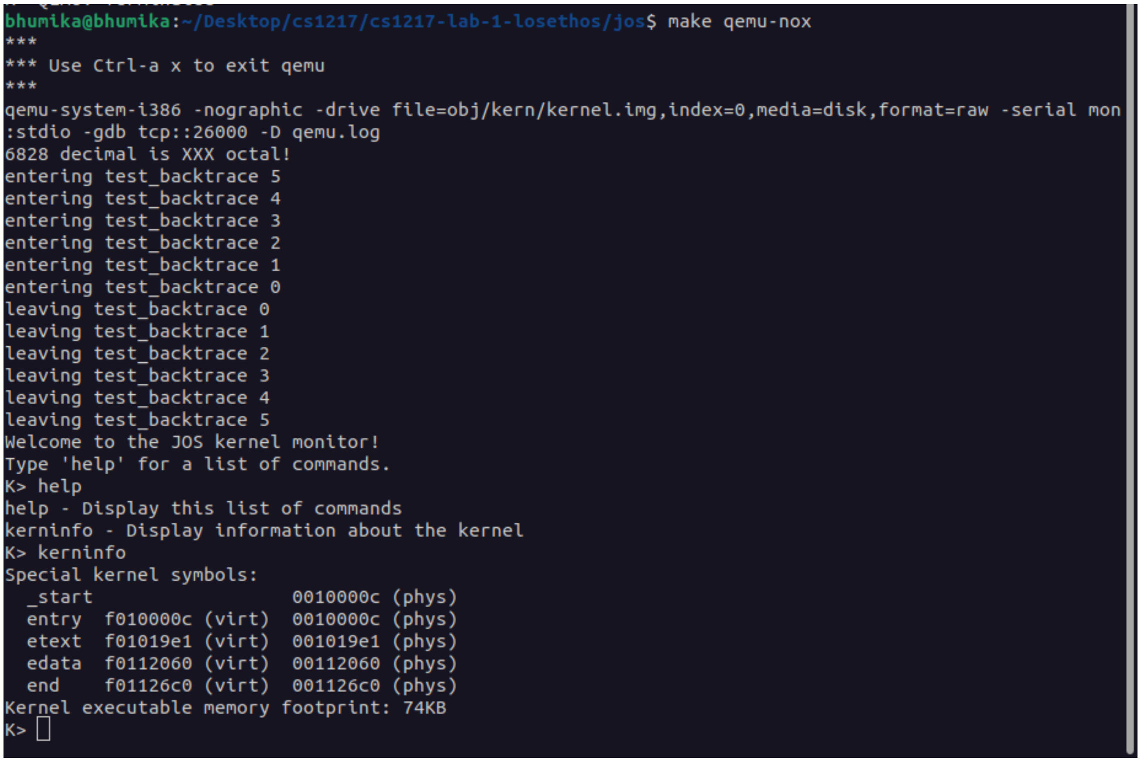
CS1217 - Spring 2023 - Lab 1

Bhumika Mittal, Saptarishi Dhanuka

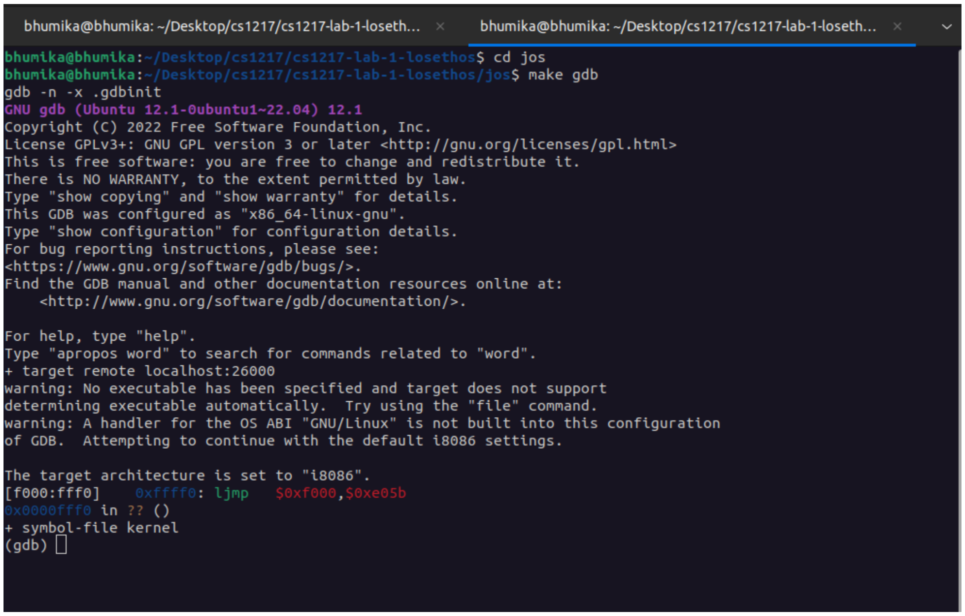
Part 1: PC Bootstrap



```
bhumika@bhumika:~/Desktop/cs1217/cs1217-lab-1-losethos/jos$ make qemu-nox
***
*** Use Ctrl-a x to exit qemu
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -serial mon
:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is XXX 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> kerninfo
Special kernel symbols:
  _start                  0010000c (phys)
  entry  f010000c (virt)  0010000c (phys)
  etext  f01019e1 (virt)  001019e1 (phys)
  edata  f0112060 (virt)  00112060 (phys)
  end    f01126c0 (virt)  001126c0 (phys)
Kernel executable memory footprint: 74KB
K>
```

Make qemu: help and kerninfo

The ROM BIOS



```
bhumika@bhumika: ~/Desktop/cs1217/cs1217-lab-1-loseth...    bhumika@bhumika: ~/Desktop/cs1217/cs1217-lab-1-loseth...
bhumika@bhumika:~/Desktop/cs1217/cs1217-lab-1-losethos$ cd jos
bhumika@bhumika:~/Desktop/cs1217/cs1217-lab-1-losethos/jos$ make gdb
gdb -n -x .gdbinit
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is set to "i8086".
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb)
```

What the BIOS does with its first few instructions is initialize various registers like dx (data register), ss (stack segment register), esp, edx, then jumps to an earlier location in the BIOS. cli clears the interrupt flag, thus ensuring no interrupts occur and cld clears the direction flag, thus ensuring that string operations increment forward. Then out and in instructions are used for talking with hardware devices through the use of ports and registers like al. Ports about the CMOS RAM Real Time Clock, Data Port and system devices are used as per their port numbers. Then it loads the interrupt descriptor table and the global descriptor table registers and sets the first bit of control register cr0 to 1, which enables protected mode. It then jumps to a memory location and switches to 32 bit mode.

Part 2: The Boot Loader

Exercise 3. *At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?*

In the xv6 operating system, the switch from 16-bit to 32-bit mode happens during the boot process. The boot loader switches the processor from real mode to 32-bit protected mode, because it is only in this mode that software can access all the memory above 1MB in the processor's physical address space.

When the boot loader loads the kernel image into memory, it sets the processor's mode to 32-bit protected mode before jumping to the kernel's entry point.

The exact point where the processor start executing 32-bit mode is the line 60 in the boot.S file

```
movw    $PROT_MODE_DSEG, %ax
```

In real mode, the PE bit in the cr0 control register is set to 0, indicating that the processor is operating in 16-bit real mode. The following instruction switches the PE bit to 1 in the cr0, thus enabling the 32-bit protected mode.

```
mov    %eax,%cr0
```

The following line (line 55) in jos/boot/boot.S file, switches processor into 32-bit mode.

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

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

After the for loop which calls readseg(), the last instruction of the bootloader in main.c is

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

In boot.asm, the last instruction is right below the above line as:

```
7d71: ff 15 18 00 01 00      call   *0x10018
```

Since the last instruction of the bootloader is calling *0x10018, we must see what is at this location. We first set a breakpoint at 7d71 and then single step.

```
Breakpoint 1, 0x00007c00 in ?? ()
(gdb) br * 0x7d71
Breakpoint 2 at 0x7d71
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d71:      call   *0x10018

Breakpoint 2, 0x00007d71 in ?? ()
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
```

Hence the first instruction of the kernel it just loaded is:

```
=> 0x10000c:    movw   $0x1234,0x472
```

(gdb) x/i *0x10018 also gives:

```
   0x10000c:    movw   $0x1234,0x472
```

Where is the first instruction of the kernel?

As shown above the first instruction of the kernel is at **0x10000c**

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?

The necessary information for the boot loader is given in the Executable and Linkable Format (ELF) header. In main.c, we define ELFHDR as a pointer to struct ELF with value 0x1000. Then, after calling readseg once, we define eph to be a pointer to a struct Proghdr which points to the end of the program header table since we used e_phnum to define it (e_phnum is number of elements in program header table which describes information that the bootloader needs to prepare for the kernel to execute).

Hence, it decides how many sectors based on the value of the phnum field in ELFHDR and simply loops through them all.

Loading the Kernel

Exercise 5. *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.*

We changed the Makefrag to read, instead of 7C00,

```
-Ttext 0x7D00
```

Then after make clean and make, we debugged with gdb by setting a breakpoint at 0x7c00 and continuing execution.

By single stepping, the first instruction to be different was :

0x7c1e: lgdtw 0x7d64 in the 7D00 case, whereas it was **0x7c1e: lgdtw 0x7c64** in the proper 7C00 case. We can see that the argument for lgdtw has changed from **0x7c64** to **0x7d64,** which is wrong. However, we can still continue execution from here.

The following instruction breaks the whole thing completely:
0x7c2d: ljmp $0x8,$0x7d32

It give the following error:

```
ld: warning: section `.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 396 bytes (max 510)
+ mk obj/kern/kernel.img
xv6@xv6host:~/lab1/cs1217-lab-1-losethos/jos$ make qemu-nox-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
***
*** Now run 'make gdb'.
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=
raw -serial mon:stdio -gdb tcp::26000 -D qemu.log  -S
EAX=00000011 EBX=00000000 ECX=00000000 EDX=00000080
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00006f20
EIP=00007c2d EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
CS =0000 00000000 0000ffff 00009b00 DPL=0 CS16 [-RA]
SS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
DS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
FS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
GS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     006ee8ec 000073ff
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

```
The target architecture is set to "i8086".
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) br * 0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7d32
0x00007c2d in ?? ()
(gdb)
```

The lgdtw instruction gets a wrong argument hence a wrong value is loaded into the global descriptor table register

The ljmp instruction which switches from 16 bits to 32 bits breaks everything if the wrong link address is given in boot/Makefrag

Also note that, after the above instruction, the bootloader goes into the infinite spin loop (which only gets executed when the bootmain returns

```
OUTPUT    DEBUG CONSOLE    TERMINAL    GITLENS

(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,$0x7d32
0x00007c2d in ?? ()
(gdb) si
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7d32
0x00007c2d in ?? ()
(gdb) si
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7d32
0x00007c2d in ?? ()
(gdb) si
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7d32
0x00007c2d in ?? ()
(gdb) si
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7d32
0x00007c2d in ?? ()
(gdb)
```

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?*

First we set a breakpoint at 0x7c00 (where the BIOS enters the boot loader) and continue to it and single step. We see that 8 words of memory around the given memory location are all 0

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

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) si
[   0:7c01] => 0x7c01:  cld
0x00007c01 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
```

Then we set a breakpoint at 0x7d71, which is where the boot loader enters the kernel. Examining 8 words of memory shows change.

```
Breakpoint 2 at 0x7d71
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x7d71:      call   *0x10018

Breakpoint 2, 0x00007d71 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x1000b812      0x220f0011      0xc0200fd8
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(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:    adc    %dl,(%ecx)
   0x100019:    add    %cl,(%edi)
   0x10001b:    and    %al,%bl
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x1000b812      0x220f0011      0xc0200fd8
```

At the second breakpoint, the boot loader has loaded the kernel into memory starting from the kernel's load address 0x00100000, and hence the 8 words we see are the first few words of memory of the kernel program. Meanwhile, at the first breakpoint, the BIOS has just entered the boot loader and hence there is nothing at address 0x00100000, which is why they are different.

Part 3: The Kernel

Using virtual memory to work around position dependence

Exercise 7. *What is the first instruction after the new mapping is established that would fail to work properly if the mapping weren't in place?*

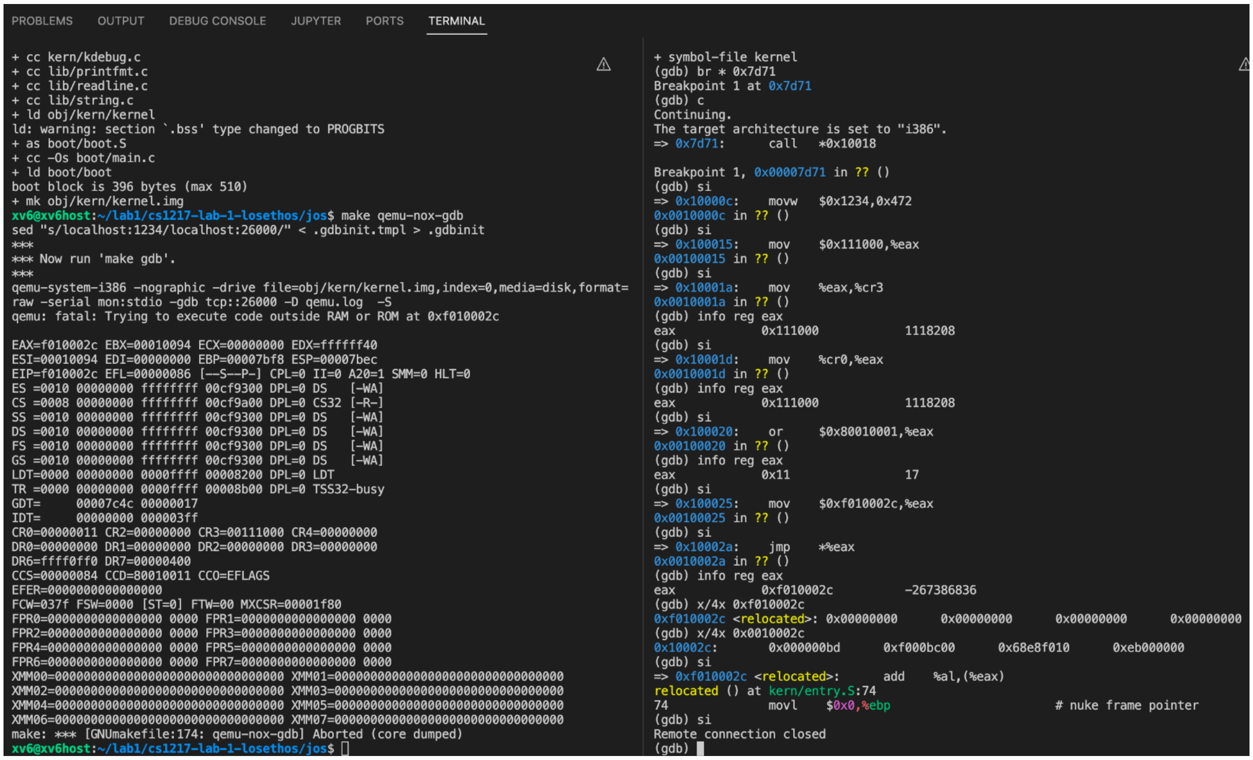
The following screenshot shows that after mov %eax, %cr0, a mapping has been established between 0x00100000 and 0xf0100000, along with the other memory location ranges mentioned in the question. The JOS kernel has set up a mapping between the virtual address 0xf0100000 and the physical address 0x00100000, so any memory accesses to 0xf0100000 will be translated to 0x00100000. Here, memory references are getting translated by the virtual memory hardware to physical addresses.

```
(gdb) si
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe   0x7205c766
0x100010:       0x34000004      0x1000b812      0x220f0011   0xc0200fd8
(gdb) x/5x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe   0x7205c766
0x100010:       0x34000004
(gdb) x/4x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe   0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start-268435468>:  0x00000000      0x00000000   0x00000000      0x00000000
(gdb) si
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe   0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start-268435468>:  0x00000000      0x00000000   0x00000000      0x00000000
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/4x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
(gdb) x/4x 0xf0100000
0xf0100000 <_start-268435468>:  0x1badb002      0x00000000      0xe4524ffe   0x7205c766
```

The first instruction that wouldn't work as it is supposed to is:

```
jmp *%eax
```

After commenting out *movl %eax, %cr0*, running *make clean* then *make* and using gdb, we get the following results and error:



This is because we first move the address of relocated to eax, which is 0xf010002c, which we can see does not have anything in it since mapping has not been established. Then we are jumping to that address, which is incorrect hence leading to an error.

Without the virtual memory mapping in place, this instruction would set the stack pointer to an incorrect physical address, causing memory access errors and ultimately leading to the segmentation fault.

Formatted Printing to the Console

Exercise 8. *A small fragment of code has been omitted - the code necessary to print octal numbers using patterns of the form "`%o`". Find and fill in this code fragment.*

Added the following code necessary to print octal numbers using patterns of the form "%o".

```
            num = getuint(&ap, lflag);
            if (altflag && num != 0){
                putch('0', putdat);
            }
            base = 8;
            goto number;
```

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

printf.c is responsible for the implementation of the cprintf() function which prints output to the console. The function that *console.c* exports that is used by *printf.c* is the **cputchar()** function, which in turn just calls the *cons_putc()* function, which is also defined in console.c

printf.c uses the cputchar() function in the following way: when the user calls the cprintf() function, it calls another function vcprintf() which formats the string using vprintfmt(), and outputs it to the console using the putch() function, which ultimately calls cputchar().

Therefore, console.c manages the printing of the characters to the console whereas printf.c formats the output and passes it to the console.c to display the output. cputchar() and (indirectly) consputc() is used by printf.c to **output each character to the console**.

Explain the code from `console.c`:

This code is a very simple implementation of scrolling the console display. The main purpose of this code is if the cursor is going off the screen, scroll up.

If the current position of the cursor (crt_pos) is greater than or equal to the total size of the console buffer (CRT_SIZE), then we need to scroll up because the display is at the bottom of the screen. To do this, the memmove() function (it copy n bytes from src to dst) takes crt_buf + CRT_COLS as source, and moves (CRT_SIZE - CRT_COLS) * sizeof(uint16_t) number of bytes to crt_buf. Here, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t) is the size of the remaining console buffer. Basically, this is moving contents of the console buffer up by one row.

Then the for loop effectively clears the last row of the console buffer and sets its attributes to 0x0700 (white text on black background). The loop iterates through the last row, setting each character to a space.

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);
```

We can put this code in monitor.c just after

```
cprintf("Type 'help' for a list of commands.\n");
```

and use gdb to step into the function to help us to see the step by step execution of the code. We can also consult kernel.asm

Output of code: “x 1, y 3, z 4”

Step by step execution of the code:

a. Declare and initialize x=1, y=3, and z=4
b. Call cprintf function with the arguments: format string "x %d, y %x, z %d\n" and the variables x,y,z
c. Declare a va_list (a type defined in stdarg.h that is used to iterate through a variable number of arguments passed to a function) ap and an integer cnt.
d. Call vcprintf with arguments as the format string and the argument list.
e. vcprintf will iterate over the format string and calls putch function for each character. When %d is encountered, then it calls printnum to print the number in decimal format. Similarly for %x and %o it prints the number in hexadecimal and octal format. For \n, it calls putch
f. printnum function calls putch for each character in the string (the integer argument is converted to string)
g. putch calls cons_putc from console.c and prints each character to the console. It also checks if the character is a newline. If it is a newline character, it updates the crt_pos by one (if it is at the end of the display, it scrolls up).
h. cons_putc writes to the console buffer at the current crt_pos and keeps updating it.
i. vcprintf then returns and transfers the control back to the caller, cprintf, which also then returns.

NOTE - va_start initializes the va_list to the first argument after fmt and va_end cleans up the va_list.

fmt points to the format string "x %d, y %x, z %d\n" and **ap** points to argument *x* since va_list is initialized as the first argument

We used gdb for this part along with logically tracing the code and finding where the functions would execute and what their arguments would be:

- vcprintf(fmt, ap)
  fmt points to "x %d, y %x, z %d\n" and ap points to the value in x. Gdb gave me the values as vcprintf (fmt=0xf0101d0e "x %d, y %x, z %d\n", ap=0xf010ef54 "\001") since the value in *x* is 1
- cons_putc () with argument 120, which is *x* in ASCII
- cons_putc() which argument 32 // ASCII for space
- Then call to va_arg. Before the call, ap points to “\001”, but the next call will set ap to the next number to be printed i.e. “\003”
- cons_putc(49) // ascii for 1
- Then cons_putc() is called again and again with arguments as the ascii values for comma, space, y, space.
- va_arg()
  Before: points to “\003”
  After: points to “\004”
- cons_putc() called multiple times with arguments as ascii values for 3, comma, space, z, space
- va_arg()
  Before: points to “\004”
  After: Garbage value
- cons_putc() called twice with arguments as ascii values for 4 and newline.

For va_arg() answers we checked the values of the memory locations with gdb starting from the initial value of ap = 0xf010ef54 till 0xf010ef60 and they checked out with the logical analysis

```
(gdb) x/4d 0xf010ef54
0xf010ef54:     1       0       0       0
(gdb) x/4d 0xf010ef58
0xf010ef58:     3       0       0       0
(gdb) x/4d 0xf010ef5c
0xf010ef5c:     4       0       0       0
(gdb) x/4d 0xf010ef60
```

NOTE – We then removed the lines we added to monitor.c and didn’t commit it

What is the output?

Output – **He110 World**

57616 in decimal is E110 in hex and 0x00646c72 is stored as 72 6c 64 00 which is the ASCII for “rld\0”.

If the x86 were instead big-endian, we would need to reverse the order of bytes in i such that it still contains the ASCII for “rld\0”. Converting 0x00646c72 to big-endian, we get 0x726c6400. In order to yield the same output, set **i = 0x726c6400**. There is **no need to change 57616** to a different value because this value is passed as an argument to cprintf in a way such that it prints this unsigned hexadecimal integer as ASCII. It is not affected by the byte order of x86.

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

Output: x=3 y=734328248

The format string contains two integer placeholders but only one int argument is provided. Observe that the first placeholder correctly takes the value 3 but the second one takes a **garbage value**. When the code is executed, it sometimes prints a garbage value or can even crash the program sometimes.

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?

One way to do this is to pass the number of arguments to cprintf as the first parameter, before the format string. We can then iterate through the arguments using a loop inside cprintf and the va_arg macro, which will allow us to retrieve each argument from the variable argument list. By this method, we would be able to determine the location of each argument on the stack.

The Stack

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?*

The kernel initializes its stack through the following instructions in entry.S

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

backstacktop is defined in bootasm.S and is the top of the stack, defined as the end of the bootstack section (defined by .space directive, which reserves KSTKSIZE bytes of memory for the kernel stack). From kernel.sym we can see that the stack starts from the address 0xf0107000.

KSTKSIZE defines the size of the stack. The stack pointer is initialized to point to the end of the bootstack section (which is bootstacktop, located at 0xf010f000 in memory). Kernel starts from here in memory and grows downwards (hence, we observe lower memory addresses) in memory. Also, the difference between 0xf010f000 and 0xf0107000 tells the size of the stack which is 8kb.

The kernel reserves the space for its stack using the .space directive as follows:

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

Exercise 10. *How many 32-bit words does each recursive nesting level of* `test_backtrace` *push on the stack, and what are those words*?

The address of the test_backtrace function is **0xf0100040**.

```
Breakpoint 1, test_backtrace (x=3) at kern/init.c:13
13      {
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=2) at kern/init.c:13
13      {
(gdb) p $sp
$5 = (void *) 0xf010ef7c
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=1) at kern/init.c:13
13      {
(gdb) p $sp
$6 = (void *) 0xf010ef5c
(gdb)
```

We check the address right after each call of test_backtrace. The difference between the addresses of two successive calls is 32 bytes. Therefore, each recursive nesting level of test_backtrace pushes 8 words (each word is 4 bytes, hence 32 bytes is 8 words) on the stack.

From the kernel.asm, and checking the stack values, it contains the following address.

```
Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) x/8x 0xf0100040
0xf0100040 <test_backtrace>:    0x56e58955      0x0172e853      0xc3810000      0x000102be
0xf0100050 <test_backtrace+16>: 0x8308758b      0x8d5608ec      0xff17d883      0xfae850ff
(gdb) x/8x $esp
0xf010efdc:     0xf01000f4      0x00000005      0x00001aac      0x00000660
0xf010efec:     0x00000000      0x00000000      0x00010094      0x00000000
```

This includes the value of x(5), x-1(4), the return address, previous ebp, and some arguments.

```
    // Test the stack backtrace function (lab 1 only)
    test_backtrace(5);
f01000e8:    c7 04 24 05 00 00 00    movl   $0x5,(%esp)
f01000ef:    e8 4c ff ff ff          call   f0100040 <test_backtrace>
f01000f4:    83 c4 10                add    $0x10,%esp

    // Drop into the kernel monitor.
    while (1)
        monitor(NULL);
f01000f7:    83 ec 0c                sub    $0xc,%esp
f01000fa:    6a 00                   push   $0x0
f01000fc:    e8 a0 07 00 00          call   f01008a1 <monitor>
f0100101:    83 c4 10                add    $0x10,%esp
f0100104:    eb f1                   jmp    f01000f7 <i386_init+0x51>
```

```
// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
f0100040:   55                      push   %ebp
f0100041:   89 e5                   mov    %esp,%ebp
f0100043:   56                      push   %esi
f0100044:   53                      push   %ebx
f0100045:   e8 72 01 00 00          call   f01001bc <__x86.get_pc_thunk.bx>
f010004a:   81 c3 be 02 01 00       add    $0x102be,%ebx
f0100050:   8b 75 08                mov    0x8(%ebp),%esi
    cprintf("entering test_backtrace %d\n", x);
f0100053:   83 ec 08                sub    $0x8,%esp
f0100056:   56                      push   %esi
f0100057:   8d 83 d8 17 ff ff       lea    -0xe828(%ebx),%eax
f010005d:   50                      push   %eax
f010005e:   e8 fa 09 00 00          call   f0100a5d <cprintf>
    if (x > 0)
f0100063:   83 c4 10                add    $0x10,%esp
f0100066:   85 f6                   test   %esi,%esi
f0100068:   7e 29                   jle    f0100093 <test_backtrace+0x53>
        test_backtrace(x-1);
f010006a:   83 ec 0c                sub    $0xc,%esp
f010006d:   8d 46 ff                lea    -0x1(%esi),%eax
f0100070:   50                      push   %eax
f0100071:   e8 ca ff ff ff          call   f0100040 <test_backtrace>
f0100076:   83 c4 10                add    $0x10,%esp
```

The backtrace code can’t detect the number of arguments because the function is not keeping track of the number of arguments passed to it. To fix this, we would need to include debugging symbol table in the kernel, which will provide information about the number and type of arguments for each function. Using this information, the code can then keep track of the number of arguments passed to it.

Exercise 11. *Implement the backtrace function as specified above.*

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
   // Your code here.
   uint32_t *ebp;  // base pointer
   asm volatile("movl %%ebp, %0" : "=r" (ebp));        // get base pointer
- asm volatile is used to prevent compiler from optimizing the code
   uint32_t eip;   // instruction pointer
   uint32_t args[5];   // arguments
   int i;

   cprintf("Stack backtrace:\n");
   while(ebp != 0) {        // base pointer is 0 at the end of the stack
       eip = *(ebp + 1);   // get the instruction pointer
       cprintf("ebp %08x  eip %08x  args", ebp, eip);      // print the
base pointer and instruction pointer - %08x is used to print the
hexadecimal value of the pointer
       for(i = 0; i < 5; i++) {
           args[i] = *(ebp + 2 + i);   // get the arguments
           cprintf(" %08x", args[i]);  // print the arguments
       }
       cprintf("\n");
       ebp = (uint32_t *)*ebp; // get the base pointer of the previous
stack frame
   }
   return 0;
}
```

Output:

```
Type 'help' for a list of commands.
K> backtrace
Stack backtrace:
 bp f010ff58 eip f0100a71 args 00000001 f010ff80 00000000 f0100ad5 f0100a84
             kern/monitor.c:136: monitor+333
 bp f010ffd8 eip f0100101 args 00000000 00001aac 00000660 00000000 00000000
             kern/init.c:43: i386_init+91
 bp f010fff8 eip f010003e args 00000023 00001003 00002003 00003003 00004003
             kern/entry.S:83: <unknown>+0
K> 
```

(This also includes the output from ex 12 as the screenshot was taken later.)

This can be also implemented using the read_ebp() function, as mentioned, as follows:

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
   // Your code here.
   uint32_t *ebp = (uint32_t *)read_ebp(); //base pointer
   uint32_t eip = ebp[1];  //eip is the second element of the stack frame.
The first element is the return address of the caller
   cprintf("Stack backtrace:\n");
   while(ebp != 0){
       cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\r \n",
ebp, eip, ebp[2], ebp[3], ebp[4], ebp[5], ebp[6]);  //print the stack
frame - ebp, eip, and the first 5 arguments
       ebp = (uint32_t *)*ebp;      //move to the next stack frame
       eip = ebp[1];    //update eip - without this, the eip will be the
same as the previous stack frame (we dn't want that)
   }
   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.*

To print the line number, we add the following code to kdebug.c

```
       stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);       //
N_SLINE is the type for line numbers in stabs - as mentioned in the header
file
       if (lline <= rline) {                              // if the
line number is found
               info->eip_line = stabs[lline].n_desc;              //
n_desc is the line number in the stab and is stored in the stabs structure
- in this line we are storing the line number in the eip_line variable
       } else {
               return -1;                                       // if
the line number is not found, return -1
       }
```

Using the bt function in gdb to verify the implementation of backtrace line number, we can observe that there is some difference in the line numbers

```
Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) bt
#0  test_backtrace (x=5) at kern/init.c:13
#1  0xf01000f4 in i386_init () at kern/init.c:39
#2  0xf010003e in relocated () at kern/entry.S:80
(gdb) 
```

Example – kern/init.c is at 43 in our implementation whereas it is at 30 in gdb. This is because the gdb implementation gives the line number where the function is called.

In debuginfo_eip, where do __STAB_* come from?

Observe the following from the kern/kernel.ld

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

__STAB_BEGIN__ and __STAB_END__ are symbols defined in the kernel.ld. These are the beginning and end addresses in the .stab section.

From ***objdump -h obj/kern/kernel,*** observe the following -

```
 2 .stab         00003889  f01022ec  001022ec  000032ec  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
 3 .stabstr      00001629  f0105b75  00105b75  00006b75  2**0
                 CONTENTS, ALLOC, LOAD, READONLY, DATA
```

We can also observe these in kernel.asm

```
    // Search the entire set of stabs for the source file (type N_SO).
    lfile = 0;
f0100c50:   c7 45 e4 00 00 00 00    movl   $0x0,-0x1c(%ebp)
    rfile = (stab_end - stabs) - 1;
f0100c57:   c7 c0 ec 22 10 f0       mov    $0xf01022ec,%eax
f0100c5d:   c7 c2 74 5b 10 f0       mov    $0xf0105b74,%edx
f0100c63:   29 c2                   sub    %eax,%edx
f0100c65:   c1 fa 02                sar    $0x2,%edx
f0100c68:   69 d2 ab aa aa aa       imul   $0xaaaaaaab,%edx,%edx
f0100c6e:   83 ea 01                sub    $0x1,%edx
f0100c71:   89 55 e0                mov    %edx,-0x20(%ebp)
```