```
**********************************************************
   18577 Thu Feb 12 20:32:16 2015
new/usr/src/uts/armv6/loader/fakeloader.c
loader: allow 1MB device maps
There's no reason we shouldn't allow 1MB PTEs for use on device memory.
loader: map as much as possible using 1MB pages
Chances are that we never actually executed this bit of code since all the
maps we ever deal with are either very short or much larger than 1MB.
unix: enable caches in locore
The loader should really be as simple as possible to be as small as
possible.  It should configure the machine so that unix can make certain
assumptions but it should leave more complex initialization to unix.
**********************************************************
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright (c) 2014 Joyent, Inc.  All rights reserved.
 * Copyright (c) 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
 */

#include "fakeloader.h"

#include <sys/types.h>
#include <sys/param.h>
#include <sys/elf.h>
#include <sys/atag.h>
#include <sys/sysmacros.h>
#include <sys/machparam.h>

#include <vm/pte.h>

/*
 * This is the stock ARM fake uniboot loader.
 *
 * Here's what we have to do:
 *   o Read the atag header and find the combined archive header
 *   o Determine the set of mappings we need to add for the following:
 *              - unix
 *              - boot_archive
 *              - atags
 *   o Enable unaligned access
 *   o Enable virtual memory
 *   o Enable the caches + virtual memory
 *
 * There are several important constraints that we have here:
 *
 *   o We cannot use any .data! Several loaders that come before us are broken
 *     and only provide us with the ability to map our .text and potentially our
 *     .bss. We should strive to avoid even that if we can.
 */

#ifdef  DEBUG
#define FAKELOAD_DPRINTF(x)     fakeload_puts(x)
#else
#define FAKELOAD_DPRINTF(x)
#endif  /* DEBUG */
```

```
/*
 * XXX ASSUMES WE HAVE Free memory following the boot archive
 */
static uintptr_t freemem;
static uintptr_t pt_arena;
static uintptr_t pt_arena_max;
static uint32_t *pt_addr;
static int nl2pages;

/* Simple copy routines */
void
bcopy(const void *s, void *d, size_t n)
{
        const char *src = s;
        char *dest = d;

        if (n == 0 || s == d)
                return;

        if (dest < src && dest + n < src) {
                /* dest overlaps with the start of src, copy forward */
                for (; n > 0; n--, src++, dest++)
                        *dest = *src;
        } else {
                /* src overlaps with start of dest or no overlap, copy rev */
                src += n - 1;
                dest += n - 1;
                for (; n > 0; n--, src--, dest--)
                        *dest = *src;
        }
}
______unchanged_portion_omitted_

static void
fakeload_map_1mb(uintptr_t pa, uintptr_t va, int prot)
{
        int entry;
        armpte_t *pte;
        arm_l1s_t *l1e;

        entry = ARMPT_VADDR_TO_L1E(va);
        pte = &pt_addr[entry];
        if (ARMPT_L1E_ISVALID(*pte))
                fakeload_panic("armboot_mmu: asked to map a mapped region!\n");
        l1e = (arm_l1s_t *)pte;
        *pte = 0;
        l1e->al_type = ARMPT_L1_TYPE_SECT;

        if (prot & PF_DEVICE) {
                l1e->al_bbit = 1;
                l1e->al_cbit = 0;
                l1e->al_tex = 0;
                l1e->al_sbit = 1;
        } else {
        /* Assume it's not device memory */
                l1e->al_bbit = 1;
                l1e->al_cbit = 1;
                l1e->al_tex = 1;
                l1e->al_sbit = 1;
        }
#endif /* ! codereview */

        if (!(prot & PF_X))
                l1e->al_xn = 1;
        l1e->al_domain = 0;
	if (prot & PF_W) {
		l1e->al_ap2 = 1;
		l1e->al_ap = 1;
	} else {
		l1e->al_ap2 = 0;
		l1e->al_ap = 1;
	}
	l1e->al_ngbit = 0;
	l1e->al_issuper = 0;
	l1e->al_addr = ARMPT_PADDR_TO_L1SECT(pa);
}

/*
 * Set freemem to be 1 MB aligned at the end of boot archive. While the L1 Page
 * table only needs to be 16 KB aligned, we opt for 1 MB alignment so that way
 * we can map it and all the other L2 page tables we might need. If we don’t do
 * this, it’ll become problematic for unix to actually modify this.
 */
static void
fakeload_pt_arena_init(const atag_initrd_t *aii)
{
	int entry, i;
	armpte_t *pte;
	arm_l1s_t *l1e;

	pt_arena = aii->ai_start + aii->ai_size;
	if (pt_arena & MMU_PAGEOFFSET1M) {
		pt_arena &= MMU_PAGEMASK1M;
		pt_arena += MMU_PAGESIZE1M;
	}
	pt_arena_max = pt_arena + 4 * MMU_PAGESIZE1M;
	freemem = pt_arena_max;

	/* Set up the l1 page table by first invalidating it */
	pt_addr = (armpte_t *)pt_arena;
	pt_arena += ARMPT_L1_SIZE;
	bzero(pt_addr, ARMPT_L1_SIZE);
	for (i = 0; i < 4; i++)
		fakeload_map_1mb((uintptr_t)pt_addr + i * MMU_PAGESIZE1M,
		    (uintptr_t)pt_addr + i * MMU_PAGESIZE1M,
		    PF_R | PF_W);
}

/*
 * This is our generally entry point. We’re passed in the entry point of the
 * header.
 */
static uintptr_t
fakeload_archive_mappings(atag_header_t *chain, const void *addr,
    atag_illumos_status_t *aisp)
{
	atag_illumos_mapping_t aim;
	fakeloader_hdr_t *hdr;
	Elf32_Ehdr *ehdr;
	Elf32_Phdr *phdr;
	int nhdrs, i;
	uintptr_t ret;
	uintptr_t text = 0, data = 0;
	size_t textln = 0, dataln = 0;

	hdr = (fakeloader_hdr_t *)addr;

	if (hdr->fh_magic[0] != FH_MAGIC0)
		fakeload_panic("fh_magic[0] is wrong!\n");
	if (hdr->fh_magic[1] != FH_MAGIC1)
		fakeload_panic("fh_magic[1] is wrong!\n");
```

```
	if (hdr->fh_magic[2] != FH_MAGIC2)
		fakeload_panic("fh_magic[2] is wrong!\n");
	if (hdr->fh_magic[3] != FH_MAGIC3)
		fakeload_panic("fh_magic[3] is wrong!\n");

	if (hdr->fh_unix_size == 0)
		fakeload_panic("hdr unix size is zero\n");
	if (hdr->fh_unix_offset == 0)
		fakeload_panic("hdr unix offset is zero\n");
	if (hdr->fh_archive_size == 0)
		fakeload_panic("hdr archive size is zero\n");
	if (hdr->fh_archive_offset == 0)
		fakeload_panic("hdr archive_offset is zero\n");

	ehdr = (Elf32_Ehdr *)((uintptr_t)addr + hdr->fh_unix_offset);

	if (ehdr->e_ident[EI_MAG0] != ELFMAG0)
		fakeload_panic("magic[0] wrong");
	if (ehdr->e_ident[EI_MAG1] != ELFMAG1)
		fakeload_panic("magic[1] wrong");
	if (ehdr->e_ident[EI_MAG2] != ELFMAG2)
		fakeload_panic("magic[2] wrong");
	if (ehdr->e_ident[EI_MAG3] != ELFMAG3)
		fakeload_panic("magic[3] wrong");
	if (ehdr->e_ident[EI_CLASS] != ELFCLASS32)
		fakeload_panic("wrong elfclass");
	if (ehdr->e_ident[EI_DATA] != ELFDATA2LSB)
		fakeload_panic("wrong encoding");
	if (ehdr->e_ident[EI_OSABI] != ELFOSABI_SOLARIS)
		fakeload_panic("wrong os abi");
	if (ehdr->e_ident[EI_ABIVERSION] != EAV_SUNW_CURRENT)
		fakeload_panic("wrong abi version");
	if (ehdr->e_type != ET_EXEC)
		fakeload_panic("unix is not an executable");
	if (ehdr->e_machine != EM_ARM)
		fakeload_panic("unix is not an ARM Executible");
	if (ehdr->e_version != EV_CURRENT)
		fakeload_panic("wrong version");
	if (ehdr->e_phnum == 0)
		fakeload_panic("no program headers");
	ret = ehdr->e_entry;

	FAKELOAD_DPRINTF("validated unix’s headers\n");

	nhdrs = ehdr->e_phnum;
	phdr = (Elf32_Phdr *)((uintptr_t)addr + hdr->fh_unix_offset +
	    ehdr->e_phoff);
	for (i = 0; i < nhdrs; i++, phdr++) {
		if (phdr->p_type != PT_LOAD) {
			fakeload_puts("skipping non-PT_LOAD header\n");
			continue;
		}

		if (phdr->p_filesz == 0 || phdr->p_memsz == 0) {
			fakeload_puts("skipping PT_LOAD with 0 file/mem\n");
			continue;
		}

		/*
		 * Create a mapping record for this in the atags.
		 */
		aim.aim_header.ah_size = ATAG_ILLUMOS_MAPPING_SIZE;
		aim.aim_header.ah_tag = ATAG_ILLUMOS_MAPPING;
		aim.aim_paddr = (uintptr_t)addr + hdr->fh_unix_offset +
		    phdr->p_offset;
		aim.aim_plen = phdr->p_filesz;
		aim.aim_vaddr = phdr->p_vaddr;
		aim.aim_vlen = phdr->p_memsz;
		/* Round up vlen to be a multiple of 4k */
		if (aim.aim_vlen & 0xfff) {
			aim.aim_vlen &= ~0xfff;
			aim.aim_vlen += 0x1000;
		}
		aim.aim_mapflags = phdr->p_flags;
		atag_append(chain, &aim.aim_header);

		/*
		 * When built with highvecs we need to account for the fact that
		 * _edata, _etext and _end are built assuming that the highvecs
		 * are normally part of our segments. ld is not doing anything
		 * wrong, but this breaks the assumptions that krtld currently
		 * has. As such, unix will use this information to overwrite the
		 * normal entry points that krtld uses in a similar style to
		 * SPARC.
		 */
		if (aim.aim_vaddr != 0xffff0000) {
			if ((phdr->p_flags & PF_W) != 0) {
				data = aim.aim_vaddr;
				dataln = aim.aim_vlen;
			} else {
				text = aim.aim_vaddr;
				textln = aim.aim_vlen;
			}
		}
	}

	aisp->ais_stext = text;
	aisp->ais_etext = text + textln;
	aisp->ais_sdata = data;
	aisp->ais_edata = data + dataln;

	/* 1:1 map the boot archive */
	aim.aim_header.ah_size = ATAG_ILLUMOS_MAPPING_SIZE;
	aim.aim_header.ah_tag = ATAG_ILLUMOS_MAPPING;
	aim.aim_paddr = (uintptr_t)addr + hdr->fh_archive_offset;
	aim.aim_plen = hdr->fh_archive_size;
	aim.aim_vaddr = aim.aim_paddr;
	aim.aim_vlen = aim.aim_plen;
	aim.aim_mapflags = PF_R | PF_W | PF_X;
	atag_append(chain, &aim.aim_header);
	aisp->ais_archive = aim.aim_paddr;
	aisp->ais_archivelen = aim.aim_plen;

	return (ret);
}

static void
fakeload_mkatags(atag_header_t *chain)
{
	atag_illumos_status_t ais;
	atag_illumos_mapping_t aim;

	bzero(&ais, sizeof (ais));
	bzero(&aim, sizeof (aim));

	ais.ais_header.ah_size = ATAG_ILLUMOS_STATUS_SIZE;
	ais.ais_header.ah_tag = ATAG_ILLUMOS_STATUS;
	atag_append(chain, &ais.ais_header);
	aim.aim_header.ah_size = ATAG_ILLUMOS_MAPPING_SIZE;
	aim.aim_header.ah_tag = ATAG_ILLUMOS_MAPPING;
	atag_append(chain, &aim.aim_header);
}
```

```
static uintptr_t
fakeload_alloc_l2pt(void)
{
	uintptr_t ret;

	if (pt_arena & ARMPT_L2_MASK) {
		ret = pt_arena;
		ret &= ~ARMPT_L2_MASK;
		ret += ARMPT_L2_SIZE;
		pt_arena = ret + ARMPT_L2_SIZE;
	} else {
		ret = pt_arena;
		pt_arena = ret + ARMPT_L2_SIZE;
	}
	if (pt_arena >= pt_arena_max) {
		fakeload_puts("pt_arena, max\n");
		fakeload_ultostr(pt_arena);
		fakeload_puts("\n");
		fakeload_ultostr(pt_arena_max);
		fakeload_puts("\n");
		fakeload_puts("l2pts alloced\n");
		fakeload_ultostr(nl2pages);
		fakeload_puts("\n");
		fakeload_panic("ran out of page tables!");
	}

	bzero((void *)ret, ARMPT_L2_SIZE);
	nl2pages++;
	return (ret);
}

/*
 * Finally, do all the dirty work. Let's create some page tables. The L1 page
 * table is full of 1 MB mappings by default. The L2 Page table is 1k in size
 * and covers that 1 MB. We're going to always create L2 page tables for now
 * which will use 4k and 64k pages.
 */
static void
fakeload_map(armpte_t *pt, uintptr_t pstart, uintptr_t vstart, size_t len,
    uint32_t prot)
{
	int entry, chunksize;
	armpte_t *pte, *l2pt;
	arm_l1pt_t *l1pt;

	/*
	 * Make sure both pstart + vstart are 4k aligned, along with len.
	 */
	if (pstart & MMU_PAGEOFFSET)
		fakeload_panic("pstart is not 4k aligned");
	if (vstart & MMU_PAGEOFFSET)
		fakeload_panic("vstart is not 4k aligned");
	if (len & MMU_PAGEOFFSET)
		fakeload_panic("len is not 4k aligned");

	/*
	 * We're going to logically deal with each 1 MB chunk at a time.
	 */
	while (len > 0) {
		if (vstart & MMU_PAGEOFFSET1M) {
			chunksize = MIN(len, MMU_PAGESIZE1M -
			    (vstart & MMU_PAGEOFFSET1M));
		} else {
			chunksize = MIN(len, MMU_PAGESIZE1M);
		}
```

```
                entry = ARMPT_VADDR_TO_L1E(vstart);
                pte = &pt[entry];

                if (!ARMPT_L1E_ISVALID(*pte)) {
                        uintptr_t l2table;

                        if (!(vstart & MMU_PAGEOFFSET1M) &&
                            !(pstart & MMU_PAGEOFFSET1M) &&
                            len >= MMU_PAGESIZE1M) {
                            len == MMU_PAGESIZE1M) {
                                fakeload_map_1mb(pstart, vstart, prot);
                                vstart += MMU_PAGESIZE1M;
                                pstart += MMU_PAGESIZE1M;
                                len -= MMU_PAGESIZE1M;
                                continue;
                        }

                        l2table = fakeload_alloc_l2pt();
                        *pte = 0;
                        l1pt = (arm_l1pt_t *)pte;
                        l1pt->al_type = ARMPT_L1_TYPE_L2PT;
                        l1pt->al_ptaddr = ARMPT_ADDR_TO_L1PTADDR(l2table);
                } else if ((*pte & ARMPT_L1_TYPE_MASK) != ARMPT_L1_TYPE_L2PT) {
                        fakeload_panic("encountered l1 entry that’s not a "
                            "pointer to a level 2 table\n");
                } else {
                        l1pt = (arm_l1pt_t *)pte;
                }

                /* Now that we have the l1pt fill in l2 entries */
                l2pt = (void *)(l1pt->al_ptaddr << ARMPT_L1PT_TO_L2_SHIFT);
                len -= chunksize;
                while (chunksize > 0) {
                        arm_l2e_t *l2pte;

                        entry = ARMPT_VADDR_TO_L2E(vstart);
                        pte = &l2pt[entry];

#ifdef  MAP_DEBUG
                        fakeload_puts("4k page pa->va, l2root, entry\n");
                        fakeload_ultostr(pstart);
                        fakeload_puts("->");
                        fakeload_ultostr(vstart);
                        fakeload_puts(", ");
                        fakeload_ultostr((uintptr_t)l2pt);
                        fakeload_puts(", ");
                        fakeload_ultostr(entry);
                        fakeload_puts("\n");
#endif

                        if ((*pte & ARMPT_L2_TYPE_MASK) !=
                            ARMPT_L2_TYPE_INVALID)
                                fakeload_panic("found existing l2 page table, "
                                    "overlap in requested mappings detected!");
                        /* Map vaddr to our paddr! */
                        l2pte = ((arm_l2e_t *)pte);
                        *pte = 0;
                        if (!(prot & PF_X))
                                l2pte->ale_xn = 1;
                        l2pte->ale_ident = 1;
                        if (prot & PF_DEVICE) {
                                l2pte->ale_bbit = 1;
                                l2pte->ale_cbit = 0;
                                l2pte->ale_tex = 0;
                                l2pte->ale_sbit = 1;
```

```
                        } else {
                                l2pte->ale_bbit = 1;
                                l2pte->ale_cbit = 1;
                                l2pte->ale_tex = 1;
                                l2pte->ale_sbit = 1;
                        }
                        if (prot & PF_W) {
                                l2pte->ale_ap2 = 1;
                                l2pte->ale_ap = 1;
                        } else {
                                l2pte->ale_ap2 = 0;
                                l2pte->ale_ap = 1;
                        }
                        l2pte->ale_ngbit = 0;
                        l2pte->ale_addr = ARMPT_PADDR_TO_L2ADDR(pstart);

                        chunksize -= MMU_PAGESIZE;
                        vstart += MMU_PAGESIZE;
                        pstart += MMU_PAGESIZE;
                }
        }
}
```

______***unchanged_portion_omitted_***

```
void
fakeload_init(void *ident, void *ident2, void *atag)
{
        atag_header_t *hdr;
        atag_header_t *chain = (atag_header_t *)atag;
        const atag_initrd_t *initrd;
        atag_illumos_status_t *aisp;
        atag_illumos_mapping_t *aimp;
        uintptr_t unix_start;

        fakeload_backend_init();
        fakeload_puts("Hello from the loader\n");
        initrd = (atag_initrd_t *)atag_find(chain, ATAG_INITRD2);
        if (initrd == NULL)
                fakeload_panic("missing the initial ramdisk\n");

        /*
         * Create the status atag header and the initial mapping record for the
         * atags. We’ll hold onto both of these.
         */
        fakeload_mkatags(chain);
        aisp = (atag_illumos_status_t *)atag_find(chain, ATAG_ILLUMOS_STATUS);
        if (aisp == NULL)
                fakeload_panic("can’t find ATAG_ILLUMOS_STATUS");
        aimp = (atag_illumos_mapping_t *)atag_find(chain, ATAG_ILLUMOS_MAPPING);
        if (aimp == NULL)
                fakeload_panic("can’t find ATAG_ILLUMOS_MAPPING");
        FAKELOAD_DPRINTF("created proto atags\n");

        fakeload_pt_arena_init(initrd);

        fakeload_selfmap(chain);

        /*
         * Map the boot archive and all of unix
         */
        unix_start = fakeload_archive_mappings(chain,
            (const void *)(uintptr_t)initrd->ai_start, aisp);
        FAKELOAD_DPRINTF("filled out unix and the archive’s mappings\n");

        /*
         * Fill in the atag mapping header for the atags themselves. 1:1 map it.
         */
        aimp->aim_paddr = (uintptr_t)chain & ~0xfff;
        aimp->aim_plen = atag_length(chain) & ~0xfff;
        aimp->aim_plen += 0x1000;
        aimp->aim_vaddr = aimp->aim_paddr;
        aimp->aim_vlen = aimp->aim_plen;
        aimp->aim_mapflags = PF_R | PF_W | PF_NORELOC;

        /*
         * Let the backend add mappings
         */
        fakeload_backend_addmaps(chain);

        /*
         * Turn on unaligned access
         */
        FAKELOAD_DPRINTF("turning on unaligned access\n");
        fakeload_unaligned_enable();
        FAKELOAD_DPRINTF("successfully enabled unaligned access\n");

        /*
         * To turn on the MMU we need to do the following:
         *  o Program all relevant CP15 registers
         *  o Program 1st and 2nd level page tables
         *  o Invalidate and Disable the I/D-cache
         *  o Fill in the last bits of the ATAG_ILLUMOS_STATUS atag
         *  o Turn on the MMU in SCTLR
         *  o Jump to unix
         */

        /* Last bits of the atag */
        aisp->ais_freemem = freemem;
        aisp->ais_version = 1;
        aisp->ais_ptbase = (uintptr_t)pt_addr;

        /*
         * Our initial page table is a series of 1 MB sections. While we really
         * should map 4k pages, for the moment we’re just going to map 1 MB
         * regions, yay team!
         */
        hdr = chain;
        FAKELOAD_DPRINTF("creating mappings\n");
        while (hdr != NULL) {
                if (hdr->ah_tag == ATAG_ILLUMOS_MAPPING)
                        fakeload_create_map(pt_addr,
                            (atag_illumos_mapping_t *)hdr);
                hdr = atag_next(hdr);
        }

        /*
         * Now that we’ve mapped everything, update the status atag.
         */
        aisp->ais_freeused = freemem - aisp->ais_freemem;
        aisp->ais_pt_arena = pt_arena;
        aisp->ais_pt_arena_max = pt_arena_max;

        /* Cache disable */
        FAKELOAD_DPRINTF("Flushing and disabling caches\n");
        armv6_dcache_flush();
        armv6_dcache_disable();
        armv6_dcache_inval();
        armv6_icache_disable();
        armv6_icache_inval();

        /* Program the page tables */
        FAKELOAD_DPRINTF("programming cp15 regs\n");
```

```
        fakeload_pt_setup((uintptr_t)pt_addr);


        /* MMU Enable */
        FAKELOAD_DPRINTF("see you on the other side\n");
        fakeload_mmu_enable();

        FAKELOAD_DPRINTF("why helo thar\n");

        /* Renable caches */
        armv6_dcache_enable();
        armv6_icache_enable();

        /* we should never come back */
        fakeload_exec(ident, ident2, chain, unix_start);
        fakeload_panic("hit the end of the world\n");
}
```

______unchanged_portion_omitted_

****
2607 Thu Feb 12 20:32:17 2015
new/usr/src/uts/armv6/loader/fakeloader_core.s
loader: document the page table setup
loader: simplify MMU enabling code
****
______***unchanged_portion_omitted_***

```
#endif  /* __lint */

#if defined(__lint)

fakeload_pt_setup(uintptr_t ptroot)
{}

#else /* __lint */

        /*
         * We need to set up the world for the first time. We’ll do the
         * following in order:
         *
         * o Set the TTBCR to always use TTBR0
         * o Set domain 0 to manager mode
         * o Program the Page table root
         */
        ENTRY(fakeload_pt_setup)
        /* use TTBR0 only (should already be true) */
#endif /* ! codereview */
        mov     r1, #0
        mcr     p15, 0, r1, c2, c0, 2

        /* set domain 0 to manager mode */
#endif /* ! codereview */
        mov     r1, #3
        mcr     p15, 0, r1, c3, c0, 0

        /* set TTBR0 to page table root */
        orr     r0, r0, #0x18           /* Outer WB, no WA Cachable */
        orr     r0, r0, #0x2            /* Sharable */
        orr     r0, r0, #0x1            /* Inner Cachable */
        orr     r0, r0, #0x1b
        mcr     p15, 0, r0, c2, c0, 0
        bx      lr
        SET_SIZE(fakeload_pt_setup)

#endif /* __lint */

#if defined(__lint)

/* ARGSUSED */
void
fakeload_mmu_enable(void)
{}

#else   /* __lint */

        /*
         * We first make sure that the ARMv6 pages are enabled (bit 23) and then
         * enable the MMU (bit 0).
         */
        ENTRY(fakeload_mmu_enable)
        mrc     p15, 0, r0, c1, c0, 0
        orr     r0, #0x800000           /* enable ARMv6 pages */
        orr     r0, #0x1                /* enable MMU */
        orr     r0, #0x800000
        mcr     p15, 0, r0, c1, c0, 0
```

```
        mrc     p15, 0, r0, c1, c0, 0
        orr     r0, #0x1
        mcr     p15, 0, r0, c1, c0, 0
        bx      lr
        SET_SIZE(fakeload_mmu_enable)
```
______***unchanged_portion_omitted_***

```
**********************************************************
    5444 Thu Feb 12 20:32:17 2015
new/usr/src/uts/armv6/ml/cache.s
armv6: p15 cache functions say that value passed in should be zero
**********************************************************
______unchanged_portion_omitted_

        ENTRY(armv6_icache_inval)
        mov     r0, #0
#endif /* ! codereview */
        mcr     p15, 0, r0, c7, c5, 0           @ Invalidate i-cache
        bx      lr
        SET_SIZE(armv6_icache_inval)

        ENTRY(armv6_dcache_inval)
        mov     r0, #0
#endif /* ! codereview */
        mcr     p15, 0, r0, c7, c6, 0           @ Invalidate d-cache
        ARM_DSB_INSTR(r2)
        bx      lr
        SET_SIZE(armv6_dcache_inval)

        ENTRY(armv6_dcache_flush)
        mov     r0, #0
#endif /* ! codereview */
        mcr     p15, 0, r0, c7, c10, 4          @ Flush d-cache
        ARM_DSB_INSTR(r2)
        bx      lr
        SET_SIZE(armv6_dcache_flush)

        ENTRY(armv6_text_flush_range)
        add     r1, r1, r0
        sub     r1, r1, r0
        mcrr    p15, 0, r1, r0, c5              @ Invalidate i-cache range
        mcrr    p15, 0, r1, r0, c12             @ Flush d-cache range
        ARM_DSB_INSTR(r2)
        ARM_ISB_INSTR(r2)
        bx      lr
        SET_SIZE(armv6_text_flush_range)

        ENTRY(armv6_text_flush)
        mov     r0, #0
#endif /* ! codereview */
        mcr     p15, 0, r0, c7, c5, 0           @ Invalidate i-cache
        mcr     p15, 0, r0, c7, c10, 4          @ Flush d-cache
        ARM_DSB_INSTR(r2)
        ARM_ISB_INSTR(r2)
        bx      lr
        SET_SIZE(armv6_text_flush)

#endif

#ifdef __lint

/*
 * Perform all of the operations necessary for tlb maintenance after an update
 * to the page tables.
 */
void
armv6_tlb_sync(void)
{}

#else   /* __lint */

        ENTRY(armv6_tlb_sync)
        mov     r0, #0
```

```
        mcr     p15, 0, r0, c7, c10, 4          @ Flush d-cache
        ARM_DSB_INSTR(r0)
        mcr     p15, 0, r0, c8, c7, 0           @ invalidate tlb
        mcr     p15, 0, r0, c8, c5, 0           @ Invalidate I-cache + btc
        ARM_DSB_INSTR(r0)
        ARM_ISB_INSTR(r0)
        bx      lr
        SET_SIZE(armv6_tlb_sync)

#endif  /* __lint */
```

```
**********************************************************
    5123 Thu Feb 12 20:32:17 2015
new/usr/src/uts/armv6/ml/glocore.s
unix: enable caches in locore
The loader should really be as simple as possible to be as small as
possible.  It should configure the machine so that unix can make certain
assumptions but it should leave more complex initialization to unix.
**********************************************************
/*
 * This file and its contents are supplied under the terms of the
 * Common Development and Distribution License ("CDDL"), version 1.0.
 * You may only use this file in accordance with the terms of version
 * 1.0 of the CDDL.
 *
 * A full copy of the text of the CDDL should have accompanied this
 * source.  A copy of the CDDL is also available via the Internet at
 * http://www.illumos.org/license/CDDL.
 */

/*
 * Copyright 2013 (c) Joyent, Inc. All rights reserved.
 * Copyright (c) 2015 Josef 'Jeff' Sipek <jeffpc@josefsipek.net>
 */

#include <sys/asm_linkage.h>
#include <sys/machparam.h>
#include <sys/cpu_asm.h>

#include "assym.h"

/*
 * Every story needs a beginning. This is ours.
 */

/*
 * Each of the different machines has its own locore.s to take care of getting
 * the machine specific setup done.  Just before jumping into fakebop the
 * first time, we call this machine specific code.
 */

/*
 * We are in a primordial world here. The loader is going to come along and
 * boot us at _start. As we've started the world, we also need to set up a
 * few things about us, for example our stack pointer. To help us out, it's
 * useful to remember what the loader set up for us:
 *
 * - unaligned access are allowed (A = 0, U = 1)
 * - virtual memory is enabled
 *   - we (unix) are mapped right were we want to be
 *   - a UART has been enabled & any memory mapped registers have been 1:1
 *     mapped
 *   - ATAGs have been updated to tell us what the mappings are
 * - I/D L1 caches have may be disabled
 * - I/D L1 caches have been enabled
 */

        /*
         * External globals
         */
        .globl  _locore_start
        .globl  mlsetup
        .globl  sysp
        .globl  bootops
        .globl  bootopsp
        .globl  t0
```

```
        .data
        .comm   t0stack, DEFAULTSTKSZ, 32
        .comm   t0, 4094, 32


/*
 * Recall that _start is the traditional entry point for an ELF binary.
 */
        ENTRY(_start)
        ldr     sp, =t0stack
        ldr     r4, =DEFAULTSTKSZ
        add     sp, r4
        bic     sp, sp, #0xff

        /*
         * establish bogus stacks for exceptional CPU states, our exception
         * code should never make use of these, and we want loud and violent
         * failure should we accidentally try.
         */
        cps     #(CPU_MODE_UND)
        mov     sp, #-1
        cps     #(CPU_MODE_ABT)
        mov     sp, #-1
        cps     #(CPU_MODE_FIQ)
        mov     sp, #-1
        cps     #(CPU_MODE_IRQ)
        mov     sp, #-1
        cps     #(CPU_MODE_SVC)

        /* Enable highvecs (moves the base of the exception vector) */
        mrc     p15, 0, r3, c1, c0, 0
        orr     r3, r3, #(1 << 13)
        mcr     p15, 0, r3, c1, c0, 0

        /*
         * Go ahead now and enable the L1 I/D caches.  (Involves
         * invalidating the caches and the TLB.)
         */
        mov     r4, #0
        mov     r5, #0
        mcr     p15, 0, r4, c7, c7, 0   /* invalidate caches */
        mcr     p15, 0, r4, c8, c7, 0   /* invalidate tlb */
        mcr     p15, 0, r5, c7, c10, 4  /* DSB */
        mrc     p15, 0, r4, c1, c0, 0
        orr     r4, #0x04       /* D-cache */
        orr     r4, #0x1000     /* I-cache */
        mcr     p15, 0, r4, c1, c0, 0

#endif /* ! codereview */
        /* invoke machine specific setup */
        bl      _mach_start

        bl      _fakebop_start
        SET_SIZE(_start)


#if defined(__lint)

/* ARGSUSED */
void
_locore_start(struct boot_syscalls *sysp, struct bootops *bop)
{}

#else   /* __lint */

        /*
 **We got here from _kobj_init() via exitto().  We have a few different**
 **tasks that we need to take care of before we hop into mlsetup and**
 **then main. We're never going back so we shouldn't feel compelled to**
 **preserve any registers.**
 *
 *  o Enable our I/D-caches
 *  o Save the boot syscalls and bootops for later
 *  o Set up our stack to be the real stack of t0stack.
 *  o Save t0 as curthread
 *  o Set up a struct REGS for mlsetup
 *  o Make sure that we're 8 byte aligned for the call
 */

ENTRY(_locore_start)


/*
 * We've been running in t0stack anyway, up to this point, but
 * _locore_start represents what is in effect a fresh start in the
 * real kernel -- We'll never return back through here.
 *
 * So reclaim those few bytes
 */
ldr     sp, =t0stack
ldr     r4, =(DEFAULTSTKSZ - REGSIZE)
add     sp, r4
bic     sp, sp, #0xff

/*
 * Save flags and arguments for potential debugging
 */
str     r0, [sp, #REGOFF_R0]
str     r1, [sp, #REGOFF_R1]
str     r2, [sp, #REGOFF_R2]
str     r3, [sp, #REGOFF_R3]
mrs     r4, CPSR
str     r4, [sp, #REGOFF_CPSR]

/*
 * Save back the bootops and boot_syscalls.
 */
ldr     r2, =sysp
str     r0, [r2]
ldr     r2, =bootops
str     r1, [r2]
ldr     r2, =bootopsp
ldr     r2, [r2]
str     r1, [r2]

/*
 * Set up our curthread pointer
 */
ldr     r0, =t0
mcr     p15, 0, r0, c13, c0, 4

/*
 * Go ahead now and enable the L1 I/D caches.
 */
mrc     p15, 0, r0, c1, c0, 0
orr     r0, #0x04       /* D-cache */
orr     r0, #0x1000     /* I-cache */
mcr     p15, 0, r0, c1, c0, 0

/*
 * mlsetup() takes the struct regs as an argument. main doesn't take
 * any and should never return. Currently, we have an 8-byte aligned
```

```
 * stack.  We want to push a zero frame pointer to terminate any
 * stack walking, but that would cause us to end up with only a
 * 4-byte aligned stack.  So, to keep things nice and correct, we
 * push a zero value twice - it's similar to a typical function
 * entry:
 *      push { r9, lr }
 */
mov     r9,#0
push    { r9 }          /* link register */
push    { r9 }          /* frame pointer */
mov     r0, sp
bl      mlsetup
bl      main
/* NOTREACHED */
ldr     r0,=__return_from_main
ldr     r0,[r0]
bl      panic
SET_SIZE(_locore_start)
```
______***unchanged_portion_omitted_***