cp 9. 페이징 - 분할할당 / 고정크기 / 페이징

페이지 프로세스의 주소공간을 0번지부터
동일한 크기의 페이지로 나눔.

부름

프레임 : 물리메모리도 0번지부터 페이지크기로 나눈걸 프레임으로

페이지와 프레임에 번호붙임
페이지크기 - 주로 4KB
페이지 테이블 - 각 페이지에 대해 페이지번호 & 프레임번호를 1:1로 저장한 테블

<<페이징 기법>> p. 115 관련

프로세스의 주소공간과 물리메모리를 페이지단위로 분할.
프로세스의 각 페이지를 물리메모리의 프레임에 분산할당하여 관리

<페이징 우수성> ① 용이한구현 ② 높은이식성 ③ 높은융통성
④ 메모리활용과 시간 오버헤드면에서 우수
- 외부단편화 X, 내부단편화 발생하지만 매우작음
(예) 페이지크기가 4KB라서 발생해도 작다

- 홀선택알고리즘을 실행할 필요 X

p. 116

프로세스공간: 4GB
페이지크기: 4KB
프로세스크기: 페이지개수 X 4KB = 2KB

프로세스가 시스템호출 실행 할때,
커널공간 페이지 K에 담긴커널코드 실행

커널코드도 논리주소로 되어 있으며, 시스템 호출을 통해
커널코드가 실행될 때, 현재 프로세스의 페이지테이블을 이용하여 물리주소로 변환

32비트 CPU의 경우, 페이지 크기가 4kB인 경우

① (페이지크기와 상관없이) 물리주소의 범위는 $0 \sim 2^{32}-1$ 이다

물리메모리의 최대크기는 $2^{32} = 4GB$

② 프로세스 주소공간의 크기는 2^{32} (한 주소당 1byte) = 4GB

↑ → (물리메모리와 관계없음)

③ 한 프로세스당 최대 페이지 개수?

$4GB / 4kB = 2^{32} / 2^{12} = 2^{20}$ 개

④ 프로세스당 하나의 페이지테이블이 있는데, 크기?

페이지테이블 항목크기 32비트 (4B) - (각항목에 프레임번호 ~~있다~~)

4 바이트 $\times 2^{20} = 2^{22}$ 바이트 $= 4MB$

각 페이지가 2^{20}

페이지테이블 $0 \sim 2^{20}-1$ 까지 주소가 있음

이게 4byte라고 한다면, $2^{20} \times 4byte$ = 페이지테이블의 크기

⑤ 응용프로그램이 하나의 프로세스라고 하자, 프로그램의 최대 크기? = 운영체제가 설정한 사용자공간크기와 동일.

⑥ 페이지테이블은 대부분 비어있는 희소테이블로 낭비가 있다

⑦ 페이지테이블은 메모리에 저장

⑧ 커널코드는 물리주소, 즉 커널은 실행시 물리주소로 변환필요

페이지테이블은 현재프로세스의 페이지테이블이 사용.

고정크기분할이라 외부단편화X

페이징 - 외부단편화 X , 내부단편화 O

스택·힙에서 생성되는 페이지는 계속변해서 제외한다면,
프로세스 마지막 페이지에만 단편화 발생
평균크기(4KB 가정) = 페이지의 ½크기 ⇒ 2KB

32비트 CPU에서, 페이지 크기가 2KB
메모리 1GB, 프로세스 A는 사용자공간에서 54321 바이트

① 물리 메모리 프레임크기 = 페이지크기 = 2KB

② 〃 프레임개수 = 메모리 1GB / 프레임크기 2KB = $2^{30}/2^{11} = 2^{19}$

③ 프로세스주소공간크기 = 2^{32} = 4GB
페이지 개수 = 2^{32} 프로세스주소공간 / 2KB (2^{11}) = 2^{21}

④ 프로세스 A의 페이지 개수?
프로세스 A의 실제크기 = 54321 바이트
54321 바이트 / 2KB(2048) = 26.5
1KB 1024 ⇒ 27개 페이지
⇒ 물리프레임도 27개

(32비트)
⑤ 페이지 항목 크기가 4 바이트 → 페이지항목크기 X 페이지개수
프로세스 A의 페이지 테이블 크기? = 페이지항목크기 X 페이지개수

2^{21} X 4 바이트 = 2^{23} 바이트 = 8MB
페이지의개수 2^2 8MB는 최소테이블

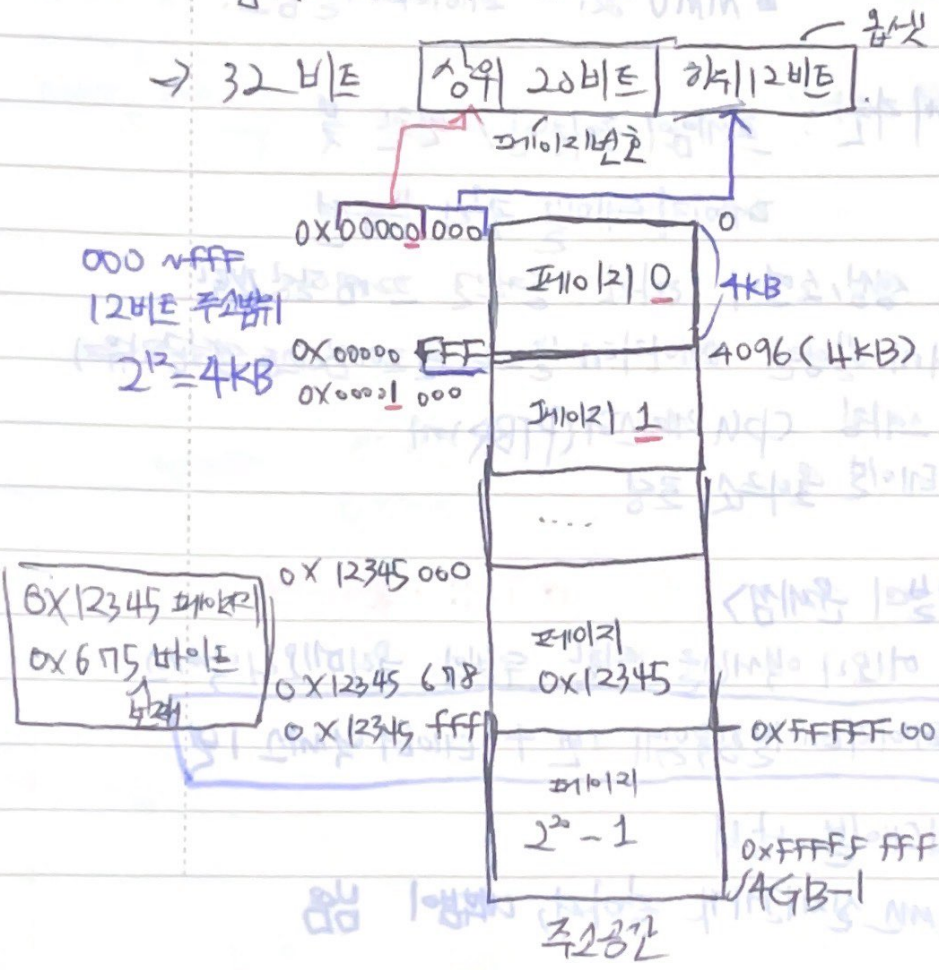
⑥ 페이징에서 단편화 메모리 평균크기?
페이지의 반 이므로 ~~4KB~~ 1KB

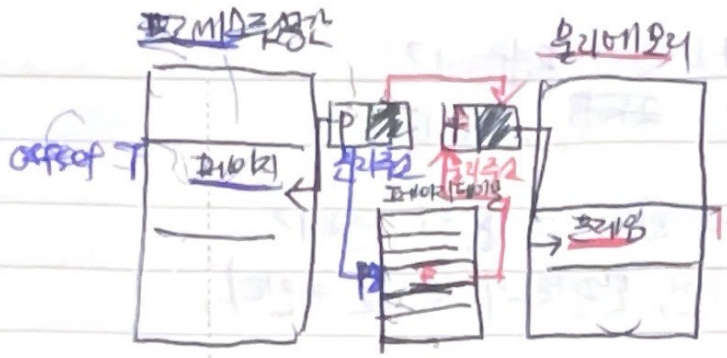
⑦ 페이지크기와 단편화의 크기X의 관계?
페이지크기가 커지면, 단편화의 크기도 커진다.

⑧ 페이지크기와 페이지 테이블의 크기 관계?
// 가 크면, 그페이지 개수 작아지고.
페이지 테이블크기도 작아짐

<< 페이징의 논리주소 >> [페이지번호 p | 옵셋]

4KB = 페이지크기, (=2^{12}) 각 바이트 주소는 2^{12}개
⇒ 옵셋크기는 12비트
⇒ 32비트 [상위 20비트 | 하위 12비트] ← 옵셋
페이지번호





즉, offset은 그대로, 프레임번호만 바뀌는 식이다.

<< 페이징 구현 >>

① 하드웨어 지원 ; ■ CPU 지원 : 페이지 테이블이 있는 메모리의 주소를 가진 레지스터 필요 (= Page Table Base Register PTBR)

⇒ 운체에 의해 제어

■ MMU 장치 - 페이지테이블 탐색.

② 운영체제 지원 : 프레임의 적절할당 / 반환 및 페이지 테이블 관리 기능구현.

- 프로세스 생성, 소멸에 따라 동적으로 프레임 할당 / 반환.
- 물리 메모리에 할당된 페이지테이블과 빈 프레임리스트 생성 관리유지
- 컨텍스트 스위칭 CPU 레지스터 (PTBR)에 페이지 테이블 물리주소 로딩

< 페이지테이블의 문제점 > ↱ TCB 사용으로해결

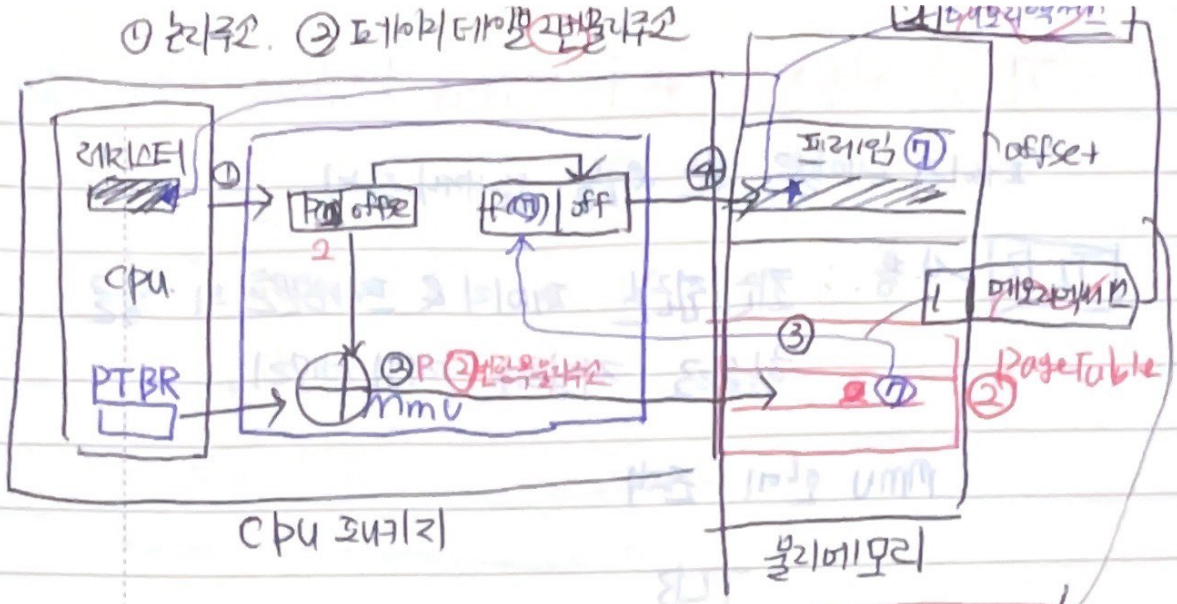
1) 한번의 메모리 액세스를 위한 두번의 물리메모리 액세스

페이지테이블항목읽기 1번 + 데이터 액세스 1번

2) 페이지테이블 낭비

프로세스 실제크기가 작아서, 대부분이 남음

① 논리주소, ② 페이지테이블 물리주소



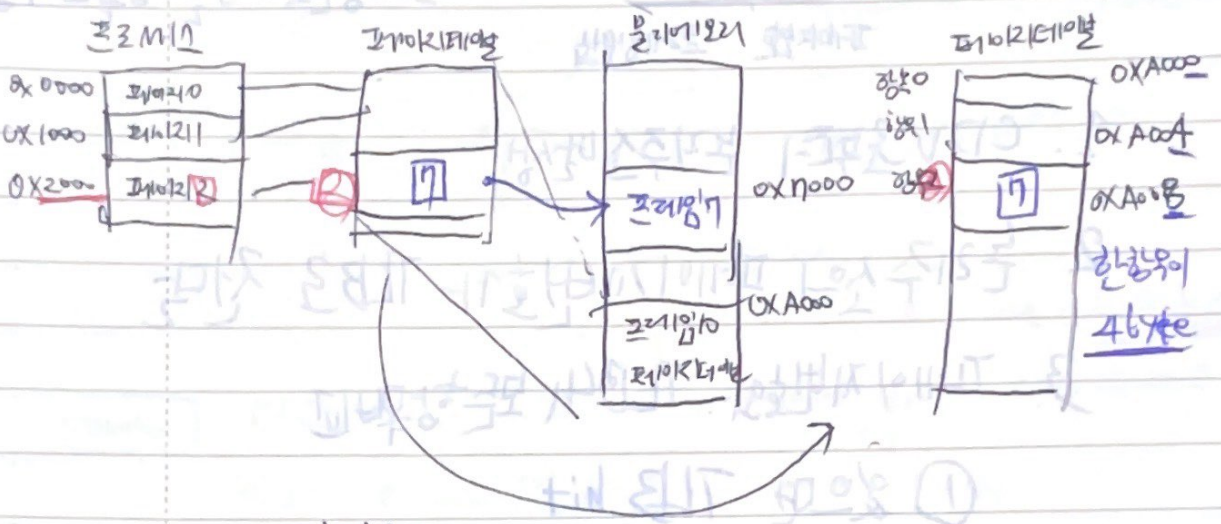
③ 페이지테이블항목 프레임 읽기

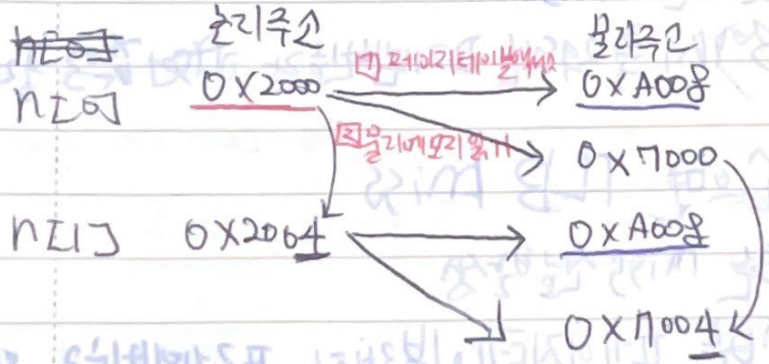
④ 물리주소

두번 메모리 접근

1 페이지테이블 접근

2 물리메모리 읽기





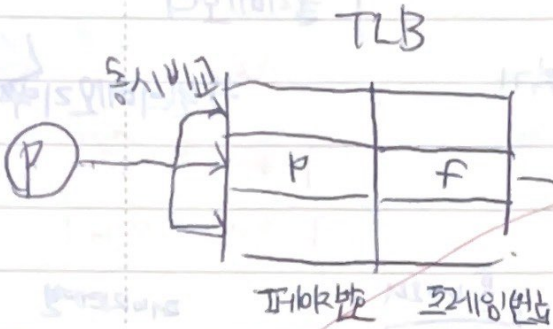
이렇게 두번 접근할 필요가 있을까?

TLB를 활용한 오버이 물리메모리 액세스 해설

페이지 테이블을 읽는 시간을 없애기 위해

TLB 사용 : 최근 접근한 페이지 & 프레임번호의 쌍을 항목으로 저장하는 캐시 메모리

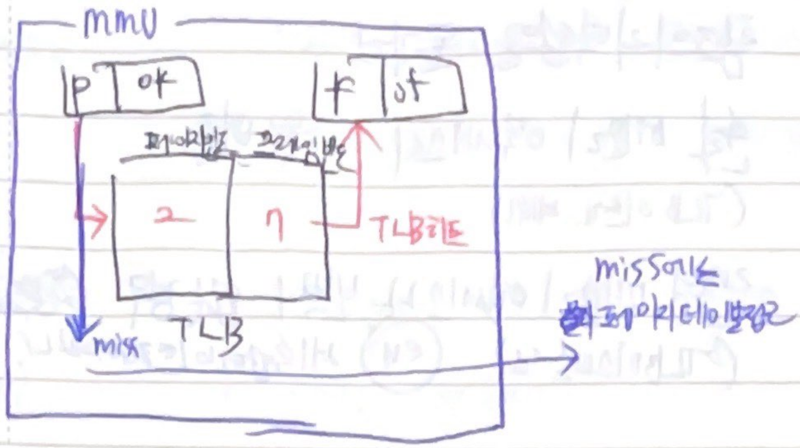
mmu 안에 존재



페이지번호 p와 프레임번호 f 쌍을 항목으로 저장

1. CPU 로부터 논리주소 발생
2. 논리주소의 페이지번호가 TLB로 전달
3. 페이지번호와 TLB내 모든항목비교
   - ① 있으면 TLB hit
     TLB에서 출력되는 프레임번호랑 오프셋 값으로 물리주소완성
   - ② 없으면 TLB miss
     - TLB는 miss 신호발생
     - mmu는 페이지테이블로부터 프레임번호를 읽어와서 물리주소 완성
     - 미스난 페이지의 항목을 TLB에 삽입

이전과 거의 비슷한데…



32비트 CPU, 4KB 페이지

n[0] 논리 0X2000 → TLB miss → TLB   물리주소 0X A008
0X7000 ← n[0] 물리주소

n[1] 0X2004 → TLB hit → [2 | 7] 0X7004 ← n[1] "

그럼 8000바이트는?
예) n[2000]

페이지 4KB일때,

$$= 4 \cdot 2^{10} B = 4 \cdot 1024 = 4096 byte$$

1page당 크기임

페이지2 n[0] 0X2000 miss → [ ] 0X A008 0X7000 ← 물리

n[1] 0X2004 hit → [2 | 7] 0X7004 n[1] 물리

8+4 = 12 = C

페이지3 n[1024] 0X3000 miss → [2 | 7] 0X A00C
0X 9000 ← 물리

n[1025] 0X3004 hit → [2 | 7, 3 | 9] 0X 9004

TLB 요론

- TLB는 참조의 지역성으로 효과적.
  순차 메모리 액세스시 실행 빠름.
  (TLB 히트가 빈번)
  랜덤 메모리 액세스나 반복이 없는 경우 실행 느림.
  (TLB 미스 발생) (예시) 계속 업데이트 해야 하니.

- TLB 성능 높이기.
  - TLB 항목(크기) 늘리기
  - 페이지 크기 늘리기. 그러므로 TLB 히트 증가 → 성능 향상
    〃 내부 단편화 증가 → 메모리 낭비

- TLB Reach
  - TLB 성능 지표 / TLB 항목 수 × 페이지 크기
  - 모든 TLB 항목이 채워졌을 때 미스 없이 작동되는 메모리 액세스 범위

< TLB 이용한 컨텍스트 스위칭 >

- 컨텍스트 스위칭 CPU가 실행되는 프로세스가 변경되므로, 새로운 프로세스의 주소 공간에서 실행해야 함.
  → 새로운 페이지 테이블 사용

- TLB 고려한 컨텍스트 스위칭 과정
  1. CPU의 모든 레지스터를 PCB에 저장
  2. 새 프로세스의 PCB에 있는 페이지 테이블 주소를 MMU(CPU) 내 PTBR에 적재
  3. TLB 지우기 (예) 이전 프로세스의 매핑정보가 있음 / 비우면 다른 프로세스로 매핑되는 현상 발생
  4. 새 프로세스 컨텍스트를 PCB에서 CPU에 적재