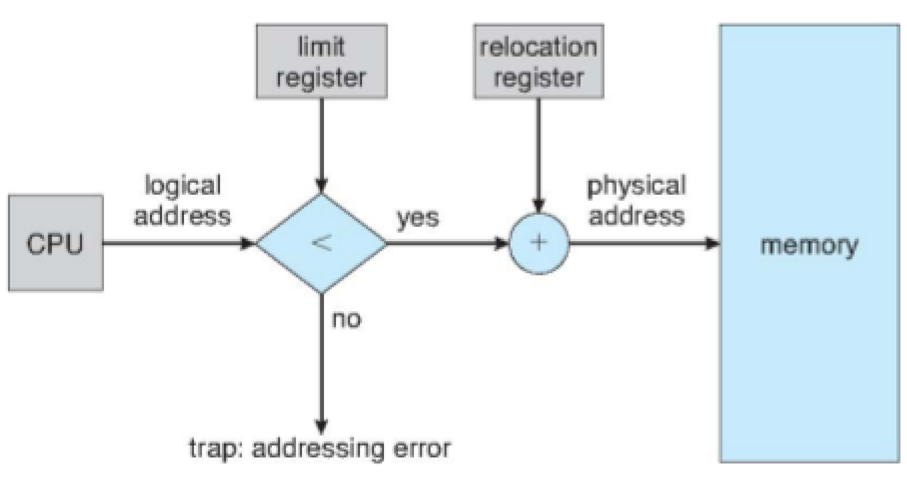
Continous Memory Allocation

Relocation registe의 역할

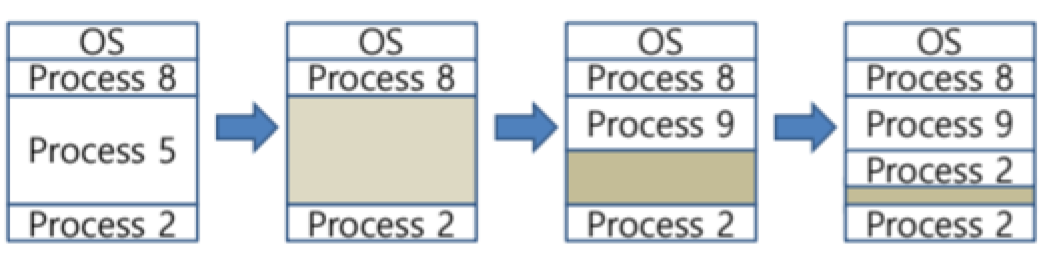
- user 프로세스들이 서로 영향을 주지 않도록 protect
- user 프로세스가 OS의 code나 data를 건들지 않도록 protect



- base regsiter가 physical address의 시작 부분을 가짐
- limit register가 logical address의 범위 값을 가짐
- MMU가 logical address를 dynamic하게 mapping
- 덕분에 커널 코드도 필요할 때만 적재되고 사라짐을 반복
  - kernel code being transient (일시성)

Multiple-partition allocation

- multiprogramming의 degree는 파티션의 개수에 따라 제한됨



- 효율을 위해 partition의 size는 variable
- 프로세스 도착 시 수용해줄 수 있는 hole에 할당
- OS는 free partitions과 allocated partitions에 대한 정보를 유지해야 함

Hole

- 사용가능한 메모리 블록
- 다양한 크기의 hole은 메모리 여기저기에 존재

free hole에 크기 n을 할당하는 방법?

- free hole list에서 적당한 hole을 찾는 방법?
- first fit
  - 그냥 n보다 큰 hole을 찾자마자 할당
- best fit
  - n보다 큰 hole 중 가장 작은 hole에 할당
  - size 기준으로 정렬 시키지 않는 이상 list 전부를 탐색해야
  - 근데 이 방법이 제일 hole을 적게 남기긴 함
- worst fit
  - 가장 큰 홀에 할당
  - best fit과 마찬가지로 list 전부 탐색해야
  - hole이 가장 많이 남게 됨 (근데 왜 함??)

Fragmentation

- external fragmentation
  - 전체적으로 request를 satisfy할 공간은 충분하나 그 공간이 continous하지 않음.
- internal fragmentation
  - 할당된 메모리가 요청한 메모리보다 살짝 큼
  - 남는 공간은 외부에서 쓰지 못하므로 낭비

External fragmentation을 줄이는 방법?

- compaction
  - free memory를 하나의 large 블록으로 합쳐지도록 메모리 contents를 shuffle
  - execution-time address binding같은 relocation이 dynamic할 때만 가능
- logical adress를 연속적이지 않아도 쓸 수 있게 만들기
  - 빈 공간 아무데나 프로세스를 할당시킬 수 있음
  - paging & Segmentation

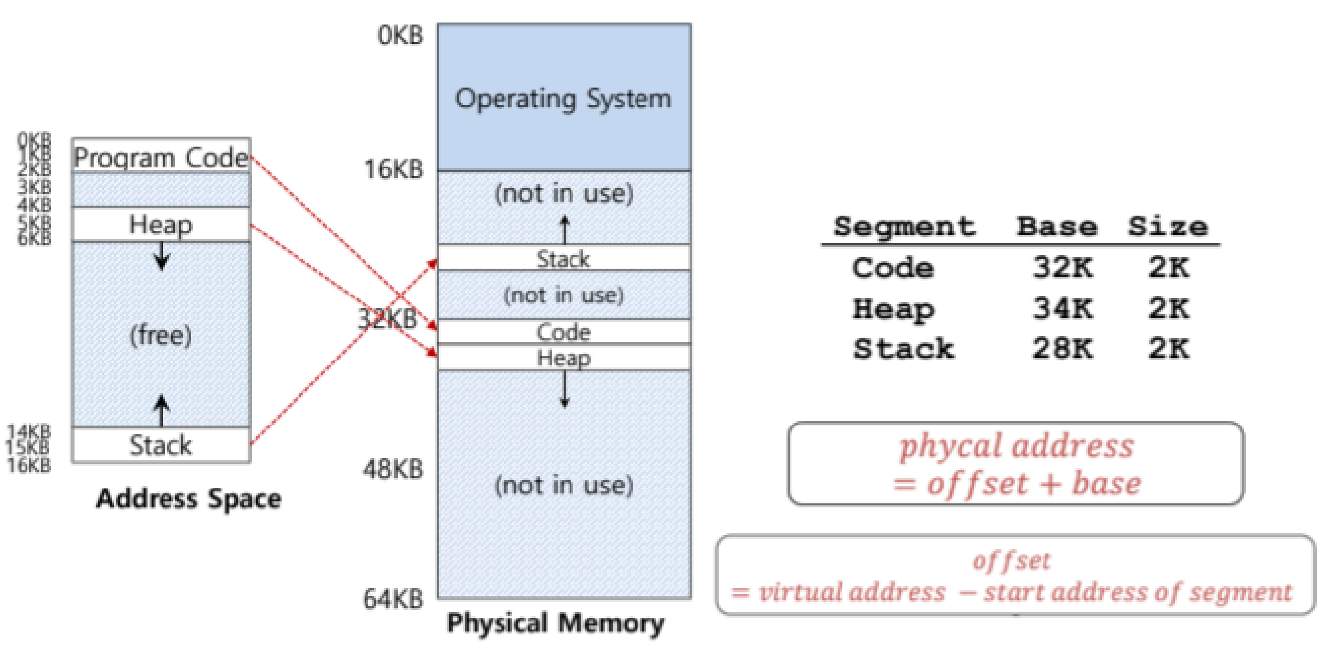
Inefficiency of the Base and Bound Approach

- base and bound는 각 프로세스 마다 base와 bound가 존재
- base and bound 방식은 base와 bound가 하나여야 하므로 연속된 큰 공간을 요구
- fragment를 활용하기 힘듦
  => Segmentation!!!

Segmentation

- address space를 logical segments들로 divide
  - segment는 특정 길이의 연속적인 address space의 일부
  - 각각의 segment는 address space 내 logical entity에 상응
  - logically-different한 segment가 있음
    - code, stack, heap
- segment 특징
  - segment끼리 physical memory의 다른 부분에 배치 가능
  - 각자 grow or shrink
  - 각 세그먼트 마다 base and bounds 가 존재

Physical memory에 segment 배치



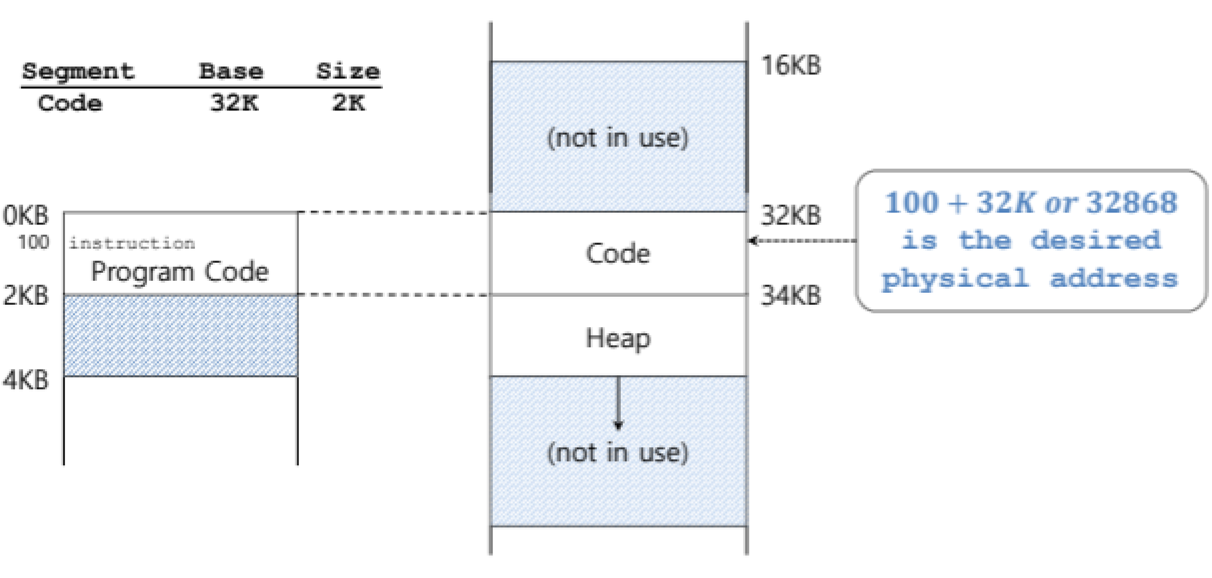
Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

$$phycal\ address = offset + base$$

$$offset = virtual\ address - start\ address\ of\ segment$$

- code, heap, stack을 physical memory의 서로 다른 장소에 배치
- 근데 이러면 stack이나 heap이 커져서 다른 프로세스를 침범하지 않나?
  - 미리 크게 할당? 근데 이렇게 하는 게 프로세스 전체를 놓는 것 보단 낫긴 할 듯

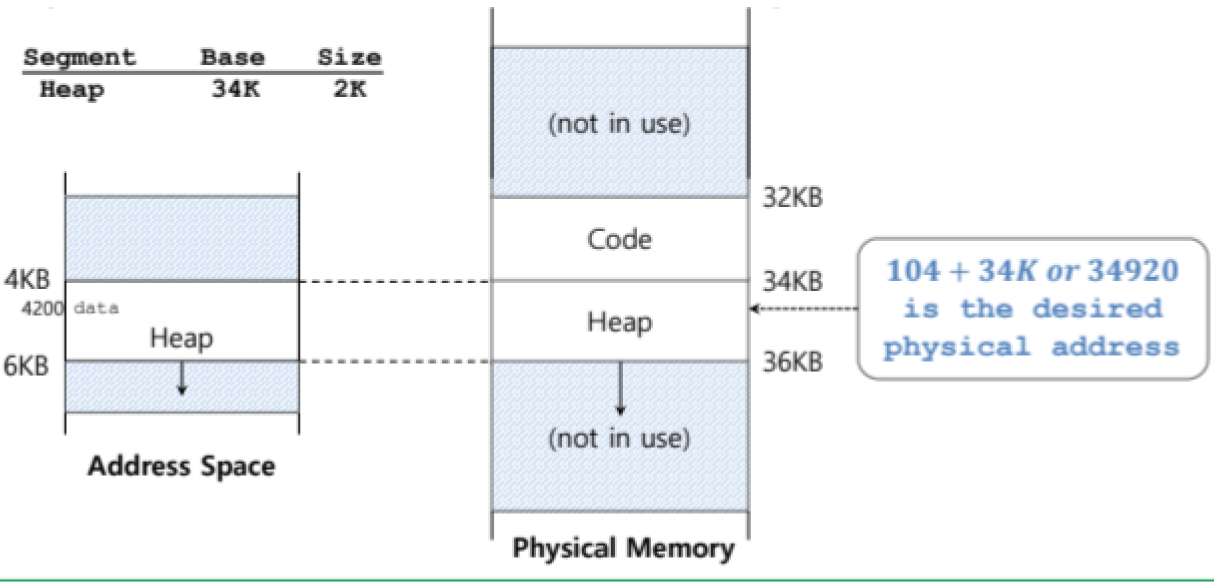
Address Translation on Segmentation



Segment	Base	Size
Code	32K	2K

$100 + 32K$ or 32868 is the desired physical address

- instruction이 address space에서 100에 위치했다면 physical address는 32768 + 100
  - 주의 : 여기서 100은 virtual address가 아닌 segmentation의 offset으로 봐야 함

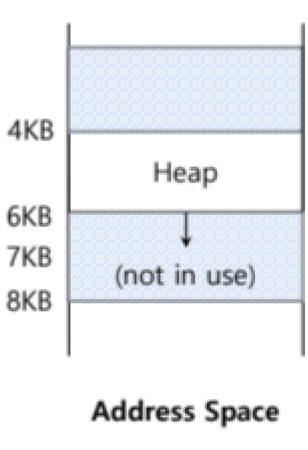


Segment	Base	Size
Heap	34K	2K

$104 + 34K$ or 34920 is the desired physical address

- data의 virtual address는 4200이지만 offset이 중요. offset은 104 (4200 - 4096)
- 그래서 physical address는 34K + 104

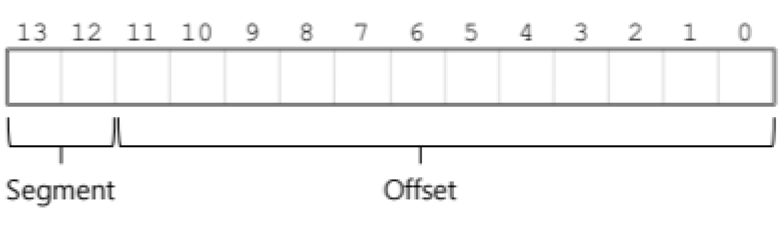
Segmentation Fault (Violation)



- 만약 7KB같은 heap 주소를 넘는 참조가 발생한다면 OS는 segmentation fault 발생시킴
- hardware는 주소가 out of bounds인지 탐지

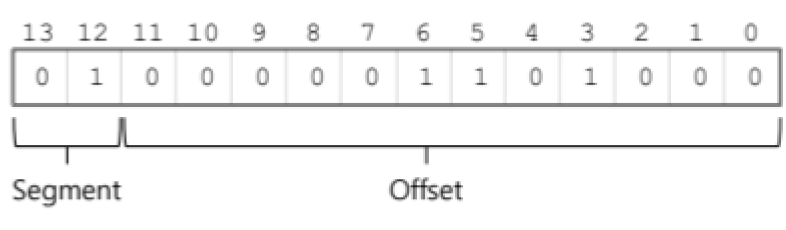
주소가 어떤 Segment를 참조하는지 아는 방법?

- address space에서 virtual address의 bit에 근거하여 segment를 생성
- top few bits가 segment를 가르킴
- 나중에 비트마스킹으로 segement bit를 추출하는 코드 나옴



- 만약 virtual address가 4200일 경우
- 4200은 2진수로 01000001101000

Segment	bits
Code	00
Heap	01
-	10
Stack	11

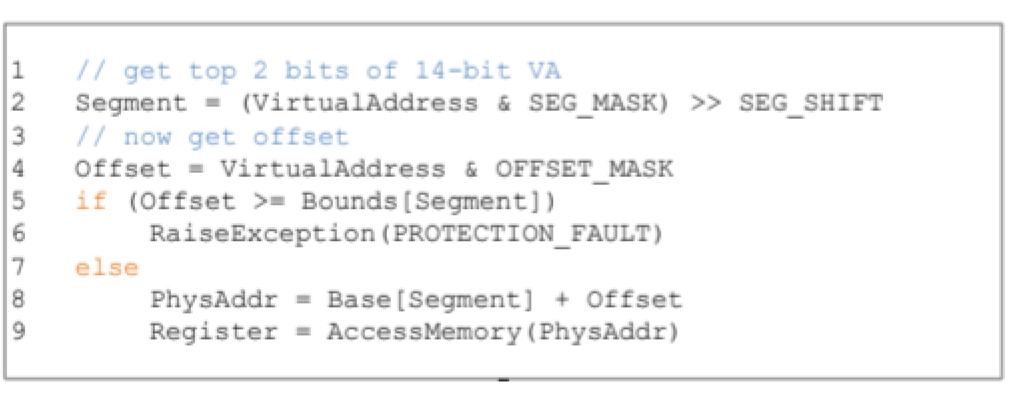


- 근데 이러면 segment의 크기는 4개로 고정되는 거 아님??
  - 맞음. segment의 크기를 늘리고 싶으면 offset bit 수를 늘리면 됨
  - segment 수를 늘리고 싶으면 segment bit 수를 늘리면 될 듯?

비트마스킹으로 segment랑 offset 추출

- `SEG_MASK = 0x3000(11000000000000)`
- `SEG_SHIFT = 12`
- `OFFSET_MASK = 0xFFF (00111111111111)`

- 상위 14 비트이므로 >> 12 하면 상위 2비트 추출

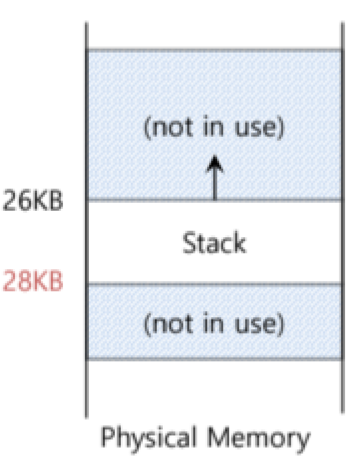


```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

- Bounds에는 각 segment의 bound 값이 있나 봄
- offset이 해당하는 segment의 bound를 넘으면 PROTECTION_FAULT
- physical address 계산 후 해당 메모리에 접근하여 레지스터에 저장하는 코드

Stack Segment를 참조하는 방법?

- 스택은 grows backward
  - address space에서의 방향과 physical memory에서의 방향이 다를 수 있음
  - base가 시작점이 아님. base는 오히려 종점임.
  - 그래서 offset을 다르게 계산하는 방법이 필요
- 추가적인 hardware support가 필요
  - hardware가 stack이 어느 방향으로 자라는지 check
  - 1 : positive direction, 0 : negative direction

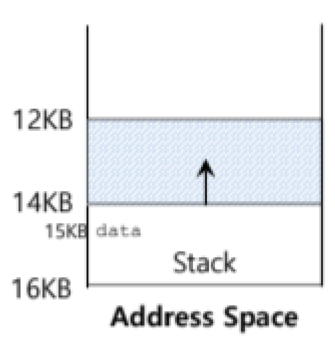


- physical memory에서 28KB가 base라 가정

Segment Register(with Negative-Growth Support)

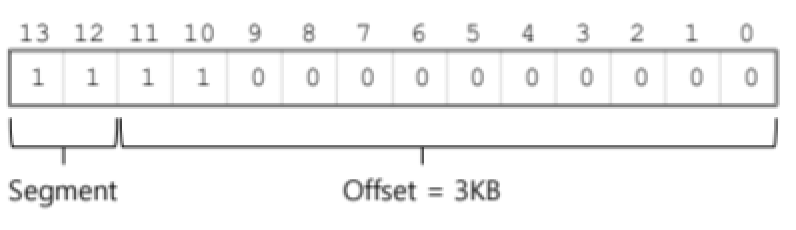
Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

- 주소가 줄어드는 쪽이므로 negative direction



Address Space

- address space 속 stack의 base는 12KB
- stack은 거꾸로 자라므로 16KB부터 시작
- data는 15KB에 있음
- 15KB = 11 1100 0000 0000



- 최대 segment size는 4KB (offset이 12bit이기 때문)
- offset가 3KB임. 이는 base에서 측정한 수치
- 하지만 stack은 거꾸로 자라므로 3KB - 4KB가 진정한 stack 속 데이터의 offset
- physical memory = 28KB - 1KB = 27KB

Support for Sharing

- 서로 다른 address space에서도 segment 공유 가능
- Code Sharing은 현대 시스템에서도 사용 중
- extra hardware support가 필요
  - protection bits 생성

Segment Register Values(with Protection)

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

  - 각 세그먼트 마다 read, write, execute에 대한 허가권을 나타내는 bit 생성

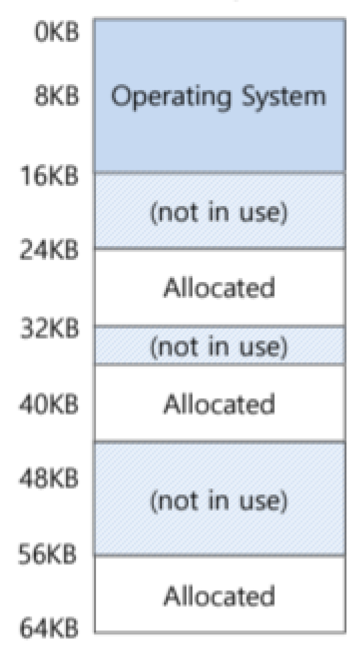
Fine-grained and Coarse-Grained

- Coarse-Grained : 적은 수의 segmentation
  - code, heap, stack
- Fine-Grained : 초기 시스템에서 쓰인 more flexible segmentation for address space
  - many segments
  - segment table이 존재
  - segment table를 구현하기 위한 hardware support가 필요

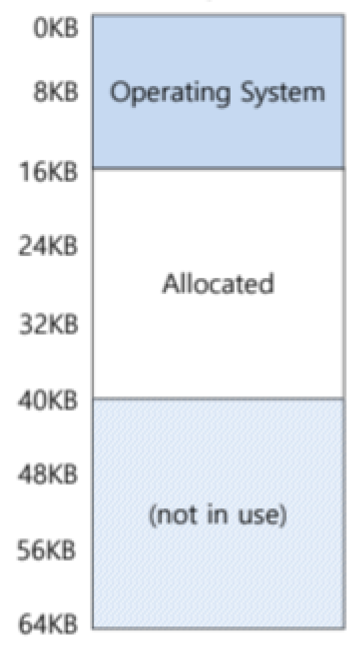
OS Support : Fragmentation

- External Fragmentation : 새 segment를 만들기 어렵게 하는 little free holes
  - 총합 24KB free space 가 있어도 one contiguous segment가 아닌 경우
  - OS가 20KB request를 만족시킬 수 없는 경우
- Compaction : segment들을 rearranging
  - 대신 비용이 좀 듦
  - 실행 중인 process를 중단해야하고
  - data를 어딘가로 복사해야하며
  - segment의 register 값들도 변경시켜야 함

Not compacted



Compacted



Segmentation 장단점

Pros

- sparse한 address space allocation 가능
  - stack과 heap이 독립적으로 grow
  - internal fragmentation이 없음
- fast, easy, well - suited to hardware
- segment를 공유하기 쉬움
  - 서로 같은 base/limit pair를 알고 있으면 됨
  - segment level에서의 code/data sharing
    - 만약 A, B 프로세스가 같은 libc를 사용한다면 segment 공유
    - 이 libc를 shraed library라고도 함
- 각 segment 마다 dynamic relocation 지원

Cons

- 각 segment들이 contiguous하게 할당되어야 함
  - external fragmentation 발생
  - large segment를 수용할 충분한 공간이 없을 수 있음
- 아직 Segmentation은 충분히 flexible하지 않아 보임
- stack, heap, code로 3개의 segment로 만들어도 이 셋 중 하나라도 엄청 크다면...?

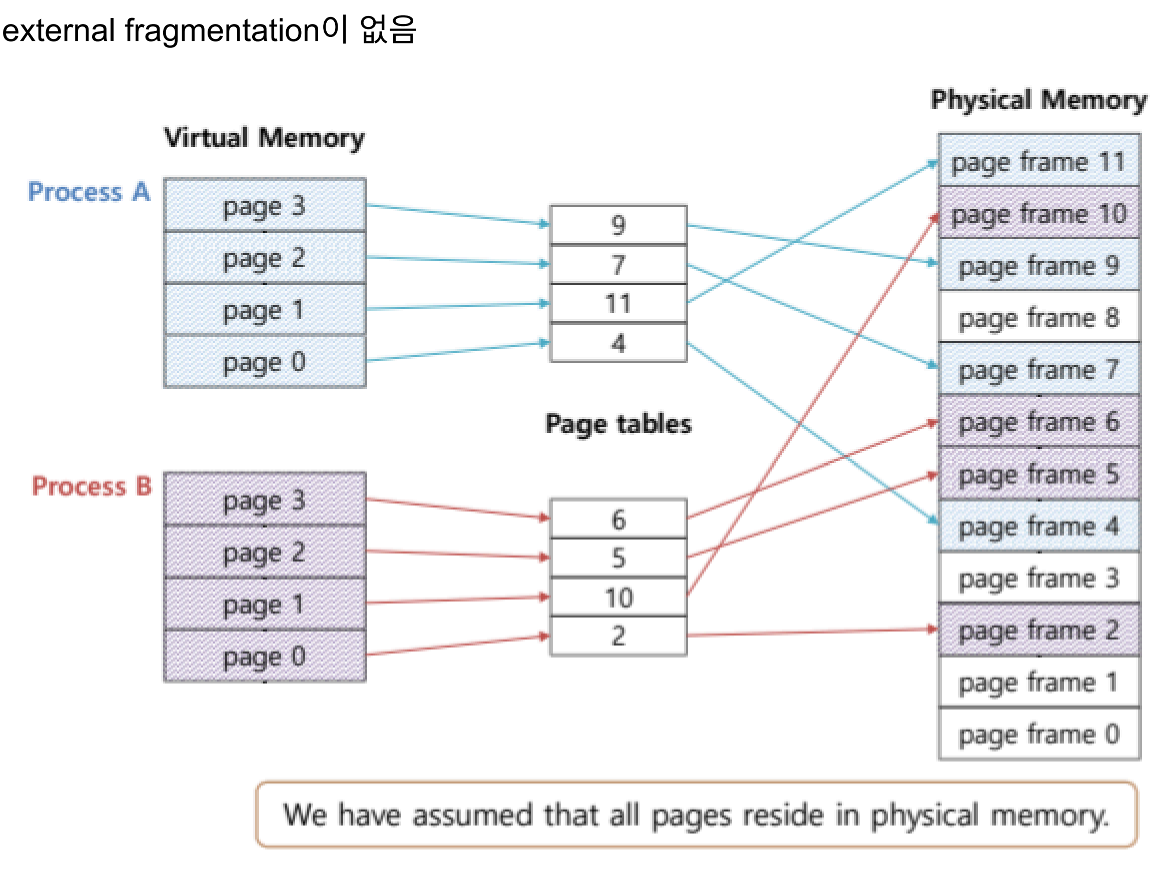
Paging

Concept

- space-management problem에 대한 두가지 접근 방법이 존재
  - Segmentation : variable size of logical segments (code, stack, heap, etc)
    - fragmentation 발생
    - 할당이 점점 challenging이 됨
  - Paging : address space를 page 단위로 split
- paging으로 physical memory를 여러 page 나눔
  - 이를 page frame이라 함
- 각 프로세스 당 virtual address를 physical address로 변환시키기 위해 page table 필요

paging pros

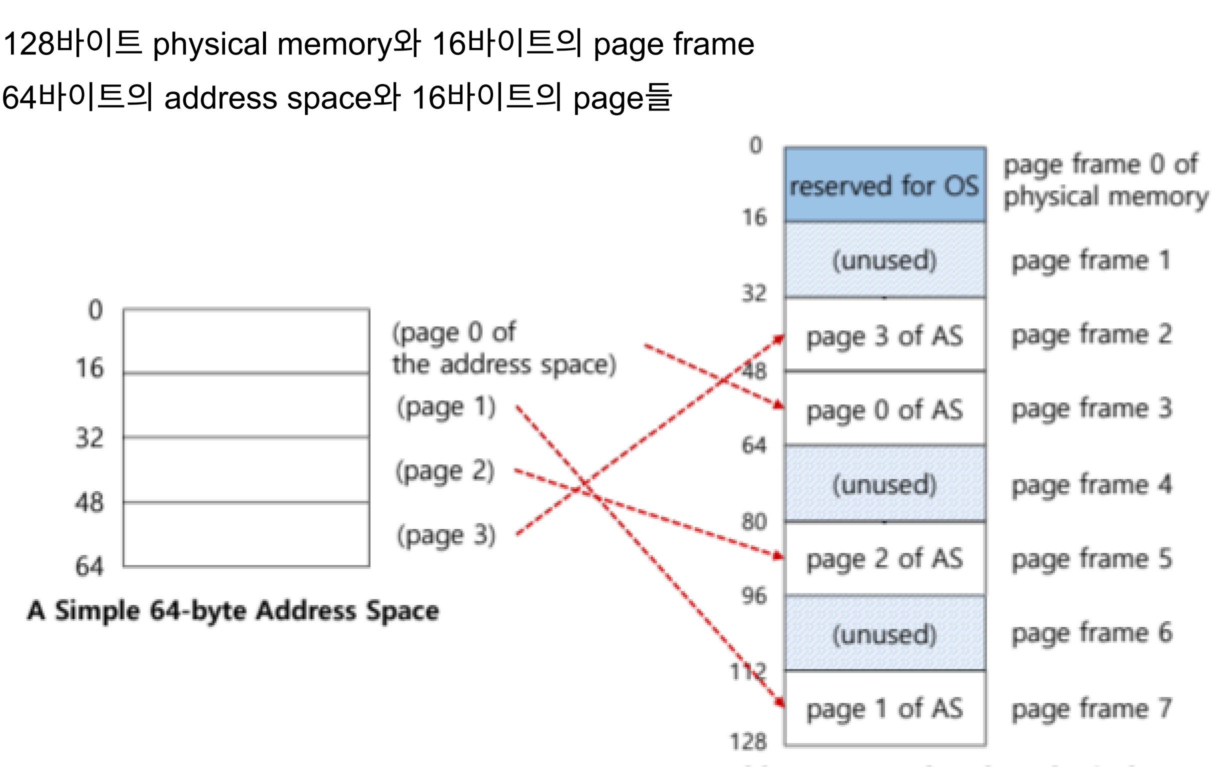
- 프로세스의 physical address space가 noncontiguous여도 괜찮아짐
  - virtual memory를 같은 크기의 블록으로 divide (page)
  - physical memory를 같은 크기의 블록으로 divide (frames)
  - page와 frame은 2^n
- memory management가 쉬워짐
  - OS가 모든 free frame을 주시
  - 크기 n page들로 이루어진 프로그램을 실행시키기 위해 크기 n의 free frame을 찾아야 함
  - virtual address를 physical address로 translate하기 위해 page table 설정
  - external fragmentation이 없음



We have assumed that all pages reside in physical memory.

Example : A Simple Paging

- 128바이트 physical memory와 16바이트의 page frame
- 64바이트의 address space와 16바이트의 page들



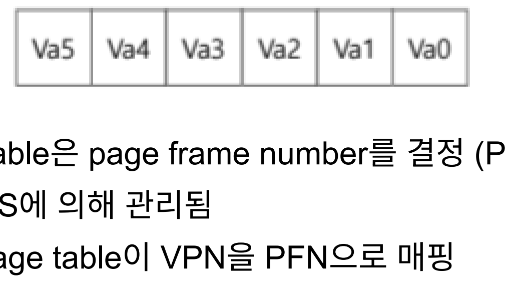
A Simple 64-byte Address Space

64-Byte Address Space Placed In Physical Memory

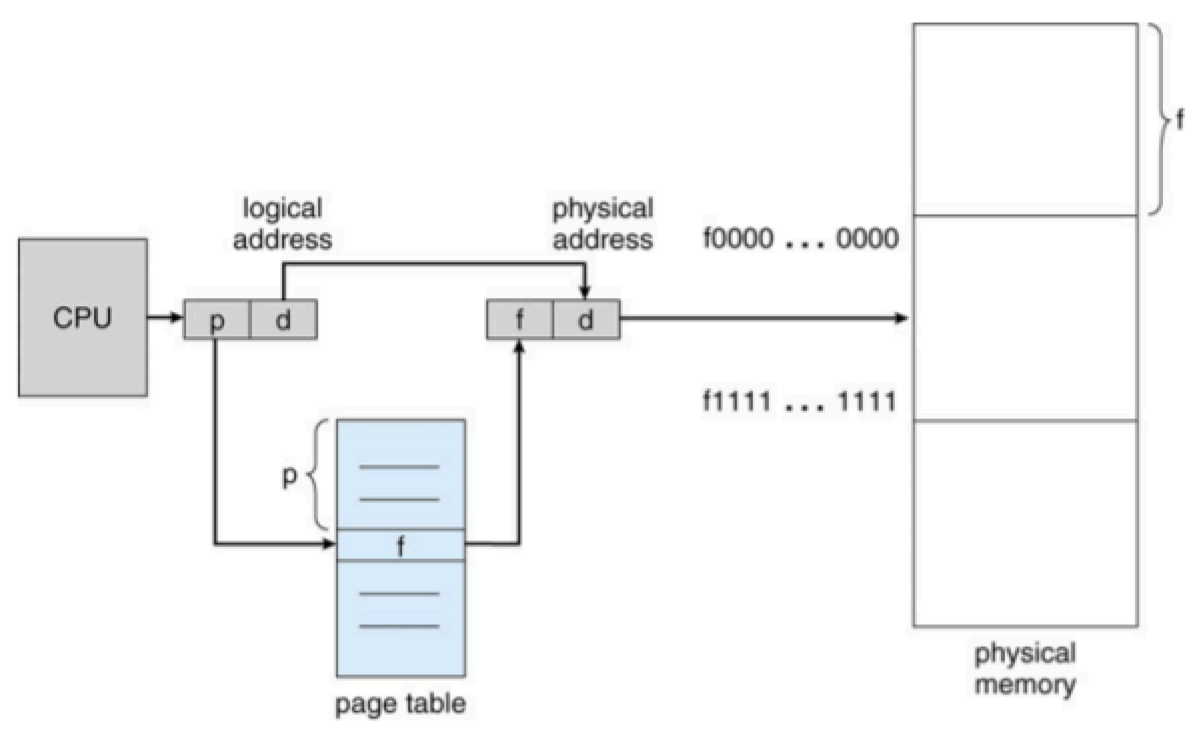
- page가 16바이트 이므로 하나의 address space에 4개의 pages
- physical memory를 미리 page frame으로 나눠놔서 page를 끼워맞추기만 하면 됨

Address Translation

- paging에서 주소 변환하는 방법?
- Two components in the virtual address
  - VPN : virtual page number
  - offset : 페이지 안에서의 offset
  - VPN은 page table 인덱스 역할

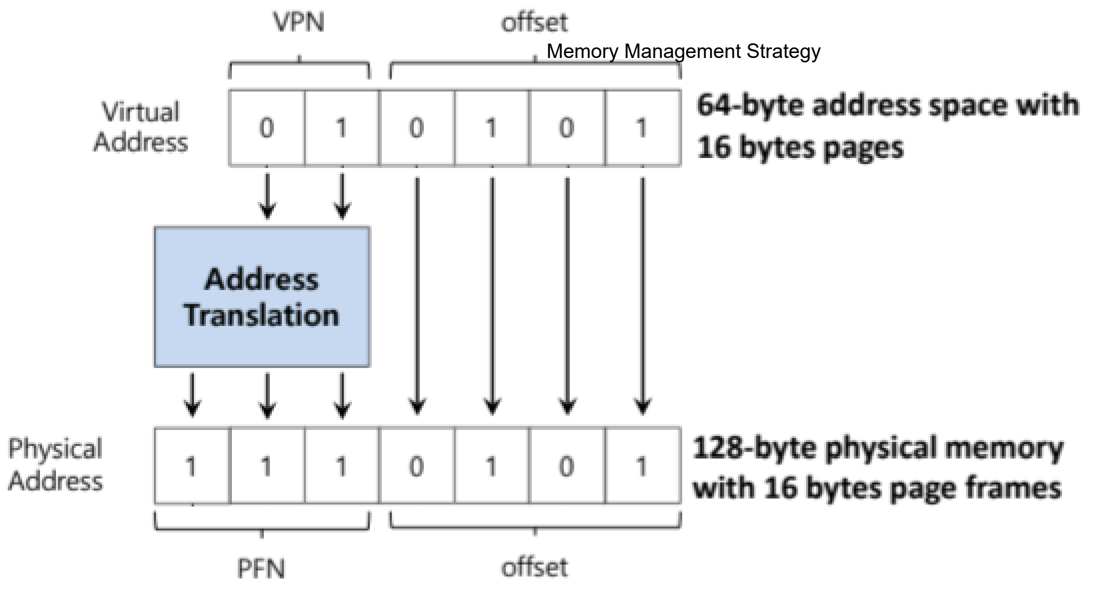


- page table은 page frame number를 결정 (PFN)
  - OS에 의해 관리됨
  - page table이 VPN을 PFN으로 매핑
  - virtual address space 속 page 마다 하나의 page table entry (PTE, 항목)가 존재
- physical address는 PFN과 Offset으로 이루어짐

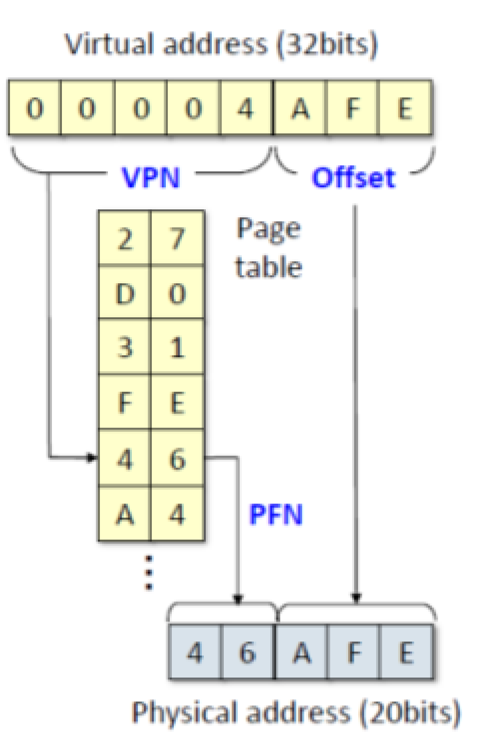


- logical address에서 page table을 통해 VPN을 PFN으로 변환시켜 physical address를 얻음

Example : Address Translation



- VPN을 address translation을 통해 PFN으로 변환
- offset은 그대로



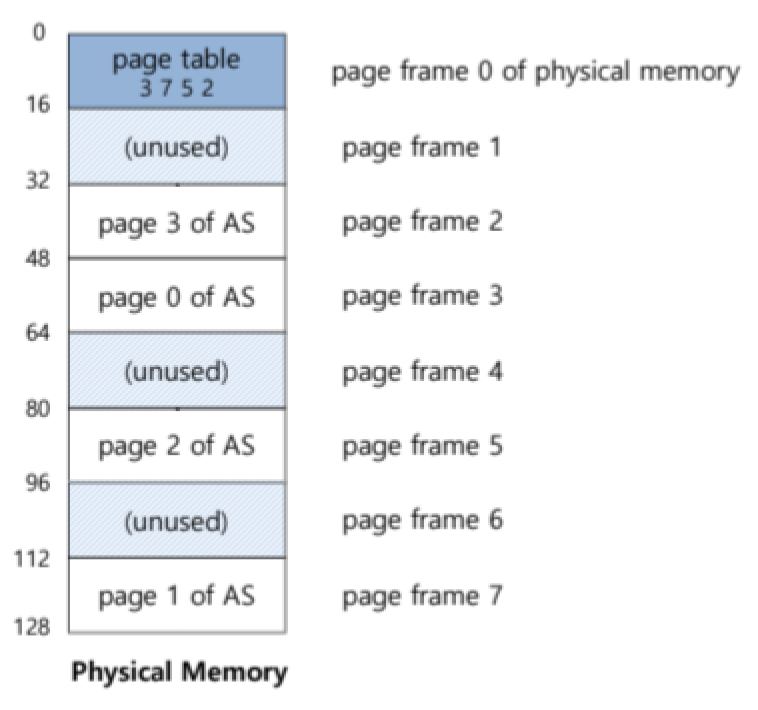
- VPN이 4번이므로 page table의 4번 인덱스 참조
- 참조된 값이 PFN을 의미
- offset은 그대로

internal fragmentation 계산해보기

- Page size = 2,048 bytes
- Process size = 72,766 bytes
- 35 pages + 1,086 bytes
  - 마지막 페이지에서 프로세스는 1086바이트만 사용함
- Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - 마지막 페이지에서 962바이트가 놀고 있음
- Worst case fragmentation = 1 frame – 1 byte
  - 만약 마지막 페이지에 프로세스가 1바이트만 사용된다면...?
- On average fragmentation = 1 / 2 frame size
  - 평균적으로 프로세스는 마지막 페이지의 절반만 사용한다고 함
- 프레임 크기를 작게하면 어떨까?
  - 각 page table entry를 위한 메모리가 필요
- 시간이 지나면서 페이지 크기가 늘어남
  - Solaris supports two page sizes – 8 KB and 4 MB

Page table은 어디에 저장될까?

- page table은 생각보다 엄청 커질 수 있음
- 각 프로세스에 대응하는 page table이 메모리에 저장되어야 함



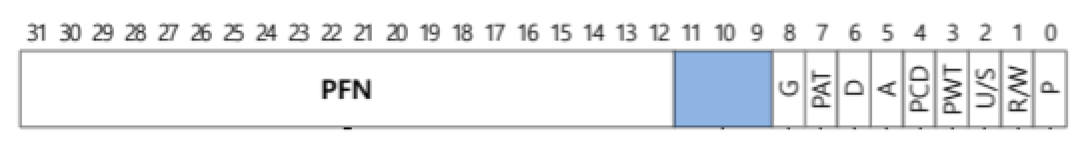
- page table도 physical memory의 frame에 저장됨

Page table엔 무엇이 저장되는가?

- page table은 그저 virtual address를 physical address로 매핑하는 데 사용되는 자료구조임
- 간단한 형태를 가짐
  - linear page table, an array
- OS는 VPN으로 인덱스를 매김

Page Table Entry의 common flags

- PTE에 공통 플래그가 존재
- valid bit : 지금 translation이 유효한지 나타냄
  - virtual address가 사용될 때 마다 check
- protection bit : page를 read, write, execute할 수 있는지 나타냄
- present bit : physical memory에 있는지 disk에 있는지 나타냄
  - disk에 있다는 건 swap out 됐다는 것
- dirty bit : page가 메모리로 온 뒤 수정되었는지 나타냄
- reference bit (accessed bit) : page가 access 되고 있는지 나타냄



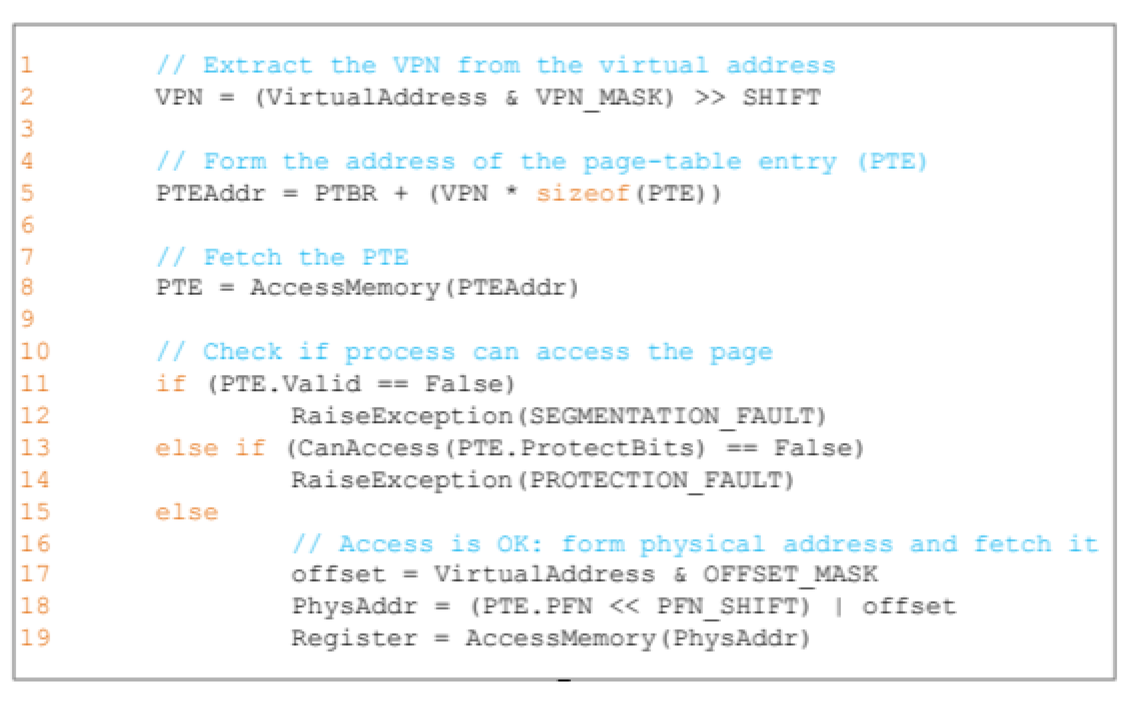
An x86 Page Table Entry(PTE)

- P : present
- R/W : read/write
- U/S : supervisor
- A : accessed
- D : dirty

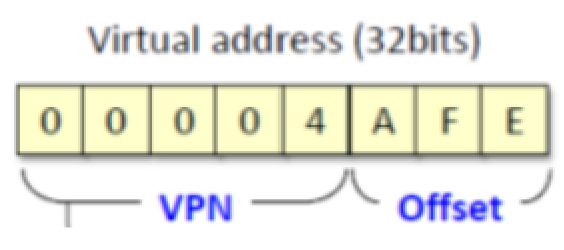
Paging : Too Slow

- 원하는 PTE의 위치를 얻기 위해선 page table의 시작 주소가 필요함
- 매번 memory reference마다 paging을 위해 OS는 one extra memory reference를 수행해줘야함

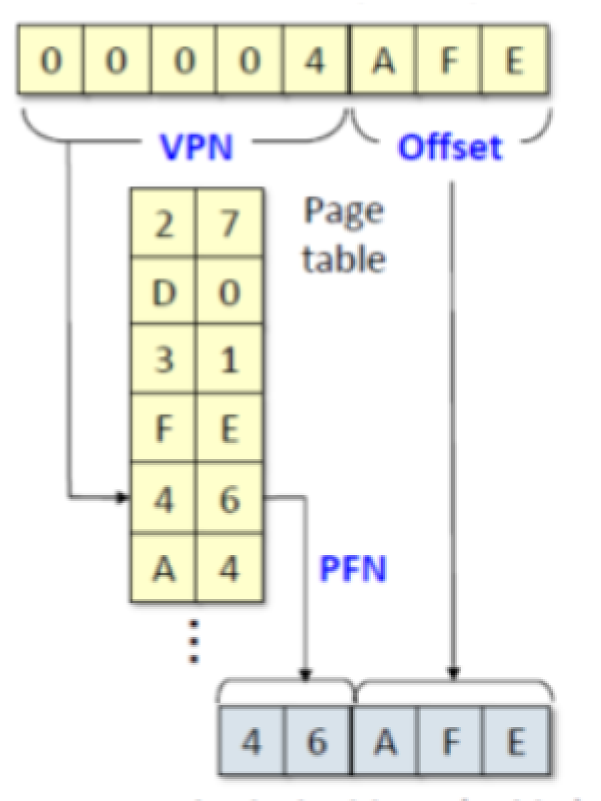
Accessing memory with paging



```
        // Extract the VPN from the virtual address
        VPN = (VirtualAddress & VPN_MASK) >> SHIFT

        // Form the address of the page-table entry (PTE)
        PTEAddr = PTBR + (VPN * sizeof(PTE))

        // Fetch the PTE
        PTE = AccessMemory(PTEAddr)

        // Check if process can access the page
        if (PTE.Valid == False)
                RaiseException(SEGMENTATION_FAULT)
        else if (CanAccess(PTE.ProtectBits) == False)
                RaiseException(PROTECTION_FAULT)
        else
                // Access is OK: form physical address and fetch it
                offset = VirtualAddress & OFFSET_MASK
                PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
                Register = AccessMemory(PhysAddr)
```



- virtual address 에서 vpn마스킹 비트를 통해 vpn 비트 추출
- page table 시작 주소에 VPN * sizeof(PTE)를 통해 해당 PTE의 주소 계산
- PTE 항목 접근
- common flag check
  - PTE의 common flag 중 valid bit가 유효하지 않으면 SEGMENTATION_FAULT
  - protectbit도 유효하지 않다면 PROTECTION_FAULT
- flag가 유효하다고 판단되면 access
  - virtual address에서 offset 추출
  - PTE의 frame number와 offset을 합해서 physical address 계산



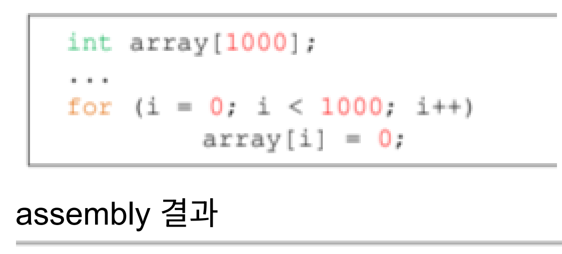
  - 해당 physical address에 access해서 레지스터 값 얻기

Memory Trace

- 다음과 같은 simple memory access 코드 실행

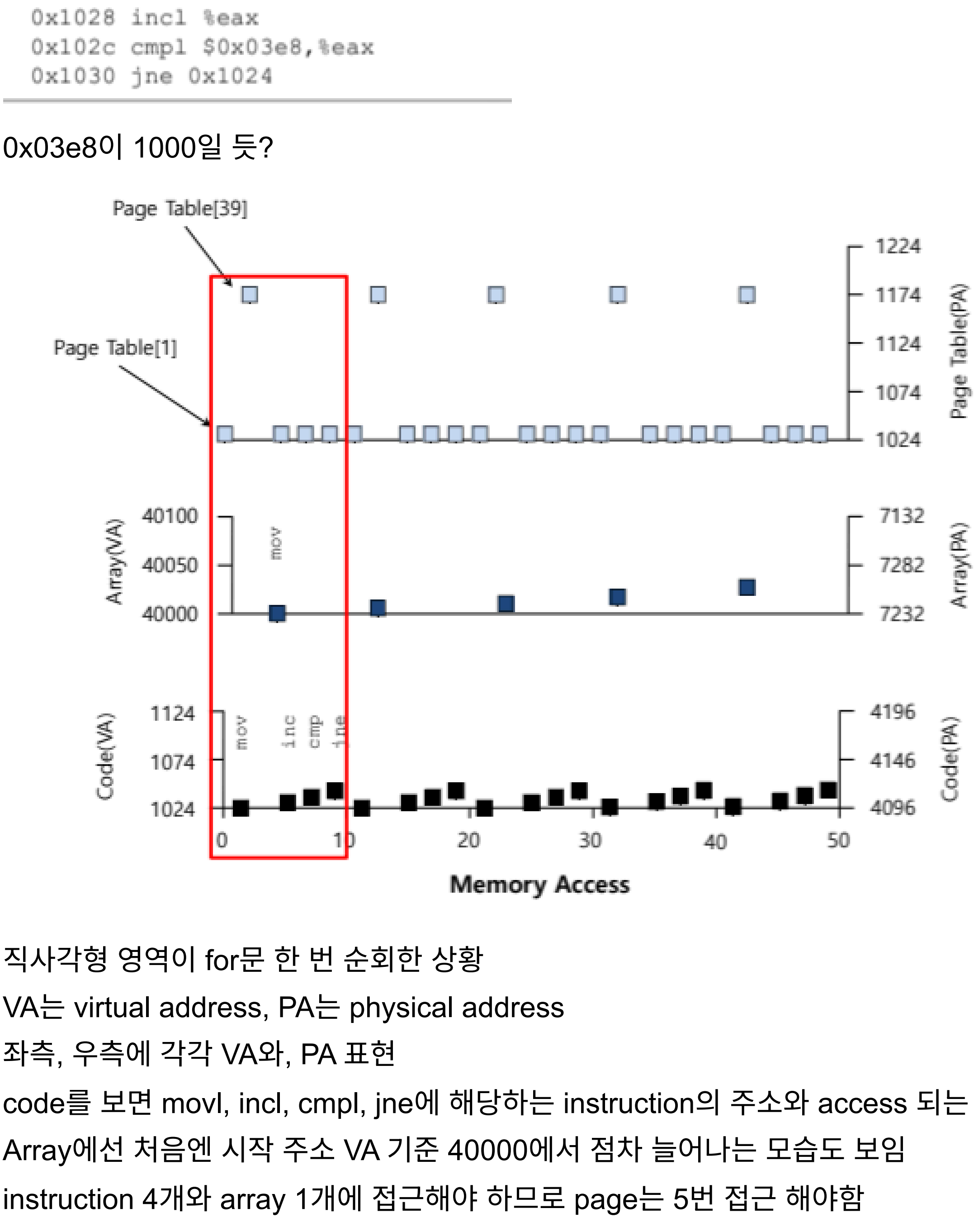
```
int array[1000];
...
for (i = 0; i < 1000; i++)
        array[i] = 0;
```

- assembly 결과



```
0x1024 movl $0x0,(%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

- 0x03e8이 1000일 듯?



- 직사각형 영역이 for문 한 번 순회한 상황
- VA는 virtual address, PA는 physical address
- 좌측, 우측에 각각 VA와, PA 표현
- code를 보면 movl, incl, cmpl, jne에 해당하는 instruction의 주소와 access 되는 모습이 보임
- Array에선 처음엔 시작 주소 VA 기준 40000에서 점차 늘어나는 모습도 보임
- instruction 4개와 array 1개에 접근해야 하므로 page는 5번 접근 해야함
- instruction은 동일한 page에 있나봄. 다만 array는 다른 page에 있음을 알 수 있음

Demand paging

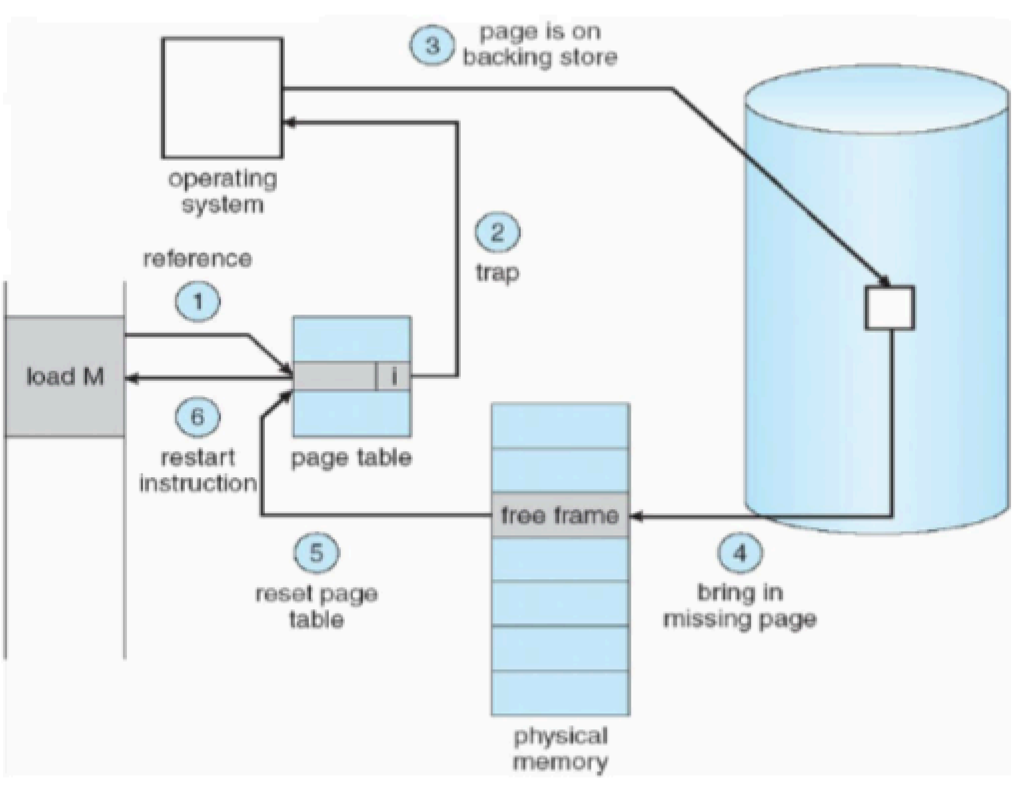
- 모든 데이터를 한 번에 메모리에 올리지 않음
- 나머지는 디스크에 남겨둠
- OS는 main memory를 프로세스가 할당하는 모든 데이터의 cache로 사용함
  - 페이지를 필요할 때만 memory로 가져옴
  - page는 physical memory frame에서 쫓겨날 수 있음
  - 쫓겨난 page는 disk로 감
    - dirty page만 evicted(written)
    - dirty : 수정
  - page의 이동은 프로세스가 모름 (몰라도 됨, transparent)
- 장점
  - Less I/O
  - Less memory
  - faster response
  - 메모리 절약으로 더 많은 프로세스 생성 가능

Page Fault

- invalid PTE 접근 시 CPU가 exception raise
- Major page faults
  - 프로그램이 필요한 페이지를 요청했는데 그 페이지가 메모리에 없는 경우
  - page가 유효하지만 메모리에 load되지 않은 경우
  - OS는 contents가 어디있는지에 대한 정보를 maintain
  - 그래서 해당 page가 디스크에 어딨는지 알고 있음
  - 가져오려면 disk I/O가 필요 (수천 배 느림)
- Minor page faults
  - 메모리 어딘가에 페이지가 있음
  - 피에지 테이블에 아직 정보가 없어 잠깐 혼란이 생김
  - 금방 해결되는 상황
  - disk I/O 없이도 해결 가능

  - 프로그램이 stack이나 heap을 처음 사용할 때는 아직 메모리가 할당되지 않은 상태임
    - 이 상태에 접근하면 page fault
    - lazy allocation..?
  - OS가 미리 예측해서 페이지를 메모리에 불러오는 데 아직 page table에 등록이 안 되어있을 수 있음

Handling Page Faults



1. page table 참조
2. page fault 발생 (trap)
3. OS는 정보를 maintain하므로 이를 통해 디스크 내에서 page 가져옴
4. memory에 적재
5. page table reset
6. instruction 다시 시작

Page table 구현

- page table은 main memory에 존재
- page table base register (PTBR)은 pabe table을 pointing
- page table length register (PTLR)은 page table의 크기를 indicating
- 이 방법에선 모든 data와 instruction 접근이 두 memory access를 수행하게 됨
  - page table 접근
  - data/instruction 접근
- two memory access problem은 special fast-lookup hardware cached를 사용함으로써 해결
  - special fast-lookup hardware cached : Translation Look-aside buffers (TLB)
  - 암튼 이런 게 있나봄

Paging : Pros

- external fragmentation이 없음
- allocation과 free가 빠름
  - free page frame에 대한 list와 bitmap이 존재
  - allocation : contiguos한 free space를 찾지 않아도 됨
  - free : 인접한 free space를 coalesce(합치다)할 필요 없음
- memory 일부를 disk로 page out하기 쉬움
  - 페이지 크기는 디스크 블록 크기의 배수로 정해짐
  - paged-out page에 참조하는 걸 탐지하기 위해 valid bit 사용
  - 몇몇 page가 디스크에 있더라도 프로세스 실행 가능
- page를 share하고 protect 하기 쉬움

Paging : Cons

- internal fragmentation 존재
  - page가 클수록 메모리 낭비도 커질 가능성
- memory reference overhead
  - instruction당 메모리 참조 수가 두배
  - 이는 hardware support(TLBs)로 해결 가능하긴 함
- page table을 저장할 공간 필요
  - 페이지 당 하나의 PTE가 필요
    - 페이지 크기가 4KB인 32bit address space에서 PTE의 개수는 2^20
    - PTE당 4바이트 필요. 이는 page table 당 4MB가 필요함을 의미
    - 시스템 내 100개의 프로세스가 존재하면 page tables의 크기는 총 400MB
  - Solution
    - valid PTE들만 저장
    - page table을 page...?