10. Virtual Memory

ECE30021/ITP30002 Operating Systems

Agenda

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

Background

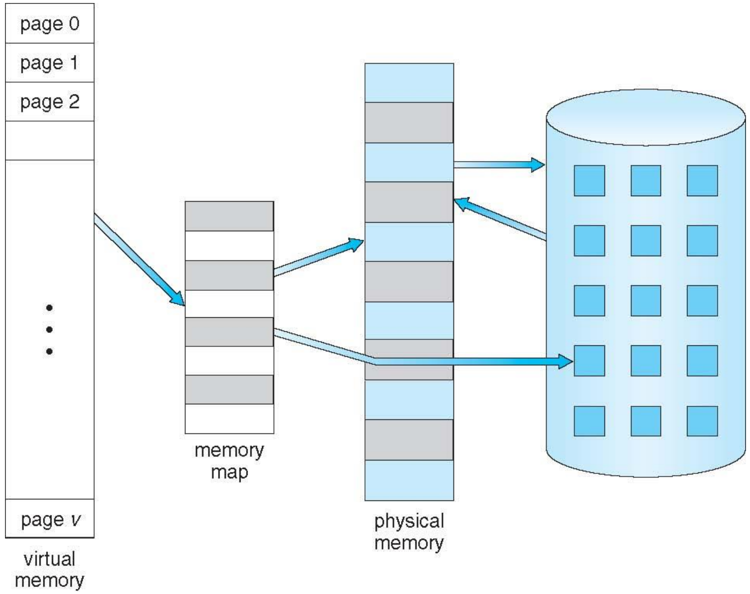
- Instructions should be in physical memory to be executed
  - ➔ In order to execute a program, should we load entire program in memory?
- Some parts are rarely used
  - Error handling codes
  - Arrays/lists larger than necessary
  - Rarely used routines
- Alternative
  - <u>Executing program which is only partially in memory</u>

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



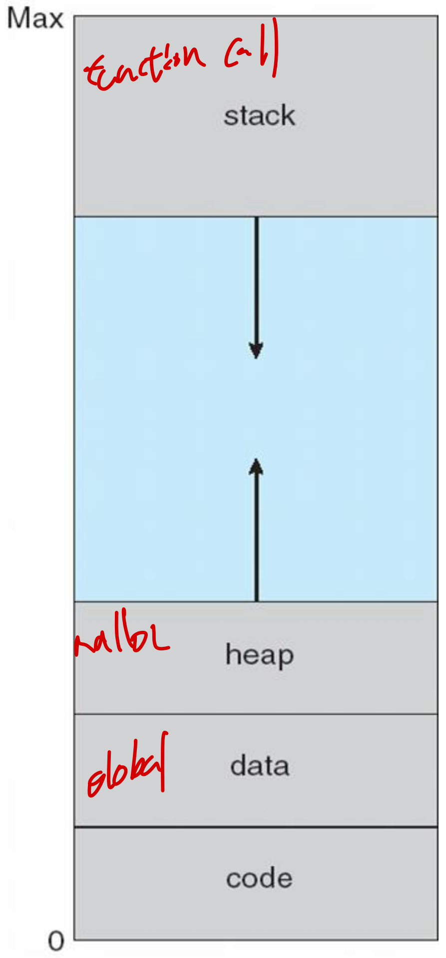
Virtual Memory

- Virtual memory: a separation of logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation (e.g. COW)
  - More programs running concurrently
  - Less I/O needed to load or swap processes

page 단위

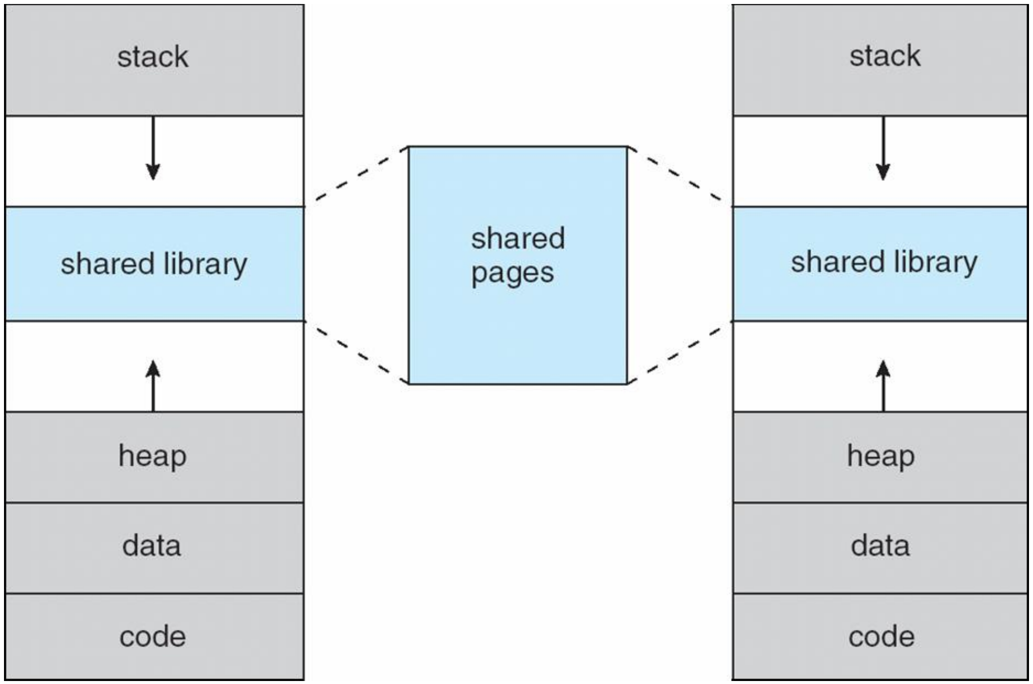
Virtual Address Space

- Virtual address space: logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space.
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
  - Programmers don't have to concern about memory management
  - Virtual address space can be sparse



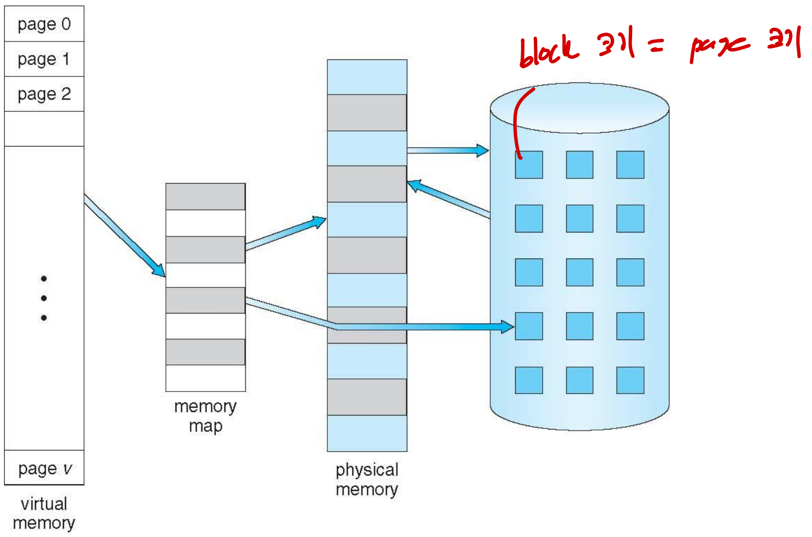
Shared Library Using Virtual Memory

- Shared page



Implementation Techniques

- <u>Demand paging</u>
  - Paging + swapping



- Demand segmentation
  - Segmentation + swapping

Agenda

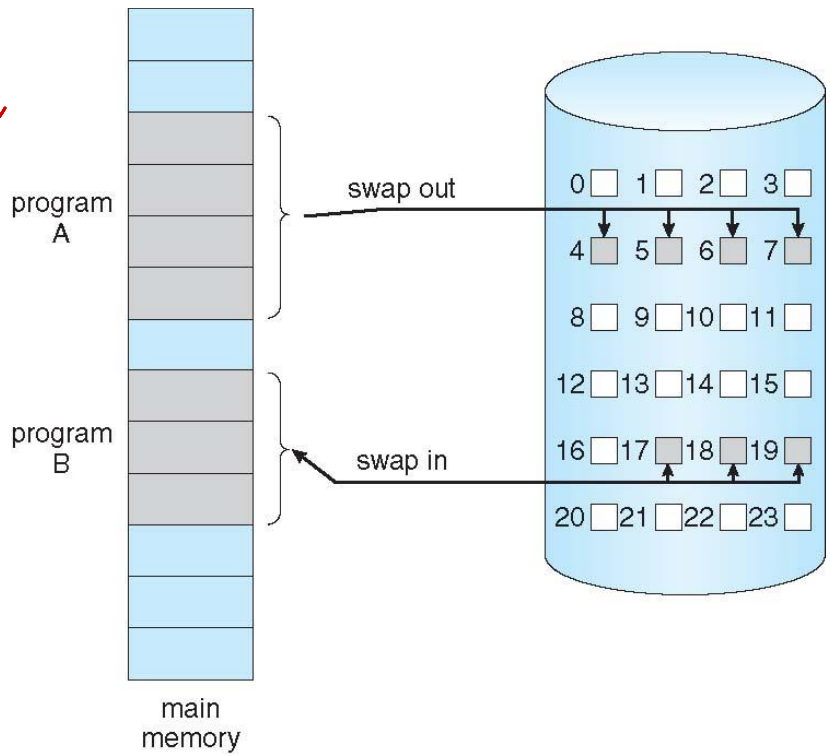
- Background
- <u>Demand paging</u>
- Copy-on-Write (COW)
- Page replacement
- Allocation of frames
- Thrashing
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- Operation-system examples

Demand Paging

- Similar to paging system with swapping

꼭 필요할때만 하겠다 page swapper

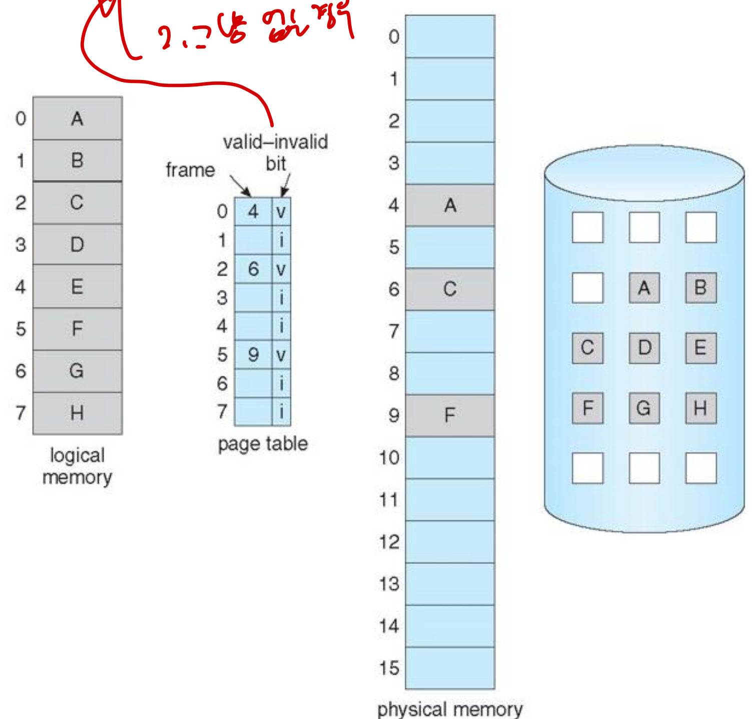
- Lazy swapper (or pager)
  - Pages are only loaded when they are demanded during execution.



Demand Paging

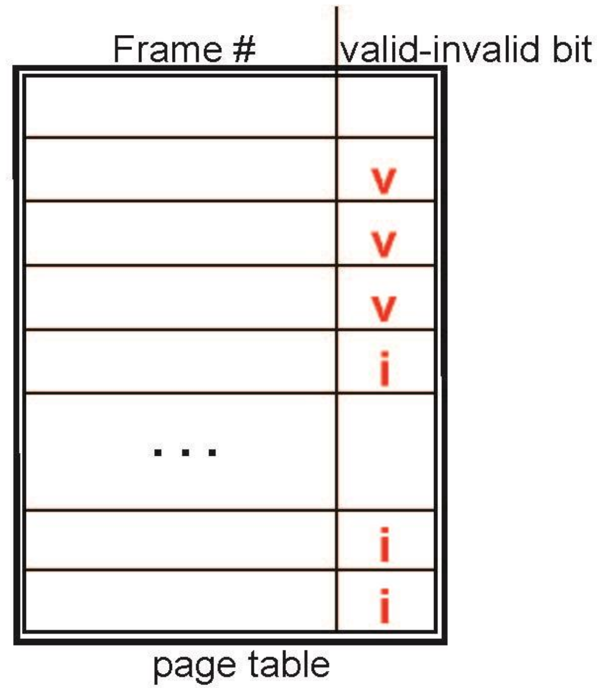
- Pages are loaded **only when they are demanded** during program execution
  - Requires H/W support to distinguish the page on memory or on disk

1. 있긴 있는데 loading이 안되었을 경우
2. 그냥 없는 경우



Page Table with Valid/Invalid Bit

- Valid/invalid bit of each page
  - Valid: the page is valid and exists in physical memory
  - 에러 or 해킹 시도
  - Invalid: the page is not valid (not in the valid logical address space of the process) or not loaded in physical memory
- If program tries to access ..
  - Valid page: execution proceeds normally
  - Invalid page: cause **page-fault** trap to OS

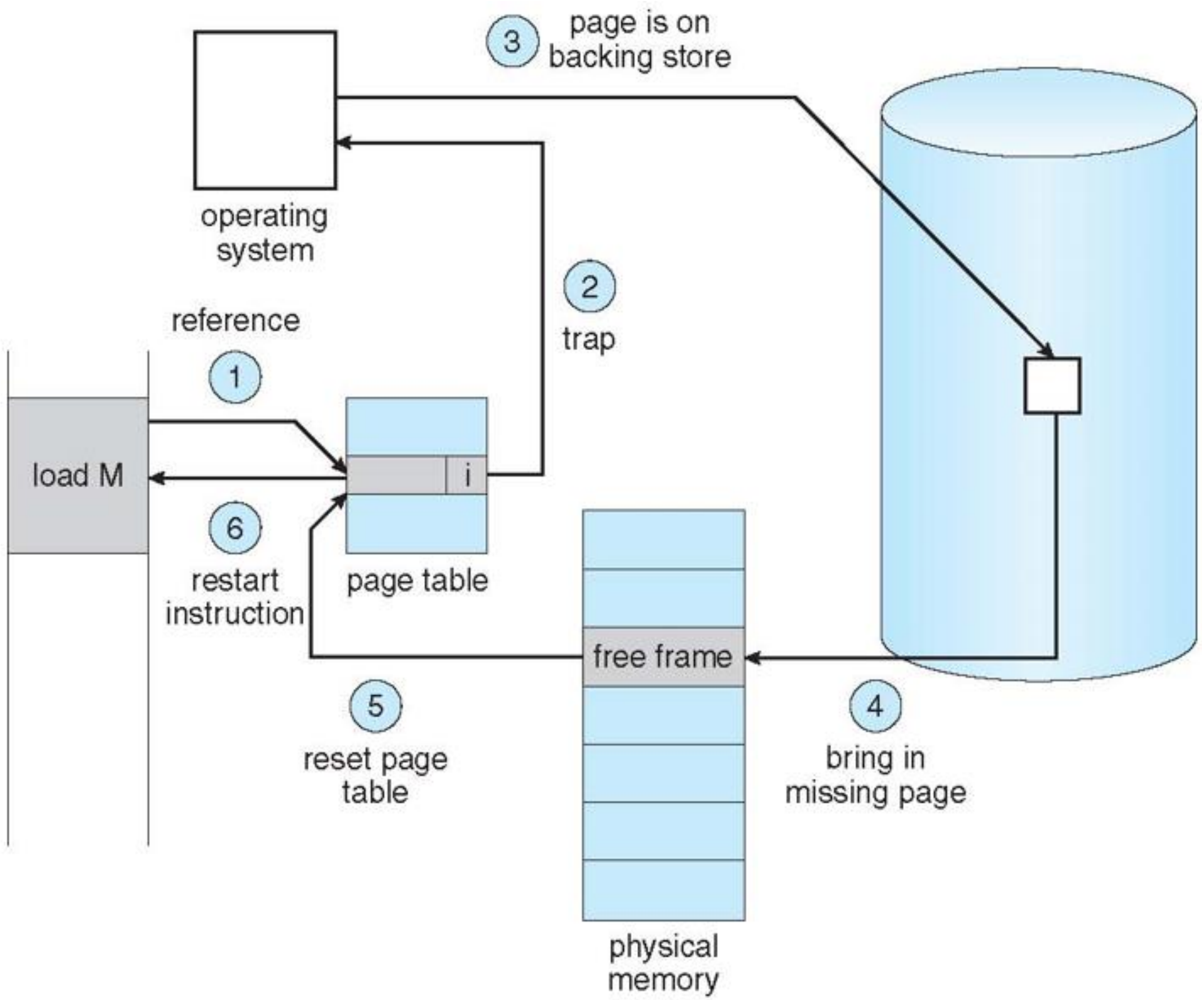


Basic Concepts

page fault 발생시 valid-invalid bit이 i 값을 갖는다. 존재x
코드x

- 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)
  - Instruction restart

Performance of Demand Paging

- **Effective access time**

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

  - ma : main memory access time (10~200 nano sec.)
  - p : probability of page fault ex) 0.00

- **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

  ≈ 200 + 7,999,800 * p

  - Proportional to **page fault rate** $ma \times 40$

  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

Agenda

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

Copy-on-Write

- fork() copies process
  - Duplicates pages belong to the parent

process creation, 메모리 복사 overhead 최소화

parent process가 child process로 복사되었을 때 논리적으로는 분리되었으나 물리적으로는 같은 내용을 가지고 있지만

둘 중 한 process가 writing을 하면 더이상 내용이 같지 않아 공유할 수 없으므로 실제 물리적으로 분리되어 memory content의 복사가 이루어져야한다.

- 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.

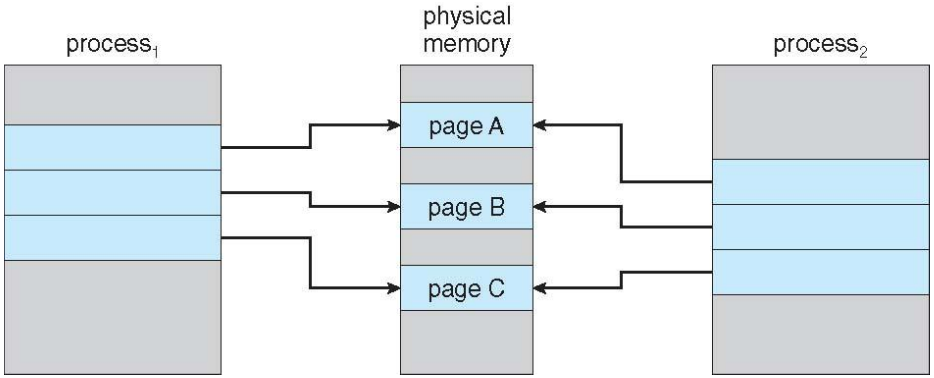
  cf. vfork() – logically shares memory with parent (obsolete)

- Many OS's provides a list of free frames for COW or stack/heap that can be expanded

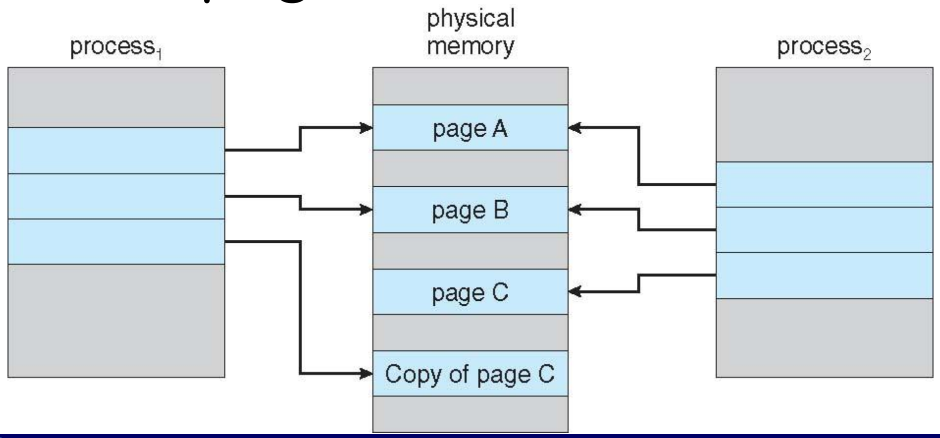
  ➔ Zero-fill-on-demand (ZFOD)
    - Zero-out pages before being assigned to a process

Copy-on-Write

- Before P1 modifies page C



- After P1 modifies page C



Agenda

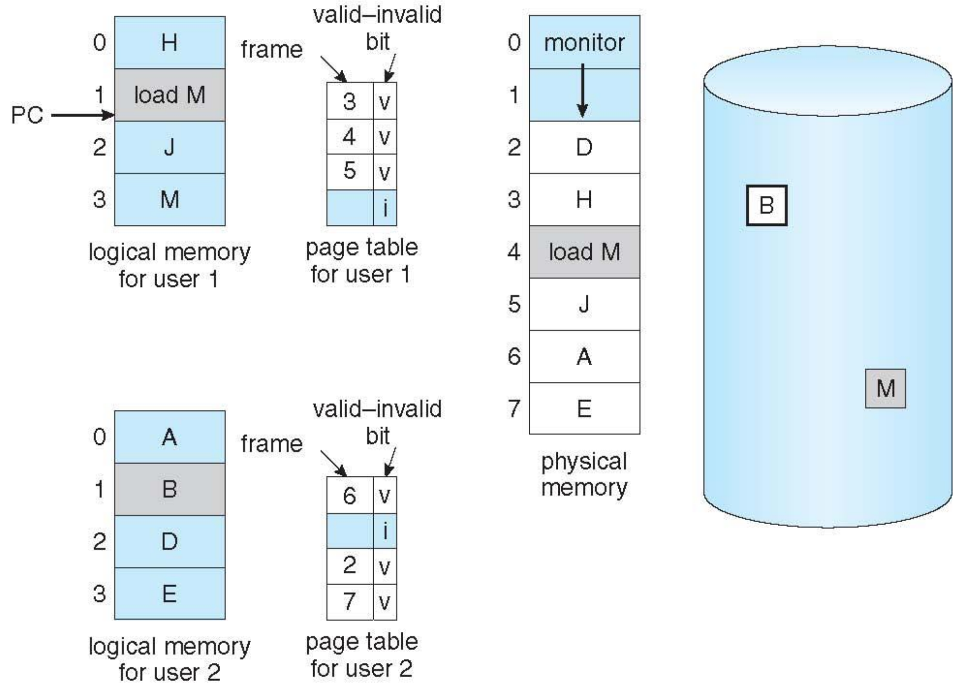
- Background
- Demand paging paging + swapping
- Copy-on-Write (COW)
- <u>Page replacement</u>
- <u>Allocation of frames</u>
- Thrashing
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- Operation-system examples

Two major problems

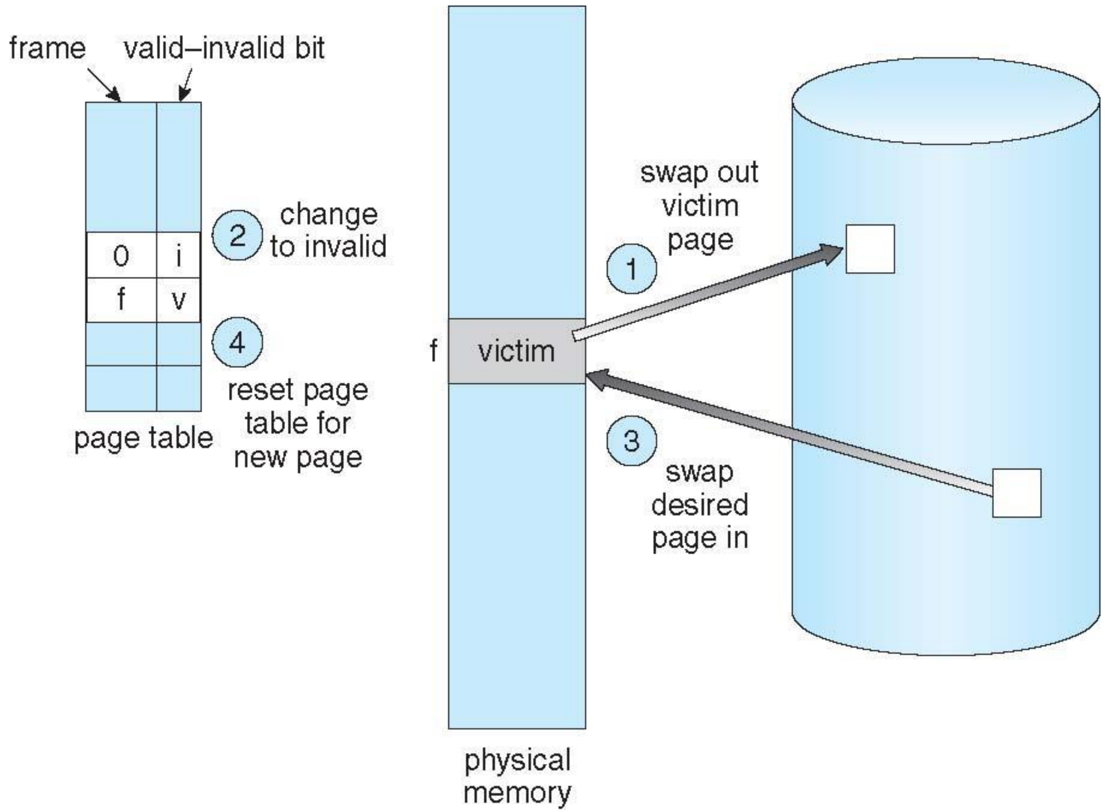
- Two major problems in demand paging
  - Page-replacement algorithm
  - Frame-allocation algorithms
- Even slight improvement can yield large gain in performance.
  - Effective access time = $(1-p) * ma + p *$ <page fault time>

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

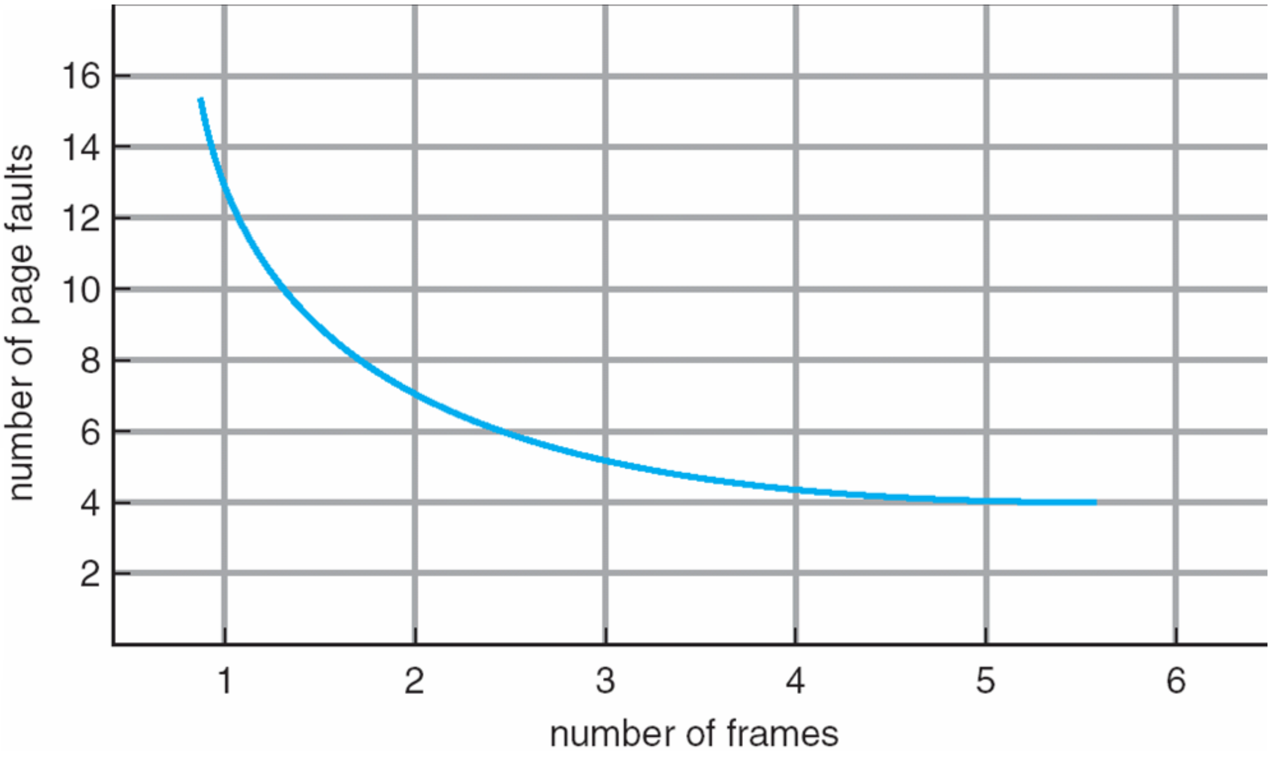


Page Replacement



Page Faults vs. Number of Frames

- In general, the more frames, the fewer page faults

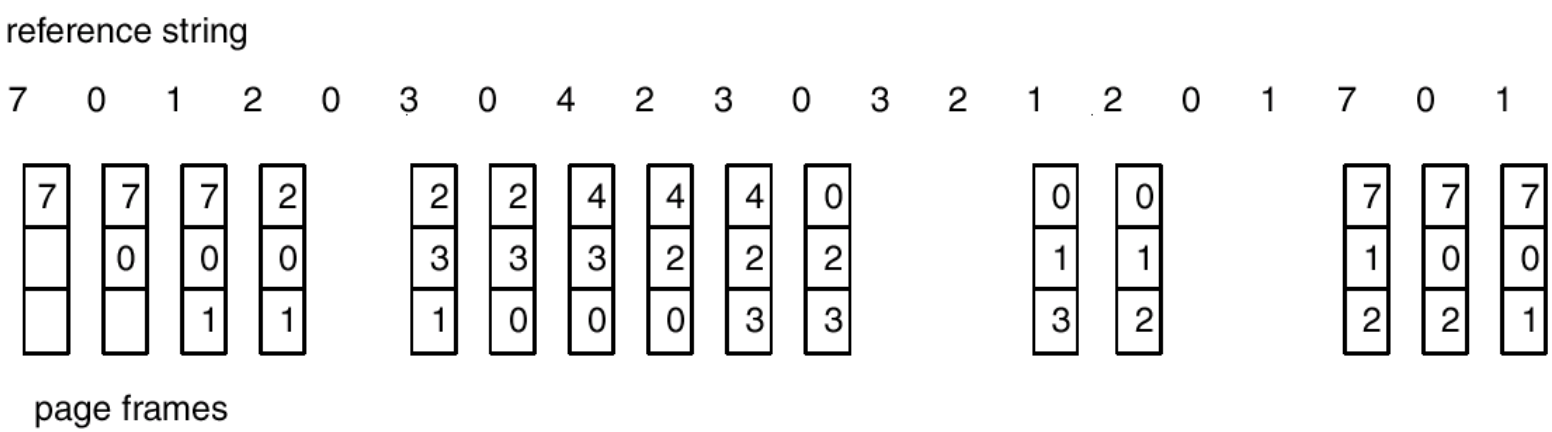


Page Replacement Algorithms

- 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

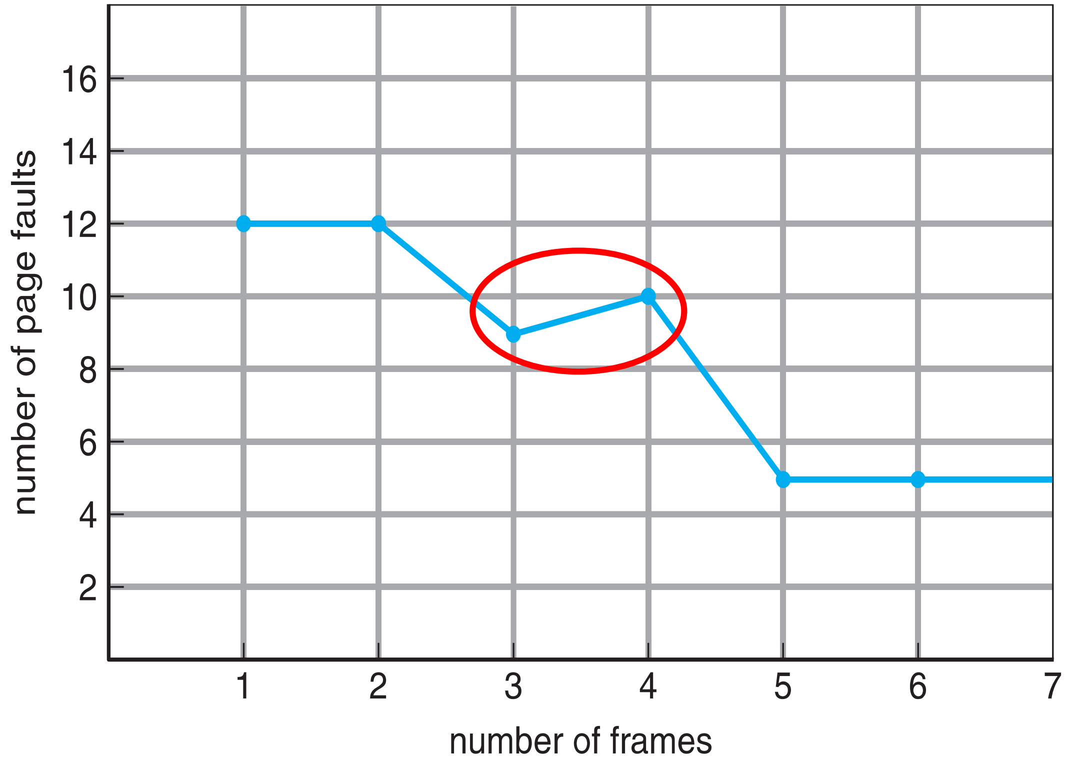


  - # of page faults: 15
- Problem: Belady's anomaly

FIFO Page Replacement

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

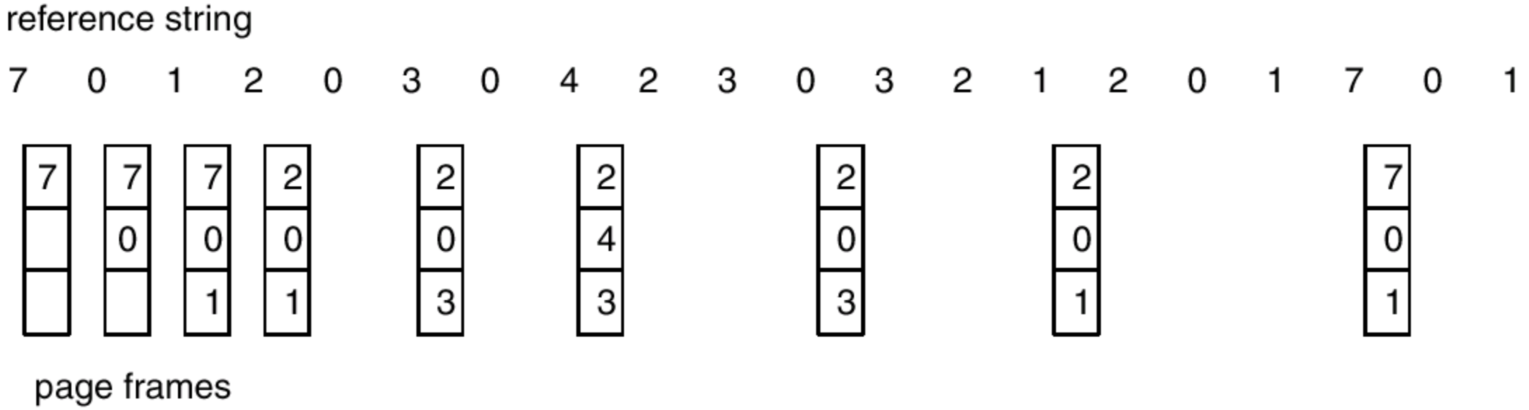
(Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5)



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

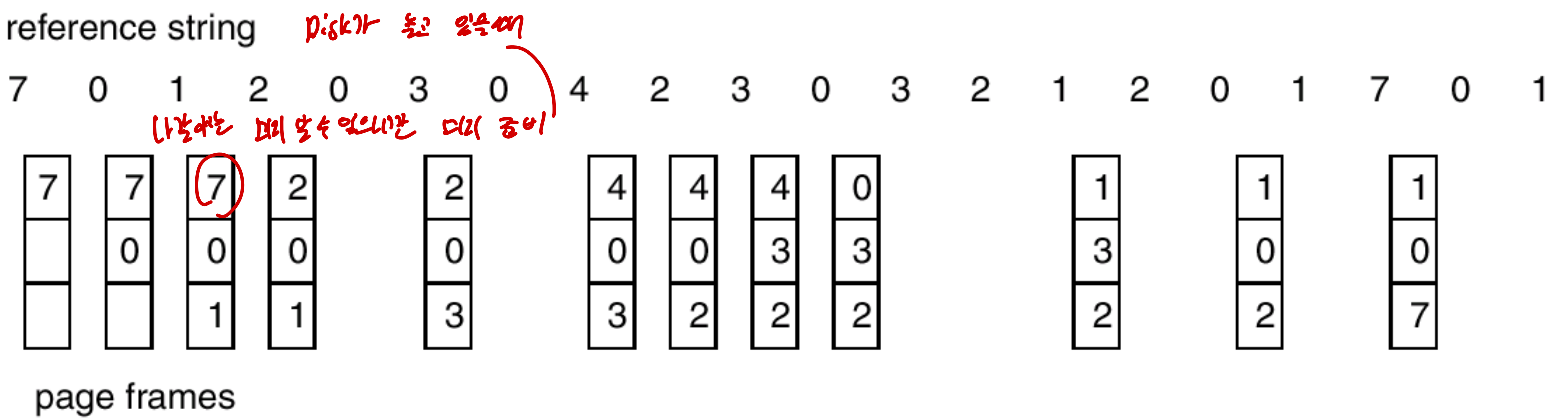


  - # of page faults: 9

- Problem: It requires future knowledge

LRU Page Replacement

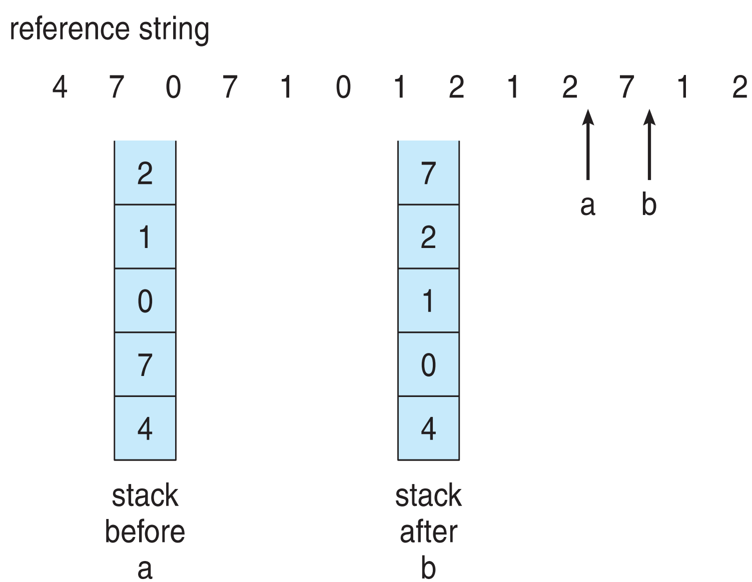
- LRU (Least Recently Used): replace page that has not been used for longest period of time



  - # 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



이것도 느림

↳ 하드웨어의 지원이 필요함

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

$\boldsymbol{n}$ frames

1	2	3	4	5	6	7	...

$\boldsymbol{n+1}$ frames

1	2	3	4	5	6	7	...

LRU-Approximation Page Replacement

bit가 1개밖에 없음

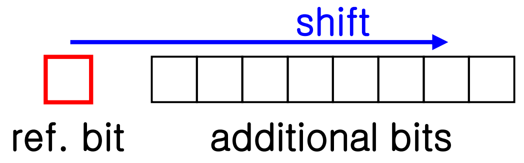
- **Motivation**
  - LRU algorithm is good, but few system provide sufficient supports for LRU
  - However, many systems support **reference bit** for each page
    - We can determine which pages have been referenced, but not their order.

- **LRU-approximation algorithms**
  - Additional-reference-bit algorithm
  - Second-chance algorithm

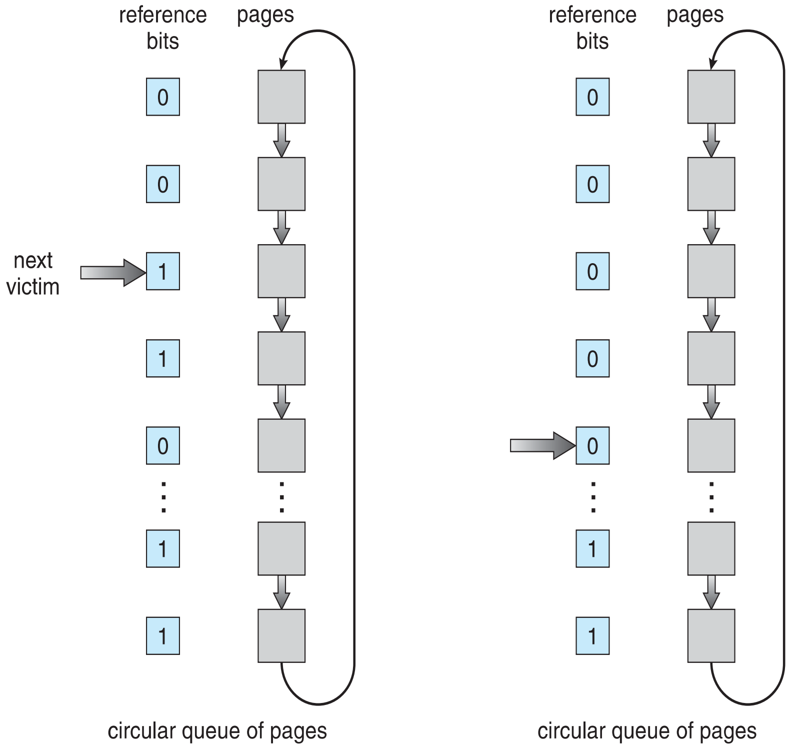
LRU-Approximation Page Replacement

- 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.

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 크기에 비례해서 주는 방법
  - 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

process의 경계 생각 X, 더 효율적인 선택 가능, 남의 공간도 과감 뺏음

- 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

내꺼 중에서 바꿔라

- 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

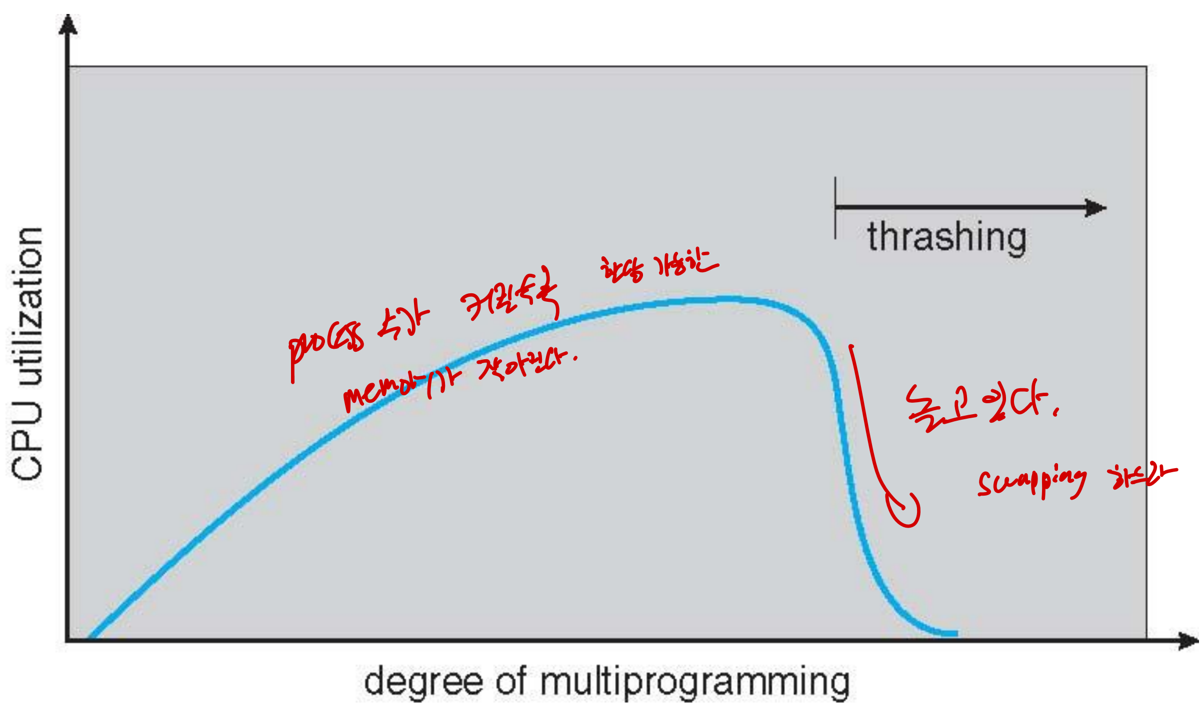
- Background
- Demand paging
- Copy-on-Write (COW)
- Page replacement
- Allocation of frames
- <u>Thrashing</u>
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- Operation-system examples

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.

Thrashing -> process 별로 Thrashing이 안 일어날 만큼은 memory를 줘야함.

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



ready queue에 있는 process의 수

Thrashing

memory access가 균일하게 일어나지 않는다.

시간에 따라 어떨 땐 얘네 어떨 땐 쟤가 많이 쓰인다.

access 되는 page 마다 같이 쓰이는 page가 있다.

(붙어다니는 애들의 수 > frame 수

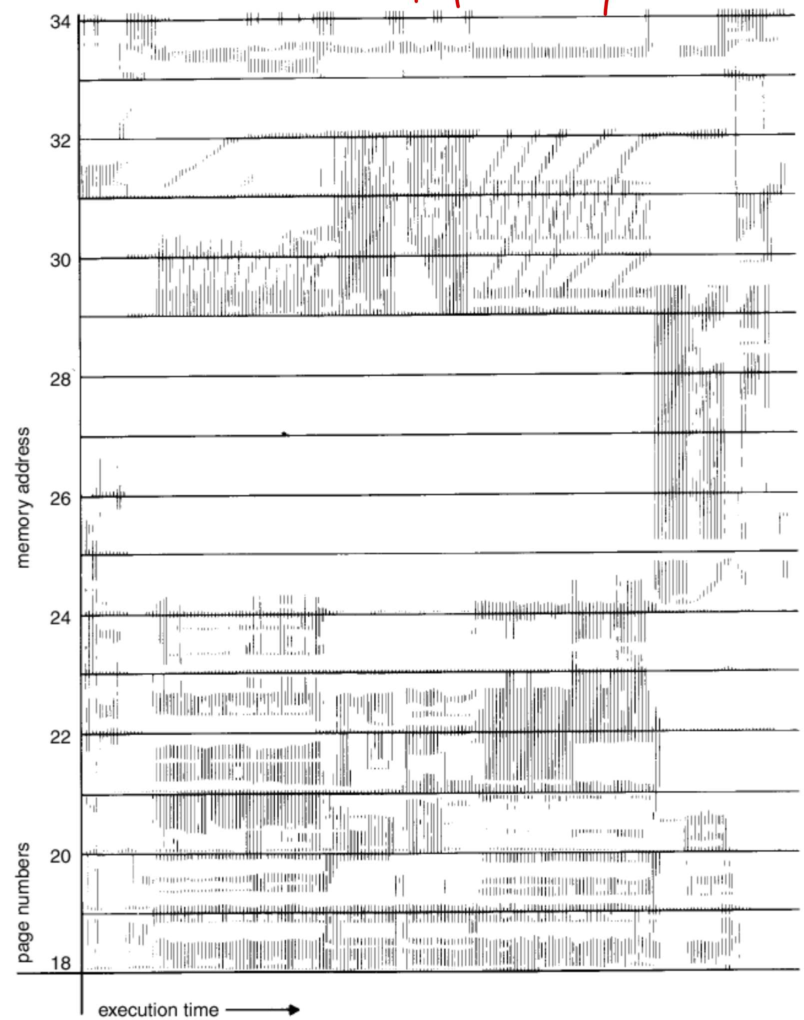
↳ page fault rate이 급격하게 증가

- To prevent thrashing, a process must be provided with as many frames as it needs.

  –> How to know how many frames it needs?

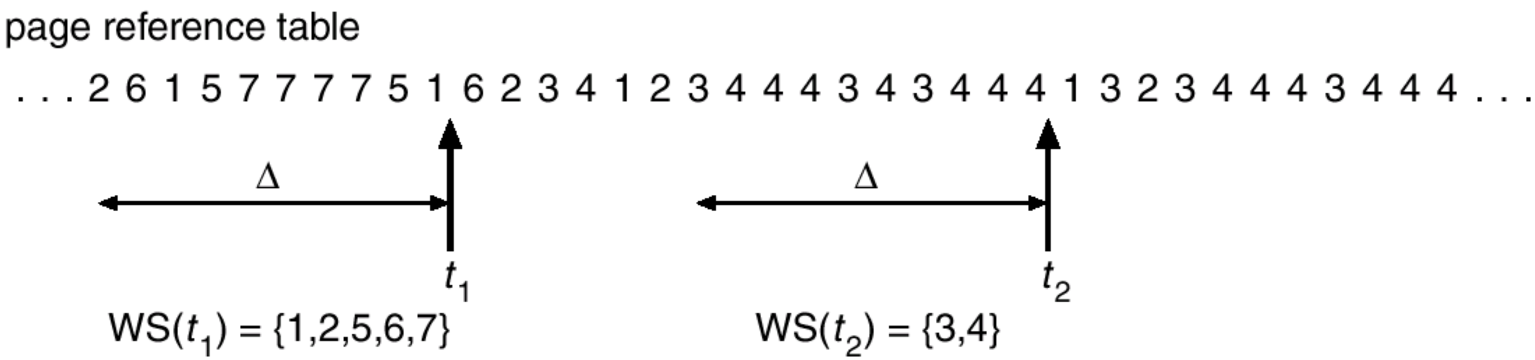
- Locality model
  - Locality: set of pages actively used together
  - A program is generally composed of several localities

회색이 memory access 흔적



Working–Set Model

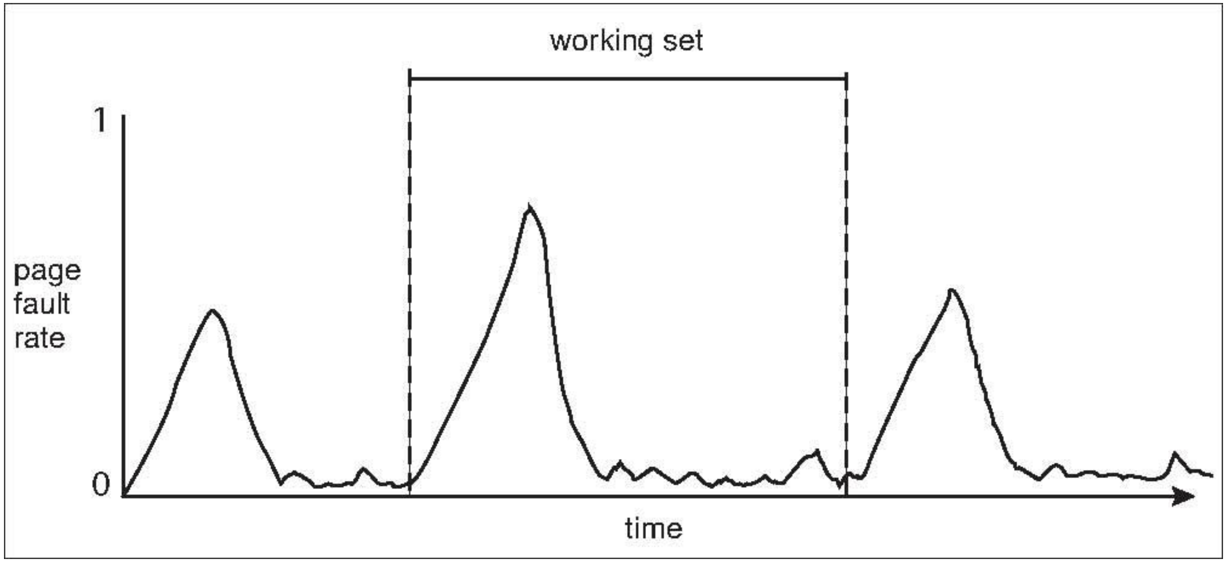
- Working set: set of pages in the most recent Δ page references
  - Parameter Δ: working–set window



  - 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.

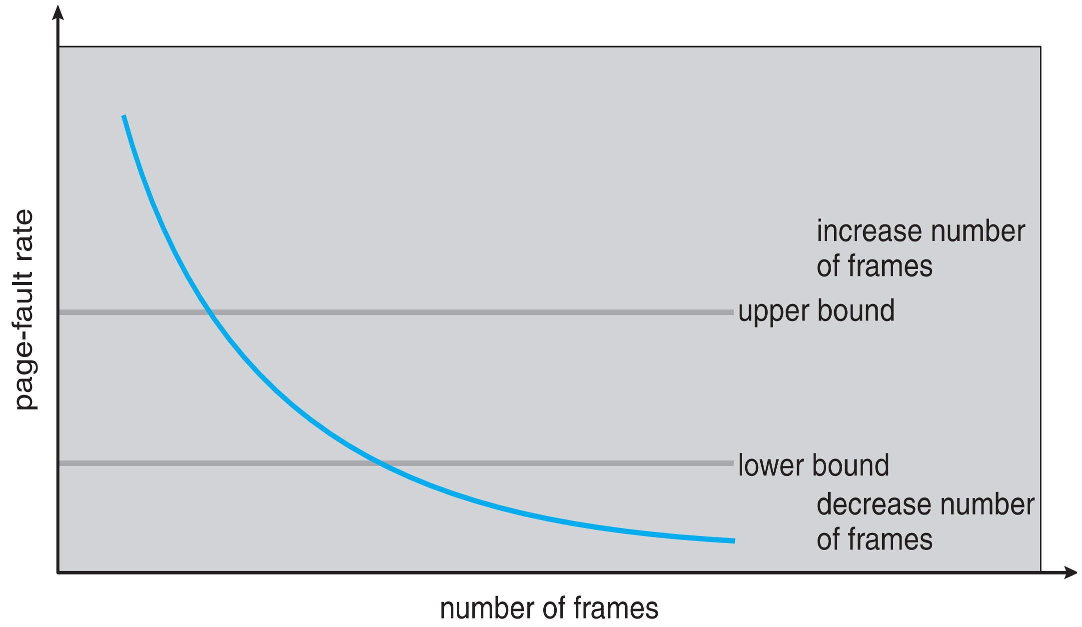
Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



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

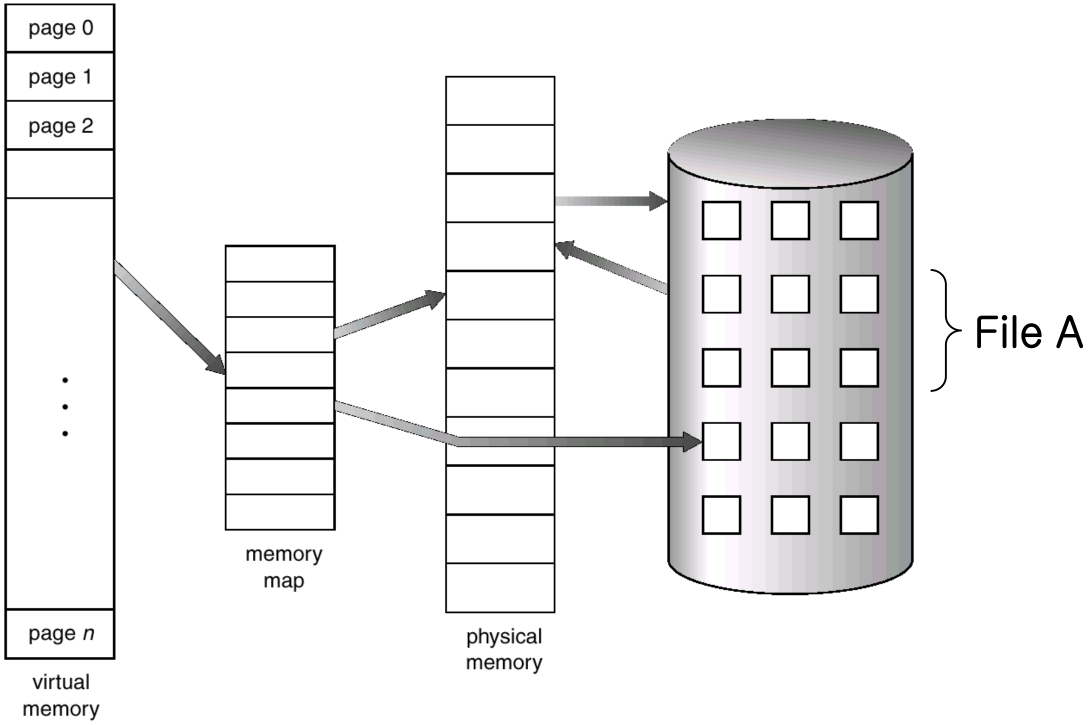
- Background
- Demand paging
- Copy-on-Write (COW)
- Page replacement
- Allocation of frames
- Thrashing
- <u>Memory-mapped files</u>
- Allocating kernel memory
- Other considerations
- Operation-system examples

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.

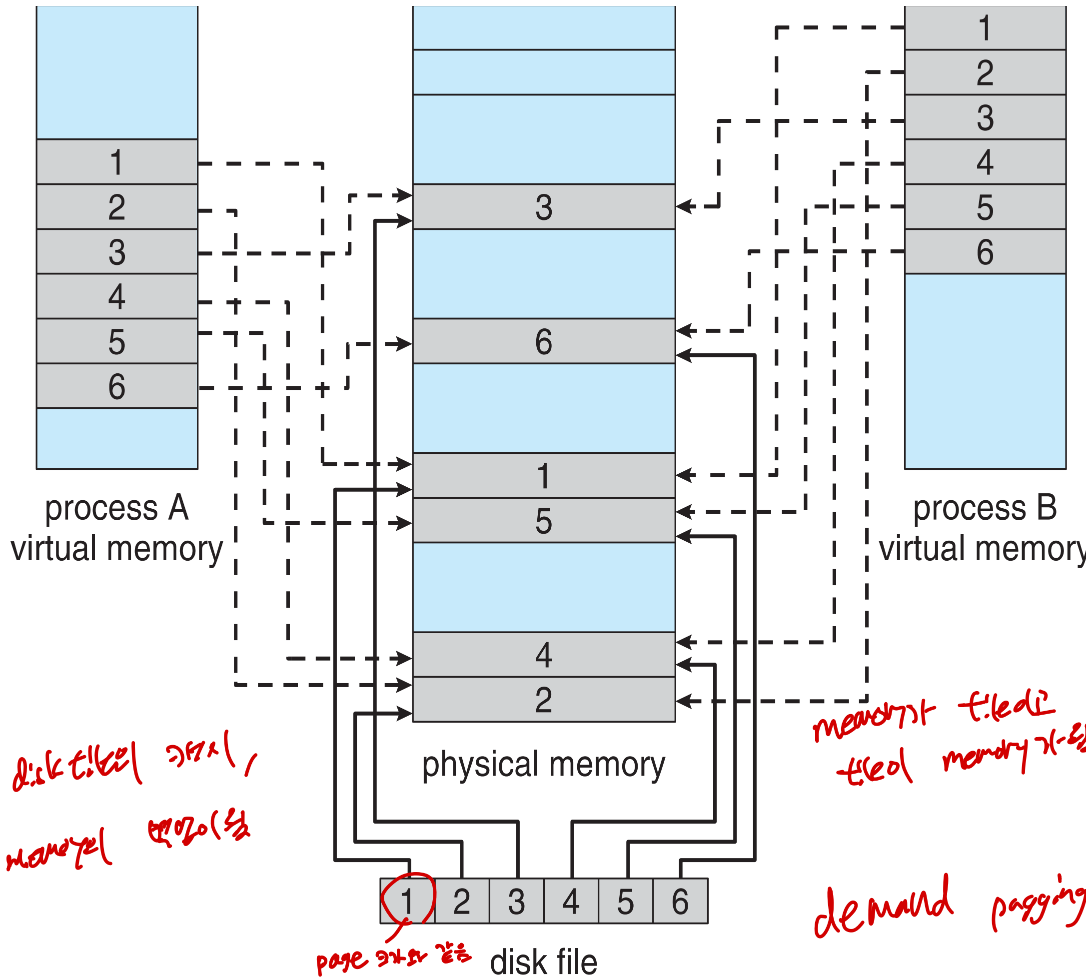
- Advantages
  - Reduces overhead of read() and write()
  - File sharing

Memory-Mapped Files



Memory-Mapped Files

demand paging의 고급기능을 여기서 사용 가능



physical memory가 disk file의 캐시,
disk file이 physical memory의 백업이 됨

memory가 file이고
file이 memory가 됨

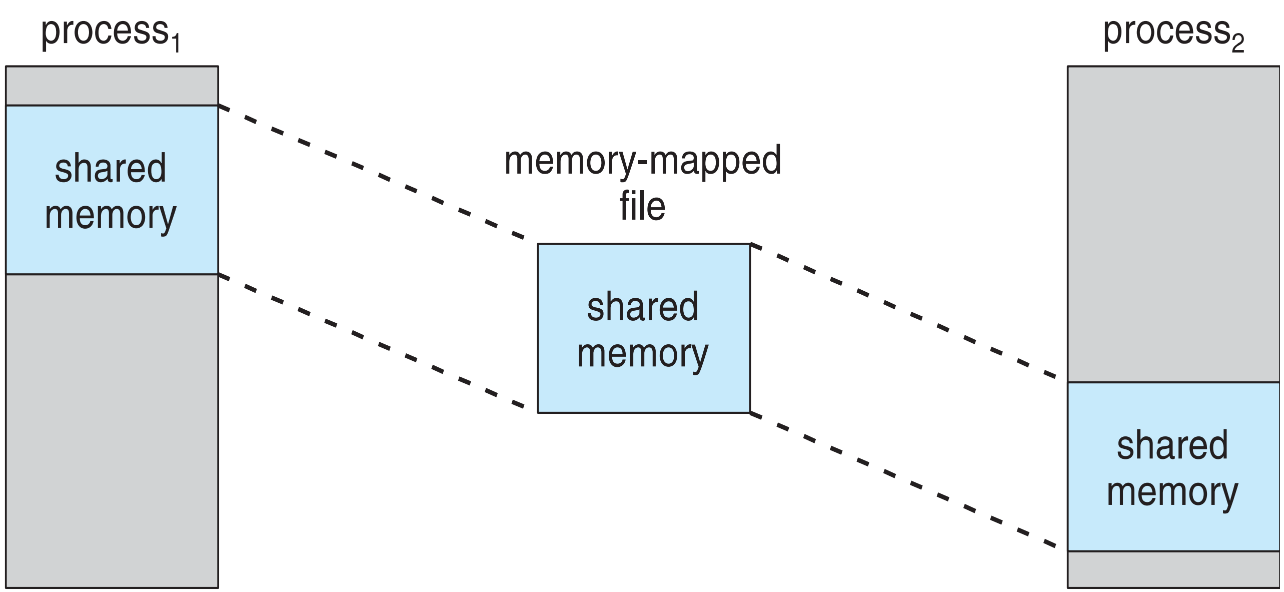
demand paging과 같아짐

page 크기와 같음

Memory–mapped file
Shared by two processes

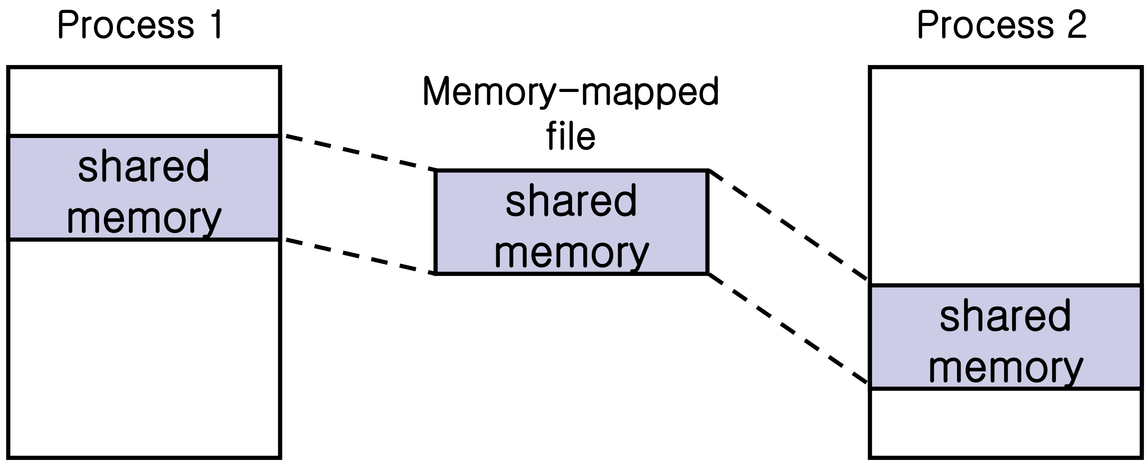
Shared Memory in Win32 API

- Sharing memory-mapped file is similar to shared memory
  - On POSIX and Windows, shared memory is accomplished by memory mapping files



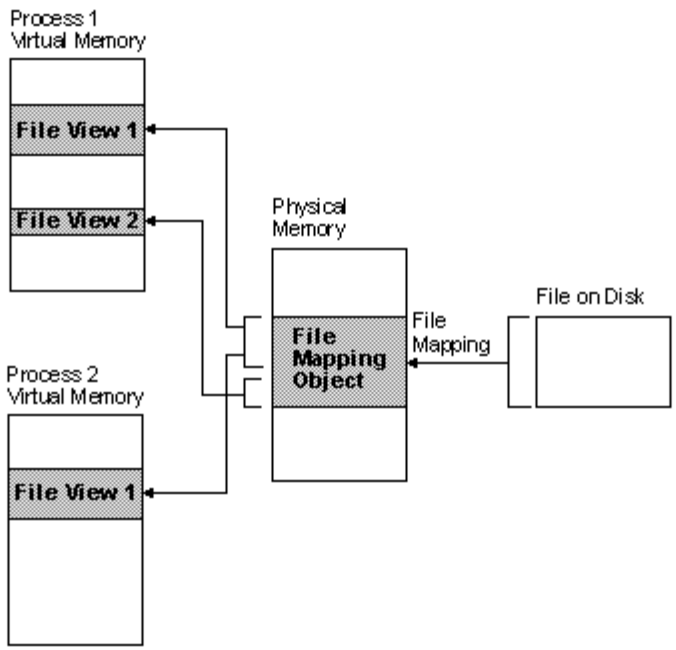
Shared Memory in Win32 API

- **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.

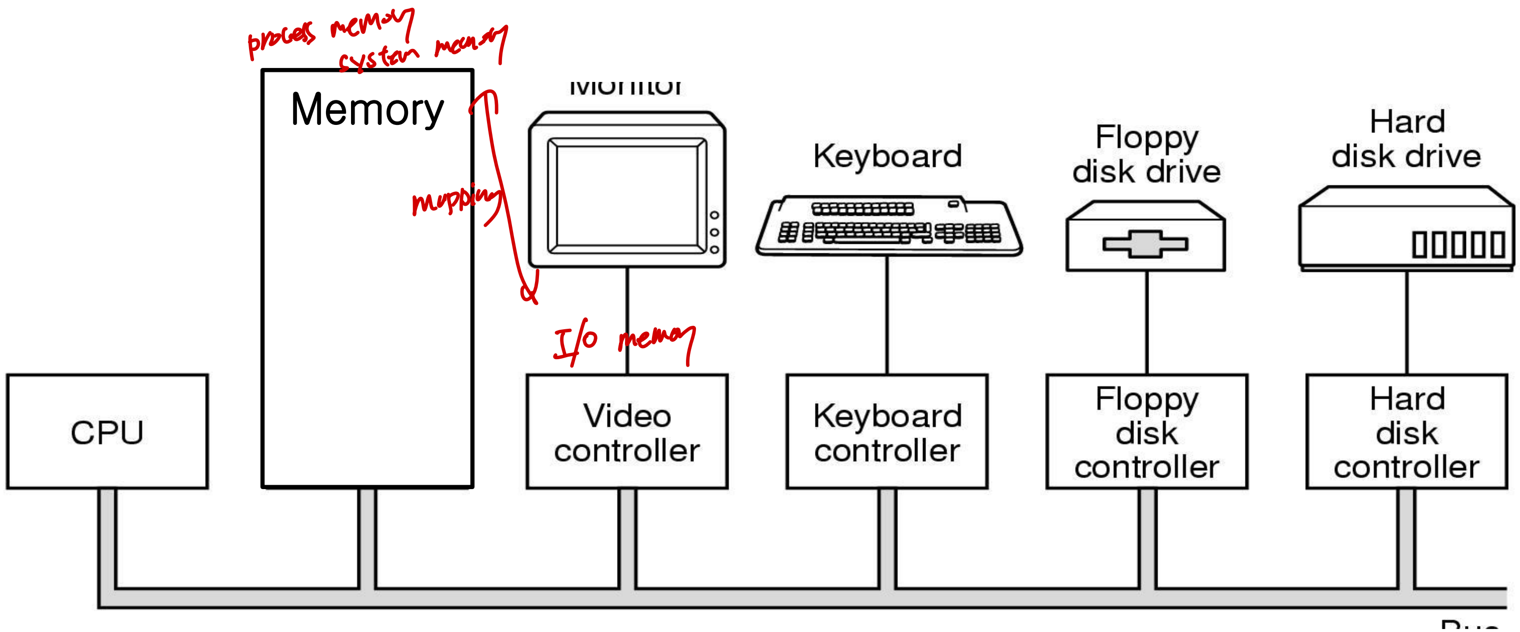


Reading Assignment

- 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 (COW)
- Page replacement
- Allocation of frames
- Thrashing
- Memory-mapped files
- <u>Allocating kernel memory</u> paging보다 아래에 있는 조직
- <u>Other considerations</u>
- Operation-system examples

Allocating Kernel Memory

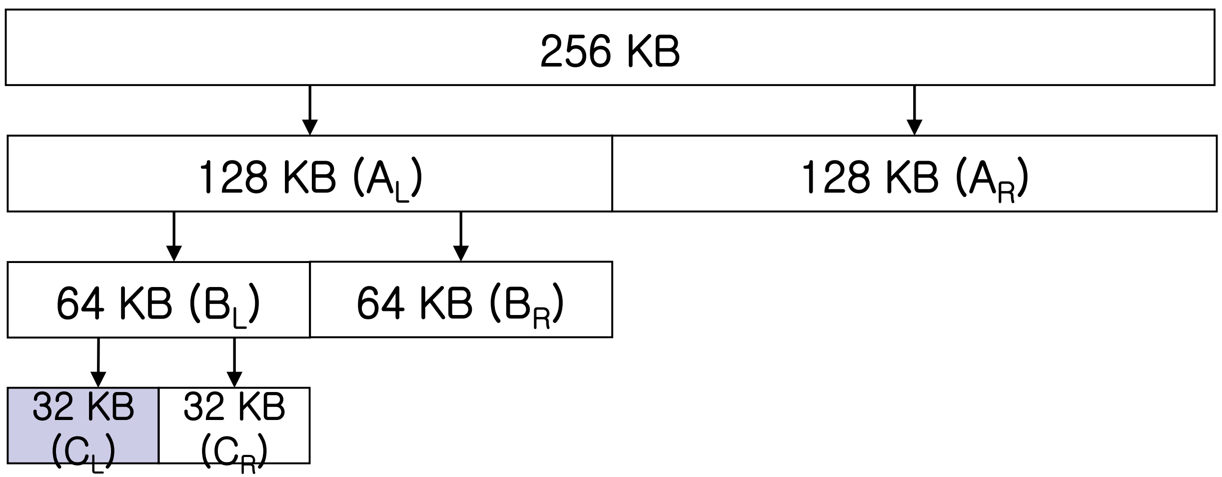
- Allocation of kernel memory requires special handling
  - Kernel requests memory for data structures of varying sizes
    - Many OS's do not subject kernel code/data to the paging system
  - Certain H/W devices interact directly with physical memory
    - Memory should reside in physically contiguous 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 frac 관리 편하다
  - Disadvantage: internal fragmentation 12 < 32

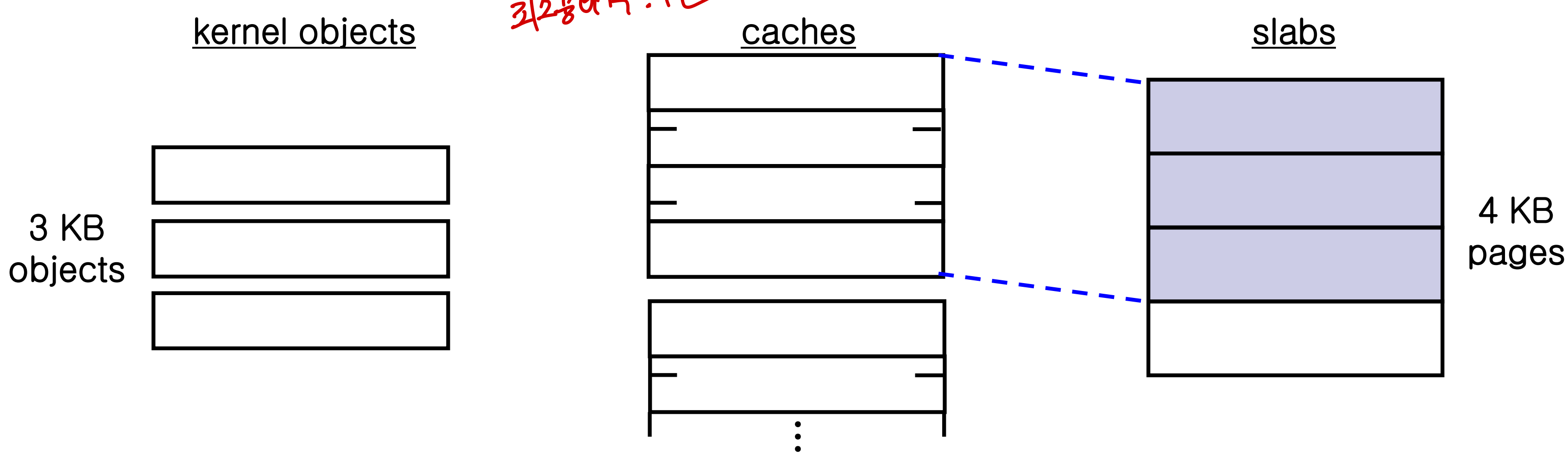
structure로 단순 -> 크기 일정

ex) PCB, TCB, shared memory block

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 똑같은 크기의 structure가 할당/해제되는
  - A slab is made up of one or more physically contiguous pages
  - A cache consists of one or more slabs

최소공배수 : 12



Slab Allocation

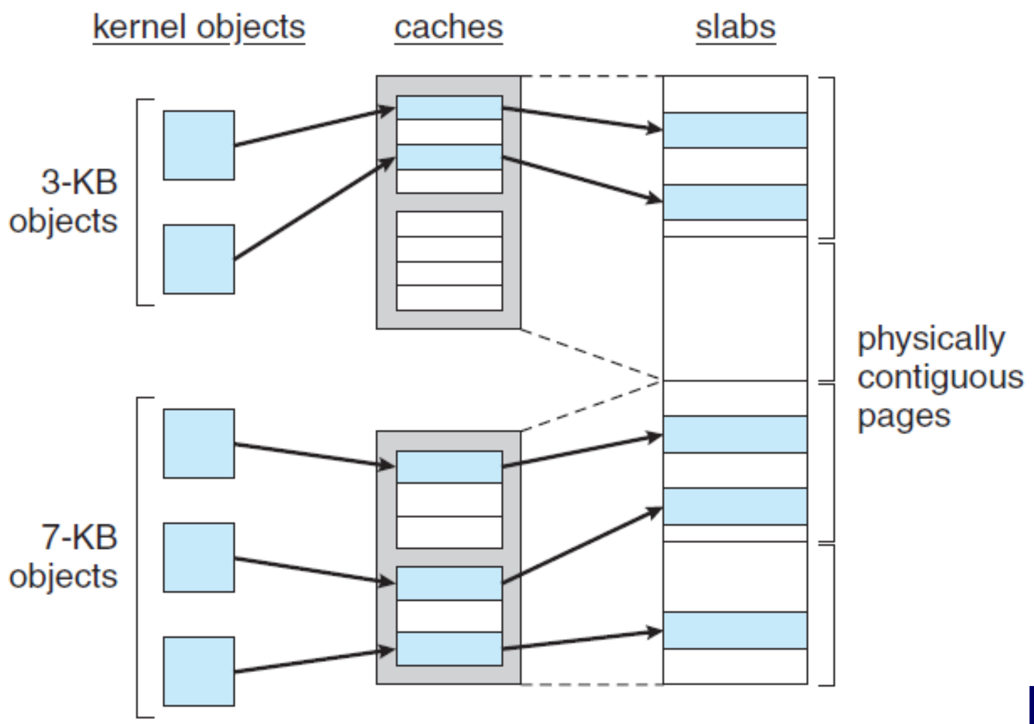
- Single cache is for each unique kernel data structure
  - Each cache filled with objects – instantiations of the data structure.

  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

미리 예측해서

page faults를 줄이는 것

- 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 (같이 다니는 page들의 set)

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

Page Size 증가하는 추세

intel page size (4K / 4M

- Issues about page size

Memory 소모 ↳ overhead

page 찾을 때 걸리는 시간

	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

한번 날때 cost가 큼

- Historical trend: page size is getting larger

Translation Lookaside Buffer

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>

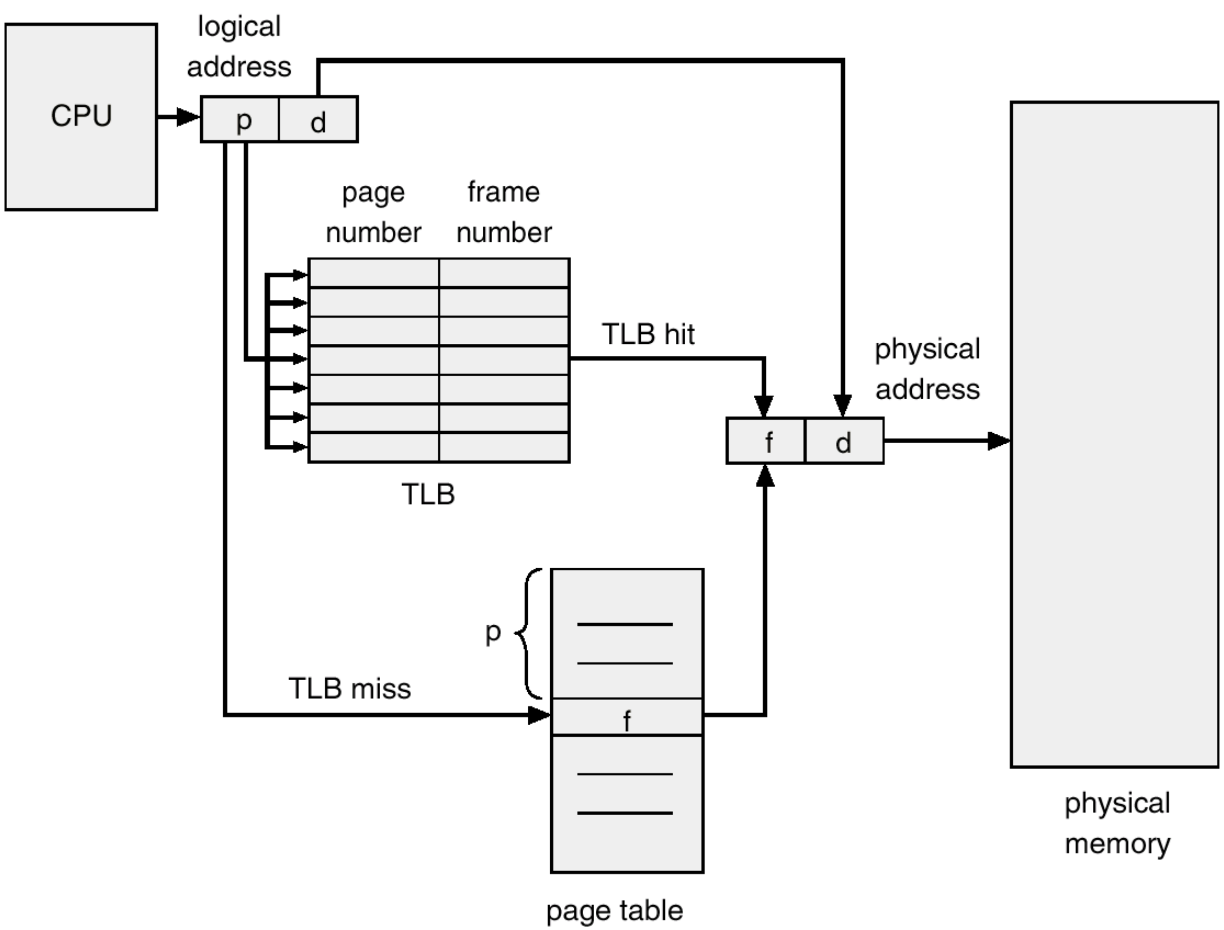
- TLB reach can increase by increasing page size
  - However, with large page, fragmentation also increases.

  ➔ 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
TLB
physical address
f
d
p
TLB miss
f
page table
physical memory

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;
```

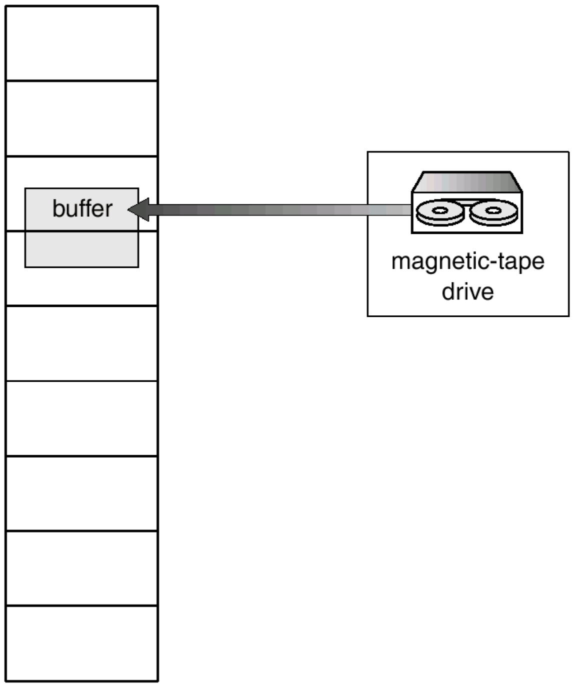
A

```
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 (COW)
- Page replacement
- Allocation of frames
- Thrashing
- Memory-mapped files
- Allocating kernel memory
- Other considerations
- <u>Operation-system examples</u>

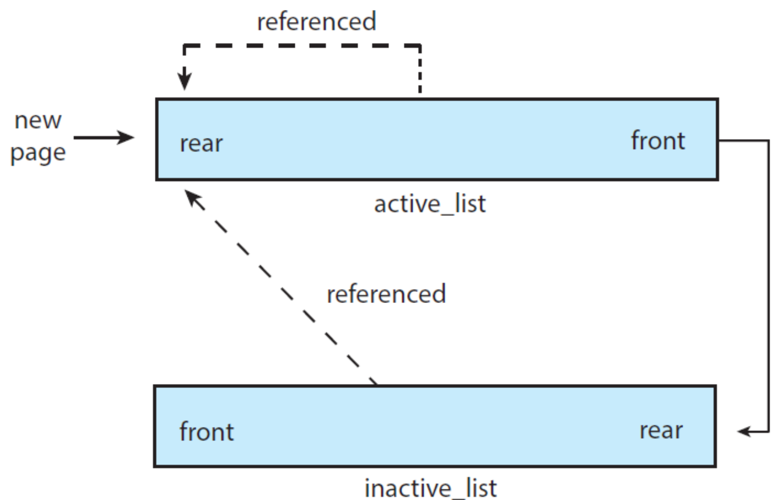
Linux

- Demand paging
  - Allocates pages from a list of free frames.
  - A global page-replacement policy similar to the LRU-approximation clock algorithm

- Accessed bit
  - Each page has an accessed bit that is set whenever the page is referenced.
  - Periodically, the accessed bits for pages in the active list are reset

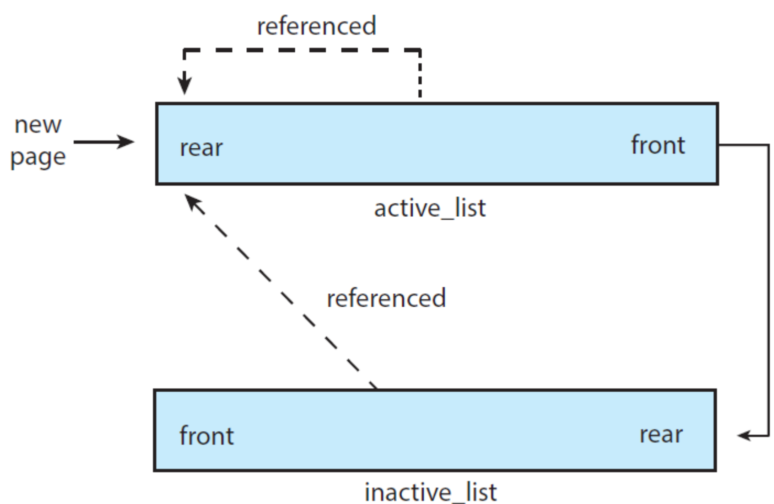
Linux

- Linux maintains two types of page lists
  - Active list contains the pages that are considered in use
    - Over time, the least recently used page will be at the front of the active list.
  - Inactive list contains pages that have not recently been referenced and are eligible to be reclaimed.
    - If a page in the inactive list is referenced, it moves back to the rear of the active list



Linux

- Kernel swap daemon process kswapd
  - A page-out daemon process
  - Periodically awakens and checks the amount of free memory in the system
  - If free memory falls below a certain threshold, kswapd begins scanning pages in the inactive list and reclaiming them for the free list



Windows

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

- Working-set minimum/maximum
  - Working-set minimum (default=50): the virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active
  - Working-set maximum (default: 345): the virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active and available memory is low.