

Chapter 9: Virtual Memory



Chapter 9: Virtual Memory

- ❑ Background
- ❑ Demand Paging
- ❑ Copy-on-Write
- ❑ Page Replacement
- ❑ Allocation of Frames
- ❑ Thrashing
- ❑ Memory-Mapped Files
- ❑ Allocating Kernel Memory
- ❑ Other Considerations
- ❑ Operating-System Examples



가상 메모리 개념

□ Virtual memory

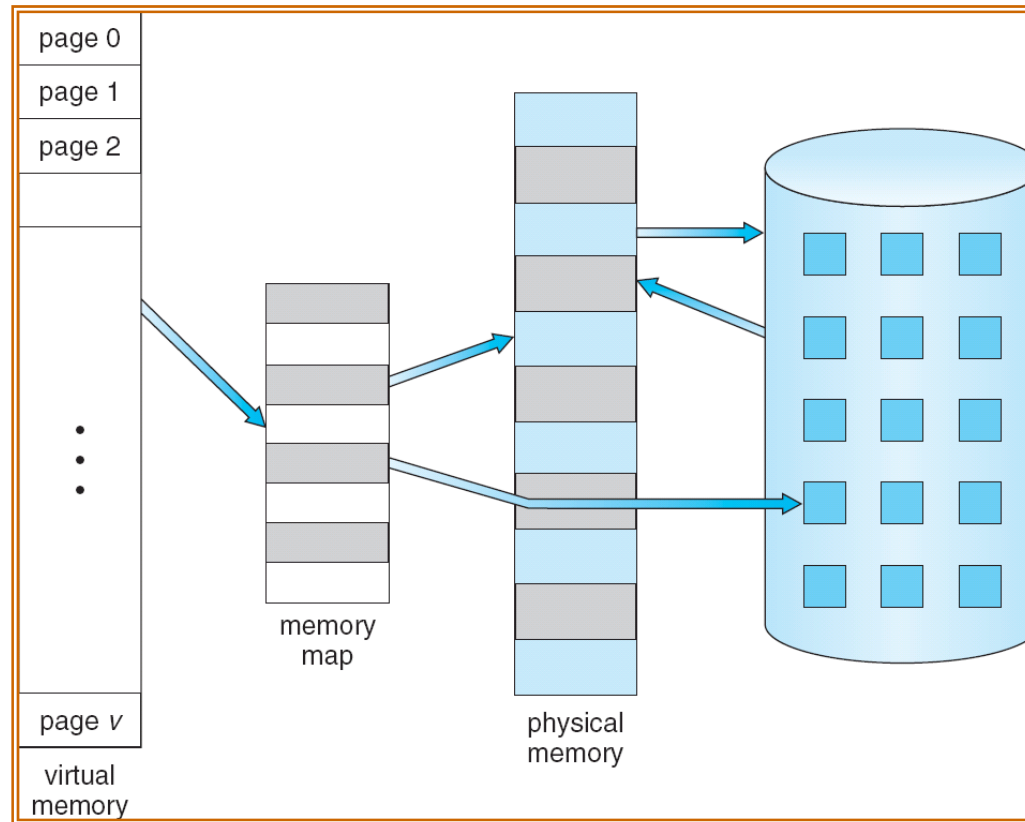
- 프로그램의 block들(페이지 또는 세그먼트)을 디스크에 저장하고 있다가, 실행도중 수시로 이들을 block 별로 메모리로 적재하거나 디스크로 교체하는 기법

□ 가상기억장치 관리 정책(페이징 정책)

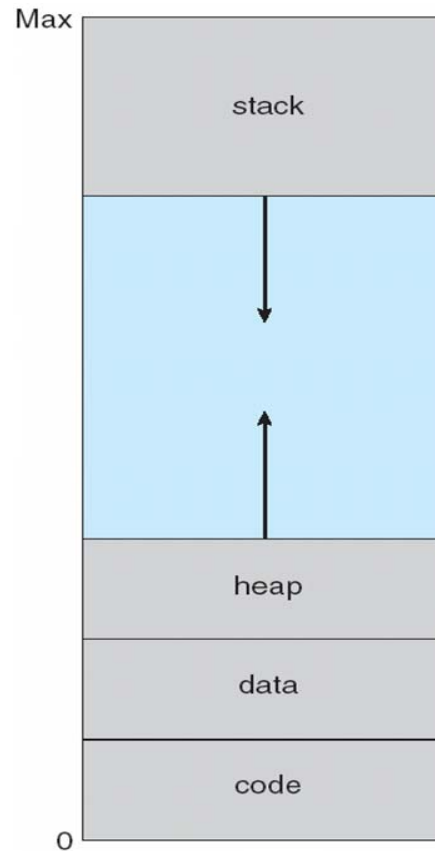
- 페이징 알고리즘에서 결정해야 하는 정책
 - 1) 페이지 반입정책(FETCH) : Demand Paging,...
 - 2) 페이지 배치 정책(PLACEMENT)
 - 3) 페이지 교체 정책(REPLACEMENT) : LRU,...
- 메모리는 임의 접근 장치이므로 PLACEMENT는 문제가 되지 않고 나머지만 성능에 영향을 줌



Virtual Memory That is Larger Than Physical Memory



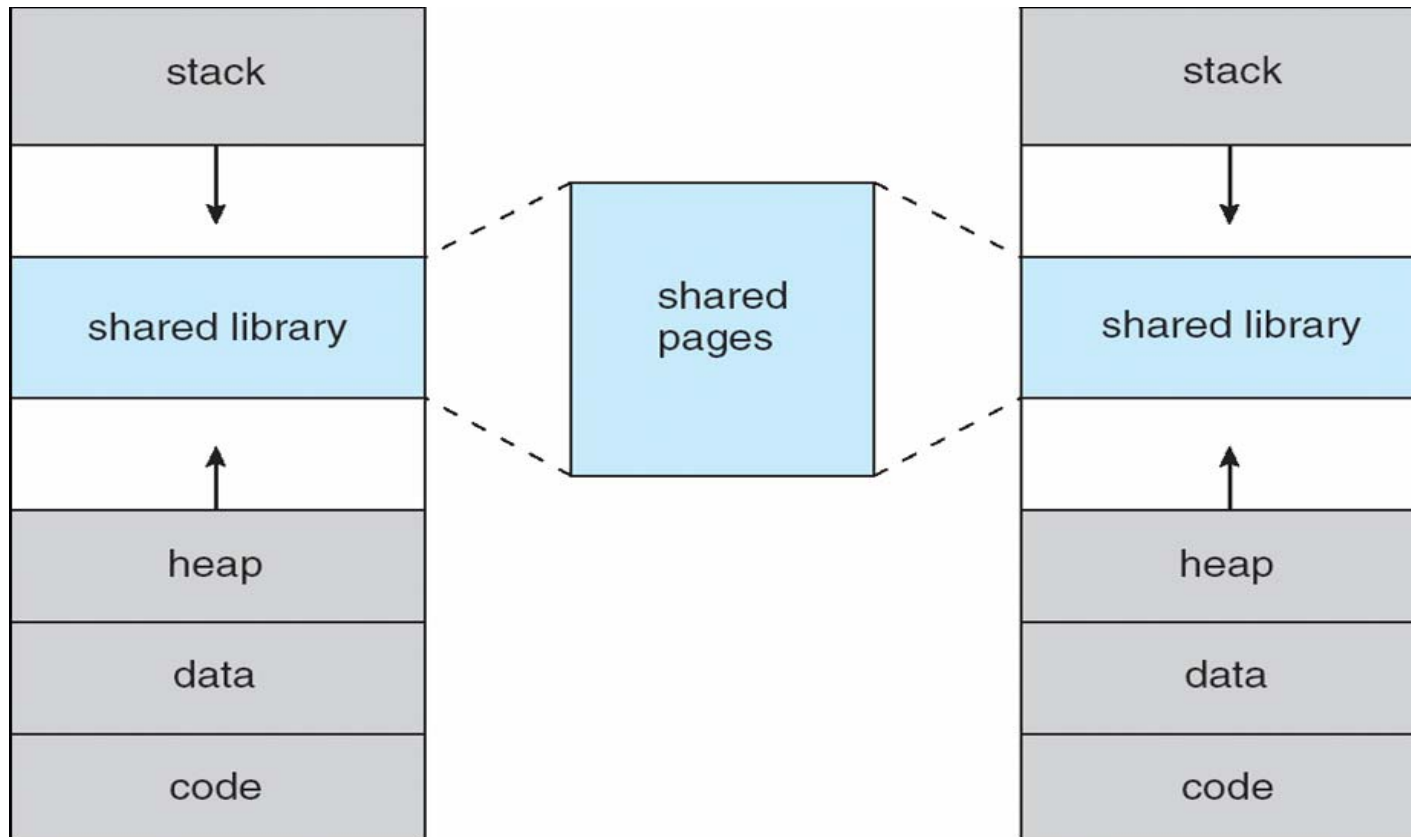
Virtual-address Space



Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc



Shared Library Using Virtual Memory



요구 페이징(Demand Paging)

□ 요구 페이징의 정의

- 프로세스가 실행하면서 실제로 필요로 될때만 메모리로 페이지를 가져오는 반입(fetch) 정책

- **Lazy swapper** – never swaps a page into memory unless page will be needed

□ Page is needed \Rightarrow reference to it

- invalid reference \Rightarrow abort

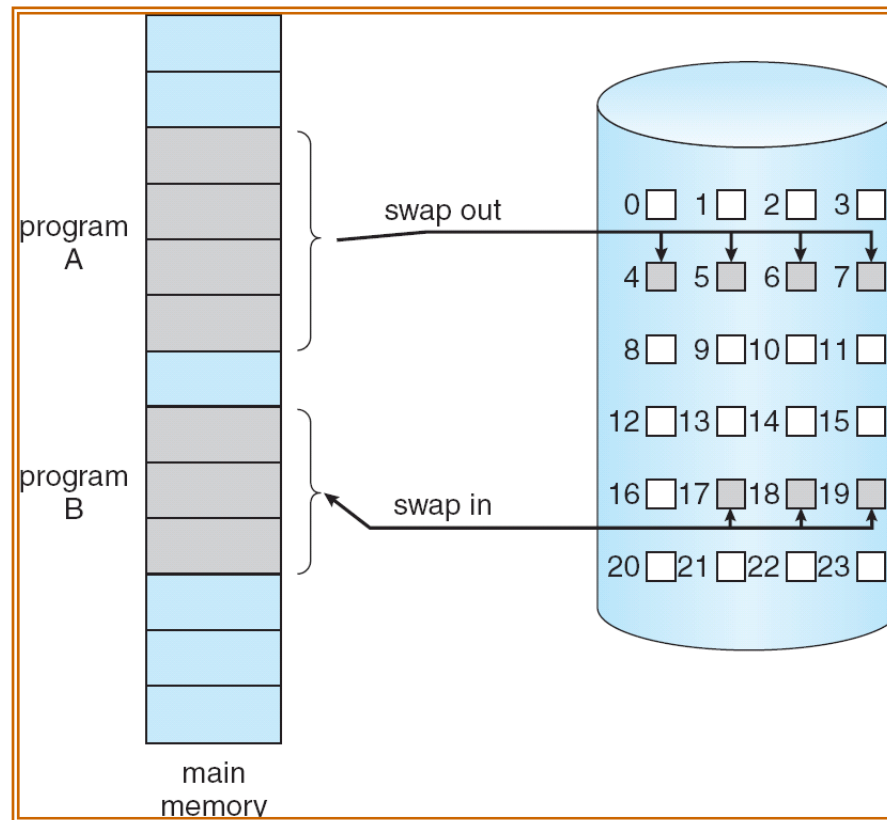
- not-in-memory \Rightarrow **Page Fault** \Rightarrow bring to memory

- 프로그램이 페이지를 요청할때 그 페이지가 메모리에 미처 적재 되지 못한 경우 **Page Fault** 발생

- Prefetch 를 통해 해결 \Rightarrow 선반입된 페이지가 사용되지 않는 상황 발생



Transfer of a Paged Memory to Contiguous Disk Space



요구 페이징 : Valid-Invalid Bit

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

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
....	
	i
	i

page table

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

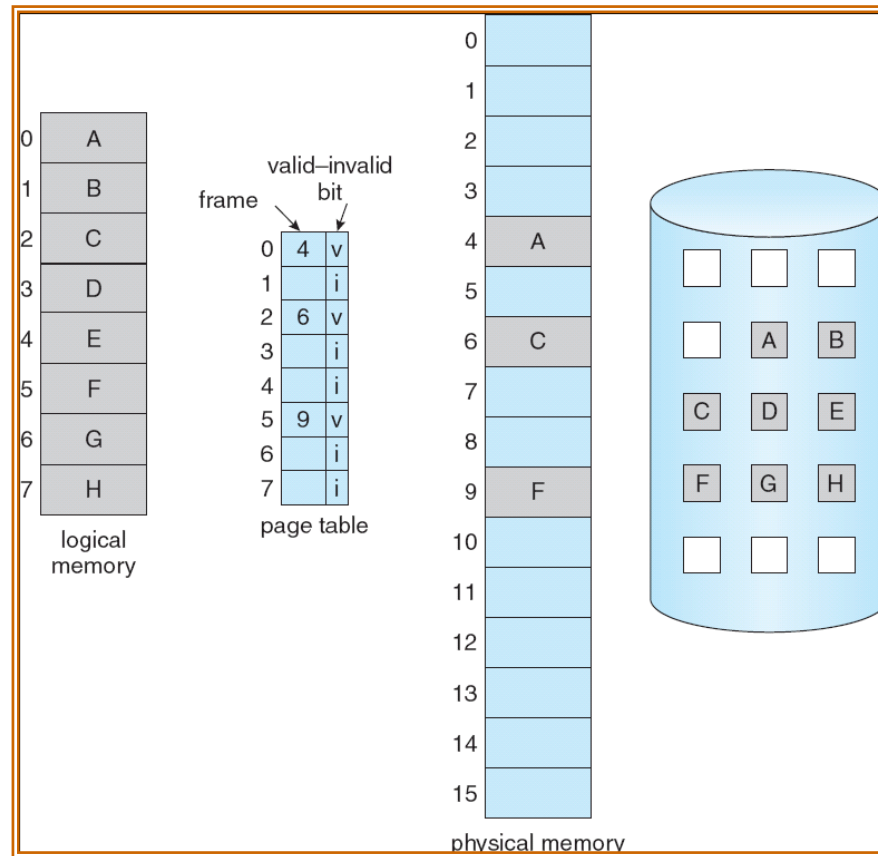


Aspects of Demand Paging

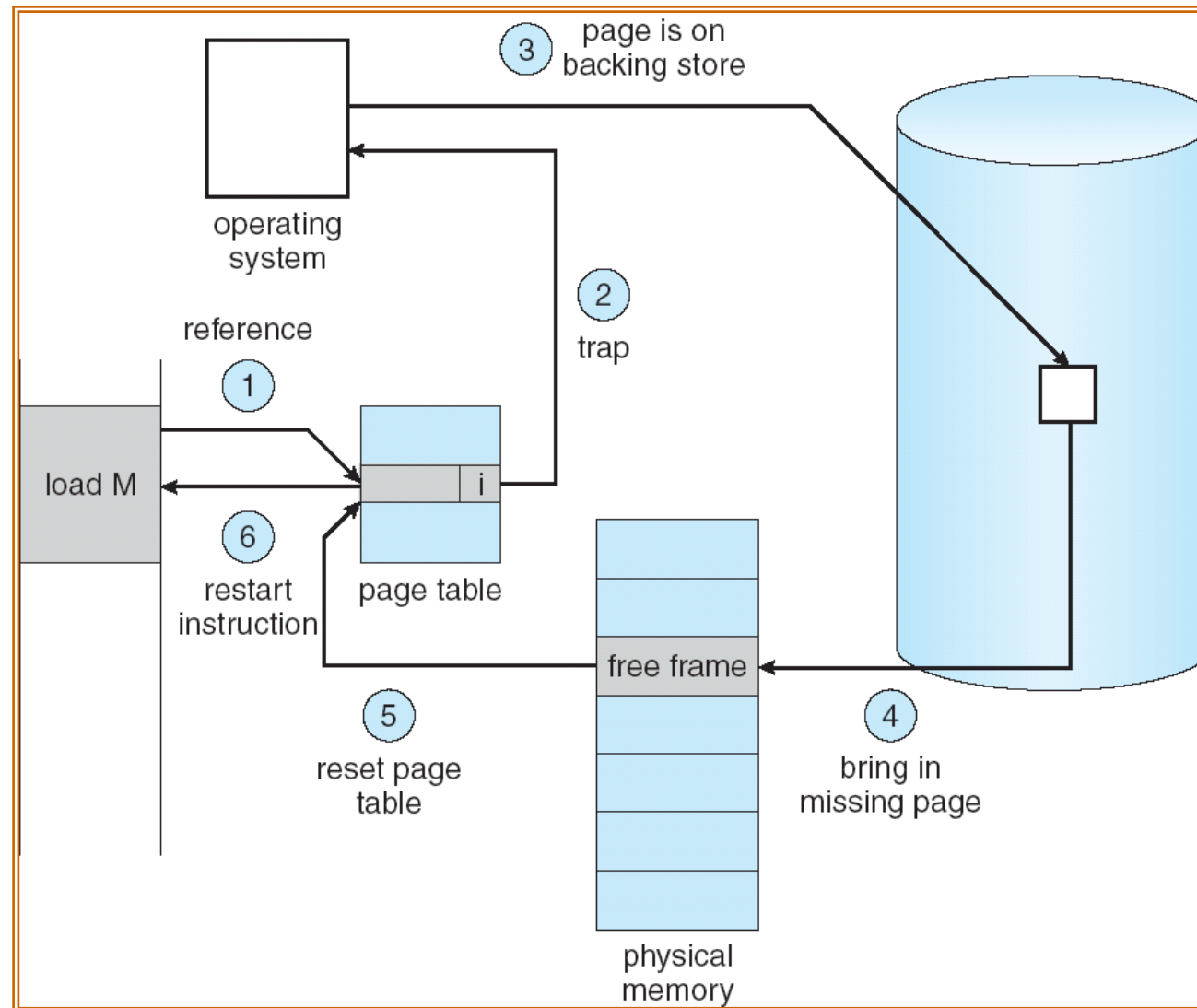
- ❑ Extreme case – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging**
- ❑ Actually, a given instruction could access multiple pages -> multiple page faults
 - 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**



Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault



Steps in Handling a Page Fault

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.



Demand Paging의 성능

- Page Fault Rate $0 \leq p \leq 1.0$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

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

프로그램은 페이지 부재가 발생할 때마다
디스크에서 그 페이지가 올라올때까지 기다려야 하므로
페이지 부재율(Page Fault)를 낮추지 않으면
프로그램의 수행이 매우 느려짐



Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p \times (8 \text{ milliseconds})$
 $= (1 - p) \times 200 + p \times 8,000,000$
 $= 1 \times 200 - p \times 200 + p \times 8,000,000$
 $= 200 + p \times 7,999,800$
- If one access out of 1,000 causes a page fault(0.0001), then
EAT = 8.2 microseconds.
This is a slowdown by a factor of 40!!



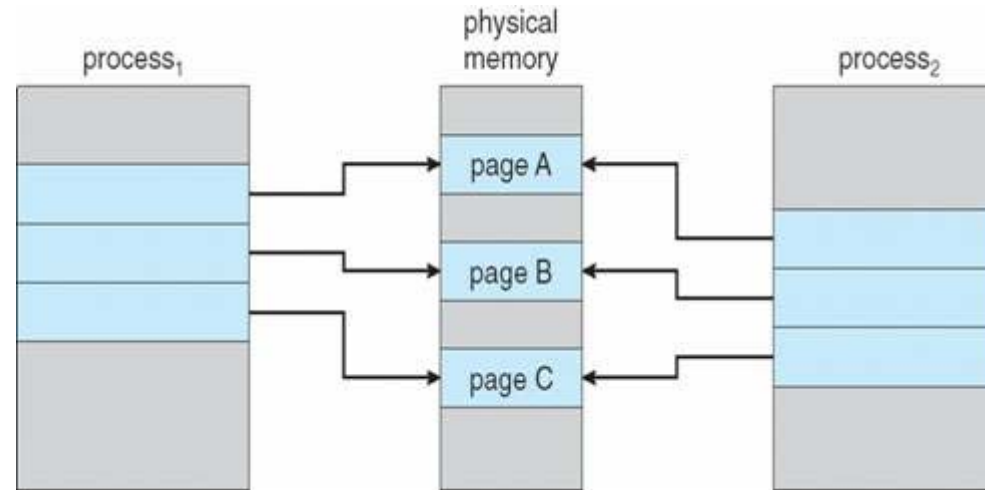
Copy-on-Write

- 일반적인 프로세스
 - 부모 프로세스가 자식 프로세스를 생성할 때 프로세스 메모리를 모두 복제하여 자식을 생성
 - 자식 프로세스 생성시간이 많이 걸림
- Copy-on-Write (COW) :
 - 부모 프로세스와 자식 프로세스가 초기에 동일 페이지를 공유하도록 허용
 - 두 프로세스 중 한 프로세스가 공유페이지를 수정할 때, 그 페이지를 복사하여 생성
- COW allows more efficient process creation as only modified pages are copied

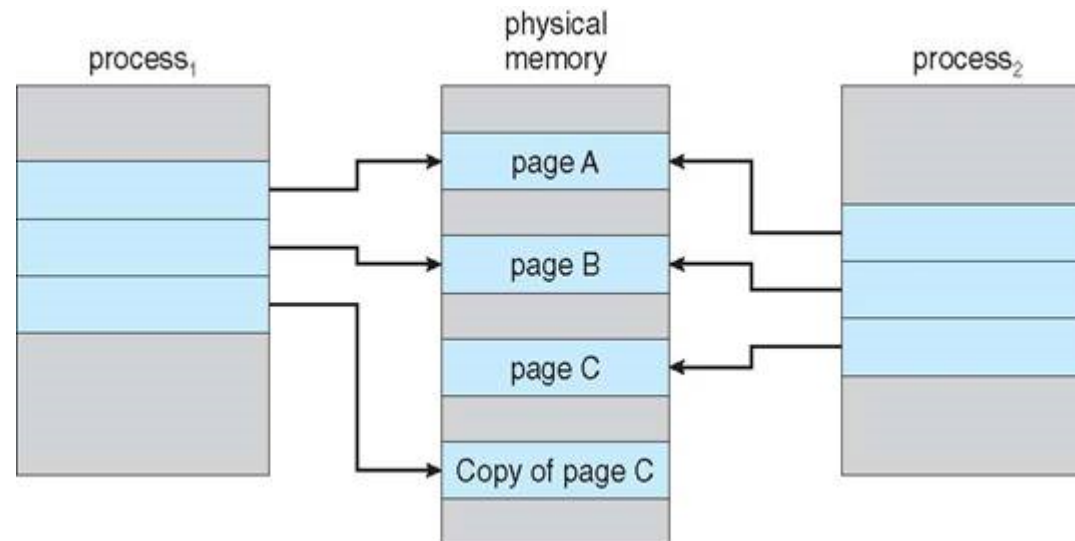


Copy-on-Write

Before Process 1
Modifies Page C



After Process 1
Modifies Page C

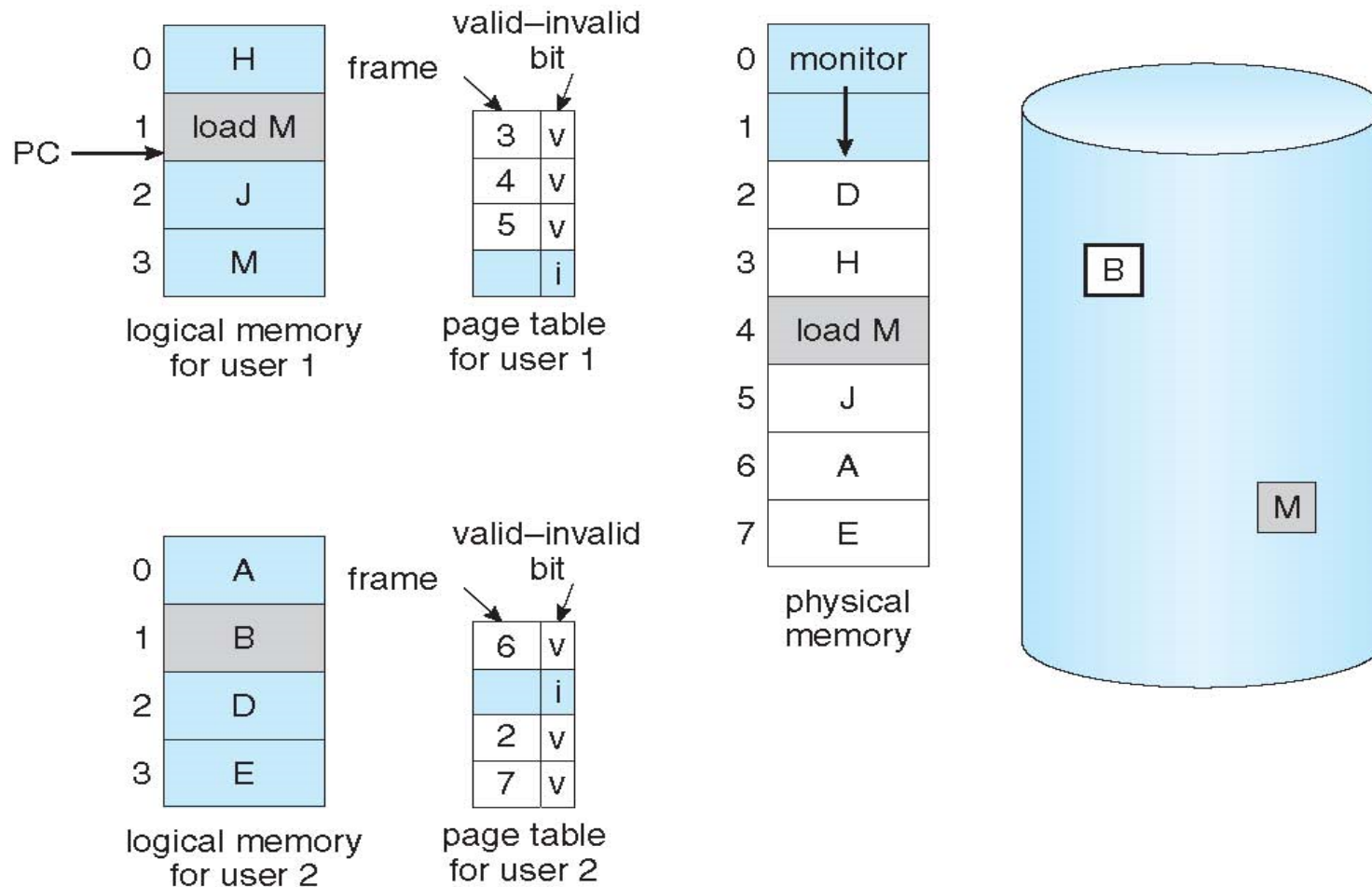


페이지 교체 알고리즘(Page Replacement)

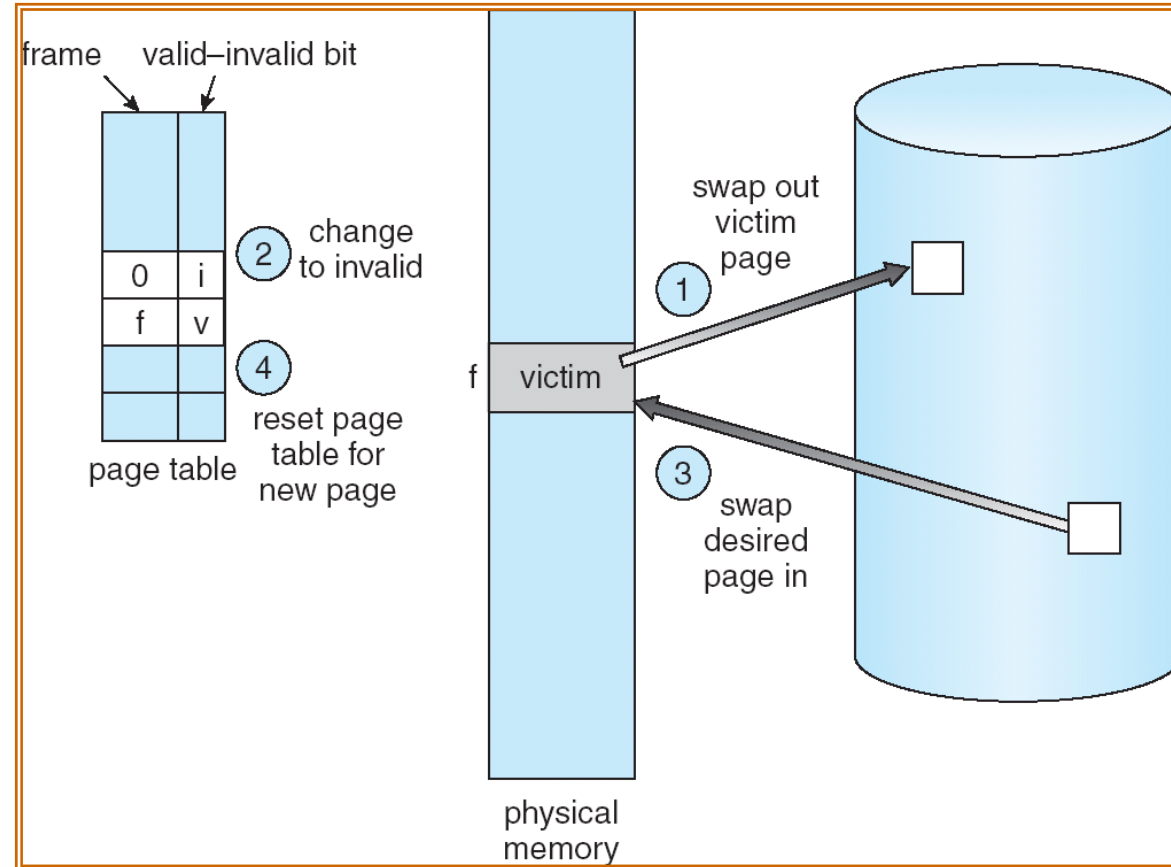
- 페이지 교체 알고리즘(Page Replacement Algorithm)
 - Page Fault가 발생하여 새로운 페이지를 메모리로 적재하고자 할때 빈공간 없는 경우 기존의 페이지중 하나를 디스크로 내리고 새 페이지를 적재하여야 함
 - 이 때, 교체될 페이지를 고르기 위해 사용되는 알고리즘
- 알고리즘의 종류
 - Belady의 최적 알고리즘(OPT 알고리즘)
 - FIFO(First-In First-Out) 알고리즘
 - LRU(Least Recently Used) 알고리즘
 - LFU(Least Frequently Used) 알고리즘
 - NUR(Not Used Recently) 알고리즘



Need For Page Replacement



Page Replacement



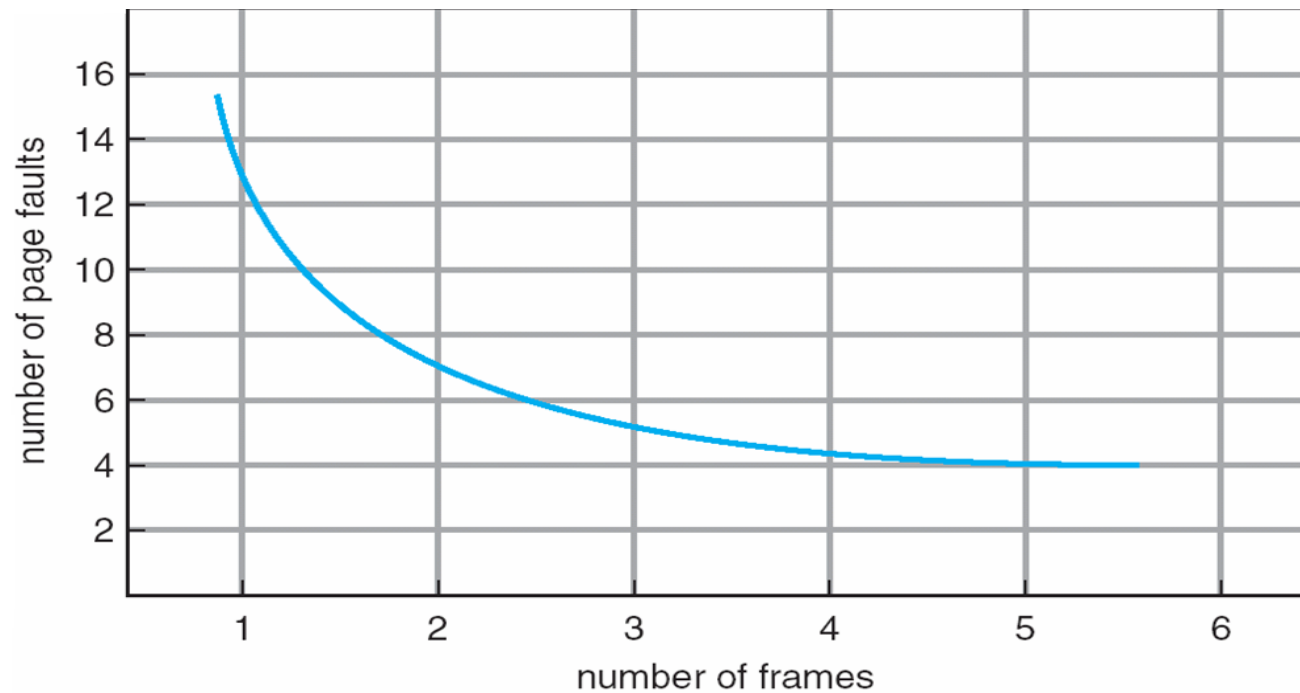
Page and Frame Replacement Algorithms

- ❑ **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
- ❑ In all our examples, the reference string 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



초기 가정



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

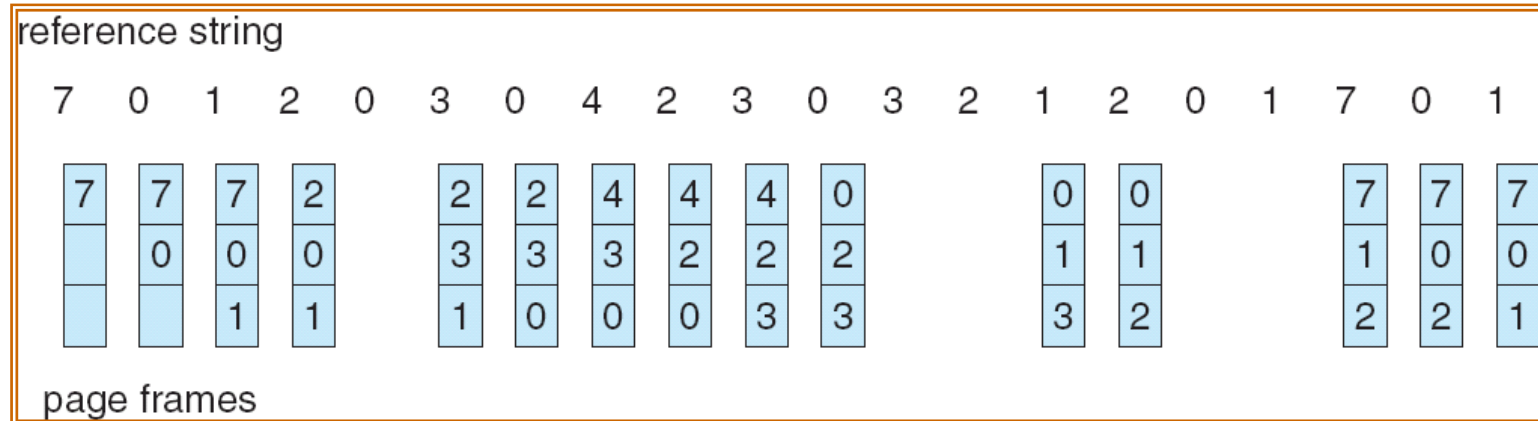
- 3 frames (3 pages can be in memory at a time per process)

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

15 page faults



FIFO Page Replacement



Page Fault : 15



First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

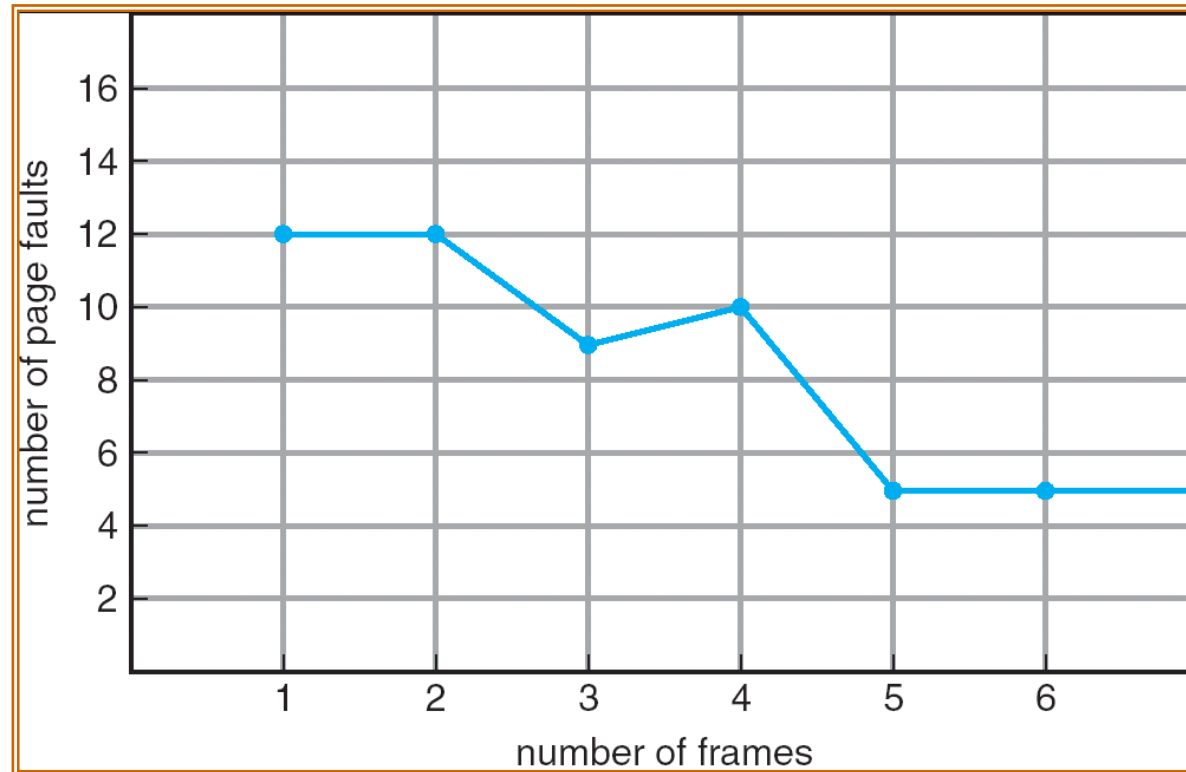
- 4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- Belady's Anomaly: more frames \Rightarrow more page faults



FIFO Illustrating Belady's Anomaly



주기억장치의 할당량을 늘려주었는데도 불구하고,
page fault가 증가하는 현상



Belady의 최적 알고리즘(OPT)

□ 정의

- Belady가 제안한 것으로, 페이지 부재를 최소화하기 위해서 향후 가장 오랫동안 사용되지 않을 페이지를 교체시키는 알고리즘

- 성능이 가장 좋지만 프로세스가 향후 어떤 페이지를 필요로 할지 예측할 수 없기 때문에 구현 불가능
 - 다른 알고리즘의 성능 비교 알고리즘으로 이용



Optimal Algorithm

- ❑ Replace page that will not be used for longest period of time
- ❑ 4 frames example

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

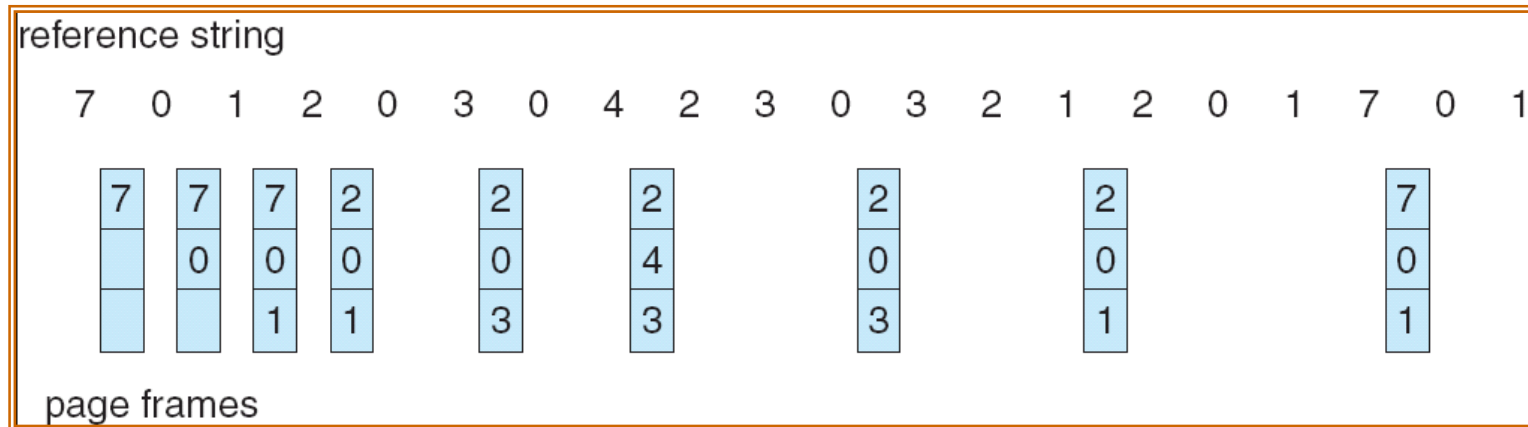
1	4
2	
3	
4	5

**6 page
faults**

- ❑ How do you know this?
- ❑ Used for measuring how well your algorithm performs



Optimal Page Replacement



Page Fault : 9



Least Recently Used (LRU) Algorithm

□ 개념

- 최근에 가장 오랫동안 사용하지 않은 페이지를 교체하는 기법

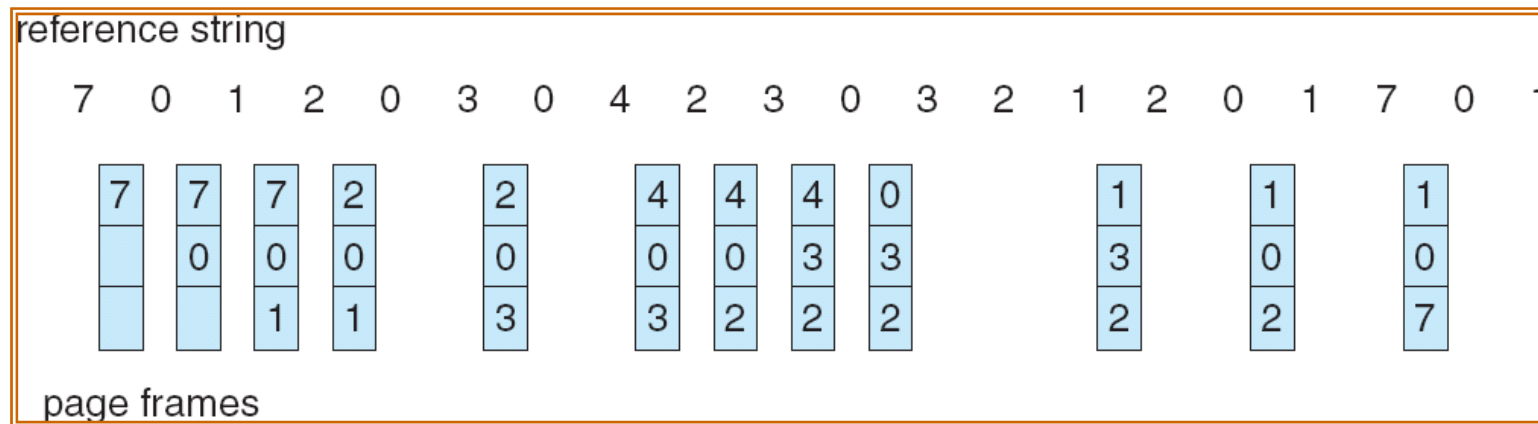
□ LRU 기법의 예

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

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



LRU Page Replacement

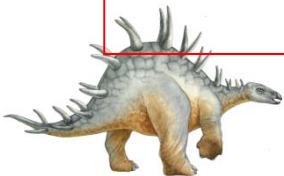


Page Fault : 12

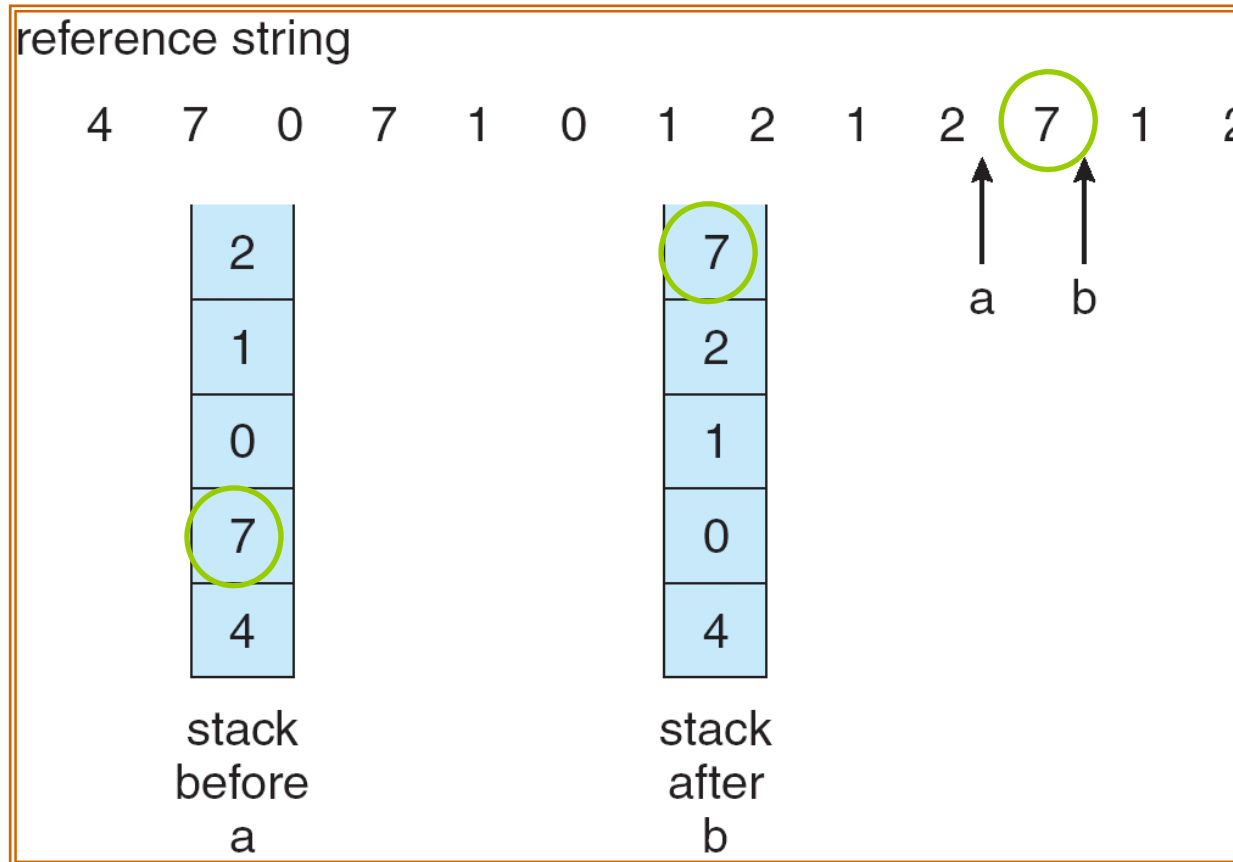


LRU Algorithm (Cont.)

- LRU 알고리즘의 구현방법
 - Counter 이용 구현 방법
 - 가장 최근 페이지 참조시간을 포함한 counter 사용
 - 페이지 교체를 위해 counter를 검색해야함
 - Stack을 이용한 구현
 - keep a stack of page numbers in a **double link** form:
 - Page referenced:
 - move it to the top
 - requires 6 pointers to be changed
 - 교체할 페이지를 찾기 위한 검색이 필요없음



Use Of A Stack to Record The Most Recent Page References



ALRU(Approximating LRU) Algorithms

□ 개념

- LRU 기법이 counter 또는 stack과 같은 높은 비용을 요구하므로 reference bit를 이용한 대략적 구현 방법

□ Reference bit

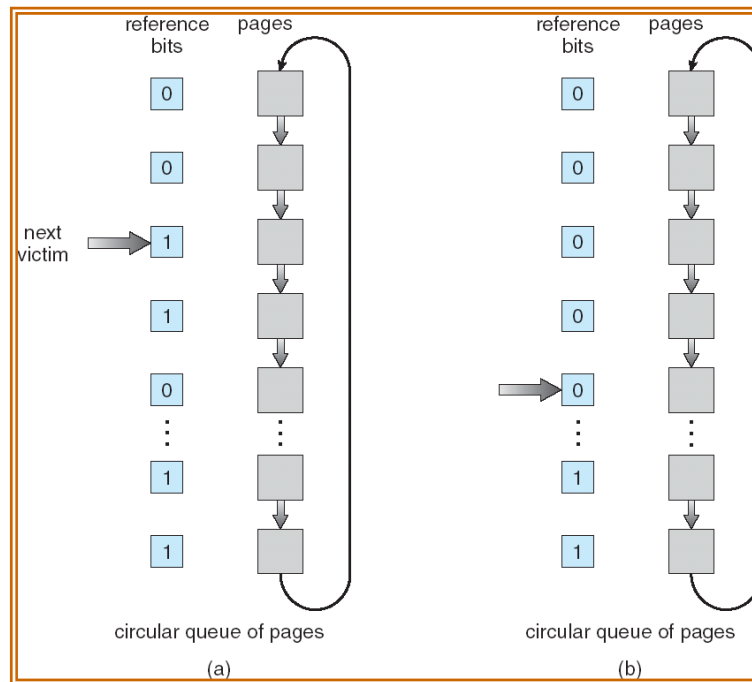
- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace the one which is 0 (if one exists)
 - We do not know the order, however



Second-Chance (clock) Page-Replacement Algorithm

가장 오랫동안 주기억 장치에 있던 페이지 중 자주 사용되는 페이지의 교체를 방지하기 위한 것으로 FIFO 기법의 단점을 보완한 기법

1. 교체 대상이 되기전에 참조 비트를 검사하여 1일 경우 한번더 기회를 부여
2. 각 페이지에 프레임을 FIFO순으로 유지시키면서 LRU 근사 알고리즘처럼 참조 비트를 가짐



기타 Counting 기반 Algorithms

□ LFU(Least Frequently Used) Algorithm

- 사용 빈도가 가장 낮은 페이지를 교체하는 기법

참조 페이지	2	3	1	3	1	2	4	5
페이지 프레임	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3
			1	1	1		1	1
							4	5
부재 발생	○	○	○				○	○

□ MFU Algorithm(Most Frequently Used)

- 사용빈도가 높은 페이지를 교체하는 기법



기타 Algorithms

□ NUR(Not Used Recently)

- LRU와 비슷한 알고리즘으로 최근에 사용하지 않은 페이지를 교체하는 기법
 - 최근의 사용여부를 확인하기 위해 각 페이지 마다 2개의 비트를 사용, 참조비트와 변형 비트를 사용
- 참조 비트와 변형 비트의 값에 따라 순서가 결정되고 페이지 교체
- NUR 교체 순서

참조비트	변형비트	교체순서
0	0	1
0	1	2
1	0	3
1	1	4



페이지 할당 알고리즘

- 페이지 프레임들을 프로세스들에게 분배해주는 방법
 - 균등 할당법(Equal Allocation)
 - 모든 프로세스에게 똑같은 수의 프레임을 배정하는 방법
 - 비례 할당법(Proportional Allocation)
 - 각 프로세스마다 다르게 할당하는 방법
 - Priority Allocation



Global vs. Local Allocation

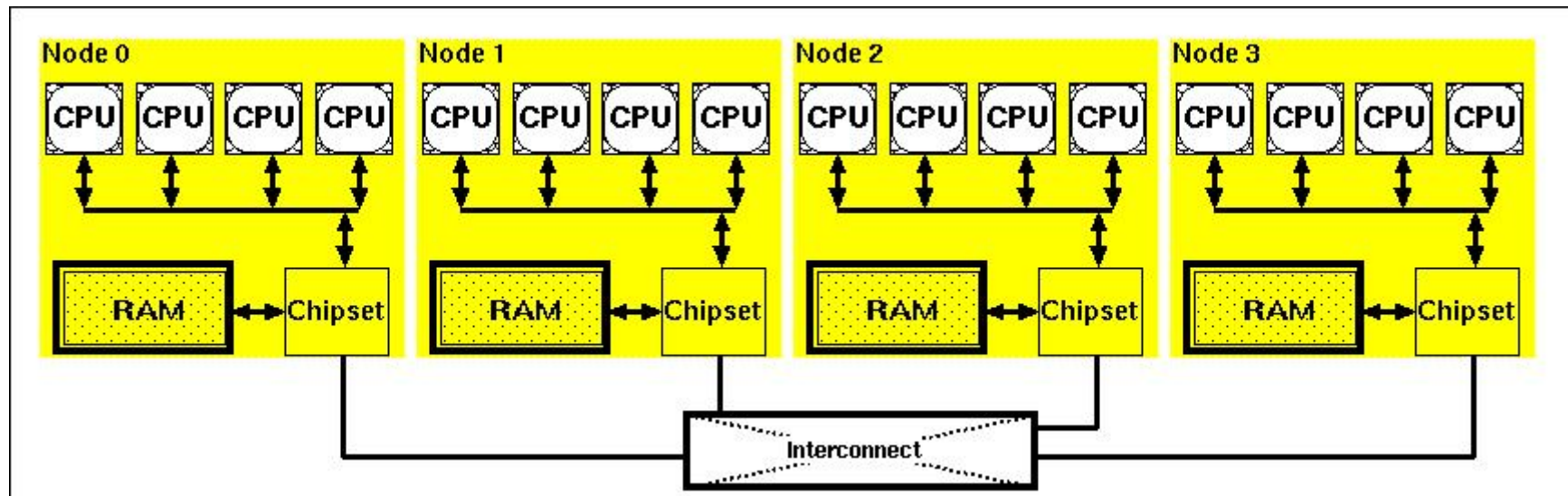
- ❑ **Global replacement** – process selects a replacement frame **from the set of all frames**; one process can take a frame from another
 - But then process execution time can vary greatly
 - But greater throughput so more common

- ❑ **Local replacement** – **each process** selects from only its own set of allocated frames
 - More consistent per-process performance
 - But possibly underutilized memory



Allocation in NUMA

- NUMA = Non-Uniform Memory Access
 - 메모리가 네트워크를 통해 연결되어 있음
 - 최소 지연시간을 갖는 메모리 프레임에 할당



<http://home.arcor.de/efocht/sched/>



스래싱(Thrashing)

□ 스래싱의 정의

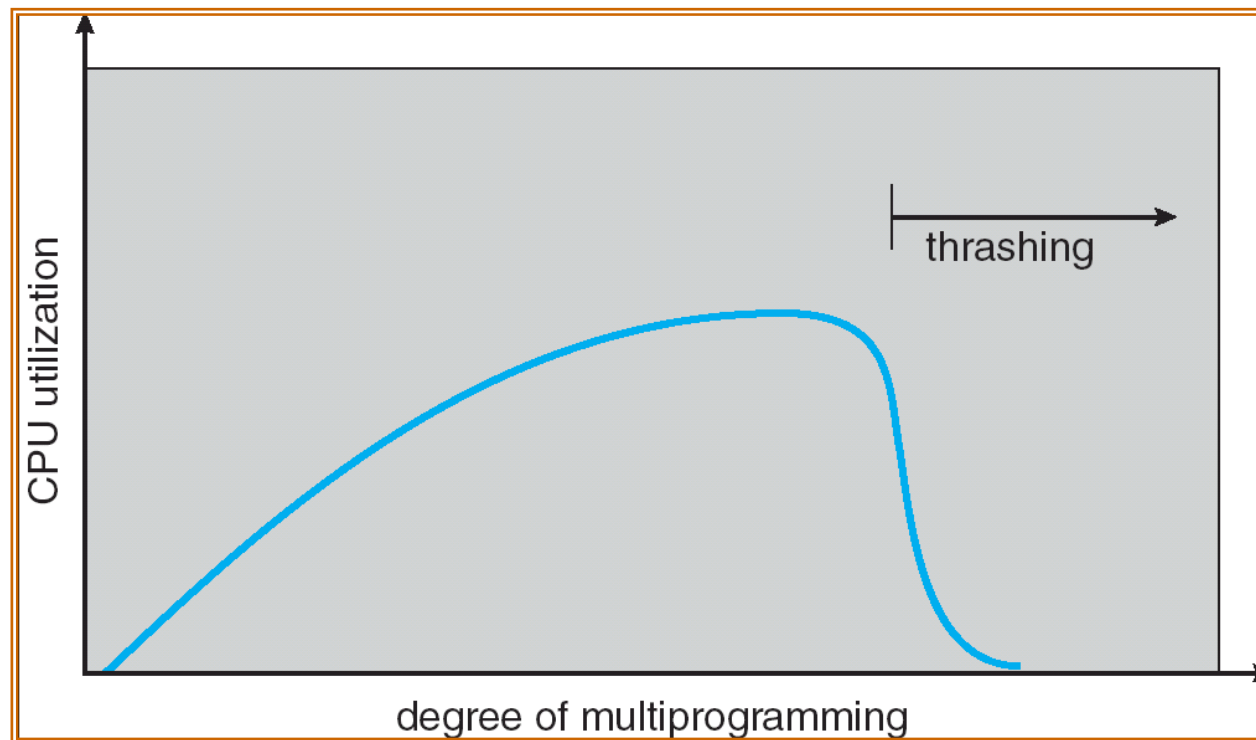
- 한 프로세스가 충분한 페이지 크기를 갖지 않아 너무 자주 **page fault**가 발생하여,
- 프로그램 수행보다 페이지 교환에 더 많은 시간이 소요되는 현상

□ 스래싱의 원인

- 멀티 프로그래밍의 정도가 높아짐에 따라 **CPU** 이용률은 높아지게 되나, 프로세스당 할당된 메모리의 페이지 프레임수가 너무 적게 되어 **Page Fault**가 급격하게 증가되고,
- 이에 따라 **CPU** 이용률이 급격하게 감소되어 역전 현상이 발생



Thrashing (Cont.)



스래싱(Thrashing) 현상 해결 방법

□ 해결방법

- 부족한 자원을 증설
- 일부 프로세스를 중단
 - 낮은 우선순위 프로세스를 중단
- 성능자료의 지속적 관리 및 분석으로 임계치를 예상하여 운영
- 페이지 부재율을 조절하여 대처함
- Working Set 방법을 이용



구역성(Locality) : 메모리 참조에서의 구역성

□ 정의

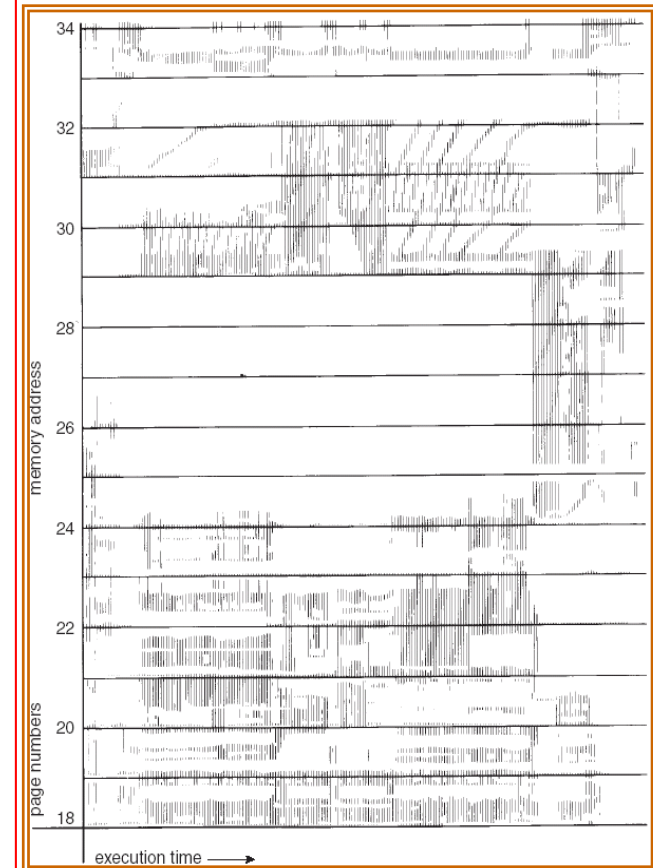
- 실행 중인 프로세스가 일정 시간 동안 메모리의 일정 부분만을 집중적으로 참조하는 경우가 발생
- Working Set 이론의 기반

■ 시간구역성

- 처음에 참조된 기억장소가 가까운 미래에도 계속 참조될 가능성이 높음
 - 반복(Loop), 스택(Stack), Subroutine, Counting, 집계(Totaling) 등

■ 공간 구역성

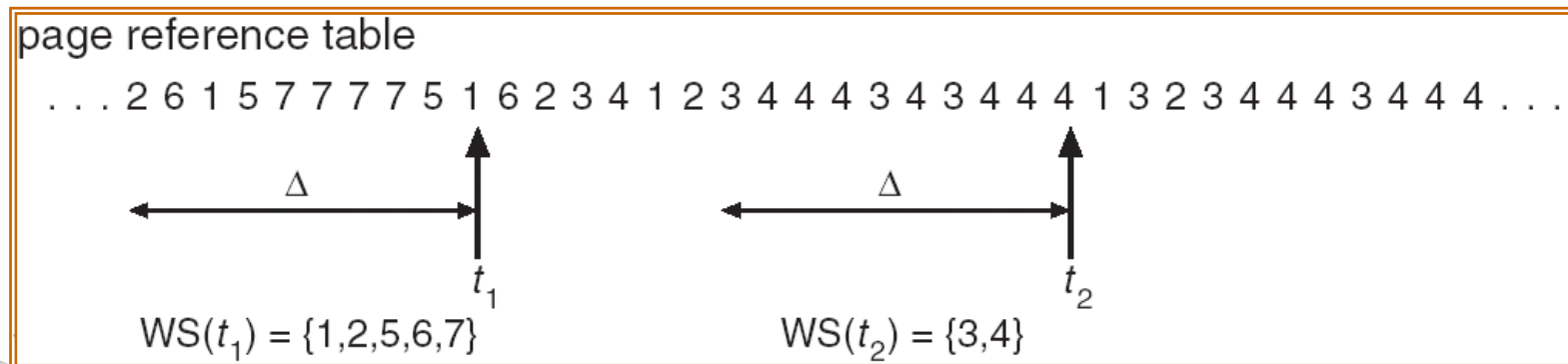
- 어떤 기억장소가 참조되었을 때, 그 근처의 기억장소가 계속 참조될 가능성이 높음
 - 배열 순례, 순차적 코드



Working-Set Model

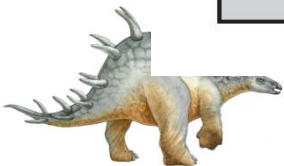
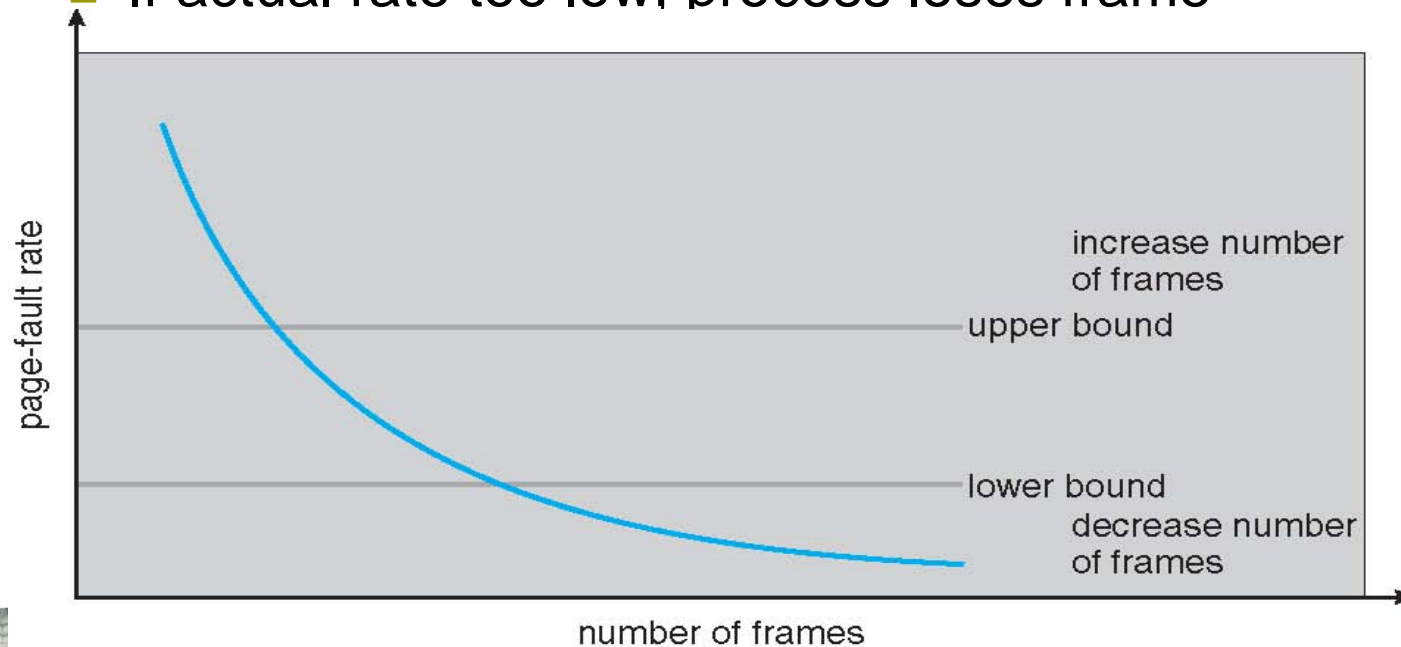
□ Working Set의 정의

- 실행중인 프로세스가 최근 T초 동안에 참조한 페이지들의 집합
- 해당 프로세스에게 할당해 주어야 할 최소한의 페이지 수를 뜻하며,
 - Working Set의 크기보다 적게 페이지 프레임을 할당해 주면 스래싱이 발생할 수 있음

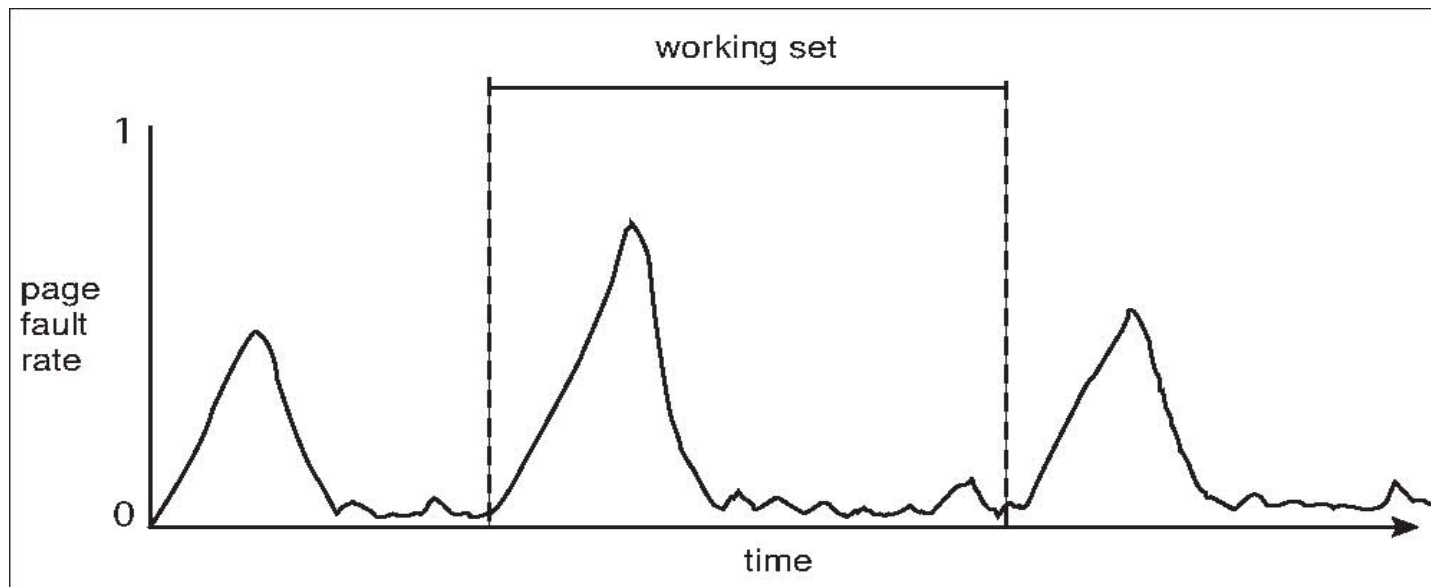


Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** rate and use local replacement policy
 - If actual rate too low, process loses frame

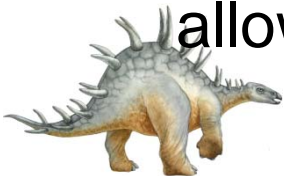


Working Sets and Page Fault Rates



Memory-Mapped Files

- ❑ Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- ❑ A file is initially read using demand paging
 - A page-sized portion of the file is read from the file system into a physical page
 - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- ❑ Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- ❑ Also allows several processes to map the same file allowing the pages in memory to be shared

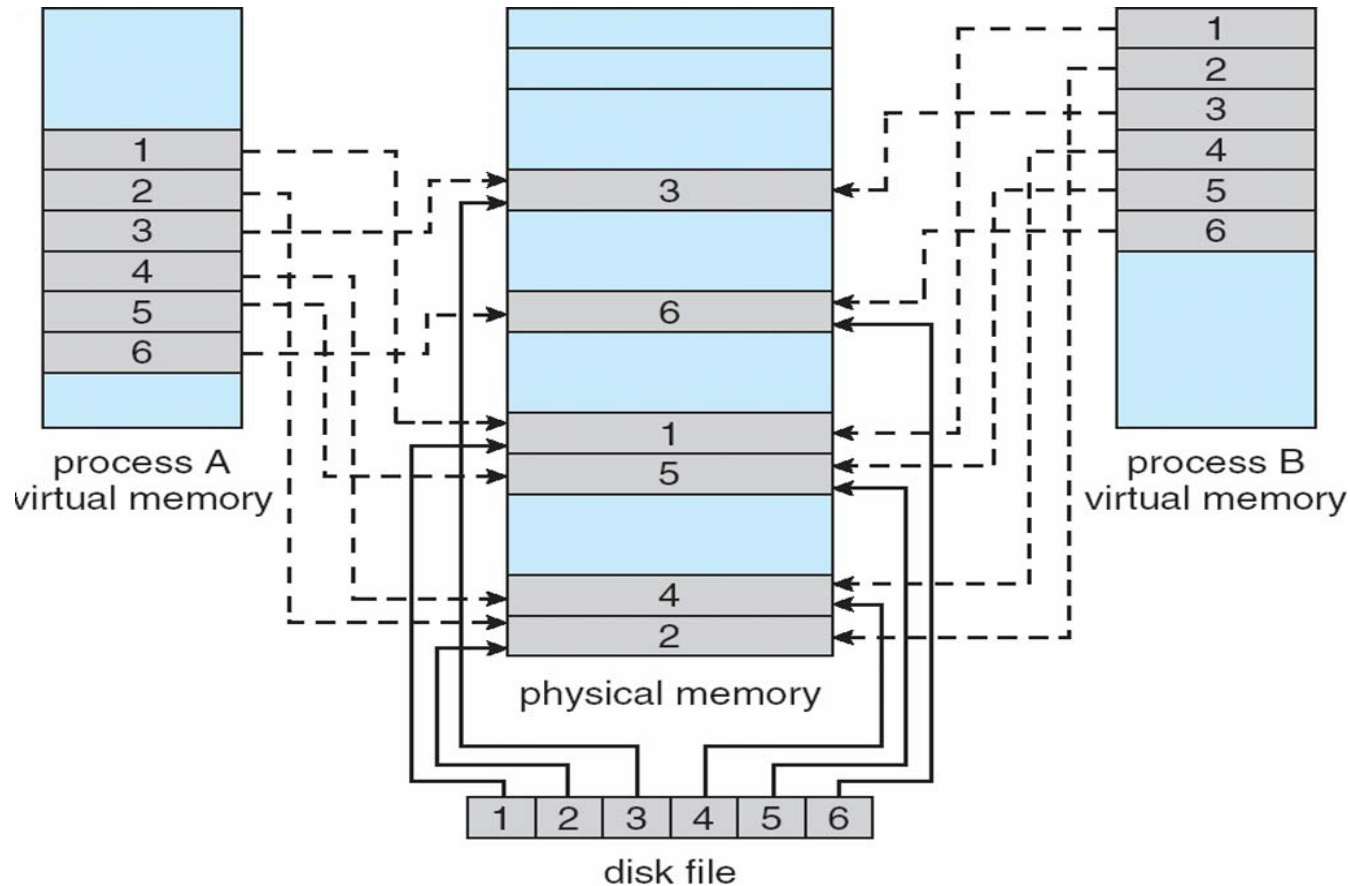


Memory-Mapped File Technique for all I/O

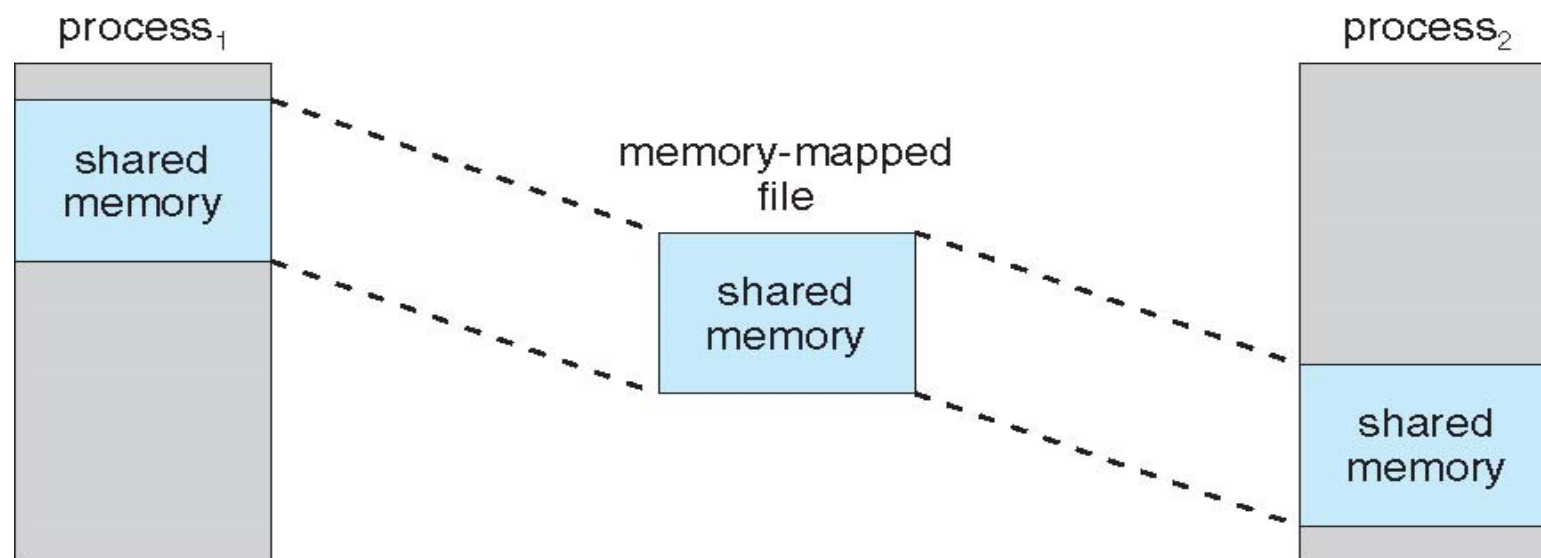
- ❑ Some OSes use memory mapped files for standard I/O
- ❑ Process can explicitly request memory mapping a file via `mmap()` system call
 - Now file mapped into process address space
- ❑ For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
 - But map file into kernel address space
 - Process still does `read()` and `write()`
 - ❑ Copies data to and from kernel space and user space
 - Uses efficient memory management subsystem
 - ❑ Avoids needing separate subsystem
- ❑ COW can be used for read/write non-shared pages
- ❑ Memory mapped files can be used for shared memory (although again via separate system calls)



Memory Mapped Files



Memory-Mapped Shared Memory in Windows

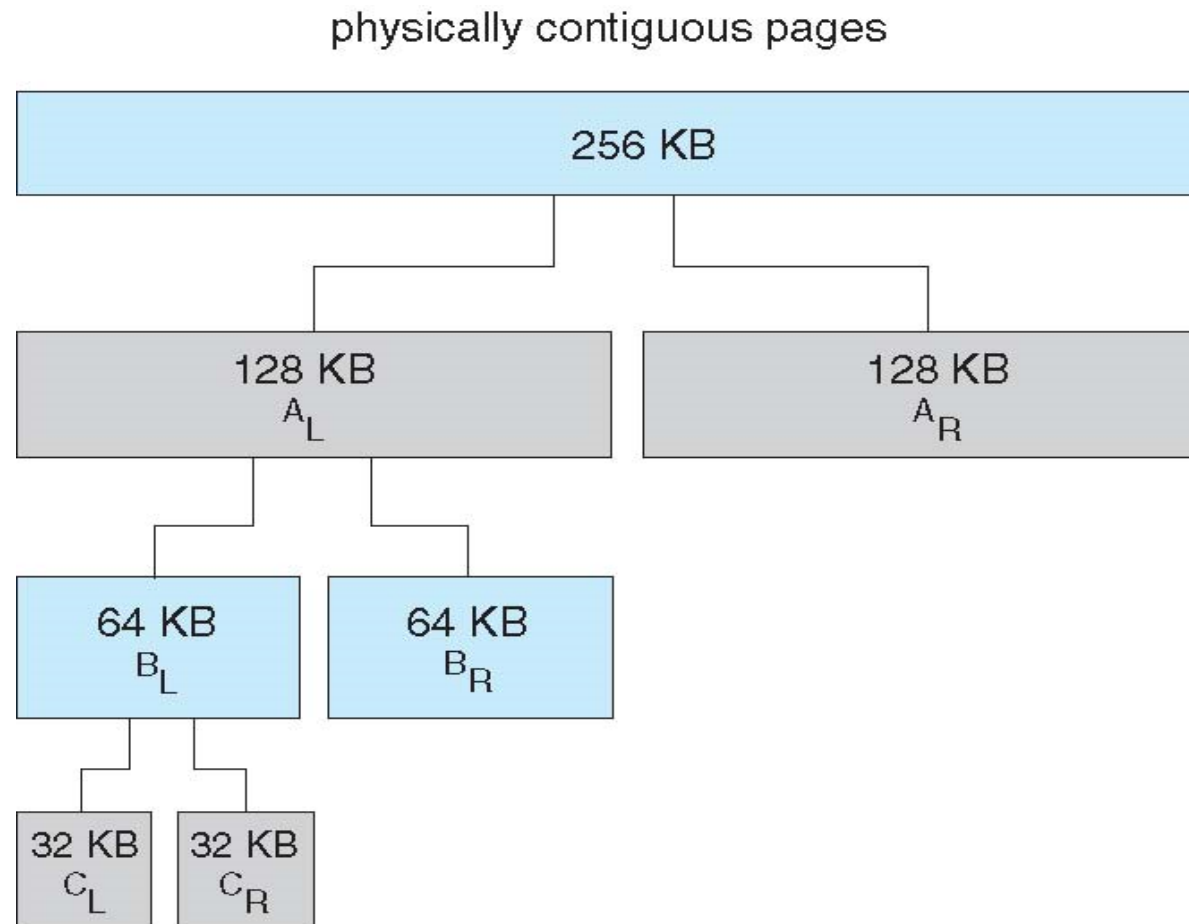


Allocating Kernel Memory

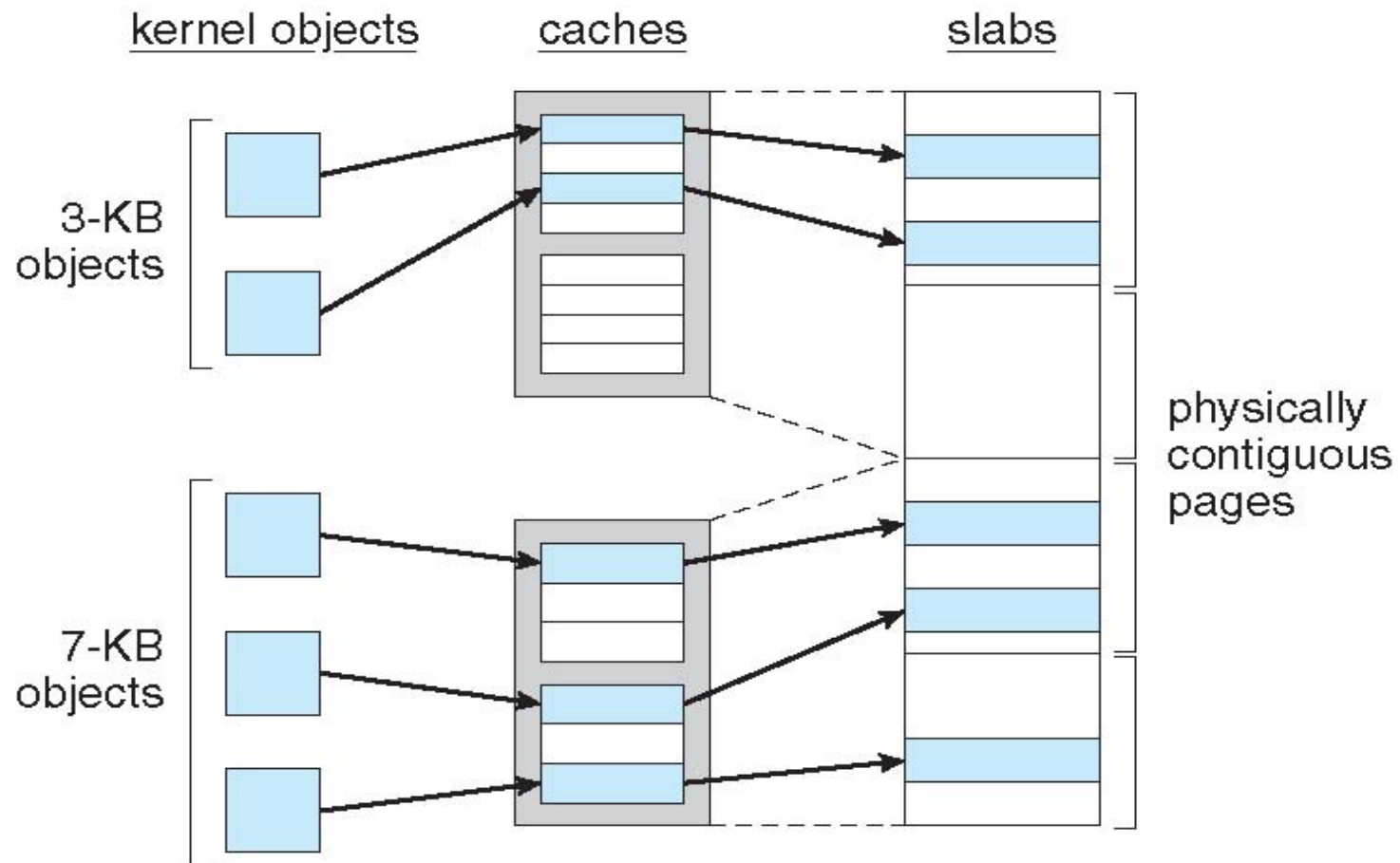
- Treated differently from user memory
- Often allocated from a free-memory pool
 - Kernel requests memory for structures of varying sizes
 - Some kernel memory needs to be contiguous
 - I.e. for device I/O



Buddy System Allocator



Slab Allocation



Other Issues – 페이지크기(Page Size)

- 가상 메모리에서 페이지 크기의 영향
 - 페이지 크기가 작을 경우
 - 페이지 단편화가 감소되고, 한 개의 페이지를 주기억 장치로 이동하는 시간이 줄어듦
 - 프로그램 수행에 필요한 내용만 주기억 장치에 적재될 수 있고, 지역성이 높아져 기억 장치 효율이 높아짐
 - 페이지 정보를 갖는 페이지 맵 테이블의 크기가 커지고 맵핑 속도가 늦어짐
 - 디스크 접근횟수가 많아져서 전체적인 입출력 시간은 늘어남
 - 페이지 개수가 증가



Other Issues – 페이지크기(Page Size)

- 가상 메모리에서 페이지 크기의 영향
 - 페이지 크기가 클 경우
 - 페이지 정보를 갖는 페이지 맵 테이블의 크기가 작아지고, 맵핑 속도가 빨라짐
 - 디스크 접근 횟수가 줄어들어 전체적인 입출력 효율성이 증가함
 - 페이지 단편화가 증가되고 한 개의 페이지를 주기억 장치로 이동하는 시간이 늘어남
 - 프로그램 수행에 불필요한 내용까지 주기억장치에 적재될수 있음



End of Chapter 9

