Memory-Mapped Files & Virtual Memory

May 23, 2018
Byung-Gon Chun

Acknowlegments. Slides and/or picture in the following are adapted from UW, Columbia, and UC Berkeley slides

Memory Management

- Address Translation
  - Basic concept
  - Flexible
  - Efficient
- Caching
- Virtual memory

Memory-Mapped Files

Demand Paging

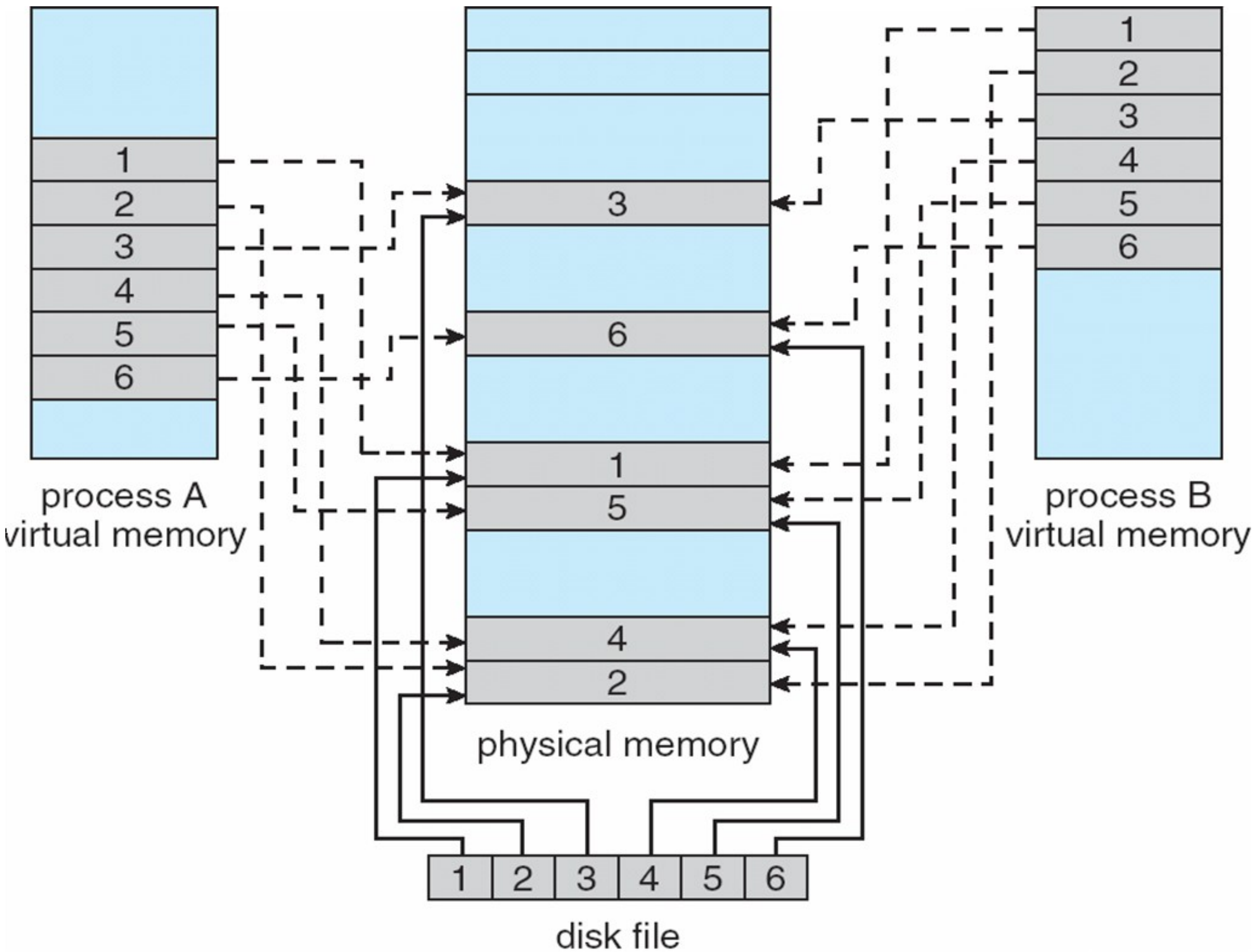
- With demand paging, applications can access more memory than is physically present on the machine, by using memory pages as a cache for disk blocks.

- When the application accesses a missing memory page, it is transparently brought in from disk.
  - Simpler case of a demand paging: a single, memory-mapped file
  - More complex case: managing multiple processes competing for space in main memory

Models for Application File I/O

- Explicit read/write system calls
  - Data copied to user process using system call
  - Application operates on data
  - Data copied back to kernel using system call
- Memory-mapped files
  - Open file as a memory segment
  - Program uses load/store instructions on segment memory, implicitly operating on the file
  - Page fault if portion of file is not yet in memory
  - Kernel brings missing blocks into memory, restarts process

Memory Mapped File Example



Advantages to Memory-mapped Files

- Programming simplicity, esp for large files
  - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
  - Data brought from disk directly into page frame
- Pipelining
  - Process can start working before all the pages are populated
- Interprocess communication
  - Shared memory segment vs. temporary file

Implementation

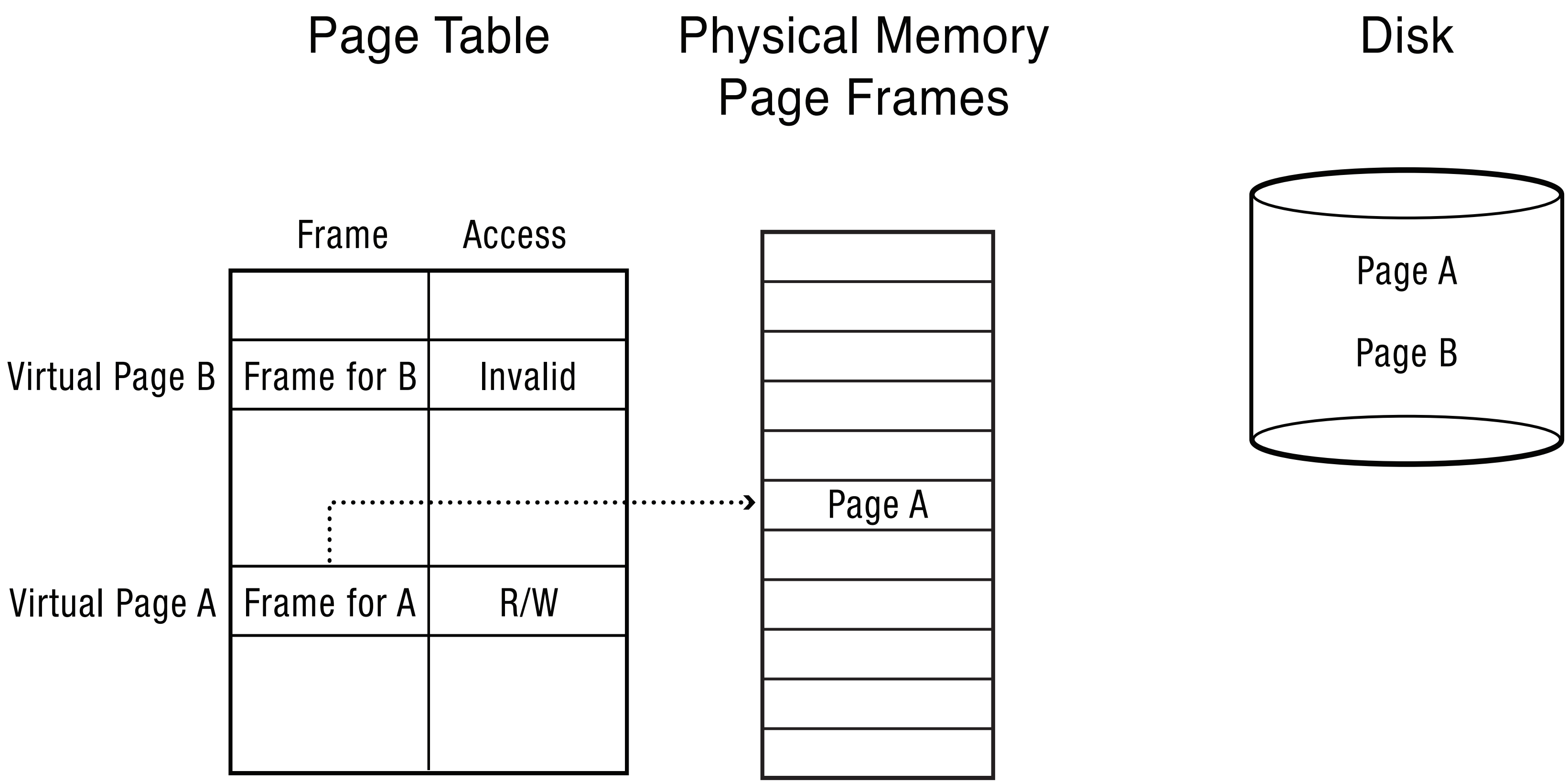
- Map the file into a portion of the virtual address space

- The kernel initializes a set of page table entries for that region of the virtual address space, setting each entry to invalid.

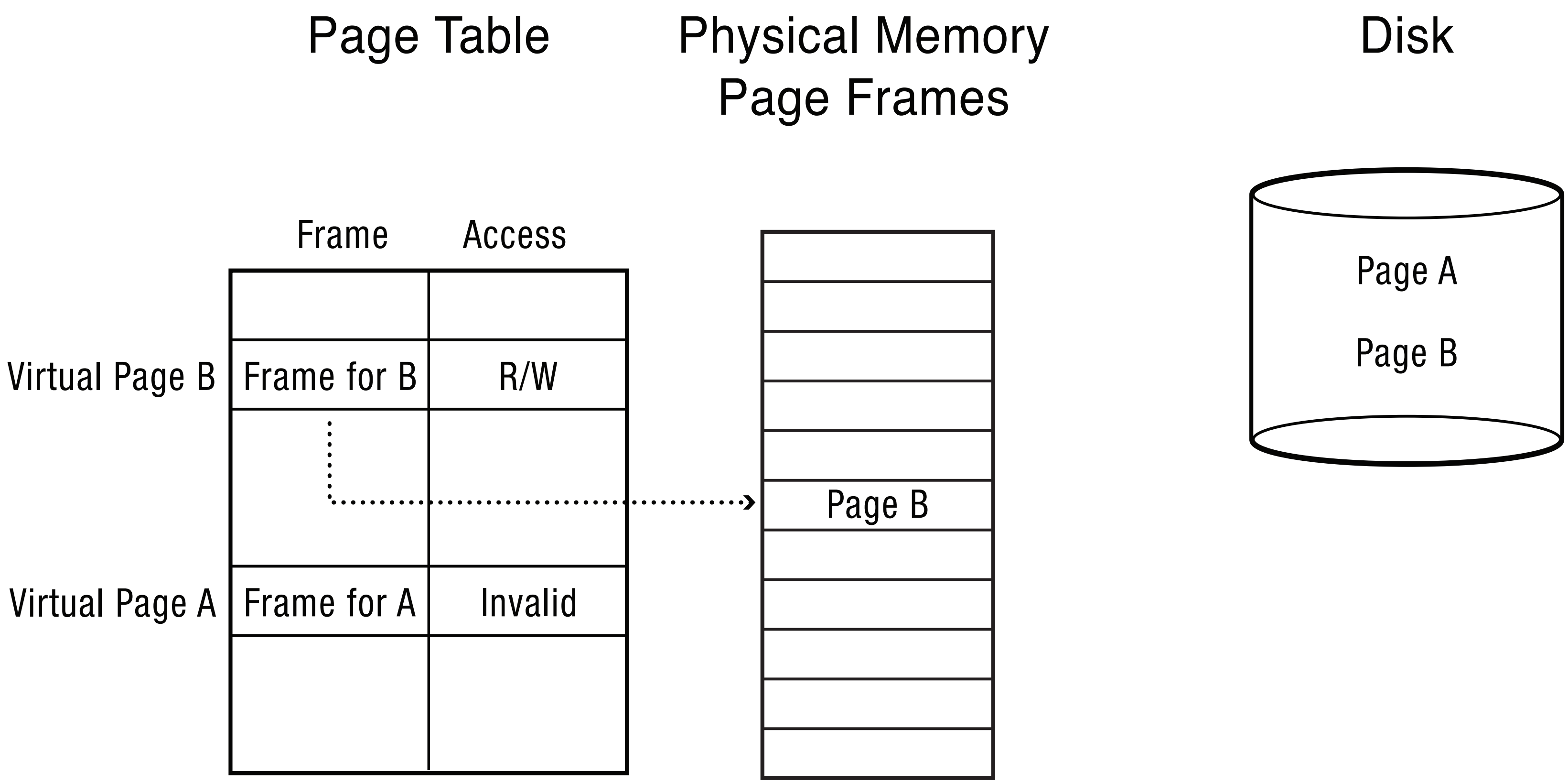
Implementation

- When the process issues an instruction that touches an invalid mapped address, a sequence of events occurs
  - TLB miss
  - Page table exception
  - Convert virtual address to file offset
  - Disk block read: allocate an empty page frame and issue a disk operation to read the required file block into the allocated page frame
  - Disk interrupt
  - Page table update
  - Resume process
  - TLB miss
  - Page table fetch

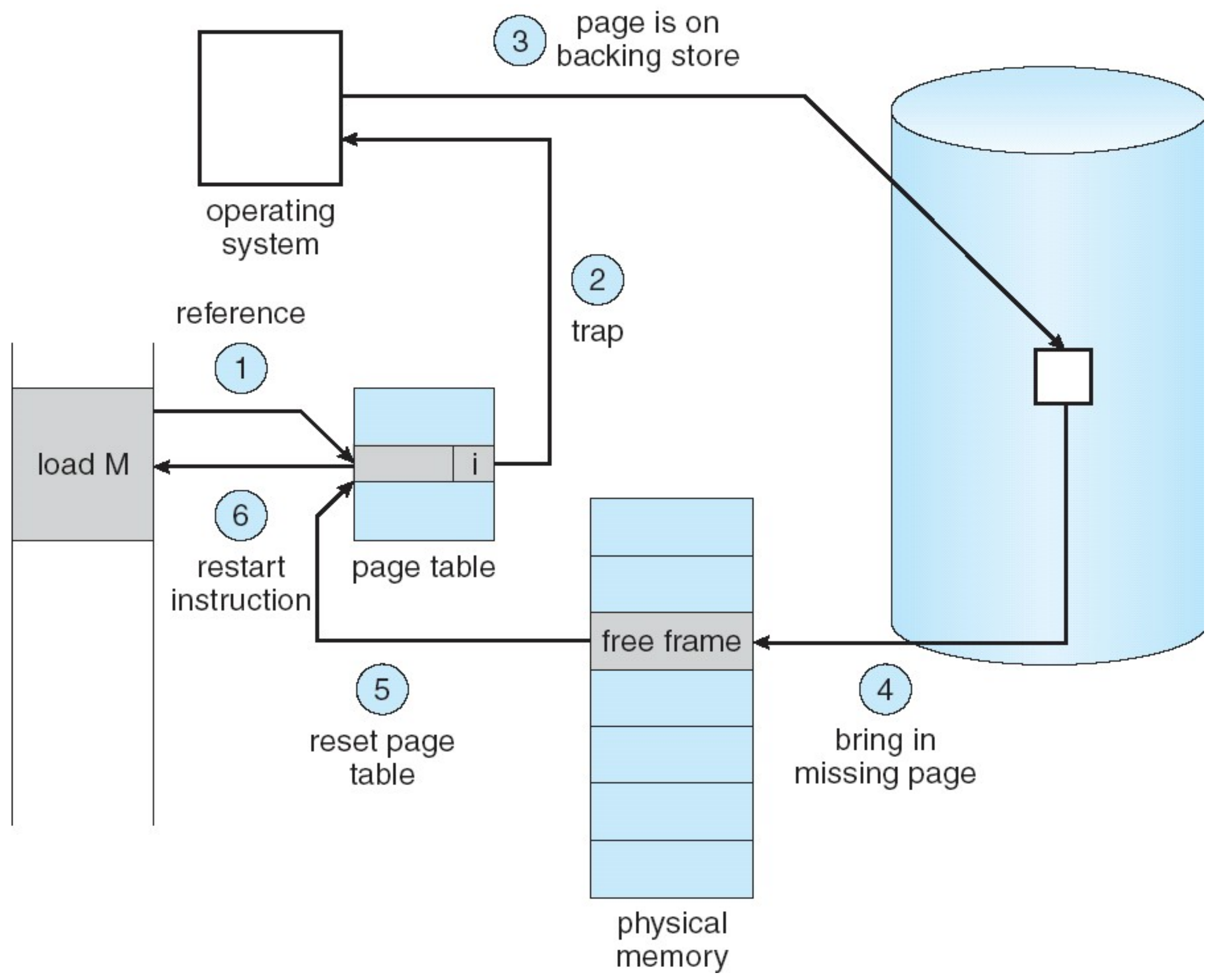
Demand Paging (Before)



Demand Paging (After)



Steps In Handling A Page Fault



Allocating a Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
  - If page frame is shared
  - Use a core map
- Set each page table entry to invalid
- Remove any TLB entries
  - Copies of now invalid page table entry
- Write changes on page back to disk, if the evicted page was modified

How Do We Know If Page Has Been Modified?

- Every page table entry has some bookkeeping
  - Has page been modified?
    - Set by hardware on store instruction
    - In both TLB and page table entry
  - Has page been recently used?
    - Set by hardware on in page table entry on every TLB miss

- Bookkeeping info can be reset by the OS kernel
  - When changes to page are flushed to disk
  - To track whether page is recently used

Virtual Memory

From Memory-Mapped Files to Demand-Paged Virtual Memory

- Every process segment backed by a file on disk
  - Code segment -> code portion of executable
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data file
  - Memory-mapped files -> memory-mapped files
  - When process ends, delete temp files
- Unified memory management across file buffer and process memory

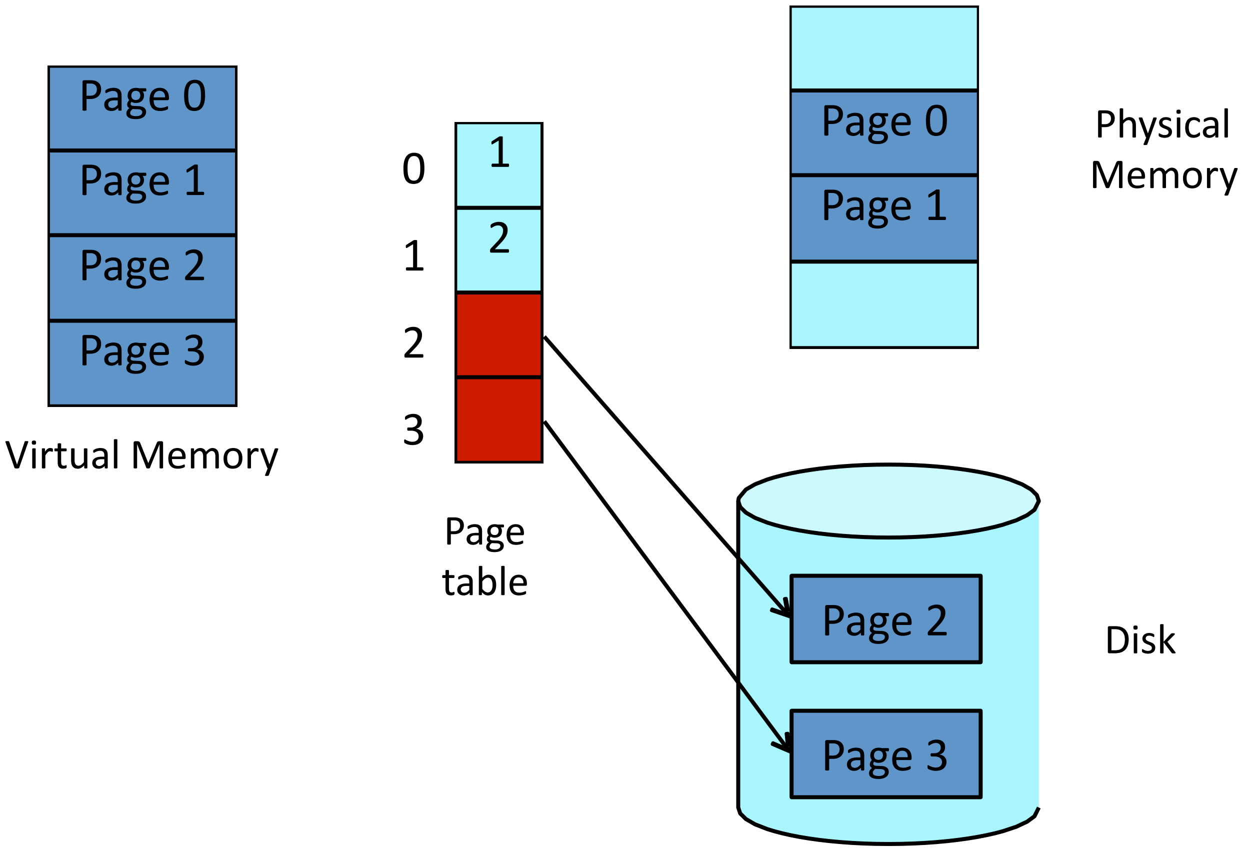
Virtual Memory Motivation

- Previous approach to memory management
  - Must completely load user process in memory
  - One large AS or too many ASes => out of memory
- Observation: locality of reference
  - Temporal: access memory location accessed just now
  - Spatial: access memory location adjacent to locations accessed just now
- Implication: process only needs a small part of address space at any moment!
  - Can load programs faster (don’t load everything)
  - Can fit more programs in memory (better utilization)

Virtual Memory Idea

- OS and hardware produce illusion of disk as fast as main memory, or main memory as large as disk
- Process runs when not all pages are loaded in memory
  - Only keep referenced pages in main memory
  - Keep unreferenced pages on slower, cheaper backing store (disk)
  - Bring pages from disk to memory when necessary

Virtual Memory Illustration



Virtual Memory Operations

- Detect reference to page on disk
- Recognize disk location of page
- Choose free physical page
  - OS decision: if no free page is available, must replace a physical page
- Bring page from disk into memory
  - OS decision: when to bring page into memory?
- Above steps need hardware and software cooperation

Detect Reference to Page on Disk and Recognize Disk Location of Page

- Overload the present bit of page table entries
- If a page is on disk, clear present bit in corresponding page table entry and store disk location using remaining bits
- Page fault: if bit is cleared then referencing resulting in a trap into OS
- In OS page fault handler, check page table entry to detect if page fault is caused by reference to true invalid page or page on disk

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$
\begin{aligned}
EAT = (1 - p) \text{ x memory access} \\
+ p \text{ (page fault overhead} \\
+ \text{swap page out} \\
+ \text{swap page in} \\
+ \text{restart overhead)}
\end{aligned}
$$

Demand Paging Example

- Disparity in memory and disk access times is huge. E.g.,
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one out of 1,000 accesses faults, then EAT = 8.2 us, or 40x slower!
- If want performance degradation < 10 percent
  - 200 + 7,999,800 x p < 220, or 7,999,800 x p < 20
  - p < .0000025
  - Less than one page fault in every 400,000 memory accesses

OS Decisions

- Page selection
  - When to bring pages from disk to memory?
- Page replacement
  - When no free pages available, must select victim page in memory and throw it out to disk

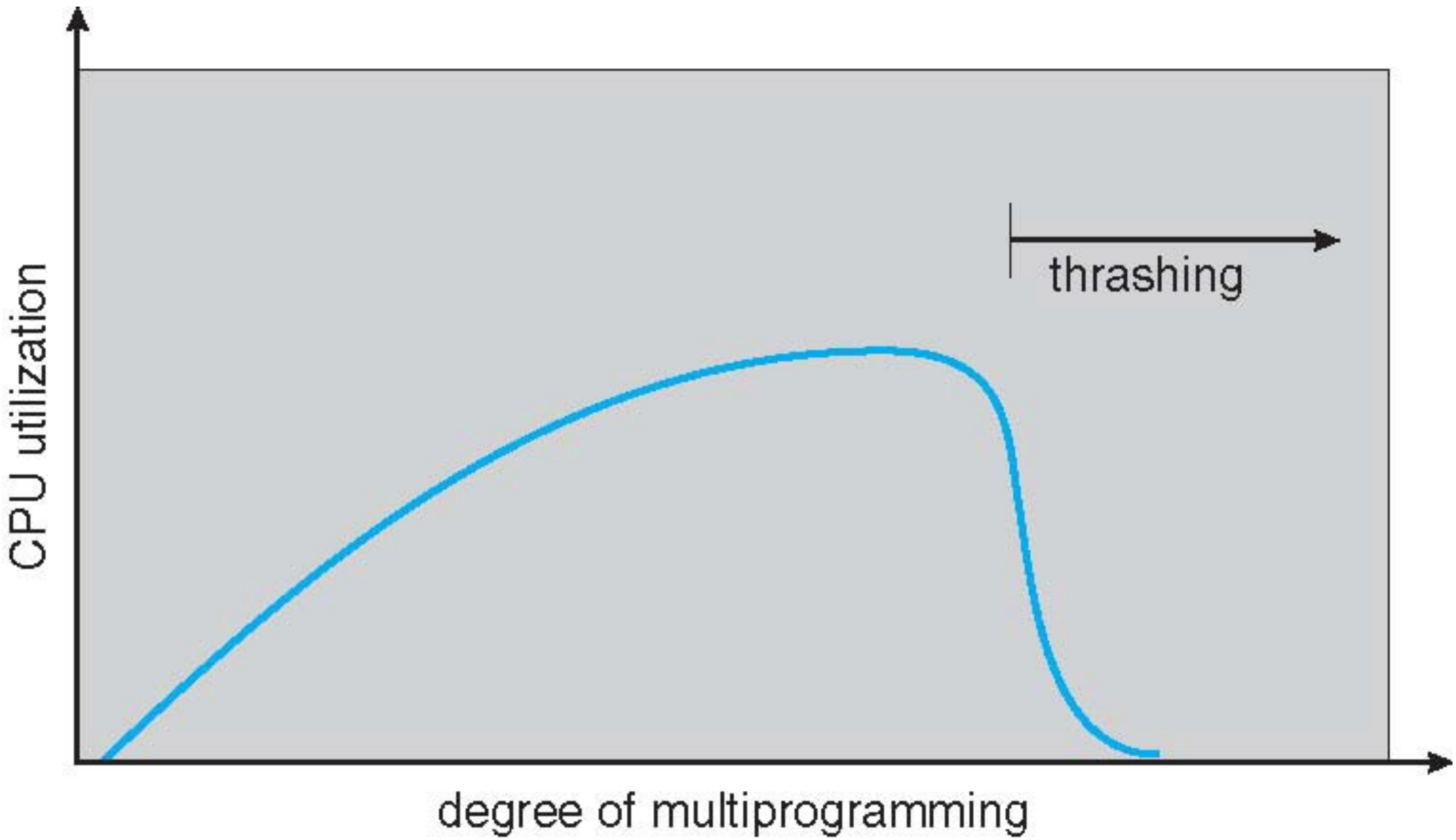
Page Selection Algorithms

- Demand paging: load page on page fault
  - Start up process with no pages loaded
  - Wait until a page absolutely must be in memory
- Request paging: user specifies which pages are needed
  - Requires users to manage memory by hand
  - Users do not always know best
  - OS trusts users (e.g., one user can use up all memory)
- Prepaging: load page before it is referenced
  - When one page is referenced, bring in next one
  - Do not work well for all workloads
    - Difficult to predict future

Thrashing

- What if we need more pages regularly than we have?
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
- Leads to:
  - High page fault rate
  - Lots of I/O wait
  - Low CPU utilization
  - No useful work done
- Thrashing ≡ system busy just swapping pages in and out

Effects of Thrashing



Page Replacement
Implementing LRU (Take 1): Hardware

- A counter for each page

- Every time page is referenced, save system clock (time) into the counter of the page

- Page replacement: scan through pages to find the one with the oldest clock

- Problem: have to search all pages/counters!

Implementing LRU (Take 2): Software

- A doubly linked list of pages
- Every time page is referenced, move it to the front of the list
- Page replacement: remove the page from back of list
  - Avoid scanning of all pages
- Problem: too expensive
  - Requires 6 pointer updates for each page reference
  - High contention on multiprocessor

LRU Concept vs. Reality

- LRU is considered to be a reasonably good algorithm
- Problem is in implementing it efficiently
  - Hardware implementation: counter per page, copied per memory reference, have to search pages on page replacement to find oldest
  - Software implementation: no search, but pointer swap on each memory reference, high contention
- In practice, settle for efficient approximate LRU
  - Find an old page, but not necessarily the oldest
  - LRU is approximation anyway, so approximate more

Clock (Second-Chance) Algorithm

- Goal: remove a page that has not been referenced recently
  - good LRU approximate algorithm

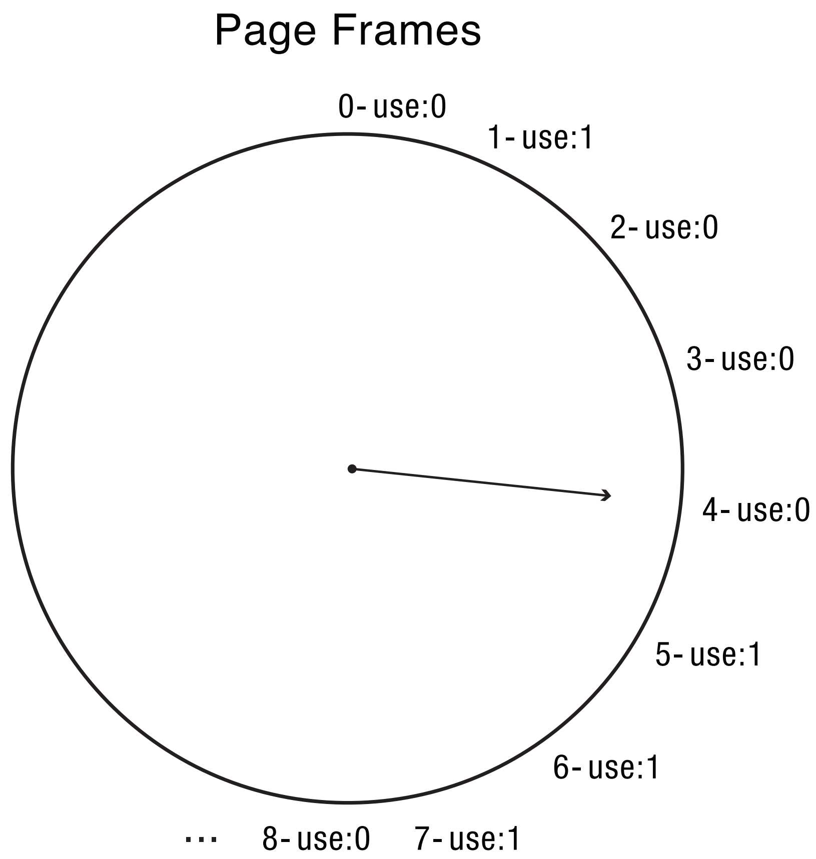
- Idea
  - A reference bit per page
  - Memory reference: hardware sets bit to 1
  - Page replacement: OS finds a page with reference bit cleared
  - OS traverses all pages, clearing bits over time

Clock Algorithm Implementation

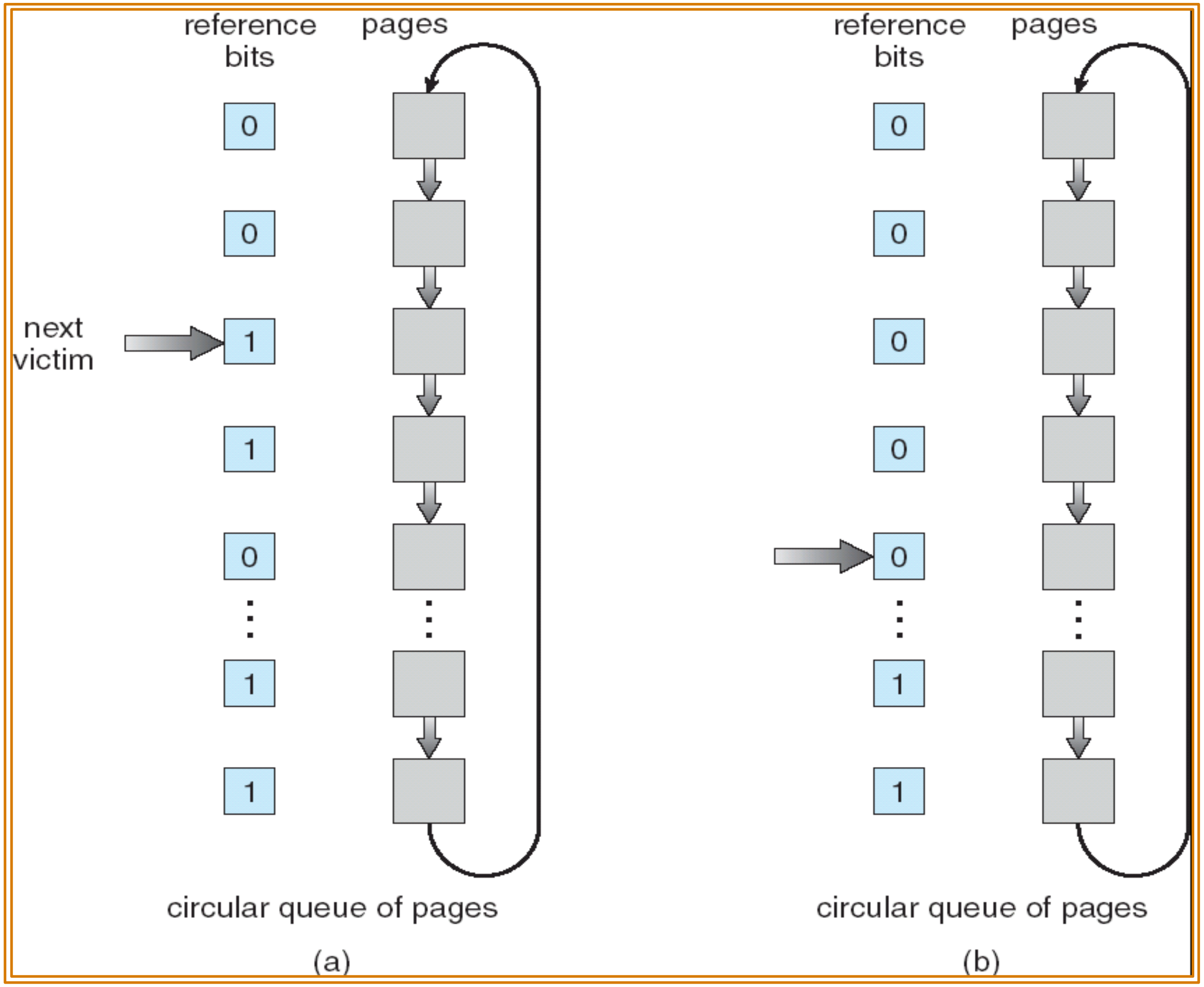
- Combining FIFO with LRU: give the victim page that FIFO selects a second chance
- Keep pages in a circular list = clock
- Pointer to next victim = clock hand
- To replace a page, OS examines the page pointed to by hand
  - If ref bit == 1, clear, advance hand
  - Else return current page as victim

Clock Algorithm (aka Second Chance Algorithm): Approximating LRU

- Periodically, sweep through all pages
- If page is unused, reclaim
- If page is used, mark as unused



A Single Step in Clock Algorithm



Least Recently Used (LRU) Algorithm

- Throw out page that hasn't been used in longest time. Can use FIFO to break ties

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	1	1	5
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	5	5	5	5	4	4
			4	4	4	4	4	4	3	3	3

8 page faults

Advantage: with locality, LRU approximates Optimal

Clock Algorithm Example

1	2	3	4	1	2	5	1	2	3	4	5
1 1	1 1	1 1	1 1	1 1	1 1	5 1	5 1	5 1	5 1	4 1	4 1
	2 1	2 1	2 1	2 1	2 1	2 0	1 1	1 1	1 1	1 0	5 1
		3 1	3 1	3 1	3 1	3 0	3 0	2 1	2 1	2 0	2 0
			4 1	4 1	4 1	4 0	4 0	4 0	3 1	3 0	3 0

10 page faults

Advantage: simple to implement!

Clock Algorithm Extension

- Problem of clock algorithm: does not differentiate dirty v.s. clean pages

- Dirty page: pages that have been modified and need to be written back to disk
  - More expensive to replace dirty than clean pages
  - One extra disk write (about a few ms)

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Clock Algorithm Extension (Cont.)

- Use dirty bit to give preference to dirty pages
- On page reference
  - Read: hardware sets reference bit
  - Write: hardware sets dirty bit
- Page replacement
  - reference = 0, dirty = 0 => victim page
  - reference = 0, dirty = 1 => skip (don't change)
  - reference = 1, dirty = 0 => reference = 0, dirty = 0
  - reference = 1, dirty = 1 => reference = 0, dirty = 1
  - advance hand, repeat
  - If no victim page found, run swap daemon to flush unreferenced dirty pages to the disk, repeat