

9. 가상 메모리 Virtual Memory

1. 가상 메모리 개요

프로그램을 일부만 적재하고도 실행이 가능케 하는 기법:

Paging + 부분 적재 + 요구될 때 적재.

Demand Paging과 Page Fault의 처리

- 2. Demand Paging (요구 페이징)
- 3. Page Replacement (페이지 교체)

Process에게 적절한 Frame 개수는?

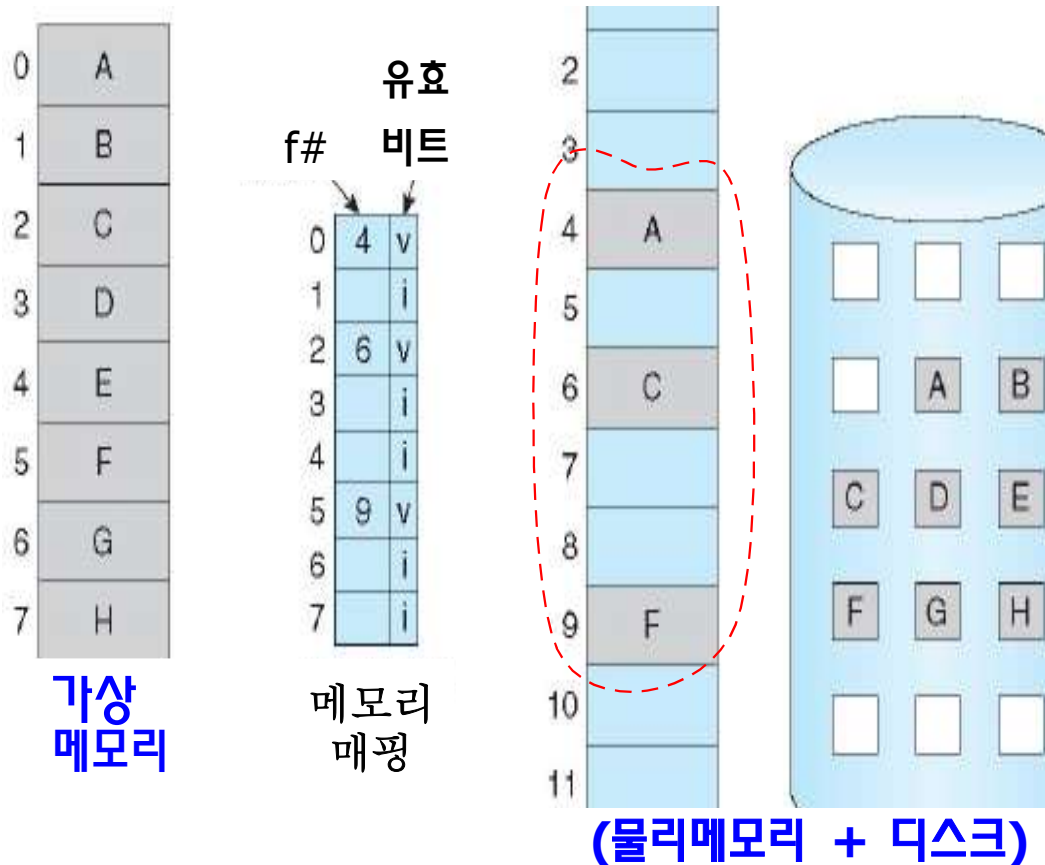
- 4. Frame 할당
- 5. Thrashing 완패, 허우적거림

And...

1. 가상 메모리

프로그램을 일부만 메모리에 적재하고도 실행이 가능하게 하는 기법

= **Paging** + 프로그램 부분 적재 + **Demand Paging**(실행도중 요구되는 page 적재)



Program은 물리메모리 크기에 제약 받지 않으므로 아주 큰 **가상주소공간 (Virtual Memory)**을 가지게 됨.
(논리메모리를 물리메모리로부터 분리)

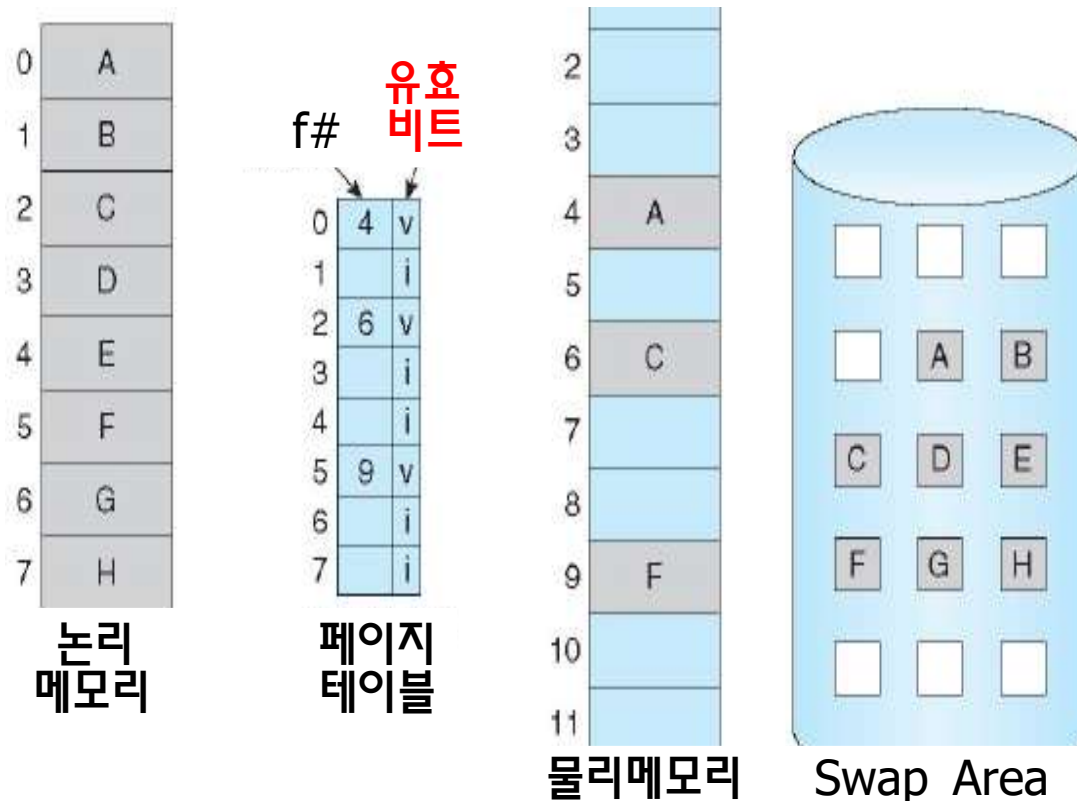
장점:

- 프로그램 일부 적재
⇒ Multiprogramming 정도 ↑
⇒ CPU 이용률, 처리량 *Throughput* ↑
- 스와핑 시간 ↓
⇒ 더 빠른 실행 (*Turnaround Time* ↓)
- Page 공유 가능
⇒ Process 생성 시간 단축.

2. 요구 페이징 Demand Paging :

page가 **실제 필요(요구)**할 때 메모리에 적재함.

※ **Pure Demand Paging** (순수 요구 페이징) OS는 단지 프로세스의 1st Instruction의 주소만 설정하고 (어떤 page도 적재하지 않고) 프로세스를 시작시킴.



- **유효비트**) Page 적재 여부 표시함:
Valid, Invalid.

※ 적재 되지 않은 페이지 참조를
Page Fault(페이지 부재)라 함.

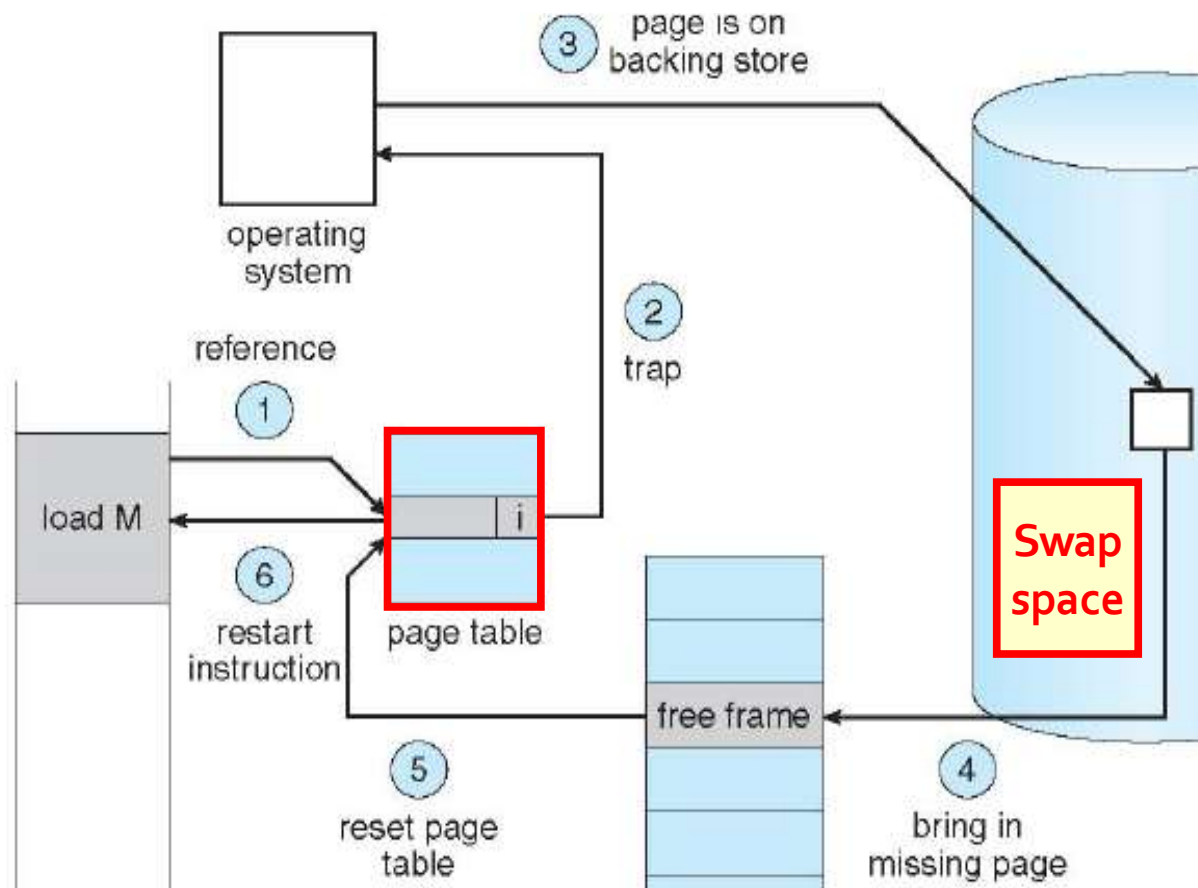
paging H/W가 유효비트를 검사.
invalid인 경우 **page fault trap** 발생.

※ 프로그램은 한 동안 특정 부분이
집중적으로 참조(참조 지역성)되는
특징을 가지고 있다.

⇒ 요구 페이징 성능이 상당히 좋다.

□ Page Fault(페이지 부재) 처리 과정(순서)

- ① 페이지 참조
- ② **page fault** 발생 (**Paging H/W**)
- ③-⑤ **page** 적재, 페이지테이블 갱신 (**Pager**)
- ⑥ instruction 재시작



지원 하드웨어:

- Page table with **Validity Bit**
- **Paging Hardware** (or MMU)
- **Swap Space**
물리메모리 확장 공간으로 사용되는 디스크 공간:
 - A huge file in file system
 - or Raw disk partition

(note) **페이지 교체**
(**Page Replacement**)

□ 요구 페이징의 처리 단계와 성능

요구 페이징 처리 단계	1. trap to OS 2. process context 보존 3. 인터럽트 원인이 page fault임을 확인 4. 디스크 내 페이지 위치 파악
	5. disk에서 free frame으로 읽기 요청: a. 디스크 대기 큐에서 기다림 b. Seek Time (5ms), Latency Time (3ms) c. 페이지를 frame으로 transfer (보통 0.05ms) (6. 대기하는 동안 CPU를 다른 프로세스에게 할당) 7. read 완료 인터럽트 수신 8. 다른 프로세스의 context 보존 9. page table 갱신; dispatching 될 때까지 대기 10. process context 환원; 실행 재계

유효 접근 시간 *Effective Access Time*

$$= (1 - \text{페이지부재율}) \times \text{메모리접근시간} + \text{페이지부재율} \times \text{페이지부재시간}$$

페이지부재시간

$$= \text{interrupt service 시간} + \text{page read 시간} + \text{process restart 시간} \\ \cong \text{page read 시간}$$

★ 메모리접근 시간 = 보통 10ns~200ns

★ page read 시간 = 보통 8ms

- $EAT = (1-p) \times 200\text{ns} + p \times 8\text{ms} = 200 + p \times 7,999,800(\text{ns}) \Rightarrow$ 페이지 부재율에 비례
 (예) 페이지 부재율 = 1/1000 이면 $EAT \cong 8.2\text{ms} \Rightarrow$ 약 40배 성능 저하

- 성능저하를 10% 이내로 낮추려면:

$$200 + 7,999,800 \times \text{페이지 부재율} < 220$$

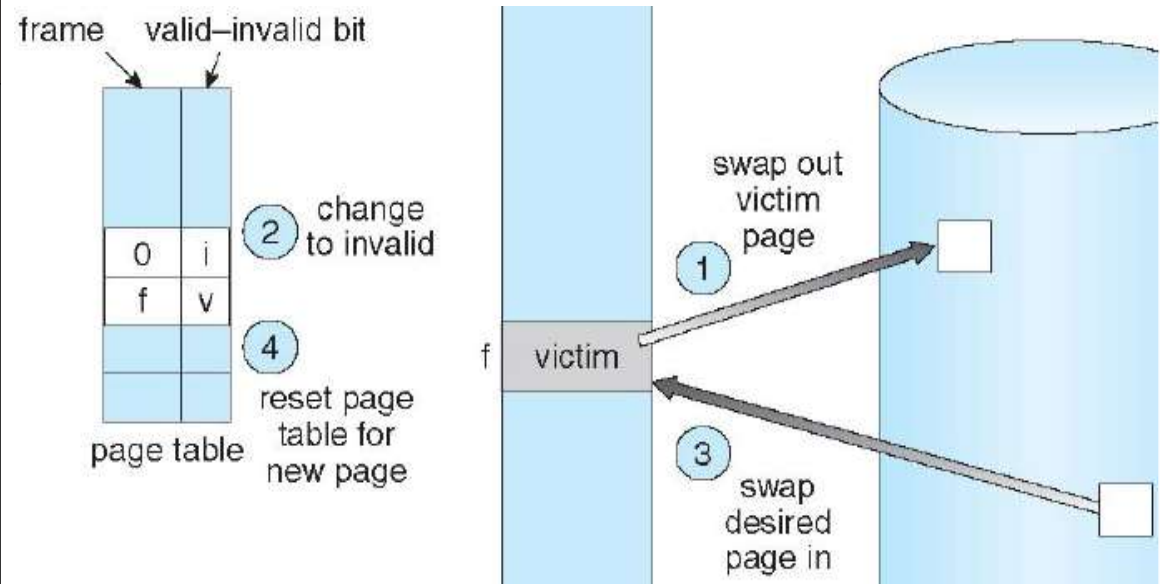
$$\text{페이지부재율} < 0.0000025 \text{ (약 40만 번 중 1번 Page Fault가 일어나는 정도)}$$

3. 페이지 교체 Page Replacement:

Page Fault 時 Free Frame이 없는 경우, 페이지 교체가 필요함.

Page-Fault Service Routine

- ① 요구된 페이지 위치(on disc) 검색;
- ② free frame 검색:
if (존재) {
 free frame을 사용함;
} else {
 희생자 페이지 선택;
 ★ **페이지 교체 알고리즘**
 ① 희생자 페이지를 swap out;
 ② 페이지 및 프레임 테이블 갱신;
}
- ③ 요구된 페이지를 swap in;
④ 페이지 및 프레임 테이블 갱신;
- ⑤ 프로세스 실행 재개



- 페이지 교체 時 page transfer 2번 수행함.
- **Modification or Dirty Bit** (in page table)
 - page 내용 변경 時 “1”로 설정.
 - dirty_bit == 1인 경우에만 swap-out 함.

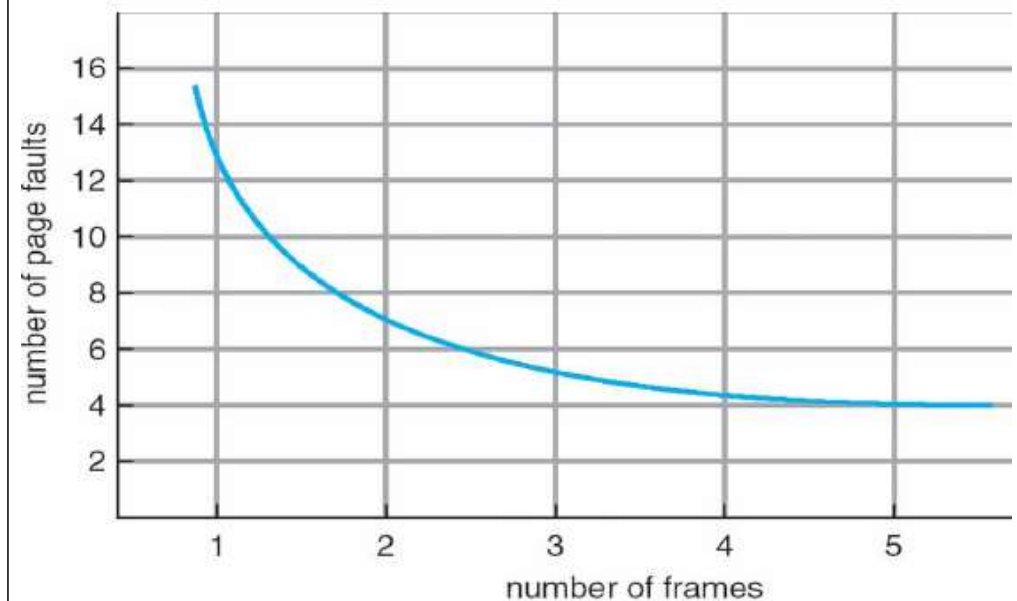
□ Page Fault에 영향을 주는 요소

페이지 교체 알고리즘 Page Replacement Algorithm

- 페이지 부재율 낮은 알고리즘이 좋음.
- 평가: 교체알고리즘 × 메모리 참조열
→ Page Fault 횟수
- 참조열 (*Reference String*)
알고리즘 평가에 사용된
memory reference string
or *page reference string*.

프레임 할당 알고리즘 Frame Allocation Algorithm

- 보통 Frame數 ↑ ⇨ 페이지 부재율 ↓

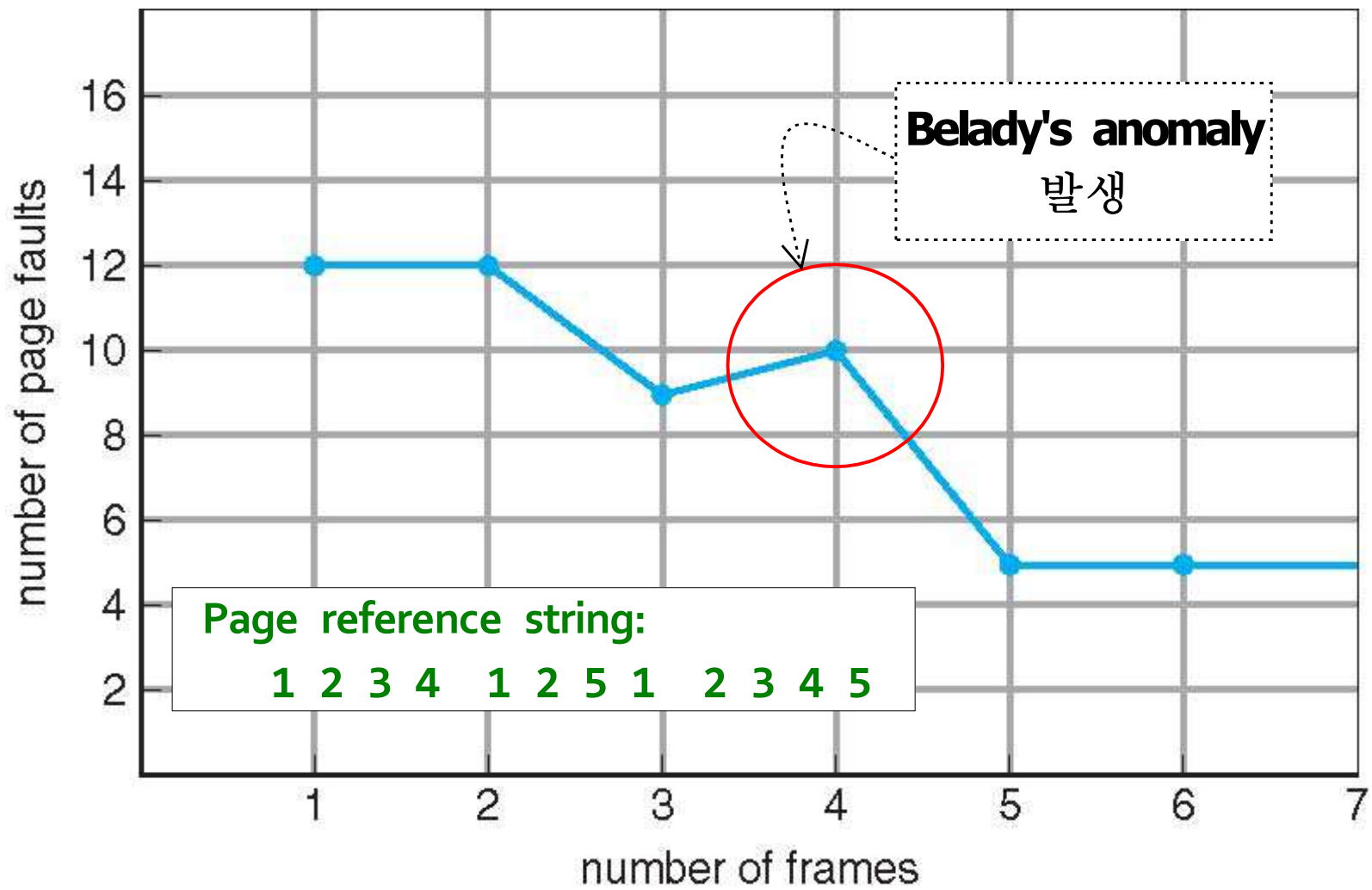


- 프로세스에게 frames을 할당하는 방법에 따라 페이지 부재율이 영향 받음.

페이지 교체 알고리즘	희생자 페이지
FIFO	가장 먼저 적재된 페이지
최적	앞으로 가장 오랫동안 사용되지 <u>않을</u> 페이지
LRU Least Recently Used (가장 덜 최근에 즉, 가장 오래 전에)	가장 오래 전에 사용된 페이지 • 최적 알고리즘의 근사 알고리즘: 가까운 과거를 가까운 미래의 근사치로 사용함.
LRU 근사	추가 참조 비트 알고리즘 2차 기회 알고리즘 개선된 2차 기회 알고리즘

□ FIFO 페이지 교체 FIFO Page Replacement Algorithm

전략	<ul style="list-style-type: none"> Victim Page = 가장 먼저 적재된 페이지 (<i>oldest page</i>) FIFO Queue 사용하여 구현.
예	<p>reference string</p> <p>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</p> <p>page frames</p> <p>Page fault rate: 15/20</p>
비고	<ul style="list-style-type: none"> 이해와 구현이 쉬우나, 성능이 항상 좋지는 않음. 할당된 <u>frame</u> 수가 증가해도 <u>페이지 부재율이 증가하는 경우 발생 가능.</u> (Belady's Anomaly 이상, 변칙)



□ 최적 페이지 교체 Optimal Page Replacement (OPT or MIN) Algorithm

전략	<ul style="list-style-type: none"> Victim Page = 앞으로 가장 오랫동안 사용되지 않을 페이지 (<i>least <u>near</u> used</i>) <i>future</i> reference string에 대한 정보가 필요함.
예	<p>reference string</p> <p>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</p> <p>page fault & no free frames</p> <p>victim</p> <p>page frames</p> <p>Page fault rate: 9/20</p>
비고	<ul style="list-style-type: none"> frame 수가 고정된 경우 가장 낮은 페이지 부재율 보장함 → 최적 Alg. 미래 참조 패턴 필요 ⇨ 구현이 어려움. 알고리즘 비교 평가에 사용됨.

□ LRU 페이지 교체 Least-Recently-Used Page Replacement Algorithm

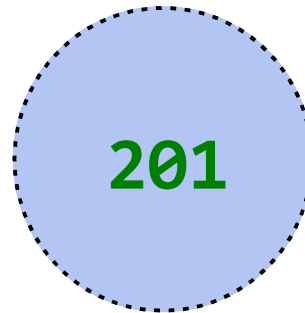
전략	<ul style="list-style-type: none"> Victim Page = 가장 오래 전에 사용된 페이지 (<i>least recently used</i>) (note) Optimal alg.의 근사 alg. : <u>가까운 과거</u>를 <u>가까운 미래</u>의 근사치로 사용함. 각 페이지마다 “마지막 사용 시각”을 유지하는 것이 요구됨.
예	<p>reference string</p> <p>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</p> <p>page frames</p>
비고	<ul style="list-style-type: none"> Page fault rate: OPT(9/20) < LRU(12/20) < FIFO(15/20) 여러 OS에서 사용되며, <u>좋은</u> 알고리즘으로 간주됨. 마지막 사용 시각을 유지하기 위한 H/W 지원이 필요함: <u>Counter</u> or <u>Stack</u>

- **Counter**(in a Page table) = 페이지 참조 시각(논리).

- 페이지 참조時 마다 **logical clock**(**마지막 사용 시각**)으로 설정됨.
- **Victim** = **Counter** 값이 가장 작은(가장 오래 전에 참조된) page

	frame#	계수기
0		50
1		100
2		70
...		
n		200

page table
with **Counter Field**



Logical Clock (in CPU):
메모리 참조時 마다
“1”씩 증가

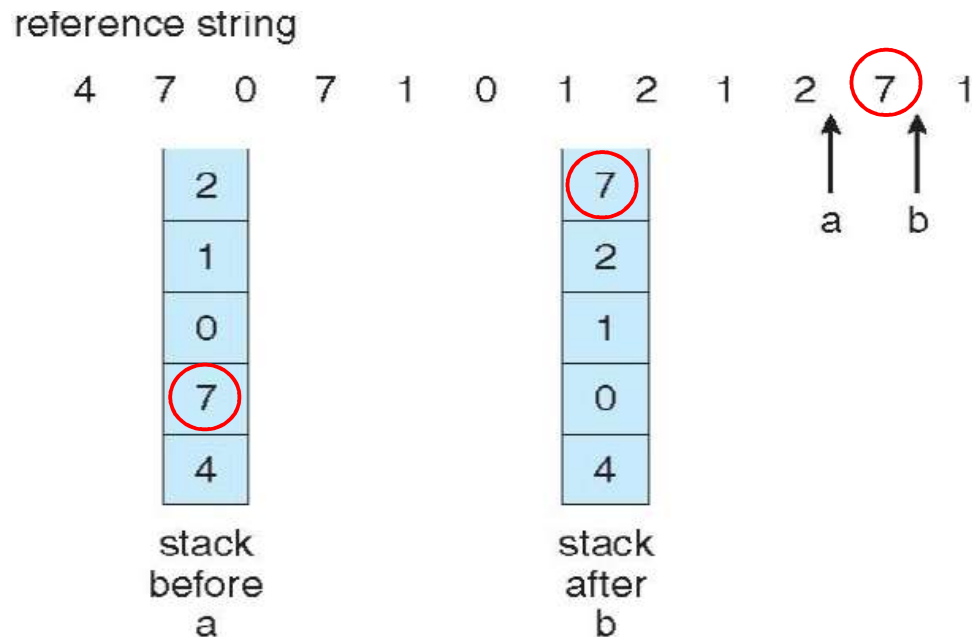
	frame#	계수기
0		50
1		100
2		201
...		
n		200

page2 참조 時

- (note) 메모리 참조 時 마다 counter field 갱신(추가 memory write 1회)
- (note) 희생자 선정 時 page table 검색 필요.

- **Stack**을 사용하는 방법

- 새로운 페이지 참조 시 **top**에 저장함.
- 기존 페이지 참조 시 **top**으로 이동함.
- Victim = **Bottom entry**
(가장 오래전에 참조된 페이지)



- stack(doubly linked) 갱신 비용 ↑
- 희생자(bottom of stack) 선정은 효율적.

(note) **Stack Algorithm**

- Belady's anomaly를 야기하지 않는
페이지 교체 알고리즘
- { pages in **n** frames }
⊆ { pages in **(n+1)** frames }

(예)

LRU 알고리즘
최적 알고리즘

□ LRU 근사 페이지 교체 LRU-Approximation Page Replacement Algorithm

※ LRU 알고리즘 구현을 위해 TLB Register 이상의 하드웨어 필요.
그러나 이를 충분히 지원하는 Computer는 거의 없음.

per-page 참조비트를 사용하여 구현함:

- 페이지 참조時 마다 “1”로 설정함 (by hardware).
- (참조 순서(시간)는 알 수 없으나) 페이지 참조 여부를 알 수 있음.

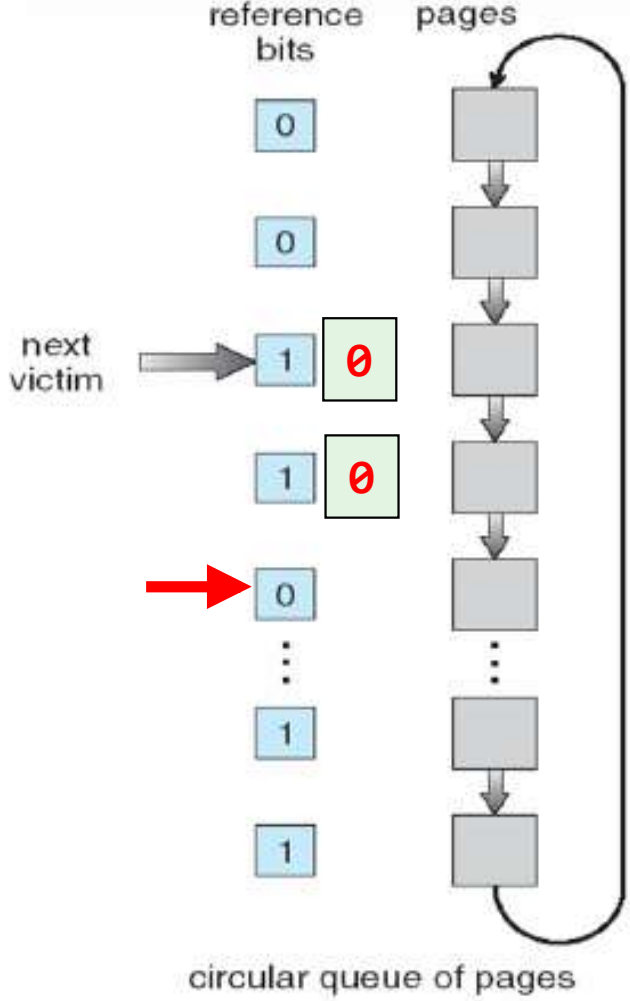
- 참조비트를 사용하는 LRU 근사 알고리즘:

추가 참조 비트, 2차 기회, 개선된 2차 기회 알고리즘

• 추가(부가) 참조 비트 알고리즘 Additional-Reference Bits Algorithm

M/H	<p>각 page에 대해 shift-right가 가능한 Reference Bit와 History Register를 둬:</p> <div data-bbox="602 363 1751 477" data-label="Diagram"> <pre> graph LR A[참조비트] --> B[히스토리 레지스터] C[페이지 참조 정보가 shift-right 되어 들어옴.] </pre> </div> <ul style="list-style-type: none"> • 페이지 참조時 마다 참조 비트가 “1”로 설정되며 (by hardware) • 주기적으로 <u>shift-right</u>가 일어남 (by timer interrupt).
원리	<ul style="list-style-type: none"> • history register는 페이지 참조 역사를 나타냄: 값이 작을수록 더 오래 전에 참조되었음을 나타냄. ※ shift-right 하기때문. <p>(예) 1000 0000 ⇨ 가장 최근에 참조됨.</p> <p>(예) 0000 0001 ⇨ 가장 오래 전에 참조됨.</p> <p>(예) 0000 0000 ⇨ 최근 8구간 동안 한 번도 참조되지 않음.</p> <ul style="list-style-type: none"> • victim = history register 값이 최소인 페이지, 즉 가장 오래전에 접근된 페이지. <p>(예) p1: 0100 1100 > p2: 0011 1111</p> <p>⇨ p2가 더 오래 전에 참조됨, p2가 victim.</p>

• 이차 기회 알고리즘 (or Clock Algorithm) Second-Chance Algorithm

알고리즘	<ul style="list-style-type: none"> 기본적으로 FIFO 알고리즘. front: next victim을 가리킴. (가장 오래 전에 적재된 페이지) <pre> while (참조비트==1) { //가장 오래되었지만 최근 참조됨. 참조비트=0; 페이지를 rear로 옮김; //2차 기회 부여 front 이동; } //가장 오래되면서 참조비트==0인 page 발견. victim 확정. 페이지 교체; </pre>	
구현	<ul style="list-style-type: none"> Circular Queue 이용하여 구현. pointer(or 시계 바늘): next victim을 가리킴. <pre> while (참조비트 == 1) { 참조비트 = 0; pointer 이동; } </pre> <p>페이지 교체;</p>	

● 개선된 이차 기회 알고리즘 Enhanced Second-Chance Algorithm

알고리즘

- 기본적으로 이차 기회 알고리즘 + Page의 등급의 이용.
- (참조비트, 변경비트)를 이용하여 page 등급 설정:

낮음	(0,0) : 최근 사용×, 변경×	교체하기에 가장 좋은 페이지
↑	(0,1) : 교체 時 <i>disk write</i> 필요	
	(1,0) : 조만간 다시 사용될 가능성 높음	
높음	(1,1) : 최근 사용○, 변경○	교체하기에 가장 나쁜 페이지

- victim = 등급이 가장 낮은 페이지 중 최초 검색된 페이지
※ circular queue를 여러 번 검사할 가능성 있음.

비고

- Second-Chance Algorithm과의 차이 (장점)
변경된 페이지의 우선순위를 높여 disk I/O 횟수를 줄임.

□ 계수 기반 페이지 교체 Counting-Based Page Replacement

LFU (least frequently used)	MFU (most frequently used)
각 페이지의 참조 횟수(counter)에 기반 한 알고리즘	
가장 적게 참조된 페이지 (page with min. counter)	가장 많이 참조된 페이지 (page with max. counter)
(문제점) 한동안 집중적으로 사용된 후 더 이상 사용되지 않는 페이지가 메모리에 남게 됨.	(근거) 가장 적게 참조된 페이지는 <u>방금</u> <u>적재된</u> 페이지
(해결방안) counter를 주기적으로 shift-right ($\div 2$) ⇒ 시간이 지남에 따라 counter의 영향을 지수적으로 감소시킴.	

※ 두 알고리즘 잘 사용되지 않음: 높은 구현 비용, 최적의 근사 알고리즘이 아님.

□ 페이지 버퍼링 Page Buffering Algorithm

- **Free Frame Pool**을 이용한 **page buffering** (페이지 교체 성능 ↑)

page fault 時:

- ① 먼저 요구된 페이지를 free frame pool로 읽어 옴. ⇨ 조기 실행 가능
- ② 연후에 희생자 페이지를 page out.
- ③ victim이 사용하던 frame을 pool에 추가. (⇨ 항상 pool이 유지됨)

Modified page list 병행 사용 가능

- disk idle 時 변경된 페이지를 write to disc하고 변경비트를 reset함.
- 실제 페이지가 교체될 때 disk write가 불필요할 가능성 높아짐.

소유자를 가지는 **Free frame pool**

- *free frame*의 **소유자** 페이지를 기록함.
- page fault 時 먼저 pool에서 페이지를 찾음
- victim을 잘못 선정할 경우 불이익을 줄일 수 있음. (환원이 용이함.)

4. 프레임 할당

□ 프로세스에게 할당된 Frame 개수

- 할당된 **frame** 개수 ↓ ⇨ 페이지 부재율 ↑ (높아지고), 성능이 떨어짐.
- 각 프로세스가 필요로 하는 **최소** 프레임 개수는?
 - **instruction**의 실행이 완료되려면 자신이 참조하는 모든 page가 적재되어야 함.
(instruction 실행 도중 (page-fault) interrupt를 처리할 수 없기 때문에.)
 - (예) **add a b c** → 1 code frame, frames that store a, b, and c.
 - (예) **add a b @c** → 주소 지정 방식에 따라 frame 개수가 영향을 받음.
 - 따라서 최소 프레임 개수는 instruction set architecture에 의해 결정됨.
- 각 프로세스에게 할당 가능한 **최대** 프레임 개수는?
 - 가용 물리메모리의 크기에 따라 결정됨.
- 프로세스에게 할당되어야 할 프레임의 **적정** 개수는?

□ 프레임 할당 알고리즘

- **균등 할당** Equal Allocation - 모든 프로세스에게 **같은 수의 frames** 할당.
- **비례 할당** Proportional Allocation - **프로세스 크기에 비례**하여 **frames** 할당.
 ※ 위 두 알고리즘에서는 할당량이 current multiprogramming degree에 의존적.
- **우선순위 할당** Priority allocation - **프로세스 우선순위에 비례**하여 **frames** 할당.

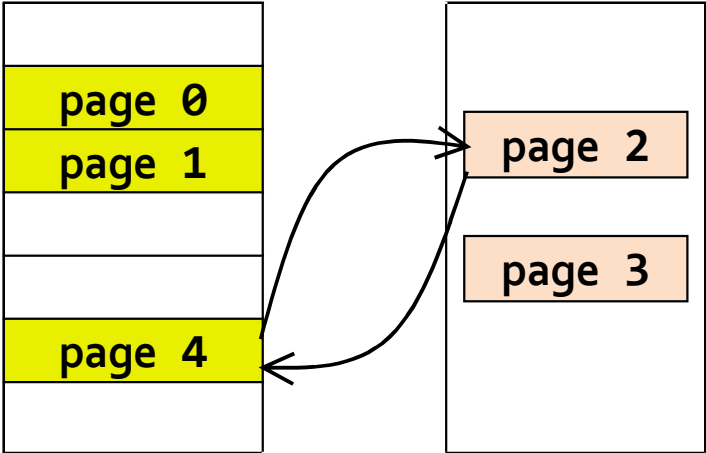
□ 지역 할당 Local Allocation와 전역 할당 Global Allocation

	지역 할당 (교체)	전역 할당 (교체)
victim 대상	프로세스 자신의 frame 내에서 선택	다른 프로세스의 frame도 대상이 됨 (예) (우선순위 + 전역) 할당
frame 총수	페이지 교체 전후 프로세스의 frame 총수는 불변	다른 프로세스에서 victim이 선정될 수 있음 (페이지 교체 후 할당된 frame 개수 1 증가)
페이지 부재율	<u>자신의 참조열</u> 에 만 의존적	자신의 참조열 뿐만 아니라 같이 실행중인 <u>다른 프로세스들</u> 에도 의존적
비고	사용 빈도가 낮은 frame들을 <u>다른</u> 프로세스가 사용할 수 없음	- 일반적으로 system throughput이 높으며 - 보다 많이 사용됨.

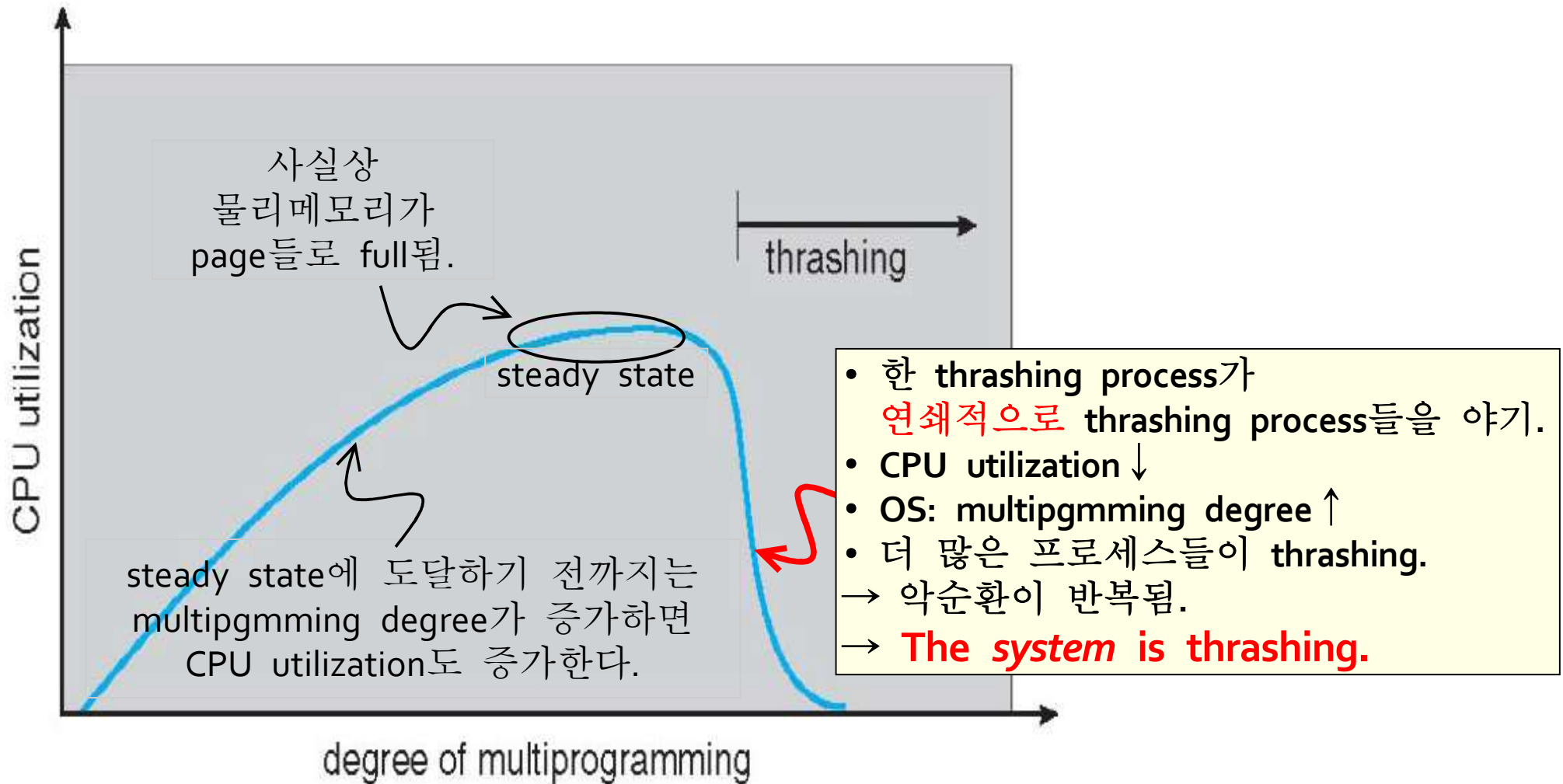
5. Thrashing

(사전적 의미) 完敗, 허우적거림.

A process is *thrashing* if it is spending more time in paging than an executing.

Situation	Consequence
<p>A process has:</p> <ul style="list-style-type: none"> - 5 <i>actively-used</i> pages, page 0~4. - maximum 3 frames  <p>Physical Memory</p> <p>Swap Space</p>	<ul style="list-style-type: none"> ◦ page fault 발생; 곧 참조될 페이지를 교체; 악순환이 반복되어 → 연쇄적으로 page fault가 발생함. The process quickly faults again, and again, and again, replacing pages that it must bring back in immediately. ◦ The process is spending most of its time paging rather than executing. ◦ The process is Thrashing.

□ 전역 교체와 Thrashing



(note) 지역 교체의 경우는 한 프로세스의 thrashing이 다른 프로세스에게 파급되지 않음;
그러나 thrashing process들에 의해 다른 프로세스들의 paging 대기 시간은 길어짐.

□ Thrashing 예방 방안

작업 집합 모델 Working Set Model	페이지 부재 빈도 Page Fault Frequency
thrashing 이 일어나지 않도록 프로세스가 <u>필요로 하는</u> 만큼의 프레임을 할당함.	페이지 부재율 을 관찰하고 이에 따라 프로세스의 frame 수를 <u>조절(증가/감소)</u> 함.
적정한 frame 개수를 찾는 방법은?	페이지 부재율 상·하한 설정 방법은?

작업집합 모델

Working Set Model

Thrashing이 일어나지 않도록 프로세스가 필요로 하는 만큼의 프레임을 할당함.

- 작업 집합 모델

- 한 프로세스가 필요로 하는 프레임 개수 구하는 방법.
- *Locality*에 기반하며, 이의 Approximation 인 *Working Set*을 사용함.

- ※ 지역성 모델

- ※ 작업 집합 모델

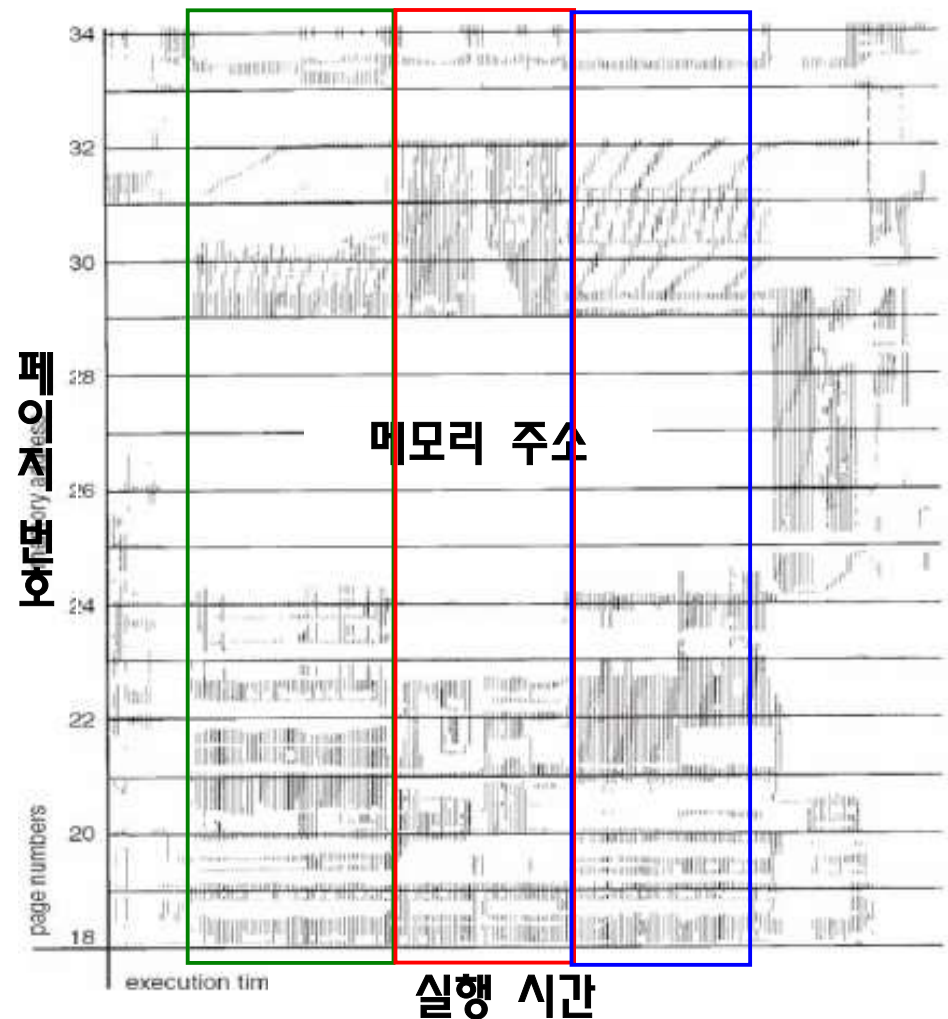
□ 지역성 모델 Locality Model of a process execution

- 프로그램은 한 동안 특정 부분이 집중 참조되는 특징을 가진다.
- **Locality, 지역**
 - 동시에 활발하게 사용되는 페이지 집합.
 - 예) 함수 호출시 locality:
{함수코드, 지역변수들, 전역변수 일부}
 - 프로그램/데이터 구조에 의해 정의됨.
 - demand paging, caching의 기반이 됨.

Locality Model:

- 일반적으로 프로그램은 여러 개의 **지역(locality)**으로 구성되며 (상호 중첩 가능)
- 프로세스 실행이 진행됨에 따라 하나의 **locality**에서 다른 **locality**로 이동함.

<메모리 참조 패턴의 지역성>

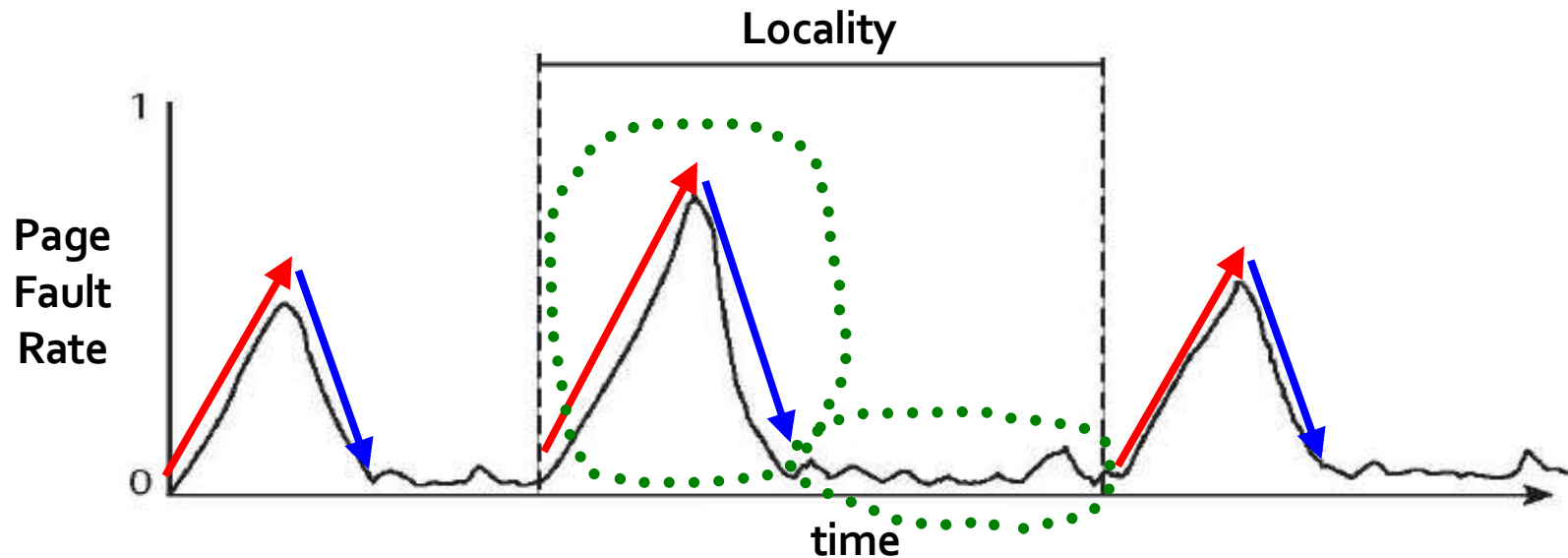


프로세스가 새로운 locality에 진입할 때

- 충분한 frame을 할당받지 못하면 ($\text{allocated frames} \ll \text{process' locality}$)
locality 내에서 실행되는 동안 계속하여 page fault 발생 → The process is thrashing.
- if ($\sum \text{locality size} > \text{total memory size}$) → The system is thrashing.

충분한 frame을 할당받으면

locality에 속하는 모든 page가 적재될 때까지는 page fault 발생 $\uparrow\downarrow$,
그 이후부터 locality를 벗어나기 전까지 page fault를 발생 않음.



새로 진입한 Locality의 충분한 Frame의 수는?

□ 작업집합 모델 Working-Set Model

- 한 프로세스가 필요로 하는 프레임 개수

= Locality 크기

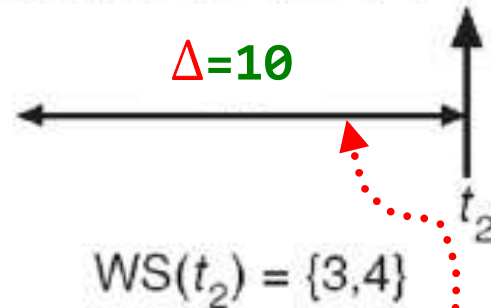
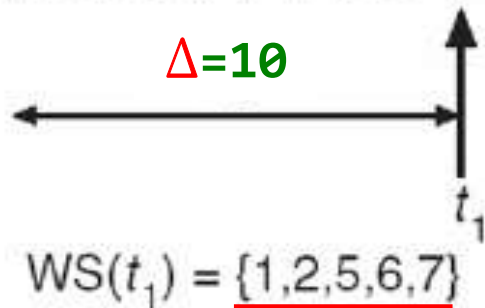
\cong 작업집합(*working set*) 크기.

- Locality = 함께 활발하게 사용되는 페이지 집합
- Working Set = 최근 작업집합 구간에서 참조된 페이지 집합 (Locality의 근사)

- 용어: 작업집합 구간, 작업집합, 작업집합 크기

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



Δ 작업집합 구간 (working set window)
 관찰할 페이지 참조 횟수 (상수).
 ※ 메모리 참조 때마다 이동함.

WS (working set) process의 작업집합
 최근 작업집합구간 Δ 에서 참조된 페이지 집합.

WSS (working set size) process의 작업집합 크기
 작업집합 內 페이지 개수.

● 작업집합

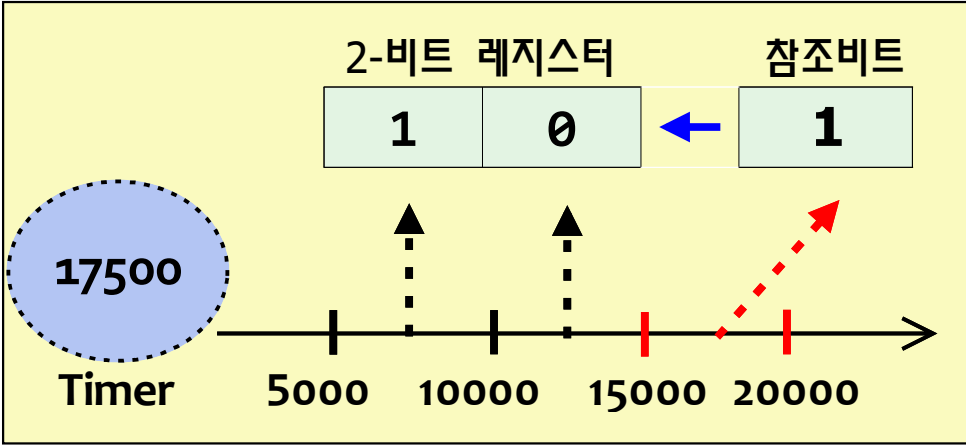
- 최근 작업집합 구간에서 참조된 페이지 집합
- **locality**(함께 활발히 사용되는 페이지 집합)의 근사 값
- 작업집합의 **정확성**은 구간 크기에 의존적
 - 구간이 작으면, WSS가 작아짐 \Rightarrow 한 locality 전체를 포함하지 못함
 - 구간이 커지면, WSS가 커짐 \Rightarrow 여러 locality를 포함
 - $\Delta = \infty$ 이면 참조되는 모든 페이지 포함

● 작동 원리

- 각 프로세스_i에게 **작업집합 크기_i** 만큼의 프레임 할당.
- if ($\sum WSS_i > \text{가용프레임 총수}$) // thrashing 발생 가능하므로
 swap out some processes. // multiprogramming degree 낮춤

\Rightarrow thrashing 방지하면서,
multiprogramming degree 최대한 유지 & CPU 이용률 최적화 함.

• 작업집합 추적 방법

하드웨어	작업집합 추적
 <p>2-비트 레지스터 1 0</p> <p>참조비트 1</p> <p>Timer 17500</p> <p>5000 10000 15000 20000</p>	<p>Let $\Delta = 10,000$ 메모리참조</p> <p>① 페이지 참조 時, 참조비트 $\leftarrow 1$;</p> <p>② timer interrupt 발생 時, 각 페이지에 대해 히스토리레지스터 \leftarrow 참조 비트; 참조비트 = 0;</p>
<ul style="list-style-type: none"> Fixed-Interval Timer 주기적으로 인터럽트 발생 (예) 5,000 메모리 참조마다 인터럽트 발생 Reference Bit per page 현재 주기 내 페이지 참조 여부를 나타냄 History Register per page 최근 2 주기 내 참조 여부를 나타냄 	<p>$WS_i = \{3 \text{ 비트 중 } 1 \text{ 비트라도 "1" 인 페이지}\}$</p>

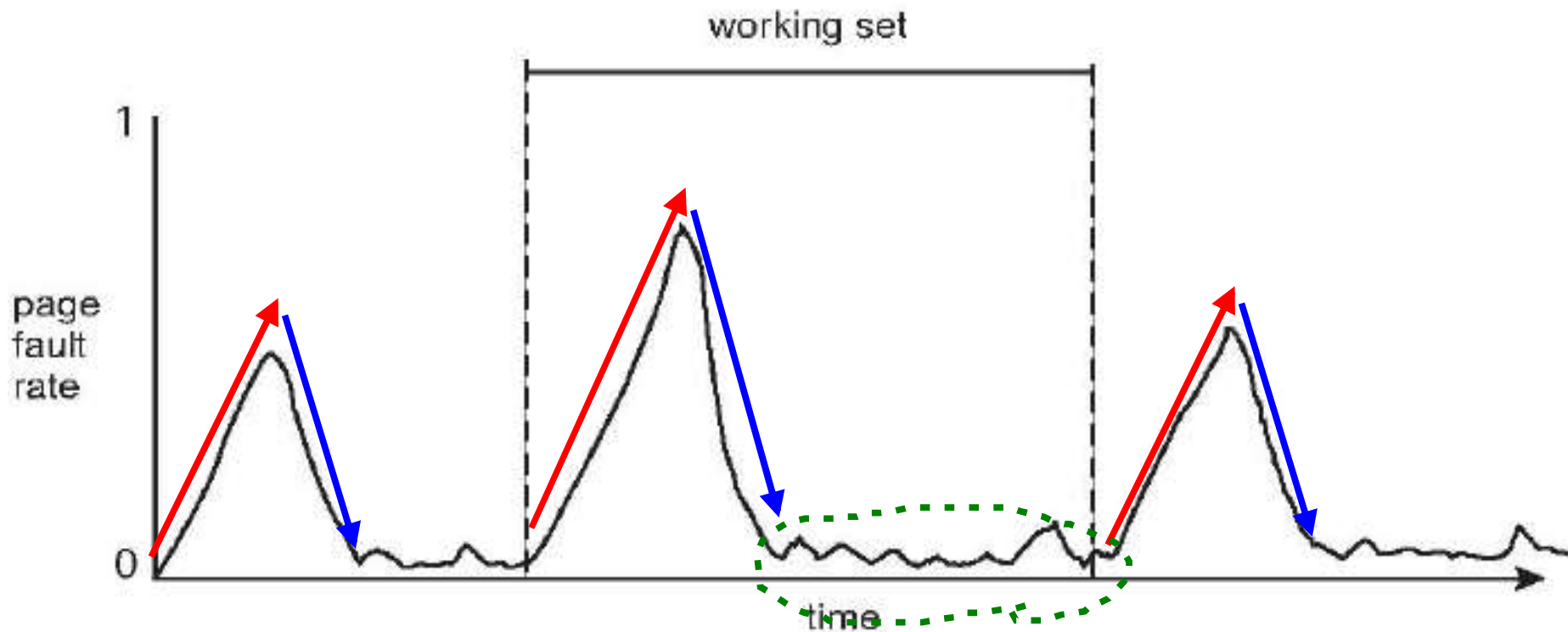
(note) 작업집합의 정확도를 높이려면 timer 주기는 짧게, history register 크게 함.

- 작업집합과 페이지 부재율과의 연관성

새로운 locality에 진입하면 page fault rate \uparrow ,

*working set*이 적재되면 page fault rate \downarrow

그 이후부터 Locality를 벗어나기 전까지 Page Fault를 발생 않음.



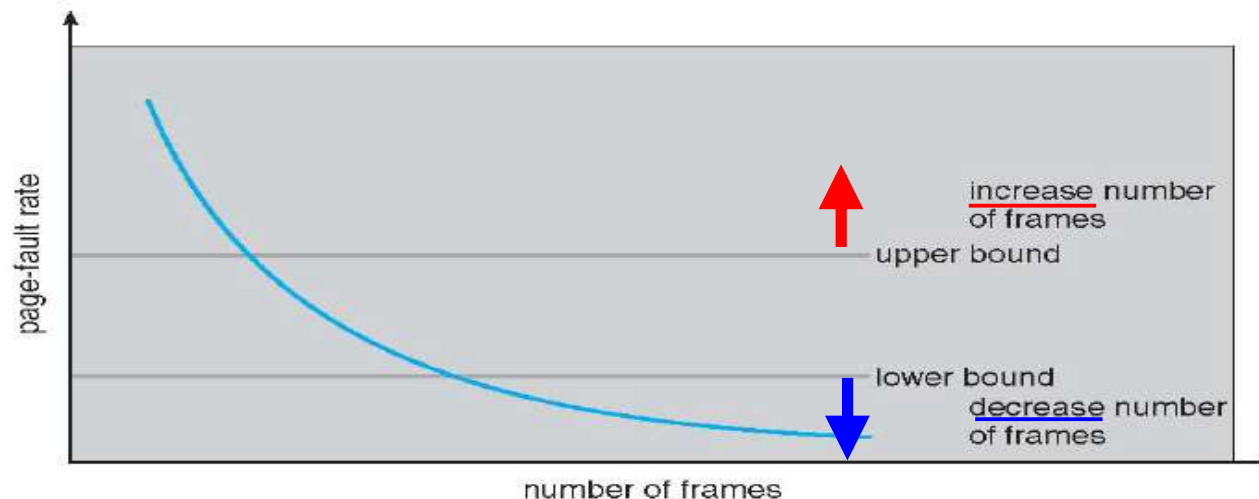
페이지 부재빈도 기법 PFF

Page-Fault Frequency

Thrashing이 일어나지 않도록 페이지 부재율을 **관찰**하고 **조절**함.

(note) 작업집합 모델 보다 더 직접적으로 thrashing 조절함.

- 바람직한 page fault rate 범위(상한, 하한) 설정
- 관찰한 페이지 부재율에 따라 프로세스에게 할당할 frame수 조절(추가/회수)
 - 실재 페이지 부재율 > **상한** \Rightarrow 프레임 하나 추가
if (no free frame) swap out some process.
 - 실재 페이지 부재율 < **하한** \Rightarrow 프레임 하나 회수



Windos XP:

- 프로세스 생성時 작업집합의 상하한 설정
상하한) 시스템 보장 최소/최대 페이지 수
- **free memory**가 임계치에 도달하면 최소 작업집합 초과 수 만큼 페이지 회수.
(*automatic working set trimming*)

Solaris:

- **free memory**가 임계치(*lotsfree* parameter) 이하로 떨어지면 **paging** 시작함.
2차 기회 페이지 알고리즘과 유사한 교체알고리즘 사용함.
- **pageout** 時 **shared library** 제외함.

6. 메모리 사상 파일 Memory-Mapped File

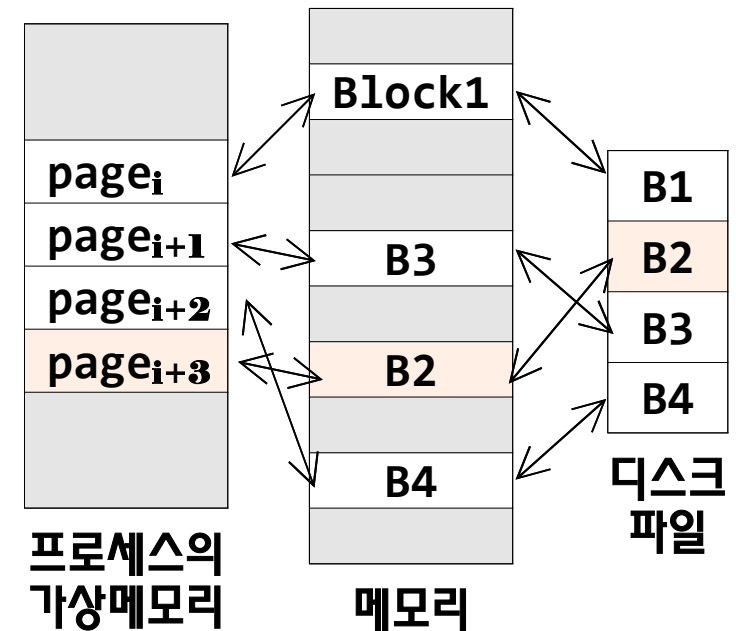
- 가상메모리 기법을 file I/O에 적용한 기법.
- **Process**의 가상주소 공간의 일부를 파일에 할애.

• 동작 원리

- ① 파일을 **process**의 페이지에 사상함.
- ② 최초 파일 접근 시 **page fault** 발생되며,
이때 **disk block**을 **page(memory)**로 읽어 옴.
So, memory-mapped file.
- ③ 차후 모든 **file I/O**는 **memory R/W**로 대신됨.
- ④ 변경된 page는 주기적 또는 close() 호출 시
실제 디스크에 write 됨.

• 장점

Disk I/O 성능 개선.
파일 공유 허용함.

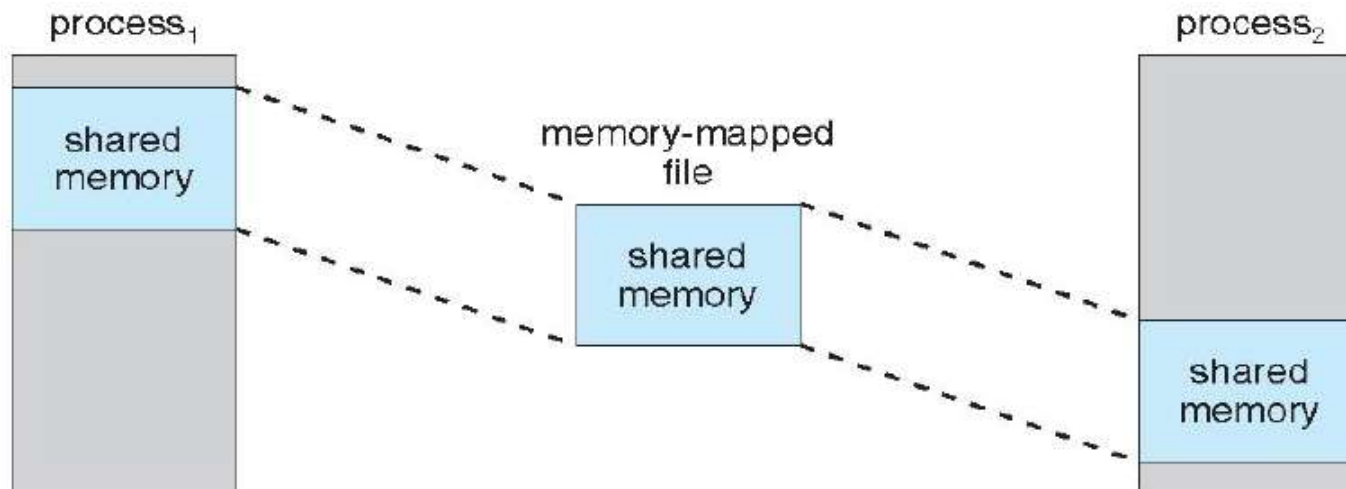


(예) Solaris

- 모든 파일 입출력을 **memory**에 **mapping** 시킴.
 - 파일을 **mmap**로 지정하면 **process** 주소공간에 사상
 - 파일을 **open/read/write**로 접근하면 **kernel** 주소공간에 사상

(예) Windows

- **memory-mapped file**을 이용하여 **shared memory** 지원



7. Kernel Memory 할당

- 다양한 크기의 kernel data structure를 할당해야 함.
- 두 가지 할당 방법:
 - Buddy system
 - Slab allocation

Buddy System

메모리를 2^n 단위로 할당함.

buddy) 친구, (단)짝.

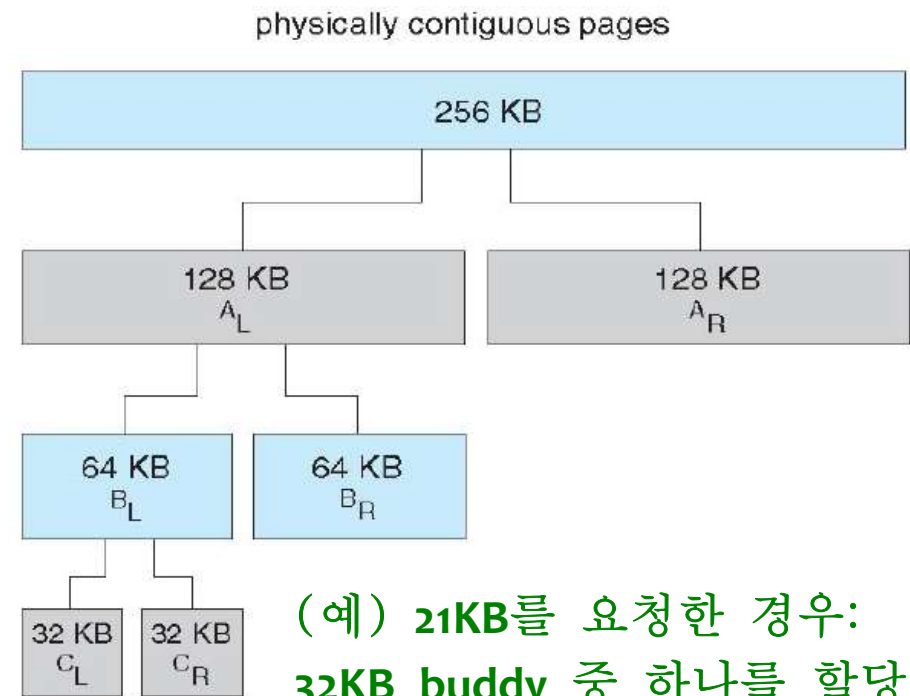
할당 방법:

요청된 크기(S)의 메모리를 얻을 때까지 **segment**를 두 개의 **buddy**로 나누고,
2개 **buddy** 중 하나를 할당함.

※ 반환 시 이웃하는 2개의 **free buddy**는
하나로 합병함.

(note)

- 내부 단편화 발생함.
- 고비용.



Slab Allocation

효율성 제고를 위해 **kernel object "pool"**을 이용함.
(예) Solaris, Linux, FreeBSD, etc.

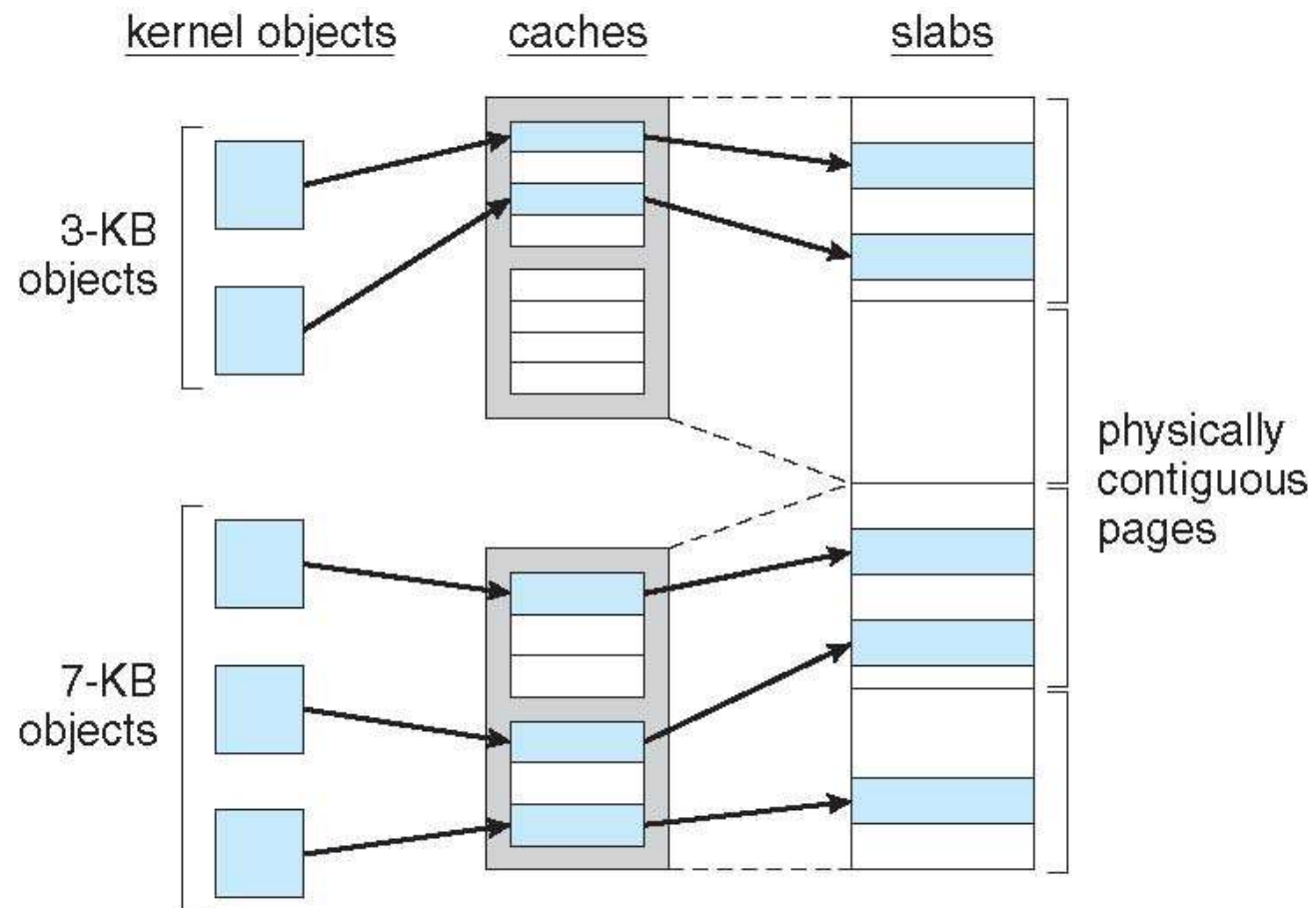
- 여러 유형(크기)의 kernel data structure
(예) semaphore, process descriptor
- A **cache** = A pool of free kernel objects of the same kernel data structure type
(예) semaphore cache, proc. desc. cache
 - State of a cache: *full, empty, partial.*
full) All objects in the slab are marked used.
- When a new kernel object is needed, **slab allocator** assign any free object from the cache of that kernel type.
- A **slab** = One or more contiguous pages
 - A **slab** is assigned to a **cache**.

Kernel:

Requests memory for an kernel object.

Slab Allocator:

- First attempts to satisfy the request with a free object in a partial slab.
- If none exists, a free object is assigned from an empty slab.
- If no empty slab are available, a new **slab** is allocated from contiguous physical pages and assigned to a **cache**; memory for the object is allocated from this slab.



8. Page Fault에 대한 기타 고려 사항

- **Pre-paging** (사전 페이징)
- Page size
- TLB reach
- Program structure
- I/O interlock

□ Pre-paging

과도한 page fault를 방지하기 위해

프로세스가 필요로 하는 페이지 전부 또는 일부를 사전에 적재하는 것.

- 작은 파일은 전체를 사전에 적재함. (Solaris)
- 일시 정지된 프로세스 실행 재계時 working set을 한꺼번에 적재.
- page fault時 해당 페이지와 다음 n pages를 pre-paging. (Windows XP)

• Pre-gaging 비용 vs. Page fault 서비스 비용

s 개 페이지 pre-paging 하여 a ($0 \leq a \leq 1$) 비율만큼 사용된 경우:

- $(1-a)s$ 개 불필요한 페이지를 pre-paging한 비용
- (as) 번의 page fault 처리 비용

□ 페이지 크기

- 페이지 크기: 항상 2^n 크기, 일반적으로 4KB ~ 4MB.

Large Page	Small Page
페이지 테이블 크기 ↓	내부 단편 크기 ↓
디스크 입출력 시간 ↓	Improved locality
페이지 부재율 낮아짐	
The historical trend is toward <u>larger</u> page sizes, even for mobile systems.	

□ TLB Reach

- TLB 적중률을 높이려면? TLB 용량 증가 (고비용) vs TLB 범위 증가
- **TLB reach** = TLB로부터 접근 가능한 메모리 크기 = **TLB 크기 × 페이지 크기**
- TLB reach를 늘리려면? 페이지크기 ↑ (내부단편 ↑) vs. Multiple page sizes

(예) **UltraSPARC**: 8KB, 64KB, 512KB, 4MB

Solaris: 8KB, 4MB

□ 프로그램 구조

- 프로그램 구조

(예) 128-word page, $a[128][128]$ 초기화, row-major 저장 방식

<pre>for (i=0; i<128; i++) for (j=0; j<128; j++) a[i][j] = 0;</pre>	<pre>for (j=0; j<128; j++) for (i=0; i<128; i++) a[i][j] = 0;</pre>
128 page faults	128×128 page faults

- 자료 구조

(예) Locality: stack은 높음, hash table은 낮음

- 프로그래밍 언어

C의 포인터, 객체지향프로그래밍언어: locality ↓

□ 입출력 상호 잠금 I/O interlock

Problem:

입출력이 user address space에서 이루어 질 때
(우선순위가 높은) 다른 프로세스에 의해
입출력장치 큐에서 대기 중인 프로세스의 page(입출력 버퍼)가 교체되면?
(깨어났을 때 해당 frame은 이미 다른 프로세스가 사용 중.)

Solution 1) 입출력을 kernel address space에서만 실행함

I/O device ↔ system buffer (입출력 완료時) ↔ user buffer ⇒ 자료 2번 이동

Solution 2) I/O interlock¹⁾

- 각 frame에 lock bit를 둠
- 입출력 버퍼 페이지가 저장된 frame을 lock 시킴 (*Page Pinning*_{고정})
(⇒ 페이지 교체에서 제외됨)
- 위험성 수반 (예) OS의 오류로 **unlock** 되지 않음
(note) Solaris는 free page pool이 너무 작거나,
page lock 요청이 너무 많으면 적절히 무시함.

interlock

- 어떤 제어장치가 동작하고 있을 때 다른 제어 장치가 동작해서는 안 되는 경우,
양자 간 안정을 위해 두어지는 보호 장치.
- 한쪽이 안전 상태가 되지 않으면 다른 쪽의 동작을 할 수 없게 하는 것
- 소정의 전제 조건이 만족되어야만 다음 조작이 되도록 한 장치.