```
**********************************************************
   33052 Thu Apr 25 02:42:33 2019
new/usr/src/boot/sys/boot/common/multiboot2.c
10807 loader fails to boot Dell R510 in UEFI mode
Reviewed by: Robert Mustacchi <rm@joyent.com>
Reviewed by: Jerry Jelinek <jerry.jelinek@joyent.com>
Reviewed by: Toomas Soome <tsoome@me.com>
**********************************************************
```

```
/*
 * 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 2017 Toomas Soome <tsoome@me.com>
 * Copyright 2019, Joyent, Inc.
 */

/*
 * This module adds support for loading and booting illumos multiboot2
 * kernel. This code is only built to support the illumos kernel, it does
 * not support xen.
 */

#include <sys/cdefs.h>
#include <sys/stddef.h>

#include <sys/param.h>
#include <sys/exec.h>
#include <sys/linker.h>
#include <sys/module.h>
#include <sys/stdint.h>
#include <sys/multiboot2.h>
#include <stand.h>
#include <stdbool.h>
#include <machine/elf.h>
#include "libzfs.h"

#include "bootstrap.h"
#include <sys/consplat.h>

#include <machine/metadata.h>
#include <machine/pc/bios.h>

#define SUPPORT_DHCP
#include <bootp.h>

#if !defined(EFI)
#include "../i386/btx/lib/btxv86.h"
#include "libi386.h"
#include "vbe.h"

#else
#include <efi.h>
#include <efilib.h>
#include "loader_efi.h"

static void (*trampoline)(uint32_t, struct relocator *, uint64_t);
#endif
```

```
#include "platform/acfreebsd.h"
#include "acconfig.h"
#define ACPI_SYSTEM_XFACE
#include "actypes.h"
#include "actbl.h"

extern ACPI_TABLE_RSDP *rsdp;

/* MB data heap pointer. */
static vm_offset_t last_addr;

static int multiboot2_loadfile(char *, uint64_t, struct preloaded_file **);
static int multiboot2_exec(struct preloaded_file *);

struct file_format multiboot2 = { multiboot2_loadfile, multiboot2_exec };
static bool keep_bs = false;
static bool have_framebuffer = false;
static vm_offset_t load_addr;
static vm_offset_t entry_addr;

/*
 * Validate tags in info request. This function is provided just to
 * recognize the current tag list and only serves as a limited
 * safe guard against possibly corrupt information.
 */
static bool
is_info_request_valid(multiboot_header_tag_information_request_t *rtag)
{
        int i;

        /*
         * If the tag is optional and we do not support it, we do not
         * have to do anything special, so we skip optional tags.
         */
        if (rtag->mbh_flags & MULTIBOOT_HEADER_TAG_OPTIONAL)
                return (true);

        for (i = 0; i < (rtag->mbh_size - sizeof (*rtag)) /
            sizeof (rtag->mbh_requests[0]); i++)
                switch (rtag->mbh_requests[i]) {
                case MULTIBOOT_TAG_TYPE_END:
                case MULTIBOOT_TAG_TYPE_CMDLINE:
                case MULTIBOOT_TAG_TYPE_BOOT_LOADER_NAME:
                case MULTIBOOT_TAG_TYPE_MODULE:
                case MULTIBOOT_TAG_TYPE_BASIC_MEMINFO:
                case MULTIBOOT_TAG_TYPE_BOOTDEV:
                case MULTIBOOT_TAG_TYPE_MMAP:
                case MULTIBOOT_TAG_TYPE_FRAMEBUFFER:
                case MULTIBOOT_TAG_TYPE_VBE:
                case MULTIBOOT_TAG_TYPE_ELF_SECTIONS:
                case MULTIBOOT_TAG_TYPE_APM:
                case MULTIBOOT_TAG_TYPE_EFI32:
                case MULTIBOOT_TAG_TYPE_EFI64:
                case MULTIBOOT_TAG_TYPE_ACPI_OLD:
                case MULTIBOOT_TAG_TYPE_ACPI_NEW:
                case MULTIBOOT_TAG_TYPE_NETWORK:
                case MULTIBOOT_TAG_TYPE_EFI_MMAP:
                case MULTIBOOT_TAG_TYPE_EFI_BS:
                case MULTIBOOT_TAG_TYPE_EFI32_IH:
                case MULTIBOOT_TAG_TYPE_EFI64_IH:
                case MULTIBOOT_TAG_TYPE_LOAD_BASE_ADDR:
                        break;
                default:
                        printf("unsupported information tag: 0x%x\n",
                            rtag->mbh_requests[i]);
                        return (false);
                }
        return (true);
}
```

______unchanged_portion_omitted_

```
#if defined(EFI)
static bool
overlaps(uintptr_t start1, size_t size1, uintptr_t start2, size_t size2)
{
        if (start1 < start2 + size2 &&
            start1 + size1 >= start2) {
                printf("overlaps: %zx-%zx, %zx-%zx\n",
                    start1, start1 + size1, start2, start2 + size2);
                return (true);
        }

        return (false);
}
#endif

static int
multiboot2_exec(struct preloaded_file *fp)
{
        multiboot2_info_header_t *mbi = NULL;
        struct preloaded_file *mfp;
        multiboot2_info_header_t *mbi;
        char *cmdline = NULL;
        struct devdesc *rootdev;
        struct file_metadata *md;
        int i, error, num;
        int rootfs = 0;
        size_t size;
        struct bios_smap *smap;
#if defined(EFI)
        multiboot_tag_module_t *module, *mp;
        struct relocator *relocator = NULL;
        EFI_MEMORY_DESCRIPTOR *map;
        UINTN map_size, desc_size;
        struct relocator *relocator;
        struct chunk_head *head;
        struct chunk *chunk;
        vm_offset_t tmp;

        efi_getdev((void **)(&rootdev), NULL, NULL);

        /*
         * We need 5 pages for relocation. We’ll allocate from the heap: while
         * it’s possible that our heap got placed low down enough to be in the
         * way of where we’re going to relocate our kernel, it’s hopefully not
         * likely.
         */
        if ((relocator = malloc(EFI_PAGE_SIZE * 5)) == NULL) {
                printf("relocator malloc failed!\n");
                error = ENOMEM;
                goto error;
        }

        if (overlaps((uintptr_t)relocator, EFI_PAGE_SIZE * 5,
            load_addr, fp->f_size)) {
                printf("relocator pages overlap the kernel!\n");
                error = EINVAL;
                goto error;
        }

#else
        i386_getdev((void **)(&rootdev), NULL, NULL);
```

```
        if (have_framebuffer == false) {
                /* make sure we have text mode */
                bios_set_text_mode(VGA_TEXT_MODE);
        }
#endif

        mbi = NULL;
        error = EINVAL;
        if (rootdev == NULL) {
                printf("can’t determine root device\n");
                goto error;
        }

        /*
         * Set the image command line.
         */
        if (fp->f_args == NULL) {
                cmdline = getenv("boot-args");
                if (cmdline != NULL) {
                        fp->f_args = strdup(cmdline);
                        if (fp->f_args == NULL) {
                                error = ENOMEM;
                                goto error;
                        }
                }
        }

        error = mb_kernel_cmdline(fp, rootdev, &cmdline);
        if (error != 0)
                goto error;

        /* mb_kernel_cmdline() updates the environment. */
        build_environment_module();

        if (have_framebuffer == true) {
                /* Pass the loaded console font for kernel. */
                build_font_module();
        }

        size = mbi_size(fp, cmdline);   /* Get the size for MBI. */

        /* Set up the base for mb_malloc. */
        i = 0;
        for (mfp = fp; mfp->f_next != NULL; mfp = mfp->f_next)
                i++;

#if defined(EFI)
        /* We need space for kernel + MBI + # modules */
        num = (EFI_PAGE_SIZE - offsetof(struct relocator, rel_chunklist)) /
            sizeof (struct chunk);
        if (i + 2 >= num) {
                printf("Too many modules, do not have space for relocator.\n");
                error = ENOMEM;
                goto error;
        }

        last_addr = efi_loadaddr(LOAD_MEM, &size, mfp->f_addr + mfp->f_size);
        mbi = (multiboot2_info_header_t *)last_addr;
        if (mbi == NULL) {
                error = ENOMEM;
                goto error;
        }
        last_addr = (vm_offset_t)mbi->mbi_tags;
#else
        /* Start info block from the new page. */
        last_addr = i386_loadaddr(LOAD_MEM, &size, mfp->f_addr + mfp->f_size);

        /* Do we have space for multiboot info? */
        if (last_addr + size >= memtop_copyin) {
                error = ENOMEM;
                goto error;
        }

        mbi = (multiboot2_info_header_t *)PTOV(last_addr);
        last_addr = (vm_offset_t)mbi->mbi_tags;
#endif  /* EFI */

        {
                multiboot_tag_string_t *tag;
                i = sizeof (multiboot_tag_string_t) + strlen(cmdline) + 1;
                tag = (multiboot_tag_string_t *)mb_malloc(i);

                tag->mb_type = MULTIBOOT_TAG_TYPE_CMDLINE;
                tag->mb_size = i;
                memcpy(tag->mb_string, cmdline, strlen(cmdline) + 1);
                free(cmdline);
                cmdline = NULL;
        }

        {
                multiboot_tag_string_t *tag;
                i = sizeof (multiboot_tag_string_t) + strlen(bootprog_info) + 1;
                tag = (multiboot_tag_string_t *)mb_malloc(i);

                tag->mb_type = MULTIBOOT_TAG_TYPE_BOOT_LOADER_NAME;
                tag->mb_size = i;
                memcpy(tag->mb_string, bootprog_info,
                    strlen(bootprog_info) + 1);
        }

#if !defined(EFI)
        /* Only set in case of BIOS. */
        {
                multiboot_tag_basic_meminfo_t *tag;
                tag = (multiboot_tag_basic_meminfo_t *)
                    mb_malloc(sizeof (*tag));

                tag->mb_type = MULTIBOOT_TAG_TYPE_BASIC_MEMINFO;
                tag->mb_size = sizeof (*tag);
                tag->mb_mem_lower = bios_basemem / 1024;
                tag->mb_mem_upper = bios_extmem / 1024;
        }
#endif

        num = 0;
        for (mfp = fp->f_next; mfp != NULL; mfp = mfp->f_next) {
                num++;
                if (mfp->f_type != NULL && strcmp(mfp->f_type, "rootfs") == 0)
                        rootfs++;
        }

        if (num == 0 || rootfs == 0) {
                /* We need at least one module - rootfs. */
                printf("No rootfs module provided, aborting\n");
                error = EINVAL;
                goto error;
        }

        /*
         * Set the stage for physical memory layout:
         * - We have kernel at load_addr.
```

```
         * - Modules are aligned to page boundary.
         * - MBI is aligned to page boundary.
         * - Set the tmp to point to physical address of the first module.
         * - tmp != mfp->f_addr only in case of EFI.
         */
#if defined(EFI)
        tmp = roundup2(load_addr + fp->f_size + 1, MULTIBOOT_MOD_ALIGN);
        module = (multiboot_tag_module_t *)last_addr;
#endif

        for (mfp = fp->f_next; mfp != NULL; mfp = mfp->f_next) {
                multiboot_tag_module_t *tag;

                num = strlen(mfp->f_name) + 1;
                num += strlen(mfp->f_type) + 5 + 1;
                if (mfp->f_args != NULL) {
                        num += strlen(mfp->f_args) + 1;
                }
                cmdline = malloc(num);
                if (cmdline == NULL) {
                        error = ENOMEM;
                        goto error;
                }

                if (mfp->f_args != NULL)
                        snprintf(cmdline, num, "%s type=%s %s",
                            mfp->f_name, mfp->f_type, mfp->f_args);
                else
                        snprintf(cmdline, num, "%s type=%s",
                            mfp->f_name, mfp->f_type);

                tag = (multiboot_tag_module_t *)mb_malloc(sizeof (*tag) + num);

                tag->mb_type = MULTIBOOT_TAG_TYPE_MODULE;
                tag->mb_size = sizeof (*tag) + num;
#if defined(EFI)
                /*
                 * We can assign module addresses only after BS have been
                 * switched off.
                 */
                tag->mb_mod_start = 0;
                tag->mb_mod_end = mfp->f_size;
#else
                tag->mb_mod_start = mfp->f_addr;
                tag->mb_mod_end = mfp->f_addr + mfp->f_size;
#endif
                memcpy(tag->mb_cmdline, cmdline, num);
                free(cmdline);
                cmdline = NULL;
        }

        md = file_findmetadata(fp, MODINFOMD_SMAP);
        if (md == NULL) {
                printf("no memory smap\n");
                error = EINVAL;
                goto error;
        }

        smap = (struct bios_smap *)md->md_data;
        num = md->md_size / sizeof (struct bios_smap); /* number of entries */

        {
                multiboot_tag_mmap_t *tag;
                multiboot_mmap_entry_t *mmap_entry;

                tag = (multiboot_tag_mmap_t *)
```

```
                    mb_malloc(sizeof (*tag) +
                    num * sizeof (multiboot_mmap_entry_t));

                tag->mb_type = MULTIBOOT_TAG_TYPE_MMAP;
                tag->mb_size = sizeof (*tag) +
                    num * sizeof (multiboot_mmap_entry_t);
                tag->mb_entry_size = sizeof (multiboot_mmap_entry_t);
                tag->mb_entry_version = 0;
                mmap_entry = (multiboot_mmap_entry_t *)tag->mb_entries;

                for (i = 0; i < num; i++) {
                        mmap_entry[i].mmap_addr = smap[i].base;
                        mmap_entry[i].mmap_len = smap[i].length;
                        mmap_entry[i].mmap_type = smap[i].type;
                        mmap_entry[i].mmap_reserved = 0;
                }
        }

        if (bootp_response != NULL) {
                multiboot_tag_network_t *tag;
                tag = (multiboot_tag_network_t *)
                    mb_malloc(sizeof (*tag) + bootp_response_size);

                tag->mb_type = MULTIBOOT_TAG_TYPE_NETWORK;
                tag->mb_size = sizeof (*tag) + bootp_response_size;
                memcpy(tag->mb_dhcpack, bootp_response, bootp_response_size);
        }

#if !defined(EFI)
        multiboot_tag_vbe_t *tag;
        extern multiboot_tag_vbe_t vbestate;

        if (VBE_VALID_MODE(vbestate.vbe_mode)) {
                tag = (multiboot_tag_vbe_t *)mb_malloc(sizeof (*tag));
                memcpy(tag, &vbestate, sizeof (*tag));
                tag->mb_type = MULTIBOOT_TAG_TYPE_VBE;
                tag->mb_size = sizeof (*tag);
        }
#endif

        if (rsdp != NULL) {
                multiboot_tag_new_acpi_t *ntag;
                multiboot_tag_old_acpi_t *otag;
                uint32_t tsize;

                if (rsdp->Revision == 0) {
                        tsize = sizeof (*otag) + sizeof (ACPI_RSDP_COMMON);
                        otag = (multiboot_tag_old_acpi_t *)mb_malloc(tsize);
                        otag->mb_type = MULTIBOOT_TAG_TYPE_ACPI_OLD;
                        otag->mb_size = tsize;
                        memcpy(otag->mb_rsdp, rsdp, sizeof (ACPI_RSDP_COMMON));
                } else {
                        tsize = sizeof (*ntag) + rsdp->Length;
                        ntag = (multiboot_tag_new_acpi_t *)mb_malloc(tsize);
                        ntag->mb_type = MULTIBOOT_TAG_TYPE_ACPI_NEW;
                        ntag->mb_size = tsize;
                        memcpy(ntag->mb_rsdp, rsdp, rsdp->Length);
                }
        }

#if defined(EFI)
#ifdef  __LP64__
        {
                multiboot_tag_efi64_t *tag;
                tag = (multiboot_tag_efi64_t *)
                    mb_malloc(sizeof (*tag));
```

```

                tag->mb_type = MULTIBOOT_TAG_TYPE_EFI64;
                tag->mb_size = sizeof (*tag);
                tag->mb_pointer = (uint64_t)(uintptr_t)ST;
        }
#else
        {
                multiboot_tag_efi32_t *tag;
                tag = (multiboot_tag_efi32_t *)
                    mb_malloc(sizeof (*tag));

                tag->mb_type = MULTIBOOT_TAG_TYPE_EFI32;
                tag->mb_size = sizeof (*tag);
                tag->mb_pointer = (uint32_t)ST;
        }
#endif /* __LP64__ */
#endif /* EFI */

        if (have_framebuffer == true) {
                multiboot_tag_framebuffer_t *tag;
                extern multiboot_tag_framebuffer_t gfx_fb;
#if defined(EFI)

                tag = (multiboot_tag_framebuffer_t *)mb_malloc(sizeof (*tag));
                memcpy(tag, &gfx_fb, sizeof (*tag));
                tag->framebuffer_common.mb_type =
                    MULTIBOOT_TAG_TYPE_FRAMEBUFFER;
                tag->framebuffer_common.mb_size = sizeof (*tag);
#else
                extern multiboot_color_t *cmap;
                uint32_t size;

                if (gfx_fb.framebuffer_common.framebuffer_type ==
                    MULTIBOOT_FRAMEBUFFER_TYPE_INDEXED) {
                        uint16_t nc;
                        nc = gfx_fb.u.fb1.framebuffer_palette_num_colors;
                        size = sizeof (struct multiboot_tag_framebuffer_common)
                            + sizeof (nc)
                            + nc * sizeof (multiboot_color_t);
                } else {
                        size = sizeof (gfx_fb);
                }

                tag = (multiboot_tag_framebuffer_t *)mb_malloc(size);
                memcpy(tag, &gfx_fb, sizeof (*tag));

                tag->framebuffer_common.mb_type =
                    MULTIBOOT_TAG_TYPE_FRAMEBUFFER;
                tag->framebuffer_common.mb_size = size;

                if (gfx_fb.framebuffer_common.framebuffer_type ==
                    MULTIBOOT_FRAMEBUFFER_TYPE_INDEXED) {
                        memcpy(tag->u.fb1.framebuffer_palette, cmap,
                            sizeof (multiboot_color_t) *
                            gfx_fb.u.fb1.framebuffer_palette_num_colors);
                }
#endif /* EFI */
        }

#if defined(EFI)
        /* Leave EFI memmap last as we will also switch off the BS. */
        {
                multiboot_tag_efi_mmap_t *tag;
                UINTN key;
                EFI_STATUS status;
                tag = (multiboot_tag_efi_mmap_t *)
                    mb_malloc(sizeof (*tag));

                map_size = 0;
                status = BS->GetMemoryMap(&map_size,
                    (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap, &key,
                    &desc_size, &tag->mb_descr_vers);
                if (status != EFI_BUFFER_TOO_SMALL) {
                        error = EINVAL;
                        goto error;
                }
                status = BS->GetMemoryMap(&map_size,
                    (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap, &key,
                    &desc_size, &tag->mb_descr_vers);
                if (EFI_ERROR(status)) {
                        error = EINVAL;
                        goto error;
                }
                tag->mb_type = MULTIBOOT_TAG_TYPE_EFI_MMAP;
                tag->mb_size = sizeof (*tag) + map_size;
                tag->mb_descr_size = (uint32_t)desc_size;

                map = (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap;
                /*
                 * Find relocater pages. We assume we have free pages
                 * below kernel load address.
                 * In this version we are using 5 pages:
                 * relocator data, trampoline, copy, memmove, stack.
                 */
                for (i = 0, map = (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap;
                    i < map_size / desc_size;
                    i++, map = NextMemoryDescriptor(map, desc_size)) {
                        if (map->PhysicalStart == 0)
                                continue;
                        if (map->Type != EfiConventionalMemory)
                                continue;
                        if (map->PhysicalStart < load_addr &&
                            map->NumberOfPages > 5)
                                break;
                }
                if (map->PhysicalStart == 0)
                        panic("Could not find memory for relocater");

                if (keep_bs == 0) {
                        status = BS->ExitBootServices(IH, key);
                        if (EFI_ERROR(status)) {
                                printf("Call to ExitBootServices failed\n");
                                error = EINVAL;
                                goto error;
                        }
                }

                last_addr += map_size;
                last_addr = roundup2(last_addr, MULTIBOOT_TAG_ALIGN);
        }
#endif /* EFI */

        /*
         * MB tag list end marker.
         */
        {
                multiboot_tag_t *tag = (multiboot_tag_t *)
                    mb_malloc(sizeof (*tag));
                tag->mb_type = MULTIBOOT_TAG_TYPE_END;
                tag->mb_size = sizeof (*tag);
        }
```

```
        mbi->mbi_total_size = last_addr - (vm_offset_t)mbi;
        mbi->mbi_reserved = 0;

#if defined(EFI)
        /*
         * At this point we have load_addr pointing to kernel load
         * address, module list in MBI having physical addresses,
         * module list in fp having logical addresses and tmp pointing to
         * physical address for MBI.
         * Now we must move all pieces to place and start the kernel.
         */
        relocator = (struct relocator *)(uintptr_t)map->PhysicalStart;
        head = &relocator->rel_chunk_head;
        STAILQ_INIT(head);

        i = 0;
        chunk = &relocator->rel_chunklist[i++];
        chunk->chunk_vaddr = fp->f_addr;
        chunk->chunk_paddr = load_addr;
        chunk->chunk_size = fp->f_size;

        STAILQ_INSERT_TAIL(head, chunk, chunk_next);

        mp = module;
        for (mfp = fp->f_next; mfp != NULL; mfp = mfp->f_next) {
                chunk = &relocator->rel_chunklist[i++];
                chunk->chunk_vaddr = mfp->f_addr;

                /*
                 * fix the mb_mod_start and mb_mod_end.
                 */
                mp->mb_mod_start = efi_physaddr(module, tmp, map,
                    map_size / desc_size, desc_size, mp->mb_mod_end);
                if (mp->mb_mod_start == 0)
                        panic("Could not find memory for module");

                mp->mb_mod_end += mp->mb_mod_start;
                chunk->chunk_paddr = mp->mb_mod_start;
                chunk->chunk_size = mfp->f_size;
                STAILQ_INSERT_TAIL(head, chunk, chunk_next);

                mp = (multiboot_tag_module_t *)
                    roundup2((uintptr_t)mp + mp->mb_size,
                    MULTIBOOT_TAG_ALIGN);
        }
        chunk = &relocator->rel_chunklist[i++];
        chunk->chunk_vaddr = (EFI_VIRTUAL_ADDRESS)(uintptr_t)mbi;
        chunk->chunk_paddr = efi_physaddr(module, tmp, map,
            map_size / desc_size, desc_size, mbi->mbi_total_size);
        chunk->chunk_size = mbi->mbi_total_size;
        STAILQ_INSERT_TAIL(head, chunk, chunk_next);

        trampoline = (void *)(uintptr_t)relocator + EFI_PAGE_SIZE;
        memmove(trampoline, multiboot_tramp, EFI_PAGE_SIZE);

        relocator->rel_copy = (uintptr_t)trampoline + EFI_PAGE_SIZE;
        memmove((void *)relocator->rel_copy, efi_copy_finish, EFI_PAGE_SIZE);

        relocator->rel_memmove = (uintptr_t)relocator->rel_copy + EFI_PAGE_SIZE;
        memmove((void *)relocator->rel_memmove, memmove, EFI_PAGE_SIZE);
        relocator->rel_stack = relocator->rel_memmove + EFI_PAGE_SIZE - 8;

        trampoline(MULTIBOOT2_BOOTLOADER_MAGIC, relocator, entry_addr);
#else
        dev_cleanup();
        __exec((void *)VTOP(multiboot_tramp), MULTIBOOT2_BOOTLOADER_MAGIC,
            (void *)entry_addr, (void *)VTOP(mbi));
#endif /* EFI */
        panic("exec returned");

error:
        if (cmdline != NULL)
        free(cmdline);

#if defined(EFI)
        free(relocator);

        if (mbi != NULL)
                efi_free_loadaddr((vm_offset_t)mbi, EFI_SIZE_TO_PAGES(size));
#endif

        return (error);
}
______unchanged_portion_omitted_
```