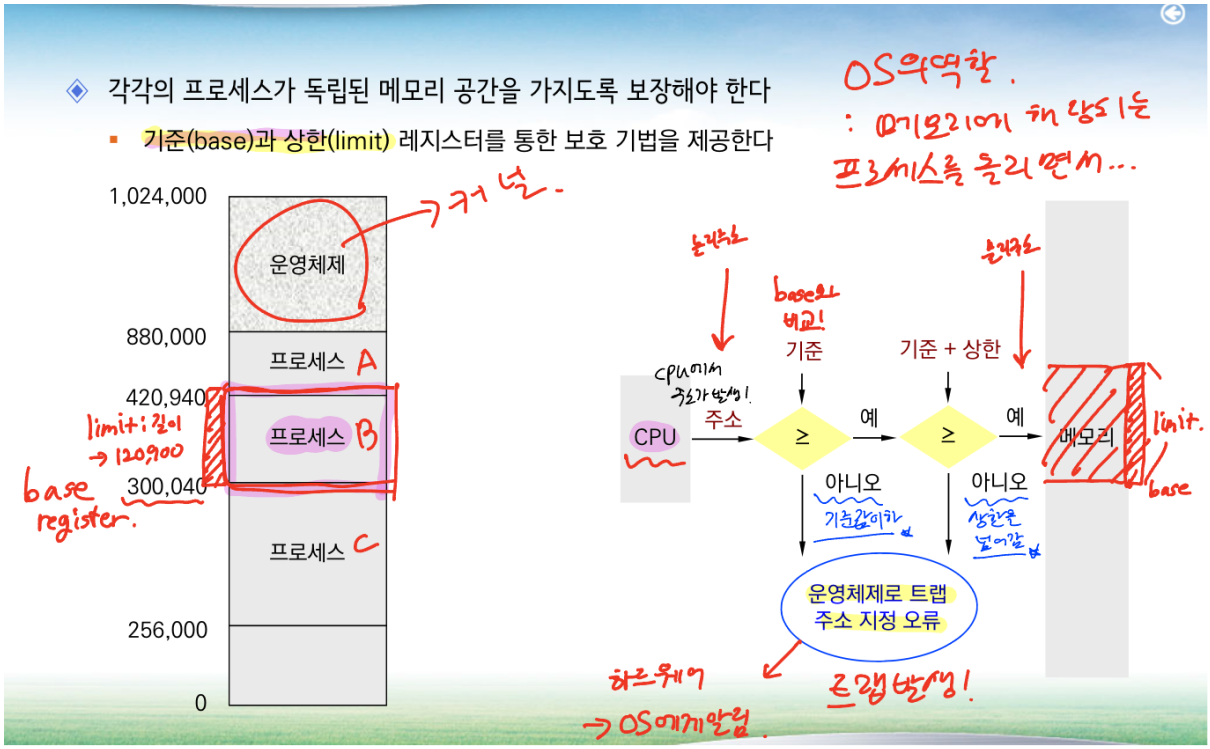
기말고사 정리

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

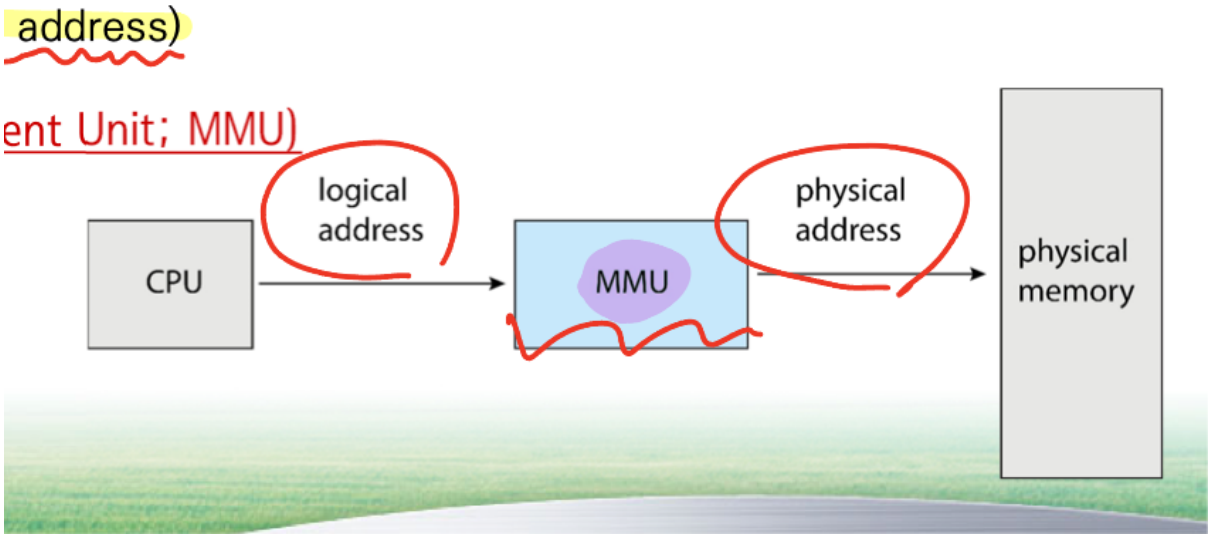
  → OS의 커널에 일부 코드 존재
- 적재 시간 바인딩 : 메모리에 올릴 때 주소를 결정
- 실행 시간 바인딩 : 실행 중간에 메모리의 주소가 변경될 수 있음

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 이때, 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 떄 페이지들은 파일 시스템이나 예비 저장장치로부터 사용 가능한 프레임으로 적재됨

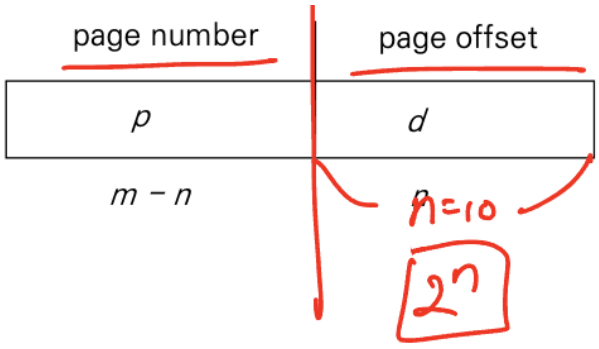
이를 통해 페이징은 논리 주소 공간의 연속성을 유지할 필요 없이 메모리를 효율적으로 관리할 수 있음

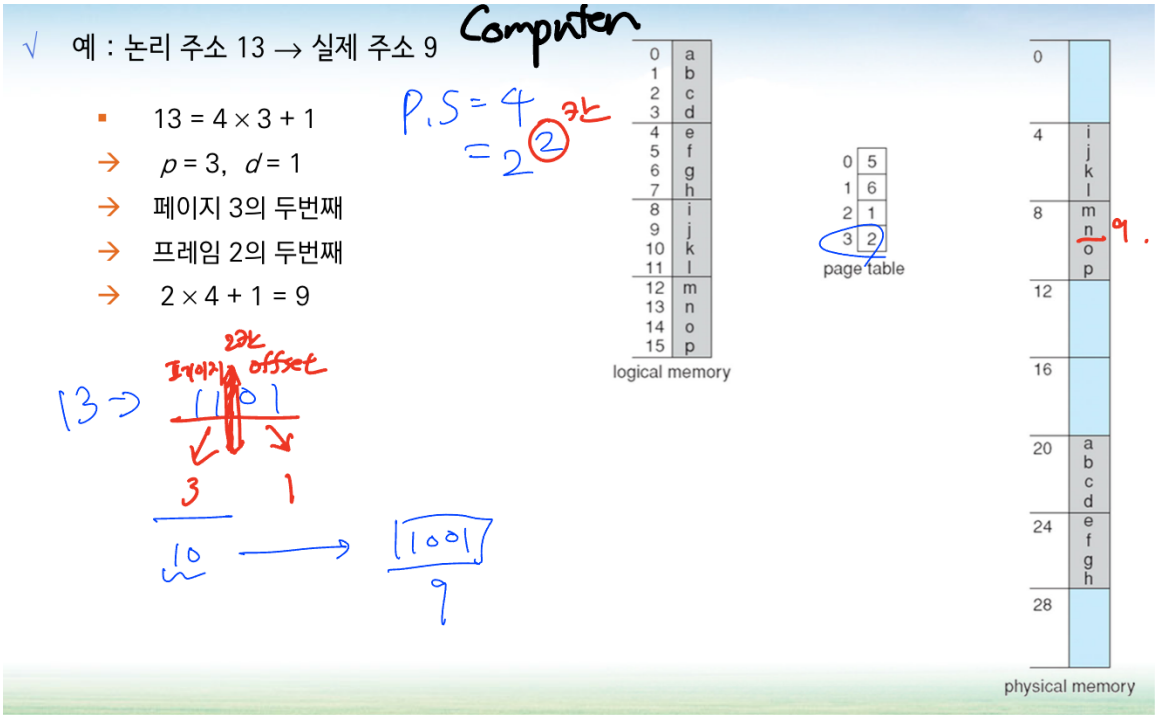
페이지 테이블

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 발생하며 동적 재배치의 형태로 페이지들이 적재됨

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

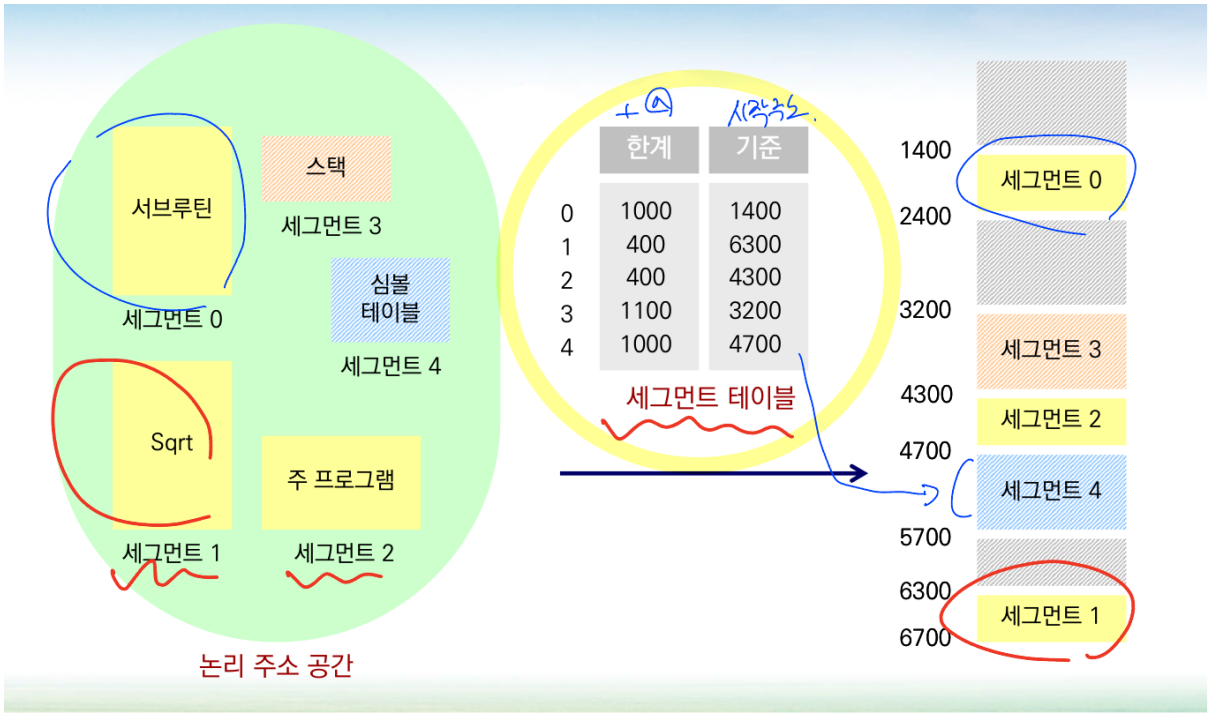
세그멘테이션

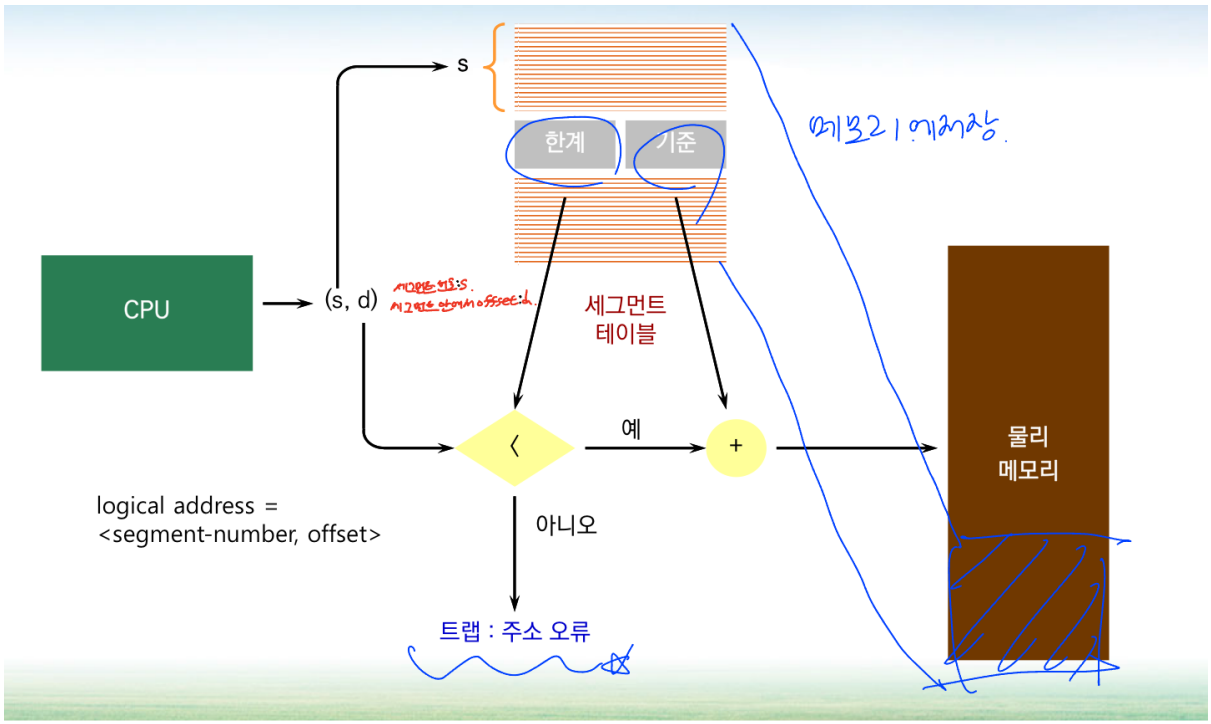
프로그래머가 생각하는 모양을 그대로 지원하는 메모리 관리 기법

각 세그먼트는 프로그램의 특정 부분을 나타냄 세그먼트마다 사이즈가 다를 수 있음

세그멘테이션을 통해 프로그램을 논리적인 단위로 구성하고 메모리를 효율적으로 사용할 수 있음

주소 변환에는 세그먼트 테이블이 사용되며 이는 메모리에 저장되어 있고, 내부 단편화가 발생할 수 있음




CPU
(s, d)
s
한계
기준
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?

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

길이 < 벗어나면 (같으면X).

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

같은크기도 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

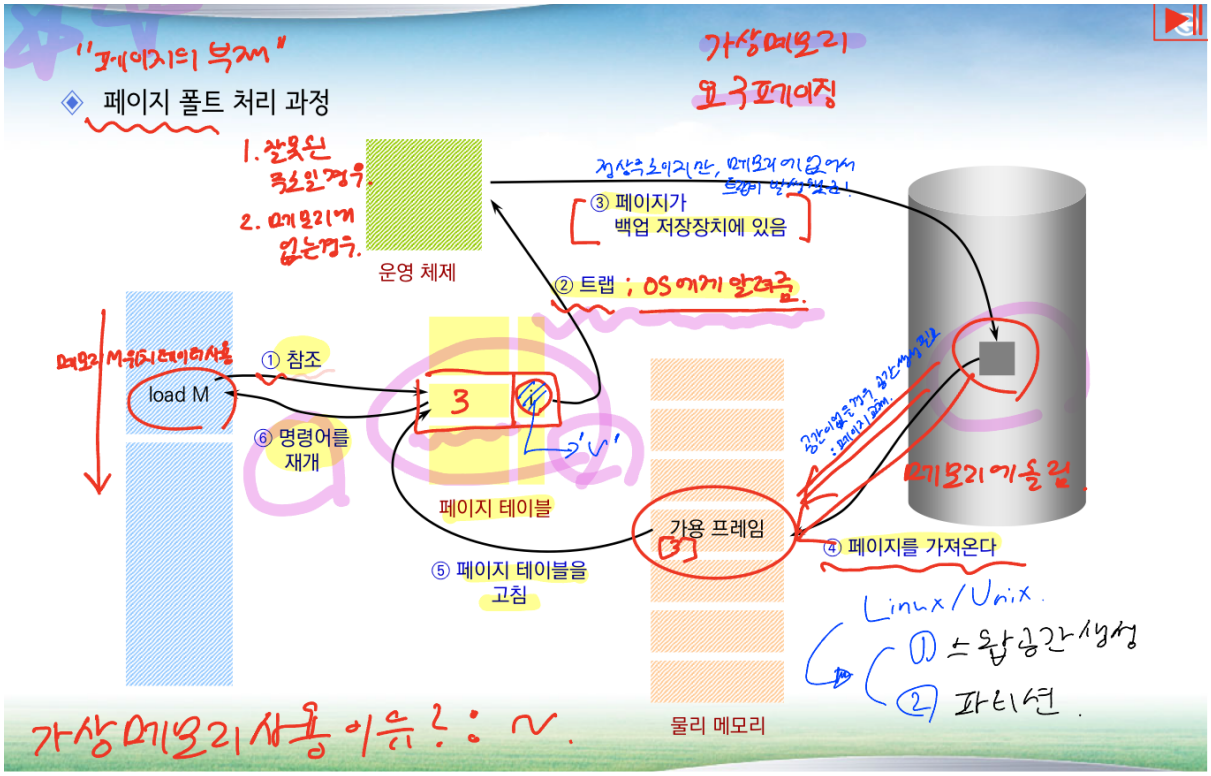
가상 주소 공간

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 주소일 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