Chapter 9. Main Memory

9.1 Background

Process

- 프로세스 == 실행중인 프로그램
  - 즉, 메인 **메모리에 로드**되어야 실행된다.

Memory

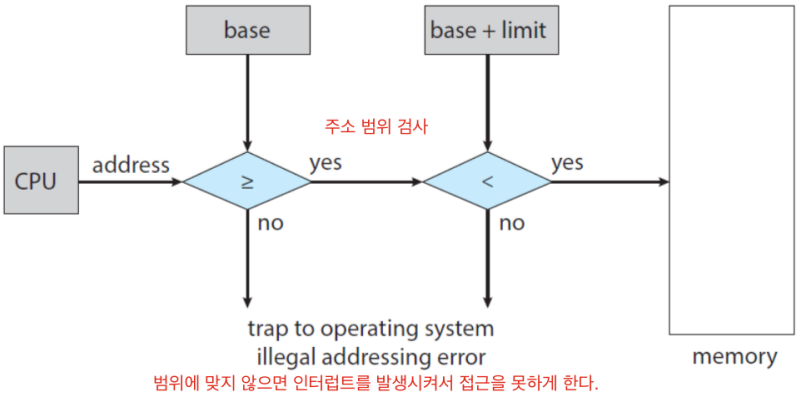
- byte의 집합으로 구성
- **address**를 가짐
- cpu는 pc(program counter)를 사용하여 명령어를 메모리에서 가져와서 load/store하며 실행

Memory Space

- 메모리 공간을 분리하여 관리해야함 → 멀티프로세싱
- **base, limit** 두개의 레지스터를 사용
  - 유저가 접근하는 주소가 올바른지 판단하는데 사용
  - 하드웨어적으로 구현
- 올바르지 않은 주소로 접근하려 한다 > segmentation fault를 던져 접근을 막음

Protection of memory space

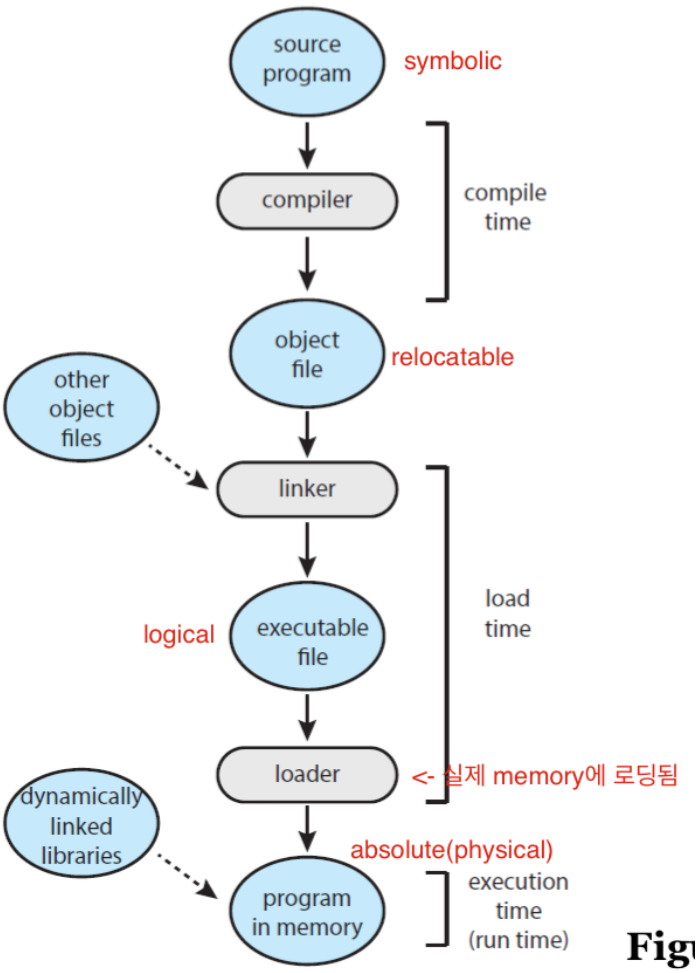
- CPU하드웨어 적으로 구현되어야 함
  - 모든 접근에 대해 판단함



Address Binding

- 프로그램은 a.out, a.exe 등 처럼 **바이너리 파일**로 디스크에 저장되어 있음
  - 이 상태는 아직 프로세스가 아님, 그냥 저장되어있는 프로그램일 뿐
  - 메모리에 불러서 run을 할 수 있는 상태
  - 메모리에 부를때, 항상 주소 공간을 00000000부터 시작하는 것은 아님
- 우리는 프로그램 code로 **symbolic**하게 메모리 번지를 정해줌
  - ex) int a;
  - 그저 주소공간의 이름을 지어준다는 뜻
- **컴파일러**가 **relocatble address**(logical)로 바인딩 해줌
- **linker, loader**가 **absolute address**(physical)로 다시 바인딩 해줌

relocatable address : 메모리에 언제든 넣을 수 있는 주소 (ex : 14byte)
absolute address : 절대적인 주소 > 실제로 메모리의 어떤 공간에 적재하는 것 > physical address



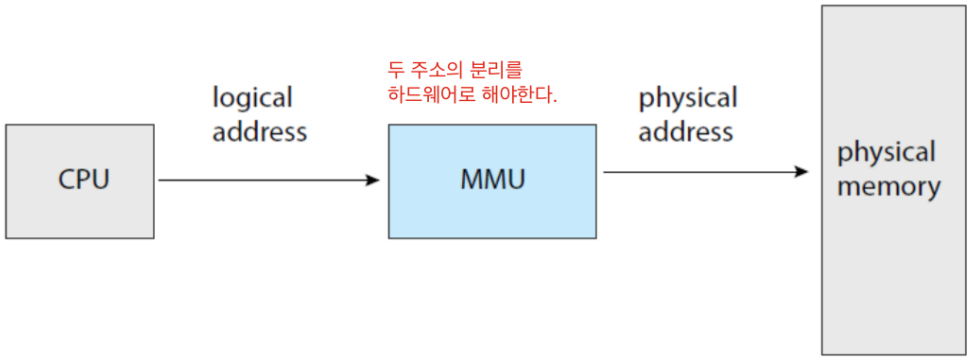
Fig

Logical vs. Physical Address space

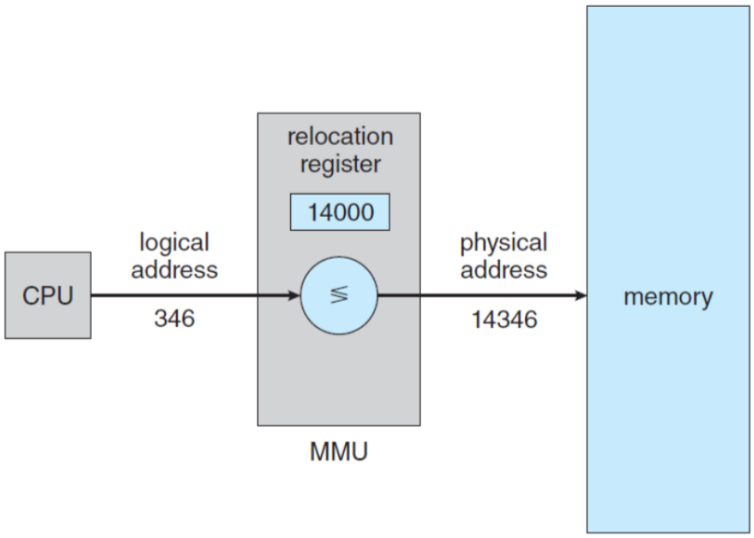
- logical address : cpu에 의해 만들어짐
- physical address : 실제 memory의 주소
- logical/physical address sapce
  - 두 주소공간은 서로 관련이 없어야한다.
  - 두 주소공간은 매핑된다.
  - 두 주소공간은 표현하는 방식이 분리되어야 한다.(달라야 한다.)

MMU (Memory Menagement Unit)

- logical address와 physical address space를 매핑해주는 하드웨어 기기



- **relocation register** : MMU의 base register
- user는 physical address를 몰라도, 원하는 곳에 접근이 가능



Dynamic Loading

- 실행파일이 있을 때, 파일의 모든 것을 메모리에 올려야하나?
  - No, 너무 크기가 크다.
- dynamic loading : 메모리 공간의 효율을 높이기 위해 사용하는 로딩기법
  - 필요한 루틴만 부르자
  - process가 memory 공간보다 많은 공간을 요구하는 경우, 사용될 수 있다.
  - ex) error handling하는 코드는 자주 사용되지 않지만, 양이 많다.

Dynamic Linking and Shared Libraries

- **DLLs** : Dynamically Linked Libraries
- **static linking** : 바이너리 파일이 만들어질 때, system library와 program code가 합쳐진다. (프로그램이 실행되기 전에)
- **dynamic linking** : 프로그램이 실행될 때, 동적으로 링킹이 일어난다.
- **shared library**

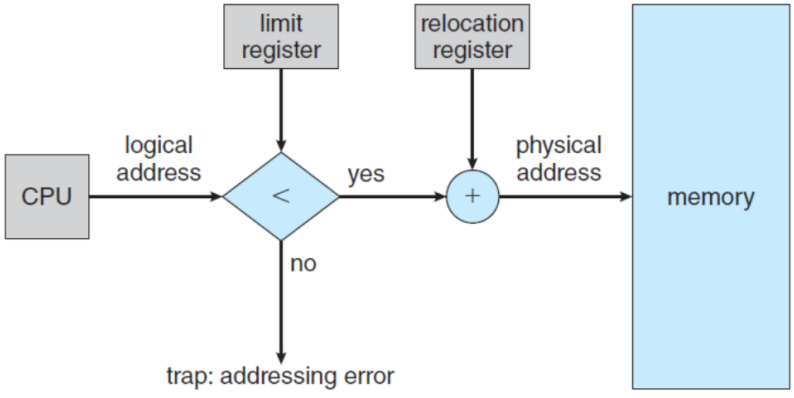
9.2 Contiguous Memory Allocation

Contiguous Memory Allocation

- 유저 프로세스를 그냥 **통째로** 메모리에 옮기는 것
- 어떤 하나의 section에 통째로 process를 올리므로, **연속적인 메모리 주소**를 가진다.
- 그러므로, **single section**으로 이루어져 있다.

Memory Protection

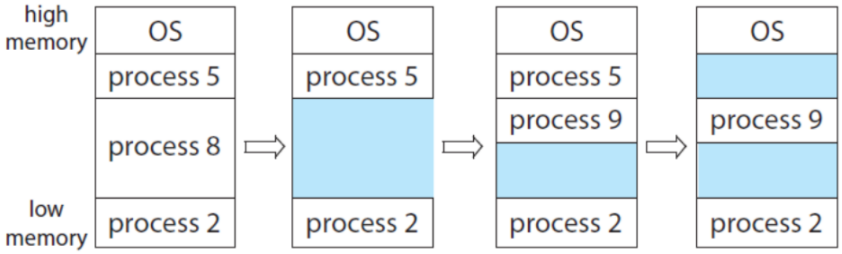
- 간단하다.
- **limit + relocation** 해서 process를 통째로 메모리에 할당하면 끝.



Memory Allocation

- 위 처럼 할당하면,
  - 프로세스 각각의 크기는 다양하기 때문에 **variable partition**을 가진다.
  - 하나의 프로세스를 **빈공간의 어디에 할당**해야하는지 이슈가됨

- **hole** : 가용한 메모리 영역(블록)
  - 즉, **hole을 어떻게 관리**할 것 인가가 이슈가 됨
  - 보통은 hole을 linked-list로 관리하게 됨



Dyniamic Storage Allocation

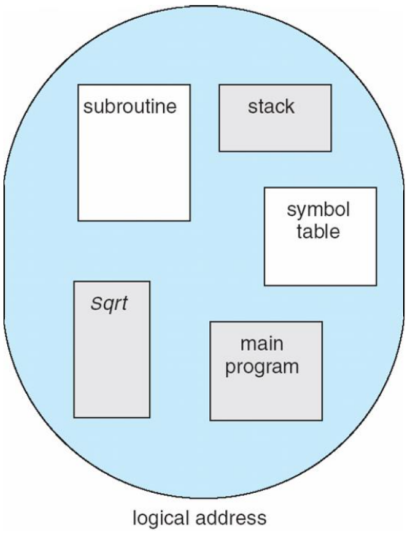
- 어떻게 프로세스를 hole에 할당할 것인가?
  - First-fit : 프로세스가 들어갈 수 있는 첫번째 hole에 할당
  - Best-fit : 프로세스가 할당될 수 있는 가장 작은 hole에 할당
  - Worst-fit : 프로세스가 할당될 수 있는 가장 큰 hole에 할당

Fragmentation

- **external fragmentation (외부 단편화)**
  - 프로세스를 contiguous하게 할당할 경우 생긴다.
  - 이 holes(fragmentation)은 프로세스를 contiguous하게 할당할 수 없게 만든다.
- **internal fragmentation (내부 단편화)**
  - 메모리를 같은 크기로 쪼개서 프로세스를 할당할 경우, 발생할 수 있다.
  - paging기법을 사용했을 때, 발생할 수 있다.

Segmentation

- contiduous와 paging의 그 사이
- 종류별로 쪼개서 올리자 (program영역, stack영역.. 등)



9.3 Paging

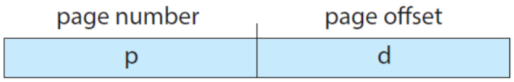
Paging

- physical address가 연속적이지 않음.
- 하나의 프로세스를 같은 크기로 쪼개서 메모리에 로드
- 장점은?
  - 외부 단편화가 (거의)발생하지 않는다.
  - 단편화를 모으는 작업(compaction)이 필요없다.
- 구현?
  - operating system + hardware 작업을 적절히 합쳐서 구현

Basic Method for Paging

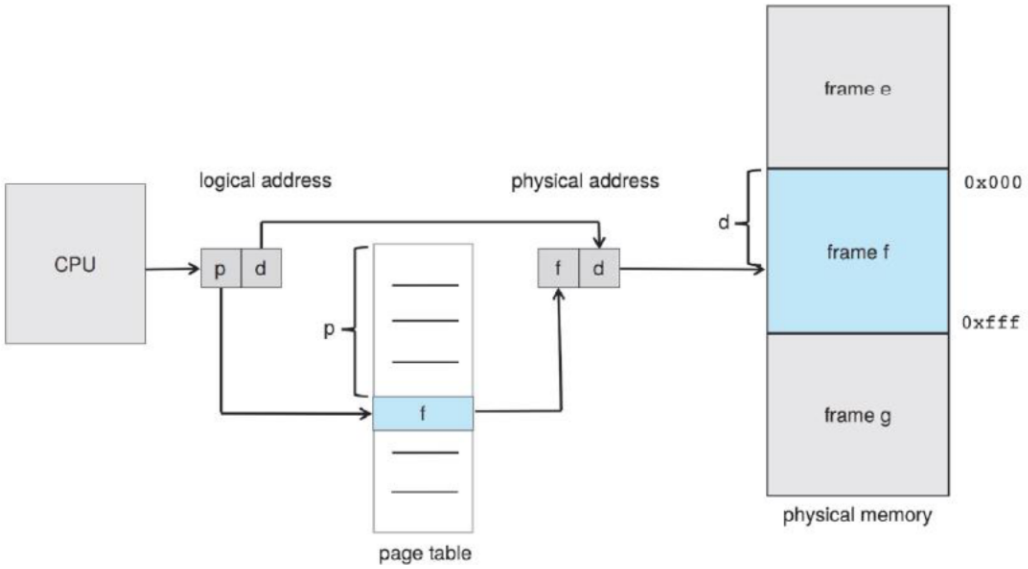
- physical memory를 고정된 size의 block(frame)으로 쪼갠다.
  - (1MB의 memory를 1kB씩 1024개의 frame으로 쪼갠다.)
- logical memory를 frame과 같은 사이즈의 block(page)로 쪼갠다.
  - (64kB의 프로세스를 1kB씩 64개의 page로 쪼갠다.)
- logical address의 공간은 physical address공간과 완전히 분리된다.
- 모든 주소공간은 page number(p)와 page offset(d)로 표현한다.

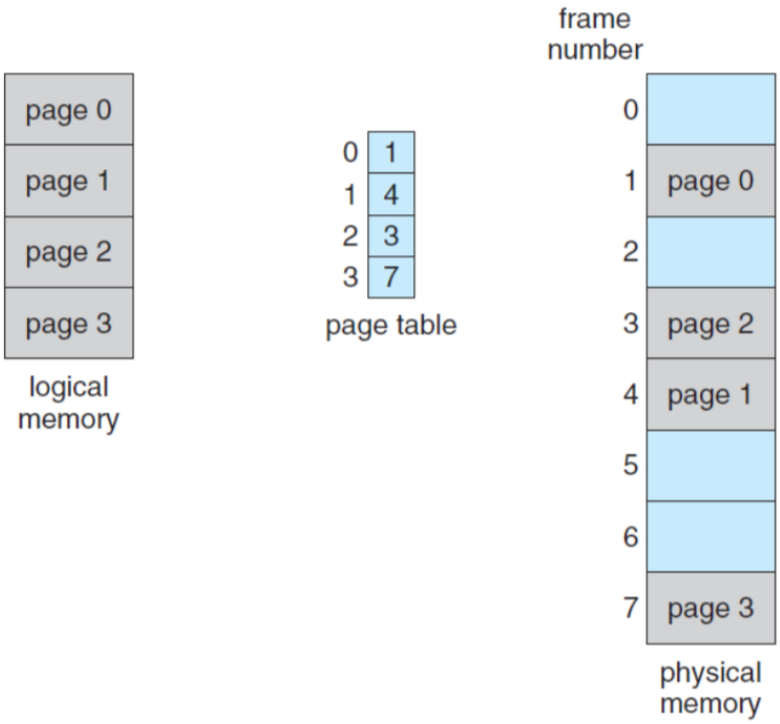
- (p번째 페이지, d번째 객체를 불러와라.)



The page number

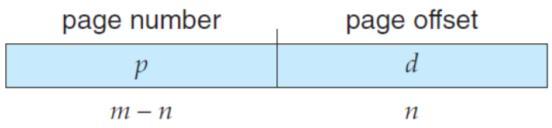
- page table을 관리해준다.
  - cpu는 **logical address(page number)**를 호출하면 **page table**을 통해 **physical address(frame number)**로 바꾸어 실제 메모리에 접근이 가능하다.



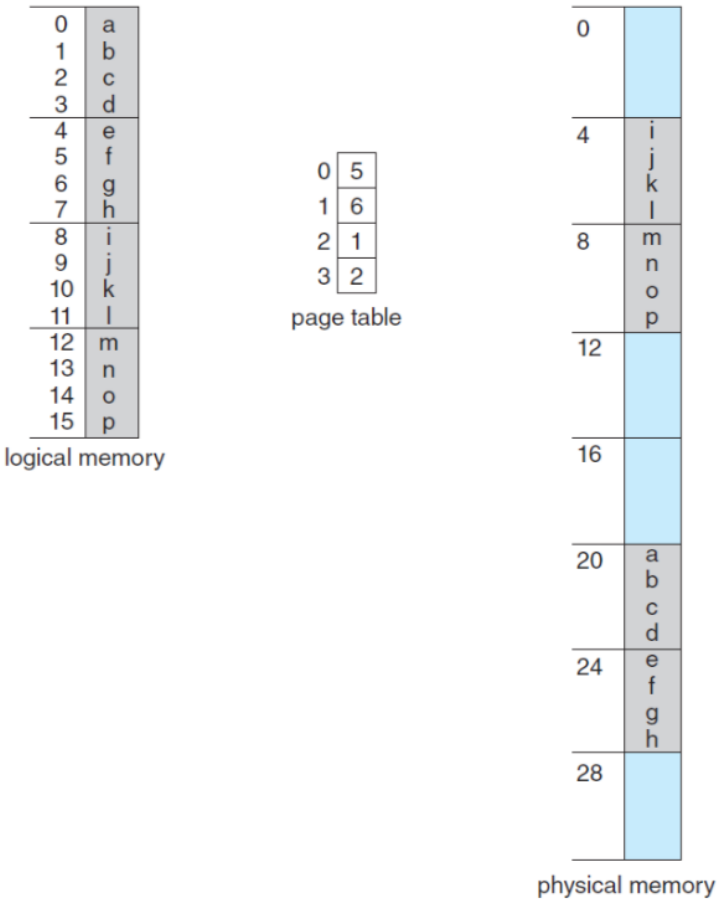


Page size (Frame size)

- 하드웨어에서 정의한다.
- 2의 거듭제곱 이어야 편하다.
- logical address공간이 2^m이고, page size가 2^n이면
  - `page number = logical address space/page size` 이므로, m-n 개가 된다.
  - `page offset = n bit`

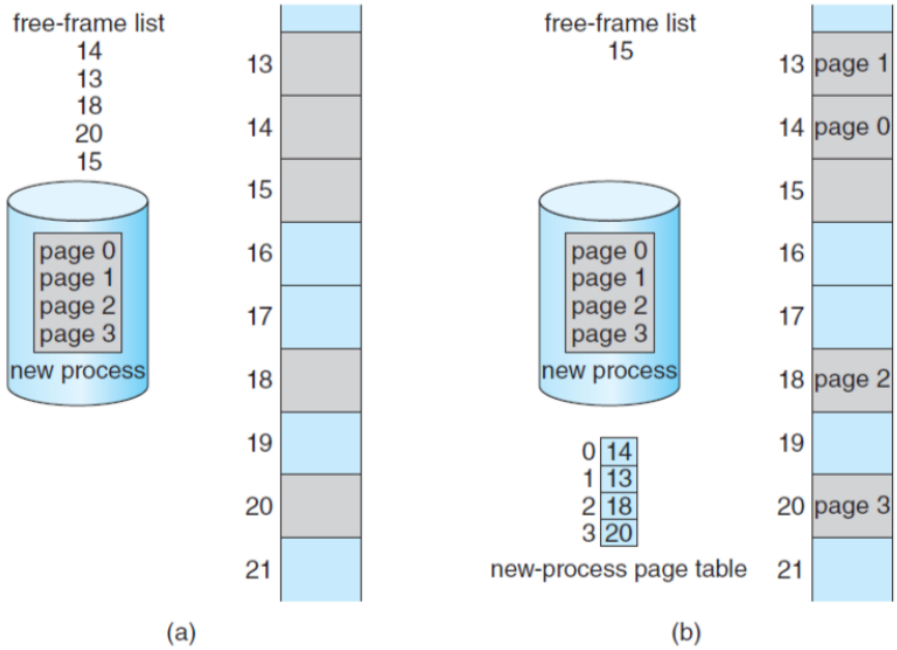


- n=2, m=4



free-frame list

- new process가 실행을 위해 도착 > free-frame list의 앞부터 읽으면서 page를 할당해 줌



Hardware Support

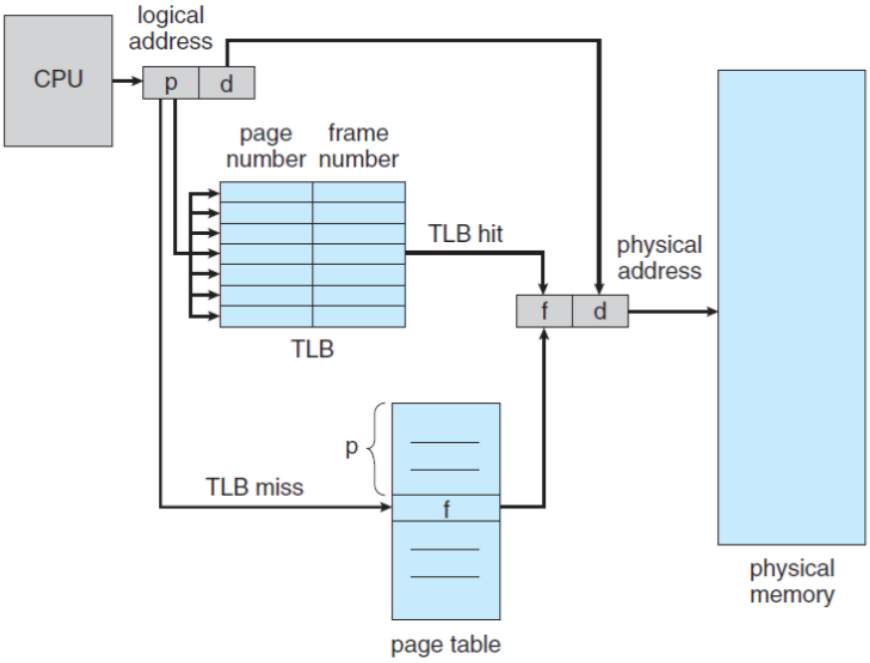
- cpu스케줄러가 다음에 실행할 process를 선택할 때,
  - **page table도 다시 로드**되어야 한다. (context switch 되어야 한다.)
- 하지만, 큰 프로그램의 경우 page table의 크기도 너무 큰 경우가 있다.
  - register value(pointer)를 PCB에 저장해두고 사용

PTBR (page-table base register)

- page table은 memory에 두고, **PTBR을 통해 접근**한다.
- context switch의 속도는 빨라진다. 하지만 여전히 memory access 속도는 느리다.
- memory에 두번 접근해야 함. page table에 접근, 실제 데이터에 접근

Translation Look-aside Buffer (TLB)

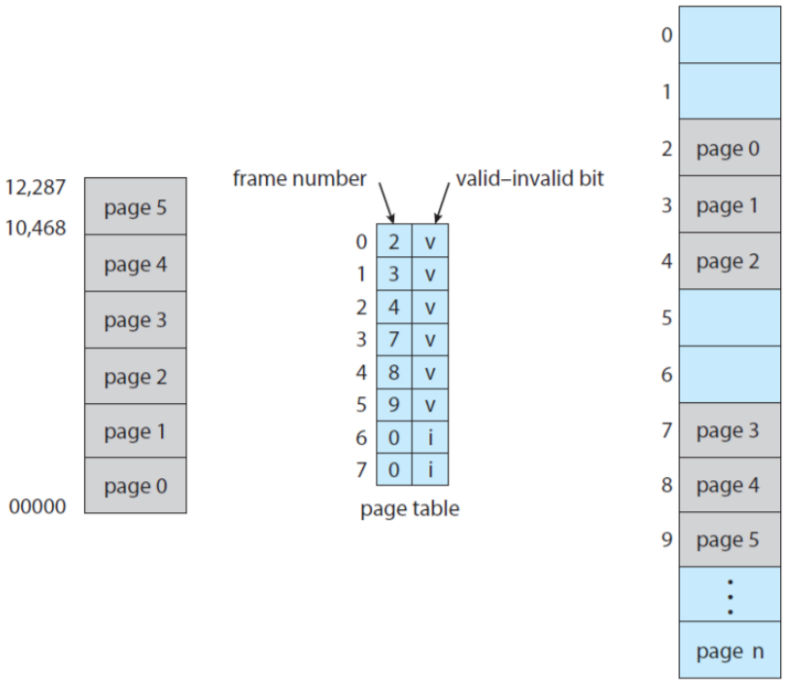
- **cache** memory (hardware)를 사용한다.



- 메모리 액세스 시간을 효과적으로 할 수 있다.
- **TLB hit** : TLB에 원하는 page number가 있는 경우
- **TLB miss** : TLB에 원하는 page number가 없는 경우
- **hit ratio** : 히트율을 계산해볼 수 있다.

Memory protection

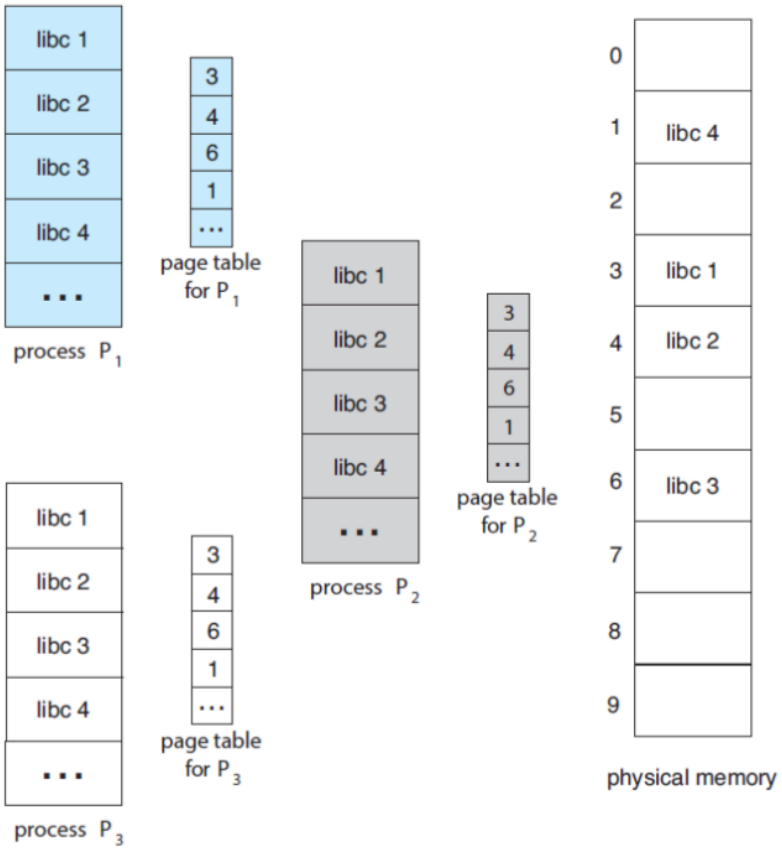
- contiguous의 경우, base/limit register를 통해 간단하게 관리했다.
- paging에서는 **valid-invalid bit**를 하나 추가한다.
- **valid** : 올바른 접근을 한 것이므로 legal
- **invalid** : 올바르지 않은 접근을 한 것이므로 illegal
- illegal address에 접근한 경우, valid-invalid bit를 통해 trap을 걸어준다.



Shared Pages

- paging의 장점은 공통된 부분을 sharing 할 수 있다는 것.
- 예를 들어, libc(standard C library)가 있다고 한다면, printf같은 함수는 다같이 공통으로 쓰는 함수임.
  - 그런데 프로세스별로 메모리 공간에 이 libc의 복사본을 가지고 있다고 가정하면, 굉장히 비효율적일 것임
  - 이것은 reentrant code 이므로 공유할 수 있음.
- **reentrant code란?**

  non-self-modifying code : 즉, 실행하면서 수정할 일이 없는 코드



- process 별로 libc는 가지고 있고, memory에는 하나만 로드해서 다같이 사용하자.

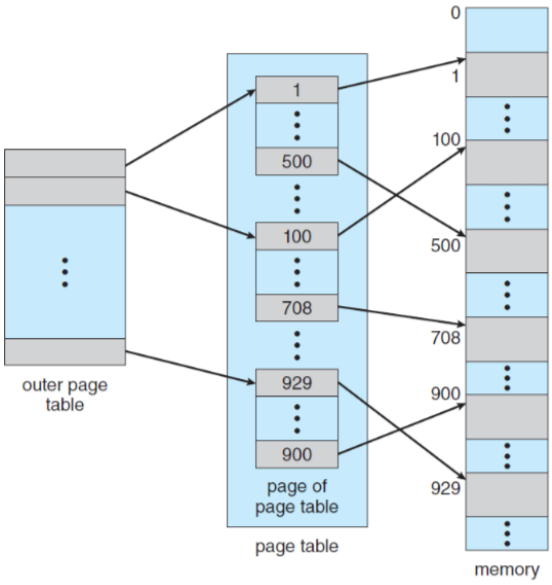
9.4 Structure of the Page Table

Structuring the Page Table

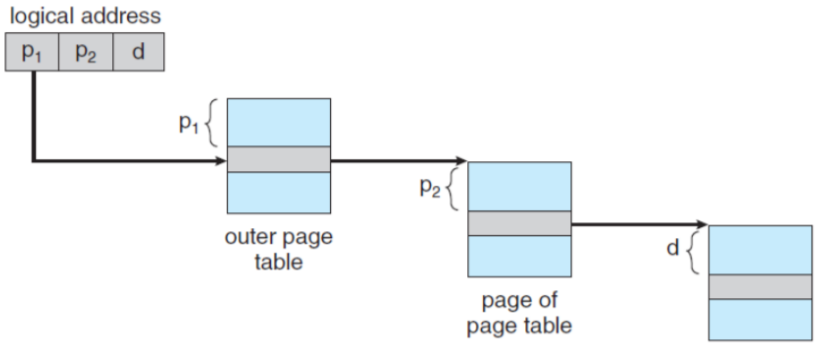
- 너무 크기가 커진 page table을 관리하자.

(1) Hierarchical Paging

- 계층적으로 관리
- page table에 대한 page table을 만드는 방식

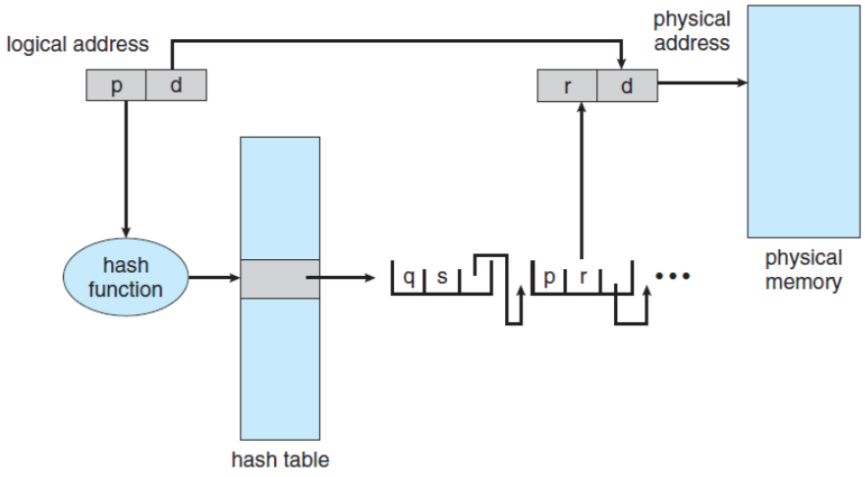


- 계층은 여러개 줄 수 있다.



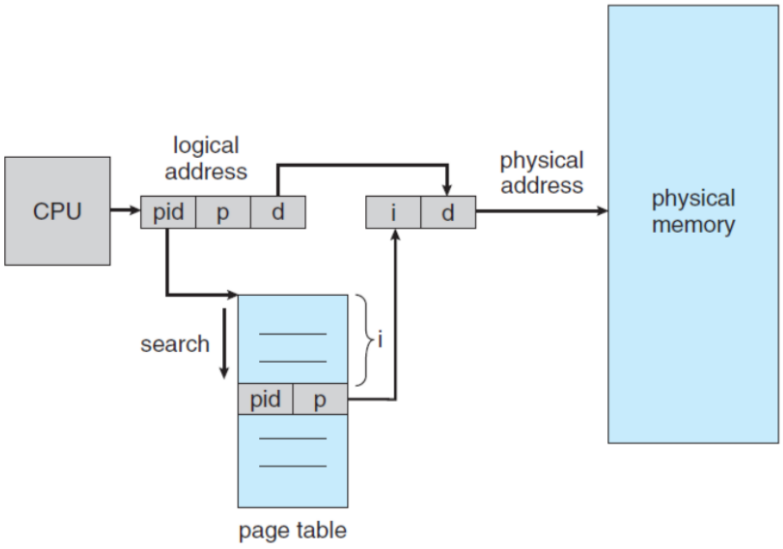
(2) Hashed Page Tables

- 해시테이블(함수)로 관리하자



(3) Inverted Page Tables

- 어떤 process가 어떤 page를 가지고 있는지 저장하자.
  - pid, page number의 쌍을 저장



e.

9.5 Swapping

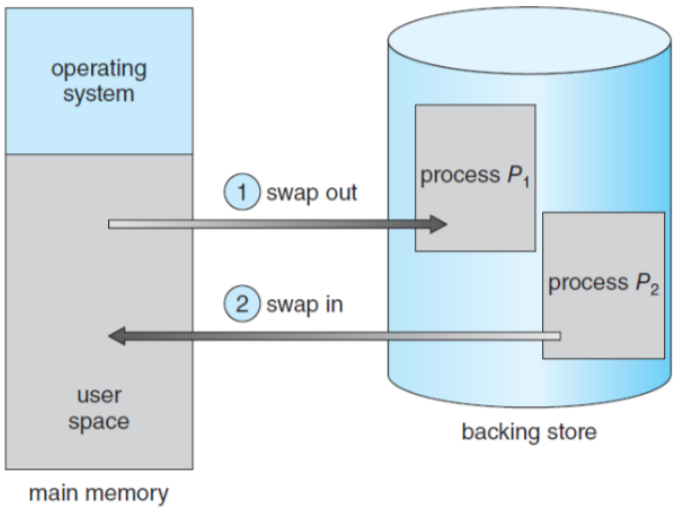
Swapping

- 실제 물리 메모리 공간보다 큰 process를 실행할 수 있을까?
  - Yes, 할 수 있다.

- 프로세스는 실행을 위해 메모리에 있어야 한다.
  - 하지만, 프로세스의 모든 부분이 메모리에 있을 필요는 없다.
  - 사용하지 않는 부분은 backing store(하드디스크)로 swap out 할 수 있다.
  - 그리고 필요할 때, 다시 메모리로 swap in 할 수 있다.

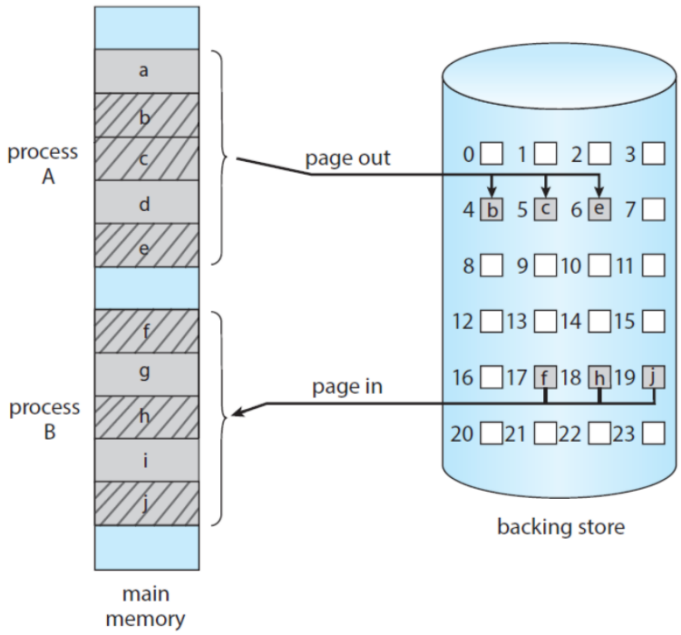
Standard Swapping

- process 전체를 swap out, swap in 하는 것
- 하지만, 전체를 swap하는 방식은 비효율적



Swapping with Paging

- 전체 프로세스를 swap하기보다, **page단위**로 swapping을 한다.
- swap in, swap out 대신, **page in, page out** 이라고 부른다.


a
b
c
d
e
f
g
h
i
j
process A
process B
page out
page in
0 1 2 3
4 b 5 c 6 e 7
8 9 10 11
12 13 14 15
16 17 f 18 h 19 j
20 21 22 23
backing store
main memory