

new/usr/src/boot/sys/boot/common/multiboot2.c

1

```
*****
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>
*****
1 /*
2  * This file and its contents are supplied under the terms of the
3  * Common Development and Distribution License ("CDDL"), version 1.0.
4  * You may only use this file in accordance with the terms of version
5  * 1.0 of the CDDL.
6  *
7  * A full copy of the text of the CDDL should have accompanied this
8  * source. A copy of the CDDL is also available via the Internet at
9  * http://www.illumos.org/license/CDDL.
10 */

12 /*
13  * Copyright 2017 Toomas Soome <tsoome@me.com>
14  * Copyright 2019, Joyent, Inc.
15 */

17 /*
18  * This module adds support for loading and booting illumos multiboot2
19  * kernel. This code is only built to support the illumos kernel, it does
20  * not support xen.
21 */

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

26 #include <sys/param.h>
27 #include <sys/exec.h>
28 #include <sys/linker.h>
29 #include <sys/module.h>
30 #include <sys/stdint.h>
31 #include <sys/multiboot2.h>
32 #include <stand.h>
33 #include <stdbool.h>
34 #include <machine/elf.h>
35 #include "libzfs.h"

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

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

43 #define SUPPORT_DHCP
44 #include <bootp.h>

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

51 #else
52 #include <efi.h>
53 #include <efilib.h>
54 #include "loader_efi.h"

56 static void (*trampoline)(uint32_t, struct relocater *, uint64_t);
57 #endif
```

new/usr/src/boot/sys/boot/common/multiboot2.c

2

```
59 #include "platform/acfreebsd.h"
60 #include "acconfig.h"
61 #define ACPI_SYSTEM_XFACE
62 #include "actypes.h"
63 #include "actbl.h"

65 extern ACPI_TABLE_RSDP *rsdp;

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

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

73 struct file_format multiboot2 = { multiboot2_loadfile, multiboot2_exec };
74 static bool keep_bs = false;
75 static bool have_framebuffer = false;
76 static vm_offset_t load_addr;
77 static vm_offset_t entry_addr;

79 /*
80  * Validate tags in info request. This function is provided just to
81  * recognize the current tag list and only serves as a limited
82  * safe guard against possibly corrupt information.
83  */
84 static bool
85 is_info_request_valid(multiboot_header_tag_information_request_t *rtag)
86 {
87     int i;

89     /*
90      * If the tag is optional and we do not support it, we do not
91      * have to do anything special, so we skip optional tags.
92      */
93     if (rtag->mbh_flags & MULTIBOOT_HEADER_TAG_OPTIONAL)
94         return (true);

96     for (i = 0; i < (rtag->mbh_size - sizeof (*rtag)) /
97          sizeof (rtag->mbh_requests[0]); i++)
98         switch (rtag->mbh_requests[i]) {
99             case MULTIBOOT_TAG_TYPE_END:
100             case MULTIBOOT_TAG_TYPE_CMDLINE:
101             case MULTIBOOT_TAG_TYPE_BOOT_LOADER_NAME:
102             case MULTIBOOT_TAG_TYPE_MODULE:
103             case MULTIBOOT_TAG_TYPE_BASIC_MEMINFO:
104             case MULTIBOOT_TAG_TYPE_BOOTDEV:
105             case MULTIBOOT_TAG_TYPE_MMAP:
106             case MULTIBOOT_TAG_TYPE_FRAMEBUFFER:
107             case MULTIBOOT_TAG_TYPE_VBE:
108             case MULTIBOOT_TAG_TYPE_ELF_SECTIONS:
109             case MULTIBOOT_TAG_TYPE_APM:
110             case MULTIBOOT_TAG_TYPE_EFI32:
111             case MULTIBOOT_TAG_TYPE_EFI64:
112             case MULTIBOOT_TAG_TYPE_ACPI_OLD:
113             case MULTIBOOT_TAG_TYPE_ACPI_NEW:
114             case MULTIBOOT_TAG_TYPE_NETWORK:
115             case MULTIBOOT_TAG_TYPE_EFI_MMAP:
116             case MULTIBOOT_TAG_TYPE_EFI_BS:
117             case MULTIBOOT_TAG_TYPE_EFI32_IH:
118             case MULTIBOOT_TAG_TYPE_EFI64_IH:
119             case MULTIBOOT_TAG_TYPE_LOAD_BASE_ADDR:
120                 break;
121             default:
122                 printf("unsupported information tag: 0x%x\n",
123                        rtag->mbh_requests[i]);
124                 return (false);
125         }
```

```

125     }
126     return (true);
127 }
    unchanged_portion_omitted

803 #if defined(EFI)
804 static bool
805 overlaps(uintptr_t start1, size_t size1, uintptr_t start2, size_t size2)
806 {
807     if (start1 < start2 + size2 &&
808         start1 + size1 >= start2) {
809         printf("overlaps: %zx-%zx, %zx-%zx\n",
810             start1, start1 + size1, start2, start2 + size2);
811         return (true);
812     }
813
814     return (false);
815 }
816 #endif

818 static int
819 multiboot2_exec(struct preloaded_file *fp)
820 {
821     multiboot2_info_header_t *mbi = NULL;
822     struct preloaded_file *mfp;
823     multiboot2_info_header_t *mbi;
824     char *cmdline = NULL;
825     struct devdesc *rootdev;
826     struct file_metadata *md;
827     int i, error, num;
828     int rootfs = 0;
829     size_t size;
830     struct bios_smap *smap;
831 #if defined(EFI)
832     multiboot_tag_module_t *module, *mp;
833     struct relocater *relocater = NULL;
834     EFI_MEMORY_DESCRIPTOR *map;
835     UINTN map_size, desc_size;
836     struct relocater *relocater;
837     struct chunk_head *head;
838     struct chunk *chunk;
839     vm_offset_t tmp;
840
841     efi_getdev((void **)&rootdev, NULL, NULL);
842
843     /*
844      * We need 5 pages for relocation. We'll allocate from the heap: while
845      * it's possible that our heap got placed low down enough to be in the
846      * way of where we're going to relocate our kernel, it's hopefully not
847      * likely.
848      */
849     if ((relocater = malloc(EFI_PAGE_SIZE * 5)) == NULL) {
850         printf("relocater malloc failed!\n");
851         error = ENOMEM;
852         goto error;
853     }
854
855     if (overlaps((uintptr_t)relocater, EFI_PAGE_SIZE * 5,
856         load_addr, fp->f_size)) {
857         printf("relocater pages overlap the kernel!\n");
858         error = EINVAL;
859         goto error;
860     }
861 #else
862     i386_getdev((void **)&rootdev, NULL, NULL);

```

```

863     if (have_framebuffer == false) {
864         /* make sure we have text mode */
865         bios_set_text_mode(VGA_TEXT_MODE);
866     }
867 #endif

868
869     mbi = NULL;
870     error = EINVAL;
871     if (rootdev == NULL) {
872         printf("can't determine root device\n");
873         goto error;
874     }
875
876     /*
877      * Set the image command line.
878      */
879     if (fp->f_args == NULL) {
880         cmdline = getenv("boot-args");
881         if (cmdline != NULL) {
882             fp->f_args = strdup(cmdline);
883             if (fp->f_args == NULL) {
884                 error = ENOMEM;
885                 goto error;
886             }
887         }
888     }
889
890     error = mb_kernel_cmdline(fp, rootdev, &cmdline);
891     if (error != 0)
892         goto error;
893
894     /* mb_kernel_cmdline() updates the environment. */
895     build_environment_module();
896
897     if (have_framebuffer == true) {
898         /* Pass the loaded console font for kernel. */
899         build_font_module();
900     }
901
902     size = mbi_size(fp, cmdline); /* Get the size for MBI. */
903
904     /* Set up the base for mb_malloc. */
905     i = 0;
906     for (mfp = fp; mfp->f_next != NULL; mfp = mfp->f_next)
907         i++;
908
909 #if defined(EFI)
910     /* We need space for kernel + MBI + # modules */
911     num = (EFI_PAGE_SIZE - offsetof(struct relocater, rel_chunklist)) /
912         sizeof(struct chunk);
913     if (i + 2 >= num) {
914         printf("Too many modules, do not have space for relocater.\n");
915         error = ENOMEM;
916         goto error;
917     }
918
919     last_addr = efi_loadaddr(LOAD_MEM, &size, mfp->f_addr + mfp->f_size);
920     mbi = (multiboot2_info_header_t *)last_addr;
921     if (mbi == NULL) {
922         error = ENOMEM;
923         goto error;
924     }
925     last_addr = (vm_offset_t)mbi->mbi_tags;
926 #else
927     /* Start info block from the new page. */

```

```

927     last_addr = i386_loadaddr(LOAD_MEM, &size, mfp->f_addr + mfp->f_size);

929     /* Do we have space for multiboot info? */
930     if (last_addr + size >= memtop_copyin) {
931         error = ENOMEM;
932         goto error;
933     }

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

939     {
940         multiboot_tag_string_t *tag;
941         i = sizeof (multiboot_tag_string_t) + strlen(cmdline) + 1;
942         tag = (multiboot_tag_string_t *)mb_malloc(i);

944         tag->mb_type = MULTIBOOT_TAG_TYPE_CMDLINE;
945         tag->mb_size = i;
946         memcpy(tag->mb_string, cmdline, strlen(cmdline) + 1);
947         free(cmdline);
948         cmdline = NULL;
949     }

951     {
952         multiboot_tag_string_t *tag;
953         i = sizeof (multiboot_tag_string_t) + strlen(bootprog_info) + 1;
954         tag = (multiboot_tag_string_t *)mb_malloc(i);

956         tag->mb_type = MULTIBOOT_TAG_TYPE_BOOT_LOADER_NAME;
957         tag->mb_size = i;
958         memcpy(tag->mb_string, bootprog_info,
959             strlen(bootprog_info) + 1);
960     }

962 #if !defined(EFI)
963     /* Only set in case of BIOS. */
964     {
965         multiboot_tag_basic_meminfo_t *tag;
966         tag = (multiboot_tag_basic_meminfo_t *)
967             mb_malloc(sizeof (*tag));

969         tag->mb_type = MULTIBOOT_TAG_TYPE_BASIC_MEMINFO;
970         tag->mb_size = sizeof (*tag);
971         tag->mb_mem_lower = bios_basemem / 1024;
972         tag->mb_mem_upper = bios_extmem / 1024;
973     }
974 #endif

976     num = 0;
977     for (mfp = fp->f_next; mfp != NULL; mfp = mfp->f_next) {
978         num++;
979         if (mfp->f_type != NULL && strcmp(mfp->f_type, "rootfs") == 0)
980             rootfs++;
981     }

983     if (num == 0 || rootfs == 0) {
984         /* We need at least one module - rootfs. */
985         printf("No rootfs module provided, aborting\n");
986         error = EINVAL;
987         goto error;
988     }

990     /*
991     * Set the stage for physical memory layout:
992     * - We have kernel at load_addr.

```

```

993     * - Modules are aligned to page boundary.
994     * - MBI is aligned to page boundary.
995     * - Set the tmp to point to physical address of the first module.
996     * - tmp != mfp->f_addr only in case of EFI.
997     */
998 #if defined(EFI)
999     tmp = roundup2(load_addr + fp->f_size + 1, MULTIBOOT_MOD_ALIGN);
1000     module = (multiboot_tag_module_t *)last_addr;
1001 #endif

1003     for (mfp = fp->f_next; mfp != NULL; mfp = mfp->f_next) {
1004         multiboot_tag_module_t *tag;

1006         num = strlen(mfp->f_name) + 1;
1007         num += strlen(mfp->f_type) + 5 + 1;
1008         if (mfp->f_args != NULL) {
1009             num += strlen(mfp->f_args) + 1;
1010         }
1011         cmdline = malloc(num);
1012         if (cmdline == NULL) {
1013             error = ENOMEM;
1014             goto error;
1015         }

1017         if (mfp->f_args != NULL)
1018             snprintf(cmdline, num, "%s type=%s %s",
1019                 mfp->f_name, mfp->f_type, mfp->f_args);
1020         else
1021             snprintf(cmdline, num, "%s type=%s",
1022                 mfp->f_name, mfp->f_type);

1024         tag = (multiboot_tag_module_t *)mb_malloc(sizeof (*tag) + num);

1026         tag->mb_type = MULTIBOOT_TAG_TYPE_MODULE;
1027         tag->mb_size = sizeof (*tag) + num;
1028 #if defined(EFI)
1029         /*
1030          * We can assign module addresses only after BS have been
1031          * switched off.
1032          */
1033         tag->mb_mod_start = 0;
1034         tag->mb_mod_end = mfp->f_size;
1035 #else
1036         tag->mb_mod_start = mfp->f_addr;
1037         tag->mb_mod_end = mfp->f_addr + mfp->f_size;
1038 #endif
1039         memcpy(tag->mb_cmdline, cmdline, num);
1040         free(cmdline);
1041         cmdline = NULL;
1042     }

1044     md = file_findmetadata(fp, MODINFOMD_SMAP);
1045     if (md == NULL) {
1046         printf("no memory smap\n");
1047         error = EINVAL;
1048         goto error;
1049     }

1051     smap = (struct bios_smap *)md->md_data;
1052     num = md->md_size / sizeof (struct bios_smap); /* number of entries */

1054     {
1055         multiboot_tag_mmap_t *tag;
1056         multiboot_mmap_entry_t *mmap_entry;

1058         tag = (multiboot_tag_mmap_t *)

```

```

1059         mb_malloc(sizeof (*tag) +
1060         num * sizeof (multiboot_mmap_entry_t));

1062     tag->mb_type = MULTIBOOT_TAG_TYPE_MMAP;
1063     tag->mb_size = sizeof (*tag) +
1064     num * sizeof (multiboot_mmap_entry_t);
1065     tag->mb_entry_size = sizeof (multiboot_mmap_entry_t);
1066     tag->mb_entry_version = 0;
1067     mmap_entry = (multiboot_mmap_entry_t *)tag->mb_entries;

1069     for (i = 0; i < num; i++) {
1070         mmap_entry[i].mmap_addr = smap[i].base;
1071         mmap_entry[i].mmap_len = smap[i].length;
1072         mmap_entry[i].mmap_type = smap[i].type;
1073         mmap_entry[i].mmap_reserved = 0;
1074     }
1075 }

1077 if (bootp_response != NULL) {
1078     multiboot_tag_network_t *tag;
1079     tag = (multiboot_tag_network_t *)
1080     mb_malloc(sizeof (*tag) + bootp_response_size);

1082     tag->mb_type = MULTIBOOT_TAG_TYPE_NETWORK;
1083     tag->mb_size = sizeof (*tag) + bootp_response_size;
1084     memcpy(tag->mb_dhcpack, bootp_response, bootp_response_size);
1085 }

1087 #if !defined(EFI)
1088     multiboot_tag_vbe_t *tag;
1089     extern multiboot_tag_vbe_t vbestate;

1091     if (VBE_VALID_MODE(vbestate.vbe_mode)) {
1092         tag = (multiboot_tag_vbe_t *)mb_malloc(sizeof (*tag));
1093         memcpy(tag, &vbestate, sizeof (*tag));
1094         tag->mb_type = MULTIBOOT_TAG_TYPE_VBE;
1095         tag->mb_size = sizeof (*tag);
1096     }
1097 #endif

1099     if (rsdp != NULL) {
1100         multiboot_tag_new_acpi_t *ntag;
1101         multiboot_tag_old_acpi_t *otag;
1102         uint32_t tsize;

1104         if (rsdp->Revision == 0) {
1105             tsize = sizeof (*otag) + sizeof (ACPI_RSDP_COMMON);
1106             otag = (multiboot_tag_old_acpi_t *)mb_malloc(tsize);
1107             otag->mb_type = MULTIBOOT_TAG_TYPE_ACPI_OLD;
1108             otag->mb_size = tsize;
1109             memcpy(otag->mb_rsdp, rsdp, sizeof (ACPI_RSDP_COMMON));
1110         } else {
1111             tsize = sizeof (*ntag) + rsdp->Length;
1112             ntag = (multiboot_tag_new_acpi_t *)mb_malloc(tsize);
1113             ntag->mb_type = MULTIBOOT_TAG_TYPE_ACPI_NEW;
1114             ntag->mb_size = tsize;
1115             memcpy(ntag->mb_rsdp, rsdp, rsdp->Length);
1116         }
1117     }

1119 #if defined(EFI)
1120 #ifdef __LP64__
1121     {
1122         multiboot_tag_efi64_t *tag;
1123         tag = (multiboot_tag_efi64_t *)
1124         mb_malloc(sizeof (*tag));

```

```

1126         tag->mb_type = MULTIBOOT_TAG_TYPE_EFI64;
1127         tag->mb_size = sizeof (*tag);
1128         tag->mb_pointer = (uint64_t)(uintptr_t)ST;
1129     }
1130 #else
1131     {
1132         multiboot_tag_efi32_t *tag;
1133         tag = (multiboot_tag_efi32_t *)
1134         mb_malloc(sizeof (*tag));

1136         tag->mb_type = MULTIBOOT_TAG_TYPE_EFI32;
1137         tag->mb_size = sizeof (*tag);
1138         tag->mb_pointer = (uint32_t)ST;
1139     }
1140 #endif /* __LP64__ */
1141 #endif /* EFI */

1143     if (have_framebuffer == true) {
1144         multiboot_tag_framebuffer_t *tag;
1145         extern multiboot_tag_framebuffer_t gfx_fb;
1146 #if defined(EFI)

1148         tag = (multiboot_tag_framebuffer_t *)mb_malloc(sizeof (*tag));
1149         memcpy(tag, &gfx_fb, sizeof (*tag));
1150         tag->framebuffer_common.mb_type =
1151         MULTIBOOT_TAG_TYPE_FRAMEBUFFER;
1152         tag->framebuffer_common.mb_size = sizeof (*tag);
1153 #else
1154         extern multiboot_color_t *cmap;
1155         uint32_t size;

1157         if (gfx_fb.framebuffer_common.framebuffer_type ==
1158         MULTIBOOT_FRAMEBUFFER_TYPE_INDEXED) {
1159             uint16_t nc;
1160             nc = gfx_fb.u.fbl.framebuffer_palette_num_colors;
1161             size = sizeof (struct multiboot_tag_framebuffer_common)
1162             + sizeof (nc)
1163             + nc * sizeof (multiboot_color_t);
1164         } else {
1165             size = sizeof (gfx_fb);
1166         }

1168         tag = (multiboot_tag_framebuffer_t *)mb_malloc(size);
1169         memcpy(tag, &gfx_fb, sizeof (*tag));

1171         tag->framebuffer_common.mb_type =
1172         MULTIBOOT_TAG_TYPE_FRAMEBUFFER;
1173         tag->framebuffer_common.mb_size = size;

1175         if (gfx_fb.framebuffer_common.framebuffer_type ==
1176         MULTIBOOT_FRAMEBUFFER_TYPE_INDEXED) {
1177             memcpy(tag->u.fbl.framebuffer_palette, cmap,
1178             sizeof (multiboot_color_t) *
1179             gfx_fb.u.fbl.framebuffer_palette_num_colors);
1180         }
1181 #endif /* EFI */
1182     }

1184 #if defined(EFI)
1185     /* Leave EFI memmap last as we will also switch off the BS. */
1186     {
1187         multiboot_tag_efi_mmap_t *tag;
1188         UINTN key;
1189         EFI_STATUS status;

```

```

1191     tag = (multiboot_tag_efi_mmap_t *)
1192         mb_malloc(sizeof(*tag));

1194     map_size = 0;
1195     status = BS->GetMemoryMap(&map_size,
1196         (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap, &key,
1197         &desc_size, &tag->mb_descr_vers);
1198     if (status != EFI_BUFFER_TOO_SMALL) {
1199         error = EINVALID;
1200         goto error;
1201     }
1202     status = BS->GetMemoryMap(&map_size,
1203         (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap, &key,
1204         &desc_size, &tag->mb_descr_vers);
1205     if (EFI_ERROR(status)) {
1206         error = EINVALID;
1207         goto error;
1208     }
1209     tag->mb_type = MULTIBOOT_TAG_TYPE_EFI_MMAP;
1210     tag->mb_size = sizeof(*tag) + map_size;
1211     tag->mb_descr_size = (uint32_t)desc_size;

1213     map = (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap;
1214     /*
1215      * Find relocater pages. We assume we have free pages
1216      * below kernel load address.
1217      * In this version we are using 5 pages:
1218      * relocater data, trampoline, copy, memmove, stack.
1219      */
1214     for (i = 0, map = (EFI_MEMORY_DESCRIPTOR *)tag->mb_efi_mmap;
1215         i < map_size / desc_size;
1216         i++, map = NextMemoryDescriptor(map, desc_size)) {
1217         if (map->PhysicalStart == 0)
1218             continue;
1219         if (map->Type != EfiConventionalMemory)
1220             continue;
1221         if (map->PhysicalStart < load_addr &&
1222             map->NumberOfPages > 5)
1223             break;
1224     }
1225     if (map->PhysicalStart == 0)
1226         panic("Could not find memory for relocater");

1215     if (keep_bs == 0) {
1216         status = BS->ExitBootServices(IH, key);
1217         if (EFI_ERROR(status)) {
1218             printf("Call to ExitBootServices failed\n");
1219             error = EINVALID;
1220             goto error;
1221         }
1222     }

1224     last_addr += map_size;
1225     last_addr = roundup2(last_addr, MULTIBOOT_TAG_ALIGN);
1226 }
1227 #endif /* EFI */

1229     /*
1230     * MB tag list end marker.
1231     */
1232     {
1233         multiboot_tag_t *tag = (multiboot_tag_t *)
1234             mb_malloc(sizeof(*tag));
1235         tag->mb_type = MULTIBOOT_TAG_TYPE_END;
1236         tag->mb_size = sizeof(*tag);
1237     }

```

```

1239     mbi->mbi_total_size = last_addr - (vm_offset_t)mbi;
1240     mbi->mbi_reserved = 0;

1242 #if defined(EFI)
1243     /*
1244     * At this point we have load_addr pointing to kernel load
1245     * address, module list in MBI having physical addresses,
1246     * module list in fp having logical addresses and tmp pointing to
1247     * physical address for MBI.
1248     * Now we must move all pieces to place and start the kernel.
1249     */
1250     relocater = (struct relocater *) (uintptr_t)map->PhysicalStart;
1251     head = &relocater->rel_chunk_head;
1252     STAILQ_INIT(head);

1253     i = 0;
1254     chunk = &relocater->rel_chunklist[i++];
1255     chunk->chunk_vaddr = fp->f_addr;
1256     chunk->chunk_paddr = load_addr;
1257     chunk->chunk_size = fp->f_size;

1259     STAILQ_INSERT_TAIL(head, chunk, chunk_next);

1261     mp = module;
1262     for (mfp = fp->f_next; mfp != NULL; mfp = mfp->f_next) {
1263         chunk = &relocater->rel_chunklist[i++];
1264         chunk->chunk_vaddr = mfp->f_addr;

1266         /*
1267         * fix the mb_mod_start and mb_mod_end.
1268         */
1269         mp->mb_mod_start = efi_physaddr(module, tmp, map,
1270             map_size / desc_size, desc_size, mp->mb_mod_end);
1271         if (mp->mb_mod_start == 0)
1272             panic("Could not find memory for module");

1274         mp->mb_mod_end += mp->mb_mod_start;
1275         chunk->chunk_paddr = mp->mb_mod_start;
1276         chunk->chunk_size = mfp->f_size;
1277         STAILQ_INSERT_TAIL(head, chunk, chunk_next);

1279         mp = (multiboot_tag_module_t *)
1280             roundup2((uintptr_t)mp + mp->mb_size,
1281                 MULTIBOOT_TAG_ALIGN);
1282     }
1283     chunk = &relocater->rel_chunklist[i++];
1284     chunk->chunk_vaddr = (EFI_VIRTUAL_ADDRESS)(uintptr_t)mbi;
1285     chunk->chunk_paddr = efi_physaddr(module, tmp, map,
1286         map_size / desc_size, desc_size, mbi->mbi_total_size);
1287     chunk->chunk_size = mbi->mbi_total_size;
1288     STAILQ_INSERT_TAIL(head, chunk, chunk_next);

1290     trampoline = (void *) (uintptr_t)relocater + EFI_PAGE_SIZE;
1291     memmove(trampoline, multiboot_trampoline, EFI_PAGE_SIZE);

1293     relocater->rel_copy = (uintptr_t)trampoline + EFI_PAGE_SIZE;
1294     memmove((void *)relocater->rel_copy, efi_copy_finish, EFI_PAGE_SIZE);

1296     relocater->rel_memmove = (uintptr_t)relocater->rel_copy + EFI_PAGE_SIZE;
1297     memmove((void *)relocater->rel_memmove, memmove, EFI_PAGE_SIZE);
1298     relocater->rel_stack = relocater->rel_memmove + EFI_PAGE_SIZE - 8;

1300     trampoline(MULTIBOOT2_BOOTLOADER_MAGIC, relocater, entry_addr);
1301 #else
1302     dev_cleanup();

```

```
1303     __exec((void *)VTOP(multiboot_trampoline), MULTIBOOT2_BOOTLOADER_MAGIC,
1304            (void *)entry_addr, (void *)VTOP(mbi));
1305 #endif /* EFI */
1306     panic("exec returned");

1308 error:
1293     if (cmdline != NULL)
1309         free(cmdline);

1311 #if defined(EFI)
1312     free(relocator);

1314     if (mbi != NULL)
1315         efi_free_loadaddr((vm_offset_t)mbi, EFI_SIZE_TO_PAGES(size));
1316 #endif

1318     return (error);
1319 }
_____unchanged_portion_omitted_____
```