

CS1217 - Spring 2023 - Lab 1

Gautam Ahuja, Nistha Singh

01. Exercise 01

Reading Exercise

02. Exercise 02

On doing a single step of kernel:

```
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::26000 -D qemu.log -S

The target architecture is set to "i386".
[f000:ffff] 0xffff0: jmp $0x000,0x05b
0x0000ffff in ?? ()
+ symbol-file kernel
(gdb) si
[f000:e05b] 0xfe05b: cmpl $0x0,%cs:0x6ac8
0x0000e05b in ?? ()
(gdb) si
[f000:e062] 0xfe062: jne 0xfd2e1
0x0000e062 in ?? ()
(gdb) si
[f000:e066] 0xfe066: xor %dx,%dx
0x0000e066 in ?? ()
(gdb) si
[f000:e068] 0xfe068: mov %dx,%ss
0x0000e068 in ?? ()
(gdb) si
[f000:e06a] 0xfe06a: mov $0x7000,%esp
0x0000e06a in ?? ()
(gdb) si
[f000:e070] 0xfe070: mov $0xf34c2,%edx
0x0000e070 in ?? ()
(gdb) si
[f000:e076] 0xfe076: jmp 0xfd15c
0x0000e076 in ?? ()
(gdb) si
[f000:d15c] 0xfd15c: mov %eax,%ecx
0x0000d15c in ?? ()
(gdb) si
[f000:d15f] 0xfd15f: cli
0x0000d15f in ?? ()
(gdb) si
[f000:d160] 0xfd160: cld
0x0000d160 in ?? ()
(gdb) si
[f000:d161] 0xfd161: mov $0x8f,%eax
0x0000d161 in ?? ()
(gdb) si
[f000:d167] 0xfd167: out %al,$0x70
0x0000d167 in ?? ()
```

- 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.

```
gautam-ahuja@LAPTOP-FV76  +  -
***
*** 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
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> QEMU: Terminated
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::26000 -D qemu.log -S
```

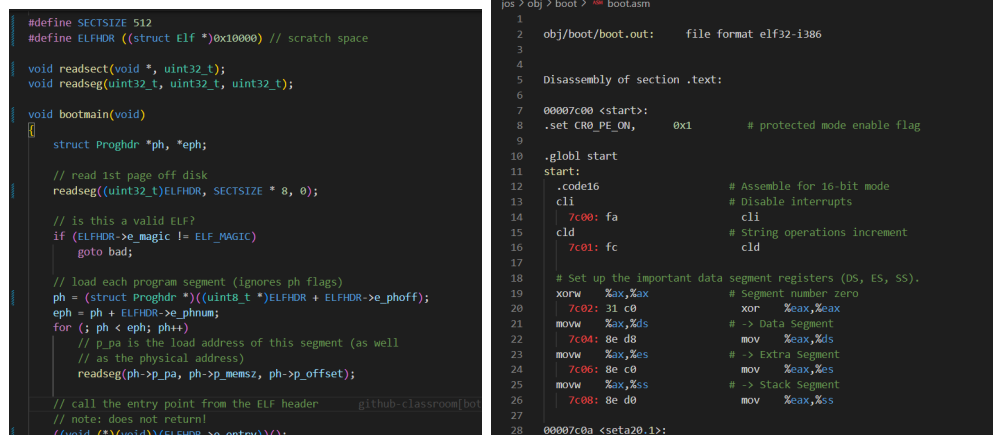
- Few steps further it sets up ports 0x70 and 0x71 in the %al register. The %al register is used to set the I/O ports and operation.

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.



```
#define SECTSIZE 512
#define ELFHDR ((struct Elf *)0x10000) // scratch space

void readsect(void *, uint32_t);
void readseg(uint32_t, uint32_t, uint32_t);

void bootmain(void)
{
    struct Proghdr *ph, *eph;

    // read 1st page off disk
    readseg((uint32_t)ELFHDR, SECTSIZE * 8, 0);

    // is this a valid ELF?
    if (ELFHDR->e_magic != ELF_MAGIC)
        goto bad;

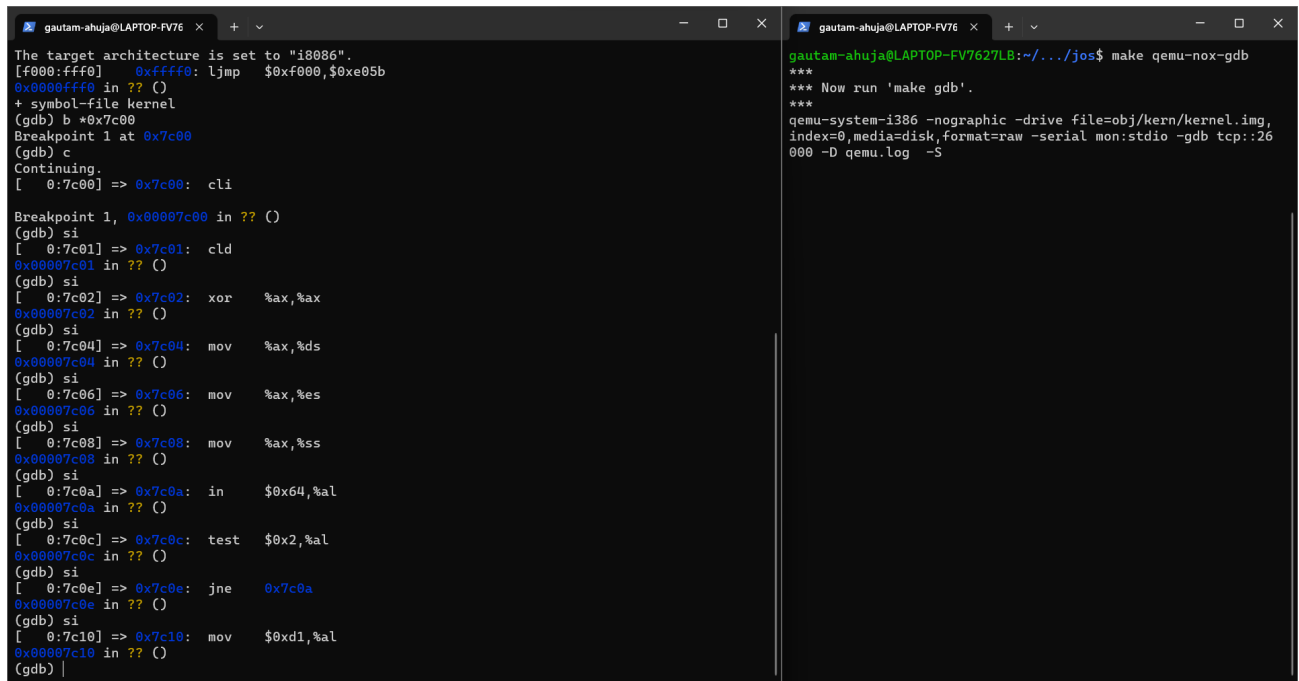
    // load each program segment (ignores ph flags)
    ph = (struct Proghdr *)((uint8_t *)ELFHDR + ELFHDR->e_phoff);
    eph = ph + ELFHDR->e_phnum;
    for (; ph < eph; ph++)
    {
        // p_pa is the load address of this segment (as well
        // as the physical address)
        readseg(ph->p_pa,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

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

// bad: does not return!
void bad(void)
{
    while (1)
        ;
}
```

```
1
2 obj/boot/boot.out: file format elf32-i386
3
4
5 Disassembly of section .text:
6
7 00007c00 <start>:
8 .set CR0_PE_ON, 0x1 # protected mode enable flag
9
10 .globl start
11 start:
12 .code16 # Assemble for 16-bit mode
13 cli # Disable interrupts
14 7c00: fa cld # String operations increment
15 7c01: fc cld
16
17 # Set up the important data segment registers (DS, ES, SS).
18 xorw %ax,%ax # Segment number zero
19 7c02: 31 c0 xor %eax,%eax
20 movw %ax,%ds # -> Data Segment
21 7c04: 8e d8 mov %eax,%ds
22 movw %ax,%es # -> Extra Segment
23 7c06: 8e c0 mov %eax,%es
24 movw %ax,%ss # -> Stack Segment
25 7c08: 8e d0 mov %eax,%ss
26
27 00007c0a <seta20,1>:
```

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



```
The target architecture is set to "i8086".
[f000:ffff] 0xfffff0: ljmp $0xf000,$0xe05b
0x0000ffff in ?? ()
+ symbol-file kernel
(gdb) b *0x7c00
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[ 0:7c00] => 0x7c00: cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) si
[ 0:7c01] => 0x7c01: cld
0x00007c01 in ?? ()
(gdb) si
[ 0:7c02] => 0x7c02: xor %ax,%ax
0x00007c02 in ?? ()
(gdb) si
[ 0:7c04] => 0x7c04: mov %ax,%ds
0x00007c04 in ?? ()
(gdb) si
[ 0:7c06] => 0x7c06: mov %ax,%es
0x00007c06 in ?? ()
(gdb) si
[ 0:7c08] => 0x7c08: mov %ax,%ss
0x00007c08 in ?? ()
(gdb) si
[ 0:7c0a] => 0x7c0a: in $0x64,%al
0x00007c0a in ?? ()
(gdb) si
[ 0:7c0c] => 0x7c0c: test $0x2,%al
0x00007c0c in ?? ()
(gdb) si
[ 0:7c0e] => 0x7c0e: jne 0x7c0a
0x00007c0e in ?? ()
(gdb) si
[ 0:7c10] => 0x7c10: mov $0xd1,%al
0x00007c10 in ?? ()
(gdb) |
```

```
gautam-ahuja@LAPTOP-FV76:~/.../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
```

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(0x1F6, 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 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 mov     $0x1f2,%edx
        7c8c: ee         out     %al,(%dx)
        7c8d: ba f3 01 00 mov     $0x1f3,%edx
        7c92: 89 c8       mov     %ecx,%eax
        7c94: ee         out     %al,(%dx)
    }
}

```

Below is the image which shows beginning and end 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    -0xc(%ebx)
    7d61: ff 73 f4    push    -0xc(%ebx)
    7d64: ff 73 ec    push    -0x14(%ebx)
    7d67: e8 6e 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 kernel, as seen on the right screen.

```

(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:~/.../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 gdt desc`. This sets 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:

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# 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
```

```
# Switch from real to protected mode, using a bootstrap GDT
# and segment translation that makes virtual addresses
# identical to their physical addresses, so that the
# effective memory map does not change during the switch.
lgdt gdt desc
7c1e: 0f 01 16 lgdtl (%esi)
7c21: 64 7c 0f fs jl 7c33 <protcseg+0x1>
movl %cr0, %eax
7c24: 20 c0 and %al, %al
orl $CR0_PE_ON, %eax
7c26: 66 83 c8 01 or $0x1, %ax
movl %eax, %cr0
7c2a: 0f 22 c0 mov %eax, %cr0

# Jump to next instruction, but in 32-bit code segment.
# Switches processor into 32-bit mode.
```

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`.

```
josh> objdump -d kernel.asm
11: .globl entry
12: entry:
13: movw $0x1234,0x472 # warm boot
14: f0100000: 02 b6 ad 1b 00 00 add %ebx,%eax
15: f0100006: 00 00 add %al,%eax
16: f0100008: fe 4f 52 decb 0x52(%edi)
17: f010000b: e4 .byte 0xe4
18:
19: f010000c: entry:
20: f010000c: 66 c7 05 72 04 00 movw $0x1234,0x472
21: f0100011: 34 12
```

```
josh> km -> entry.S
41:
42: .globl entry
43: entry:
44: movw $0x1234,0x472 # warm boot
45:
46: # We haven't set up virtual memory yet, so we're running from
47: # the physical address the boot loader loaded the kernel at: 1MB
48: # (plus a few bytes). However, the C code is linked to run at
49: # KERNELBASE+1MB. Hence, we set up a trivial page directory that
50: # translates virtual addresses (KERNELBASE, KERNELBASE+4MB) to
51: # physical addresses (0, 4MB). This area region will be
52: # sufficient until we set up our real page table in main_init
53: # in lab 2.
```

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`

```
josh> boot -C main.c -o bootmain(void)
48:
49: // load each program segment (ignores ph flags)
50: ph = (struct Proghdr *)((uint8_t *)ELFHDR + ELFHDR->e_phoff);
51: eph = ph + ELFHDR->e_phnum;
52: for (; ph < eph; ph++)
53: // p_pa is the load address of this segment (as well
54: // as the physical address)
55: readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
56:
```

```
josh> inc -C elf.h -o SectionDef
```

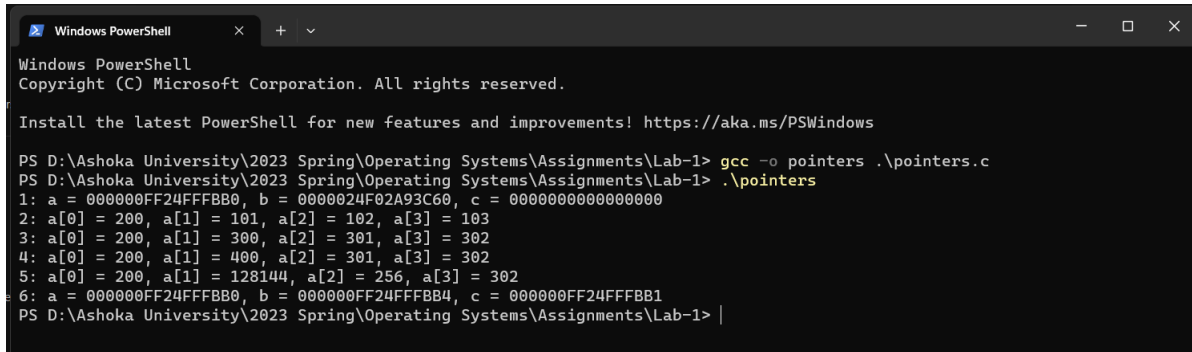
```
24: struct Proghdr {
25:     uint32_t p_type;
26:     uint32_t p_offset;
27:     uint32_t p_va;
28:     uint32_t p_pa;
29:     uint32_t p_filesz;
30:     uint32_t p_memsz;
31:     uint32_t p_flags;
32:     uint32_t p_align;
33: };
34:
```

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.

```
void readseg(uint32_t pa, uint32_t count, uint32_t offset)
Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
Might copy more than asked
readseg(ph->p_pa, ph->p_memsz, ph->p_offset);
```

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 = 000000FF24FFFB80, 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 = 000000FF24FFFB80, b = 000000FF24FFFB84, c = 000000FF24FFFB81
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-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
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ |
```

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 > M Makefrag
25
26 $(OBJDIR)/boot/boot: $(BOOT_OBJS)
27 @echo + ld boot/boot
28 $(V)$(LD) $(LDFLAGS) -N -e start -Ttext 0x7D00 -o $@.out $^
29 $(V)$(OBJDUMP) -S $@.out >$@.asm
30 $(V)$(OBJCOPY) -S -O binary -j .text $@.out $@
31 $(V)perl boot/sign.pl $(OBJDIR)/boot/boot
32 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:

```

(gdb) si
=> 0xf0741:    mov     0x4(%esp),%eax
0x000f0741 in ?? ()
(gdb) si
=> 0xf0745:    call    0xfec70
0x000f0745 in ?? ()
(gdb) si
=> 0xfec70:    mov     %eax,%ecx
0x000fec70 in ?? ()
(gdb) si
=> 0xfec72:    movsbl %dl,%edx
0x000fec72 in ?? ()
(gdb) si
=> 0xfec75:    call    *(%ecx)
0x000fec75 in ?? ()
(gdb) si
=> 0xfec65:    mov     %edx,%eax
0x000fec65 in ?? ()
(gdb) si
=> 0xfec67:    mov     0xf693c,%dx
0x000fec67 in ?? ()
(gdb) si
=> 0xfec6e:    out     %al,(%dx)
0x000fec6e in ?? ()
(gdb) si
=> 0xfec6f:    ret
0x000fec6f in ?? ()
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
The target architecture is set to "i8086".
[ 0:7c2d] => 0x7c2d:  jmp     $0x8,$0x7d32
0x00007c2d in ?? ()
(gdb)

+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
ld: warning: section '.bss' type changed to PROGBITS
+ as boot/boot.S
+ cc -O5 boot/main.c
+ ld boot/boot
boot block is 396 bytes (max 510)
+ mk obj/kern/kernel.img
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$ make qemu-nox-gdb
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
EAX=00000011 EBX=00000000 ECX=00000000 EDX=00000000
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00006f20
EIP=00007c2d EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 H
LT=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=ffff0fff DR7=00000400
EFER=0000000000000000
Triple fault. Halting for inspection via QEMU monitor.

```

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.

```

gautam-ahuja@LAPTOP-FV76: ~$ gdb -i boot.asm
The target architecture is set to "i386".
[f000:fff0] 0xffff0: jmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) br * 0x7c00
Breakpoint 1 at 0x7c00
(gdb) br * 0x7d71
Breakpoint 2 at 0x7d71
(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
(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/8x 0x00100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x1000b812 0x220f0011 0xc0200fd8
(gdb)
  
```

```

gautam-ahuja@LAPTOP-FV76: ~$ 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
  
```

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 copied 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 instruction is located at `0xf0100025` inside `kernel.asm`.

```
josh> obj> kern> ASM kernel.asm
30 f010001a: 0f 22 d8          mov    %eax,%cr3
31      # Turn on paging.
32      movl    %cr0, %eax
33 f010001d: 0f 20 c0          mov    %cr0,%eax
34      orl    $(CR0_PE|CR0_PG|CR0_WP), %eax
35 f0100020: 0d 01 00 01 80    or     $0x80010001,%eax
36      movl    %eax, %cr0
37 f0100025: 0f 22 c0          mov    %eax,%cr0
38
```

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

```
gautam-ahuja@LAPTOP-FV76: ~$ 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 i386 settings.

The target architecture is set to "i386".
[f000:ffff] 0xfffff0: jmp    $0xf000,$0xe05b
0x0000ffff in ?? ()
+ symbol-file kernel
(gdb) br * 0x0100025
Breakpoint 1 at 0x100025
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0x100025: mov    %eax,%cr0

Breakpoint 1, 0x0100025 in ?? ()
(gdb) x/8x 0x0100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x1000b812 0x220f0011 0xc0200fd8
(gdb) x/8x 0xf0100000
0xf0100000 <_start-268435468>: 0x00000000 0x00000000 0x00000000 0x00000000
0xf0100010 <entry+4>: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) si
=> 0x100028: mov    $0xf010002f,%eax
0x0100028 in ?? ()
(gdb) x/8x 0x0100000
0x100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0x100010: 0x34000004 0x1000b812 0x220f0011 0xc0200fd8
(gdb) x/8x 0xf0100000
0xf0100000 <_start-268435468>: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
0xf0100010 <entry+4>: 0x34000004 0x1000b812 0x220f0011 0xc0200fd8
(gdb) |

gautam-ahuja@LAPTOP-FV767LB: ~/.../josh$ 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
```

Before the execution of instructions, the higher addresses are empty as they are not being accessed by the kernel as they exceed 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 addresses `0x100000-0x100010` are exactly the same as `0xf0100000-0xf0100010`. This is because they point to the same location in memory where the kernel is located.

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

```

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/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 i386 settings.

The target architecture is set to "i386".
[f000:fff0] 0xfffff0: jmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) e
Continuing.
Remote connection closed
(gdb)

index=0 media=disk,format=raw -serial mon:stdio -gdb tcp::26
000 -D qemu.log -S
qemu: fatal: Trying to execute code outside RAM or ROM at 0x
f010002c

EAX=f010002c EBX=00010094 ECX=00000000 EDX=ffffff40
ESI=00010094 EDI=00000000 EBP=00007bf8 ESP=00007bec
EIP=f010002c EFL=00000086 [--S--P--] CPL=0 II=0 A20=1 SMM=0 H
LT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
FS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
GS =0010 00000000 ffffffff 00cf9300 DPL=0 DS [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TS32-busy
GDT= 00007c4c 00000017
IDT= 00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00111000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
CCS=00010094 CCD=80010011 CCO=LOGICL
EFER=0000000000000000
FCW=037f FSW=0000 [ST=0] FTW=00 MXCSR=00001f80
FPR0=0000000000000000 0000 FPR1=0000000000000000 0000
FPR2=0000000000000000 0000 FPR3=0000000000000000 0000
FPR4=0000000000000000 0000 FPR5=0000000000000000 0000
FPR6=0000000000000000 0000 FPR7=0000000000000000 0000
XMM00=00000000000000000000000000000000 XMM01=0000000000000000
0000000000000000
XMM02=00000000000000000000000000000000 XMM03=0000000000000000
0000000000000000
XMM04=00000000000000000000000000000000 XMM05=0000000000000000
0000000000000000
XMM06=00000000000000000000000000000000 XMM07=0000000000000000
0000000000000000
make: *** [GNUmakefile:174: qemu-nox-gdb] Aborted
gautam-ahuja@LAPTOP-FV7627LB:~/.../jos$

```

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`.

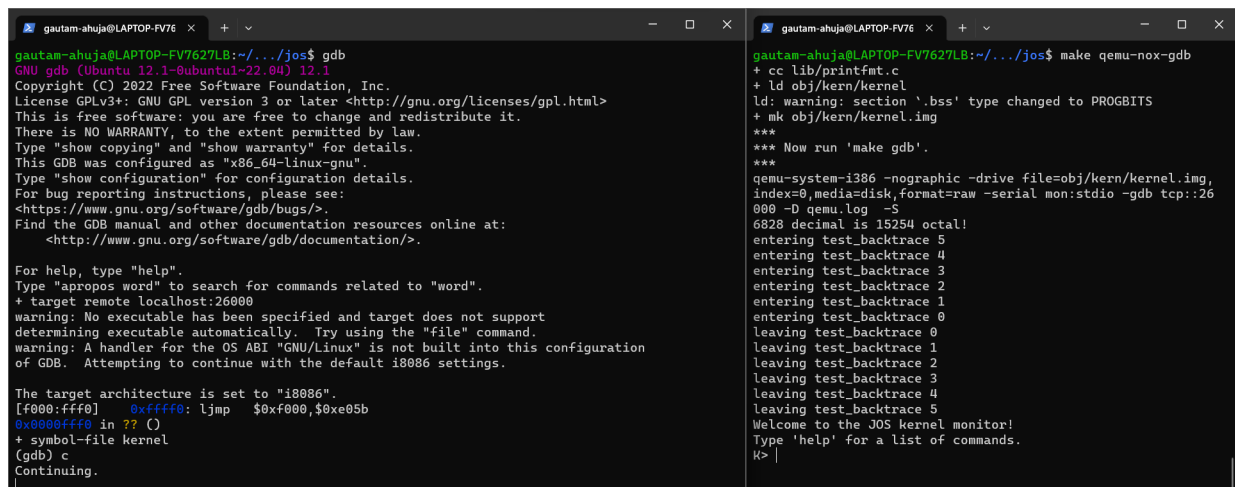
08. Exercise 08

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.

```
jos > lib > C printfmt.c > vprintfmt(void*)(int, void *, void *, const char *, va_list)
200 // unsigned decimal
201 case 'u':
202     num = getuint(&ap, lflag);
203     base = 10;
204     goto number;
205
206 // (unsigned) octal
207 case 'o':
208     // Replace this with your code.
209     // putchar('X', putdat);
210     // putchar('X', putdat);
211     // putchar('X', putdat);
212     // break;
213     num = getuint(&ap, lflag);
214     base = 8;
215     goto number;
216
```

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.



```
gautam-ahuja@LAPTOP-FV76: ~/.../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/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 i386 settings.

The target architecture is set to "i386".
[f000:fff0] 0xfffff0: jmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb) c
Continuing.

gautam-ahuja@LAPTOP-FV76: ~/.../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
***
*** 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!
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>
```

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. If the position of cursor is greater than 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.

```
jos > kern > C init.c > b386_init(void)
34 cons_init();
35
36 printf("6828 decimal is %o octal\n", 6828);
37 //
38 int x = 1, y = 3, z = 4;
39 printf("x %d, y %x, z %d\n", x, y, z);
40 // You, 1 minutes ago + uncommitted changes
41
```

```
jos > obj > kern > kernel.asm
160 //
161 int x = 1, y = 3, z = 4;
162 cprintf("x %d, y %x, z %d\n", x, y, z);
163 f01000e8: 6a 04          push $0x4
164 f01000ea: 6a 03          push $0x3
165 f01000ec: 6a 01          push $0x1
166 f01000ee: 8d 83 8a 17 ff lea -0xe876(%ebx),%eax
167 f01000f4: 50            push %eax
168 f01000f5: e8 4f 09 00 00 call f0100a49 <printf>
169 //
```

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

```

warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB. Attempting to continue with the default i386 settings.

The target architecture is set to "i386".
[fffff0] 0xfffff0: jmp $0xf000,$0xe05b
0x0000ffff in ?? ()
+ symbol-file kernel
(gdb) br * 0xf01000e8
Breakpoint 1 at 0xf01000e8: file kern/init.c, line 39.
(gdb) c
Continuing.
The target architecture is set to "i386".
=> 0xf01000e8 <i386_init+66>: push $0x4

Breakpoint 1, i386_init () at kern/init.c:39
39      cprintf("x %d, y %x, z %d\n", x, y, z);
(gdb) si
=> 0xf01000ea <i386_init+68>: push $0x3
0xf01000ea 39      cprintf("x %d, y %x, z %d\n", x, y, z);
(gdb) si
=> 0xf01000ec <i386_init+70>: push $0x1
0xf01000ec 39      cprintf("x %d, y %x, z %d\n", x, y, z);
(gdb) si 10
=> 0xf0100a56 <vcprintf+13>: call 0xf0100a12 <vcprintf>
0xf0100a56 32      cnt = vcprintf(fmt, ap);
(gdb) si 10
=> 0xf0100a2b <vcprintf+25>: push 0xc(%ebp)
21      vprintfmt((void*)putch, &cnt, fmt, ap);
(gdb) si
=> 0xf0100a2e <vcprintf+28>: push 0x8(%ebp)
0xf0100a2e 21      vprintfmt((void*)putch, &cnt, fmt, ap);
(gdb) p fmt
$1 = 0xf0101a92 "x %d, y %x, z %d\n"
(gdb) p ap
$2 = (va_list) 0xf010efd4 "\001"
(gdb) x/8x 0xf010efd4
0xf010efd4: 0x00000001 0x00000003 0x00000004 0xf0101a77
0xf010efe4: 0x00001aac 0x00000660 0x00000000 0x00000000
(gdb)

```

```

gautam-ahuja@LAPTOP-FV76: ~/.../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 15254 octal!

```

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

```

000 -D qemu.log -S
6828 decimal is 15254 octal!
x=3 y=1632entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3

```

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.

```
josh> kern > ASM entry.S
69 relocated:
70
71 # Clear the frame pointer register (EBP)
72 # so that once we get into debugging C code,
73 # stack backtraces will be terminated properly.
74 movl    $0x0,%ebp      # nuke frame pointer
75
76 # Set the stack pointer
77 movl    $(bootstacktop),%esp
78
79 # now to C code
80 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`

```
josh> obj > kern > ASM kernel.asm
53 movl    $0x0,%ebp      # nuke frame pointer
54 f010002f: bd 00 00 00 00      mov     $0x0,%ebp
55
56 # Set the stack pointer
57 movl    $(bootstacktop),%esp
58 f0100034: bc 00 f0 10 f0      mov     $0xf010f000,%esp
59
```

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  x  +  v
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 > asm kernel.asm
73 // Test the stack backtrace function (lab 1 only)
74 void
75 test_backtrace(int x)
76 {
77     f0100040: 55          push    %ebp
78     f0100041: 89 e5      mov     %esp,%ebp
79     f0100043: 56          push    %esi
80     f0100044: 53          push    %ebx
81     f0100045: e8 72 01 00 00 call   f01001bc <_x86.get_pc_thunk.bx>
82     f010004a: 81 c3 be 02 01 00 add     $0x102be,%ebx
83     f0100050: 8b 75 08    mov     0x8(%ebp),%esi
84     | cprintf("entering test_backtrace %d\n", x);
85     f0100053: 83 ec 08    sub     $0x8,%esp
86     f0100056: 56          push    %esi
87     f0100057: 8d 83 18 17 ff ff lea     -0xe8e8(%ebx),%eax
88     f010005d: 50          push    %eax
89     f010005e: e8 d6 09 00 00 call   f0100a39 <printf>
90     | if (x > 0)
91     f0100063: 83 c4 10    add     $0x10,%esp
92     f0100066: 85 f6      test    %esi,%esi
93     f0100068: 7e 29      jle     f0100093 <test_backtrace+0x53>
94     | test_backtrace(x-1);
95     f010006a: 83 ec 0c    sub     $0xc,%esp
96     f010006d: 8d 46 ff    lea     -0x1(%esi),%eax
97     f0100070: 50          push    %eax
98     f0100071: e8 ca ff ff ff call   f0100040 <test_backtrace>
99     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: ~$ 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:2600/" < .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::26
000 -D qemu.log -S
6828 decimal is 15254 octal!

gautam-ahuja@LAPTOP-FV76: ~$ make gdb
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 0x6a06a000
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>: 0xf68510c0 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 <1386_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 0xc0ec8329 0x50ff468d 0xffffcae8
0xf0100075 <test_backtrace+53>: 0x10c483ff 0x5608ec83 0x1734838d 0xe850ffff
(gdb)

```


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:

```
jos > kern > C monitor.c > ...
56
57 int
58 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
59 {
60     // Your code here.
61     // Get the current base stack pointer (ebp)
62     uint32_t ebp = read_ebp();
63     // In a stack frame the first value 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
64     // Now we have to iterate over the stack frames and print the values of the stack frame.
65     printf("Stack backtrace:");
66     while(ebp != 0){
67         printf("\n");
68         // %08x is used to print the value in hexadecimal format with 8 digits.
69         printf("ebp %08x eip %08x args %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));
70         // Update the base pointer to the previous stack frame
71         ebp = *((uint32_t*)ebp);
72     }
73     printf("\n");
74     return 0;
75 }
76
```

The output of the code is:

```
gautam-ahuja@LAPTOP-FV76: ~/.../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
aerial mon:stdio -gdb tcp:26000 -D qemu.log -S
6028 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-FV76: ~/.../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/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 i386 settings.

The target architecture is set to "i386".
[f000:ffff] 0xffff0: jmp $0xf000,$0xe05b
0xffff0 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`.

```
josh@kern > c kdebug.c > debuginfo_eip(uintptr_t, Eipdebuginfo *)
182
183 // Already searched within that file's stabs for the function definition
184 // Now, search within the function definition for the line number.
185 // Here we use the N_SLINE stab type.
186 stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
187 if (lline <= rline)
188     info->eip_line = stabs[lline].n_desc;
189 else
190     // Line number not found, return -1.
191     return -1;
192
```

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

```
josh@kern > kernel.ld
23
24
25 /* Include debugging information in kernel memory */
26 .stab : {
27     PROVIDE(__STAB_BEGIN__ = .);
28     *(.stab);
29     PROVIDE(__STAB_END__ = .);
30     BYTE(0) /* Force the linker to allocate space
31             for this section */
32 }
33
```

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

```
gautam-ahuja@LAPTOP-FV76: ~$ 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-FV76: ~$ 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-FV76: ~$
```

```

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`.

```

jos> kern > C monitor.c > mon_backtrace(int, char **, Trapframe *)
20
57 int
58 mon_backtrace(int argc, char **argv, struct Trapframe *tf)
59 {
60     // Your code here.
61     // Q11 Edit: Get the current base stack pointer (ebp)
62     uint32_t ebp = read_ebp();
63     // Q12 Edit: Create a structure to store the debug information
64     struct Eipdebuginfo info;
65
66     // In a stack first value is the return address, second value is the base pointer of the previous stack frame (eip). The rest are the arguments or local variables
67     // 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
68     cprintf("Stack backtrace:");
69     while(ebp != 0){
70         cprintf("\n");
71         // %08x is used to print the value in hexadecimal format with 8 digits.
72         // Q11 Edit: Print Stack Information
73         cprintf("ebp %08x eip %08x args %08x %08x %08x %08x", ebp, *((uint32_t*)ebp + 1), *((uint32_t*)ebp + 2), *((uint32_t*)ebp + 3), *((uint32_t*)ebp + 4), *
74         *((uint32_t*)ebp + 5), *((uint32_t*)ebp + 6));
75
76         // Q12 Edit: Print Debug Information, send the eip value to the function as it is the address of the instruction that caused the call.
77         debuginfo_eip(*((uint32_t*)ebp + 1), &info);
78         // Print the debug information
79         cprintf(" %s:%d: %s\n", info.eip_file, info.eip_line, info.eip_fn_name, info.eip_fn_name, *((uint32_t*)ebp + 1) - info.eip_addr);
80
81         // Update the base pointer to the previous stack frame
82         ebp = *((uint32_t*)ebp);
83     }
84     cprintf("\n");
85     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.

```
josh > kern > C monitor.c > ...

23
    You, 11 seconds ago | 2 authors (github-classroom[bot] and others)
24 static struct Command commands[] = {
25     { "help", "Display this list of commands", mon_help },
26     { "kerninfo", "Display information about the kernel", mon_kerninfo },
27     // Adding a new command for backtrack
28     { "backtrace", "Display the stack backtrace", mon_backtrace },
29 };
30
```

After all edits, the final output is:

The image shows two terminal windows side-by-side. The left window displays the output of the JOS kernel monitor, showing a stack backtrace and various kernel messages. The right window shows a GDB session with various status messages and commands.

```

gautam-ahuja@LAPTOP-FV76: X + v X
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: X + v X
gautam-ahuja@LAPTOP-FV76:~$ 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/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 i
nto this configuration
of GDB. Attempting to continue with the default i386 setti
ngs.

The target architecture is set to "i386".
[f000:ffff] 0xffff0: jmp $0xf000,$0xe05b
0x0000ffff0 in ?? ()
+ symbol-file kernel
(gdb) c
Continuing.
```