Lab1 Document

5150809010012

程浩

代码解释：

1. 实现%o 打印八进制数

```
// (unsigned) octal
case 'o':
  // Replace this with your code.
  // putch('X', putdat);
  // putch('X', putdat);
  // putch('X', putdat);
  putch('0', putdat);
  num = getuint(&ap, lflag);
  base = 8;
  goto number;
  break;
```

2. 实现%d 中，‘d’前有‘+‘显示打印的符号（showFlag 是检测前面是否有+号以打印出符号）

```
case '+':
  padc = '+';
  showFlag = true;
  goto reswitch;
```

```
// (signed) decimal
case 'd':
  num = getint(&ap, lflag);
  if ((long long) num < 0) {
    putch('-', putdat);
    num = -(long long) num;
  }
  // code I add to solute sign
  else if (showFlag)
  {
    putch('+', putdat);
  }
  base = 10;
  goto number;
```

3. 实现%d 中，‘d’前有‘-’将所需打印字符居左并以‘-’后的数字个空格填充

```c
// your code here:
if(padc=='-')
{
    padc = ' ';
    uint64_t baseNum = 0;
    int numlen = 0;

    // first change the base of the number
    for(; num >= base; ++numlen , num /= base)
    {
        baseNum *= base ;
        baseNum += num % base;
    }
    putch("0123456789abcdef"[num],putdat);

    //then print the number
    for( width -= numlen ; numlen > 0 ; --numlen, baseNum /= base)
    {
        putch("0123456789abcdef"[baseNum % base], putdat);
    }
    // the other is fiiled with space
    while (--width > 0)
        putch(padc, putdat);
    return ;
}
// first recursively print all preceding (more significant) digits
if (num >= base) {
    printnum(putch, putdat, num / base, base, width - 1, padc);
} else {
    // print any needed pad characters before first digit
    while (--width > 0)
        putch(padc, putdat);
}
```

4. 实现%n，打印 char*类型的变量，并添加对空指针和溢出的处理

```c
const char *null_error = "\nerror! writing through NULL pointer! (%n argument)\n";
const char *overflow_error = "\nwarning! The value %n argument pointed to has been overflowed!\n";

// Your code here

char *posPtr ; //position pointer
if ((posPtr = va_arg(ap, char *)) == NULL)
{
    printfmt(putch,putdat,"%s", null_error);
}
else if(*((uint32_t *)putdat) > 254 )
{
    printfmt(putch, putdat,"%s",overflow_error);
    *posPtr = -1;
}
else
{
    *posPtr = *(char *)putdat;
}
break;
```

5. 利用 kedebug.c 中提供的 debuginfo_eip，并对它进行部分修改，实现完整的

debuginfo 函数，并按照 lab1 文档中所给的输出格式输出

```
// Your code here.

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

```
// Your code here.

int *ebp = (int *)read_ebp();
int eip = *(ebp + 1);
cprintf("Stack backtrace:\n");
struct Eipdebuginfo info;


while(ebp)
{
cprintf("  eip %08x  ebp %08x  args %08x %08x %08x %08x %08x\n",
    eip, ebp, ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);
debuginfo_eip((uintptr_t)eip, &info);
cprintf("         %s:%u %.*s+%u\n",
    info.eip_file, info.eip_line, info.eip_fn_namelen, info.eip_fn_name, eip - (int)info.eip_fn_addr);
ebp = (int *)(*ebp);
eip = *(ebp + 1);
}
overflow_me();
cprintf("Backtrace success\n");
return 0;
```

6. 在 do_overflow 中替换 return address（hint：

```
// hint: You can use the read_pretaddr function to retrieve
//       the pointer to the function call return address;

char str[256] = {};
int nstr = 0;
char *pret_addr;

// Your code here.

pret_addr = (char *) read_pretaddr();
nstr = (uint32_t)do_overflow;
int i;
for (i = 0; i < 4; ++i)
  cprintf("%*s%n\n", pret_addr[i] & 0xFF, "abcdefg", pret_addr + 4 + i);
for (i = 0; i < 4; ++i)
  cprintf("%*s%n\n", (nstr >> (8*i)) & 0xFF, "abcdefg", pret_addr + i);
```

部分 exercise 解答：

Exercise 2

代码执行后可以发现，BIOS 的主要功能就是在控制，初始化，检测各种底层设备，比如时钟，GDTR 寄存器，以及设置中断向量表。其中，他作为 PC 其中后第一段程序，他的重要功能就是负责把操作系统导入内存，然后把控制权交给操作系统。当在磁盘中找打了操作系统，BIOS 就会把该磁盘的第一个扇区（启动区）先加载到内存中，其中，一个很重要的 boot loader 程序会负责将操作系统导入到内存中。

Exercise 3

- 首先，在 boot.S 文件中，计算机工作在实模式下，此时地址为 16bit 模式；当执行了 long jump 后（也就是这一句 ljmp $PROT_MODE_CSEG, $protcseg），正式进入 32bit 地址模式。
- 它之行的最后一条指令是 boot/main.c 中 bootmain 中的最后一条"((void (*)(void)) (ELFHDR->e_entry))();"，即跳转到操作系统内核的起始指令处；该程序第一条指令位于 kern/entry.S 中，"movw $0x1234,0x472 "。
- 操作系统中的扇区信息都是存放在操作系统文件中的 program header table 中的，在这个表中的每个表项（对应操作系统某一段）的内容包括了这个段的大小，段起始地址的 offset 等信息，所以找到这个表就能确认要读取多少扇区；这个表一般存放在操作系统内核文件的 ELF 头部中。

Exercise 4

```
jos@cosmic:~/桌面/OSlab/jos-2019-spring$ ./pointers
1: a = 0x7ffe175db8f0, b = 0x55d81af29260, c = 0x7fa9f61d7a95
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7ffe175db8f0, b = 0x7ffe175db8f4, c = 0x7ffe175db8f1
```

Exercise 5

BIOS entered the boot loader.

```
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

Boot loader entered the kernel.

```
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x3000b812      0x220f0011      0xc0200fd8
(gdb)
```

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

因为在 bootmain 执行的最后，它会将各个程序段送到地址 0x100000 处，由上图以及程序入口地址 0x10000c 可推测，这里面存放的是指令段 .text 的内容。

```
(gdb) b *0x7d71
Breakpoint 1 at 0x7d71
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x7d71:      call   *0x10018

Breakpoint 1, 0x00007d71 in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
```

Exercise 6

将 Makefrag 中的 boot loader 链接地址从原先的 0x7C00 修改为 0x7D00

```
$(OBJDIR)/boot/boot: $(BOOT_OBJS)
	@echo + ld boot/boot
	$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o $@.out $^
	$(V)$(OBJDUMP) -S $@.out >$@.asm
	$(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
	$(V)perl boot/sign.pl $(OBJDIR)/boot/boot
```

```
(gdb)
[   0:7c2a] => 0x7c2a:  mov    %eax,%cr0
0x00007c2a in ?? ()
(gdb)
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87c32
0x00007c2d in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c32:      mov    $0x10,%ax
0x00007c32 in ?? ()
```

修改前

```
(gdb)
[   0:7c2d] => 0x7c2d:  ljmp   $0xb866,$0x87d32
0x00007c2d in ?? ()
(gdb)
[f000:e05b]    0xfe05b: cmpw   $0x48,%cs:(%esi)
0x0000e05b in ?? ()
```

修改后

可以发现在 long jump 时发生了错误，跳到不知道哪儿去了。

Exercise 7

```
(gdb)
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000      0x00000000
```

movl　　%eax, %cr0 执行前

```
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/4x 0x100000
0x100000:       0x02    0xb0    0xad    0x1b
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x02    0xb0    0xad    0x1b
```

movl　　%eax, %cr0 执行后

发现，在执行过后存放在内核虚拟地址 0xf0100000 中的内容已经映射到了物理地址 0x100000 中；接着，注释掉 movl　　%eax, %cr0，编译后调试

```
(gdb) x/4x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start+4026531828>: 0x00000000      0x00000000      0x00000000      0x00000000
```

由于此时没有将内核的虚拟地址值映射到低地址，所以执行前后无变化

```
(gdb) si
=> 0xf010002c <relocated>:      add    %al,(%eax)
relocated () at kern/entry.S:74
74              movl    $0x0,%ebp                       # nuke frame pointer
(gdb) x/4x $ebp
0x7bf8: 0x00000000      0x00007c4a      0xc031fcfa      0xc08ed88e
(gdb) x/4x *$ebp
Attempt to dereference a generic pointer.
(gdb) x/4x $ebp
0x7bf8: 0x00000000      0x00007c4a      0xc031fcfa      0xc08ed88e
(gdb) si
Remote connection closed
```

但是在执行到 movl　　$0x0,%ebp 这个语句时发生了错误，原因是超出了访存范围。

Exercise 8&9

1. Console.c export 了除 static 修饰的函数，其中 printf 中的 putch()调用了 console.c 中的 cputchar。
2.
3.
4.
5.
6.