6.828 Fall 2007 Lab 2: Memory Management

Handed out Wednesday, September 12, 2007
Due Thursday, September 27, 2007

Introduction

In this lab, your will write the memory management code for your operating system. Memory management is comprised of two components.

The first component is managing the physical memory of the computer so that the kernel can allocate and free physical memory as needed. The x86 divides physical memory into 4096-byte regions 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*, where we set up the PC'S Memory Management Unit (MMU) hardware to map the virtual addresses used by software to physical addresses. You will modify JOS to set up virtual memory mappings according to a specification we provide.

Getting started

Download the code for lab 2 from http://pdos.lcs.mit.edu/6.828/2007/labs/lab2/lab2-handout.tar.gz and untar it into your 6.828 directory, just as you did for lab 1. You will then need to merge the changes between our lab 1 and lab 2 source code trees into your own kernel code resulting from completing lab 1.

In this and future labs you will progressively build on this same kernel. With each new lab we will hand out a source tree containing additional files and possibly some changes to existing files. Compare the new source tree against the one we provided for the previous lab in order to figure out what new code you need to incorporate into your kernel. You may find it useful to keep a "pristine" copy of our source tree for each lab around along with your modified versions. You should expect to become intimately familiar with the Unix `diff` utility if you aren't already, and `patch` can be highly useful as well. "Diff-and-merge" is an important and unavoidable component of all real OS development activity, so any time you spend learning to do this effectively is

time well spent.

One option is to just merge in your changes manually. If you remember what functions you modified, you can copy the changes into the lab2 code. To actually see what changes you made, and try to patch them in to the code, run the following sequence of commands. Be warned that these utilities are not perfect, and merging in the changes by hand may be simpler.

```
cd ~/6.828

# this creates a tar of what you handed in, for backup purposes
tar czvf lab1-handin.tar.gz lab1

mkdir given-code
cd given-code
tar xzf ../lab1.tar.gz
cd ..
mv given-code/lab1 lab1-unchanged

# now we have the handed out lab1 code in lab1-unchanged

diff -r -u lab1-unchanged lab1 > lab1-changes.txt

# It is very important to look at the patch file.  All of the changes
# in it should be for code that you added to lab 1 and want to bring
# to lab 2.  If there are other changes (like changes to the
# makefiles), then you should NOT run the 'patch' command below.
# Instead, you should apply the patch by hand.  If you decide to apply
# it with patch, then run the commands below.

cd lab2
patch -p1 -u < ../lab1-changes.txt

# if any chunks failed, look at the rejects
# files (.rej) and merge those changes in yourself.
```

Anyone serious about software development should consider using a source code management system like SVN or CVS. The NYU version of this class has some potentially useful instructions on setting up a CVS repository. The course staff is a big fan of CVS, by the way.

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
- kern/kdebug.h

- kern/kdebug.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 Page 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. The last two files provide support for extending the kernel monitor.

Pay particular attention to memlayout.h and pmap.h, since this lab requires you to use and understand many of the definitions they contain.

Lab Requirements

In this lab and subsequent labs, do all of the regular exercises described in the lab and *at least one* challenge problem. (Some challenge problems are more challenging than others, of course!) Additionally, 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 your chosen challenge problem. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called answers.txt (plain text) or answers.html (HTML format) in the top level of your lab2 directory before handing in your work.

Hand-In Procedure

When you are ready to hand in your lab code and write-up, run gmake handin in the lab2 directory. This will first do a gmake clean to clean out any .o files and executables, and then create a tar file called lab2-handin.tar.gz with the entire contents of your lab2 directory. You should submit the tar file here.

As before, we will be grading your solutions with a grading program. You can run gmake grade in the lab2 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.

> **Exercise 1.** Modify your stack backtrace function to display, for each EIP, the function name, source file name, and line

number corresponding to that EIP. To help you we have provided `debuginfo_eip`, which looks up `eip` in the symbol table and is defined in `kern/kdebug.c`.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- run `i386-jos-elf-objdump -h obj/kern/kernel`
- run `i386-jos-elf-objdump -G obj/kern/kernel`
- run `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, and look at init.s.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

Extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
Stack backtrace:
kern/monitor.c:74: mon_backtrace+10
  ebp f0119ef8  eip f01008ce  args 00000001 f0119f20 00000000 00000000 2000000a
kern/monitor.c:143: monitor+10a
  ebp f0119f78  eip f01000e5  args 00000000 f0119fac 00000275 f01033cc fffffffc
kern/init.c:78: _panic+51
  ebp f0119f98  eip f010133e  args f01033ab 00000275 f01033cc f0103473 f01030bc
kern/pmap.c:711: page_check+9e
  ebp f0119fd8  eip f0100082  args f0102d20 00001aac 000006a0 00000000 00000000
kern/init.c:36: i386_init+42
  ebp f0119ff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
```

The `read_eip()` function may help with the first line. You may find that the some functions are missing from the backtrace. For example, you will probably see a call to `monitor()` but not to `runcmd()`. This is because the compiler in-lines some function calls. Other optimizations may cause you to see unexpected line numbers. If you get rid of the `-O2` from `GNUMakefile`, the backtraces may make more sense (but your kernel will run more slowly).

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 Page` 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 managment code will need to allocate physical memory in which to store page tables.

> **Exercise 2.** 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 `i386_vm_init()` in `pmap.c`, as indicated by comments there. For now, just add the code needed before the call to `check_page_alloc()`.
>
> `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 own assumptions are correct.

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 3.** Read chapters 5 and 6 of the Intel 80386 Reference Manual, if you haven't done so already. Although JOS relies most heavily on page translation, you will also need a basic understanding of how segmentation works in protected mode to understand what's going on in JOS.

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

> **Exercise 4.** Review the debugger section in the Bochs user manual, and make sure you understand which debugger commands deal with which kinds of addresses. In particular, note the various `vb`, `lb`, and `pb` breakpoint commands to set breakpoints at virtual, linear, and physical addresses. The default `b` command breaks at a *physical* address. Also note that the `x` command examines data at a *linear* address, while the command `xp` takes a physical address. Sadly there is no `xv` at all.

The JOS kernel source uses different type names for virtual and physical addresses: the type `uintptr_t` represents 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! Every pointer value in JOS should be a virtual address (once paging is set up), since only virtual addresses can be dereferenced: kernel memory references go through the paging hardware. To summarize:

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

> **Question**
>
> 1. Assuming that the following JOS kernel code compiles correctly and doesn't crash, 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 boot loader sets up the x86 segmentation

hardware so that the kernel appears to run at its link address of 0xf0100000, 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 0xf0100000, but its *linear* and *physical* starting addresses are both 0x00100000. The kernel's linear and physical addresses are the same because we have not yet initialized or enabled page translation.

In the virtual memory layout you are going to set up for JOS, we will stop using the x86 segmentation hardware for anything interesting, and instead start using page translation to accomplish everything we've already done with segmentation and much more. That is, after you finish this lab and the JOS kernel successfully enables paging, linear addresses will be the same as (the offset portion of) the kernel's *virtual* addresses, rather than being the same as physical addresses as they are when the boot loader first enters the kernel.

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.

In future labs you will often map the same physical page at multiple virtual addresses (or in the address spaces of multiple environments), so you will keep a count of the number of times each physical page is mapped in the corresponding `Page`'s `pp_ref`. The count should be equal to the number of pointers to the page in the page table(s) *below* `UTOP` (The mappings above `UTOP` are mostly just set up at boot time by the kernel and are not tracked by the reference-counting system). When the count goes to zero, the physical page can be freed.

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

```
pgdir_walk()
boot_map_segment()
page_lookup()
page_remove()
page_insert()
```

`page_check()`, called from `i386_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 32-bit 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. 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.

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 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. We do this as follows.

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 6.** Fill in the missing code in `i386_vm_init()` after the call to `page_check()`.
>
> Your code should now pass the `check_boot_pgdir()` check.

> Answer these questions:
>
> 1. 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?]

2. After `check_boot_pgdir()`, `i386_vm_init()` maps the first four MB of virtual address space to the first four MB of physical memory, then deletes this mapping at the end of the function. Why is this mapping necessary? What would happen if it were omitted? Does this actually limit our kernel to be 4MB? What must be true if our kernel were larger than 4MB?
3. (From Lecture 4) 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?

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!

Note: If you compiled bochs yourself, be sure that the appropriate configuration options were specified. By default bochs does not support some extended page table features.

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!)

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! 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! 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.)
Note: If you compiled bochs yourself, be sure that the appropriate configuration options were specified. By default bochs does not support some extended page table features.

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.

Challenge! Extend the JOS kernel monitor with commands to allocate and free pages explicitly, and display whether or not

any given page of physical memory is currently allocated. For example:

```
K> alloc_page
        0x13000
K> page_status 0x13000
        allocated
K> free_page 0x13000
K> page_status 0x13000
        free
```

Think of other commands or extensions to these commands that may be useful for debugging, and add them.

This completes the lab. Type `gmake handin` in the `lab2` directory and then upload `lab2-handin.tar.gz` here.

Version: $Revision: 1.11 $. Last modified: $Date: 2007/09/11 18:42:09 $