Topic 11
Lecture 11a
Memory Management

CSCI 240

Data Structures and Algorithms

Prof. Dominick Atanasio

Dictum

- All programmers want an infinite abount of RAM that is as fast as L1 cache and is nonvolatile. Wishfull thinking!

Memory

- The concept of a memory hierarchy
- a few MBs of very fast, expensive, volatile cache memory
- a few GBs of medium-speed, medium-priced, volatile main memory
- A few terabytes of slow, cheap, nonvolatile magnetic or solid-state disk storage
- It is the job of the operating system to abstract this hierarchy into a useful model and then manage the abstraction.

Memory Manager

- The part of the operating system that manages (part of) the memory hierarchy is called the memory manager
  - keep track of which parts of memory are in use
  - allocate memory to processes
  - deallocate it when they are done

A MEMORY ABSTRACTION: ADDRESS SPACES

- The Notion of an Address Space
- an abstraction for memory
  - a kind of abstract memory space for programs to live in
  - Address space is the set of addresses that a process can use to address memory

Swapping

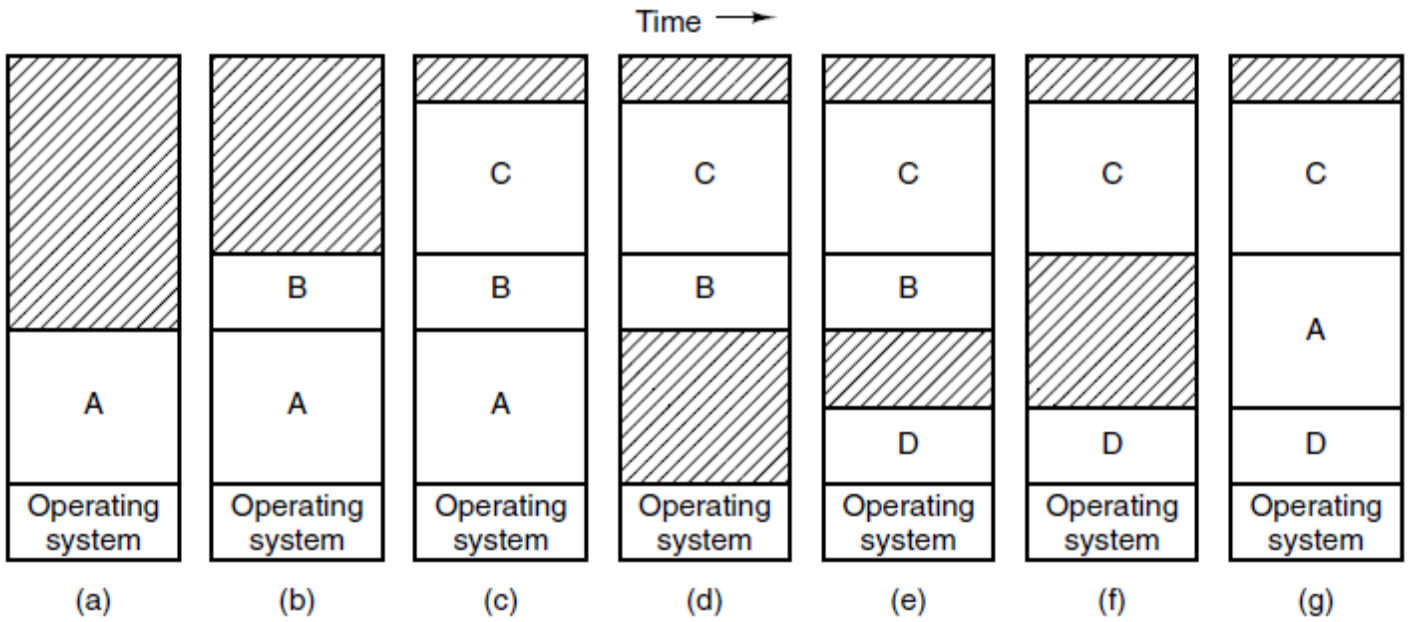
- If the physical memory of the computer is large enough to hold all the processes, then most schemes will more or less work.
- In practice, the total amount of RAM needed by all the processes is often much more than can fit in memory
- Modern operating systems have many processes running at any time
- Large user applications, like Photoshop, can easily require 500 MB just to start and many gigabytes when processing data

Approaches

- Two general approaches to dealing with memory overload
- Swapping
- bring in each process in its entirety, run it for a while, then putting it back on the disk if memory is full
- Virtual memory
  allows programs to run even when they are only partially in main memory

Swapping (1)

- Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory



Memory Compaction

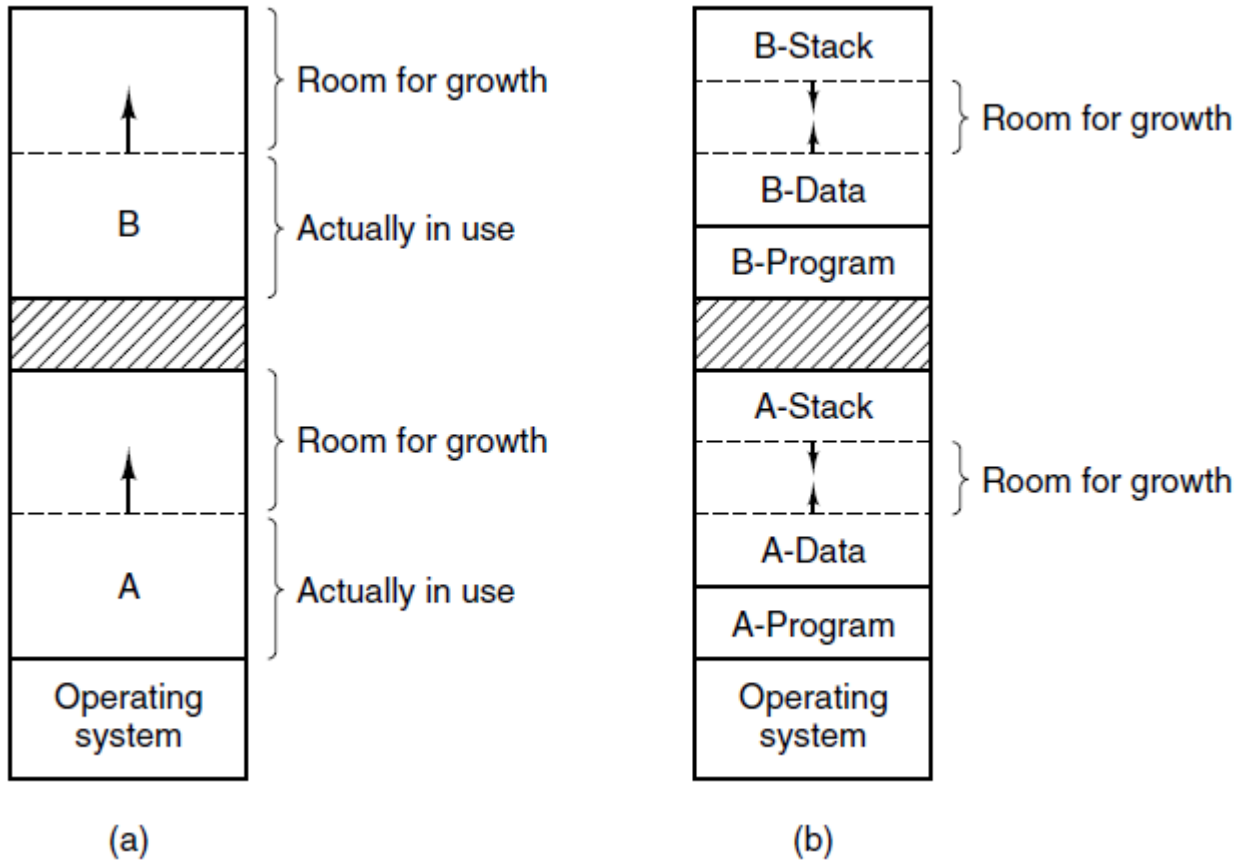
- Swapping can create multiple holes in memory
- It is possible to combine them all into one big one by moving all the processes downward
- Requires a lot of CPU time

Fixed vs Dynamic Size

- Fixed: the size of a process' address space never changes, allocation is simple
- Dynamic: address space may change to accommodate a process' needs
  - If a hole is adjacent to the process, the process can be allowed to grow into the hole otherwise may need to be moved
  - a good idea is to allocate a little extra memory whenever a process is swapped in or moved
  - when swapping processes to disk, only the memory actually in use should be swapped

Swapping (2)

- Figure 3-5. (a) Allocating space for a growing data segment.
  (b) Alternative, allocating space for a growing stack and a growing data segment.



Managing Free Memory

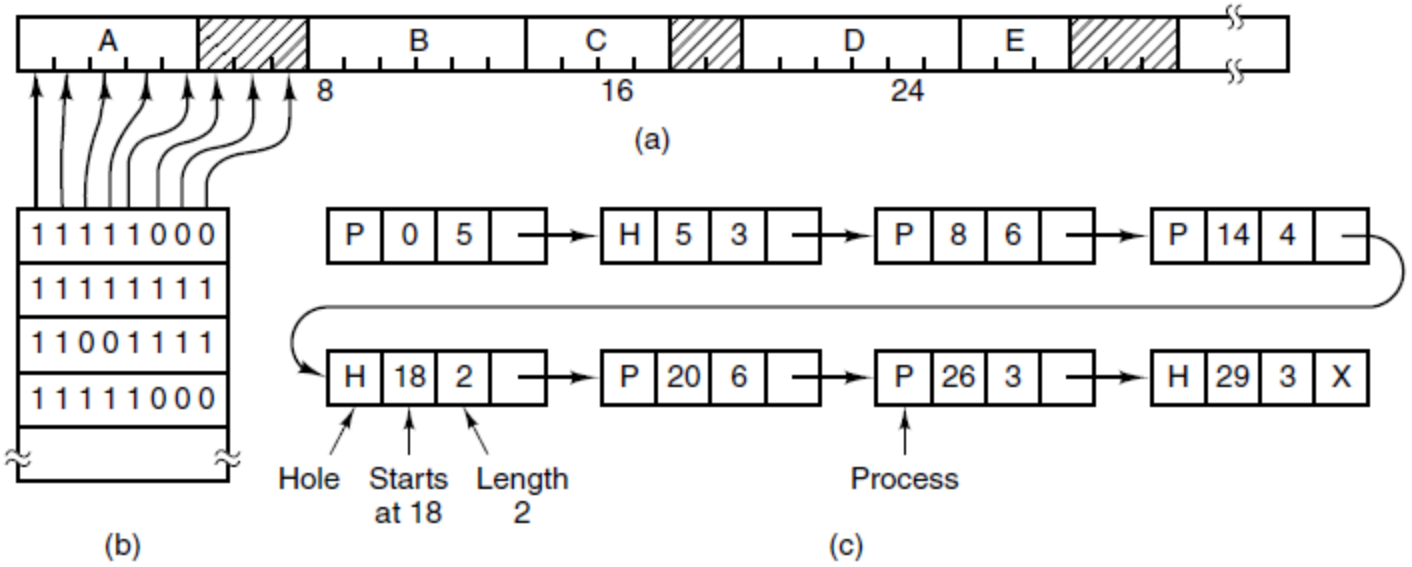
- When memory is assigned dynamically, the operating system must manage it.
- In general terms, there are two ways to keep track of memory usage: bitmaps and free lists

Memory Management with Bitmaps

- With a bitmap, 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
  - 0 if the unit is free and 1 if it is occupied

Memory Management with Bitmaps

- Figure 3-6. (a) A part of memory with five processes and three holes. The tick-marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.



Memory Management with Bitmaps

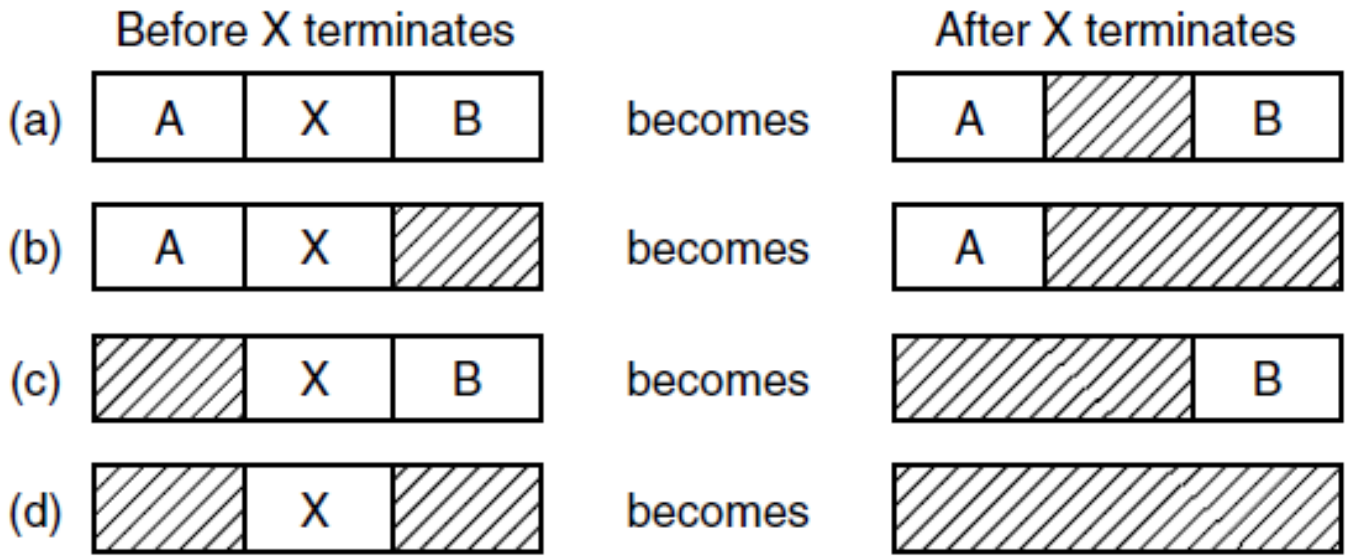
- The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap
- With an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map
  - the bitmap will take up only 1/32 of memory
- With larger allocation unit
  - Bitmap reduces in size
  - May be waste memory

Memory Management with Linked Lists

- Maintain a linked list of allocated and free memory segments
  - a segment either contains a process or is an empty hole between two processes (P or H)
  - Also contains the address at which it starts, the length, and a pointer to the next item
- Segment list might sorted by address
  - updating the list is straightforward
  - several algorithms can be used to allocate memory for a created process

Memory Management with Linked Lists

- Figure 3-7. Four neighbor combinations for the terminating process, X.



Memory Management Algorithms

- First fit: scan list of segments finds first hole big enough
- Next fit: Same but keeps track of last location
- Best fit: search the entire list for hole that’s just right
- Worst fit: always take the largest available hole
- Quick fit: keep lists for common sizes requested

Virtual Memory

- There is a need to run programs that are too large to fit in memory
- Solution adopted in the 1960s, split programs into little pieces, called overlays
  - Kept on the disk, swapped in and out of memory
- Virtual memory : each program has its own address space, broken up into chunks called pages

Virtual Memory

- In the 60’s programs that were so large they couldn’t fit in memory
- Solution: an overlay system
  - At program start, an overlay manager was loaded
  - It immediately started overlay 0
  - When overlay 0 finished overlay 1 was loaded over 0
  - Some systems were complex loading multiple overlays to the capacity of RAM
  - Later called virtual memory

Virtual Memory

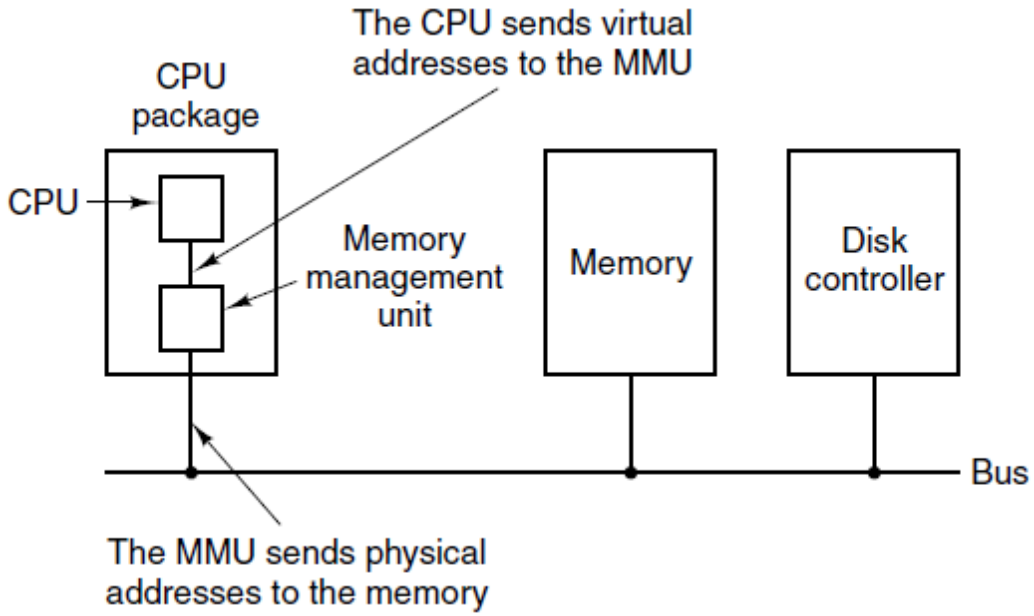
- Virtual memory is a generalization of the base-and-limit-register
- instead of having separate relocation for just the text and data segments, the entire address space can be mapped onto physical memory in fairly small units
- works well in a multiprogramming system
- many pieces of programs in memory at once
- while a program is waiting for pieces of itself to be read in, the CPU can be given to another process

Paging

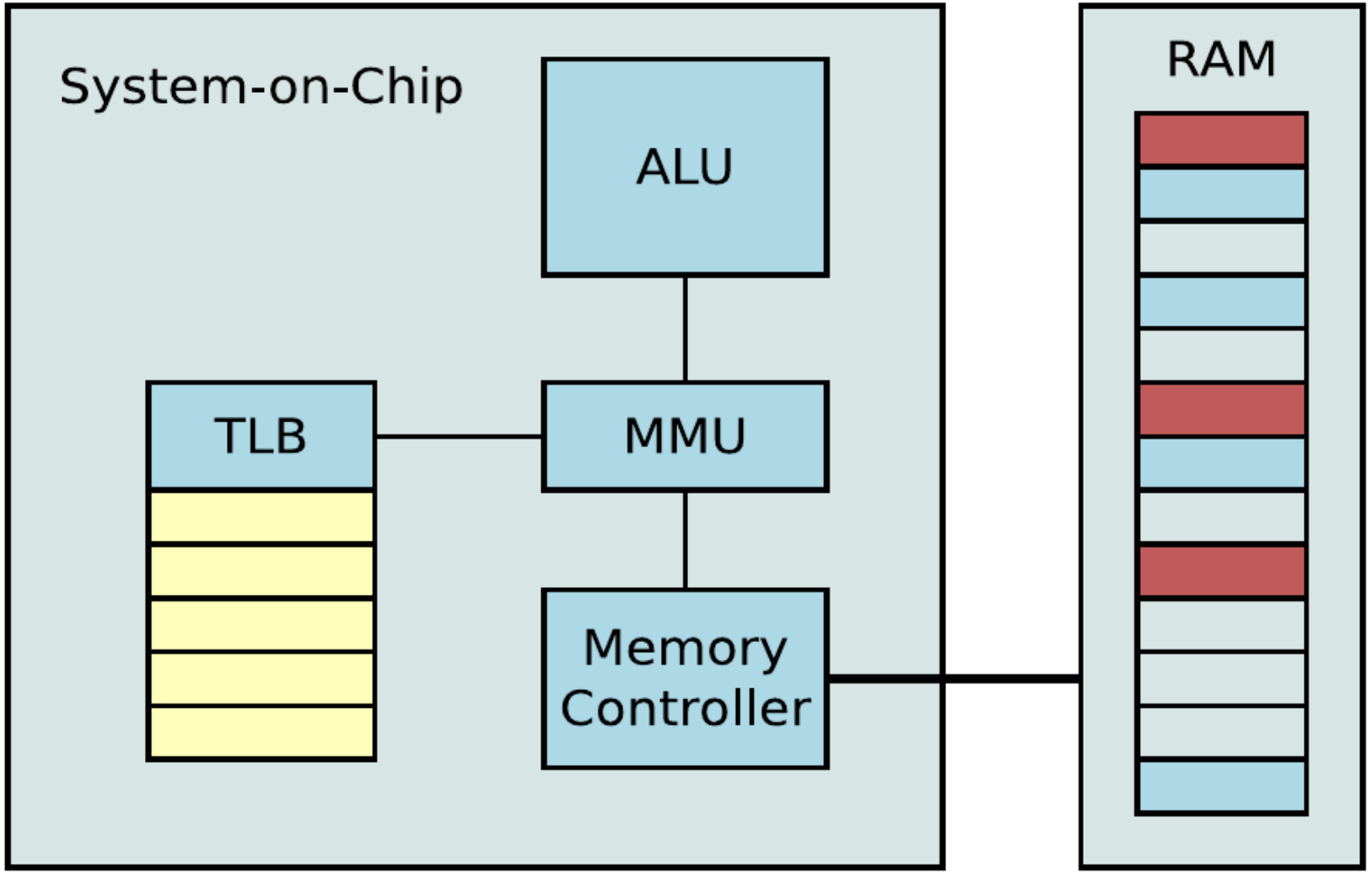
- Most virtual memory systems use a technique called paging
- Programs reference a set of memory addresses
- E.g. MOV REG,1000
  - copy the contents of mem address 1000 to REG
  - program-generated addresses are called virtual addresses and form the virtual address space
  - MMU (Memory Management Unit) maps the virtual addresses onto the physical memory

Paging (1)

- Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.



Virtual Memory System (hardware)

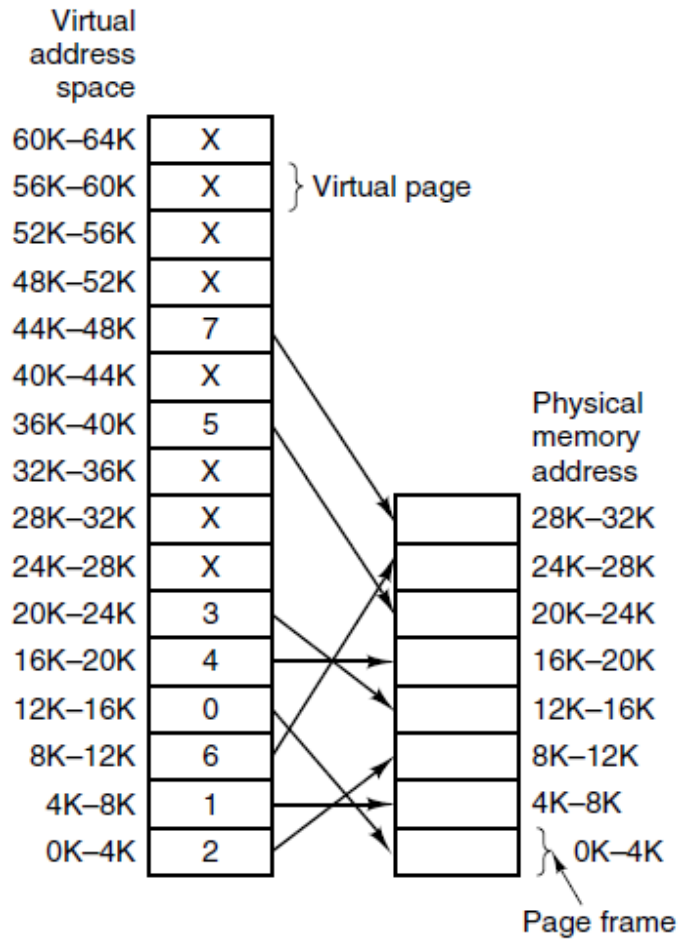


Mapping

- A very simple example of how this mapping works
- Computer that generates 16-bit addresses, from 0 up to 64K − 1
  - These are the virtual addresses given to each process
  - Every process sees the same addresses
- Assume, however, the computer has only 32 KB of physical RAM
- Virtual address space consists of fixed-size units called pages.
- The corresponding units in the physical memory are called page frames

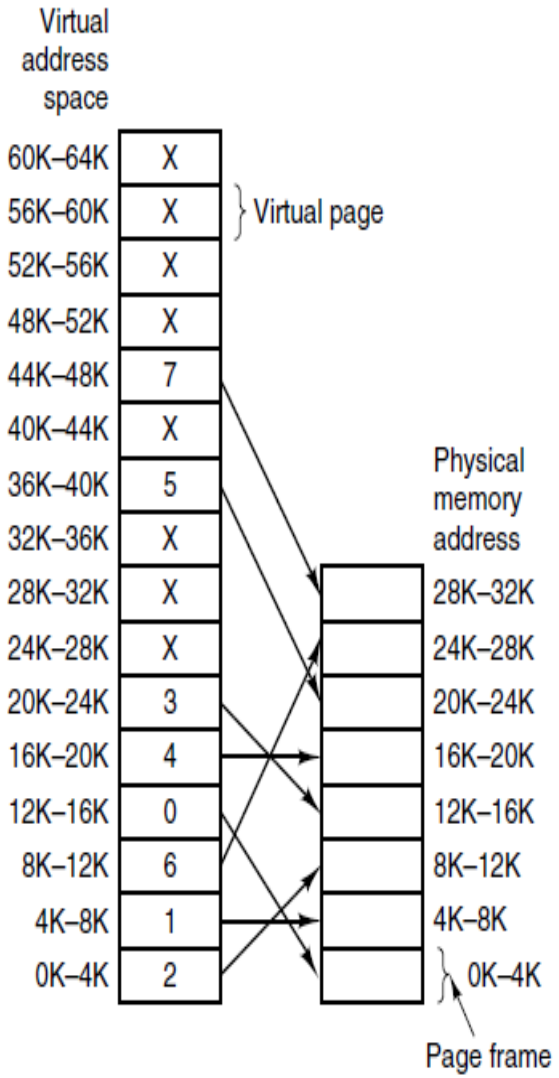
Paging (2)

- Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287



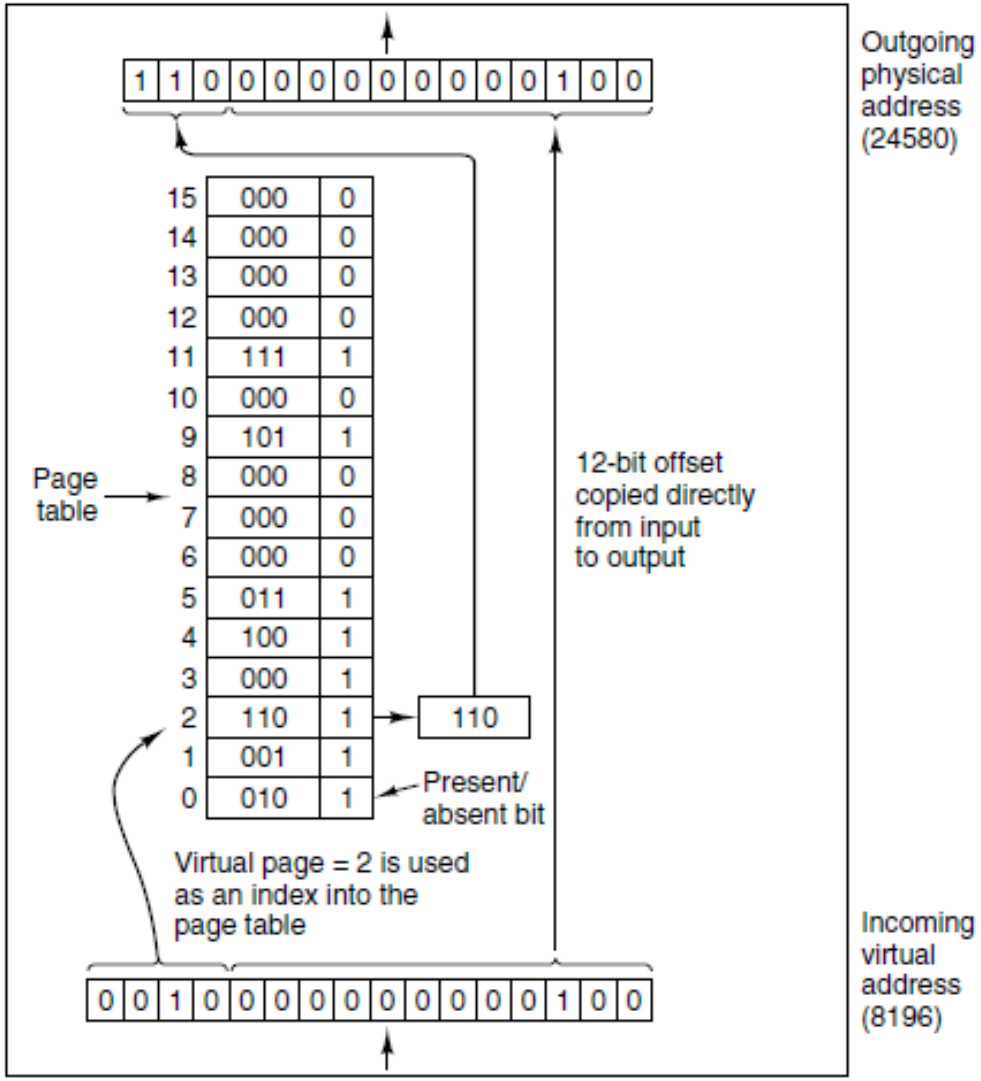
Page Faults

- What happens if the program references an unmapped address (e.g. MOV $5, 32780)?
- Byte 12 within virtual page 8 (starting at 32768)
- MMU determines that the page is unmapped
- This causes the CPU to trap to the operating system (Page Fault)
- The OS chooses a page frame to use, if it's contents are Dirty, the OS first writes it to disk then loads the requested page from the disk to that frame
- The OS updates the map, and restarts the trapped instruction



Paging (3)

- Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

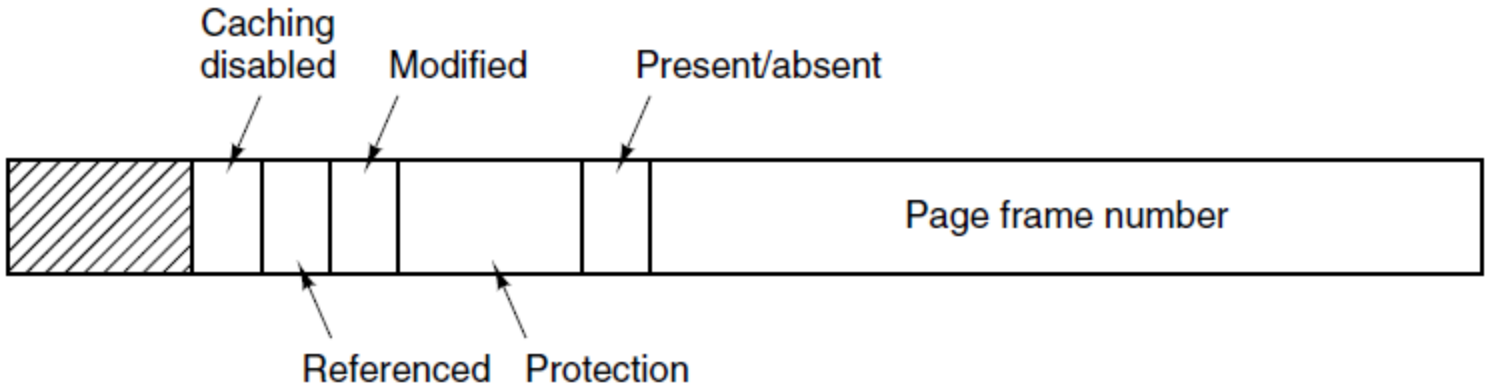


Page Table Entry (PTE)

- The layout of an entry in the page table is highly machine dependent
- Page frame number is a fixed size and sets the limit of how much RAM a machine can have
- Present/absent bit. If set to 1, the entry is valid and can be used
- The Protection bit(s) tell what kinds of access are permitted
  - Simple scheme (1 bit): 0 = read/write, 1 = read only
  - Better (3 bits): read, write, execute
- Referenced bit is set whenever a page is referenced, used by OS to choose a page to evict when a page fault occurs
- Modified: (dirty) used to tell if the page has been changed since loading
- Caching: important for pages that map onto device registers. This bit turns on/off caching of the word from device

Structure of a Page Table Entry

- Figure 3-11. A typical page table entry.



What About the Disk Addresses?

- Note that the disk address used to hold the page when it is not in memory is not part of the page table.
- The page table holds only that information the hardware needs to translate a virtual address to a physical address
- The location of the pages on disk is information that only the OS need to track; this information, and how it is organized, is OS dependent

Speeding Up Paging

- Major issues faced:
- The mapping from virtual address to physical address must be fast.
  - Virtual-to-physical mapping must be done on every memory reference
  - Instructions live in memory too which sometimes causes multiple lookups per instruction. If an instruction execution takes 1 nsec, the page table lookup must be done a fraction nsec to avoid a major bottleneck
- If the virtual address space is large, the page table will be large.
  - 32-bit address space has 1 million pages. 64-bit address space? Let's not go there
    - Page table must have 1 million entries. And remember that each process needs its own page table

Approaches

- An array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number
  - When a process starts, the OS loads the registers with the process' page table, taken from main memory
  - Very expensive
  - having to load the full page table at every context switch would destroy performance
- Translation Lookaside Buffer (associative cache)
  - The page table is in memory
  - Most programs tend to make a large number of references to a small number of pages; only a small fraction of the page table entries are heavily read
  - It is usually inside the MMU and consists of a small number of page table entries
  - fields have a one-to-one correspondence with the fields in the page table

Translation Lookaside Buffers

- Figure 3-12. A TLB to speed up paging.
- Valid bit is used to indicate if the entry is in use or not.

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

TLB in Action

- A virtual address is presented to the MMU for translation
- hardware checks to see if its virtual page number is present in the TLB by comparing it to all the entries
  - This is done in parallel , simultaneously
- If a valid match is found and the access does not violate the protection bits, the page frame # is taken directly from the TLB
  - the instruction is trying to write on a read-only page, a protection fault is generated
- If the virtual page number is not in the TLB
  - MMU detects the miss and does an ordinary page table lookup
  - It evicts one of the entries from the TLB and replaces it with the new lookup
    - When the entry is evicted from the TLB, the modified bit is copied back into the page table entry in memory
    - if that page is used again soon, the second time it will result in a TLB hit rather than a miss

Software TLB Management

- In most designs, TLB management and handling TLB faults are done entirely by the MMU hardware
- However, many RISC machines, including the SPARC, MIPS, do nearly all of this page management in software
- When a TLB miss occurs, instead of the MMU doing a lookup in page tables to find reference, it generates a TLB fault and hands-off the problem to the operating system
- The OS must find the page, overwrite an entry in the TLB with the new one, then restart the faulted instruction
- if the TLB is moderately large, software management can be acceptably efficient
- OS management could lead to smarter decisions on TLB entry replacement
- A soft miss occurs when the page referenced is not in the TLB
- A hard miss occurs when the page itself is not in the page table
  - At least a million times slower than a soft miss

Types of Misses

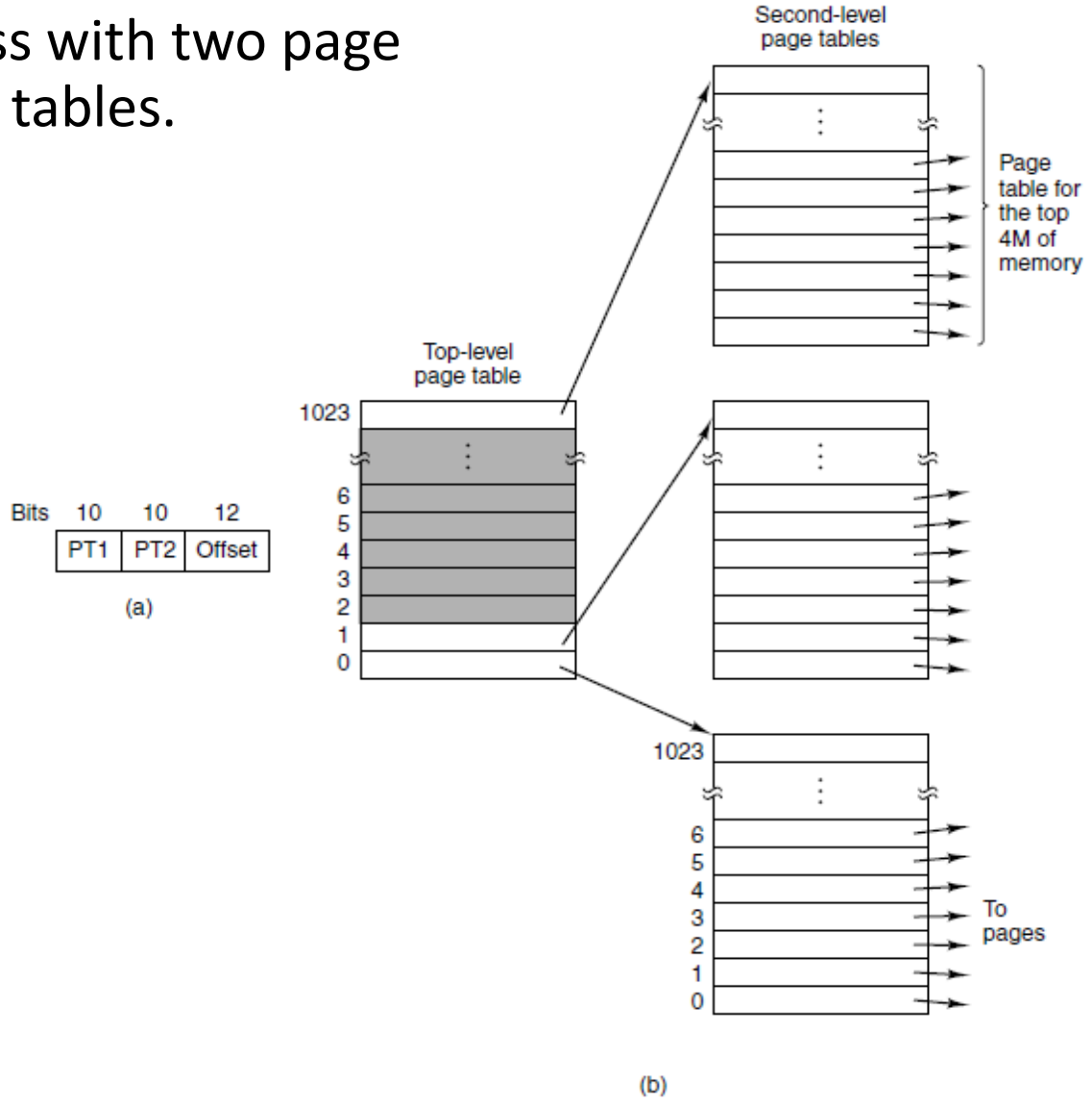
- Some misses are softer or harder than others
- Three possibilities:
- minor page fault: the page may actually be in memory, but not in this process' page table
  - Brought in by another process; just need map the page appropriately in the page tables.
- major page fault: the page needs to be brought in from disk
- segmentation fault: the program accessed an invalid address. the OS usually kills the program

Multilevel Page Tables

- TLBs can be used to speed up virtual-to-physical address translation over the original page-table-in-memory scheme
- How to deal with very large virtual address spaces?
- The multilevel page table method avoids keeping all the page tables in memory all the time
- Example: Suppose a process needs 12 megabytes: the bottom 4 MB of memory for program text, the next 4 MB for data, and the top 4 MB for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

Multilevel Page Tables

- Figure 3-13. (a) A 32-bit address with two page table fields. (b) Two-level page tables.



Page Replacement Algorithms

- When a page fault occurs, the operating system has to choose a page to evict (remove from memory) to make room for the incoming page
- If the page to be removed has been modified while in memory, it must be rewritten to the disk,
- The page to be read in just overwrites the page being evicted
- The OS could choose a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen
- The problem of ‘‘page replacement’’ occurs in other areas of computer design as well
  - most computers have one or more memory caches consisting of recently used memory blocks
  - Web servers keep a certain number of heavily used Web pages in a cache

Page Replacement Algorithms

- Optimal algorithm
- Not recently used algorithm
- First-in, first-out (FIFO) algorithm
- Second-chance algorithm
- Clock algorithm
- Least recently used (LRU) algorithm

Optimal Page Replacement Algorithm

- Easy to describe but impossible to implement
- The page with the highest label should be removed
  - The moment that a page fault occurs with some set of pages is in memory
  - One of these pages will be referenced on the very next instruction
  - Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later
- This algorithm is unrealizable
  - the OS has no way of knowing when the pages will be referenced
  - Similar problem as the shortest-job-first scheduling algorithm

Not Recently Used (NRU) Algorithm

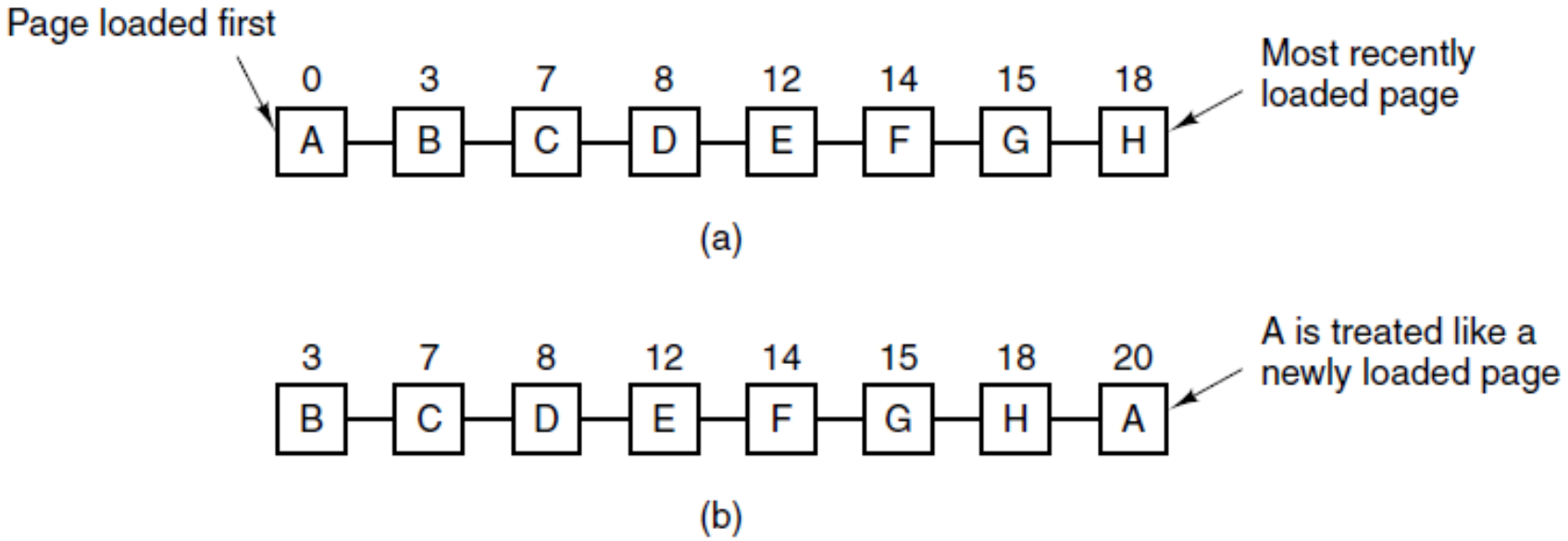
- At page fault, system inspects pages
- Categories of pages based on the current values of their R and D bits:
  - Hardware sets these bits but the OS clears them
- Class 0: not referenced, not modified.
- Class 1: not referenced, modified.
- Class 2: referenced, not modified.
- Class 3: referenced, modified.
- NRU algorithm removes a page at random from the lowest-numbered non-empty class

First-In, First-Out (FIFO) Algorithm

- A linked list (circular works) is maintained of all pages in memory
- The head of the list is the oldest (think queue)
- When a page fault occurs , the oldest page is discarded and the new page reference is added to the tail of the list
- An old page that is used heavily might be evicted just because it’s been in memory a long time
- In its purest form, this algorithm is not used much
- The second chance algorithm is a modification of FIFO

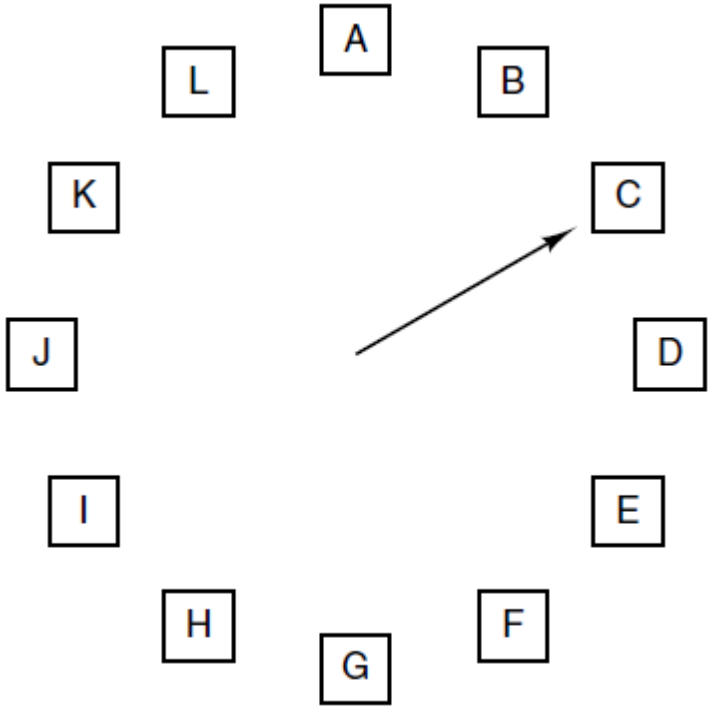
Second-Chance Algorithm

- Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and A has its R bit set. The numbers above the pages are their load times.
- Reasonable but inefficient because of its constant maintenance of the list



Clock Page Replacement Algorithm

- Figure 3-16. The clock page replacement algorithm.



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Least Recently Used (LRU)

- A good approximation to the optimal algorithm
- based on page usage in the last few instructions
  - Those heavily used will probably be used again
- idea suggests a realizable algorithm
  - when a page fault occurs, evict the page that has been unused for the longest time
  - Not cheap to implement
  - To fully implement, a linked list of all pages in memory must be maintained with most recently used page at the front and the least recently used page at the rear

LRU with Special Hardware

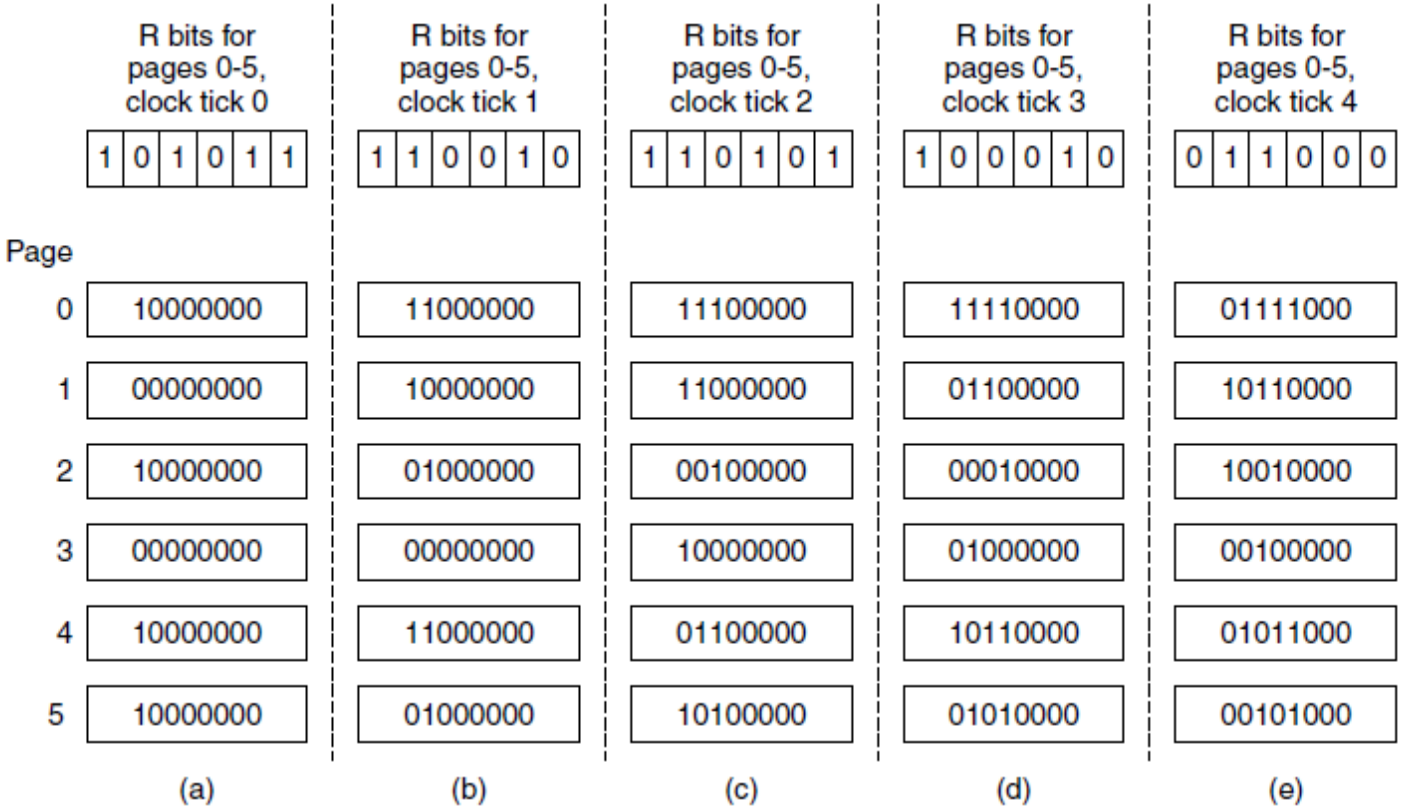
- Equip the hardware with a 64-bit counter
- Automatically increment after each instruction
- Add a field to the page table for this counter (again 64b)
- After each memory reference, this field is updated with the current count in the entry that was referenced
- When a page fault occurs, scan the table for the entry with the lowest count
- Few, if any, machines have the required hardware to implement this system
- It will also make the page table much larger

LRU with Software

- NFU (Not Frequently Used) (not to be confused with the NRU)
  - A software counter associated with each page
  - At each clock interrupt, the operating system scans all the pages in memory, For each page, the R bit is added to the counter
  - counters roughly keep track of how often each page has been referenced
  - At a page fault, the page with the lowest counter is evicted
  - Main problem, with NFU never forgets anything
  - A small modification:
    - the counters are each shifted right 1 bit before the R bit is added
    - the R bit is added to the leftmost rather than the rightmost bit
      - known as aging

Simulating LRU in Software

- Figure 3-17. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).



LRU with Software (NFU)

- This algorithm differs from LRU in two important ways
- Consider pages 3 and 5 in previous figure, neither has been referenced in two time periods; both were referenced the period prior, on a page fault one of those two should be evicted
  The problem, we don't know which one was most recently accessed between tick 1 and tick 2
- We've lost our earlier memory because of the bit shifting. There must be a limited number of bits devoted to this process
  In practice, however, 8 bits is generally enough if a clock tick is around 20 msec, if a page has not been referenced in 160 msec, it probably is not that important