MIT 6.828: Operating System Engineering

Lab Report
Lab 1: C, Assembly, Tools, and Bootstrapping

CSNLP
csnlp16@126.com
https://csnlp.github.io/

Contents

1 Environment Setup

Our environment: 1. Host: Mac Air 2. Virtualbox: Ubuntu 14.04

Install The QEMU MIT6.828 uses QEMU as simulator. To install QEMU, we use the command like: sudo apt-get install qemu

2 Part 1: PC Bootstrap

Our environment: 1. Host: Mac Air 2. Virtualbox: Ubuntu 14.04

Install The QEMU MIT6.828 uses QEMU as simulator. To install QEMU, we use the command like: sudo apt-get install qemu

2.1 Getting Started with x86 assembly

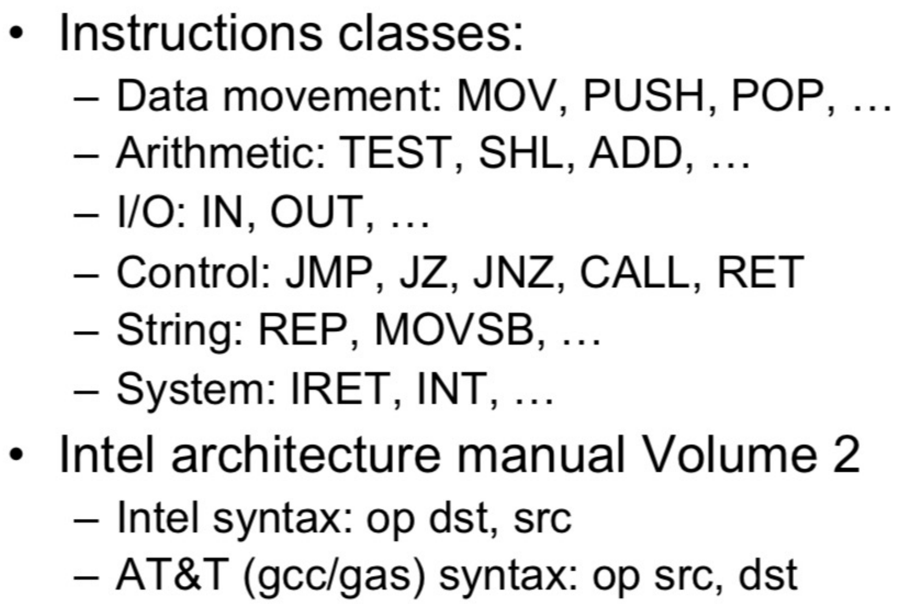


Figure 1: This is a caption

2.2 Simulating the x86

2.3 XX

2.4 The PC's Physical Address Space

I believe PC physical address space is considerably important.

2.5 The ROM BIOS

2.5.1 Exercise 2

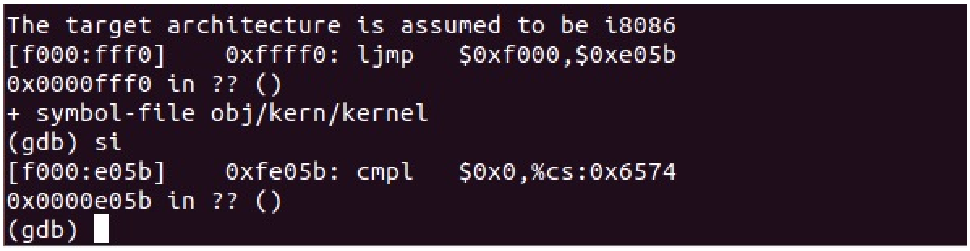
Use GDB's si (Step Instruction) command to trace into the ROM BIOS for a few more instructions, and try to guess what it might be doing. You might want to look at Phil Storrs I/O Ports Description, as well as other materials on the 6.828 reference materials page. No need to figure out all the details - just the general idea of what the BIOS is doing first.

Here is the experiment steps:

1. Open a terminal and type: make qemu-gdb.

2. Open another terminal and type: make gdb

We can see the output as follows:



```
The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file obj/kern/kernel
(gdb) si
[f000:e05b]    0xfe05b: cmpl   $0x0,%cs:0x6574
0x0000e05b in ?? ()
(gdb)
```

Figure 2: This is a caption

3 Part 2: The Boot Loader

The boot loader must do

1. Switchs from the real-mode to 32-bit protected mode.
2. Reads the kernel from the hard disk by directly accessing the IDE disk device registers via the x86's special I/O instructions.

The first assembly instruction is

```
[f000: fff0 ] 0xffff0 : ljmp $0xf000 ,$0xe05b
```

Listing 1: The first assembly instruction of boot program

The first instruction is to jump from the **real mode** to **protective mode**. With si instruction, we can see the following instructions:

```
[f000:e05b]    0xfe05b: cmpl    $0x0,%cs:0x6574
[f000:e062]    0xfe062: jne     0xfd2b6
[f000:e066]    0xfe066: xor     %ax,%ax
[f000:e068]    0xfe068: mov     %ax,%ss
[f000:e06a]    0xfe06a: mov     $0x7000,%esp
[f000:e070]    0xfe070: mov     $0xf3c24,%edx
[f000:e076]    0xfe076: jmp     0xfd124
[f000:d124]    0xfd124: mov     %eax,%ecx
[f000:d127]    0xfd127: cli
[f000:d128]    0xfd128: cld
[f000:d129]    0xfd129: mov     $0x8f,%eax
[f000:d12f]    0xfd12f: out     %al,$0x70
[f000:d131]    0xfd131: in      $0x71,%al
[f000:d133]    0xfd133: in      $0x92,%al
[f000:d135]    0xfd135: or      $0x2,%al
```

Listing 2: The first assembly instruction of boot program

The question asked in

1. At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
   Answer: we can see in **boot/boot.S** that

```
ljmp    $PROT_MODE_CSEG, $protcseg
```

Listing 3: The switch from 16 to 32 bit mode

2. What is the last instruction of the boot loader executed, and what is the first instruction of the kernel it just loaded?
   Answer: it the command in **boot/main.c**

```
((void (*)(void)) (ELFHDR->e_entry))();
```

Listing 4: Last instruction of boot loader

Additional Information:

3. Where is the first instruction of the kernel?
4. 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?

3.1 Loading The Kernal

3.1.1 What's ELF

3.1.2 Exercise 5

*[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. Then change the link address in **boot/Makefrag** to something wrong, run **make clean**, recompile the lab with **make**, and trace into the boot loader again to see what happens. Don't forget to change the link address back and **make clean** again afterward!]*

We can see from **boot/Makefrag** that

```
$(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
```

Listing 5: makefrag

We do the following operations

1. Open a terminal and type **make qemu-gdb**
2. Open a second terminal and type **make gdb**
3. Set a breakpoint and continue execution to this breakpoint: type in the second terminal that **b *0x7c00** and **c**
4. Repeat the command **si** in the second terminal.

Close these two terminal.

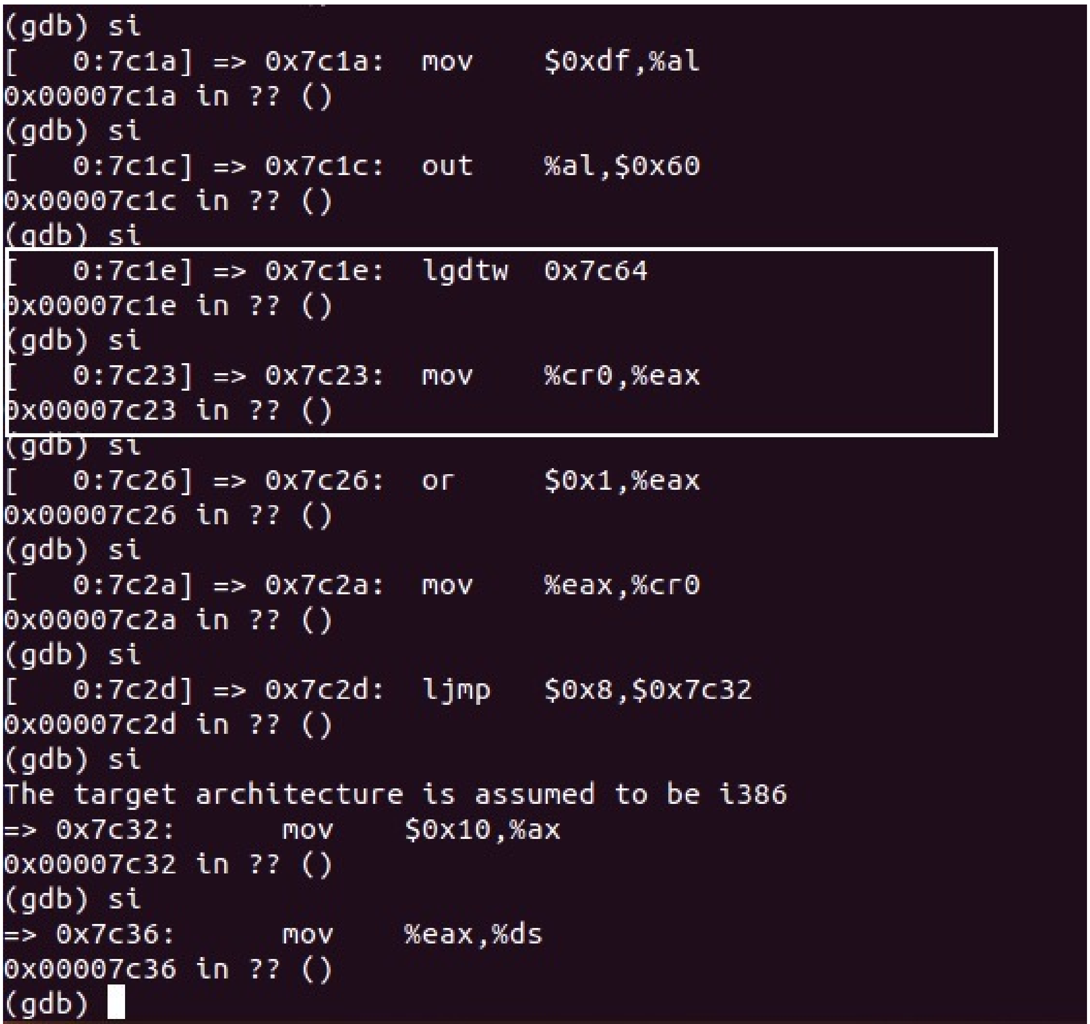
Then, run **make clean**. After that change the address in the **boot/Makefrag** to **0x7d00** and then **make**.

1. Open a terminal and type **make qemu-gdb**
2. Open a second terminal and type **make gdb**
3. Set a breakpoint and continue execution to this breakpoint: type in the second terminal that **b *0x7c00** and **c**
4. Repeat the command **si** in the second terminal.

We can see that:

3.1.3 Exercise 6

*[We can examine memory using GDB's x command. The GDB manual has full details, but for now, it is enough to know that the command **x/Nx ADDR** prints N words of memory at ADDR. (Note that both 'x's in the command are lowercase.) Warning: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).*



```
(gdb) si
[   0:7c1a] => 0x7c1a:  mov    $0xdf,%al
0x00007c1a in ?? ()
(gdb) si
[   0:7c1c] => 0x7c1c:  out    %al,$0x60
0x00007c1c in ?? ()
(gdb) si
[   0:7c1e] => 0x7c1e:  lgdtw  0x7c64
0x00007c1e in ?? ()
(gdb) si
[   0:7c23] => 0x7c23:  mov    %cr0,%eax
0x00007c23 in ?? ()
(gdb) si
[   0:7c26] => 0x7c26:  or     $0x1,%eax
0x00007c26 in ?? ()
(gdb) si
[   0:7c2a] => 0x7c2a:  mov    %eax,%cr0
0x00007c2a in ?? ()
(gdb) si
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb) si
The target architecture is assumed to be i386
=> 0x7c32:      mov    $0x10,%ax
0x00007c32 in ?? ()
(gdb) si
=> 0x7c36:      mov    %eax,%ds
0x00007c36 in ?? ()
(gdb)
```

Figure 3: The correct version with the link address as 0x7c00

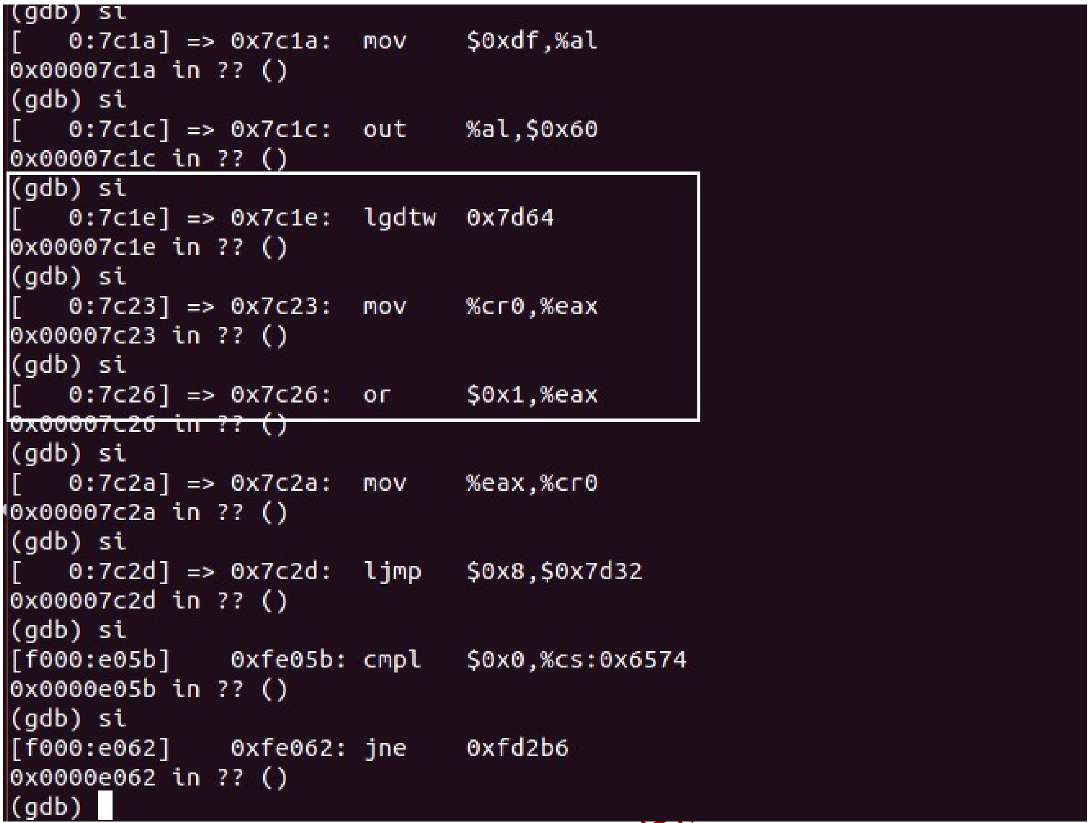


Figure 4: The wrong version with the link address as 0x7d00

Reset the machine (exit QEMU/GDB and start them again). 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? (You do not really need to use QEMU to answer this question. Just think.)]

We can clearly observe that after the kernal was loaded (b *0x1000c and c), the memory 0x100000 was loaded with kernal code.

```
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 100000
0x186a0:        0x00000000      0x00000000      0x00000000      0x00000000
0x186b0:        0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/8x 0x100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw    $0x1234,0x472

reakpoint 2, 0x0010000c in ?? ()
gdb) x/8x 0x100000
x100000:        0x1badb002      0x00000000      0xe4524ffe      0x7205c766
x100010:        0x34000004      0x0000b812      0x220f0011      0xc0200fd8
gdb) 
```

Figure 5: Exercise 6

4 Part 3: The Kernel

4.1 Using virtual memory to work around position dependence

4.1.1 Exercise 7

[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. Make sure you understand what just happened.*
What is the first instruction after the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the ***movl %eax, %cr0*** *in* ***kern/entry.S****, trace into it, and see if you were right.]*

Okay, first we have observe that **0x10000C** is the entry address of kernal. Let's firstly check the content in memory **0x00100000** and at **0xf0100000** before the instruction: **movl %eax, %cr0**.

```
(gdb) x/4xb 0xf0100000
0xf0100000 <_start+4026531828>: 0x00    0x00    0x00    0x00
(gdb) x/4xb 0x00100000
0x100000:       0x02    0xb0    0xad    0x1b
```

Figure 6: Exercise 7: After the instruction

After this instruction: **movl %eax, %cr0** , we can see that

```
(gdb) x/4xb 0xf0100000
0xf0100000 <_start+4026531828>: 0x02    0xb0    0xad    0x1b
(gdb) x/4xb 0x00100000
0x100000:       0x02    0xb0    0xad    0x1b
(gdb)
```

Figure 7: Exercise 7: After the instruction

4.2 Formatted Printing to the Console

4.2.1 The Analysis of kern/printf.c, lib/printfmt.c, and kern/console.c

We have the following observations:

1. kern/printf.c has the following two functions (it has have a static method putch() for its self.):
   - cprintf(): which use the vprintfmt() in file lib/printfmt.c
   - vcprintf(): use the vcprintf() in the lib/printfmc
   - putch(): use the cputchar() in the kern/console.c.
2. dd

So, let's start from kern/console.c.

4.2.2 Analysis of kern/console.c

```
void
cputchar(int c)
{
        cons_putc(c);
}
```

Listing 6: The cputchar function in kern/console.c

Then, let's see cons_putc function

```
// output a character to the console
static void
cons_putc(int c)
{
        serial_putc(c);
        lpt_putc(c);
        cga_putc(c);
}
```

Listing 7: The cons_putc function in kern/console.c

4.2.3 Analysis of lib/printfmt.c

4.2.4 Exercise 8

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.

```
// Main function to format and print a string.
void printfmt(void (*putch)(int, void*), void *putdat, const char *fmt, ...);

void
vprintfmt(void (*putch)(int, void*), void *putdat, const char *fmt, va_list ap)
{
  register const char *p;
  register int ch, err;
  unsigned long long num;
  int base, lflag, width, precision, altflag;
  char padc;

  while (1) {
      //print all the characters before %
    while ((ch = *(unsigned char *) fmt++) != '%') {
      if (ch == '\0')
        return;
      putch(ch, putdat);
    }

    // Process a %-escape sequence
    padc = ' ';
    width = -1;
    precision = -1;
    lflag = 0;
    altflag = 0;
  reswitch:
    switch (ch = *(unsigned char *) fmt++) {

    // flag to pad on the right
    case '-':
      padc = '-';
      goto reswitch;

    // flag to pad with 0's instead of spaces
    case '0':
      padc = '0';
      goto reswitch;

    // width field
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
      for (precision = 0; ; ++fmt) {
        precision = precision * 10 + ch - '0';
        ch = *fmt;
        if (ch < '0' || ch > '9')
          break;
      }
      goto process_precision;

    case '*':
```

```
      precision = va_arg(ap, int);
      goto process_precision;

    case '.':
      if (width < 0)
        width = 0;
      goto reswitch;

    case '#':
      altflag = 1;
      goto reswitch;

    process_precision:
      if (width < 0)
        width = precision, precision = -1;
      goto reswitch;

    // long flag (doubled for long long)
    case 'l':
      lflag++;
      goto reswitch;

    // character
    case 'c':
      putch(va_arg(ap, int), putdat);
      break;

    // error message
    case 'e':
      err = va_arg(ap, int);
      if (err < 0)
        err = -err;
      if (err >= MAXERROR || (p = error_string[err]) == NULL)
        printfmt(putch, putdat, "error %d", err);
      else
        printfmt(putch, putdat, "%s", p);
      break;

    // string
    case 's':
      if ((p = va_arg(ap, char *)) == NULL)
        p = "(null)";
      if (width > 0 && padc != '-')
        for (width -= strnlen(p, precision); width > 0; width--)
          putch(padc, putdat);
      for (; (ch = *p++) != '\0' && (precision < 0 || --precision >= 0); width--)
        if (altflag && (ch < ' ' || ch > '~'))
          putch('?', putdat);
        else
          putch(ch, putdat);
      for (; width > 0; width--)
        putch(' ', putdat);
      break;

    // (signed) decimal
    case 'd':
      num = getint(&ap, lflag);
      if ((long long) num < 0) {
```

```
        putch('-', putdat);
        num = -(long long) num;
      }
      base = 10;
      goto number;

    // unsigned decimal
    case 'u':
      num = getuint(&ap, lflag);
      base = 10;
      goto number;

    // (unsigned) octal
    case 'o':
      // Replace this with your code.
      putch('X', putdat);
      putch('X', putdat);
      putch('X', putdat);
      break;

    // pointer
    case 'p':
      putch('0', putdat);
      putch('x', putdat);
      num = (unsigned long long)
        (uintptr_t) va_arg(ap, void *);
      base = 16;
      goto number;

    // (unsigned) hexadecimal
    case 'x':
      num = getuint(&ap, lflag);
      base = 16;
    number:
      printnum(putch, putdat, num, base, width, padc);
      break;

    // escaped '%' character
    case '%':
      putch(ch, putdat);
      break;

    // unrecognized escape sequence - just print it literally
    default:
      putch('%', putdat);
      for (fmt--; fmt[-1] != '%'; fmt--)
        /* do nothing */;
      break;
    }
  }
}
```

Listing 8: The vprintfmt() function in lib/printfmt.c

1. Print all the character before %.
2. Then it process the special character just before % character:

```
while ((ch = *(unsigned char *) fmt++) != '%') {
        if (ch == '\0')
                return;
        putch(ch, putdat);
}
```

Figure 8: Exercise 7: After the instruction

- long: l
- character: c
- error: e
- decimal: d
- ...

Actually, case 'o' is where we should fill in our code:

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

Listing 9: Case octal

4.3 The Questions Following Exercise 8

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?]*
   Answer: console.c export the cputchar() function.

2. *[Explain the following from console.c:]*

```
      // What is the purpose of this?
        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;
        }
```

Listing 10: Subpart of console.c

The **memmove** [1] command is used to put the memory at **crt_buf + CRT_COLS** to **crt_buf**. This reason is to solve the problem that the current crt_pos is already bigger than current **CRT_SIZE**. After that

3. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86. Trace the execution of the following code step-by-step:

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

```

Listing 11: question 3 code

- In the call to cprintf(), to what does fmt point? To what does ap point?
- 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.

Answer: Let's first revisit the **cprintf()** in **kern/printfc.c**:

```c
static void
putch(int ch, int *cnt)
{
        cputchar(ch);
        *cnt++;
}

int
vcprintf(const char *fmt, va_list ap)
{
        int cnt = 0;

        vprintfmt((void*)putch, &cnt, fmt, ap);
        return cnt;
}

int
cprintf(const char *fmt, ...)
{
        va_list ap;
        int cnt;

        va_start(ap, fmt);
        cnt = vcprintf(fmt, ap);
        va_end(ap);

        return cnt;
}

```

Listing 12: question 3 code

[1] memmove(void destination, const void source, size_t num)

We can see that fmt is the pointer to this string. **va_list ap** is the list of the input parameter in the string fmt, i.e., **{x, y, z}**.

We can see that **cprintf()** calls vcprintf(), then the **vcprint()** call the **vprintfmt()**. During the process of **vprintfmt()**, it jumps to the **case d**.

```
                case 'd':
                        num = getint(&ap, lflag);
                        if ((long long) num < 0) {
                                putch('-', putdat);
                                num = -(long long) num;
                        }
                        base = 10;
                        goto number;
```

We can see that it first output the minus symbol '-' if number is negative. Then **goto number**. Actually, **goto number** is for

- **case 'd'**
- **case 'u'**
- **case 'p'**

4. Run the following code

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

Listing 13: Question 4 Code

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters. The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

Answer: It print He110 World

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? **cprintf("x=%d y=%d", 3);**
Answer: We append the line in the **kern/monitor.c** file and find the result as: In our case

Figure 9: Question 5

4.4 The Stack

4.4.1 The ESP and EBP Register

1. ESP

    - The x86 stack pointer (esp register) points to the lowest location on the stack that is currently in use. Everything below that location in the region reserved for the stack is free.
    - **Pushing** a value onto the stack involves decreasing the stack pointer and then writing the value to the place the stack pointer points to.
    - **Popping** a value from the stack involves reading the value the stack pointer points to and then increasing the stack pointer.

2. EBP

    - d

4.4.2 Exercise 9

[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?]

Answer: we need to carefully read the **kern/entry.S** file.

```
/* See COPYRIGHT for copyright information. */

#include <inc/mmu.h>
#include <inc/memlayout.h>

# Shift Right Logical
#define SRL(val, shamt)   (((val) >> (shamt)) & ~(-1 << (32 - (shamt))))


###################################################################
# The kernel (this code) is linked at address ~(KERNBASE + 1 Meg),
# but the bootloader loads it at address ~1 Meg.
#
# RELOC(x) maps a symbol x from its link address to its actual
# location in physical memory (its load address).
###################################################################

#define RELOC(x) ((x) - KERNBASE)

#define MULTIBOOT_HEADER_MAGIC (0x1BADB002)
#define MULTIBOOT_HEADER_FLAGS (0)
#define CHECKSUM (-(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS))

###################################################################
# entry point
###################################################################

.text

# The Multiboot header
.align 4
```

```
.long MULTIBOOT_HEADER_MAGIC
.long MULTIBOOT_HEADER_FLAGS
.long CHECKSUM

# '_start' specifies the ELF entry point.  Since we haven't set up
# virtual memory when the bootloader enters this code, we need the
# bootloader to jump to the *physical* address of the entry point.
.globl    _start
_start = RELOC(entry)

.globl entry
entry:
  movw  $0x1234,0x472     # warm boot

  # We haven't set up virtual memory yet, so we're running from
  # the physical address the boot loader loaded the kernel at: 1MB
  # (plus a few bytes).  However, the C code is linked to run at
  # KERNBASE+1MB.  Hence, we set up a trivial page directory that
  # translates virtual addresses [KERNBASE, KERNBASE+4MB) to
  # physical addresses [0, 4MB).  This 4MB region will be
  # sufficient until we set up our real page table in mem_init
  # in lab 2.

  # Load the physical address of entry_pgdir into cr3.  entry_pgdir
  # is defined in entrypgdir.c.
  movl  $(RELOC(entry_pgdir)), %eax
  movl  %eax, %cr3
  # Turn on paging.
  movl  %cr0, %eax
  orl $(CR0_PE|CR0_PG|CR0_WP), %eax
  movl  %eax, %cr0

  # Now paging is enabled, but we're still running at a low EIP
  # (why is this okay?).  Jump up above KERNBASE before entering
  # C code.
  mov $relocated, %eax
  jmp *%eax
relocated:

  # Clear the frame pointer register (EBP)
  # so that once we get into debugging C code,
  # stack backtraces will be terminated properly.
  movl  $0x0,%ebp     # nuke frame pointer

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

  # now to C code
  call  i386_init

  # Should never get here, but in case we do, just spin.
spin: jmp spin


.data
###################################################################
# boot stack
###################################################################
```

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

Listing 14: kern/entry.S

We can see that

```
	movl	$0x0,%ebp		# nuke frame pointer
	# Set the stack pointer
	movl	$(bootstacktop),%esp
```

is used to initialize its stack.

1. Determine where the kernel initializes its stack? **Answer:** in the kern/entry.S file. The **movl $0x0,%ebp** and **movl $(bootstacktop),%esp** command.

2. Where in memory its stack is located?
   To better understand this problem, we should separate it into the following steps: we give the instruction in kern/entry.S again.

```
	  # Load the physical address of entry_pgdir into cr3.  entry_pgdir
	# is defined in entrypgdir.c.
	movl	$(RELOC(entry_pgdir)), %eax
	movl	%eax, %cr3
	# Turn on paging.
	movl	%cr0, %eax
	orl	$(CR0_PE|CR0_PG|CR0_WP), %eax
	movl	%eax, %cr0

	# Now paging is enabled, but we're still running at a low EIP
	# (why is this okay?).  Jump up above KERNBASE before entering
	# C code.
	mov	$relocated, %eax
	jmp	*%eax
relocated:

	# Clear the frame pointer register (EBP)
	# so that once we get into debugging C code,
	# stack backtraces will be terminated properly.
	movl	$0x0,%ebp			# nuke frame pointer

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

	# now to C code
	call	i386_init

```

- Load the physical address of table entry_pgdir to cr3.
- Set the PE, PG, WP of cr0 as 1.
- Jump up above KERNBASE before entering before entering C code. The **$relocated** value is **0xf010002**

- Set the ebp as 0x00.
- Set the esp as 0xf0110000.

The stack size is 32KB. So, the stack locates 0xf0108000-0xf0111000

4.4.3 Exercise 10

[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?
Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.]

Answer: Let's firstly check the C code of test_backtrace in kern/init.c:

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
        cprintf("entering test_backtrace %d\n", x);
        if (x > 0)
                test_backtrace(x-1);
        else
                mon_backtrace(0, 0, 0);
        cprintf("leaving test_backtrace %d\n", x);
}
```

From this code, we can figure out why these code when we make qemu.

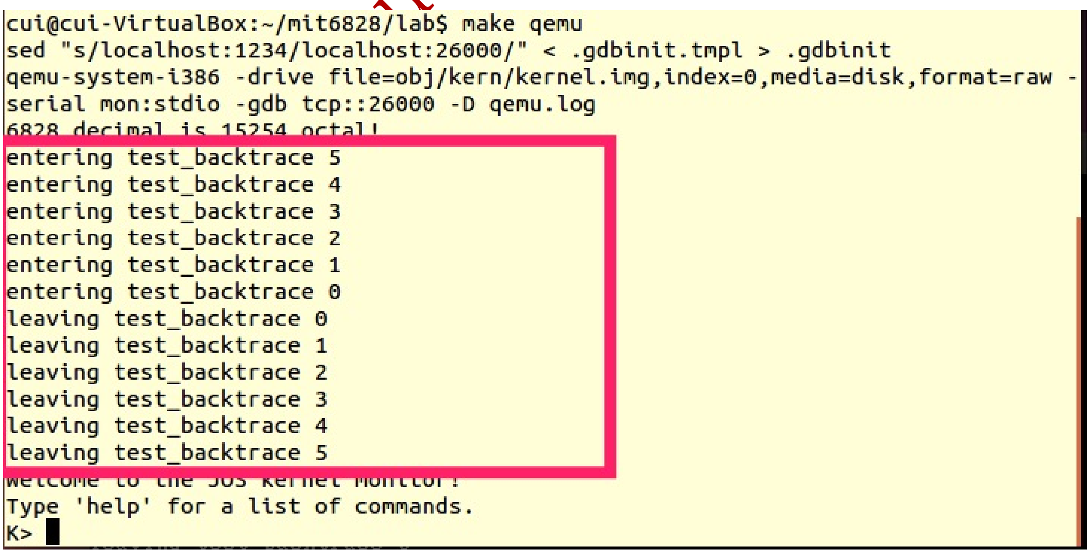


Figure 10: Exercise 6

Because there is test_backtrace(5) in the kern/init.c.
Then we check the test_backtrace in obj/kern/kernal.asm

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       53                      push   %ebx
f0100044:       83 ec 14                sub    $0x14,%esp
f0100047:       8b 5d 08                mov    0x8(%ebp),%ebx
        cprintf("entering test_backtrace %d\n", x);
f010004a:       89 5c 24 04             mov    %ebx,0x4(%esp)
f010004e:       c7 04 24 e0 18 10 f0    movl   $0xf01018e0,(%esp)
f0100055:       e8 d7 08 00 00          call   f0100931 <cprintf>
        if (x > 0)
f010005a:       85 db                   test   %ebx,%ebx
f010005c:       7e 0d                   jle    f010006b <test_backtrace+0x2b>
                test_backtrace(x-1);
f010005e:       8d 43 ff                lea    -0x1(%ebx),%eax
f0100061:       89 04 24                mov    %eax,(%esp)
f0100064:       e8 d7 ff ff ff          call   f0100040 <test_backtrace>
f0100069:       eb 1c                   jmp    f0100087 <test_backtrace+0x47>
        else
                mon_backtrace(0, 0, 0);
f010006b:       c7 44 24 08 00 00 00    movl   $0x0,0x8(%esp)
f0100072:       00
f0100073:       c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
f010007a:       00
f010007b:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
f0100082:       e8 18 07 00 00          call   f010079f <mon_backtrace>
        cprintf("leaving test_backtrace %d\n", x);
f0100087:       89 5c 24 04             mov    %ebx,0x4(%esp)
f010008b:       c7 04 24 fc 18 10 f0    movl   $0xf01018fc,(%esp)
f0100092:       e8 9a 08 00 00          call   f0100931 <cprintf>
}
```

5 Reference

1. bysui's github and blog
   - github: https://github.com/bysui/mit6.828
   - blog: https://blog.csdn.net/bysui
2. SmallPond's github and blog
   - github: https://github.com/SmallPond/MIT6.828_OS
   - blog: https://me.csdn.net/Small_Pond
3. SimpCosm's github
   - github: https://github.com/SimpCosm/6.828
4. fatsheep9146's blog
   - blog: https://www.cnblogs.com/fatsheep9146/category/769143.html

https://github.com/csnlp/MIT6828