9. Memory management 2

External fragmentation을 제거하기 위해 Paging을 사용하려면, Page table이 필요하다.

이때 page table의 구조에 따라서 page table 참조 방식이 달라진다.

기존 방식의 문제점

32-bit address and 4KB (= 2^12) page size를 가정하자.

- Page에서 4B는 Entry의 크기이고, 하나의 Page에는 2^10개의 page entry가 존재한다.

이 경우 32 - 12 = 20개의 bit가 남고, 따라서 한 Process는 **2^20개의 logical page**를 가질 수 있다.

- 백만 개의 page table entries
- Logical address는 어차피 순서대로 등록되므로, logical page 순서를 따로 관리할 필요 없이 Frame 정보만 기록하면 된다.

따라서 각 page table entry는 4 byte이므로, **전체 Page table의 크기는 4MB**이다.

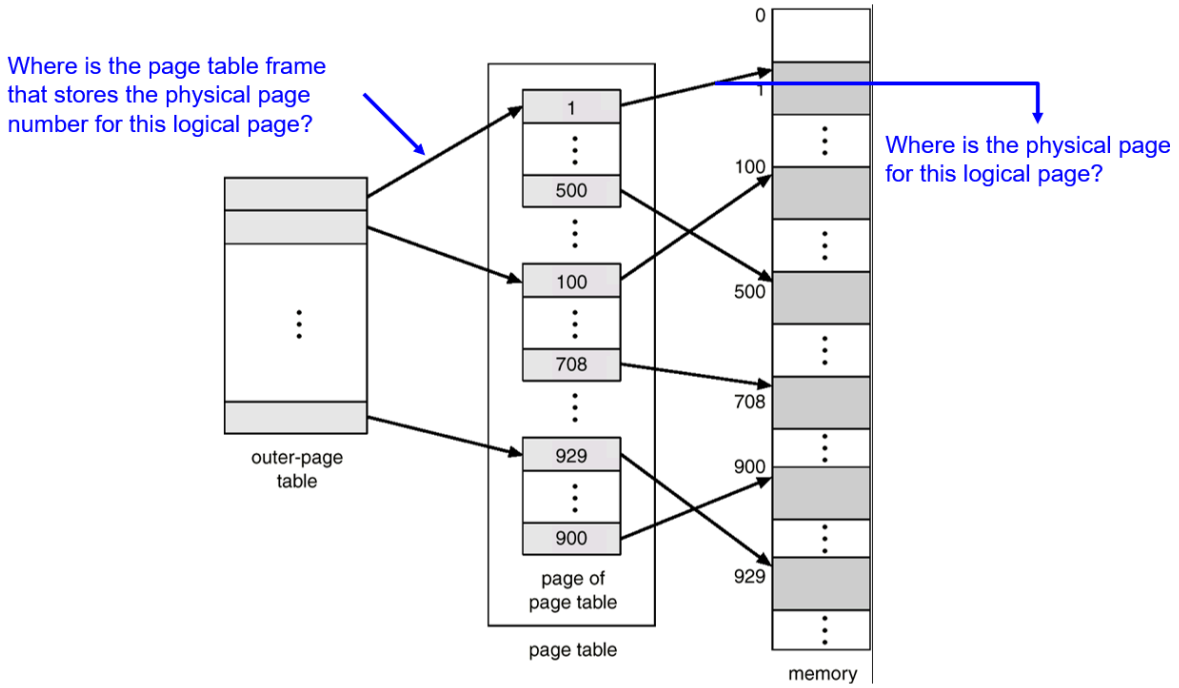
- Page table을 저장하기 위해서만, **각 Process 당 2^10 개의 Page가 필요**하다.
  - 이때, 2^10개의 page frame은 불연속적으로 저장되어도 되지만, 한 개의 frame 내부에서는 연속적으로 저장되어야 한다.
- 각 Page의 size가 2^12이기 때문이다.

Hierarchical paging

위의 문제를 해결해보자

1. **우선 전체 Page table은 항상 Disk에 저장하고, 자주 사용하는 Page table만 Memory로 올리자**
    - 이를 위해선 내가 원하는 Page table이 어느 Frame에 저장되어 있는지 알아야 한다.
2. Page table은 Page unit에 **on-damand 방식으로 메모리에 올라와야 한다.**
    - Page table을 가리키는 또 다른 Page table이 필요하다.

Two-Level page-Table Scheme



- Outer page table 이 새롭게 필요하다.
    - 각 Page table이 저장된 frame이 어디에 있는지 저장한다.
- CPU는 **Logical address만을 이용**하여 Outer-page table, Page table, Memory 모두에 접근이 가능하게 된다.

정리)

- Memory에는 한 process 당 page가 2^20개 존재할 수 있다.
- 한 Page table은 2^10개의 page entry를 가지므로, 2^20개의 page frame을 모두 가리키기 위해선 page table이 2^10개가 필요하다.

- Outer-page table은 page table 2^10개를 가리켜야 하는데, 이는 하나의 page table로 가능하다.

Example

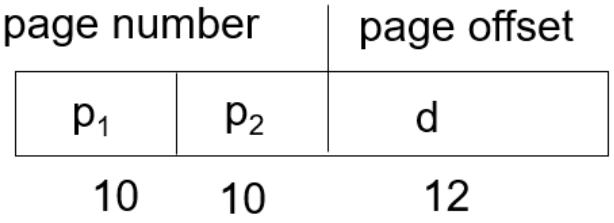
Logical address (32-bit machine with 4K page size)

- `Page number` : 20 bit
- `Page offset` : 12 bit

위 경우에는 page table을 가리키는 page table이 추가로 필요하다.

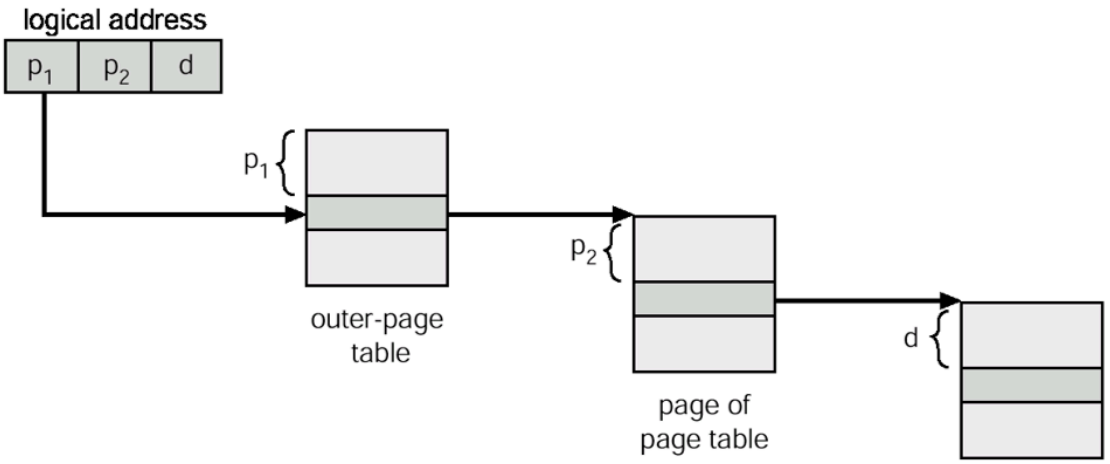
- 상위 `10-bit page numer` : Outer-page에 대한 index
- 그다음 `10-bit page offset` : Inner page table에서 몇 번째 Entry인지 판단하는 Offset

Thus, a logical address is as follows:



- 위의 예시에 따르면 Logical address는 이렇게 나누어 진다.
- p1은 Outer table에 해당하는 index, p2는 inner page table에 해당하는 index이다.

이를 이용한 Address-Translation scheme은 아래와 같다.



<Forward-mapped page table>

- Outer page table은 **Inner page table의 시작 주소를 제공**해준다.
- 각 Outer page table, Inner page table, memory 모두 2^12의 size를 갖지만, memory의 offset만 2^12인 이유는 **Page table은 Entry에 대한 index로** 접근하지만, **Memory는 byte 기준**으로 접근하기 때문이다.

이 과정은 전부 Page를 2의 지수승으로 나누었기 때문에 가능한 것이다.

이러한 Multi-level paging **의 성능에 대해 알아보자**

- Multi-level에 경우, 한 page table이 저장하고 있는 정보가 Data block의 주소가 아니라, Next level page table의 주소일 수 있다.
- 각 Page table은 메모리에서 연속적으로 저장된 것이 아니기 때문에, **각 Level마다 여러 번의 메모리 접근이 필요할 수 있다.**
- 현재 위의 예시인 Two-level의 경우, Outer, Inner, Memory 총 3번의 Memoty access를 하게 된다.

Two-level의 경우, **기존에 비해 한 번의 추가적인 Memory access가 필요하지만, TLB가 이를 감안하고도 성능을 향상 시키는 데 도움을 준다.**

- 4-level의 경우 hit rate가 98%

EAT = 0.98 x 120 + 0.02 x 520 = 128 nanoseconds,
which is only a 28 percent slowdown in memory access time (assuming TLB access time = 20 ns, memory access time = 100 ns)

- 이 경우, 4-level을 가정했음에도 불구하고, TLB hit이 발생한다면, 주소 변환 없이 직접 메모리에 접근하는 경우보다 28%밖에 느리지 않음을 확인할 수 있다.
- **성능 저하가 TLB 덕에 심하지 않다면, 우리는 용량 문제를 해결하기 위해 Multi-level을 사용할 수 있다.**

32 bit address 이상의 경우는 어떨까??

64 bit address의 경우를 생각해보자.

- 이 경우에는 6-level paging이 필요하다.
- 최근 64 bit address의 경우에는 주소 계산에 48 bit만 사용하고, 4-level paging을 사용한다.
- **Logical address가 32bit보다 크면, Outer page table을 하나의 page table에 저장하지 못 할 수 있다.**
  - 그렇다면 page table access가 증가하며, Memory 접근 횟수가 증가한다.
  - Two-level 이상의 paging 기법을 사용하면 해결되나, 이는 overhead가 발생한다.

상식)

32bit는 4GB DRAM까지 커버할 수 있다.

- 4GB DRAM이라 하더라도, 0.8GB 정도는 사용하지 않기 때문이다.
- 8GB DRAM은 커버할 수 없다
  - 4GB 이상의 DRAM을 사용하기 위해서는 64 bit cpu/os 를 사용해야 한다.

Hashed Page Table

결국 **Page table은 Logical address로 Physical address를 찾는 문제**이다.

- Hierarchy page table은 사실상 Tree 자료구조 이다.

그렇다면 Search 문제에서 가장 효율적인 **Hashing을 사용**할 수는 없을까?

- Best case에는 O(1)에 Search할 수 있다.
- Hash table == Page table

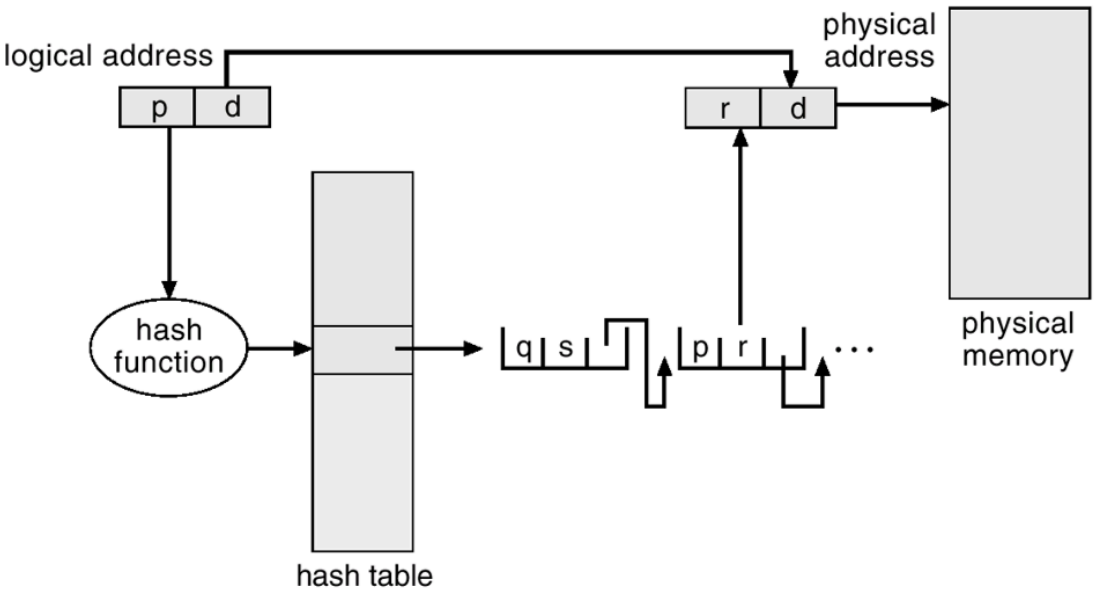
Address space가 32 bit보다 큰 경우에 성능이 좋다.

- 32 bit 까지는 Two-level page table을 사용하는 것이 성능이 더 좋기 때문이다.

Virual page number (Logical address의 상위 p bit)를 key로 사용하여, Hashing 하자.

Collision **이 발생하는 경우에는 Chaining 기법을 활용한다.**

- 같은 index에 Hashing 되는 다른 element는 chain처럼 연결해 놓는다.



1. **P bit를 hash function의 input으로 사용하여, hash table을 look**한다.

2. 이때, Collision이 발생하면, Chain에서 맞는 것을 찾아 반환한다.
    - 이때, O(n)의 시간이 소요되고, 이는 Hierarchy page table에 비해 손해이다.
    - **Hash table에서 collision을 최소화해야 한다.**
3. Collision이 발생하지 않으면 바로 알맞는 것을 return한다.
4. 이후 하위 d bit (page offset)을 뒤에 더하여서 physical memory에 접근한다.

성능 판단 기준은 `Hash funtion` **의 종류,** `Hash table` **의 size**에 따라서 달라진다.

Inverted Page Table

기존 문제점: Page table이 커야할 필요가 있는가?

- 기존 page table size는 page의 개수에 비례했다.
- Logical page 하나 당 하나의 Page table entry가 필요했다.
- **그러나 실제로는 DRAM에 한 process의 모든 page가 올라와 있지 않다!**

Process가 자신의 모든 Logical page를 DRAM에 올린다면, process 당 4GB의 공간이 필요하다.

- 그러나, 우리는 하나의 process만을 실행하는 것도 아닌데 DRAM은 보통 4GB 뿐이다.
- **즉, 각 process의 모든 page가 DRAM에 올라가 있는 경우는 거의 없다!**

기존 우리는 Process의 address space를 너무 크게 잡아서 실제 사용하지 않는 공간이 너무 많다.

- 따라서 사용하지 않는 부분은 DRAM에 올리지 않더라도, Page table에서 매핑 정보는 유지하되, Invalid하도록 만들어야 한다.

이를 해결하기 위해 **Physical page를 Logical Page로 Mapping 하자.**

- Physical page의 개수는 DRAM의 용량에 의해 정해진다.

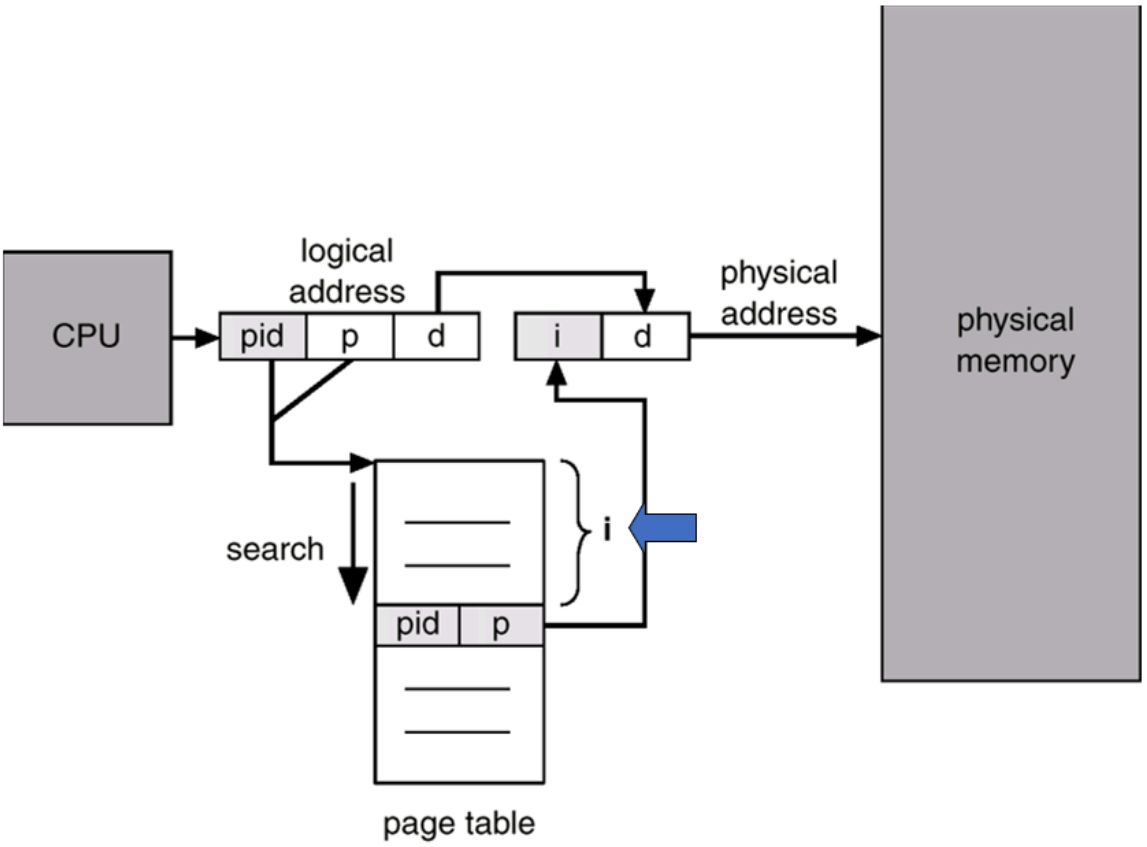
- DRAM이 4GB라면, Page 하나는 4KB이기 때문에 2^20개의 Physical page가 존재할 수 있다.
- 그렇다면, DRAM의 각 physical page마다 각 page에 어떤 process의 어떤 page가 저장되어져 있다는 정보를 제공하면 어떠할까?
    - 각 Process마다 2^20개의 page에 대한 Mapping 정보가 필요해 Page table이 커질 필요 없이 **전체 Process에 대해 2^20개의 Page table에 대한 정보만 관리하면 된다.**
    - Page table에는 Invalid 따로 없이 **실제 DRAM에서 사용 중인 Page에 대한 정보**만 들어가게 된다.
    - 각 Process에 대해 여러 개의 Page table이 존재하는 것이 아니라, **하나의 Page table만 존재**해도 된다.

결론: 하나의 Physical page frame 마다 하나의 page table entry가 존재하게 하자.

- 각 Page table entry는 `process id`를 관리해야 한다.
- **One system-wide page table**
    - Enrty 개수 = Physical page frame의 개수
    - 모든 Process가 이 Page table을 공유해야 한다.

Inverted Page Table Architecture

`Fully associative` 방식 이용



- 먼저 Page table에서 **pid + p** 에 맞는 enrty를 찾는다.
- Matching 되는 것이 있다면, 해당 entry에는 해당 process의 p번째 page가 저장되어 있을 것이다.
- Matching 되었을 때, Page table에서의 index i를 physical page에 접근하는데 사용한다.
    - 정확하게 DRAM의 page frame과 매칭되어 있기 때문에 가능하다.

문제점)

1. 이전처럼 Page table에서 Logical address를 이용하여 indexing하는 것이 아니라, **직접 Page table을 돌며 Matching 되는 것을 찾아야 한다.**
    - DRAM을 일일히 search 해야 하기 때문에 느리다.
    - Memory에 올라가는 자료 구조이기 때문에, TLB와 같은 특별한 하드웨어를 사용할 수 없다.
    - Pid, p를 이용하여 Hashing하는 방법이 하나의 해결책이 될 수 있다.

2. **Page sharing이 불가능하다.**
    - Page sharing을 위해서는 같은 frame index에 두 개의 pid가 적히도록 해야 한다.
    - 사전에 몇 개의 process가 하나의 page frame을 공유할 지 모른다.
    - 결국 Page sharing을 지원하려면, **Page table의 구현 자체가 어려워진다.**

해결법)

1. pid, p를 key로 하는 **Hash table 사용**
    - Hash table을 lookup하는데 한 번의 추가적인 메모리 접근이 필요하긴 하다.
2. **Inverted page table과 별개로 TLB 사용**
    - TLB miss가 발생한 경우에만 Inverted page table을 확인한다.

Segmentation

Paging의 문제점

Paging은 DRAM과 Logical address space 모두 page 단위로 자른다.

만약 이렇게 **의미 단위가 아니라 크기 단위만 가지고 자르게 되면 잃는 부분이 생길 수 있다.**

크기 단위로 다르게 되면 코드나 배열의 일부분은 첫 번째 Page에, 다른 부분은 다른 Page에 존재하게 될 수도 있다.

- 이 경우, 코드나 배열을 공유할 때, Page 전체를 보고 공유해야 하는데, Page의 일부에는 공유 가능한 것 다른 부분에는 공유 불가능 한 것이 저장되어져 있어 공유가 불가능한 상황이 생길 수 있다.
- 한 코드를 공유할 때에도 여러 Page를 공유해야 되서 귀찮다.

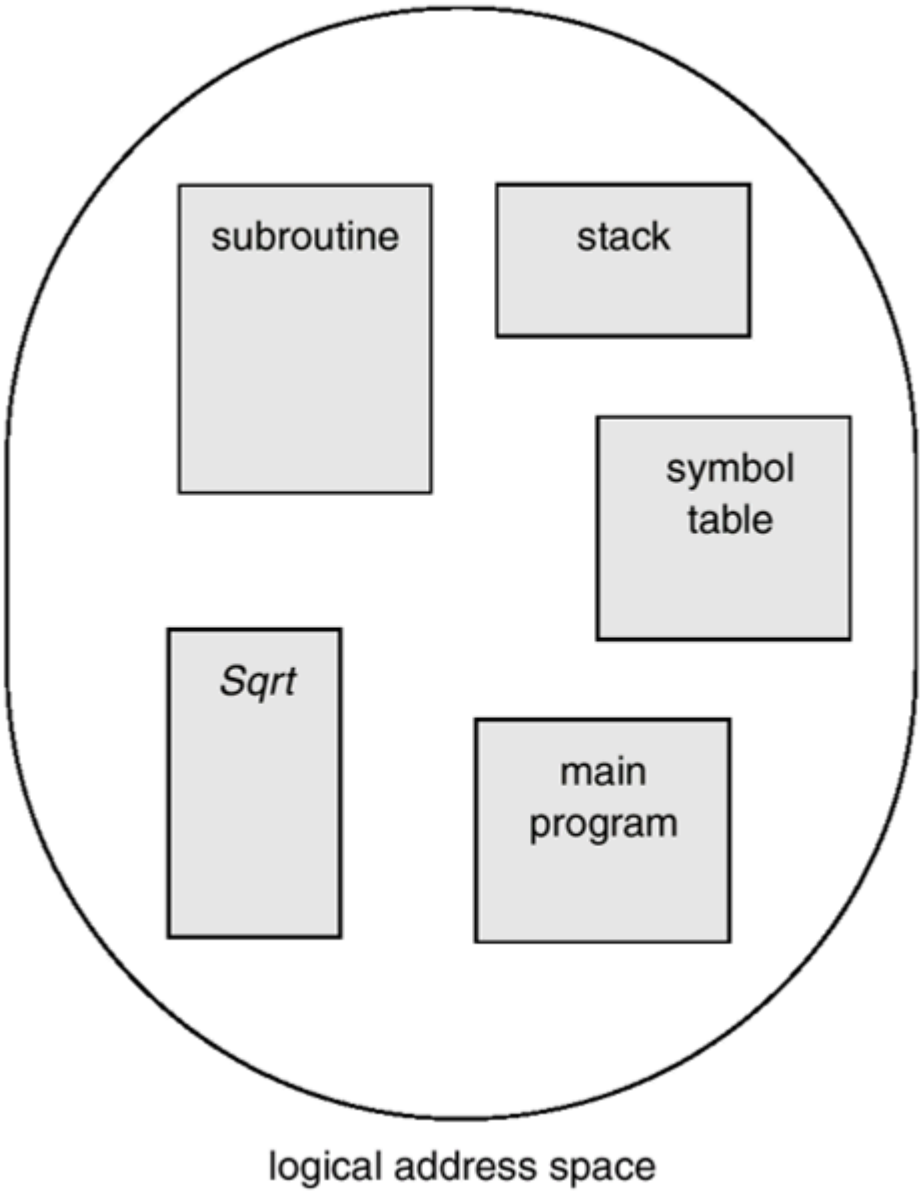
이를 해결하기 위해 `Segmentation` 을 사용하자

`Segment` 은 **Logical unit**이다.

- main(), global variable, stack, symbol table 등이 segment가 된다.
- 각 Segment는 각각의 의미 단위가 다르기 때문에, 가변 길이를 갖는다.

Segmentation 은 **이러한 Segment의 의미 단위를 유지하며 나누는 방법**이다.

- 각 Segment는 Memory에 Continous하게 저장될 수 있도록 한다.
- Segment간에는 연속일 필요는 없다.



- **User's view of a program**

- 각 사각형이 Segment이다.
- 이 경우, Logical address space는 의미 단위들의 집합이다.

Sementation Architecture

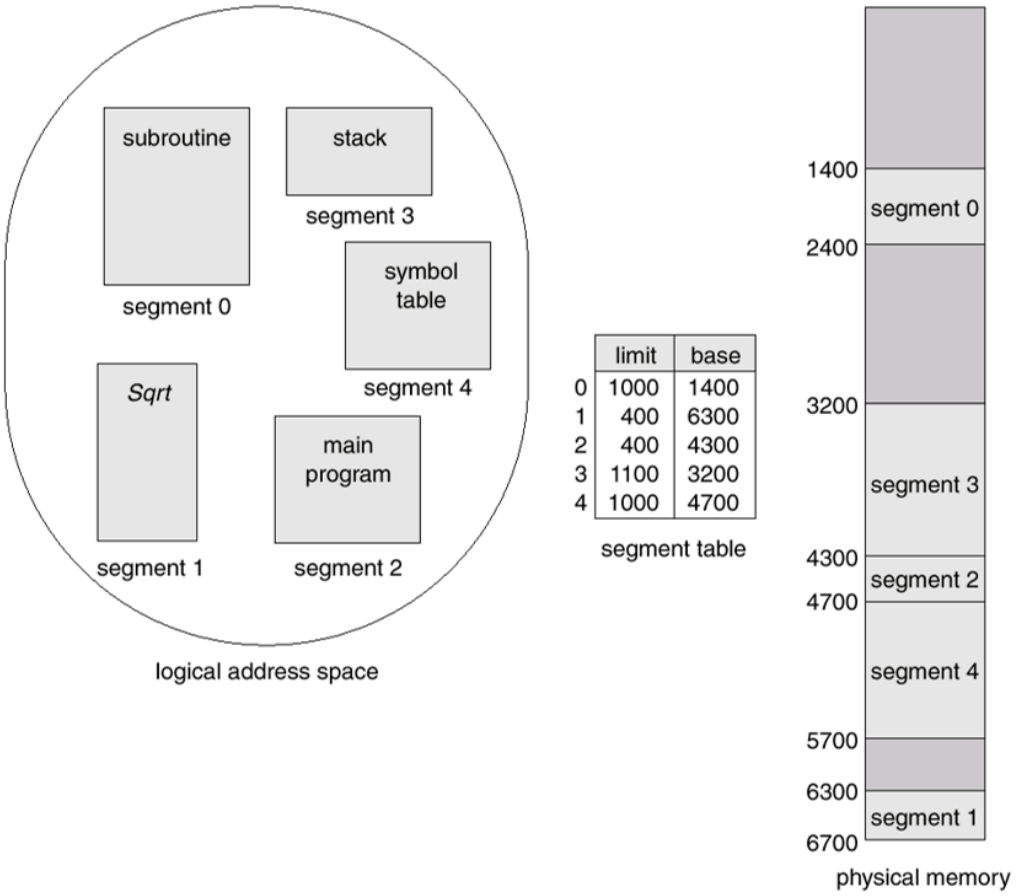
- Logical address structure
  - < segment-number, offset >

- Segment Table
  - Two dimentional logical address를 physical address로 바꾸어 준다.
  - 각 Table은 Segment에 대한 base와 limit을 갖는다.
  - base: Segment가 저장된 memory의 시작 주소
  - Limit: Segment의 길이
  - 이 자료구조는 OS가 관리한다.

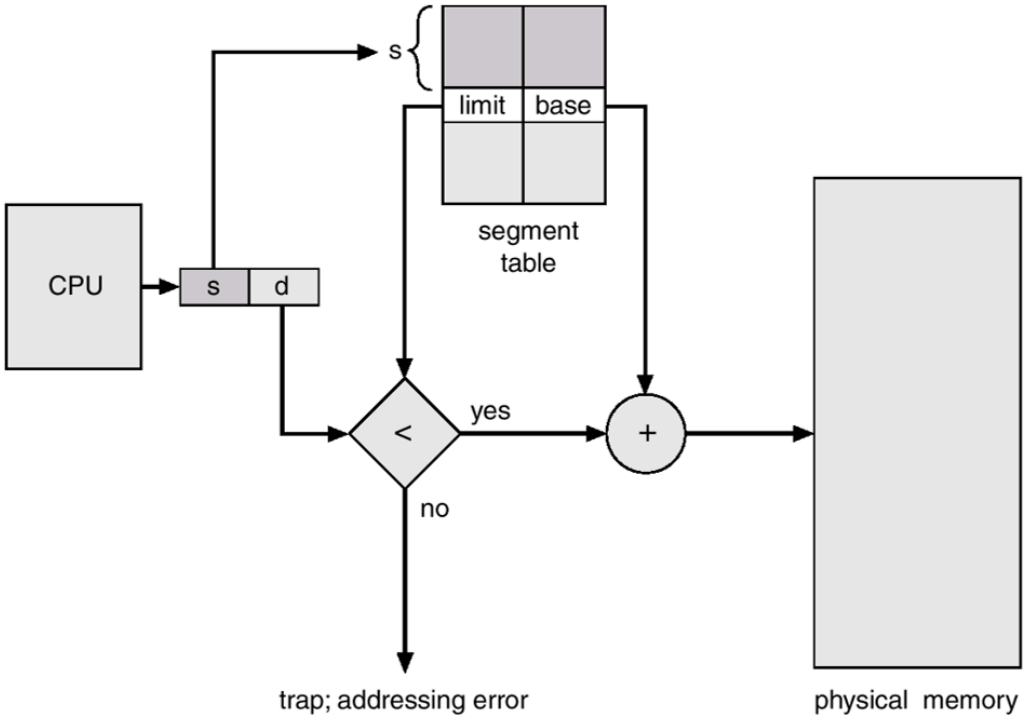
- Segment table base register (STBR)
  - Segment table의 시작 주소를 가리킨다.

- Segment table length register (STLR)
  - Segment table의 길이를 가리킨다.
  - Segment table의 길이 = Segment의 개수
  - Segment num인 s가 (s < STLR) 인 경우에만 정상 동작해야 한다.

Example



- 이 그림과 같이 Segment3, 2, 4가 꼭 연속적으로 저장될 필요는 없다.



1. s (Segment number)이 STLR의 값보다 작은지 확인한다
2. STBR에 저장된 값에 s를 더하여 limit과 Base를 확인한다.
3. (2)에서 얻은 Limit이 d보다 큰지 확인한다.
4. (3) 과정이 통과라면 Base + d로 memory에 접근한다.

Segmentation의 특징

Protection

- 각 Segment table은 **valid / invalid bit**을 갖는다.
  - illegal segment인지 확인한다.
  - OS가 실행시킨 process의 segment를 미리 알 방법은 없다.
  - 따라서 미리 **크기를 어느 정도 넉넉하게 지정해놓고, Invalid bit을 활용**한다.
- **각 Segment는 R / W / X bits를 갖는다.**
  - Paging은 의미 단위가 아니기 때문에 불가능 했던 행동
  - EX) Code segment = Read - Only

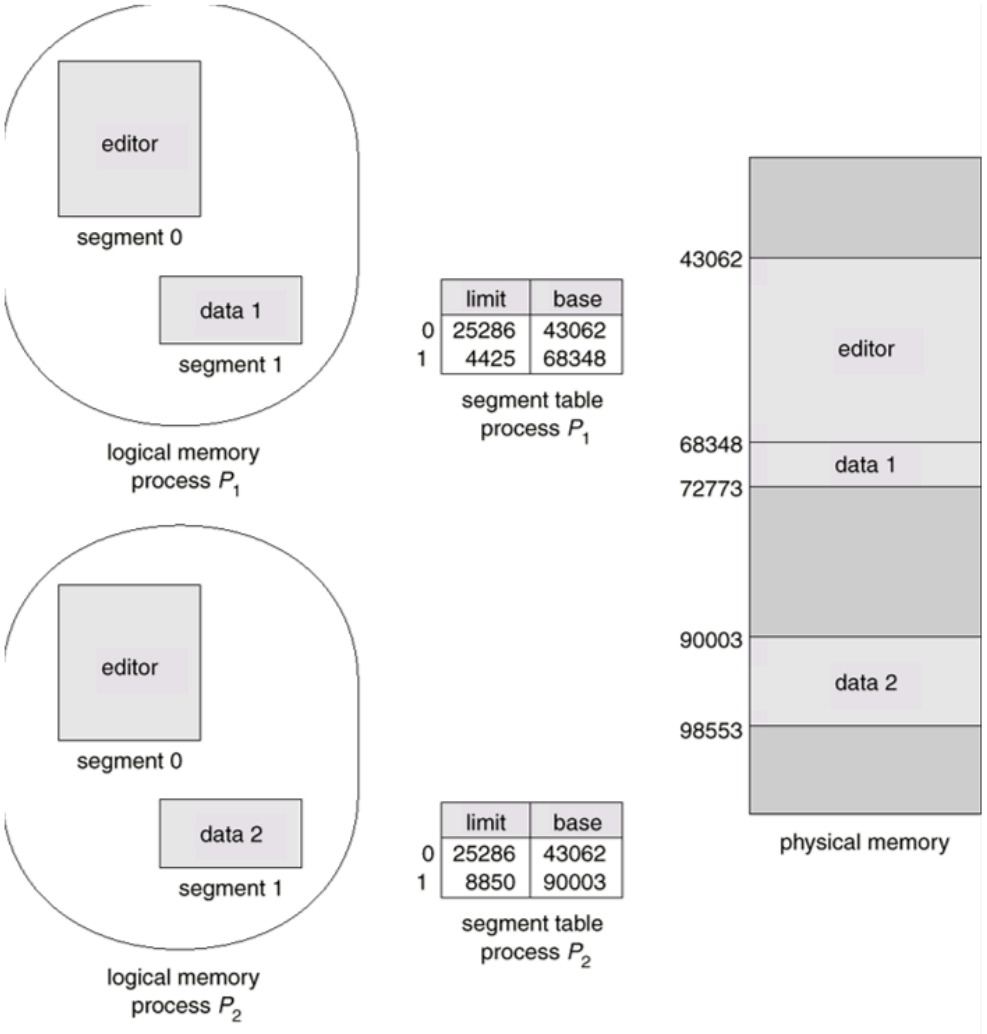
Sharing

- Segment level에서 sharing이 가능해져 Paging보다 강점이 많다.
- 각 Segment가 R / W / X의 protection bit를 가져 권한 부여가 가능하다.
- 여러 Process가 하나의 Segment를 공유하고자 하는 경우에는 Segment number가 같도록 해야한다.

Allocation

- Segment의 길이가 일정하지 않기 때문에, Dynamic storage-allocation 문제가 발생한다.
  - 이런 경우에는 Best fit / first fit 중에 한 방법 이용
- **External fragmentation 발생 가능하지만, Internal Fragmentation은 발생하지 않는다.**
  - 남은 메모리 공간보다, 할당해야 하는 Segment의 크기가 큰 경우 External fragmentation 문제가 발생!

Example of segment sharing



- **각 Process의 segment table에서 공유 하는 부분에 대해서는 Segment number가 같아야 한다.**
- 이를 Segment 0 을 통해 확인할 수 있다.

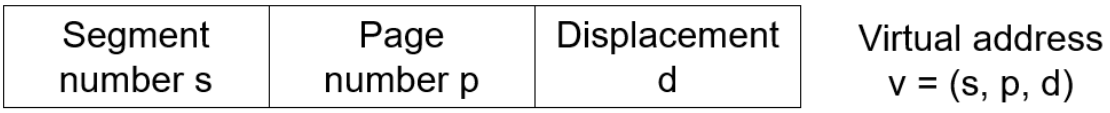
Segment with paging

문제점)

Segment만 사용 시, External fragmentation이 발생한다.

이를 해결하기 위해, **DRAM은 Page frame 단위로 자르고, Logical address는 우선 Segment 단위로 자르고 Segment 내부적으로는 page 단위로 자르는 방법**을 사용해보자.

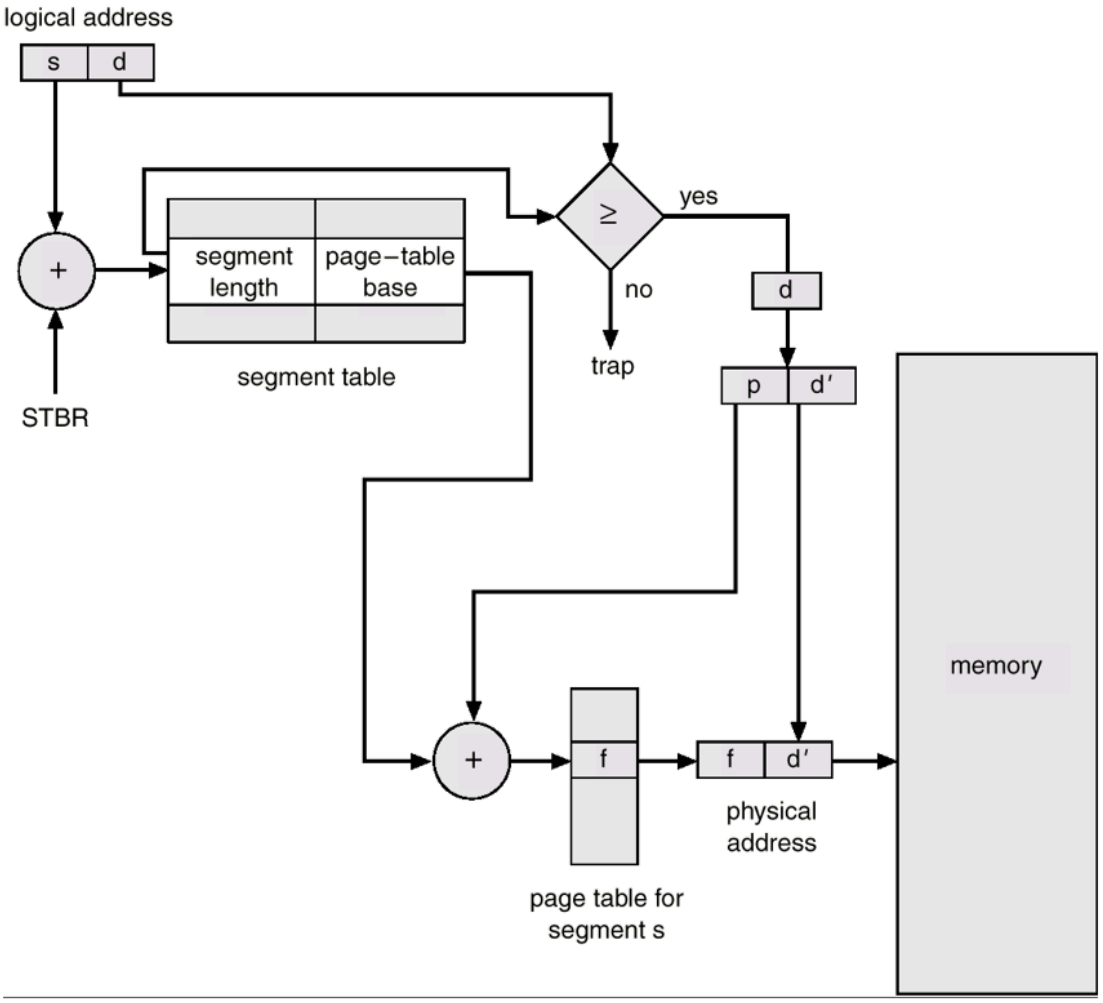
위 방법을 위해선, Logical address가 살짝 달라져야 한다.



(Virtual address format in a paged and segmented system)

- `Segment number`는 동일하다.
- `Segment offset` 은 **Page number와 displacement로 나뉘어 진다.**
- Segment table은 기존처럼 Segment의 Base address를 저장하는 것이 아니라, **이 Segment에 대한 Page table의 Base address를 저장하도록** 한다.
- **즉, 몇 번째 Segment의 몇 번째 Page의 몇 번째 Byte**

Address translation



1. s가 STLR의 값보다 작은지 확인한다.
2. s + STBR으로 Segment table에 접근한다.
3. d가 Segment table의 Segment length보다 작다면 이동한다.
4. d를 p와 d′으로 나눈다.
5. p + (2)에서 확인한 Page-table base로 Page table을 확인한다.
6. (5)에서 얻은 f (memory 상에서의 Page시작 주소) + d′으로 memory에 접근한다.

장점)

- 결국 memory는 page 단위로 조작되어, External fragmentation이 발생하지 않는다.

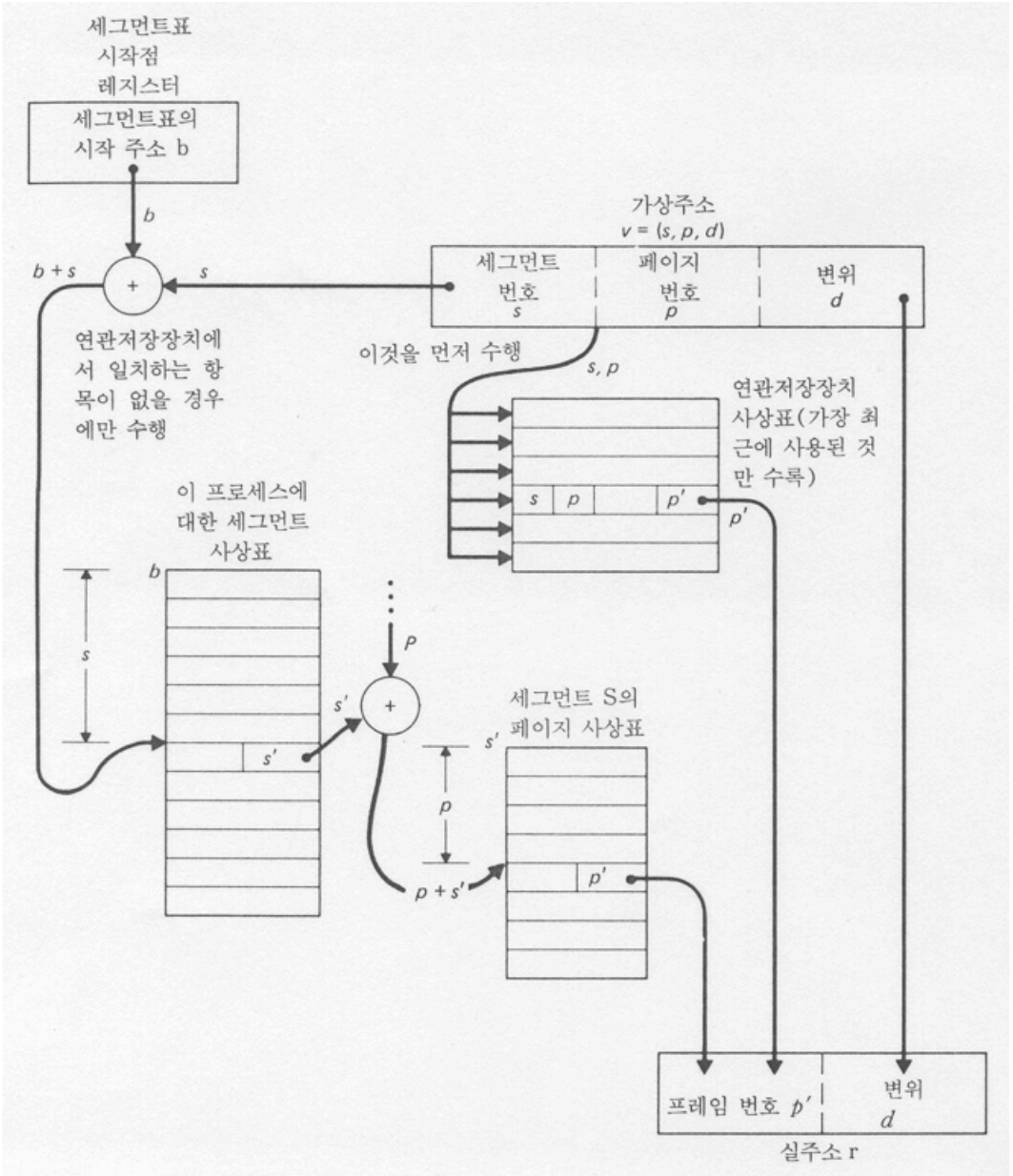
단점)

- Internal fragmentation이 발생하게 된다.

결국, 메모리 관리는 Paging과 동일하게 된다. 하지만 Segment Table을 사용하기 때문에, Segment 단위로의 Sharing이나 접근 방법은 유지 된다.

- **Segment 의미 단위는 유지하고, 공간 효율성은 Paging과 비슷하게 만든다.**

Segmentation with paging (TLB 추가 버전)



- Segment number로 TLB를 확인한다.
- TLB hit의 경우 Page table entry를 바로 확인할 수 있다.
- OS 종류마다 실제로 구현되는 방법이 조금씩은 다르나, 기본 개념은 이 예시와 같다.