OSlab1 文档

5140379064　　陈俊

一、启动流程：

最初，CS = 0xf000，IP = 0xfff0，即位于 0xffff0 处。通过 ljump 指令，跳到了 0xfe05b 处，进行了一系列硬件设置（如初始化 DMA 等），将 boot 启动程序加载到内存 0x7c00，并通过 jmp 指令跳到 0x7c00 处开始运行 boot 程序。

Boot 程序初始化了 GDT，并且通过修改 cr0 上 PE 的方式，将系统从 16-bit real mode 切换到 32-bit 的保护模式,并且修改了一系列的段寄存器,之后调用了 bootmain 函数。在 bootmain 函数中，先往内存中映射启动 Kernel 的 ELF 的 header，并根据 header 往内存中读入该 ELF 文件的各段数据，之后启动 Kernel，调用 entry 函数。

在 entry 函数中，开启了页表机制，加载了 cr3，同时把 cr0 的 PG 设为开启。在加载 cr3 过程中,程序已将物理地址 0-4MB 映射到虚拟地址的 0-4MB 以及 KernelBase - KernelBase+4MB。之后使用了 jmp 指令，因为在 kernel.ld 中 link address 设置为 0xf0100000，就完成了到高虚拟地址的转化。同时，也设置了一个简单的栈，方便之后函数的调用。这就是大致的启动流程。

二、Printf 相关

Printf 是一个典型的 c 不定长参数函数，在这个 lab 中，我们看到了它的实现。在 printf.c 中，cprintf 获得了参数 fmt，并把之后的参数放在 va_list 中，并将这两个参数传给 vcprintf 函数。vcprintf 函数调用了 lib/printfmt.c 中的 vprintfmt 函数，用于对 fmt 进行解析，并将 putch 函数作为一个函数指针参数传了进去。putch 函数用于字符的打印，它是 cputchar 的一个封装，调用 console.c 的 cputchar，并进行打印字数的统计。

在 lab 中，我主要修改了 printfmt.c 文件。对于八进制的实现，只需要按照十六进制的写法，在开头通过 putch 打印 0 字符，并通过 getuint 获得 va_list 中的数字，并将 base 改为 8，之后 goto number 调用 printnum 函数实现。在 getuint 中，通过调用 va_arg 函数，获得当前参数对应类型的值，并将 va 指针移到下一个参数。而对于正数的‘+’显示，只需要添加一个 flag，当有‘+’时 flag 设为 1，在 case d 的情况下进行额外的判断即可。

对于%n 的实现，只需要将当前打印的字符写入参数中对应的地址即可。在 putch 中,我们额外对打印的个数进行了累加,而这个累加的值是 vcprintf 函数中的 count，也就是 vpirntfmt 中的 putdat 参数。因此，只需要通过 va_arg 获得地址，把 putdat 写入地址即可。当然还需额外判断两种错误情况，溢出由于是 signed char，所以条件应变成大于 127 而不是 255。

在 printnum 中，数字根据进制一个个打印，并且根据 padc 在前方添加占位符以符合宽度。我需实现通过‘-’号，让其在后方添加占位符。但根据它原本的递归调用，占位符永远只能出现在前方。因此，我新写了一个递归函数，而将原函数当作最外层的一个壳。在原函数中调用新函数，而新函数递归调用新函数。这样相当于原函数只会被调用一次。若 padc 不是‘-’,新函数和原函数一切照旧。若是‘-’号，则在最后添加位于尾部的空格占位符。这样可以保证只有最外层的函数会在‘-’号时在尾部添加空占位符，也就实现了题目的要求。

三、Stack

完成这部分 lab 需要我掌握栈帧的分布。在调用函数时，我们首先会将参数从最后到第一个依次压栈，然后调用 call 指令，压入返回地址，进入新函数的栈帧。最开始调用的两条指令一定是 push %ebp，movl %esp，%ebp，因此新调用函数的 ebp 里，存放着上一个函数的 ebp 值，再之后四个字节是新函数的返回地址，之后是传入的参数。因此，只需通过 read_ebp 读到现在的 ebp，往上读返回地址和 5 个参数（虽然不一定 5 个），之后把 ebp 改为 ebp 里存着的旧的 ebp，循环，就可以完成 backtrace。

之后，需要打印 debug 信息，要调用 kdebug.c 中的 debuginfo_eip。通过查询 stab 文档，知道 N_SLINE 是查找行数的参数，并且行数存储在 Stab 类的 n_desc 中，在 debuginfo_eip 中添加这部分行数的功能，并在 backtrace 中调用 debuginfo_info，传入返回地址，打印相关信息，就可以实现需要的功能。

下一个要求类似于 ICS 课程中的 lab3，需要修改栈帧中的返回地址。题目要求 overflow_me 要调用 start_overflow，从 start_overflow 调用 do_overflow,并且需要正常返回。我们知道，overflow_me 调用 start_overflow,栈帧中 ebp 所处位置的上方存放着返回地址，为使得 start_overflow 返回 do_overflow，需要将原返回地址改为 do_overflow 的地址。同时，为了正常返回，进入 do_overflow 后，它的返回地址又是再上面四个字节，也就是 pret_addr+4，因此需要将原返回地址挪入这个新的地址，就可以完成正常返回。题目要求用 printf 中的%n，因此只能 1 个 byte1 个 byte 写，先把原返回地址保存起来，把新地址改为 do_overflow 地址，新地址上方四个字节（+4）改为原返回地址，分别用%n 写入 8 次，就可以达到目的。

四、Time 命令实现

题干中提到使用 rdtsc，这是一条汇编指令，获得 CPU 的 cycles。因此，我借鉴了 read_pretaddr 的写法，在 retsc 函数内调入了 rdtsc 汇编指令，获得一个 uint64_t 的值。之后，将 argv[1]与 command 中的命令比较，如果匹配到名字相同，就调用该命令的相关函数，argc 变为 argc-1,argv 变为原先 argv[1]的地址。在调用前后分别调用 rdtsc，将 cycles 数相减，就获得了相差的 cycle 数。再将 time 指令挂载到 commands 中，就可以在 kernel 中调用 time 函数了。