CMPSCI 377 Operating Systems **Fall 2008**

Lecture 15: October 23

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

15.1 Paging and Virtual Memory

In modern operating systems, applications do not access physical memory directly; instead, they make use of *virtual memory.* Virtual memory addresses start at 0, and each application has the illusion of having the whole address space to itself. Notice that while the programs manage memory at the level of bytes, the OS manages the memory in *groups* of bytes, called pages. This makes it easier to manage the whole memory: think of Tetris with all squares. Both the virtual and the physical memory are treated this way (ie, by being divided into pages), except that when we refer to pages in the physical memory, we call them frames.

Every virtual page has a virtual address, which is mapped to a physical address in the RAM. When some application needs to access a virtual page, the corresponding virtual address is translated into a "real" physical address, and then the actual data might be read and/or written[1]. Notice that some virtual pages do not map to any actual physical memory address, because not all virtual pages are necessarily being used at any given time; these unused virtual pages are called *invalid.* If a program ever tries to access an invalid page, it segfaults.

So far we only know that virtual memory is somehow mapped into physical memory. But why do we need virtual memory? Some of the reasons for using it are the following: *1)* because we want each individual program to have the illusion of having the whole address space for itself; and *2)* because we want the address space being used by each process to be isolated from other applications, ie, we don't want applications to be able to damage other applications. Notice that since each program thinks it has the whole memory to itself, programs can use *a lot* of virtual memory; in fact, a computer might use huge amounts of virtual memory, much more than the amount of actual physical memory available. Of course this is no magic trick; having much more virtual memory than actual physical RAM only works because, in practice, processes use relatively small amounts of RAM. If, however, all processes suddenly decide to use more virtual memory than can be mapped into physical memory, then we need to use some other mechanism, such as disk swapping[2].

A typical virtual memory layout includes a stack, a mmap region, a heap, and the code of the program. Since mmap and heap grow into each other's direction, in the worst case they could smash into each other. The same could happen with the stack: if a program recurses for a large number of times, the stack could grow over the mmap region, and in this case something bad and unpredictable will happen.

15.1.1 The Memory Management Unit

When a program issues a memory load or store operation, the virtual addresses (VAs) used in those operations have to be translated into "real" physical memory addresses (PAs). It is presisely this type of translation that the MMU (Memory Management Unit) does. The MMU maintains a *page table* (big hash table) that maps VAs into PAs. Since memory accesses are happening all the time, the MMU is designed to be *really*

[1]Notice that although arrays are contiguous in virtual memory, parts of it can be mapped into non-contiguous physical memory addresses.

[2]to be mentioned later on.

fast[3]. Notice, however, that we can't map every single byte of virtual memory to a physical address; that would require a huge page table. Instead, the MMU maps virtual pages to physical pages. Also, since we want to isolate each program's address space from other application's address spaces, the MMU must keep a separate page table for each process; this implies that multiple different processes could eventually use the same virtual address to refer to some data, and that would not be a problem since these addresses would be mapped into different physical addresses. The page table also marks all virtual pages that are allocated (and therefore are being used), so that it is possible to know which ones pages have valid mappings into PAs. All virtual pages that are not being mapped into a physical address are marked as being *invalid*; segfaults occur when a program tries to reference or access a virtual address that is not a valid. Besides valid bits, entries in the page table also store lots of other information, such as "read" and "write" bits to indicate which pages can be read/written.

15.1.2 Virtual addresses

Virtual addresses are made up of two parts: the first one contains a page number, and the second one contains an offset inside that page. Suppose our pages are 4kb (4096 bytes) long, and that our machine uses 32 bit addresses. Then we can have at most 2^{32} addressable bytes of memory; therefore, we could fit at most $\frac{2^{32}}{2^{12}} = 2^{20}$ pages. This means that we need 20 bits to address any single page; in other words, our virtual address will be split up into two parts. The first part will be formed by its 20 most significative bits, which will be used to address an entry in the page table; also, the $32 - 20 = 12$ less significant bits of the VA will be used as an offset inside the page. Of course, with 12 bits of offset we can address $2^{12} = 4096$ bytes, which is exactly what we need in order to address every byte inside our 4kb pages.

Now suppose that we have one such page table per process. A page table with 2^{20} entries, each entry with, say, 4 bytes, would require 4mb of memory! This is somehow disturbing because a machine with 80 processes would need more than 300 megabytes just for storing page tables! The solution to this dilemma is to use *multi-level page tables*; this approach allows page tables to point to other page tables, and so on. Suppose a 1-level system; in this case, each virtual address can be divided into something like this: an offset (10 bits), a level-1 page table entry (12 bits), and a level-0 page table entry (10 bits). Then if we read the 10 most significant bits of a virtual address, we obtain an entry index in the level-0 page; if we follow the pointer given by that entry, we get a pointer to a level-1 page table. The entry to be accessed in this page table is given by the next 12 bits of the virtual address. We can again follow the pointer specified on that level-1 page table entry, and finally arrive at a physical page. The last 10 bits of the VA address will give us the offset within that PA page.

A drawback of using this hierarchical approach is that for every load or store instruction we have to perform several indirections, which of course makes everything slower. One way to minimize this problem is to use something called Translation Lookaside Buffer (TLB); the TLB is a fast, fully associative memory that caches page table entries. Tipically, TLBs can cache from 8 to 2048 page table entries.

Finally, notice that if the total virtual memory in use (ie, the sum of the virtual memory used by all processes) is larger than the physical memory, we could start using RAM as a cache for disk. In this case, disk could used to store memory pages that are not being used or that had to be removed from RAM to free space for some other needed page, which itself had been moved to the disk sometime in the past. This approach obviously requires locality, in the sense that the whole set of working pages must fit in RAM. If it does not, then we will incur in a lot of disk accesses; in the worst case, these accesses could cause thrahshing, ie, the system doing nothing except a lot of disk reads and writes.

[3]In fact, it is generally implemented in hardware.

15.1.3 A Day in the Life of a page

Suppose your program allocates some memory with malloc; what happens in this case is that the allocator gives part of a memory page to your program. The OS then updates the corresponding page table in order to mark the virtual address of that page as *valid*; by doing so, it will later on be able to tell that that page is indeed being mapped to a real physical page. Everytime that a virtual address is modified, both the corresponding virtual and physical pages are marked as being *dirty.* If the OS ever needs to evict the page where the data is, it has to copy that page to disk (ie, it *swaps it*); then, that page is marked valid, non-resident, non-readable and non-writable. If we ever touch that page again (ie, if we try to read or write it), the OS may have to evict some other page in order to bring our page back to RAM. One obvious implication of this is that page faults are slow to resolve, since disk accesses are performed. Thus, one possible optimization is for the OS, when idle, to write dirty pages to disk; by doing so, when it comes the time to really evict those pages, the OS won't have to write them to disk, and therefore will be able to kick them out much faster.

15.2 Sharing

Paging allows sharing of memory across processes, since memory used by a process no longer needs to be contiguous. This shared code must be reentran, that means the processes that are using it cannot change it (e.g. no data in reentrant code). Sharing of pages is similar to the way threads share text and memory with each other. The shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address. The OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program. Sharing of pages across processes can greatly reduce overall memory requirements for commonly used applications.