Operating Systems Development Series

Operating Systems Development - Multiboot

by Mike, 2010

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

This tutorial covers the multiboot standard and how to develop a multiboot-complient operating system. While the series goes into the multiboot structure, there is more to creating a multiboot complient system. By your system being multiboot complient, it will be capable of being loaded by any multiboot complient boot loader. Cool, huh? This means any multi boot complient bootloader can boot your OS.

Multiboot Specification

Abstract

The multiboot standard was originally created in 1995 and is overseen by the **Free Software Foundation**. They provide a written specification that defines a standard way to allow **multi-booting**. **Multi-booting** allows a computer system to install, and run, multiple operating systems and system environments. A **dual-boot** computer system is an example of multibooting, with two operating systems installed.

Multibooting is made possible by another, unofficial software trick: **Partitioning**. **Partitioning** creates the effect of multiple logical disks on one physical disk. Partitioning separates the storage on a storage medium (typically a hard disk) for different uses. For example, the first partition can be from sector 0 to sector *n* and contain an NTFS formatted Windows operating system. The other partitions can be of any file system - containing data or another operating system software.

Because partitioning is a software trick, it is up to the boot loader to be able to detect these partitions by reading the software **Partition Table** and, typically, display a list of the partitions that contain an operating system to boot. This is the **boot menu**. The Multiboot Specification defines the state of the computer when the bootloader transfers control to an operating system and how data is passed to the operating system.

The multiboot standard can be used on disks that do not support multibooting as well. This means, if your multi-boot complient bootloader can boot from a floppy disk, you can make your floppy disk OS boot from it.

Operating System Image

Typical bootloaders can be configured to boot different types of operating system images. Typically this is the Kernel or another OS Loading

program. The multiboot specification does not provide details on the format of the image. Because of this, you can use any format that you want - flat binary, ELF, or even PE files.

However, this file requires an additional header - the **Multiboot Header**. This header must be located somewhere within the first 8k of your image and aligned on a dword (32 bit) boundery. Any multiboot complient bootloader will be able to find this header and obtain information from it. This is how the boot loader will know how to load and execute your image.

Here is the structure format:

```
typedef struct _MULTIBOOT_INFO {

   uint32_t magic;       //all required...
   uint32_t flags;
   uint32_t checksum;
   uint32_t headerAddr;  //all optional, set if bit 16 in flags is set...
   uint32_t loadAddr;
   uint32_t loadEndAddr;
   uint32_t bssEndAddr;
   uint32_t entryPoint;
   uint32_t modType;     //all optional, set if bit 2 in flags is set...
   uint32_t width;
   uint32_t height;
   uint32_t depth;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;
```

You should make sure no padding is added. In MSVC, this can be done by adding a **#pragma pack (push, 1)** and **#pragma pack(pop,1)** around the structure declaration above.

The above is the only structure that you need to get your OS booted by a multi boot complient bootloader, such as GRUB. Lets look at the members:

- **magic:** must always be 0x1BADB002
- **flags:**
  - **Bit 0**
    - 0: All boot modules and OS image must be aligned in page (4k) bounderies.
  - **Bit 1**
    - 1: Boot loader must pass memory information to the operating system.
  - **Bit 2**
    - 1: Boot loader must pass the video mode table to the operating system.
  - **Bit 16**
    - 1: Offsets 12-28 of the multiboot header are valid. (That is, members header_addr through entry_addr in the multiboot header are valid.) If this bit is set, the boot loader will use these values instead of parsing the image format and obtaining the values

from it. Multiboot complient bootloaders can provide support for native executable file formats, such as ELF or PE that it can load.

- **checksum:** This must be a value that which, when added to **magic** and **flags** must be a 32 bit unsigned sum of 0.
- **headerAddr:** Address of multiboot header
- **loadAddr:** Base address to load to
- **loadEndAddr:** End load address. If 0, bootloader assumes the end is the end of the OS image file
- **bssEndAddr:** End of BSS segment. Bootloader null's this segment. If 0, no BSS segment is assumed
- **entryPoint:** Address of entry point function. Yes, thats right, the entry point of your operating system
- **modType:**
  - 0: Linear graphics mode
  - 1: EGA Standard text mode
  - Everything else is reserved
- **width:** width of display in text columns or pixels. If 0, the bootloader assumes no preference
- **height:** height of display in text columns or pixels. If 0, the bootloader assumes no preference
- **depth:** Number of **Bits Per Pixels (BPP)** in a graphics mode. If 0, the bootloader assumes no preference

That is all there is to it. The boot loader can load and execute your operating system in two ways: By loading and reading its executable image format (ELF and PE are examples) or by using the information found in this structure.

The boot loader looks for this structure in your image. Because of this, you need to fill out and create this structure.

Implementing the Multi boot Header

There are different solutions for implimenting the multiboot header into your operating system. Different solutions for different toolchains. Lets look at some of them.

Visual C++ 2005, 2008

This is a recent trick I discovered and posted on another forum. It uses some extensions provided by Microsoft Visual C++ to define the header in your kernel.

We first declare the structure, making sure there is no extra padding:

```
#pragma pack (push, 1)

/**
*   Multiboot structure
*/
typedef struct _MULTIBOOT_INFO {

   uint32_t magic;
   uint32_t flags;
   uint32_t checksum;
```

```
   uint32_t headerAddr;
   uint32_t loadAddr;
   uint32_t loadEndAddr;
   uint32_t bssEndAddr;
   uint32_t entryPoint;
   uint32_t modType;
   uint32_t width;
   uint32_t height;
   uint32_t depth;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;

#pragma pack(pop,1)
```

Now all that we need to do is define this structure somewhere. Remember that this header must be defined on a dword (32 bit) boundery and within the first 8K of your kernel? This trick uses section alignment to insure the proper alignment of the structure. We set up the section alignment in the **Linker Options** of the IDE and it is guaranteed to be dword aligned. So, all we need to do is create a new program section and define the structure in it. Neat, huh?

Lets do that now:

```
//! Bad example:
#pragma section(".text")
__declspec(allocate(".text"))
MULTIBOOT_INFO _MultibootInfo = {

   MULTIBOOT_HEADER_MAGIC,
   MULTIBOOT_HEADER_FLAGS,
   CHECKSUM,
   HEADER_ADDRESS,
   LOADBASE,
   0, //load end address
   0, //bss end address
   KeStartup
};
```

This works but is problimatic. This allocates the structure in the .text section, but *where?* This is going to be a problem - Multiboot specification requires the structure be located in the first 8K of the image, but MSVC is still free to place it outside the 8K region.

To fix this problem, we must use the section naming convention. The section naming conventions used in MSVC follow the format **name$loc** where **name** is the name of the section, and **loc** is an alpha-numeric value that represents where, in the section, it represents. Its in alphanumeric order: **section$a** is first, **section$b** is second and so on. So, by using **.text$0** we are representing the beginning of the .text segment. But of

course, just replacing the above **.text** to **.text$a** wont work - My, or my no, that would be too easy. :)

Instead, we define our own section - lets call it **.a$0**. We can create this as a code segment and merge it into the .text section:

```
//! Complete example
#pragma code_seg(".a$0")
__declspec(allocate(".a$0"))
MULTIBOOT_INFO _MultibootInfo = {

   MULTIBOOT_HEADER_MAGIC,
   MULTIBOOT_HEADER_FLAGS,
   CHECKSUM,
   HEADER_ADDRESS,
   LOADBASE,
   0, //load end address
   0, //bss end address
   KeStartup
};

#pragma comment(linker, "/merge:.text=.a")
```

Thats all that there is to it. **KeStartup** is your entry point function, **OADBASE** is the base address of your kernel (like 1MB for example), **HEADER_ADDRESS** is the address of the multiboot header (which happens to be **LOADBASE+0x400** do to .text always starting at offset 0x400), magic is **0x1BADB002**, flags of **0x00010003** and the checksum being **-(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)**.

Here is the complete example:

```
#pragma pack (push, 1)

/**
*   Multiboot structure
*/
typedef struct _MULTIBOOT_INFO {

   uint32_t magic;
   uint32_t flags;
   uint32_t checksum;
   uint32_t headerAddr;
   uint32_t loadAddr;
   uint32_t loadEndAddr;
   uint32_t bssEndAddr;
   uint32_t entryPoint;
```

```
   uint32_t modType;
   uint32_t width;
   uint32_t height;
   uint32_t depth;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;

#pragma pack(pop,1)

/**
*   Kernel entry
*/
void KeStartup ( PMULTIBOOT_INFO* loaderBlock ) {

   __halt ();
}

//! loading address
#define LOADBASE                     0x100000

//! header offset will always be this
#define   ALIGN                           0x400
#define   HEADER_ADDRESS         LOADBASE+ALIGN

#define MULTIBOOT_HEADER_MAGIC        0x1BADB002
#define MULTIBOOT_HEADER_FLAGS        0x00010003
#define STACK_SIZE                    0x4000
#define CHECKSUM                      -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)

#pragma code_seg(".a$0")
__declspec(allocate(".a$0"))
MULTIBOOT_INFO _MultibootInfo = {

   MULTIBOOT_HEADER_MAGIC,
   MULTIBOOT_HEADER_FLAGS,
   CHECKSUM,
   HEADER_ADDRESS,
   LOADBASE,
   0, //load end address
   0, //bss end address
   KeStartup
};

#pragma comment(linker, "/merge:.text=.a")
```

Assuming this kernel has the base address of 1MB, and is compiled with Visual C++ to produce a valid PE executable, this should be bootable by any multiboot complient bootloader.

Machine State

When the bootloader executes our operating system, the registers must be the following values:

- **EAX** - Magic Number. Must be 0x2BADB002. This will indicate to the kernel that our boot loader is multiboot standard
- **EBX** - Containes the physical address of the Multiboot information structure
- **CS** - Must be a 32-bit read/execute code segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact value is undefined.
- **DS,ES,FS,GS,SS** - Must be a 32-bit read/write data segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact values are all undefined.
- **A20 gate** must be enabled
- **CR0** - Bit 31 (PG) bit must be cleared (paging disabled) and Bit 0 (PE) bit must be set (Protected Mode enabled). Other bits undefined

All other registers are undefined. Most of this is already done in our existing boot loader. The only additional two things we must add are for the EAX register and EBX. The most important one for us is stored in EBX. This will contain the physical address of the multiboot information structure. Lets take a look!

Multi boot Information Structure

Now that our operating system is being booted by the boot loader, whats next? Multiboot complient boot loaders also creates an information structure providing information to the operating system. These are passed by a pointer to the structure in the **EBX** register.

This is possibly one of the most important structures contained in the multiboot specification. The information in this structure is passed to the kernel from the EBX register, **This allows a standard way for the boot loader to pass information to the kernel.**

This is a fairly big structure but isnt to bad. Not all of these members are required. The specification states that the operating system must use the flags member to determin what members in the structure exist and what do not.

Here is the entire structure format. Simular to the multi-boot header structure, it is recommended to insure there is no padding added.

```
typedef struct _MULTIBOOT_INFO {

        uint32_t flags;                 //required
        uint32_t memLower;              //if bit 0 in flags are set
        uint32_t memUpper;              //if bit 0 in flags are set
        uint32_t bootDevice;            //if bit 1 in flags are set
        uint32_t commandLine;           //if bit 2 in flags are set
        uint32_t moduleCount;           //if bit 3 in flags are set
        uint32_t moduleAddress;         //if bit 3 in flags are set
        uint32_t syms[4];               //if bits 4 or 5 in flags are set
        uint32_t memMapLength;          //if bit 6 in flags is set
```

```
        uint32_t memMapAddress;         //if bit 6 in flags is set
        uint32_t drivesLength;          //if bit 7 in flags is set
        uint32_t drivesAddress;         //if bit 7 in flags is set
        uint32_t configTable;           //if bit 8 in flags is set
        uint32_t apmTable;              //if bit 9 in flags is set
        uint32_t vbeControlInfo;        //if bit 10 in flags is set
        uint32_t vbeModeInfo;           //if bit 11 in flags is set
        uint32_t vbeMode;               // all vbe_* set if bit 12 in flags are set
        uint32_t vbeInterfaceSeg;
        uint32_t vbeInterfaceOff;
        uint32_t vbeInterfaceLength;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;
```

This structure isnt as complex as it looks. If the corrosponding bit in the flags member is set, it means that the members (shown above) are valid. Because of this, **flags** is, technically, the only required member, all other members are optional.

Lets look at the members here:

- **memLow, memUpper:** Amount of low and upper memory in KB. Low memory starts at 0, upper memory starts at 1MB.
- **bootDevice:** Boot device (see below)
- **commandLine:** Pointer to C-string containing your kernel command line
- **moduleCount:** The number of additional boot modules that were loaded by the boot loader
- **moduleAddress:** address of first **module structure** (see below)
- **syms:** Location of symbol table. See below
- **memMapLength:** Number of entries in system memory map
- **memMapAddress:** Address of memory map
- **drivesLength, drivesAddress:** see below
- **configTable:** Address of BIOS ROM config table (returned from GET CONFIGUATION BIOS INT call)
- **apmTable:** Address of **Advanced Power Management (APM)** table
- **vbeControlInfo, VbevbeModeInfo:** Address of **Video Bios Extensions (VBE)** structures.
- **vbeMode:** VBE mode
- **vbeInterfaceSeg, vbeInterfaceOff, vbeInterfaceLength:** Used to access VBE 2.0 protected mode interface

This chapter does not go over VBE nor APM so we wont cover them here. Memory map information has been described in Chapter 17, including the format of the **System Memory Map**.

The ROM configuation for **configTable** is the table obtained from BIOS INT 0x15 Function 0xC0.

Thats all there is to it. This structure isnt that bad :) There are a few members we havnt looked at though: bootDevice, moduleAddress, syms, drivesLength, and drivesAddress. Lets look at those in detail.

bootDevice

The **bootDevice** member follows the format:

- 1st word: BIOS drive number
- 2nd, 3rd, 4th words: Partition

The BIOS drive number is the number used by the BIOS INT 0x13 services to represent the drive. The other words reoresent the partitions: Words 2,3, and 4 represent Partition 1,2,and 3. Partition 1 is the top level partition, partition 2 is the sub-partition in that partition and so on. Unused partitions are marked as 0xFF.

moduleAddress

This is a pointer to the first module structure. A module structure entry follows the format:

```
typedef struct _MODULE_ENTRY {

	uint32_t moduleStart;
	uint32_t moduleEnd;
	char     string[8];

}MODULE_ENTRY, *PMODULE_ENTRY;
```

moduleStart and **moduleEnd** contain the start and end addresses of the loaded module. "string" represents that module, typically can be a command line or path name or 0 if there is none.

drivesLength, drivesAddress

The **drivesLength** member contains the size of all of the drive structures. **drivesAddress** contains a pointer to the first drive structure. A drive structure entry has the format:

```
typedef struct _DRIVE_ENTRY {

	uint32_t size;                  //size of structure
	uint8_t driveNumber;
	uint8_t driveMode;
	uint16_t driveCylinders;
	uint8_t driveHeads;
	uint8_t driveSectors;
	uint8_t ports [0];              //can be any number of elements

}DRIVE_ENTRY, *PDRIVE_ENTRY;
```

Lets take a closer look at each member:

- **driveNumber:** Number used by BIOS
- **driveMode:**
  - 0: CHS
  - 1: LBA
- **driveCylinders,driveHeads,driveSectors:** Drive Geometry
- **ports:** contain a list of I/O port numbers used by the BIOS for drive access, terminated by 0

Thats all there is to this structure. Only one more member to cover, that strange **syms** member...lets take a look!

syms

The **syms** member is declared in the structure in this chapter as **uint32_t syms[4]** but that is not entirely true. It is actually several members that occupy those bytes that follow the members:

- syms[0] = uint32_t sym_num
- syms[1] = uint32_t sym_size
- syms[2] = uint32_t sym_addr
- syms[3] = uint32_t sym_shndx

While the specifications state that any format for the operating system image can be used (Such as ELF, PE, flat binary, or anything), this one is specific to ELF formats only. Technically the system image (like the kernel) is able to parse itself to obtain symbolic information. However the miltiboot standard allows the boot loader to pass ELF-specific symbolic information to the operating system as well through these members. **sym_num** is the number of symbol entries in the ELF section header. **size** indicates the size of each entry, **addr** contains the address of the symbol table in the ELF binary.

Conclusion

Thats all that there is to the Multi boot standard! Technically all that you need is to define the **Multiboot Header** properly in order to get your system booted by any multiboot complient bootloader. However, you can use the **multiboot information** structure to obtain information that is normally obtained at boot time.

If you would like to support multi boot in the series, you must insure your kernel is loaded at a **physical** address not a virtual one. This is because paging is disabled when a multiboot complient bootloader passes control to your kernel. A good typical address is 1MB, which can be loaded by alot of bootloaders, such as GRUB. You can, of course, enable paging and use it later on of course :)

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System Software Suite.

Questions or comments? Feel free to Contact me.

Home