Project 2

Due: TBA

Last updated: 2025/06/02 14:32:01

Update history:

```
2025/06/02: initial version
```

---

1. Introduction

In this assignment, you will add memory management code to 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.

1.1. Code Distribution

As with the first assignment, you will download a code distribution using one of these commands, depending on how you want to retrieve the archive:

```
$ get csci352 project2
$ get csci352 project2.tar
```

The first command unpacks the distribution into a subdirectory named `project2/`; the second retrieves it in a `tar` file named `project2.tar`, which you can copy to another system or unpack yourself.

This assignment's distribution contains several new or modified source files, which you should browse through:

- `inc/memlayout.h` (modified)
- `kern/init.c` (modified)
- `kern/pmap.c`
- `kern/pmap.h`
- `kern/kclock.h`
- `kern/kclock.c`

`memlayout.h` describes the layout of the virtual address space; the additions to the file are related to virtual memory. The `pmap.c` and `pmap.h` files contain functions related to the creation and management of memory-related data structures.

The changes to `init.c` are the removal of the code that performed testing of your project 1 changes (i.e., the `test_backtrace()` function, and the code in `i386_init()` that called it), and the addition of code to test your project 2 changes (a call to `mem_init()` in the `i386_init()` function, and addition of relevant `#include` statements).

`kclock.c` and `kclock.h` manage 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`, as this assignment 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 assignment.

1.2. Requirements

In the sections that follow, you will find a series of questions and exercises enclosed in boxed paragraphs:

Things to do are “to do” lists, and summarize important things you'll need to do (typically, adding or modifying code).

Check your understanding activities are things you should be particularly diligent about doing, as they will often provide you with important information pertaining to what you're trying to accomplish.

Challenges are things you could consider as possible modifications to the kernel. They are “thought experiments” and are not work to be done for this assignment.

You will also find some paragraphs with thin black borders. These contain information that may be particularly important, or they may point you toward other reference material that could help explain architectural or coding details.

You may do this assignment individually, or together, as you prefer. While there is some benefit to working alone on a project such as this, working together gives you the opportunity to “brainstorm” when looking at strange code or when you run into problems as you modify the code base.

At the end of this writeup is a section entitled *Future Work* which contains several “challenge” questions. These are *not* to be turned in; rather, they are to get you thinking about other ways the basic system could be modified or enhanced.

---

2. Assignment Details

2.1. Part 0: Moving from Project 1 to Project 2

Things to do:

To begin, you should insert your work from the first project into the source code for this project. Because the relevant files haven't changed between distributions, this should only require copying the versions from project 1 into the corresponding files in the project 2 distribution.

2.2. Part 1: Physical Memory 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 will begin by writing the physical page allocator. This code keeps track of which pages are free with a linked list of `struct PageInfo` (defined in `memlayout.h`) objects. (Unlike xv6, these objects *not* embedded in the free pages themselves), Each `PageInfo` object corresponds to a physical page in memory. You must 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.

Things to do:

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()`

The functions `check_page_free_list()` and `check_page_alloc()` in this file 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.

All of these assignments will require you to do a bit of detective work to figure out exactly what you need to do. This assignment doesn't describe all the details of the code you'll need to add to JOS. Look for comments in the parts of the JOS source that you must modify; those comments often contain specifications and hints. You will also need to look at related parts of JOS, at the Intel documentation, and the relevant lecture notes.

2.3. Part 2: Virtual Memory

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

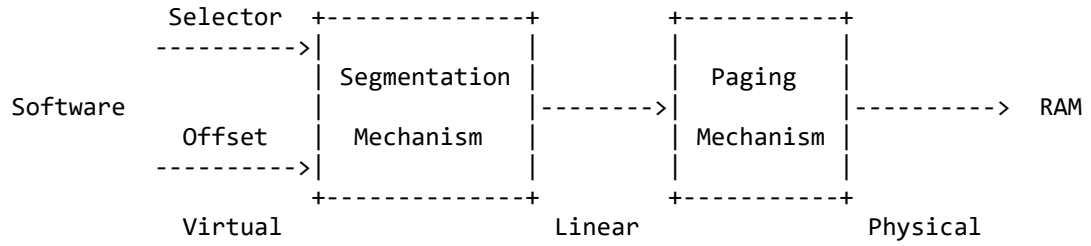
Check your understanding:

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). You should also skim the sections that cover 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.

2.3.1. 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`, we install a Global Descriptor Table (GDT) that effectively disabled 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. Eventually, something a bit more complicated will be required in order to set up privilege levels, but as far as memory translation is concerned, it's safe to ignore segmentation and focus solely on page translation.

In project 1, the code in `kern/entry.S` installs a simple page table so that the kernel can 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 this assignment, this must be expanded 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.

Check your understanding:

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 MIT 6.828 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 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 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 unsigned integers (`uint32_t`), so the compiler won't stop you from assigning one type to another. Because 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, because 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

Check your understanding:

Assuming that the following JOS kernel code is correct, what type should the variable `x` have (`uintptr_t` or `physaddr_t`)?

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

The 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 only 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. The macro `KADDR(pa)` (defined in `kern/pmap.h`) is provided to do this.

The 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. The macro `PADDR(va)` (also defined in `kern/pmap.h`) performs this subtraction.

2.3.2. Reference counting

In “real” operating systems is is often the case that the same physical page is mapped at multiple virtual addresses simultaneously, or is in the address spaces of multiple processes. To keep track of this, a *reference count* is kept for each physical page, which is simply a count of the number of times that physical page is mapped into address spaces.

The `struct PageInfo` declaration in `inc/memlayout.h` contains a field named `pp_ref` specifically for this. 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 (e.g., `page_insert()` might be one such place) and sometimes the function calling `page_alloc()` must do it directly.

2.3.3. 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.

> ***Things to do:***
>
> 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()
> ```
>
> The function `check_page()`, called from `mem_init()`, tests your page table management routines. You should make sure it reports success before proceeding.

2.4. Part 3: Kernel Address Space

JOS divides the processor's 32-bit linear address space into two parts. *User environments* (processes) 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 give the kernel such a high link address: 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 may find it helpful to refer to the JOS memory layout diagram in `inc/memlayout.h`.

2.4.1. Permissions and Fault Isolation

Because 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.

2.4.2. Initializing the Kernel Address Space

The address space above `UTOP` is the kernel part of the address space. See `inc/memlayout.h` for the layout you should use. You'll use the functions you just wrote to set up the appropriate linear-to-physical mappings.

Things to do:

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.

Check your understanding:

- 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]

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

---

3. Turning In Your Solution

Because you are changing many files, rather than having you keep track of those files and submitting them separately, you will submit your entire project directory for this assignment. When you are ready to submit your work, execute this series of commands:

```
$ cd project2
$ make realclean
```

```
$ submit -v wrc-grd is-2 .
```

The first command places you in the project source directory; the second command removes all the "cruft" created by the compilation and linking process, to minimize the size of your submission.

The `submit` command is similar to `try` in that it bundles up the things you want to submit and deposits them in the target account (`wrc-grd`, in this case). Unlike `try`, it will accept entire directory hierarchies, which is what we're doing here. The last command-line argument, a single dot (`'.'`) character, is a reference to the working directory; this command says "bundle up everything that is in or under this directory" as the submission.

The `-v` argument tells `submit` to be verbose - it will tell you the names of the things it's archiving. You can verify your submission with the `cksubmit` command:

```
$ cksubmit -v wrc-grd is-2
```

You'll see an `ls`-style list of all the files that were submitted.

If you are working as a team, each team should submit only *one copy* of its answers. All members of the team will receive the same grade for this assignment.

---

4. Future Work

Here are some ideas for possible enhancements to the system.

4.1. Kernel Address Space

Challenge 1:

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 Intel manuals (PDF). Make sure you design the kernel to use this optimization only on processors that support it!

4.2. Kernel Monitor Extensions

Challenge 2:

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 addresses 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!)

4.3. Address Space Layout Alternatives

The address space layout we use in JOS is not the only one possible. An operating system might map the kernel at low linear addresses while leaving the *upper* part of the linear address space for user processes. x86 kernels generally do not take this approach, however, because one of the x86's backward-compatibility modes, known as *virtual 8086 mode*, is "hard-wired" in the processor to use the bottom part of the linear address space, and thus cannot be used at all if the kernel is mapped there.

It is even possible, though much more difficult, to design the kernel so as not to have to reserve *any* fixed portion of the processor's linear or virtual address space for itself, but instead effectively to allow user-level processes unrestricted use of the *entire* 4GB of virtual address space while still fully protecting the kernel from these processes *and* protecting different processes from each other!

Challenge 3:

Each user-level environment maps the kernel. Change JOS so that the kernel has its own page table and so that a user-level environment runs with a minimal number of kernel pages mapped. That is, each user-level environment maps just enough pages mapped so that the user-level environment can enter and leave the kernel correctly. You also have to come up with a plan for the kernel to read/write arguments to system calls.

Challenge 4:

Write up an outline of how a kernel could be designed to allow user environments unrestricted use of the full 4GB virtual and linear address space. Hint: do the previous challenge exercise first, which reduces the kernel to a few mappings in a user environment. (Hint: the technique is sometimes known as "*follow the bouncing kernel*".) In your design, be sure to address exactly what has to happen when the processor transitions between kernel and user modes, and how the kernel would accomplish such transitions. Also describe how the kernel would access physical memory and I/O devices in this scheme, and how the kernel would access a user environment's virtual address space during system calls and the like. Finally, think about and describe the advantages and disadvantages of such a scheme in terms of flexibility, performance, kernel complexity, and other factors you can think of.

Challenge 5:

Since our 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 challenge problem about `PTE_PS`.)

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.
