9. Virtual Memory Management

ECE30021/ITP30002 Operating Systems

Agenda

- Background
- Demand paging
- Copy-on-write
- Page replacement
- Allocation of frames
- Thrashing
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- Operation-system examples

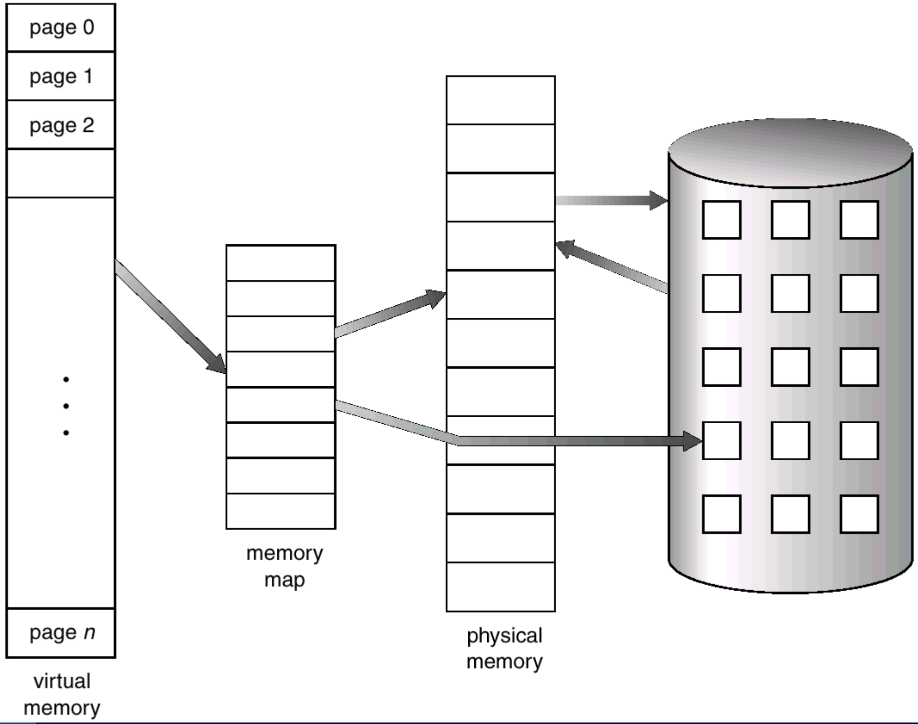
Background

- If we can run a program by loading in parts …
  - A program is not constrained by the amount of physical memory
  - More program can run at the same time
  - Less I/O is need to load or swap programs

Virtual Memory

- Virtual memory: a separation of logical memory from physical memory
  - Contents not currently reside in main memory can be addressed.

    ➔ H/W and OS will load the required memory from auxiliary storage automatically.
  - User programs can reference more memory than actually exists

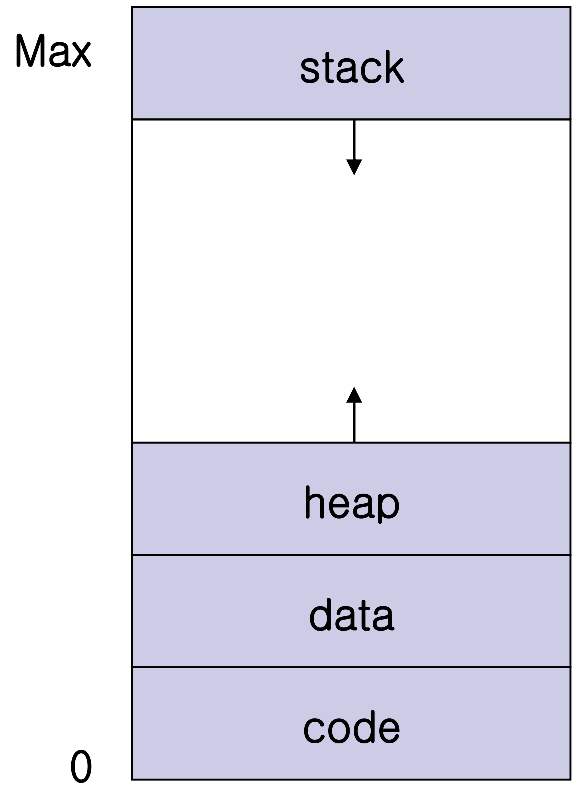


Virtual Memory

- Virtual memory (cont.)
  - Programmers don't have to concern about memory management
  - Virtual address space can be sparse

- Sparse address space
  - Address space can include holes

  Ex) Stack and heap can grow



<virtual address space>

Virtual Memory

- Additional benefits
  - An address space can be shared by many processes.
  - Efficient in process creation by fork()
    - Copy on Write (COW)

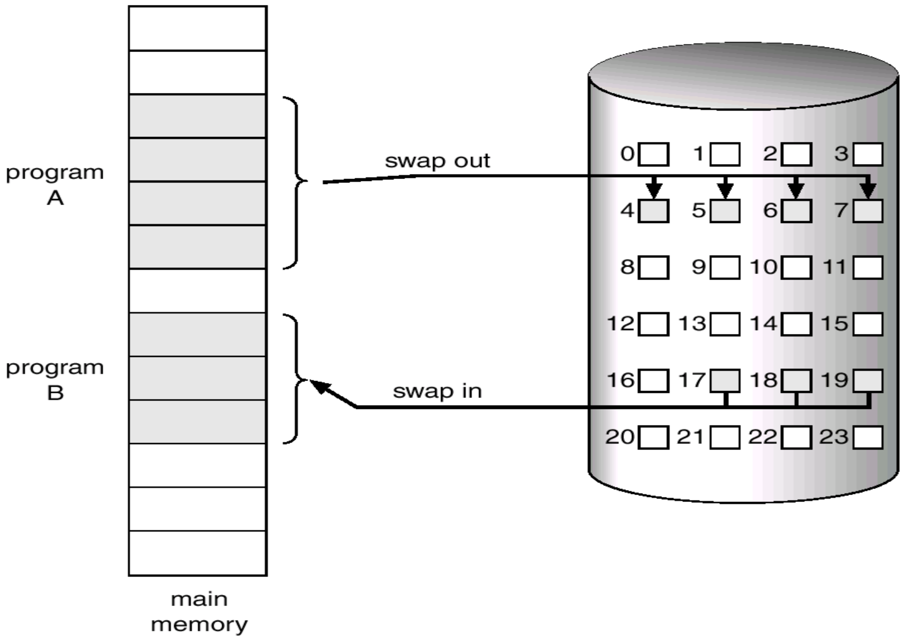
- Implemented by …
  - Demand paging
  - Demand segmentation

Agenda

- Background
- <u>Demand paging</u>
- Copy-on-write
- Page replacement
- Allocation of frames
- Thrashing
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- Operation-system examples

Demand Paging

- Similar to paging system with swapping
- Lazy swapper (or pager)
  - Pages are only loaded when they are demanded during execution
  - A swapper that never swaps a page unless it will be needed

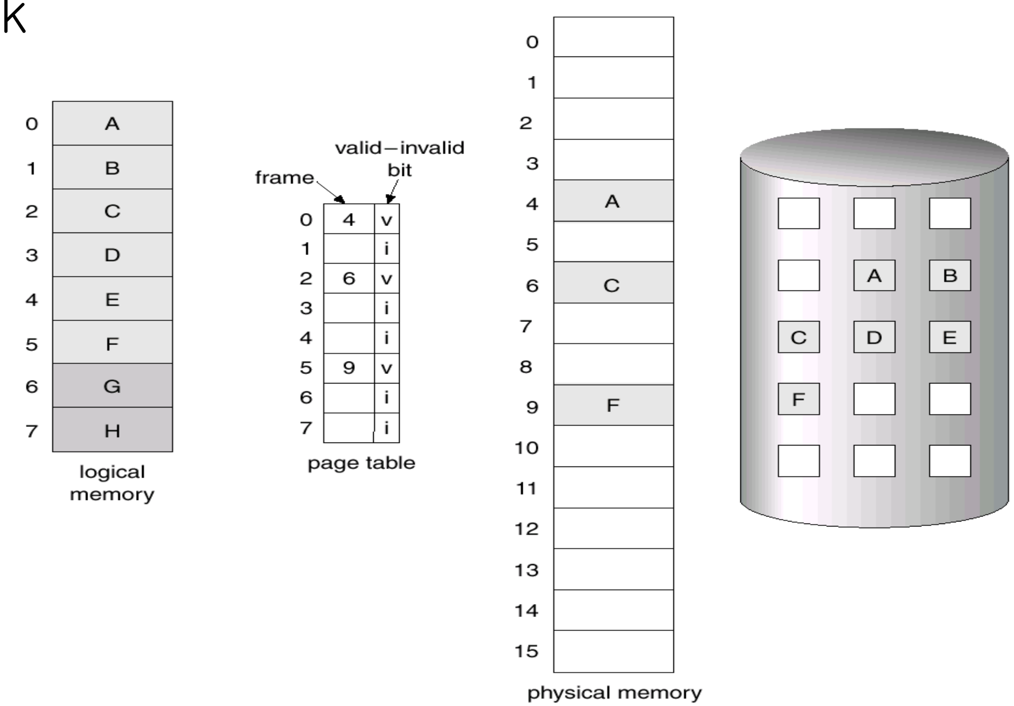


Basic Concepts

- When a process is to be swapped in, pager guesses which pages will be used.

-> Only those pages are swapped into memory.

  - Requires H/W support to distinguish the page on memory or on disk



Basic Concepts

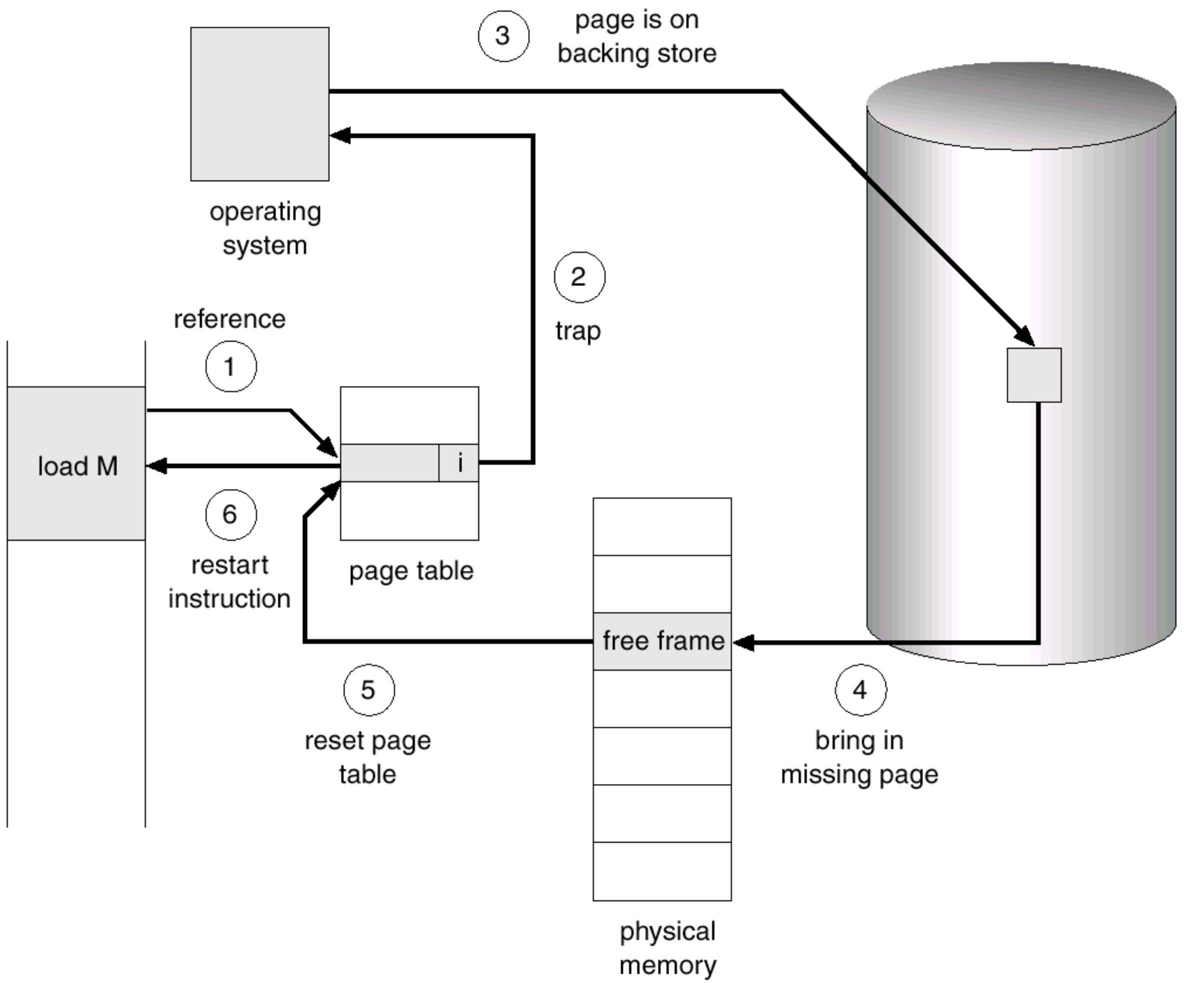
- Valid/invalid bit of each page
  - Valid: page is valid and exists in physical memory
  - Invalid: page is invalid or not exist in physical memory

- If program tries to access ..
  - Valid page: execution proceeds normally
  - Invalid page: cause page-fault trap to OS

Basic Concepts

- Handling page-fault
  - Check an internal table to determine whether the reference was valid or not
  - If the reference was invalid, terminates the process
  - If valid, page it in.
    - Find a free frame
    - Read desired page into the free frame
    - Modify internal table
    - Restart the instruction that caused the page-fault trap

Handling Page-Fault



Basic Concepts

- In pure demand paging, a page is not brought into memory until it is required

- Performance degradation
  - Theoretically, some program can cause multiple page faults per instructions
  - But actually, this behavior is exceedingly unlikely.
    - locality of references

- H/W supports for demand paging
  - Page table (support for valid-invalid bit, …)
  - Secondary memory (swap space)

Performance of Demand Paging

- Effective access time

  Effective access time = (1−*p*)**ma* + *p* * <page fault time>

  - *ma* : memory access time (10~200 nano sec.)
  - *p* : probability of page fault

- Page fault time
  - Service page-fault interrupt -> 1~100 μsec.
  - Read in the page -> about 8 msecs
  - Restart the process -> 1~100 μsec.

Performance of Demand Paging

Effective access time = $(1-p)*ma + p *$ <page fault time>

- Example
  - Memory access time: 200 nano sec
  - Page-fault service time: 8 milliseconds

- Then…

  Effective access time (in nano sec.)

  $= (1-p) * 200 + 8{,}000{,}000 * p$

  $\approx 200 + 7{,}999{,}800 * p$

  - Proportional to **page fault rate**

  Ex) p == 1/1000, effective access time = 8.2 μsec. (40 times)

  - Page fault rate should be kept low

Execution of Program in File System

- Ways to execute a program in file system
  - Option1: copy entire file into swap space at starting time
    - Usually swap space is faster than file system
  - Option2: initially, demand pages from files system and all subsequent paging can be done from swap space
    - Only needed pages are read from file system

Copy-on-Write

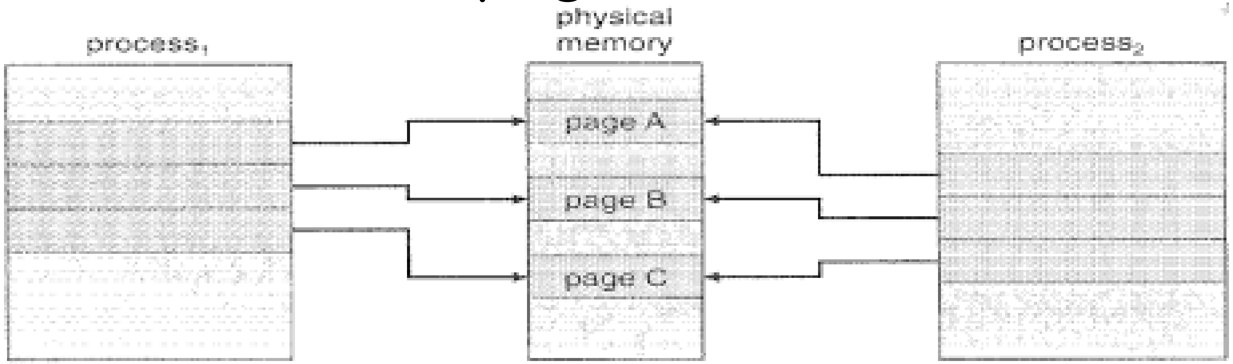
- fork() copies process
  - Duplicates pages belong to the parent
- Copy-on-write (COW)
  - When the process is created, pages are not actually duplicated but just shared.
    - Process creation time is reduced.
  - When either process writes to a shared page, a copy of the page is created.

  cf. vfork() – logically shares memory with parent (obsolete)
- Many OS's provides a pool of free pages for COW or stack/heap that can be expanded

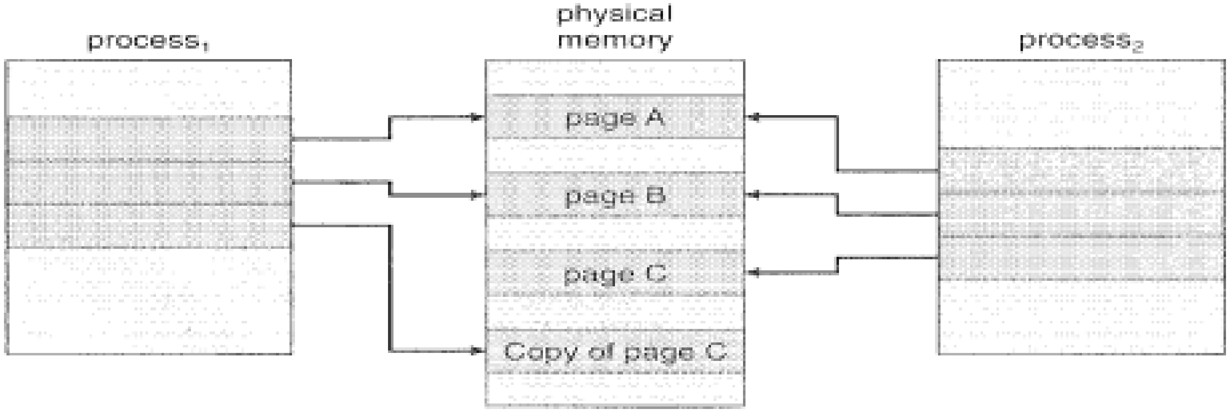
  ➔ Zero-fill-on-demand (ZFOD)
    - Zero-out pages before being assigned to a process

Copy-on-Write

- Before P1 modifies page C



- After P1 modifies page C



Agenda

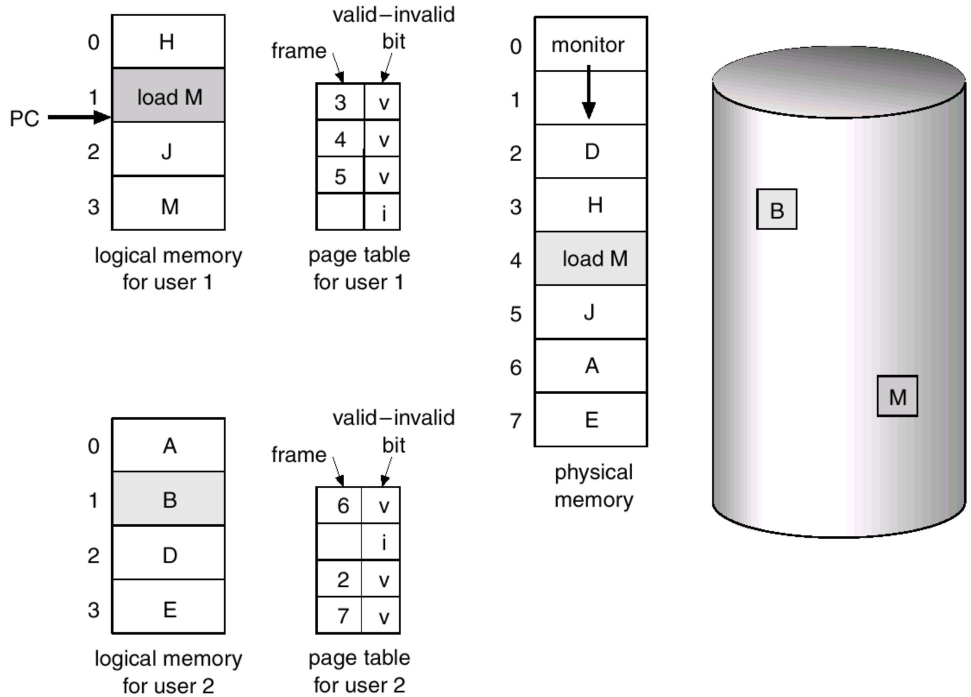
- Background
- Demand paging
- Copy-on-write
- <u>Page replacement</u>
- <u>Allocation of frames</u>
- Thrashing
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- Operation-system examples

Two major problems

- Two major problems in demand paging
  - Page-replacement algorithm
  - Frame-allocation algorithms
- Even slight improvement can yield large gain in performance.

Page Replacement

- Page replacement
  - If no frame is free at a page fault, we find a frame not being used currently, and swap out
  - Writing overhead can be reduced by modify-bit (or dirty-bit) for each frame

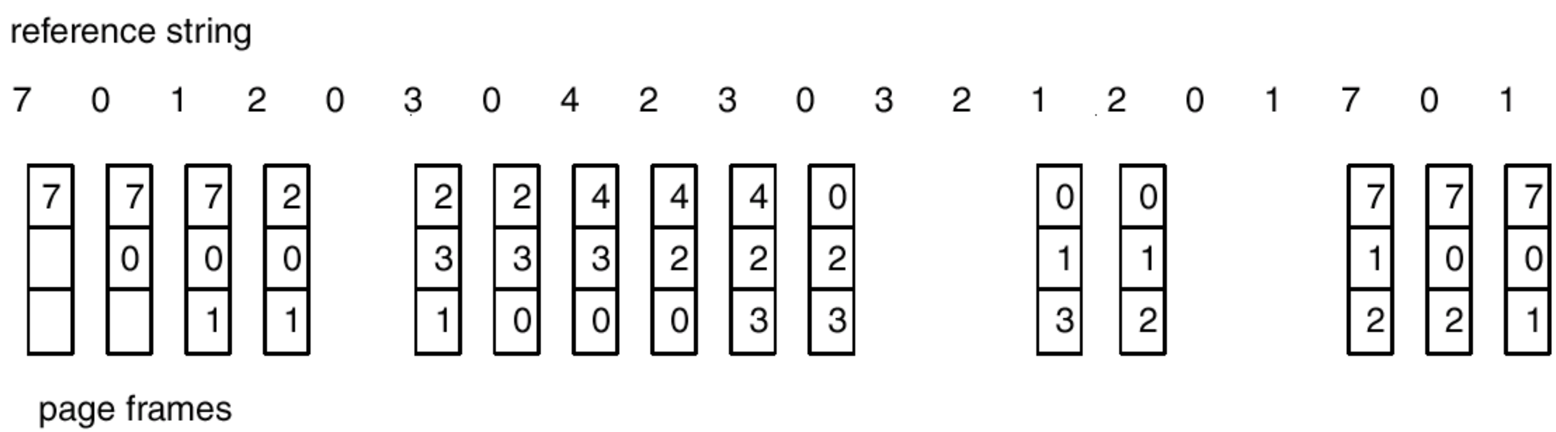


Page Replacement Algorithms

- FIFO page replacement
- Optimal page replacement (in theory)
- Least-recently-used (LRU) page replacement
- LRU-approximation page replacement
- ETC.

FIFO Page Replacement

- First-in, first-out: when a page should be replaced, the oldest page is chosen.
  - Easy, but not always good

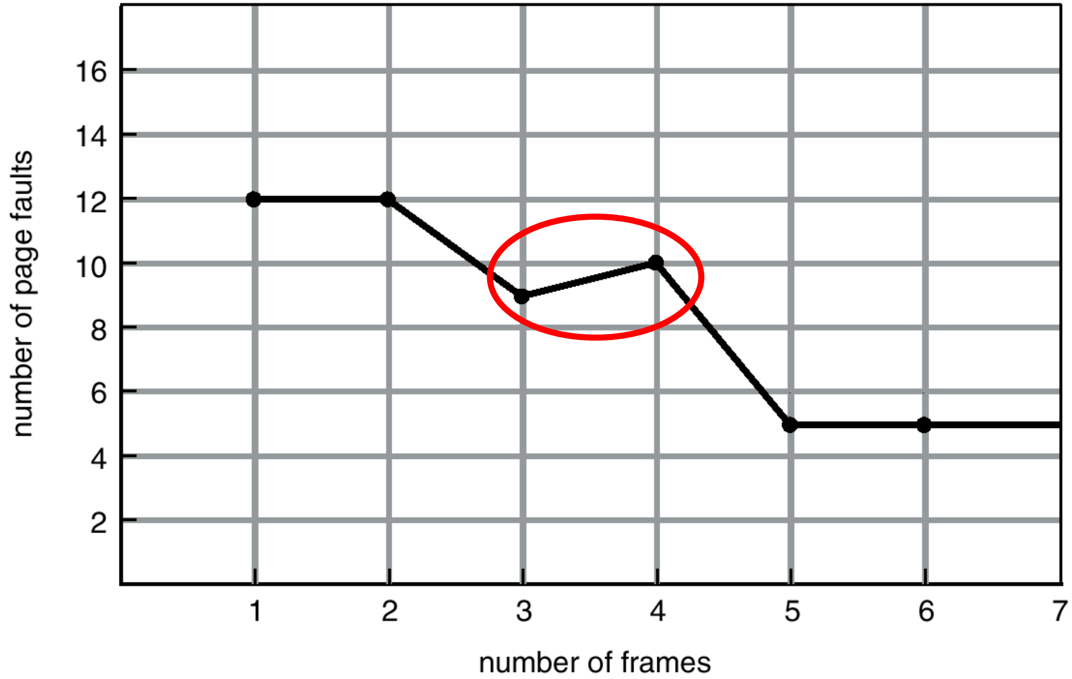


  - # of page faults: 15
- Problem: Belady's anomaly

FIFO Page Replacement

- Belady's anomaly: # of faults for 4 frames is greater than # of faults for 3 frames

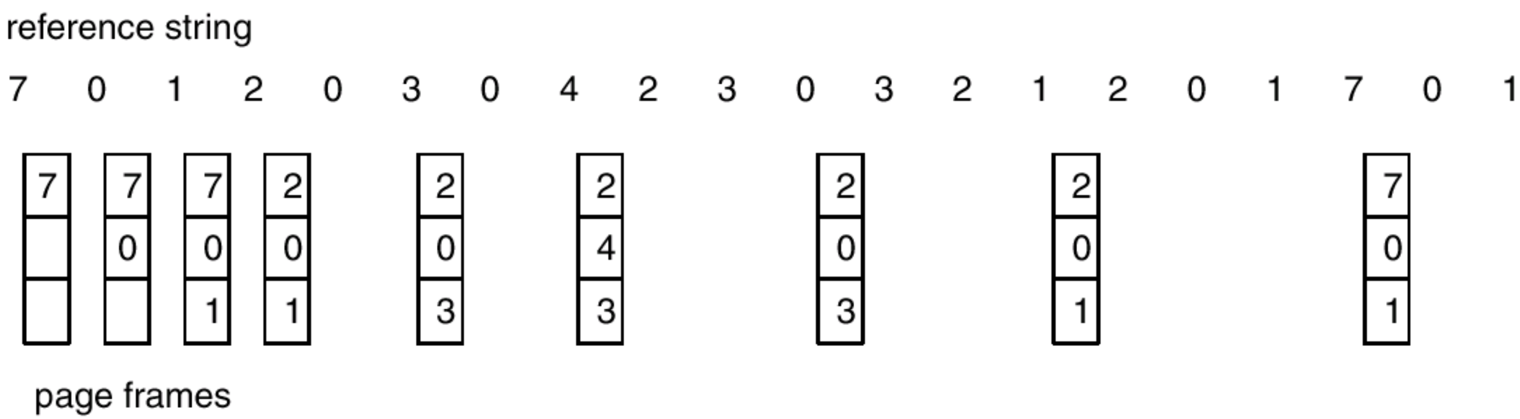
  (Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5)



Page-fault rate may increase as the number of frames increase.

Optimal Page Replacement

- Replace the page that will not be used for the longest period of time

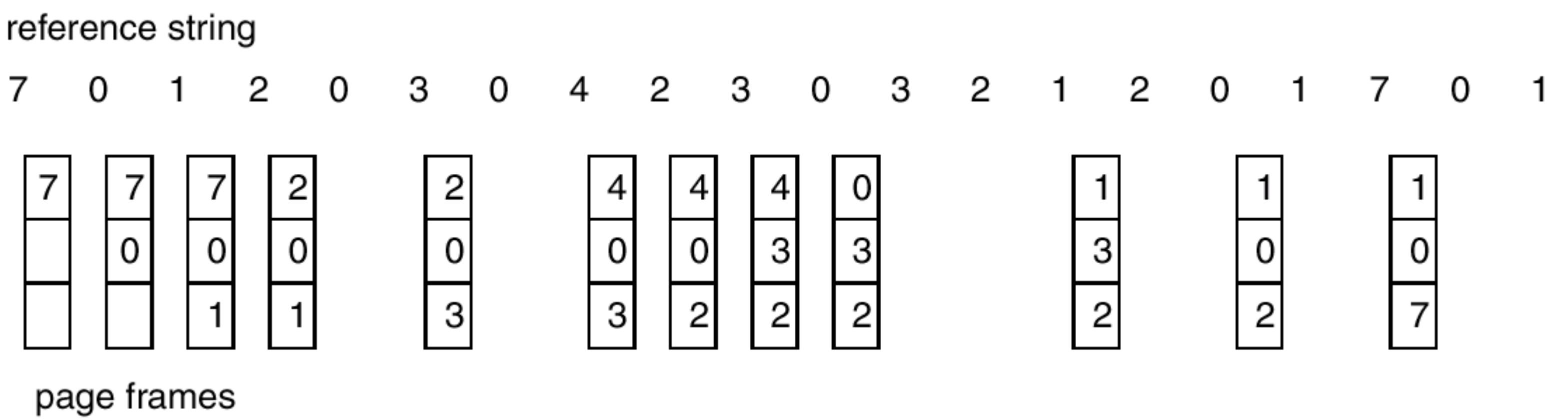


  - # of page faults: 9

- Problem: It requires future knowledge

LRU Page Replacement

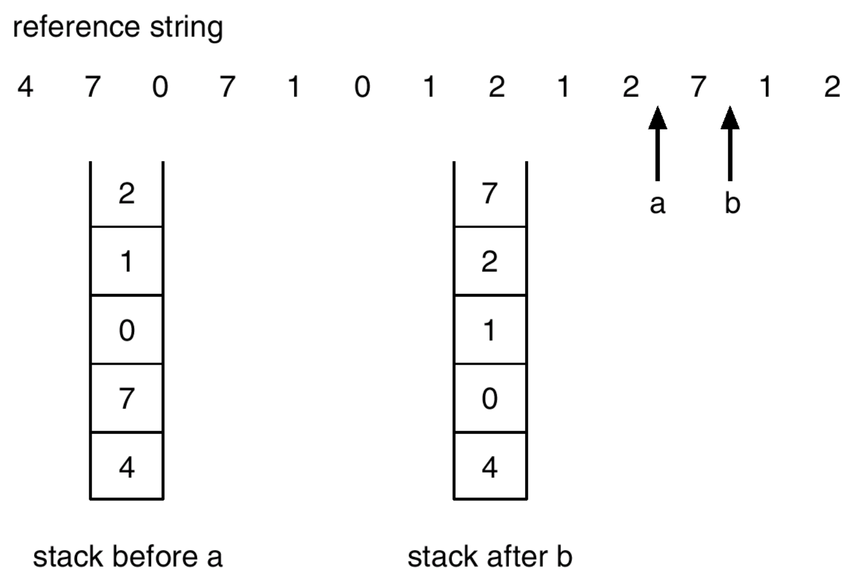
- LRU (Least Recently Used): replace page that has not been used for longest period of time



  - # of page faults: 12
  - LRU is considered to be good and used frequently

Implementation of LRU

- Using counter (logical clock)
  - Associate with each page-table entry a time-of-used field
  - Whenever a page is referenced, clock register is copied to its time-of-used field
- Using stack of page numbers
  - If a page is referenced, remove it and put on the top of the stack



Stack Algorithm

- Does LRU cause the Belady's anomaly?
- Stack algorithm: an algorithm for which the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n+1 frames.
  - Never exhibit Belady's anomaly
  - LRU is a stack algorithm

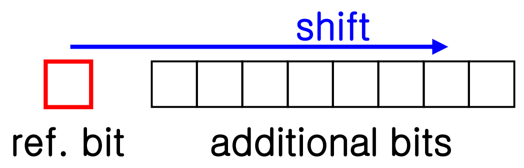
LRU-Approximation Page Replacement

- Motivation
  - LRU algorithm is good, but few system provide sufficient supports for LRU
  - However, many systems support reference bit for each page
    - We can determine which pages have been referenced, but not their order.
- LRU-approximation algorithms
  - Additional-reference-bit algorithm
  - Second-chance algorithm

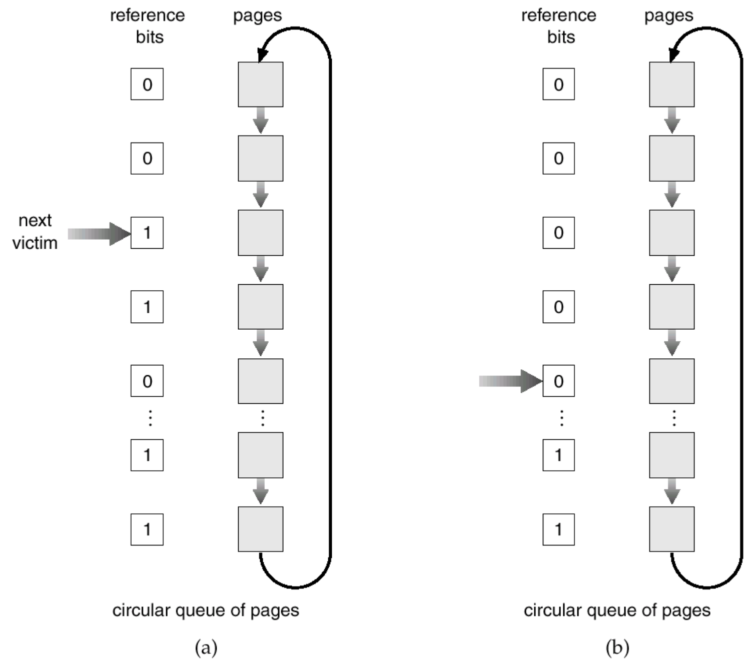
LRU-Approximation Page Replacement

- Additional-reference-bit algorithm
  - Gain additional ordering information by recording the reference bits at regular intervals.

    Ex) keep 8-bit byte for each page and records current state of reference bits of each page



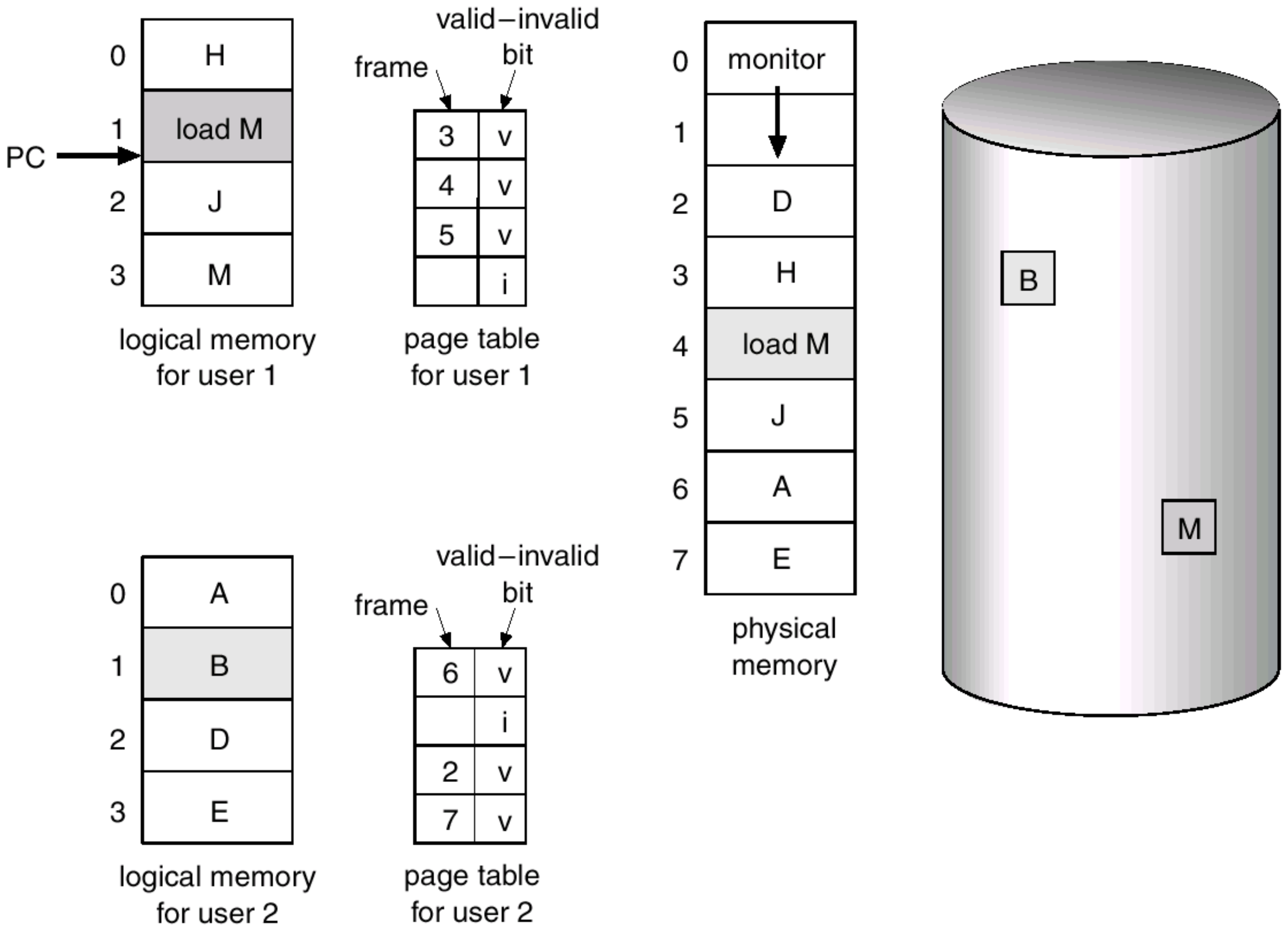
- Second-chance algorithm
  - Basically, FIFO algorithm
  - If reference bit of the chosen page is 1, give a second chance
    - The reference bit is cleared.



Page-Buffering Algorithms

- Pool of free frames
  - Some system maintain a list of free frames.
  - When a page fault occurs, a victim frame is chosen from the pool.
  - Frame number of each free page can be kept for next use.
- Keep a list of free frames and remember which page was in each frame.
  - The old page can be reused.
  - OS typically applies ZFOD(zero-fill on demand) technique.
- List of modified pages
  - Whenever the page device is idle, a modified page is written to disk.

Page-Buffering Algorithms



Allocation of Frames

- How do we allocate the fixed amount of free memory among various processes?
  - How many frames does each process get?

- Minimum number of frames for each process
  - # of frames for each process decreases
    -> page-fault rate is increases
    -> performance degradation
  - Minimum # of frames should be large enough to hold all different pages that any single instruction can reference.

Allocation Algorithms

- Equal allocation
  - Split m frames among n processes

  -> m/n frames for each process

- Proportional allocation
  - Allocate available memory to each process according to its size

    $$a_i = s_i/S * m$$

    - a_i: # of frames allocated to process p_i
    - s_i: size of process p_i
    - $S = \Sigma s_i$
  - Variation: frame allocation based on …
    - Priority of process
    - Combination of size and priority

Global vs. Local Allocation

- Global replacement: a process can select a replacement frame from the set of all frames, including frames allocated to other processes
  - A process cannot control its own page-fault rate
- Local replacement: # of frames for a process does not change
  - Less used pages of memory can't be used by other process

-> global replacement is more common method.

Agenda

- Background
- Demand paging
- Copy-on-write
- Page replacement
- Allocation of frames
- <u>Thrashing</u>
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- Operation-system examples

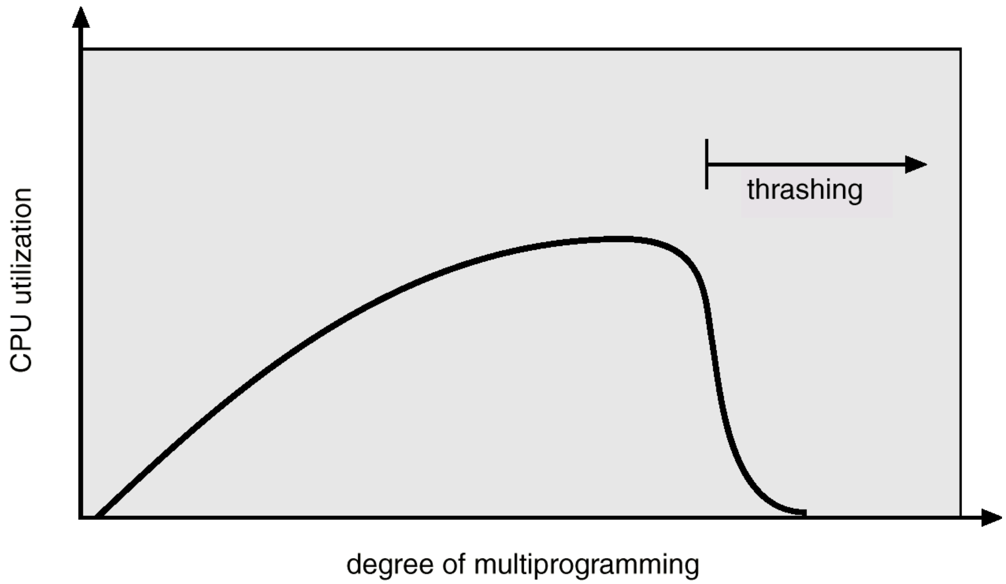
Thrashing

- If a process does not have enough frames to support pages in active use, it quickly faults again and again.

- A process is thrashing if it is spending more time paging than executing.

Trashing

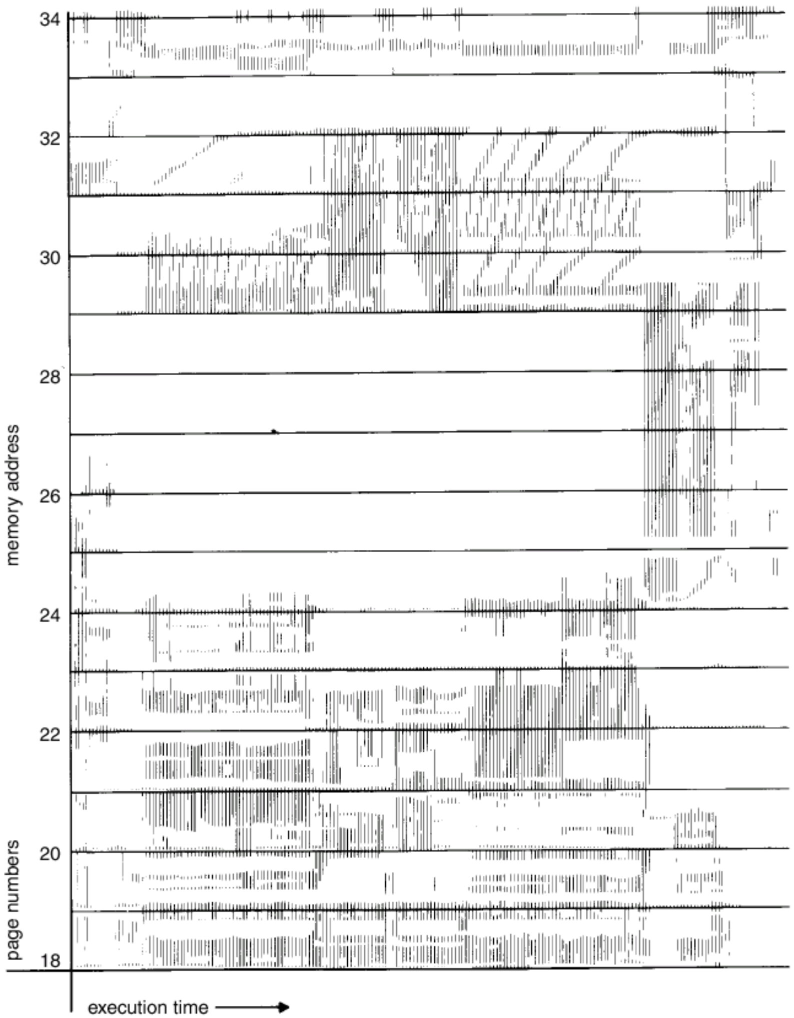
- If trashing sets in, CPU utilization drops sharply.
  - Degree of multiprogramming should be decreased



Trashing

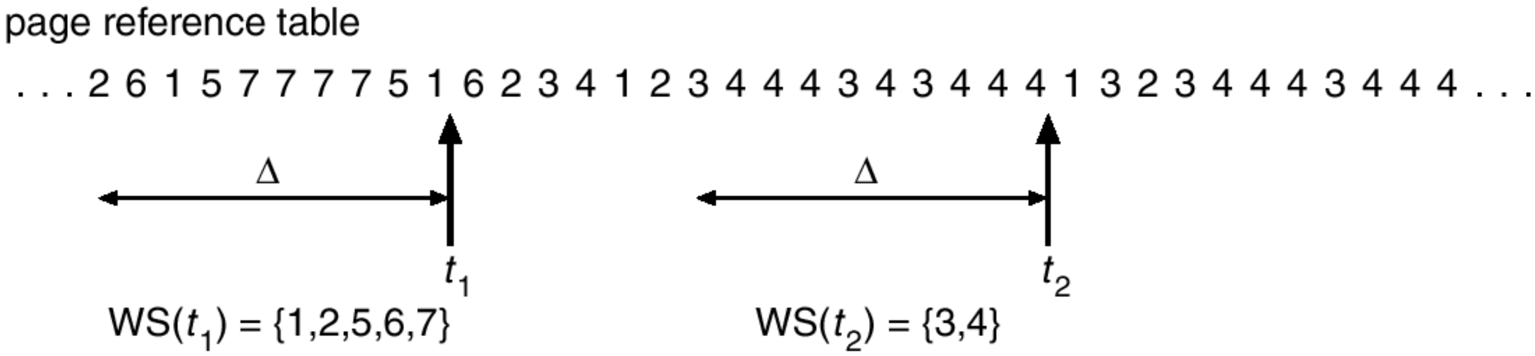
- To prevent thrashing, a process must be provided with as many frames as it needs.
  - -> How to know how many frames it needs?

- Locality model
  - Locality: set of pages actively used together
  - A program is generally composed of several localities



Working-Set Model

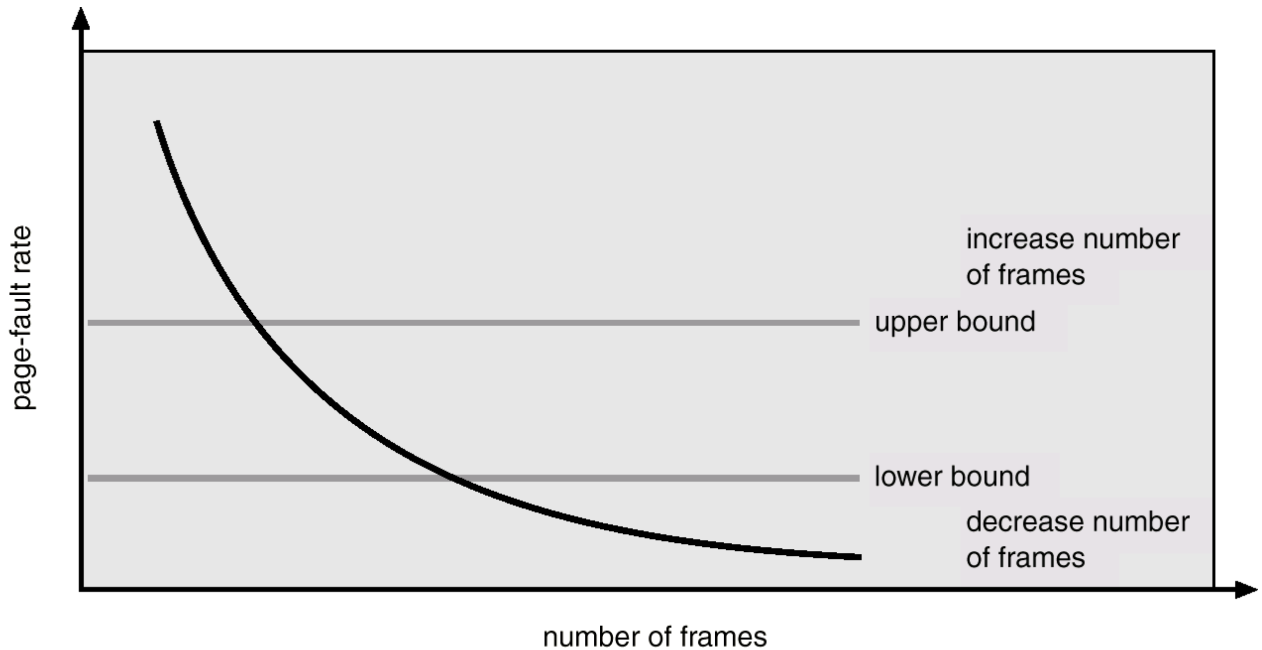
- Working set: set of pages in the most recent Δ page references
  - Parameter Δ: working-set window



  - WSS_i: working set size of process p_i
  - Process p_i needs WSS_i frames
- If total demand is greater than # of available frames, thrashing will occur.

Page-Fault Frequency

- Alternative method to control trashing: control degree of multiprogramming by page-fault frequency (PFF)
  - If PFF of a process is too high, allocate more frame
  - If PFF of a process is too low, remove a frame from it



Agenda

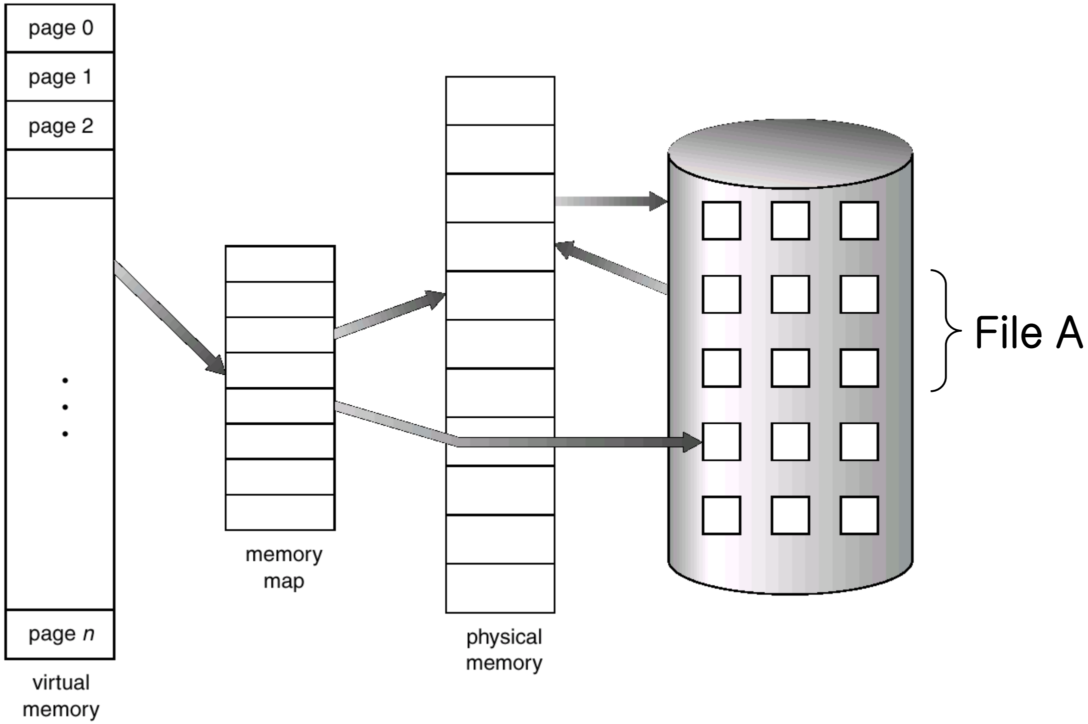
- Background
- Demand paging
- Copy-on-write
- Page replacement
- Allocation of frames
- Thrashing
- <u>Memory-mapped files</u>
- Allocating kernel memory
- Other considerations
- Operation-system examples

Memory-Mapped Files

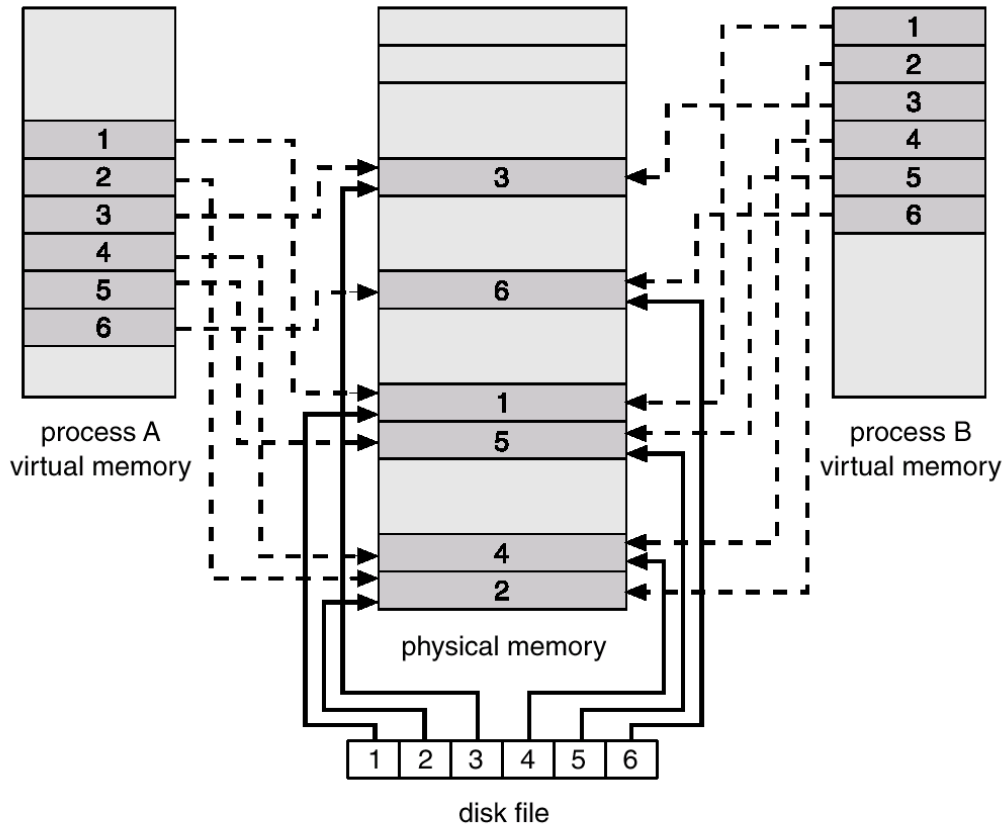
- Memory-mapped file: a part of virtual address space is logically associated with a file
  - Map a disk block to a page in memory
  1. Initial access proceeds through demand paging results in page fault.
  2. A page-sized portion of the file is read from the file system into a physical page.
  3. Subsequent access is through memory access routine.
  4. When the file is closed, memory-mapped data are written back to disk.

- Advantages
  - Reduces overhead of read() and write()
  - File sharing

Memory-Mapped Files



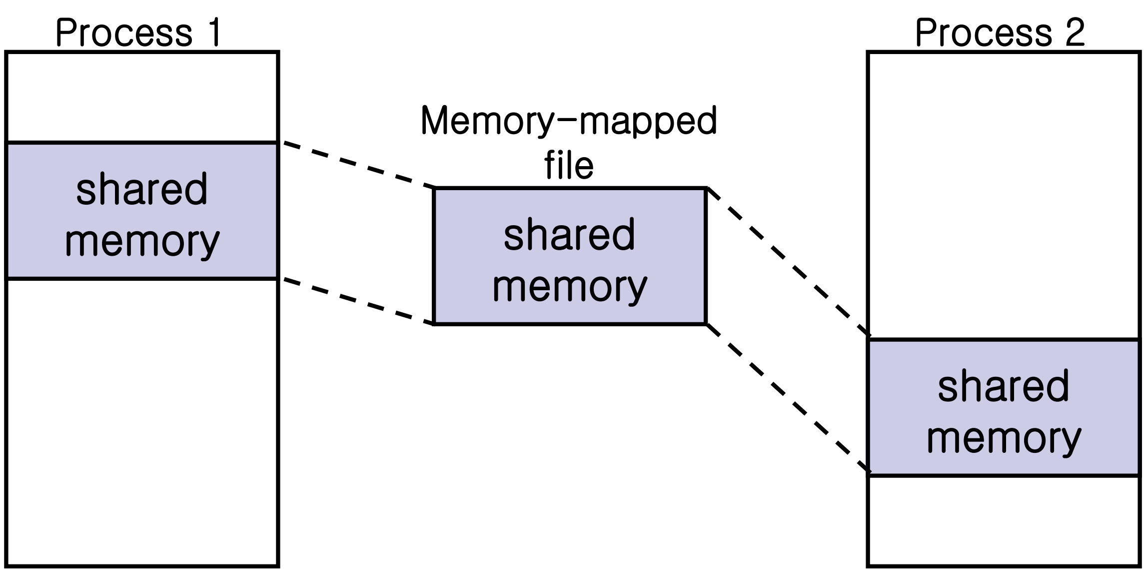
Memory-Mapped Files



Memory-mapped file
Shared by two processes

Shared Memory in Win32 API

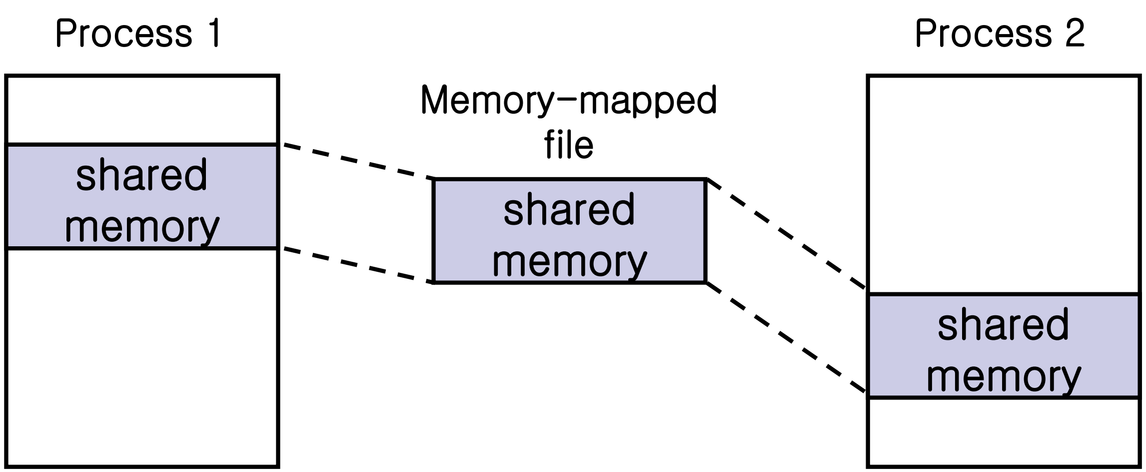
- Sharing memory-mapped file is similar to shared memory
- On Windows NT, 2000, XP, shared memory is accomplished by memory mapping files



Shared Memory in Win32 API

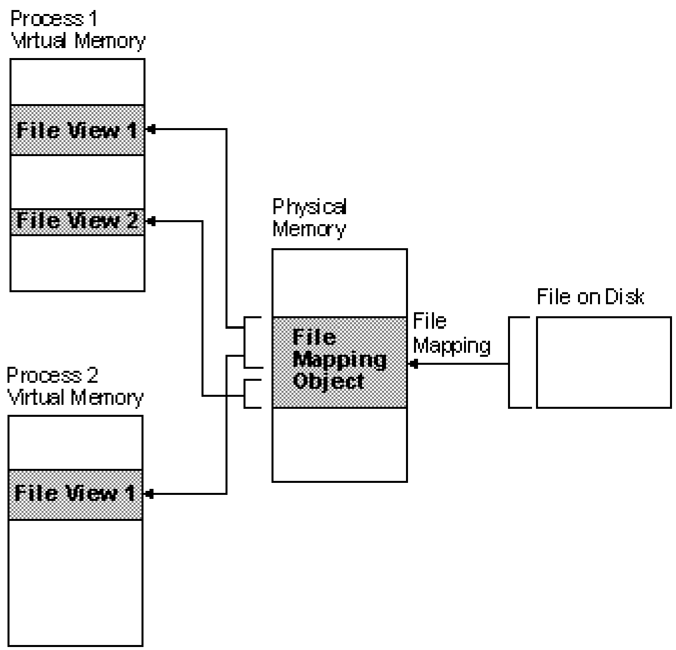
- Process 1
  1. Create a file to be shared
     - CreateFile(…)
  2. Create file mapping (named shared memory object)
     - CreateFileMapping(…)
  3. Establish a view of mapped file in virtual address space
     - MapViewOfFile(…)

- Process 2
  1. Open file mapping
     - OpenFileMapping(…)
  2. Establish a view of mapped file in virtual address space
     - MapViewOfFile(…)



Memory Mapped File on Windows

- File mapping is the association of a file's contents with a portion of the virtual address space of a process.
  - The system creates a file mapping object to maintain this association.
  - A file view is the portion of virtual address space that a process uses to access the file's contents.
  - It also allows the process to work efficiently with a large data file.
  - Multiple processes can also use memory-mapped files to share data.

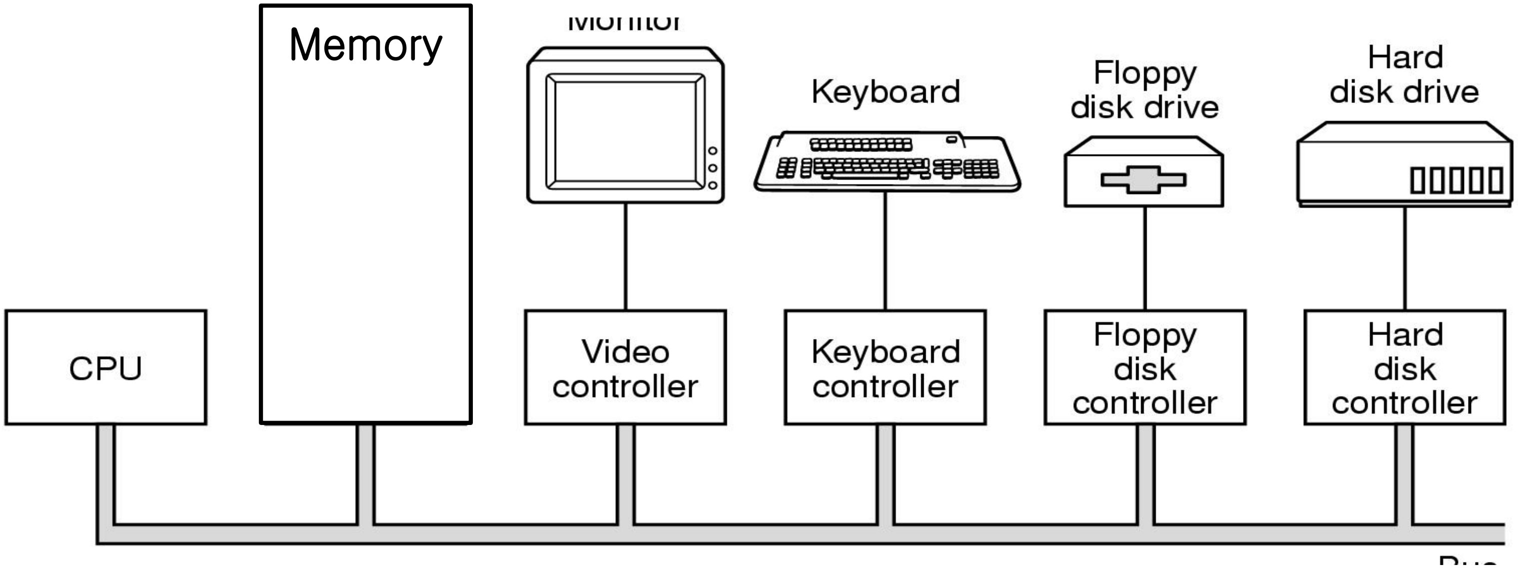


Reading Assignment

- Visit and study the following web pages to learn memory mapped file on Windows
  - File mapping
    - https://msdn.microsoft.com/en-us/library/windows/desktop/aa366556(v=vs.85).aspx
  - File mapping functions
    - https://msdn.microsoft.com/en-us/library/windows/desktop/aa366781(v=vs.85).aspx#file_mapping_functions

Memory-Mapped I/O

- I/O devices are accessed through …
  - Device registers in I/O controller to hold command and data
  - Usually, special instructions transfer data between device registers and memory



Memory-Mapped I/O

- Memory-mapped I/O: ranges of memory addresses are set aside and mapped to the device registers
  - In IBM PC, each location on screen is mapped to a memory location
  - Serial/parallel ports – data transfer through reading/writing device registers (ports)

- Programmed I/O (PIO)

  Ex) Sending a long string of bytes through memory-mapped I/O
    - CPU sets a byte to a register and set a bit in the control register to signal that the data is available.
    - Device receives the data and clears the bit in the control register to signal CPU that it is ready for the next byte.

  cf. Interrupt driven I/O

Agenda

- Background
- Demand paging
- Copy-on-write
- Page replacement
- Allocation of frames
- Thrashing
- Memory-mapped files
- <u>Allocating kernel memory</u>
- <u>Other considerations</u>
- Operation-system examples

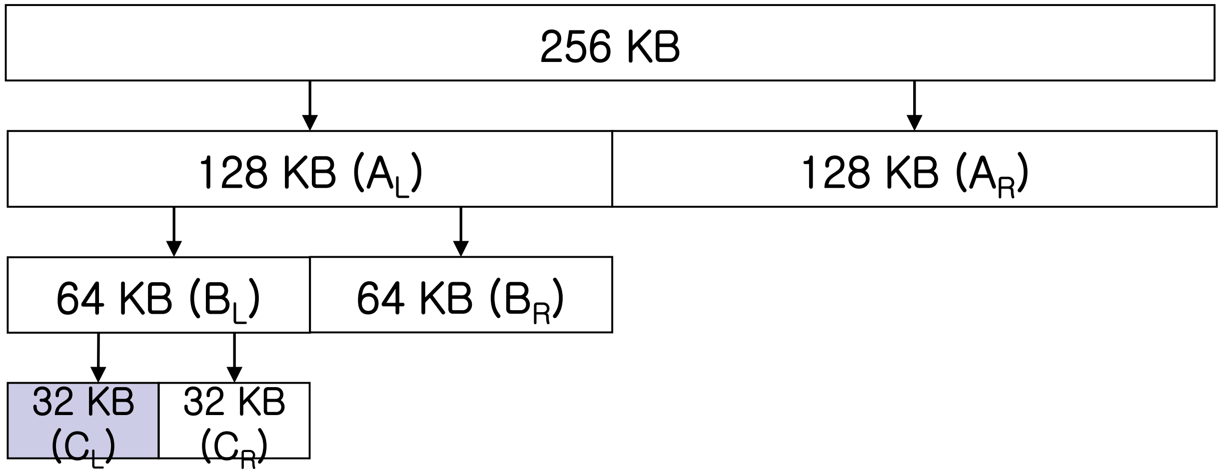
Allocating Kernel Memory

- Allocation of kernel memory requires special handling
  - Kernel requests memory for data structures of varying sizes
    - Many OS's do not subject kernel code/data to the paging system
  - Certain H/W devices interact directly with physical memory
    - Memory should reside in physically contiguous pages.
- Strategies for kernel memory allocation
  - Buddy system
  - Slab allocation

Buddy System

- Buddy system: allocates memory from a fixed-size segment consisting of physically contiguous pages
  - Power-of-2 allocator

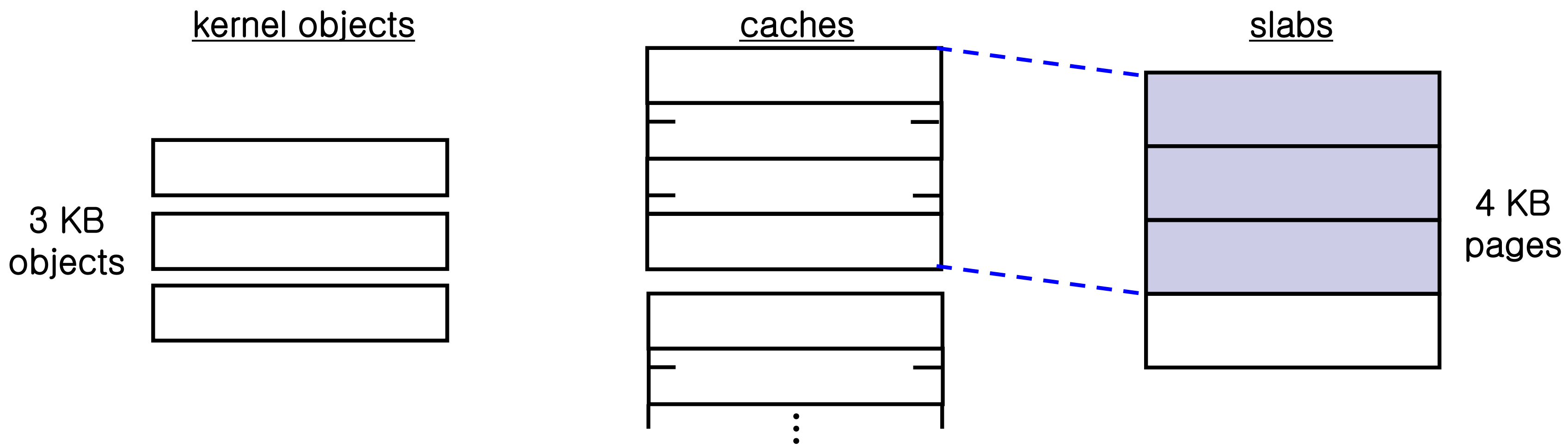
  Ex) initially 256 KB is available, 21 KB was requested



  - Advantage: easy to combine adjacent buddies
  - Disadvantage: internal fragmentation

Slab Allocation

- Motivation: mismatch between allocation size and requested size
  - Page-size granularity vs. byte-size granularity
  - Applied since Solaris 2.4 and Linux 2.2
- Cache for each unique kernel data structure
  - A slab is made up of one or more physically contiguous pages
  - A cache consists of one or more slabs



Slab Allocation

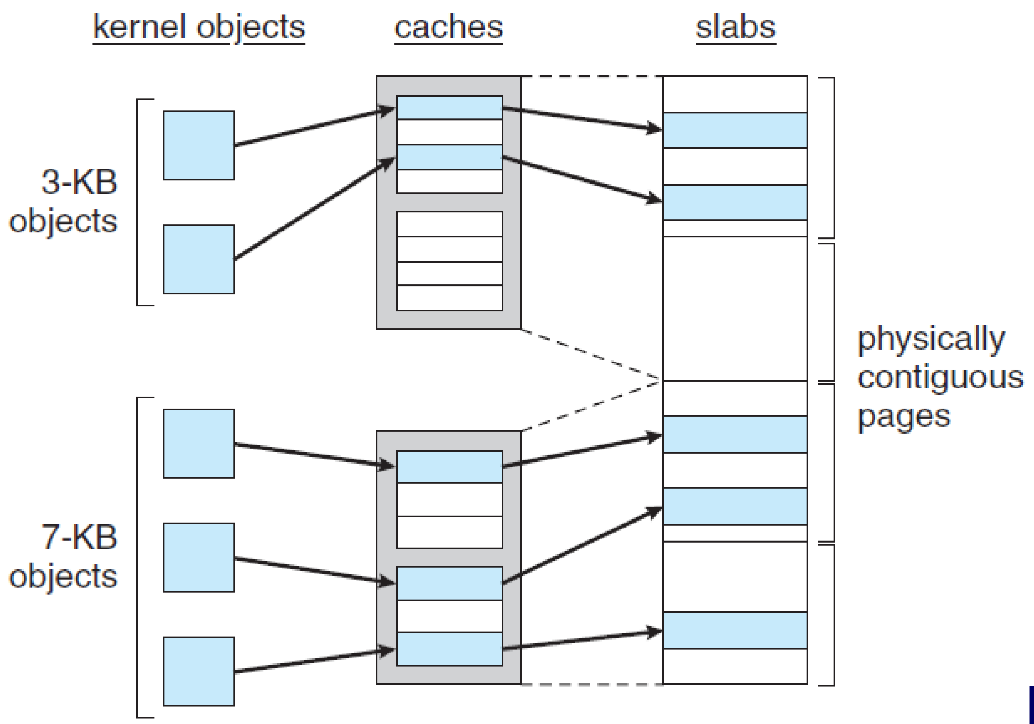
- Single cache is for each unique kernel data structure
  - Each cache filled with objects – instantiations of the data structure.

  Ex) cache for process descriptor, cache for file objects, cache for semaphore, …
- When cache created, filled with objects marked as free.
- When structures stored, objects marked as used.
- If slab is full of used objects, next object allocated from empty slab.
  - If no empty slabs, new slab is allocated.

Slab Allocation

- Benefits
  - No memory waste due to fragmentation
  - Memory requests can be satisfied quickly

➔ Suitable for data structures that are allocated and deallocated frequently.



Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page tables
- Program structure
- I/O interlock

Prepaging

- A problem of pure demand paging: a large number of page faults

- Prepaging: bring all pages that will be needed at one time to reduce page faults.

  Ex) working-set model

  - Important issue: cost of prepaging vs. cost of servicing corresponding page faults

Page Size

- Issues about page size

	smaller page	larger page
Size of page table	large	small
Memory utilization	better	worse
I/O latency	large	small
Locality	good	bad
Page fault	many	few

  - Historical trend: page size is getting larger

TLB Reach

- To improve TLB hit ratio, size of TLB should be increased.

-> but associate memory is expensive, power hungry

- TLB reach: amount of memory accessible from TLB
  - TLB reach = <# of entries in TLB> * <page size>

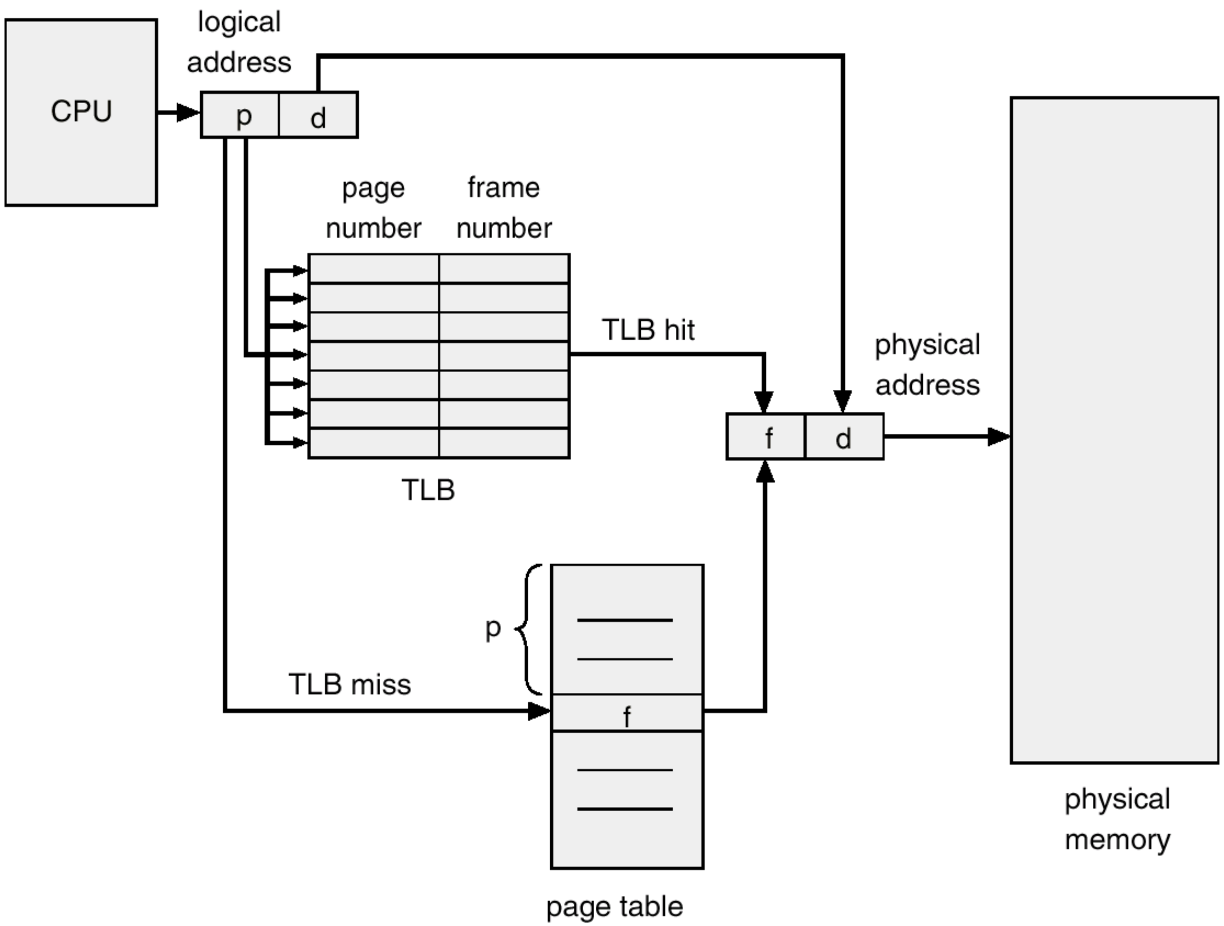
- TLB reach can increase by increasing page size
  - However, with large page, fragmentation also increases.

  ➔ S/W managed TLB (OS support several different page sizes)

  Ex) UltraSparc, MIPS, Alpha

  Cf) PowerPC, Pentium: H/W managed TLB

TLB Reach



CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

Inverted Page Tables

- Inverted page table reduces amount of physical memory needed to memory translation
- However, it no longer contains complete information about logical address space of a process
  - Demand paging requires complete information about logical address space to process page fault
- Remedy: maintaining external page table for each process
  - External page table can be paged out and in

Program Structure

- User don't' have to know about nature of memory. But, if user knows underlying demand paging, performance can be improved

  Ex) If page size is 128 words, B is better than A

```
for(j = 0; j < 128; j++)
    for(i = 0; i < 128; i++)
        data[i][j] = 0;
```

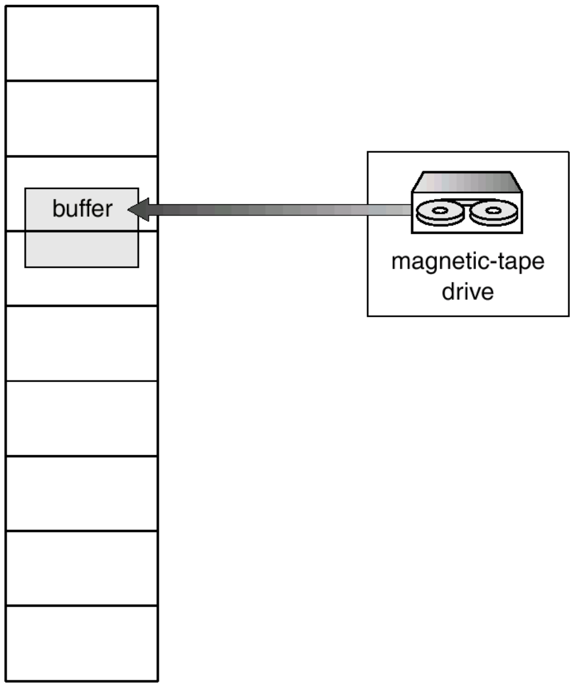
A

```
for(i = 0; i < 128; i++)
    for(j = 0; j < 128; j++)
        data[i][j] = 0;
```

B

I/O Interlock

- Memory space related to I/O transfer should not be replaced
- Remedies
  - I/O transfer is performed only through system memory
  - Locking pages in memory



Agenda

- Background
- Demand paging
- Copy-on-write
- Page replacement
- Allocation of frames
- Thrashing
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- <u>Operation-system examples</u>

Windows XP

- Demand paging with clustering
  - At page fault, Windows XP brings not only the fault page but also several following pages

- Working-set minimum/maximum
  - Working-set minimum: minimum # of pages guaranteed for a process
    - For most applications, 50
  - Working-set maximum: maximum # of pages that can be assigned to a process if sufficient memory is available.
    - For most applications, 345

Windows XP

- Virtual memory manager maintains list of free frames
  - If there are some free frames and if page fault occurs for a process that is is below working-set maximum

  -> A free page is allocated to the process
  - If amount of free memory falls below a threshold, and a process has more pages than working-set minimum

  -> Automatic working-set trimming

Solaris

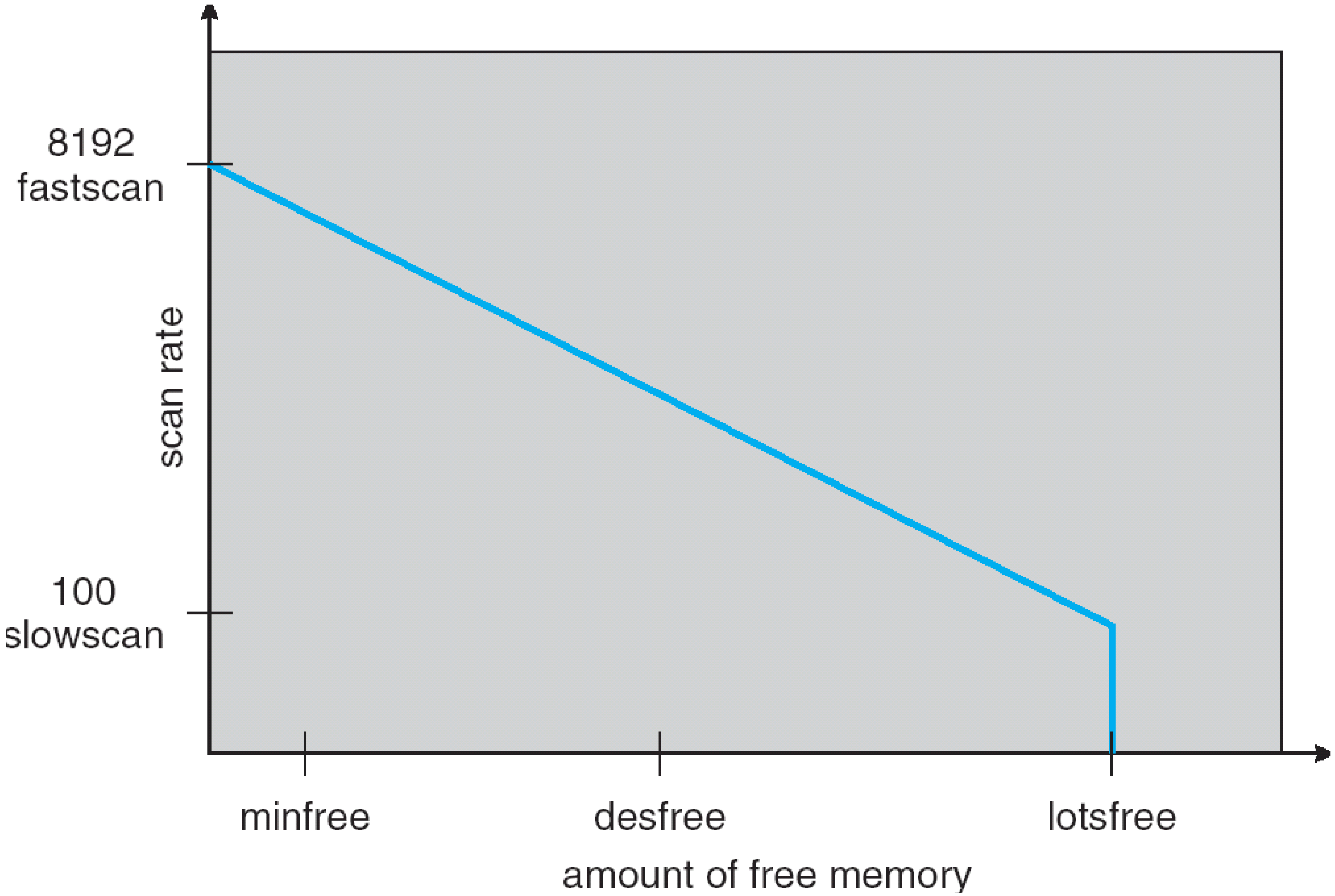
- Maintains a list of free pages to assign faulting processes

- Three thresholds for amount of free memory
  - *Lotsfree* – threshold parameter (amount of free memory) to begin paging
  - *Desfree* – threshold parameter to increasing paging
    - If it is unable to keep free memory at desfree for 30 sec, kernel begins swapping
  - *Minfree* – pageout is called for every request for a new page

Solaris

- Pageout scans pages using modified second-chance algorithm
  - *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
  - Pageout is called more frequently depending upon the amount of free memory available

Solaris


8192
fastscan
100
slowscan
scan rate
minfree
desfree
lotsfree
amount of free memory