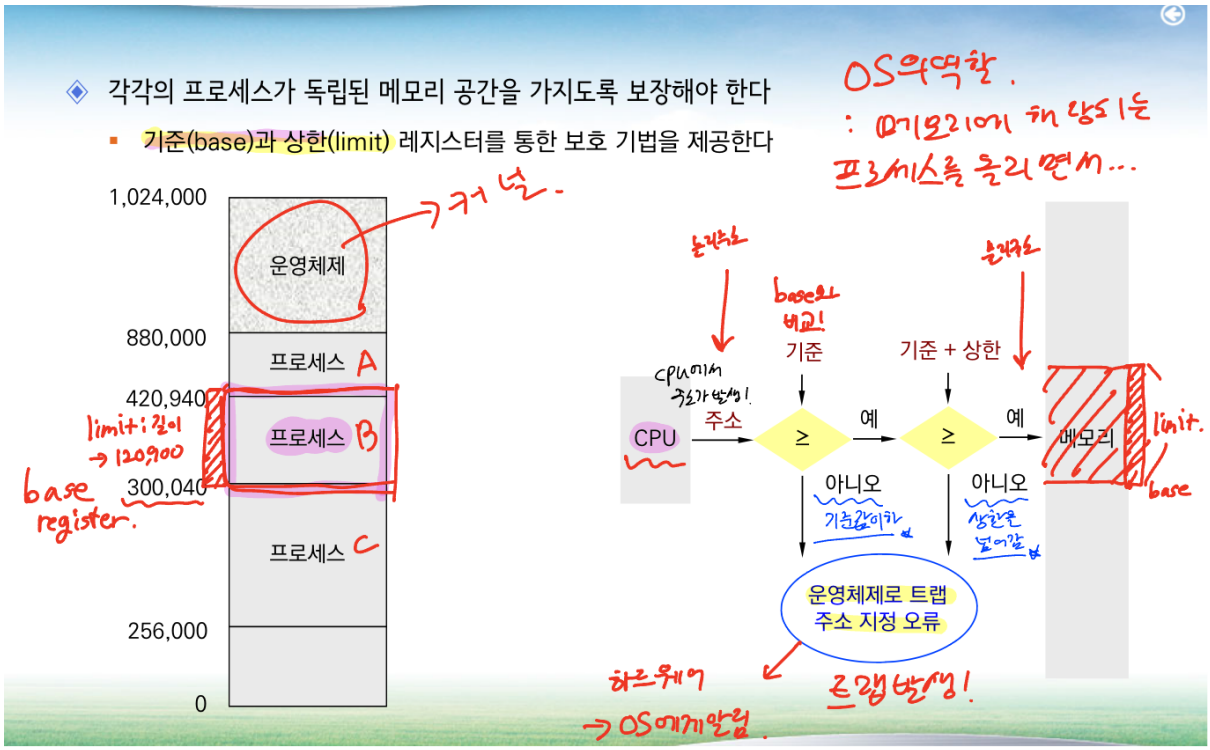
기말고사 정리

Chap 9. 메인 메모리

하드웨어가 메모리를 관리하는 방법?

하드웨어는 각 프로세스의 독립된 메모리 공간을 보장하기 위해 기준 레지스터와 상한 레지스터를 사용하여 보호 기법을 제공.

잘못된 주소 접근 시 트랩을 발생시켜 운영체제에 알리며, 운영체제는 기준과 상한 레지스터 값을 설정



주소의 할당

바인딩이란 ? : 주소 값을 어떻게 세팅하는가..?

명령어와 데이터의 바인딩은 이루어지는 시점에 따라 다음과 같이 구분 됨

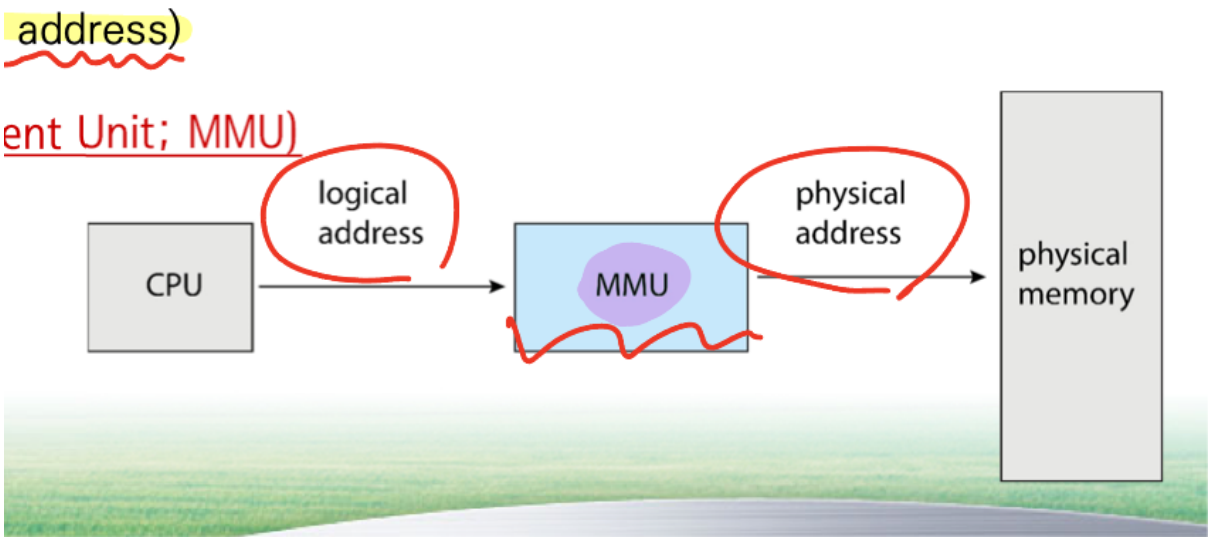
- 컴파일 시간 바인딩 : 메모리의 주소가 컴파일 시간에 결정되는 것

  → OS의 커널에 일부 코드 존재
- 적재 시간 바인딩 : 메모리에 올릴 때 주소를 결정
- 실행 시간 바인딩 : 실행 중간에 메모리의 주소가 변경될 수 있음

컴파일시/적재 시 주소 바인딩 기법의 경우 → 논리, 물리 주소가 같음 / **논리 주소=가상 주소**

실행시간 주소 바인딩 기법의 경우 → 논리, 물리 주소가 다름

메모리 관리기 `MMU` 를 통해 **가상 주소에서 물리 주소로의 변환을 수행하며 재배치 레지스터를 사용** → Memory Management Unit



동적 적재

프로세스 실행을 위해 필요한 코드를 모두 메모리에 미리 올려야 함. 이를 효율적으로 하기 위해 동적 적재를 사용

각 코드는 호출되기 전까지 메모리에 올라오지 않고 디스크에서 대기하며, 필요할 때만 적재되고, 큰 코드 양이 필요한 경우에 유용하게 활용됨

동적 연결 및 공유 라이브러리

동적 연결 라이브러리는 프로그램이 실행 시 연결되는 라이브러리로 `.dll` 파일 (dynamic linking library)

많은 실행 파일들에서 공통으로 사용하며 루틴을 바꿀 때 유용하다는 장점이 있으며 운영체제의 도움을 필요로 함

메모리 보호

프로세스간 메모리 접근을 제어하는 것이며, 상한 레지스터와 기준(재배치) 레지스터를 사용해 각 프로세스의 메모리 영역을 정의한다. MMU를 통해 논리 주소를 물리 주소로 변환하여 보호 기능을 수행

동적 메모리 할당 문제 해결책

1. 최초 적합

   → 첫 번째 사용 가능한 가용 공간에 할당

2. 최적 적합

   → 사용 가능한 공간 중에서 가장 작은 가용 공간을 할당

3. 최악 적합

   → 가장 큰 가용 공간을 할당

단편화

외부 단편화

프로세스들이 메모리에 적재되고 제거될 때 할당되지 않고 작은 조각들로 메모리가 나뉘어져 있는 상태

외부 단편화 해결책

1. **메모리의 내용을 한쪽으로 밀어서 큰 공간을 만드는 방법 : 밀집**

   → 시간 낭비가 심함

2. **요구되는 메모리 크기보다 더 크게 할당**

   → 이때, 할당 했지만 안쓰는 공간을 내부 단편이라고 함

3. 한 프로세스의 논리 주소 공간을 여러 개의 비연속적인 공간에 나누어 할당하는 방법

   → **세그멘테이션과 페이징을 사용**

페이징

논리 주소 공간을 페이지로 분할하여 관리하는 메모리 기법

물리 메모리는 동일한 크기의 프레임으로, 논리 메모리는 동일한 크기의 페이지로 분할됨

프로세스가 실행될 때 페이지들은 파일 시스템이나 예비 저장장치로부터 사용 가능한 프레임으로 적재됨

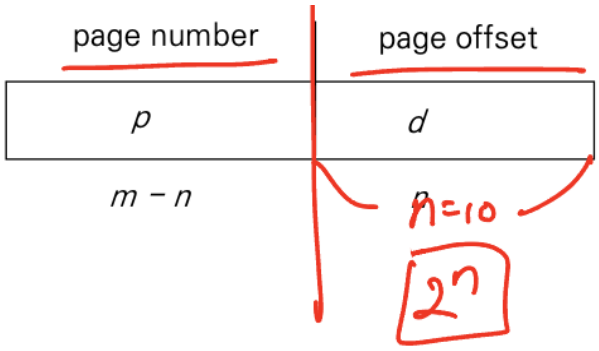
이를 통해 페이징은 논리 주소 공간의 연속성을 유지할 필요 없이 메모리를 효율적으로 관리할 수 있음

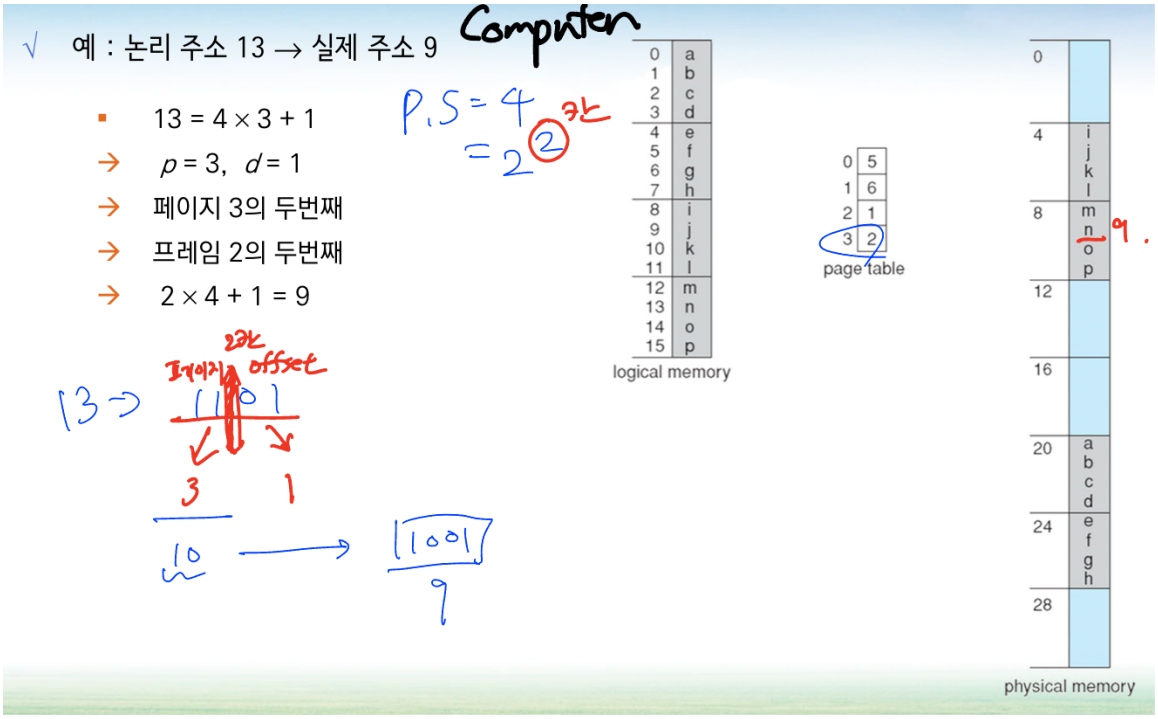
페이지 테이블

논리 주소와 물리 주소 간의 매핑 정보를 저장하는 데이터 구조로 MMU를 통해 논리 주소를 물리 주소로 변환

페이지 테이블은 다음과 같이 구성

- 페이지 번호(number)
  - 페이지 테이블의 index로 사용
- 페이지 변위(offset)
  - 참조되는 프레임 안에서의 위치





페이징의 특징

- 외부 단편화가 발생하지 않으나 내부 단편화는 발생

- 동적 재배치의 형태로 페이지들이 적재

- 메모리에 대한 사용자의 인식과 실제 내용이 서로 다르지만, MMU에 의해 해소됨 (ex : 배열)

페이지 테이블의 구현

페이지 테이블은 메모리에 저장됨

페이지 테이블 기준 레지스터(PTBR)가 페이지 테이블을 가리키며,

페이지 테이블 길이 레지스터(PTLR)을 통해 프로세스가 제시한 주소가 유효한 범위 내에 있는지 확인

데이터 접근 시 두번의 메모리 접근이 일어나며 메모리 엑세스 시간 저하가 발생하는데, 이를 해결하기 위해 페이지와 프레임의 매핑 정보를 빠른 연관 메모리에 저장한 TLBs를 사용

실질 메모리 접근 시간 : EAT

메모리에 페이지 테이블을 저장하는 페이징 시스템을 가정할 때 다음 각 질문에 답하시오.

a. 물리 메모리 참조가 50nsec가 걸린다면 페이징 시스템을 적용했을 때 실제 메모리 참조는 얼마나 걸리는가?

b. TLB를 추가하고 모든 페이지 테이블 참조의 75%를 TLB에서 찾을 수 있다면 실제 메모리 접근 시간은 얼마인가?
(TLB에 존재하는 페이지 테이블 항목을 찾는데 2nsec가 걸린다고 가정한다)

a. 실제 메모리 참조에는 페이지 테이블을 위해 한번, 데이터/명령을 위해 한번 총 두번의 메모리 접근이 필요

→ 총 두번의 메모리 접근이 필요 : $50nsec \times 2 = 100nsec$

b.

TLB hit : $TLB + M/A = 52nsec$

TLB miss : $TLB + M/A + M/A = 102nsec$

→ M/A = Memory Access

$$EAT = TLB\ hit \times hit\ ratio\ +\ TLB\ miss \times (1 - hit\ ratio)$$
$$52 \times 0.75\ +\ 102 \times 0.25 = 64.5nsec$$

메모리 보호 / 가상 메모리에서 의미있게 사용됨

페이징 환경에서 메모리 보호는 페이지 테이블의 보호 비트를 통해 구현됨

유효 비트는 합법적인 페이지를 나타내고, 무효 비트는 논리 주소 공간에 속하지 않는 페이지를 나타냄

OS는 이를 이용해 페이지에 대한 접근을 허용하거나 거부

페이징의 장점

페이징을 통해 같은 프레임 번호를 사용하여 공통의 코드를 공유할 수 있으므로 메모리 절약이 가능

페이지 테이블의 구조

계층적 페이징

페이지 테이블을 페이지 단위로 여러 단계로 나누어 구성하는 방식

일반적으로 2단계 페이징이 사용되나, 최근 시스템은 64비트를 사용하므로 4단계 페이징을 사용하는 경우가 많음

→ 현재 대부분의 OS에서는 계층적 페이징을 사용

해시 페이지 테이블

해시를 사용하여 페이지 테이블 항목을 빠르게 찾는 방식으로, 시스템 크기가 32비트 보다 커지면 많이 사용됨

역 페이지 테이블 → 현재는 사용되지 않음

페이징에서의 스와핑

페이지를 메모리에서 백업 저장장치로 이동 시키는 과정 : 페이징-아웃

페이지를 백업 저장장치에서 메모리로 이동 시키는 과정 : 페이징-인

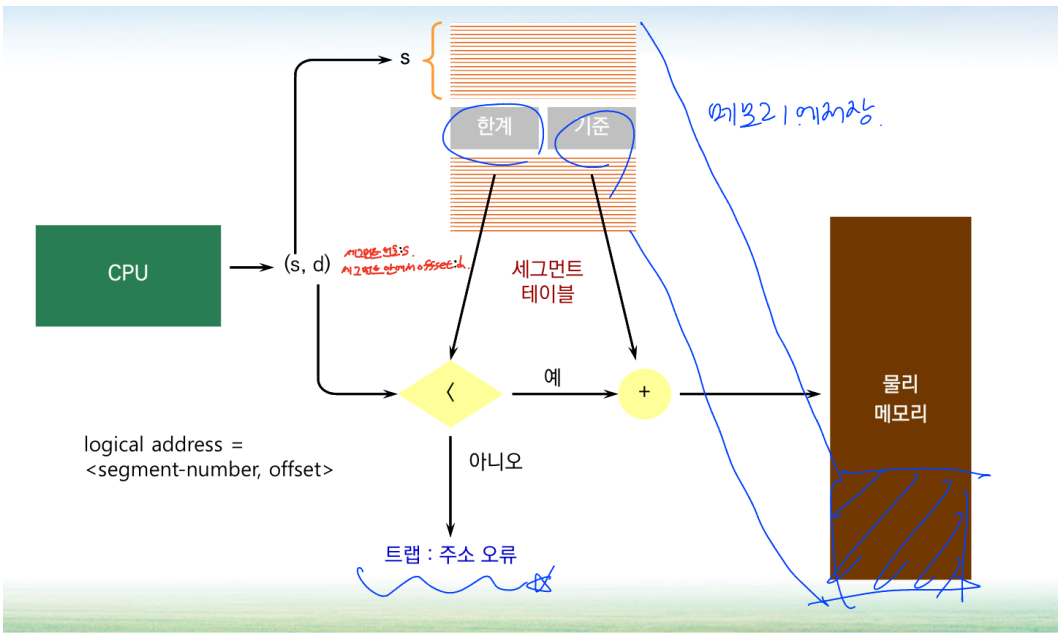
세그멘테이션

프로그래머가 생각하는 모양을 그대로 지원하는 메모리 관리 기법

각 세그먼트는 프로그램의 특정 부분을 나타내며 세그먼트마다 사이즈가 다를 수 있음

세그멘테이션을 통해 프로그램을 논리적인 단위로 구성, 메모리를 효율적으로 사용 가능

주소 변환에는 메모리에 저장된 세그먼트 테이블이 사용되고 내부 단편화가 발생할 수 있음


s
한계
기준
CPU
(s, d)
세그먼트
테이블
<
예
+
물리
메모리
logical address =
<segment-number, offset>
아니오
트랩 : 주소 오류

페이징-세그먼트 사례

만약 세그멘테이션과 페이징을 지원하지 않는다면 연속 메모리 할당을 사용해야 하지만,

둘다 지원하는 경우에는 OS에서 둘 다 선택해서 사용할 수 있음

4. 다음과 같이 세그먼트 테이블이 주어질 때 다음 각 논리 주소에 대한 물리 주소는 무엇인가?

길이 < 범위 (같으면X).

세그먼트	기본 = 시작점	길이 = 한계
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

범위.

a. (0, 430)
b. (1, 10)
c. (2, 500)
d. (3, 400)
e. (4, 112)
f. (4, 96).

a. 219 + 430 = 649.
b. 2300 + 10 = 2310.
c. 오류발생! → 주소오류
d. 1327 + 400 = 1727.
e. 오류발생! → 주소오류
f. 오류발생! → 주소오류

같은 크기도 오류발생!

Chap 10. 가상 메모리

가상 메모리

실제 메모리와 논리 메모리를 분리하여 동작하는 기법

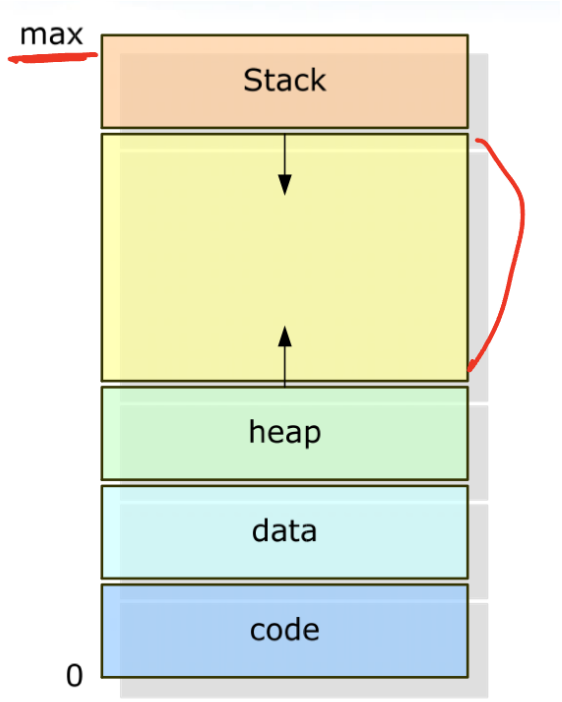
프로세스 전체가 메모리에 존재하지 않아도 실행이 가능

가상 메모리를 활용하면 예비저장공간에는 전부 들어가 있음

가상 메모리 사용 이유?

1. 물리적인 메모리 크기에 의해 제약받지 않음
2. 많은 프로그램을 동시해 수행할 수 있음
3. 페이지 교체와 스왑 과정에 필요한 I/O 횟수가 줄어듬

가상 주소 공간

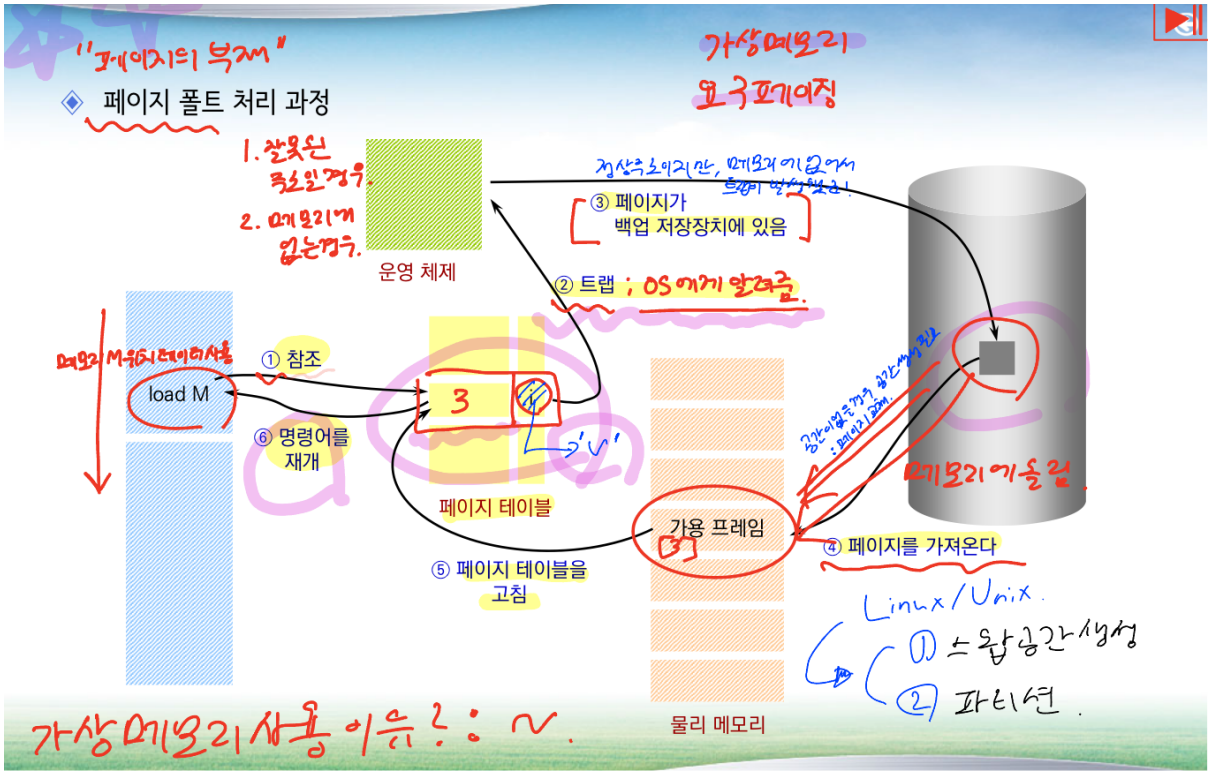


프로세스가 메모리에 저장되는 과정에서 스택과 힙의 중간에 비어있는 공간으로, 성긴 공간(sparse)이라 한다.

요구 페이징

사용자가 필요할 때 해당되는 페이지를 메모리에 올리는 것

페이지 폴트(페이지 부재) 처리 과정 → 유효/무효 비트에 무효로 표시되어 있는 경우



1. CPU로 부터 주소가 발생 → 페이지 테이블로 이동
2. 메모리에 접근이 불가능하여 트랩을 발생시켜 OS에게 알리며 OS가 동작
3. OS는 아래 두 가지로 해석

   a. 잘못된 주소인 경우 b. 메모리에 없는 경우(페이지 폴트) → 트랩이 페이지 폴트 상황임을 판단
4. 보조저장장치에서 페이지의 위치를 찾아 메모리에 올림 → OS가 보조저장장치의 페이지 위치도 관리를 의미
5. 가용 프레임으로 읽기요구를 내어, 디스크에 있는 데이터를 메모리에 올림

   a. 데이터를 메모리에 올리는 동안 CPU는 다른 사용자에게 할당됨

   : 이때, 가용 프레임 공간은 반드시 있어야 하므로, 없을 시 공간을 만들어야함
   → 페이지 교체 과정(희생자)
6. 저장장치가 다 읽었다고 인터럽트를 알림(I/O 완료 상황)
7. 주소를 페이지 테이블에 수정하여 올림
8. 명령을 재개

순수 요구 페이징

실제 프로세스는 메모리에 전혀 올라와있지 않은 상태를 순수 요구 페이징이라고 함

처음에 PCB, 페이지 테이블 공간 등 커널을 위한 자료구조들은 메모리에 만들어져 있음

이후 코드에서 첫 번째 문장이 수행되는 순간 그 문장을 위해서 필요한 페이지부터 하나씩 올리는 것

→ C에서는 `main()`

실제로는 처음 일부 페이지를 메모리에 올려놓고 실행

요구 페이징을 지원하기 위해 필요한 하드웨어

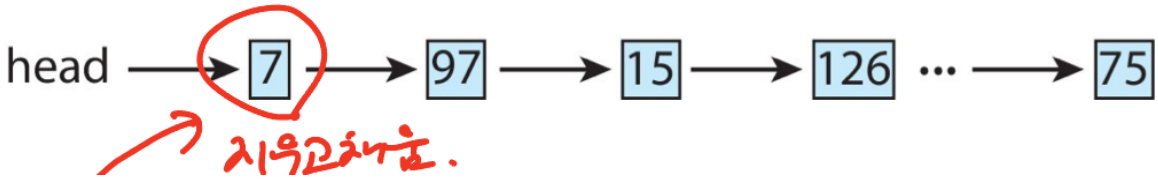
- 페이지 테이블 : 유효/무효 비트 설정
- 보조저장장치 : 스왑공간 → 메인 메모리에 없는 모든 페이지를 가지고 있어야 하므로
- 페이지 폴트 처리 후에 재시작할 수 있는 명령어를 CPU가 제공해야함

가용 프레임 리스트

페이지 폴트를 해결하기 위해 OS는 zero-fill-on-demand라는 기법을 사용해 가용 프레임 리스트를 할당

시스템이 시작되면 모든 가용 메모리가 가용 프레임 리스트에 넣어짐

가용 프레임이 요청되면 가용 프레임 리스트의 크기가 줄어듬



요구 페이징 성능

페이징으로 외부 단편화는 해결 가능, 두 번의 메모리 접근 과정으로 성능은 떨어짐

→ 성능을 높이는 방법 : 페이지 교체 알고리즘을 통해 **페이지 폴트율을 낮추는 것**

페이지 폴트율 : p (메모리에 적재 되어있지 않을 확률) / 메모리 접근 시간 : ma

페이지 폴트 시간

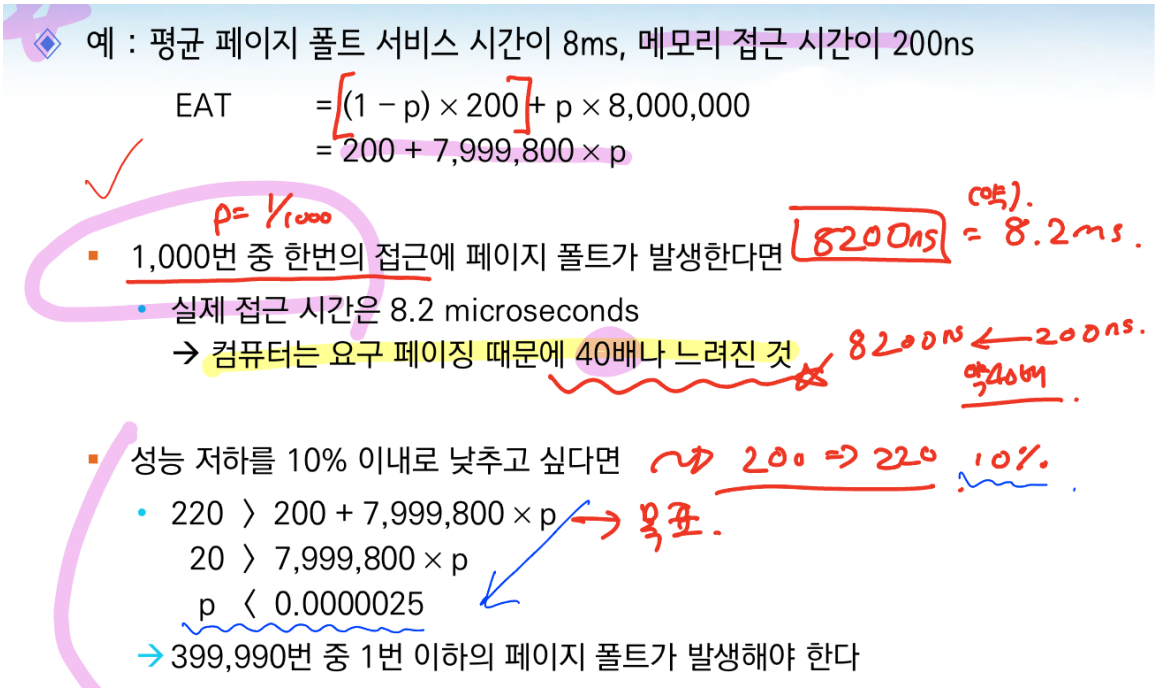
= page fault overhead + **[swap page out] + swap page in** + restart overhead

: 밑줄 친 부분이 많은 시간을 차지

swap page in : 디스크에 있는 것을 메모리에 올림 (프로세스는 아무 일도 하지 못함)

swap page out : 가용 메모리 공간이 없는 경우 있는걸 빼고 올라오는 시간

실질 접근 시간 : EAT = $(1-p) \times ma + p \times$ 페이지 폴트 시간



→ **Hard real time system에서는 정확한 예측이 불가능하므로 가상 메모리 사용 X**

쓰기 및 복사 / Copy on write

- 부모 프로세스가 자식 프로세스를 생성할때 쓰기 시 복사 방식을 사용 가능

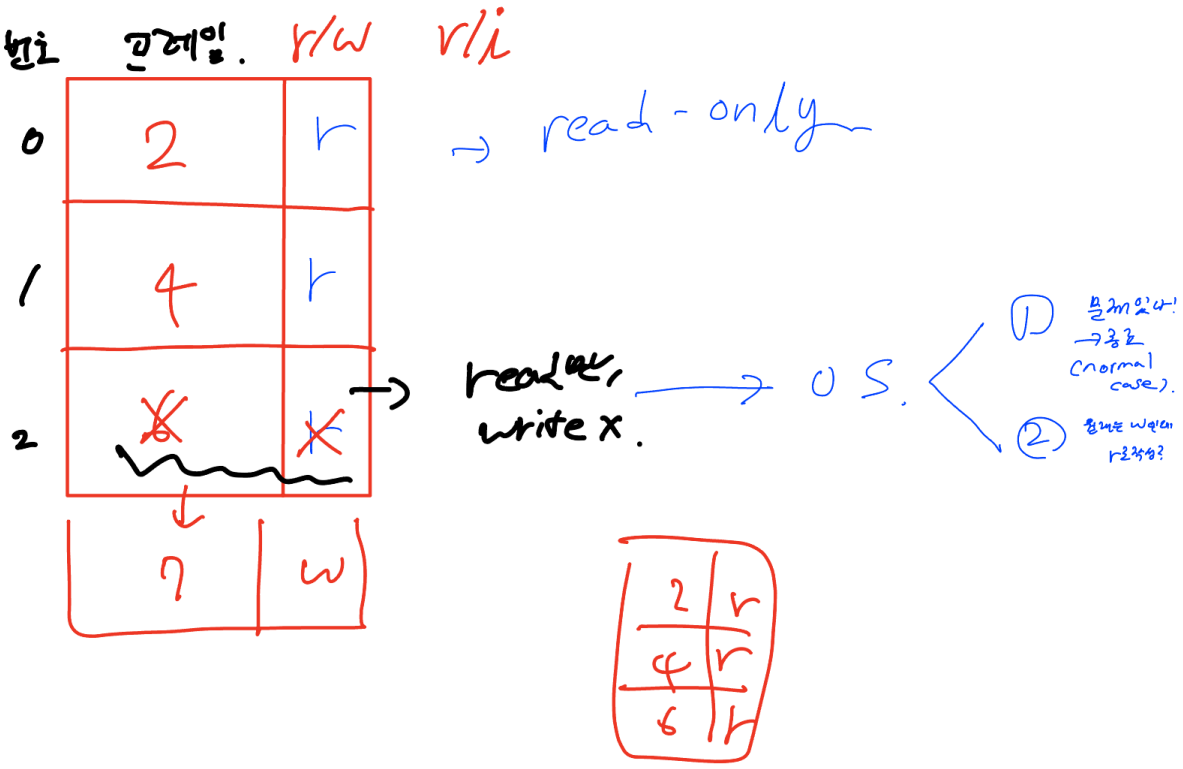
- 부모 프로세스의 페이지 테이블을 자식 프로세스가 그대로 공유하지만 데이터를 실제 수정하려는 시점에서 복사를 수행하는 것 → 각 프로세스는 독립적인 데이터를 사용할 수 있음

- 복사본을 데이터 사용 시점마다 매번 생성하는 것은 비효율적 → r/w bit를 사용해 표시하며 read, write 여부를 표시

- OS에서 두 가지 경우로 판단

1. read only page를 write하려 하는 경우 : 프로그램을 종료
2. read only page가 아닌 copy on write를 사용하기 때문에 write인 경우 : 복사를 수행

- 2번 진행 후 각자의 테이블을 가지고 있기 때문에 더 이상의 복사가 필요 없어짐



- fork() 외에 vfork()라는 시스템 콜을 제공

- 부모 프로세스가 생성되는 자식 프로세스가 exec()가 바로 실행 될 것을 가정 후 사용

- 기본 커널 자료구조만 만들고 메모리를 별도로 할당하지 않음

페이지 교체

- 프로세스가 실행되는 동안 페이지 폴트가 발생하고 OS는 필요로 하는 페이지가 보조저장장치에 저장된 위치를 알아내지만 가용한 프레임 목록에 가용한 프레임이 없음을 발견하게 됨

- OS는 프로그램을 중단하거나 표준 스와핑을 사용하는 방법을 선택할 수 있으나 좋은 선택은 아님

- 대부분의 OS는 페이지 스와핑과 페이지 교체를 결합하여 사용

페이지 교체 알고리즘

예 : 특정 프로세스의 참조 주소 추적

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

- 페이지 크기 = 100 → 참조열 : 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

참조열 : 메모리에 대한 페이지 접근이 일어나는데, 한번 메모리에 올라오면 **절대 페이지 폴트가 발생하지 않으므로** 아래와 같이 축약해서 작성한 것,

14161111161111611611 → 14161616161

페이지 교체 알고리즘의 성능을 평가하는 기준 : 가장 낮은 페이지 폴트 비율

- FIFO 페이지 교체 알고리즘
- 최적 페이지 교체 알고리즘
- LRU 페이지 교체 알고리즘
- LRU 근사 페이지 교체 알고리즘

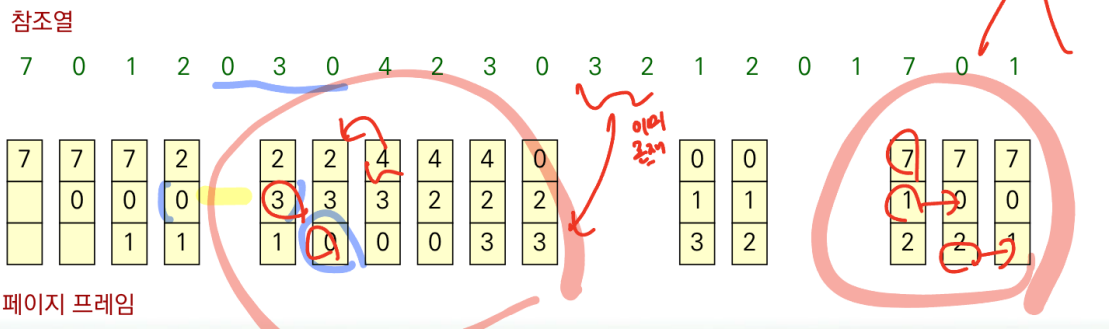
FIFO 페이지 교체 알고리즘

- 페이지에서 교체되어야 할 때 가장 오래된 페이지를 선택 / FIFO 큐의 사용

◈ First-In-First-Out (FIFO) Page Replacement Algorithm

◈ 가장 간단한 페이지 대치 알고리즘

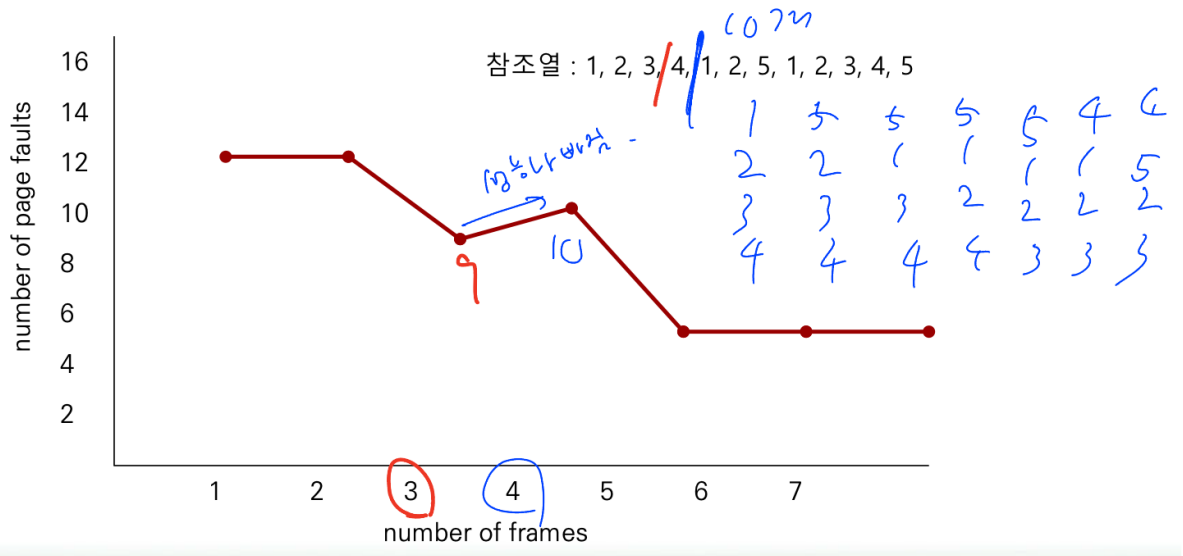
- 페이지가 교체되어야 할 때 가장 오래된 페이지를 선택한다
- FIFO 큐의 사용
  - 큐의 머리 부분 페이지를 교체하고, 새로 올라온 페이지는 큐의 끝에 삽입하면 된다
- 이해하기 쉽고 프로그램하기도 쉽다



→ **페이지 폴트의 개수가 20개중 15개이다. 로 표현**

→ ~~**페이지 폴트율이 15/20 이다.**~~

Belady의 모순 → 프레임의 수가 많아진다고 성능이 좋아지는 것은 아님



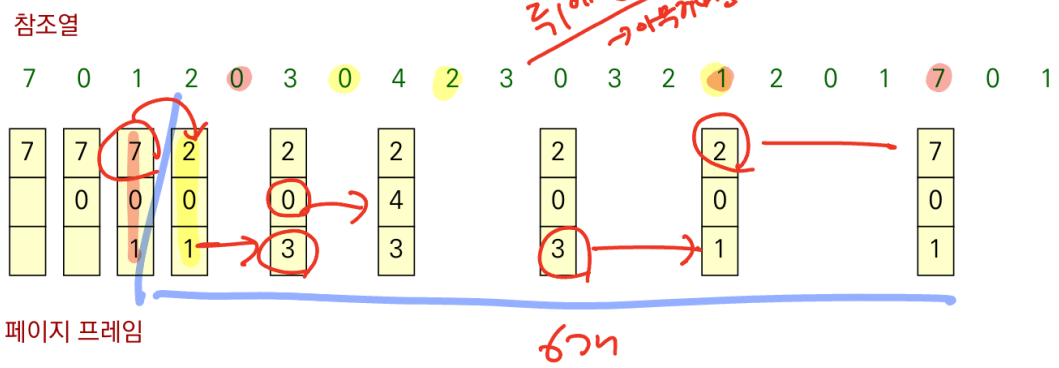
최적 페이지 교체

- 앞으로 가장 오랜 기간 동안 사용되지 않을 페이지를 교체
- 모든 알고리즘 가운데 페이지 폴트 비율이 가장 낮음
- **현실적으로 최적 페이지 교체 알고리즘은 구현하기 어려움**

→ 참조열에 대한 미래의 지식을 알기 어려움 / 주로 비교 연구를 위해 사용

◈ 앞으로 가장 오랜 기간 동안 사용되지 않을 페이지를 교체한다

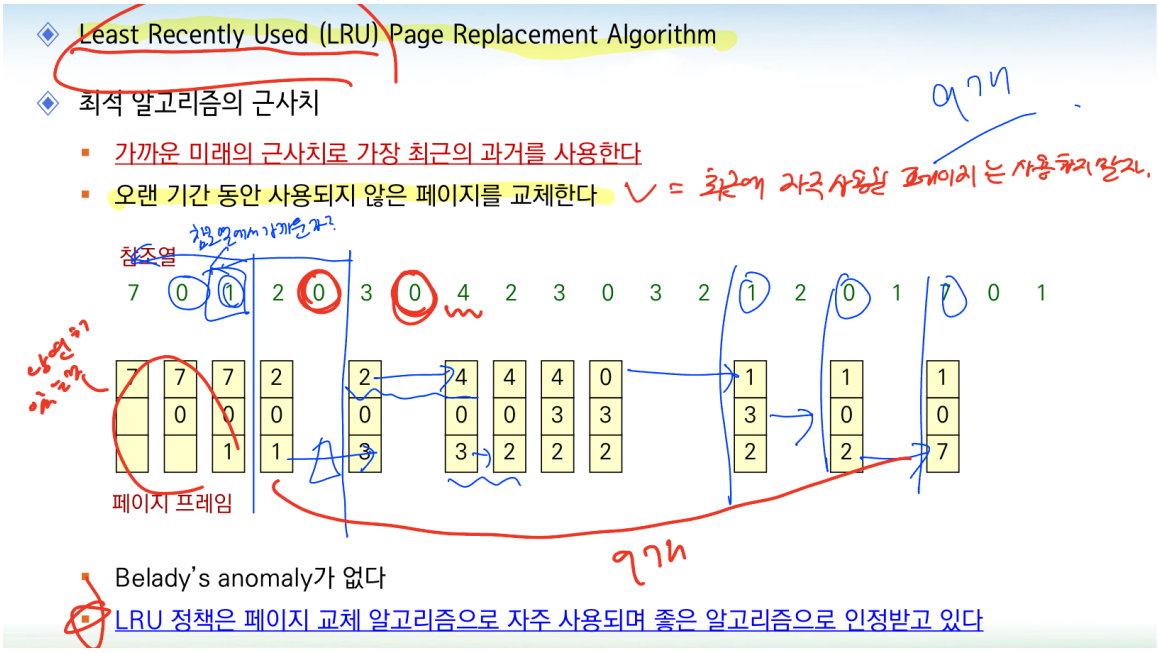
- 모든 알고리즘 가운데 페이지 폴트 비율이 가장 낮다
- Belady's anomaly를 일으키지 않는다



- 현실적으로 최적 페이지 교체 알고리즘은 구현하기 어렵다
  - 참조열에 대한 미래의 지식을 알기 어렵기 때문 / 주로 비교 연구를 위해 사용

LRU 페이지 교체

- Least Recently Used 페이지 교체
- 최적 알고리즘의 근사지
- 가까운 미래의 근사치로 가장 최근의 과거를 사용
- 오랜 기간 동안 사용되지 않은 페이지를 교체 → **최근에 사용된 페이지는 교체하지 않는다.**



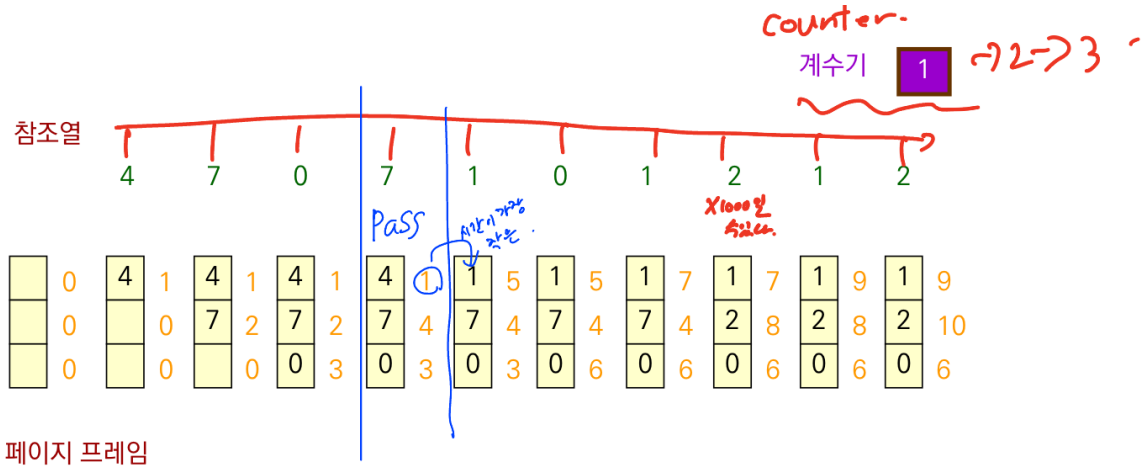
> ! **LRU 정책은 페이지 교체 알고리즘으로 자주 사용되고 좋은 알고리즘으로 인정받음**

LRU 페이지 교체 알고리즘의 구현 문제 : 하드웨어의 지원이 필요

- 프레임들을 최근 사용된 순서로 파악할 수 있어야함

1. 계수기 (counter)의 사용

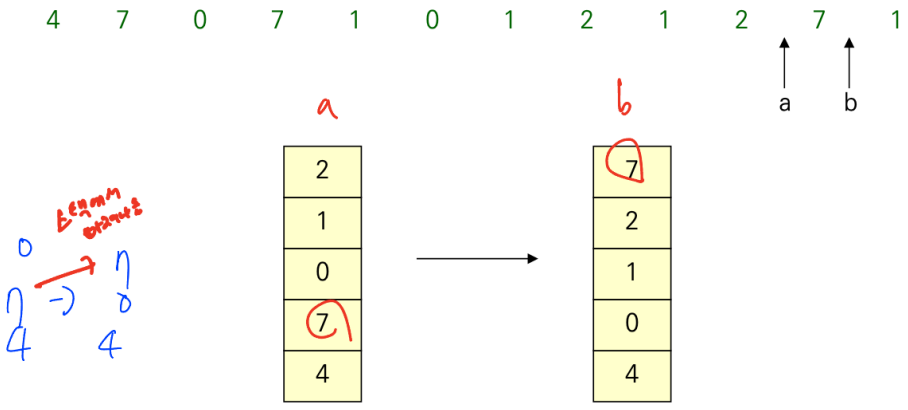
- 페이지 항목마다 사용 시간 필드를 넣고, CPU에 논리 클럭이나 계수기를 추가
- 메모리 접근 시마다 증가 → **가장 작은 시간 값을 가진 페이지를 교체**



: 기본적으로 공간도 많이 사용되며, 하드웨어가 필수적으로 요구됨 → **오버헤드 부담 추가**

2. 스택(stack)의 사용

- 페이지 번호의 스택을 유지, 페이지를 참조할 때마다 페이지 번호를 스택 top에 유지
- bottom에 있는 페이지가 가장 오래 전에 사용된 페이지



> ! 시스템들이 LRU 페이지 교체를 위한 하드웨어 지원을 충분히 할 수 없음

→ LRU 특성을 사용하는 알고리즘으로 구현을 심플하게 바꾼 것 : **LRU 근사 교체**

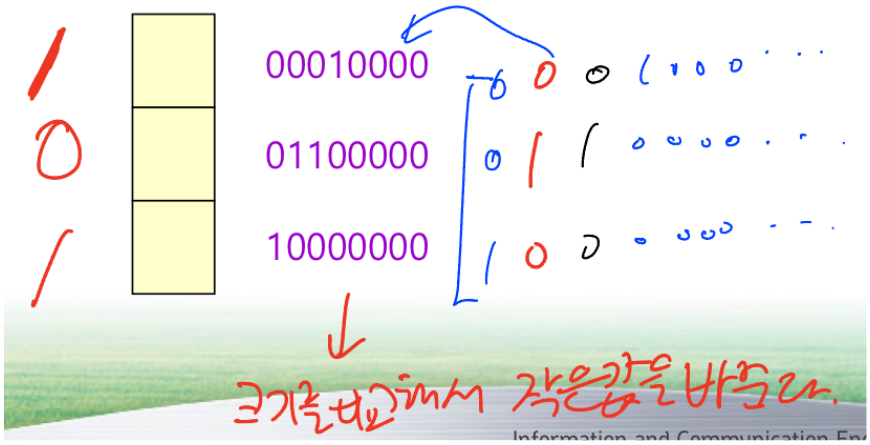
LRU 근사 페이지 교체

- 많은 시스템들이 참조 비트(reference bit)의 형태로 어느 정도 지원

- OS에서는 이것을 활용하여 아래 알고리즘을 구현

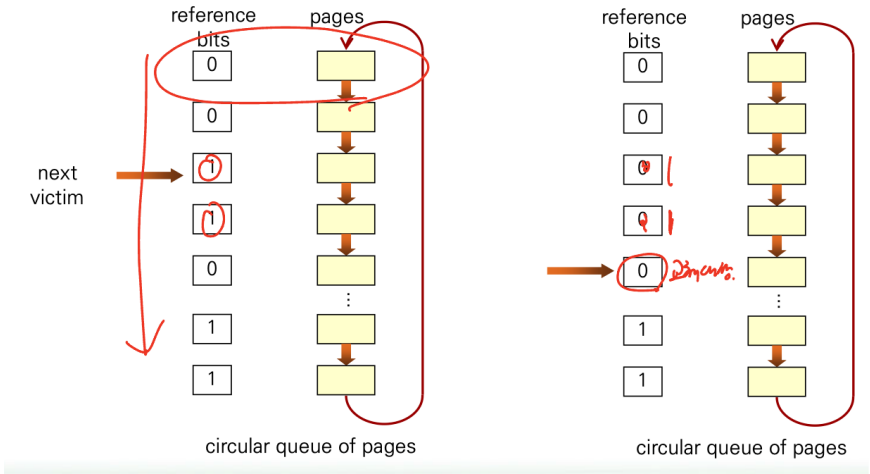
- 부가적 참조 비트 알고리즘

  - 계수기가 아닌 8비트의 참조 비트와 타이머 인터럽트 사용
  - 사용된 비트는 표시를 하고 타이머트 인터럽트가 발생할 때마다 right shift + 교체된 페이지 표시 진행
  - 8비트의 크기를 비교해서 가장 작은 값을 바꿈



- 2차 기회 알고리즘

  - 페이지가 선택될 때마다 참조 비트를 확인하고 참조 비트가 0이면 페이지를 교체
  - 1이면 다시 한번 기회를 주고 다음 페이지로 넘어감 (교체 대상에서 제외)



- **개선된 2차 기회 알고리즘**

  - 참조 비트와 변경 비트를 사용하면 더 개선할 수 있음
  - 두 개의 비트를 조합하여 사용하면 4가지 등급이 가능 → 성능을 높일 수 있음
    - (0, 0) 최근에 사용되지도 변경되지도 않은 경우
      - **교체하기 가장 좋은 페이지** → **내용이 바뀐적이 없으므로 swap in/out 시간 소요가 없음**
    - (0, 1) 최근에 사용되지는 않았지만 변경은 된 경우
      - swap in/out 과정이 있으므로 사용하지 않는 것이 좋음
    - (1, 0) 최근에 사용은 되었으나 변경되지 않은 경우
    - (1, 1) 최근에 사용도 되었고 변경도 된 경우

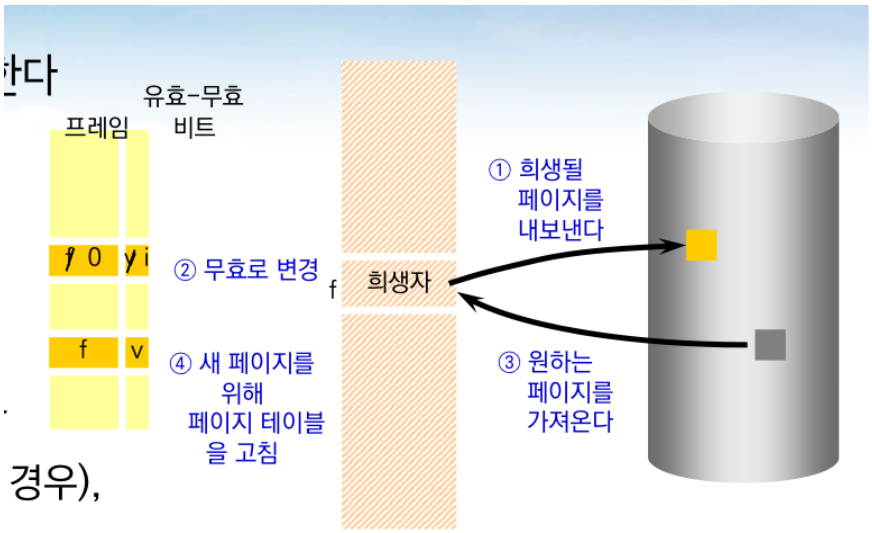
- **페이지 버퍼링 알고리즘**

  - **페이지 교체 알고리즘과 병행하여 버퍼링 기법이 사용될 수 있음**
  - 가용 공간을 미리 확보해놓는 것으로 가용 프레임 풀(pool)을 유지
    - 희생될 프레임을 바로 swap out 하는 것이 아닌 pool에 등록 진행

    → 이로 인해 프로세스는 희생된 페이지가 swap out 되기를 기다리지 않고 빠르게 실행 가능
  - 변경된 페이지들의 리스트를 유지
    - 페이지 장치가 쉬는 동안 변경된 페이지를 선택해 디스크에 기록하고 변경 비트를 0으로 설정
  - **부가적인 효과** : 가용 프레임의 저장소를 유지하면서 각 프레임의 어느 페이지가 있는가를 기억
    - 필요하면 가용 프레임 풀에서 직접 다시 사용될 수 있음

페이지 교체 과정

1. 보조저장장치에서 필요한 페이지의 위치를 알아냄
2. 빈 페이지 프레임을 찾음
    a. 비어 있는 프레임이 있다면 그것을 사용
    b. 비어 있는 프레임이 없다면 희생될 프레임을 선정하기 위해 페이지 교체 알고리즘을 가동
    c. 희생될 페이지를 보조저장장치에 기록하고(필요한 경우), 관련 테이블을 수정
3. 빼앗은 프레임에 새 페이지를 읽어오고 테이블을 수정
4. 페이지 폴트가 발생한 지점에서부터 프로세스를 계속함



프레임 할당

최소 프레임수

- 유효 프레임 수보다 많이 할당할 수 없지만 할당 가능한 최소 프레임이 있음

- 프로세스에 할당되는 프레임 수가 줄어들면 페이지 폴트율이 증가하고 프로세스 실행이 늦어짐

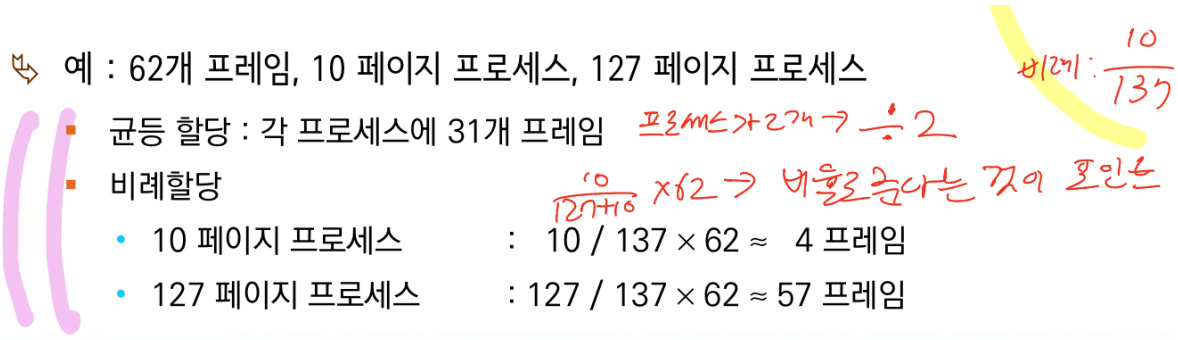
- 프로세스 당 최소 2개의 프레임 필요

할당 알고리즘

- 균등 할당 : 모든 프로세스에 똑같은 프레임을 할당, 나머지는 가용 프레임 풀로 이동

- 비례 할당 : 프로세스의 크기에 비례하여 각 프로세스에 프레임을 할당

→ 프로세스가 크다고해서 꼭 프레임의 개수가 많은 것은 아님



- 우선순위도 고려하여 우선순위가 높은 프로세스에 많은 기억 장소를 할당하여 수행 속도를 높이는 것이 바람직

전역 대 지역 할당

페이지 교체 알고리즘은 전역 교체와 지역 교체로 나뉘어짐

- 전역 교체 : 모든 프로세스의 프레임을 대상으로 교체를 수행하는

- 지역 교체 : 프로세스에게 할당된 프레임 중에서만 교체될 희생자를 선택

→ 전역 교체 알고리즘이 일반적으로 더 좋은 성능을 나타냄

스레싱(Thrashing)

스레싱의 원인

- OS는 CPU의 이용률이 낮아지면 다중 프로그래밍 정도를 높임

- 페이지 부재가 많아지고 이로 인한 페이지 교체가 과도하게 발생하여 시스템 성능이 저하
→ **스레싱**

- 스레싱의 이유

1. 다중 프로그래밍 정도(degree of multiprogramming)를 너무 높임
2. 프로세스에 할당되는 실제 프레임 수가 부족한 경우
3. 프로세스 실행 시간보다 페이지 교체 시간이 더 클 때 발생

→ CPU 이용률이 급격히 저하됨

- 아래는 스레싱 현상을 방지하기 위해서 사용하는 전략

작업 집합 모델

- 메모리 참조 지역성을 기반으로 하는 메모리 관리 모델

- 프로세스의 작업 집합을 관리하고 각 프로세스에게 작업 집합의 크기에 맞는 충분한 프레임을 할당

- 작업 집합 추적은 정확하게 가능하나, 오버헤드가 있음

페이지 폴트 빈도

- OS에서 페이지 폴트율을 체크후 프래임 개수를 조절하는 것

- 페이지 폴트율이 일정치 이상 높아지면 프레임 개수를 늘리고 반대의 경우 프레임 개수를 감소시켜 균형을 맞춤