

**CS 630: OPERATING SYSTEMS DESIGN**  
**COURSE PROJECT REPORT**

**MIT LAB 2**  
**MEMORY MANAGEMENT**

**HARSH SHAH**  
**KRISHNA MURALI**

## Introduction

MIT Labs deals with implementation of exokernel-style operating systems kernel named JOS under x86 platform and then port it to ARM architecture. Lab 2 deals with the memory management part of the operation system. This project is intended to develop memory management of the operating system and try to answer few questions by solving the given exercises.

## Lab 1: Booting a PC

### Environment Configuration

Hardware Environment:

Memory: 8GB  
Processor: Intel® Core™ i5-3230M CPU @ 2.60GHz × 4  
Graphics: Intel® Ivybridge Mobile  
OS Type: 64 bit  
Disk: 750GB

Software Environment:

OS: Ubuntu 16.04 LTS(x86\_64)  
Gcc: Gcc 5.4.0  
Make: GNU Make 4.1  
Gdb: GNU gdb 7.11.1

### Test Compiler Toolchain

```
krishnamurali@krishnamurali-virtual-machine:~/python$ objdump -iBFD header file version (GNU Binutils for Ubuntu) 2.26.1elf64-x86-64(header little endian, data little endian)i386elf32-i386
```

### QEMU Emulator

```
$ git clone http://web.mit.edu/ccutler/www/qemu.git -b 6.828-2.3.0
$ cd qemu
$ ./configure --disable-kvm --target-list="i386-softmmu x86_64-softmmu"
$ make
$ sudo make install
```

### PC Bootstrap

#### Simulating the x86

```
krishnamurali@krishnamurali-virtual-machine:~/lab$ make
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
```

```
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
+ as boot/boot.S
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
```

After compiling, we now have our boot loader(obj/boot/boot) and our kernel(obj/kern/kernel), So where is the disk? The kernel.img is the disk image, which is acting as the virtual disk here. From kern/Makefrag we can see that both our boot loader and kernel have been written to the image (using the dd command).

Now we can be running the QEMU like running a real PC.

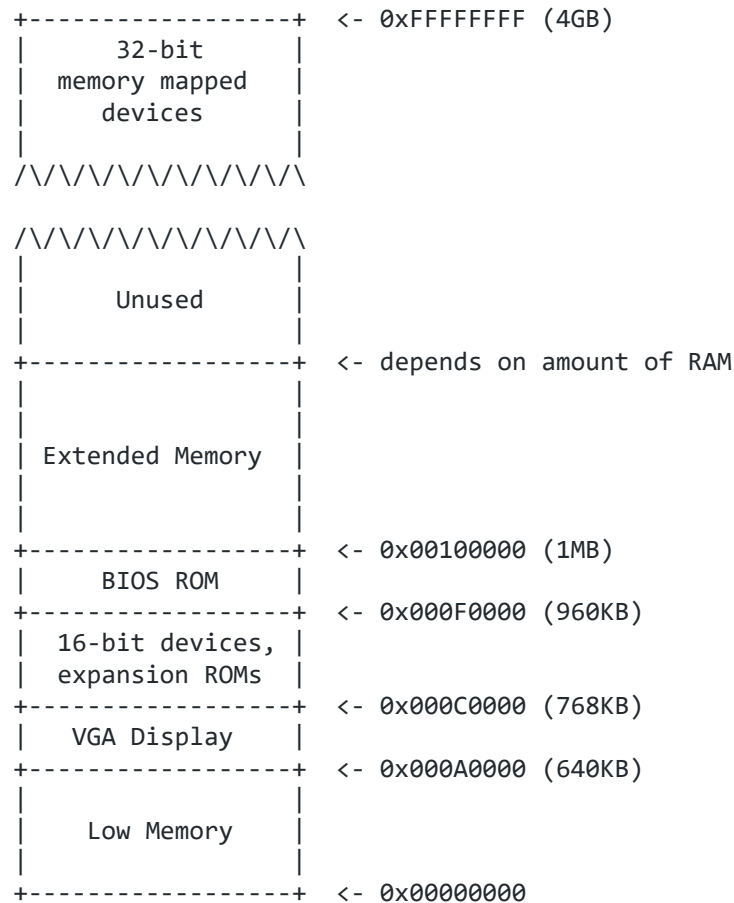
```
krishnamurali@krishnamurali-virtual-machine:~/lab$ make qemu
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
qemu -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log
WARNING: Image format was not specified for 'obj/kern/kernel.img' and probing guessed raw.
```

Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.

Specify the 'raw' format explicitly to remove the restrictions.

```
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>
```

## PC Physical Address Space



## The ROM BIOS

```
The target architecture is assumed to be i8086
[f000:ffff] 0xffff0: ljmp $0xf000,$0xe05b
0x0000ffff in ?? ()
```

With GDB, we know ljmp the first instruction to be executed after power-up, i.e.

- The IBM PC starts executing at physical address 0x000ffff0, which is at the very top of the 64KB area reserved for the ROM BIOS.
- The PC starts executing with CS = 0xf000 and IP = 0xffff0.
- The first instruction to be executed is a jmp instruction, which jumps to the segmented address CS = 0xf000 and IP = 0xe05b.

Currently, we are still in the real mode, so address translation works according to the formula

**physical address = 16 \* segment + offset**

## Function of BIOS

- sets up an interrupt descriptor table
- initializes various devices such as the VGA display
- searches for a bootable device such as a floppy, hard drive, or CD-ROM
- when it finds a bootable disk, the BIOS reads the boot loader from the disk and transfers control to it

## The Boot Loader

A boot loader is a computer program that loads an operating system or some other system software for the computer after completion of the power-on self-tests; it is the loader for the operating system itself. Within the hard reboot process, it runs after completion of the self-tests, then loads and runs the software.

In the conventional hard drive boot mechanism (which we use here), the boot loader(obj/boot/boot) resides in the first sector of our boot device, which we also call boot sector. After finishing its work, BIOS loads the 512-byte boot sector into memory at physical addresses 0x7c00 through 0x7dff, then uses a jmp instruction to set the CS:IP to 0000:7c00, passing control to the boot loader.

## Function of boot loader

- switches the processor from real mode to 32-bit protected mode
- reads the kernel from the hard disk
- transfers control to kernel

## Significance of 32-bit Protected Mode

In real mode, memory is limited to only 1MB. Valid address ranges from 0x00000 to 0xFFFFF, this requires a 20-bit number, which will not fit to any of 8086's 16-bit registers. Intel solved this problem by segment: offset pair we talked above. However, real segmented address have disadvantages:

- A single segment can only refer to 64K of memory (16 bit of offset). When a program has more than 64K of code, the program must be split into sections (called *segments*) and the value of CS must be changed. Similar problem occurs with large amounts of data and the DS register. This can be very awkward.
- Each byte in memory does not have an unique segmented address. The physical address 04808 can be referenced by 047C:0048, 047D:0038, 047E:0028 or 047B:0058. This can complicate the comparison of segmented addresses

In 80286, Intel invented 16-bit protected mode. We still use the selector: offset pair to realize address translation. However, the former of the pair is not called segment any more, it's now called selector. In real mode, the former value of the pair is a paragraph number of physical memory. In protected mode, a selector value is an index into a descriptor table. In both modes, programs are divided into segments. In real mode, these segments are at fixed positions in physical memory and the selector value denotes the paragraph number of the beginning of the segment. In protected mode, the segments are not at fixed positions in physical memory. In fact, they do not have to be in memory at all.

Protected mode uses a technique called virtual memory. In 16-bit protected mode, segments are moved between memory and disk as needed. All of this is done transparently by the operating system. The program does not have to be written differently for virtual memory to work.

In protected mode, each segment is assigned an entry in a descriptor table. This entry has all the information that the system needs to know about the segment. This information includes: is it currently in memory; if in memory, where is it; access permissions (e.g., read-only). The index of the entry of the segment is the selector value that is stored in segment registers.

One big disadvantage of 16-bit protected mode is that *offsets are still 16-bit quantities*. Because of this, segment sizes are still limited to at most 64K. This makes the use of large arrays problematic.

The 80386 introduced 32-bit protected mode. There are two major differences between 386 32-bit and 286 16-bit protected modes:

- Offsets are expanded to be 32-bits. Thus, segments can have sizes up to 4GB.
- Segments can be divided into smaller 4K-sized units called pages. The virtual memory system works with pages now instead of segments. This means that only parts of segment may be in memory at any one time. In 286 16-bit mode, either the entire segment is in memory or none of it is. This is not practical with the larger segments that 32-bit mode allows.

### Additional information about protected mode

From 8086 to 80286, Intel introduced protected mode which enable protection to memory and other peripheral device on hardware level (by Privilege levels and other mechanisms to restrict memory access). On the other hand, it introduced virtual memory, which enable independence of physical space and logic space and raises utilization of memory. While in 80386, Intel expanded offsets to 32 bits (which allow 4G memory space) and introduced paging (which raises utilization of memory more)

### Exercise 3

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

In boot.S, the `ljmp $PROT_MODE_CSEG, $protcseg` causes the switch from 16- to 32-bit mode in the boot.S:

```
# 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
```

### Working principle of ljmp instruction

To enable 32-bit protected mode, we have to prepare the GDT first (we can use the `lgdt` command), then we enable the PE bit on CR0. Note that to complete the process of loading a new GDT, the segment registers need to be reloaded. The CS register must be loaded using a far jump

For the `ljmp` instruction, In Real Address Mode or Virtual 8086 mode, the former pointer provides 16 bits for the CS register. In protected mode, the former 16-bit now works as selector. And `PROT_MODE_CSEG(0x8)` ensure that we still work in the same segment. The offset `$protcseg` is exactly the next instruction. Till now, we have switch to 32-bit mode, but we still work in the same program logic segment.

## Reason for PROT\_MODE\_CSEG at 0x8, PROT\_MODE\_DSEG at 0x10

# Bootstrap GDT

.p2align 2                      # force 4 byte alignment

gdt:

SEG\_NULL                      # null seg

SEG(STA\_X|STA\_R, 0x0, 0xffffffff)    # code seg

SEG(STA\_W, 0x0, 0xffffffff)    # data seg

gdtdesc:

.word 0x17                      # sizeof(gdt) - 1

.long gdt                      # address gdt

Null segment, code segment and data segment all start at 0x0, their limit are 0xffffffff(4G).

## Q: What is the last instruction of the boot loader executed?

In main.c

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

in obj/boot/boot.asm, it's

```
// call the entry point from the ELF header
// note: does not return!
((void (*)(void)) (ELFHDR->e_entry))();
7d6b:        ff 15 18 00 01 00        call    *0x10018
```

After this instruction, the boot loader transfers control to the kernel.

## What is the first instruction of the kernel it just loaded?

In entry.S

```
movw     $0x1234,0x472                      # warm boot
```

In obj/kern/kernel.asm

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

## Where is the first instruction of the kernel?

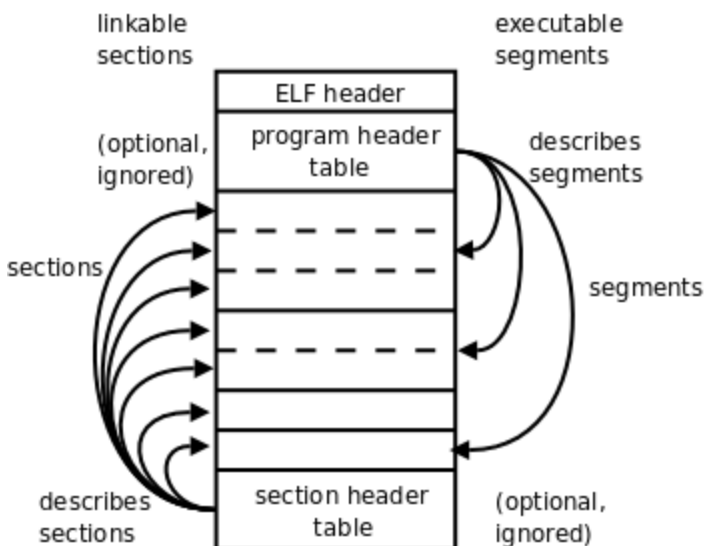
Since the last instruction the boot loader executed is call \*0x10018, the first instruction of the kernel should be at \*0x10018.

```
(gdb) x/1x 0x10018
0x10018: 0x0010000c
```

**Q: How does the boot loader decide how many sectors it must read to fetch the entire kernel from disk? Where does it find this information?**

1. The boot loader read the first page of the disk image.
2. Then, it reads the program head table to get all the program segment information.
3. Last, it loads each of the segment to its ph->ppa address. p\_pa is the load address of this segment (as well as the physical address)

```
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);
```



As we can see, An ELF file has two views: the program header shows the segments used at run-time, whereas the section header lists the set of sections of the binary. We now only focus on run-time view.

## Loading the Kernel

**link address(VMA)** The link address of a section is the memory address from which the section expects to execute.

**load address(LMA)** The load address of a section is the memory address at which that section should be loaded into memory

- program header table of obj/boot/boot.out

start address 0x00007c00



Program Header:

```
LOAD off 0x00000074 vaddr 0x00007c00 paddr 0x00007c00 align 2**2
      filesz 0x00000230 memsz 0x00000230 flags rwx
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
      filesz 0x00000000 memsz 0x00000000 flags rwx
```

- program header table of obj/kern/kernel

start address 0x0010000c

Program Header:

```
LOAD off 0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
      filesz 0x0000712f memsz 0x0000712f flags r-x
LOAD off 0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
      filesz 0x0000a300 memsz 0x0000a944 flags rw-
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
      filesz 0x00000000 memsz 0x00000000 flags rwx
```

we can see that for boot, the link address is the same with the load address. However, for the kernel, the link address is different with the load address.

## Exercise 5

Change the boot loader's link address in boot/Makefrag from 0x7c00 to 0x7e00 and recompile the lab.

We can see that the start address of the boot have been changed to 0x7e00. But BIOS still load the boot loader to 0x7c00. We set a breakpoint at 0x7c00, the first few instruction still work. However, when it comes to address reference at lgdt instruction, it would do the wrong thing.

```
0x00007c18 in ?? ()
(gdb) si
[ 0:7c1a] => 0x7c1a: mov    $0xdf,%al
0x00007c1a in ?? ()
(gdb) si
[ 0:7c1c] => 0x7c1c: out    %al,$0x60
0x00007c1c in ?? ()
(gdb) si
[ 0:7c1e] => 0x7c1e: lgdtw  0x7e64
0x00007c1e in ?? ()
(gdb) x/6xb 0x7e64
0x7e64: 0x00  0x00  0x00  0x00  0x00  0x00
(gdb) x/6xb 0x7c64
0x7c64: 0x17  0x00  0x4c  0x7e  0x00  0x00
(gdb) si
[ 0:7c23] => 0x7c23: mov    %cr0,%eax
0x00007c23 in ?? ()
(gdb) c
Continuing.
```

```

00007e00 <start>:
.set CR0_PE_ON,      0x1          # protected mode enable flag

.globl start
start:
.code16                      # Assemble for 16-bit mode
cli                          # Disable interrupts
7e00:          fa          cli
.cld                        # String operations increment
7e01:          fc          cld

# Set up the important data segment registers (DS, ES, SS).
xorw    %ax,%ax             # Segment number zero
7e02:          31 c0          xor    %eax,%eax
movw    %ax,%ds             # --> Data Segment
7e04:          8e d8          mov    %eax,%ds
movw    %ax,%es             # --> Extra Segment
7e06:          8e c0          mov    %eax,%es
movw    %ax,%ss             # --> Stack Segment
7e08:          8e d8          mov    %eax,%ss

00007e0a <seta20.1>:
# Enable A20:
# For backwards compatibility with the earliest PCs, physical
# address line 20 is tied low, so that addresses higher than
# 1MB wrap around to zero by default. This code undoes this.
seta20.1:
inb     $0x64,%al           # Wait for not busy
7e0a:          e4 64          in     $0x64,%al
testb   $0x2,%al
7e0c:          a8 02          test    $0x2,%al

```

## Exercise 6

Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint?

Before the BIOS enters the boot loader

```

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

```

Before the boot loader enters the kernel

```

(gdb) x/8x 0x100000
0x100000:      0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:      0x34000004      0x0000b812      0x220f0011      0xc0200fd8

```

Because the boot loader loads the kernel to 0x100000

## Kernel

### Working principle of Paging

At lab1, we map the first 4MB of physical memory using hand-written, statically-initialized page directory and page table in kern/entrypgdir.c.

Up until kern/entry.S sets the CRO\_PG flag, memory references are treated as physical addresses. Once CRO\_PG is set, memory references are virtual addresses that get translated by the virtual memory hardware to physical addresses. entry\_pgdir translates

- virtual addresses 0xf0000000 through 0xf0400000 to physical addresses 0x00000000 through 0x00400000
- virtual addresses 0x00000000 through 0x00400000 to physical addresses 0x00000000 through 0x00400000.

Any virtual address that is not in one of these two ranges will cause a hardware exception which, since we haven't set up interrupt handling yet, will cause QEMU to dump the machine state and exit.

Right now, each memory reference must look up to the `entry_pgdir` first. After finding its PTE index, its real physical is known.

### Exercise 7

**Q: Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at `0x00100000` and at `0xf0100000`. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at `0x00100000` and at `0xf0100000`. Make sure you understand what just happened.**

```
(gdb) x/10i 0x10000c
=> 0x10000c:    movw  $0x1234,0x472
0x100015:      mov   $0x110000,%eax
0x10001a:      mov   %eax,%cr3
0x10001d:      mov   %cr0,%eax
0x100020:      or    $0x80010001,%eax
0x100025:      mov   %eax,%cr0
0x100028:      mov   $0xf010002f,%eax
0x10002d:      jmp   *%eax
0x10002f:      mov   $0x0,%ebp
0x100034:      mov   $0xf0110000,%esp
(gdb) stepi 5
=> 0x100025:    mov   %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4bx 0x00100000
0x100000:      0x02    0xb0    0xad    0x1b
(gdb) x/4bx 0xf0100000
0xf0100000 <_start+4026531828>: 0x00    0x00    0x00    0x00
(gdb) stepi
=> 0x100028:    mov   $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/4bx 0x00100000
0x100000:      0x02    0xb0    0xad    0x1b
(gdb) x/4bx 0xf0100000
0xf0100000 <_start+4026531828>: 0x02    0xb0    0xad    0x1b
```

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

`jmp *%eax` would fail because `0xf010002c` is outside of RAM  
qemu: fatal: Trying to execute code outside RAM or ROM at `0xf010002c`

## Formatted Printing to the Console

### Exercise 8

**Q: We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.**

```
case 'o':
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
// ...
```

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

console.c exports cputchar getchar iscons, while cputchar is used as a parameter when printf.c calls vprintfmt in printfmt.c.

**Q. Explain the following from console.c:**

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

When the screen is full, scroll down one row to show newer information.

**Run the following code.**

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

Output is He110 World

Because 57616= 0xe110 so first part is He110

i=0x00646c72 is treated as string so 'r'=(char)0x72 'l'=(char)0x6c 'd'=(char)0x64, and 0x00 is marked as end of string

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

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

x=3 y=SOME\_VALUE because y is decimal value of 4 bytes, is placed above 3 in stack.

## The Stack

### Exercise 9

Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

In entry.S

```
# 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
```

In obj/kern/kern.asm

```
        # Set the stack pointer
        movl    $(bootstacktop),%esp
f0100034:    bc 00 00 11 f0                mov    $0xf0110000,%esp
```

So, the stack starts at 0xf0110000, its range is 0xf0108000-0xf0110000.

### Exercise 10

Q: To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words? Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.

Before `test_backtrace(5)`, we push the function parameter to stack, then we push the return address to stack. Above all, we are still at the caller-stack. Next, we enter C calling conventions

```
push    %ebp
mov     %esp,%ebp
push    %ebx
sub     $0xc,%esp
```

These belong to the callee-stack.

## Exercise 11

```
krishnamurali@krishnamurali-virtual-machine: ~/lab
entering test_backtrace 0
Stack backtrace:
ebp f010ff18 eip f010007b args 00000000 00000000 00000000 00000000 f0100936
ebp f010ff38 eip f0100068 args 00000000 00000001 f010ff78 00000000 f0100936
ebp f010ff58 eip f0100068 args 00000001 00000002 f010ff98 00000000 f0100936
ebp f010ff78 eip f0100068 args 00000002 00000003 f010ffb8 00000000 f0100936
ebp f010ff98 eip f0100068 args 00000003 00000004 00000000 00000000 00000000
ebp f010ffb8 eip f0100068 args 00000004 00000005 00000000 00010094 00010094
ebp f010ffd8 eip f01000d4 args 00000005 00001aac 00000644 00000000 00000000
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
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 f010ff68 eip f0100902 args 00000001 f010ff80 00000000 f010ffc8 f0112540
ebp f010ffd8 eip f01000e1 args 00000000 00001aac 00000644 00000000 00000000
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
K>
```

## Exercise 12

**Q: Modify your stack backtrace function to display, for each eip, the function name, source file name, and line number corresponding to that eip.**

In This Question, we know the eip(the return address), and we want to know which function this eip belong to, which file this function belong to. And which line this eip in the file.

To get all this info, we need get the debug info Eipdebuginfo, this struct stores all the information we need.

// Debug information about an instruction pointer

```
struct Eipdebuginfo {
    const char *eip_file;           // Source code filename for EIP
    int eip_line;                   // Source code linenumber for EIP

    const char *eip_fn_name;        // Name of function containing EIP
                                     // - Note: not null terminated!
    int eip_fn_namelen;              // Length of function name
    uintptr_t eip_fn_addr;           // Address of start of function
    int eip_fn_narg;                 // Number of function arguments
};
```



To get this struct, we need the `debuginfo_eip(addr, info)` function. This function searches the STAB table to Fill in the 'info' structure with information about the specified instruction address, 'addr'. The `debuginfo_eip` function has been given in `kdebug.c`, we just need to add this.

```
// Your code here.
    stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
info->eip_line = stabs[lline].n_desc;
```

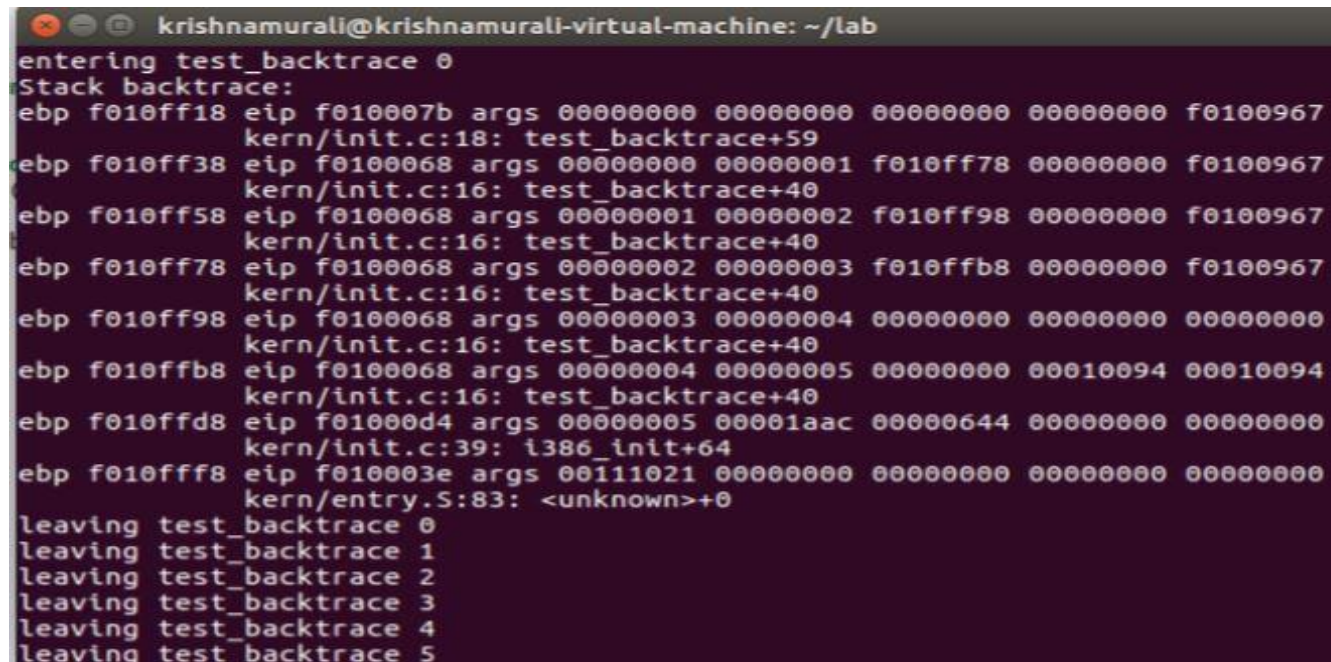
In `monitor.c`

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
    // Your code here.
    unsigned int *ebp = ((unsigned int*)read_ebp());
    printf("Stack backtrace:\n");

    while(ebp) {
        printf("ebp %08x ", ebp);
        printf("eip %08x args", ebp[1]);
        for(int i = 2; i <= 6; i++)
            printf(" %08x", ebp[i]);
        printf("\n");

        unsigned int eip = ebp[1];
        struct Eipdebuginfo info;
        debuginfo_eip(eip, &info);
        printf("\t%s:%d: %.s+%-d\n",
            info.eip_file, info.eip_line,
            info.eip_fn_name, info.eip_fn_name,
            eip-info.eip_fn_addr);

        ebp = (unsigned int*)(*ebp);
    }
    return 0;
}
```



```
krishnamurali@krishnamurali-virtual-machine: ~/lab
entering test_backtrace 0
Stack backtrace:
ebp f010ff18 eip f010007b args 00000000 00000000 00000000 00000000 f0100967
    kern/init.c:18: test_backtrace+59
ebp f010ff38 eip f0100068 args 00000000 00000001 f010ff78 00000000 f0100967
    kern/init.c:16: test_backtrace+40
ebp f010ff58 eip f0100068 args 00000001 00000002 f010ff98 00000000 f0100967
    kern/init.c:16: test_backtrace+40
ebp f010ff78 eip f0100068 args 00000002 00000003 f010ffb8 00000000 f0100967
    kern/init.c:16: test_backtrace+40
ebp f010ff98 eip f0100068 args 00000003 00000004 00000000 00000000 00000000
    kern/init.c:16: test_backtrace+40
ebp f010ffb8 eip f0100068 args 00000004 00000005 00000000 00010094 00010094
    kern/init.c:16: test_backtrace+40
ebp f010ffd8 eip f01000d4 args 00000005 00001aac 00000644 00000000 00000000
    kern/init.c:39: i386_init+64
ebp f010fff8 eip f010003e args 00111021 00000000 00000000 00000000 00000000
    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
```

### **Goals achieved in Lab 1**

- We got familiarized with x86 assembly language, the QEMU x86 emulator, and the PC's power-on bootstrap procedure.
- We examined the boot loader for our 6.828 kernel, which resides in the boot directory of the lab tree.
- Finally, we delved into the initial template for our 6.828 kernel itself, named JOS, which resides in the kernel directory.



## Lab 2: Memory Management

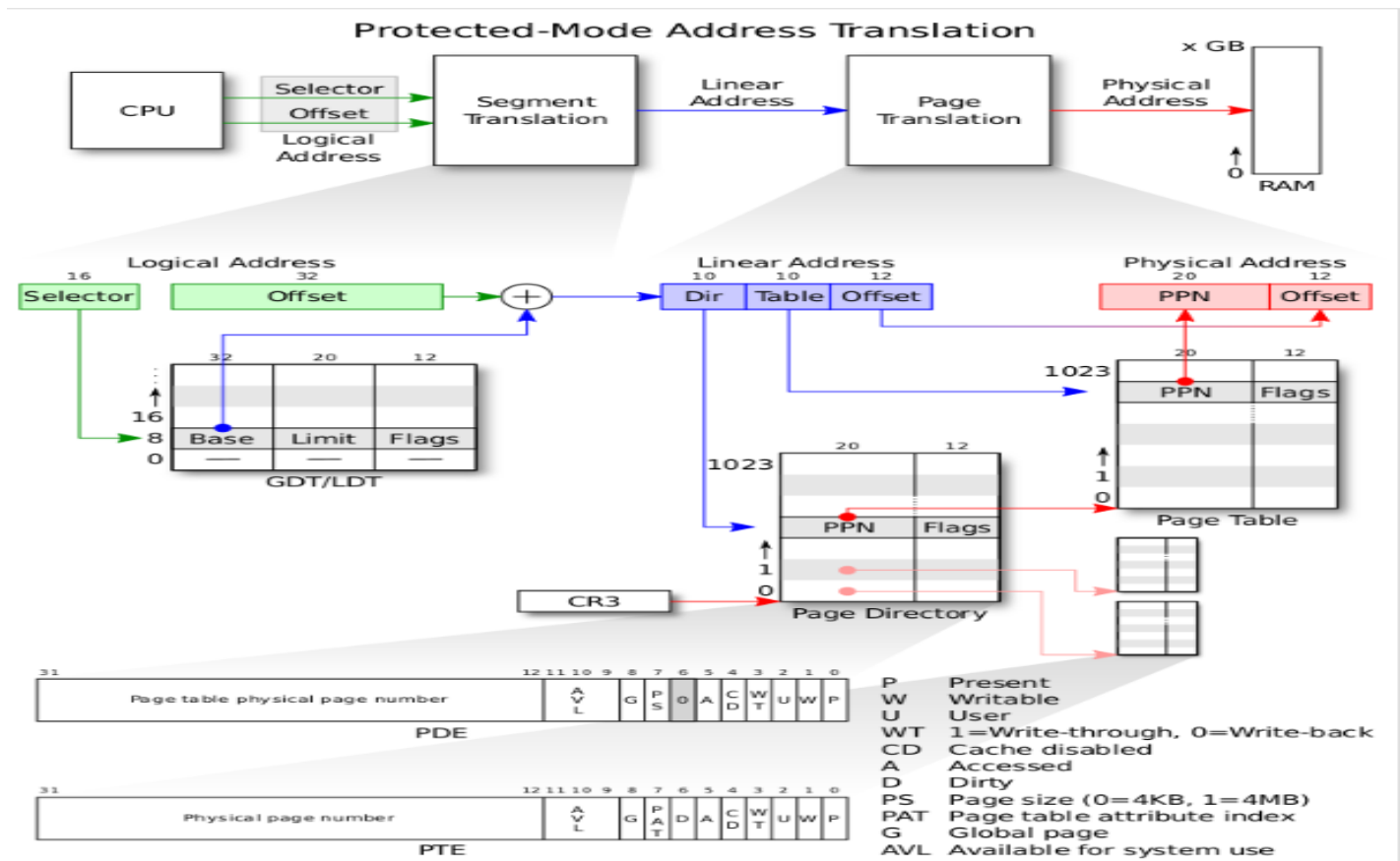
### PHYSICAL PAGE MANAGEMENT:

The operating system keeps the track of which parts of physical RAM are free and which are currently in use. JOS manages the PC's physical memory with *page granularity* so that it can use the MMU to map and protect each piece of allocated memory.

There are two stages of memory management:

1. Segment – based memory management:  
Allocate => segments  
Arbitrate => segment registers
2. Page – based memory management:  
Allocate => pages -> page frames  
Arbitrate => page tables

The **image below** summarizes both phases of the transformation from a logical address to a physical address when paging is enabled. The page translation mechanism uses 2-level page tables to convert the Dir, Table, and Offset fields of a linear address into the physical address:



The addressing mechanism uses the `Dir` field as an index into a page directory, uses the `Table` field as an index into the page table determined by the page directory, and uses the `Offset` field to address a byte within the page determined by the page table. The physical page number (PPN) specifies the physical starting address of a page. Because pages are located on 4K boundaries, the low-order 12 bits (from bit 12 to bit 23) are always zero. In a page directory, the PPN is the address of a page table. In a second-level page table, the PPN is the address of the physical page frame that contains the desired memory operand.

**Exercise 1.** In the file `kern/pmap.c`, you must implement code for the following functions (probably in the order given).

`boot_alloc()`  
`mem_init()` (only up to the call to `check_page_free_list(1)`)  
`page_init()`  
`page_alloc()`  
`page_free()`

`check_page_free_list()` and `check_page_alloc()` test your physical page allocator. You should boot JOS and see whether `check_page_alloc()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your assumptions are correct.

`boot_alloc`:

```
// ROUNDUP to make sure nextfree is kept aligned
// to a multiple of PGSIZE
cprintf("boot_alloc memory at %x\n", nextfree);
cprintf("Next memory at %x\n", ROUNDUP((char *) (nextfree+n), PGSIZE));
if (n != 0) {
    char *next = nextfree;
    nextfree = ROUNDUP((char *) (nextfree+n), PGSIZE);
    return next;
} else return nextfree;
```

Q: Then extended memory [`EXTPHYSMEM`, ...]. Some of it is in use, some is free. Where is the kernel in physical memory? Which pages are already in use for page tables and other data structures?

`page_init`:

```
void
page_init(void)
{
    size_t i;
    for (i = 1; i < npages_basemem; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

```

    }
    int med = (int)ROUNDUP(((char*)pages) + (sizeof(struct PageInfo) * npages) - 0xf0000000, PGSIZE)/PGSIZE;
    cprintf("%d\n", ((char*)pages) + (sizeof(struct PageInfo) * npages));
    cprintf("med=%d\n", med);
    for (i = med; i < npages; i++) {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}

```

Memory after  $(\text{int})\text{ROUNDUP}(((\text{char}^*)\text{pages}) + (\text{sizeof}(\text{struct PageInfo}) * \text{npages}) - 0xf0000000, \text{PGSIZE})/\text{PGSIZE}$ ; is free.

page\_alloc:

```

struct PageInfo *
page_alloc(int alloc_flags)
{
    if (page_free_list) {
        struct PageInfo *ret = page_free_list;
        page_free_list = page_free_list->pp_link;
        if (alloc_flags & ALLOC_ZERO)
            memset(page2kva(ret), 0, PGSIZE);
        return ret;
    }
    return NULL;
}

```

page\_free:

```

void
page_free(struct PageInfo *pp)
{
    pp->pp_link = page_free_list;
    page_free_list = pp;
}

```

```

$ make qemu-nox
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
boot_alloc memory at f0116000
Next memory at f0117000
boot_alloc memory at f0117000
Next memory at f0138000
npages: 16639
npages_basemem: 160
pages: f0117000
-267159560

```

```

med=312
boot_alloc memory at f0138000
Next memory at f0138000
check_page_free_list done
check_page_alloc() succeeded!

```

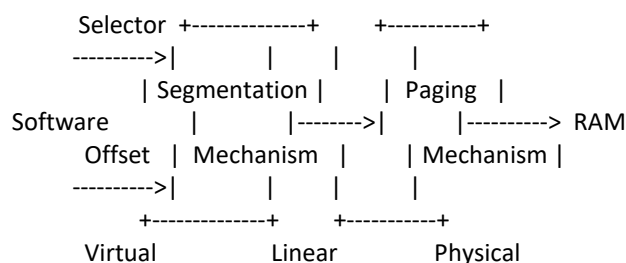
### Understanding of Exercise 1:

When implementing the `page_init` function, it's worth mentioning that the page tables were not allocated to physical memory address `UTEMP` (defined in `inc/memlayout.h`); instead, because in function `mem_init` the page directories and page tables were allocated using function `boot_alloc`, therefore, simply using `boot_alloc(0)` can point to the first free page after the memory chunk that stores page tables.

## **VIRTUAL MEMORY:**

### **Virtual, Linear, and Physical Addresses**

In x86 terminology, a *virtual address* consists of a segment selector and an offset within the segment. A *linear address* is what you get after segment translation but before page translation. A *physical address* is what you finally get after both segment and page translation and what ultimately goes out on the hardware bus to your RAM.



Recalling the part 3 of lab 1, we installed a simple page table so that the kernel could run at its link address of `0xf0100000`, even though it is actually loaded in physical memory just above the ROM BIOS at `0x00100000`. This page table mapped only 4MB of memory. In the virtual memory layout you are going to set up for JOS in this lab, we'll expand this to map the first 256MB of physical memory starting at virtual address `0xf0000000` and to map a number of other regions of virtual memory.

**Exercise 3.** While GDB can only access QEMU's memory by virtual address, it's often useful to be able to inspect physical memory while setting up virtual memory. Review the QEMU [monitor commands](#) from the lab tools guide, especially the `xp` command, which lets you inspect physical memory. To access the QEMU monitor, press **Ctrl-a c** in the terminal (the same binding returns to the serial console).

Use the `xp` command in the QEMU monitor and the `x` command in GDB to inspect memory at corresponding physical and virtual addresses and make sure you see the same data.

Our patched version of QEMU provides an `info pg` command that may also prove useful: it shows a compact but detailed representation of the current page tables, including all mapped memory ranges, permissions, and flags. Stock QEMU also provides an `info mem` command that shows an overview of which ranges of virtual memory are mapped and with what permissions.

Inspecting memory at corresponding physical (in QEMU monitor) and virtual addresses (in GDB):

```
1  (qemu) x/16x 0x100000
2  00100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
3  00100010: 0x34000004 0x2000b812 0x220f0011 0xc0200fd8
4  00100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
5  00100030: 0x00000000 0x112000bc 0x0002e8f0 0xfeeb0000
6
7  (gdb) x/16x 0xf0100000
8  0xf0100000 <_start+4026531828>: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
9  0xf0100010 <entry+4>: 0x34000004 0x2000b812 0x220f0011 0xc0200fd8
10 0xf0100020 <entry+20>: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
11 0xf0100030 <relocated+1>: 0x00000000 0x112000bc 0x0002e8f0 0xfeeb0000
```

Inspecting detailed representation of the current page tables in QEMU monitor:

```
1  (qemu) info pg
2  VPN range      Entry      Flags      Physical page
3  [00000-003ff]  PDE[000]    ----A----P
4  [00000-00000]  PTE[000]    -----WP 00000
5  [00001-0009f]  PTE[001-09f] ---DA---WP 00001-0009f
6  [000a0-000b7]  PTE[0a0-0b7] -----WP 000a0-000b7
7  [000b8-000b8]  PTE[0b8]    ---DA---WP 000b8
8  [000b9-000ff]  PTE[0b9-0ff] -----WP 000b9-000ff
9  [00100-00102]  PTE[100-102] ----A---WP 00100-00102
10 [00103-00110]  PTE[103-110] -----WP 00103-00110
11 [00111-00111]  PTE[111]    ---DA---WP 00111
12 [00112-00113]  PTE[112-113] -----WP 00112-00113
13 [00114-003ff]  PTE[114-3ff] ---DA---WP 00114-003ff
14 [f0000-f03ff]  PDE[3c0]    ----A---WP
15 [f0000-f0000]  PTE[000]    -----WP 00000
16 [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
17 [f00a0-f00b7]  PTE[0a0-0b7] -----WP 000a0-000b7
18 [f00b8-f00b8]  PTE[0b8]    ---DA---WP 000b8
19 [f00b9-f00ff]  PTE[0b9-0ff] -----WP 000b9-000ff
20 [f0100-f0102]  PTE[100-102] ----A---WP 00100-00102
21 [f0103-f0110]  PTE[103-110] -----WP 00103-00110
22 [f0111-f0111]  PTE[111]    ---DA---WP 00111
23 [f0112-f0113]  PTE[112-113] -----WP 00112-00113
24 [f0114-f03ff]  PTE[114-3ff] ---DA---WP 00114-003ff
```

Show ranges of virtual memory that are mapped and their permissions in QEMU monitor:

```
1 (qemu) info mem
2 0000000000000000-0000000000400000 0000000000400000 -r-
3 00000000f0000000-00000000f0400000 0000000000400000 -rw
```

Ctrl – a c is used to switch between the QEMU serial console and monitor.

Ctrl – a x is used to exit the QEMU.

### Question:

Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr\_t or physaddr\_t?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

### Answer:

uintptr\_t, because va is used in applications.

```
static physaddr_t
check_va2pa(pde_t *pgdir, uintptr_t va)
{
    pte_t *p;

    pgdir = &pgdir[PDX(va)];
    if (!(*pgdir & PTE_P))
        return ~0;
    p = (pte_t*) KADDR(PTE_ADDR(*pgdir));
    if (!p[PTX(va)] & PTE_P)
        return ~0;
    return PTE_ADDR(p[PTX(va)]);
}
```

### Reference counting

The same physical page mapped at multiple virtual addresses simultaneously (or in the address spaces of multiple environments) will keep a count of the number of references to each physical page in the pp\_ref field of the struct PageInfo corresponding to the physical page. When this count goes to zero for a physical page, that page can be freed because it is no longer used. In general, this count should be equal to the number of times the physical page appears *below* UTOP in all page tables (the mappings above UTOP are mostly set up at

boot time by the kernel and should never be freed, so there's no need to reference count them). We'll also use it to keep track of the number of pointers we keep to the page directory pages and, in turn, of the number of references the page directories have page table pages.

## Page Table Management

**Exercise 4.** In the file `kern/pmap.c`, you must implement code for the following functions.

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

`check_page()`, called from `mem_init()`, tests your page table management routines. You should make sure it reports success before proceeding.

`pgdr_walk`:

```
pte_t *
pgdir_walk(pte_t *pgdir, const void *va, int create)
{
    int dindex = PDX(va), tindex = PTX(va);
    //dir index, table index
    if (!(pgdir[dindex] & PTE_P)) {    //if pde not exist
        if (create) {
            struct PageInfo *pg = page_alloc(ALLOC_ZERO);    //alloc a zero page
            if (!pg) return NULL;    //allocation fails
            pg->pp_ref++;
            pgdir[dindex] = page2pa(pg) | PTE_P | PTE_U | PTE_W;
            //we should use PTE_U and PTE_W to pass checkings
        } else return NULL;
    }
    pte_t *p = KADDR(PTE_ADDR(pgdir[dindex]));

    return p+tindex;
}
```

`boot_map_region`:

```
static void
boot_map_region(pte_t *pgdir, uintptr_t va, size_t size, physaddr_t pa, int perm)
{
    int i;
    for (i = 0; i < size/PGSIZE; ++i, va += PGSIZE, pa += PGSIZE) {
        pte_t *pte = pgdir_walk(pgdir, (void *) va, 1);    //create
        if (!pte) panic("boot_map_region panic, out of memory");
        *pte = pa | perm | PTE_P;
    }
}
```

page\_lookup:

```
struct PageInfo *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    pte_t *pte = pgdir_walk(pgdir, va, 0);    //not create
    if (!pte || !(*pte & PTE_P)) return NULL;  //page not found
    if (pte_store)
        *pte_store = pte;    //found and set
    return pa2page(PTE_ADDR(*pte));
}
```

page\_remove:

```
void
page_remove(pde_t *pgdir, void *va)
{
    pte_t *pte;
    struct PageInfo *pg = page_lookup(pgdir, va, &pte);
    if (!pg || !(*pte & PTE_P)) return; //page not exist
    // - The ref count on the physical page should decrement.
    // - The physical page should be freed if the refcount reaches 0.
    page_decref(pg);
    // - The pg table entry corresponding to 'va' should be set to 0.
    *pte = 0;
    // - The TLB must be invalidated if you remove an entry from
    //   the page table.
    tlb_invalidate(pgdir, va);
}
```

page\_insert:

```
int
page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
    pte_t *pte = pgdir_walk(pgdir, va, 1);    //create on demand
    if (!pte)    //page table not allocated
        return -E_NO_MEM;
    //increase ref count beforehand to avoid the corner case that pp is freed before it is inserted.
    pp->pp_ref++;
    if (*pte & PTE_P)    //page colides, tle is invalidated in page_remove
        page_remove(pgdir, va);
    *pte = page2pa(pp) | perm | PTE_P;
    return 0;
}
```

#### Understanding of Exercise 4:

It's important to understand that the `page_alloc` in `pgdir_walk` allocates a page, in other words, it creates a page table, so `page2pa(pp)` in `pgdir_walk` returns the page table's address. However, the function argument `pp` in `page_insert` represents a physical page frame, therefore the `page2pa(pp)` in function `page_insert` returns the page frames' address.



## KERNEL SPACE ADDRESS:

### Permissions and Fault Isolation

Since kernel and user memory are both present in each environment's address space, we will have to use permission bits in our x86 page tables to allow user code access only to the user part of the address space. Otherwise bugs in user code might overwrite kernel data, causing a crash or more subtle malfunction; user code might also be able to steal other environments' private data. Note that the writable permission bit (PTE\_W) affects both user and kernel code!

The user environment will have no permission to any of the memory above ULIM, while the kernel will be able to read and write this memory. For the address range [UTOP,ULIM), both the kernel and the user environment have the same permission: they can read but not write this address range. This range of address is used to expose certain kernel data structures read-only to the user environment. Lastly, the address space below UTOP is for the user environment to use; the user environment will set permissions for accessing this memory.

### Initializing the Kernel Address Space

#### Exercise 5:

**Question:** What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point?

A: See the table below:

1	Entry	Base Virtual Address	Points to (logically):
2	960	0xf0000000	mapped physical memory
3	959	0xffff8000	kernel stack
4	957	0xef400000	page table
5	956	0xef000000	page structures
6	0	0x00000000	start of memory (see entrypghdir.c)

**Q: We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?**

A: The permission bits/flags of PDE/PTE are used to protect the kernel memory.

**Q: What is the maximum amount of physical memory that this operating system can support? Why?**

A: The operating system uses a 4MB space at UPAGES to store the PageInfo structures. It can store 512 structures (a.k.a., 512 physical page frames) because each structure occupies 8B. The total amount of physical memory it can support is thus  $512 \times 4KB = 2GB$ . The maximum memory that can be used by the kernel is 256MB. Base address of mapped physical memory starts from virtual address 0xf0000000, which allows up to 256MB only.

**Q: How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?**

A: Up to 6MB + 4KB space overhead. page directory:  $2^{10} \times 4B = 4KB$ ; per PTE refers to 4KB physical memory, then per PT refers to  $2^{10} \times 4KB = 4MB$  physical memory, needs  $2GB/4MB = 512$  pages tables which takes  $512 \times 4KB = 2MB$ ; PageInfo structures: 4MB.

**Q: Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?**

A: After the instruction "jmp \*%eax" we transition to running at virtual kernel address. The low 4MB of virtual address was mapped to the low 4MB physical address. The transition allows user programs to use the low address space.

## Challenge Question:

*Challenge!* Extend the JOS kernel monitor with commands to:

- Display in a useful and easy-to-read format all of the physical page mappings (or lack thereof) that apply to a particular range of virtual/linear addresses in the currently active address space. For example, you might enter 'showmappings 0x3000 0x5000' to display the physical page mappings and corresponding permission bits that apply to the pages at virtual addresses 0x3000, 0x4000, and 0x5000.
- Explicitly set, clear, or change the permissions of any mapping in the current address space.
- Dump the contents of a range of memory given either a virtual or physical address range. Be sure the dump code behaves correctly when the range extends across page boundaries!
- Do anything else that you think might be useful later for debugging the kernel. (There's a good chance it will be!)

Below is the code followed by the output.

First, I need a function to translate string to address:

```
uint32_t xtoi(char* buf) {
    uint32_t res = 0;
    buf += 2; //0x...
    while (*buf) {
        if (*buf >= 'a') *buf = *buf - 'a' + '0' + 10; //aha
        res = res * 16 + *buf - '0';
        ++buf;
    }
    return res;
}
```

a function that prints pte\_t:

```
void pprint(pte_t *pte) {
    cprintf("PTE_P: %x, PTE_W: %x, PTE_U: %x\n",
        *pte & PTE_P, *pte & PTE_W, *pte & PTE_U);
}
```

Showmappings:

```
int
showmappings(int argc, char **argv, struct Trapframe *tf)
{
    if (argc == 1) {
        cprintf("Usage: showmappings 0xbegin_addr 0xend_addr\n");
        return 0;
    }
    uint32_t begin = xtoi(argv[1]), end = xtoi(argv[2]);
    cprintf("begin: %x, end: %x\n", begin, end);
    for (; begin <= end; begin += PGSIZE) {
```

```

    pte_t *pte = pgdir_walk(kern_pgdir, (void *) begin, 1); //create
    if (!pte) panic("boot_map_region panic, out of memory");
    if (*pte & PTE_P) {
        cprintf("page %x with ", begin);
        pprint(pte);
    } else cprintf("page not exist: %x\n", begin);
}
return 0;
}

```

```

K> showmappings
Usage: showmappings 0xbegin_addr 0xend_addr
K> showmappings 0xf011a000 0xf012a000
begin: f011a000, end: f012a000
page f011a000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011b000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011c000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011d000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011e000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011f000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0120000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0121000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0122000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0123000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0124000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0125000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0126000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0127000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0128000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0129000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f012a000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
K> showmappings 0xef000000 0xef010000
begin: ef000000, end: ef010000
page ef000000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef001000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef002000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef003000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef004000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef005000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef006000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef007000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef008000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef009000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00a000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00b000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00c000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00d000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00e000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00f000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef010000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
K>

```

Here's setm that can set or clear a flag in a specific page:

```
int setm(int argc, char **argv, struct Trapframe *tf) {
    if (argc == 1) {
        cprintf("Usage: setm 0xaddr [0|1 :clear or set] [P|W|U]\n");
        return 0;
    }
    uint32_t addr = xtoi(argv[1]);
    pte_t *pte = pgdir_walk(kern_pgdir, (void *)addr, 1);
    cprintf("%x before setm: ", addr);
    pprint(pte);
    uint32_t perm = 0;
    if (argv[3][0] == 'P') perm = PTE_P;
    if (argv[3][0] == 'W') perm = PTE_W;
    if (argv[3][0] == 'U') perm = PTE_U;
    if (argv[2][0] == '0') //clear
        *pte = *pte & ~perm;
    else //set
        *pte = *pte | perm;
    cprintf("%x after setm: ", addr);
    pprint(pte);
    return 0;
}
```

```
K> setm
Usage: setm 0xaddr [0|1 :clear or set] [P|W|U]
K> setm 0xf011a000 1 U
f011a000 before setm: PTE_P: 1, PTE_W: 2, PTE_U: 0
f011a000 after setm: PTE_P: 1, PTE_W: 2, PTE_U: 4
K> setm 0xf011a000 0 U
f011a000 before setm: PTE_P: 1, PTE_W: 2, PTE_U: 4
f011a000 after setm: PTE_P: 1, PTE_W: 2, PTE_U: 0
K>
```

Here's showvm that is used to view memory:

```
int showvm(int argc, char **argv, struct Trapframe *tf) {
    if (argc == 1) {
        cprintf("Usage: showvm 0xaddr 0xn\n");
        return 0;
    }
    void** addr = (void**) xtoi(argv[1]);
    uint32_t n = xtoi(argv[2]);
    int i;
    for (i = 0; i < n; ++i)
        cprintf("VM at %x is %x\n", addr+i, addr[i]);
    return 0;
}
```

```
K> showmappings
Usage: showmappings 0xbegin_addr 0xend_addr
K> showmappings 0xf011a000 0xf012a000
begin: f011a000, end: f012a000
page f011a000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011b000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011c000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011d000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011e000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f011f000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0120000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0121000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0122000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0123000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0124000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0125000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0126000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0127000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0128000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f0129000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
page f012a000 with PTE_P: 1, PTE_W: 2, PTE_U: 0
K> showmappings 0xef000000 0xef010000
begin: ef000000, end: ef010000
page ef000000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef001000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef002000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef003000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef004000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef005000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef006000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef007000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef008000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef009000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00a000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00b000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00c000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00d000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00e000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef00f000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
page ef010000 with PTE_P: 1, PTE_W: 0, PTE_U: 4
K>
```