CSE 506: Lab 2: Memory Management

Due 11:59 PM, Wednesday, September 17, 2014

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 amd64 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

In this and future labs you will progressively build up your kernel. We will also provide you with some additional source. To fetch that source, use Git to commit your Lab 1 source, fetch the latest version of the course repository, and then create a local branch called `lab2` based on our lab2 branch, `origin/lab2`:

```
kermit% cd ~/CSE506/lab
kermit% git commit -am 'my solution to lab1'
Created commit 254dac5: my solution to lab1
 3 files changed, 31 insertions(+), 6 deletions(-)
kermit% git pull

Already up-to-date.
kermit% git checkout -b lab2 origin/lab2
Branch lab2 set up to track remote branch refs/remotes/origin/lab2.
Switched to a new branch "lab2"
kermit%
```

The **`git checkout -b`** command shown above actually does two things: it first creates a local branch `lab2` that is based on the `origin/lab2` branch provided by the course staff, and second, it changes the contents of your `lab` directory to reflect the files stored on the `lab2` branch. Git allows switching between existing branches using **`git checkout`** ***`branch-name`***, though you should commit any outstanding changes on one branch before switching to a different one.

You will now need to merge the changes you made in your `master` (lab1) branch into the `lab2` branch, as follows:

```
kermit% git merge master
Merge made by recursive.
 kern/kdebug.c  |   11 +++++++++--
 kern/monitor.c |   19 +++++++++++++++++++
 lib/printfmt.c |    7 +++----
 3 files changed, 31 insertions(+), 6 deletions(-)
kermit%
```

In some cases, Git may not be able to figure out how to merge your changes with the new lab assignment (e.g. if you modified some of the code that is changed in the second lab assignment). In that case, the **`git merge`** command will tell you which files are *conflicted*, and you should first resolve the conflict (by editing the relevant files) and then commit the resulting files with **`git commit -a`**.

Lab 2 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.

Hand-In Procedure

When you are ready to hand in your lab code and write-up, note in `slack.txt` how many late hours you have used for this assignment. (This is to help us agree on the number that you have used.) Then run **`make handin`** in the `lab` directory. *If you submit multiple times, we will take the latest submission and count late hours accordingly.*

In this and all other labs, you may complete challenge problems for extra credit. If you do this, please create a file called challenge2.txt, which includes a short (e.g., one or two paragraph) description of what you did to solve your chosen challenge problem and how to test it. If you implement more than one challenge problem, you must describe each one. Be sure to list the challenge problem number. If you complete challenges from previous labs, please list them in this file (not a previous lab challenge file), with both the lab and problem number.

As before, we will be grading your solutions with a grading program. You can run **`make grade`** in the `lab` directory to test your kernel with the grading program. You may change any of the kernel source and header files you need to in order to complete the lab, but needless to say you must not change or otherwise subvert the grading code.

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.

JOS is "told" the amount of physical memory it has by the bootloader. JOS's bootloader passes the kernel a multiboot info structure which possibly contains the physical memory map of the system. The memory map may exclude regions of memory that are in use for reasons including IO mappings for devices (e.g., the "memory hole"), space reserved for the BIOS, or physically damaged memory. For more details on how this structure looks and what it contains, refer to the specification. A typical physical memory map for a PC with 10 GB of memory looks like below.

```
        e820 MEMORY MAP
            address: 0x0000000000000000, length: 0x000000000009f400, type: USABLE
            address: 0x000000000009f400, length: 0x0000000000000c00, type: RESERVED
            address: 0x00000000000f0000, length: 0x0000000000010000, type: RESERVED
            address: 0x0000000000100000, length: 0x00000000dfefd000, type: USABLE
            address: 0x00000000dfffd000, length: 0x0000000000003000, type: RESERVED
            address: 0x00000000fffc0000, length: 0x0000000000040000, type: RESERVED
            address: 0x0000000100000000, length: 0x00000001a0000000, type: USABLE
```

You'll now write the physical page allocator. It keeps track of which pages are free with a linked list of `struct PageInfo` objects, 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.

```
        boot_alloc()
        page_init()
        page_alloc()
        page_free()
```

You also need to add some code to `x64_vm_init()` in `pmap.c`, as indicated by comments there. For now, just add the code needed before the call to `check_page_alloc()`.

You probably want to work on `boot_alloc()`, then `x64_vm_init()`, then `page_init()`, `page_alloc()`, and `page_free()`.

`check_page_alloc()` tests 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.

This lab, and all the CSE 506 labs, will require you to do a bit of detective work to figure out exactly what you need to do. This assignment 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 notes from previous Operating Systems courses.

Part 2: Virtual Memory

Before doing anything else, familiarize yourself with the AMD64's long-mode memory management architecture: namely *segmentation* and *page translation*.

> **Exercise 2.** Read chapters 4 and 5 of the AMD64 Architecture Programmer's Reference Manual, if you haven't done so already. Read the sections about page translation and page-based protection closely (5.1). Although JOS relies most heavily on page translation, you will also need a basic understanding of how segmentation works in long mode to understand what's going on in JOS.

Virtual, Linear, and Physical Addresses

In AMD64 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. Be sure you understand the difference between these three types or "levels" of addresses!

```
           Selector  +--------------+         +-----------+
          ---------->|              |         |           |
                     | Segmentation |         |  Paging   |
Software             |              |-------->|           |---------->  RAM
            Offset   |  Mechanism   |         | Mechanism |
          ---------->|              |         |           |
                     +--------------+         +-----------+
            Virtual                   Linear                Physical
```

A C pointer is the "offset" component of the virtual address. In `kern/bootstrap.S`, we installed 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. In lab 3, we'll have to interact a little more with segmentation to set up privilege levels, but as for memory translation, we can ignore segmentation throughout the JOS labs and focus solely on page translation.

> **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.

From code executing on the CPU, once we're in protected/long mode, 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 virtual addresses, and `physaddr_t` represents physical addresses. Both these types are really just synonyms for 64-bit integers (`uint64_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;
```

In Part 3 of Lab 1 we noted that the kernel's first step is to set up simple segmentation and paging (in kern/bootstrap.S) so that the kernel runs at its link address of 0x8004100000, even though it is actually loaded in physical memory just above the ROM BIOS at 0x00100000. In other words, the kernel's *virtual* starting address at this point is 0x8004100000, but its *physical* starting address is 0x00100000. The kernel's virtual and linear addresses are same because of the flat segmentation hardware in AMD64, while its linear and physical addresses differ because of the paging hardware (Remember we mapped the upper 256 MB 0xf000000 through 0xffffffff back to 0x0 through 0xfffffff)

However, the JOS kernel sometimes needs to read or modify memory for which it only knows 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, like any other software, cannot bypass virtual memory translation and thus cannot directly load and store to physical addresses. One reason JOS remaps of all of physical memory starting from physical address 0 at virtual address 0x8004000000 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 0x8004000000 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. The kernel addresses its global variables and memory that `boot_alloc()` allocates, with addresses in the region where the kernel was loaded, starting at 0x8004000000, 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 0x8004000000. You should use `PADDR(va)` to do that subtraction.

Reference counting

In future labs you will often have the same physical page mapped at multiple virtual addresses simultaneously (or in the address spaces of multiple environments). You 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 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.

```
        pml4e_walk()
        pdpe_walk()
        pgdir_walk()
        boot_map_region()
        page_lookup()
        page_remove()
        page_insert()
```

`page_check()`, called from `x64_vm_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 linear address space into two parts. User environments (processes), which we will begin loading and running in lab 3, 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 linear (and therefore virtual) address space for the kernel.

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 amd64 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.

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

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. Fill in the missing code in `x64_vm_init()` after the call to `page_check()`.

Your code should now pass the `check_boot_pml4e()` check.

Question

2. What entries (rows) in the page directory have been filled in at this point for the 4th page directory pointer entry(Make sure you understand why 4th pdpe entry)? 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):
511	?	Page table for top 2MB of phys memory
510	?	?
.	?	?
.	?	?
.	?	?
184	0xF0000000	?
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. Read the simple page table setup code in `kern/bootstrap.S`.
The bootloader tests whether the CPU supports long (64-bit) mode. It initializes a simple set of page tables for the first 4GB of memory. These pages map virtual addresses in the lowest 3GB to the same physical addresses, and then map the upper 256 MB back to the lowest 256 MB of memory. At this point, the bootloader places the CPU in long mode. Note that our bootloader transitioning to long mode isn't strictly necessary; typically, a bootloader only runs in long mode to load a 64-bit kernel at a high (>4 GB) virtual memory address.
Note that once we transfer control to the kernel, the kernel assumes the CPU supports 64-bit mode. Assuming the kernel was loaded in the lower 4GB of virtual address space, the kernel itself could test whether the CPU supports long mode and determine dynamically whether to run in 64 or 32-bit mode. Of course, this would substantially complicate the boot process.

Challenge 1! (10 bonus points) 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. You might want to refer to AMD64_Architecture_Programmers_Manual.pdf.

Challenge 2! (1 bonus point each, up to 5 points) 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!)

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 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! (10 bonus points) 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: 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 4! (10 bonus points) 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 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.

This completes the lab. Type **`make handin`** in the `lab` directory.

Last updated: 2014-09-03 20:05:45 -0400 [validate xhtml]