

Local vs. Physical Address

v Logical Address (= virtual address)

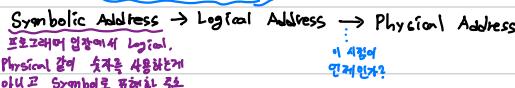
- 각 프로세스마다 독립적으로 가지는 주소 공간
각자 가지게 되는 주소
- 각 프로세스마다 0번지부터 시작
- CPU가 보는 주소는 logical address임

메모리에 물리화할 때 시작 위치는 바뀌지만
그 안에 있는 코드의 address는 그대로 남아 있기 때문에.

v Physical Address

- 메모리에 실제 옮겨가는 위치

* 주소 바인딩: 주소를 결정하는 것 → 메모리 어도우 물려갈 것인지



주소 바인딩 (Address Binding)

• Compile time binding → 컴파일 될 때 주소변환

- 물리적 메모리 주소 (Physical address)가 컴파일 시 알려짐
- 시작 위치 변경 시 재컴파일
- 컴파일러는 절대코드 (absolute code) 생성

Logical address가 Physical address로 사용되면서

• Load time binding → 실행 시작될 때 주소 변환

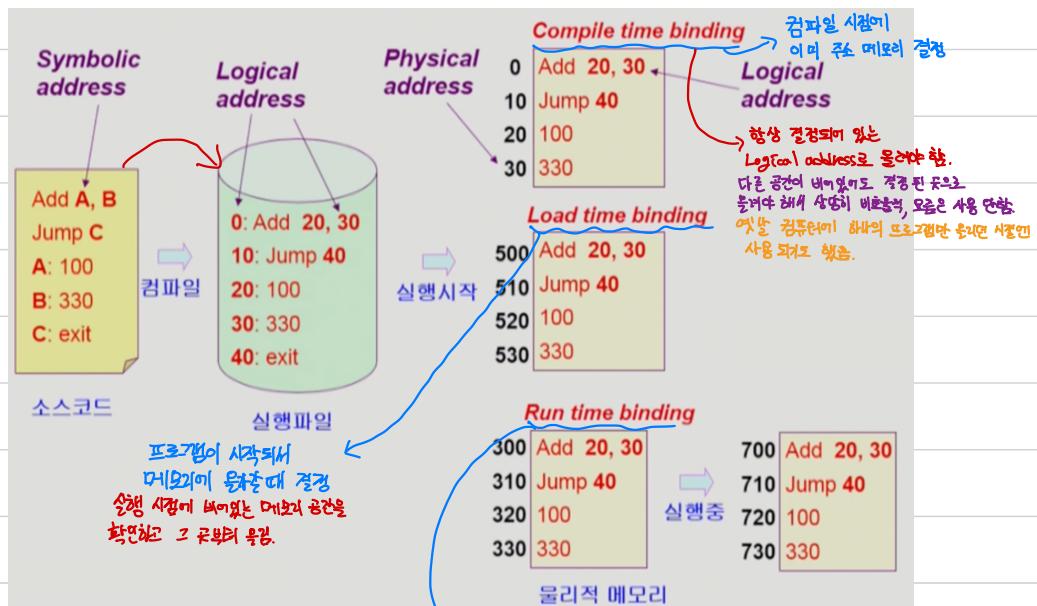
- Loader의 책임하여 물리적 메모리 주소 부여
- 컴파일러가 재배치 가능코드 (relocatable code)를 생성한 경우 가능

• Execution time binding (= Run time Binding)

- 수행된 후에도 이후에도 프로세스의 메모리 상 위치를 움직일 수 있음
- CPU가 주소를 참조할 때마다 binding을 첨결 (address mapping table)
- 하드웨어적인 지원 필요 (e.g. base and limit registers, MMU)

실행된 후에도

중간에 물리적 메모리
주소 바꿀 수 있는 방법



) 실행시에 주소가 결정되는 건 Load time과 동일 하지만 주소가 실행 중에 바뀔 수도 있다는 차이점.
지금의 컴퓨터는 Runtime binding 지원

Memory-Management Unit (MMU)

주소 변환을 처음 배우기 때문에 Logical address가 physical address로
그대로 통과된다 가능하고 진행.

• MMU (Memory -Management unit)

v logical address를 physical address로 매핑해 주는 hardware device

• MMU Scheme

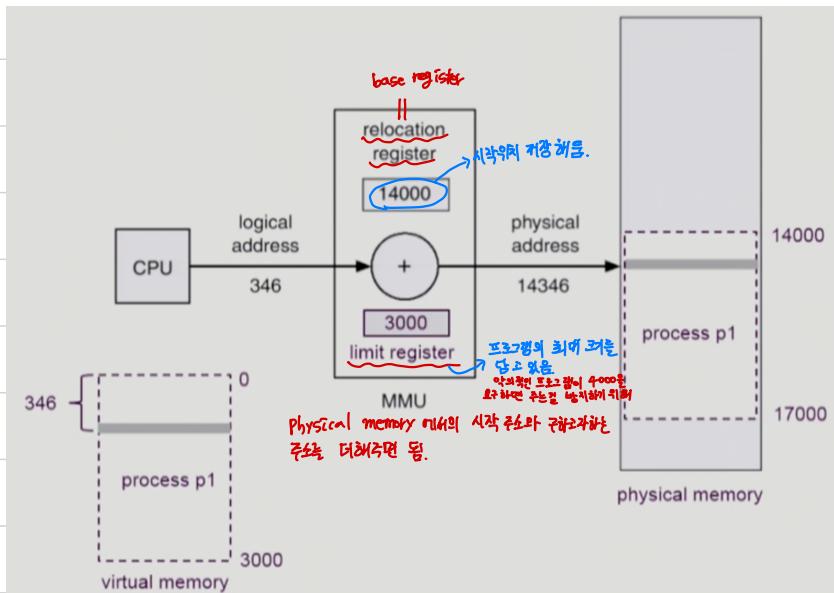
v 사용자 프로세스가 CPU에서 수행되며 성능하락은 모든 주소값에 대해 base register(s) relocation register)의 값을 더한다.

• User program

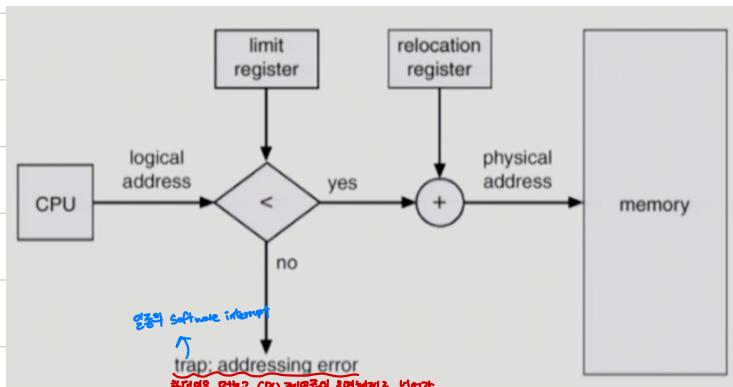
v logical address 만을 다룬다

v 실제 physical address를 볼 수 없는데 알 필요가 없다

Dynamic Relocation



Hardware Support for Address Translation



운영체계 및 사용자 프로세스 간의 메모리 보호를 위해 사용하는 레지스터

- **Relocation register**: 접근 할 수 있는 물리적 메모리 주소의 최소값
(=base register)

- **Limit register**: 논리적 주소의 범위

Some Terminologies

- Dynamic Loading
- Dynamic Linking
- Overlays
- Swapping

Dynamic Loading

- 프로세스 전체를 메모리에 데리다 옮기는 것이 아니라 루틴이 불려질 때 메모리에 Load 하는 것
 - memory utilization의 향상
 - 가끔씩 사용되는 많은 양의 코드의 경우 유통 예) 오류 처리 루틴 때문에 혹시 그런 상황이 생기면 그때 메모리에 옮김.
 - 운영체제의 특별한 지원 없이 프로그램 자체에서 구현 가능 (OS는 라이브러리를 통해 지원 가능)
- * Loading: 메모리로 옮기는 것
- 운영체제의 페리포드가 아니게
dynamic Loading은 프로그래머가 직접 하는 것.
- 라이브러리 후 사용되어야 고정이
어려울 때 사용

프로그램이 dynamic loading을 위해 이곳에서는 기 위해 dynamic loading이지만

프로그램이 명시하지 않고 운영체제가 알아서 불려놓고 끌어내는 것도 dynamic loading이라고 써야 쓰기도 한다 정도까지 알아 두자.

Overlays

- 메모리에 프로세스의 부분 중 실제 필요한 정보만을 옮김.
→ 이 내용만 보면 dynamic load이랑 차이X
- 프로세스의 크기가 메모리보다 클 때 유용
→ 과제인 컴퓨터 메모리가 너무 작아서
프로그래머가 어떤 때 어떤 부분을 메모리에 옮기지
않고 있음.
- 운영체제의 지원없이 사용자에 의해 구현
- 작은 공간의 메모리를 사용하면 초기 시스템에서 수작업으로 프로그래머가 구현

v Manual Overlay

- v 프로그래밍이 매우 복잡

* dynamic loading과 overlay의 차이

⇒ dynamic loading은 OS의 라이브러리를 사용해서
프로그램이 고정하면 되지만
overlay는 프로그래머가 직접 다 고정 해야 함.

Swapping

→ 이게 유판 swapping
프로그램 구성하는게 전부다 줄여내는 것.

• Swapping

- v 프로세스를 일시적으로 메모리에서 backing store로 풀어내는 것

하드디스크와 같이 메모리에서 풀어내는 것을 저장하는 곳

• Backing store (= swap area)

v 디스크

- 많은 사용자의 프로세스 어디지를 담을 만큼 충분히 빠르고 큰 저장 공간

• Swap in / Swap out

- v 일반적으로 증기 스케줄러(swapper)에 의해 swap out 시킬 프로세스 선택

v priority-based CPU scheduling algorithm

- priority가 낮은 프로세스를 swapped out 시킴
- priority가 높은 프로세스는 메모리에 물려 둘음.

- v Compile time 혹은 load time binding에서는 원래 메모리 위주로 swap in 해야 함.

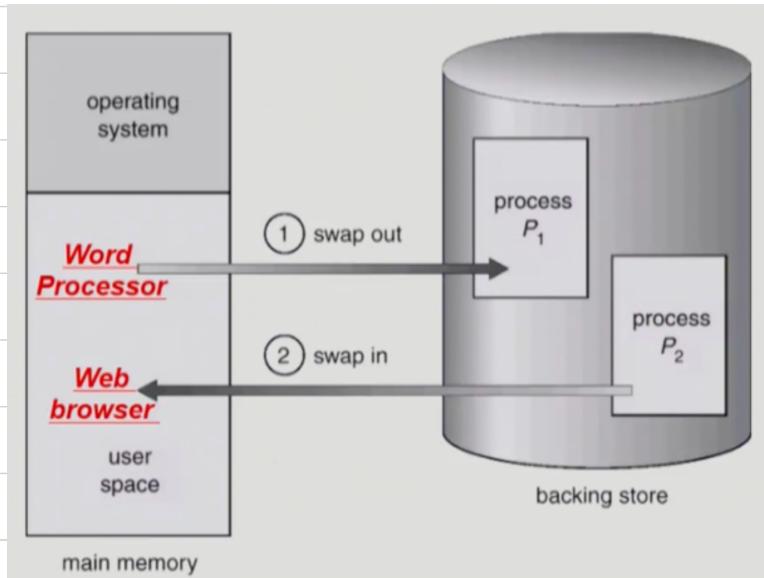
- v Execution time binding에서는 충분히 메모리 영역 아무곳이나 물질 수 있음.

- v Swap time은 대부분 transfer time (swap 되는 양이 비례하는 시간)임.

→ 이게 지원 되어야지 swapping을

효율적으로 할 수 있다.

Schematic View of Swapping



Dynamic Linking

- Linking을 실행 시간(execution time)까지 미루는 기법
- Static Linking
 - 라이브러리가 프로그램의 실행 파일 코드에 포함됨
 - 실행 파일의 크기가 커짐
 - 동일한 라이브러리를 각각의 프로세스가 메모리에 물리적으로 메모리 낭비 (eg printf 함수의 라이브러리 코드)
- Dynamic Linking
 - 라이브러리가 실행시(연결(link))됨
 - 라이브러리의 호출 부분이 라이브러리 후원의 위치를 찾기 위한 stub이라는 작은 코드를 둠
 - 라이브러리가 이미 메모리에 있으면 그 주소로 가고 없으면 디스크에서 읽어옴
 - 운영체제의 도움이 필요

Allocation of Physical Memory

- 메모리는 일반적으로 두 영역으로 나뉘어 사용

v OS 상주 영역

- interrupt vector와 함께 낮은 주소 영역 사용

v 사용자 프로세스 영역

- 높은 주소 영역 사용

• 사용자 프로세스 영역의 할당 방법

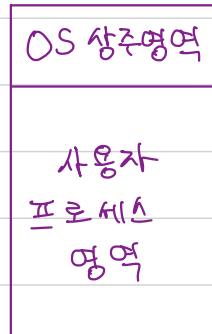
v Contiguous allocation 연속 할당, 통제로 물리적

- 각각의 프로세스가 대체로 연속적인 공간에 전체 모드를 하는 것
- Fixed partition allocation
- Variable partition allocation

v Noncontiguous allocation 불연속 할당, 장기 전개와 현재 사용..

- 하나의 프로세스가 메모리의 여러 영역에 분산되어 물리적 할당 가능

- Paging
- Segmentation
- Paged Segmentation

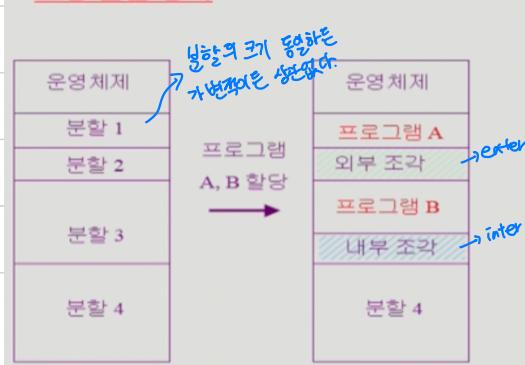


Contiguous Allocation

v 고정분할 (Fixed partition) 방식

- 물리적 메모리를 몇 개의 영구적 분할 (Partition)로 나눔
- 분할의 크기가 모두 동일한 방식과 서로 다른 방식이 존재
- 분할당 하나의 프로그램 적재
- 움동성이 많음
 - 동시에 메모리에 load되는 프로그램의 수가 고정됨
 - 최대 수행 가능 프로그램 크기 제한
- Internal fragmentation 발생 (External fragmentation도 발생)

고정 분할 방식

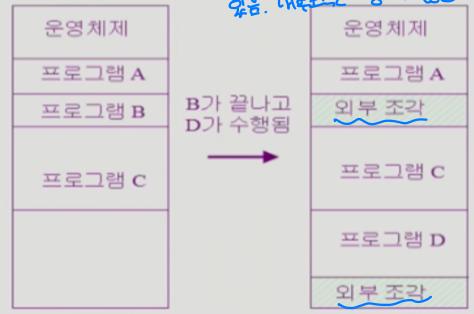


v 가변분할 (Variable partition) 방식

- 프로그램의 크기를 고려해서 할당
- 분할의 크기, 개수가 동적으로 변함
- 가상적 관리기법 필요
- External fragmentation 발생

가변 분할 방식

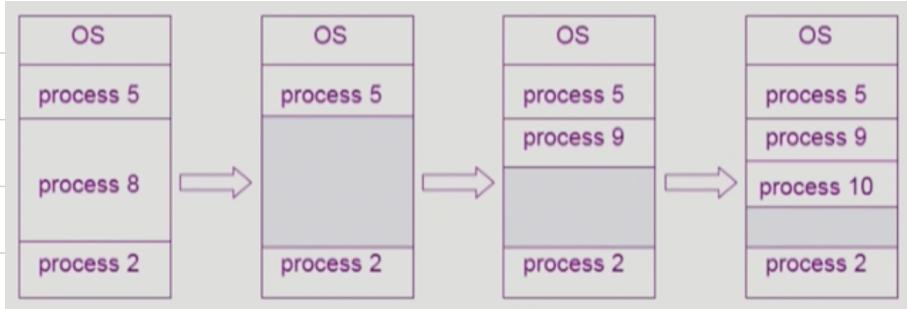
가변 분할을 쓰더라도 프로그램들의 크기가 고열하지 않기 때문에 외부조각이 생길 수 있다. 프로그램 A,B,C가 수행 중 있음. 내부조각은 생기지 않는다.



Contiguous Allocation

• Hole

- ▷ 가용 메모리 공간
- ▷ 다양한 크기의 Hole들이 메모리 어려움에 흩어져 있음
- ▷ 프로세스가 도착하면 수용 가능한 Hole을 할당
- ▷ 운영체제는 다음의 정보를 유지
 - a) 할당 공간
 - b) 가용 공간 (Hole)



• Dynamic Storage-Allocation Problem

▷ 가변 분할 방식에서 size n 인 요청을 만족하는 가장 적절한 hole을 찾는 문제

• first-fit

▷ size가 n 이상인 것 중 최초로 할당되는 hole에 할당

• Best-fit

▷ size가 n 이상인 가장 작은 hole을 찾아서 할당

▷ Hole들의 리스트가 크기로 정렬되지 않은 경우 모든 hole의 리스트를 탐색 해야함

▷ 많은 수의 아주 작은 hole들이 생성됨

• Worst-fit

▷ 가장 큰 hole에 할당

▷ 역시 모든 리스트를 탐색 해야함

▷ 상대적으로 아주 큰 hole들이 생성됨

* First-fit과 Best-fit이 Worst-fit 보다 속도와 공간이용률 측면에서
효과적인 것으로 알려짐(실험적인 결과)

Contiguous Allocation

• Compaction

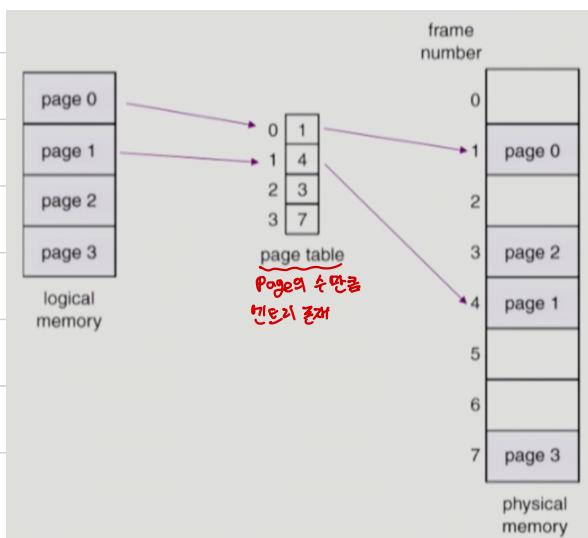
- v external fragmentation 문제를 해결하는 한가지 방법
- v 사용중인 메모리 영역을 한 군데로 몰고 빈 공간을 다른 한 곳으로 놓아 큰 빈 공간을 만드는 것
- v 매우 비용이 많이 드는 복잡한 과정
- v 최소한의 메모리 이동으로 Compaction 하는 방법 (매우 복잡한 문제)
- v Compaction은 프로세스의 주소가 실행 시간에 동적으로 재배치 가능한 경우에만 수행될 수 있다
Run time binding 이 지원 되어야 가능

Paging

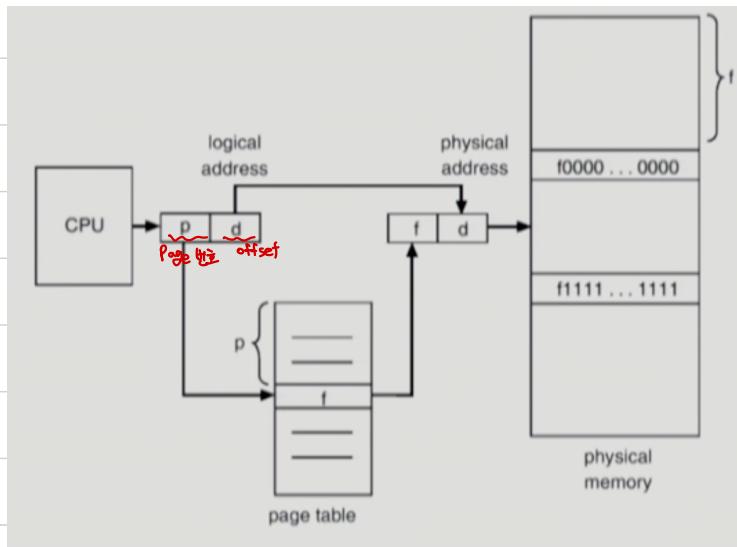
- Process의 virtual memory를 동일한 사이즈의 Page 단위로 나눔
- Virtual memory의 내용이 Page 단위로 noncontiguous하게 저장됨
- 일부는 backing storage에, 일부는 physical memory에 저장

• Basic Method

- v Physical memory를 동일한 크기의 frame으로 나눔
- v logical memory를 동일한 크기의 Page로 나눔 (frame과 같은 크기)
- v 모든 frame들을 관리
- v Page table을 사용하여 logical address를 physical address로 변환
- v External fragmentation 발생 안함
- v Internal fragmentation 발생 가능



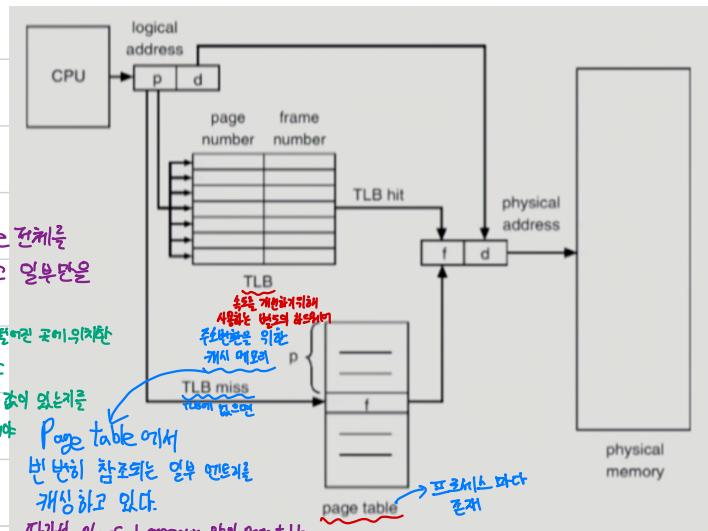
Address Translation Architecture



Implementation of page Table

- Page table은 main memory에 상주
- Page-table base register (PTBR) 가 page table을 가리킴
- Page-table length register (PTLR) 가 테이블 크기를 노출
- 모든 메모리 접근 연산에는 2번의 memory access 필요
- Page table 접근 1번, 실제 data/instruction 접근 1번
주소변환 위해 실제 주소가 주소위해
- 속도 향상을 위해 associative register 혹은 translation look-aside buffer (TLB) 라 불리는 고속의 lookup hardware cache 사용

Paging Hardware with TLB



Associative Register

• Associative registers (TLB): parallel search 가능

▼ TLB에는 Page Table 중 일부만 존재
→ TLB는 통칭 불필요한 검색하는게 아니라 전체를 Search 해야 하는 대문자

• Address translation

▼ Page table 중 일부가 associative register에 보관되어 있음

▼ 만약 해당 Page가 associative register에 있는 경우 그대로 frame#을 얻음

▼ 그렇지 않은 경우 main memory에 있는 page table로부터 frame#을 얻음

▼ TLB는 context switch 때 flush (remove old entries)

→ 프로세스마다 주소변환
엔트리가 다르기 때문에

Effective Access Time

- Associative register look up time = ϵ
- memory cycle time = 1
- Hit ratio = α

v associative register에 H 찾아가는 비율

- Effective Access Time (EAT)

$$EAT = (1 + \epsilon)\alpha + (2 + \epsilon)(1 - \alpha)$$
$$= 2 + \epsilon - \alpha$$

Two-Level Page Table

속도는 줄여들지 않지만
공간의 효율성을 위해 사용

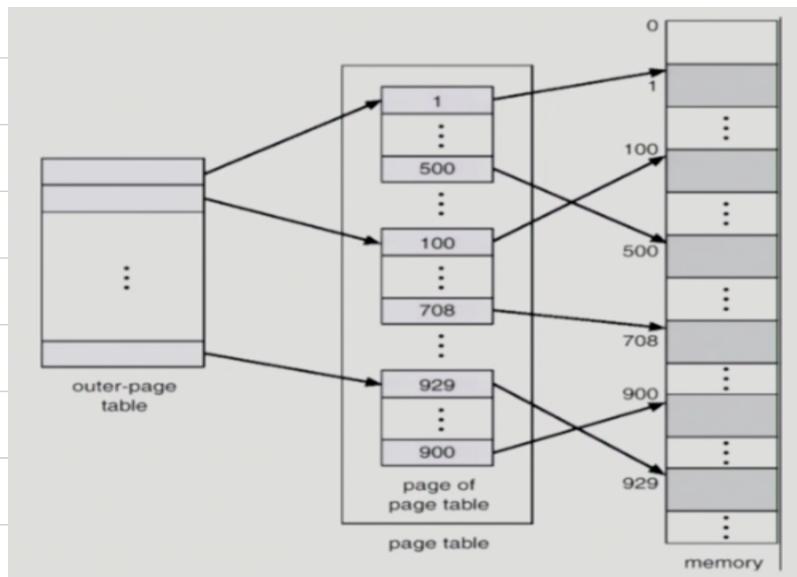
- 현대의 컴퓨터는 address space가 매우 큰 프로그램 지원

V 32비트 address 사용시 : 2^{32} (4G)의 주소 공간 \Rightarrow 최대인 6464도

- Page size가 4KB 개의 page table entry 필요
- 각 page entry가 4B이므로 4M의 page table 필요
- 그러나 대부분의 프로그램은 4K의 주소 공간을 지속적 일정한 사용하므로 page table 공간이 심하게 낭비됨

→ Page table 자체를 page로 구성

→ 사용되지 않는 주소 공간에 대한 outer page table의 엔트리 값은 NULL
(대체로는 inner page table이 없음)



Two-Level Paging Example

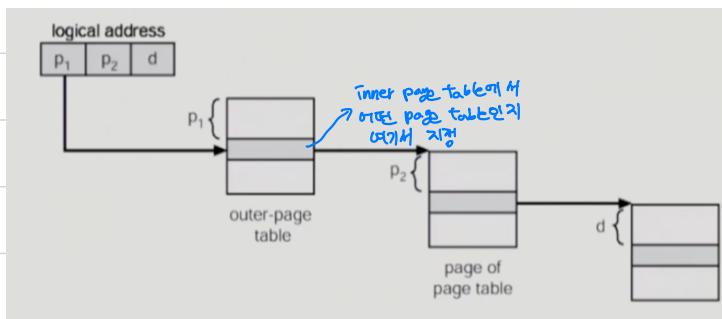
- logical address (on 32-bit machine with 4k page size)의 구성
 - 20 bit의 page number
 - 12 bit의 page offset
- Page table 자체가 Page로 구성되기 때문에 page number는 다음과 같이 나뉜다.
(각 Page table entry가 4B)
 - 10-bit의 page number
 - 10-bit의 page offset
- 따라서, logical address는 다음과 같다.

Page number	Page offset
P_1	P_2
10	10
12	

- P_1 은 outer page table의 index이고
- P_2 는 outer page table의 page address의 범위(displacement)

Address-Translation Scheme

- 2단계/两级映射 예시의 Address-translation scheme



Multilevel Paging and Performance

- Address space 더 넓으면 다단계 페이지 테이블 필요

- 각 단계의 페이지 테이블이 메모리에 존재 하므로 logical address의 physical address 변환에 더 많은 메모리 접근 필요

- TLB를 통해 메모리 접근 시간을 줄일 수 있음

- 4단계 주소변환 테이블을 사용하는 경우

▼ 메모리 접근 시간이 100ns, TLB 접근 시간이 20ns 이고

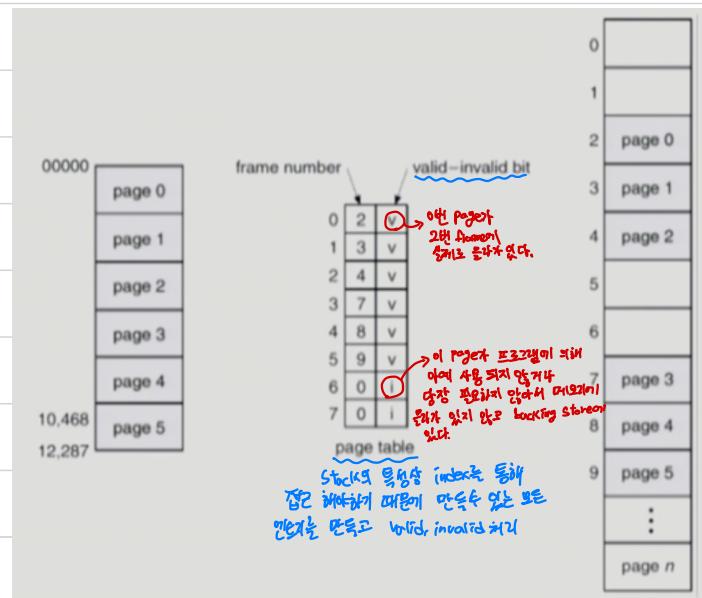
▼ TLB hit ratio가 98%인 경우

$$\text{Effective memory access time} = 0.98 \times 120 + 0.02 \times 520 = 128 \text{ nano seconds}$$

결과적으로 주소변환을 위해 28ns만 소요

TLB 접근 20ns
주소변환 400ns
메모리 접근 100ns

Valid(v) / Invalid(i) Bit in a Page Table



Memory Protection

- Page table의 각 entry마다 아래의 bit를 둔다.

v Protection bit

- Page에 대한 접근 권한 (read/write/read-only) → stack, data의 경우에 접근할 수 있게 권한을 주야는!

Code 같은 경우 변화면 안되기 때문에
read-only!

마호에 브인 프로세스만

Page table에 접근 가능하기
때문에 다른 프로세스로부터
보호 받는 의미는 아님.
과정 어떤 연산에 관해
접근 권한이 있느냐를 나타냄.

v Valid - Invalid bit

- "valid"는 해당 주소의 frame에 그 프로세스를 구성하는 유효한 내용이 있음을 뜻함 (접근 허용)

- "invalid"는 해당 주소의 frame에 유효한 내용이 없음을 뜻함 (접근 불허)

*;) 프로세스가 그 주소 부분을 사용하지 않는 경우

::;) 해당 페리미터가 메모리에 물려와 있지 않고 swap area에 있는 경우

Inverted Page Table

→ Page Table의 문제가 되겠지 ⇒ 굉장히 많은 용량을 차지하고 있다는 것

- Page Table의 매우 큰 이유

v 모든 Process 별로 그 logical address에 대응하는 모든 Page에 대해 Page Table entry가 존재

v 대응하는 Page가 메모리에 있든 아니든 간에 Page Table에는 entry로 존재

• Inverted page table

v Page frame 하나당 Page Table에 하나의 entry를 둠 것 (System-wide)

v 각 Page Table entry는 각각의 물리적 메모리의 page frame이 담고 있는 내용 표시 (process-id, process의 logical address,)

v 단점

• 테이블 전체를 탐색 해야함 → 매우 부적절!!

v 조치

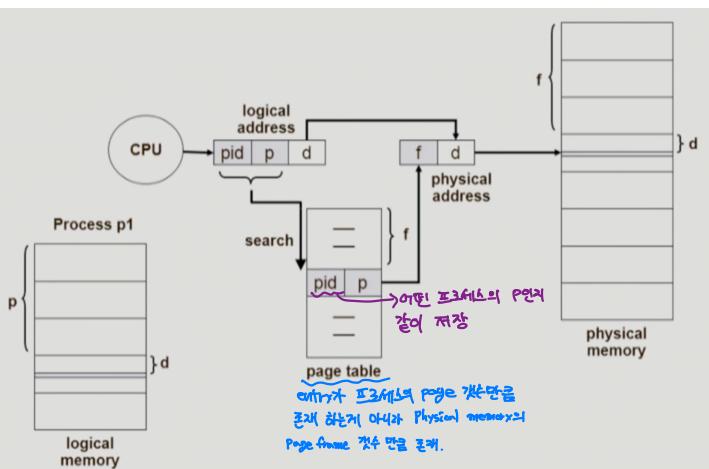
• associative register 사용 (expensive)

TLB처럼 병렬적 검색 가능하게 Page Table을

메모리에 아닌 별도의 associative register에 넣음.

Inverted Page Table Architecture

원래 Page Table을 통한 주소변환은 역발상으로 되길래 놓은 것



index p 만큼 떨어진 곳을 바로 검색하는데 Page Table의 장점인디!

Inverted page table은 Page 번호가 존재하면 어디를 전부 검색해야지만 주소변환을 할 수 있다.

logical address → physical addresses를 찾는 기본 page table과 달리 physical address → logical address를 찾는

Inverted page table은 기본 page table과 다르고 할 수 있는데 왜 사용하는가 ?? ⇒ System-Wide적으로 Page Table을 하나만 둘 필요가 공간을 줄이기 위해 사용. 대신 시간적인 overhead.. 한 번에 꾸준변환 되는게 아니라 page table 전부다 search해봐야 Page P#가 어떤 frame에 있는지 알 수 있기 때문에.

Shared Page

• Shared Code

↳ 프로세스간의 IPC와는 다른.

↳ read/write 가능해서 변경수행 공유, but shared code는 read-only

↳ v Re-entrant Code (= Pure Code)

↳ 1 ↳ v read-only로 하여 프로세스 간에 하나의 코드만 메모리에 둘림
(e.g., text editors, Compilers, Window systems)

↳ v Shared code는 모든 프로세스의 logical address space에서 동일한 위치에 있어야 함.

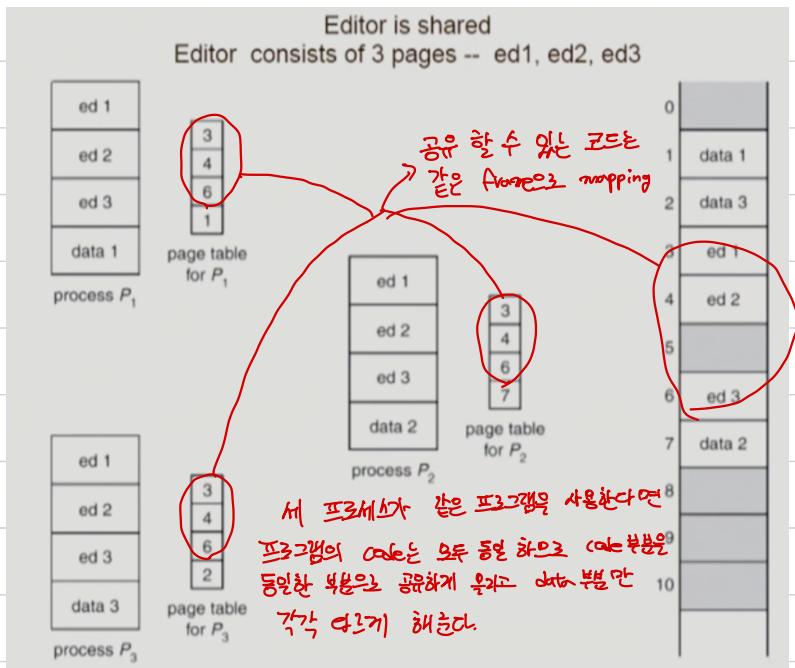
↳ 2

• Private code and data

↳ 각 프로세스들은 독자적으로 메모리에 둘림

↳ v Private data는 logical address space의 아무 곳에 와도 무방

Shared Pages Example



Segmentation

- 프로그램은 의미 단위인 여러개의 Segment로 구성

- 작게는 프로그램을 구성하는 함수 하나 하나를 세그먼트로 정의
- 크게는 프로그램 전체를 하나의 세그먼트로 정의 가능
- 일반적으로는 code, data, stack 부분이 하나씩의 세그먼트로 정의됨

- Segment는 다음과 같은 logical unit 들임

main(),
function,
global Variables,
Stack,
Symbol table, arrays

Segmentation Architecture

- logical address는 다음의 두 가지로 구성

⟨segment-number, offset⟩

• Segment table

↳ each table entry has:

- base - starting physical address of the segment
- limit - length of the segment

• Segment-table base register (STBR)

↳ 물리적 메모리에서의 segment table의 위치

• Segment-table length register (STLR)

↳ 프로그램이 사용하는 segment의 수

Segment number s is legal if $s < STLR$

Segmentation Hardware

→ 의미 단위로 자르기 때문에
길이가 고정하지 않을 수 있어서
길이를 table entry에 같이 가지고
있게 한다.

