Lecture 14: March 9

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

14.1 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 fixed size groups of bytes, called **pages**. This makes it easier to manage the whole memory: think of Tetris with all squares. The OS then maps each page of virtual memory to a chunk of physical memory referred to as a **frame**.

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. While virtual memory pages give a program the impression of a contiguous region of memory, in fact the pages may be mapped to arbitrary frames in RAM in any order. Some virtual pages may not map to any actual physical memory address at all. This is 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".

The previous lecture illustrated how applications typically expect contiguous regions of memory to store their data in, but that this can lead to both internal and external fragmentation. The use of virtual memory can eliminate part of this problem by giving each process a virtual address space that appears to be contiguous, but is in fact mapped to arbitrary regions of physical memory by the operating system. This can *eliminate the external fragmentation* issues described previously that occurred because small memory regions between processes could not be used; with virtual memory, as long as a memory region is at least the size of a page, it can be allocated and mapped into the virtual address space of a process.

The virtual memory abstraction provides some additional benefits as well: *1)* it gives each individual program the illusion of having the whole address space for itself; and *2)* it isolates the address space being used by each process from other applications–all processes think their address space starts at address zero and reaches until the end of the system's RAM, so they cannot attempt to pick a random address in order to try to peak into the memory region of another application. 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 on a regular basis. 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 (described later on) to ensure that the memory regions actively being used are always available.

14.1.1 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). Since this translation could potentially be required for every assembly instruction, address translation must be done very quickly. As a result, modern computers include a hardware MMU (Memory Management Unit) that performs these actions. The MMU maintains a *page table* (a big hash table) that maps VAs into PAs. 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 works at coarser granularity and 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 ensures that even if multiple different processes use the same virtual address to refer to some data, 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 valid. Besides valid bits, entries in the page table can also store other information, such as "read" and "write" bits to indicate which pages can be read/written.

14.1.2 Virtual addresses

Virtual addresses are made up of two parts: the first one contains a page number (p), and the second one contains an offset inside that page (d). Suppose our pages are 4KB (4096 bytes = 2^{12} 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. Thus the first part of a virtual address will be formed by its 20 most significant bits, which will be used to address an entry in the page table (p); the $32 - 20 = 12$ least significant bits of the VA will be used as an offset inside the page (d). 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.

For a second example, consider a system with only 256 bytes of memory and a page size of 16 bytes; assume that within a page, memory can be addressed at word level (4 byte) granularity. Such a system can support $\frac{256}{16} = 16$ pages. In order to address the 16 pages, we need 4 bits ($2^4 = 16$). Since each page is divided into $\frac{16}{4} = 4$ addressable words, we need 2 bits to calculate the offset. Thus the 4 most significant bits in a virtual address will be used to select the page, p, and the 2 least significant bits in each address will be used for d, the offset within a page. Given a virtual address, the correct page and offset can be easily found by writing out the address in binary and separating the bits into the p and d chunks. Converting each of these pieces back into decimal will reveal the correct page table entry and the offset within that page. In order to find the true *physical* address, you must then use p as an index into the page table to determine the actual physical frame where the page is stored. The final physical address is then calculated by using the frame address plus the offset within the page.

14.1.3 Translation Look-aside Buffer

In order to speed up the translation of virtual to physical addresses, modern computers contain a Translation Look-aside Buffer (TLB) as part of the MMU hardware. The TLB acts as a cache of page to frame mappings. When an address must be translated, if it is found in the TLB then it can be very quickly mapped to the correct physical address. On a TLB miss, then the address translation must be done using the full set of page tables stored in the system's main memory. Since most applications exhibit locality in how they access memory, most addresses can be found within the TLB without requiring an expensive lookup.

14.2 Starting and Switching Processes

When a new process starts, it will typically request a certain number of pages, say k. In a system using paging, k free frames will be found, and the page table will be updated with mappings for each virtual page to a physical frame address. The OS will then mark all of the TLB entries as invalid, causing the TLB cache to be flushed of any old data. The OS then starts the process, and as it executes the OS will load the TLB entry for each page as it is accessed. Eventually, the TLB may run out of space, at which point the TLB must use some sort of eviction policy to determine which pages to remove from the TLB cache to create space.

The contents of a page table and the TLB are specific to an individual process. This means that every time the OS context switches between processes, it must update both of these data structures. In general, this is done by extending the Process Control Block (PCB) described in earlier lectures with the process's page table and possibly a copy of the TLB (otherwise the TLB must be flushed and rebuilt on each context switch). When a context switch occurs, the following steps are performed: (1) the current page table base register must be copied to the PCB, (2) the contents of the TLB is copied into the PCB, (3) the TLB is flushed, (4) the next PCB is loaded, and the page table base register is restored from it, and (5) the TLB is reloaded from the PCB. These actions are part of the reason that context switch overheads can reduce performance in an OS.

14.3 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 reentrant, 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.