CS1217 - Spring 2023 - Lab 1

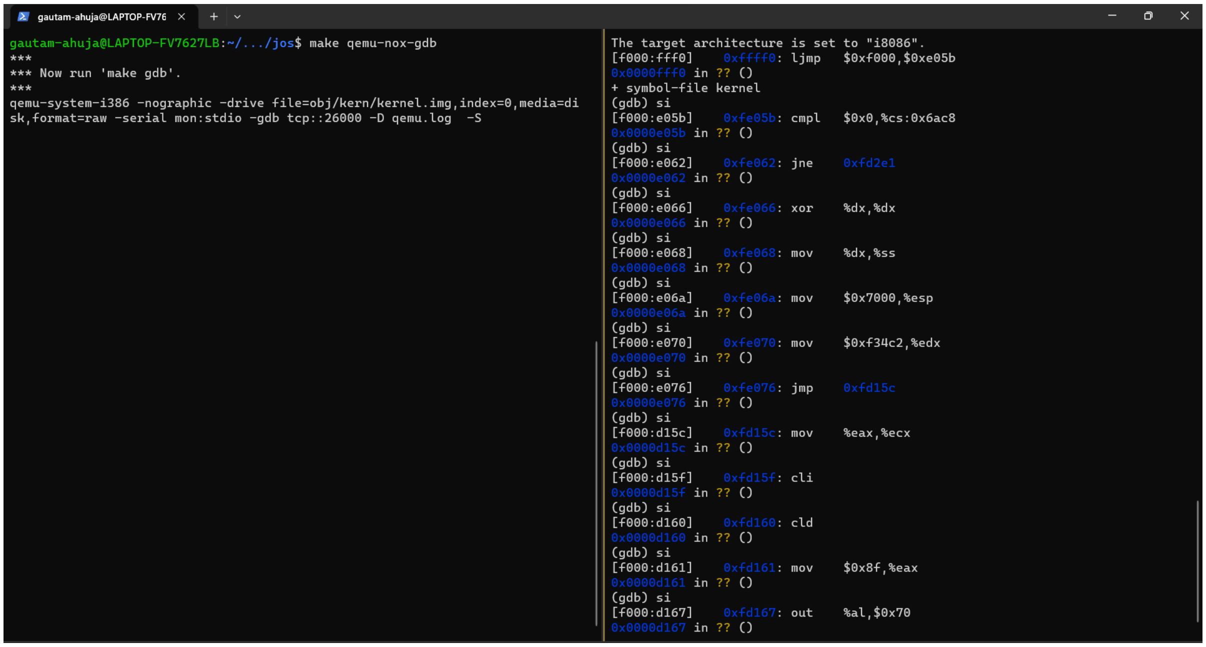
Gautam Ahuja, Nistha Singh

01. Exercise 01

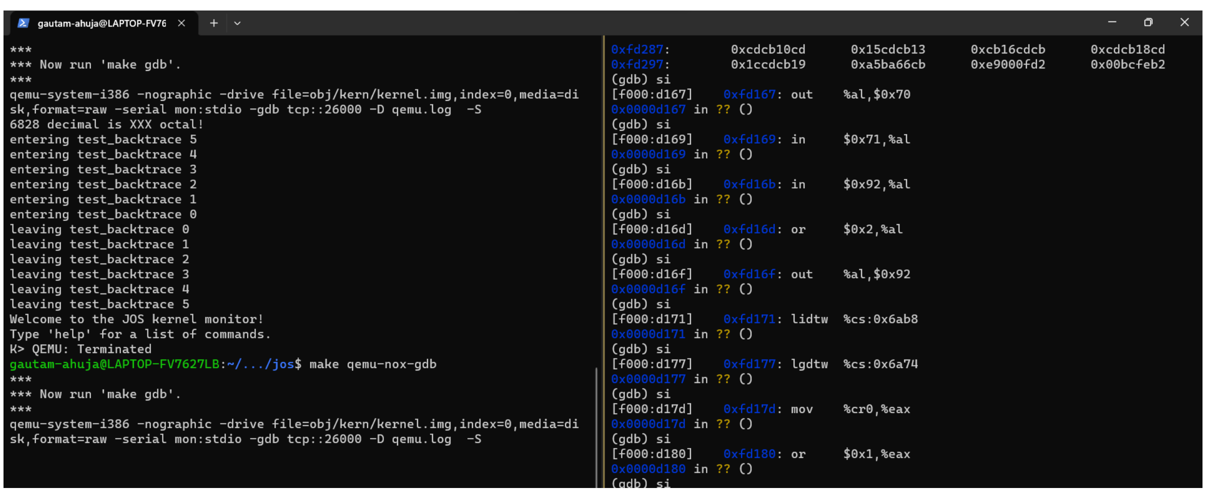
Reading Exercise

02. Exercise 02

On doing a single step of kernel:



- We see that the first instruction `ljmp $0xf000, $0xe05b` is a jump instruction which moves the control within earlier location in BIOS.
  This sets the value in `%cs` to `0xf000` and `%ip` to `0xe05b`.
- In second instruction, it compares the content the address at `0x6ac8` offset from `%cs` to `0x0`.
- Since the contents are equal it skips over `jne 0xfd2e1` and sets values in registers `%ss, %esp, %edx, %ecx,` etc.

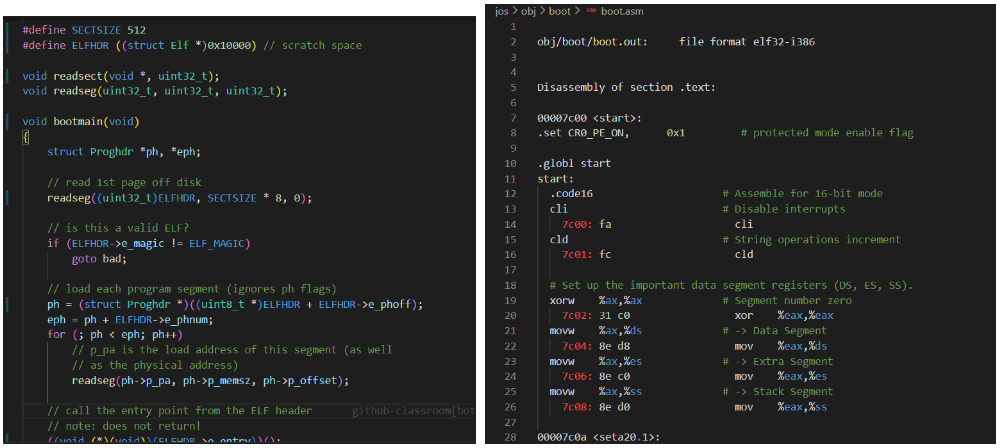


- Few steps further it sets up ports `0x70` and `0x71` in the `%al` register.
  The `%al` register is used to set the I/O ports and operation.

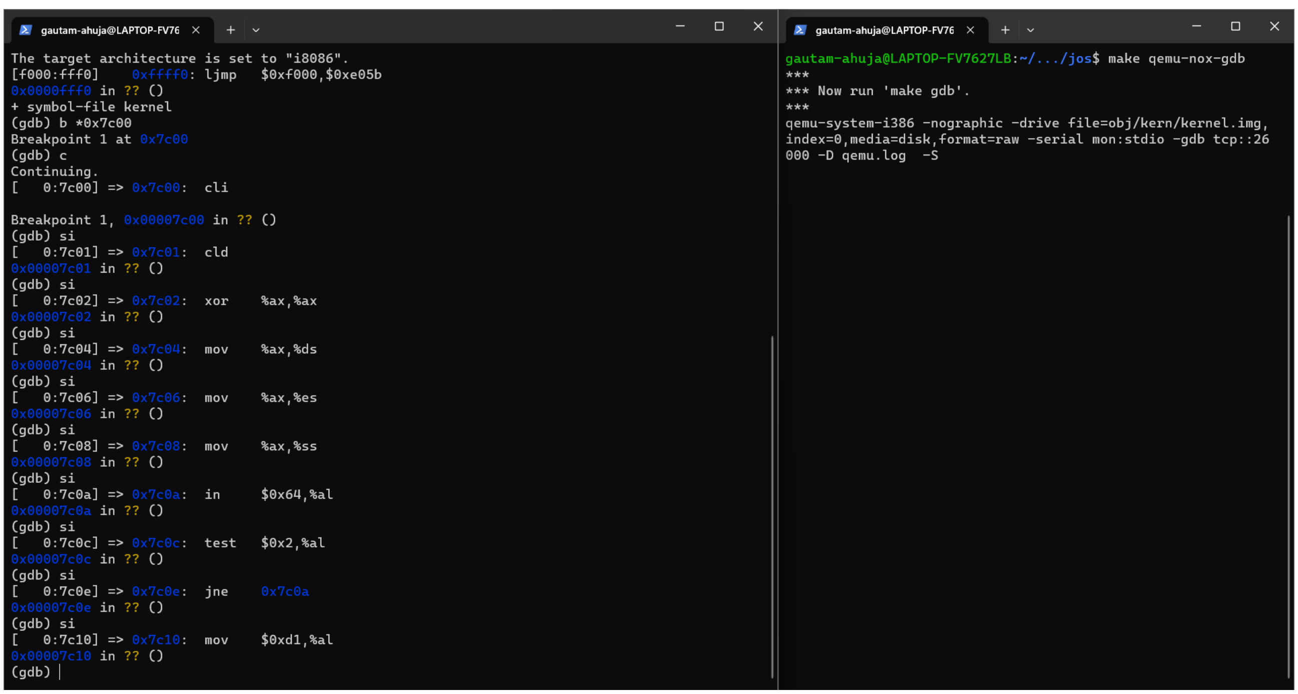
Here the BIOS is setting up the I/O operations (ports, displays, etc) for the kernel to run as initial instructions.

03. Exercise 03

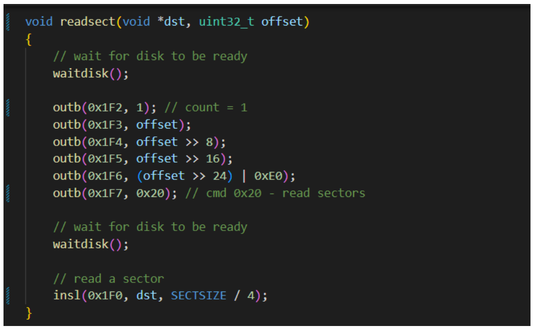
After BIOS has completed all checks and done operations for I/O and PCI ports, it loads the boot loader. The 512 bytes of bootloader are loaded into *The Low Memory* from `0x7c00` to `0x7dff` and the control is passed on the bootloader instructions.
In our case the contents of `boot/main.c` (its compiled assembly - `boot.asm`) gets to run.



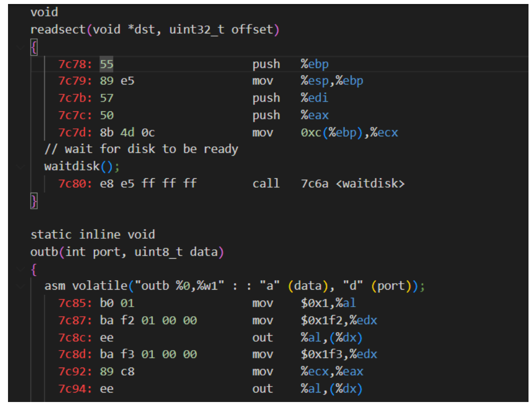
We can set a breakpoint at `0x7c00` and then single spet (`si`) through the bootloader to understand how it loads all the instructions.



Here we can see the instructions corresponds to the `boot.asm` file. Here are the corresponding contents of `readsect()` in both `C` and `asm`.

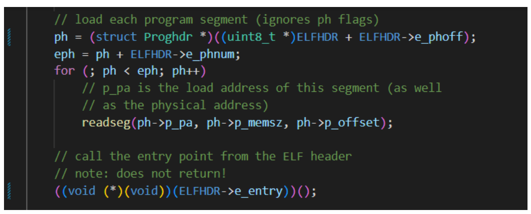


```
void readsect(void *dst, uint32_t offset)
{
    // wait for disk to be ready
    waitdisk();

    outb(0x1F2, 1); // count = 1
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20); // cmd 0x20 - read sectors

    // wait for disk to be ready
    waitdisk();

    // read a sector
    insl(0x1F0, dst, SECTSIZE / 4);
}
```

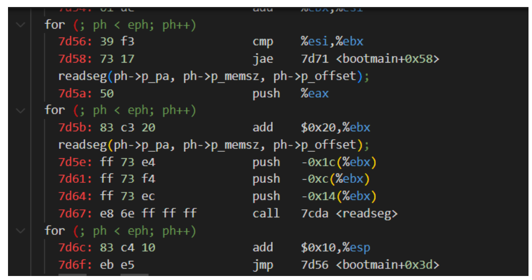


```
void
readsect(void *dst, uint32_t offset)
{
    7c78: 55                    push   %ebp
    7c79: 89 e5                 mov    %esp,%ebp
    7c7b: 57                    push   %edi
    7c7c: 50                    push   %eax
    7c7d: 8b 4d 0c              mov    0xc(%ebp),%ecx
  // wait for disk to be ready
  waitdisk();
    7c80: e8 e5 ff ff ff        call   7c6a <waitdisk>
}

static inline void
outb(int port, uint8_t data)
{
  asm volatile("outb %0,%w1" : : "a" (data), "d" (port));
    7c85: b0 01                 mov    $0x1,%al
    7c87: ba f2 01 00 00        mov    $0x1f2,%edx
    7c8c: ee                    out    %al,(%dx)
    7c8d: ba f3 01 00 00        mov    $0x1f3,%edx
    7c92: 89 c8                 mov    %ecx,%eax
    7c94: ee                    out    %al,(%dx)
```

Below is the image which shows beginning and ed of `for` loop that reads the remaining sectors of the kernel from the disk. We can see (in `asm` file) that loop starts after address `0x7d54` and ends at `0x7d6f`.

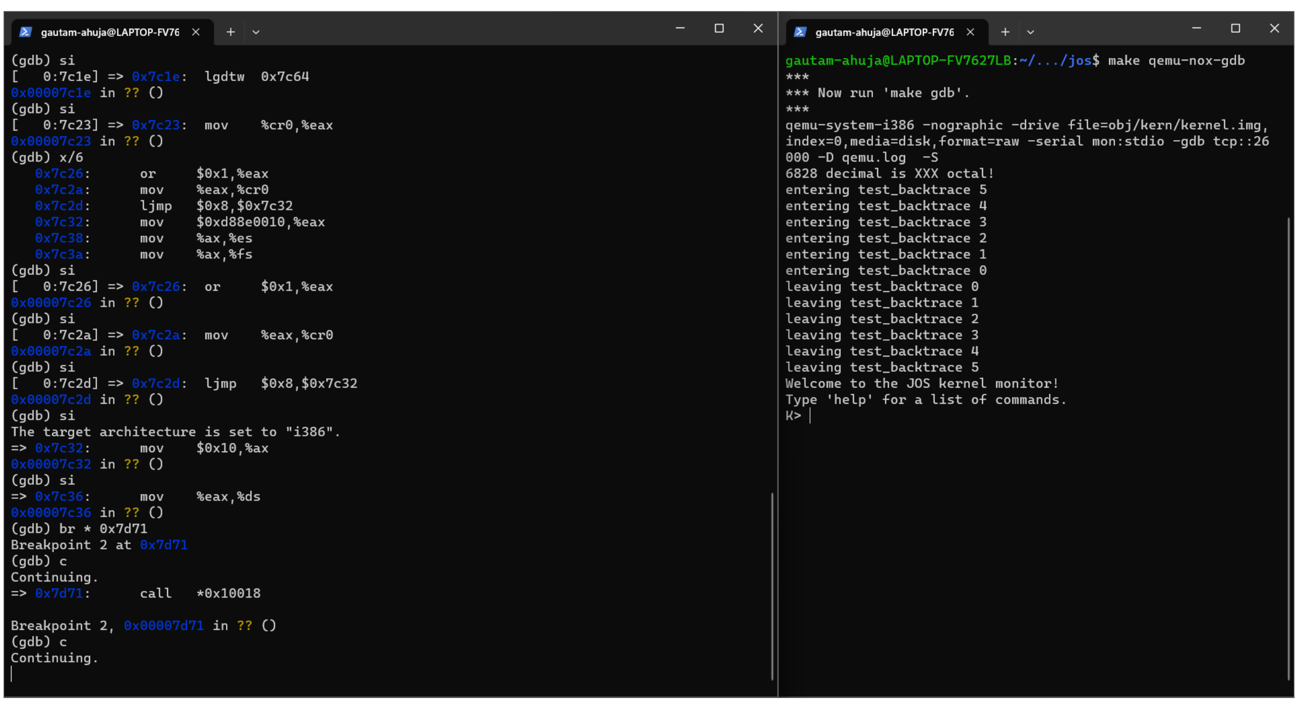


```
    // load each program segment (ignores ph flags)
    ph = (struct Proghdr *)((uint8_t *)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++)
        // p_pa is the load address of this segment (as well
        // as the physical address)
        readseg(ph->p_pa, ph->p_memsz, ph->p_offset);

    // call the entry point from the ELF header
    // note: does not return!
    ((void (*)(void))(ELFHDR->e_entry))();
```



```
  for (; ph < eph; ph++)
    7d56: 39 f3                 cmp    %esi,%ebx
    7d58: 73 17                 jae    7d71 <bootmain+0x58>
    readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
    7d5a: 50                    push   %eax
  for (; ph < eph; ph++)
    7d5b: 83 c3 20              add    $0x20,%ebx
    readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
    7d5e: ff 73 e4              push   -0x1c(%ebx)
    7d61: ff 73 f4              push   -0xc(%ebx)
    7d64: ff 73 ec              push   -0x14(%ebx)
    7d67: e8 6e ff ff ff        call   7cda <readseg>
  for (; ph < eph; ph++)
    7d6c: 83 c4 10              add    $0x10,%esp
    7d6f: eb e5                 jmp    7d56 <bootmain+0x3d>
```

After the loop ends, the bootloader ends and we enter into the kernel through `e_entry` entrypoint. This instruction is at the address `0x7d71`.
We can see below after setting a breakpoint at `0x7d71` and continuing, we enter the enter the kernel, as seen on the right screen.

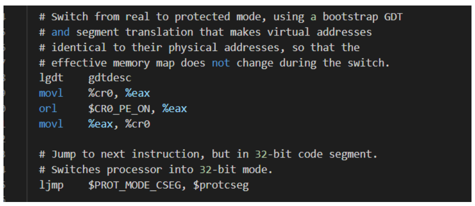


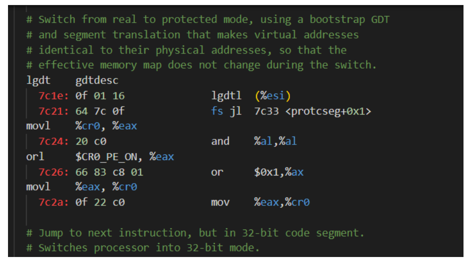
```
gautam-ahuja@LAPTOP-FV76
(gdb) si
[   0:7c1e] => 0x7c1e:  lgdtw  0x7c64
0x00007c1e in ?? ()
(gdb) si
[   0:7c23] => 0x7c23:  mov    %cr0,%eax
0x00007c23 in ?? ()
(gdb) x/6
   0x7c26:      or     $0x1,%eax
   0x7c2a:      mov    %eax,%cr0
   0x7c2d:      ljmp   $0x8,$0x7c32
   0x7c32:      mov    $0xd88e0010,%eax
   0x7c38:      mov    %ax,%es
   0x7c3a:      mov    %ax,%fs
(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 set to "i386".
=> 0x7c32:      mov    $0x10,%ax
0x00007c32 in ?? ()
(gdb) si
=> 0x7c36:      mov    %eax,%ds
0x00007c36 in ?? ()
(gdb) br * 0x7d71
Breakpoint 2 at 0x7d71
(gdb) c
Continuing.
=> 0x7d71:      call   *0x10018

Breakpoint 2, 0x00007d71 in ?? ()
(gdb) c
Continuing.
```

```
gautam-ahuja@LAPTOP-FV76
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ make qemu-nox-gdb
***
*** 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::26
000 -D qemu.log  -S
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>
```

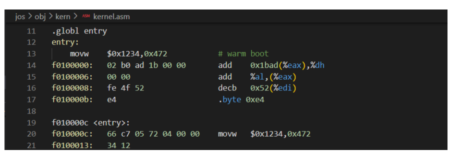
Answers to graded questions:

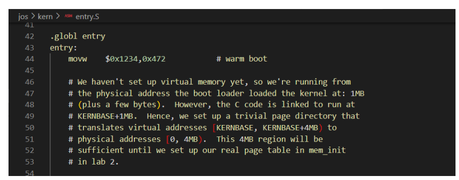
1. The instruction `ljmp $PROT_MODE_CSEG, $protcseg` causes the switch from 16- to 32-bit mode. After that the code execute instructions under `protcseg` label which is `32-bit protected mode`.
   Currently the bootloader is in `16-bit (real)` mode, to executing in `32-bit (protected)` mode it needs to go from `16-bit -> 32-bit` and `real -> protected`. This is done by instructions following `lgdt gdtdesc`. This sets a up a bootstrap GDT (global descriptor table) which does the `16-bit -> 32-bit` conversion and the later instruction do `16-bit -> 32-bit` conversion. We can see even figure it out through comments mentioned in the code:





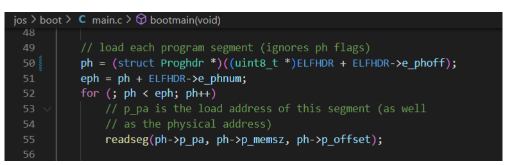
2. The last instruction of bootloader (as discussed above) is `((void (*)(void))(ELFHDR->e_entry))();` in `boot/main.c` and `call *0x10018` in `boot.asm`
   This sets up the entry point to kernel and loads the kernel. The first instruction that kernel executes is: `movw $0x1234, 0x472` as seen both in `kernel.asm` and `entry.S`

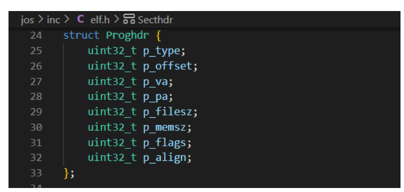




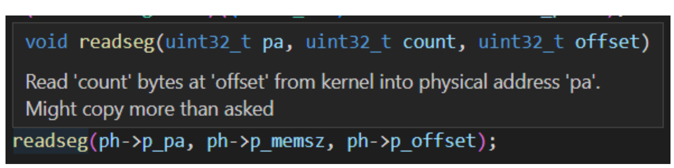
3. The first kernel instruction is located at `0xf010000c` as seen in `kernel.asm`.

4. We look at the code section in `main.c` and `elf.h`





   Here we can see that the `main.c` loads all the ELF headers (# of headers in `e_phoff`) at and typedef them to `Proghdr` at address `0x10000` (above BIO). Then in each header (through a for loop iteration) it reads the segment by passing the physical address and offset and number of bytes to be counted each stored in `p_pa, p_offset, p_memsz` respectively.



04. Exercise 04

Downloading and running the file `pointers.c` gives the following output:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS D:\Ashoka University\2023 Spring\Operating Systems\Assignments\Lab-1> gcc -o pointers .\pointers.c
PS D:\Ashoka University\2023 Spring\Operating Systems\Assignments\Lab-1> .\pointers
1: a = 000000FF24FFFBB0, b = 0000024F02A93C60, c = 0000000000000000
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 000000FF24FFFBB0, b = 000000FF24FFFBB4, c = 000000FF24FFFBB1
PS D:\Ashoka University\2023 Spring\Operating Systems\Assignments\Lab-1>
```

05. Exercise 05

After running the command `objdump -h obj/kern/kernel` we get:

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000019e1  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000006bc  f0101a00  00101a00  00002a00  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003739  f01020bc  001020bc  000030bc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001529  f01057f5  001057f5  000067f5  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00009300  f0107000  00107000  00008000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got          00000008  f0110300  00110300  00011300  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .got.plt      0000000c  f0110308  00110308  00011308  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .data.rel.local 00001000  f0111000  00111000  00012000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  8 .data.rel.ro.local 00000044  f0112000  00112000  00013000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .bss          00000661  f0112060  00112060  00013060  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 10 .comment      0000002b  00000000  00000000  000136c1  2**0
                  CONTENTS, READONLY
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$
```

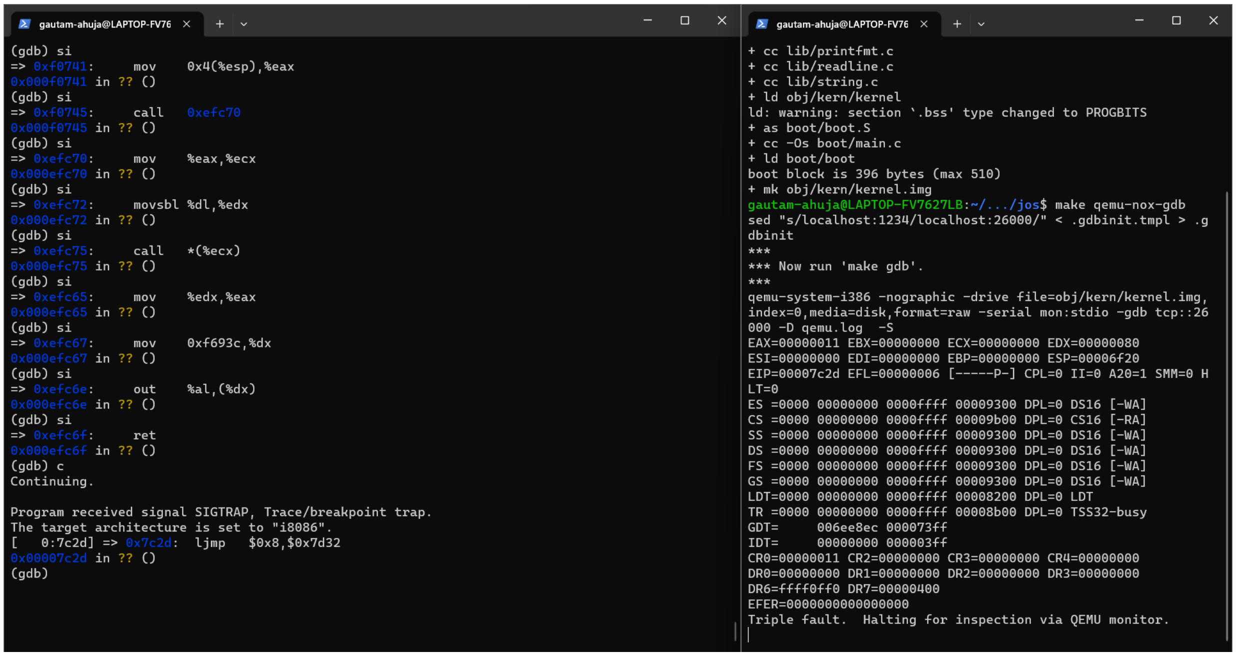
After running the command `objdump -h obj/boot/boot.out` we can see that the boot is linked to address `0x7c00` to load from there.

```
gautam-ahuja@LAPTOP-FV76
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ objdump -h obj/boot/boot.out

obj/boot/boot.out:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000018c  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     0000009c  00007d8c  00007d8c  00000200  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00000684  00000000  00000000  0000029c  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      0000043f  00000000  00000000  00000920  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .comment      0000002b  00000000  00000000  00000d5f  2**0
                  CONTENTS, READONLY
```

Now to see what (and where) happens when we first change the link address, in `boot/Makefrag`. Since the Boot is supposed to be at address `0x7c00`, if we change the address to `0x7d00` then:

```
jos > boot > Makefrag

$(OBJDIR)/boot/boot: $(BOOT_OBJS)
	@echo + ld boot/boot
	$(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7D00 -o $@.out $^
	$(V)$(OBJDUMP) -S $@.out >$@.asm
	$(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
	$(V)perl boot/sign.pl $(OBJDIR)/boot/boot
	github-classroom[bot], 2 weeks ago • Initial commit …
```

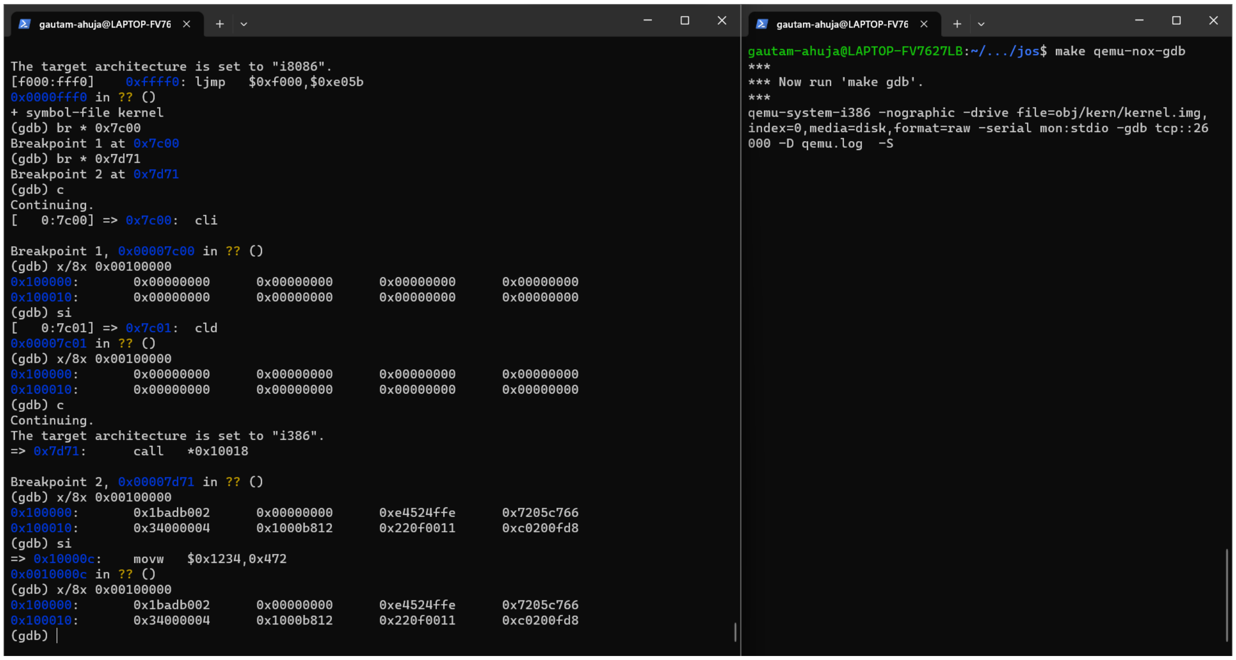
As we see in the 2nd screenshot, we can guess that the first instruction to fail be inside `boot.out` at `0x7c00` address, as it was supposed to be loaded from there but the new address is at `0x7d00`. We can see that our boot failed:



Since BIOS still load the bootloader to `0x7c00`, the first few instruction still work. However, the instruction `ljmp $0x8, $0x7d32` would "break" the kernel as there is no link to jump to.

06. Exercise 06

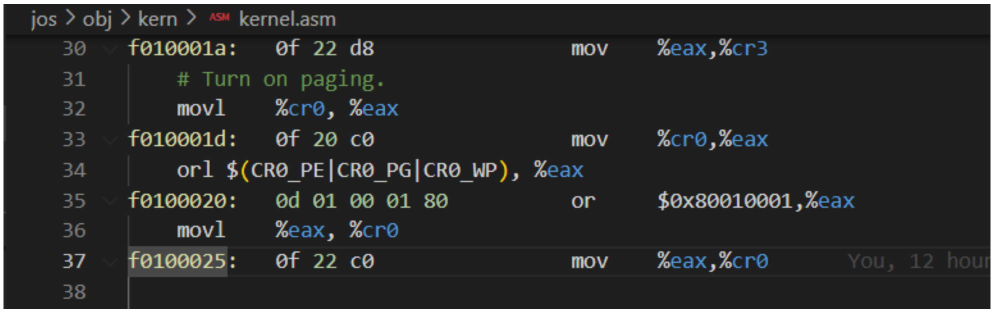
As done in question 3, the BIOS enters the bootloader at `0x7c00` and the bootloader enters the kernel at `0x7d71` as seen in `boot.asm` file.
Therefore we set two breakpoints, one at each address and then check the values in memory. We can also see that right after the last instruction in bootloader, the kernel's first instruction is at `0x10000c` and memory values are the same.



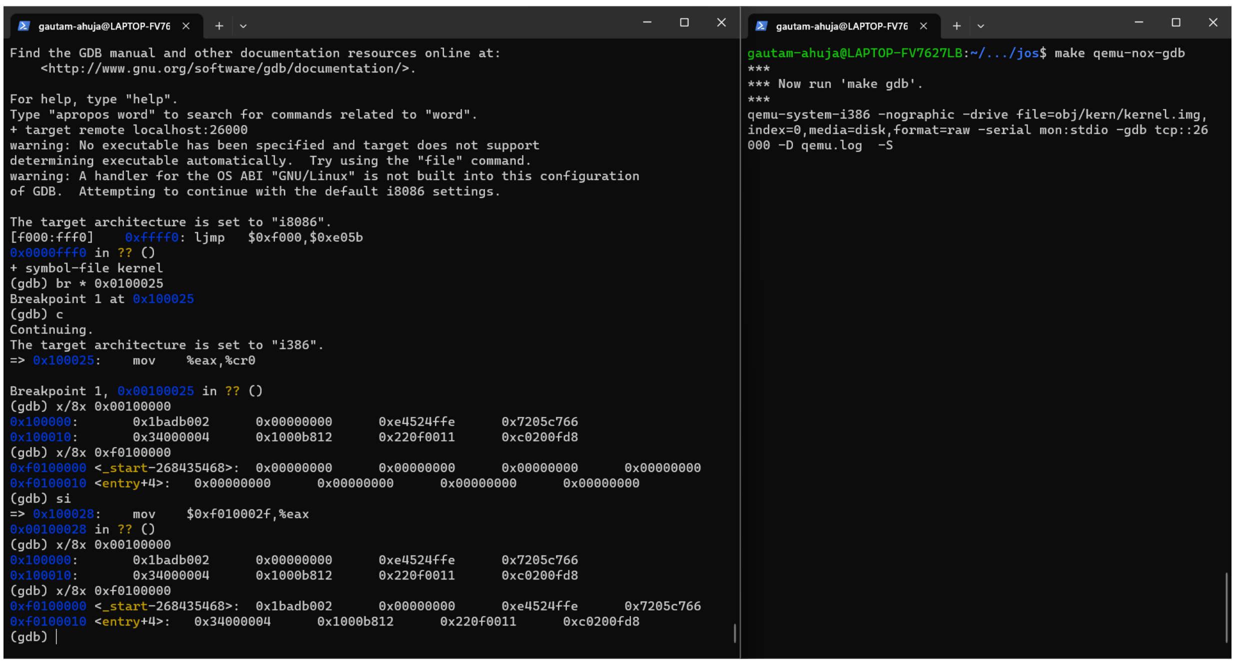
At the first breakpoint, the bootloader is loaded into the Low Memory region of the stack, hence the higher address `0x10000c` which lies above BIOS region is empty. Also the bootloader is in `16-bit` architecture so it could not access the higher addresses. Since there are no instructions acting on the memory, it is empty.
Later when the bootloader loads the kernel, it is copies into the Extended Memory region. The instruction which follow before, is the reason why the memory stack at `0x10000c` is non-empty and has some values - the .text region for kernel as seen in question 5 `objdump` command. The kernel is also in `32-bit` architecture, hence it can access, read and write the memory at higher addresses.

07. Exercise 07

As per question we first load into the kernel and top at `movl %eax, %cr0`. This isntruction is located at `0xf0100025` inside `kernel.asm`.

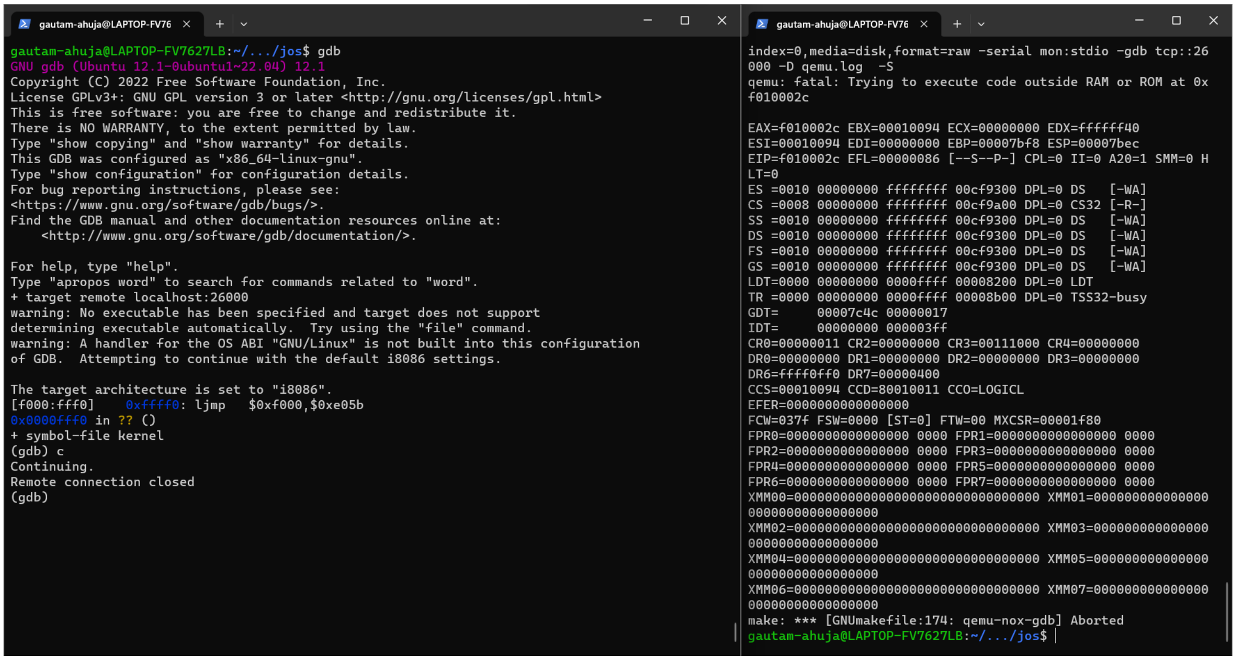


Since we know that this is mapped to `0x0100025`, we set a breakpoint at the address and then see the results.



Before the execution of instructions, the higher addresses are empty as they are not being accessed by the kernel as they exceeds physical memory's size of 4MB.
After the execution of the instruction, paging is available and virtual addresses can be set. We see the physical addresse `0x100000-0x100010` are exactly same as `0xf0100000-0xf0100010`. This is because they point point to the same location in memory where kernel is located.

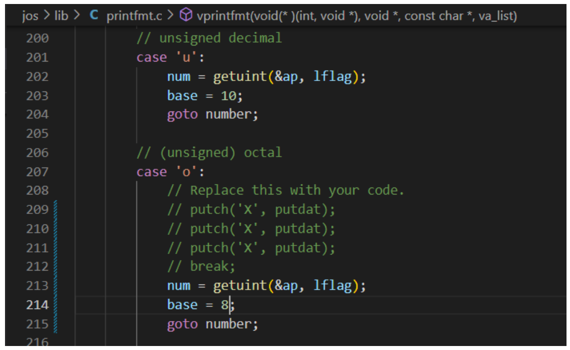
Now we comment out this line and rerun the kernel to see what happens.



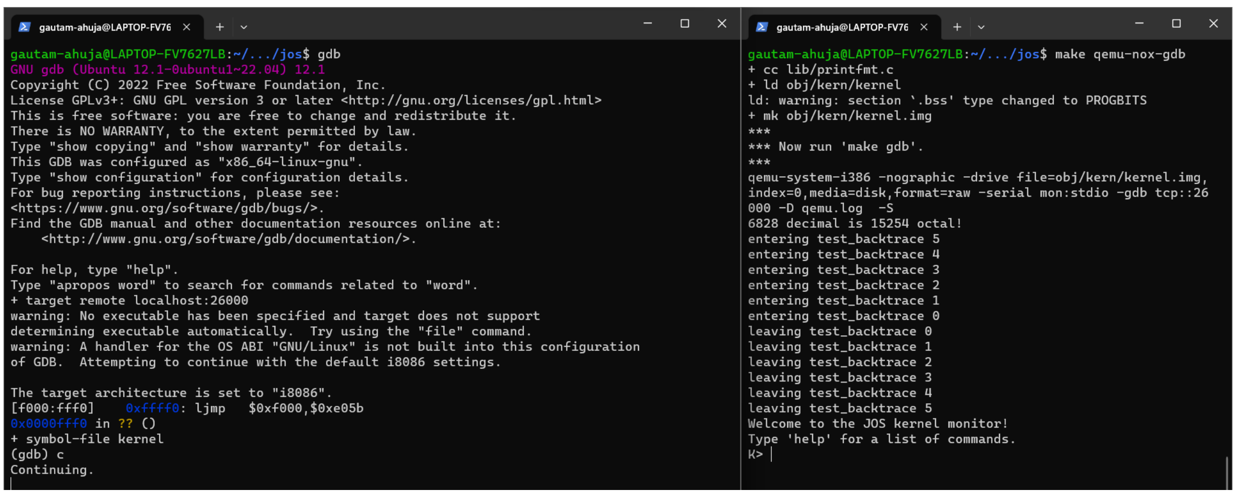
We can see that the kernel is not able to access the addresses as they are out of memory since paging is not present.
Since the paging is not available, the next instruction `mov $0xf010002f,%eax` and `jmp %eax` fails to execute as the address is inaccessible. This results in error `qemu: fatal: Trying to execute code outside RAM or ROM at 0xf010002c`.

08. Exercise 08

We refer to the file `lib/printfmt.c` to change and print octal numbers.
We change the `case 'o':` and add the code (by referring to above `case 'u':`)so it prints octal numbers.



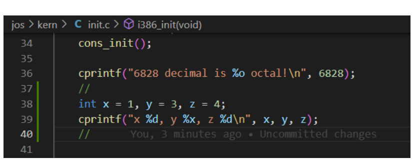
We can see that we have added it correctly as on loading of kernel we see `6828 decimal is 15254 octal!` instead of `6828 decimal is XXX octal!` on kernel.

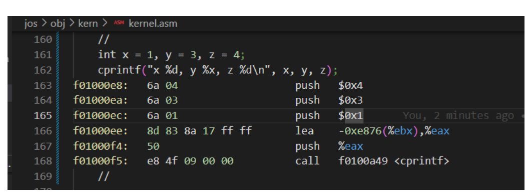


1. The `printf.c` and `console.c` are related through the `cputchar()` function. The `putch()` function in `printf.c` calls the `cputchar()` function which is used to put a single character on the screen.

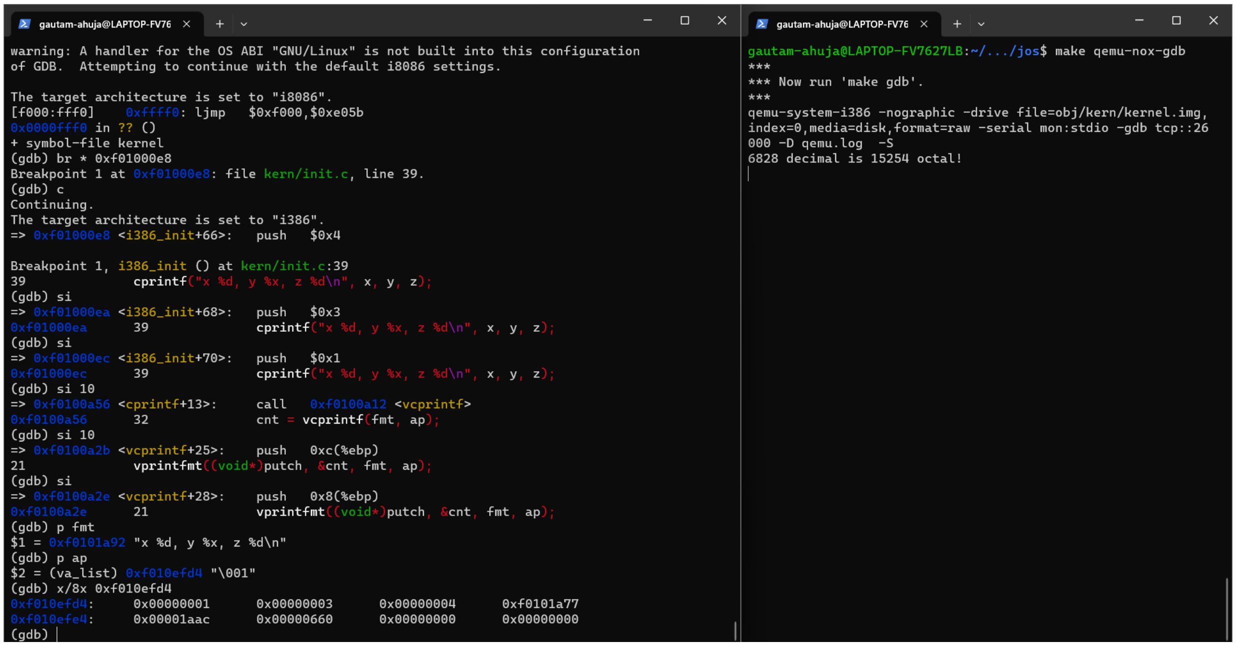
2. The code is used to scroll down the screen.
   `crt_pos` is the position of cursor on the screen. It the position of cursor is greater then screen size then a scroll is required. The `memmove()` function (located in `lib/string.c` moves the entire contents of the screen one row up. The next `for` loop fills the last row (`crt_buf` is the buffer row) with spaces. And the cursor is moved one position up from the end.

3. We insert the given code in the `init.c` file and check the corresponding address inside the new `kernel.asm` file.





At the breakpoint for `0xf01000e8` in GDb, we see:

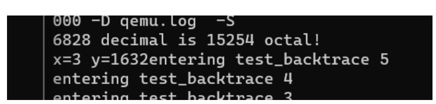


- We can see that `fmt` points to the address of string `x %d, y %x, z %d` and `ap` points to the arguments values of x, y, and z.

- The order of sequence is as follows: first, `vcprintf(fmt, ap)` is executed as seen which then points both the arguments for `printf`. Then, `va_arg(ap, int)` is called and finally `cons_putc(c)`

4. The output of the specific code will be: `He110 World`.
   The `57616` is input as hexadecimal (`the %x flag`). Hence it is converted into its hex form which is `0x110`. The second number `0x00646c72` is passed as a string. Since `x86` is little-endian, the value is stored as `| 00 | 64 | 6c | 72 |` where left is high address and right is low address. The corresponding ASCII character values are `d|l|r`. Therefore it outputs the string `"rld"` which is concatenated to `"Wo"`.
   If `x86` was a big-endian, the second argument needed to be : `0x726c6400`. The `57616` does not need to change because it is fetched as an integer and converted to hex and printed directly. It is not using ASCII conversion.

5. The output on our computer is: `x=3 y=1632`

   The first input (`3`) is defined. However, the second input is not. When the `ap` goes to fetch value from 4 bytes down the stack (as seen in above question), it can be any random value since we have not defined it. Therefore we see a random value.

6. The `ctype()` doesn't need to be changed. The setting up of arguments on the stack is the work of compiler and in return it provides the addresses of all the arguments in the stack. `ctype()` takes the addresses corresponding to the arguments and then prints the argument. The `fmt` points to the address of string and the `ap` points to the corresponding argument to string. Hence, no change is necessary.

09. Exercise 09

The stack is loaded after the bootloader moves control to the kernel.
We check the entrypoint and file `entry.S` to locate the initialization of the stack.

```
jos > kern > entry.S
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
```

Here we see that stack is initialized by `movl $0x0,%ebp` and `movl $(bootstacktop),%esp`.
In the `kernel.asm`, we see that the stack top is initiated at `0xf010f000`

```
jos > obj > kern > kernel.asm
    movl    $0x0,%ebp           # nuke frame pointer
f010002f:   bd 00 00 00 00          mov    $0x0,%ebp

    # Set the stack pointer
    movl    $(bootstacktop),%esp
f0100034:   bc 00 f0 10 f0          mov    $0xf010f000,%esp
```

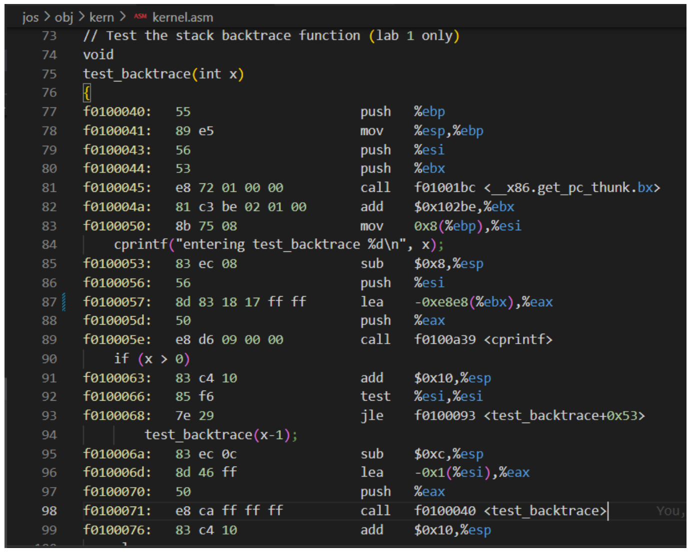
As discussed earlier, the top memory is reserved by `ELF`. And below it is the stack. We know the stack grows from top to bottom.
We will find the top of the start of the stack as:

```
gautam-ahuja@LAPTOP-FV76
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ objdump -D obj/kern/kernel | grep bootstack
f0107000 <bootstack>:
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ |
```

Here we see that stack starts at lower address of `0xf0107000`.
Therefore the range of the stack is `0xf0107000 - 0xf010f000` which is equivalent to 32 KB.

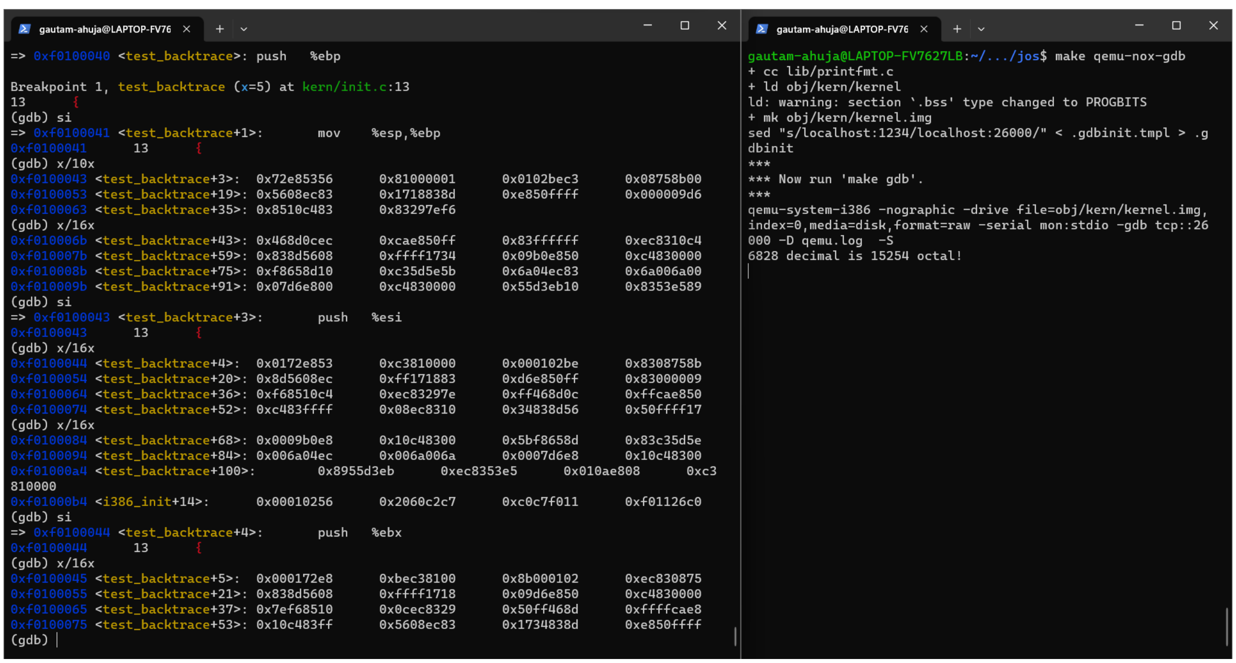
10. Exercise 10

We see the address for `test_backtrace` in the file `kernel.asm`.



```
jos > obj > kern > kernel.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:   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 18 17 ff ff       lea    -0xe8e8(%ebx),%eax
f010005d:   50                      push   %eax
f010005e:   e8 d6 09 00 00          call   f0100a39 <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
```

Here in each recursive call (`x>0`), we have 32 bytes of spaces being allocated in memory. 4 bytes (32-bit word) each to push the registers: `push %ebp`, `push %esi`, `push %ebx`, `push %esi`, `push %eax`, `push %eax` and a 8 byte allocation for stack `sub $0x8,%esp`.
We can see the traceback:



```
gautam-ahuja@LAPTOP-FV76
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=5) at kern/init.c:13
13      {
(gdb) si
=> 0xf0100041 <test_backtrace+1>:       mov    %esp,%ebp
0xf0100041      13      {
(gdb) x/10x
0xf0100043 <test_backtrace+3>:  0x72e85356      0x81000001      0x0102bec3      0x08758b00
0xf0100053 <test_backtrace+19>: 0x5608ec83      0x1718838d      0xe850ffff      0x000009d6
0xf0100063 <test_backtrace+35>: 0x8510c483      0x83297ef6
(gdb) x/16x
0xf010006b <test_backtrace+43>: 0x468d0cec      0xcae850ff      0x83ffffff      0xec8310c4
0xf010007b <test_backtrace+59>: 0x838d5608      0xffff1734      0x09b0e850      0xc4830000
0xf010008b <test_backtrace+75>: 0xf8658d10      0xc35d5e5b      0x6a04ec83      0x6a006a00
0xf010009b <test_backtrace+91>: 0x07d6e800      0xc4830000      0x55d3eb10      0x8353e589
(gdb) si
=> 0xf0100043 <test_backtrace+3>:       push   %esi
0xf0100043      13      {
(gdb) x/16x
0xf0100044 <test_backtrace+4>:  0x0172e853      0xc3810000      0x000102be      0x8308758b
0xf0100054 <test_backtrace+20>: 0x8d5608ec      0xff171883      0xd6e850ff      0x83000009
0xf0100064 <test_backtrace+36>: 0xf68510c4      0xec83297e      0xff468d0c      0xffcae850
0xf0100074 <test_backtrace+52>: 0xc483ffff      0x08ec8310      0x34838d56      0x50ffff17
(gdb) x/16x
0xf0100084 <test_backtrace+68>: 0x0009b0e8      0x10c48300      0x5bf8658d      0x83c35d5e
0xf0100094 <test_backtrace+84>: 0x006a04ec      0x006a006a      0x0007d6e8      0x10c48300
0xf01000a4 <test_backtrace+100>:        0x8955d3eb      0xec8353e5      0x010ae808      0xc3
810000
0xf01000b4 <i386_init+14>:      0x00010256      0x2060c2c7      0xc0c7f011      0xf01126c0
(gdb) si
=> 0xf0100044 <test_backtrace+4>:       push   %ebx
0xf0100044      13      {
(gdb) x/16x
0xf0100045 <test_backtrace+5>:  0x000172e8      0xbec38100      0x8b000102      0xec830875
0xf0100055 <test_backtrace+21>: 0x838d5608      0xffff1718      0x09d6e850      0xc4830000
0xf0100065 <test_backtrace+37>: 0x7ef68510      0x0cec8329      0x50ff468d      0xffffcae8
0xf0100075 <test_backtrace+53>: 0x10c483ff      0x5608ec83      0x1734838d      0xe850ffff
(gdb)
```

```
gautam-ahuja@LAPTOP-FV76
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ make qemu-nox-gdb
+ cc lib/printfmt.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .g
dbinit
***
*** 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::26
000 -D qemu.log  -S
6828 decimal is 15254 octal!
```

11. Exercise 11

We saw in the previous question, when we push values into the stack, `%ebp` (stack base pointer) is at the top followed by `%eip` which is the base pointer of the previous stack frame (return address). The rest are the arguments or local variables stored in form of array.
Therefore we may edit the code as:

```c
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    // Get the current base stack pointer (ebp)
    uint32_t ebp = read_ebp();
    // In a stack first valye is the return address, second value is the base pointer of the previous stack frame (eip). The rest are the arguments or local variables stored in form of array
    // Now we have to iterate over the stack frames and print the values of the stack frame.
    cprintf("Stack backtrace:");
    while(ebp != 0){
        cprintf("\n");
        // %08x is used to print the value in hexadecimal format with 8 digits.
        cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x", ebp, *((uint32_t*)ebp + 1), *((uint32_t*)ebp + 2), *((uint32_t*)ebp + 3), *((uint32_t*)ebp + 4), *((uint32_t*)ebp + 5), *((uint32_t*)ebp + 6));
        // Update the base pointer to the previous stack frame
        ebp = *((uint32_t*)ebp);
    }
    cprintf("\n");
    return 0;
}
```

The output of the code is:

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ make qemu-nox-gdb
+ cc kern/monitor.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
***
*** Now run 'make gdb'.
***
qemu-system-i386 -nographic -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -s
erial mon:stdio -gdb tcp::26000 -D qemu.log  -S
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
Stack backtrace:
ebp f010ef18 eip f01000a1 args 00000000 00000000 00000000 f010004a f0110308
ebp f010ef38 eip f0100076 args 00000000 00000001 f010ef78 f010004a f0110308
ebp f010ef58 eip f0100076 args 00000001 00000002 f010ef98 f010004a f0110308
ebp f010ef78 eip f0100076 args 00000002 00000003 f010efb8 f010004a f0110308
ebp f010ef98 eip f0100076 args 00000003 00000004 00000000 f010004a f0110308
ebp f010efb8 eip f0100076 args 00000004 00000005 00000000 f010004a f0110308
ebp f010efd8 eip f01000f4 args 00000005 00001aac 00000660 00000000 00000000
ebp f010eff8 eip f010003e args 00000003 00001003 00002003 00003003 00004003
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>
```

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ gdb
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/l
icenses/gpl.html>
This is free software: you are free to change and redistribu
te 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 no
t support
determining executable automatically.  Try using the "file"
command.
warning: A handler for the OS ABI "GNU/Linux" is not built i
nto this configuration
of GDB.  Attempting to continue with the default i8086 setti
ngs.

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

We started by taking `ebp = read_ebp()` as `uint32_t` and later in code we did type casting to `(uint32_t*)` because the value of `ebp` is a pointer to the stack frame which is of type `(uint32_t*)`.

12. Exercise 12

As per question, we first look into the `kdebug.c` file and complete the `stab_binsearch` for line number which is `N_SLINE` as mentioned in `stab.h`.

```
jos > kern > C kdebug.c > debuginfo_eip(uintptr_t, Eipdebuginfo *)
    // Already searched within that file's stabs for the function definition
    // Now, search within the function definition for the line number.
    // Here we use the N_SLINE stab type.
    stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
    if (lline <= rline)
        info->eip_line = stabs[lline].n_desc;
    else
        // Line number not found, return -1.
        return -1;
```

The `stab[lline]` points to the line number in string table. the next `.n_desc` is a variable in `stabs` structure which contains the address of the current line number. If address is found, we store it in `eip_line` variable of `info` structure.

In `kernel.asm`, we see that `__STAB_BEGIN__` and `__STAB_END__` (`__STAB_*`) both refer to self `0xF0100000` address. This is where the kernel is loaded

```
jos > 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 */
    }
```

When doing a `objdump -h obj/kern/kernel`, we see that kernel has a `.stab` and `.stabstr` section.

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001b21  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000728  f0101b40  00101b40  00002b40  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         000038dd  f0102268  00102268  00003268  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001620  f0105b45  00105b45  00006b45  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00009300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got          00000008  f0111300  00111300  00012300  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .got.plt      0000000c  f0111308  00111308  00012308  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .data.rel.local 00001000  f0112000  00112000  00013000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  8 .data.rel.ro.local 00000060  f0113000  00113000  00014000  2**5
                  CONTENTS, ALLOC, LOAD, DATA
  9 .bss          00000661  f0113060  00113060  00014060  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 10 .comment      0000002b  00000000  00000000  000146c1  2**0
                  CONTENTS, READONLY
```

A run of `objdump -G obj/kern/kernel` command, we can read all the symbols which belongs to the `.stab` section of the code (as seen in `grep` command)

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ objdump -G obj/kern/kernel | grep stab
Contents of .stab section:
492    FUN    0      0      f0100b19 2551   stab_binsearch:f(0,1)=(0,1)
494    RSYM   0      0      00000000 2579   stabs:P(0,2)=*(0,3)=xsStab:
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ |
```

```
gautam-ahuja@LAPTOP-FV76
                     CONTENTS, READONLY
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ objdump -G obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Contents of .stab section:

Symnum n_type n_othr n_desc n_value  n_strx String

-1     HdrSym 0      1212   0000161f 1
0      SO     0      0      f0100000 1      {standard input}
1      SOL    0      0      f010000c 18     kern/entry.S
2      SLINE  0      44     f010000c 0
3      SLINE  0      57     f0100015 0
4      SLINE  0      58     f010001a 0
5      SLINE  0      60     f010001d 0
6      SLINE  0      61     f0100020 0
7      SLINE  0      62     f0100025 0
8      SLINE  0      67     f0100028 0
9      SLINE  0      68     f010002d 0
10     SLINE  0      74     f010002f 0
11     SLINE  0      77     f0100034 0
12     SLINE  0      80     f0100039 0
13     SLINE  0      83     f010003e 0
14     SO     0      2      f0100040 31     kern/entrypgdir.c
15     OPT    0      0      00000000 49     gcc2_compiled.
16     GSYM   0      0      00000000 64     entry_pgdir:G(0,1)=ar(0,2)=r(0,2);0;4294967295;;
0;1023;(0,3)=(0,4)=(0,5)=r(0,5);0;4294967295;
17     LSYM   0      0      00000000 158    pde_t:t(0,3)
```

Hence the __STAB_* comes to read symbols from kernel for debugging using `debuginfo_eip()`.

Now, we edit out `mon_backtrace()` function to display, for each `eip`, the function name, source file name, and line number corresponding to that `eip`.

```c
jos > kern > C monitor.c > mon_backtrace(int, char **, Trapframe *)
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    // Q11 Edit: Get the current base stack pointer (ebp)
    uint32_t ebp = read_ebp();
    // Q12 Edit: Create a structure to store the debug information
    struct Eipdebuginfo info;

    // In a stack first valye is the return address, second value is the base pointer of the previous stack frame (eip). The rest are the arguments or local variables stored in form of array. Now we have to iterate over the stack frames and print the values of the stack frame.      You, 1 second ago • Uncommitted changes
    cprintf("Stack backtrace:");
    while(ebp != 0){
        cprintf("\n");
        // %08x is used to print the value in hexadecimal format with 8 digits.
        // Q11 Edit: Print Stack Information
        cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x", ebp, *((uint32_t*)ebp + 1), *((uint32_t*)ebp + 2), *((uint32_t*)ebp + 3), *((uint32_t*)ebp + 4), *((uint32_t*)ebp + 5), *((uint32_t*)ebp + 6));

        // Q12 Edit: Print Debug Information, send the eip value to the function as it is the address of the instruction that caused the call.
        debuginfo_eip(*((uint32_t*)ebp + 1), &info);
        // Print the debug information
        cprintf(" %s:%d: %.*s+%d\n", info.eip_file, info.eip_line, info.eip_fn_namelen, info.eip_fn_name, *((uint32_t*)ebp + 1) - info.eip_fn_addr);

        // Update the base pointer to the previous stack frame
        ebp = *((uint32_t*)ebp);
    }
    cprintf("\n");
    return 0;
}
```

We collect the debug information of the previous stack base pointer using `debuginfo_eip()` and then print the relevant information. To add a new command, we add a new entry into the `Command` structure.

jos > kern > monitor.c > ...

You, 11 seconds ago | 2 authors (github-classroom[bot] and others)

```c
static struct Command commands[] = {
	{ "help", "Display this list of commands", mon_help },
	{ "kerninfo", "Display information about the kernel", mon_kerninfo },
	// Adding a new command for backtrack
	{ "backtrace", "Display the stack backtrace", mon_backtrace },
};
```

After all edits, the final output is:

gautam-ahuja@LAPTOP-FV76

```
ebp f010ff58 eip f0100076 args 00000001 00000002 f010ff98 f010004a f0111308
             kern/init.c:16: test_backtrace+54

ebp f010ff78 eip f0100076 args 00000002 00000003 f010ffb8 f010004a f0111308
             kern/init.c:16: test_backtrace+54

ebp f010ff98 eip f0100076 args 00000003 00000004 00000000 f010004a f0111308
             kern/init.c:16: test_backtrace+54

ebp f010ffb8 eip f0100076 args 00000004 00000005 00000000 f010004a f0111308
             kern/init.c:16: test_backtrace+54

ebp f010ffd8 eip f01000f4 args 00000005 00001aac 00000660 00000000 00000000
             kern/init.c:45: i386_init+78

ebp f010fff8 eip f010003e args 00000003 00001003 00002003 00003003 00004003
             kern/entry.S:83: <unknown>+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> backtrace
Stack backtrace:
ebp f010ff58 eip f0100a99 args 00000001 f010ff80 00000000 f0100afd f0100aac
             kern/monitor.c:147: monitor+333

ebp f010ffd8 eip f0100101 args 00000000 00001aac 00000660 00000000 00000000
             kern/init.c:49: i386_init+91

ebp f010fff8 eip f010003e args 00000003 00001003 00002003 00003003 00004003
             kern/entry.S:83: <unknown>+0

K>
```

gautam-ahuja@LAPTOP-FV76

```
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ gdb
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/l
icenses/gpl.html>
This is free software: you are free to change and redistribu
te 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 no
t support
determining executable automatically.  Try using the "file"
command.
warning: A handler for the OS ABI "GNU/Linux" is not built i
nto this configuration
of GDB.  Attempting to continue with the default i8086 setti
ngs.

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