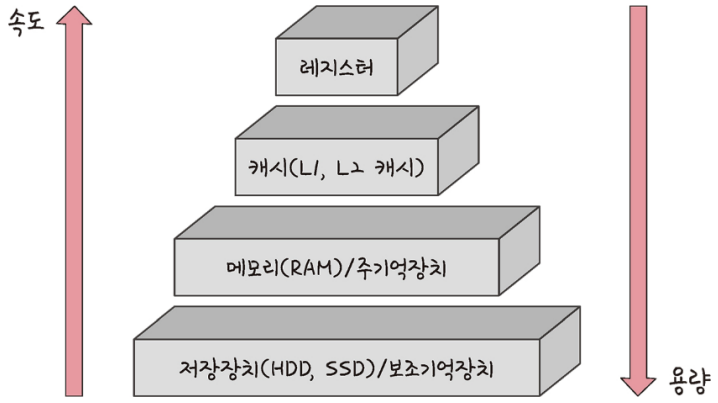
Chap 03 운영체제_3.2 메모리

3.2 메모리

> CPU는 메모리에 올라와 있는 프로그램의 명령어를 실행하는 것

3.2.1 메모리 계층



▲ 그림 3-8 메모리 계층

계층 구조의 이유 : 경제성

로딩중: 하드디스크 또는 인터넷에서 데이터를 읽어 RAM으로 전송하는 과정 중

✅ 캐시

= 데이터를 미리 복사해 놓는 임시 저장소

= 빠른 장치와 느린 장치에서 속도 차이에 따른 병목 현상을 줄이기 위한 메모리

= 다시 계산하는 시간 절약

*캐싱 계층 = 속도 차이(메모리와 CPU) 해결을 위해 계층과 계층 사이에 있는 계층

▪지역성의 원리

: 캐시를 직접 설정 할 경우 (계층x)

[시간 지역성]

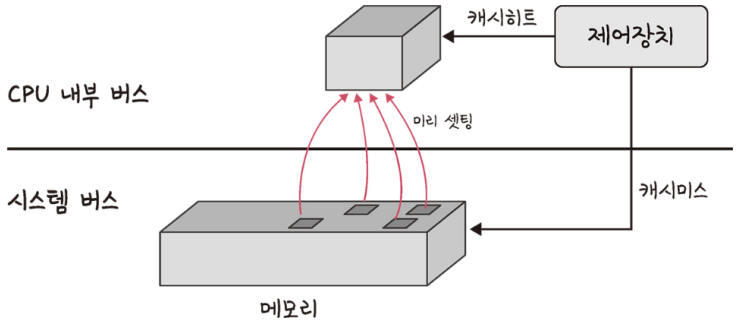
최근 사용한 데이터를 다시 접근하려는 특성

ex) for문의 변수 i

[공간 지역성]

최근 접근한 데이터를 이루고 있는 공간 or 가까운 공간에 접근하는 특성

✅ 캐시히트와 캐시미스



▲ 그림 3-9 캐시히트와 캐시미스

캐시히트 = 캐시에서 원하는 데이터를 찾음 (속도 빠름)

캐시미스 = 해당 데이터가 캐시에 없다면 주메모리로 가서 데이터를 찾아봄

▪캐시매핑

: 캐시가 히트되기 위해 매핑하는 방법

▼ 표 3-1 캐시매핑 분류

이름	설명
직접 매핑 (directed mapping)	메모리가 1~100이 있고 캐시가 1~10이 있다면 1:1~10, 2:1~20… 이런 식으로 매핑하는 것을 말합니다. 처리가 빠르지만 충돌 발생이 잦습니다.
연관 매핑 (associative mapping)	순서를 일치시키지 않고 관련 있는 캐시와 메모리를 매핑합니다. 충돌이 적지만 모든 블록을 탐색해야 해서 속도가 느립니다.
집합 연관 매핑 (set associative mapping)	직접 매핑과 연관 매핑을 합쳐 놓은 것입니다. 순서는 일치시키지만 집합을 둬서 저장하며 블록화되어 있기 때문에 검색은 좀 더 효율적입니다. 예를 들어 메모리가 1~100이 있고 캐시가 1~10이 있다면 캐시 1~5에는 1~50의 데이터를 무작위로 저장시키는 것을 말합니다.

▪웹 브라우저의 캐시

쿠기/로컬 스토리지/세션 스토리지

→ 보통 사용자의 커스텀한 정보나 인증 모듈 관련 사항들을 웹 브라우저에 저장해서 추후 서버에 요청할 경우 쓰임

→ 오리진에 종속

ex) 검색어 자동완성

[쿠키]

- 서버와 클라이언트 간의 데이터 교환 지원
- 만료 기한 있는 key-value 저장소
- **크기 제한**: 최대 4KB.
- **유효 기간**: 설정 가능 (만료 날짜까지 유지).
- **용도**: 사용자 세션 관리, 추적, 인증 정보 저장.
- **오리진**: 특정 도메인과 경로에 종속.

[로컬 스토리지]

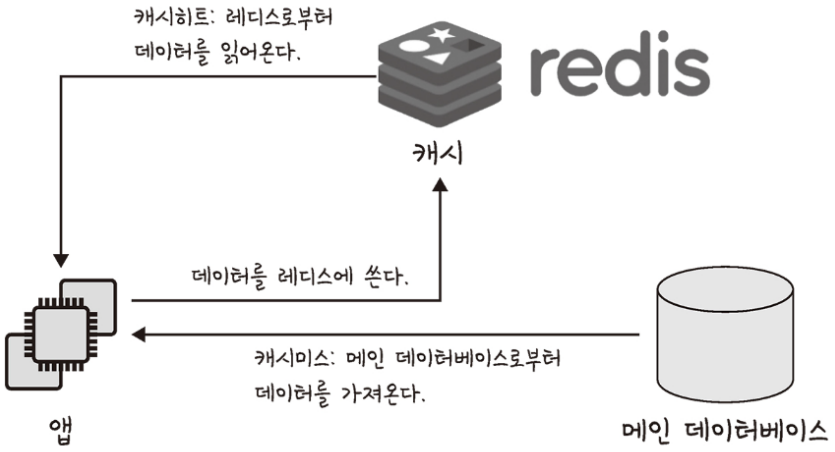
- 세션 간 데이터 유지
- 만료 기한 없는 key-value 저장소
- **크기 제한**: 약 5~10MB.
- **유효 기간**: 브라우저에 무기한 저장.
- **용도**: 사용자 설정, 장기 데이터 저장.
- **오리진**: 특정 오리진에 종속.

[세션 스토리지]

- 동일한 세션 내에서만 데이터 유지
- 만료 기한 없는 key-value 저장소
- **크기 제한**: 약 5~10MB.
- **유효 기간**: 브라우저 세션(탭/창)이 닫힐 때까지 유지.
- **용도**: 임시 데이터 저장, 탭 간 데이터 공유.
- **오리진**: 특정 오리진에 종속, 동일 탭에서만 접근 가능.

▪데이터베이스의 캐싱 계층

Redis 데이터베이스 계층을 캐싱 계층으로 두어 성능 향상



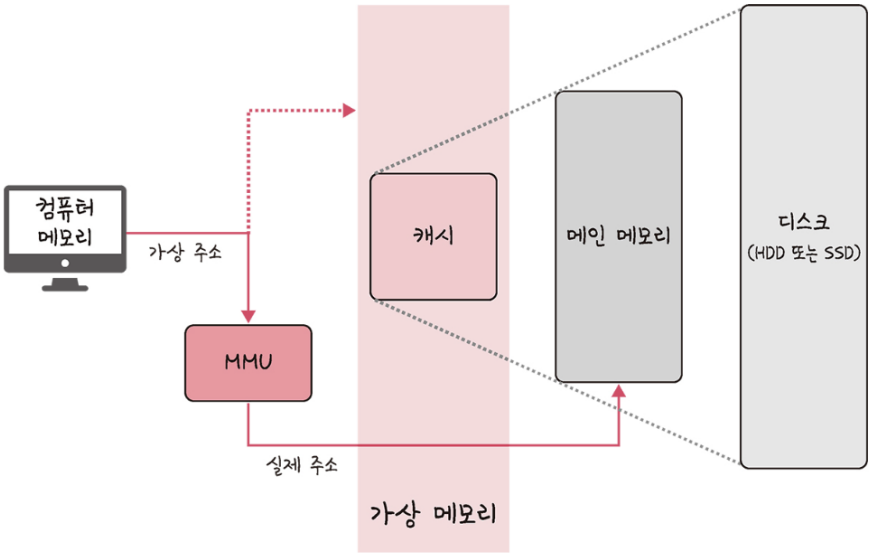
▲ 그림 3-10 레디스 캐싱 계층 아키텍처

ex) 모니터링 ...?

3.2.2 메모리 관리

✅ 가상 메모리

= 컴퓨터가 실제로 이용 가능한 메모리 자원을 추상화 → 사용자들에게 큰 메모리로 보이게 함



▲ 그림 3-11 가상 메모리

→ 가상 메모리는 프로세스의 주소 정보가 들어 있는 '페이지 테이블'로 관리

→ 속도 향상을 위해 TLB 사용

(TLB:메모리와 CPU사이에 있는 주소 변환을 위한 캐시)

▪스와핑

▪페이징 폴트

✅ 스레싱

= 메모리의 페이지 폴트율이 높은 것

→ 컴퓨터의 성능 저하 초래

→ 메모리에 너무 많은 프로세스 동시에 올라가면 스와핑 발생 → cpu 이용률 낮아짐

→ cpu 이용률 낮아지면 OS는 계속해서 프로세스 올림 → 악순환 발생

→ [해결] 작업 세트 or PFF

▪작업세트

= 프로세스의 과거 사용 이력인 지역성을 통해 결정된 페이지 집합을 만듬 → 미리 메모리에 로드

= 작업 세트 개수만큼 메모리 용량을 받지 못할 경우 그 프로세스의 메모리를 빼앗아 스레싱 방지

⇒ 탐색 비용, 스와핑 감소

ex)

과거에 접근된 페이지 번호가 `1, 2, 3, 4, 4`이고 `x`가 `5`라면 Working Set은 중복을 제거한 `1, 2, 3, 4`으로 4개이다.(4는 중복이니 제거)

▪PFF

페이지 폴트 빈도를 조절하는 방법 (상한선, 하한선 만듬)

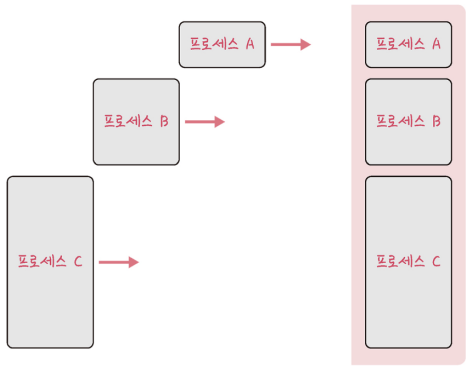
→ 상항선 도달시 프레임 늘리고 하한선 도달시 프레임 줄임

✅ 메모리 할당

메모리의 할당 크기를 기반으로 할당 → 연속 할당/불연속 할당

▪연속 할당

= 프로그램이 통째로 메모리 한 장소에 올라가는 것



▲ 그림 3-13 **연속 할당**

[고정 분할 방식]

메모리를 미리 나누어 관리 (영구적 분할로 나누는 것)

→ 메모리가 미리 나눠짐으로 융통성 없음 , 내부 단편화 발생

*융통성 = 메모리 관리의 유연성

*내부 단편화 = 프로그램이 필요로 하는 메모리보다 큰 블록을 할당받게 되어 사용되지 않는 공간 발생

[가변 분할 방식]

매 시점 프로그램의 크기에 맞게 동적으로 메모리 나눠 사용

→ 내부 단편화 발생x, 외부 다편화 발생 o

→ 중간에 프로그램 종료되어 중간에 빈 공간 발생시 새로 올릴 프로그램이 크기가 맞지 않을 수 잇음

*외부 단편화 = 메모리를 나눈 크기보다 프로그램이 커서 들어가지 못하는 공간 발생

ex) 100MB → 55,45로 나눴지만 프로그램 크기가 70으로 들어가지 못하는 상황

▪불연속 할당

= 메모리를 연속적으로 할당x

[페이징 기법]

= 메모리를 동일한 크기의 페이지로 나누고 메모리의 서로 다른 위치에 프로세스 할당

→ 홀의 크기가 균일하지 않은 문제 해결

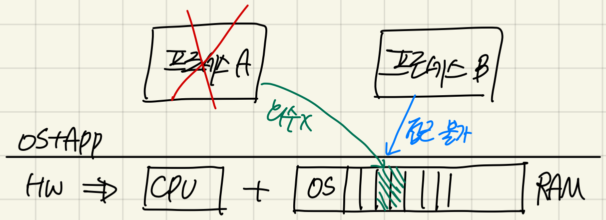
→ 주소 변환 복잡 (페이지 테이블 조회 후 변환 수행)

[세그멘테이션]

= 페이지 단위가 아닌 의미 단위(세그먼트)로 나누는 방식

가상메모리

Step1) "MS-DOC" OS 사용 시절



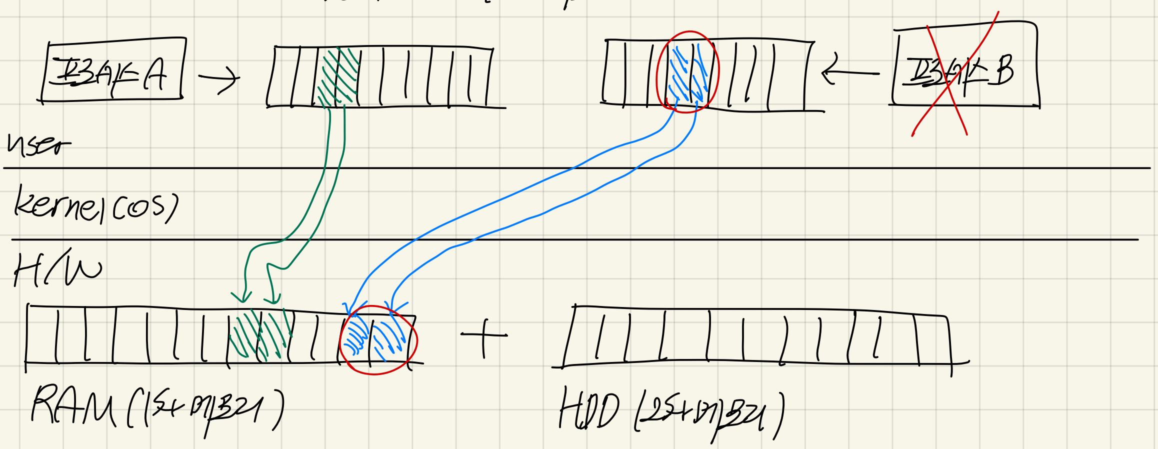
① 메모리 회수 불가
② 메모리 낭비 과도
→ 재부팅으로 해결

HW, Application 관련↑, OS관련↓
전체 시스템에 문제 많음.

Step2) 보호모드 삽입 (커널영역 등장)
+가상메모리 등장

가상메모리 구조

*stack, head, code 모두 가상메모리

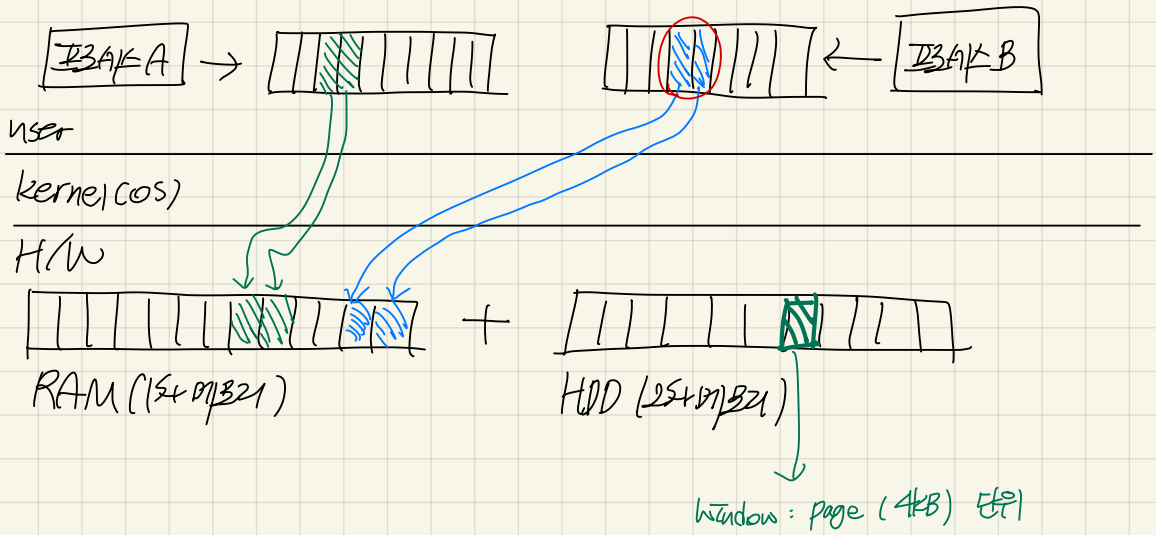


① 프로세스 B dead
② 가상메모리의 실제 메모리 위치 → OS는 알고있음
③ OS가 RAM의 (프로세스 B가 사용하던) 메모리 사용 가능하도록 처리 (= 회수)
④ 프로세스 죽/살 OS는 손상X

⇒ 가상 Memory의 사용 이유

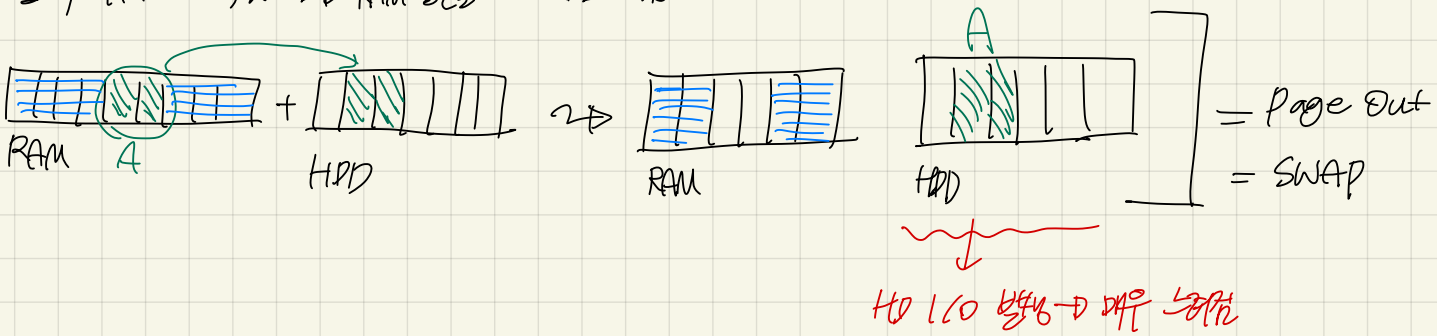
*가상메모리 사용의 예)
애플 phone → 프로세스 동시에 사용 + 중간에 죽이기 ... 등등 해도
성능 변화 X

가상 메모리 발표

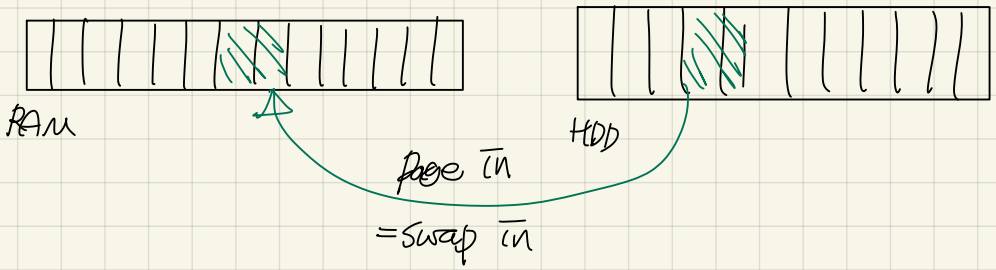


① 메모리 부족 현상 발생

② 프로세스 A가 동작 X → RAM 공간을 HDD로 이동



③ RAM의 공간이 확보될 경우
(이동시킨 프로세스 A 재실행의 상황)

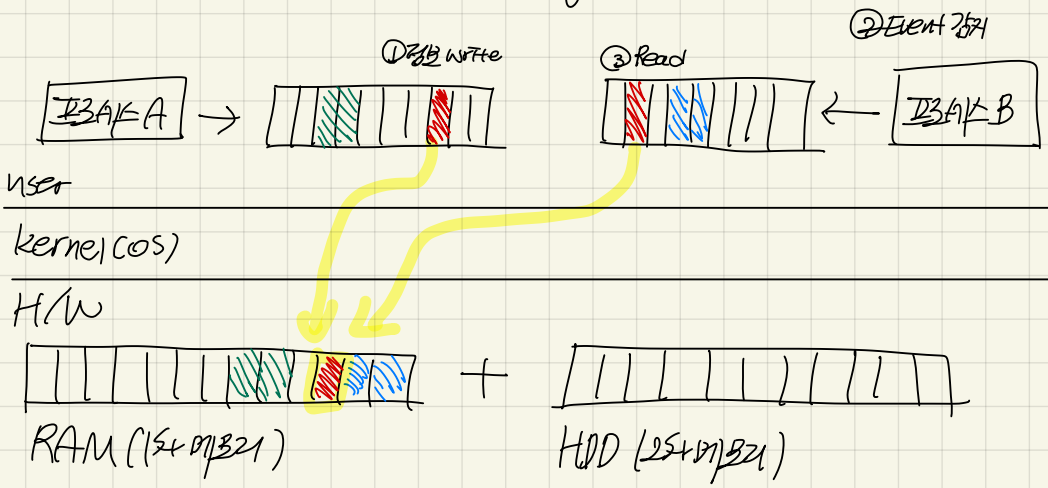


" 주기억장치 (RAM)과 하드디스크 둘에 하나의 논리적 메모리를 유지할 수 있다. "

→ 현재 사용하는 메모리는 OS가 관리 → 메모리 용량만 충분하면 됨.

가상메모리와 IPC

↳ Shared Memory



* 가상메모리는 독립적 (침범 불가)

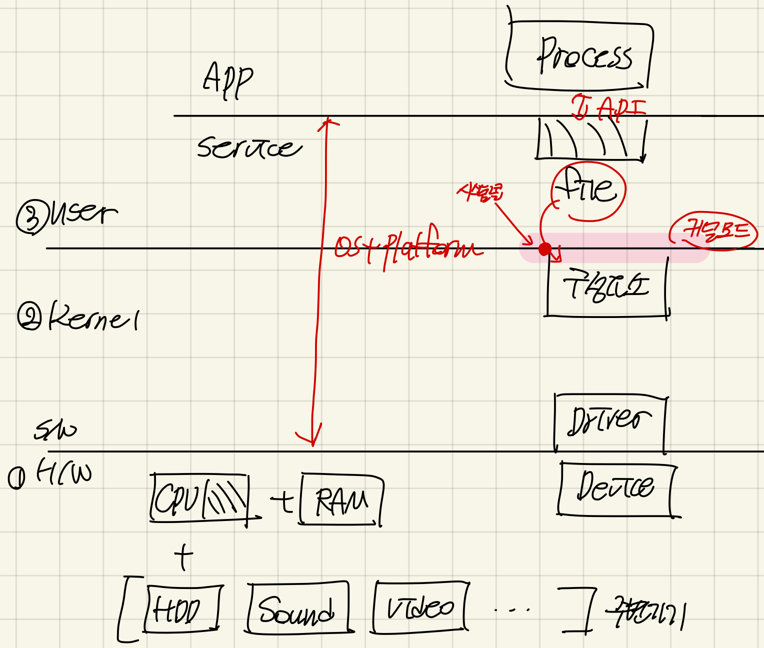
→ 의도적으로 공유 가능

①, ②, ③ ⇒ 프로세스간의 정보를 운영자 수준 보다 안에 의도적으로 같게끔 mapping

→ 자동으로 통신 가능

ex) "복사 붙여넣기" 가능

컴퓨터



ex1) print "hello world"

① API 중 print 함수로 call

② API 내부에서 Interface를 통해 함수가 내려감

↓ (커널 모드 진입)

③ system call 중 print 함수 이므로 write 함수가 call됨.

④ Device driver를 제어하기 시작

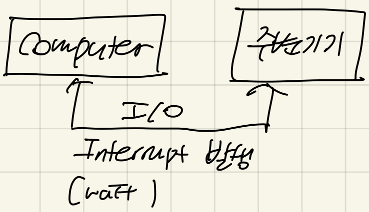
IRQ ↓ 출력. ↑ call했던 다음 recall

다시 Interrupt 발생

* 이때 ↺ 활동선 프로세스가 대기 할 경우 = blocking I/O

↺ 활동안 프로세스 대기X, 다른 작업 진행 = un blocking I/O

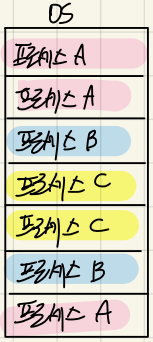
Interrupt란?



가상메모리 관리 (페이징)

연속메모리 할당 → 문제1) 외부단편화

문제2) 물리 메모리보다 큰 프로세스 실행 불가.



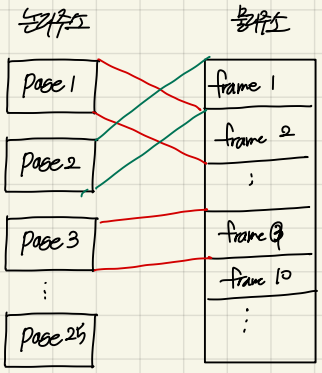
"실행되고자 하는 프로그램을 일부만 메모리에 적재"

① 페이징이란?

프로세스를 일정 크기로 자르고 → 불연속적으로 할당

- 논리주소 공간을 page 단위로 자름
- 물리주소 공간을 frame 이라는 page와 동일한 단위로 자름

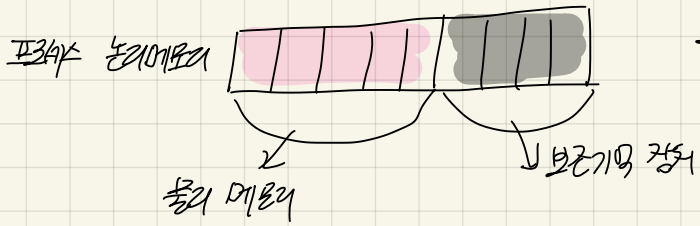
⇒ Page를 frame에 할당하는 가상메모리 관리 기법



→ 외부 단편화 X

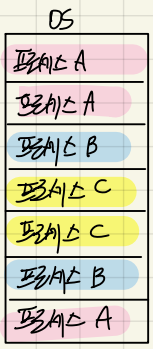
② 페이징에서의 스와핑

- 프로세스 단위X, Page 단위로 스왑 in/out
- 메모리에 적재될 필요 없는 페이지들은 보조기억장치로 swap out
- 실행 필요 page는 메모리로 swap in



⇒ 물리 메모리 보다 큰 프로세스도 실행 가능

③ 페이지 테이블



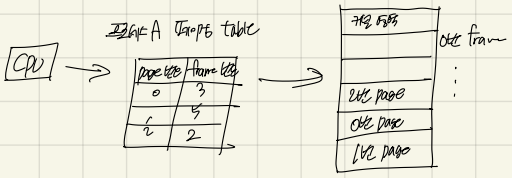
→ 프로세스를 이루는 Page가 어느 frame에 적재되어 있는지 관리 필요.

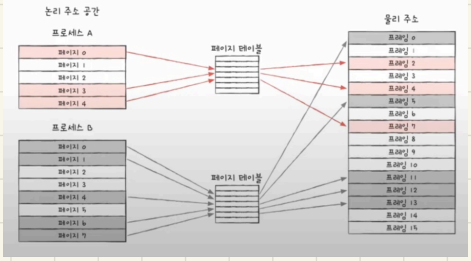
⇓

페이징 table

= page 번호와 frame 번호를 짝지어주는 이정표

= 프로세스마다 페이지 table 有

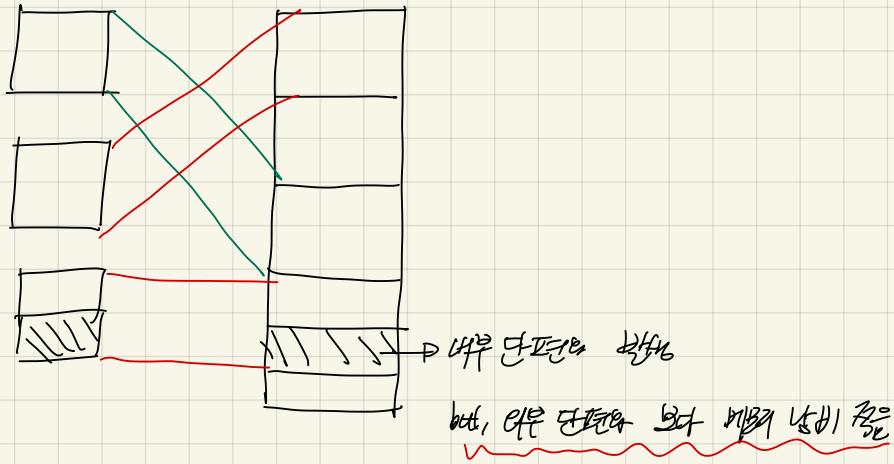




→ CPU 입장에서 바라본 논리주소는 연속적

→ CPU는 논리 주소를 순차적으로 실행해갈 뿐.

④ 내부 단편화 발생 가능



BUT, 외부 단편화 보다 메모리 낭비 적음

⑤ TLB

↳ CPU 페이지 table의 캐시 메모리.

※ 페이지 table이 메모리에 있을 경우 메모리 접근 시간 X2배 → 캐시 메모리 이용

자주 사용하는 것 위주로

HTTP

↳ text ⇝ hypertext