

MIT6.828 Lab1: Booting a PC

Zhuofan Zhang

August 2020

Contents

1	PC Bootstrap	1
1.1	Getting Started with x86 assembly	1
1.2	Simulating the x86	2
1.3	The PC's Physical Address Space	2
1.4	The ROM BIOS	3
2	The Boot Loader	4
2.1	Loading the Kernel I: boot.S	4
2.2	Loading the Kernel II: bootmain.c	5
3	The Kernel	9
3.1	Using virtual memory to work around position dependence	9
3.2	Formatted Printing to the Console	9
3.3	The Stack	12

Chapter 1

PC Bootstrap

1.1 Getting Started with x86 assembly

本节内容主要为熟悉基本的 x86 汇编语言，此处归纳总结一些基本的语法内容。

```
##### Three kinds of OPERANDs #####
# %xxx -- Register
# $num -- Immediate(eg: $0x1F, $-577)
# (%xxx) -- memory access

##### Addressing Mode #####
# Immediate Addressing
# Register Addressing -- eg: MOV AX, BX (means "copy data in AX to BX")
# Direct Addressing(memeory written in instruction) -- eg: MOV AX, [2000H]
# Indirect Addressing -- eg: MOV AX, [BX]
# Indexed Addressing
# Offsetting Addressing

##### Directives #####
# .global {symbol1,symbol2,...}
# -- declares each symbol in the list to be global(can be seen by the linker).
# .set {symbol} {expression} -- assigns the value of expression to symbol.
# .data -- The .data directive changes the current section to .data.
# .codexx
# -- instructs the assembler to interpret subsequent instructions as exact bits.(eg:
    .code16, .code32)
```

除了基本的语法规则外，需注意 x86 为分段内存管理 (Segment): 寄存器 CS/DS/EX 分别为代码段、数据段及额外段寄存器，当指令中未指定段地址时默认使用这三个寄存器内的值。具体情况根据 CPU 运行于实模式 (Real Mode) 或保护模式 (Protect Mode) 而不同，将在后文详述。

Exercise 1

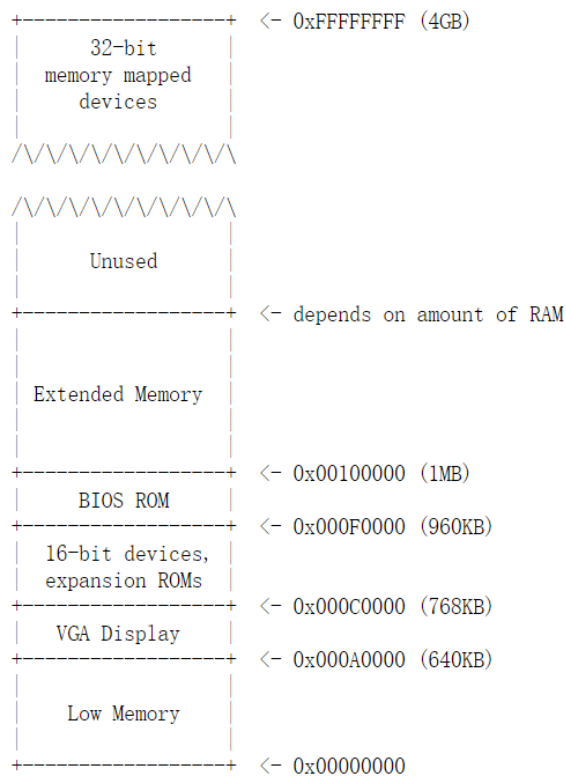
Familiarize yourself with the assembly language materials. You don't have to read them now, but you'll almost certainly want to refer to some of this material when reading and writing x86 assembly.

1.2 Simulating the x86

本节主要内容是介绍课程实验中工具的基本使用方法。实验使用 QEMU 仿真器对 x86 硬件进行模拟，并借助 GDB 进行调试。

1.3 The PC's Physical Address Space

在启动 PC 前，首先了解一些关于物理地址空间 (*Physic Address Space*) 的内容。x86-CPU 系列有一个相对固定的物理地址空间，如下图所示。



早期的 8088 处理器只使用 1MB 内存，对应于此物理空间的低位 1MB，除去固定提供给 BIOS 和 VGA 等使用的地址，余下的 Low Memory 即为早期处理器实际可用的内存空间。其后的处理器为保证前向兼容而保留这一模式，增加 Extend Memory。此外，BIOS 在整个地址空间的最高处保留一小段空间，提供给 32-bit 的 PCI 设备使用。

当前处理器扩展至 64-bit 后，为保证前向兼容，32-bit 物理空间的模式依然得到保留（也即，为 PCI 设备保留的这段地址依然存在）。本实验构建的 JOS 操作系统只使用 256MB 的物理内存，因此所有的内容假设建立在 32-bit 物理空间上。

1.4 The ROM BIOS

PC 的启动流程基本可以概括为：1.CPU 正常启动；2. 读取并运行 BIOS；3.BIOS 进行设备初始化，加载 BootLoader；4.BootLoader 加载内核，并将控制流交给内核。

早期 BIOS 烧写在 ROM 中，一旦设置便不可修改。现代 PC 可以使用其他方案进行初始化，但统一归类为 BIOS 程序（或常见写作 BIOS Legacy）。BIOS 会将主板可以检测到的所有 IO 设备初始化，最后将控制交给加载完成的 BootLoader。

使用 GDB 进行程序追踪可发现，CPU 启动后第一条指令的位置：

```
[f000:fff0]    0xfffff0: ljmp    $0xf000,$0xe05b
```

可以看到，它正好落在上一节提到的 BIOS 地址空间中。

不同的 BIOS 基本功能都包括上电自检 (Power On Self Test, POST) 及主板设备的初始化。BIOS 系统按照一定顺序从硬盘或 USB 存储设备/网络上寻找操作系统，并从第一个扇区中找到 BootLoader，将控制交给它，开始对内核的加载。

Exercise 2

Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing.

这个 Exercise 是探究 BIOS 执行的指令及作用。BIOS 基本功能我们在本节中已阐述，具体的命令属于 dirty knowledge，此处不展开详细讨论。

Chapter 2

The Boot Loader

2.1 Loading the Kernel I: boot.S

BIOS 在找到可启动的设备后，会将设备中第一个扇区（512KB）的内容加载到 0x7c00 位置，并跳转至此处运行。这一个扇区中的代码一般用于将 OS 加载到内存中，称之为加载器 (*Boot-Loader*)。JOS 的 BootLoader 源码由 boot/boot.S 及 boot/main.c 两部分组成。

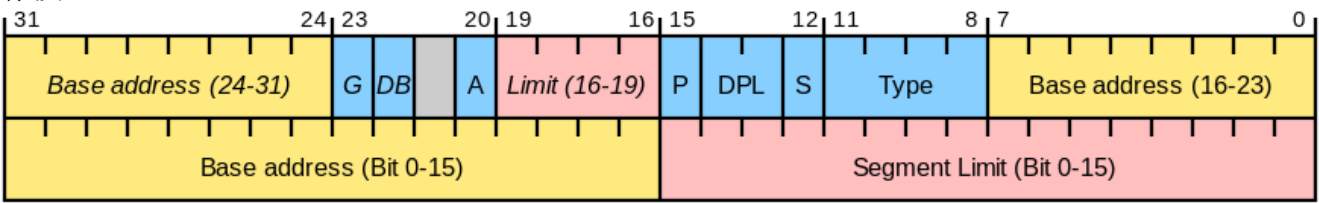
boot.S 文件中的源码主要完成了以下工作：1. 关闭中断；2. 段寄存器置 0；3. A20 地址线使能；4. 从实模式切换到保护模式，加载 GDT；5. 将控制交给 boot/main.c/bootmain()。

A20 线使能同样源于向前兼容性问题。8086/8088 处理器仅有 20 条地址线（实模式下 0-FFFF:FFFF 地址空间）；而在 80286 后的 x86 处理器地址线多于 20 条，为了向前兼容，在实模式下对 A20 线置零；当切换到保护模式后，解除地址空间的限制。

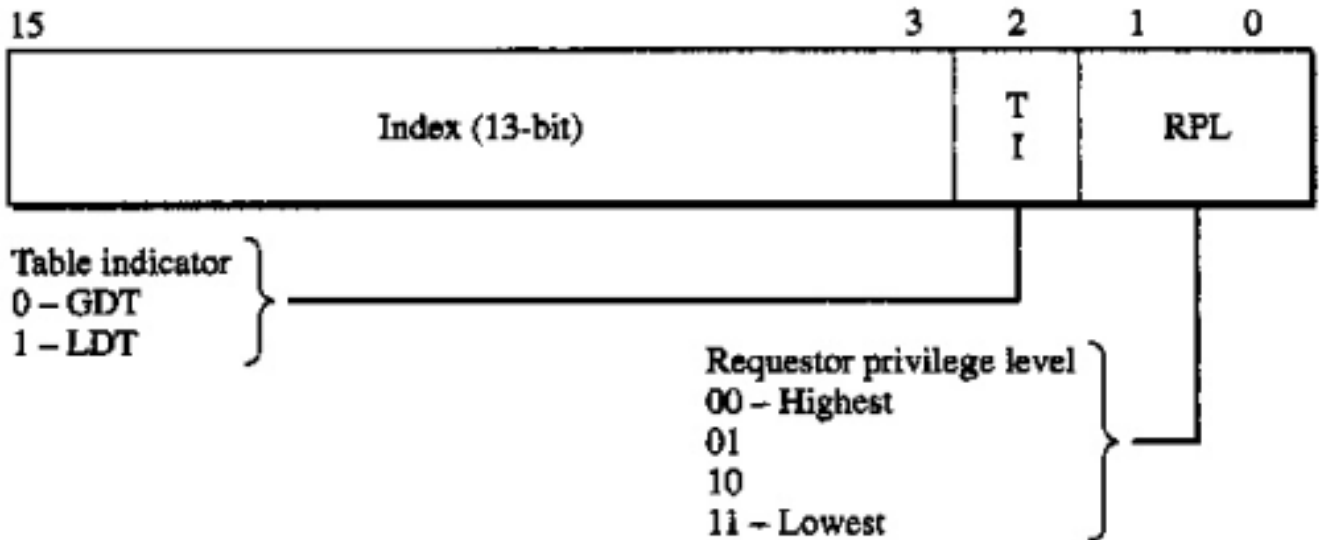
从实模式 (*Real Mode*) 切换到保护模式 (*Protect Mode*) 是此段源码的核心功能。

实模式是早期 x86 处理器的工作模式。当时的处理器使用 20 条地址线（1MB 的物理地址空间），而寄存器均为 16-bit 大小。为对整个地址空间的索引，使用了段基址：段偏移的分段策略：计算地址时将段基址左移 4 位后与偏移量相加，得到最终的物理地址，其中段基址一般由段寄存器给出。由于段基址和段偏移两个信息即可推出实际地址，故称为“实”模式。

保护模式 (*Protect Mode*) 是进入 32-bit 地址空间后，为了提高寻址安全性及灵活性所设计的新策略。为了与之前的架构体系相兼容，仍使用段寄存器组，但此时其中存放的不再是段基址，而是段选择子 (*Segment Selector*)。段选择子是使用全局描述符表 (*Global Descriptor Table, GDT*) 后产生的概念。GDT 中存放的表项也称为段描述符，其结构如下图所示，每个段描述符描述一个内存段。

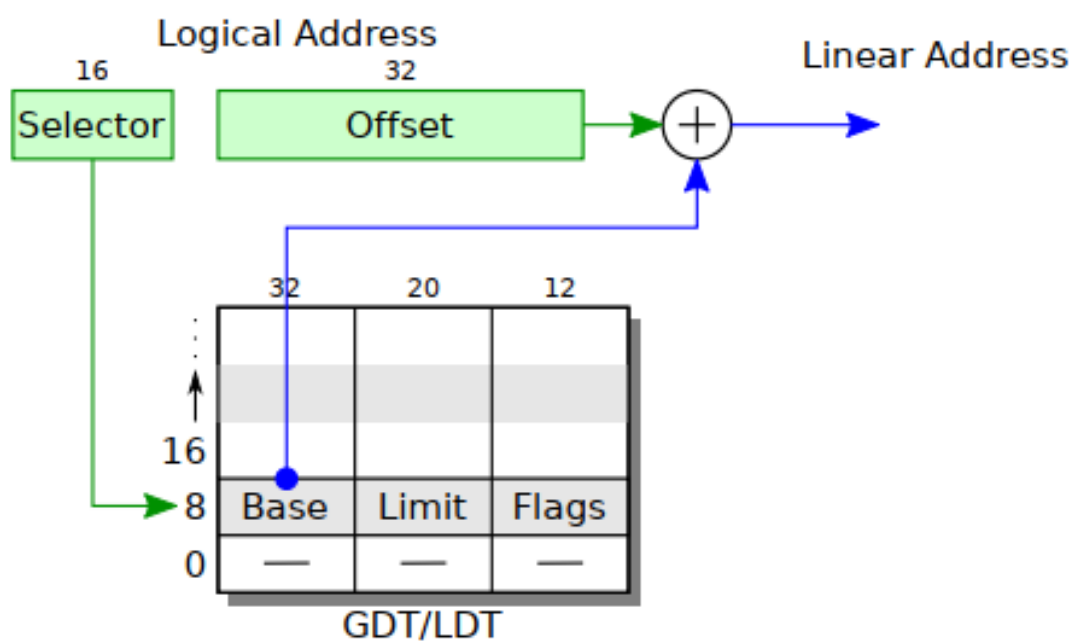


在此基础上，段选择子可看作 GDT 的索引，其结构如下图所示。



需要补充的是，除了 GDT 外，x86 架构提供另一个结构：本地描述符表 (*Local Descriptor Table, LDT*)。LDT 与 GDT 拥有相同的结构和类似的功能，两者的区别在于：GDT 主要是为操作系统提供内存索引服务，而 LDT 则为用户进程等提供相似的服务。因此，GDT 一般在操作系统中作为孤本保存，而 LDT 通常有多个不同副本，且 LDT 会被保存在 GDT 中。

结合上文，我们最终可以得到在 x86 保护模式下的寻址方式（如下图所示）。



2.2 Loading the Kernel II: bootmain.c

在 boot.S 完成准备工作后，控制权交到 boot/main.c/bootmain() 中。这部分源码的核心是将内核加载到内存中，并将控制权交给内核，完成启动。在本次实验中，内核以 ELF 文件形式存在。

加载内核的核心是 readseg() 和 readsect() 两个函数。根据 main.c 中提供的定义，内核 ELF

文件将被加载到物理地址 0x100000 处。bootmain() 会首先加载 ELF Header 部分检查文件是否有效并确认文件大小，最后再将整个内核加载到物理内存中。

Exercise 3

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in boot/boot.S and compare the original boot loader source code with both the disassembly in obj/boot/boot.asm and GDB.

这个 Exercise 让我们追踪从 0x7c00 开始的代码（即 BootLoader），并比较调试命令与 boot.asm（BootLoader 反汇编结果）。通过比较我们可以发现，反汇编文件中给出的命令地址与 GDB 调试时产生的地址是相同的：这说明，Boot 部分的逻辑地址与虚拟地址是相同的。我们可以通过 OBJDUMP 工具进行验证：

```
# objdump -h obj/boot/boot.out
boot.out:      file format elf32-i386

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          0000017e  00007c00  00007c00  00000054  2**2
    CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame      000000cc  00007d80  00007d80  000001d4  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab          000006d8  00000000  00000000  000002a0  2**2
    CONTENTS, READONLY, DEBUGGING
  3 .stabstr       000007df  00000000  00000000  00000978  2**0
    CONTENTS, READONLY, DEBUGGING
  4 .comment       00000011  00000000  00000000  00001157  2**0
    CONTENTS, READONLY
```

这与 Kernel 部分是不同的：

```
# objdump -h obj/kern/kernel
kernel:      file format elf32-i386

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
  0 .text          00004eae  f0100000  00100000  00001000  2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata        00001720  f0104ec0  00104ec0  00005ec0  2**5
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab          000089b9  f01065e0  001065e0  000075e0  2**2
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr       00009d41  f010ef99  0010ef99  0000ff99  2**0
    CONTENTS, ALLOC, LOAD, READONLY, DATA
```


4	.data	000c9f51	f0119000	00119000	0001a000	2**12
		CONTENTS, ALLOC, LOAD, DATA				
5	.bss	00000ef4	f01e2f60	001e2f60	000e3f60	2**5
		CONTENTS, ALLOC, LOAD, DATA				
6	.comment	00000011	00000000	00000000	000e4e54	2**0
		CONTENTS, READONLY				

Exercise 4

Read about programming with pointers in C.

阅读任务，略。

Exercise 5

Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong.

修改 MakeFlag 文件中 BootLoader 的链接地址 (0x7c00)，再追踪出现 bug 的指令。由于 BIOS 对 BootLoader 加载的位置是硬编码的，因此实际上 BootLoader 仍加载到了 0x7c00 处，并跳转到此处进行。因此理论上，BIOS 可以顺利地将控制转交给 BootLoader。问题出现在链接地址的改变对 BootLoader 链接时重定向的改变：当链接器根据修改后的链接地址重定向后，如源码中的 lgdt 和 ljmp 命令的操作数都将出现问题。lgdt 命令处出现问题，但 JOS 依然成功加载了 GDT，最后是在 ljmp 指令处发生了错误。

Exercise 6

Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint?

在 BIOS 引导进入 boot loader 前，0x0010000 处内容为空；进入 kernel 前，0x0010000 处已经有了内容，前后的区别在于 kernel 被加载进来了。

Question

At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?

What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?

Where is the first instruction of the kernel?

How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?

在 **ljmp \$PROT_MODE_CSEG, \$protcseg** 处切换到了保护模式。在此之前 BootLoader 设置好了 GDT, **ljmp** 命令修改 CS 段寄存器值并进行长跳转。

BootLoader 的最后一条命令是 **call bootmain** , 最后一条命令是 **((void (*)(void)) (ELFHDR->e_entry))();** 。

根据 BootLoader 的反汇编结果, 其最后一条指令是 **call *0x10018** , 我们可以得到 Kernel 的第一条指令是 **movw \$0x1234,0x472** (对应于 Kernel 的源码: kern/entry.S)。

根据 ELF Header 提供的信息判断读取节数。

Chapter 3

The Kernel

3.1 Using virtual memory to work around position dependence

在前面的内容中我们已经知道，在 BootLoader 阶段，虚拟地址与物理地址是完全相同的（OBJDUMP 对 Boot 的反汇编结果）。而 Kernel 则不同，它会将自己映射到 32-bit 物理地址空间的高位：以 JOS 为例，它将内核映射到 0xf0100000 虚拟地址处。由于许多 PC 并不具有如此大的物理内存，因此会选择将高位虚拟地址映射到低位物理地址。

JOS 在启动真正的分页服务前，会使用 kern/entrypgdir.c 进行初映射，将高位/低位的 4MB 虚拟地址全部映射到低位 4MB 物理地址。

Exercise 7

Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at 0x00100000 and at 0xf0100000.

设置 CR0 寄存器后，处理器进入分页模式，运行上述命令后 entrypgdir.c 设置的分页生效。因此在此处断点检查会发现两个地址内容不同，高位地址为空；步进后再次检查，发现两地址内容相同，因为它们均映射到低位物理地址处。

3.2 Formatted Printing to the Console

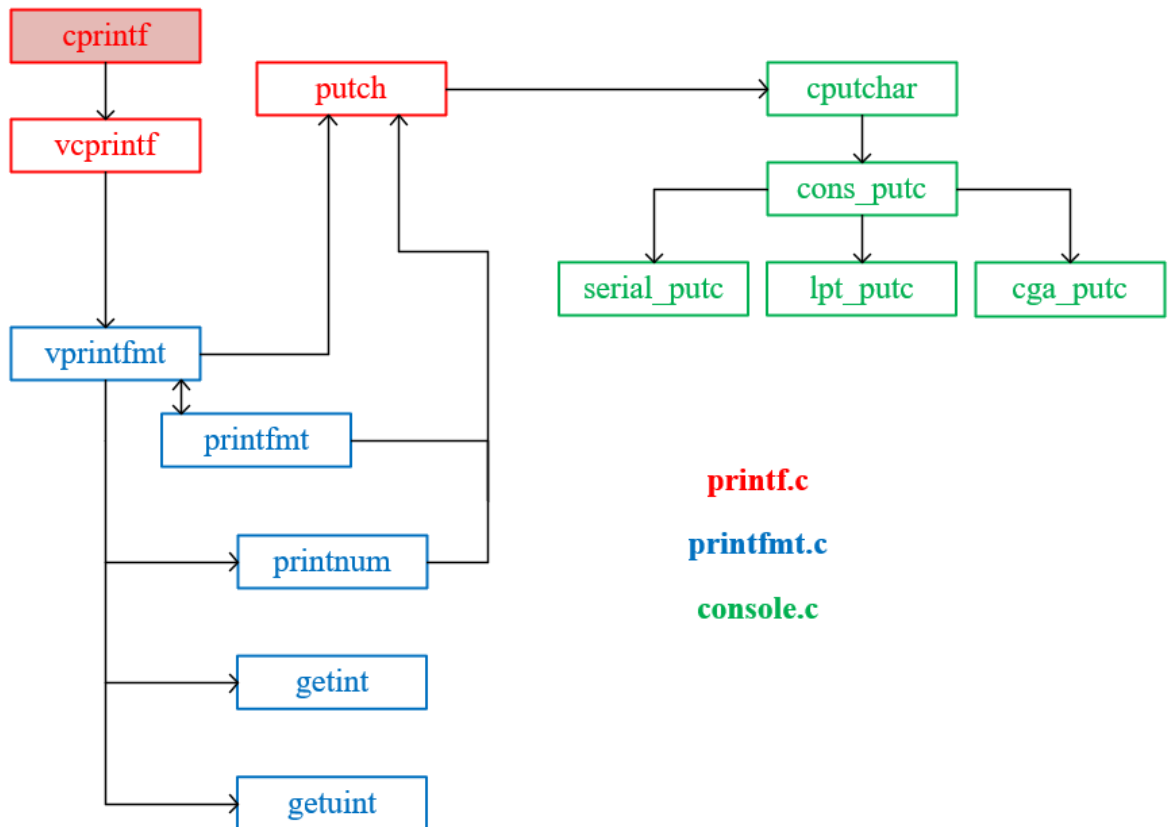
这一节要求我们阅读 kern/printf.c, lib/printfmt.c 和 kern/console.c 的源码，理解它们之间的关系。

分析较高调用层（如 kern/init.c）中对输出函数的使用，我们可以知道用户层的接口应该是 cprintf。本节内容结合 Question 及 Exercise 8 来展开。

注：*cpprintf* 与 *printf* 的区别在于后者可以被重定向到输出屏幕之外的位置（标准输出流），前者只能输出到控制台。

Question

1. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?



根据对三个文件的分析，我们可以大致得到调用关系如上图。其中 `printf.c` 提供最外层的用户接口，并调用由 `printfmt.c` 及 `console.c` 提供的底层实现接口；`console.c` 主要提供了 `cputchar` 函数，它调用底层接口将字符输出到控制台（屏幕）上。

2. Explain the following from `console.c`:

```
if (crt_pos >= CRT_SIZE)
{
    int i;
    memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
    for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
        crt_buf[i] = 0x0700 | ' ';
    crt_pos -= CRT_COLS;
}
```

这段代码出现在 `cga_putc()` 函数中。CGA 指的是彩色图形适配器 (Color Graphics Adapter)，这个函数用于将字符串显示到 CGA 设备上。根据代码分析，猜测为当输出内容溢出屏幕显示范

围时，将终端上第一行移出屏幕，其他行整体上移，并将新数据写到显示器最后一行上。

3. Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

(i) In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?

(ii) List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `vcprintf` list the values of its two arguments.

(i) `cprintf` 中的 `fmt` 参数指向待输入的格式串；`ap` 指向参数 `x,y,z` 构成的可变参数列表 (1,3,4)。

(ii) `cons_putc` 的参数是每次打印到控制台的字符 ASCII 码；每次调用 `va_arg` 后，`va_list` 参数会逐渐减少至空；`vcprintf` 的参数为：“x %d, y %x, z %d\n”，(1,3,4)。

4. Run the following code.

```
unsigned int i = 0x00646c72;
cprintf("H%x Wo%s", 57616, &i);
```

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise.

`%x` 是以 16 进制打印相应整数，`DEC(57616)=HEX(110)`；`%s` 为打印字符串，而 `i` 以无符号整形存储（4 字节），且 x86 为小端机器，故 4 个字节依次为 `HEX(72)`，`HEX(6c)`，`HEX(64)`，`HEX(00)`，分别对应 ASCII 码的 `r`，`l`，`d` 和空字符，故最终打印结果为：HE110 Wolrd。

5. In the following code, what is going to be printed after ‘y=’? (note: the answer is not a specific value.) Why does this happen?

`y` 并没有指定参数，故非固定值。实际情况是，指令将从栈中取出变量 3 之后的一个整形长度值。

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

将变长数组参数与格式串参数位置进行调换即可。

Exercise 8

We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

根据前文对打印函数体系调用结构的分析,我们知道处理转义字符的分支存在于 printfmt.c/vprintfmt() 函数中。因此我们在其中加入处理八进制的分支,仿照处理十六进制的分支进行补充:

```
case 'o':
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
```

3.3 The Stack

本节的内容主要关于 C 程序对 x86 的栈使用,并讨论了 JOS 的内核栈相关内容。

Exercise 9

Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

在 kern/entry.S 中的一行代码初始化了内核栈:

```
# Set the stack pointer
movl $(bootstacktop),%esp
```

\$bootstacktop 变量定义在同一文件下,在切换到.data 段后定义:

```
.data
#####
# boot stack
#####
    .p2align PGSHIFT    # force page alignment
    .globl    bootstack
bootstack:
    .space    KSTKSIZE
    .globl    bootstacktop
bootstacktop:
```

使用 GDB 进行断点调试，链接后地址为 0xf0117000（虚拟地址）；通过 .space 指令，内核初始化了一个大小为 $KSTACKSIZE=8*PGSIZE=32KB$ 的区域作为栈生长空间，而栈顶位于 0xf0117000，向低位地址生长。

在 32-bit 的 x86 体系中，栈一般只存储 32-bit 的数据——也即，栈是 4 字节对齐的，因此存放栈顶地址的 ESP 寄存器的值应当是可被 4 整除的。包括 call 在内的一系列 x86 指令，都会默认使用 ESP 所指的栈（Hard-wired 的）。

除了 ESP 寄存器外，x86 体系中还存在一个称为栈帧指针 (Stack Frame Pointer) 的 EBP 寄存器，协助对栈进行维护：当 C 程序发生函数调用时，它会将当前函数的 EBP 寄存器值入栈，并将当前 ESP 值复制到 EBP 寄存器中。有了这一规则的补充，我们可以利用栈对调用链进行回溯。

Exercise 10

To become familiar with the C calling conventions on the x86, find the address of the test_backtrace function in obj/kern/kernel.asm, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of test_backtrace push on the stack, and what are those words?

这个练习的目的是帮助我们了解 C 程序的过程调用规则。我们设置断点对 test_backtrace 进行跟踪。这个函数将递归调用自己，每次被调用时，参数都将减一。根据步进调试，我们可以得到该函数调用的流程：

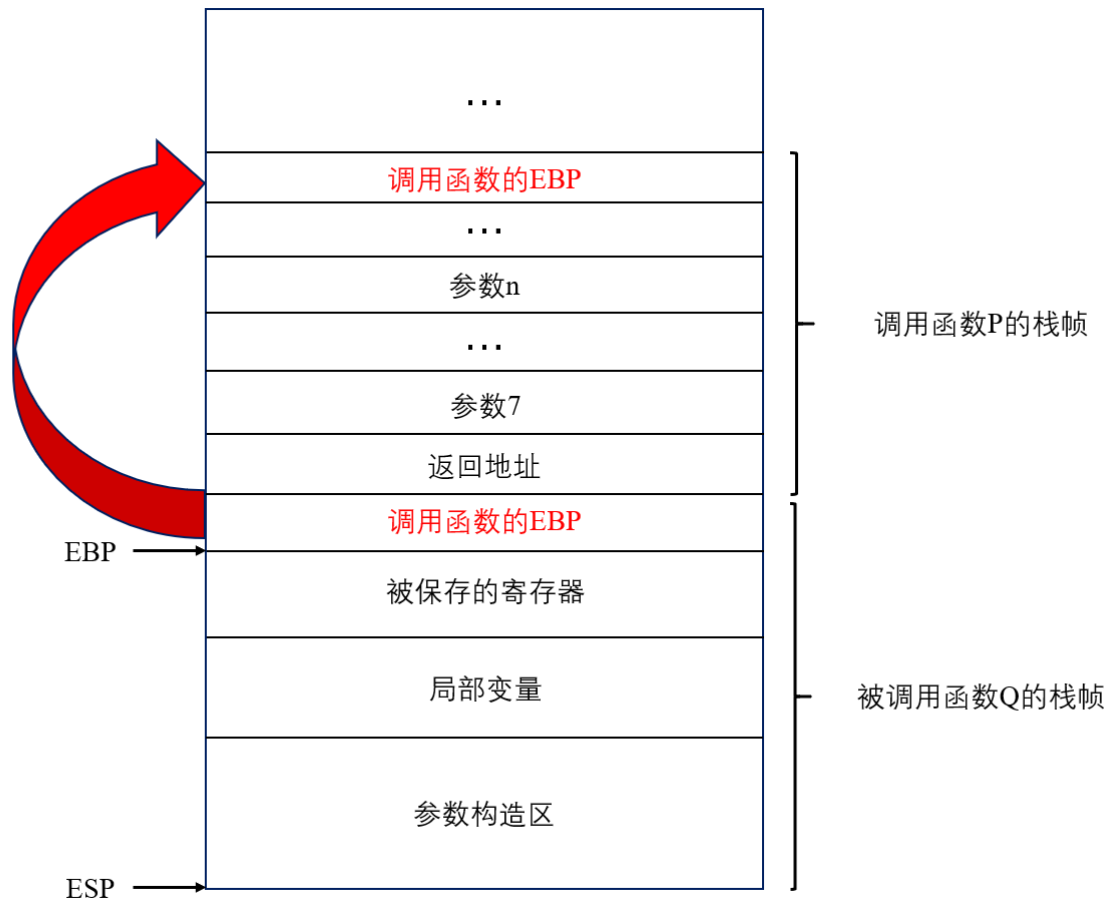
1. call 指令将返回地址入栈（%esp 值减少 4 字节）
2. 将 %ebp 入栈（%esp 值减少 4 字节，累计 8 字节）
3. 将 %esp 值赋给 %ebp
4. 将 %ebx 入栈（约定保护的寄存器值）（%esp 值减少 4 字节，累计 12 字节）
5. 将 %esp 减去 20(0x14) 个字节（保证地址 4 字节对齐）。

由此可得，每次调用 test_traceback 函数时，都将产生一个 32 字节的栈帧，故每次调用有 8 个 32-bit 字入栈。

上面的 Exercise 提供了一个栈帧的实例，我们进一步分析栈在过程调用中的变化。

大部分的过程使用的栈帧都是定长的：也即，在编译过程就确定并分配了定长栈帧。当发生调用时，首先有返回地址和 EBP 寄存器值依次入栈，并修改 EBP 值；同时，它将需要保存的寄存器入栈（如上例中的 EBX），并将调用参数放入已保存的寄存器中，用于数据传送；随后为局部变量划定空间并入栈。我们知道，x86 体系用于传递参数的寄存器仅有 6 个，若需要更多的参数，可在栈顶继续逆序构造（第 7 个参数将位于栈顶）。

由于 C 本身支持定长数组，且 C 标准库中包含像 alloca 这样提供栈空间分配的函数，因此栈也需要支持变长特性，这一特性可以依赖 EBP 实现。在早期 x86 代码中，每个函数都使用帧指针。如今只有栈帧长可变的情况下才使用。



Exercise 11

The backtrace function should display a listing of function `call` frames in the following format:

Stack backtrace:

```
ebp f0109e58 eip f0100a62 args 00000001 f0109e80 f0109e98 f0100ed2 00000031
```

```
ebp f0109ed8 eip f01000d6 args 00000000 00000000 f0100058 f0109f28 00000061
```

...

Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused.

这是要求我们实现 `backtrace` 功能，其原理依据正是我们上文对栈帧结构的分析。实际需要填写的函数是 `mon_backtrace`，并可以使用 `read_rbp()` 接口获得 `EBP` 寄存器的值：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    cprintf("Stack backtrace:\n");
```



```

uint32_t* _ebp = (uint32_t*)read_ebp();
while(_ebp != NULL)
{
    cprintf("ebp %08x ",_ebp);
    uint32_t _eip = *(_ebp + 1);
    cprintf("eip %08x args",_eip);

    uint32_t _tmp;
    for(size_t i = 0; i < 5;++i)
    {
        _tmp = *(_ebp + i + 2);
        cprintf(" %08x",_tmp);
    }
    cprintf("\n");
    _ebp = (uint32_t*)(*_ebp);
}
return 0;
}

```

Exercise 12

Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.

In debuginfo_eip, where do __STAB_ come from? This question has a long answer.*

这个 Exercise 要求我们利用符号表信息进一步增加 backtrace 的功能。提示中的 __STAB_* 的相关定义指的是符号表 (Symbol Table, stab)，我们可以使用 OBJDUMP 查看这张符号表：

```

# objdump -G obj/kern/kernel

obj/kern/kernel:      file format elf32-i386
Contents of .stab section:
Symnum n_type n_othr n_desc n_value  n_strx String
-1      HdrSym 0      3594    0000a955 1
0       SO     0      0      f0100000 1      {standard input}
1       SOL    0      0      f010000c 18     kern/entry.S
...
14      SO     0      2      f0100040 31     kern/entrypgdir.c
15      OPT    0      0      00000000 49     gcc2_compiled.
...

```

kern/kdebug.c 中提供了两个辅助函数接口：stab_binsearch 和 debuginfo_eip，前者用于利用地址查找符号表，后者则调用该函数，用于填充信息。我们补全 debuginfo_eip 函数：

```
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
info->eip_line = lline > rline ? -1 : stabs[rline].n_desc;
```

最后利用这个接口补全 backtrace 的功能。

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    // The data structure to save information about EIP
    struct Eipdebuginfo info;
    uint32_t arg[4];
    uint32_t* ebp = (uint32_t*)read_ebp();
    while(ebp != NULL)
    {
        uint32_t eip = *(ebp + 1);
        for(int i = 0; i < 4; i++)
            arg[i] = *(ebp + i + 2);
        cprintf("ebp %08x eip %08x args ", ebp, eip);
        for(int i = 0; i < 4; i++)
            cprintf("%08x ", arg[i]);
        cprintf("\n");
        if(debuginfo_eip(eip, &info) == 0)
        {
            cprintf("%s:%d: ", info.eip_file, info.eip_line);
            cprintf("%.s", info.eip_fn_namelen, info.eip_fn_name);
            cprintf("+%d\n", eip - (uint32_t)info.eip_fn_addr);
        }
        else
            cprintf("Error happened when reading symbol table\n");
        ebp = (uint32_t*) (*ebp);
    }
    return 0;
}
```