Lab 3: Memory Management

Due Date : 18th April, 2023 at 11:59pm

General Instructions

1. This lab will **constitute 8%** of your total grade (**Total 230 points**).
2. **Only one** person of each team (of 2) should submit the lab. This should be the same person who had submitted previous assignments and labs.
3. Along with solutions, each team is also required to submit a document outlining detailed contributions of each team member.
4. Please refer to the class webpage for notes on late assignment submission policy.

Introduction

In this lab, you will write the memory management code for your operating system. Memory management has **two components**.

The **first** component is a physical memory allocator for the kernel, so that the kernel can allocate memory and later free it. Your allocator will operate in units of 4096 bytes, called `pages`. Your task will be to maintain data structures that record which physical pages are free and which are allocated, and how many processes are sharing each allocated page. You will also write the routines to allocate and free pages of memory.

The **second** component of memory management is *virtual memory*, which maps the virtual addresses used by kernel and user software to addresses in physical memory. The x86 hardware's memory management unit (MMU) performs the mapping when instructions use memory, consulting a set of page tables. You will modify JOS to set up the MMU's page tables according to a specification we provide.

Getting started

For this lab, we have updated the git repository with relevant code files. First you need to get the latest code from the git repository using the following commands.

GitHub Classroom Link - https://classroom.github.com/a/_zMAczHG
Team Naming Convention - <OLD_TEAM_NAME>-2, so if your team name was **xyz**, and for Lab 2 it was **xyz-1,** for Lab 3 it should be **xyz-2.** Make sure the team members are still the same.

This repository contains the following new source files, which you should browse through:

```
inc/memlayout.h
kern/pmap.c
kern/pmap.h
kern/kclock.h
kern/kclock.c
```

`memlayout.h` describes the layout of the virtual address space that you must implement by modifying `pmap.c`.
`memlayout.h` and `pmap.h` define the `PageInfo` structure that you'll use to keep track of which pages of physical memory are free.
`kclock.c` and `kclock.h` manipulate the PC's battery-backed clock and CMOS RAM hardware, in which the BIOS records the amount of physical memory the PC contains, among other things.
The code in `pmap.c` needs to read this device hardware in order to figure out how much physical memory there is, but **that part of the code is done for you**: you do not need to know the details of how the CMOS hardware works.
Pay particular attention to `memlayout.h` and `pmap.h`, since this lab requires you to use and understand many of the definitions they contain. You may want to review `inc/mmu.h`, too, as it also contains a number of definitions that will be useful for this lab.

Lab Requirements

Write up brief answers to the questions posed in the lab and a short (e.g., one or two paragraph) description of what you did to solve the problem. Place the write-up in a file called answers-lab3.pdf in the top level of your lab directory before handing in your work.

Hand-In Procedure

Submit your code files on the main branch on your team's GitHub Repository. Make sure your code is well-documented, those will help us to evaluate the code and will also serve as guides for your demos. Submit the PDF write-up through Google Classroom.

Part 1: Physical Page Management

The operating system must keep 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.

You'll now write the physical page allocator. It keeps track of which pages are free with a linked list of `struct PageInfo` objects (which, unlike xv6, are *not* embedded in the free pages themselves), each corresponding to a physical page. You need to write the physical page allocator before you can write the rest of the virtual memory implementation, because your page table management code will need to allocate physical memory in which to store page tables.

Exercise 1. In the file `kern/pmap.c`, you must implement code for the following functions (probably in the order given). [25 Points]

`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. State all your assumptions clearly in the pdf.

This lab will require you to do ***a bit of detective work to figure out exactly what you need to do***. This document does not describe all the details of the code you'll have to add to JOS. **Look for comments** in the parts of the JOS source that you have to modify; those comments often **contain specifications and hints**. You will also need to look at related parts of JOS, at the Intel manuals, and perhaps at your class notes.

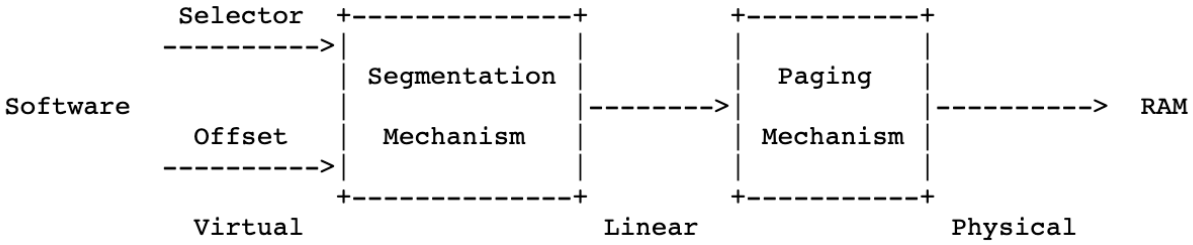
Part 2: Virtual Memory

Before doing anything else, familiarize yourself with the x86's protected-mode memory management architecture: namely *segmentation* and *page translation*.

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 the paging hardware 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.[0 Points]

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.



A C pointer is the "offset" component of the virtual address. In `boot/boot.S`, a Global Descriptor Table (GDT) has been installed that effectively disables segment translation by setting all segment base addresses to `0` and limits to `0xffffffff`. Hence the "selector" has no effect and the linear address always equals the offset of the virtual address. For now, for memory translation, we can ignore segmentation throughout the JOS labs and focus solely on page translation.

Recall that in 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 address space 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 the virtual address space.

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). [20 Points]

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.

The MIT's 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 addresses are mapped and with what permissions.

From code executing on the CPU, 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 and translated by the MMU, which means all pointers in C are virtual addresses.

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.

The JOS kernel can dereference a `uintptr_t` by first casting it to a pointer type. In contrast, the kernel can't sensibly dereference a physical address, since the MMU translates all memory references. If you cast a `physaddr_t` to a pointer and dereference it, you may be able to load and store to the resulting address (the hardware will interpret it as a virtual address), but you probably won't get the memory location you intended.

To summarize:

C type	Address type
T*	Virtual
`uintptr_t`	Virtual
`physaddr_t`	Physical

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;
```

The JOS kernel sometimes needs to read or modify memory for which it knows only the physical address. For example, adding a mapping to a page table may require allocating physical memory to store a page directory and then initializing that memory. However, the kernel cannot bypass virtual address translation and thus cannot directly load and store to physical addresses. One reason JOS remaps all of physical memory starting from physical address 0 at virtual address `0xf0000000` is to help the kernel read and write memory for which it knows just the physical address. 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.
The JOS kernel also sometimes needs to be able to find a physical address given the virtual address of the memory in which a kernel data structure is stored. Kernel global variables and memory allocated by `boot_alloc()` are in the region where the kernel was loaded, starting at `0xf0000000`, the very region where we mapped all of physical memory. 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.

Reference counting

There might be cases in future labs where (often) you might have the same physical page mapped at multiple virtual addresses simultaneously (or in the address spaces of multiple environments). In those cases, you will need to 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 to page table pages.

Be careful when using `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.

Page Table Management

Now you'll write a set of routines to manage page tables: to insert and remove linear-to-physical mappings, and to create page table pages when needed.

Exercise 4. In the file `kern/pmap.c`, you must implement code for the following functions. [25 Points]

```
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.

Part 3: Kernel Address Space

JOS divides the processor's 32-bit linear address space into two parts. User environments (processes), which will be loaded and run, will have control over the layout and contents of the lower part, while the kernel always maintains complete control over the upper part. The dividing line is defined somewhat arbitrarily by the symbol `ULIM` in `inc/memlayout.h`, reserving approximately 256MB of virtual address space for the kernel. This explains why 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. You'll find it helpful to refer to the JOS memory layout diagram in `inc/memlayout.h` both for this part and for later labs.

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 space 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

Now you'll set up the address space above `UTOP`: the kernel part of the address space. `inc/memlayout.h` shows the layout you should use. You'll use the functions you just wrote to set up the appropriate linear to physical mappings.

Exercise 5. 1. Fill in the missing code in `mem_init()` after the call to `check_page()`.

Your code should now pass the `check_kern_pgdir()` and `check_page_installed_pgdir()` checks. [70 Points Total]

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	?	Page table for top 4MB of phys memory
1022	?	?
.	?	?
.	?	?
.	?	?
2	`0x00800000`	?
1	`0x00400000`	?
0	`0x00000000`	[see next question]

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?

4. What is the maximum amount of physical memory that this operating system can support? Why?

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?

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?

Additional Exercises (All of the following need to be solved and submitted)

Exercise 6 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! [30 Points]

Exercise 7 Extend the JOS kernel monitor with commands to: [30 Points]

- 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.)

Exercise 8 Since our JOS kernel's memory management system only allocates and frees memory on page granularity, we do not have anything comparable to a general-purpose malloc/free facility that we can use within the kernel. This could be a problem if we want to support certain types of I/O devices that require *physically contiguous* buffers larger than 4KB in size, or if we want user-level environments, and not just the kernel, to be able to allocate and map 4MB *superpages* for maximum processor efficiency. (See the earlier problem about `PTE_PS`.) [30 Points total]

Generalize the kernel's memory allocation system to support pages of a variety of power-of-two allocation unit sizes from 4KB up to some reasonable maximum of your choice. Be sure you have some way to divide larger allocation units into smaller ones on demand, and to coalesce multiple small allocation units back into larger units when possible. Think about the issues that might arise in such a system.

This completes the lab.