs[:diffs_sz_thresh]
204                                 print >> log, \
205                                       "... truncated due to length: " \
206                                       "use -v to override ..."
207                         else :
208                                 print >> log, diffs
209                         print >> log, "\n"
210                 log.flush()
211                 log_lock.release()


214 #####
215 # diff generating routines
216 #

218 #
219 # Return human readable diffs from two temporary files
220 #
221 def diffFileData(tmpf1, tmpf2) :

223         binaries = False

225         # Filter the data through od(1) if the data is detected
226         # as being binary
227         if isBinary(tmpf1) or isBinary(tmpf2) :
228                 binaries = True
229                 tmp_od1 = tmpf1 + ".od"
230                 tmp_od2 = tmpf2 + ".od"
231
232                 cmd = od_cmd + " -c -t x4" + " " + tmpf1 + " > " + tmp_od1
233                 os.system(cmd)
234                 cmd = od_cmd + " -c -t x4" + " " + tmpf2 + " > " + tmp_od2
235                 os.system(cmd)
236
237                 tmpf1 = tmp_od1
238                 tmpf2 = tmp_od2

240         try:
241                 data = commands.getoutput(diff_cmd + " " + tmpf1 + " " + tmpf2)
242                 # Remove the temp files as we no longer need them.
243                 if binaries :
244                         try:
245                                 os.unlink(tmp_od1)
246                         except OSError, e:
247                                 error("diffFileData: unlink failed %s" % e)
248                         try:
249                                 os.unlink(tmp_od2)
250                         except OSError, e:
251                                 error("diffFileData: unlink failed %s" % e)
252         except:
253                 error("failed to get output of command: " + diff_cmd + " " \
254                     + tmpf1 + " " + tmpf2)

256                 # Send exception for the failed command up
257                 raise
258                 return
```

```
260         return data

262 #
263 # Return human readable diffs betweeen two datasets
264 #
265 def diffData(base, ptch, d1, d2) :

267         t = threading.currentThread()
268         tmpFile1 = tmpDir1 + os.path.basename(base) + t.getName()
269         tmpFile2 = tmpDir2 + os.path.basename(ptch) + t.getName()

271         try:
272                 fd1 = open(tmpFile1, "w")
273         except:
274                 error("failed to open: " + tmpFile1)
275                 cleanup(1)

277         try:
278                 fd2 = open(tmpFile2, "w")
279         except:
280                 error("failed to open: " + tmpFile2)
281                 cleanup(1)

283         fd1.write(d1)
284         fd2.write(d2)
285         fd1.close()
286         fd2.close()

288         return diffFileData(tmpFile1, tmpFile2)

290 #####
291 # Misc utility functions
292 #

294 # Prune off the leading prefix from string s
295 def str_prefix_trunc(s, prefix) :
296         snipLen = len(prefix)
297         return s[snipLen:]

299 #
300 # Prune off leading proto path goo (if there is one) to yield
301 # the deliverable's eventual path relative to root
302 # e.g. proto.base/root_sparc/usr/src/cmd/prstat => usr/src/cmd/prstat
303 #
304 def fnFormat(fn) :
305         root_arch_str = "root_" + arch

307         pos = fn.find(root_arch_str)
308         if pos == -1 :
309                 return fn

311         pos = fn.find("/", pos)
312         if pos == -1 :
313                 return fn

315         return fn[pos + 1:]

317 #####
318 # Usage / argument processing
319 #

321 #
322 # Display usage message
323 #
324 def usage() :
325         sys.stdout.flush()
```



```
326         print >> sys.stderr, """Usage: wsdiff [-dvVst] [-r results ] [-i filelis
327         -d      Print debug messages about the progress
328         -v      Do not truncate observed diffs in results
329         -V      Log *all* ELF sect diffs vs. logging the first diff found
330         -t      Use onbld tools in $SRC/tools
331         -r      Log results and observed differences
332         -s      Produce sorted list of differences
333         -i      Tell wsdiff which objects to compare via an input file list"""
334         sys.exit(1)

336 #
337 # Process command line options
338 #
339 def args() :

341         global debugon
342         global logging
343         global vdiffs
344         global reportAllSects
345         global sorted

347         validOpts = 'di:r:vVst?'

349         baseRoot = ""
350         ptchRoot = ""
351         fileNamesFile = ""
352         results = ""
353         localTools = False

355         # getopt.getopt() returns:
356         #       an option/value tuple
357         #       a list of remaining non-option arguments
358         #
359         # A correct wsdiff invocation will have exactly two non option
360         # arguments, the paths to the base (old), ptch (new) proto areas
361         try:
362                 optlist, args = getopt.getopt(sys.argv[1:], validOpts)
363         except getopt.error, val:
364                 usage()

366         if len(args) != 2 :
367                 usage();

369         for opt,val in optlist :
370                 if opt == '-d' :
371                         debugon = True
372                 elif opt == '-i' :
373                         fileNamesFile = val
374                 elif opt == '-r' :
375                         results = val
376                         logging = True
377                 elif opt == '-s' :
378                         sorted = True
379                 elif opt == '-v' :
380                         vdiffs = True
381                 elif opt == '-V' :
382                         reportAllSects = True
383                 elif opt == '-t':
384                         localTools = True
385                 else:
386                         usage()

388         baseRoot = args[0]
389         ptchRoot = args[1]

391         if len(baseRoot) == 0 or len(ptchRoot) == 0 :
```

```
392                 usage()

394         if logging and len(results) == 0 :
395                 usage()

397         if vdiffs and not logging :
398                 error("The -v option requires a results file (-r)")
399                 sys.exit(1)

401         if reportAllSects and not logging :
402                 error("The -V option requires a results file (-r)")
403                 sys.exit(1)

405         # alphabetical order
406         return  baseRoot, fileNamesFile, localTools, ptchRoot, results

408 #####
409 # File identification
410 #

412 #
413 # Identify the file type.
414 # If it's not ELF, use the file extension to identify
415 # certain file types that require special handling to
416 # compare. Otherwise just return a basic "ASCII" type.
417 #
418 def getTheFileType(f) :

420         extensions = { 'a'      :       'ELF Object Archive',
421                        'jar'    :       'Java Archive',
422                        'html'   :       'HTML',
423                        'ln'     :       'Lint Library',
424                        'db'     :       'Sqlite Database' }

426         try:
427                 if os.stat(f)[ST_SIZE] == 0 :
428                         return 'ASCII'
429         except:
430                 error("failed to stat " + f)
431                 return 'Error'

433         if isELF(f) == 1 :
434                 return 'ELF'

436         fnamelist = f.split('.')
437         if len(fnamelist) > 1 : # Test the file extension
438                 extension = fnamelist[-1]
439                 if extension in extensions.keys():
440                         return extensions[extension]

442         return 'ASCII'

444 #
445 # Return non-zero if "f" is an ELF file
446 #
447 elfmagic = '\177ELF'
448 def isELF(f) :
449         try:
450                 fd = open(f)
451         except:
452                 error("failed to open: " + f)
453                 return 0
454         magic = fd.read(len(elfmagic))
455         fd.close()

457         if magic == elfmagic :
```



```
458                 return 1
459         return 0

461 #
462 # Return non-zero is "f" is binary.
463 # Consider the file to be binary if it contains any null characters
464 #
465 def isBinary(f) :
466         try:
467                 fd = open(f)
468         except:
469                 error("failed to open: " + f)
470                 return 0
471         s = fd.read()
472         fd.close()

474         if s.find('\0') == -1 :
475                 return 0
476         else :
477                 return 1

479 #####
480 # Directory traversal and file finding
481 #

483 #
484 # Return a sorted list of files found under the specified directory
485 #
486 def findFiles(d) :
487         for path, subdirs, files in os.walk(d) :
488                 files.sort()
489                 for name in files :
490                         yield os.path.join(path, name)

492 #
493 # Examine all files in base, ptch
494 #
495 # Return a list of files appearing in both proto areas,
496 # a list of new files (files found only in ptch) and
497 # a list of deleted files (files found only in base)
498 #
499 def protoCatalog(base, ptch) :

501         compFiles = []          # List of files in both proto areas
502         ptchList = []           # List of file in patch proto area

504         newFiles = []           # New files detected
505         deletedFiles = []       # Deleted files

507         debug("Getting the list of files in the base area");
508         baseFilesList = list(findFiles(base))
509         baseStringLength = len(base)
510         debug("Found " + str(len(baseFilesList)) + " files")
511
512         debug("Getting the list of files in the patch area");
513         ptchFilesList = list(findFiles(ptch))
514         ptchStringLength = len(ptch)
515         debug("Found " + str(len(ptchFilesList)) + " files")

517         # Inventory files in the base proto area
518         debug("Determining the list of regular files in the base area");
519         for fn in baseFilesList :
520                 if os.path.islink(fn) :
521                         continue

523                 fileName = fn[baseStringLength:]
```

```
524                 compFiles.append(fileName)
525         debug("Found " + str(len(compFiles)) + " files")

527         # Inventory files in the patch proto area
528         debug("Determining the list of regular files in the patch area");
529         for fn in ptchFilesList :
530                 if os.path.islink(fn) :
531                         continue

533                 fileName = fn[ptchStringLength:]
534                 ptchList.append(fileName)
535         debug("Found " + str(len(ptchList)) + " files")

537         # Deleted files appear in the base area, but not the patch area
538         debug("Searching for deleted files by comparing the lists")
539         for fileName in compFiles :
540                 if not fileName in ptchList :
541                         deletedFiles.append(fileName)
542         debug("Found " + str(len(deletedFiles)) + " deleted files")

544         # Eliminate "deleted" files from the list of objects appearing
545         # in both the base and patch proto areas
546         debug("Eliminating deleted files from the list of objects")
547         for fileName in deletedFiles :
548                 try:
549                         compFiles.remove(fileName)
550                 except:
551                         error("filelist.remove() failed")
552         debug("List for comparison reduced to " + str(len(compFiles)) \
553             + " files")

555         # New files appear in the patch area, but not the base
556         debug("Getting the list of newly added files")
557         for fileName in ptchList :
558                 if not fileName in compFiles :
559                         newFiles.append(fileName)
560         debug("Found " + str(len(newFiles)) + " new files")

562         return compFiles, newFiles, deletedFiles

564 #
565 # Examine the files listed in the input file list
566 #
567 # Return a list of files appearing in both proto areas,
568 # a list of new files (files found only in ptch) and
569 # a list of deleted files (files found only in base)
570 #
571 def flistCatalog(base, ptch, flist) :
572         compFiles = []          # List of files in both proto areas
573         newFiles = []           # New files detected
574         deletedFiles = []       # Deleted files

576         try:
577                 fd = open(flist, "r")
578         except:
579                 error("could not open: " + flist)
580                 cleanup(1)

582         files = []
583         files = fd.readlines()
584         fd.close()

586         for f in files :
587                 ptch_present = True
588                 base_present = True
```



```
590                 if f == '\n' :
591                         continue

593                 # the fileNames have a trailing '\n'
594                 f = f.rstrip()

596                 # The objects in the file list have paths relative
597                 # to $ROOT or to the base/ptch directory specified on
598                 # the command line.
599                 # If it's relative to $ROOT, we'll need to add back the
600                 # root_'uname -p' goo we stripped off in fnFormat()
601                 if os.path.exists(base + f) :
602                         fn = f;
603                 elif os.path.exists(base + "root_" + arch + "/" + f) :
604                         fn = "root_" + arch + "/" + f
605                 else :
606                         base_present = False

608                 if base_present :
609                         if not os.path.exists(ptch + fn) :
610                                 ptch_present = False
611                 else :
612                         if os.path.exists(ptch + f) :
613                                 fn = f
614                         elif os.path.exists(ptch + "root_" + arch + "/" + f) :
615                                 fn = "root_" + arch + "/" + f
616                         else :
617                                 ptch_present = False

619                 if os.path.islink(base + fn) :  # ignore links
620                         base_present = False
621                 if os.path.islink(ptch + fn) :
622                         ptch_present = False

624                 if base_present and ptch_present :
625                         compFiles.append(fn)
626                 elif base_present :
627                         deletedFiles.append(fn)
628                 elif ptch_present :
629                         newFiles.append(fn)
630                 else :
631                         if os.path.islink(base + fn) and \
632                             os.path.islink(ptch + fn) :
633                                 continue
634                         error(f + " in file list, but not in either tree. " + \
635                             "Skipping...")

637         return compFiles, newFiles, deletedFiles


640 #
641 # Build a fully qualified path to an external tool/utility.
642 # Consider the default system locations. For onbld tools, if
643 # the -t option was specified, we'll try to use built tools in $SRC tools,
644 # and otherwise, we'll fall back on /opt/onbld/
645 #
646 def find_tool(tool) :

648         # First, check what was passed
649         if os.path.exists(tool) :
650                 return tool

652         # Next try in wsdiff path
653         for pdir in wsdiff_path :
654                 location = pdir + "/" + tool
655                 if os.path.exists(location) :
```

```
656                         return location + " "

658                 location = pdir + "/" + arch + "/" + tool
659                 if os.path.exists(location) :
660                         return location + " "

662         error("Could not find path to: " + tool);
663         sys.exit(1);


666 #####
667 # ELF file comparison helper routines
668 #

670 #
671 # Return a dictionary of ELF section types keyed by section name
672 #
673 def get_elfheader(f) :

675         header = {}

677         hstring = commands.getoutput(elfdump_cmd + " -c " + f)

679         if len(hstring) == 0 :
680                 error("Failed to dump ELF header for " + f)
681                 raise
682                 return

684         # elfdump(1) dumps the section headers with the section name
685         # following "sh_name:", and the section type following "sh_type:"
686         sections = hstring.split("Section Header")
687         for sect in sections :
688                 datap = sect.find("sh_name:");
689                 if datap == -1 :
690                         continue
691                 section = sect[datap:].split()[1]
692                 datap = sect.find("sh_type:");
693                 if datap == -1 :
694                         error("Could not get type for sect: " + section + \
695                                 " in " + f)
696                 sh_type = sect[datap:].split()[2]
697                 header[section] = sh_type

699         return header

701 #
702 # Extract data in the specified ELF section from the given file
703 #
704 def extract_elf_section(f, section) :

706         data = commands.getoutput(dump_cmd + " -sn " + section + " " + f)

708         if len(data) == 0 :
709                 error(dump_cmd + "yielded no data on section " + section + \
710                     " of " + f)
711                 raise
712                 return

714         # dump(1) displays the file name to start...
715         # get past it to the data itself
716         dbegin = data.find(":") + 1
717         data = data[dbegin:];

719         return (data)

721 #
```



```
722 # Return a (hopefully meaningful) human readable set of diffs
723 # for the specified ELF section between f1 and f2
724 #
725 # Depending on the section, various means for dumping and diffing
726 # the data may be employed.
727 #
728 text_sections = [ '.text', '.init', '.fini' ]
729 def diff_elf_section(f1, f2, section, sh_type) :

731         t = threading.currentThread()
732         tmpFile1 = tmpDir1 + os.path.basename(f1) + t.getName()
733         tmpFile2 = tmpDir2 + os.path.basename(f2) + t.getName()

735         if (sh_type == "SHT_RELA") : # sh_type == SHT_RELA
736                 cmd1 = elfdump_cmd + " -r " + f1 + " > " + tmpFile1
737                 cmd2 = elfdump_cmd + " -r " + f2 + " > " + tmpFile2
738         elif (section == ".group") :
739                 cmd1 = elfdump_cmd + " -g " + f1 + " > " + tmpFile1
740                 cmd2 = elfdump_cmd + " -g " + f2 + " > " + tmpFile2
741         elif (section == ".hash") :
742                 cmd1 = elfdump_cmd + " -h " + f1 + " > " + tmpFile1
743                 cmd2 = elfdump_cmd + " -h " + f2 + " > " + tmpFile2
744         elif (section == ".dynamic") :
745                 cmd1 = elfdump_cmd + " -d " + f1 + " > " + tmpFile1
746                 cmd2 = elfdump_cmd + " -d " + f2 + " > " + tmpFile2
747         elif (section == ".got") :
748                 cmd1 = elfdump_cmd + " -G " + f1 + " > " + tmpFile1
749                 cmd2 = elfdump_cmd + " -G " + f2 + " > " + tmpFile2
750         elif (section == ".SUNW_cap") :
751                 cmd1 = elfdump_cmd + " -H " + f1 + " > " + tmpFile1
752                 cmd2 = elfdump_cmd + " -H " + f2 + " > " + tmpFile2
753         elif (section == ".interp") :
754                 cmd1 = elfdump_cmd + " -i " + f1 + " > " + tmpFile1
755                 cmd2 = elfdump_cmd + " -i " + f2 + " > " + tmpFile2
756         elif (section == ".symtab" or section == ".dynsym") :
757                 cmd1 = elfdump_cmd + " -s -N " + section + " " + f1 + \
758                     " > " + tmpFile1
759                 cmd2 = elfdump_cmd + " -s -N " + section + " " + f2 + \
760                     " > " + tmpFile2
761         elif (section in text_sections) :
762                 # dis sometimes complains when it hits something it doesn't
763                 # know how to disassemble. Just ignore it, as the output
764                 # being generated here is human readable, and we've already
765                 # correctly flagged the difference.
766                 cmd1 = dis_cmd + " -t " + section + " " + f1 + \
767                         " 2>/dev/null | grep -v disassembly > " + tmpFile1
768                 cmd2 = dis_cmd + " -t " + section + " " + f2 + \
769                         " 2>/dev/null | grep -v disassembly > " + tmpFile2
770         else :
771                 cmd1 = elfdump_cmd + " -w " + tmpFile1 + " -N " + \
772                         section + " " + f1
773                 cmd2 = elfdump_cmd + " -w " + tmpFile2 + " -N " + \
774                         section + " " + f2

776         os.system(cmd1)
777         os.system(cmd2)

779         data = diffFileData(tmpFile1, tmpFile2)

781         # remove temp files as we no longer need them
782         try:
783                 os.unlink(tmpFile1)
784         except OSError, e:
785                 error("diff_elf_section: unlink failed %s" % e)
786         try:
787                 os.unlink(tmpFile2)
```

```
788         except OSError, e:
789                 error("diff_elf_section: unlink failed %s" % e)

791         return (data)

793 #
794 # compare the relevant sections of two ELF binaries
795 # and report any differences
796 #
797 # Returns: 1 if any differenes found
798 #          0 if no differences found
799 #         -1 on error
800 #

802 # Sections deliberately not considered when comparing two ELF
803 # binaries. Differences observed in these sections are not considered
804 # significant where patch deliverable identification is concerned.
805 sections_to_skip = [ ".SUNW_signature",
806                      ".comment",
807                      ".SUNW_ctf",
808                      ".debug",
809                      ".plt",
810                      ".rela.bss",
811                      ".rela.plt",
812                      ".line",
813                      ".note",
814                      ".compcom",
815                      ]

817 sections_preferred = [ ".rodata.str1.8",
818                        ".rodata.str1.1",
819                        ".rodata",
820                        ".data1",
821                        ".data",
822                        ".text",
823                        ]

825 def compareElfs(base, ptch, quiet) :

827         global logging

829         try:
830                 base_header = get_elfheader(base)
831         except:
832                 return
833         sections = base_header.keys()

835         try:
836                 ptch_header = get_elfheader(ptch)
837         except:
838                 return
839         e2_only_sections = ptch_header.keys()

841         e1_only_sections = []

843         fileName = fnFormat(base)

845         # Derive the list of ELF sections found only in
846         # either e1 or e2.
847         for sect in sections :
848                 if not sect in e2_only_sections :
849                         e1_only_sections.append(sect)
850                 else :
851                         e2_only_sections.remove(sect)

853         if len(e1_only_sections) > 0 :
```



```
854                 if quiet :
855                         return 1

857                 data = ""
858                 if logging :
859                         slist = ""
860                         for sect in e1_only_sections :
861                                 slist = slist + sect + "\t"
862                         data = "ELF sections found in " + \
863                                 base + " but not in " + ptch + \
864                                 "\n\n" + slist

866                 difference(fileName, "ELF", data)
867                 return 1
868
869         if len(e2_only_sections) > 0 :
870                 if quiet :
871                         return 1
872
873                 data = ""
874                 if logging :
875                         slist = ""
876                         for sect in e2_only_sections :
877                                 slist = slist + sect + "\t"
878                         data = "ELF sections found in " + \
879                                 ptch + " but not in " + base + \
880                                 "\n\n" + slist

882                 difference(fileName, "ELF", data)
883                 return 1

885         # Look for preferred sections, and put those at the
886         # top of the list of sections to compare
887         for psect in sections_preferred :
888                 if psect in sections :
889                         sections.remove(psect)
890                         sections.insert(0, psect)

892         # Compare ELF sections
893         first_section = True
894         for sect in sections :

896                 if sect in sections_to_skip :
897                         continue

899                 try:
900                         s1 = extract_elf_section(base, sect);
901                 except:
902                         return

904                 try:
905                         s2 = extract_elf_section(ptch, sect);
906                 except:
907                         return

909                 if len(s1) != len (s2) or s1 != s2:
910                         if not quiet:
911                                 sh_type = base_header[sect]
912                                 data = diff_elf_section(base, ptch, \
913                                                         sect, sh_type)

915                                 # If all ELF sections are being reported, then
916                                 # invoke difference() to flag the file name to
917                                 # stdout only once. Any other section difference
918                                 # should be logged to the results file directly
919                                 if not first_section :
```

```
920                                         log_difference(fileName, \
921                                             "ELF " + sect, data)
922                                 else :
923                                         difference(fileName, "ELF " + sect, \
924                                             data)
925
926                         if not reportAllSects :
927                                 return 1
928                         first_section = False
929
930         return 0
931
932 #####
933 # recursively remove 2 directories
934 #
935 # Used for removal of temporary directory strucures (ignores any errors).
936 #
937 def clearTmpDirs(dir1, dir2) :
938
939         if os.path.isdir(dir1) > 0 :
940                 shutil.rmtree(dir1, True)
941
942         if os.path.isdir(dir2) > 0 :
943                 shutil.rmtree(dir2, True)
944
945
946 #####
947 # Archive object comparison
948 #
949 # Returns 1 if difference detected
950 #         0 if no difference detected
951 #        -1 on error
952 #
953 def compareArchives(base, ptch, fileType) :
954
955         fileName = fnFormat(base)
956         t = threading.currentThread()
957         ArchTmpDir1 = tmpDir1 + os.path.basename(base) + t.getName()
958         ArchTmpDir2 = tmpDir2 + os.path.basename(base) + t.getName()
959
960         #
961         # Be optimistic and first try a straight file compare
962         # as it will allow us to finish up quickly.
963         #
964         if compareBasic(base, ptch, True, fileType) == 0 :
965                 return 0
966
967         try:
968                 os.makedirs(ArchTmpDir1)
969         except OSError, e:
970                 error("compareArchives: makedir failed %s" % e)
971                 return -1
972         try:
973                 os.makedirs(ArchTmpDir2)
974         except OSError, e:
975                 error("compareArchives: makedir failed %s" % e)
976                 return -1
977
978         # copy over the objects to the temp areas, and
979         # unpack them
980         baseCmd = "cp -fp " + base + " " + ArchTmpDir1
981         status, output = commands.getstatusoutput(baseCmd)
982         if status != 0 :
983                 error(baseCmd + " failed: " + output)
984                 clearTmpDirs(ArchTmpDir1, ArchTmpDir2)
985                 return -1
```



```
986
987         ptchCmd = "cp -fp " + ptch + " " + ArchTmpDir2
988         status, output = commands.getstatusoutput(ptchCmd)
989         if status != 0 :
990                 error(ptchCmd + " failed: " + output)
991                 clearTmpDirs(ArchTmpDir1, ArchTmpDir2)
992                 return -1
993
994         bname = string.split(fileName, '/')[-1]
995         if fileType == "Java Archive" :
996                 baseCmd = "cd " + ArchTmpDir1 + "; " + "jar xf " + bname + \
997                           "; rm -f " + bname + " META-INF/MANIFEST.MF"
998                 ptchCmd = "cd " + ArchTmpDir2 + "; " + "jar xf " + bname + \
999                           "; rm -f " + bname + " META-INF/MANIFEST.MF"
1000        elif fileType == "ELF Object Archive" :
1001                baseCmd = "cd " + ArchTmpDir1 + "; " + "/usr/ccs/bin/ar x " + \
1002                          bname + "; rm -f " + bname
1003                ptchCmd = "cd " + ArchTmpDir2 + "; " + "/usr/ccs/bin/ar x " + \
1004                          bname + "; rm -f " + bname
1005        else :
1006                error("unexpected file type: " + fileType)
1007                clearTmpDirs(ArchTmpDir1, ArchTmpDir2)
1008                return -1
1009
1010        os.system(baseCmd)
1011        os.system(ptchCmd)
1012
1013        baseFlist = list(findFiles(ArchTmpDir1))
1014        ptchFlist = list(findFiles(ArchTmpDir2))
1015
1016        # Trim leading path off base/ptch file lists
1017        flist = []
1018        for fn in baseFlist :
1019                flist.append(str_prefix_trunc(fn, ArchTmpDir1))
1020        baseFlist = flist
1021
1022        flist = []
1023        for fn in ptchFlist :
1024                flist.append(str_prefix_trunc(fn, ArchTmpDir2))
1025        ptchFlist = flist
1026
1027        for fn in ptchFlist :
1028                if not fn in baseFlist :
1029                        difference(fileName, fileType, \
1030                                   fn + " added to " + fileName)
1031                        clearTmpDirs(ArchTmpDir1, ArchTmpDir2)
1032                        return 1
1033
1034        for fn in baseFlist :
1035                if not fn in ptchFlist :
1036                        difference(fileName, fileType, \
1037                                   fn + " removed from " + fileName)
1038                        clearTmpDirs(ArchTmpDir1, ArchTmpDir2)
1039                        return 1
1040
1041                differs = compareOneFile((ArchTmpDir1 + fn), \
1042                    (ArchTmpDir2 + fn), True)
1043                if differs :
1044                        difference(fileName, fileType, \
1045                                   fn + " in " + fileName + " differs")
1046                        clearTmpDirs(ArchTmpDir1, ArchTmpDir2)
1047                        return 1
1048
1049        clearTmpDirs(ArchTmpDir1, ArchTmpDir2)
1050        return 0
```

```
1052 #####
1053 # (Basic) file comparison
1054 #
1055 # There's some special case code here for Javadoc HTML files
1056 #
1057 # Returns 1 if difference detected
1058 #         0 if no difference detected
1059 #        -1 on error
1060 #
1061 def compareBasic(base, ptch, quiet, fileType) :

1063         fileName = fnFormat(base);

1065         if quiet and os.stat(base)[ST_SIZE] != os.stat(ptch)[ST_SIZE] :
1066                 return 1

1068         try:
1069                 baseFile = open(base)
1070         except:
1071                 error("could not open " + base)
1072                 return -1
1073         try:
1074                 ptchFile = open(ptch)
1075         except:
1076                 error("could not open " + ptch)
1077                 return -1

1079         baseData = baseFile.read()
1080         ptchData = ptchFile.read()

1082         baseFile.close()
1083         ptchFile.close()

1085         needToSnip = False
1086         if fileType == "HTML" :
1087                 needToSnip = True
1088                 toSnipBeginStr = "<!-- Generated by javadoc"
1089                 toSnipEndStr = "-->\n"

1091         if needToSnip :
1092                 toSnipBegin = string.find(baseData, toSnipBeginStr)
1093                 if toSnipBegin != -1 :
1094                         toSnipEnd = string.find(baseData[toSnipBegin:], \
1095                                                 toSnipEndStr) + \
1096                                                 len(toSnipEndStr)
1097                         baseData = baseData[:toSnipBegin] + \
1098                                    baseData[toSnipBegin + toSnipEnd:]
1099                         ptchData = ptchData[:toSnipBegin] + \
1100                                    ptchData[toSnipBegin + toSnipEnd:]

1102         if quiet :
1103                 if baseData != ptchData :
1104                         return 1
1105         else :
1106                 if len(baseData) != len(ptchData) or baseData != ptchData :
1107                         diffs = diffData(base, ptch, baseData, ptchData)
1108                         difference(fileName, fileType, diffs)
1109                         return 1
1110         return 0


1113 #####
1114 # Compare two objects by producing a data dump from
1115 # each object, and then comparing the dump data
1116 #
1117 # Returns: 1 if a difference is detected
```



```
1118 #          0 if no difference detected
1119 #         -1 upon error
1120 #
1121 def compareByDumping(base, ptch, quiet, fileType) :

1123         fileName = fnFormat(base);
1124         t = threading.currentThread()
1125         tmpFile1 = tmpDir1 + os.path.basename(base) + t.getName()
1126         tmpFile2 = tmpDir2 + os.path.basename(ptch) + t.getName()

1128         if fileType == "Lint Library" :
1129                 baseCmd = lintdump_cmd + " -ir " + base + \
1130                           " | egrep -v '(LINTOBJ|LINTMOD):'" + \
1131                           " | grep -v PASS[1-3]:" + \
1132                           " > " + tmpFile1
1133                 ptchCmd = lintdump_cmd + " -ir " + ptch + \
1134                           " | egrep -v '(LINTOBJ|LINTMOD):'" + \
1135                           " | grep -v PASS[1-3]:" + \
1136                           " > " + tmpFile2
1137         elif fileType == "Sqlite Database" :
1138                 baseCmd = "echo .dump | " + sqlite_cmd + base + " > " + \
1139                           tmpFile1
1140                 ptchCmd = "echo .dump | " + sqlite_cmd + ptch + " > " + \
1141                           tmpFile2
1142
1143         os.system(baseCmd)
1144         os.system(ptchCmd)

1146         try:
1147                 baseFile = open(tmpFile1)
1148         except:
1149                 error("could not open: " + tmpFile1)
1150                 return
1151         try:
1152                 ptchFile = open(tmpFile2)
1153         except:
1154                 error("could not open: " + tmpFile2)
1155                 return

1157         baseData = baseFile.read()
1158         ptchData = ptchFile.read()

1160         baseFile.close()
1161         ptchFile.close()

1163         if len(baseData) != len(ptchData) or baseData != ptchData :
1164                 if not quiet :
1165                         data = diffFileData(tmpFile1, tmpFile2);
1166                         try:
1167                                 os.unlink(tmpFile1)
1168                         except OSError, e:
1169                                 error("compareByDumping: unlink failed %s" % e)
1170                         try:
1171                                 os.unlink(tmpFile2)
1172                         except OSError, e:
1173                                 error("compareByDumping: unlink failed %s" % e)
1174                         difference(fileName, fileType, data)
1175                 return 1

1177         # Remove the temporary files now.
1178         try:
1179                 os.unlink(tmpFile1)
1180         except OSError, e:
1181                 error("compareByDumping: unlink failed %s" % e)
1182         try:
1183                 os.unlink(tmpFile2)
```

```
1184         except OSError, e:
1185                 error("compareByDumping: unlink failed %s" % e)

1187         return 0

1189 #####
1190 #
1191 # SIGINT signal handler. Changes thread control variable to tell the threads
1192 # to finish their current job and exit.
1193 #
1194 def discontinue_processing(signl, frme):
1195         global keep_processing

1197         print >> sys.stderr, "Caught Ctrl-C, stopping the threads"
1198         keep_processing = False

1200         return 0

1202 #####
1203 #
1204 # worker thread for changedFiles processing
1205 #
1206 class workerThread(threading.Thread) :
1207     def run(self):
1208         global wset_lock
1209         global changedFiles
1210         global baseRoot
1211         global ptchRoot
1212         global keep_processing

1214         while (keep_processing) :
1215                 # grab the lock to changedFiles and remove one member
1216                 # and process it
1217                 wset_lock.acquire()
1218                 try :
1219                         fn = changedFiles.pop()
1220                 except IndexError :
1221                         # there is nothing more to do
1222                         wset_lock.release()
1223                         return
1224                 wset_lock.release()

1226                 base = baseRoot + fn
1227                 ptch = ptchRoot + fn

1229                 compareOneFile(base, ptch, False)


1232 #####
1233 # Compare two objects. Detect type changes.
1234 # Vector off to the appropriate type specific
1235 # compare routine based on the type.
1236 #
1237 def compareOneFile(base, ptch, quiet) :

1239         # Verify the file types.
1240         # If they are different, indicate this and move on
1241         btype = getTheFileType(base)
1242         ptype = getTheFileType(ptch)

1244         if btype == 'Error' or ptype == 'Error' :
1245                 return -1

1247         fileName = fnFormat(base)

1249         if (btype != ptype) :
```



```
1250                 if not quiet :
1251                         difference(fileName, "file type", btype + " to " + ptype
1252                 return 1
1253         else :
1254                 fileType = btype

1256         if (fileType == 'ELF') :
1257                 return compareElfs(base, ptch, quiet)

1259         elif (fileType == 'Java Archive' or fileType == 'ELF Object Archive') :
1260                 return compareArchives(base, ptch, fileType)

1262         elif (fileType == 'HTML') :
1263                 return compareBasic(base, ptch, quiet, fileType)

1265         elif ( fileType == 'Lint Library' ) :
1266                 return compareByDumping(base, ptch, quiet, fileType)

1268         elif ( fileType == 'Sqlite Database' ) :
1269                 return compareByDumping(base, ptch, quiet, fileType)

1271         else :
1272                 # it has to be some variety of text file
1273                 return compareBasic(base, ptch, quiet, fileType)

1275 # Cleanup and self-terminate
1276 def cleanup(ret) :

1278         debug("Performing cleanup (" + str(ret) + ")")
1279         if os.path.isdir(tmpDir1) > 0 :
1280                 shutil.rmtree(tmpDir1)
1281
1282         if os.path.isdir(tmpDir2) > 0 :
1283                 shutil.rmtree(tmpDir2)
1284
1285         if logging :
1286                 log.close()

1288         sys.exit(ret)

1290 def main() :

1292         # Log file handle
1293         global log

1295         # Globals relating to command line options
1296         global logging, vdiffs, reportAllSects

1298         # Named temporary files / directories
1299         global tmpDir1, tmpDir2

1301         # Command paths
1302         global lintdump_cmd, elfdump_cmd, dump_cmd, dis_cmd, od_cmd, diff_cmd, s

1304         # Default search path
1305         global wsdiff_path

1307         # Essentially "uname -p"
1308         global arch

1310         # changed files for worker thread processing
1311         global changedFiles
1312         global baseRoot
1313         global ptchRoot

1315         # Sort the list of files from a temporary file
```

```
1316        global sorted
1317        global differentFiles

1319        # Debugging indicator
1320        global debugon

1322        # Some globals need to be initialized
1323        debugon = logging = vdiffs = reportAllSects = sorted = False


1326        # Process command line arguments
1327        # Return values are returned from args() in alpha order
1328        # (Yes, python functions can return multiple values (ewww))
1329        # Note that args() also set the globals:
1330        #       logging to True if verbose logging (to a file) was enabled
1331        #       vdiffs to True if logged differences aren't to be truncated
1332        #       reportAllSects to True if all ELF section differences are to be
1333        #
1334        baseRoot, fileNamesFile, localTools, ptchRoot, results = args()

1336        #
1337        # Set up the results/log file
1338        #
1339        if logging :
1340                try:
1341                        log = open(results, "w")
1342                except:
1343                        logging = False
1344                        error("failed to open log file: " + log)
1345                        sys.exit(1)

1347                dateTimeStr= "# %04d-%02d-%02d at %02d:%02d:%02d" % time.localti
1347                dateTimeStr= "# %d/%d/%d at %d:%d:%d" % time.localtime()[:6]
1348                v_info("# This file was produced by wsdiff")
1349                v_info(dateTimeStr)

1351        # Changed files (used only for the sorted case)
1352        if sorted :
1353                differentFiles = []

1355        #
1356        # Build paths to the tools required tools
1357        #
1358        # Try to look for tools in $SRC/tools if the "-t" option
1359        # was specified
1360        #
1361        arch = commands.getoutput("uname -p")
1362        if localTools :
1363                try:
1364                        src = os.environ['SRC']
1365                except:
1366                        error("-t specified, but $SRC not set. Cannot find $SRC/
1367                        src = ""
1368                if len(src) > 0 :
1369                        wsdiff_path.insert(0, src + "/tools/proto/opt/onbld/bin"

1371        lintdump_cmd = find_tool("lintdump")
1372        elfdump_cmd = find_tool("elfdump")
1373        dump_cmd = find_tool("dump")
1374        od_cmd = find_tool("od")
1375        dis_cmd = find_tool("dis")
1376        diff_cmd = find_tool("diff")
1377        sqlite_cmd = find_tool("sqlite")

1379        #
1380        # Set resource limit for number of open files as high as possible.
```



```
1381        # This might get handy with big number of threads.
1382        #
1383        (nofile_soft, nofile_hard) = resource.getrlimit(resource.RLIMIT_NOFILE)
1384        try:
1385                resource.setrlimit(resource.RLIMIT_NOFILE,
1386                    (nofile_hard, nofile_hard))
1387        except:
1388                error("cannot set resource limits for number of open files")
1389                sys.exit(1)

1391        #
1392        # validate the base and patch paths
1393        #
1394        if baseRoot[-1] != '/' :
1395                baseRoot += '/'

1397        if ptchRoot[-1] != '/' :
1398                ptchRoot += '/'

1400        if not os.path.exists(baseRoot) :
1401                error("old proto area: " + baseRoot + " does not exist")
1402                sys.exit(1)

1404        if not os.path.exists(ptchRoot) :
1405                error("new proto area: " + ptchRoot + \
1406                      " does not exist")
1407                sys.exit(1)

1409        #
1410        # log some information identifying the run
1411        #
1412        v_info("Old proto area: " + baseRoot)
1413        v_info("New proto area: " + ptchRoot)
1414        v_info("Results file: " + results + "\n")

1416        #
1417        # Set up the temporary directories / files
1418        # Could use python's tmpdir routines, but these should
1419        # be easier to identify / keep around for debugging
1420        pid = os.getpid()
1421        tmpDir1 = "/tmp/wsdiff_tmp1_" + str(pid) + "/"
1422        tmpDir2 = "/tmp/wsdiff_tmp2_" + str(pid) + "/"
1423        try:
1424                os.makedirs(tmpDir1)
1425        except OSError, e:
1426                error("main: makedir failed %s" % e)
1427        try:
1428                os.makedirs(tmpDir2)
1429        except OSError, e:
1430                error("main: makedir failed %s" % e)

1432        # Derive a catalog of new, deleted, and to-be-compared objects
1433        # either from the specified base and patch proto areas, or from
1434        # from an input file list
1435        newOrDeleted = False

1437        if fileNamesFile != "" :
1438                changedFiles, newFiles, deletedFiles = \
1439                                flistCatalog(baseRoot, ptchRoot, fileNamesFile)
1440        else :
1441                changedFiles, newFiles, deletedFiles = \
1442                                  protoCatalog(baseRoot, ptchRoot)

1444        if len(newFiles) > 0 :
1445                newOrDeleted = True
1446                info("\nNew objects found: ")
```

```
1448                 if sorted :
1449                         newFiles.sort()
1450                 for fn in newFiles :
1451                         info(fnFormat(fn))

1453         if len(deletedFiles) > 0 :
1454                 newOrDeleted = True
1455                 info("\nObjects removed: ")

1457                 if sorted :
1458                         deletedFiles.sort()
1459                 for fn in deletedFiles :
1460                         info(fnFormat(fn))

1462         if newOrDeleted :
1463                 info("\nChanged objects: ")
1464         if sorted :
1465                 debug("The list will appear after the processing is done")

1467         # Here's where all the heavy lifting happens
1468         # Perform a comparison on each object appearing in
1469         # both proto areas. compareOneFile will examine the
1470         # file types of each object, and will vector off to
1471         # the appropriate comparison routine, where the compare
1472         # will happen, and any differences will be reported / logged

1474         # determine maximum number of worker threads by using
1475         # DMAKE_MAX_JOBS environment variable set by nightly(1)
1476         # or get number of CPUs in the system
1477         try:
1478                 max_threads = int(os.environ['DMAKE_MAX_JOBS'])
1479         except:
1480                 max_threads = os.sysconf("SC_NPROCESSORS_ONLN")
1481                 # If we cannot get number of online CPUs in the system
1482                 # run unparallelized otherwise bump the number up 20%
1483                 # to achieve best results.
1484                 if max_threads == -1 :
1485                         max_threads = 1
1486                 else :
1487                         max_threads += max_threads/5

1489         # Set signal handler to attempt graceful exit
1490         debug("Setting signal handler")
1491         signal.signal( signal.SIGINT, discontinue_processing )

1493         # Create and unleash the threads
1494         # Only at most max_threads must be running at any moment
1495         mythreads = []
1496         debug("Spawning " + str(max_threads) + " threads");
1497         for i in range(max_threads) :
1498                 thread = workerThread()
1499                 mythreads.append(thread)
1500                 mythreads[i].start()

1502         # Wait for the threads to finish and do cleanup if interrupted
1503         debug("Waiting for the threads to finish")
1504         while True:
1505                 if not True in [thread.isAlive() for thread in mythreads]:
1506                     break
1507                 else:
1508                     # Some threads are still going
1509                     time.sleep(1)

1511         # Interrupted by SIGINT
1512         if keep_processing == False :
```



```
1513                 cleanup(1)

1515         # If the list of differences was sorted it is stored in an array
1516         if sorted :
1517                 differentFiles.sort()
1518                 for f in differentFiles :
1519                         info(fnFormat(f))

1521         # We're done, cleanup.
1522         cleanup(0)

1524 if __name__ == '__main__' :
1525         try:
1526                 main()
1527         except KeyboardInterrupt :
1528                 cleanup(1);
```