

به نام خدا



دانشگاه صنعتی امیرکبیر  
( پلی تکنیک تهران )

گزارش Lab1 سیستم عامل پیشرفته

استاد:

دکتر جوادی

دانشجویان:

امیررضا زارع – ۴۰۱۱۳۱۰۰۸

سید علیرضا حسینی – ۴۰۱۱۳۱۰۰۵

برای راه‌اندازی سیستم عامل JOS که در حد بسیار مقدماتی است، در ابتدا باید یک سری مقدماتی را فراهم کنیم. برای این پروژه از نسخه لینوکس Ubuntu 14.04 استفاده می‌شود. برای راه‌اندازی JOS باید gcc، binutils و سپس gdb را نصب کنیم. این مراحل را انجام دادیم و وارد پوشه JOS شدیم و با دستور

make qemu-nox، qemu را بالا آوردیم. قبل از این کار نیز حالت مجازی سازی ماشین مجازی را فعال کرده بودیم. در ابتدای کار فقط با دو دستور help و kerninfo روبرو هستیم. این دو دستور تست شد و به درستی کار کردند.

تست دستور make qemu-nox:

```

jos@Zare-Hosseini: ~/jos_f23_401131008_401131005
jos@Zare-Hosseini:~$ cd ./jos_f23_401131008_401131005/
jos@Zare-Hosseini:~/jos_f23_401131008_401131005$ make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-x86_64 -nographic -cpu qemu64 -m 256 -hda obj/kern/kernel.img -serial
mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>

```

تست دستور help:

```

jos@Zare-Hosseini: ~/jos_f23_401131008_401131005
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-x86_64 -nographic -cpu qemu64 -m 256 -hda obj/kern/kernel.img -serial
mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
K>

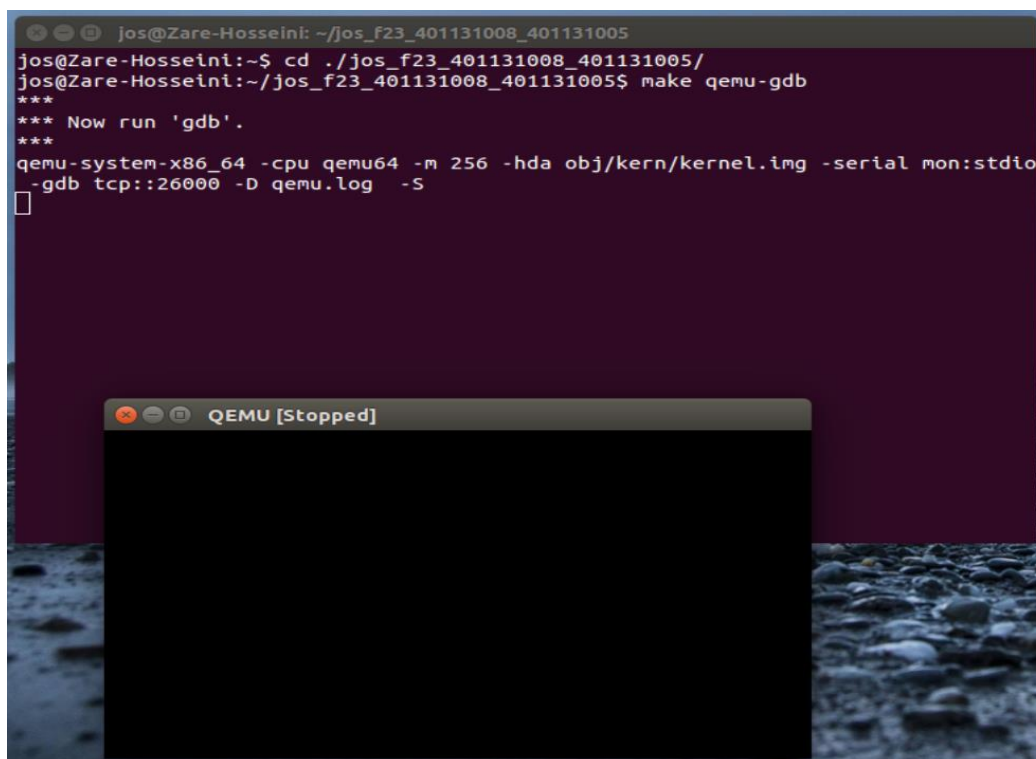
```

تست دستور kerninfo:

```
josh@Zare-Hosseini: ~/jos_f23_401131008_401131005
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> help
help - Display this list of commands
kerninfo - Display information about the kernel
K> kerninfo
Special kernel symbols:
_start      0020000c (phys)
entry       80042000c (virt) 0020000c (phys)
etext       800420932f (virt) 0020932f (phys)
edata       800421c6a0 (virt) 0021c6a0 (phys)
end         800421dd40 (virt) 0021dd40 (phys)
Kernel executable memory footprint: 120KB
K>
```

در بخش The ROM BIAS از سایت مربوط به درس، گفته دو تا ترمینال باز کنید و در یکی دستور `make qemu-gdb` و در دیگری دستور `make` و سپس `gdb` را اجرا کنید. این کار را انجام دادیم.

تست دستور `make qemu-gdb`:



در ادامه دستور make و سپس gdb را می‌نویسیم و بعد از آن با استفاده از دستور si و registers i می‌توانیم کد اسمبلی را مشاهده کنیم.

تست si:

```

jos@Zare-Hosseini: ~/jos_f23_401131008_401131005
jos@Zare-Hosseini:~$ cd ./jos_f23_401131008_401131005/
jos@Zare-Hosseini:~/jos_f23_401131008_401131005$ make qemu-gdb
***
*** Now run 'gdb'.
***
qemu-system-x86_64 -cpu qemu64 -m 256 -hda obj/kern/kernel.img -serial mon:stdio
-gdb tcp::26000 -D qemu.log -S
[ ]

QEMU [Stopped]

jos@Zare-Hosseini:~/jos_f23_401131008_401131005
of GDB. Attempting to continue with the default i8086 setting
s.

The target architecture is assumed to be i8086
[f000:fff0] 0xffff0: jmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
(gdb) si
[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x6574
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xfd2b6
0x0000e062 in ?? ()
(gdb) si
[f000:e066] 0xfe066: xor %ax,%ax
0x0000e066 in ?? ()
(gdb) si
[f000:e068] 0xfe068: mov %ax,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a] 0xfe06a: mov $0x7000,%esp
0x0000e06a in ?? ()
(gdb) s
Cannot find bounds of current function
(gdb) si

```

تست دستور register i:

```

jos@Zare-Hosseini: ~/jos_f23_401131008_401131005
jos@Zare-Hosseini:~$ cd ./jos_f23_401131008_401131005/
jos@Zare-Hosseini:~/jos_f23_401131008_401131005$ make qemu-gdb
***
*** Now run 'gdb'.
***
qemu-system-x86_64 -cpu qemu64 -m 256 -hda obj/kern/kernel.img -serial mon:stdio
-gdb tcp::26000 -D qemu.log -S
[ ]

QEMU [Stopped]

jos@Zare-Hosseini:~/jos_f23_401131008_401131005
0x0000e06a in ?? ()
(gdb) s
Cannot find bounds of current function
(gdb) si
[f000:e070] 0xfe070: mov $0xf3c24,%edx
0x0000e070 in ?? ()
(gdb) i registers
eax          0x0      0
ecx          0x0      0
edx          0x663    1635
ebx          0x0      0
esp          0x7000   0x7000
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0xe070   0xe070
eflags       0x46     [ PF ZF ]
cs           0xf000   61440
ss           0x0      0
ds           0x0      0
es           0x0      0
fs           0x0      0
gs           0x0      0
(gdb)

```

تا اینجای کار gdb انجام شد.

## تمرین ۱:

**Exercise 1.** Read or at least carefully scan the entire [PC Assembly Language](#) book, except that you should skip all sections after 1.3.5 in chapter 1, which talk about features of the NASM assembler that do not apply directly to the GNU assembler. You may also skip chapters 5 and 6, and all sections under 7.2, which deal with processor and language features we won't use. This reading is useful when trying to understand assembly in JOS, and writing your own assembly. If you have never seen assembly before, read this book carefully.

Also read the section "The Syntax" in [Brennan's Guide to Inline Assembly](#) to familiarize yourself with the most important features of GNU assembler syntax. JOS uses the GNU assembler.

We will be developing JOS for the 64-bit version of the x86 architecture (also known as amd64). The assembly is very similar to 32-bit, with a few key differences. Read [this guide](#), which explains the key differences between the assembly.

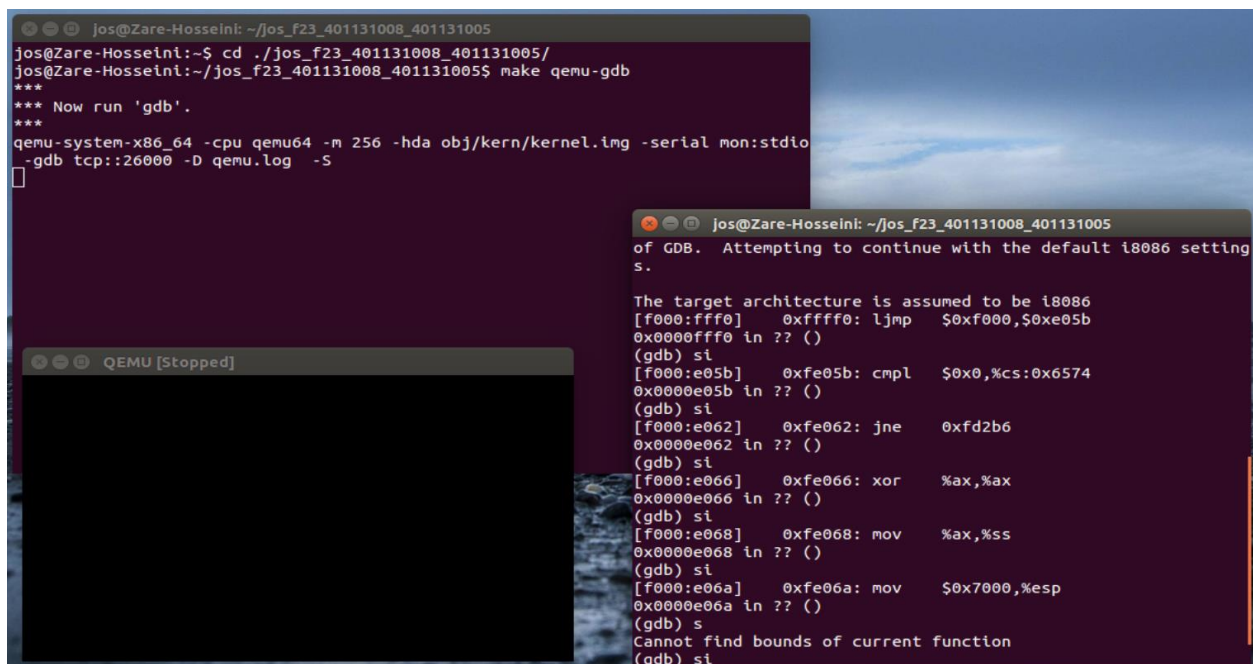
Become familiar with inline assembly by writing a simple program. Modify the program ex1.c to include inline assembly that increments the value of x by 1.

در این تمرین باید با استفاده از inline assembly به یک متغیر یک واحد اضافه می‌کردیم. کد این تمرین در مشاهده است.

## تمرین ۲:

**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 [reference materials page](#). No need to figure out all the details - just the general idea of what the BIOS is doing first.

هدف از این تمرین، آشنایی با gdb است. وارد gdb شدیم و با دستور si به پیمایش برنامه پرداختیم.



```
josh@Zare-Hosseini: ~/jos_f23_401131008_401131005
josh@Zare-Hosseini:~$ cd ./jos_f23_401131008_401131005/
josh@Zare-Hosseini:~/jos_f23_401131008_401131005$ make qemu-gdb
***
*** Now run 'gdb'.
***
qemu-system-x86_64 -cpu qemu64 -m 256 -hda obj/kern/kernel.img -serial mon:stdio
-gdb tcp::26000 -D qemu.log -S
[]

josh@Zare-Hosseini:~/jos_f23_401131008_401131005
of GDB. Attempting to continue with the default i8086 setting
s.

The target architecture is assumed to be i8086
[f000:ffff] 0xfffff0: jmp $0xf000,$0xe05b
0x0000ffff in ?? ()
(gdb) si
[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x6574
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xfd2b6
0x0000e062 in ?? ()
(gdb) si
[f000:e066] 0xfe066: xor %ax,%ax
0x0000e066 in ?? ()
(gdb) si
[f000:e068] 0xfe068: mov %ax,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a] 0xfe06a: mov $0x7000,%esp
0x0000e06a in ?? ()
(gdb) s
Cannot find bounds of current function
(gdb) si
```



## تمرین ۳:

**Exercise 3.** Take a look at the [lab tools guide](#), especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that break point. Trace through the code in `boot/boot.s`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

- 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?
- 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?

سوال ۳.۱: پردازنده از چه مرحله‌ای شروع می‌کند که کدهای ۳۲ بیتی را اجرا کند؟ چه چیزی باعث می‌شود که مد از حالت ۱۶ بیتی به ۳۲ بیتی تغییر کند؟

در فایل اسمبلی `boot.s` دستور جامپ باعث می‌شود که مد از ۱۶ به ۳۲ بیتی تغییر کند. در تصویر زیر جامپ را در خط آخر مشاهده می‌کنید.

```
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
lgdt    gdt_desc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
ljmp     $PROT_MODE_CSEG, $protcseg

.code32                                # Assemble for 32-bit mode
protcseg:
```

سوال ۳.۲: آخرین دستورالعمل اجرا شده در بوت لودر و همچنین اولین دستورالعمل کرنل چیست؟

ابتدا در بوت لودر فایل `boot.s` اجرا می‌شود و سپس از فایل `main.c`، تابع `bootmain` فراخوانی می‌شود که می‌توان نتیجه گرفت که آخرین دستوری که در بوت لودر اجرا می‌شود، در فایل `main.c` است که این دستور را در تصویر زیر مشاهده می‌کنید.

```
((void (*)(void)) ((uint32_t)(ELFHDR->e_entry)))();
```

بر اساس فایل `boot.asm` که در مسیر `obj/boot/boot.asm` قرار دارد، آخرین دستور یک جامپ می‌کند به آدرسی که `0x10018` به آن اشاره می‌کند و با `gdb` باید ببینیم جایی که این آدرس به آن اشاره می‌کند کجاست.

با اجرای دستور `(gdb) x/1x 0x10018` خروجی `0x0010000: 0x10018` تولید می‌شود. پس دستوری که در `0x0010000` است، اولین دستور کرنل است. این دستور به صورت زیر است:

```
f010000c: 66 c7 05 72 04 00 00 movw $0x1234,0x472
```

سوال ۳.۳: بوت لودر چطوری تصمیم می‌گیرد که باید چند سکتور بخواند تا بتواند کل کرنل را از دیسک واکشی کند؟ این اطلاعات را باید از کجا بگیرد؟

باید این اطلاعات را از `x`.

## تمرین ۴:

**Exercise 4.** 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!

چند دستورالعمل اول بوت لودر را باید دوباره مرور کنیم و اولین دستوری که در صورت اشتباه بودن آدرس لینک بوت لودر اشتباه انجام می‌شود را پیدا کنیم. این دستور را در تصویر زیر مشاهده می‌کنید.

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

## تمرین ۵:

**Exercise 5.** 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).

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.)

سوال ۵.۱: در نقطه‌ای که بایاس وارد بوت لودر می‌شود، کلمه حافظه را که در آدرس `0x00100000` است را بررسی کنید.

به طور کامل صفر است. چون هنوز وارد آن بخش نشده‌ایم، می‌تواند هر مقداری داشته باشد. با تست کردن این دستور در `gdb` متوجه شدیم که کامل صفر است. در تصویر زیر مشاهده می‌کنید.

```
(gdb) x/8x 0x00100000
0x100000: 0x00000000 0x00000000 0x00000000 0x00000000
0x100010: 0x00000000 0x00000000 0x00000000 0x00000000
```

حال در `make` داریم:

```
(gdb) x/8w 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x0000b812 0x220f0011 0xc0200fd8
(gdb) x/8i 0x00100000
0x100000: add    0x1bad(%eax),%dh
0x100006: add    %al, (%eax)
0x100008: decb   0x52(%edi)
0x10000b: in     $0x66,%al
0x10000d: movl   $0xb81234,0x472
```

```

0x100017: add    %dl, (%ecx)
0x100019: add    %cl, (%edi)
0x10001b: and    %al, %bl

```

با مقایسه کردن آن با فایل boot/kernel.asm می‌بینیم که این دقیقاً آغاز بخش هسته است.

## تمرین ۶:

**Exercise 6.** Use QEMU and GDB to trace into the early JOS kernel boot code (in the kern/bostrap.S directory) and find where the new virtual-to-physical mapping takes effect. Then examine the Global Descriptor Table (GDT) that the code uses to achieve this effect, and make sure you understand what's going on.

What is the first instruction *after* the new mapping is established that would fail to work properly if the old mapping were still in place? Comment out or otherwise intentionally break the segmentation setup code in kern/entry.S, trace into it, and see if you were right.

```

(gdb) b *0x100025
Breakpoint 1 at 0x100025
(gdb) c
Continuing.

```

The target architecture is assumed to be i386

```
=> 0x100025: mov    %eax,%cr0
```

```

(gdb) x/i $pc
=> 0x100113: jmp    0x100113

```

```

(gdb) x/10w 0x00100000
0x100000: 0x107000b8 0x66188900 0x047205c7 0x12340000
0x100010: 0x007c00bc 0x00cce800 0x20b80000 0x0f000000
0x100020: 0x00bfe022 0x31001020

```

```

(gdb) x/10w 0xf0100000
0xf0100000: 0x00000000 0x00000000 0x00000000 0x00000000
0xf0100010: 0x00000000 0x00000000 0x00000000 0x00000000
0xf0100020: 0x00000000 0x00000000

```

```

(gdb) si
Cannot remove breakpoints because program is no longer writable.
Further execution is probably impossible.

```

```
=> 0x100113: jmp    0x100113
```

```
0x00100113 in ?? ()
```

```

(gdb) x/10w 0x00100000
0x100000: 0x107000b8 0x66188900 0x047205c7 0x12340000
0x100010: 0x007c00bc 0x00cce800 0x20b80000 0x0f000000
0x100020: 0x00bfe022 0x31001020

```

همانطور که مشاهده می‌کنید، آدرس‌های 0xf0100000 اکنون شامل متن هسته هستند.



(gdb) x/10i 0xf0100000

```
0xf0100000 <_start+4026531828>: add    0x1bad(%eax),%dh
0xf0100006 <_start+4026531834>: add    %al, (%eax)
0xf0100008 <_start+4026531836>: decb   0x52(%edi)
0xf010000b <_start+4026531839>: in     $0x66,%al
0xf010000d <entry+1>:          movl   $0xb81234,0x472
0xf0100017 <entry+11>:        add    %dl, (%ecx)
0xf0100019 <entry+13>:        add    %cl, (%edi)
0xf010001b <entry+15>:        and    %al,%bl
0xf010001d <entry+17>:        mov    %cr0,%eax
0xf0100020 <entry+20>:        or     $0x80010001,%eax
```

اگر دستوری که با رنگ زرد هایلایت شده را ننویسیم، عملیات صفحه بندی فعال نمی‌شود و اولین دستوری که با شکست مواجه می‌شود، دستور `movl $0x0,%ebp` است و این شکست به این دلیل اتفاق می‌افتد که با توجه به دستور قبلی، مجبور بودیم به آدرس `0xf010002c` برویم و از طرفی صفحه‌بندی هم فعال نیست، `qemu` از کار می‌افتد.

## تمرین ۷:

**Exercise 7.** 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.

سوال ۷.۱: رابط بین `printf.c` و `console.c` را توضیح دهید. به طور خاص، `console.c` چه عملکردی را صادر می‌کند؟ چگونه این تابع توسط `printf.c` استفاده می‌شود؟

در فایل `console.c` تابع سطح پایینی به نام `cpuchar(int c)` وجود دارد یک `char` را در پورت موازی، پورت سریالی و همچنین در بافر CGA قرار می‌دهد که روی صفحه ظاهر می‌شود. همچنین فایل `printf.c` هم دارای تابع `putch(int ch, int* cnt)` است که این تابع از خروجی تابع `cpuchar(int c)` استفاده می‌کند.

سوال ۷.۲: کد زیر را که از `console.c` است را توضیح دهید.

```
1  if (crt_pos >= CRT_SIZE) {
2      int i;
3      memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
4      for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
5          crt_buf[i] = 0x0700 | ' ';
6      crt_pos -= CRT_COLS;
7  }
```

Crt یک ماتریس با ۲۵ سطر و ۸۰ ستون است که اندازه آن  $25 \times 80 = 2000$  است. اگر بخواهیم موقعیت `cursor` را از `crt_pos` بدست بیاوریم، با محاسبه زیر این کار انجام می‌شود:

```
memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
```

هر موقع که به انتهای کد برسیم، در این ماتریس یک سطر جدید اضافه می‌شود.

سوال ۷.۳: اجرای کد زیر را بصورت گام به گام بررسی کنید.

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

در فراخوانی `cprintf()` پارامترهای `fmt` و `ap` به چه چیزی اشاره می‌کنند؟

در ابتدا هر دو به یک چیز اشاره می‌کنند و بعد از آن `fmt` به فرمت آرگومان اشاره می‌کند و `ap` به آرگومان‌های متغیر اشاره می‌کند.

```
(gdb) info locals
fmt = 0xf0101a17 "x %d, y %x, z %d\n"
ap = 0xf0101a17 "x %d, y %x, z %d\n"
```

سوال ۷.۴: کد زیر را اجرا کنید. خروجی آن چیست؟ و نحوه رسیدن به این خروجی را به روش گام به گام تمرین قبلی توضیح دهید.

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

خروجی این کد بصورت عبارت "He110 World" است. `e110` هگزادسیمال شده‌ی عدد `۵۷۶۱۶` است. `Rld` هم مربوط به عدد `0x00646c72` است. نحوه‌ی رسیدن به این خروجی بدین صورت است که در `big endian` برای نشان دادن `rd\0` باید عدد بصورت `00 64 2C 72` باشد. پس `۰x7266400` باشد و `57616` نیز نیاز به تغییری ندارد.

سوال ۷.۵: در کد زیر، بعد از `y=` قرار است چه چیزی چاپ شود؟

```
cprintf("x=%d y=%d", 3);
```

برای `x` مقدار ۳ در خروجی ظاهر می‌شود. برای `y` مقدار خانه‌ی بعد از `x` در `stack` را قرار می‌دهد.

سوال ۷.۶: فرض کنید که `gcc` قرار است روند فراخوانی خور را تغییر دهد تا آرگومان‌ها را به ترتیب بر روی پشته پوش کند به طوریکه آخرین آرگومان، آخرین پوش باشد. چگونه باید `cprintf()` یا اینترفیس آن را تغییر دهیم که همچنان بتواند تعداد متغیری از آرگومان‌ها را به آن منتقل کند؟

برای نشان دادن تعداد آرگومان‌ها، می‌توانیم بعد از آخرین آرگومان، یک عدد پوش کنیم که تعداد آرگومان‌ها را نشان می‌دهد.

## چالش ۱:

*Challenge 1 (2 bonus points)* Enhance the console to allow text to be printed in different colors. The traditional way to do this is to make it interpret [ANSI escape sequences](#) embedded in the text strings printed to the console, but you may use any mechanism you like. There is plenty of information on [the reference page](#) and elsewhere on the web on programming the VGA display hardware. If you're feeling really adventurous, you could try switching the VGA hardware into a graphics mode and making the console draw text onto the graphical frame buffer.

این چالش می‌گوید روش کار اینکه کنسول امکان چاپ متن در رنگ‌های مختلف را داشته باشد چیست. مراحل کار به صورت زیر است.

۱- درک دنباله‌ی ANSI Escape :

توالی‌های ANSI Escape یک روش متداول برای کنترل ویژگی‌های متن در برنامه‌های ترمینال است. این دنباله‌ها با کاراکتر escape که معمولاً در زبان C بصورت '\033' است شروع می‌شوند و کدهای خاصی برای تنظیم ویژگی‌هایی مانند رنگ متن دارند. برای مثال '\033[31m' رنگ متن را به قرمز تغییر می‌دهد.

۲- تغییر تابع خروجی کنسول:

باید تابعی که مسئول چاپ متن در کنسول است را تغییر دهیم. این تابع باید دنباله‌های ANSI Escape را شناسایی و تفسیر کند و رنگ متن را مطابق با آن تنظیم کند و هنگامیکه با یک ANSI Escape برای تغییر رنگ مواجه می‌شود، باید تنظیمات رنگ را به روز کند.

۳- تنظیم پالت رنگ:

باید در مورد مجموعه‌ای از رنگ‌هایی که می‌خواهید استفاده کنید تصمیم بگیرید و آنها را به کدهای رنگی ANSI نگاشت کنید.

۴- به روزرسانی سخت‌افزار VGA :

برای تنظیم رنگ متن باید با سخت‌افزار VGA یک اینترفیس داشته باشید. این شامل تغییر رجیسترهای VGA است که رنگ‌های پیش‌زمینه و پس‌زمینه را کنترل می‌کنند.

۵- تست:

کنسول پیشرفته خود را با چاپ متن با توالی فرار رنگی مختلف آزمایش کنید و تغییر رنگ را مشاهده کنید.

## تمرین ۸:

**Exercise 8.** 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?

هسته پشته خود را در `entry.S` با اجرای خط زیر مقداردهی می‌کند:

```
movl $(bootstacktop), %esp
```

از آنجایی که پشته پایین می‌آید، `bootstacktop` جایی است که نشانگر پشته در ابتدا به آن اشاره می‌کند و به سمت آدرس‌های پایین‌تر بخش داده‌ها رشد می‌کند. آدرس `bootstacktop` در قسمت `data` در آفست برابر با `KSTKSIZE` تعریف شده است. کرنل ناحیه استک را در زیرمجموعه `data` رزرو می‌کند.

با توجه به `entry.S` داریم:

```
Movabs $(bootstcktop), %rax
```

```
Movq %rax, %rsp
```

و با توجه به `kernel.asm`:

```
Movabs $(bootstacktop), %rax
```

```
800420003d: 48 b8 00 c0 21 04 80 movabs $0x800421c000,%rax
```

```
8004200044: 00 00 00
```

```
Movq %rax,%rsp
```

```
8004200047: 48 89 c4 mov %rax,%rsp
```

پس می‌توانیم نتیجه بگیریم آدرس شروع استک `0x800421c000` است.

## تمرین ۹:

**Exercise 9.** To become familiar with the C calling conventions on the x86-64, 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 64-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 your course virtual machine. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

با دستور `$sp p` در `gdb` می‌توانیم ببینیم که هر فراخوانی این آدرس‌ها را برای نشان‌گر پشته به ما می‌دهد.

```
$14 = (void *) 0xf010ffdc (address right after test_backtrace(5) was called)
$15 = (void *) 0xf010ffbc (address right after test_backtrace(4) was called)
$16 = (void *) 0xf010ff9c (address right after test_backtrace(3) was called)
$17 = (void *) 0xf010ff7c (address right after test_backtrace(2) was called)
```

هر بار ۸ کلمه ۴ بایتی را پوش می‌کند زیرا تفاوت rbp بین دو نقطه شکست 0x20 است.

return address

saved ebp

saved ebx

abandoned

abandoned

abandoned

abandoned

var x for calling next test\_backtrace

## تمرین ۱۰:

**Exercise 10.** Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

پیاده‌سازی کد back\_trace :

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    uint32_t my_ebp; // this is the frame pointer of mon_trace itself.
    asm volatile("movl %%ebp,%0" : "=r" (my_ebp));
    cprintf("Stack backtrace:\n");
    uint32_t ebp = my_ebp;
    while (ebp != 0) {
        uint32_t eip = *((uint32_t*)ebp + 1);
        cprintf("ebp %08x eip %08x args", ebp, eip);
        for (int i = 2; i < 7; i++) {
            uint32_t arg = *((uint32_t*)ebp + i);
            cprintf(" %08x ", arg);
        }
        cprintf("\n");
        ebp = *(uint32_t*)ebp;
    }
    return 0;
}
```



## تمرین ۱۱:

**Exercise 11.** Modify your stack backtrace function to display, for each `rip`, the function arguments, the function name, source file name, and line number corresponding to that `rip`.

Add a backtrace command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_rip` and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
rbp 00000800421af00 rip 0000080042010ff
    kern/monitor.c:86: mon_backtrace+0000000000000035 args:3 0000000000000000 00000000421b909 0000000000000080
rbp 00000800421afb0 rip 00000800420144d
    kern/monitor.c:163: runcmd+000000000000001d3 args:2 0000000000000001 0000000000000002
rbp 00000800421afe0 rip 000008004201508
    kern/monitor.c:185: monitor+000000000000007d args:1 0000000000000080
rbp 00000800421aff0 rip 000008004200196
    kern/init.c:172: i386_init+00000000000000ba args:0
```

Each line gives the file name and line within that file of the stack frame's `rip`, followed by the name of the function and the offset of the `rip` from the first instruction of the function (e.g., `monitor+106` means the return `rip` is 106 bytes past the beginning of `monitor`), followed by the number of function arguments and then the actual arguments themselves.

Hint: for the function arguments, take a look the struct `Ripdebuginfo` in `kern/kdebug.h`. This structure is filled by the call to `debuginfo_rip`. The x86\_64 calling convention states that the function arguments are pushed onto the stack. Refer to [this article](#) on the calling convention to figure out how to read the actual function arguments on the stack.

Be sure to print the file and function names on a separate line, to avoid confusing the grading script.

Tip: `printf` format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in the DWARF2 tables. `printf("%.s", length, string)` prints at most `length` characters of `string`. Take a look at the `printf` man page to find out why this works.

You may find that the some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUmakefile`, the backtraces may make more sense (but your kernel will run more slowly).

فایل لینکر نگاهی ببندازیم:  
 نمادهایی هستند که در اسکریپت پیوند دهنده `kernel.ld` تعریف شده‌اند. اگر به `__STAB_BEGIN__` و `__STAB_END__`

```
.stab : {
    PROVIDE(__STAB_BEGIN__ = .);
    *(.stab);
    PROVIDE(__STAB_END__ = .);
    BYTE(0) /* Force the linker to allocate space
            for this section */
}
```

اگر بخواهیم بررسی کنیم که این نمادها چه آدرسی دارند، می‌توانیم امتحان کنیم.

```
jos@Zare-Hosseini: ~/jos_f23_401131008_401131005
jos@Zare-Hosseini:~/jos_f23_401131008_401131005$ objdump -h obj/kern/kernel

obj/kern/kernel:      file format elf64-x86-64

Sections:
Idx Name              Size      VMA               LMA               File off  Algn
  0 .bootstrap          00007004  0000000000010000  0000000000010000  00001000  2**12
    CONTENTS, ALLOC, LOAD, CODE
  1 .text               0000932f  000008004200000  0000000000200000  00009000  2**2
    CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .rodata             00000f8f  000008004209340  0000000000209340  00012340  2**5
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .eh_frame           00000dc8  00000800420a2d0  000000000020a2d0  000132d0  2**3
    CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data               000106a0  00000800420c000  000000000020c000  00015000  2**12
    CONTENTS, ALLOC, LOAD, DATA
  5 .bss                000016a0  00000800421c6a0  000000000021c6a0  000256a0  2**5
    ALLOC
  6 .debug_aranges      000002d0  0000000000000000  0000000000000000  000256a0  2**4
    CONTENTS, READONLY, DEBUGGING
  7 .debug_info         00006f4a  0000000000000000  0000000000000000  00025970  2**0
    CONTENTS, READONLY, DEBUGGING
  8 .debug_abbrev       0000146d  0000000000000000  0000000000000000  0002c8ba  2**0
    CONTENTS, READONLY, DEBUGGING
  9 .debug_line         000019b9  0000000000000000  0000000000000000  0002dd27  2**0
    CONTENTS, READONLY, DEBUGGING
```

در آخر با استفاده دستور make grade نتیجه زیر حاصل می‌شود.

```
josh@Zare-Hosseini: ~/jos_f23_401131008_401131005
+ cc kern/syscall.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ cc kern/libdwarf_rw.c
+ cc kern/libdwarf_frame.c
+ cc kern/libdwarf_lineno.c
+ cc kern/elf_rw.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld obj/boot/boot
boot block is 498 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory `/home/jos_f23_401131008_401131005'
running JOS: (1.7s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 10/10
```

اضافه شدن lab1 به گیت هاب :

```
josh@Zare-Hosseini: ~/Desktop/jos-master/lab1
to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'josh@Zare-Hosseini.(none)')
josh@Zare-Hosseini:~/Desktop/jos-master/lab1$
josh@Zare-Hosseini:~/Desktop/jos-master/lab1$
josh@Zare-Hosseini:~/Desktop/jos-master/lab1$ git config --global user.email "amr
rz.z.1379@gmail.com"
josh@Zare-Hosseini:~/Desktop/jos-master/lab1$ git config --global user.name "amir
rezazare1379"
josh@Zare-Hosseini:~/Desktop/jos-master/lab1$ git commit -m "lab1 Done!"
[master (root-commit) 54903cc] lab1 Done!
47 files changed, 6763 insertions(+)
create mode 100644 .dir-locals.el
create mode 100644 .gdbinit.tmpl
create mode 100644 .gitignore
create mode 100644 CODING
create mode 100644 GNUmakefile
create mode 100644 README.md
create mode 100644 boot/Makefrag
create mode 100644 boot/boot.S
create mode 100644 boot/main.c
create mode 100644 boot/sign.pl
create mode 100644 conf/env.mk
create mode 100644 conf/lab.mk
create mode 100644 fs/test.c
create mode 100644 fs/testshell.key
create mode 100755 grade-lab1
create mode 100644 gradelib.py
create mode 100644 inc/COPYRIGHT
create mode 100644 inc/assert.h
create mode 100644 inc/elf.h
create mode 100644 inc/error.h
create mode 100644 inc/kbereg.h
create mode 100644 inc/memlayout.h
create mode 100644 inc/mmu.h
create mode 100644 inc/stab.h
```