LAB 2 -MEMORY MANAGEMENT REPORT

BY
AMAL JOY
CS11B006

NOTEWORTHY POINTS

- **pages are units of 4096 bytes.**
- ***virtual memory* maps the virtual addresses used by kernel and user software to addresses in physical memory.**
- **struct PageInfo {**
  // Next page on the free list.
  struct PageInfo *pp_link;

  uint16_t pp_ref;
  // pp_ref is the count of pointers (usually in page table entries)
  to this page, for pages allocated using page_alloc. Pages allocated at boot time using pmap.c's boot_alloc do not have valid reference count fields.
  };

- **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.**

- **The JOS kernel often needs to manipulate addresses as opaque values or as integers, without dereferencing them, for example in the physical memory allocator. Sometimes these are virtual addresses, and sometimes they are physical addresses. To help document the code, the JOS source distinguishes the two cases: the type `uintptr_t` represents opaque virtual addresses, and `physaddr_t` represents physical addresses. Both these types are really just synonyms for 32-bit integers (`uint32_t`), so the compiler won't stop you from assigning one type to another! Since they are integer types (not pointers), the compiler *will* complain if you try to dereference them.**
- **once we're in protected mode (which we entered first thing in `boot/boot.S`), there's no way to directly use a linear or physical address. *All* memory references are interpreted as virtual addresses**

Question 1. 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;

Variable x should have type uintptr_t.
x is assigned a cast of value. value is a pointer. A pointer has a value equal to the virtual address of the data it points to. So x is assigned a virtual address, and therefore should have type uintptr_t.

NOTEWORTHY POINTS

- In order to translate a physical address into a virtual address that the kernel can actually read and write, the kernel must add 0xf0000000 to the physical address to find its corresponding virtual address in the remapped region. You should use `KADDR(pa)` to do that addition.
- Thus, to turn a virtual address in this region into a physical address, the kernel can simply subtract 0xf0000000. You should use `PADDR(va)` to do that subtraction.

- `page_alloc`. The page it returns will always have a reference count of 0, so `pp_ref` should be incremented as soon as you've done something with the returned page (like inserting it into a page table). Sometimes this is handled by other functions (for example, `page_insert`) and sometimes the function calling `page_alloc` must do it directly.
- we needed to give the kernel such a high link address in lab 1: otherwise there would not be enough room in the kernel's virtual address space to map in a user environment below it at the same time.
- 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.

For Exercise 1,4 and 5 pmap.c is attached.

Exercise 2. Look at chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.2 and 6.4). We recommend that you also skim the sections about segmentation; while JOS uses paging for virtual memory and protection, segment translation and segment-based protection cannot be disabled on the x86, so you will need a basic understanding of it.

In the second phase of address transformation, the 80386 transforms a linear address into a physical address. This phase of address transformation implements the basic features needed for page-oriented virtual-memory systems and page-level protection.
The page-translation step is optional. Page translation is in effect only when the PG bit of CR0 is set. This bit is typically set by the operating system during software initialization. The PG bit must be set if the operating system is to implement multiple virtual 8086 tasks, page-oriented protection, or

page-oriented virtual memory.

A page table is simply an array of 32-bit page specifiers. A page table is itself a page, and therefore contains 4 Kilobytes of memory or at most 1K 32-bit entries.
Two levels of tables are used to address a page of memory. At the higher level is a page directory. The page directory addresses up to 1K page tables of the second level. A page table of the second level addresses up to 1K pages. All the tables addressed by one page directory, therefore, can address 1M pages (2^(20)). Because each page contains 4K bytes 2^(12) bytes), the tables of one page directory can span the entire physical address space of the 80386 (2^(20) times 2^(12) = 2^(32)).

The page frame address specifies the physical starting address of a page. Because pages are located on 4K boundaries, the low-order 12 bits are always zero. In a page directory, the page frame address is the address of a page table. In a second-level page table, the page frame address is the address of the page frame that contains the desired memory operand.

The Present bit indicates whether a page table entry can be used in address translation. P=1 indicates that the entry can be used.

When P=0 in either level of page tables, the entry is not valid for address translation, and the rest of the entry is available for software use; none of the other bits in the entry is tested by the hardware.

If P=0 in either level of page tables when an attempt is made to use a page-table entry for address translation, the processor signals a page exception. In software systems that support paged virtual memory, the page-not-present exception handler can bring the required page into physical memory.

The concept of privilege for pages is implemented by assigning each page to one of two levels:

1. Supervisor level (U/S=0) -- for the operating system and other systems software and related data.
2. User level (U/S=1) -- for applications procedures and data.

The current level (U or S) is related to CPL. If CPL is 0, 1, or 2, the processor is executing at supervisor level. If CPL is 3, the processor is executing at user level.

When the processor is executing at supervisor level, all pages are addressable, but, when the processor is executing at user level, only pages that belong to the user level are addressable.

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.

xp

Displays memory at the specified physical address using the specified format.

format: Used to specify the output format the displayed memory. The format is broken down as */ [count][data_format][size]*

- count: number of item to display (base 10)
- data_format: 'x' for hex, 'd' for decimal, 'u' for unsigned decimal, 'o' for octal, 'c' for char and 'i' for (disassembled) processor instructions
- size: 'b' for 8 bits, 'h' for 16 bits, 'w' for 32 bits or 'g' for 64 bits. On x86 'h' and 'w' can select instruction disassembly code formats.

address:

- Direct address, for example: 0x20000
- Register, for example: $eip

Example - Display 3 instructions on an x86 processor starting at the current instruction:

```
(qemu) xp /3i $eip
```

(qemu) info pg -prints out the current page table

```
VPN range     Entry         Flags        Physical page
[ef000-ef3ff]  PDE[3bc]     -------UWP
[ef000-ef020]  PTE[000-020] -------U-P 0011a-0013a
[ef400-ef7ff]  PDE[3bd]     -------U-P
[ef7bc-ef7bc]  PTE[3bc]     -------UWP 003fd
[ef7bd-ef7bd]  PTE[3bd]     -------U-P 00119
[ef7bf-ef7bf]  PTE[3bf]     -------UWP 003fe
[ef7c0-ef7d0]  PTE[3c0-3d0] ----A--UWP 003ff 003fc 003fb 003fa 003f9 003f8 ..
[ef7d1-ef7ff]  PTE[3d1-3ff] -------UWP 003ec 003eb 003ea 003e9 003e8 003e7 ..
[efc00-effff]  PDE[3bf]     -------UWP
[efff8-effff]  PTE[3f8-3ff] --------WP 0010e-00115
[f0000-f03ff]  PDE[3c0]     ----A--UWP
[f0000-f0000]  PTE[000]     --------WP 00000
[f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
[f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
[f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
[f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
```

```
[f0100-f0105]  PTE[100-105] ----A---WP 00100-00105
[f0106-f0114]  PTE[106-114] --------WP 00106-00114
[f0115-f0115]  PTE[115]     ---DA---WP 00115
[f0116-f0117]  PTE[116-117] --------WP 00116-00117
[f0118-f0119]  PTE[118-119] ---DA---WP 00118-00119
[f011a-f011a]  PTE[11a]     ----A---WP 0011a
[f011b-f011b]  PTE[11b]     ---DA---WP 0011b
[f011c-f013a]  PTE[11c-13a] ----A---WP 0011c-0013a
[f013b-f03bd]  PTE[13b-3bd] ---DA---WP 0013b-003bd
[f03be-f03ff]  PTE[3be-3ff] --------WP 003be-003ff
[f0400-f3fff]  PDE[3c1-3cf] ----A--UWP
[f0400-f3fff]  PTE[000-3ff] ---DA---WP 00400-03fff
[f4000-f43ff]  PDE[3d0]     ----A--UWP
[f4000-f40fe]  PTE[000-0fe] ---DA---WP 04000-040fe
[f40ff-f43ff]  PTE[0ff-3ff] --------WP 040ff-043ff
[f4400-fffff]  PDE[3d1-3ff] -------UWP
[f4400-fffff]  PTE[000-3ff] --------WP 04400-0ffff
```

Question

2. **What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:**

Entry	Base Virtual Address	Points to (logically)
1023	0xffc0000	Page table for top 4MB of phys memory
1022	0xff80000	Page table for second to top 4MB of phys mem
960	0xf0000000	KERNBASE
.	?	?
2	0x00800000	UTEXT
1	0x00400000	UTEMP
0	0x00000000	BIOS

3. (From Lecture 3) **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?**
   User programs will not be able to read or write the kernel's memory because the combination of the hardware and the page table will prevent this.The CPL

bits on %cs indicate whether the process is in user or kernel mode, and each of the page table entries (as well as each of the page directory entries) has a user bit. If the process is in user mode and the program tries to access kernel memory, the hardware will not allow this while the PTE user bit is unset. And since the only way to dereference physical addresses is through the hardware, it will be impossible for the user program to get at the kernel's memory.

4. **What is the maximum amount of physical memory that this operating system can support? Why?**
   With the page table structure we currently have (1 PD with 1023 PDEs;each PT with 1024 PTE; each page with 4KB), this operating system could support up to (1 PD)*(1023 PDEs / PD)*(1 PT / PDE)*(1024 PTEs / PT)*(1 page / PTE)*(4 KB / page) = 4,190,208 KB = 4.19 GB.
5. **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?**
   When using the maximum capacity of the paging hardware, there would be 1024 distinct PTs (including the PD). Each of these PTs consists of 1024 32-bit PTEs. This totals 4.194 MB of overhead.
   We transition to running at an EIP above KERNBASE at kern/entry.S:68,

   With the jmp to <relocated>./kern/entrypgdir.c defines the initial PD and PT that are used in kern/entry.S and all the code that is run before the real paging system is set up.This paging systems maps both the [0, 4MB) and [KERNBASE, KERNBASE+4MB) sets of VAs to the [0, 4MB) set of PAs. So after paging is enabled, the low virtual addresses are still being mapped to the correct low physical addresses.The transition is necessary so that the kernel can map low virtual addresses to user code, and so that it can start setting up the rest of memory.

   ```
   # Now paging is enabled, but we're still running at a low EIP
   # (why is this okay?).  Jump up above KERNBASE before entering
   # C code.
   mov    $relocated, %eax
   jmp    *%eax
   ```

6. **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?**

Challenge! We consumed many physical pages to hold the page tables for the KERNBASE mapping. Do a more space-efficient job using the PTE_PS ("Page Size") bit in the page directory entries. This bit was *not* supported in the original 80386, but is

supported on more recent x86 processors. You will therefore have to refer to Volume 3 of the current Intel manuals. Make sure you design the kernel to use this optimization only on processors that support it!

```
#define PGSIZE_4MB PGSIZE*1024
if(perm & PTE_PS){//4mb paging
	for(offset = 0; offset < size; offset += PGSIZE_4MB) {
		pte = &pgdir[PDX(va)];

		*pte = pa|perm|PTE_P|PTE_PS;

		pa += PGSIZE_4MB;
		va += PGSIZE_4MB;
	}
}
else
{    //4kb paging
    for(offset = 0; offset < size; offset += PGSIZE) {
		pte = pgdir_walk(pgdir, (void*)va, 1);

		*pte = pa|perm|PTE_P;

		pa += PGSIZE;
		va += PGSIZE;
	}

}
```