CS162: Operating Systems **Fall 2005**

Section: Address Translation

TA: Rajesh Nishtala 10/12/05

1 Announcements

- Midterm!
- Project 2 Initial Design doc due on monday
- carefully write out your designs so that you have a working project
- group issues ... make an appointment with me. I'd rather not use design review time to sort out that stuff
- admin questions?
- won't be handing out solutions for project 1 because we don't want solutions in the open, however all you need working are join, alarm, and communicator to move on. If you didn't pass all the auto-grader tests on those, then see me and this time i can help you catch the java bugs in those functions. And make sure that you design well this time! Hopefully my harsh grades gave you an idea of how to write a design document

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

2 Virtual Memory

Lets go back to one of the roles of an operating system before. The operating system plays the role of an illusionist. One of these illusions is to make each process think it has access to the full memory space and that at 0 and ending at $2^{3}2 - 1$. In this section we'll see how the OS can fulfill this illusion and what mechanisms it can use.

2.1 Segmenting

One of the easiest things that the OS can do is segment the physical memory. Figure 1 shows an example of how segmentation works. I think the book and the lecture notes give a good description of what segmentation is so i won't go into it in much detail here. I'll just go through the quick example. Notice here that process A and process B both fit in the memory space easily. And in order to find out where to index into we simply take the

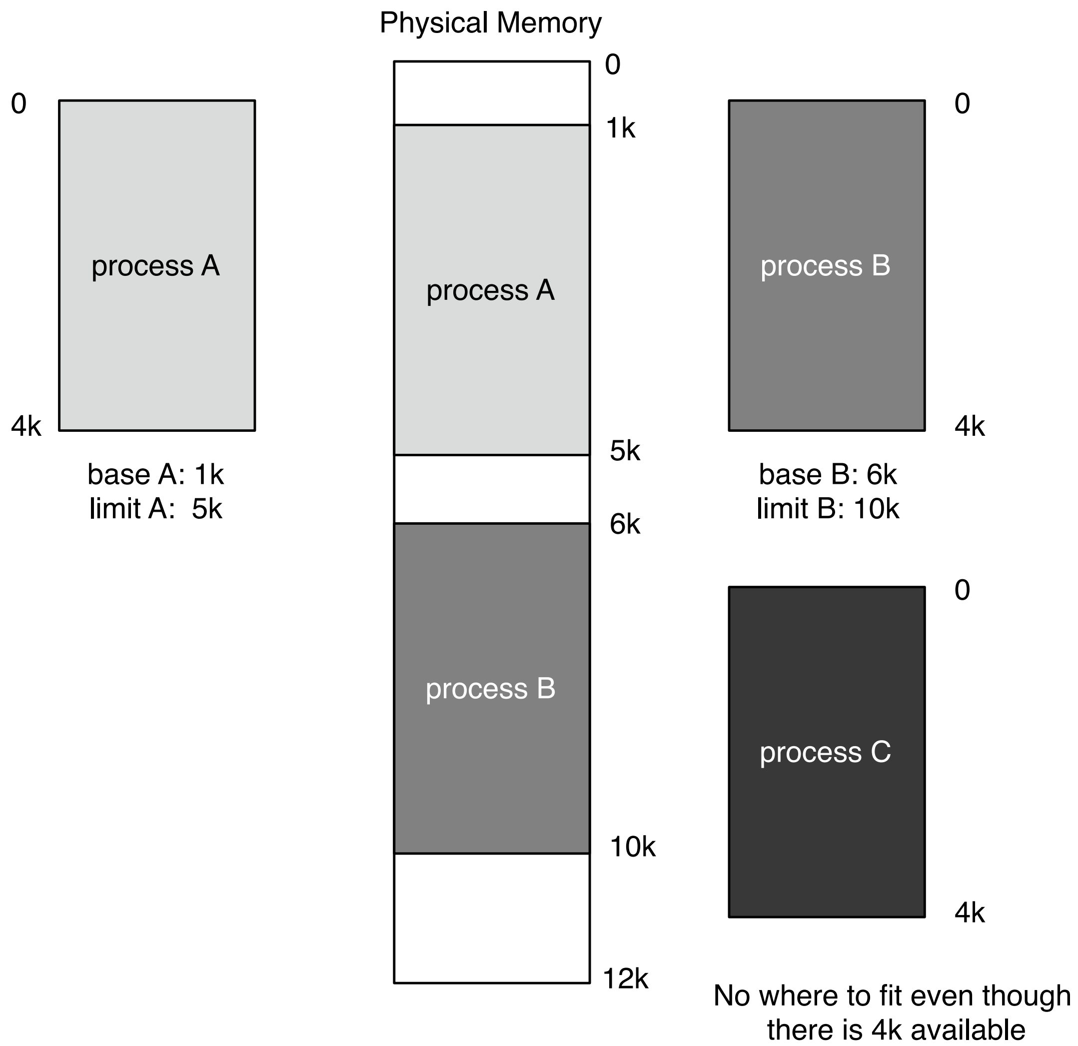

Figure 1: Segmentation Example

virtual address add in the base and ... tada ... we have the physical address. If the address that we want is less than the limit then we have a valid address.

Question: Why do we need the limit register? What does it prevent us from doing?

However one of the main problems with this scheme is that its fragmented (also termed external fragmentation) the memory space pretty badly. Notice that there is 4k of available physical memory but, since its not in a contiguous chunk we can't set up the base and limit registers. A way around this would be to migrate the entire memory space through a copy operation but that'll get really expensive for large memory spaces and many processes and large memory sizes. Another option is to keep around multiple segments but again this gets expensive because we now have an $O(n)$ lookup to see what segment to use where n is the number of segments. We want to have an $O(1)$ access. So even though segments are simple, they don't scale well at all. So we need to come up with a better solution.

2.2 Paging

To fix the problems of external fragmentation described above we create the idea of pages. Pages are fixed units of memory that the operating system. Typical page sizes for consumer OSes range anywhere from 1k through 16k. Paging allows us to break a process's address space into chunks (i.e. pages) and spread those chunks (i.e. pages) around the system. Again I think the book and the lecture notes give a good intro into the idea of paging and what pages are so I won't go into to much detail. The concept of paging fixes the external fragmentation problem from before since we can interleave the pages a lot easier but we still get the problem of a page size being too large.

Since the operating system only deals in pages, if the page size is too large and we don't use all the data then there will still be wasted space. This is termed internal fragmentation. In the worst case lets say we require 1k+1 bytes of memory to run our application. Using segmentation this is easy to setup with base and limit registers. However with paging we need to allocate a full two pages (assuming 1k pages) to satisfy the request leaving one page almost blank. So as we've harped on many times, there is a design tradeoff here. The more pages (and hence the smaller page size you have) the less likely you'll have internal fragmentation, but you need more indexing overhead to access the memory. The less pages and hence larger page sizes) you have the more likely you'll have internal fragmentation but your indexing overhead drops.

Question: What are the advantages of large pages? What types of applications can use large pages?

Question: What are the advantages of small pages? What types of applications can use small pages?

2.3 One-level Page Table and Address Translation

Once we have partitioned the memory space into pages we need some way of finding what physical pages correspond to what virtual address space. Our translation algorithm needs to take a virtual address that is in the range of 0 to $2^{32} - 1$ and translate that into an

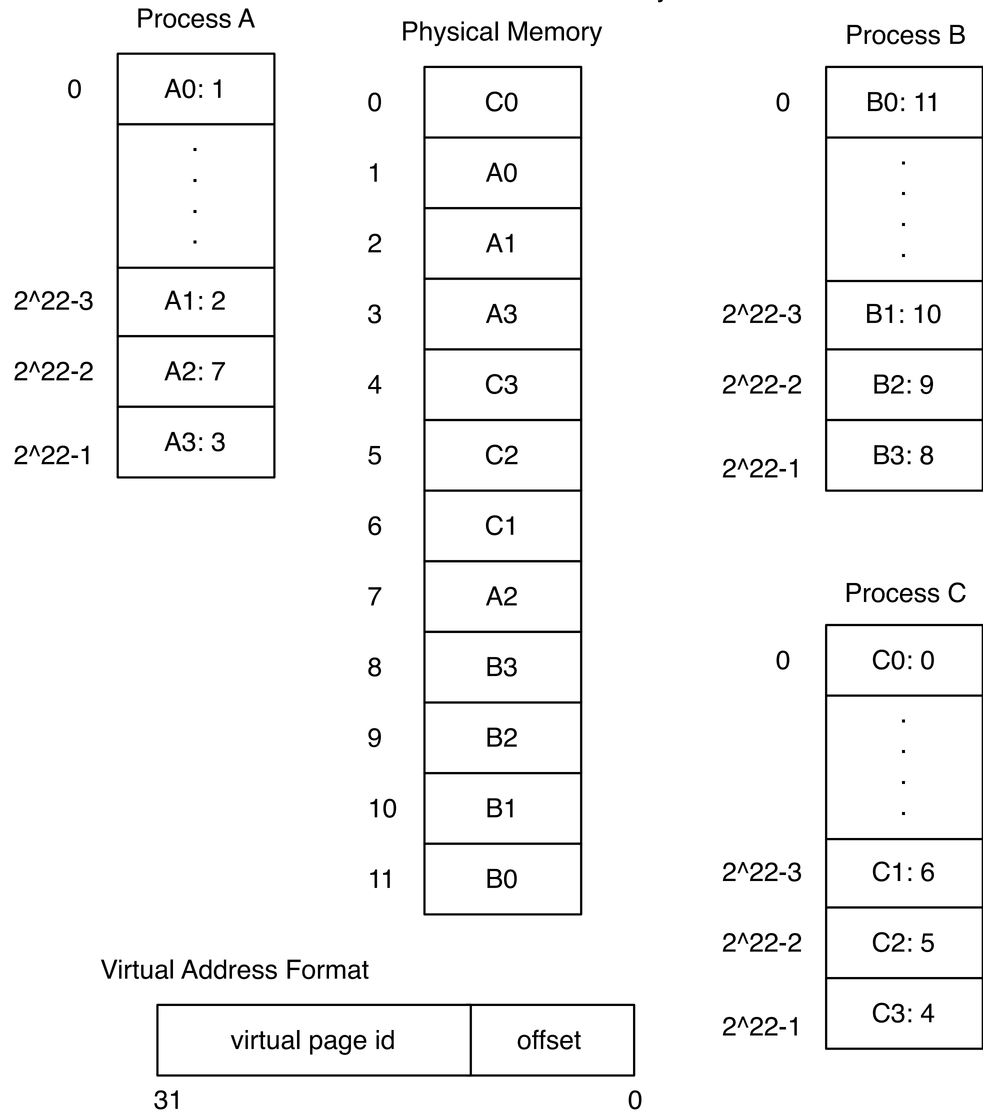

Figure 2: One Level Page Table Example

address into the physical memory. We first break our address into two regions, an offset and the virtual page ID. We then take the virtual page ID to index into the page table to get the physical page ID and attach the physical page ID to the offset and we get our physical address. Notice here that every process has its own translation table. Figure 2 shows an example of one-level page tables.

Question: If the page size is 1k, how many bits are in the offset? How many indices are in the Page Table?

Question: Why can we simply just tack on the offset to the page ID and not need to do any other complex arithmetic?

Question: Why does this scheme ensure that processes can't access regions of memory that aren't allocated to them?

Question: How can we get two processes to share a page?

Again here i encourage you to read the book and the lecture notes on the subject. One of the fundamental drawbacks is that notice that if the pages are 1k then we need $2^{22} \sim 4$ million entries in the page table to handle the index alone. Thats a lot of memory and it doesn't scale! If your system has over 100 processes (which is not uncommon for home machines), you can spend 1.6 *gigabytes* trying to hold this index.

2.4 Two-level Page Table and Address Translation

In order to fix the problem of the indices get too large we create multi-level page tables. Figure 3 shows an example of how we can construct a two level page table. The book and the lecture notes explain this idea very well so I won't go into this too much. However think about the sizes of the different tables. It is important to remember that the page tables are themselves in memory so they must fit in page sizes to be able for us to easily handle different pages. Another main advantage of the two level table is that it allows us to only look at indices of pages that have been allocated instead of keeping the entire table in the memory. Notice that in our one-level page table example above, most of that table is empty and just wasted space. The two level scheme allows us to only store the parts of this index (i.e. the page tables) that point to valid data and not even allocate the rest of it creating a huge savings. I encourage you to work it out using both the examples. Keep in mind all this address translation and segmenting the address is identical to the stuff you learned with caches in 61C. This is a common and big idea of computer science (especially in architecture and systems)

Question: Assuming that each page table entry is 4 bytes, why are there 2^8 entries in each page table? Why are there 2^{14} entries in the segment table?

Exercise: *How many physical pages need to be resident with the one-level page table to access one virtual data page? How many physical pages need to be resident with the two-level page table to access one virtual data page? How much is the difference?*

Question: What needs to be saved on a context switch and what needs to be restored? this question is a bit tricky be careful

Question: How many memory access are needed to translate a virtual address in the one-level scheme? In the two-level scheme? in an n-level scheme?

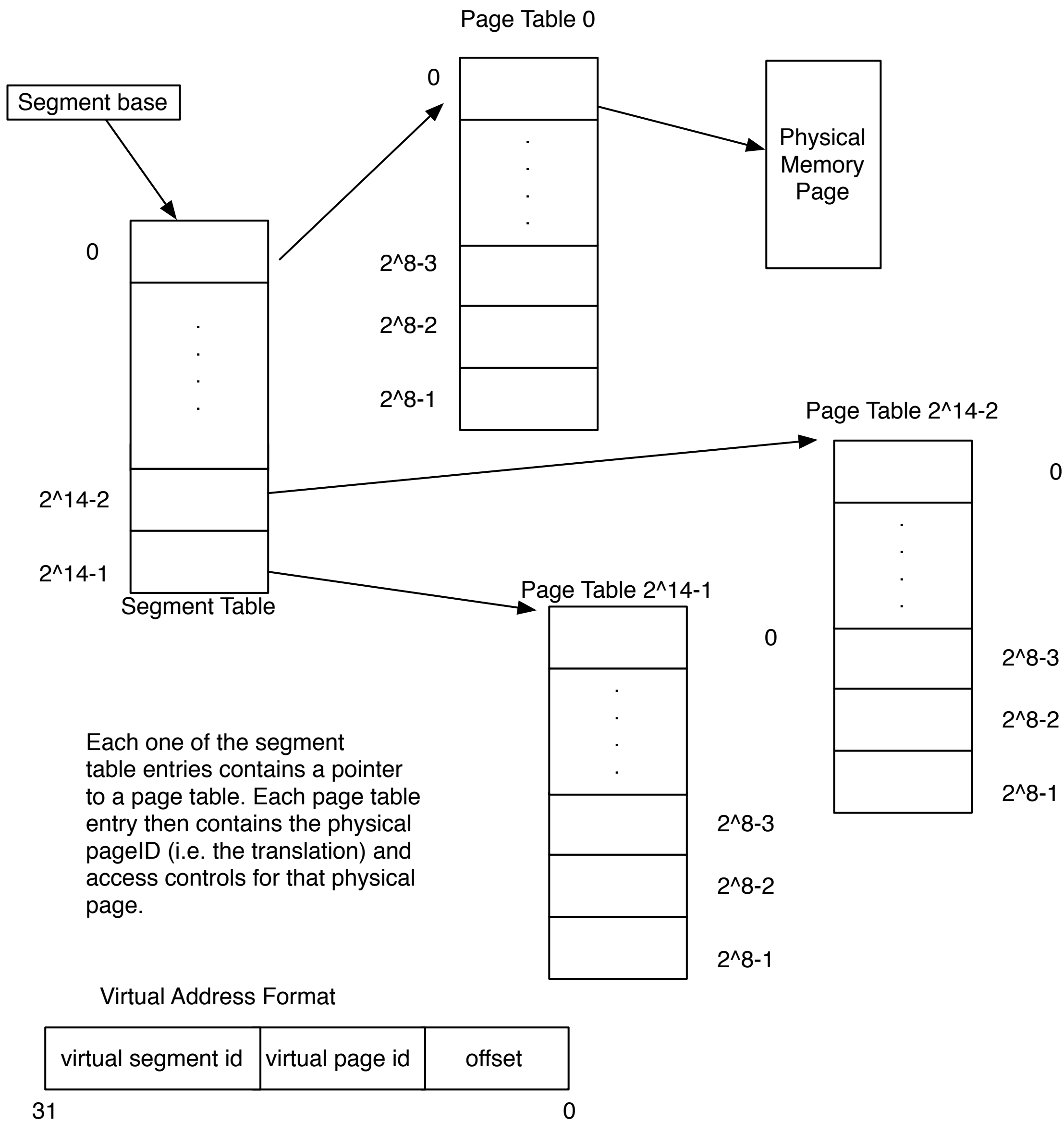

Figure 3: Two Level Page Table Example

Question: One can argue that the segment table above (with 2^{14} entries) is still too large and still very sparse. How can we easily fix that?

3 Nachos Project 2 and System Calls

Part 2 of nachos will be running user code on top of our kernel. All the user code is written in C and thus will be run using the MIPS simulator in the Machine class. We will provide a MIPS/C cross compiler to be able to compile test programs and run them on the nachos kenrel (aren't we nice that we don't make you write assembly). In order to get this to work properly we need syscalls which are mechanisms such as exec and join which allow the user programs to fork off new processes. Remember that all a shell does is exec a new process and let it start running and return to the command line.

3.1 Multi-programming

Multi-programming means that we are going to build in support for running multiple processes at once in Nachos. UserProcess.java and UserThread.java in the userprog folder are important to look at for this phase. In order to do this we need to first of all create the page tables and share the physical memory (similar to what we talked about earlier). Its up to you guys to figure how you want to set that up. Leverage the TranslationEntry.java class in machine. It'll save you lots of time and headaches.

3.2 System Call

A system call is a request by the user processes to do a privileged operation like opening a file on the disk. The user process tells the kernel "I need this specific thing done and please take care of it for me." The kernel then runs that specified piece of code that the user asks in Kernel mode and returns the result. System calls in some ways are like standard function calls in that you jump to a different portion of the code and you run some stuff and return the result. However unlike user functions system calls are predefined by the OS and are in specified locations (usually accessed through some sort of lookup table). Because they are ALL predefined the implementation can be done by the kernel writer, and thus we can assume that they are safe.

3.2.1 File System System Calls

The first set of system calls that we ask you to write are the file system system calls. Look at UserProcess.java to see how the machine handles system calls. We need to augment this list with creat, open, close, read, write, and unlink. Implementing the system calls will allow the users to work with files. Take a look at the different c routines to see how they'd use it. Again i think office hours are the best place to go into more detail.

3.2.2 Syscalls to handle Process Creation

The next set of syscalls that we want to handle are the process creation and process handling ones. Think back to how you spawned threads in Project 1. Its a similar problem here in

that we need to make sure that you spawn the processes of properly and that they don't interfere with each other. Again rather than go into detail here ... come to office hours.

3.3 Lottery Scheduler

Lottery scheduler is a fun problem. The motivation is as follows: in the priority scheduler, low priority threads never ran because high priority threads always took precedence. However now we create a lottery scheme so that high priority threads will statistically run most of the time, but low priority threads will get some time to run and there to make forward progress.

The idea is based on allocating tickets. The number of tickets is directly proportional to the priority. Everytime we want to schedule a new thread, we pick a number at random and the thread with the winning ticket gets to run. The threads that have more tickets have a higher chance but every now and then a low priority thread gets to run. Figuring out the appropriate ticket allocation scheme is up to you guys.

Question: Priority Inversion is still a problem here .... how do we solve it?

3.4 General Project Advice

I personally think that this project is easier than the last one. The tricky part is making sure that you use the VM system efficiently and get paging involved. Another weird part is that you need to bullet-proof the kernel. This means that regardless of how stupid the user programs get, the Kernel must not crash. You should test for this and have the user programs do really weird and dumb things and make sure the kernel is fine. (hint... this means tons and tons of error checking of arguments) This can be tricky because you have to handle all the corner cases. The lottery scheduler is fun and I think using random numbers to decide a course of events is again another common idea in systems programming. If you had a working priority scheduler from before it should be a straightforward extension, since you probably understand what it means to donate priority and how the chaining needs to happen. If not, see me.

4 Midterm Questions!

Fire away!