Chapter 3: Memory Management

- memory hierarchy: computers have a few megabytes of very fast, expensive, volatile cache memory, a few gigabytes of medium-speed, medium-priced, volatile main memory, and a few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage, not to mention removable storage.
- memory manager: the part of the OS than manages the memory hierarchy
    - keeps track of which parts of memory are in use, allocate memory to processes when they need it, and deallocate it when they are done.

No memory abstraction:

- early computers had no memory abstraction
- memory presented to the programmer was physical memory
- under these conditions, it was not possible to have two running programs in memory at the same time
- BIOS: where portion of the system is in the ROM
    - one process at a time can be running
- only way to get parallelism in a system with no memory abstraction is to program with multiple threads

Running Multiple Programs without a memory abstraction:

- OS has to save the entire contents of memory to a disk file, then bring in and run the next program (swapping)
- as long as there is only one program at a time in memory, there are no conflicts
- static relocation:It worked like this. When a program was loaded at address 16,384, the constant 16,384 was added to every program address during the load process (so ‘‘JMP 28’’ became ‘‘JMP 16,412’’, etc.)

The notion of an address space:

- two problems have to be solved to allow multiple applications to be in memory at the same time without interfering with each other: protection and relocation
- address space (protection solution) : creates a kind of abstract memory for programs to live in
    - the set of addresses that a process can use to address memory

Base and Limit Registers

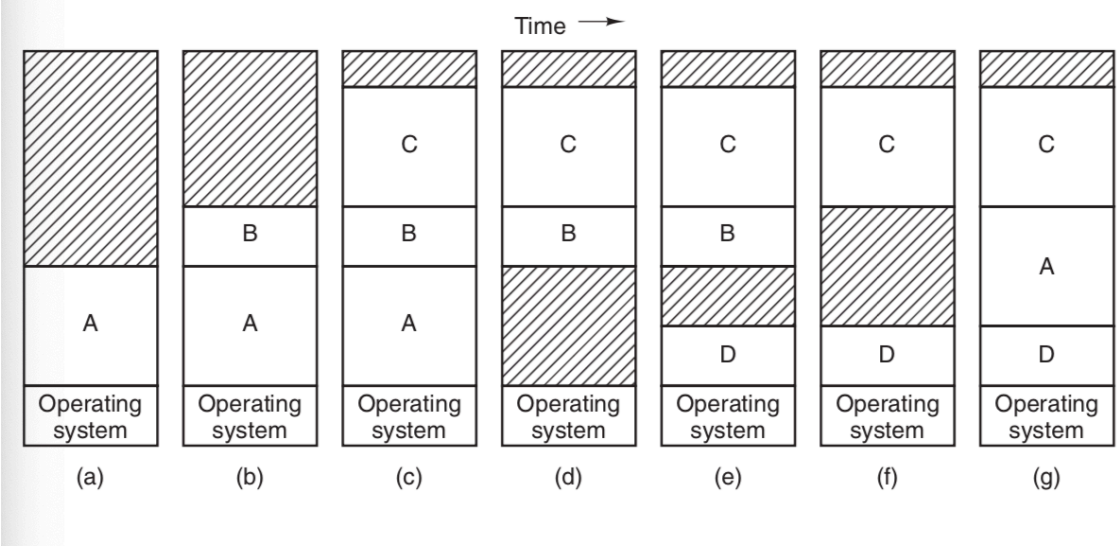
- dynamic relocation: map each process’ address space onto a different part of physical memory

in a simple way

- Equip the CPU with two special hardware registers, base and limit registers
  - when they are used the programs are loaded into consecutive memory locations wherever there is room and without relocation during loading
  - when a process is run, the base register is loaded with the physical address where its program begins in memory and the limit register is loaded with the length of the program
  - every time a process references memory, either to fetch an instruction for read or write a data word, the CPU hardware automatically adds the base values to the address generated by the process before sending the address out on the memory bus
  - simultaneously it checks whether the address offered is qual to or greater than the value in the limit register, in which case a fault is generated and the access is aborted
- The disadvantage using these registers is the need to perform an addition and a comparison on every memory reference

Swapping:

- approaches to deal with memory overload
- swapping: consists of bringing in each process n its entirety, running it for a while, then putting it back on the disk



- virtual memory: allows programs to run even when they are only partially in main memory
- when swapping creates multiple holes in memory, it is possible to combine then all into one big one by moving all the processes downward as far as possible : memory compaction

Managing Free Memory:

- keep track go memory usage: bitmaps and free lists

Memory Management with Bitmaps:

- with bitmaps: memory is divided into allocation units as small as a few words and as large as several kilobytes
- corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if its occupied
- the smaller the allocation unit, the larger the bitmap
- a bitmap provides a simple way to keep track go memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the bitmap depends only on the size of memory and the size of the allocation unit
-

Memory Management with Linked Lists:

- keeping track of memory with lined list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes
- several algorithms can be used to allocate memory for a created process:
    - first fit: the memory manager scans along the list of segments until it finds a hole that is big enough
        - hole is broken up into two pieces, one for the process and one for the unused memory
        - fastest algorithm
    - next fit: keeps track of where it is whenever it find a suitable hole, the next time it is called to find a hole, it starts searching the list from the place whether it left off last time
    - Best fit: searches the entire list, and take the smallest hole that is adequate : find hole closest to the actual size needed
        - slower
    - quick fit: maintains separate lists for some of the more common sizes requested

Virtual Memory:

- spitting programs into smaller pieces: overlays
    - when a program started all that was loaded into memory was the overlay manager
    - kept on the disk and swapped into memory by the overlay manager
- virtual memory: each program has its own address space, broken up into chunks called pages
    - each page has a range of addresses
    - pages are mapped onto physical memory

Paging:

- program generated addresses are called virtual addresses and form the virtual address space
- when virtual memory is used the virtual address goes to MMU (Memory Management Unit) that maps to the virtual addresses onto the physical memory addresses
- the page number is used as an index into the page table, yielding the number of the page frame corresponding to that virtual page

Page Tables:

- the virtual address is spit into a virtual page number and an offset
    - For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page
- the virtual page number is used as an index into the page table to find the entry for that virtual page
- from the page table entry, the page frame number is found
- the page frame number is attached to the high order end of the offset forming a physical address that can be sent to memory

Structure of a Page Table Entry:

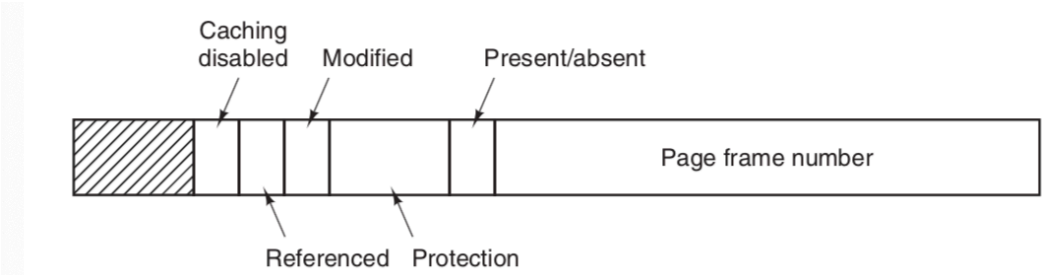


Figure 3-11. A typical page table entry.

- Protection but: access permitted, simplest form: field contains 1 bit : 0 for read/write and 1 for read only
- Modified and Referenced bit keeps track of page usage
    - dirty bit, clean bit
- Referenced bit: whenever a page is referenced: reading or writing
- last but allows caching to be dibbled for the page

Speeding Up Paging:

- In any paging system: two major issues must be faces:
    - 1. The mapping from virtual address to physical address must be fast
    - 2. if the virtual address space is large, the page table will be large
- virtual to physical mapping must be done on every memory reference

Translation Lookaside Buffers:

- a hardware device that maps virtual addresses to physical addresses without going through the page table
- each entry contains info about one page, including the virtual page number, a bit that is set when the page is modified, the protection code, and the physical page frame in which the page is located

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 3-12. A TLB to speed up paging.

Software TLB Management:

- many machines do page management in software
- TLB entries loaded by the OS
- soft miss: when the page references is not in the TLB but is in memory
- hard miss: page itself is not in emory
    - disk access is required to bring in the page
- Minor page fault: a page brought in from disk by another process, so we don't need to access the disk again

- Major page fault: page needs to be brought in from disk
- segmentation fault: program accessed an invalid address and no mapping needs to be added to the TLB

Multilevel Page Tables:

- avoid keeping all the page tables in memory all the time
- page directory: entries pointed to page tables, which point to page frames
- page directory pointer table: extends each entry in each level of the page table hierarchy from32 to 64 bits, so it can access memory about the 4 GB boundary

Inverted Page Tables:

- there is only one entry per page frame in real memory, rather than one entry per page of virtual address space
-
    -