

Chapter

## 10

## 가상 메모리

1. 물리 메모리의 한계
2. 가상 메모리 개념
3. 요구 페이징(demand paging)
4. 참조의 지역성과 작업 집합
5. 프레임 할당
6. 페이지 교체

10장 가상메모리



# 강의 목표

1. 가상 메모리 기법의 개념을 이해한다.
2. 가상 메모리 기법에 대한 여러 의문점들과 해결에 대해 이해한다.
  - 스래싱, 페이지 테이블 구성, 페이지 폴트, 페이지 할당, 스왑 영역
  - 프레임 할당, 작업 집합, 페이지 교체, 쓰기 시 복사
3. 요구 페이징의 개념과 페이지 폴트에 대해 안다.
4. 요구 페이징 시스템에서 프로세스의 실행 과정을 이해한다.
5. 프로세스의 생성에 사용되는 쓰기 시 복사에 대해 이해한다.
6. 연속된 페이지 폴트로 인해 심각한 디스크 입출력이 발생하는 스래싱 현상에 대해 이해한다.
7. 참조의 지역성, 작업 집합, 페이지 폴트 사이의 관계에 대해 안다.
8. 프로세스에게 메모리 프레임을 할당하는 정책에 대해 이해한다.
9. 페이지 교체의 개념과 다양한 페이지 교체 알고리즘을 이해한다.
  - 최적 교체, FIFO, LRU, Clock

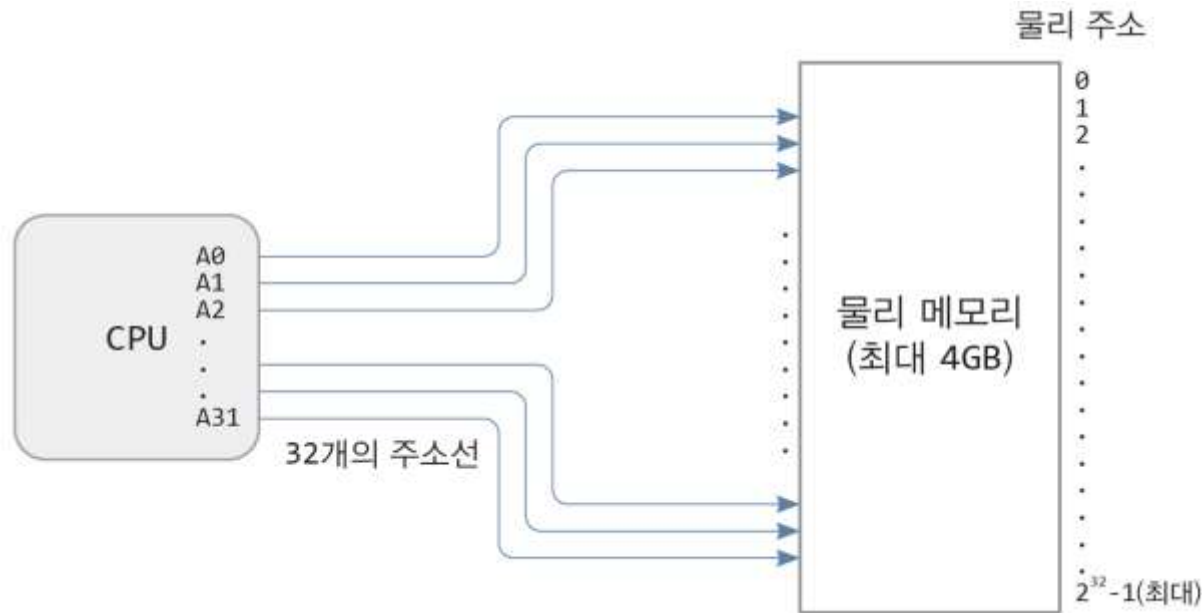
3

# 1. 물리 메모리의 한계

# 주소 공간과 물리 메모리

4

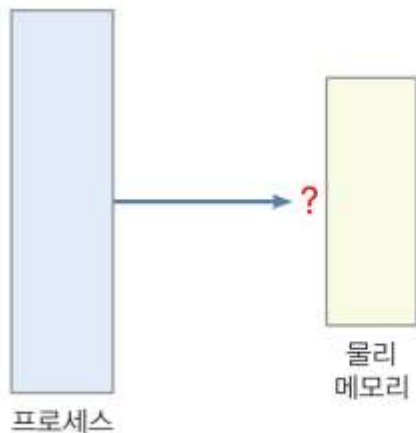
- 컴퓨터에 설치할 수 있는 물리 메모리의 최대 크기
  - ▣ 물리 메모리의 최대 크기는 CPU 주소 버스 크기에 달려 있음
  - ▣ 32비트 CPU의 물리 메모리 최대량 : 4GB
  - ▣ 64비트 CPU의 물리 메모리 최대량 :  $2^{24}$ TB
- 실제 컴퓨터에 설치되는 물리 메모리
  - ▣ 최대량에는 미치지 못하는 크기, 비용 때문
  - ▣ 현재 대부분의 64비트 컴퓨터에서 8GB ~ 32GB 수준 *비사서*



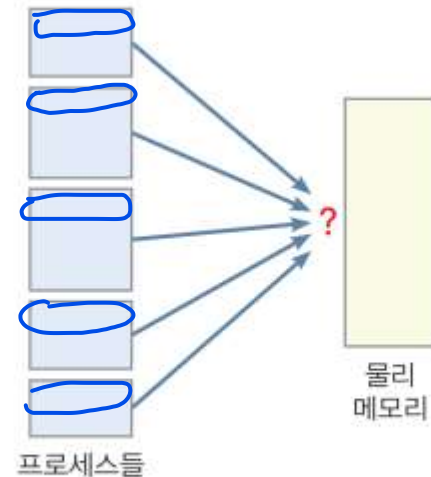
# 물리 메모리의 한계

5

- 물리 메모리의 크기 한계에서 비롯된 2가지 질문
  1. 설치된 물리 메모리보다 큰 프로세스를 실행시킬 수 있는가?
  2. 프로세스들을 합친 크기가 설치된 물리 메모리보다 클 때 이들을 모두 실행시킬 수 있는가?
- 질문에 숨어 있는 전제
  - ▣ 전제 : 프로세스 전체가 물리 메모리에 적재 되어야 한다.
  - ▣ 이 전제를 부정하여 크기 한계를 극복하는 방법을 찾아보자.
    - 과연 프로세스 전체가 완전히 메모리에 적재되어야 실행이 가능한 것일까?
    - 당장 실행에 필요한 프로세스의 일부 메모리만 적재한 채 실행시킬 수는 없는가?



(a) 프로세스가 물리 메모리보다 큰 경우



(b) 여러 프로세스들을 합친 크기가 물리 메모리보다 큰 경우

6

## 2. 가상 메모리 개념

# 가상 메모리 개요

7

## □ 가상 메모리의 필요

- 물리 메모리 크기 한계를 극복하는 해결책

## □ 가상 메모리 기법의 핵심 2가지

### 1) 물리 메모리를 디스크 공간으로 확장

- 프로세스를 물리 메모리와 하드 디스크(보조기억장치)에 나누어 저장
- 프로세스나 사용자가, 프로세스를 실행하기에 충분히 큰 물리 메모리가 있다고 착각하게 만드는 메모리 관리 기술

### 2) 스와핑(swapping)

- 물리 메모리가 부족할 때, 실행에 필요하지 않은 부분<sub>을</sub> 하드 디스크로 이동
- 실행에 필요할 때 하드 디스크에서 물리 메모리로 이동

# 가상 메모리 개념

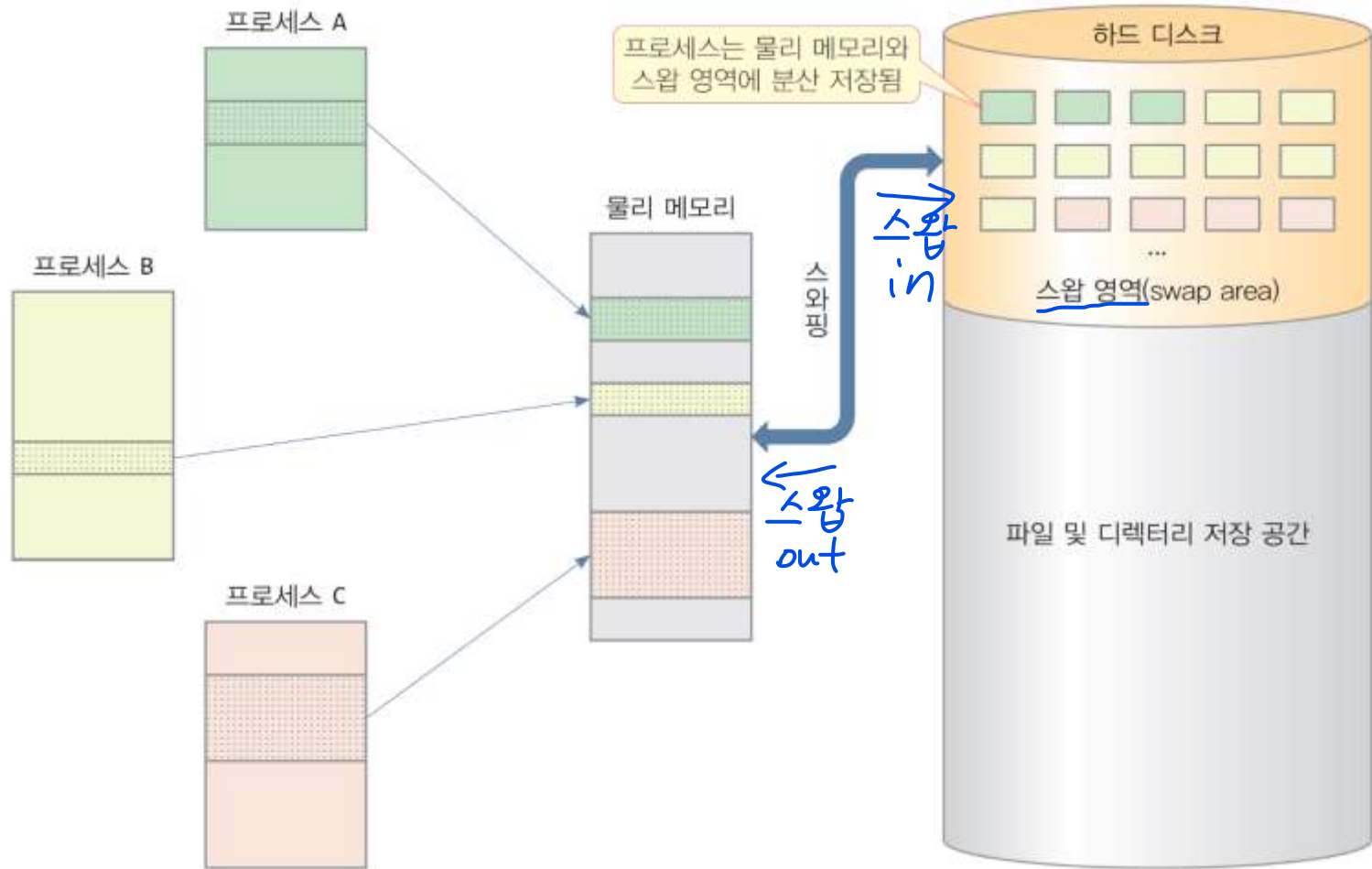
8

1. 운영체제는 물리 메모리 영역을 하드 디스크로 연장
  - 프로세스를 물리 메모리와 하드 디스크에 나누어 저장
2. 프로세스 실행 시 프로세스 전체가 물리 메모리에 적재되어 있을 필요 없음
  - 현재 실행에 필요한 코드나 데이터의 일부만 있으면 실행 가능
  - 나머지는 하드 디스크에 있다가 필요할 때 메모리에 적재해도 됨
3. 운영체제는 물리 메모리의 빈 영역이 부족하게 되면, 물리 메모리 일부분을 하드 디스크로 옮겨 물리 메모리의 빈 영역 확보
  - 이 방식으로 많은 프로세스를 메모리에 적재 가능
  - 다중프로그래밍 정도 높임, CPU 활용율과 처리율 높임
    - 다중프로그래밍 정도란 메모리에 적재하여 동시에 실행시키는 프로세스의 개수
4. 물리 메모리를 확장하여 사용하는 디스크 영역을 스왑 영역이라고 부름
  - 스왑-아웃 : 물리 메모리의 일부를 스왑 영역으로 옮기는 작업
  - 스왑-인 : 스왑 영역에서 물리 메모리로 가지고 오는 작업
5. 사용자는 컴퓨터 시스템에 무한대의 메모리가 있는 것으로 착각
  - 사용자는 큰 프로그램을 작성하는데 부담 없음
  - 사용자는 여러 프로그램을 동시에 실행시키는 데 부담 없음
6. 가상 메모리는 운영체제마다 구현 방법 다름



# 가상 메모리를 사용하는 시스템에서 프로세스와 물리 메모리, 하드 디스크의 관계

9



\* 사용자나 프로세스 : 무한대에 가까운 큰 메모리를 사용하며, 프로세스가 0번지부터 연속되어 적재되는 것으로 생각

\* 운영체제 : 프로세스를 물리 메모리와 하드디스크에 분산 저장, 프로세스의 일부만 물리 메모리에 적재

\* 디스크 : 물리 메모리에 빈 공간이 부족하여 메모리 일부를 스왑 영역에 저장

# 가상 메모리 구현

10

## □ 대표적인 가상 메모리 구현 방법

### ▣ 요구 페이징(demand paging)

페이징 기법을 토대로 프로세스의 일부 페이지들만 물리 메모리에 할당하고, 페이지가 필요할 때 물리 메모리를 할당받고 페이지를 적재시키는 메모리 관리 기법

■ 요구 페이징 = 페이징 + 스와핑(swapping)

### ▣ 요구 세그먼테이션(demand segmentation)

■ 세그먼테이션 기법 + 세그먼트 스와핑

# 풀어야 할 가상 메모리 기법에 대한 의문들

11

- (스래싱 문제) 물리 메모리와 디스크의 스왑 영역 사이에 입출력이 너무 빈번히 발생하지 않는지?
- (페이지 테이블) 페이지 테이블은 어떻게 구성할지?
- (페이지 폴트) 가상 주소를 물리 주소로 변환할 때 페이지가 프레임에 없는 경우 어떻게 처리할지?
- (페이지 할당) 프로세스의 어떤 페이지를 물리 메모리에 두고 어떤 페이지를 하드 디스크에 둘지?
- (스왑 영역) 디스크의 스왑 영역 크기는 얼마가 적당한지?
- (프레임 할당) 프로세스별로 할당할 프레임의 개수를 몇 개로 정할지?
- (작업 집합) 프로세스는 일정 시간 범위에서 실행 중에 몇 개의 프레임을 실제로 사용하고 있는가?
- (페이지 교체 알고리즘) 필요한 페이지를 디스크로부터 읽어 오기 위해 프레임 중 하나를 비워야 하는데 어떤 프레임을 비워야 하는지? 비워야 하는 프레임이 결정되면 그 프레임에 저장된 페이지는 어떻게 처리할 것인지?
- (쓰기 시 복사) 프로세스가 자식 프로세스를 생성하면 자식 프로세스의 메모리 공간은 어떻게 되는지?
- 등등...

### 3. 요구 페이징

# 요구 페이징 개념(1)

13

## □ 개념

- 현재 실행에 필요한 일부 페이지만 메모리에 적재하고 나머지는 하드 디스크에 두고, 페이지가 필요할 때 메모리에 적재하는 방식

- 프로세스의 페이지가 있는 디스크 영역 = 스왑 영역 + 실행 파일

- **요구(demand)의 의미** : 페이지가 실행에 필요할 때

## □ 요구 페이징 구현의 전형적인 형태

- 운영체제는 첫 페이지만 물리 메모리에 적재하고,  
■ 실행 중 프로세스가 다른 페이지를 필요로 할 때 페이지 적재

## □ 스왑 영역

- 메모리가 부족할 때, 메모리를 비우고 페이지를 저장해두는 하드 디스크의 영역
  - 리눅스 : 디스크 내 특별한 위치 혹은 스왑 파티션에 구성
  - Windows : C:/pagefile.sys 파일을 스왑 영역으로 사용

# 요구 페이징 개념(2)

14

## □ 페이지 테이블 항목

### □ present/valid bit : 해당 페이지가 물리 메모리에 있는지 여부

- 이 비트가 1이면, 해당 페이지가 프레임 번호의 메모리에 있음
- 이 비트가 0이면, 해당 페이지가 디스크에 있음

### □ modified/dirty bit : 해당 페이지가 수정되었는지 여부

- 이 비트가 1이면, 해당 페이지가 프레임에 적재된 이후 수정되었음, 나중에 쫓겨날 때 스왑-아웃
- 이 비트가 0이면, 해당 페이지는 수정된 적이 없음. 나중에 쫓겨날 때, 스왑 영역에 저장될 필요 없음

### □ physical address

- present bit=1이면, 해당 페이지가 적재되어 있는 프레임 번호
- present bit=0이면, 해당 페이지가 있는 디스크 블록 번호

## □ 페이지 폴트(page fault)

### □ CPU가 액세스하려는 페이지가 물리 메모리에 없을 때, 페이지 폴트 발생

- 페이지 폴트가 일어나면 빈 프레임을 할당하고, 스왑 영역이나 실행 파일로부터 페이지 적재
- 스왑-인(swap-in) = page-in
  - 페이지를 스왑 영역에서 프레임으로 읽어 들이는(복사) 행위
- 스왑-아웃(swap-out) = page-out
  - 프레임에 저장된 페이지를 스왑 영역에 저장하고 프레임을 비우는 행위

# 요구 페이징 구성

프로세스 주소 공간



\*프로세스는 5개 페이지로 구성

	present 비트	modified 비트	physical address
0	1	0	1
1	1	0	5
2	0	0	disk block number
3	0	0	disk block number
...	...	...	...
m	1	0	f
...	...	...	...

페이지 테이블

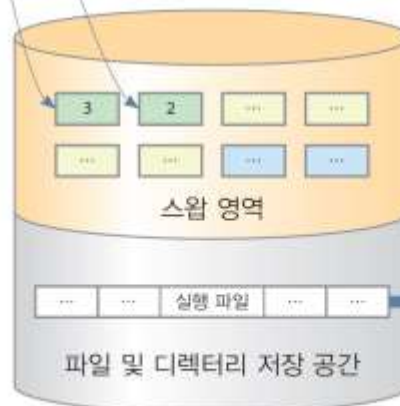
물리 메모리



충격

스와핑

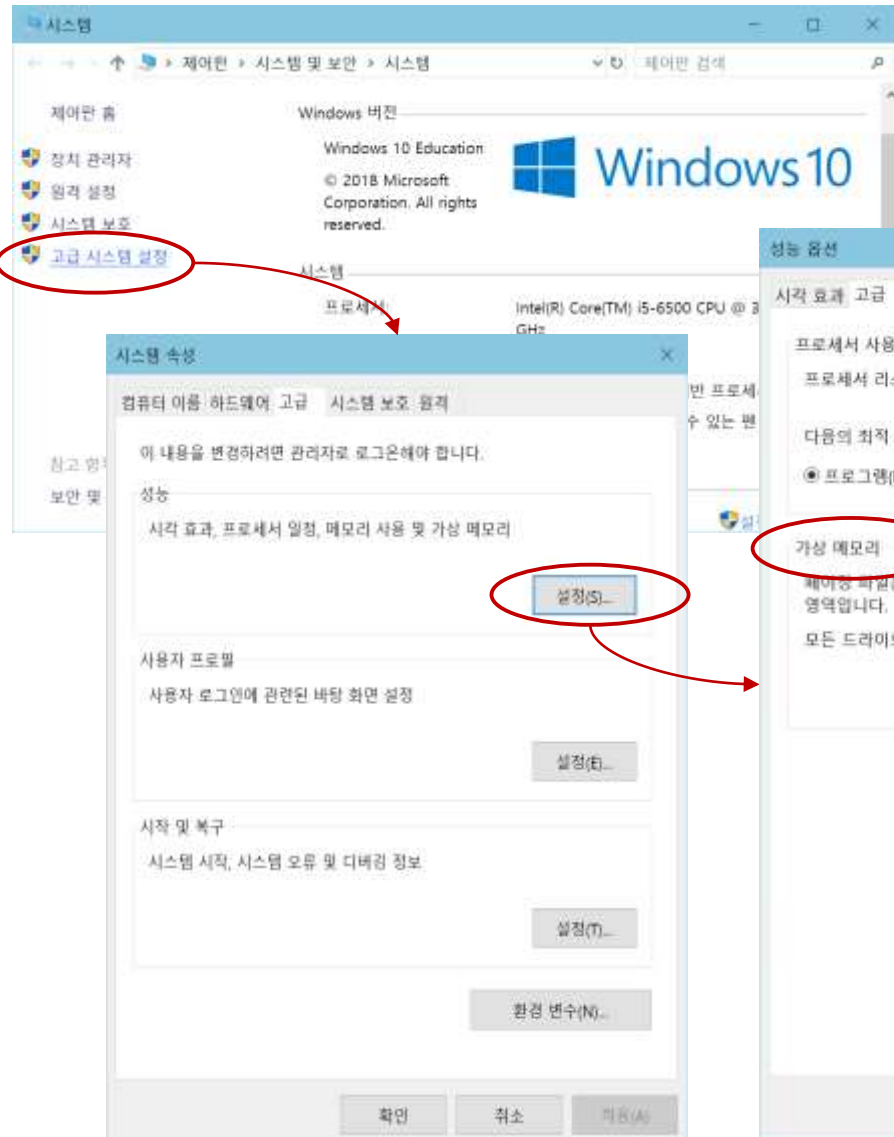
실행 파일로 부터 적재



하드 디스크

# Windows의 스왑 영역, C:\pagefile.sys

16



체크박스를 해제하면 사용자 지정 가능



현재 페이지 파일(C:/pagefile.sys)의 크기 4.8GB



# 페이지 폴트 자세히 알기(사례 중심으로)

17

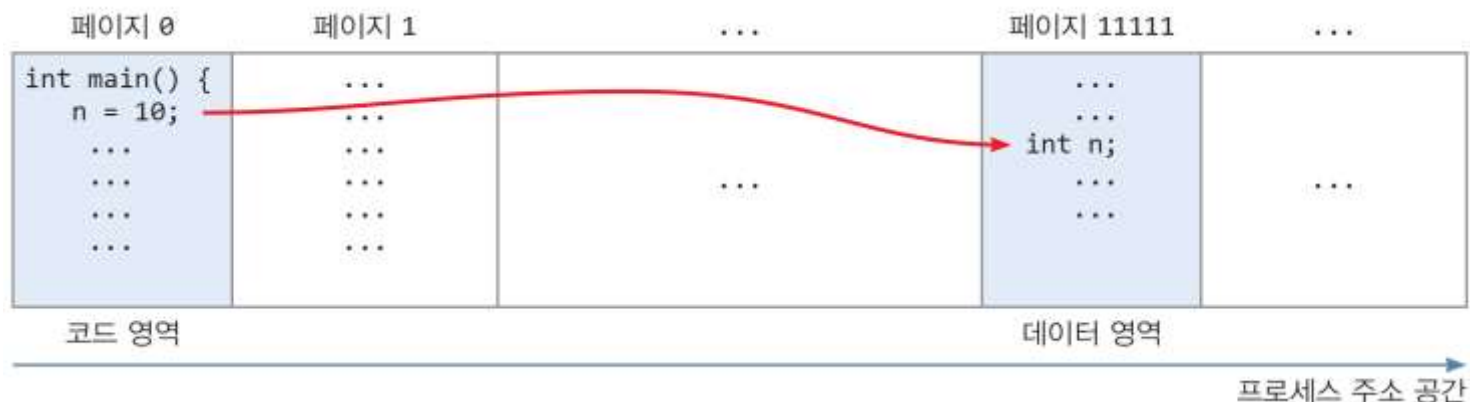
## □ 페이지 폴트

- 요구 페이지징에서 가장 중요한 상황
- CPU가 발생시킨 가상 주소의 페이지가 메모리 프레임에 없는 상황
- MMU가 가상 주소를 물리 주소로 바꾸는 과정에서 발생

## □ 사례

- main() 함수와 전역 변수 n을 가진 C 프로그램

- 전역 변수 n의 가상 주소는 0x11111234 -> 페이지 번호는 0x11111

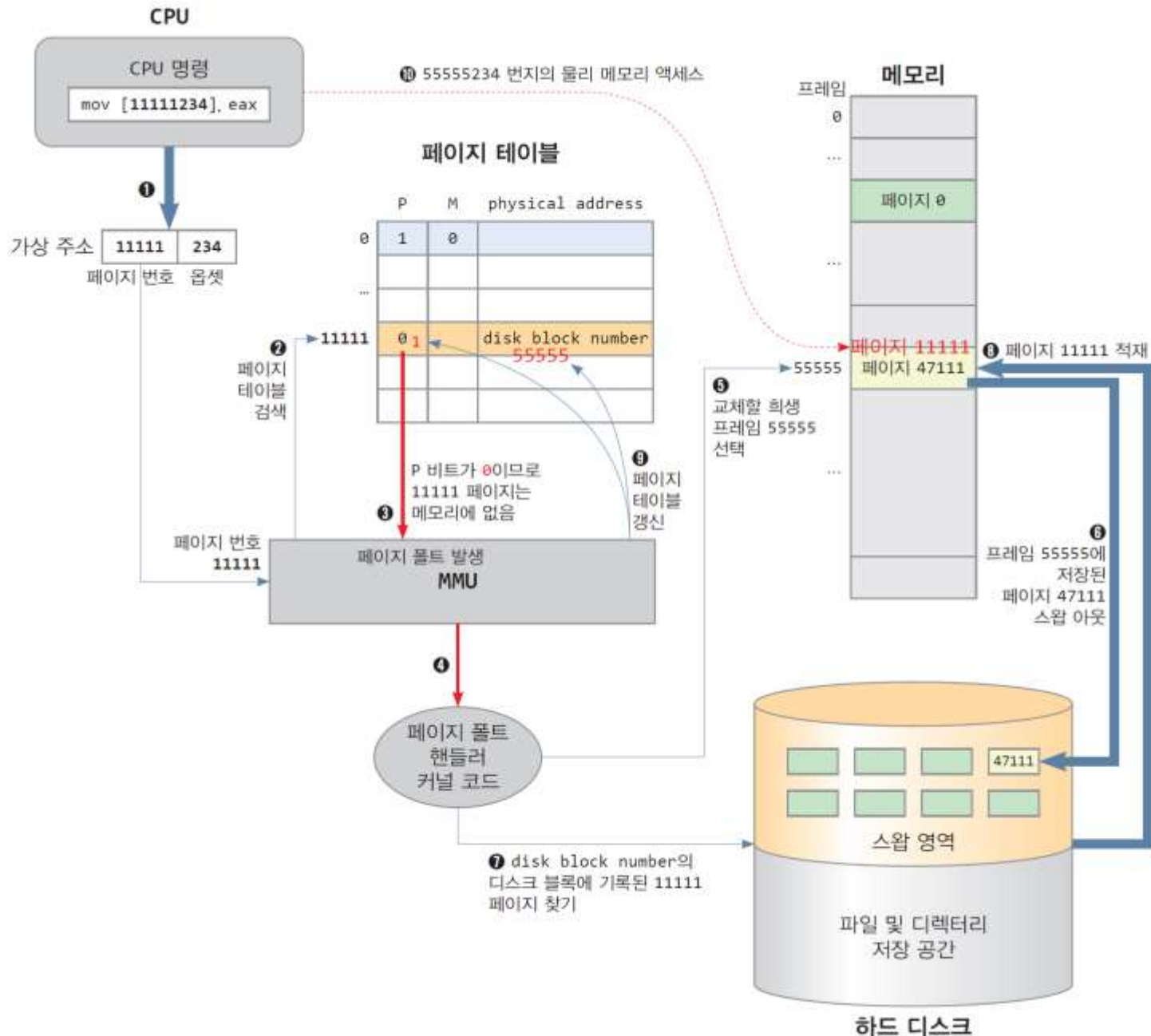


- n = 10의 컴파일된 기계어 코드

```
mov eax, 10 ; eax 레지스터에 10 저장
mov [11111234], eax ; eax 값을 0x11111234 번지에 저장
```

- mov [11111234], eax 명령을 실행하는 과정을 통해 페이지 폴트 설명

# 페이지 폴트가 발생하고 처리되는 과정

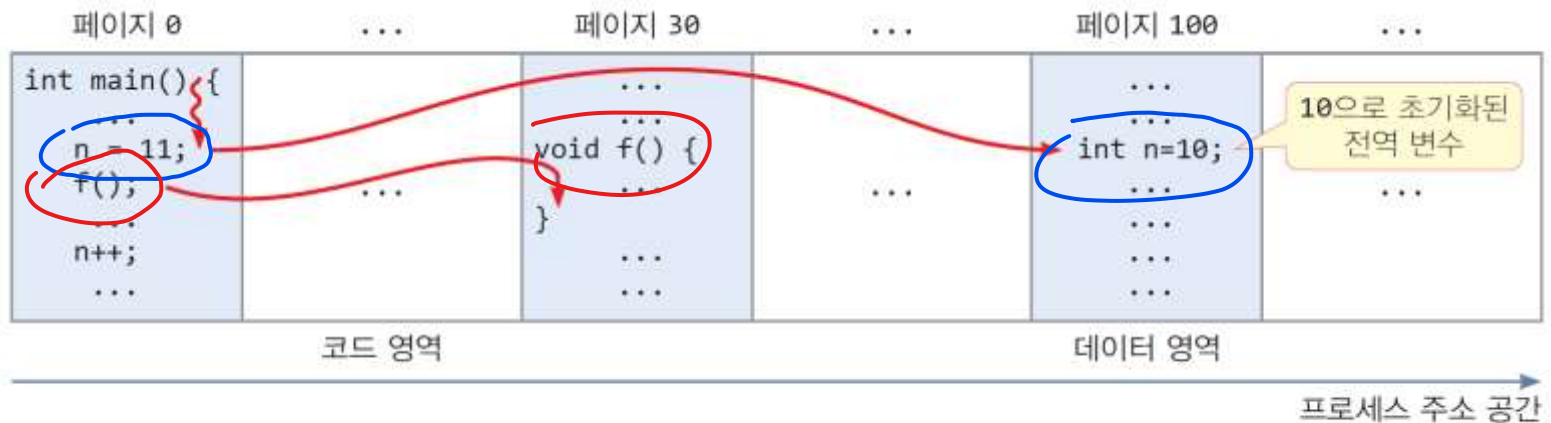


# 요구 페이징 시스템에서 프로세스 실행

19

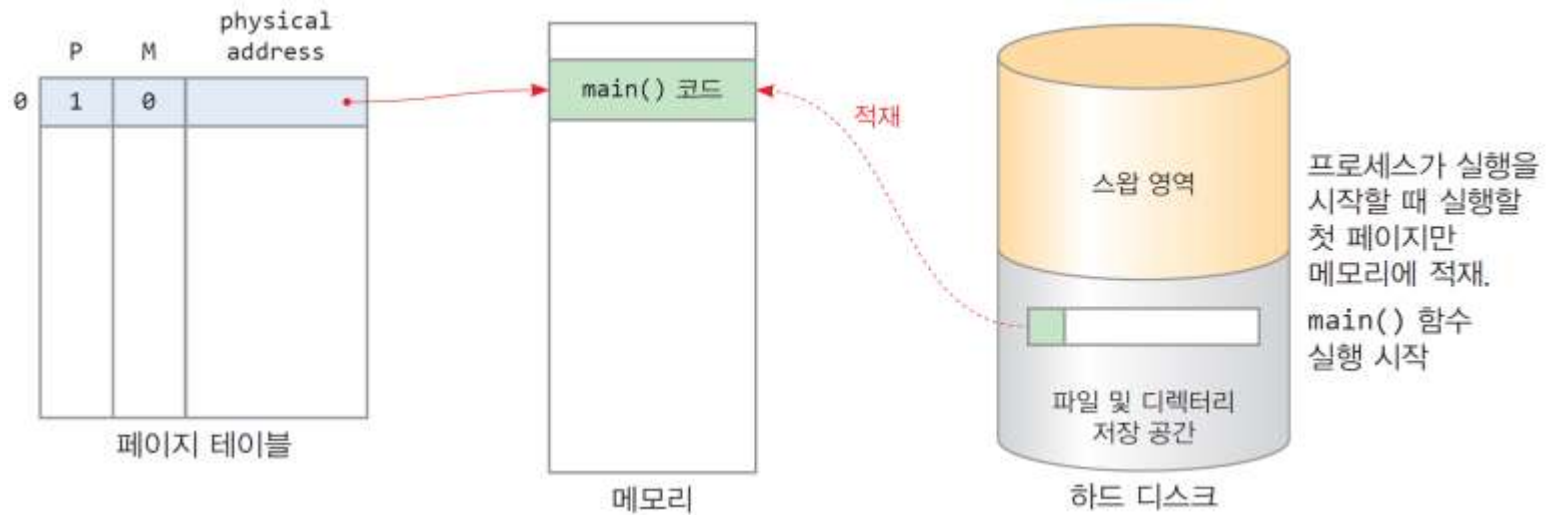
## □ 사례

- 요구 페이징에서 실행 파일로부터 프로세스가 적재되어 실행되는 전형적인 과정 설명
- 100개 이상의 페이지로 구성된 샘플 C 프로그램



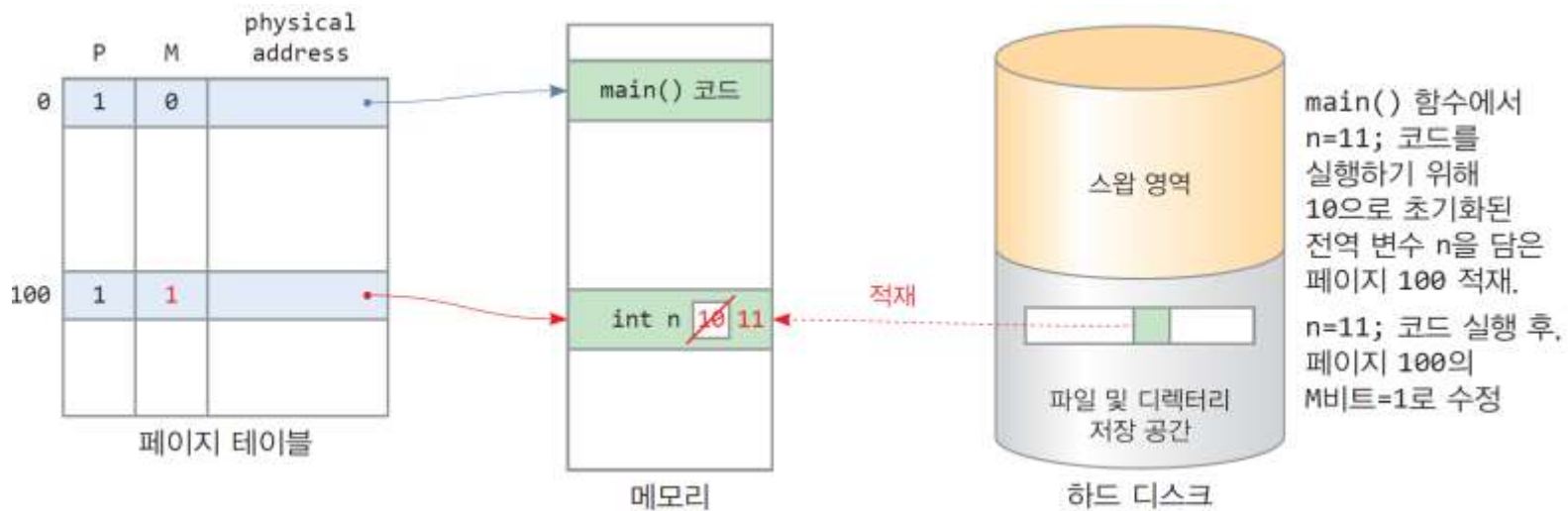
\* main() 함수가 실행되는 동안 페이지 0, 페이지 100, 페이지 30의 순서로 메모리가 필요함

## (1) 프로세스의 시작 페이지 적재



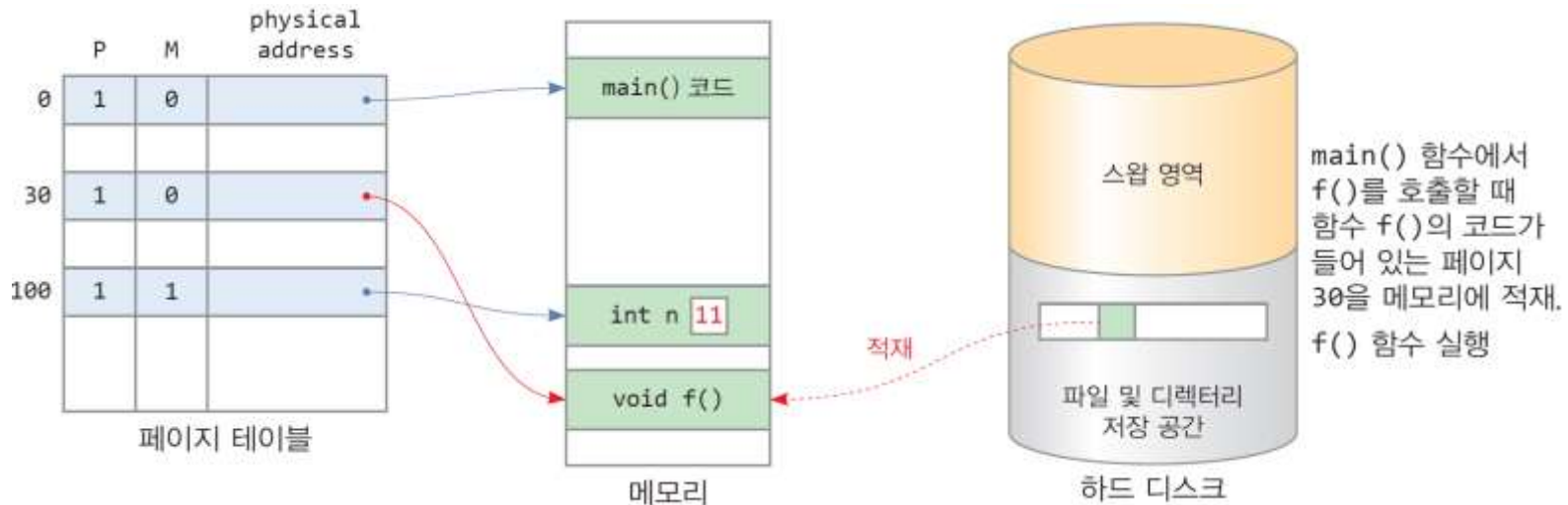
(a) 프로세스가 실행을 시작할 때 첫 페이지를 메모리에 적재

## (2) 여러 번의 페이지 폴트를 통해 실행 파일로부터 페이지들 적재



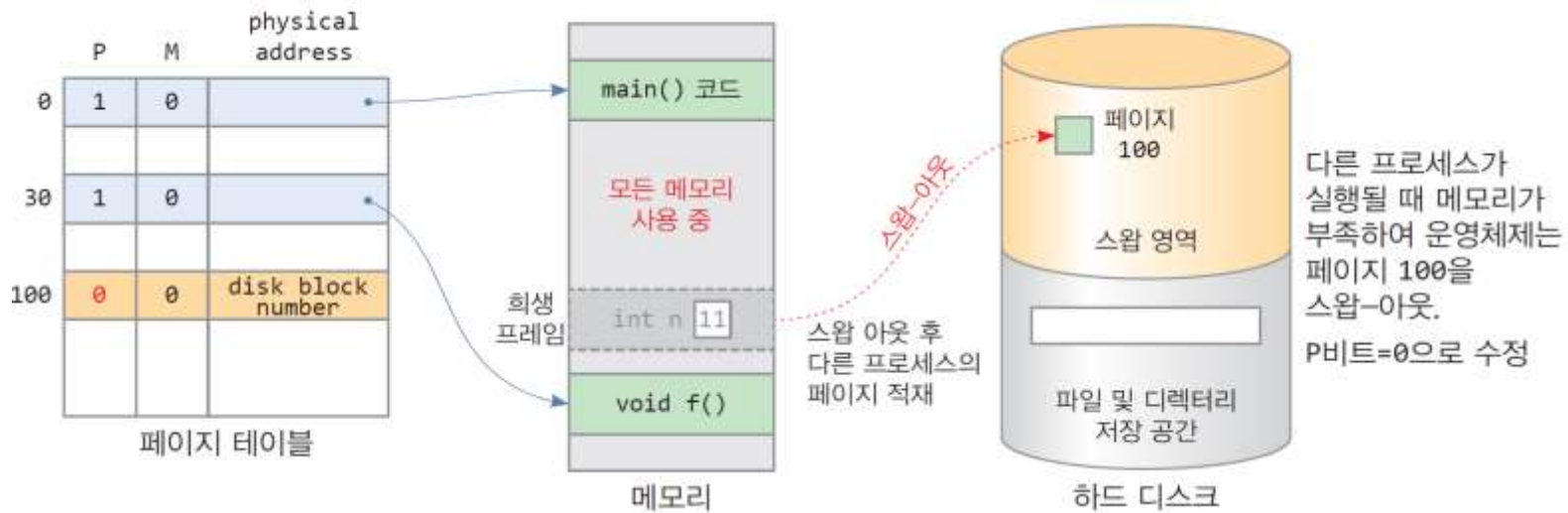
(b) 전역 변수 n이 들어 있는 페이지 100 적재 후, n=11 실행

- \* 폴트가 발생한 페이지는 '실행 파일' 이나 '스왑 영역' 의 2군데 중 한 곳에 있음
- \* 처음 액세스될 때 '실행 파일' 에서 적재



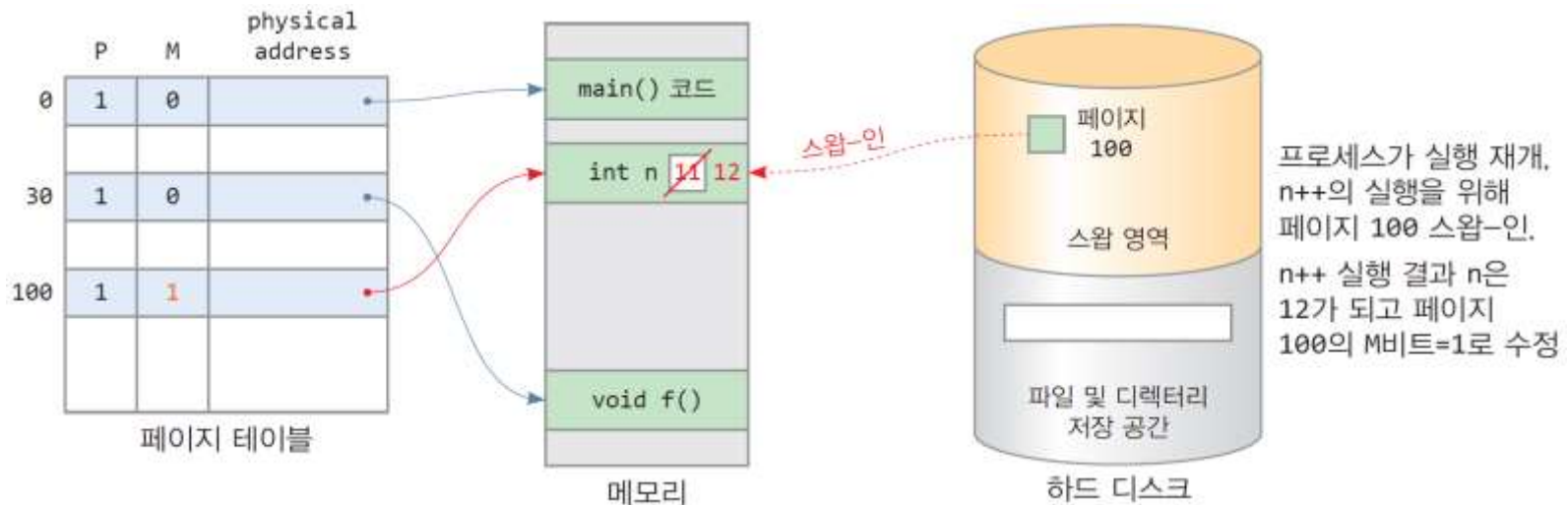
(c) 함수 f()의 코드가 들어 있는 페이지 30 적재

### (3) 메모리가 부족하면 스왑-아웃/스왑-인



(d) 메모리가 부족하여 전역 변수 n이 들어 있는 페이지 100의 스왑-아웃

### (4) 스왑-아웃된 페이지 100을 다시 스왑-인, 그리고 n++ 실행



(e) 페이지 100의 스왑-인 후 n++ 실행

#### (5) 수정된 페이지는 스왑 영역에 쓰기

시간이 흐르고 운영체제가 빈 프레임을 만들기 위해 페이지 30을 희생 페이지로 선택하였다면 다음과 같이 처리한다.

- 페이지 테이블 항목에 M비트=1이라면, 운영체제는 페이지 30이 들어 있는 프레임을 스왑-영역에 다시 기록한다.
- 하지만, M비트=0이라면(적재된 후 수정되지 않음) 스왑 영역에 저장할 필요가 없다. 그냥 페이지 30이 있었던 프레임에 다른 프로세스의 페이지를 적재하면 된다.

# 쓰기 시 복사(COW, copy on write)

24

## □ 프로세스의 생성

### ▣ 부모 프로세스가 fork() 시스템 호출로 생성

- 시스템 호출 함수는 운영체제마다 다름

### ▣ 프로세스 생성은 메모리 할당과 페이지의 메모리 적재에서 시작

## □ fork()는 자식 프로세스의 메모리를 어떻게 생성?

### ▣ (방법 1) 완전 복사

- 부모 프로세스의 모든 페이지를 완전히 복사
- 비효율적 - fork() 후 exec()하는 것이 일반적인 사례이기 때문

### ▣ (방법 2) 쓰기 시 복사

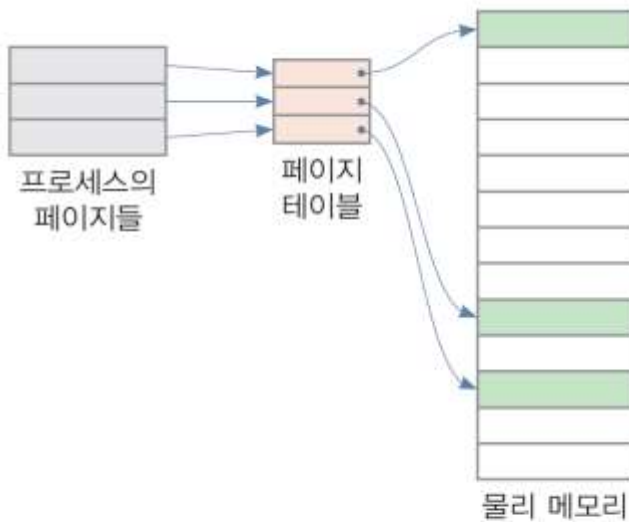
- 자식 프로세스를 위해 부모 프로세스의 페이지 테이블만 복사
- 그러므로, 초기에 자식 프로세스는 부모 프로세스의 메모리 프레임을 완전 공유
- 자식 프로세스의 페이지 테이블 항목에 '쓰기 시 복사' 표시 884p
- 자식이나 부모 중 누군가 페이지를 수정할 때, 새로운 프레임을 할당 받아 공유하고 있는 부모 프레임을 복사



# 완전 복사

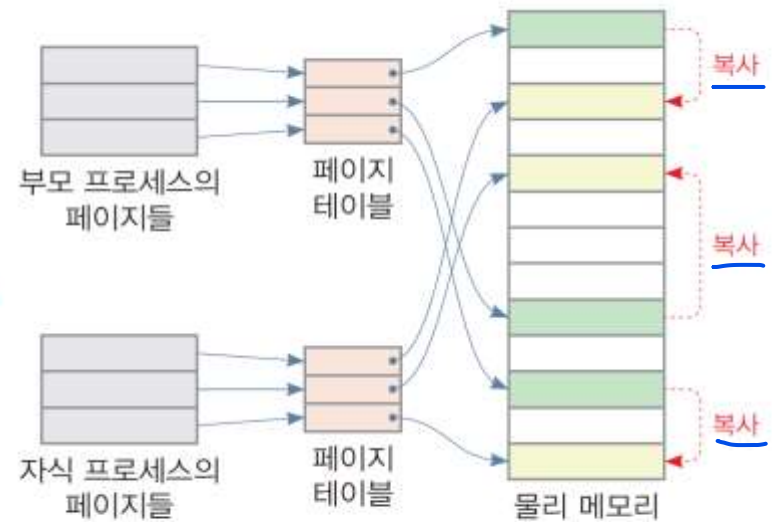
25

- 부모 프로세스의 메모리를 완전히 복사하여 자식 프로세스 생성



(a) 부모 프로세스

fork()



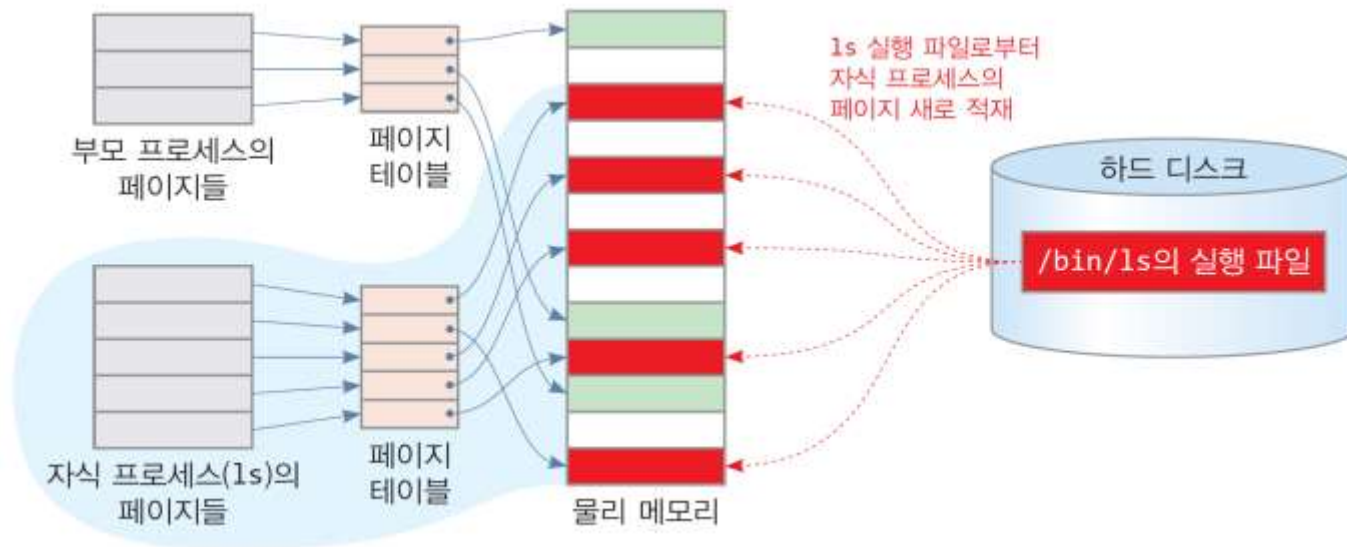
(b) 완전 복사 후 부모와 자식 프로세스의 메모리 프레임

# 완전 복사의 비효율성

26

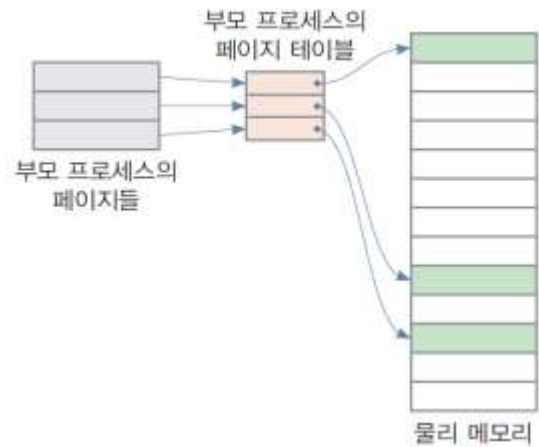
- 많은 응용프로그램들이, `fork()` 후 생성된 자식프로세스가 `execvp()`를 호출하여 곧 바로 다른 프로그램을 실행하도록 작성되기 때문
  - `fork()`가 완전 복사를 하면, 완전 복사로 인한 괜한 시간 낭비

```
int childPid = fork(); // 쉘을 복제한 자식 프로세스 생성
if(childPid == 0) { // 자식 프로세스 코드
    execvp("/bin/ls", "ls", NULL); // /bin/ls 파일을 적재하여 실행
}
```

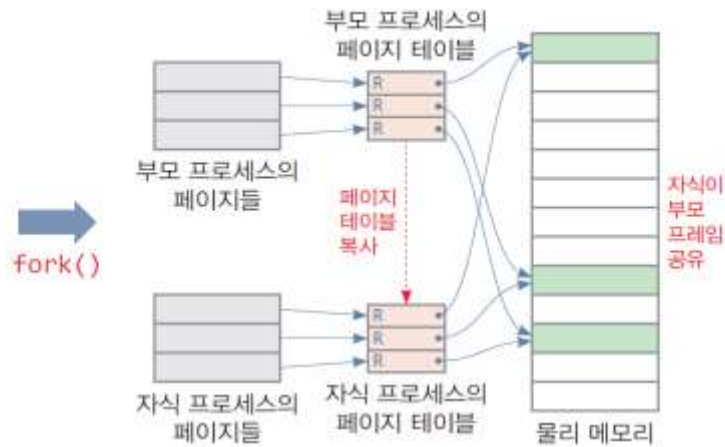


`execvp()`에 의해 자식 프로세스의 메모리가 모두 반환되고 실행파일 ls로부터 새로 페이지 적재

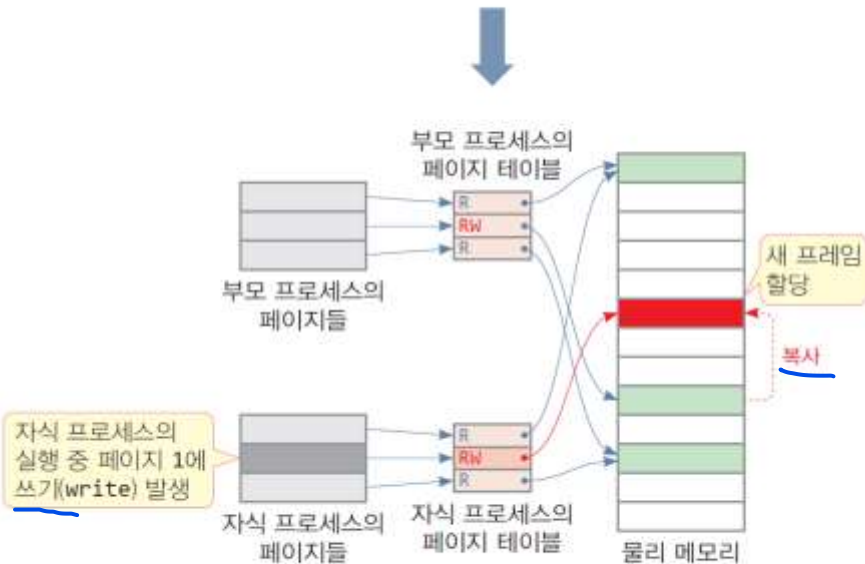
## 쓰기 시 복사(COW)로 자식 프로세스를 생성하는 과정



(a) 부모 프로세스



(b) '쓰기 시 복사' 후 자식 프로세스는 부모의 메모리 공유



(c) 자식 프로세스가 페이지 1에 쓰기를 실행할 때, 운영체제는 새 프레임을 할당하여 쓰기가 발생한 페이지 1을 복사

# 쓰기 시 복사의 장점

28

## □ 프로세스 생성 시간 절약

- ▣ 부모 프로세스의 페이지 테이블만 복사하여 자식 프로세스를 만들기 때문에 프로세스 생성이 매우 빠름
- ▣ fork()후 exec()해도 프로세스의 생성 과정에서 손해나는 것 없음

## □ 메모리 절약

- ▣ 부모와 자식 프로세스가 둘 다 읽기만 하는 페이지는 새로운 프레임 할당할 필요가 없으므로 메모리 절약
  - 예) 프로세스의 코드 페이지. 코드와 같이 읽기 용 페이지 프레임은 자동 공유

# 탐구 10-1 요구 페이지징에 대해 생각해 볼 이슈

29

**Q1. 페이지 폴트는 필연적으로 페이지 폴트를 동반한다. 그런데 페이지 폴트가 빈번하게 발생하면 디스크와 메모리 사이의 빈번한 입출력으로 인해 시스템 성능이 떨어지지 않을까?**

A. 그런 경우를 스래싱(thrashing)이라고 부르고 발생할 수도 있다. 프로그램이 실행되는 초기에는 페이지 폴트가 계속 발생하겠지만, 얼마 지나지 않아 필요한 페이지들이 메모리에 올라오게 되어 그 이후에는 간헐적으로 페이지 폴트가 발생한다.

**Q2. 프로세스의 실행 동안 페이지 폴트가 계속되면 언젠가 메모리에는 그 프로세스의 많은 페이지들이 존재하게 될 텐데 왜 처음부터 이들을 적재하지 않는가?**

A. 프로세스의 실행이 금방 종료될지 오랜 후에 종료될지 모르고, 모든 페이지가 다 사용될 지 일부 페이지만 사용될 지 모르기 때문에, 처음부터 모든 페이지를 적재하는 것은 메모리 낭비

**Q3. 한 프로세스에게 할당할 수 있는 메모리 프레임은 무한정인가 아니면 제한적인가?**

A. 일반적으로 한 프로세스에게 할당되는 메모리 프레임의 최대 개수는 제한된다. 운영체제는 메모리량의 한계 때문에 프로세스가 필요로 하는 모든 페이지에 대해 메모리 프레임을 할당할 수 없다.

**Q4. 한 프로세스에게 할당하는 프레임의 수와 페이지 폴트의 관계는?**

A. 한 프로세스에게 할당하는 프레임의 개수가 많을수록 페이지 폴트의 횟수가 작아진다.

**Q5. 페이지 폴트를 처리하는 과정에서 커널 코드나 커널 데이터가 적재된 메모리 프레임도 스왑-아웃되는 희생 프레임으로 선택되는가?**

A. 오늘날 대부분의 운영체제는 커널 코드와 커널 데이터가 적재된 프레임을 희생 프레임으로 선택하지 않는다. 즉 스왑-아웃되지 않는다. 만일 인터럽트 핸들러가 스왑-아웃되어 메모리에 없다면, 인터럽트가 발생하였을 때 인터럽트 핸들러가 실행될 수 없거나 실행되는데 매우 긴 시간이 걸릴 것이다.

# 페이지 폴트와 스래싱(thrashing)

30

- 페이지 폴트와 디스크 I/O
  - ▣ 페이지 폴트가 발생하면 필연적으로 디스크 I/O 증가
- 스래싱(Thrashing or Disk Thrashing)

스래싱은 페이지 폴트가 계속 발생하여, 메모리 프레임에 페이지가 반복적으로 교체되고, 디스크 입출력이 심각하게 증가하고, CPU 활용율이 대폭 감소하는 현상

스래싱은 빈번한 페이지 폴트로 인한 디스크 입출력 증가 현상

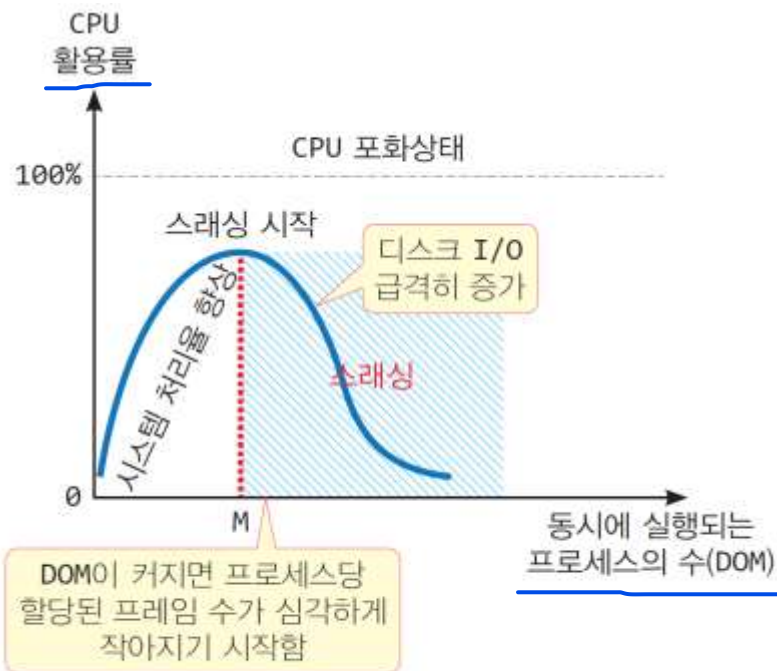
- ▣ 한 프로세스의 페이지를 적재하기 위해 다른 프로세스가 사용중인 페이지를 스왑-아웃시키는 일이 도미노처럼 발생
  - ▣ 연속되는 페이지 폴트가 처리되는 동안 스레드는 대부분의 시간을 대기하면서 보냄
- 스래싱 원인
    1. 다중프로그래밍 정도(degree of multiprogramming )가 과도한 경우
      - 메모리에 비해 너무 많은 프로세스가 실행되어,
      - 프로세스 당 할당되는 프레임 개수가 적을 때,
      - 프로세스가 필요한 충분한 페이지가 적재되지 못하여 페이지 폴트 발생
    2. 잘못된 메모리 할당/페이지 교체 알고리즘
    3. 기본적으로 메모리 량이 적을 때
    4. 우연히도 특정 시간에 너무 많은 프로세스 실행

# 스래싱 현상 관찰

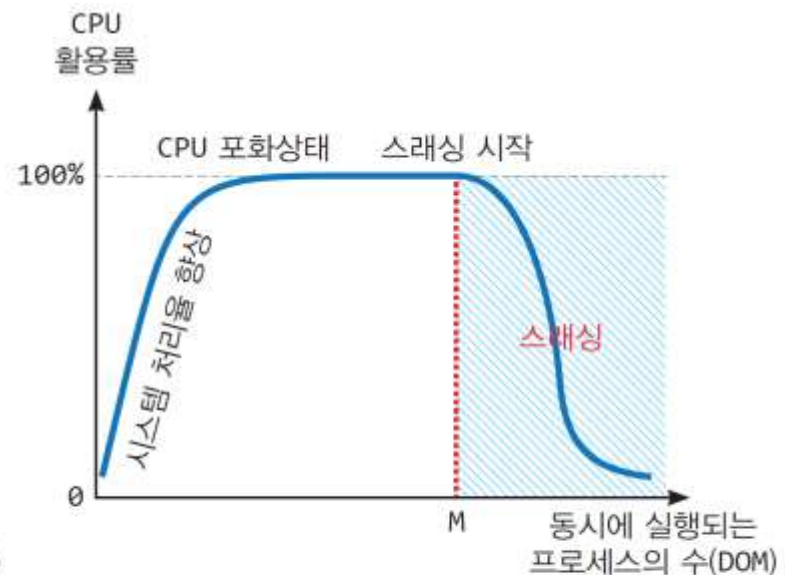
31

## □ 스래싱이 발생하는 시점

- 다중프로그래밍 정도(DOM)가 높아질수록 자연스러운 CPU 활용률 증가
- 다중프로그래밍 정도(DOM)가 임계점 M을 넘어가면 스래싱 발생
  - CPU 활용률이 급감하고 I/O비율 급상승



(a) 메모리가 작은 경우



(b) 메모리가 많은 경우

# 스래싱 해결 및 예방

32

- 스래싱 감지 방법
  - ▣ 운영체제에 따라 다름
  - ▣ 윈도우 : process explorer 등 사용
  - ▣ 리눅스 : top나 htop, vmstat 명령을 통해 작업 부하는 높지만 CPU 활용률이 낮고, 스왑-인/아웃이 모두 높은지 검사
- 해결 및 예방
  - ▣ 다중프로그래밍 정도(DOM) 줄이기
    - 몇몇 프로세스 종료
  - ▣ 하드 디스크 대신 빠른 SSD 사용
  - ▣ 메모리 늘리기

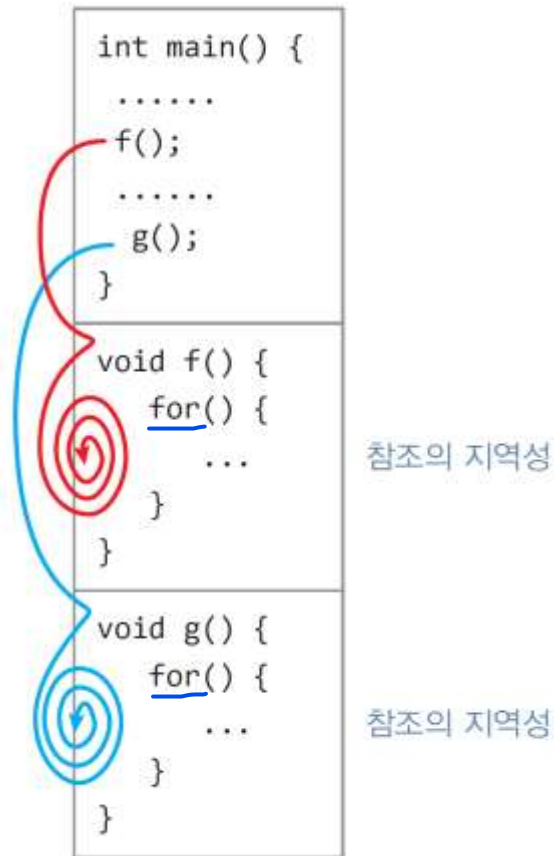


## 4. 참조의 지역성과 작업 집합

# 프로그램의 실행 특성

34

- 참조의 지역성(reference of locality)
  - ▣ CPU가 짧은 시간 범위 내에 일정 구간의 메모리 영역을 반복적으로 참조하는 경향
  - ▣ 참조의 지역성 특징
    - 모든 프로그램에서 나타나는 기본적인 실행 특성
    - 메모리를 균일하게 액세스하지 않고, 짧은 시간에 특정 부분을 집중 참조
    - locality(지역성), principle of locality(지역성의 원리)
    - 프로세스가 최근에 참조한 데이터와 코드를 다시 참조하는 경향성
    - 참조의 지역성 이동. 프로세스가 실행되는 동안 메모리 영역을 옮겨 다니면서 참조의 지역성이 나타남
    - 90/10 규칙 - 경험적 관찰에서 나온 것으로, "프로그램 코드의 10 %에서 실행 시간의 90 % 소비"
- ▣ 의미
  - 현재 프로세스의 실행 패턴을 관찰하면, 가까운 미래에 프로세스의 코드와 데이터 사용을 합리적으로 예측 -> 메모리 할당과 페이지 교체 전략에 활용



# 참조의 지역성 형태(type)

35

## □ 대표적인 참조의 지역성 형태

### ▣ 시간 지역성(Temporal Locality)

- 시간적으로 볼 때,
- 프로세스에서 지금 참조된 주소(혹은 페이지)가 가까운 미래에 다시 참조될 가능성이 큰 특성
- 코드나 데이터, 자원 등이 아주 짧은 시간 내에 다시 사용되는 특성
- 대표 사례 : 반복문

### ▣ 공간 지역성(Spatial Locality)

- 공간적(메모리 주소 면에서)으로 볼 때
- 지금 참조되는 번지(혹은 페이지)의 주변 번지들이 가까운 미래에 참조되는 특성
- 대표 사례 : 배열 사용, 순차 읽기/순차 쓰기, 코드에서 100번지를 액세스하고 있다면 다음에 104번지를 실행할 가능성 매우 높음

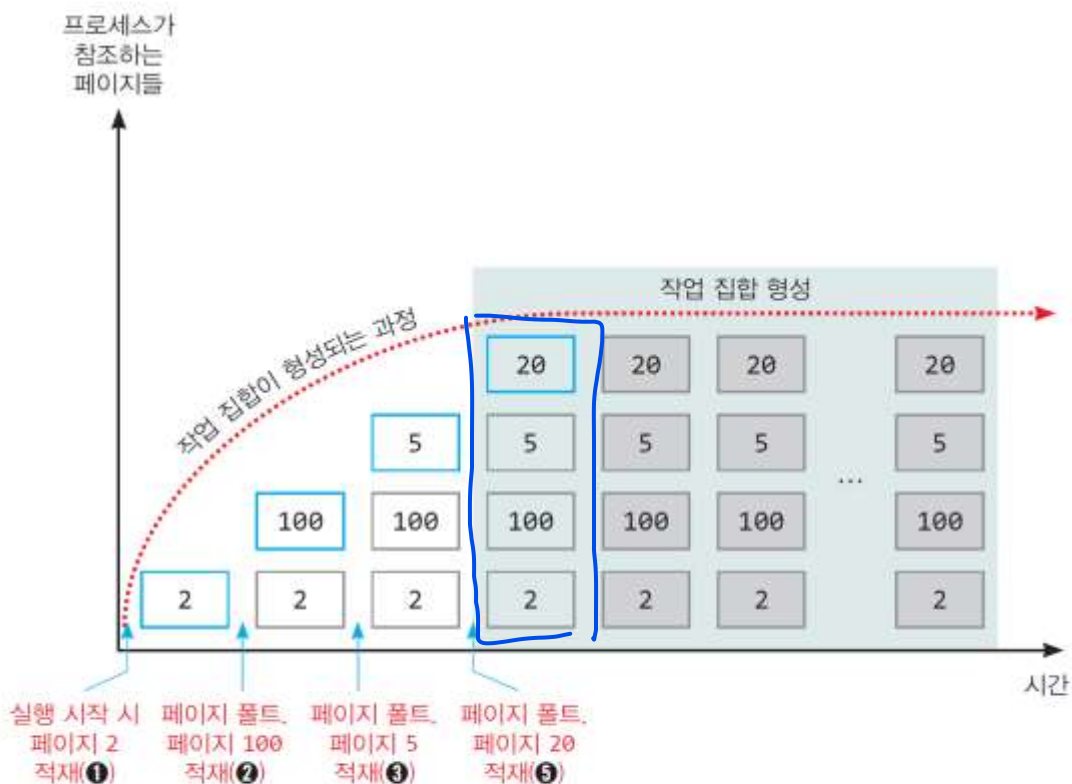
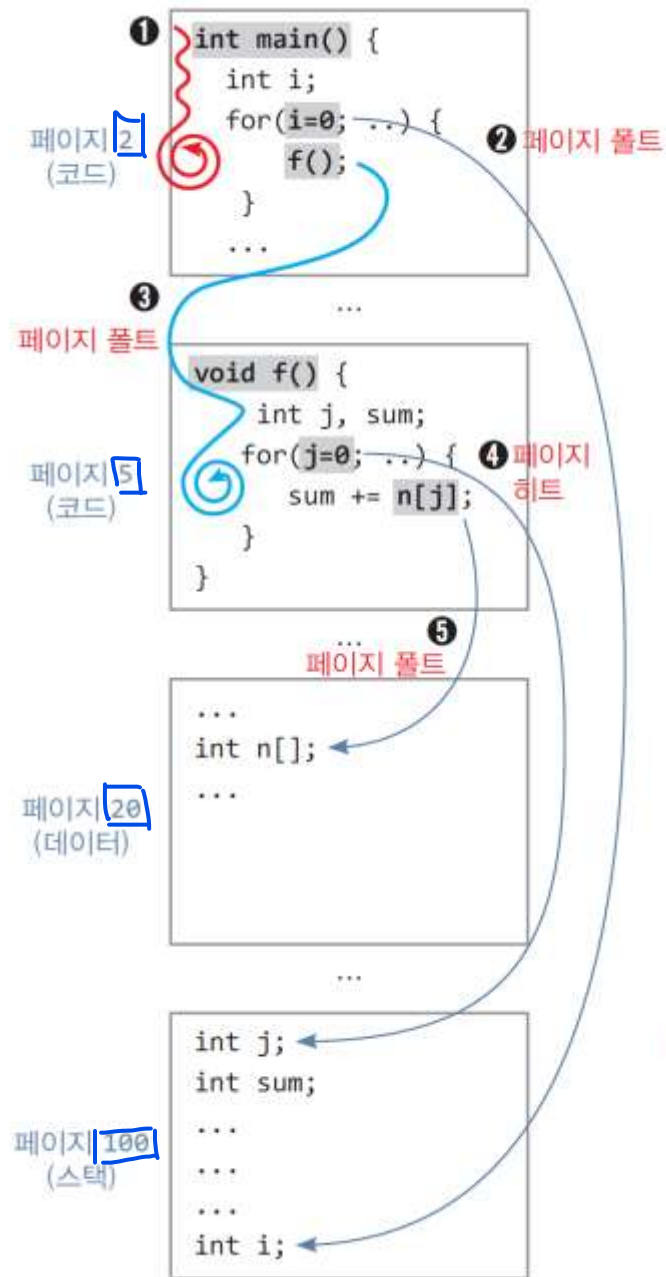
# 작업 집합과 페이지 폴트, 스래싱

36

## □ 작업 집합(working set)

- ▣ 일정 시간 범위 내에 프로세스가 액세스(참조)한 페이지들의 집합
  - 작업 집합에 포함된 페이지들이 모두 메모리에 적재되어 있는 것이 프로세스 실행의 최고 성능
  - 시간 범위가 클수록 작업 집합도 큼. 시간 범위를 얼마로 정할 것인가?
- ▣ 참조의 지역성으로 인해 일정 시간 내에 작업 집합이 뚜렷하게 형성
  - 프로세스의 실행 중 갑자기 **페이지 폴트**가 계속되면, **작업 집합**을 메모리에 적재하는 과정임 -> 시간이 지나면 페이지 폴트가 줄고 작업 집합이 뚜렷이 형성됨(다음 슬라이드)

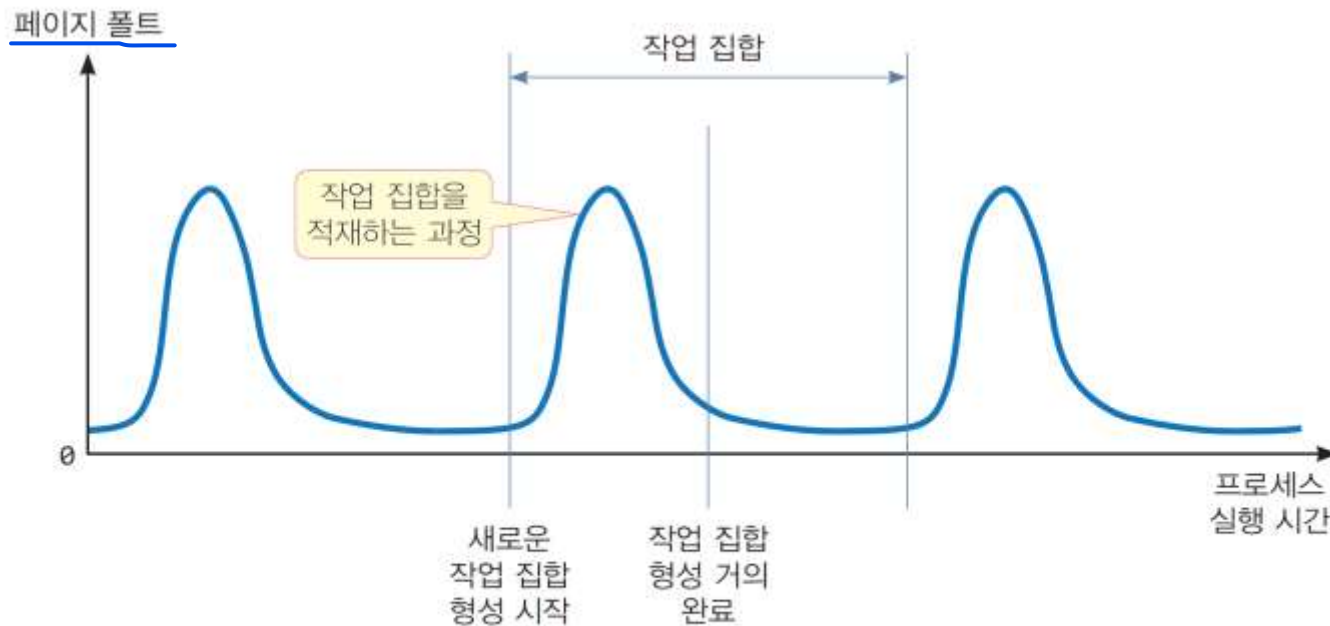
## 작업 집합이 형성되는 과정



# 작업 집합 이동

38

- 작업 집합 이동(working set shift)
  - ▣ 프로세스가 실행되는 동안 계속 작업 집합 이동
    - 시간이 지나면 새로운 작업 집합 형성



※ 새로운 작업 집합이 형성되는 과정에서 페이지 폴트가 급격히 발생하지만 곧 줄어들어 안정 상태가 된다.

# 스래싱과 작업 집합

39

- 스래싱 발발 원인 설명에 작업 집합 모델 활용
  - ▣ 1968년 Denning의 논문(Thrashing : its causes and prevention)
    - 스래싱의 원인 설명에 ‘작업 집합’ 모델 사용
    - 처음 스래싱 관측 - 1960년대 다중프로그래밍 시스템에서 갑자기 처리율이 떨어지는, 예측하지 못한 현상 발견됨
    - 1968년, 처음으로 작업 집합 모델로 스래싱 현상 설명
- 스래싱 예방
  - ▣ 스래싱은 작업 집합이 메모리에 올라와 있지 않을 때 발생
  - ▣ 1968년 Denning
    - 실험으로 증명
    - 예방책 제시 - 각 프로세스에게 작업 집합에 포함하는 페이지들을 적재할 충분한 메모리 할당

# 요구 페이지징의 필수 알고리즘 2개

40

## □ 요구 페이지징의 성능에 영향을 미치는 필수 알고리즘

### 1. 프레임 할당(frame allocation) 알고리즘

- 프로세스당 할당할 프레임 개수를 결정하는 문제
- 알고리즘의 목표
  - 프로세스의 작업 집합에 포함될 페이지들을 수용할만한 개수의 프레임 할당 - > 페이지 폴트를 줄이기 위해

### 2. 페이지 교체(page replacement) 알고리즘

- 페이지 폴트가 발생하였을 때, 빈 프레임이 없는 경우 희생 프레임(비울 프레임)을 결정하는 문제
- 알고리즘의 목표
  - 작업 집합에 속하지 않은 페이지가 담긴 프레임을 선택하여 미래에 사용될 페이지가 교체되지 않도록 유지



## 5. 프레임 할당

# 프레임 할당

42

## □ 프레임 할당의 목표

- ▣ 프로세스에게 작업 집합에 포함된 페이지들을 적재할 충분한 메모리 할당
  - 페이지 폴트를 줄이고 스래싱 예방

## □ 2가지 방법

### ▣ 균등 할당(equal allocation)

- 프로세스에게 크기와 관계없이 동일한 개수의 프레임 할당
- 장점 - 단순
- 단점 - 작은 프로세스에는 프레임 낭비,  
큰 프로세스에는 빈번한 페이지 폴트 발생 가능

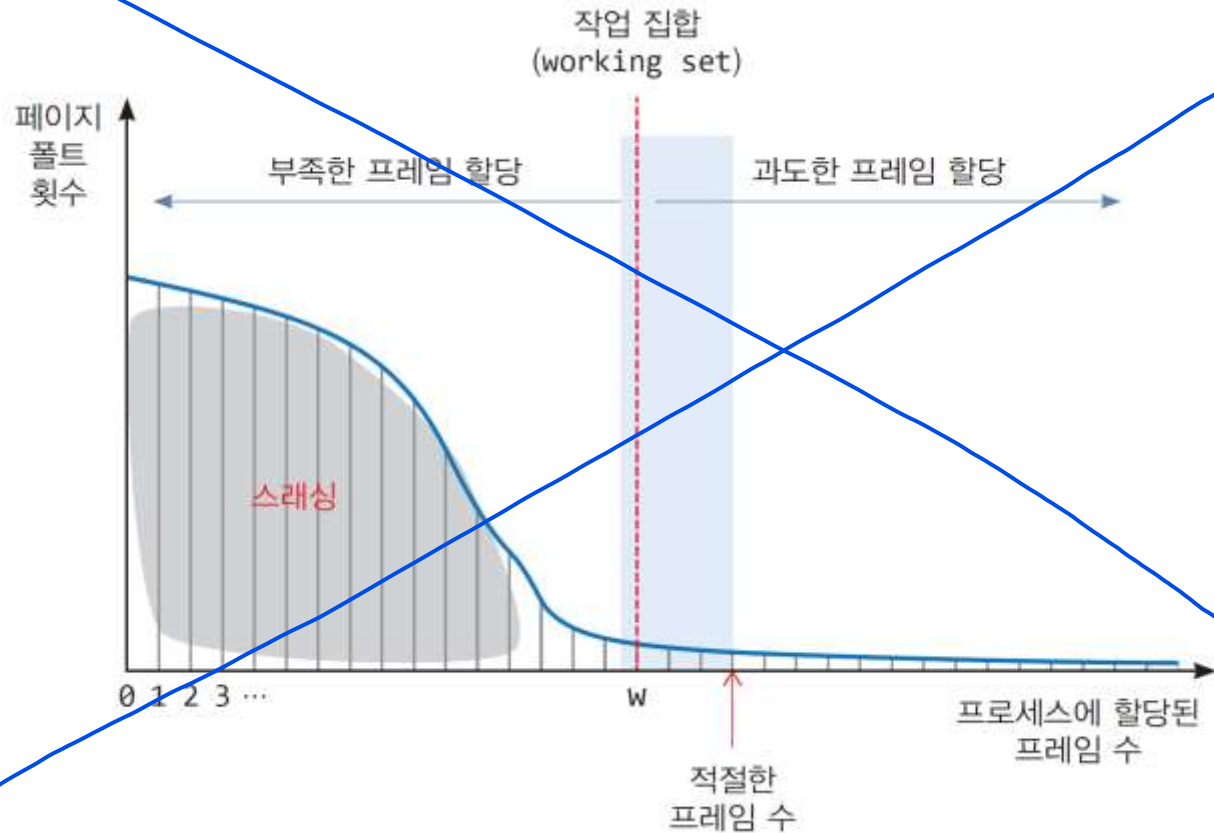
대부분 →

### 비례 할당(proportional allocation)

- 프로세스 크기에 비례하여 프레임 할당
- 장점 - 많이 필요한 프로세스에게 많은 프레임을 할당함으로써,  
전체적으로 페이지 폴트의 수를 줄임
- 단점 - 실행 전에 프로세스 크기를 완벽히 알지 못함.  
실행 중에 작업 집합을 판단할 필요 있음(Windows 사례 참고)

# 프로세스에게 할당할 적정 프레임의 개수 - 작업 집합을 약간 넘나드는 크기

43



# Tip. Windows의 프레임 할당(작업집합관리)사례

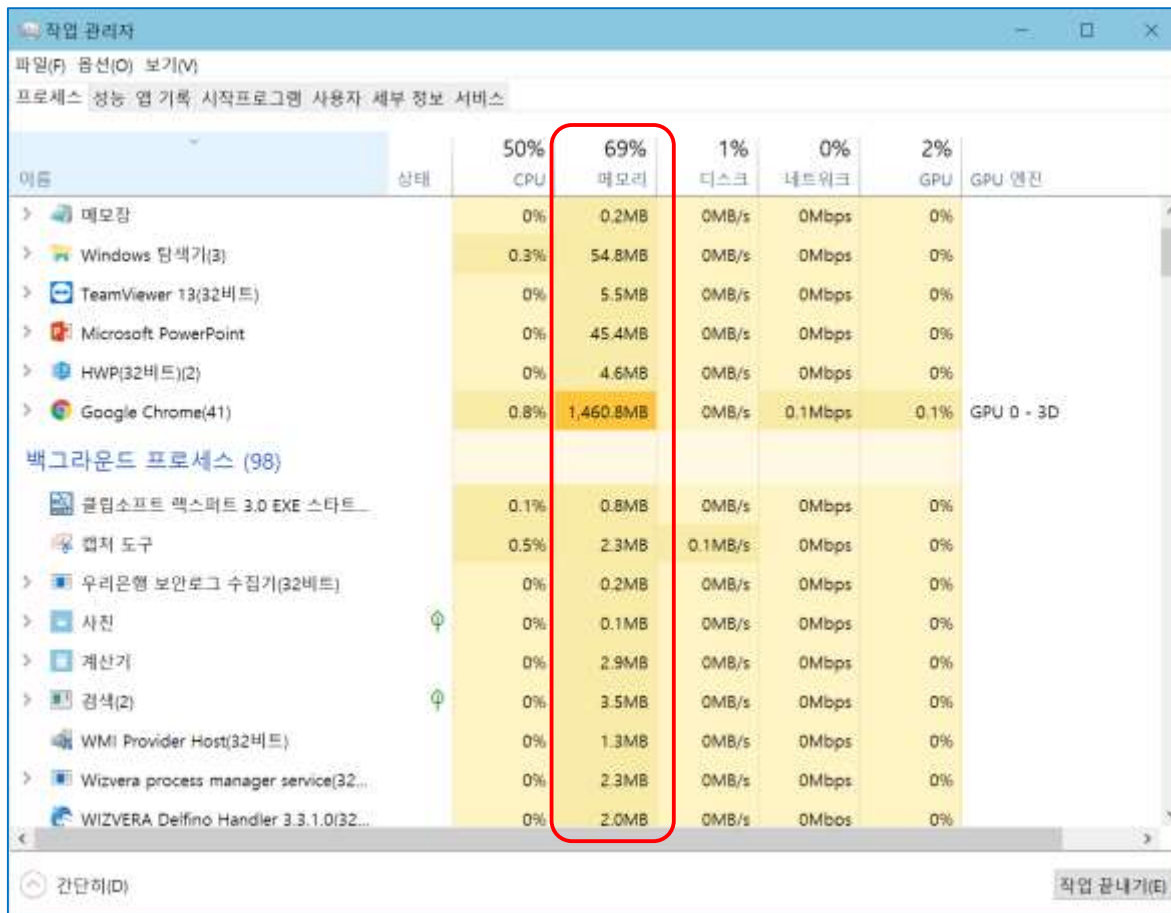
44

- 작업집합의 이동으로 프로세스의 작업집합 크기를 정확히 알기 어려움
- Windows는 프로세스 생성시 최소, 최대 할당 프레임 수를 정함
  - ▣ Windows 7에서는 최소 50개, 최대 345개
  - ▣ Windows에서는 이를 워킹셋이라고 부름
  - ▣ 프로세스가 생성될 때
    - 최소/최대 워킹셋 수가 예약되고,
    - 프로세스가 실행되면, 페이지 폴트 시 최소 수까지 메모리 할당 노력
    - 최대 수를 넘어서는 메모리 할당 불허
    - 하지만, 가용 메모리가 충분하면, 프로세스에게 최대치를 넘어서도 할당
  - ▣ 주기적으로
    - working set trimming algorithm으로 시스템 메모리 사용량을 스캔
    - 일정수준 이상 시스템 메모리가 사용되고 있으면, 프로세스들의 워킹셋(할당된 프레임 수)을 줄임 -> 스왑-아웃시킴
    - Windows가 정한 가용 메모리를 확보할 때까지 계속됨
  - ▣ 프로세스가 스스로 작업 집합 크기를 변경할 수 있는 시스템 호출 제공
    - SetProcessWorkingSetSize(), SetProcessWorkingSetSizeEx()

# Windows 작업 관리자, 메모리 사용량 의미

45

- ▣ 메모리에 적재된 프로세스의 페이지 수를 나타냄
  - 프로세스의 크기가 아님, 프로세스의 모든 페이지가 적재된 것도 아님
  - 현재 프로세스의 워킹셋이라고 부름



이름	상태	50% CPU	69% 메모리	1% 디스크	0% 네트워크	2% GPU	GPU 엔진
> 메모장		0%	0.2MB	0MB/s	0Mbps	0%	
> Windows 탐색기(3)		0.3%	54.8MB	0MB/s	0Mbps	0%	
> TeamViewer 13(32비트)		0%	5.5MB	0MB/s	0Mbps	0%	
> Microsoft PowerPoint		0%	45.4MB	0MB/s	0Mbps	0%	
> HWP(32비트)(2)		0%	4.6MB	0MB/s	0Mbps	0%	
> Google Chrome(41)		0.8%	1,460.8MB	0MB/s	0.1Mbps	0.1%	GPU 0 - 3D
백그라운드 프로세스 (98)							
클립소프트 액스퍼트 3.0 EXE 스타트...		0.1%	0.8MB	0MB/s	0Mbps	0%	
앱치 도구		0.5%	2.3MB	0.1MB/s	0Mbps	0%	
> 우리은행 보안로그 수집기(32비트)		0%	0.2MB	0MB/s	0Mbps	0%	
> 사진		0%	0.1MB	0MB/s	0Mbps	0%	
> 계산기		0%	2.9MB	0MB/s	0Mbps	0%	
> 검색(2)		0%	3.5MB	0MB/s	0Mbps	0%	
WMI Provider Host(32비트)		0%	1.3MB	0MB/s	0Mbps	0%	
> Wizvera process manager service(32...		0%	2.3MB	0MB/s	0Mbps	0%	
WIZVERA Delfino Handler 3.3.1.0/32...		0%	2.0MB	0MB/s	0Mbps	0%	

Window 탐색기의 경우,

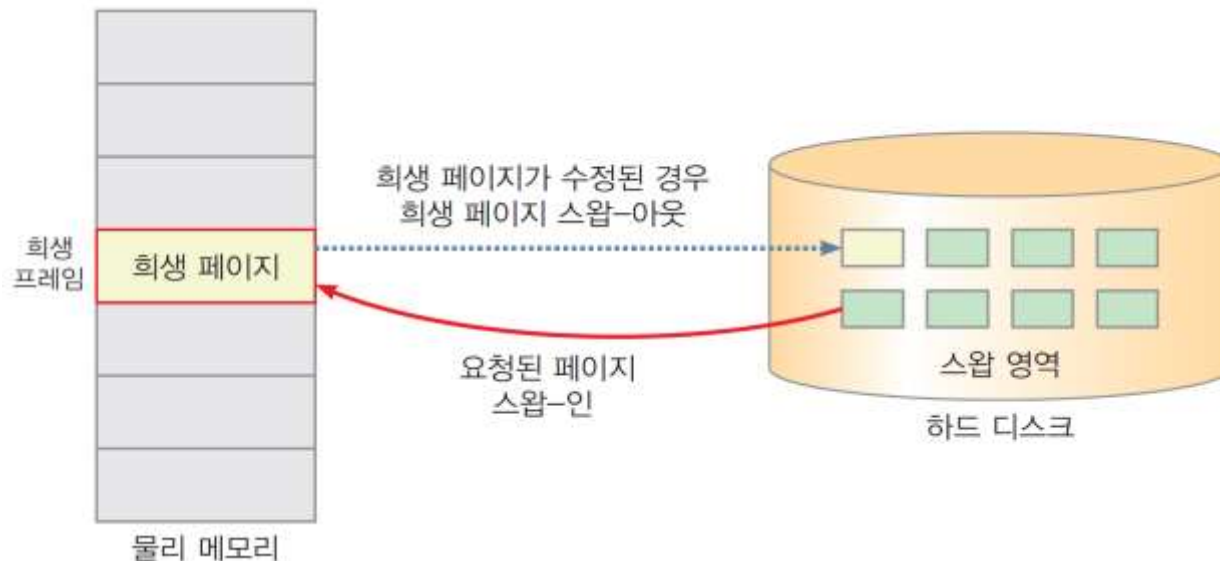
- 현재 사용중인 메모리는 54.8 MB
- Window 탐색기의 현재 작업 집합
- 작업 집합의 크기 :  
 $54.8\text{MB} / 4\text{KB} = 14029\text{개의 페이지}$

## 6. 페이지 교체

# 페이지 교체

47

- 페이지 교체(page replacement)란
  - ▣ 메모리 프레임 중 하나를 선택하여 비우고 이곳에 요청된 페이지를 적재하는 과정
- 특징
  - ▣ 페이지 폴트 핸들러에서 실행되는 작업
  - ▣ 희생 프레임(victim frame) - 비우기로 선택된 프레임
  - ▣ 희생 페이지(victim page) - 희생 프레임에 들어 있는 페이지
  - ▣ 희생 페이지는 스왑-아웃, 요청 페이지는 스왑-인
- 페이지 교체 알고리즘의 목표
  - ▣ 현재 작업 집합에 포함되지 않거나 가까운 미래에 참조되지 않을 페이지를 희생 페이지로 선택,
  - ▣ 페이지 폴트 횟수 줄임



# 희생 프레임의 선택 범위

48

## □ 지역 교체(local replacement)

- 요청한 프로세스에 할당된 프레임 중에서 희생 프레임 선택
  - per-process replacement
- 장점
  - 한 프로세스에서 발생한 스래싱이 다른 프로세스로 전파되지 않음
  - 스래싱에 대한 대책으로 적합
  - 프로세스 별로 독립적으로 페이지 폴트 처리

## □ 전역 교체(global replacement)

- 전체 메모리 프레임 중에서 선택
- 장점
  - 지역 교체보다 더 효과적인 것으로 평가
- 리눅스, Windows 등 많은 운영체제에서 사용



# 잠깐! 페이지 교체 알고리즘에 대한 시각 변화

49

- 페이지 교체 알고리즘은 1960~1970년대의 다중프로그래밍 초기 시대에 뜨거운 연구 주제
- 최근 컴퓨터 시스템의 구조 변화로 인한 시각 변화
  1. 전역 교체 방법은 큰 시간 소모
    - RAM의 용량이 커져 전역 교체 방법으로 희생 프레임을 찾을 때 시간 소모가 커져 비현실적
  2. 캐시의 중요성 부각
    - 메모리보다 CPU 캐시의 활용이 더 중요
  3. 참조의 지역성 약화로 작업 집합이 모호해지는 경향
    - 객체 지향 언어의 사용
      - getter/setter라고 부르는 작은 함수들이 많아져서 이들이 여러 페이지들에 걸쳐 실행
      - 하나의 함수에서 일정 시간 동안 실행되면서 나타났던 참조의 지역성이 흐려짐
    - 배열 대신 트리나 해시맵 등 노드들의 연결 리스트 사용
      - 자료들이 연속된 메모리에 존재하지 않고 여러 페이지에 흩어져 있게 됨
      - 데이터에 대한 참조의 지역성도 떨어지게 됨
    - 가비지 컬렉션
      - 자바나 자바스크립트, 파이썬 프로그램 등 가비지 컬렉션을 기반으로 실행되는 응용프로그램이 많아져서 메모리가 사용되는 패턴에 변화가 생김

# 페이지 교체 알고리즘의 종류

50

- 최적 교체(Optimal Page Replacement)
  - ▣ 이론적으로 최고의 알고리즘
  - ▣ 가장 먼 미래에 사용될 페이지를 교체 대상으로 결정
    - 현실에서는 미래의 페이지 사용 패턴을 알 수 없으므로, 비현실적 알고리즘
    - 다른 알고리즘의 평가를 위한 기준으로 사용
- FIFO(First in first out)
  - ▣ 가장 오래전에 적재된 페이지 선택
  - ▣ 구현 단순
- LRU(Least recently used)
  - ▣ 가장 최근에 사용되지 않았던(가장 오래전에 사용된) 페이지 선택
- Clock
  - ▣ FIFO와 LRU를 섞은 방법
  - ▣ LRU가 단순화됨

# 최적 페이지 교체(Optimal Page Replacement)

51

- 프로세스 당 3개의 프레임이 주어졌고, 지역 교체 가정
  - 미래의 어떤 페이지가 사용될 지 안다는 가정

요청 페이지	2	4	3	1	5	3	7	100	1	6	1	100
초기상태	hit	fault	hit	hit	fault	hit	fault	fault	hit	fault	hit	hit
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	4	4	4	5	5	7	100	100	100	100	100
3	3	3	3	3	3	3	3	3	3	6	6	6

페이지 1, 2, 3이  
적재되었다고  
가정

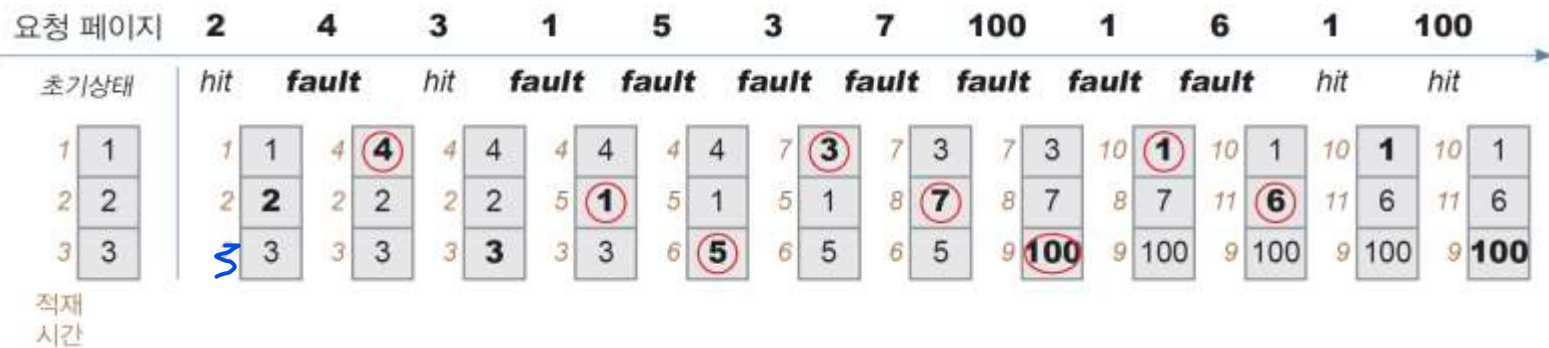
페이지 폴트 횟수 : 5

- 미래의 페이지 액세스에 대해 모르기 때문에 구현 불가능

# FIFO

52

- 프로세스 당 3개의 프레임이 주어졌고
  - ▣ 페이지 1,2,3이 적재되어 있다고 가정



페이지 폴트 횟수 : 8

- ▣ 페이지가 적재된 시간 저장
- ▣ 이해 쉽고, 구현도 쉬움
- ▣ 성능이 좋지 않음
  - 오랜 된 페이지에도 자주 사용되는 변수나 코드가 있을 수 있기 때문

# LRU

53

- 가장 최근에 사용되지 않은 페이지 선택
  - ▣ 페이지 1,2,3이 적재되어 있다고 가정

요청 페이지	2	4	3	1	5	3	7	100	1	6	1	100
초기상태	hit	fault	hit	fault	fault	hit	fault	fault	fault	fault	hit	hit
참조 시간	1	1	5	4	5	8	8	8	11	11	11	11
1	1	4	4	4	5	5	5	100	100	100	100	100
2	2	2	2	1	1	1	7	7	7	6	6	6
3	3	3	3	3	3	3	3	3	1	1	1	1

페이지 폴트 횟수 : 7

- ▣ 좋은 알고리즘으로 평가
- ▣ 많은 운영체제들이 채택하고, 변형하여 사용

# LRU 구현 방법

54

## 1. 타임 스탬프 이용

- 모든 프레임에 참조 시간을 기록할 수 있는 비트 추가
- CPU가 페이지를 참조할 때마다 참조 시간 기록
- 운영체제 교체 알고리즘 코드
  - 희생 페이지를 선택할 때, 참조 시간이 가장 오래된 것 선택
- 시간 기록 및 참조 시간 검사에 많은 오버헤드

## 2. 하드웨어 이용, 참조 비트 사용

- 페이지 테이블 항목에 1개의 참조 비트(Ref 비트) 추가
  - Ref 비트가 1이면 최근에 참조되었음
  - Ref 비트가 0이면 최근에 참조된 적 없음
- CPU가 페이지를 참조할 때마다 H/W로 참조 비트를 1로 설정
  - 참조 비트는 한 비트가 아닐 수 있음(시간 값)
- 운영체제 교체 알고리즘 코드
  - 전체 페이지 테이블 검색해서 참조 값이 가장 낮은(0인) 페이지 선택
  - 주기적으로 Ref 비트를 0으로 만들음
- 하드웨어 비용 + 페이지 테이블 항목 비용

가상 주소

	Presence Bit	Dirty Bit	Ref Bit	Physical address
0	1	0	1	
1	1	0	1	
2	0	0	0	logical block number
3	0	0	0	logical block number
4	0	1	0	logical block number
5	1	1	0	

페이지 테이블

# Clock 알고리즘

55

- FIFO와 LRU를 섞은 알고리즘
  - ▣ LRU 근사 알고리즘, FIFO 근사 알고리즘으로 불림
  - ▣ 2차 기회 알고리즘으로 불리기도 함
- 프레임당 1비트의 참조 비트(reference bit/used bit) 사용
  - ▣ 프레임을 원형 큐로 연결하여 관리
  - ▣ 페이지가 참조될 때마다 프레임의 참조 비트를 1로 셋
- 희생 프레임 선택
  - ▣ 원형 큐에서 검색을 시작하는 프레임 위치를 **포인터**라고 부름
  - ▣ 포인터에서 시작하여 시계방향으로 원형 큐를 따라 이동
    - ▣ 참조 비트가 0이면, 그 프레임을 희생 프레임으로 선택
    - ▣ 참조 비트가 1이면, 0으로 바꾸고 다음 프레임으로 이동
  - ▣ 한 바퀴를 돌게 되면 처음 프레임 선택

# Clock 알고리즘의 동작 사례

56



(1) 포인터가  
프레임 2 가리킴



(2) CPU가 프레임 5의 페이지 참조.  
프레임 5의 참조 비트를 1로 수정



(3) 페이지 교체 요청 발생.  
프레임 4 선택



(4) 페이지 교체 요청 발생.  
프레임 7 선택

페이지 교체 요청이 발생한 경우, 시계 방향으로 이동하면서  
참조 비트가 0인 페이지를 1로 수정하고 희생 페이지로 결정.  
중간에 만다는 참조 비트가 1인 페이지는 0으로 지운다.