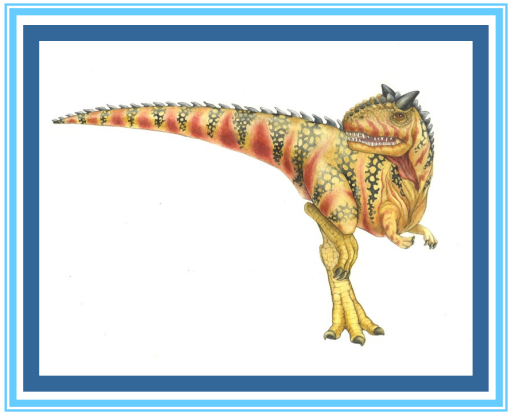
Chapter 10: Virtual Memory



Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Operating-System Examples

Objectives

- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process, and explain how it is related to program locality.
- Describe how Linux and Windows 10 manage virtual memory.

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

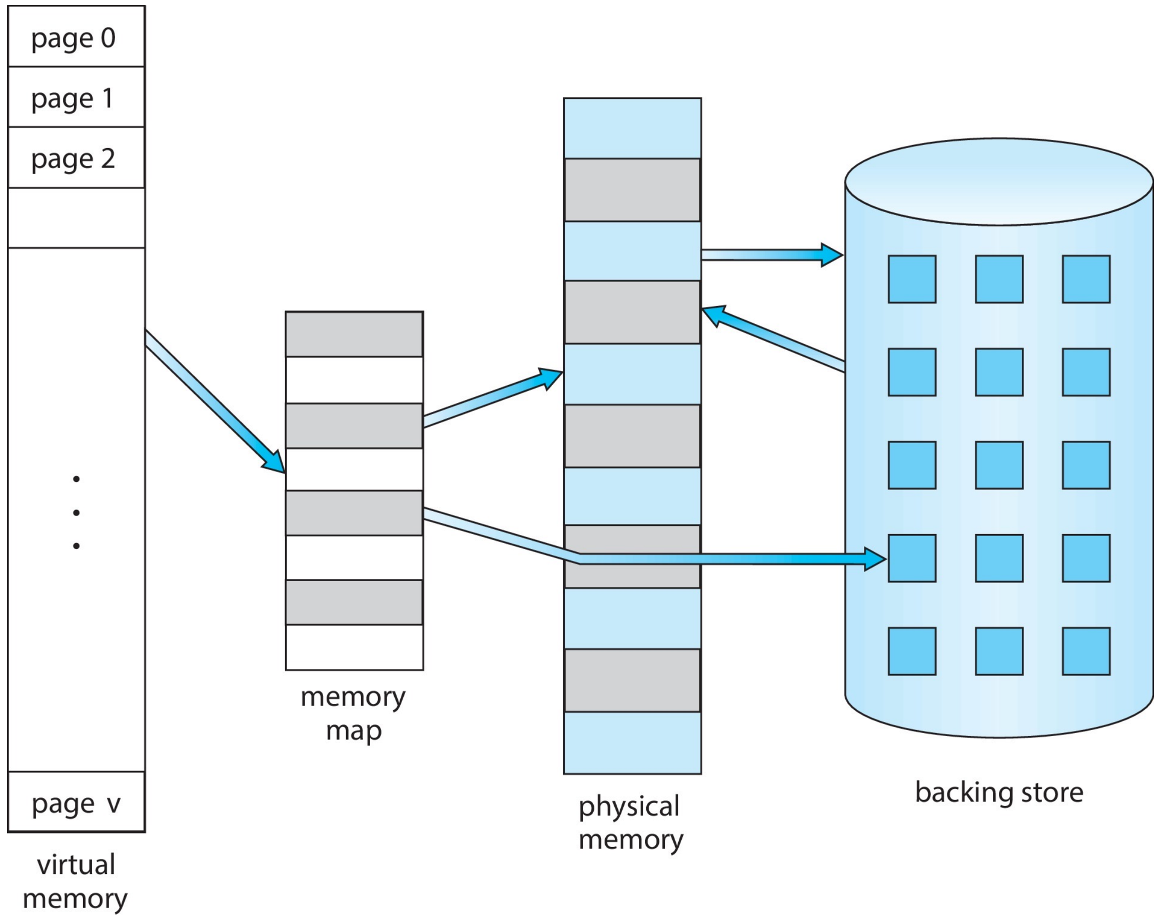
Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

Virtual memory (Cont.)

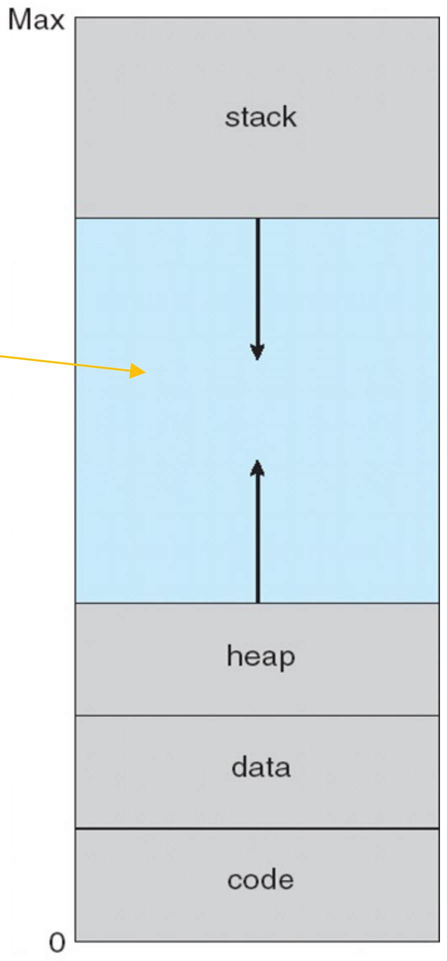
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

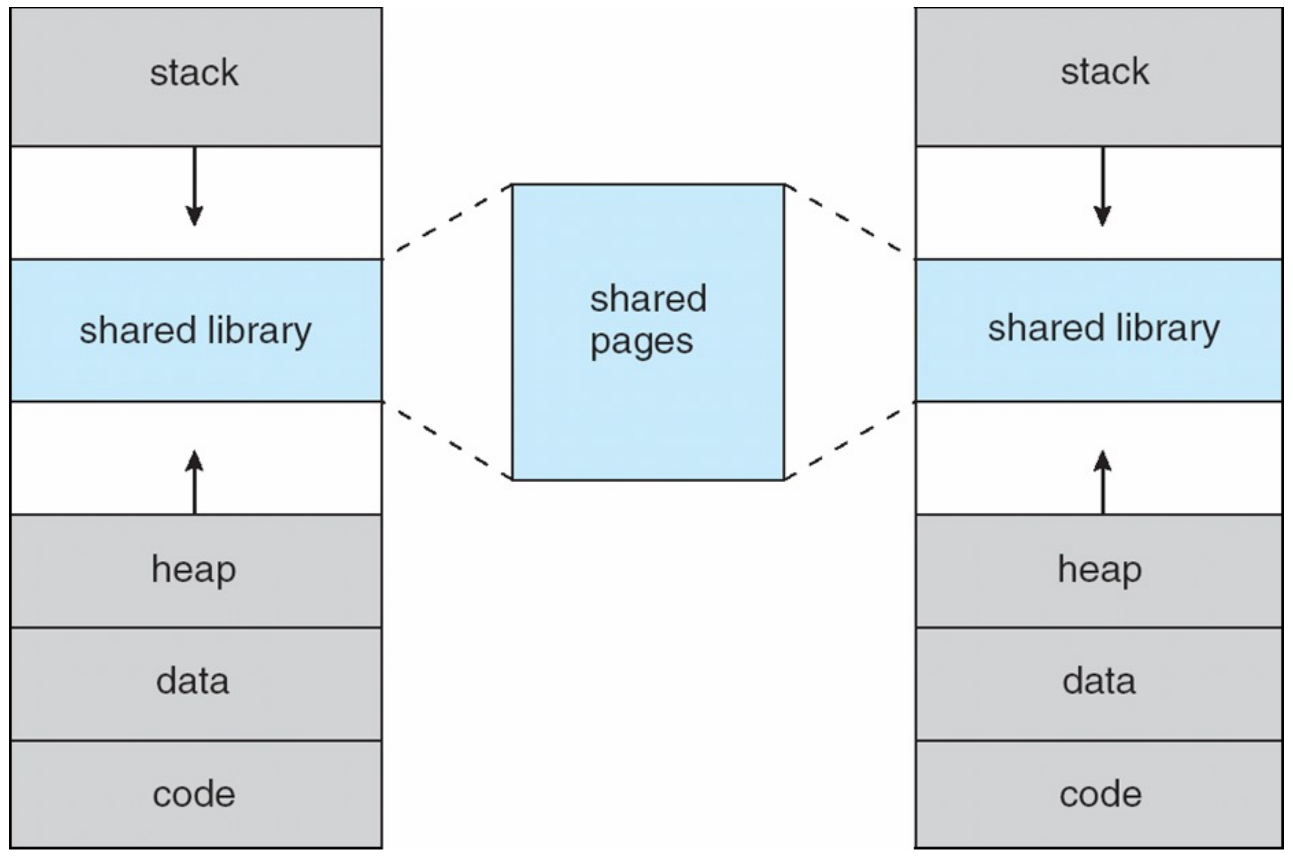


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



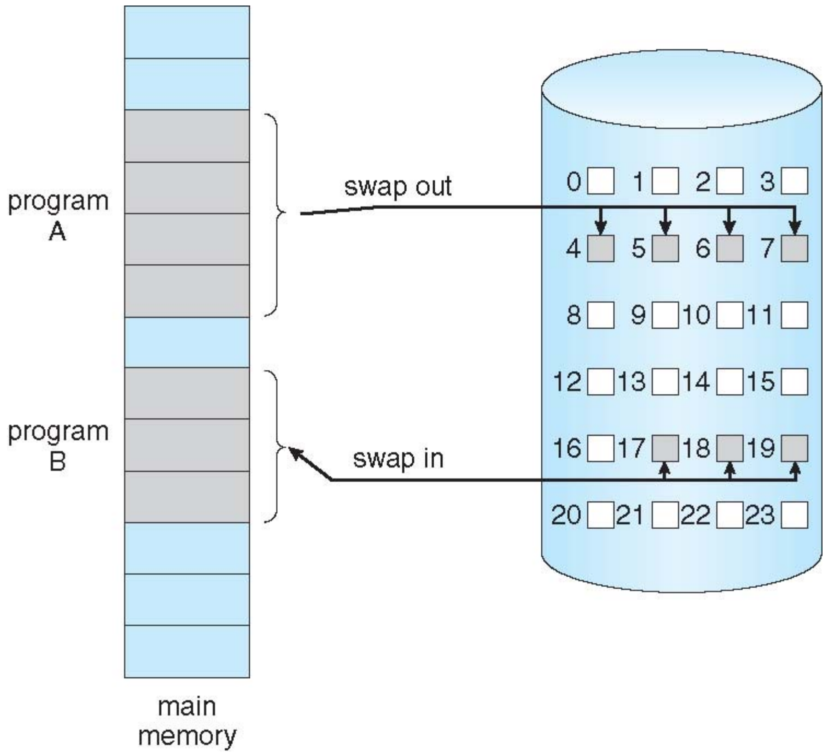
Shared Library Using Virtual Memory



Demand Paging

Swapping: 전체 프로세스
Swapping with paging: 일부 페이지
Demand paging: 필요한 페이지만

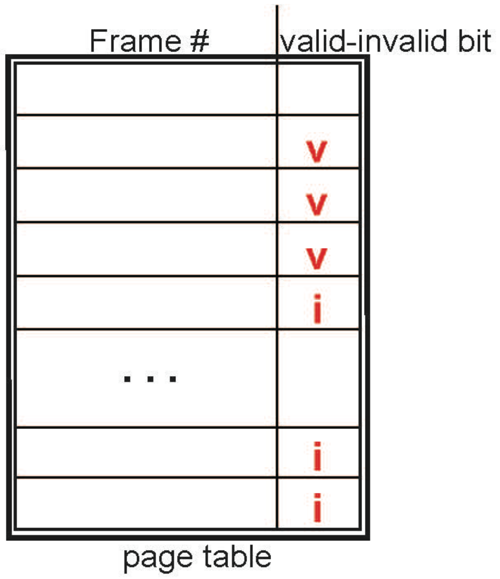
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory



이 그림은 Swapping을 나타냄

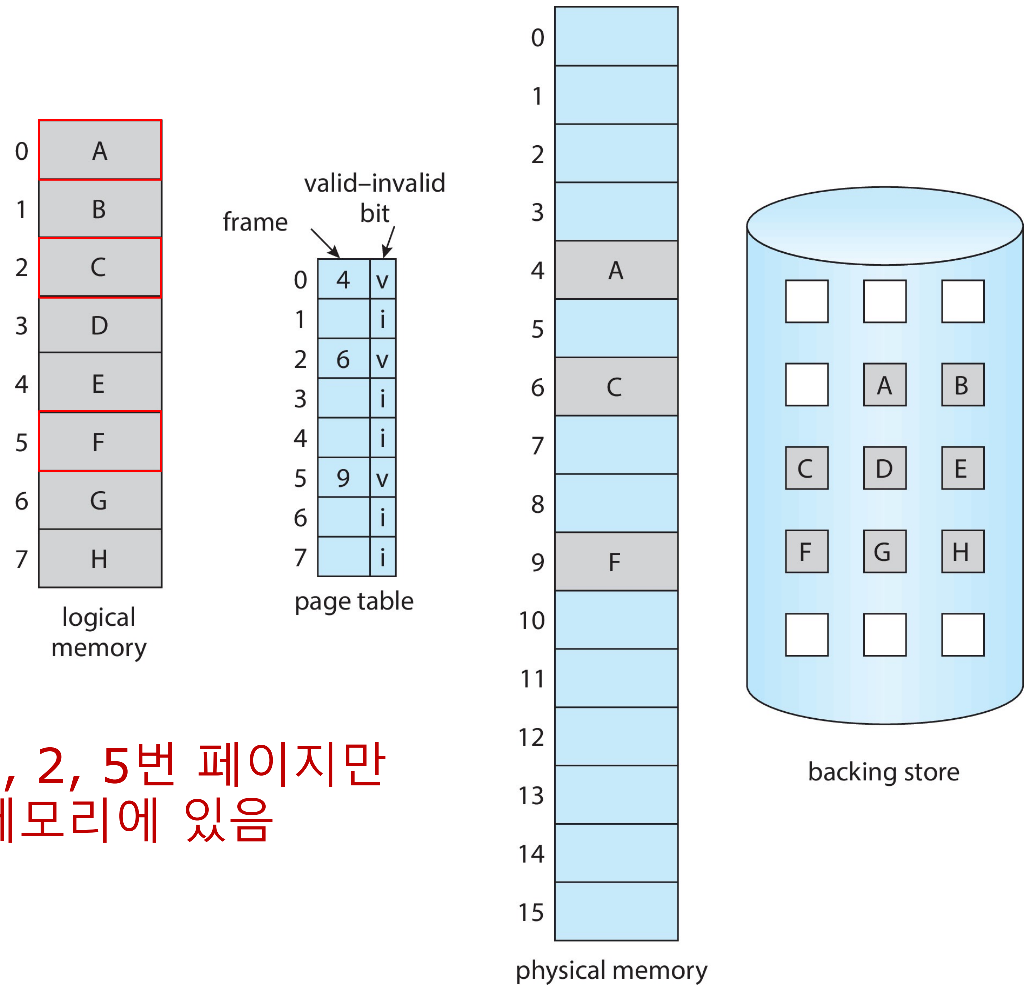
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



현재 0, 2, 5번 페이지만
메인 메모리에 있음

Steps in Handling Page Fault

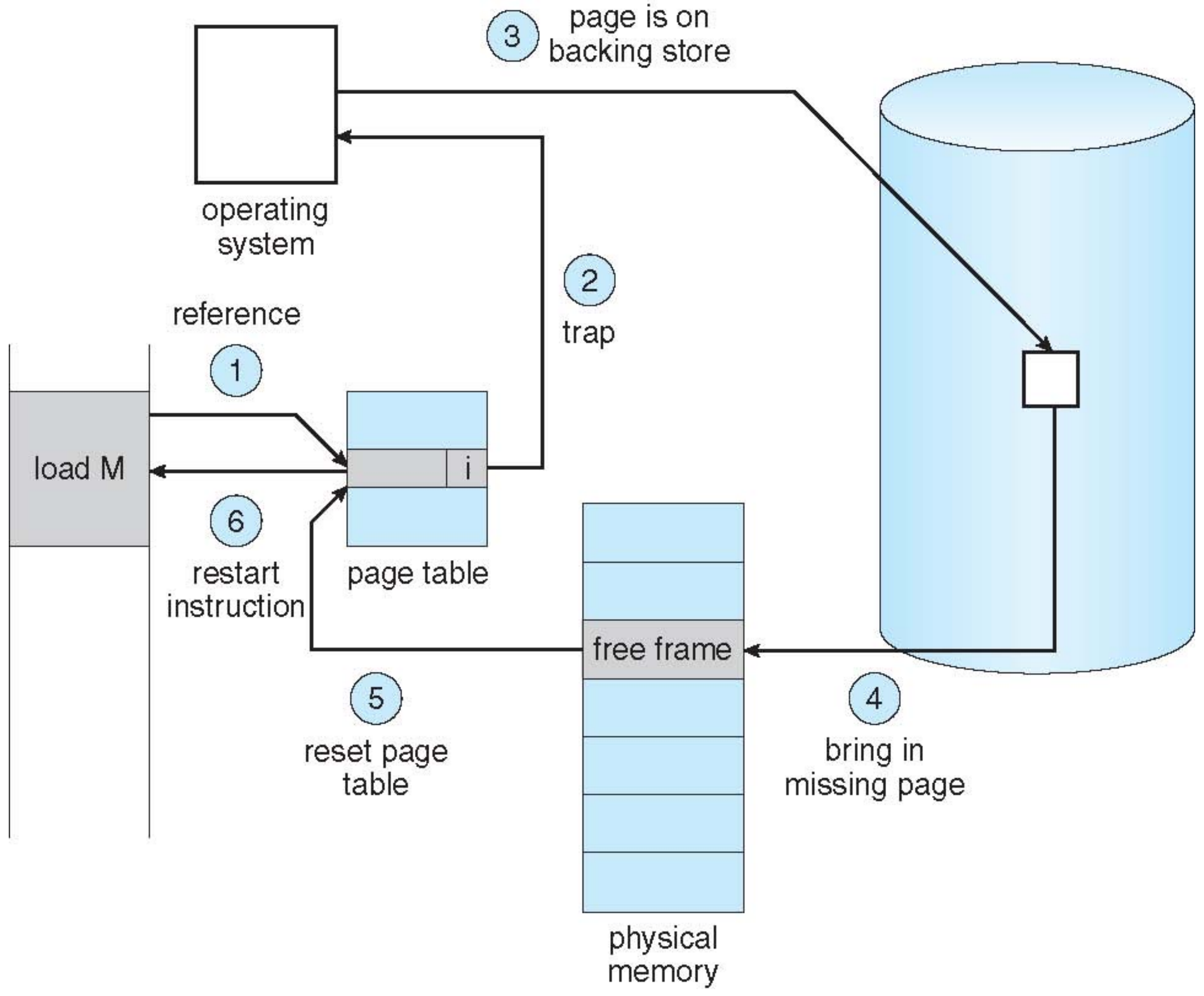
1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

Invalid bit는 2가지 의미: ① 주소 범위 밖에 있는 경우, ② 메모리에 없는 경우.

Major page fault: 페이지가 메모리에 없는 경우
Minor page fault: 메모리에 있지만 매핑이 안 되어 있는 경우 (shared library), 또는 원하는 페이지가 free-frame list에 아직 남아 있는 경우

Linux는 minor page fault가 major page fault 보다 월등히 많다.
이는 shared libraries를 많이 활용하고 있다는 증거이다.

Steps in Handling a Page Fault (Cont.)



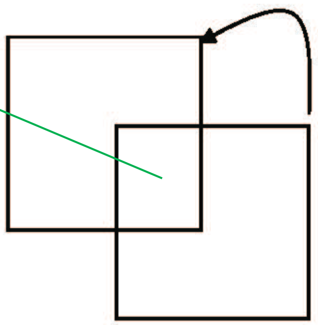
Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging** → 시작할 때 메모리에 어떤 페이지도 없는 방식
    ↔ prepaging: 메모리에 일부를 미리 로딩
- Actually, a given instruction could access multiple pages -> multiple page faults 드물지만 한 명령어에 최대 4번의 page fault도 가능
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations
  - Block move: IBM 360/370 MVC instruction

옮기는 중간에 page fault가 일어나서 명령어를 다시 실행할 때 겹치는 부분에서 원본을 손실한 경우가 발생



두 가지 해법 가능

① 페이지 폴트가 일어나기 전에 미리 페이지를 불러들이고 명령어를 실행

② 임시 레지스터에 겹치는 부분을 저장해 두었다가 page fault가 일어나면 restore함

  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Stages in Demand Paging – Worse Case

"Page fault 처리 상세 과정"

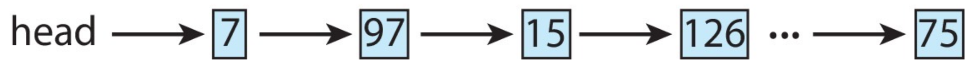
1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.

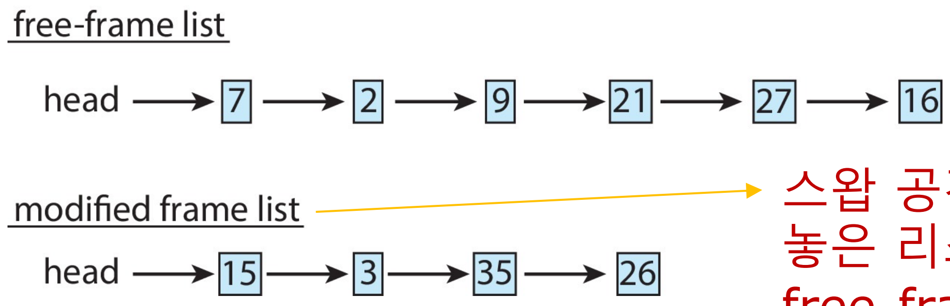


- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

Memory Compression

macOS, Windows, Android, iOS

- **Memory compression** -- rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.
- Consider the following free-frame-list consisting of 6 frames



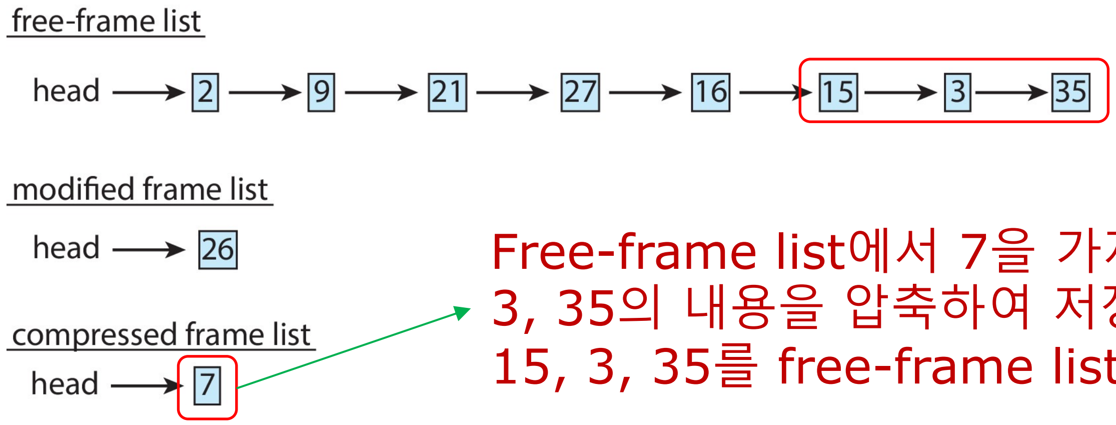
스왑 공간에 쓰기 전에 모아 놓은 리스트. 나중에 쓰게 되면 free-frame list로 옮긴다.

- Assume that this number of free frames falls below a certain threshold that triggers page replacement. The replacement algorithm (say, an LRU approximation algorithm) selects four frames -- 15, 3, 35, and 26 to place on the free-frame list. It first places these frames on a modified-frame list. Typically, the modified-frame list would next be written to swap space, making the frames available to the free-frame list.

Memory Compression (Cont.)

- An alternative to paging is **memory compression**.
- Rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages.

Android와 iOS는 swapping이나 paging을 지원하지 않는다. 대신 memory compression을 사용한다.



Free-frame list에서 7을 가져와서 15, 3, 35의 내용을 압축하여 저장하고, 15, 3, 35를 free-frame list로 옮김

macOS와 Windows 10도 memory compression을 지원한다. macOS의 memory compression은 SSD를 사용한 paging 시스템보다 빠른 것으로 알려져 있다.

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

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

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p x (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800 $\longrightarrow$ EAT는 p에 비례
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system

스왑공간은 파일시스템보다 빠르다

- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

프로세스 전체를 시작 전에 스왑공간에 복사

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system

변하지 않는 바이너리 이미지는 파일시스템에서 읽어오고 anonymous memory는 스왑공간 이용

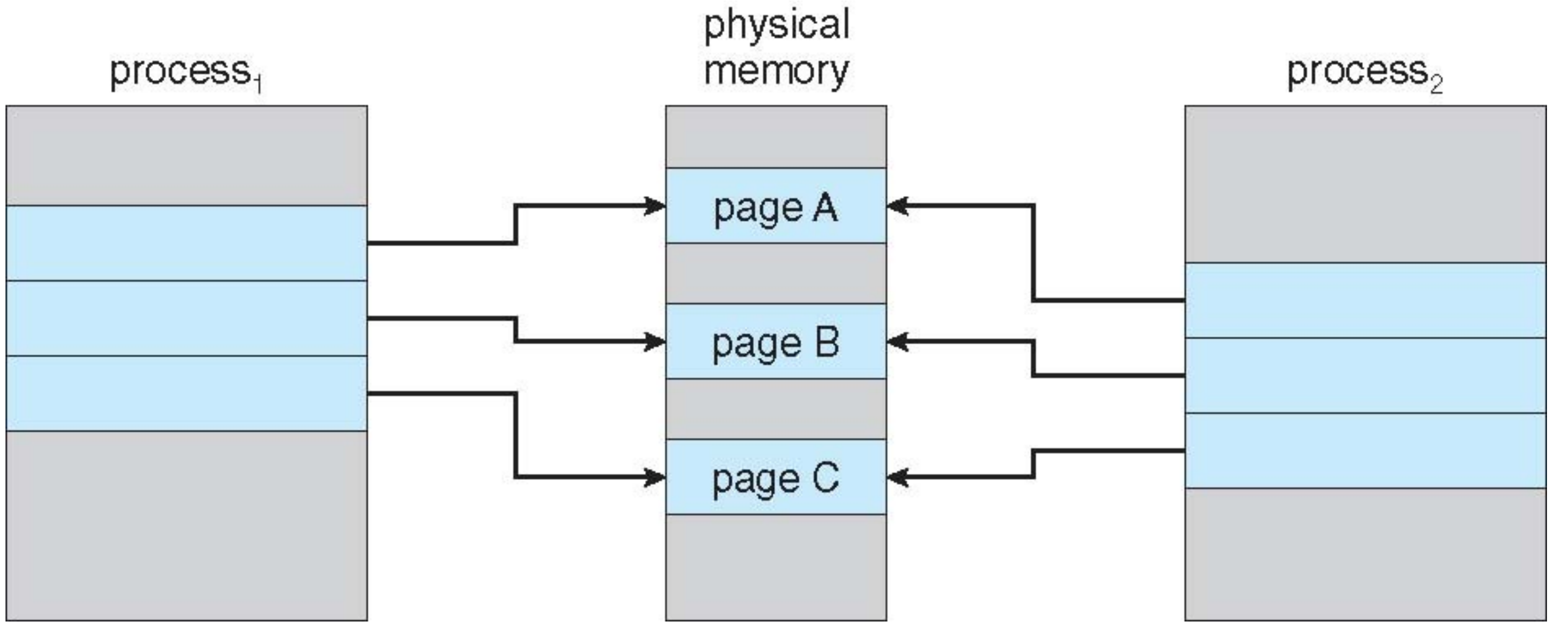
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

스와핑을 지원하지 않고 파일시스템을 사용하거나 메모리 압축 기법 사용

Copy-on-Write

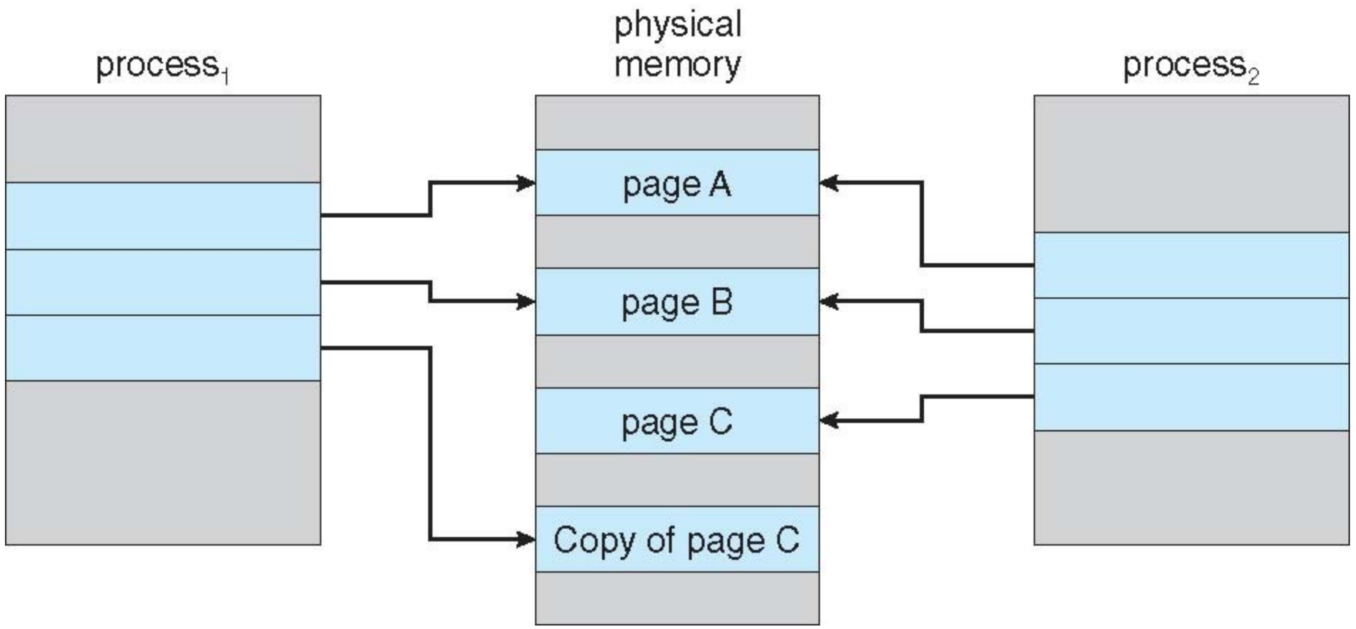
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it? Ans: Security
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent. ex) Linux, macOS, BSD Unix
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



페이지 C 변경 전

After Process 1 Modifies Page C



페이지 C 변경 후

What Happens if There is no Free Frame?

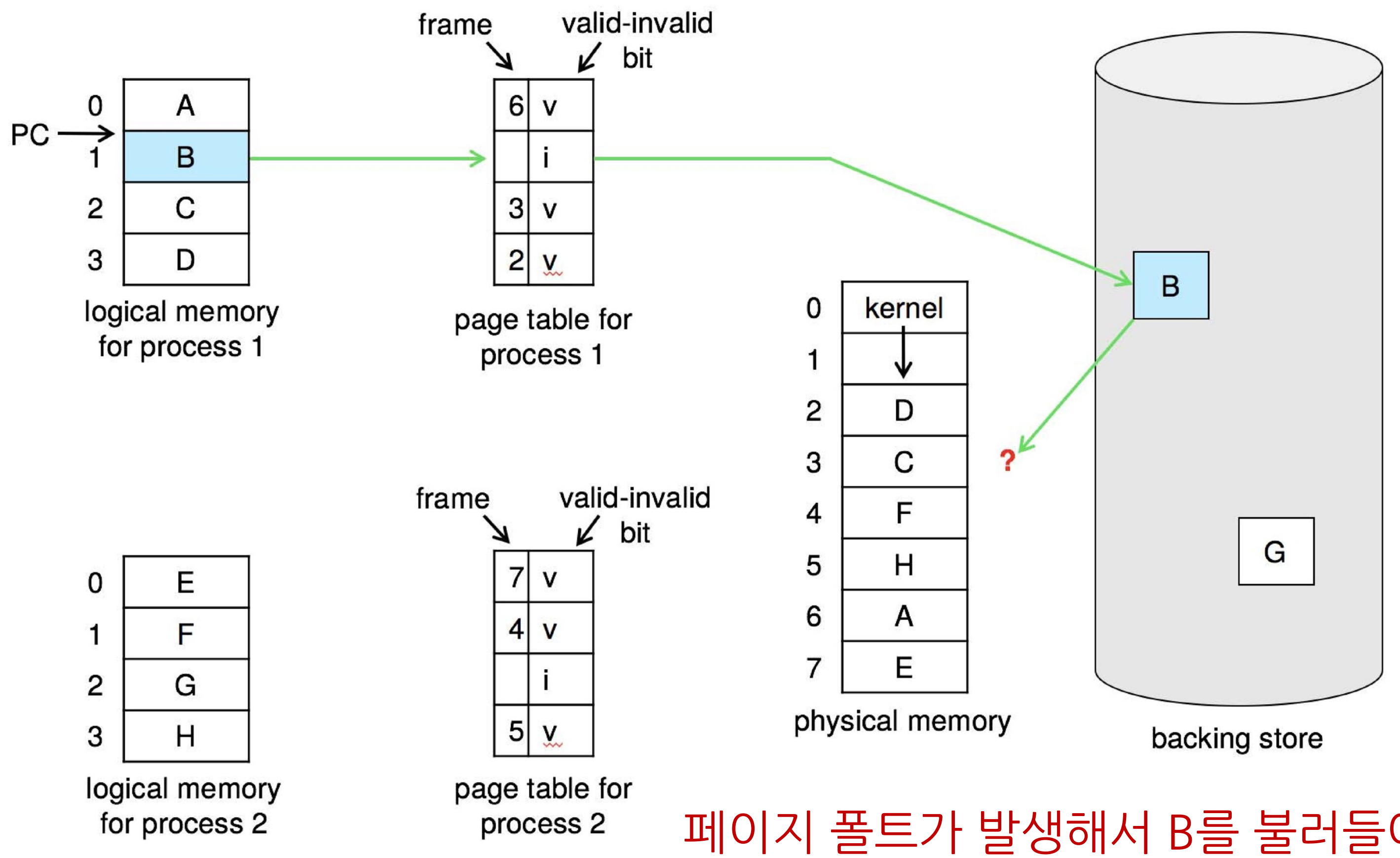
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

프레임이 모자르면 메모리에 있지만 잘 사용하지 않는
페이지를 내보내야 하는데 이것을 “페이지 교체”라고 부른다.

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



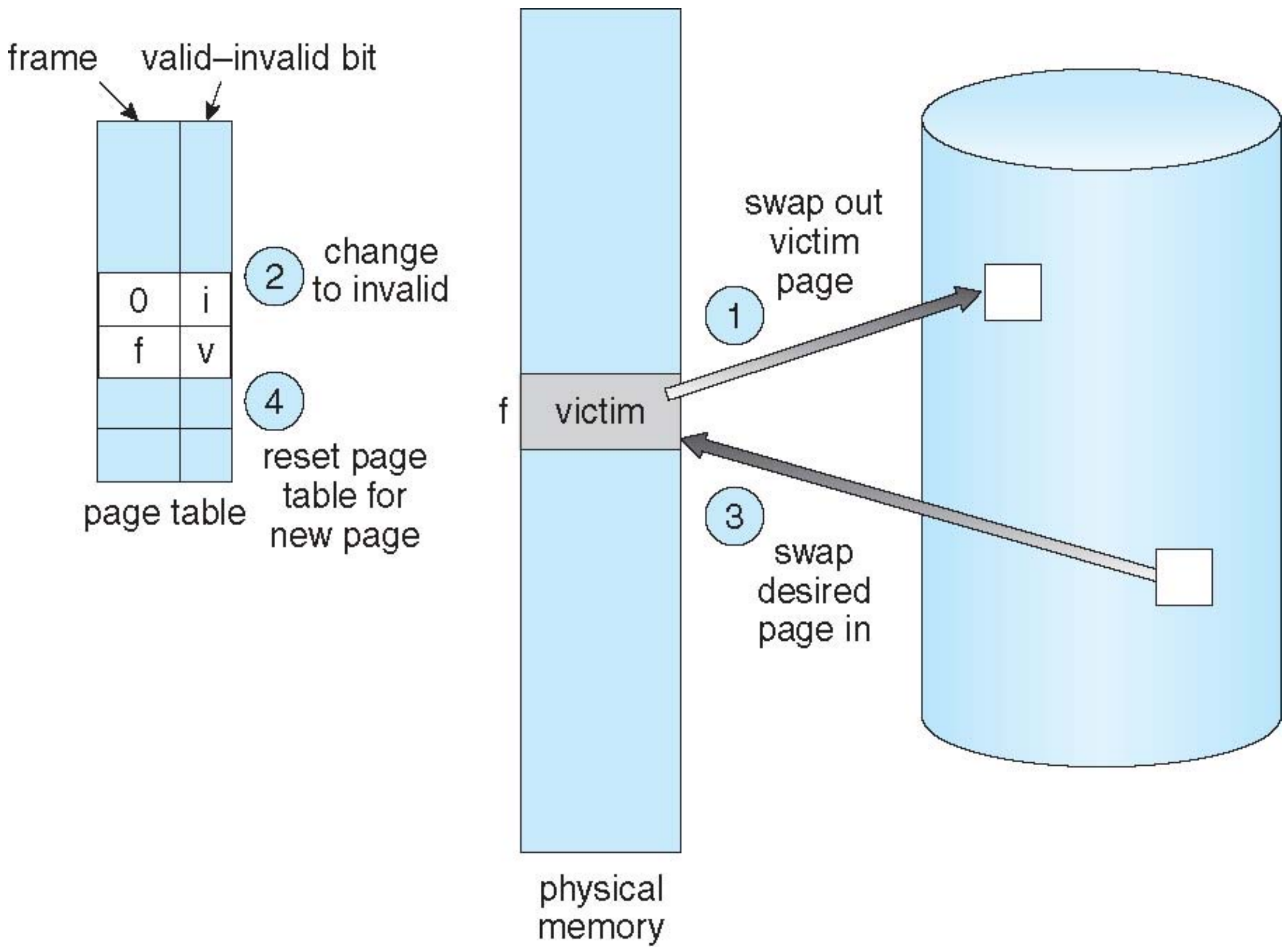
페이지 폴트가 발생해서 B를 불러들여야 하는데, 빈 프레임이 없어서 누군가를 교체해야 하는 경우

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement



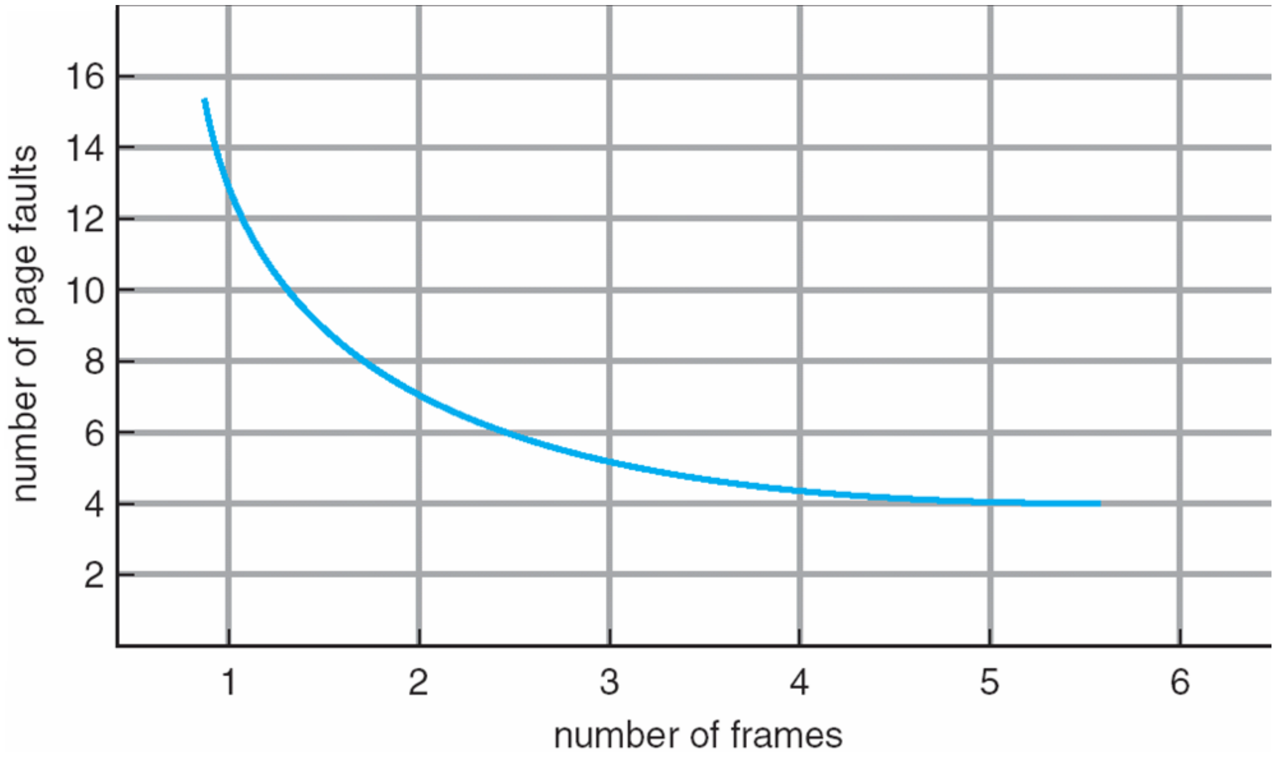
Page and Frame Replacement Algorithms

프로세스 당 몇 개의 프레임을 할당하는가?
어떤 프레임을 교체하는가?

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

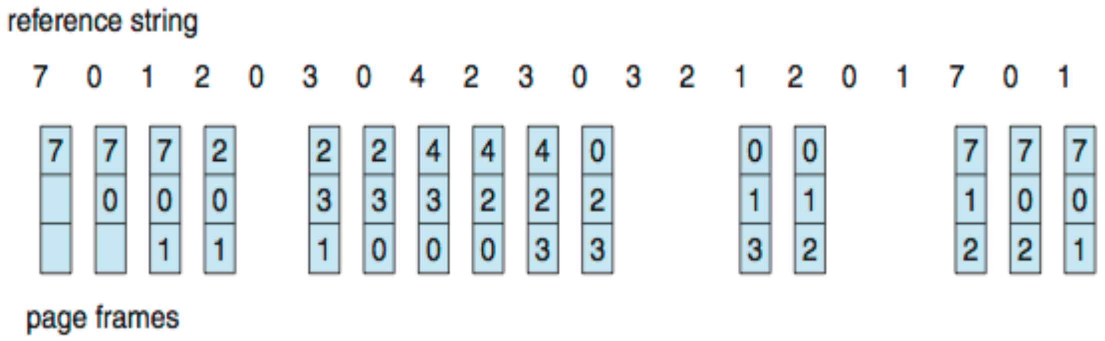
Graph of Page Faults Versus The Number of Frames



프레임을 늘리면 페이지 폴트가 감소하는 일반적인 모습

First-In-First-Out (FIFO) Algorithm

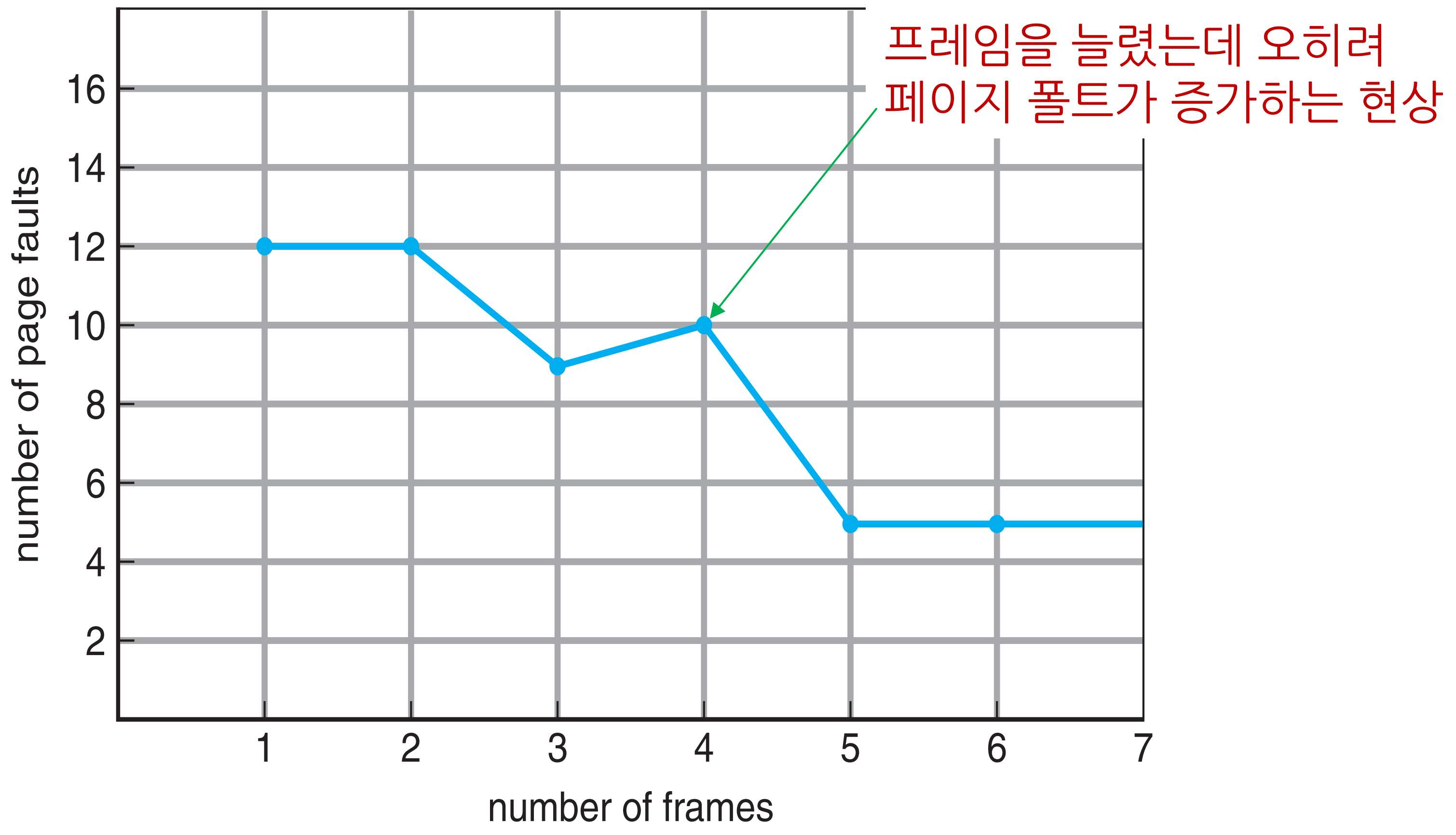
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

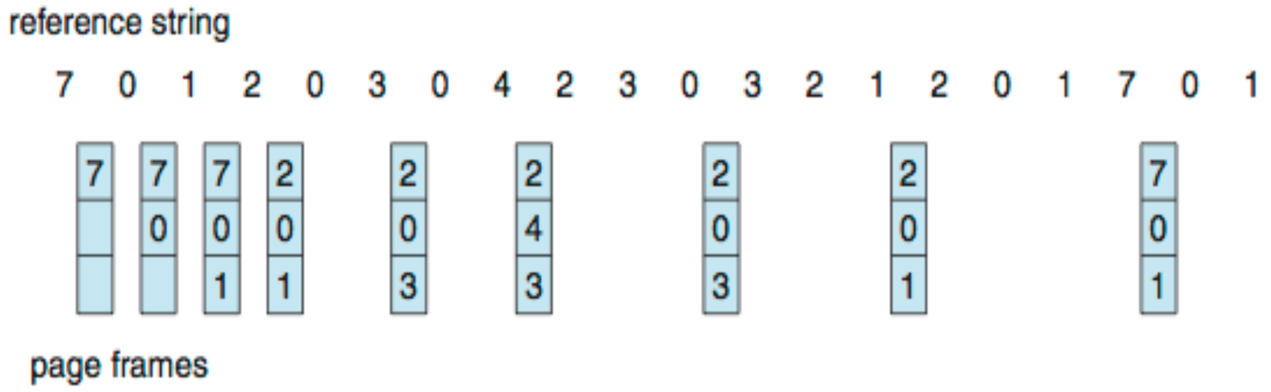
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



Optimal Algorithm

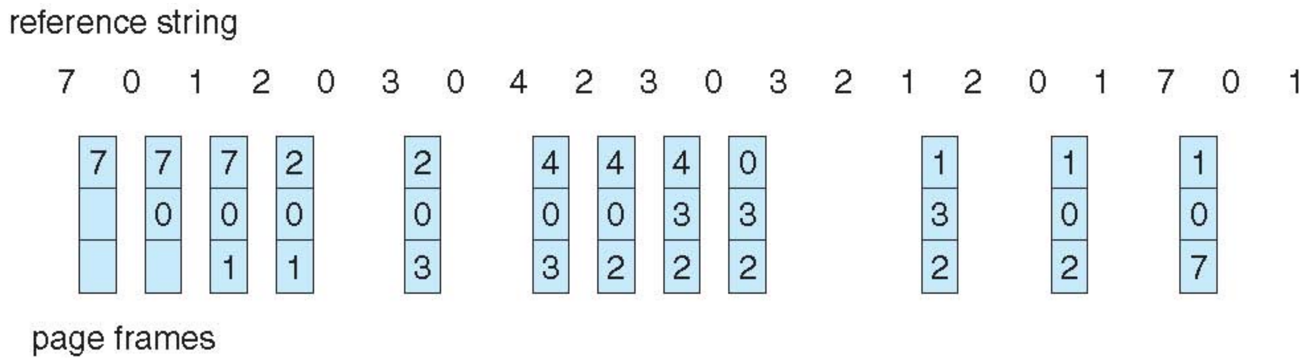
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future → 구현하기 어렵다.
- Used for measuring how well your algorithm performs



# of page faults for OPT(S) = # of page faults for OPT(S^R)
where S = reference string, S^R = reverse of S

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement? $\longrightarrow$ LRU approximation 사용

# of page faults for LRU(S) = # of page faults for LRU(S^R)

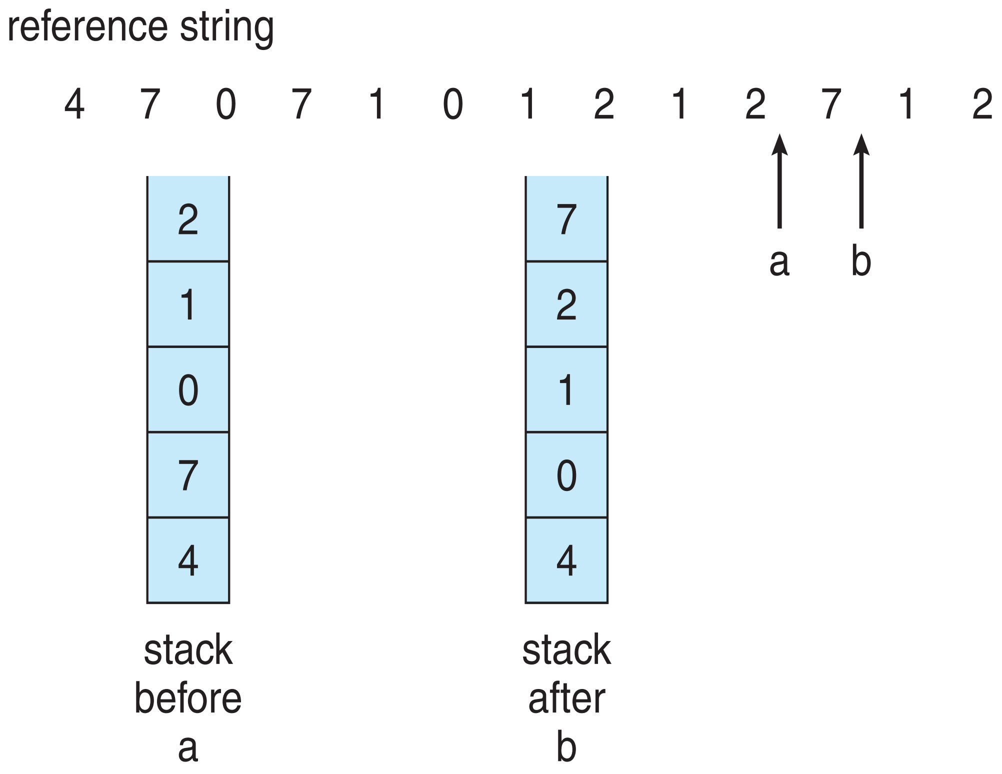
LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don’t have Belady’s Anomaly

이 두 방식을 하드웨어 도움 없이 소프트웨어로만 한다면 적어도 10배 이상은 느려진다.

N개의 프레임을 사용했을 때 남아 있는 페이지의 집합이 N+1개의 프레임을 사용했을 때 남아 있는 페이지의 집합의 부분집합일 때, 우리는 스택 알고리즘이라 부른다.

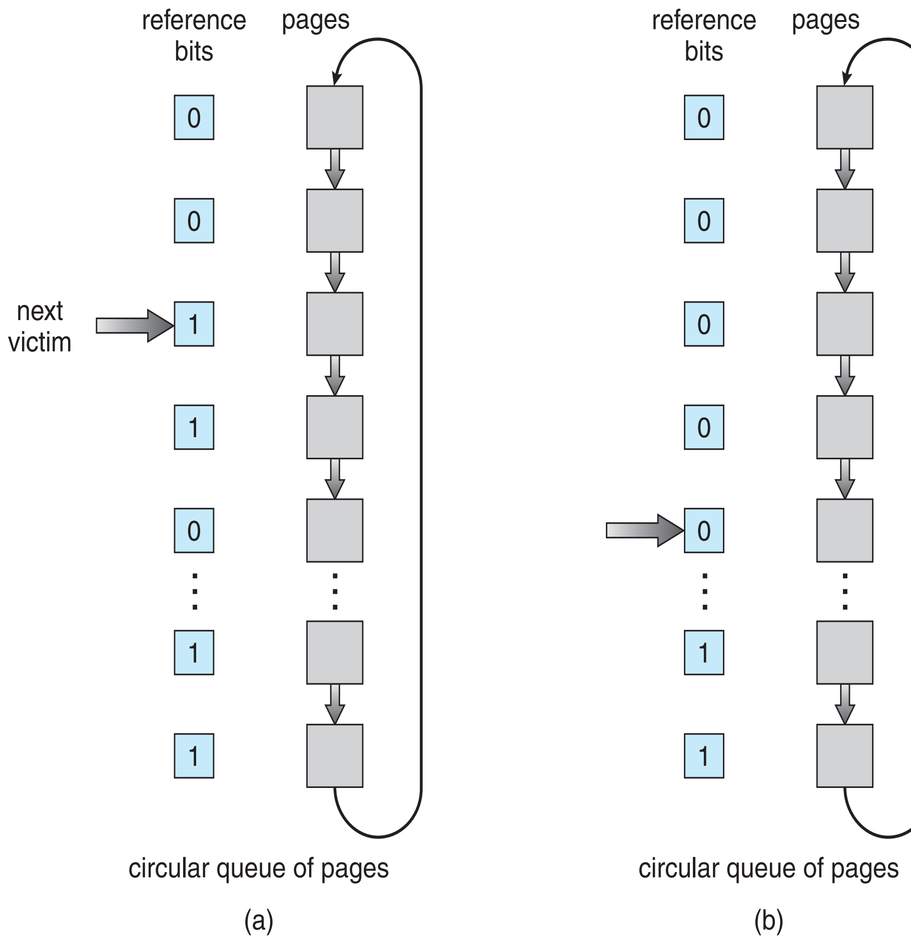
Use Of A Stack to Record Most Recent Page References



LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then: (second chance!)
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

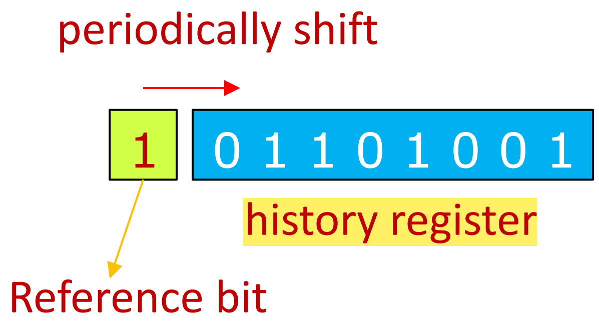
- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

Victim
우선순위

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

LRU Approximation Algorithms (Cont.)

- **Additional reference bits algorithm**
  - Timer 인터럽트를 통해 주기적으로 OS가 오른쪽으로 한 비트씩 이동



- **Demand paging and reference bit**
  - Demand paging의 경우 페이지가 불려오면 조회가 되므로 reference bit는 1로 세팅
  - Second-chance 알고리즘을 수행하는 방법, 주기적으로 모든 reference bit를 0으로 리셋하는 방법을 고려할 수 있다.

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common $\longrightarrow$ 구현이 비싸고, OPT 알고리즘에 근접하지 못함.
- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces page with smallest count
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim

Pool을 사용하여 페이지 폴트 시간에 찾지 않고 바로 할당

Pool을 사용하면 victim 페이지를 스왑 공간에 쓰는 것과 병행으로 실행 가능, 프로세스를 빨리 restart 시킬 수 있다

- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty

변경된 페이지를 페이징 디바이스가 놀 때, backing store에 써 놓으면 나중에 시간을 절약할 수 있다

- Possibly, keep free frame contents intact and note what is in them

프리 프레임의 내용이 backing store와 같으면 재사용

  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work

  DBMS와 같이 응용프로그램이 스스로 메모리와 I/O 버퍼를 관리하면 double buffering 현상이 발생할 수 있다.

- Operating system can given direct access to the disk, getting out of the way of the applications
  - Raw disk mode
    - Bypasses buffering, locking, etc

OS는 파일시스템을 우회하는 raw disk 모드를 제공함으로써 DBMS처럼 응용프로그램 자신에 최적화된 솔루션을 개발할 수 있다. 그럼에도 대부분의 응용프로그램은 그냥 파일시스템을 사용하는 것이 유리하다.

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*

CPU (instruction set, addressing mode etc.)에 따라 달라질 수 있다

- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- **Proportional allocation** – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

– s_i = size of process p_i

– $S = \sum s_i$

– m = total number of frames

– a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m =$ ~~64~~ 62

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

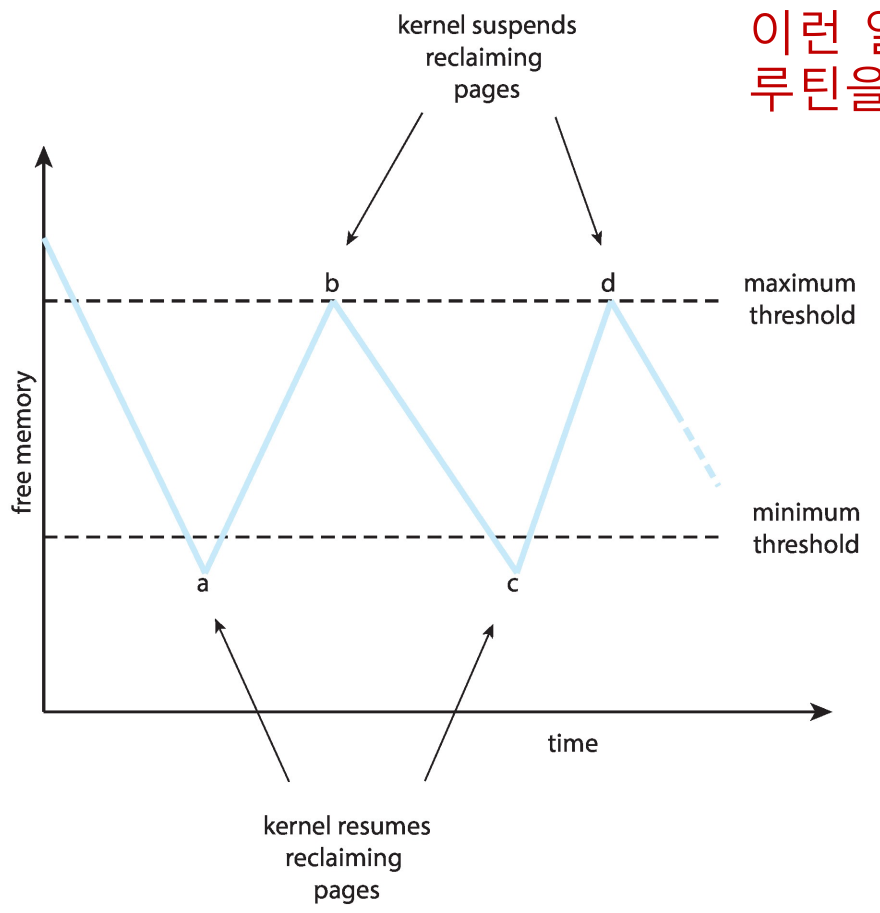
할당된 프레임의 수가 자신은 물론 다른 프로세스에 의해 영향을 받기 때문에

Reclaiming Pages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

메모리 요청이 항상 성공할 수 있도록 프리 프레임의 수가 어떤 임계치에 도달하면 페이지 교체를 미리 실행하는 전략

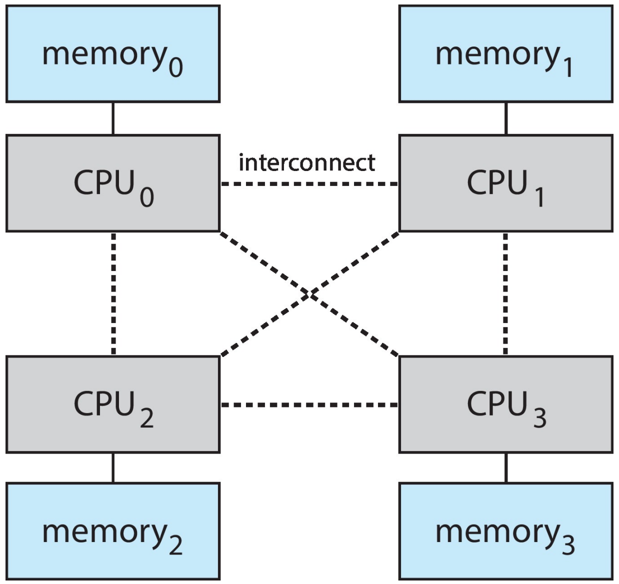
Reclaiming Pages Example



이런 일을 수행하는 커널 루틴을 "reapers"라고 한다

Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



Linux는 NUMA 노드마다 별도의 free frame list를 관리한다

Non-Uniform Memory Access (Cont.)

- Optimal performance comes from allocating memory "close to" the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups** (= locality group)
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

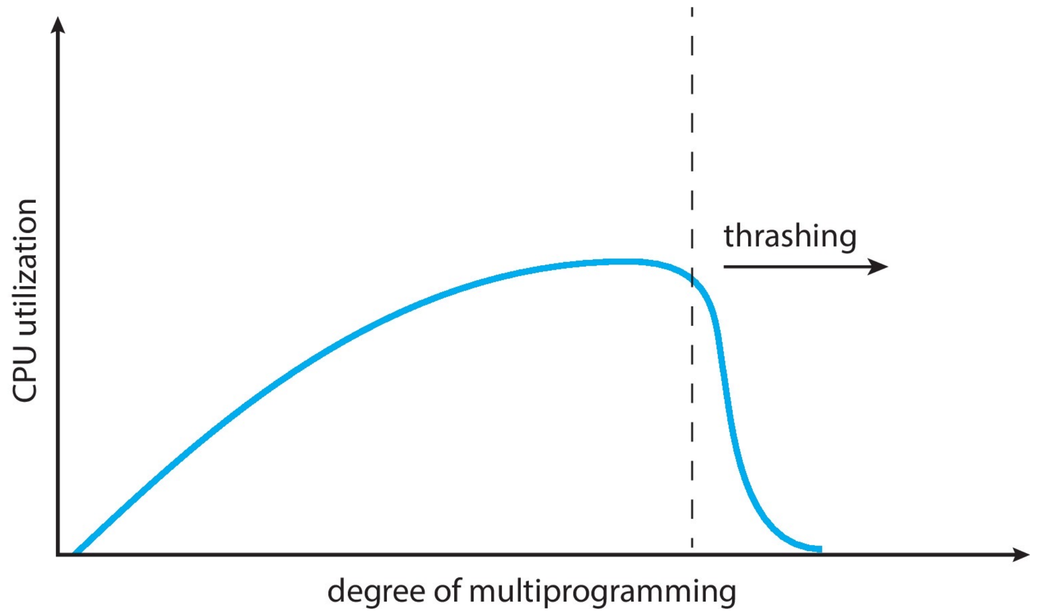
Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to: (점점 더 악화되는 이유)
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

프로세스가 프로그램 실행보다 페이징에 더 많은 시간을 보낼 때 이것을 thrashing이라고 부른다

Thrashing (Cont.)

- **Thrashing.** A process is busy swapping pages in and out



Demand Paging and Thrashing

- Why does demand paging work?

  같이 활발하게 사용되는 페이지의 집합

  $$\begin{tikzcd}\text{Locality model} \arrow[r] & \text{같이 활발하게 사용되는 페이지의 집합}\end{tikzcd}$$

  Locality model

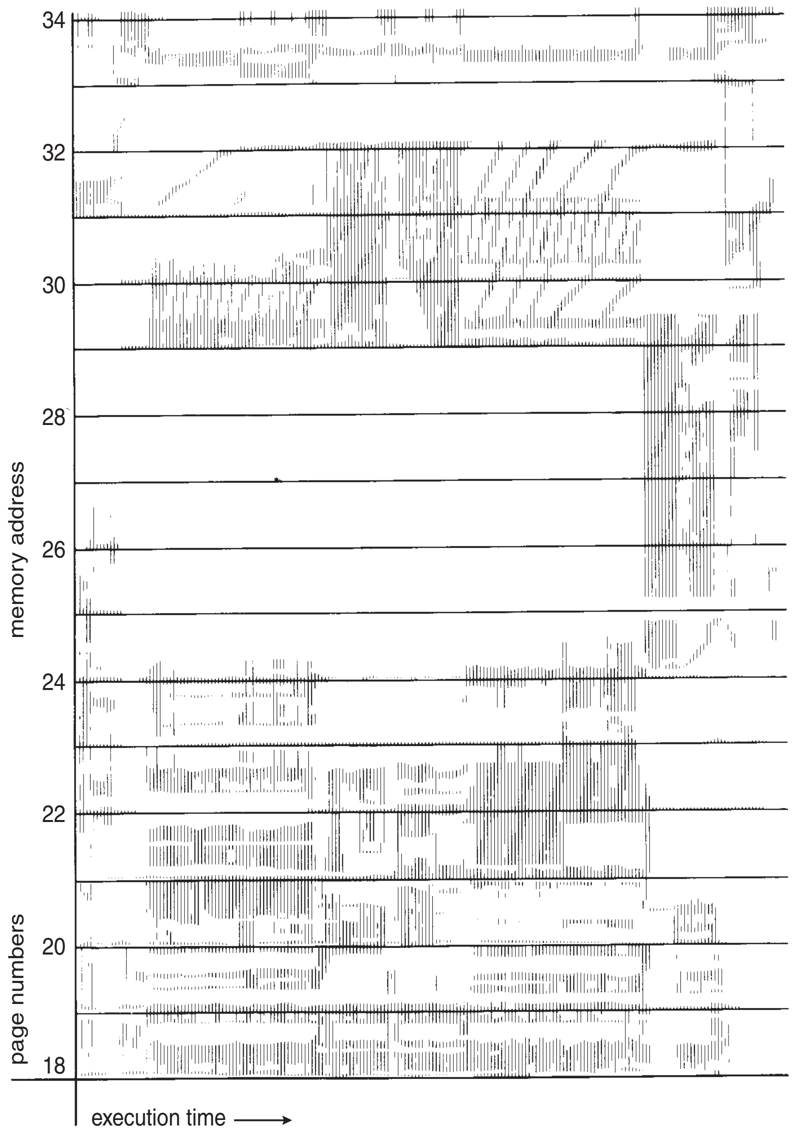
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size

- Limit effects by using local or priority page replacement

Local page replacement algorithm도 thrashing 문제를 완전히 해결하지는 못한다. 왜냐하면 어떤 프로세스가 thrashing이 일어나면 paging device가 바빠지므로 다른 프로세스는 thrashing이 아니더라도 EAT가 나빠진다.

Locality In A Memory-Reference Pattern



프로세스의
지역성을 수용할
충분한 메모리가
할당되지 않으면
thrashing이
일어난다

Working-Set Model

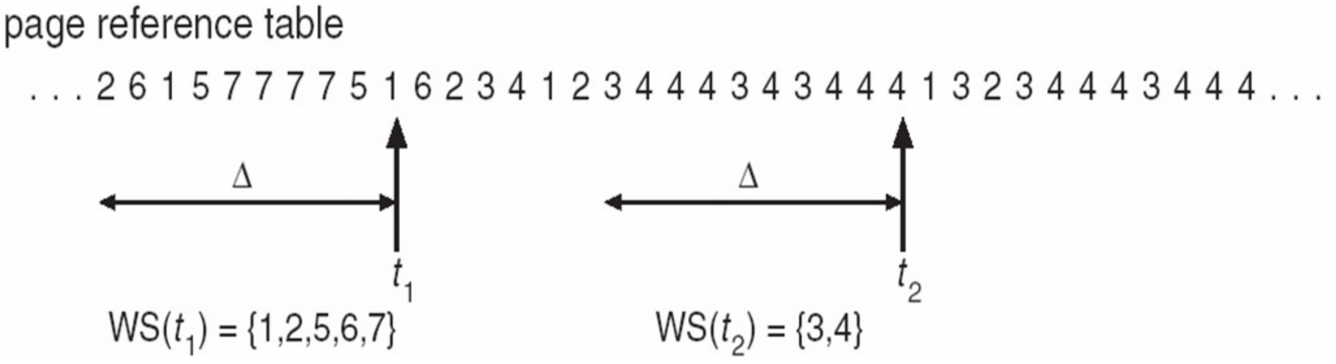
Δ시간 동안 조회된 페이지의 집합

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set size of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

메모리 프레임의 수



문제는 working set을 어떻게 추적하느냐에 있다.
Timer와 reference bit를 사용하여 근사치를 계산

Keeping Track of the Working Set

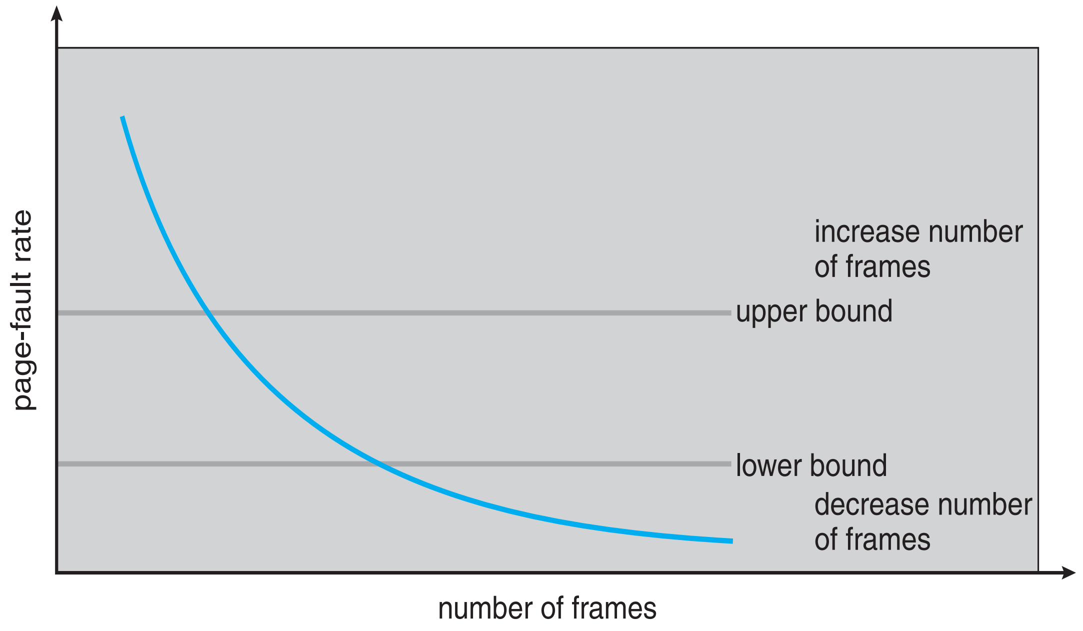
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units (비용 증가)

3 bits 사용 효과:
1 reference bit +
2 history bits

Page-Fault Frequency

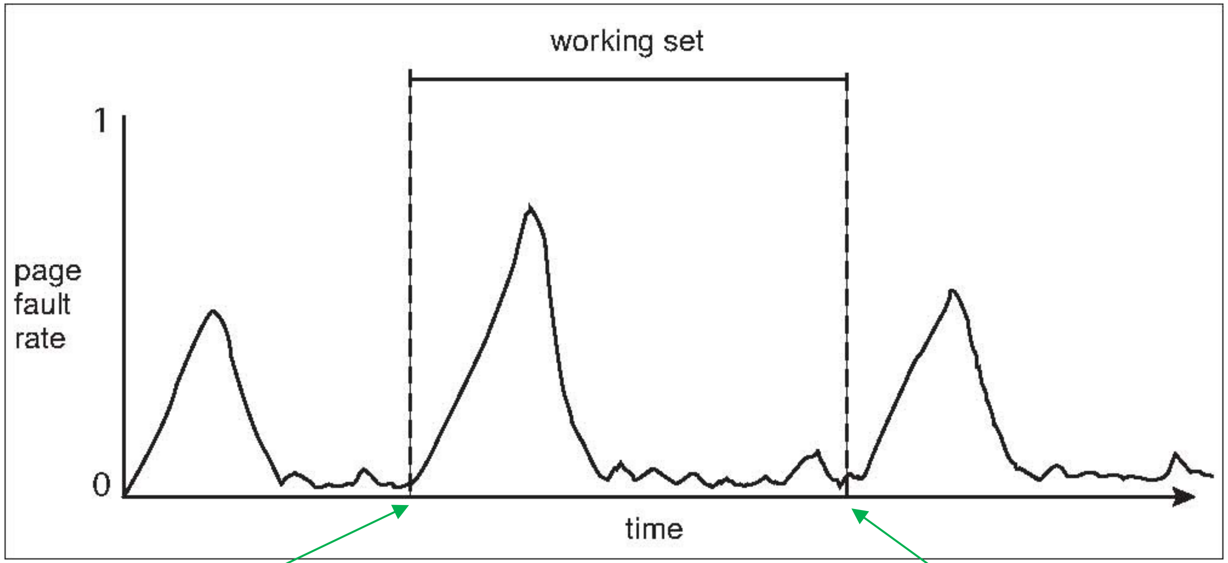
- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame

Current "best practice" → Get enough physical memory



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 fault가 일어나기 시작

Page fault가 다시 일어나기 시작

Allocating Kernel Memory

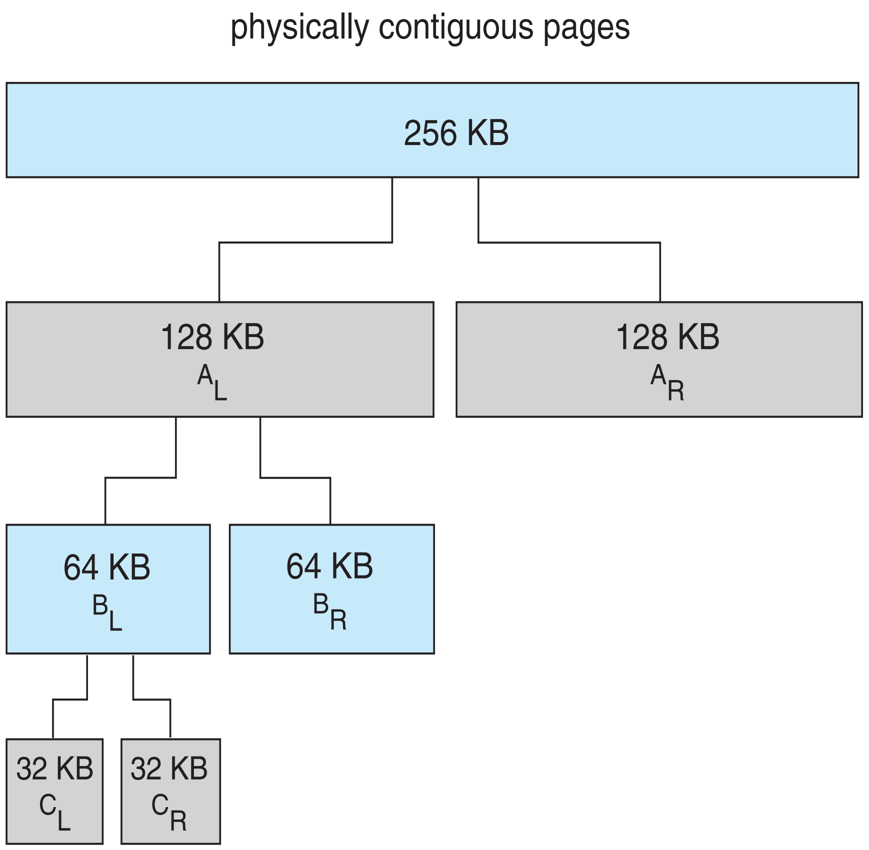
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

커널의 코드나 데이터가 페이징
시스템과 잘 맞지 않는다

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

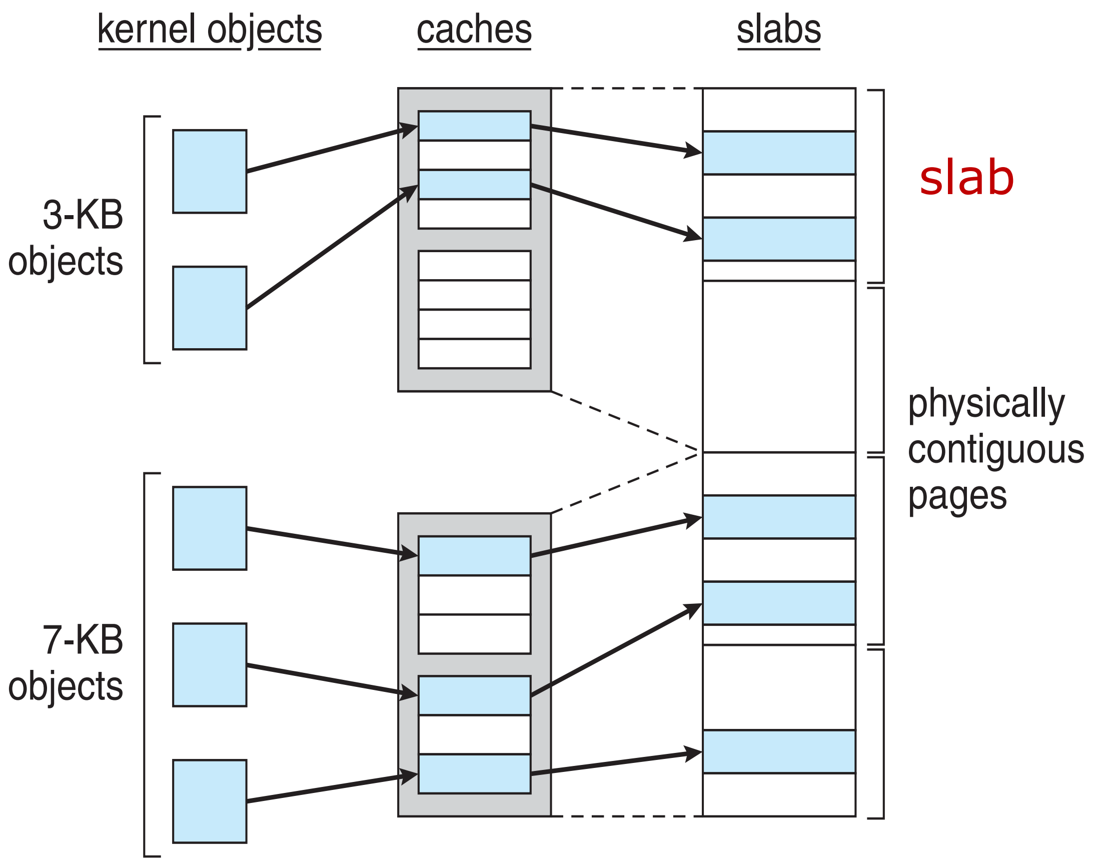
Buddy System Allocator



Slab Allocator

- Alternate strategy 하나 이상의 물리적으로 연속적인 페이지
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs → 하나 이상의 슬랩으로 구성
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure → 커널 자료구조마다 별도의 캐시 사용
- 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 allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used
  - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging
    $s * (1- \alpha)$ unnecessary pages?
  - α near zero $\Rightarrow$ prepaging loses

If cost(s prepaging) < cost(s*α page fault)
then prepaging wins

실행 파일에 대한 prepaging은 예측하기 어렵지만,
데이터 파일에 대한 prepaging은 예측이 가능할 수 있다.
특히 순차적으로 접근하는 경우는 더욱 그렇다.

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation (▴)
  - Page table size (▲)
  - Resolution (▴)
  - I/O overhead (▲(횟수), ▴(양))
  - Number of page faults (▲)
  - Locality (▴)
  - TLB size and effectiveness (▲)
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

▴ 작을수록 유리
▲ 클수록 유리

TLB Reach

TLB로 접근 가능한 메모리 양

- **TLB Reach** - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

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

128 x 128 = 16,384 page faults

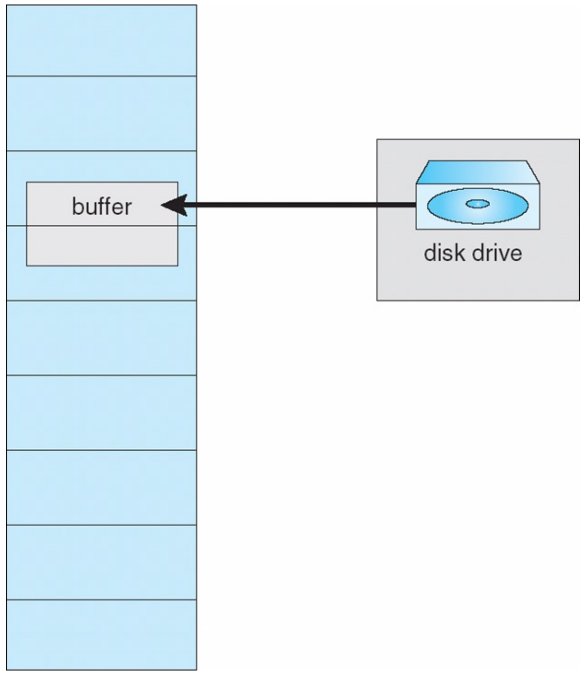
  - Program 2

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

128 page faults

I/O interlock

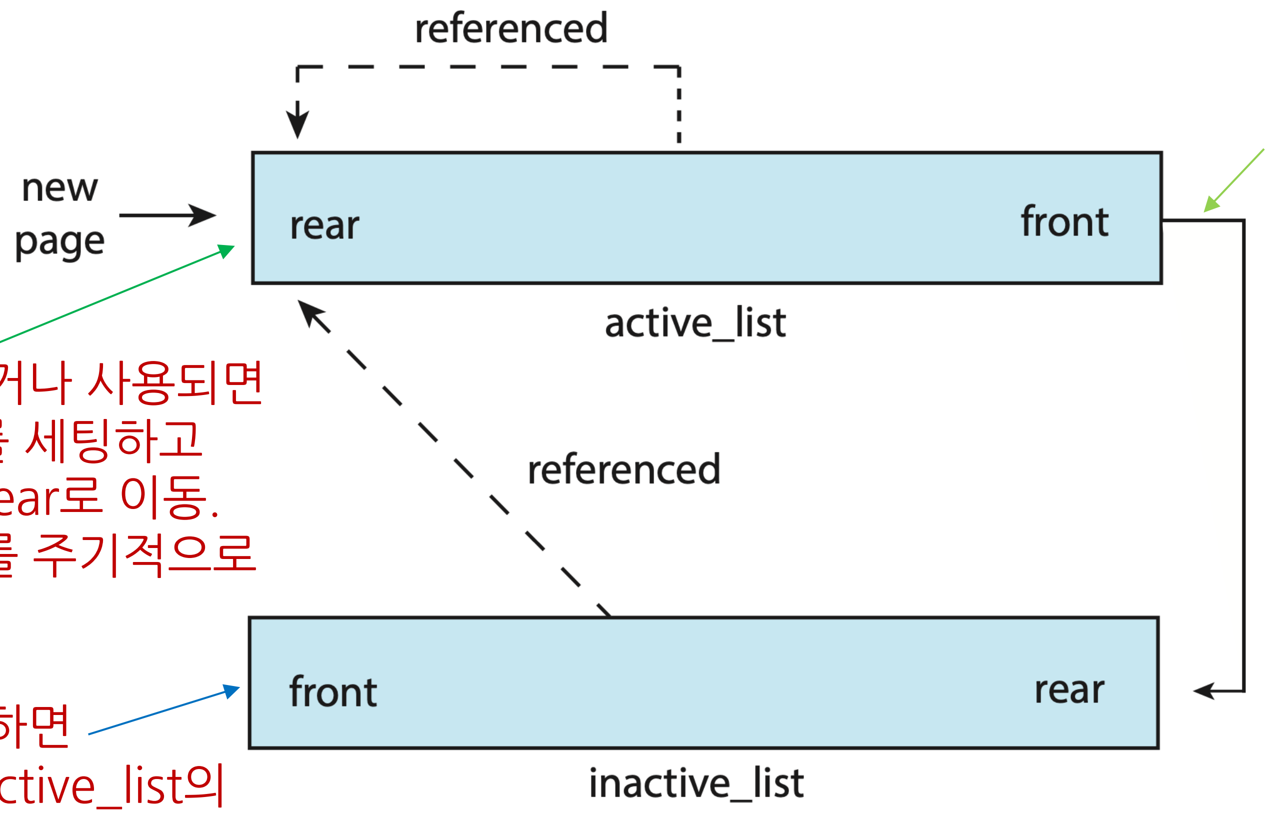
- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



I/O 버퍼로 사용될 페이지가 교체되면 문제 발생, 이를 해결하기 위한 해법 두 가지:
① I/O를 사용자 메모리에서 수행하지 않고 커널 영역에서 수행 (double buffering)
② 페이지 교체가 안 되도록 잠금 (pinning)

Linux

- Kernel memory: slab allocator
- Demand paging
- Global page-replacement policy
  - LRU-approximation clock algorithm: accessed bit (= reference bit)



이동하여 두 리스트의 균형을 맞춤

새로 할당되었거나 사용되면 accessed bit를 세팅하고 active_list의 rear로 이동. Accessed bit를 주기적으로 리셋함

메모리가 부족하면 kswapd가 inactive_list의 front부터 교체

Windows

페이지 폴트 앞 뒤 3 또는
7개 페이지도 불러들임

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum
- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

Working set maximum을 사용하는 프로세스에서 페이지
폴트가 발생하면 LRU 기반 local 페이지 교체 알고리즘을 사용.
전체 메모리가 임계치 아래로 내려가면 automatic working
set trimming이라는 global 페이지 교체 알고리즘을 사용

End of Chapter 10