8. Memory management 1

Program이 시작되기 위해서는 Disk에서 Memory로 올라와야 한다.

CPU는 Memory와 Register에만 직접 접근이 가능하다.

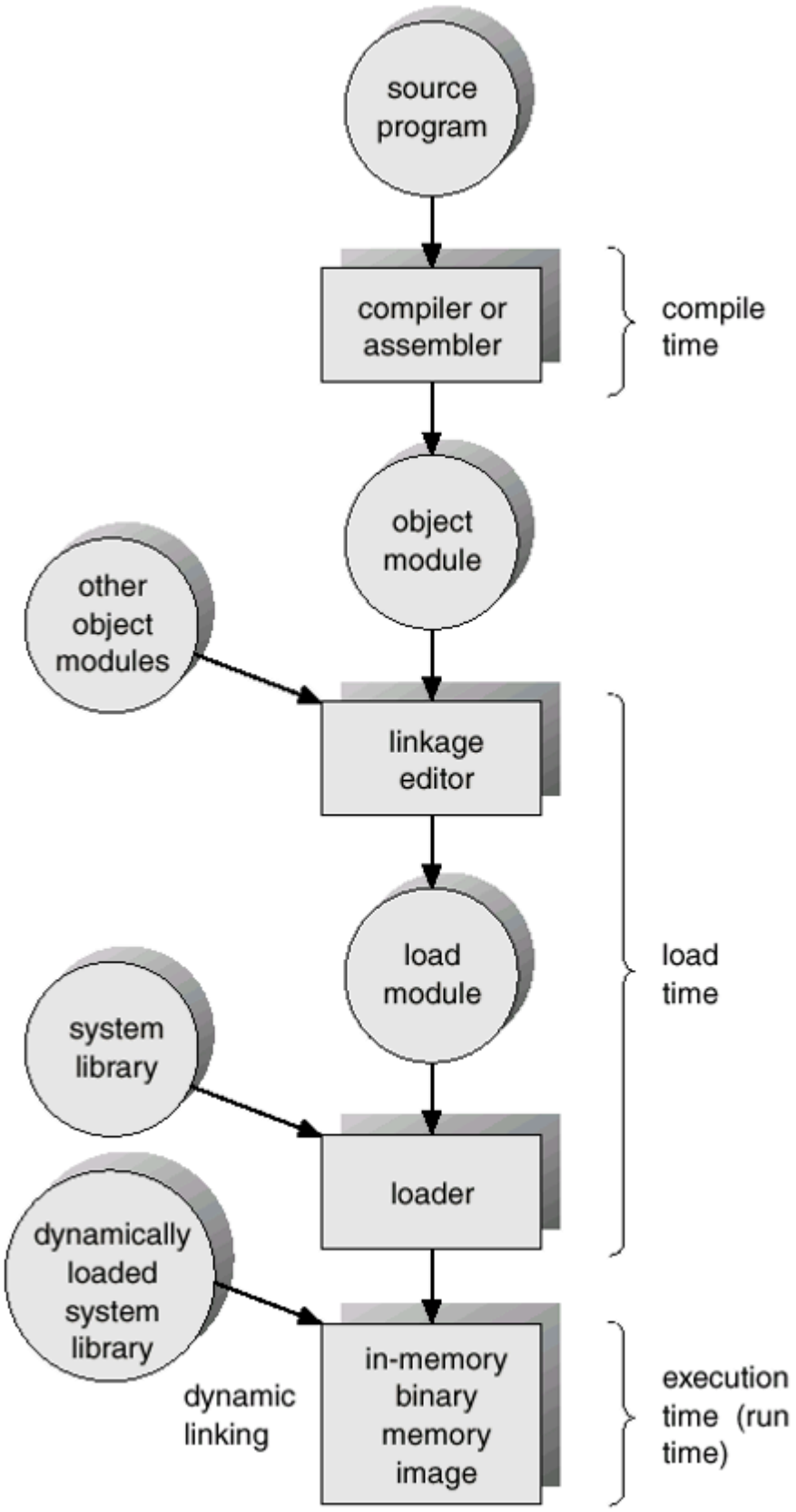
- `Register`에는 1 CPU cycle 이내에 접근 할 수 있어야 한다.
- `Memory`는 더 많은 cycle이 소요된다.

`Cache`는 Memory의 느린 접근 속도를 보완하기 위해 Register와 Main memory 사이에 위치한다.

`Protection` : 하나의 Process는 다른 Process의 Memory 영역이나 Kernel address에 접근하면 안 된다.

- Kernel address에 접근 가능한 경우는 System call을 호출한 경우이다.

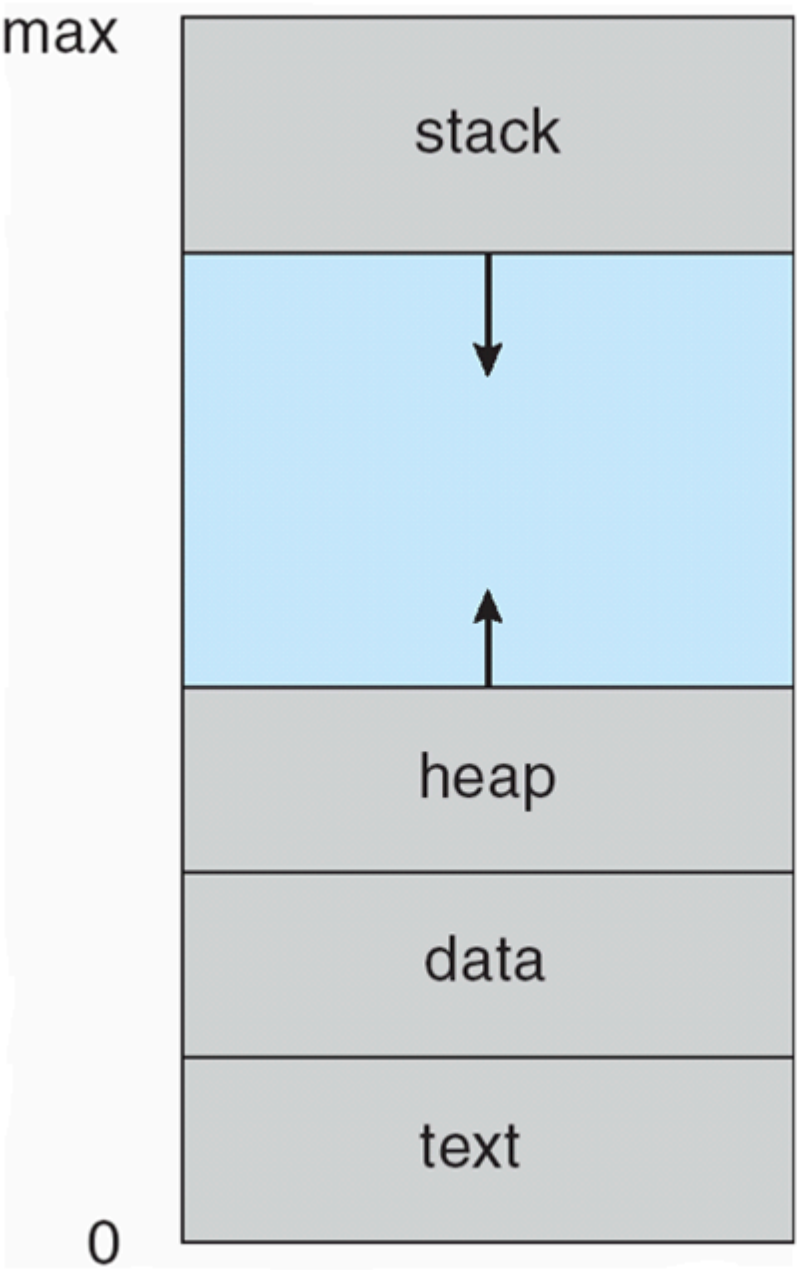
Multistep Processing of a User Program



- Linker는 하나의 Program에 존재하는 다양한 object file을 묶는다.

- `System library` : 소스 코드에서 호출한 Library
- `Loader`에서 Process를 memory에 올린다.

Process in Memory: Address space



- 0 ~ MAX 의 주소 공간을 `Address space` 라고 한다.
- `Stack` 에는 Local variable, `data` 에는 global variable이 저장된다.

    - 즉, 매핑이 필요하다.
- `Heap` : 동적 할당
- `Text` : 코드가 저장된다.
    - PC는 0부터 시작하여 1씩 증가하며 Text 영역의 코드를 확인한다.
- variable은 변수명에 따라 주소가 정해져 있어야 한다.

Binding of Instructions and Data to Memory

- 변수, 명령어를 메모리 상의 정확한 주소와 매핑시키는 것
- `Binding` 이 이루어지는 순간에 CPU는 변수의 값을 확인할 수 있다.

1. **Compile time binding**
    a. Compile time에 각 `symbol` 에 대한 절대 주소가 할당
    b. `Compliler`가 `Absolute code` 를 만들어 사용한다.
        - 절대 주소를 사용하는 Machine code이다.
    c. 실행 환경이 변하면 Recompile 해야 한다.
        - Flexibility가 낮다.
    d. Multitasking이 사용되지 않던 옛날에 사용하던 방식이다.
        - 이 경우, kernel / User만을 나누고 User 영역에는 하나의 Process만 올리면 되므로 Compile binding이 쉬웠다.
        - 최근엔 사용하지 않는다.

2. **Load time binding**
    a. `Loader` 가 Load time에 `absolute address`를 할당
        - Memory에 올릴 때, (**Memory에서의 코드 시작 주소 + Offset**)으로 절대 주소를 결정
            - 메모리에서의 시작 주소만 알면 된다!!
        - Compiler는 Compile time에 `Relocatable code` 를 생성한다.

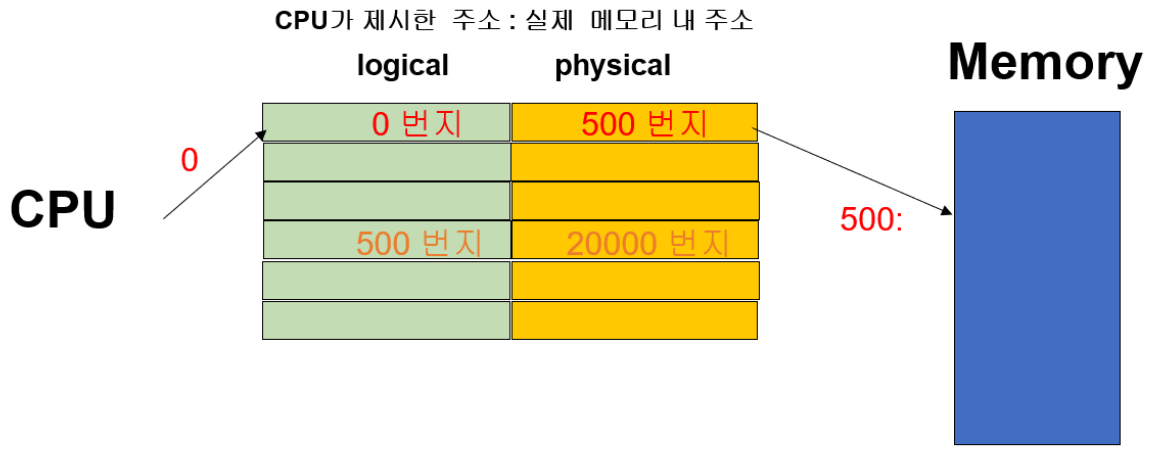
    - 어차피 offset을 사용하는 것을 알기에, Program code가 0번지부터 시작함을 가정하고 생성한다.
  - **Multitasking을 지원**한다.
  - 환경이 바뀌어도, Compile time에는 어차피 relative하게 처리하므로 Recompiling 할 필요가 없다.
  - 그러나, 한번 Load 후에는 **process address space 주소 값이 바뀔 수 없다는 단점**이 있다.

3. **Execution time binding**
   a. Load시까지 상대 주소 사용하다가 CPU가 relative address로 접근할 때, 실제 주소값으로 변환한다.
   b. CPU가 주소를 생성할 때마다, **mapping table**을 통한 Binding이 필요하다.
   c. `MMU` 같은 Hardware적인 지원이 필요하다.
      - MMU는 mapping을 지원하는 hardware unit이다.
      - Hardware를 사용하는 이유는 데이터 접근 시마다 mapping이 필요하기 때문에 이 과정이 하드웨어적으로 굉장히 빠르게 처리될 필요가 있기 때문이다.
   d. Relative address는 변하지 않을 때, 실제 Memory 상에서의 주소가 변해도 OS는 mapping table만 바꾸면 된다.
      - 이를 통해 실행 중, Memory 상에서 이동이 가능할 수 있게 한다.
      - 이는 메모리의 효율성, 안정성 측면에서 중요하다.

중요한 점

- Relative address 0번지 ≠ Absolute address 0번지 (= Kernel의 시작 주소)
- `Binding` **은 위처럼 실제 Binding 시점에 따라 3가지로 분류된다.**
- `Multi-tasking` : 모든 주소가 Relative address에서 0번지부터 시작한다면, 가능하다.

Address Mapping Table



- **Logical → Physical Mapping**해주는 table이 존재

Base & Limit Registers

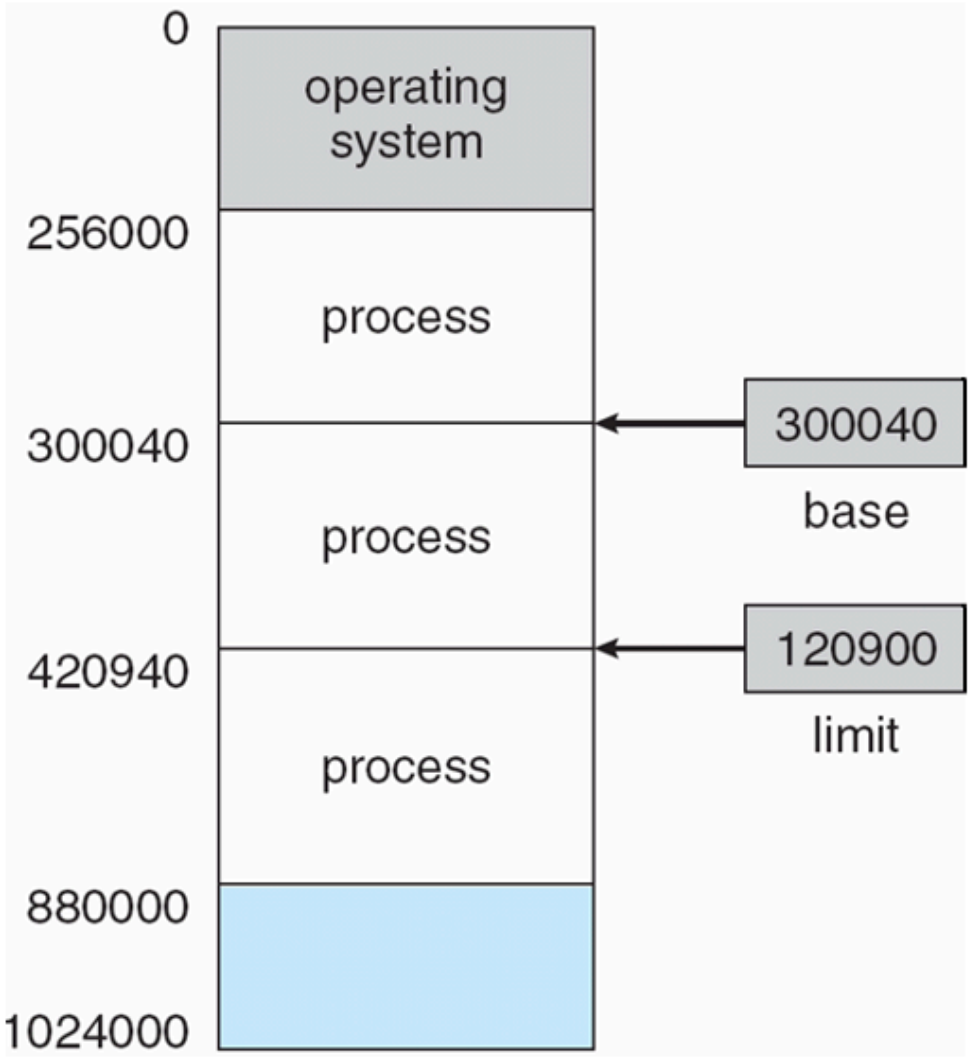
Multi-tasking의 경우에 여러 Process가 동시에 Memory에 올라가 있는 경우를 생각해보자

이 경우에는 각 process의 Memory 영역을 구분해주어야 한다.

- 각 Process의 **Memory 상에서의 시작 주소와 길이를 저장할 공간**이 필요하다.
- `PCB`에 이를 저장하는 변수가 존재한다.

`Base Register` 와 `Limit Register` 에 이 값을 저장할 수 있다.

- Process가 시작할 때, PCB에서 이 reg로 값을 Load



- 보이는 것처럼 Limit register 는 길이를 나타낸다.
- 중간에 Memory에서 쫓겨 나갔다가 다시 Memory로 올라오는 경우, Base address 는 바뀔 수 있다.
    - 이 관점에서 Base를 Reallocation register 라고도 한다.

Logical address vs Physical address

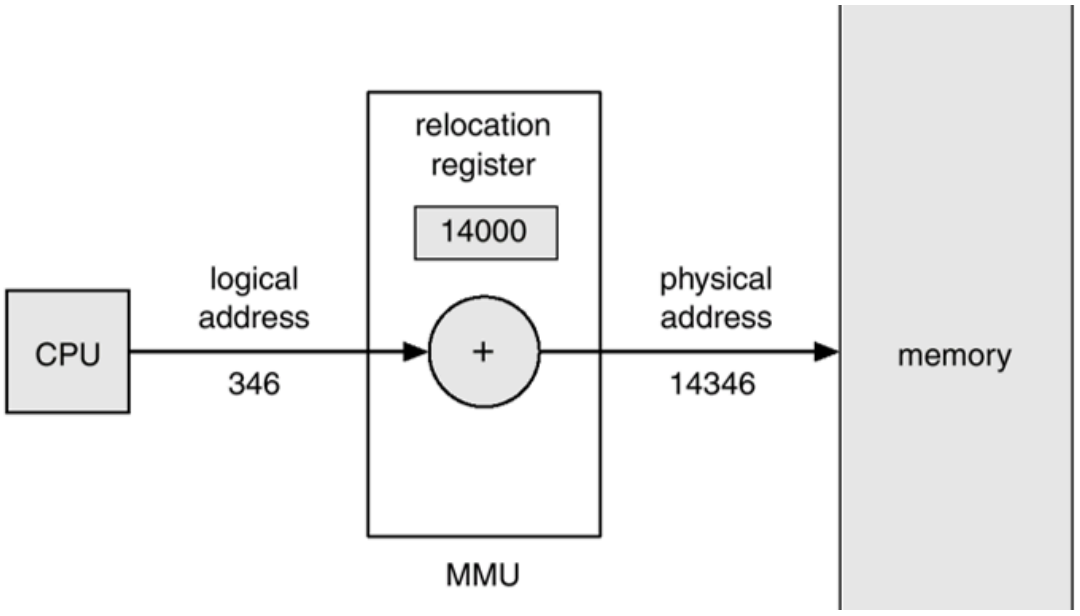
Logical address 와 Physical address 를 분리하는 이유

    - CPU는 Physical address를 모른 채로 memory에 접근하고, Physical address로의 mapping은 OS가 관리한다.
    - 이를 통해 각 변수마다 실제 주소를 몰라도 되고, process마다 독립적인 메모리 공간을 가질 수 있으며, **OS는 메모리 관리, 보호를 효율적**으로 할 수 있게 된다.

- `Compile time / Load time binding은 compile / load` 시 각 logical address가 physical address와 동일해진다.
    - CPU가 실제 DRAM 상의 주소를 알게 된다.
- 그러나 `Execution time binding`의 경우, Logical address와 Physical address가 다르다.
    - 최근에 사용하는 방법이다.

Memory Management Unit (MMU)

- Logical → Physical mapping 해주는 Hardware device



- Memory에 탑재될 때, Base Register의 값에 Logical address를 더하면 Physical address를 알 수 있다.
- 그러나 위 방법 대신, 우리는 CPU가 logical address만을 다루고, Physical address는 절대 확인하지 못 하도록 한다.

- 위의 예시처럼, CPU가 logical address를 이용하여 요청을 할 때, `MMU`가 `relocation register`에 저장되어 있는 값에 logical address를 더해서 memeory에 전달해준다.

Swapping

- `Mid-term scheduler` 에 의해 수행된다.
- Process는 일시적으로 **swapped 될 수 있다.**
  - 임시로 `Backing store` 에 저장되었다가 Memory로 복귀할 수 있다.
  - Suspend (CTRL + Z) 된 경우에 swapped 된다.
- Swapping 이후에 메모리에 복귀했을 때, Base register에 저장되어야 하는 값이 달라질 수 있다.
  - 즉, 메모리 상에서의 시작 주소가 달라질 수 있다.
  - 때문에, Swapping은 **Execution time binding**에서만 가능하다.

Backing store

- Swap-out 된 process가 저장되는 공간
- 모든 `Memory image` 를 저장할 수 있는 **크고 빠른 DIsk**
  - Memory image란? "프로세스를 실행하는 데 필요한 **전체 메모리 상태 복사본**"
- CPU가 Backing store 상에서 원하는 Memory image에 direct access 할 수 있도록 제공한다.
- Process 의 Address space가 크기 때문에, **Backing store에 저장 및 읽는 속도는 굉장히 빨라야 한다.**
  - 이를 위해 연속적으로 읽고, 쓰는 I/O mechanism을 도입한다.
  - 반대로 File system의 경우에는 빈공간에 바로 할당하고, 비연속적인 메커니즘을 사용한다.

Swap time 중에 대부분은 Data transfer time이고, Data trasfer time은 Swap 대상인 Memory의 크기에 비례한다.

Contigous allocation

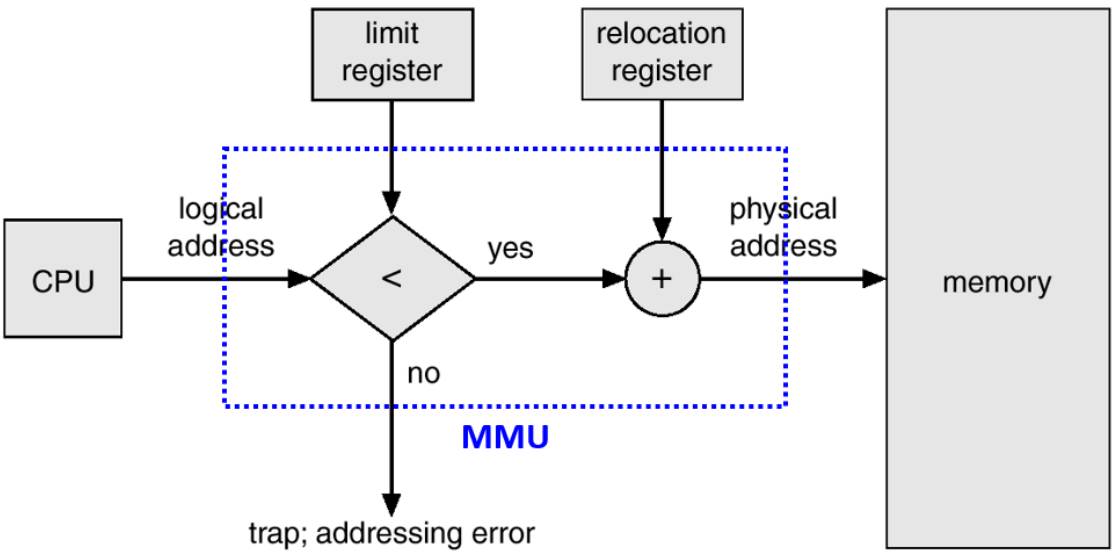
- Process를 DRAM에 연속적으로 올리는 방법
- MMU mapping의 가정: “Process의 address space가 memory 상에서 연속적으로 저장되어져 있다.”
- 하나의 Process의 address space는 DRAM에 연속적으로 저장한다.

Main memory는 보통 두 파트로 나뉜다.

1. Resident Operating System
    - Low memory에 interrupt vector와 저장
    - Interrupt vector 는 interrupt 발생 시, handler의 시작 주소를 적어놓은 vector 이다.

2. User Process
    - High memory에 저장
    - 이 영역을 보호하기 OS 영역과 다른 Process의 영역으로부터 보호하기 위해 Relocation register와 Limit register 를 사용한다.
    - Limit register는 Logical address의 범위를 지정한다.

따라서 MMU 는 아래와 같이 동작한다.



- CPU가 제공한 logical address가 limit register에 저장된 값보다 작은지 확인하는 부분은 필수적이다.
    - 이 조건이 맞지 않다면, 다른 process의 영역을 침범하는 것이므로 치명적인 오류이다.

Continous allocation의 경우 `Hole` 이 발생한다.

- Hole은 **연속적인 Memory 상의 빈 공간**을 의미한다.
- 보통 `Swapping` 으로 인하여 발생한다.
- 하나의 Process가 연속적으로 할당되기 때문에, Hole의 크기는 제 각각이다.
- Process address space가 process size마다 다르기 때문이다.

OS는 Hole을 관리해야 한다.

- 새로운 Process가 도착했을 때, 해당 Process의 크기보다 큰 Hole 중 하나에 process를 할당할 수 있어야 한다.
- 이를 위해 OS는 hole에 대한 다음 정보를 알고 있어야 한다.
    - Hole의 시작 주소
    - Hole의 크기
    - 어떤 Hole이 존재하는지
    - `Allocated space` : 현재 Process가 사용하는 공간
    - `Free space` : 현재 사용하지 않는 공간 (= Hole)

Dynamic Storage-Allocation probelm

Process가 도착했을 때, 어떤 Hole을 할당할지는 굉장히 큰 문제이다.

- **지금 시작되는 Process에 어떤 Hole을 할당하는 지에 따라서, 미래의 특정 Process가 실행되지 못 할 수도 있다.**
- 특정 시점에 항상 Optimal 하게 결정하는 것은 불가능하다.

Process에 Hole을 할당하는 아래 세 가지 방법이 있다.

1. `First fit`
    - Process 크기 이상인 Hole 중, **가장 큰 hole을 할당**

2. `Best fit`
    - Process 크기 이상인 Hole 중, **가장 작은 hole을 할당**
    - Hole이 크기 순으로 정렬되어져 있지 않은 경우에, 전체 Hole을 크기 판단을 위해서 탐색해야 한다.
    - `Leftover hole` 이 생성된다.
        - `Best fit`으로 Hole을 사용하고 남은 공간은 너무 작아서 재사용이 어렵다.

3. `Worst fit`
    - 가장 큰 Hole을 할당
    - 가장 큰 Hole을 찾기 위해, 전체 Hole을 탐색해야 한다.
    - `Large leftover hole` 이 생성된다.
        - `Best fit` 에서의 경우보다 크다.
        - 재사용이 가능할 수도 있다.

어느 방법이 좋다고 명확하게 말하기는 어렵지만, **보통 First-fit과 Best-fit이 Storate utilization 관점에서 Worst-fit보다 좋다.**

Fragmentation

1. `External Fragmentation`
    a. 할당되지 않은 공간이나, 크기가 너무 작아 연속적으로 할당될 수 없는 공간
    b. 예시: Best-fit에서 너무 작아 할당되지 못 한 작은 공간

2. Internal Fragmentation
    a. 이미 할당되었지만, 사용되지 않는 공간
    b. 예시: 요청한 메모리보다 더 많은 메모리 공간이 할당된 경우

External fragmentation을 줄여보자

→ Campaction

- 여러 External fragmentation을 **하나의 큰 메모리 공간**으로 합쳐서 사용
- Compaction이 가능하긴 위해선 Execution time binding 방식을 사용해야 한다.
    - Relocation이 Dynamic 해야 하기 때문이다.
    - Base register 가 변해야 한다.
        - 이는 Memory copy가 많이 일어나게 하므로 Heavy하다.

External fragmentation을 아예 발생하지 않게 만들어 보자

Paging

Paging

- **Address space가 contiguous 하지 않아도 되도록 만든다.**
- Memory와 Address space를 모두 **같은 size의 block 여러 개**로 쪼개놓고, 끼워넣는다.
- 하나의 Block 은 연속적이어야 하지만, 각 Block이 연속일 필요가 없다.
    - Process address space가 Noncontiguous 해도 된다.

Frame

- Physical memory를 **fixed-sized block**으로 쪼갠 단위를 의미한다.
- 2의 지수배의 크기를 갖는다. (512 bytes ~ 8MB)

Page

- Logical memory를 **fixed-sized block**으로 쪼갠 단위를 의미한다.
- 보통 4KB

Basic method

- Free frame을 tracking 해야 한다.
- n개의 page를 갖는 process를 실행하기 위해서는, memory에 n개의 free frame이 존재해야 한다.
- Page table **이 필요하다**
  - 이전에는 연속적으로 저장되므로 Base address만 mapping할 수 있으면 됐다.
  - 하지만 지금은 하나의 page만 연속적으로 저장되므로, **모든 page에 대해 mapping이 가능한 Page table이 필요하다.**

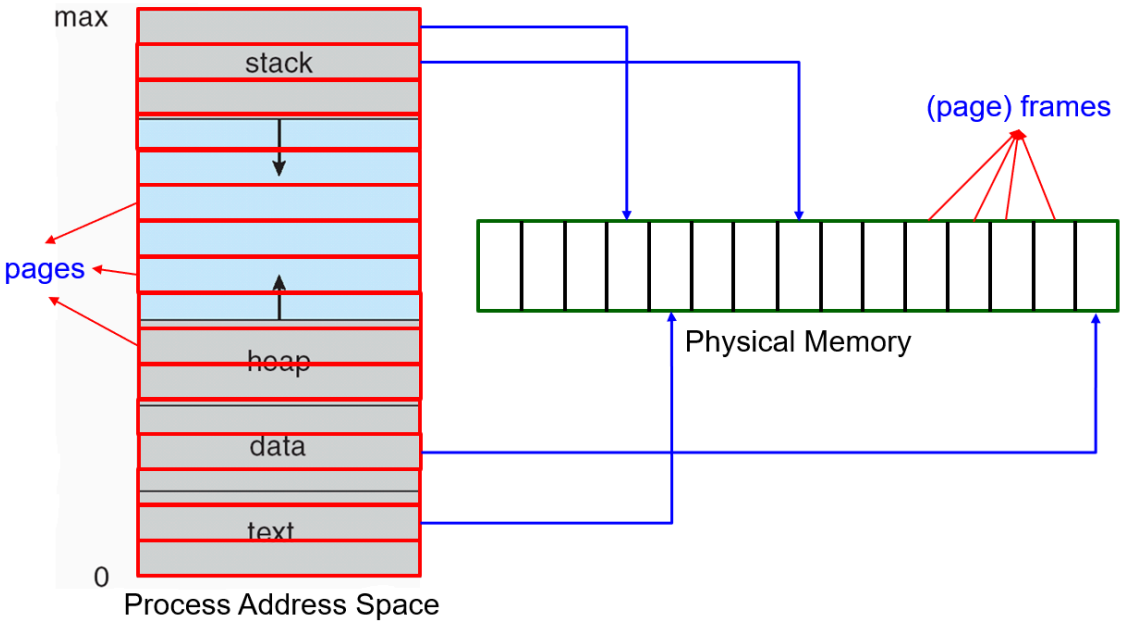
Paging 사용 시, External fragmentation은 절대 발생하지 않는다.

- 메모리에서 남는 공간은 무조건 Frame 단위이기 때문이다.

그러나 Internal fragmentation은 여전히 존재한다.

- 각 Process의 마지막 page의 경우에 1KB만 저장하면 되는 경우에 4KB page를 할당해야 한다.
- 그러나 각 process의 마지막 page에서만 발생하므로, 그렇게 심각한 문제는 아니다.

Paging을 시각적으로 나타내면 아래와 같다.



Paging에서의 Address Translation 과정을 살펴보자

먼저, page는 무조건 **2의 지수의 size**를 가져야 한다.

CPU가 제공하는 Logical address를 아래와 같이 분리한다.

- **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

page number	page offset
p	d
(m – n) bits	n bits

- For given logical address space 2^m *and page size* 2^n

- 이 경우, page의 크기는 **2^n** 이어야 한다.

Paging 시, 메모리가 연속적으로 저장되지 않아 Base register가 따로 필요하진 않다.

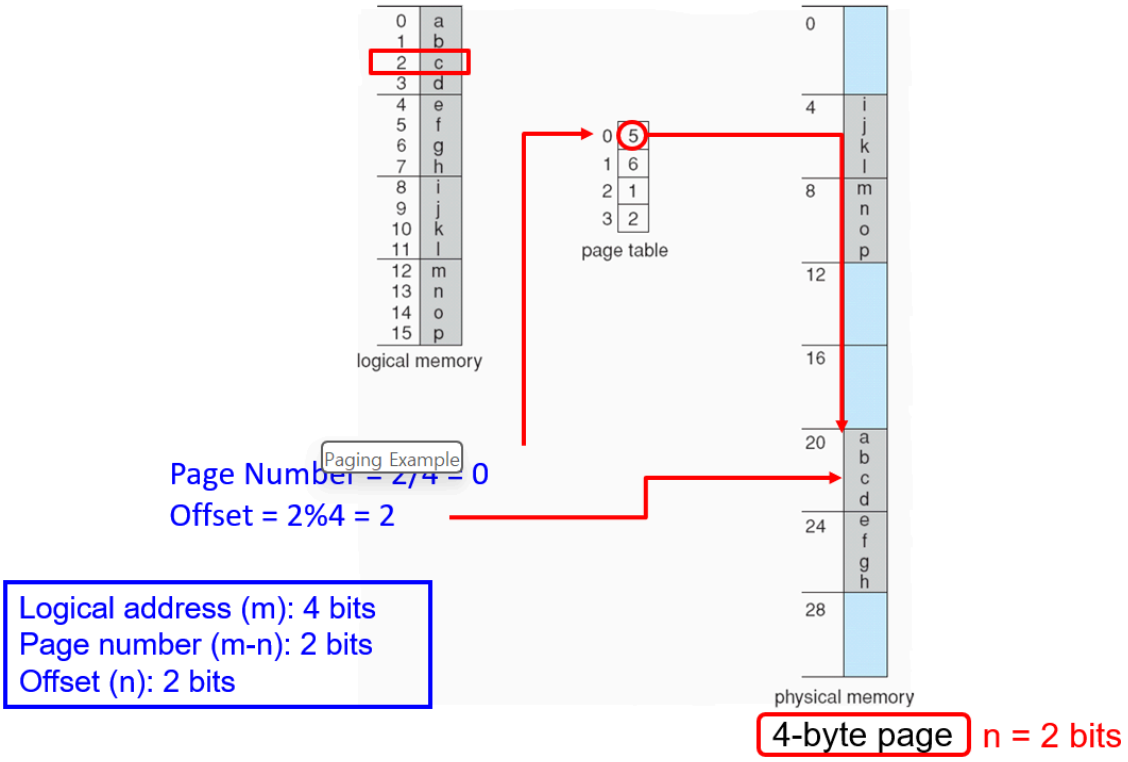
대신 page number에 해당하는 부분을 page table의 index로 사용한다.

- Page table entry의 개수는 **2^(m-n)** 이다.

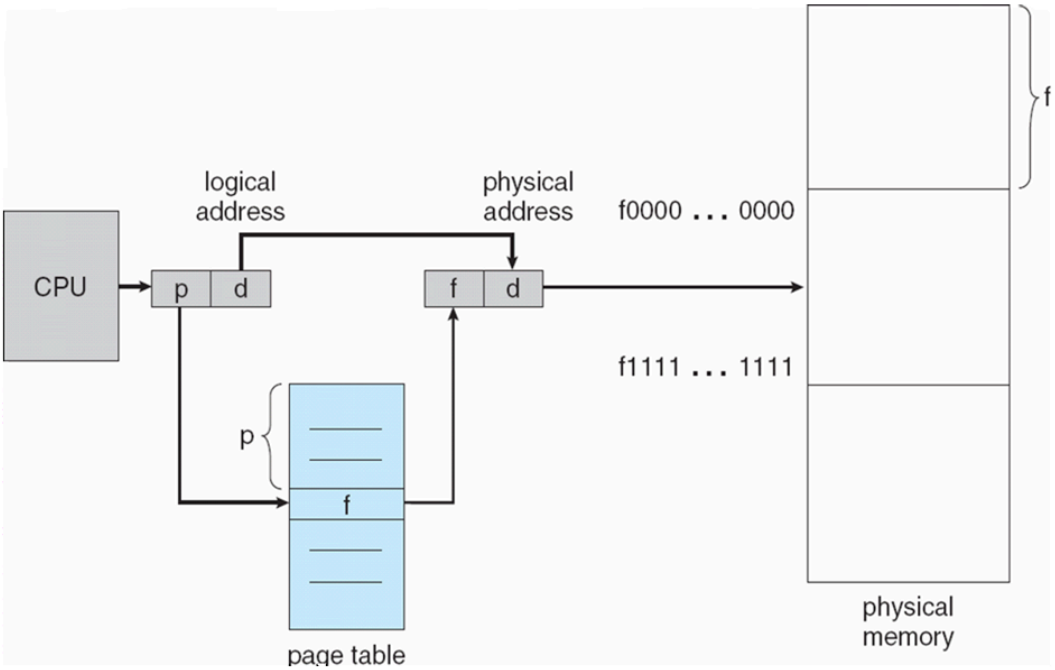
Page table은 각 page의 시작 주소를 저장하도록 한다.

- 이 시작 주소에 page offset을 더하여 원하는 데이터를 가져올 수 있다.

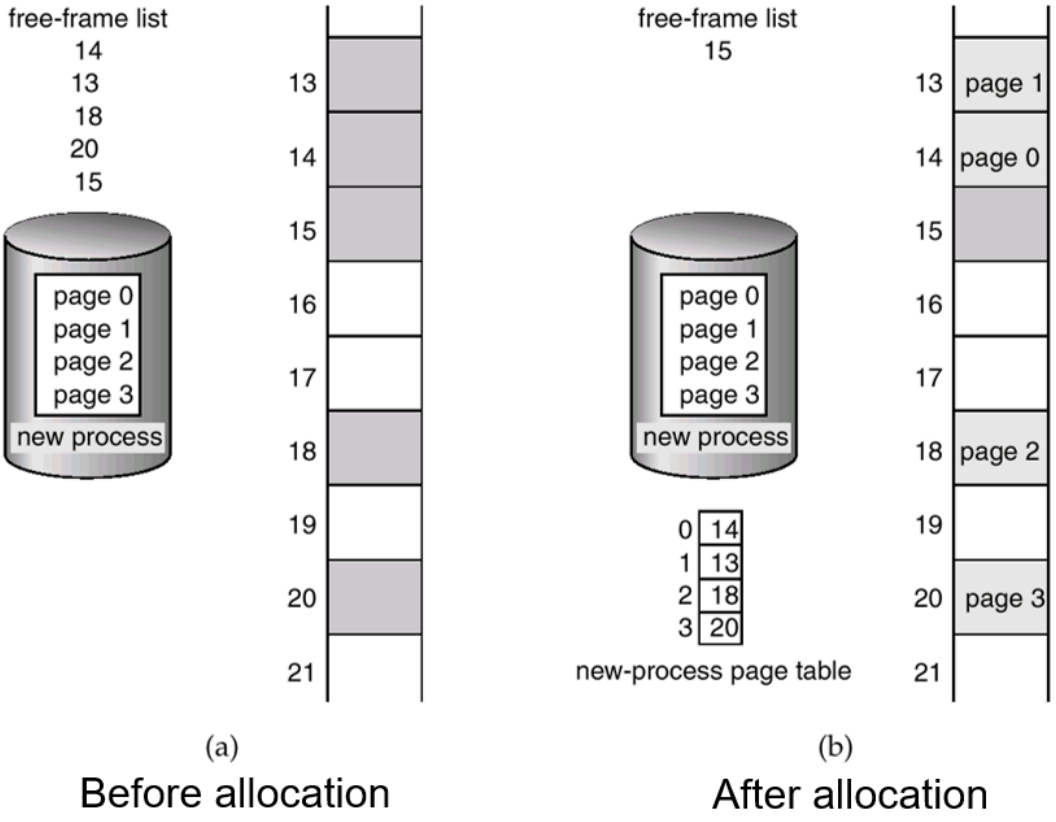
Paging Example



- PTE가 4 = 2^2 이므로, 2bit + page offset이 0 ~ 3이므로 2 bit 하여 logical address는 총 4 bit 사용
- 여기선 Page index 는 4로 나눈 몫, Page offset은 4로 나눈 나머지를 사용하도록 하였다.
- **Page num = 마지막 주소 (해당하는) / Page size**
- **Page offset = 마지막 주소 (해당하는) % Page size**



- Page number로 Page table에서 원하는 page의 시작 주소를 얻어오고, Page offset을 더해 실제 원하는 데이터에 접근할 수 있다.



- Free frame list의 위에서부터 순서대로 채운다.
- Page number에 맞게 page table을 구성한 것도 확인할 수 있다.

Q) Free frame이 부족하면?

→ Virtual Memory 를 사용한다.

Page table 구현

Page table은 Paging을 위해 OS가 관리하는 자료구조이다.

Page table은 Main memory에서 저장되고 관리되어야 한다.

- CPU가 참조해야 하기 때문이다.

- 또한 CPU가 참조하기 위해서는 Page Table의 실제 주소도 알아야 한다.
  - **Page-table base register (PTBR):** Page table의 Memory 상에서 시작 주소를 저장
  - **Page-table length register (PLTR):** Page table의 size를 저장
  - MMU가 Page table에 접근할 수 있도록 address traslation

문제점) 모든 메모리 접근 명령어마다 **page table 접근 + Data / Instruction 접근 총 2번의 Memory 접근**이 일어난다.

- Memory 접근은 Overhead가 크다, → 전체적인 실행 시간이 증가

해결) Fast-lookup hardware cache 사용

- **Translation Look-aside Buffer (TLB) 또는 Associative cache**라고 부른다.
- Memory에 있는 Page table의 일부를 담고 있는 빠른 메모리
  - Page table에 접근하기 위해 Memory를 확인하는 Overhead를 줄여줌
- **TLB는 빨라야한다.**
  - TLB size를 작게 만들면서 Locality를 활용해 TLB hit를 높여야 한다.
- TLB Miss의 경우, 그냥 Memory access 한다.
- 일반적으로 TLB는 Context switch time에 flush
- 몇몇 TLB는 address-space identifiers (ASIDs)를 각 TLB entry에 저장하도록 해 TLB Flushing을 피하도록 한다.

`Associative Memory`

- TLB의 경우, 용량이 작아 모든 Page를 저장하지 못 한다.
- 따라서, Offset을 활용해 바로 접근하지 못 하고, 직접 TLB를 돌며 찾아야 한다.
  - 이 때문에 TLB access가 오래 걸릴 수 있다.
  - 이렇게 되면 TLB 접근 횟수가 증가할 수록 TLB를 사용할 이유가 없다.
- 이를 보완하기 위해, index에 따라 **parallel search**하는 `Associativie memory` 를 사용한다.

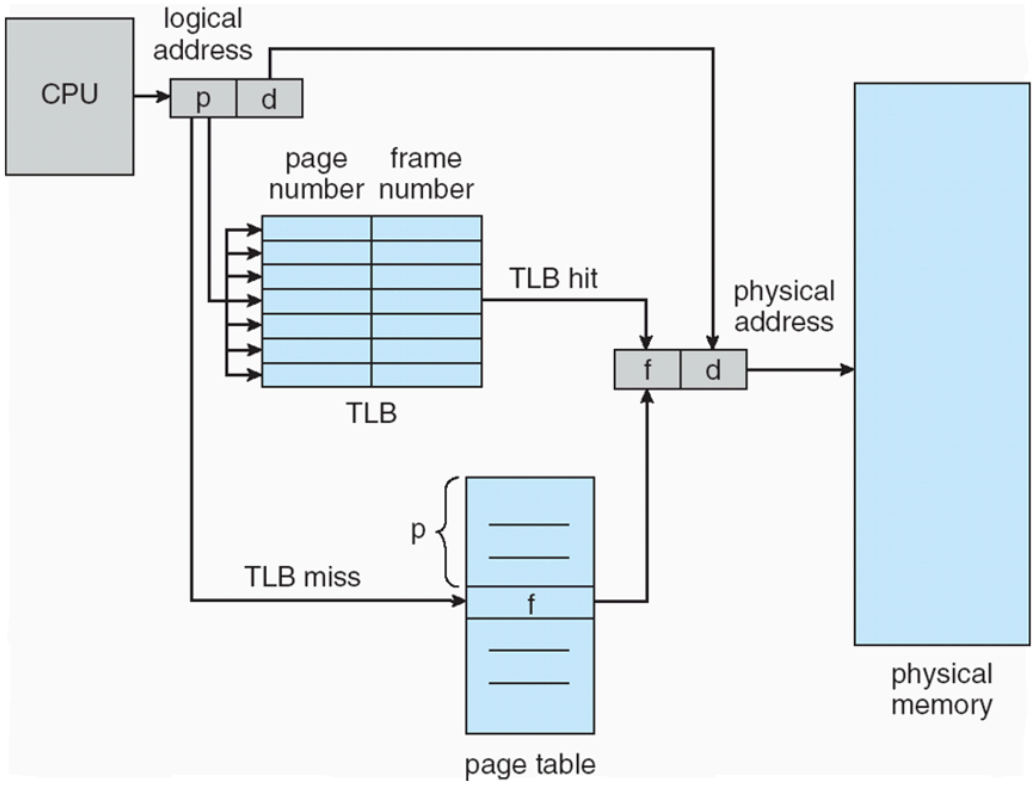
TLB를 활용한 Address translation 과정은 아래와 같다.

1. TLB look up에 원하는 page table가 있는지 확인한다.
2. TLB에 있다면 해당 frame을 out
3. TLB에 없다면, Main memory에서 해당 Page table을 찾고, TLB에 등록한다.

TLB는 context switch시 FLUSH되어야 한다.

- process마다 각각 Page table을 관리하고 있기 때문에 TLB도 달라져야 한다.

Example



- TLB를 보면, Page table 구조와 유사하게 정보를 저장 중인 것을 확인할 수가 있다.

Effective Access Time

- TLB가 있을 때, CPU가 Memory operation을 통해서 원하는 Inst, data를 가져오는 데 걸리는 평균 시간을 계산해보자.
- TLB 성능 추정에 사용할 수 있다.

- Associative lookup = α time unit
- memory access time = β
- Hit ratio = ε (percentage found in the associative memory)

- Effective Access Time (EAT)

$$\begin{aligned} &\text{<hit>} \qquad \text{<miss>} \\ \text{EAT} &= (\alpha + \beta)\,\varepsilon + (\alpha + 2\,\beta)(1 - \varepsilon) \\ &= \alpha + (2 - \varepsilon)\,\beta \end{aligned}$$

- TLB hit 시, Data / inst를 위한 memory access 한 번
  - TLB Miss 시, Data / inst를 위한 memory access 두 번
  - TLB hit rate가 클수록 EAT가 작다.
- 알파는 베타에 비해 월등히 작다.
  - 즉, TLB look up은 Memory access에 비해 훨씬 작다.
- Paging을 사용하지 않을 때와 비교해보자.
  - Paging을 사용하지 않을 경우에 EAT는 베타이다.
  - TLB Hit ratio = 1일 때는 (알파 + 베타) 이다.
    - 알파는 베타에 비해 월등히 작기에 무시 가능하다.
  - 따라서 TLB hit가 높으면 **external fragmentation 없이도 DRAM access 한 번 하는 속도**로 데이터를 가져올 수 있다.

따라서 우리의 목표는 **TLB HIT RATIO를 높이는 것이다.**

Memory protection

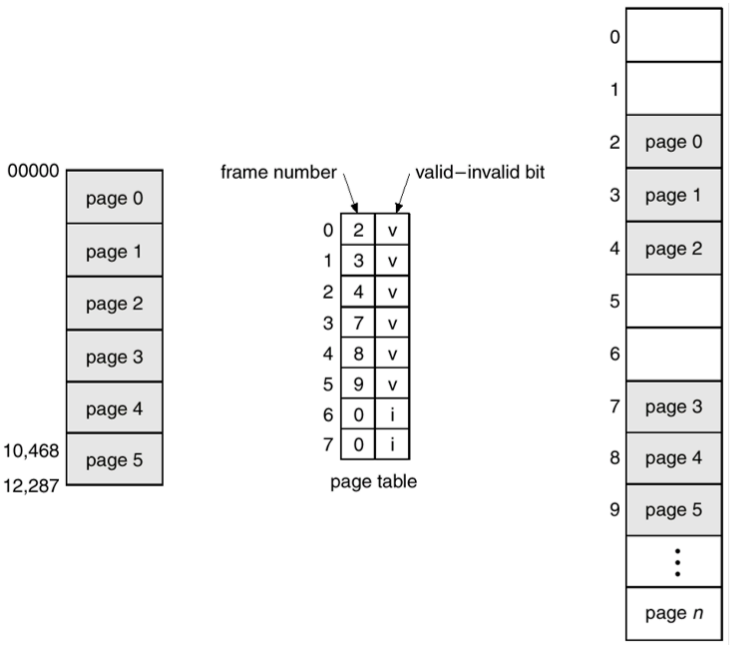
각 Frame에 Protection bit를 추가하여 구현할 수 있다.

Valid-invalid bit

각 PTE에 존재하고, 'bit = Valid'인 경우에만 해당 page에 접근 가능하다.

Example

- Page size = 2 KB
- Uses only addresses 0 to 10468
- 6 (=10469/2048) pages are allocated to the process
- Only 6 entries are used in the page table
- PTLR can be used to test the validity of the address instead of the valid-invalid bit



- 이 Process가 Logical address 10469 까지의 address만 사용이 허락되었다고 할 때, 6개의 page table entry가 있으면 된다.
- 하지만 이 page table은 8개의 enrty가 존재한다.
- 따라서 이 Process의 page table에서 **2개는 Invalid**하게 만들어야 한다.

Q) 왜 page table이 address space보다 크지??

- Address space를 0 ~ MAX라고 할 때, **Page table entry는 logical하게 가능한 page의 개수만큼 만들어져야 한다.**
- 특정 시점의 page에만 맞춰서 만든다면, stack과 heap이 변할 때마다, PTE가 부족하기 때문에 주소를 매번 갱신해야 한다.

이렇게 Valid-invalid bit를 따로 사용하는 방법도 있고, **PTLR**을 써서 PTLR에 저장된 값보다 작은 Entry num (index)까지만 유효한 entry로 지정하는 방법도 있다.

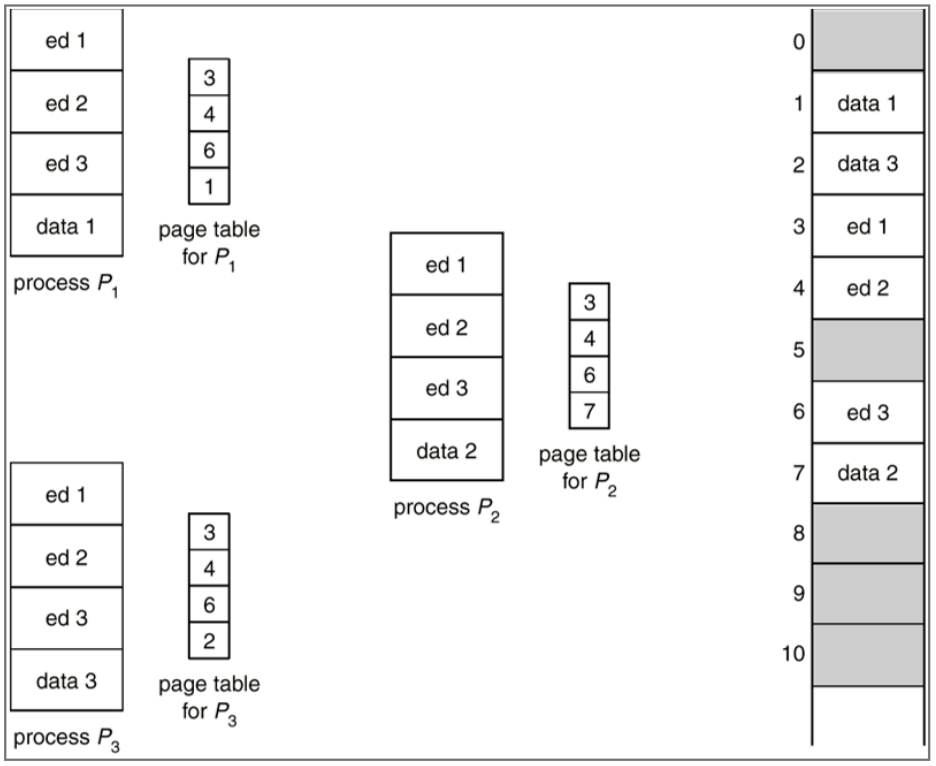
Shared pages

1. Shared code
   - Code가 **read-only고 re-entrant** 하다면 share이 가능하다.
   - 하나의 복사본
   - Continuous allocation인 경우에 text 영역만 따로 빼서 공유하는 것은 어렵다.
     - 따라서 Text section을 공유하면, Data section도 같이 공유하게 되는 경우가 생겨서 어렵다.
   - Jump 등은 logical address로 jump 하도록 함
     - Shared code는 Logical address 상에서 항상 같은 위치를 갖도록 한다.
     - 즉, 상대 주소가 동일하도록 한다.

2. Private code and data
   - 각 Process는 **Seperate copy**를 가짐
   - Private page는 logical address의 어느 공간에 있어도 상관 없다.

Example

Editor is shared (Editor consists of 3 pages -- ed1, ed2, ed3)



- P1이 ed1, 2, 3를 자신의 page table에 등록하고, memory에 올려놓음
- P2, P3가 page table을 업데이트 하려할 때, 공유되고 있는 ed1, ed2, ed3가 P1에 의해 DRAM의 어느 page 에 저장되는지 OS가 알고 있다.
- 따라서 P2, P3의 page table에는 기존에 알고 있던 page 주소을 그대로 사용한다.