[Chapter 9]

경영학과/2018100233/김정민

I. Memory Management

A. 메모리를 관리 목표

1. 편리한 추상화(abstraction)를 제공하기 위해
   - abstraction은 프로그래밍에 있어서 중요하게 생각하는 가치이다.
   - 메모리는 추상화가 가장 잘 된 영역이다.
2. 여러 프로세스가 실행되고 있을 때 한정된 메모리 자원을 할당하기 위해
   - 최소한의 오버헤드, 최대 성능
3. 멀티프로세스 환경에서 프로세스 간의 독립성 보장하기 위해
   - Virtual Memory (VM) : 프로세스마다 격리된 메모리를 가진다.

B. 메모리 관리가 필요하게 된 배경

1. Batch programming
   - Batch system에서는 메모리 관리의 필요성이 낮았다.
   - 프로그램은 물리적 메모리 주소를 직접적으로 접근했다.
   - OS가 할 일을 로드해주고, 실행시켜주고, 메모리에서 내리는 것까지 해준다.
2. Multiprogramming
   - 동시에 여러 프로세스가 메모리 상에서 메모리에 접근한다.
   - 여러 방법으로 관리할 수 있다. (partitioning, paging, segmentation, 등)
   - 요구조건
     - Protection : 한 프로세스의 데이터를 다른 프로세스가 접근하지 못하게 한다.
     - Fast translation : 데이터를 빠르게 찾을 수 있어야 한다. (효율성)
     - Fast context switching : 메모리를 업데이트하는 과정은 빨라야 한다. (효율성)

C. Issues

1. 멀티프로세스 환경 지원
   - 각각의 프로세스는 논리적으로 연속적인 메모리 공간을 가진다.
   - 그 공간의 크기는 가변적이다.
2. 할당받은 physical memory보다 더 많은 공간을 메모리로 사용할 수 있다.
   - 모든 메모리가 동시에 사용되는 것은 아니라는 점에 초점
   - spatial, temporal locality
3. Protection
4. Sharing
5. 프로세스마다 여러 공간을 지원한다.
6. 성능
   - memory access time 최소화
   - memory reference overhead 최소화
   - context switching overhead 최소화

D. Solution : Virtual Memory (VM)

1. VM은 데이터가 저장된 physical memory(PM)의 실제 주소를 모르고도 프로그램이 실행될 수 있게 해준다.
   - 프로그램은 필요한 RAM보다 더 적은 RAM을 가진 장치에서 실행될 수 있다.
2. 많은 프로그램은 해당 코드나 데이터를 동시에 모두 필요로 하지 않는다.
   - 절대 접근하지 않는 코드 혹은 데이터가 있을 수도 있다.
   - 접근할 필요가 없는 데이터를 위해 메모리를 할당할 필요는 없다.

- OS가 할당될 메모리 양을 런타임에 따라서 결정해준다.

3. 프로세스들을 격리시킨다.
   - 한 프로세스의 메모리의 주소를 다른 프로세스가 볼 수 없다.

II. Binding of Instructions and Data to Memory (메모리 주소 결정 시점)

A. Compile time

1. 컴파일할 때 물리적 메모리의 주소를 직접 할당해서 바인딩한다.
   - 데이터의 주소가 컴파일러에 의해 미리 결정된다.
   - 해당 파일은 실행 파일에 직접적으로 바인딩된다.
   - 로지컬 메모리의 주소는 상수 값(절대주소)을 가진다.
2. 단점
   - 직접적으로 물리적 메모리 주소를 할당하기 때문에, 오버랩되는 메모리를 할당받은 프로세스는 실행할 수 없다.
   - 불필요한 매핑까지 수행해서 로드타임이 길어진다.
3. 장점
   - 실행 속도는 빠르다.
4. 위와 같은 이유로 사용되지 않는다.
5. Virtual memory는 고려하지 않는 개념
   - 임베디드 시스템에서는 펌웨어 레벨로 활용한다. (MMU 대신 MPU)

B. Load time

1. 로지컬 메모리의 주소를 상대 주소로 정해진다.
   - 실행 파일에 데이터의 주소는 상대주소(offset)으로 저장되어 있다.
   - 로드할 때 메모리를 보고 sequential하게 메모리에 넣을 수 있는 곳으로 넣어준다.
     - base address는 프로세스마다 달라진다.
2. 로드 타임에 base address와 상대주소를 연산해서 메모리에 올리고, 코드 보고 처리한다.
3. 장점
   - 컴파일 이후에도 피지컬 메모리의 주소를 변경할 수 있다.
4. 단점
   - 상대주소와 base address를 통해 실제 메모리의 주소를 계산하는 과정을 모두 로드 타임에 한다.
   - 로드타임이 너무 길어진다.
     - new 상태에서 ready 상태가 되기까지 오래 걸린다. (add 연산)

C. Execution time

1. 로지컬 메모리의 주소 -> 상대 주소, 피지컬 메모리에 올리는 주소 -> 상대 주소
   - 데이터의 주소가 실행하면서, 필요한 코드나 데이터만 주소 연산을 통해 처리한다.
   - 프로그램이 실행되는 동안 데이터의 주소가 동적으로 결정된다.
2. 단점
   - 로드타임이 감소하는 대신, 실행 시간이 증가한다.
3. 해결
   - 주소 처리 연산 오버헤드를 줄이기 위해 MMU(Memory Management Unit)이 탄생했다.
   - MMU는 주소 연산에 특화되어 있는 모듈로, 독립적으로 주소 연산을 한다.
   - 실행 시간 오버헤드 최소화

D. MMU(Memory Management Unit)

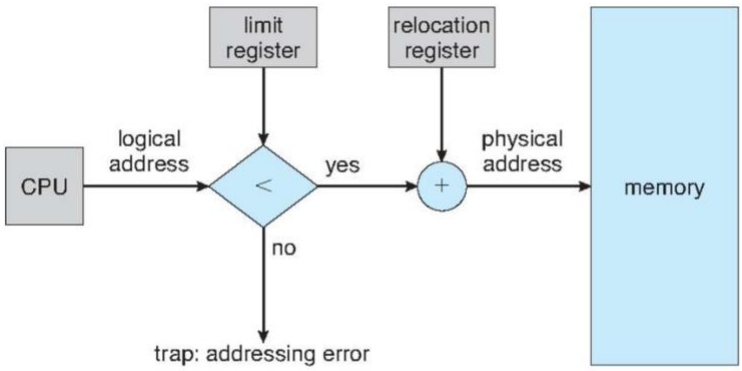
1. 로지컬 주소를 피지컬 주소로 매핑해주는 하드웨어 장치
2. CPU 안에 존재한다.

3. 유저는 메모리의 피지컬 주소를 볼 수 없다. 오로지 로지컬 주소만 볼 수 있다. (abstraction good)
4. 변수의 주소를 print() 함수로 여러 번 출력해보면, 같은 로지컬 주소가 계속 나온다.
   - 하지만 실행할 때마다 피지컬 메모리의 위치는 달라질 수 있다.

III. Memory Allocation

A. Contiguous Allocation

1. 로지컬과 피지컬 모두 연속적으로 메모리 할당
2. Address translation
   - Physical address = logical address + relocation register



   -
3. Memory protection (limit register)
   - limit register를 통해 overflow, underflow 여부를 검사한다. (그래서 부등호를 사용)
   - 다른 데이터 영역을 침범했는지를 보는 것
   - 다른 프로세스의 데이터 영역을 침범하면 프로세스가 죽어버린다.
4. Relocation register
   - Logical address에 특정 값을 더해주어 physical address를 연산한다.
5. 단점 (Hole)
   - 모든 프로세스가 피지컬 메모리의 빈 칸에 딱딱 들어맞는 데이터 크기를 가지지 않는다.
   - 따라서 중간중간 남는 공간이 생긴다.
   - external fragmentation
   - 비어있는 공간의 크기가 못쓸 정도로 작은데, 그런 공간이 계속 발생한다면 낭비되는 메모리가 많아진다.
     - 메모리 공간을 shifting하기에는 오버헤드가 너무 크다.
     - 싹 다 디스크에 내렸다가 다시 올려야 하는데, 이는 OS 오버헤드가 너무 크다.
6. 해결 (Dynamic Storage-Allocation Problem)
   - 빈 메모리 공간(Hole)에 어떻게 데이터를 올릴 것인지?
   - First-fit
     - base address부터 보다가 가장 먼저 만나는 hole(들어갈 수 있는 크기)에 할당한다.
   - Best-fit (홀의 개수, 위치, 크기 등에 대한 OS의 관리 필요)
     - 들어갈 수 있는 hole 중에서 가장 작은 것에 할당한다.
     - 모든 리스트를 검색해봐야 한다. (hole의 크기가 정렬되어 유지되지 않는다면)
     - 그나마 가장 작은 홀을 만든다.
   - Worst-fit (홀의 개수, 위치, 크기 등에 대한 OS의 관리 필요)
     - 가장 큰 홀에 할당한다.
     - 이것 또한 모든 리스트를 검색해봐야 한다. (hole의 크기가 정렬되어 유지되지 않는다면)
     - 가능한 가장 큰 홀이 만들어진다.

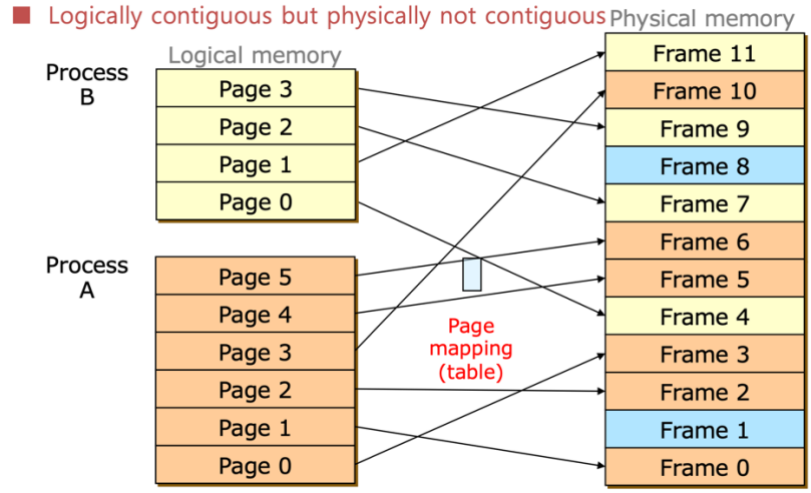
7. First-fit과 best-fit이 속도와 저장공간 utilization의 측면에서 worst-fit보다 좋다.
8. 여기까지는 contiguous allocation의 관점이다. 어찌됐건 홀(fragmentation)은 발생하기 때문에 더 최적화가 필요하다.

B. Fragmentation

1. External fragmentation
   - 프로세스 외부에 생기는 '틈' 공간
2. Internal fragment
   - 프로세스 내부에 생기는 쓰이지 않는 공간
   - 필요한 메모리보다 조금 더 할당받았을 경우
3. Compaction
   - External fragmentation을 모아서 하나의 큰 빈 공간으로 만든다.
   - relocation이 동적이고, 실행 시간에 결정될 때에만 가능한 개념이다.
   - I/O problem
     - 오버헤드가 크다.
     - I/O가 너무 많이 일어난다.
     - 그래서 contiguous allocation 안 쓴다.

IV. Paging

A. Page (logical memory) -> frame (physical memory)

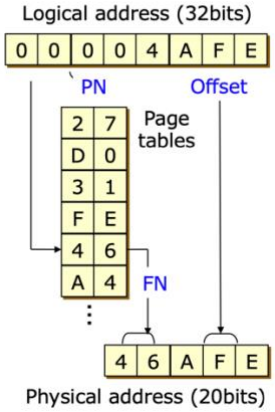


1. 정해진 크기만큼 page(슬롯)를 나눈다.
2. Page 내에서 internal fragmentation이 생긴다.
   - external fragmentation보다 낫다. (메모리 낭비 적다.)
   - 성능을 위해 이 정도는 감수한다.
3. 페이지는 같은 크기의 frame과 매핑된다.
   - frame은 연속적이지 않게(randomly) 할당된다.
   - Page table가 매핑 정보를 담고있다.

B. Address Translation

1. Logical address
   - <page number::offset> = (p,d)
   - page number : 페이지 테이블에서의 인덱스
     - 로지컬 단에서는 붙어있다.
     - 페이지 안에서도 원하는 데이터로 가기 위해 offset 활용
   - 페이지 테이블은 frame number를 결정해준다.

2. Physical address
   - <frame number::offset> = (f,d)
3. Page tables
   - 프로세스마다 존재한다.
   - OS가 관리해준다.
   - Page number를 인덱스로 사용해서 frame number를 알려준다.
     - 이 과정에서 주소 연산은 MMU가 대신 해주기 때문에 성능 오버헤드 이슈가 거의 없다.
   - 페이지마다 하나의 페이지 테이블 엔트리를 가진다. (logical/virtual address space)
4. Paging Example
   - Logical address: 32 bits
   - Physical address: 20 bits
   - Page size: 4KB
   - Offset: 12 bits
   - Page Number: 20 bits
   - Page table entries: 2^{20}



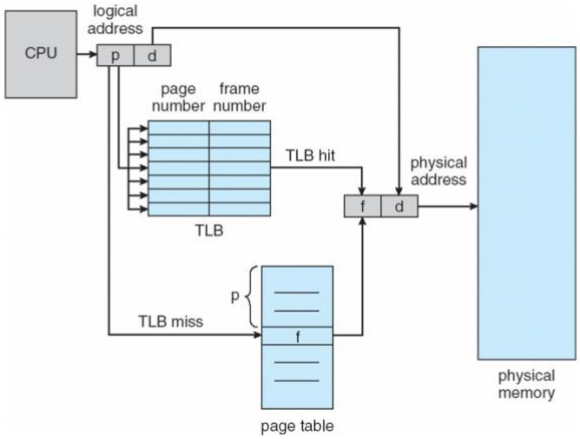
   -
   - 페이지의 크기 : 4KB
     - 2^12 바이트
     - 로지컬 주소의 하위 12비트를 사용하여 offset을 나타낼 수 있다.
   - Page Number : 20 bits
     - Page table entries (프로세스마다 페이지 테이블 엔트리의 수) : 2^20개
5. Free frames
   - free-frame list를 OS가 관리한다.
   - free-frame 개수보다 더 많은 page를 필요로 하는 프로세스를 실행하면?
     - overflow 발생
     - 디스크를 메모리처럼 사용한다. (swap in/out)

V. Page Table 및 TLB

A. Page Table 구현

1. 페이지 테이블은 메인 메모리에 있다.
2. PTBR (Page-table base register) -> 페이지 테이블을 가리키는 포인터 역할
3. PTLR (Page-table length register) -> 페이지 테이블의 사이즈를 나타낸다.
4. 모든 데이터와 코드에 대한 접근은 두 번 메모리에 접근해야 한다.
   - Page table(메인 메모리 안) +실제 메모리에 있는 데이터 혹은 코드에 접근
5. 더 효율적인 방법은 없을까?
   - Page table의 페이지를 캐시 데이터로 만든다.
   - **Translation Look-aside Buffer (TLB)**
     - MMU에 의해 관리된다.

B. TLB



1. Associative Memory
   - Parallel search (병렬 접근)이 가능하다.
   - 탐색하는데 시간이 비교적 적게 걸린다.
2. Address translation (p,d)
   - p는 associative register 안에 있고, p에 맞는 frame #을 반환한다.
   - p가 TLB에 없다면, 메모리 안에 있는 페이지 테이블로 접근해서 frame #을 얻는다.
3. TLB는 하드웨어적으로 구현된다.
   - Fully associative cache : 모든 엔트리는 병렬적으로 탐색된다.
   - 캐시의 태그는 로지컬 페이지 넘버(in 로지컬 주소)를 가진다.
   - 캐시의 값들은 PTE들을 나타낸다. (page table entries)
   - PTE(frame #) + offset -> MMU는 직접적으로 피지컬 주소를 계산할 수 있다.
4. Locality를 활용한다.
   - 프로세스들은 동시에 적당한 페이지들을 사용한다.
   - 일반적으로 TLB에 16~48 엔트리 (64 ~ 192KB)가 존재한다.
   - Hit rate가 98% 이상일 정도로 좋다.
5. Locality 종류
   - Temporal locality
     - 최근에 사용한 것 다시 쓸 확률이 높다. (반복문 카운터 등)
   - Spatial locality
     - 근처에 있는 페이지들을 사용할 가능성이 높다.
6. TLB 미스 대처
   - Hardware (MMU) : Intel x86
     - 메모리 안에 페이지 테이블이 어디 있는지 안다.
     - OS가 테이블을 유지하고, 하드웨어(MMU)는 페이지 테이블에 직접적으로 접근한다.
     - 페이지 테이블은 하드웨어적으로 정의된 형태여야 한다.
   - Software loaded TLB (OS)
     - 소프트웨어적으로 해결하는 방법은 거의 쓰이지 않는다.
     - TLB 미스가 발생하면 OS가 제대로 된 PTE를 찾고, TLB를 로드해준다.
     - 페이지 테이블은 OS가 접근하기 쉬운 어느 형태여도 괜찮다.
7. TLB 관리
   - OS가 TLB와 페이지 테이블의 일관성을 보장해준다.
     - OS가 PTE의 protection bits를 변경하면, 해당 PTE가 TLB 안에 있으면 해당 PTE를 invalidate하는

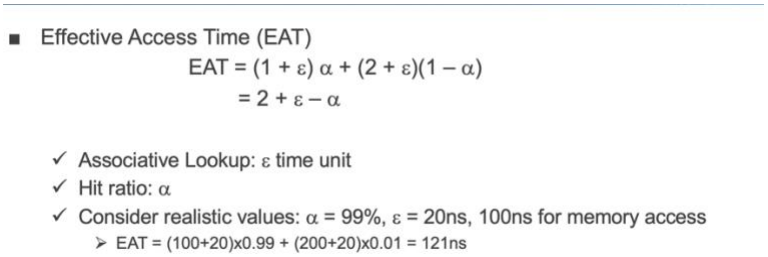
것이 필요하다.

- context switch 발생 시 TLB 다시 로드
  - context switch 발생하면 TLB의 모든 엔트리를 invalid / flush 시킨다.
  - 무조건 처음에는 TLB 미스 발생
  - 페이지 테이블에서 locality 고려해서 TLB에 한 뭉탱이를 로드한다. -> 금방 hit rate 올라간다.
  - 기존에 TLB에 있던 PTE들은 victim이 돼서 내려온다. (하드웨어적으로 구현된다.)

8. Locality 종류
   - Temporal locality
     - 최근에 사용한 것 다시 쓸 확률이 높다. (반복문 카운터 등)
   - Spatial locality
     - 근처에 있는 페이지들을 사용할 가능성이 높다.

C. Access time & Main memory Protection

1. Effective Access Time (EAT)



   - (1+앱실론), (2+앱실론)에서 1과 2는 메모리 access가 한 번 일어나느냐, 두 번 일어나느냐의 차이

2. Main Memory Protection
   - page table에서 invalid인 PTE를 만난다면?
   - 어떻게 처리할 것인지 분기 발생
     - swap-out 상태인지가 상태인지? 다시 swap-in시키고 valid 상태로 변경해야 한다.
     - exception : CPU 스스로 인터럽트를 걸어서 예외 처리해야 한다.

3. TLB는 하드웨어적으로 구현된다.
   - Fully associative cache : 모든 엔트리는 병렬적으로 탐색된다.
   - 캐시의 태그는 로지컬 페이지 넘버(in 로지컬 주소)를 가진다.

D. PTEs (Page Table Entries)

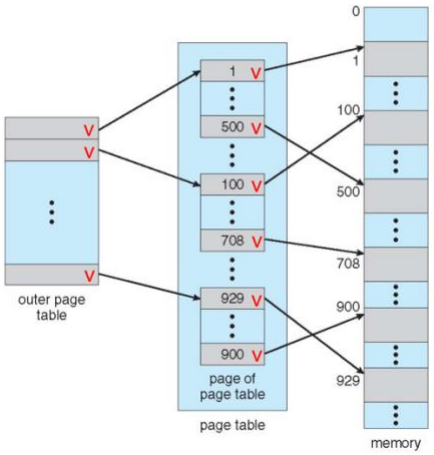
1. Valid bit (V) : PTE가 사용될 수 있는지를 나타낸다.
   - logical address가 사용될 때마다 체크된다.
2. Reference bit (R) : 페이지가 최근에 접근되었는지를 나타낸다.
   - read나 write가 발생하면 업데이트된다.
3. Modify bit (M) : 페이지가 수정되었는지에 대한 정보를 나타낸다.
   - write이 발생하면 업데이트된다.
4. Protection bit (Prot) : 어떤 연산이 페이지에 대해서 허용되었는지에 대한 정보를 나타낸다.
   - read, write, execute 등
5. frame Number (FN) : 피지컬 메모리의 페이지를 결정한다.

E. Page Table Structure

1. 페이지 테이블의 크기는 매우 크다.
   - 페이지 테이블의 크기는 32비트 주소공간이 4KB의 페이지 크기마다 32비트의 주소를 가진다.
   - 프로세스마다 4MB의 크기를 가진다.
   - 그래도 모든 페이지가 valid인 경우는 별로 없다.
   - invalid까지 모두 관리하는 것은 낭비

2. 해당 오버헤드를 줄일 수 있는 방법은?
   - 유효한 주소공간 부분만 매핑을 수행한다.
3. 어떻게 사용되고 있는 페이지만 매핑할 수 있는지?
   - 페이지 테이블 구조를 동적으로 확장 가능하게 만든다.
4. 방법
   - Hierarchical paging (이것만 쓰임)
   - Hashed page table
   - Inverted page table
5. Hierarchical paging
   - 로지컬 주소 공간을 여러 개의 단계로 나눈다.
   - Two-level paging
     - page number를 두 개로 쪼갠다. (p1, p2, d) -> p1, p2 10비트씩
     - 로지컬 주소에서 p1에 해당하는 페이지 테이블로 이동한다.
     - p2에 해당하는 페이지 테이블로 이동한다.
     - 실제 메모리에 접근한다.
     - 모든 페이지가 valid인 경우?

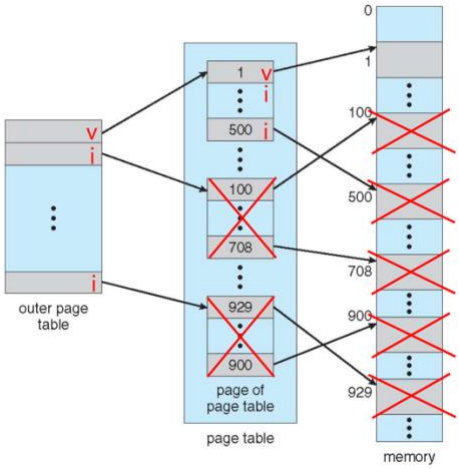
■ Two-level paging example
  ✓ Case 1) all the pages are used
  ✓ Outer page table
    ➢ 2^{10} entries
    ➢ 4B x 2^{10} = 4KB
  ✓ Page table
    ➢ 2^{10} entries x 2^{10}
    ➢ 4B x 2^{10} x 2^{10} = 4MB
■ Single-level paging
  ✓ (p, d): p=20bits
  ✓ Page table
    ➢ 2^{20} entries x 1
    ➢ 4B x 2^{20} = 4MB



◆
◆ single-level paging보다 0.4MB 더 많이 쓴다.

     - 하나의 페이지만 valid 경우?

■ Two-level paging example
  ✓ Case 2) Only one page is used
  ✓ Outer page table
    ➢ 2^{10} entries
    ➢ 4B x 2^{10} = 4KB
  ✓ Page table
    ➢ 2^{10} entries x 1
    ➢ 4B x 2^{10} x 1 = 4KB



◆
◆ Invalid인 경우는 고려하지 않아도 되기 때문에 총 8KB만 사용하므로, 리소스를 아낄 수 있다.
◆ 그리고 TLB에 어떤 페이지를 올릴지에 대한 고민도 편해진다.

6. Hashed Page Tables

- 손실압축 (나머지 연산)
- Linked list로 연산의 과정을 관리해야하는데, access 오버헤드가 너무 커서 안쓴다.

7. Inverted Page Tables
   - 페이지 테이블에 pid와 PN을 저장한다. 해당하는 오프셋을 FN으로 가진다.
   - 역연산을 하려면 MMU 아키텍처를 갈아엎어야 하므로, 오버헤드가 크다.
   - frame 개수가 너무 많으면 페이지 테이블 관리가 어려워진다.
8. Shared Pages
   - fork()가 발생한 경우에는 똑같은 코드 섹션을 공유한다.
   - 피지컬 메모리에 한 번 올린 코드 데이터를 공유함으로써 메모리를 아낄 수 있다.
   - 어느 하나의 프로세스에서 페이지에 해당하는 코드가 수정되면 별도의 프레임을 할당한다.
   - 다이내믹 링킹 개념에서 라이브러리는 여러 프로세스에서 코드 접근을 수행한다.
     - 해당 라이브러리를 shared pages에 올려놓고, 복사본을 만들지 않는다.

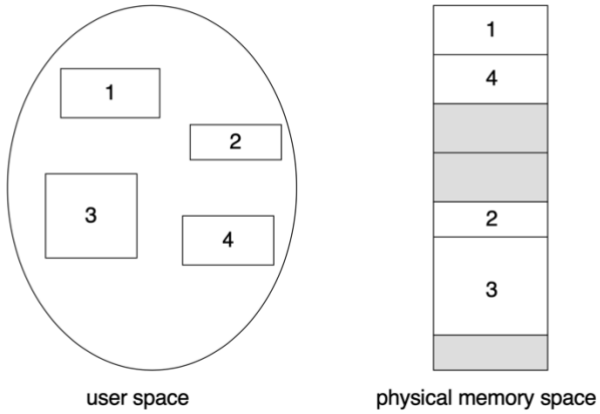
F. Paging의 장점
1. 피지컬 메모리에 할당하기가 쉽다.
   - 피지컬 메모리는 free frame list 중에서 선택되어 할당된다.
   - 할당되면, 해당 프레임은 프리 리스트에서 제외된다.
2. External Fragmentation이 발생하지 않는다.
   - 메모리를 효과적으로 활용
3. 프로그램의 데이터나 코드를 뭉탱이로 page out하기 쉽다.
   - 모든 페이지의 사이즈가 같다.
   - valid bit를 통해서 page out된 페이지를 확인할 수 있다.
   - 페이지 크기는 disk block 사이즈에 따라서 정해진다.
4. 허락되지 않은 접근으로부터 페이지를 보호하기 쉽다.
5. shared page를 구현하기 쉽다. (페이지를 공유하기 쉽다)

G. Paging의 단점
1. 여전히 internal fragmentation은 존재한다.
   - 무시(감수)할 수 있는 정도
2. 메모리에 접근할 때 오버헤드 발생
   - 페이지 테이블에 접근하고, 메모리에 접근하므로 메모리 접근이 2번 일어난다.
   - TLB로 해결
3. 페이지 테이블을 관리하는 메모리의 사이즈가 클 수 있다. (invalid까지 관리하면)
   - virtual address space에서 페이지당 하나의 PTE가 필요하다.
   - 32비트 주소가 4KB page만큼 존재한다. -> 2^20개만큼의 PTE를 나타낼 수 있다.
   - 각 PTE가 4바이트이기 때문에 페이지 테이블당 4MB의 크기를 가진다. (4바이트 * 2^20)
   - OS는 일반적으로 프로세스마다 각각의 페이지 테이블을 가진다.
     - 25 프로세스 -> 100MB의 총 페이지 테이블 크기
4. 해결
   - Hierarchical page tables ...

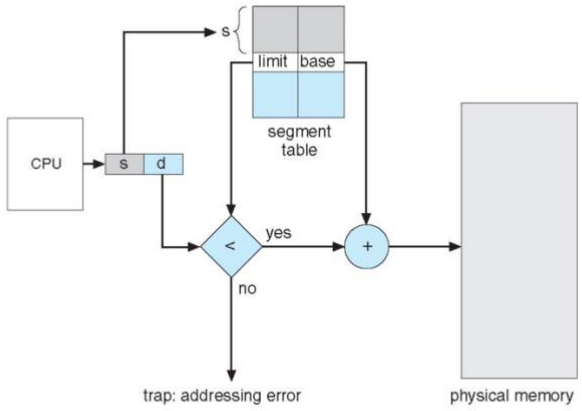
VI. Segmentation

A. 가상메모리를 코드, 심볼 테이블, 서브루틴, 스택 등 논리적인 부분(서로 다른 크기)으로 나눠서 관리한다.

■ Logical view of segmentation

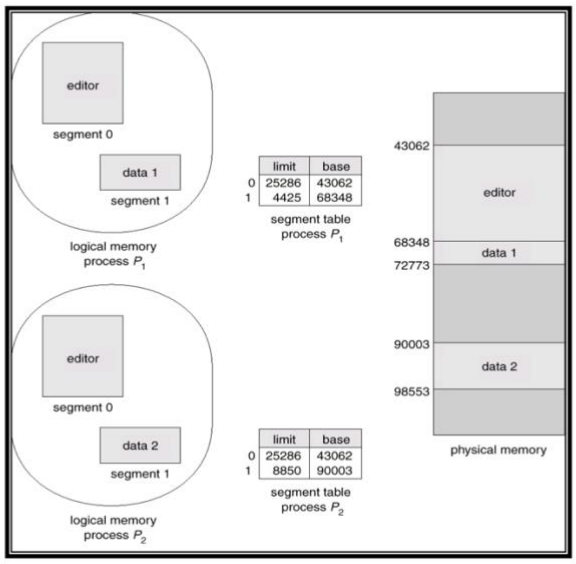


1. segment table 존재



-
- logical address에서 s를 통해 segment 테이블에서 정보를 가져온다.
- limit : bound값, base : base address
  - offset이 limit을 넘는지 체크 -> addressing error 체크
  - segment register 활용
- addressing error가 발생하는지 검사를 수행하고, 통과된다면 segment table에서 받아온 주소와 더하기 연산을 통해 실제 메모리에 올린다.

2. 여러 프로세스에서 공유하는 메모리를 피지컬 메모리에 한 번만 올려서 share도 가능하다.



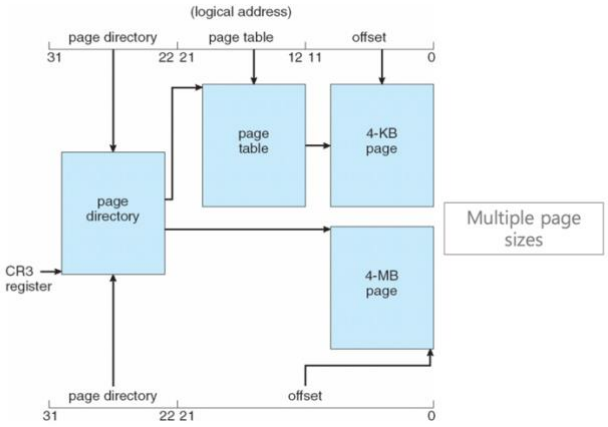
-

3. Segmentation의 장점
   - 데이터의 크기가 유연한 데이터에 대해 관리가 쉽다.
   - segment를 보호하기 쉽다.
     - segment 테이블의 각 엔트리마다 Valid bit가 존재한다.
     - Protection bits(read/write/execute) 역시 존재한다.
   - share하기 쉽다.
     - base/limit 정보에 같은 피지컬 주소를 넣는다.
     - 코드 혹은 데이터 sharing이 segment 레벨에서 발생한다. (shared libraries)
   - Internal fragmentation이 존재하지 않는다.
4. Segmentation의 단점
   - 스택, 힙과 같은 동적인 데이터들의 크기를 예측하기가 어렵다.
   - 데이터를 공유하기 위해서는 세그먼트들이 동일한 세그먼트 번호들을 가져야 한다.
     - 그렇지 않으면 indirect addressing만 가능하다.
   - segment table의 크기가 크다.
     - 메인 메모리에 존재한다. (페이지 테이블과 비슷)
     - 성능을 위해 하드웨어 캐시를 사용한다.
     - 관리하기 나름이다.
   - External fragmentation이 발생한다. (contiguous allocation과 비슷)

B. Paging vs. Segmentation
1. Block size
   - Paging : 고정
   - Segmentation : 가변
2. Linear address space
   - Paging : 1
   - Segmentation : Many
3. Memory addressing
   - Paging : one word (PN + offset)
   - Segmentation : two words (segment & offset)
4. Replacement
   - Paging : Easy (모두 같은 사이즈)
   - Segmentation : Difficult (segment가 들어갈 수 있는 위치를 찾아야만 한다.)
5. Fragmentation
   - Paging : Internal
   - Segmentation : External
6. Disk traffic
   - Paging : Efficient (페이지 사이즈에 맞춰서 최적화되어 있음)
   - Segmentation : Inefficient (페이지 사이즈가 다를 수 있다.)
7. Transparent to the programmers
   - Paging : O (피지컬 메모리로의 매핑에 대해서 개발자는 몰라도 된다.)
   - Segmentation : X (개발자가 세그먼트 정보의 일부를 알고 있어야 한다.)
8. Address space가 피지컬 메모리의 크기를 초과할 수 있나?
   - Paging : O
   - Segmentation : X

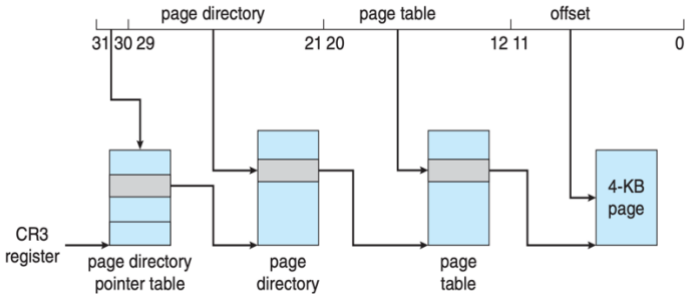
9. 코드와 데이터가 구별되고, 각각 보호될 수 있는가?
   - Paging : X (구별 없이 페이지 단위로만 관리된다.)
   - Segmentation : O (논리적 단위 = 세그먼트)
10. 사이즈가 가변적인 테이블이 쉽게 적용될 수 있는가?
   - Paging : X
   - Segmentation : O
11. 코드의 공유가 쉬운가?
   - Paging : X
   - Segmentation : O
12. 기술이 발명된 배경
   - Paging : 더 많은 피지컬 메모리를 구매하지 않고도 더 큰 linear address space를 갖기 위해
   - Segmentation : 프로그램과 데이터가 논리적으로 독립적인 address space를 가지고, sharing과 protection을 용이하게 하기 위해

C. Hybrid approaches (Segmentation with Paging)
1. 1차적으로 segment화 시킨 후에 paging 수행
   - Paged segments
   - Segmentation with Paging
   - 세그먼트는 페이지보다 큰 사이즈를 가진다.
   - 코드, 데이터, 힙 등 논리적인 용도에 따라 segment 단위로 나눈다. (multiple pages size)
   - segment를 고정된 크기로 쪼개서 paging으로 피지컬 메모리에 매핑한다.
     - external fragmentation 발생 X
     - PTE는 해당하는 세그먼트에만 할당될 수 있다.
2. IA-32
   - page size 지원
     - 4KB, 2MB, 4MB
   - IA-32 Paging
     - Two-level paging
     - Multiple page size (페이지의 사이즈가 고정되어 있지 않다.)
   - IA-32 PAE (Page Address Extension)

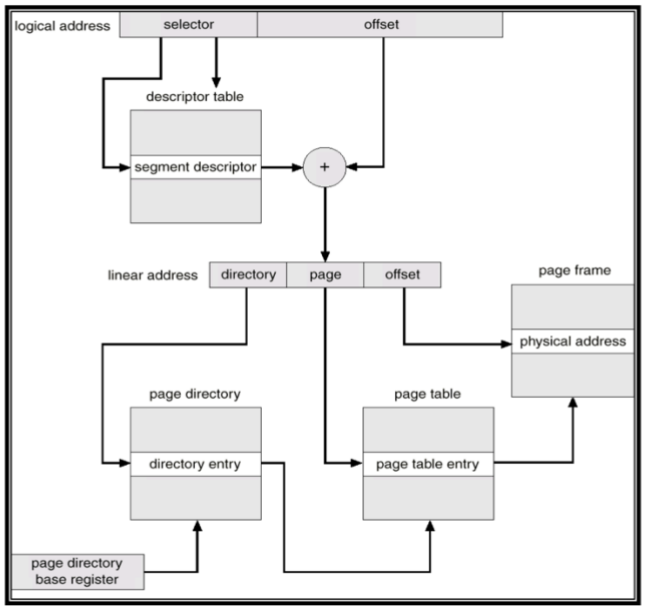


     -
     - 32비트 시스템에 적용
     - 메모리 총량 2^32 = 4기가
     - 8기가 RAM을 어떻게 사용?
     - 남은 부분의 메모리를 디스크로 사용한다.

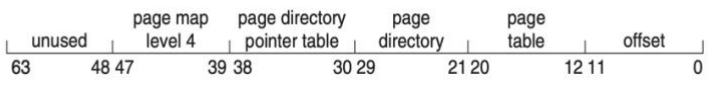
        - ◆ 엄청 빠르다.
        - ◆ 껐다 켜면, 휘발되기 때문에 다시 코드 복사해야 한다.
    - ■ 3-level scheme도 가능하다.
    - ■ 상위 2 비트는 페이지 디렉토리 포인터 테이블을 가리킨다.
    - ■ 페이지 디렉토리와 페이지 테이블 엔트리들은 64비트 크기를 가진다.
    - ■ 주소공간을 36비트로 늘릴 수 있다. (64기가 피지컬 메모리를 사용 가능하다.)
    - ■



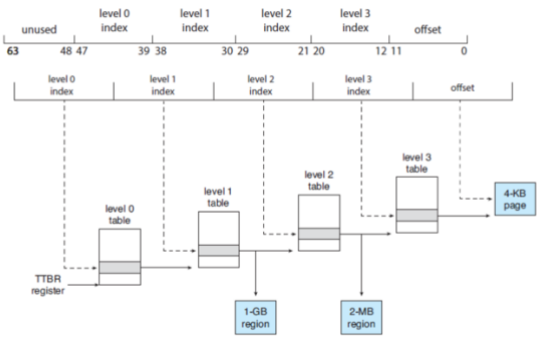
3. Alpha XP Architecture
    - page size 지원
        - ■ 8KB, 16KB, 32KB, 64KB
        - ■ 43, 47, 51, 55 bits virtual address
4. Pentium



    -
    - hierarchical page table 사용한다.
    - 43, 47, 51, 55 bits virtual address
5. x86-64 (AMD64, IA-64)
    - 메모리 버스가 64비트
    - 48-bits addressing을 수행한다.
        - ■ 16비트는 아직 사용하지 않는다.
        - ■ 필요도 없고, 엔트리 크기만 커진다.
    - Multiple page sizes (4KB, 2MB, 1GB)
    - 4-level paging hierarchy 사용
    - PAE를 사용할 수도 있다. (52비트 실제 주소 체계)
    -



6. 2-Level Paging in ARM (AArch32)
   - 페이지를 크기에 따라 4종류로 나눠서 관리한다.
     - section page (1MB)
     - large page (64KB)
     - small page (4KB)
     - tiny page (1KB)
   - Translation table -> page table
   - TTB (Translation Table Base)
7. ARMv8 (AArch64)
   - Support for access to more than 4GB of memory space
     - ✓ four-level hierarchical paging



   -
   - 4-Level Hierarchical Paging
     - 여전히 맨앞 16비트는 사용하지 않는다.
   - 4기가 이상의 메모리 공간을 추가로 지원한다.