지금까지의 문제점

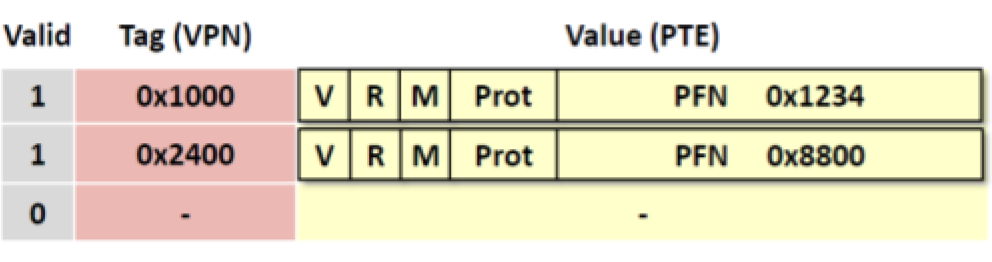
- Address traslation은 너무 느림
  - 메모리를 lookup 할려면 데이터를 가져오는 작업 외에 page table을 조회해야하는 비용이 추가되므로 비용이 두배
  - multi-level page면 비용이 더더욱 증가
- 한 번 address translation을 빠르게 만들어보자
  - virtual address에서의 fetching을 physical address에서의 fetching 만큼 효율적으로 만들어보자

TLB (Translation Lookaside Buffer)

- MMU의 일부임 (메모리 접근시간 vs MMU 접근시간)
- 자주 쓰이는 virtual address를 physical address 로 변환해서 저장해놓는 하드웨어 캐쉬

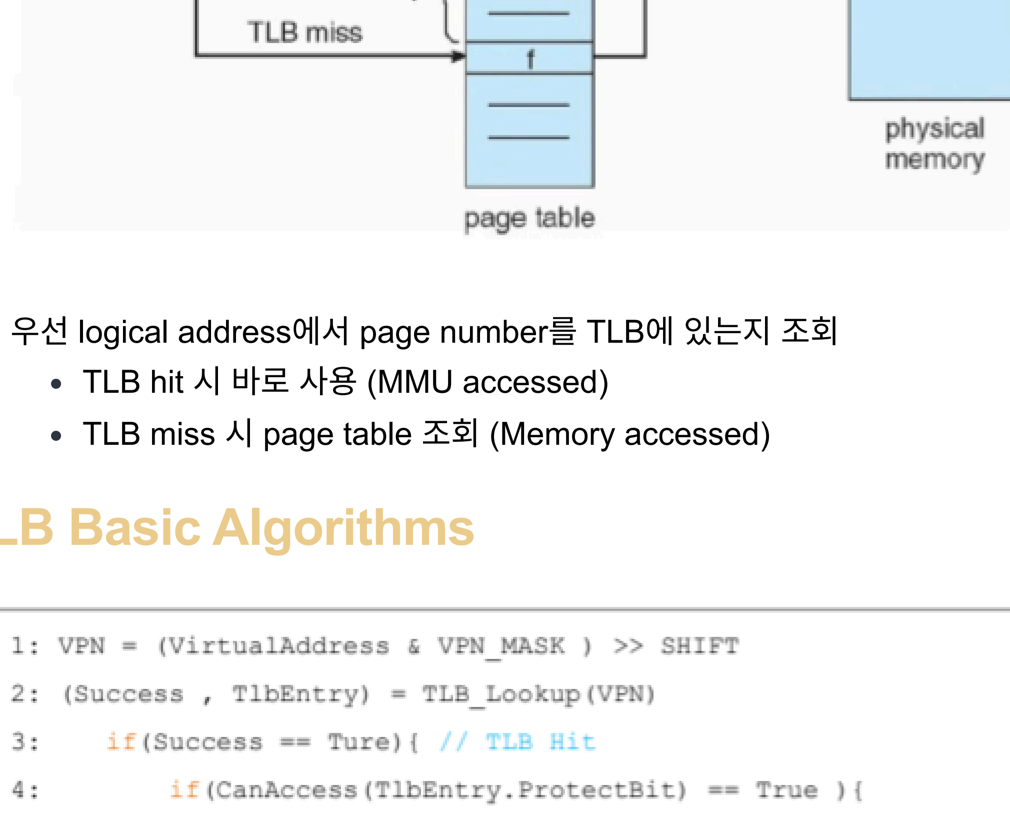
TLB 구조

- TLB는 하드웨어로 구현됨
- 프로세스는 한 번에 a handful of pages 만큼만 사용
  - TLB에 보통 16~256 entries
- 보통 fully assoiciative (완전 연관 방식)
  - 완전 연관 방식? : 캐시의 어떤 entry라도 어떤 주소와도 매핑될 수 있음
  - 검색할 때 모든 캐시 entry를 동시에 확인함
  - 유연성은 높지만 하드웨어 복잡도와 지연이 큼
  - latency를 줄이기 위해 set associative일 수도 있음
- entry의 구조



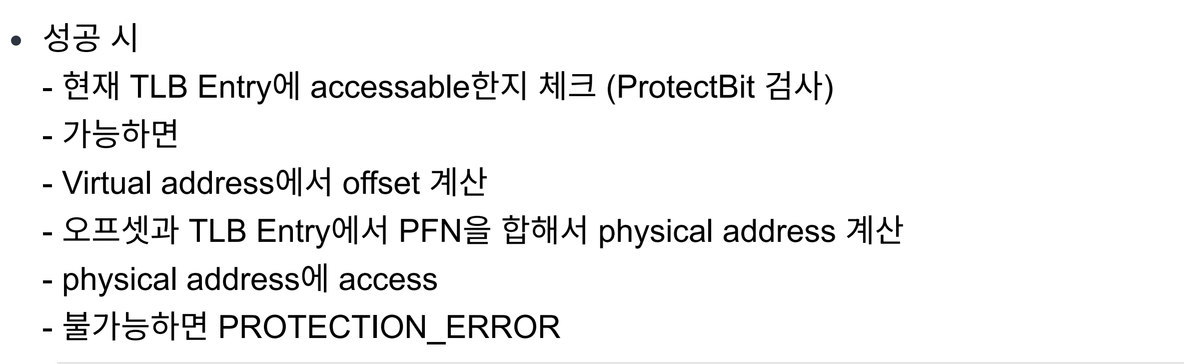
Valid	Tag (VPN)	Value (PTE)				
1	0x1000	V	R	M	Prot	PFN 0x1234
1	0x2400	V	R	M	Prot	PFN 0x8800
0	-	-				

Address Translation with TLB



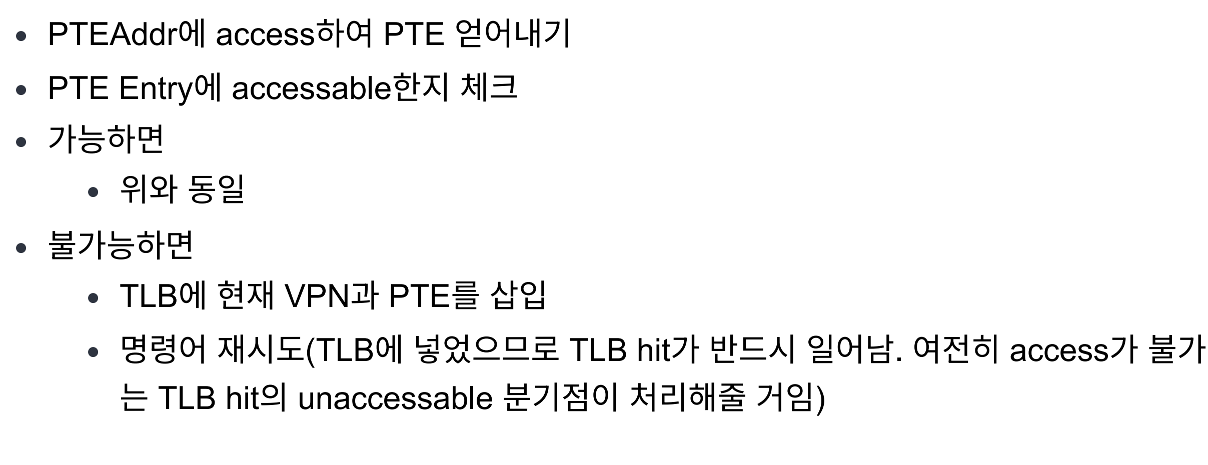
- 우선 logical address에서 page number를 TLB에 있는지 조회
  - TLB hit 시 바로 사용 (MMU accessed)
  - TLB miss 시 page table 조회 (Memory accessed)

TLB Basic Algorithms



```
VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
(Success , TlbEntry) = TLB_Lookup(VPN)
   if(Success == Ture){ // TLB Hit
       if(CanAccess(TlbEntry.ProtectBit) == True ){
           offset = VirtualAddress & OFFSET_MASK
           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
           AccessMemory( PhysAddr )
       }else RaiseException(PROTECTION_ERROR)
```

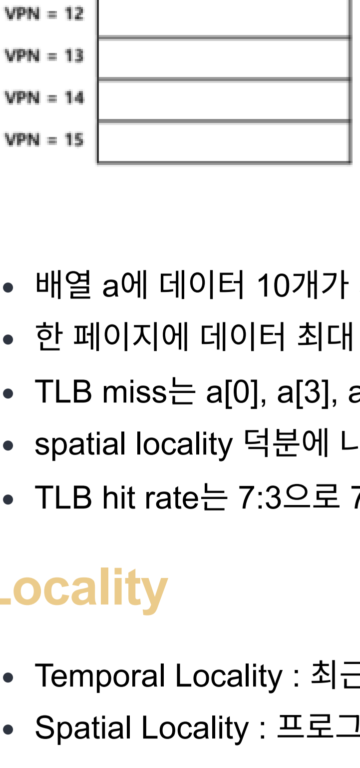
- 비트마스킹으로 VirtualAddress에서 VPN (Virtual page number) 추출
- VPN으로 TLB 조회
- 성공 시
  - 현재 TLB Entry에 accessable한지 체크 (ProtectBit 검사)
  - 가능하면
  - Virtual address에서 offset 계산
  - 오프셋과 TLB Entry에서 PFN을 합해서 physical address 계산
  - physical address에 access
  - 불가능하면 PROTECTION_ERROR

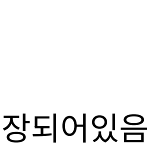


```
   }else{ //TLB Miss
       PTEAddr = PTBR + (VPN * sizeof(PTE))
       PTE = AccessMemory(PTEAddr)
       (…)
       }else{
       TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
       RetryInstruction()
       }
}
```

- 실패 시
  - PTBR(Page table의 시작주소)에 VPN과 PTE의 크기를 조합하여 PTEAddr 계산
  - PTEAddr에 access하여 PTE 얻어내기
  - PTE Entry에 accessable한지 체크
  - 가능하면
    - 위와 동일
  - 불가능하면
    - TLB에 현재 VPN과 PTE를 삽입
    - 명령어 재시도(TLB에 넣었으므로 TLB hit가 반드시 일어남. 여전히 access가 불가능한 경우 이는 TLB hit의 unaccessable 분기점이 처리해줄 거임)

Example : Accessing an array





```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

The TLB improves performance due to spatial locality

3 misses and 7 hits.
Thus TLB hit rate is 70%.

- 배열 a에 데이터 10개가 저장되어있음
- 한 페이지에 데이터 최대 4개 저장 가능하다고 가정
- TLB miss는 a[0], a[3], a[7]에서 일어날 것임
- spatial locality 덕분에 나머지는 hit 발생
- TLB hit rate는 7:3으로 70퍼센트

Locality

- Temporal Locality : 최근 access된 data or instruction은 다시 access될 가능성이 높다.
- Spatial Locality : 프로그램이 x를 방문하면 이후 x 근처를 방문할 가능성이 높다.

TLB Miss 처리

CISC에서는 하드웨어가 전적으로 TLB miss를 처리

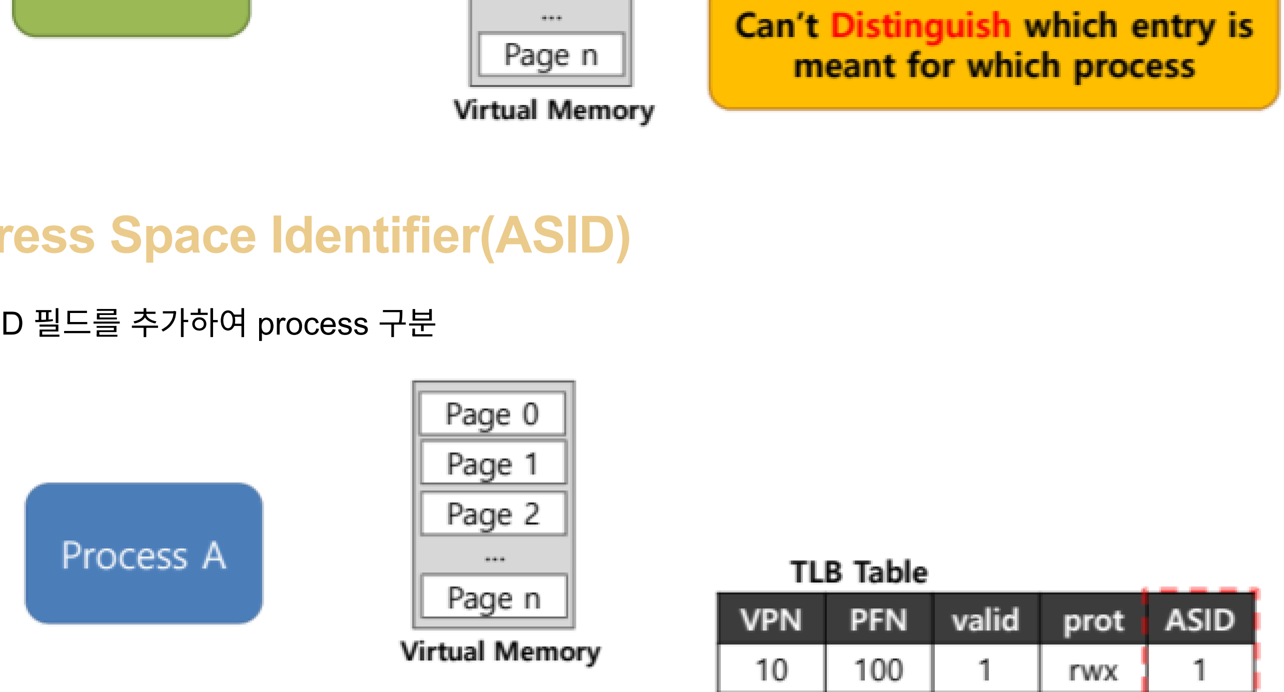
- 하드웨어는 page table이 메모리 어디에 있는지 알아야 함
- 하드웨어가 page table로 가서 frame number를 찾아 주소 변환을 하고 TLB에 update 후 재시도

RISC는 software-managed TLB를 사용

- TLB Miss 시 하드웨어가 exception을 raise(trap handler)
- OS의 trap handler가 page table을 탐색하고, TLB를 수동으로 갱신한 뒤 instruction을 재실행

TLB Issue : Context Switching

- TLB table에 entry들을 저장해놓음
- 근데 이 entry들은 누구의 process 것인지 알 수가 없음



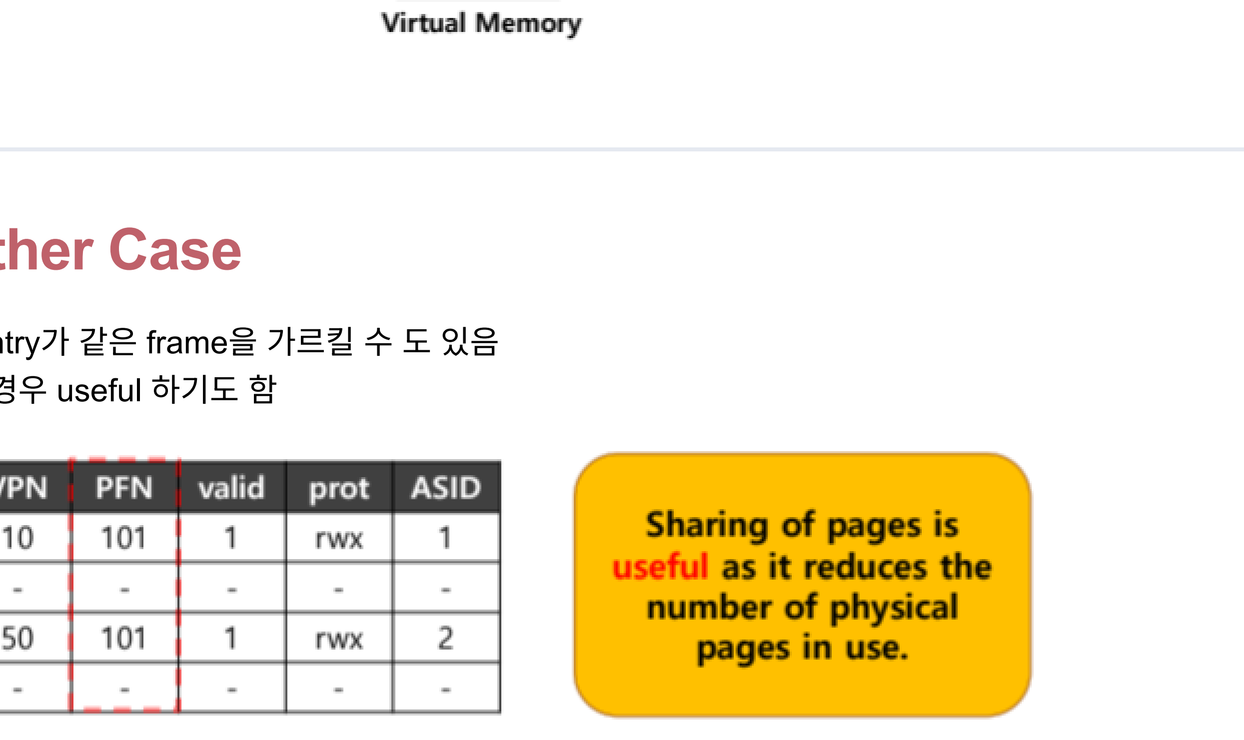
TLB Table

VPN	PFN	valid	prot
10	100	1	rwx
-	-	-	-
10	170	1	rwx
-	-	-	-

Can't Distinguish which entry is meant for which process

Address Space Identifier(ASID)

- ASID 필드를 추가하여 process 구분

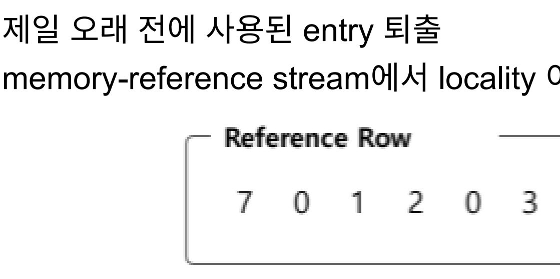


TLB Table

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
-	-	-	-	-
10	170	1	rwx	2
-	-	-	-	-

Another Case

- 두 entry가 같은 frame을 가르킬 수 도 있음
  - 이 경우 useful 하기도 함

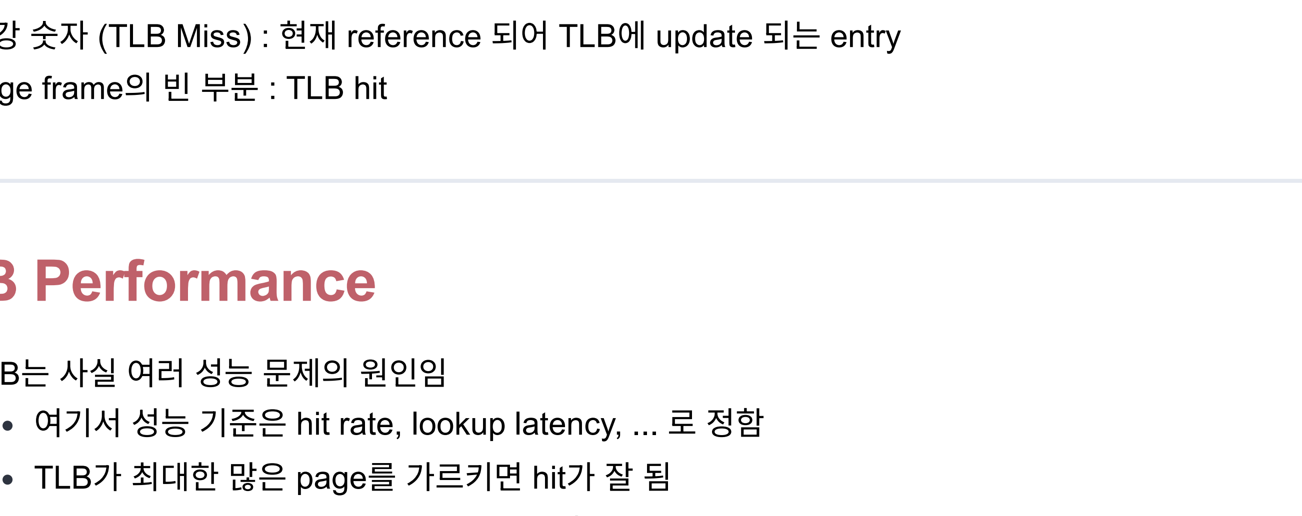


VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

Sharing of pages is useful as it reduces the number of physical pages in use.

TLB Replacement Policy

- LRU 방식 (Least Recently Used)
  - 제일 오래 전에 사용된 entry 퇴출
  - memory-reference stream에서 locality 이용



- reference row : memory를 reference하는 순서
- 빨강 숫자 (TLB Miss) : 현재 reference 되어 TLB에 update 되는 entry
- page frame의 빈 부분 : TLB hit

TLB Performance

- TLB는 사실 여러 성능 문제의 원인임
  - 여기서 성능 기준은 hit rate, lookup latency, ... 로 정함
  - TLB가 최대한 많은 page를 가르키면 hit가 잘 됨
- 문제를 해결하기 위해선 TLB reach를 늘려야 함

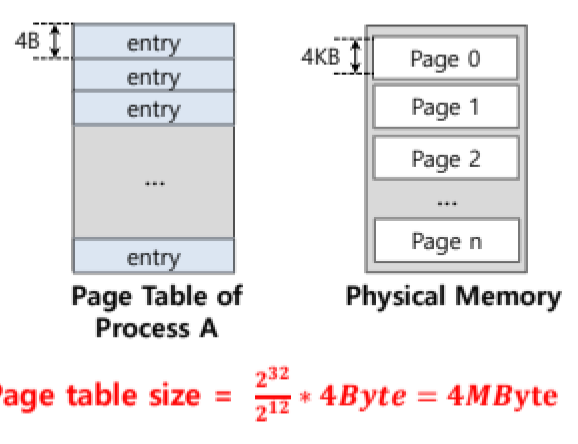
- TLB reach : TLB가 영향을 미치는 영역
  - TLB reach = # TLB entries * Page size

Memory management TLBs and Smaller Tables

- 다른 방법은 multi-level TLBs 사용
- 자료구조나 알고리즘을 TLB-friendly하게 설계하는 것도 도움이 됨

Paging: Linear Tables

- **현재 page table이 생성되는 방식은 페이지 마다 PTE를 생성해서 표현하는 방식임**
  - (linear page table)
- **이 방식대로 한 번 page table size를 구해보자**
- 페이지 크기는 4KB로 합의
- PTE는 0~11까지 기타 정보, 12~32까지 PFN
  - 총 32비트, 4바이트
- 프로세스 당 page table은 하나
- 32비트 시스템에서는 주소를 2^32 까지만 표현 가능
  - 이론 상 메모리도 4GB만 사용 가능
  - PAE 같은 기술로 더 넓은 주소에 접근 가능
- 2^32 까지 접근 가능한데 페이지 하나 당 4KB
  - 이론 상 총 2^32/2^12 개의 페이지 생성 가능
- PTE 하나 당 페이지 하나를 가르키므로 PTE도 2^32/2^12 생성
- page table은 PTE들로 이루어짐
  - PTE가 2^32/2^12개에 개당 4바이트면 page table의 크기는 2^32/2^12 * 4

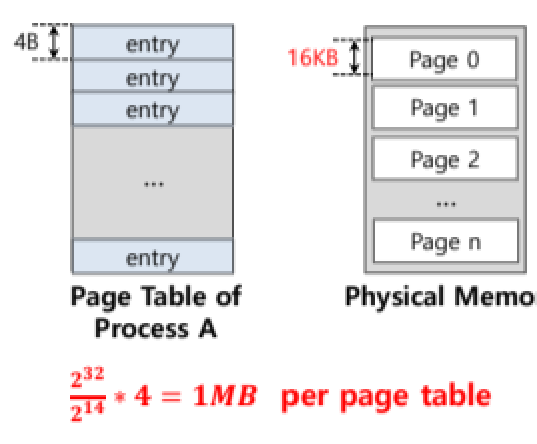


Page table size = $\frac{2^{32}}{2^{12}} * 4Byte = 4MByte$

=> 이런 방식으로는 테이지 테이블의 크기가 너무 큼

Paging : smaller Tables

- 페이지의 크기를 늘리면 어떨까?
- PTE가 가르켜야 하는 페이지 개수가 줄어드므로 page table의 크기도 줄어듦

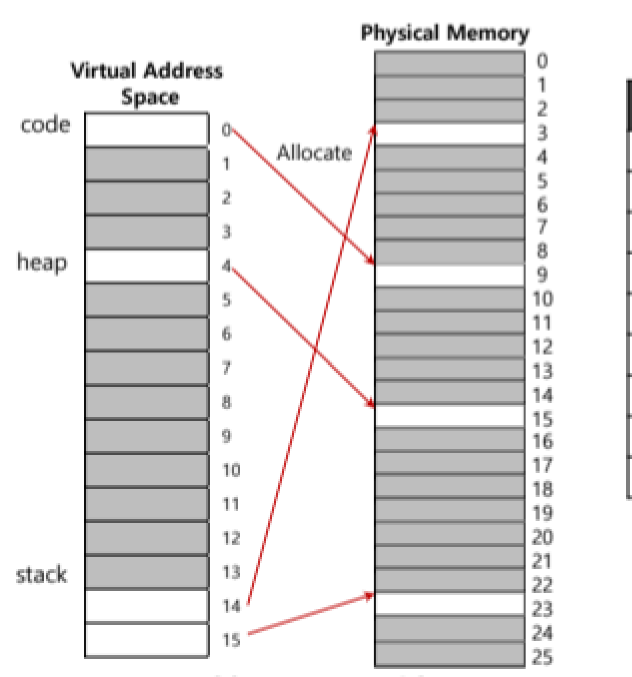


$\frac{2^{32}}{2^{14}} * 4 = 1MB$ per page table

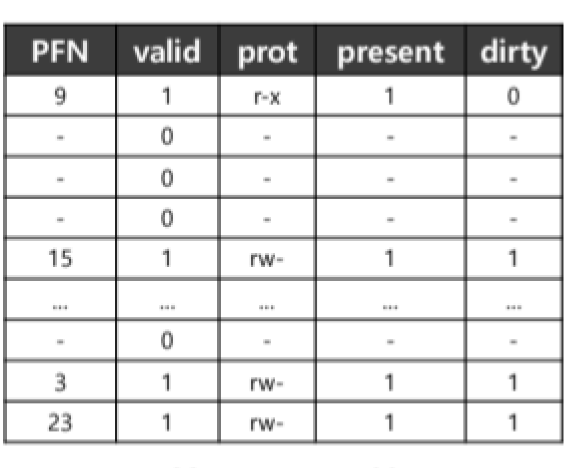
=> 하지만 Big pages는 internal fragmentation의 문제를 가짐

왜 페이지 테이블의 크기가 클까?

- 사용하지 않는 페이지도 페이지 테이블에서 표현하는 것이 원인
- 왼쪽의 virtual address space에서 차례대로 page들을 page table에 매핑
  - 프로세스마다 virtual address space를 가짐
- 정작 사용되는 페이지는 4개밖에 없음 (지금과 같은 경우 PTE 12개가 낭비)



A 16KB Address Space with 1KB Pages



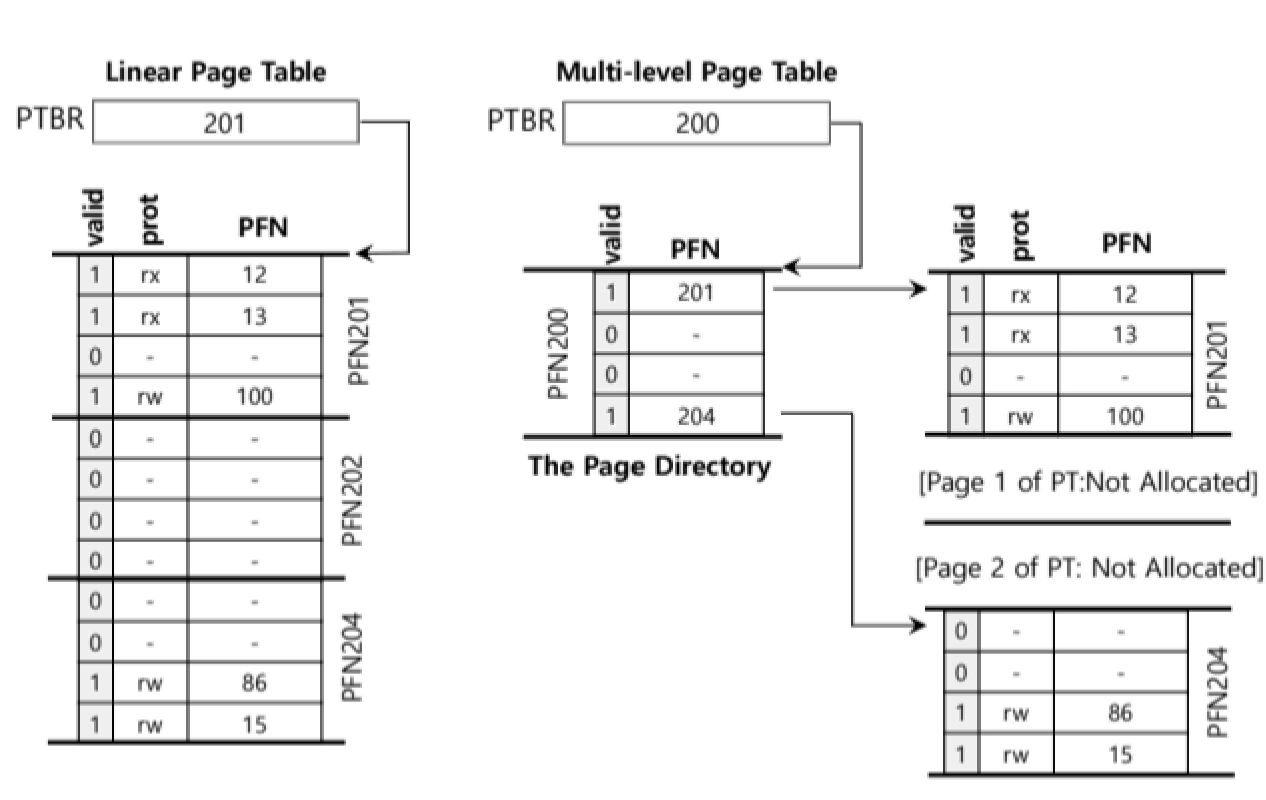
PFN	valid	prot	present	dirty
9	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...	...	...	...	...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

A Page Table For 16KB Address Space

=> Multi-level page tables 사용

Multi-level page tables

- page directory라는 자료구조 사용
- linear page table을 tree처럼 구현
  - linear page table을 다시 페이지 크기만큼 쪼갬
  - page table을 다시 page라 생각해보는 것
    - 원래 page에는 메모리에 놓이는 프로세스의 코드, 데이터들이 들어가는데 이번에는 PTE들만 들어감
  - 이 page에서 PTE들이 전부 invalid인 경우 필요없는 page므로 page directory에서 invalid로 놓고 실제 PTE들을 할당하지 않음
  - page directory는 multi level 트리와 비슷한 자료구조
- page directory의 entry는 페이지 테이블을 페이지 크기만큼 쪼갠 만큼을 나타냄
  - PDE (page directory entry)
- PDE는 valid bit와 PFN으로 구성
  - PDE의 페이지는 linear page table을 다시 page로 쪼갠 형태이므로 PDE의 PFN은 이 page들의 frame number임
  -



Linear (Left) And Multi-Level (Right) Page Tables

advantage

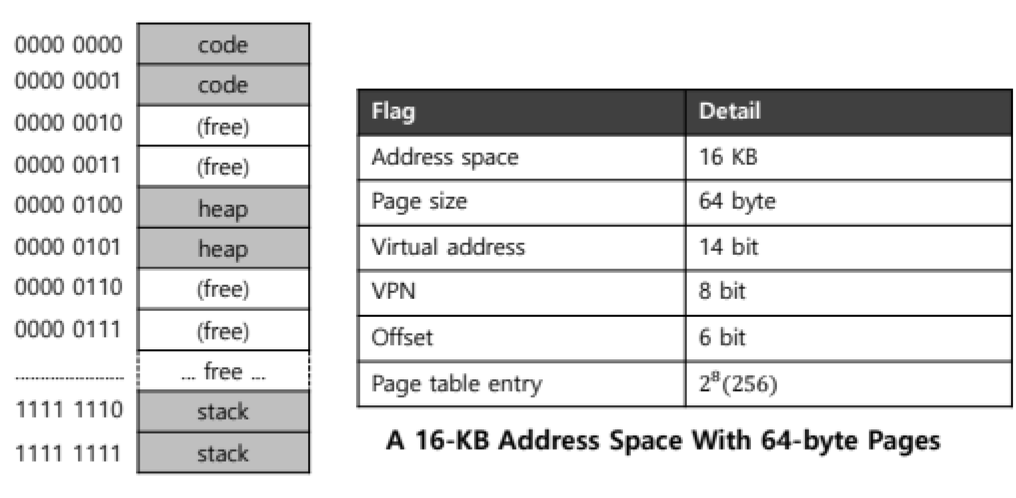
- 사용 중인 address space 양에 비례하여 page table space를 할당
- page table을 늘리거나 할당해야할 때 next free page를 OS가 grab 할 수 있음

disadvantage

- time - space 의 trade off
- complexity

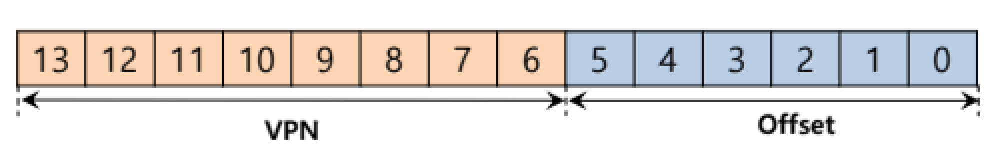
A Detailed Multi-Level Example

- 예시를 한 번 들어보자

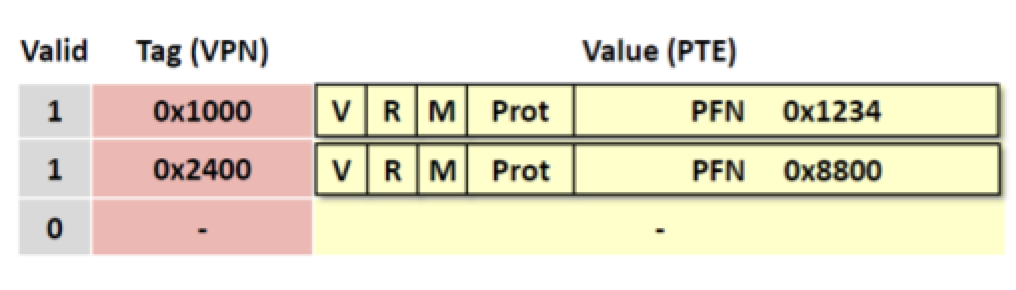


Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	2^8(256)

A 16-KB Address Space With 64-byte Pages

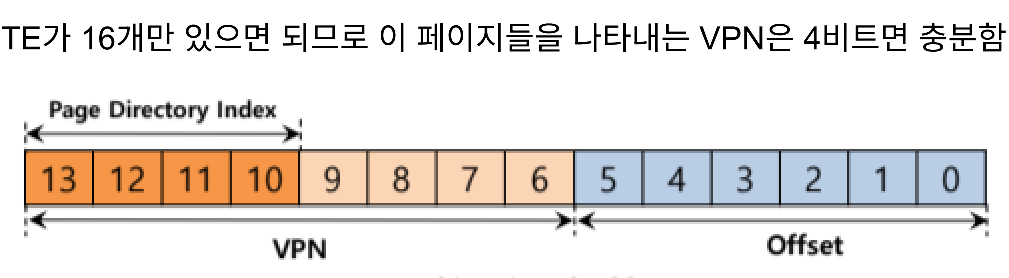


- 우선 address space를 16KB로 가정
  - 왼쪽 그림 page table에서 총 256개의 PTE 확인 (= 256개의 페이지)
  - page size를 64 바이트로 가정
    - 사용되는 메모리는 총 256 x 64 = 2^8 x 2^6 이므로 2^14 = 16KB
    - 그러므로 address space는 16KB로 계산됨
- 사용되는 메모리 즉, address space가 16KB 이므로 virtual address는 14비트로 표현 가능
  - 2^14 = 16KB
- 여기서 page size가 64바이트 이므로 offset bit는 6개로 충분
- page 개수가 256개 이므로 VPN bit는 8개로 충분
- 만약 PTE의 크기를 4바이트라 가정하고 page table이 모든 page들을 가르킨다고 가정하면 page가 256개 이므로 page table의 크기는 1KB
  - 다시보는 PTE 구조

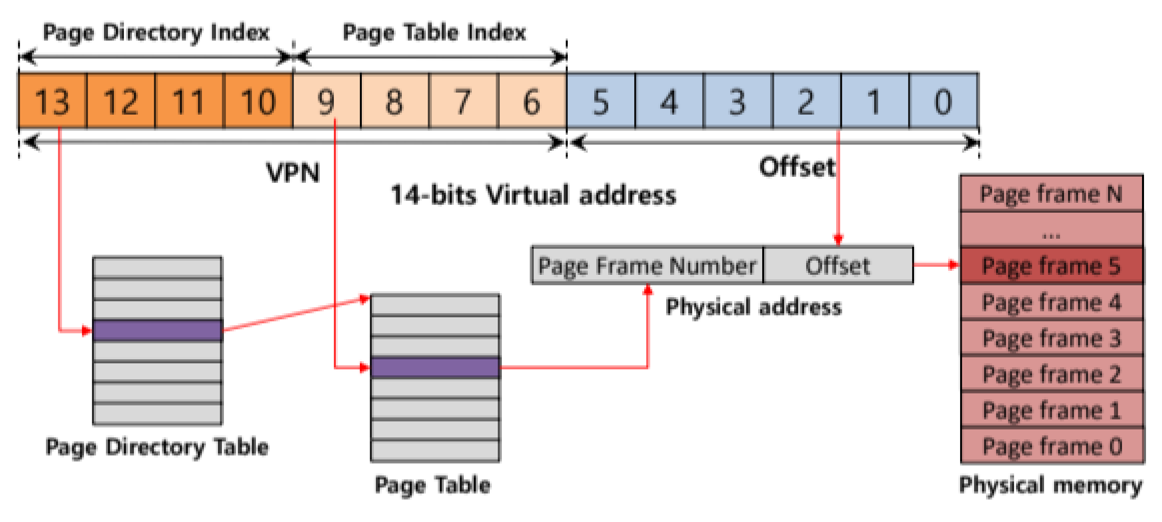


Valid	Tag (VPN)	Value (PTE)
1	0x1000	V R M Prot PFN 0x1234
1	0x2400	V R M Prot PFN 0x8800
0	-	-

  - page directory에 따라 page table을 다시 page로 쪼개보자
    - PTE를 4바이트 가정했고 page 크기가 64바이트 이므로 한 페이지당 16개의 PTE를 넣을 수 있음
    - PTE가 16개만 있으면 되므로 이 페이지들을 나타내는 VPN은 4비트면 충분함



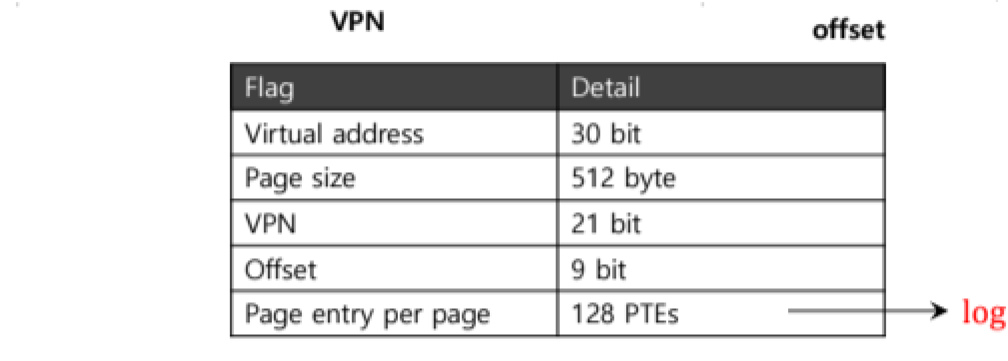
- 256개의 page들을 page directory로 변환했음
  - page table을 page로 쪼갰으므로 page table에 16개의 페이지가 존재
  - page table의 한 페이지 당 16개의 PTE가 존재
  - VPN의 상위 4비트를 page directory index로 사용하고 하위 4비트를 page table index로 사용하게 됨

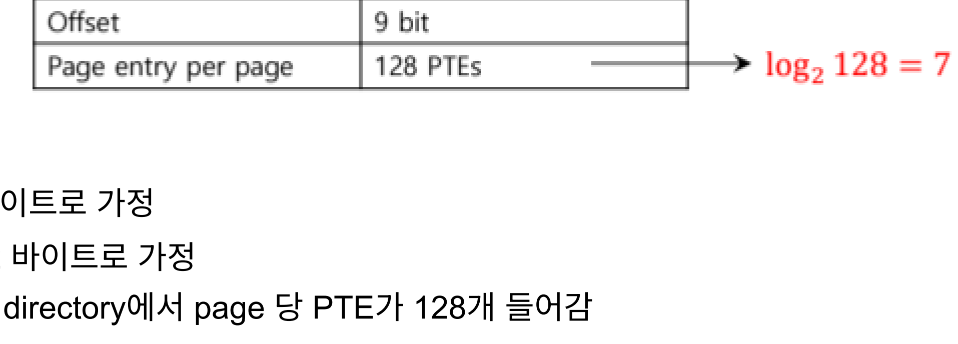


  - 현재 그림에선 page table이 page 크기로 나뉜 상태
  - 그래서 그림으로만 보면 page directory table이 page table이랑 1:1 매칭된다고 보면 됨
  - 실제로 page table은 256개의 PTE로 이루어져 있음

more than two level

- 지금까지 multi-level page table은 page 수가 적어 level 1으로 충분했음
- 만약 page 수가 더 많으면 level 1으로 부족함
- 다음과 같은 경우를 보자



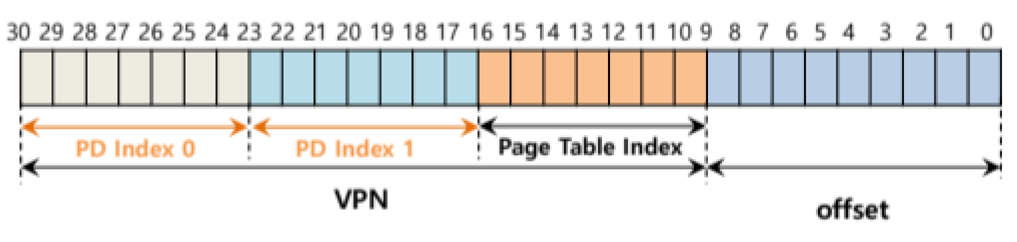


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

$\log_2 128 = 7$

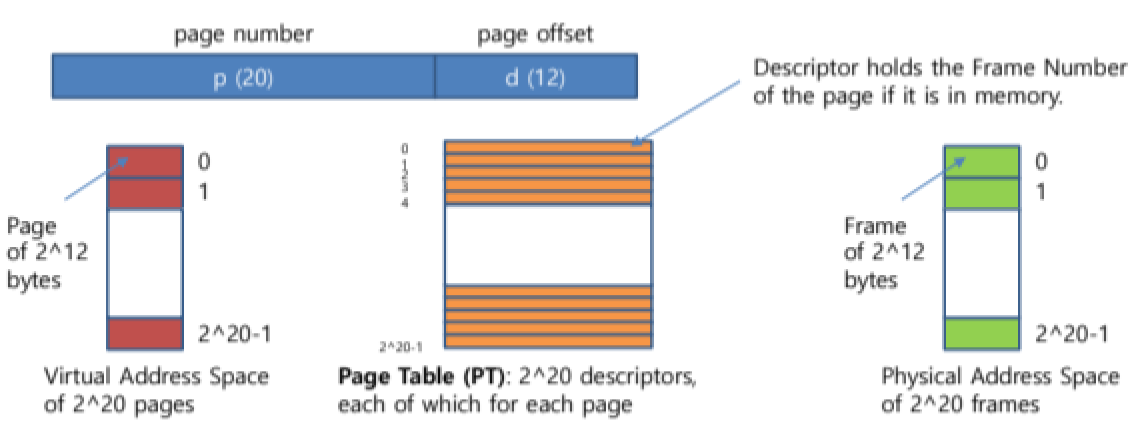
- PTE는 그대로 4바이트로 가정
- 페이지 크기를 512 바이트로 가정
  - 이론상 page directory에서 page 당 PTE가 128개 들어감
  - page directory의 한 page는 page table을 의미
    - page directory는 linear page table를 다시 page로 나눈 것. 그래서 page는 원래 linear page table의 일부임.
    => page table index를 표현하는 데 7 bit가 필요
- 총 페이지 개수는 2^21 개 (= VPN)
  - PTE의 개수도 2^21개
- linear page table의 page 개수는 2^21 / 2^7 = 2 ^ 14
  - page directory로 구현하면 page directory의 page는 2^14 개
    => page directory index를 표현하는 데 14 bit 필요
- 여기서 문제가 있음
  - page directory index가 14 bit 이므로 이론 상 page directory에 들어갈 수 있는 entry가 2^14개임
  - 근데 한 entry 당 사이즈가 4바이트인데 그럼 page directory의 크기가 64kB가 됨

- 한 페이지 사이즈를 512 바이트로 가정했으므로 page directory를 표현하는 데 필요하는 페이지가 128개가 필요해짐...
    - 2^30을 2^14로 표현한 거니 싸게 먹힌거긴 함
- 상위 레벨의 트리 구조가 필요

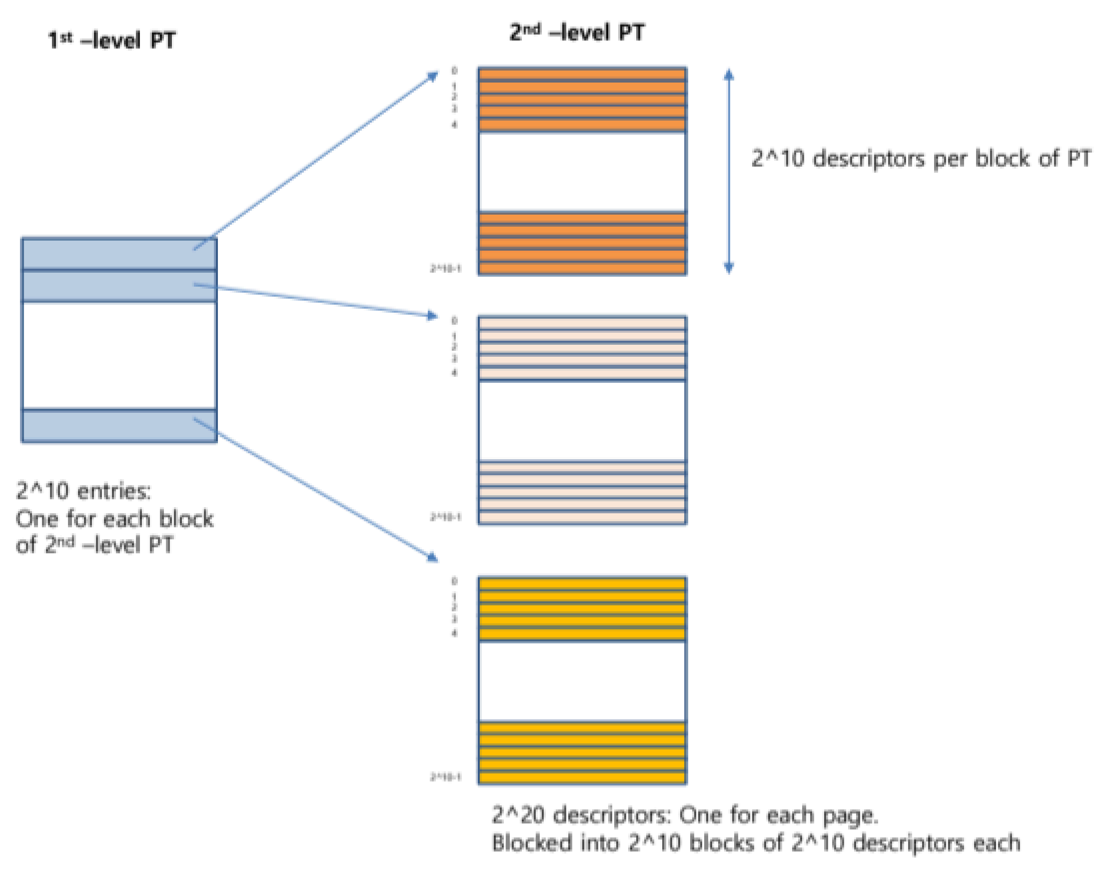


Two-Level Page table scheme Example

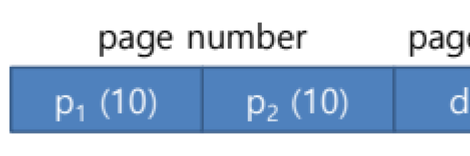
- 주소공간비트가 32bit, 페이지 크기는 4K
  - offset은 12bit, frame은 2^20개
  - page table의 entry 수는 2^20개



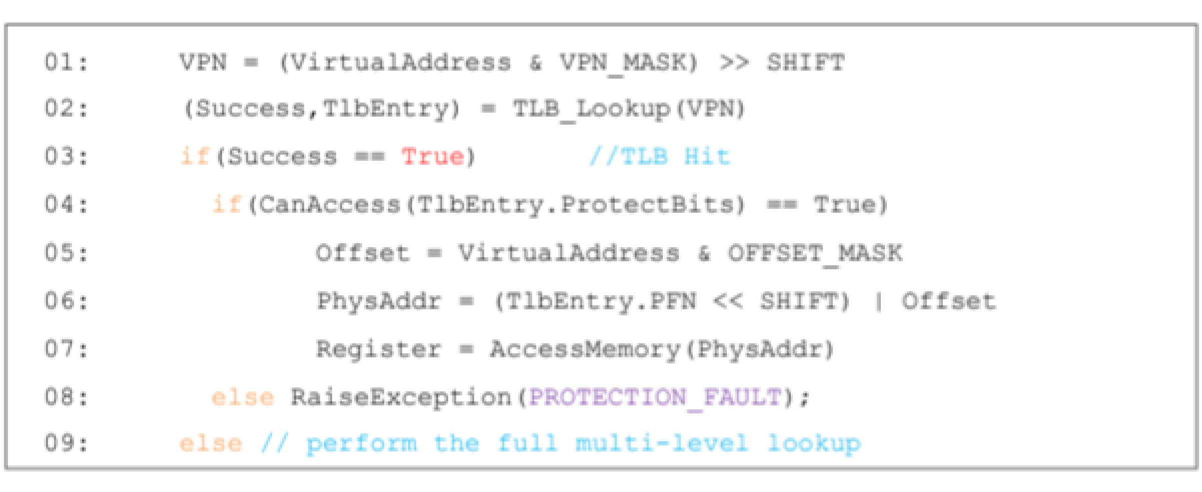
- 여기서 1st level PT가 page directory임
- 이 page table을 다시 page로 나누면 entry는 4B, 한 페이지는 4KB이므로 한 페이지당 1K의 entry가 들어갈 수 있음



- 한 페이지당 1K의 entry가 들어가고 entry는 총 2^20개 이므로 page directory의 entry는 2^10개

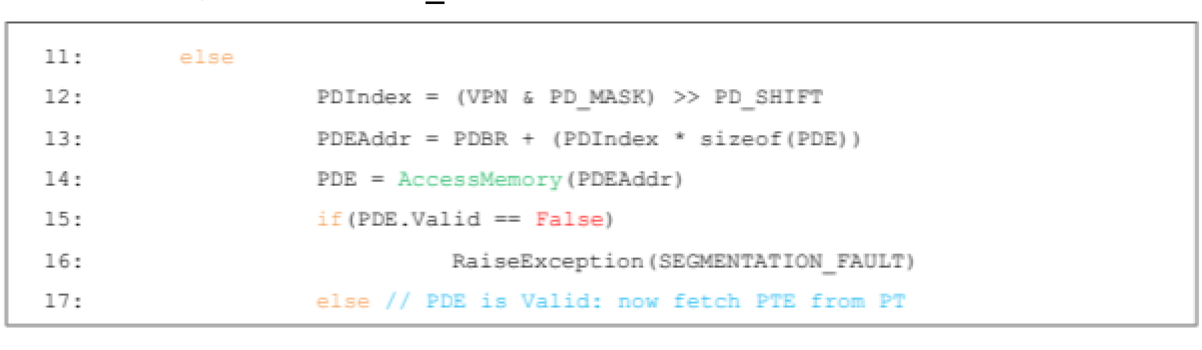


Multi-level Page table control flow



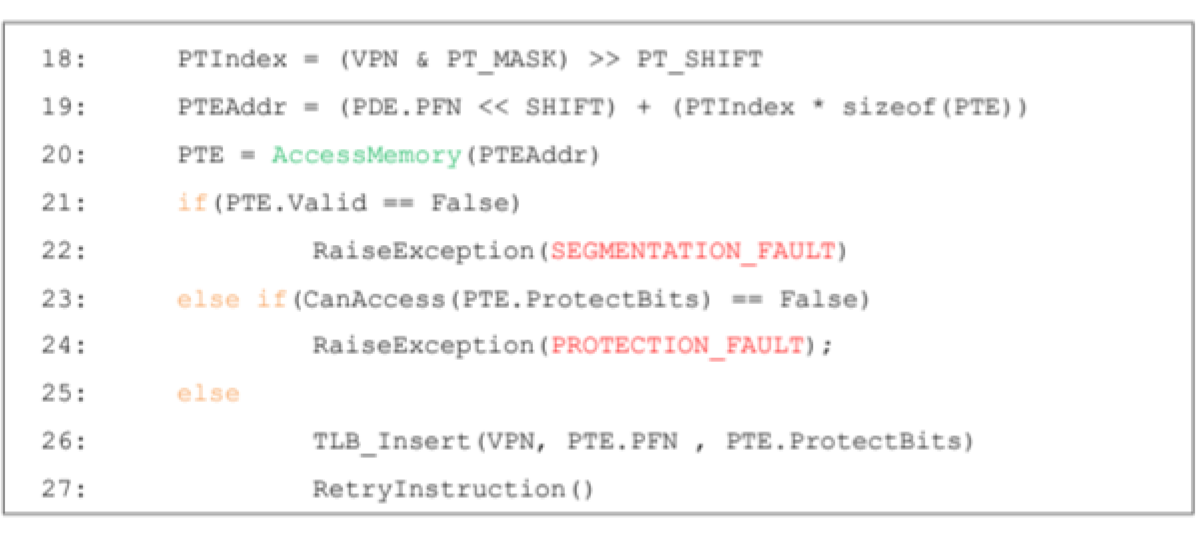
```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success,TlbEntry) = TLB_Lookup(VPN)
if(Success == True)        //TLB Hit
  if(CanAccess(TlbEntry.ProtectBits) == True)
        Offset = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
  else RaiseException(PROTECTION_FAULT);
else // perform the full multi-level lookup
```

- virtualAddress에서 VPN 추출
- TLB에서 VPN을 lookup
- TLB hit 시
  - 접근 가능하면
  - virtualAddress에서 offset 추출
  - TLB에서 꺼내온 entry에서 PFN과 offset을 조합해서 physAddr 생성
  - physAddr에 접근
  - 접근 불가능 시 PROTECTION_FAULT



```
else
        PDIndex = (VPN & PD_MASK) >> PD_SHIFT
        PDEAddr = PDBR + (PDIndex * sizeof(PDE))
        PDE = AccessMemory(PDEAddr)
        if(PDE.Valid == False)
                RaiseException(SEGMENTATION_FAULT)
        else // PDE is Valid: now fetch PTE from PT
```

- TLB miss 시
  - PDIndex 추출
  - PDEAddr 계산
  - PDEAddr에 접근해서 PDE 가져오기
  - PDE가 invalid면
  - SEGMENTATION-FAULT

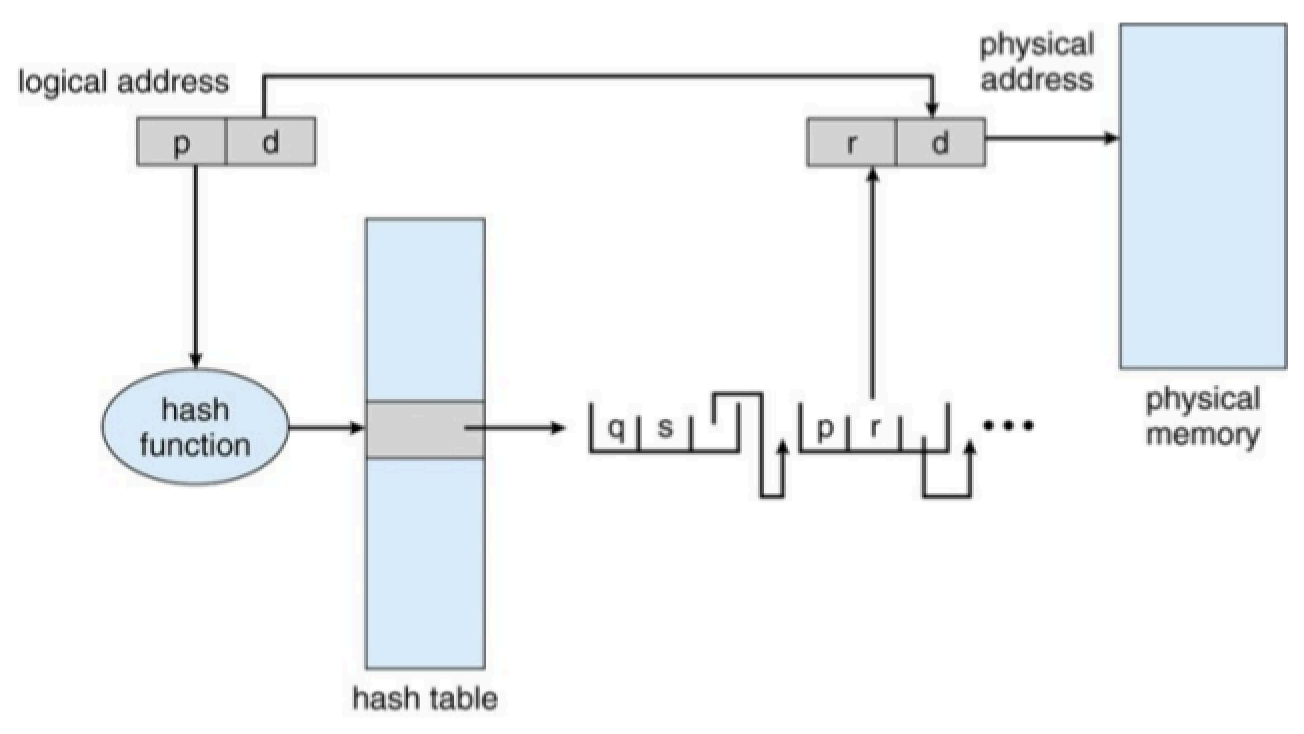


```
PTIndex = (VPN & PT_MASK) >> PT_SHIFT
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
if(PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
else if(CanAccess(PTE.ProtectBits) == False)
        RaiseException(PROTECTION_FAULT);
else
        TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
        RetryInstruction()
```

- PDE가 valid하면
    - PTIndex 계산
    - PTEAddr 계산
    - PTEAddr에 접근해서 PTE 가져오기
    - PTE가 invalid면
        - SEGMENTATION-FAULT
    - PTE에 접근 불가능하면
        - PROTECTION-FAULT
        - PTE를 TLB에 넣기
        - 다시 명령문 실행

Hashed page tables

- address space가 32bit를 넘는 건 흔한 일 (요즘은 64비트)
- VPN이 page table에 hash됨
- 원래 hash 처럼 page table에 chain이 존재
- VPN이 hash함수에 들어가 지정된 page table의 위치에서 chain을 탐색하여 본인에게 맞는 곳을 찾음
- match되면 physical frame을 얻을 수 있음



Inverted page tables

- 64비트에선 각 프로세스마다 페이지 테이블을 가지면 크기가 몇십GB가 될 수 있음
- 메모리의 실제 페이지 당 하나의 엔트리를 갖는 유일 single page를 가짐
- 엔트리엔 어떤 프로세스가 이 페이지를 사용 중인지, 그 페이지의 어떤 virtual page가 이 physical page에 매핑되어있는지에 대한 정보가 담김
- match를 찾기 위해 table을 전부 뒤져봐야 함
- hashing을 사용해서 table의 entry를 몇개만 뒤져봐도 되게 구현

pros & cons

- page table들을 저장할 memory 공간이 감소
    - process 당 page table을 가질 필요가 없음
- 대신 TLB miss 시 table을 탐색하는 데 필요한 시간이 늘어남