10.8 Virtual Memory And Memory Multiplexing

Most large computer systems virtualize memory and present an application with an abstract view of memory. Each application appears to occupy a large address space; on a system with a small physical memory, the address space can exceed the size of physical memory. The operating system multiplexes physical memory among all processes that need to use it, moving all or part of the application into physical memory as needed. That is, code and data for processes are kept on secondary storage (i.e., disk), and moved into main memory temporarily when the process is executing. Although few embedded systems rely on virtual memory, many processors include virtual memory hardware.

The chief design question in virtual memory management systems concerns the form of multiplexing. Several possibilities have been used:

- Swapping
- Segmentation
- Paging

Swapping refers to an approach that moves all code and data associated with a particular computation into main memory when the scheduler makes the computation current. Swapping works best for a long-running computation, such as a word processor that runs while a human types a document — the computation is moved into main memory, and remains resident for a long period.

Segmentation refers to an approach that moves pieces of the code and data associated with a computation into main memory as needed. One can imagine, for example, placing each function and the associated variables in a separate segment. When a function is called, the operating system loads the segment containing the function into main memory. Seldom-used functions (e.g., a function that displays an error message) remain on secondary storage. In theory, segmentation uses less memory than swapping because segmentation allows pieces of a computation to be loaded into memory as needed. Although the approach has intuitive appeal, few operating systems use dynamic segmentation.

Paging refers to an approach that divides each program into small, fixed-size pieces called *pages*. The system keeps the most recently referenced pages in main memory, and moves copies of other pages to secondary storage. Pages are fetched on demand — when a running program references memory location i, the memory hardware checks to see whether the page containing location i is *resident* (i.e., currently in memory). If the page is not resident, the operating system suspends the process (allowing other processes to execute), and issues a request to the disk to obtain a copy of the needed page. Once the page has been placed in main memory, the operating system makes the process ready. When the process retries the reference to location i, the reference succeeds.

10.9 Real And Virtual Address Spaces

In many operating systems, the memory manager supplies each computation with its own, independent address space. That is, an application is given a private set of memory locations that are numbered *0* through *M – 1*. The operating system works with the underlying hardware to map each address space to a set of memory locations in memory. As a result, when an application references address zero, the reference is mapped to the memory location that corresponds to zero for the process. When another application references address zero, the reference is mapped to a different location. Thus, although multiple applications can reference address zero, each reference maps to a separate location, and the applications do not interfere with one another. To be precise, we use the term *physical address space* or *real address space* to define the set of addresses that the memory hardware provides, and the term *virtual address space* to describe the set of addresses available to a given computation. At any time, memory functions in the operating system map one or more virtual address spaces onto the underlying physical address space. For example, Figure 10.1 illustrates how three virtual address spaces of *K* locations can be mapped onto an underlying physical address space that contains *3K* locations.

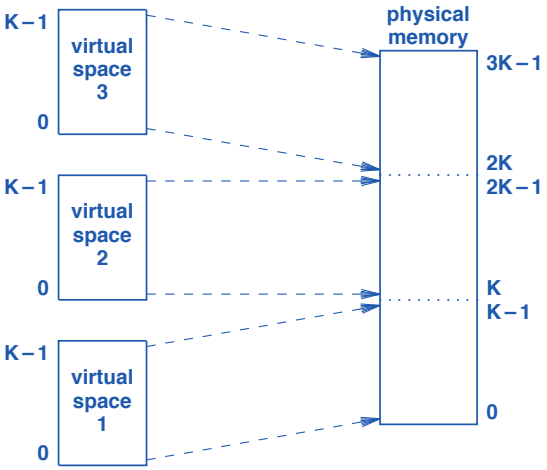

Figure 10.1 Illustration of three virtual address spaces mapped onto a single underlying physical address space.

From the point of view of a running computation, only addresses in the virtual address space can be referenced. Furthermore, because the system maps each address from a given virtual space to a specific region of memory, the running computation cannot accidentally read or overwrite memory that has been allocated to another computation. As a result, a system that provides each computation with its own virtual address

space helps detect programming errors and prevent problems. The point can be summarized:

> *A memory management system that maps each virtual address space into a unique block of memory prevents one computation from reading or writing memory allocated to another computation.*

In Figure 10.1, each virtual address space is smaller than the underlying physical address space. However, most memory management systems permit a virtual address space to be larger than the memory on the machine. For example, a demand paging system only keeps pages that are being referenced in main memory, and leaves copies of other pages on disk.

The original research on virtual memory arose from two motivations. First, the small size of physical memory limited the size of programs that were feasible. Second, with only one program running, a processor remained idle during I/O cycles. On the one hand, if multiple programs can be placed in memory, an operating system can switch the processor among them, allowing one to execute while another waits for an I/O operation. On the other hand, squeezing more programs into a fixed memory means each program will have less allocated memory, forcing the programs to be small. So, demand paging arose as a mechanism that handled both problems. By dividing programs into many pieces, a demand paging system can choose to keep parts of many programs in memory, keeping the processor busy when a program waits for I/O. By allowing pieces of a program's address space to be resident in memory without requiring space for the entire program, a demand paging system can support an address space that is larger than physical memory.

10.10 Hardware For Demand Paging

An operating system that maps between virtual and real addresses cannot operate without hardware support. To understand why, observe that each address, including addresses generated at runtime, must be mapped. Thus, if a program computes a value *C* and then *jumps* to location *C*, the memory system must map *C* to the corresponding real memory address. Only a hardware unit can perform the mapping efficiently.

The hardware needed for demand paging consists of a page table and an address translation unit. A page table resides in kernel memory, and there is one page table for each process. Typically, the hardware contains a register that points to the current page table and a second register that specifies the length; after it has created a page table in memory, the operating system assigns values to the registers and turns on demand paging. Similarly, when a context switch occurs, the operating system changes the page table registers to point to the page table for the new process. Figure 10.2 illustrates the arrangement.

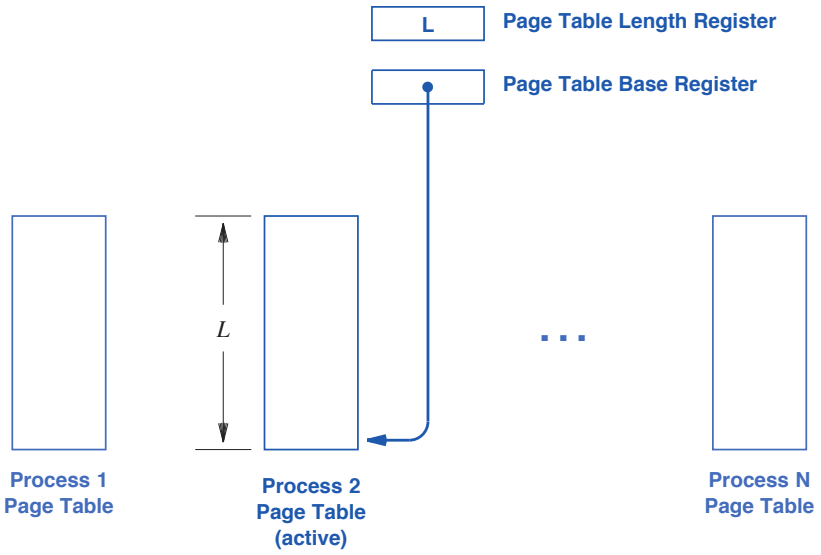

Figure 10.2 Page tables in memory and hardware registers that specify which page table to use at a given time.

10.11 Address Translation With A Page Table

Conceptually, a page table consists of an array of pointers to memory locations. In addition to a pointer, each entry contains a bit that specifies whether the entry is valid (i.e., whether it has been initialized). *Address translation* hardware uses the current page table to translate a memory address; translation is applied to the instruction addresses as well as addresses used to fetch or store data. Translation consists of array lookup: the hardware treats the high-order bits of an address as a *page number*, uses the page number as an index into the page table, and follows the pointer to the location of the page in memory.

In practice, a page table entry does not contain a complete memory pointer. Instead, pages are restricted to start in memory locations that have zeroes in the low-order bits, and the low-order bits are omitted from each page table entry. For example, suppose a computer has 32-bit addressing and uses 4096-byte pages (i.e., each page contains 2^{12} bytes). If memory is divided into a set of 4096-byte *frames*, the starting address of each frame (i.e., the address of the first byte in the frame) will have zeroes in the 12 low-order bits. Therefore, to point to a frame in memory, a page table entry only needs to contain the upper 20 bits.

To translate an address, A, the hardware uses the upper bits of A as an index into the page table, extracts the address of a frame in memory where the page resides, and then uses the low-order bits of A as an offset into the frame. We can imagine that the translation forms a physical memory address as Figure 10.3 illustrates.

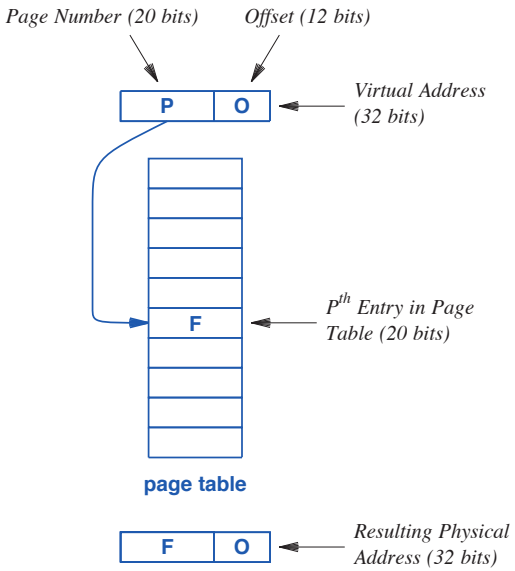

Figure 10.3 An example of virtual address translation used with paging.

Our description implies that each address translation requires a page table access (i.e., a memory access). However, such overhead would be intolerable. To make translation efficient, a processor employs a special-purpose hardware unit known as a *Translation Look-aside Buffer* (*TLB*). The TLB caches recently accessed page table entries, and can look up the answer much faster than a conventional memory access. To achieve high speed, a TLB uses a form of parallel hardware known as a *Content-Addressable Memory* (*CAM*). Given an address, CAM hardware searches a set of stored values in parallel, returning the answer in only a few clock cycles. As a result, a TLB enables a processor to run as fast with address translation turned on as it can with address translation disabled.

10.12 Metadata In A Page Table Entry

In addition to a frame pointer, each page table entry contains three bits of metadata that the hardware and operating system use. Figure 10.4 lists the bits and their meaning.

Name	Meaning
Use Bit	Set by hardware whenever the page is referenced (fetch or store)
Modify Bit	Set by hardware whenever a store operation changes data on the page
Presence Bit	Set by OS to indicate whether the page is resident in memory

Figure 10.4 The three metabits in each page table entry and their meanings.

10.13 Demand Paging And Design Questions

The term *demand paging* refers to a system where an operating system places the pages for all processes on secondary storage, and only reads a page into memory when the page is needed (i.e., on demand). Special processor hardware is required to support demand paging: if a process attempts to access a page that is not resident in memory, the hardware must suspend execution of the current instruction and notify the operating system by signaling a *page fault* exception. When a page fault occurs, the operating system finds an unused frame in memory, reads the needed page from disk, and then instructs the processor to resume the instruction that caused the fault.

When a computer first starts, memory is relatively empty, which makes finding a free frame easy. Eventually, however, all frames in memory will be filled and the operating system must select one of the filled frames, copy the page back to disk (if the page has been modified), fetch the new page, and change the page tables accordingly. The selection of a page to move back to disk forms a key problem for operating systems designers.

The design of a paging system centers on the relationship between pages and processes. When process *X* encounters a page fault, should the operating system move one of process *X*'s pages back to disk, or should the system select a page from another process? While a page is being fetched from disk, the operating system can run another process. How can the operating system ensure that at least some process has enough pages to run without also generating a page fault?† Should some pages be locked in memory? If so, which ones? How will the page selection policy interact with other policies, such as scheduling? For example, should the operating system guarantee each high-priority process a minimum number of resident pages? If the system allows processes to share memory, what policies apply to the shared pages?

One of the interesting tradeoffs in the design of a paging system arises from the balance between I/O and processing. Paging overhead and the latency a process experiences can be reduced by giving a process the maximal amount of physical memory when the process runs. However, many processes are I/O-bound, which means that a given process is likely to block waiting for I/O. When one process blocks, overall per-

†If insufficient frames exist in memory, a paging system can *thrash*, which means the frequency of page faults becomes so high that the system spends all its time paging and each process spends long periods waiting for pages.

formance is maximized if another process is ready to execute and the operating system can switch context. That is, processor utilization and overall throughput of a system can be increased by having many processes ready to execute. So the question arises: should a given process be allowed to use many frames of memory or should memory be divided among processes?

10.14 Page Replacement And Global Clock

Various page replacement policies have been proposed and tried:

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- First In First Out (FIFO)

Interestingly, a provably optimal replacement policy has been discovered. Known as *Belady's optimal page replacement algorithm*, the policy chooses to replace the page that will be referenced farthest in the future. Of course, the method is totally impractical because the operating system cannot know how pages will be used in the future. However, Belady's algorithm allows researchers to assess how well replacement policies perform.

In terms of practical systems, a single algorithm has become the de facto standard for page replacement. Known as *global clock* or *second chance*, the algorithm was devised as part of the MULTICS operating system and has relatively low overhead. The term *global* means that all processes compete with one another (i.e., when process *X* generates a page fault, the operating system can choose a frame from another process, *Y*). The alternative name of the algorithm arises because global clock is said to give used frames a "second chance" before they are reclaimed.

Global clock starts running whenever a page fault occurs. The algorithm maintains a pointer that sweeps through all the frames in memory, stopping when a free frame is found. The next time it runs, the algorithm starts at the frame just beyond where it left off.

To determine whether to select a frame, global clock checks the Use and Modify bits in the page table of the frame. If the Use/Modify bits have value (0,0), global clock chooses the frame. If the bits are (1,0), global clock resets them to (0,0) and bypasses the frame. Finally, if the bits are (1,1), global clock changes them to (1,0) and bypasses the frame, keeping a copy of the modified bit to know whether the page has been modified. In the worst case, global clock sweeps through all frames twice before reclaiming one.

In practice, most implementations use a separate process to run the global clock algorithm (which allows the clock to perform disk I/O). Furthermore, global clock does not stop immediately once a frame has been found. Instead, the algorithm continues to run, and collects a small set of candidate pages. Collecting a set allows subsequent page faults to be handled quickly and avoids the overhead associated with running the global clock algorithm frequently (i.e., avoids frequent context switching).

10.15 Perspective

Although address space management and virtual memory subsystems comprise many lines of code in an operating system, the most significant intellectual aspects of the problem arise from the choice of allocation policies and the consequent tradeoffs. Allowing each subsystem to allocate arbitrary amounts of memory maximizes flexibility and avoids the problem of a subsystem being deprived when free memory exists. Partitioning memory maximizes protection and avoids the problem of having one subsystem deprive other subsystems. Thus, a tradeoff exists between flexibility and protection.

Despite years of research, no general solution for page replacement has emerged, the tradeoffs have not been quantified, and no general guidelines exist. Similarly, despite years of research on virtual memory systems, no demand paging algorithms exist that work well for small memories. Fortunately, economics and technology have made many of the problems associated with memory management irrelevant: DRAM chip density increased rapidly, making huge memories inexpensive. As a result, computer vendors avoid memory management altogether by making the memory on each new product much larger than the memory on the previous product. Because memory is so large, the operating system can satisfy the needs of a process without taking frames from other processes. That is, a demand paging system works well not because we have devised excellent replacement algorithms, but because memories have grown so large that replacement algorithms are seldom invoked.

10.16 Summary

Low-level memory allocation mechanisms treat all of free memory as a single, exhaustible resource. High-level memory management facilities that allow memory to be partitioned into separate regions provide guarantees that prevent one subsystem from using all available memory.

The high-level memory management functions in Xinu use a buffer pool paradigm in which a fixed set of buffers is allocated in each pool. Once a pool has been created, a group of processes can allocate and free buffers dynamically. The buffer pool interface is synchronous: a given process will block until a buffer becomes available.

Large operating systems use virtual memory mechanisms to allocate a separate address space for each process. The most popular virtual memory mechanism, paging, divides the address space into fixed-size pages, and loads pages on demand. Hardware is needed to support paging because each memory reference must be mapped from a virtual address to a corresponding physical address. Paging systems only perform satisfactorily if memory is large enough to avoid running the page replacement algorithm frequently.