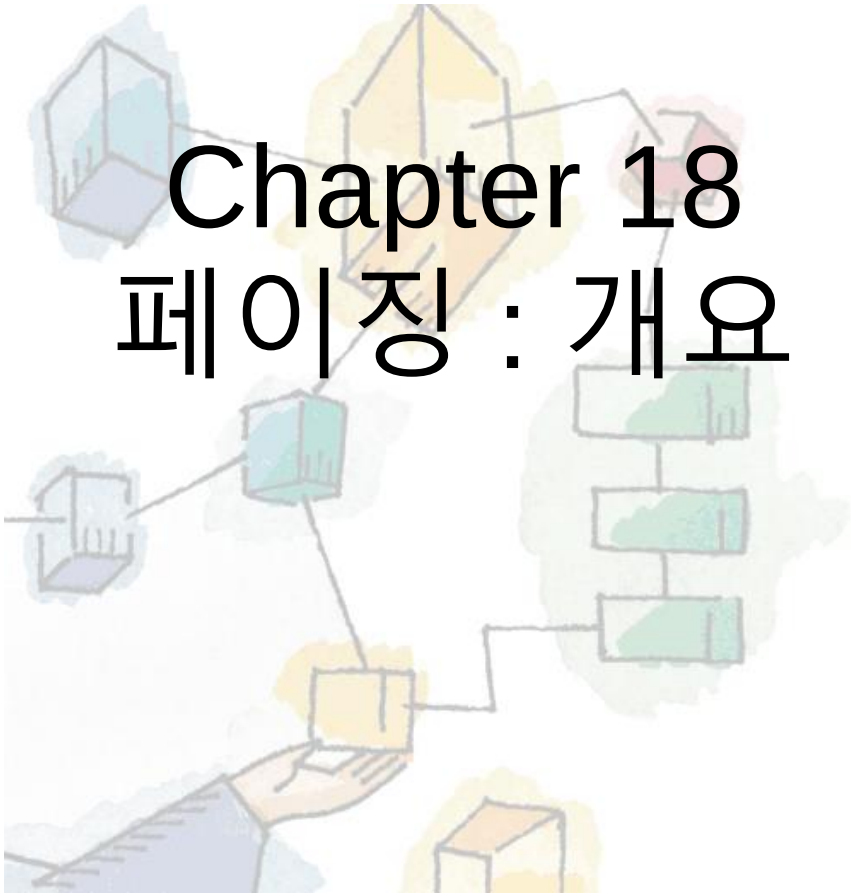
Chapter-18

운영 체제

정내훈

2023년 가을학기
게임공학과
한국공학대학교



Chapter 18
페이징 : 개요

페이징

- 페이징은 주소공간을 **페이지**라고 불리는 **동일 크기**의 조각으로 잘라서 관리한다.
  - 세그멘테이션 : 세그먼트마다 별도의 크기를 갖는다.

- 페이징에서는 물리 메모리도 동일한 크기의 **페이지 프레임**으로 잘라서 관리한다.
  - 페이지의 크기 == 페이지 프레임의 크기

- 가상 주소를 물리 주소로 변환하기 위해 프로세스마다 **페이지 테이블**을 사용한다.

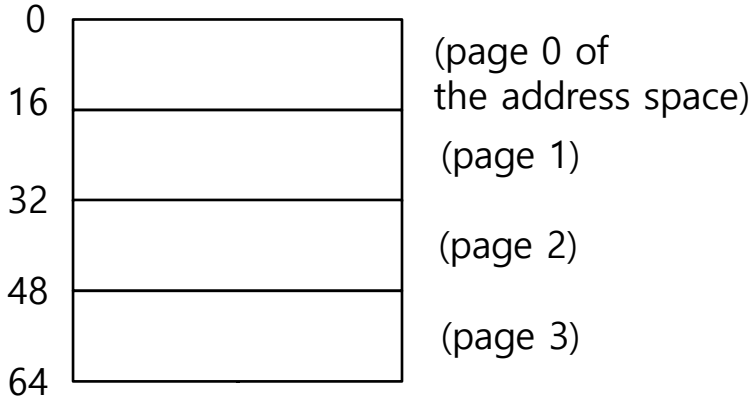
페이징의 장점

- 유연성: 주소 공간의 일관적인 관리
  - Stack 세그먼트 처럼 역방향관리를 따로 HW로 구현할 필요가 없다.

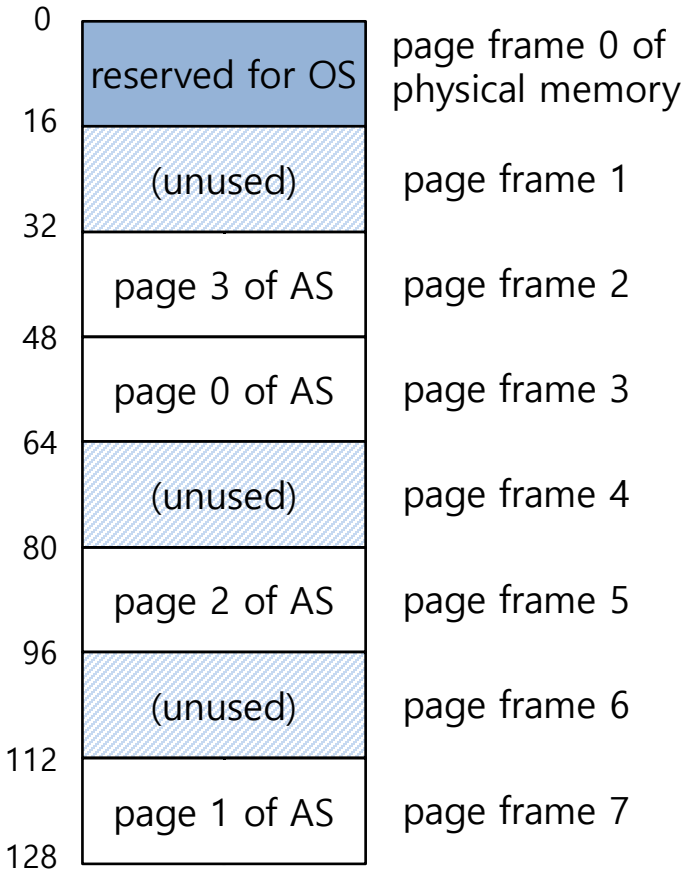
- 단순성: 빈 공간 관리가 쉽다.
  - 페이지 크기와 페이지 프레임의 크기가 같다.
  - 빈 공간 리스트의 모든 조각의 크기가 같다.

간단한 예제

- 16-byte크기의 페이지 프레임을 갖는 128-byte 물리 메모리
- 16-byte크기의 페이지를 갖는 64-byte 주소 공간



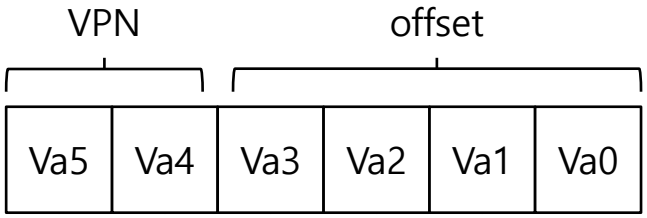
A Simple 64-byte Address Space



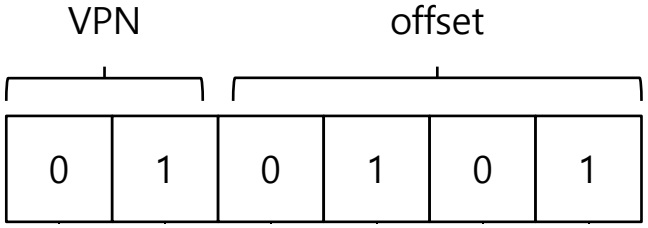
64-Byte Address Space Placed In Physical Memory

주소 변환

- 가상 주소의 분할
  - VPN(Virtual Page Number) : 가상 페이지 번호
  - Offset: 페이지 내에서의 오프셋

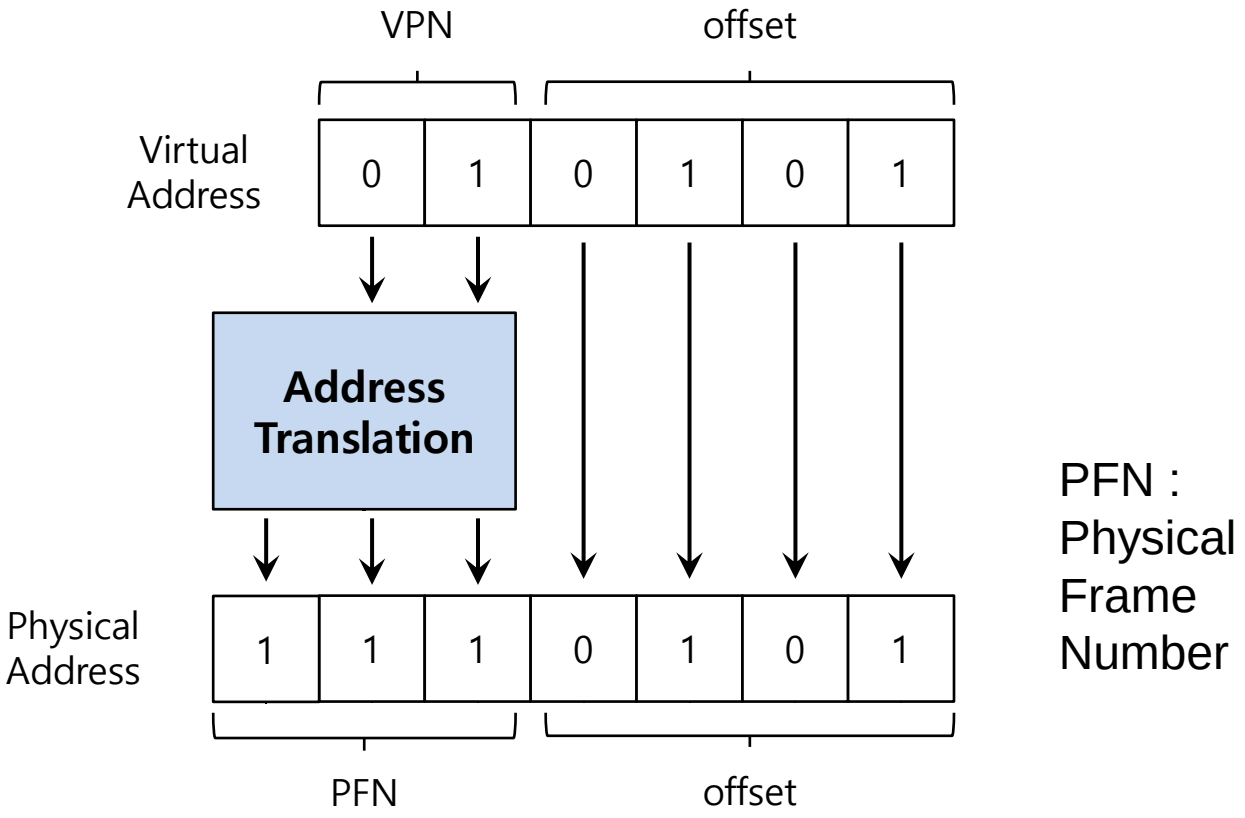


- 예: 64-byte 주소 공간에서 주소 21



예: 주소 변환

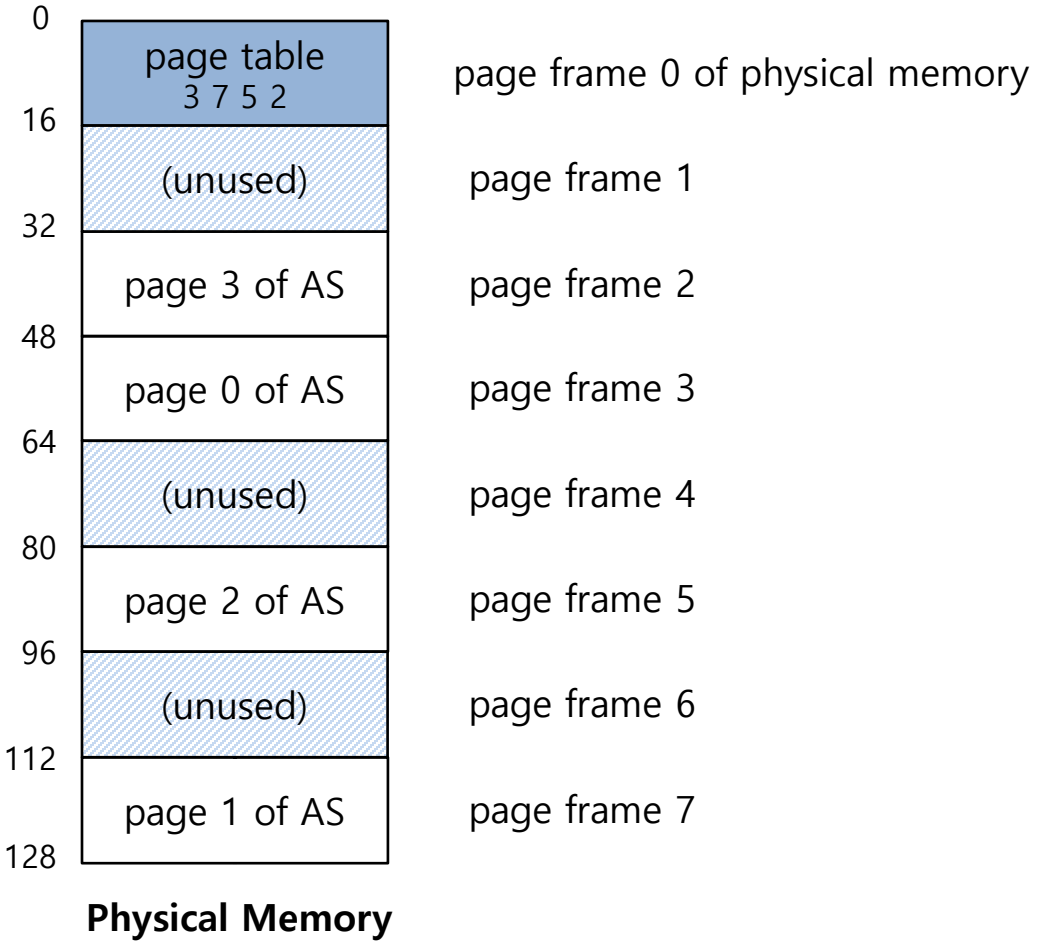
- 64-byte 주소 공간에서 가상 주소 21



페이지 테이블의 위치

- 페이지 테이블의 크기는 엄청날 수 있다.
  - 32-bit 주소 공간에 4-KB 페이지의 경우, VPN으로 20-bits 필요
    - $4MB = 2^{20}\ entries * 4\ Bytes\ per\ page\ table\ entry$
- 각 프로세스마다 별도의 페이지 테이블
  - 메모리에 존재(CPU에 올려 놓기에는 너무 크다.)

예 : 커널 물리메모리의 페이지 테이블



페이지 테이블의 내용

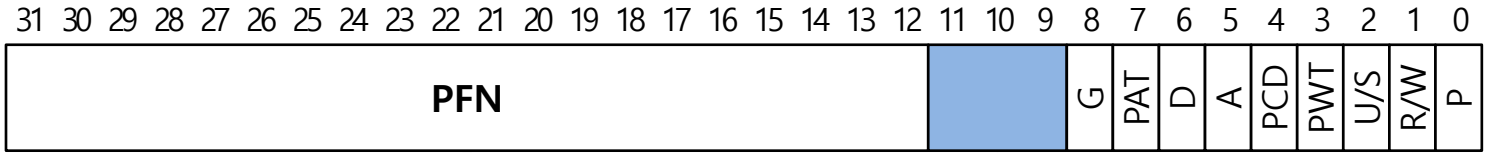
- 가상 주소를 물리 주소로 변환하기 위한 **자료 구조**.
  - 단순화 하면: 선형 페이지테이블, 배열(물리주소의)

- OS는 VPN을 **인덱스**로 페이지 테이블을 읽고 쓴다.

PTE의 추가 정보 비트들

- **Valid Bit**: 해당되는 가상 주소가 올바른가.
  - 할당이 되어 있는가?
- **Protection Bit**: 읽어도 되는가? 써도 되는가? 실행해도 되는가?
- **Present Bit**: 이 페이지가 물리 메모리에 존재하는가? 아니면 디스크에 존재하는가?
- **Dirty Bit**: 메모리에 생성된 이후 한번이라도 내용이 변경되었는가?
- **Reference Bit(Accessed Bit): CPU**가 한번이라도 참조(읽기 또는 쓰기 또는 실행)한 적이 있는가?

예: x86 Page Table Entry



An x86 Page Table Entry(PTE)

- P: present
- R/W: read/write bit
- U/S: supervisor
- A: accessed bit
- D: dirty bit
- PFN: the page frame number

페이징: 너무 느림

- 해당 PTE를 얻기 위해서는 페이지 테이블의 시작 위치를 알아야 한다.
  - CPU에 시작위치를 갖고 있는 레지스터가 있고, 운영체제가 초기화 한다.

- 모든 메모리 접근에서 HW는 추가로 메모리에 접근해야 한다.
  - 페이지 테이블 접근

페이징에서 메모리 접근(1/2)

```
1       // Extract the VPN from the virtual address
2       VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4       // Form the address of the page-table entry (PTE)
5       PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7       // Fetch the PTE
8       PTE = AccessMemory(PTEAddr)
9
```

페이징에서 메모리 접근(2/2)

```
10      // Check if process can access the page
11      if (PTE.Valid == False)
12              RaiseException(PAGE_FAULT)
13      else if (CanAccess(PTE.ProtectBits) == False)
14              RaiseException(PROTECTION_FAULT)
15      else
16              // Access is OK: form physical address and fetch it
17              offset = VirtualAddress & OFFSET_MASK
18              PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19              Register = AccessMemory(PhysAddr)
```

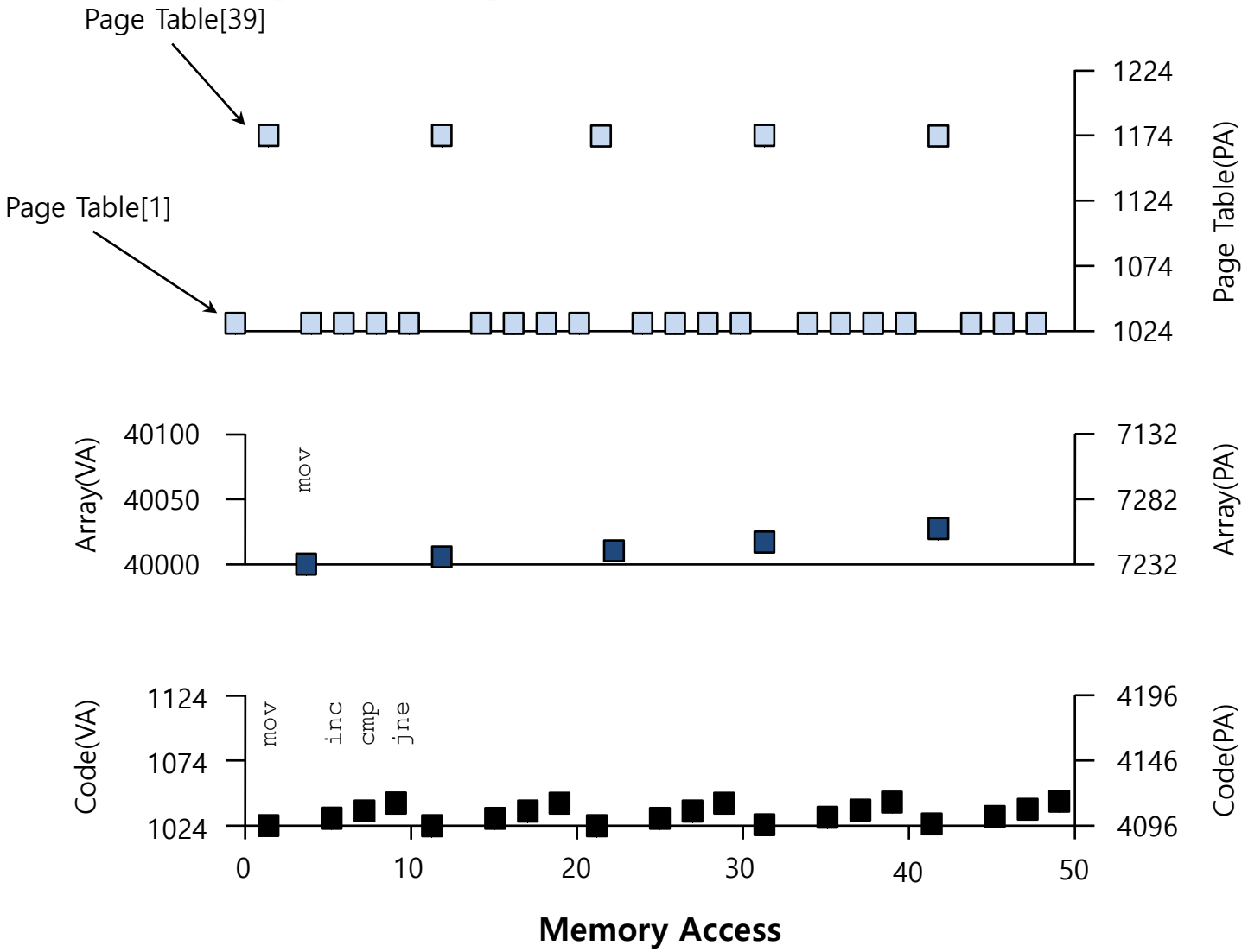
메모리 트레이스

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

```
prompt> gcc –o array array.c –Wall –o
prompt>./array
```

```
0x1024 mov [edi+eax*4],0
0x1028 inc eax
0x102c cmp 0x03e8,eax //0000 0011 1110 1000₂ = 1000₁₀
0x1030 jne 0x1024
```

가상(물리) 메모리 트레이스



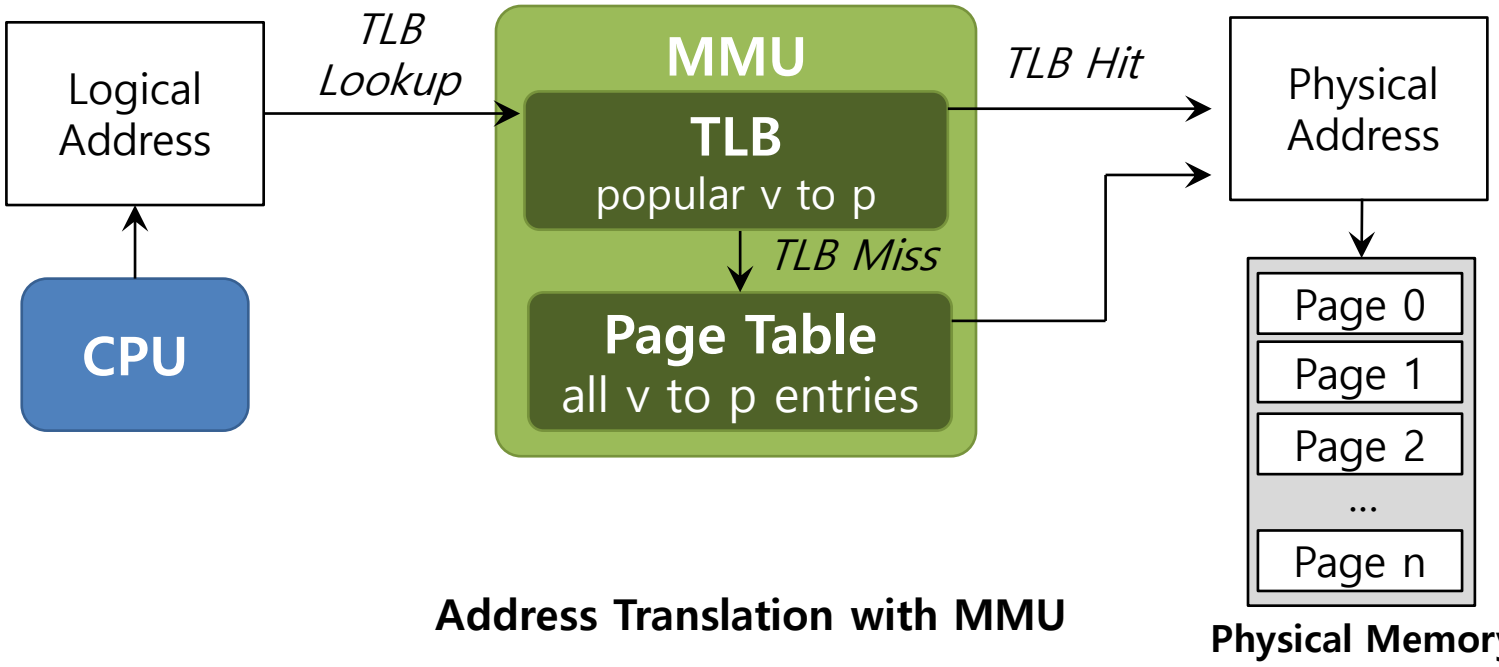
페이징 요약

- 장점
  - 외부 단편화 없음
  - 작은 내부 단편화
    - 최대 : 논리적 세그먼트 개수 * (페이지 크기 – 1)
- 단점
  - 페이지 테이블로 인한 잦은 메모리 접근

19. 페이징: 더 빠른 변환(TLB)

TLB(Translation Look-Aside Buffer)

- CPU의 memory-management unit(MMU) 메모리 관리 장치의 일부분.
- 자주 사용되는 가상 주소의 가상-실제 메모리 변환 하드웨어 캐시



Address Translation with MMU

TLB 기본 알고리즘

```
1: VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
2: (Success , TlbEntry) = TLB_Lookup(VPN)
3:    if(Success == TRUE){ // TLB Hit
4:       if(CanAccess(TlbEntry.ProtectBit) == True ){
5:           offset = VirtualAddress & OFFSET_MASK
6:           PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:           AccessMemory( PhysAddr )
8:       }else RaiseException(PROTECTION_ERROR)
```

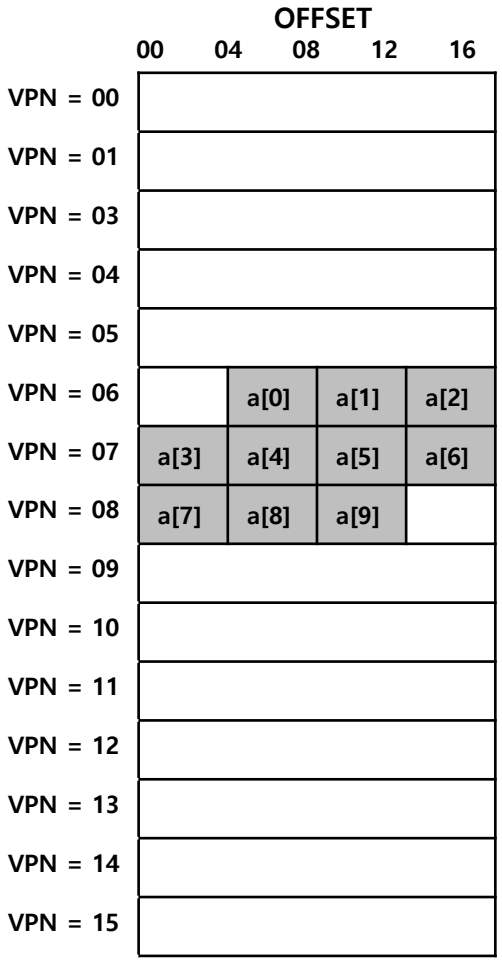
TLB 기본 알고리즘

```
11:     }else{ //TLB Miss
12:         PTEAddr = PTBR + (VPN * sizeof(PTE))
13:         PTE = AccessMemory(PTEAddr)
14:         if(PTE.Valid == False)
               RaiseException SEGFAULT) ;
15:         else{
16:            TLB_Insert( VPN , PTE.PFN , PTE.ProtectBits)
17:            RetryInstruction()
18:        }
19:    }
```

- (12-13 lines) 하드웨어가 변환을 위해서 페이지 테이블 참조
- (16 lines) 찾은 변환으로 TLB 갱신.

예: 배열 접근

- TLB로 인한 성능 향상



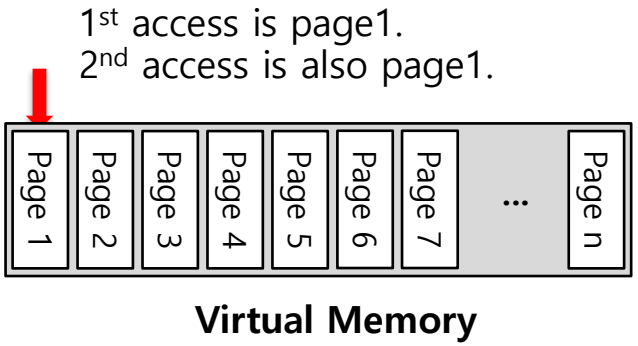

```
0:    int sum = 0 ;
1:    for( i=0; i<10; i++){
2:          sum+=a[i];
3:    }
```

The TLB improves performance due to spatial locality

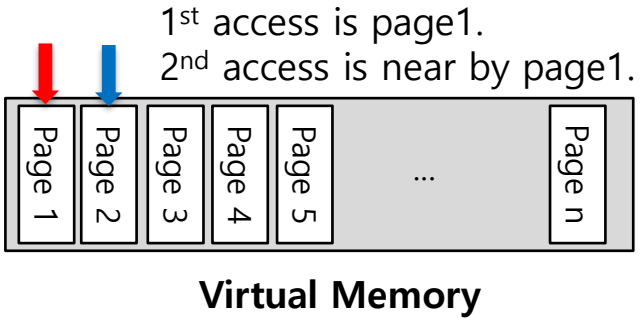
3 misses and 7 hits.
Thus TLB hit rate is 70%.

지역성(Locality)

- 시간 지역성
  - 사용한 명령어나 데이터는 곧 다시 사용할 확률이 크다.



- 공간 지역성
  - 프로그램이 x 주소의 메모리를 사용했다면 얼마 안 지나서 x 근처의 메모리를 사용할 확률이 크다.



TLB 미스는 누가 처리할까?

- CISC에서는 HW가 전부 처리.
  - HW가 페이지테이블의 위치를 정확히 알고 있다.
    - PTBR(Page Table Base Register)
  - HW가 페이지테이블을 “검색“ 해서 해당 PTE를 찾아서, TLB에 업데이트하고 해당명령어를 다시 실행한다.
  - **hardware-managed TLB.**
  - **Intel x86**

TLB 미스는 누가 처리할까?

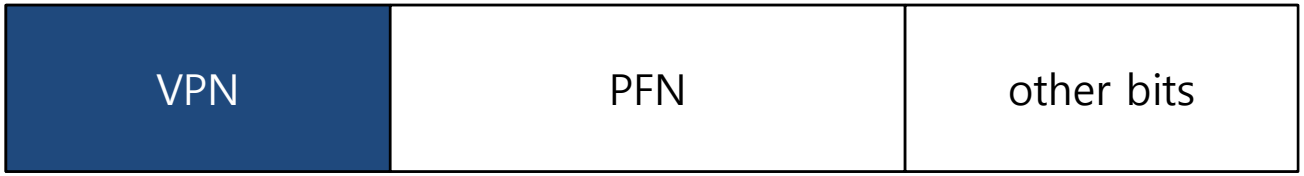
- RISC 에서는 **소프트웨어-관리 TLB** 방식 사용
  - TLB 미스가 발생하면 하드웨어 예외처리를 발생시킨다. (trap handler).
    - OS에 있는 **Trap handler 프로그램은** TLB miss를 처리한 후 미스를 발생시킨 명령어를 재실행 한다.

RISC – TLB HW

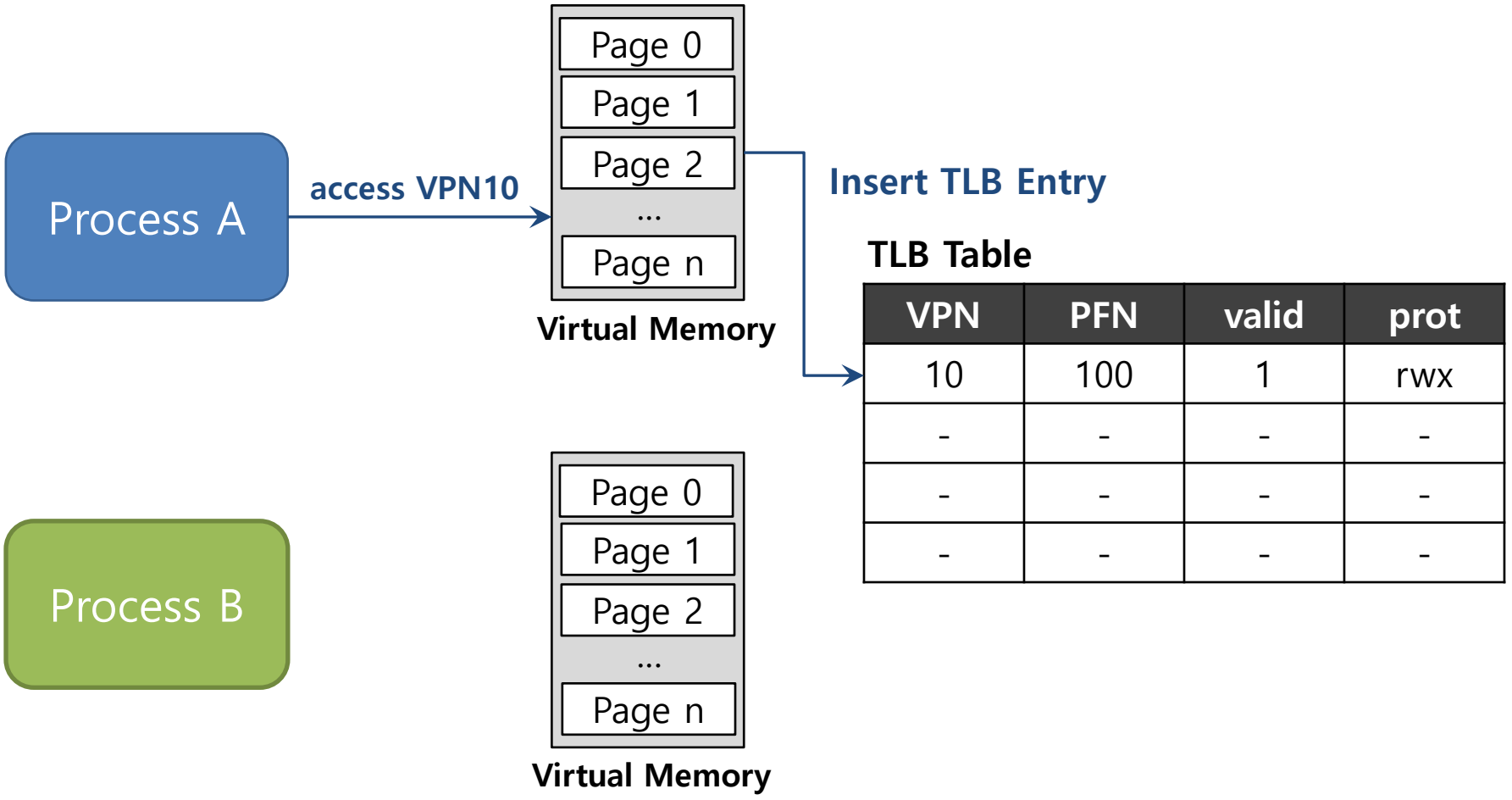
```
1:      VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2:      (Success, TlbEntry) = TLB_Lookup(VPN)
3:      if (Success == True) // TLB Hit
4:          if (CanAccess(TlbEntry.ProtectBits) == True)
5:              Offset = VirtualAddress & OFFSET_MASK
6:              PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:              Register = AccessMemory(PhysAddr)
8:          else
9:              RaiseException(PROTECTION_FAULT)
10:     else // TLB Miss
11:             RaiseException(TLB_MISS)
```

TLB의 구성

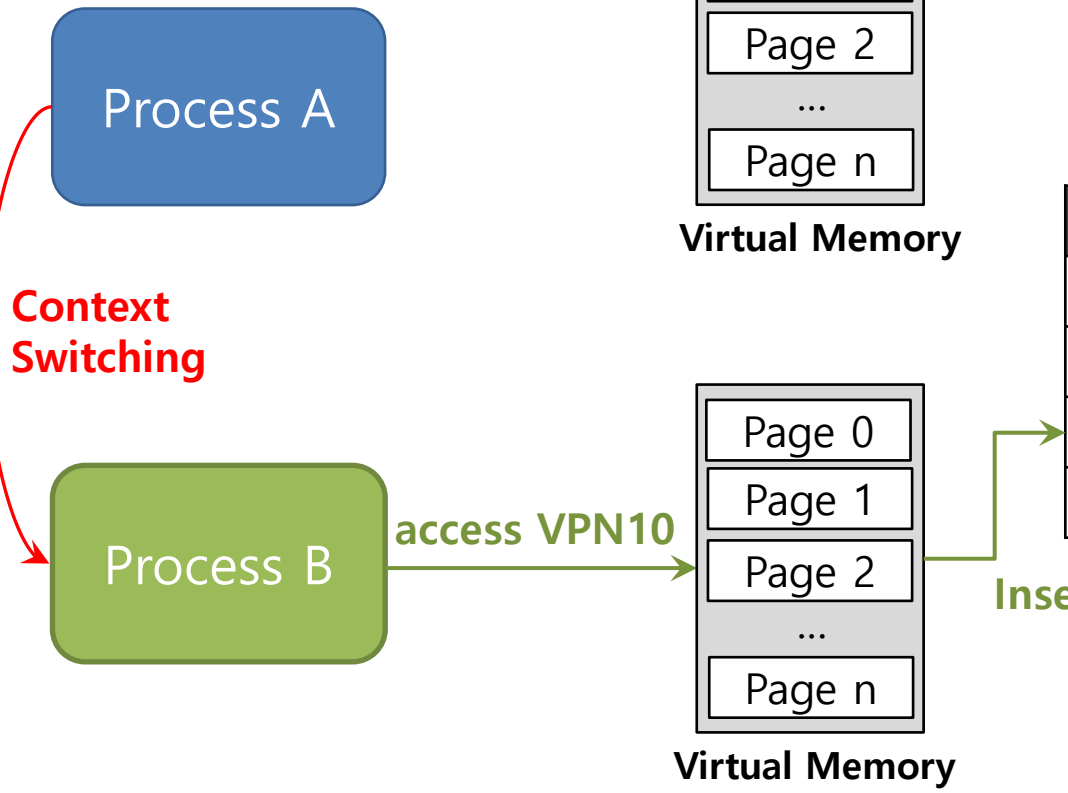
- TLB는 완전 연관(**Full Associative)** 방식으로 관리된다.
  - TLB는 일반적으로 32,64, 또는 128개의 원소를 갖는다.
  - TLB 검색은 하드웨어로 구현되며 병렬 검색을 통해 속도 저하를 막는다.
  - 여러 정보 비트(other bits)들을 갖는다: valid bits(TLB 유효, Page 유효) , protection bits, address-space identifier, dirty bit



TLB Issue: Context Switching



TLB Issue: Context Switching



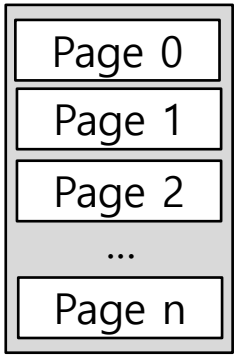
TLB Table

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

Insert TLB Entry

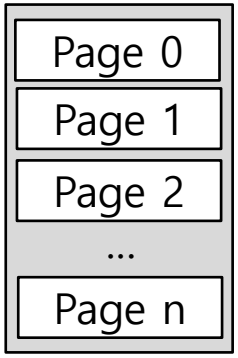
TLB Issue: Context Switching

Process A



Virtual Memory

Process B



Virtual Memory

TLB Table

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

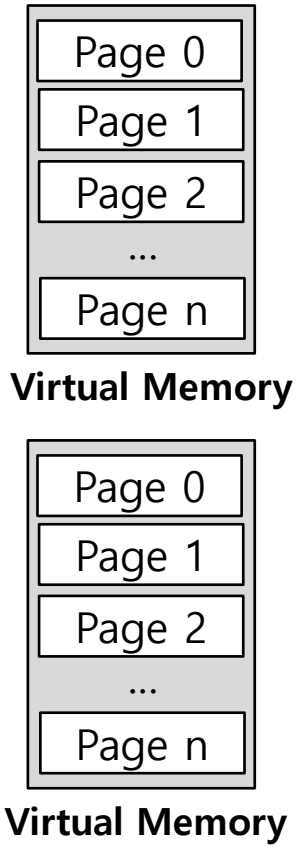
원소가 어떤 프로세스에 해당하는 지 구분이 불가능하다.

To Solve Problem

- 주소 공간 ID를 사용하여 구분 한다.(address space identifier, ASID)

Process A

Process B



TLB Table

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

- **또는 모든 valid를 0로 한다.**

Another Case

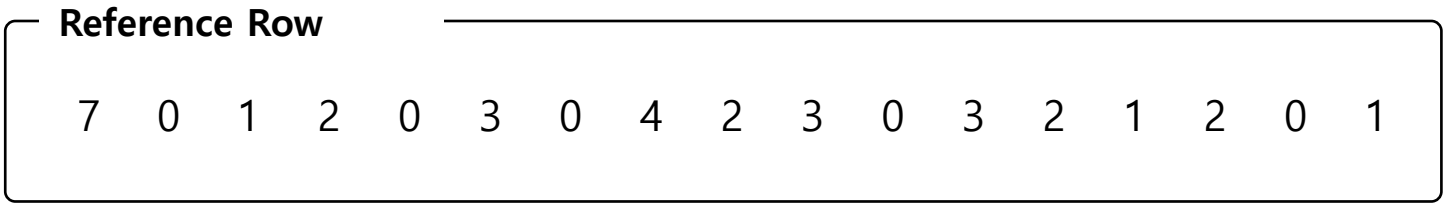
- 2개의 프로세스가 페이지를 공유.
  - 프로세스 1이 101번 물리 페이지를 프로세스 2와 공유.
  - P1이 그 페이지를 주소공간 10번째 페이지에 매핑
  - P2이 그 페이지를 주소공간 50번째 페이지에 매핑

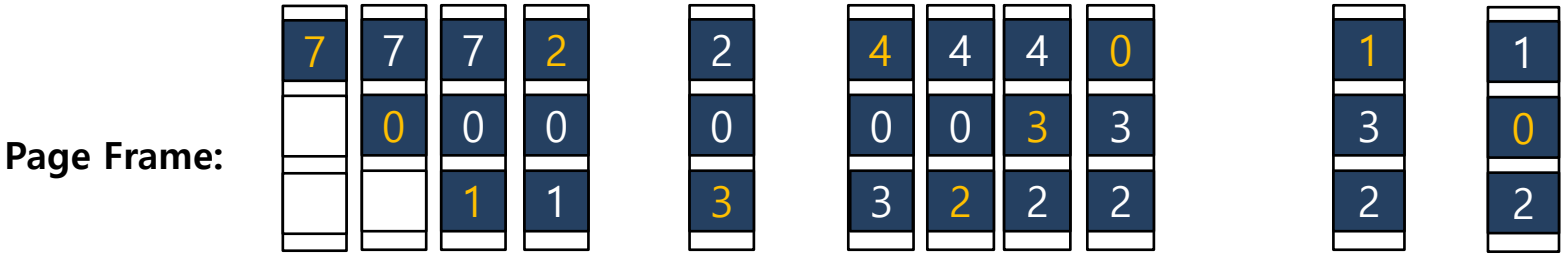
VPN	PFN	valid	prot	ASID
10	101	1	rwx	1
-	-	-	-	-
50	101	1	rwx	2
-	-	-	-	-

페이지 공유는 사용 중인 물리 페이지를 절약할 수 있기 때문에 바람직하다.

TLB 교체 정책

- LRU(Least Recently Used)
  - 사용한지 오래된 엔트리를 방출한다.
  - 지역성(locality)을 이용한다.

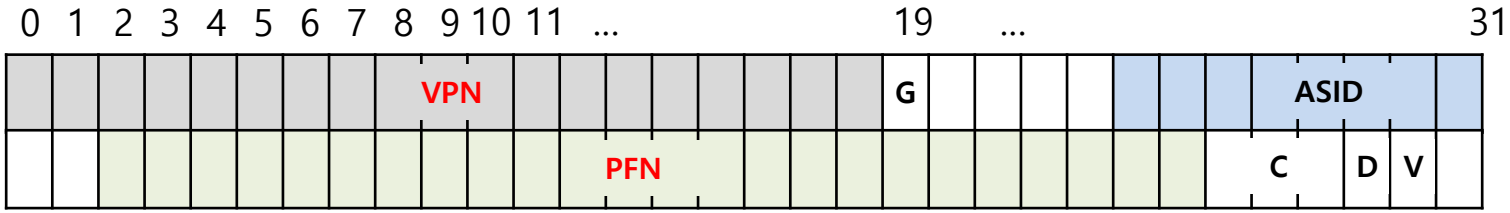




Total 11 TLB miss

A Real TLB Entry

All 64 bits of this TLB entry(example of MIPS R4000)

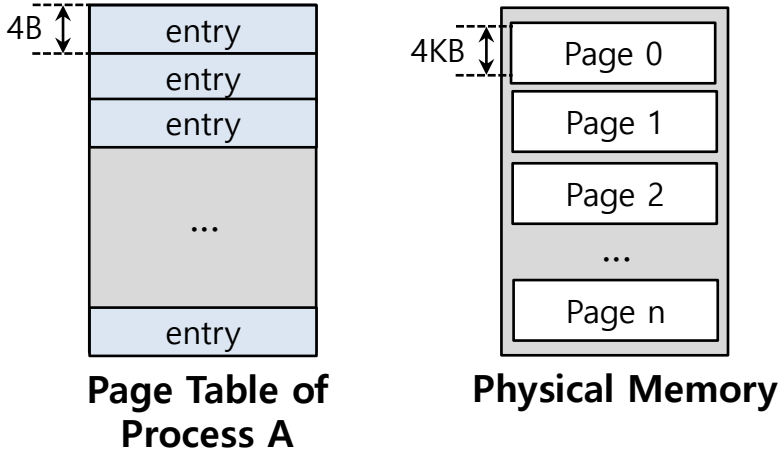


Flag	Content
19-bit VPN	The rest reserved for the kernel.
24-bit PFN	Systems can support with up to 64GB of main memory(pages).
Global bit(G)	Used for pages that are globally-shared among processes.
ASID	OS can use to distinguish between address spaces.
Coherence bit(C)	determine how a page is cached by the hardware.
Dirty bit(D)	marking when the page has been written.
Valid bit(V)	tells the hardware if there is a valid translation present in the entry.

20. Advanced Page Tables

Paging: 선형 테이블

- 모든 프로세스당 한 개씩의 페이지 테이블이 필요하다..
  - 32비트 주소공간에서 4KB크기의 페이지를 사용하고 페이지 테이블 엔트리로 4 바이트를 사용한다면.

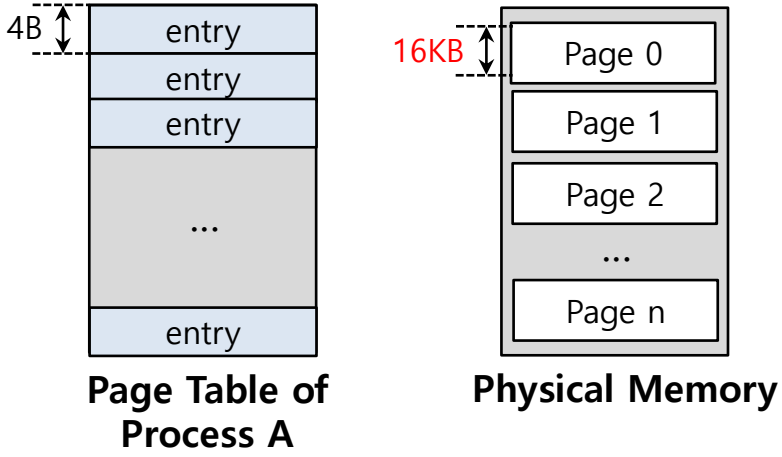


$$\text{Page table size} = \frac{2^{32}}{2^{12}} * 4Byte = 4MByte$$

페이지 테이블은 너무 커서 너무 많은 메모리를 차지한다.

Paging: 테이블 크기 줄이기

- 페이지의 크기를 키울 경우.
  - 32비트 주소공간에 16KB 크기의 페이지이고 페이지 테이블 엔트리로 4바이트를 사용할 경우.

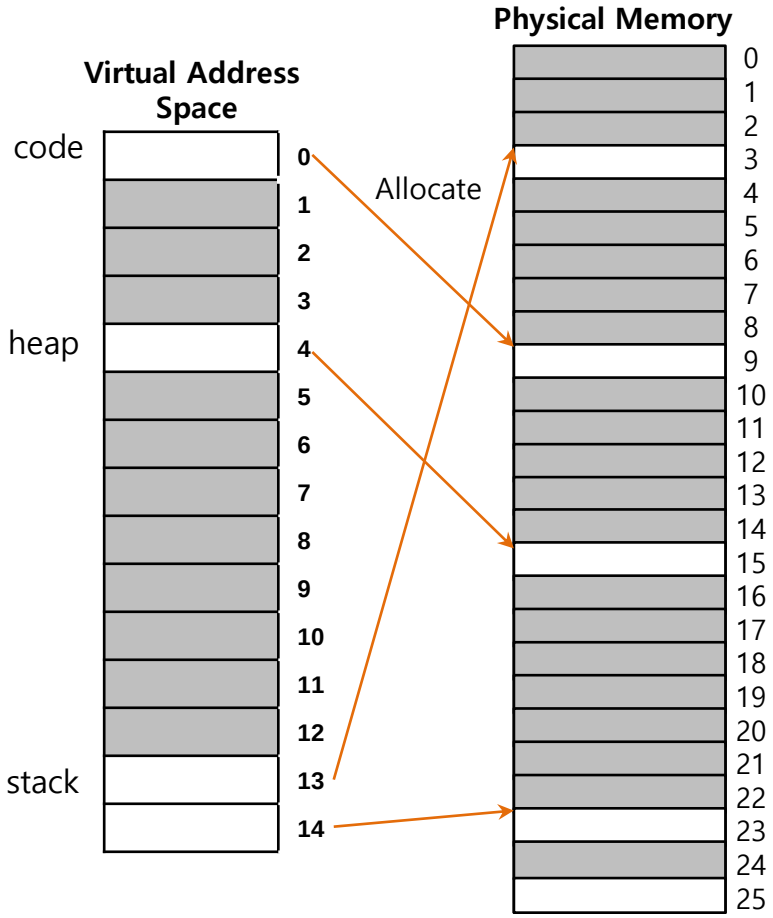


$\frac{2^{32}}{2^{14}} * 4 = 1MB$ per page table

페이지가 커지면 내부 단편이 커진다.

문제점

- 프로세스의 주소공간을 하나의 페이지테이블로 관리할 경우.



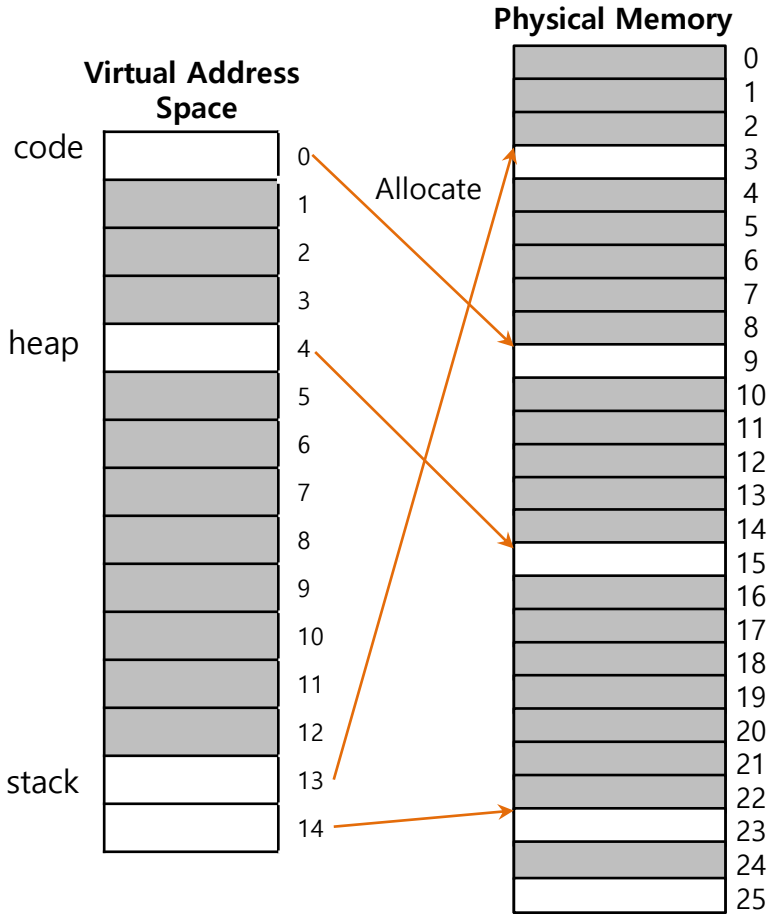
A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
10	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

문제점

- 페이지 테이블의 대부분이 미사용 공간이다.



A 16KB Address Space with 1KB Pages

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	-	-	-
-	0	-	-	-
-	0	-	-	-
15	1	rw-	1	1
...	...	...	...	...
-	0	-	-	-
3	1	rw-	1	1
23	1	rw-	1	1

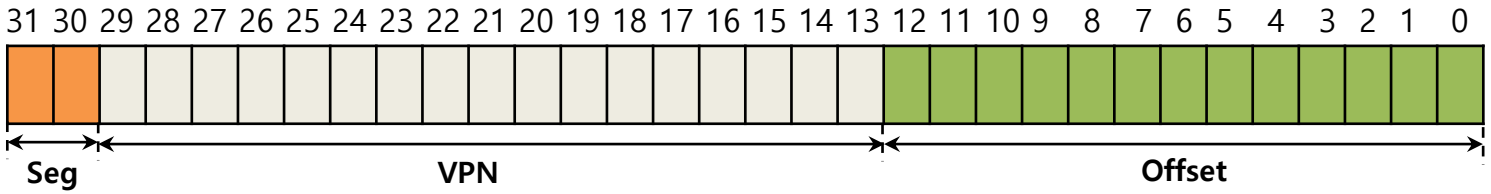
A Page Table For 16KB Address Space

하이브리드 접근법: 페이징과 세그먼트

- 페이지 테이블로 인한 메모리 부담을 줄이자.
  - 세그멘트를 사용하지만, base 주소가 실제 세그먼트의 시작 주소가 아니라 세그먼트의 페이지 테이블의 주소이다.
  - 바운드 레지스터는 페이지 테이블의 끝 주소를 갖는다.

하이브리드 방식의 예

- 모든 프로세스는 3개의 세그먼트를 갖고 있고 이에 해당하는 3개의 페이지 테이블을 갖는다.
  - 프로세스는 3개의 세그먼트 베이스 레지스터가 있고, 그 레지스터는 해당 세그먼트의 페이지테이블의 물리 시작 주소를 갖는다.



32-bit Virtual address space with 4KB pages

Seg value	Content
00	unused segment
01	code
10	heap
11	stack

하이브리드 방식에서의 TLB 미스

- HW는 **물리 주소**를 **페이지 테이블**에서 얻는다.
  - HW는 세그먼트 비트(SN)를 통해 어느 베이스 레지스터를 사용할지 결정한다.
  - 베이스 레지스터에서 읽은 물리 주소와 VPN을 통해 페이지 테이블 원소(PTE)의 주소를 합성한다.

```
01:    SN = (VirtualAddress & SEG_MASK) >> SN_SHIFT
02:    VPN = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
03:    AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

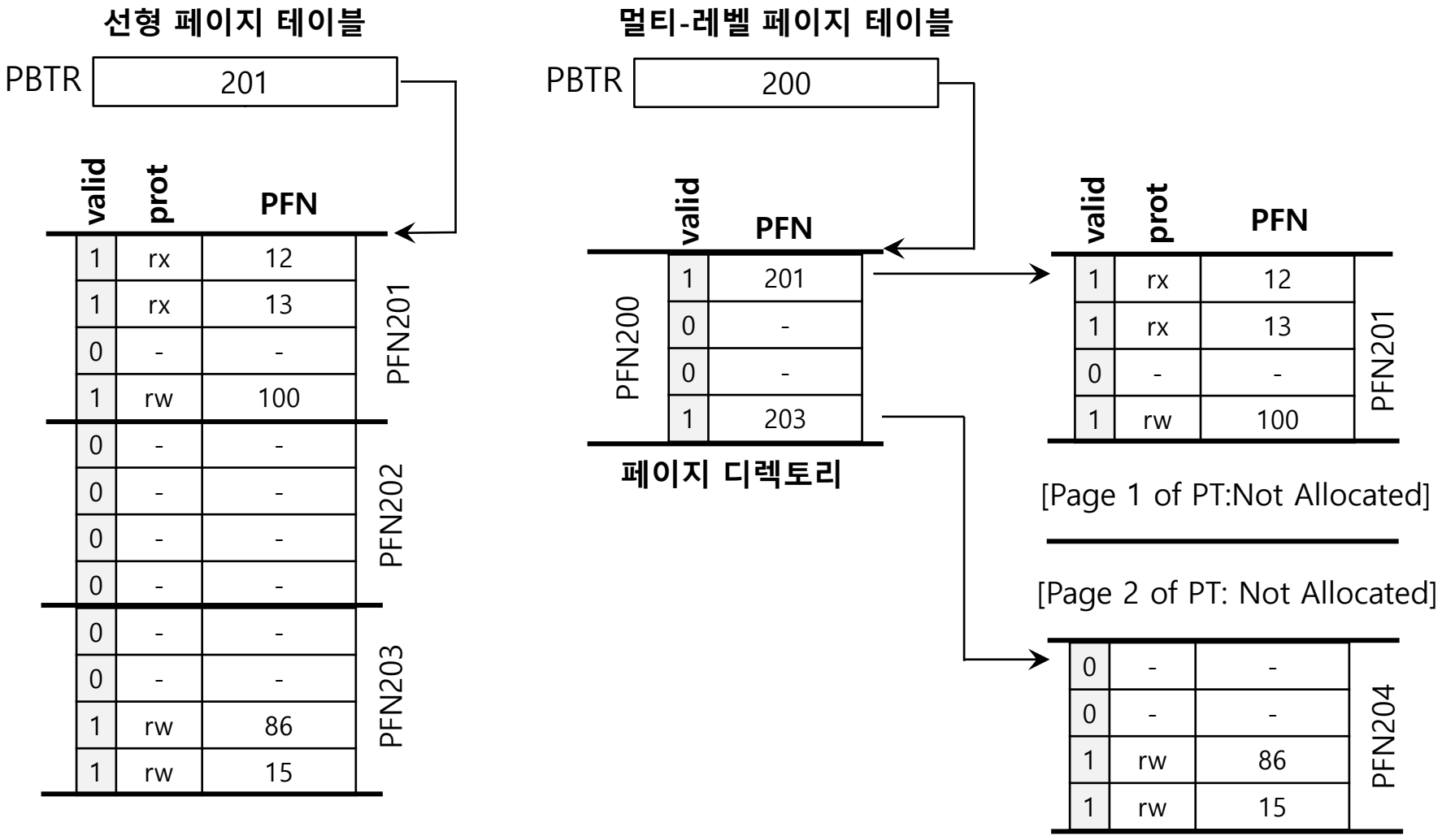
하이브리드 방식의 문제

- 하이브리드 방식에 문제가 없는 것은 아니다.
  - Heap의 중간에 안쓰이는 부분이 많다면, 기존과 같은 페이지 테이블의 낭비가 발생한다.
  - 기존의 외부 단편화가 해결되지 않는다.
    - 페이지 테이블자체가 연속된 메모리 공간 요구

멀티-레벨 페이지 테이블

- 긴 페이지 테이블을 트리형식으로 변경.
  - 페이지 테이블을 **페이지 크기**의 조각으로 자른다.
  - 어떤 조각의 속하는 페이지들이 전부 무효상태이면 그 조각은 비워둔다. (메모리를 할당하지 않는다.)
  - 조각의 유효 여부와 조각의 위치를 알기 위해 페이지 디렉토리라는 새 자료구조를 사용한다.

멀티-레벨 페이지 테이블: 페이지 디렉토리



선형(Left)과 멀티 레벨(Right) 페이지 테이블

멀티-레벨 페이지 테이블 : 페이지 디렉토리 항목

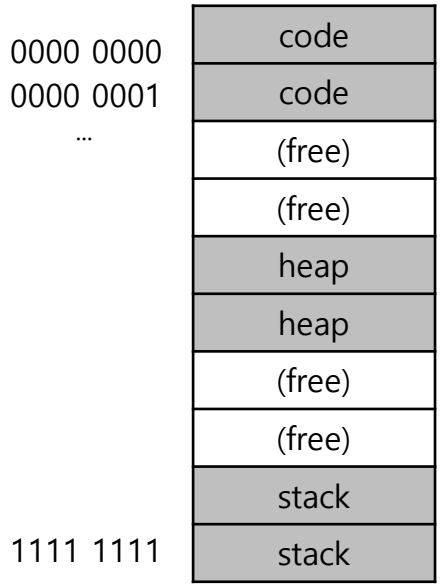
- 페이지 디렉토리 항목(Page Directory Entry, PDE)
- 페이지 디렉토리는 PDE들로 구성되어 있고 하나의 PDE는 페이지 크기의 페이지 테이블 조각 하나를 담당한다.
  - 페이지 디렉토리의 크기 = PDE개수 * PDE크기
  - PDE개수 = 페이지테이블 크기 / 페이지 크기
  - PDE크기 = PTE크기
- PDE도 유효 비트와 PFN를 갖고 있다.

멀티-레벨 페이지 테이블: 장단점

- 장점
  - 페이지 테이블의 크기가 실제 사용하고 있는 메모리 크기에 비례한다. 즉 낭비가 적다
  - 페이지 테이블의 추가, 삭제가 편하다. (연속된 메모리를 요구하지 않는다.)

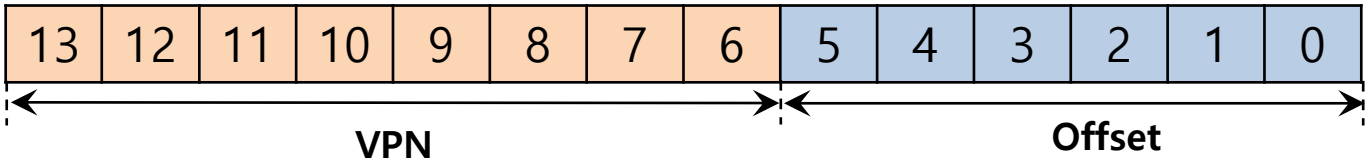
- 단점
  - 성능-공간의 상호절충 (trade-off)이다.
  - 복잡하다.

멀티-레벨 페이지 테이블: 예



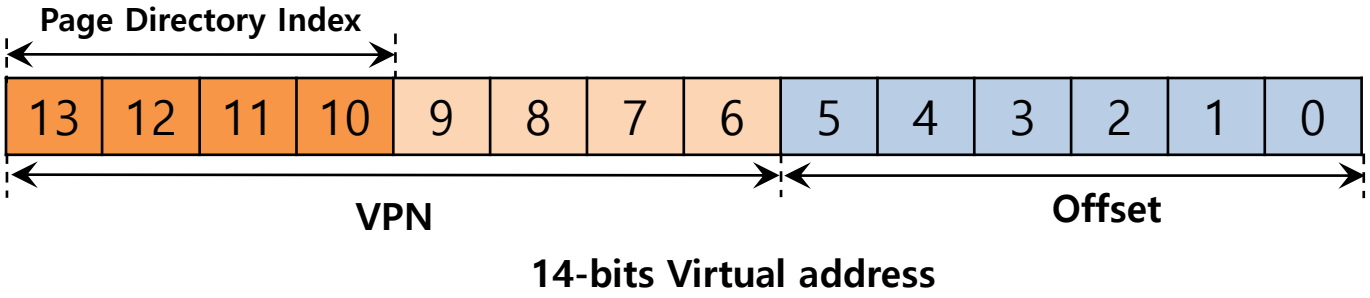
Flag	Detail
Address space	16 KB
Page size	64 byte
Virtual address	14 bit
VPN	8 bit
Offset	6 bit
Page table entry	

A 16-KB Address Space With 64-byte Pages



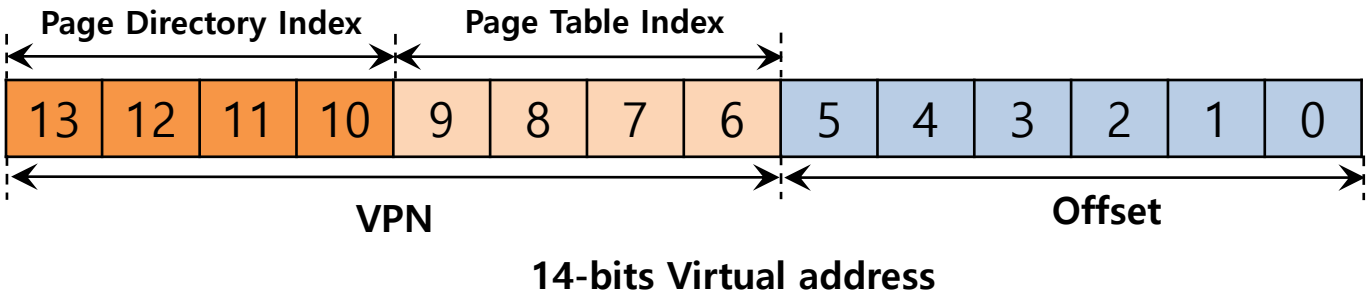
다단계의 예 : 페이지 디렉토리

- 페이지 테이블의 페이지 하나당 하나의 페이지 디렉토리 항목이 필요하다.
  - 페이지 디렉토리당 16개의 페이지 테이블.
- 페이지 디렉트리가 유효(valid)하지 않다면 -> 예외처리(exception) 발생



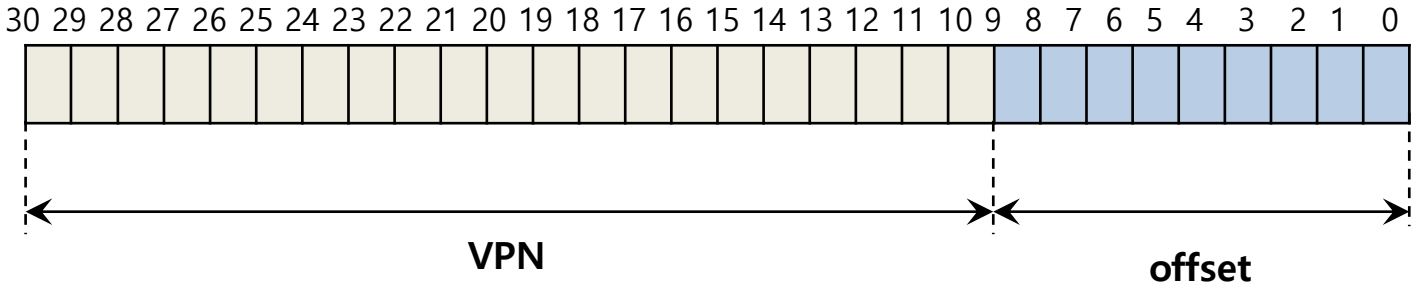
다단계의 예 : 페이지 디렉토리

- PDE가 유효하면, 더 나아간다.
  - 페이지 디렉토리 항목이 가리키는 페이지 테이블의 페이지에서 페이지 테이블 항목(PTE)를 찾는다,
- 페이지 테이블 인덱스 (page-table index)로 페이지 테이블 내부에서 PTE를 찾는다.



두 단계 이상

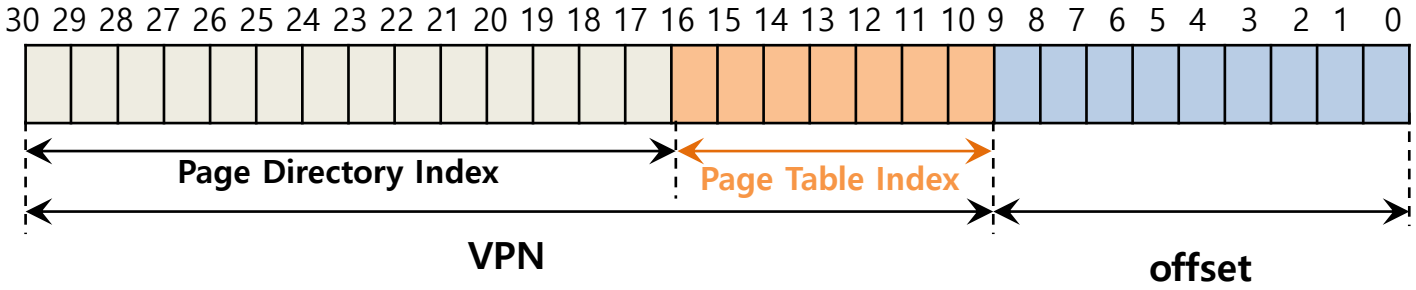
- 어떤 경우는 더 깊은 트리구조가 가능하다.



Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit

두단계 이상

- 여러 단계가 필요한 경우.

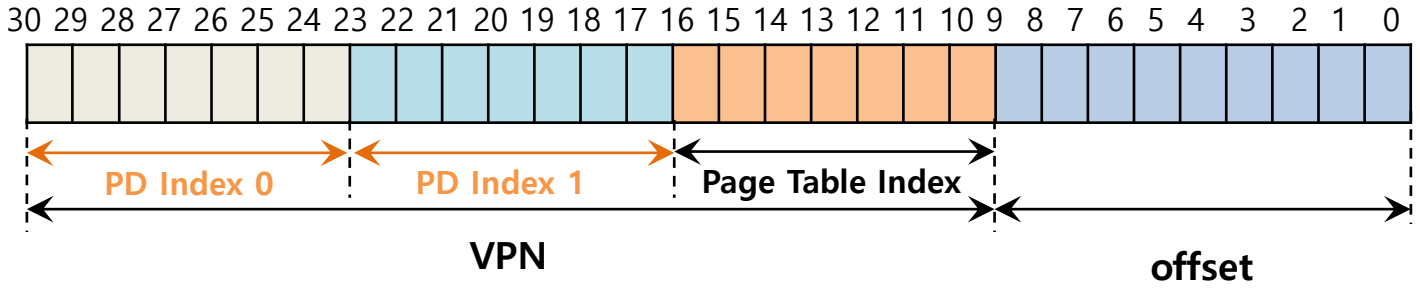


Flag	Detail
Virtual address	30 bit
Page size	512 byte
VPN	21 bit
Offset	9 bit
Page entry per page	128 PTEs

→ $\log_2 128 = 7$

두 단계 이상

- 페이지 디렉토리에 2^{14}개의 원소가 있다면, 하나에 페이지에 담을 수 없고 128개의 페이지가 필요하다.
- 이를 해결해기 위해 레벨을 더 늘린다. 페이지 디렉토리를 한단계위의 페이지 디렉토리로 관리한다.



Multi-level Page Table Control Flow

```
01:      VPN = (VirtualAddress & VPN_MASK) >> SHIFT
02:      (Success,TlbEntry) = TLB_Lookup(VPN)
03:      if(Success == True)        //TLB Hit
04:        if(CanAccess(TlbEntry.ProtectBits) == True)
05:              Offset = VirtualAddress & OFFSET_MASK
06:              PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
07:              Register = AccessMemory(PhysAddr)
08:        else RaiseException(PROTECTION_FAULT);
09:      else // TLB Miss
```

- (1 lines) 가상 페이지 넘버를 얻는다. (VPN)
- (2 lines) TLB에 해당 VPN이 있는지 검사한다.
- (4-8 lines) TLB에서 해당 원소를 꺼내서 물리주소를 얻은 후 메모리에 접근한다.

Multi-level Page Table Control Flow

```
11:                 // Page Directory Access
12:                 PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13:                 PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14:                 PDE = AccessMemory(PDEAddr)
15:                 if(PDE.Valid == False)
16:                           RaiseException(SEGMENTATION_FAULT)
17:                 else // PDE is Valid: now fetch PTE from PT
```

- (11 lines) 페이지 디렉토리 인덱스(PDIndex)를 구한다.
- (13 lines) 페이지 디렉토리 항목을 로드한다.(PDE)
- (15-17 lines) 각종 플랙을 검사하고, 실패시 예외(Exception)을 발생시킨다.

The Translation Process: Remember the TLB

```
18:     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
19:     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
20:     PTE = AccessMemory(PTEAddr)
21:     if(PTE.Valid == False)
22:             RaiseException(SEGMENTATION_FAULT)
23:     else if(CanAccess(PTE.ProtectBits) == False)
24:             RaiseException(PROTECTION_FAULT);
25:     else
26:             TLB_Insert(VPN, PTE.PFN , PTE.ProtectBits)
27:             RetryInstruction()
```

- 페이지 테이블 항목을 로딩한다.
- 이를 통해 물리 메모리 주소를 합성한다.
- TLB에 물리 메모리 주소를 등록하고 명령을 재실행 한다.

AMD 64 (우리의 CPU)

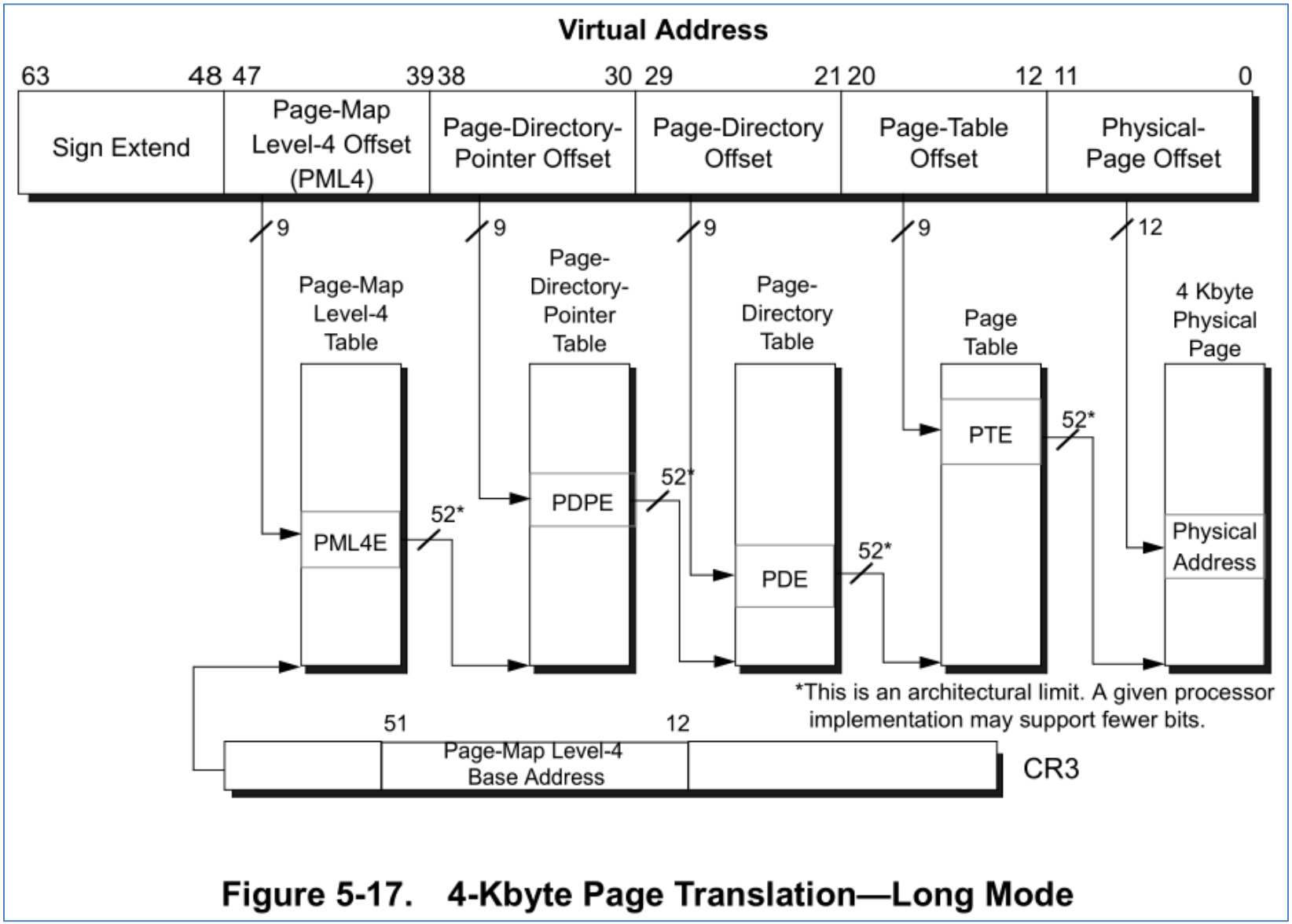


Figure 5-17. 4-Kbyte Page Translation—Long Mode

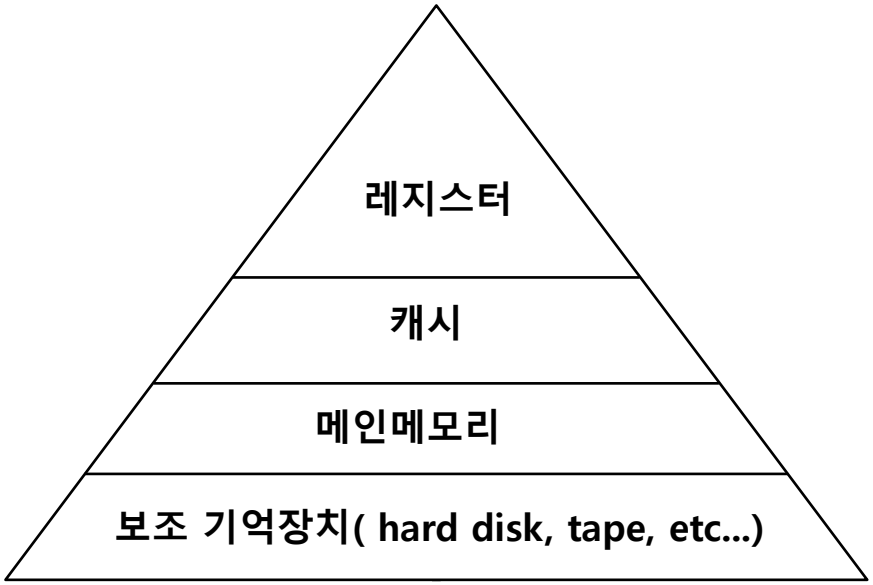
역(Inverted) 페이지 테이블

- 하나의 페이지 테이블로 모든 프로세스의 페이지를 관리
- 각각의 항목은 그 페이지가 어느 프로세스의 어떤 가상 주소가 사용하는 지를 저장한다.
- 문제 : 주소 변환에 검색이 필요하다.
  - 해싱으로 해결한다.
- 예) PowerPC

21. 물리 메모리 크기의 극복 : 메커니즘

물리 메모리 크기의 극복

- 메모리 계층(memory hierarchy)을 확장한다.
  - OS는 당장 필요하지 않은 주소 공간을 다른 곳에 치워 놓을 필요가 있다.
  - 현대 시스템에서는 보조 기억장치가 그런 용도로 사용된다.



현대 컴퓨터의 메모리 계층

물리 메모리 부족

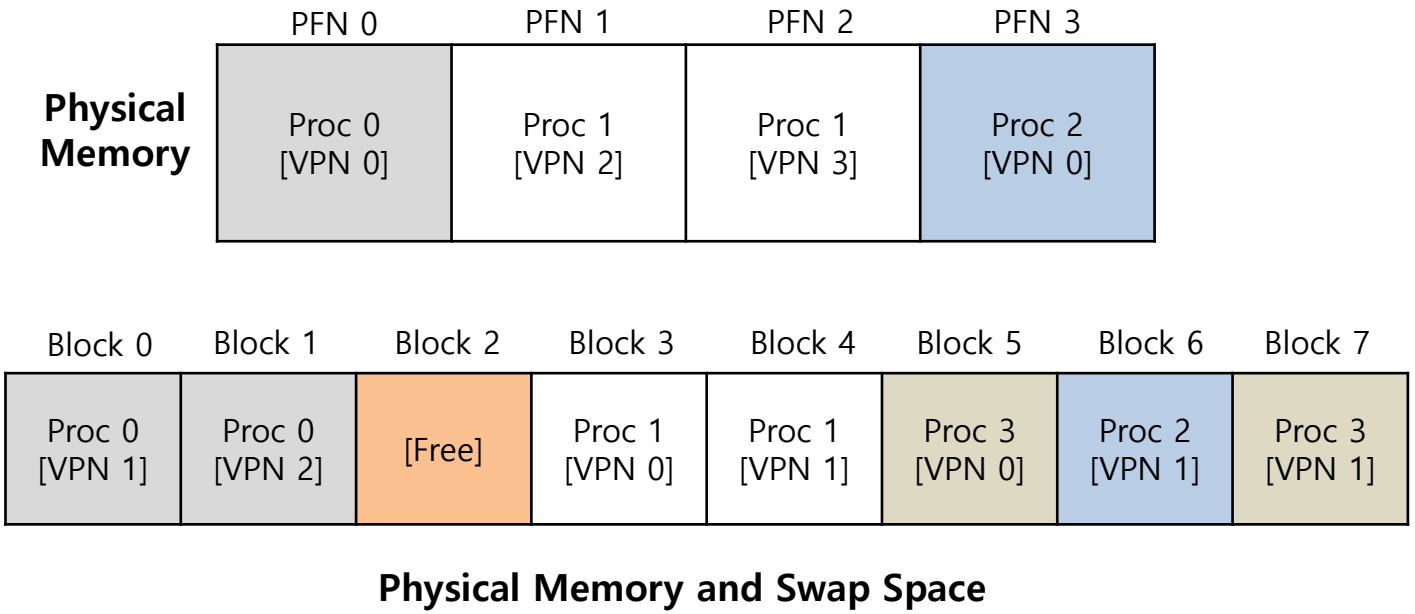
- 원인
  - 메모리를 많이 사용하는 프로그램
  - 멀티 프로그래밍
    - 메모리에 많은 프로세스를 올려 놓을 수록 CPU 사용 효율 증가
- 해결 = 스왑(Swapping)
  - 잘 사용하지 않는 페이지를 하드디스크에 저장하자.
  - 페이징이 아니면 사용하기 어려운 개념
    - 전부, 혹은 세그먼트단위로 저장해야 한다.

‘큰’ 주소공간 제공

- ‘큰‘ 주소공간?
  - 많은 메모리를 요구하는 프로그램
  - 프로그램의 확장성
- 메모리 오버레이(Memory Overlay)
  - 단일 프로세스
  - 함수를 실행하기 전에 함수의 코드와 함수가 사용할 데이터를 프로그래머가 메모리에 로딩.
    - 복잡한 사용법
- 스왑공간 (swap space)과 페이징을 사용하면 물리 메모리보다 큰 주소공간 사용가능

스왑공간(Swap Space)

- 디스크에서 페이지를 넣었다 뺐다 할 수 있는 공간 확보.
- OS는 각 페이지가 어디에 저장되었는지 알아야 한다.



Physical Memory and Swap Space

Present Bit

- 페이지를 스와핑하기위한 필수 하드웨어 기능
  - PTE에서 페이지가 실제로 실제 메모리에 존재하는가의 여부를 나타냄.
- PB가 0인 페이지를 프로그램에서 접근하면?
  - 페이지 폴트(page fault)인터럽트가 발생한다.
  - OS에서 이를 해결해야 한다.
    - 페이지 폴트 핸들러(page fault handler)에서 해결
    - 페이지폴트를 해결하고 프로세스를 계속 실행 시켜야 한다.

Value	Meaning
1	page is present in physical memory
0	The page is not in memory but rather on disk.

메모리가 가득 차면

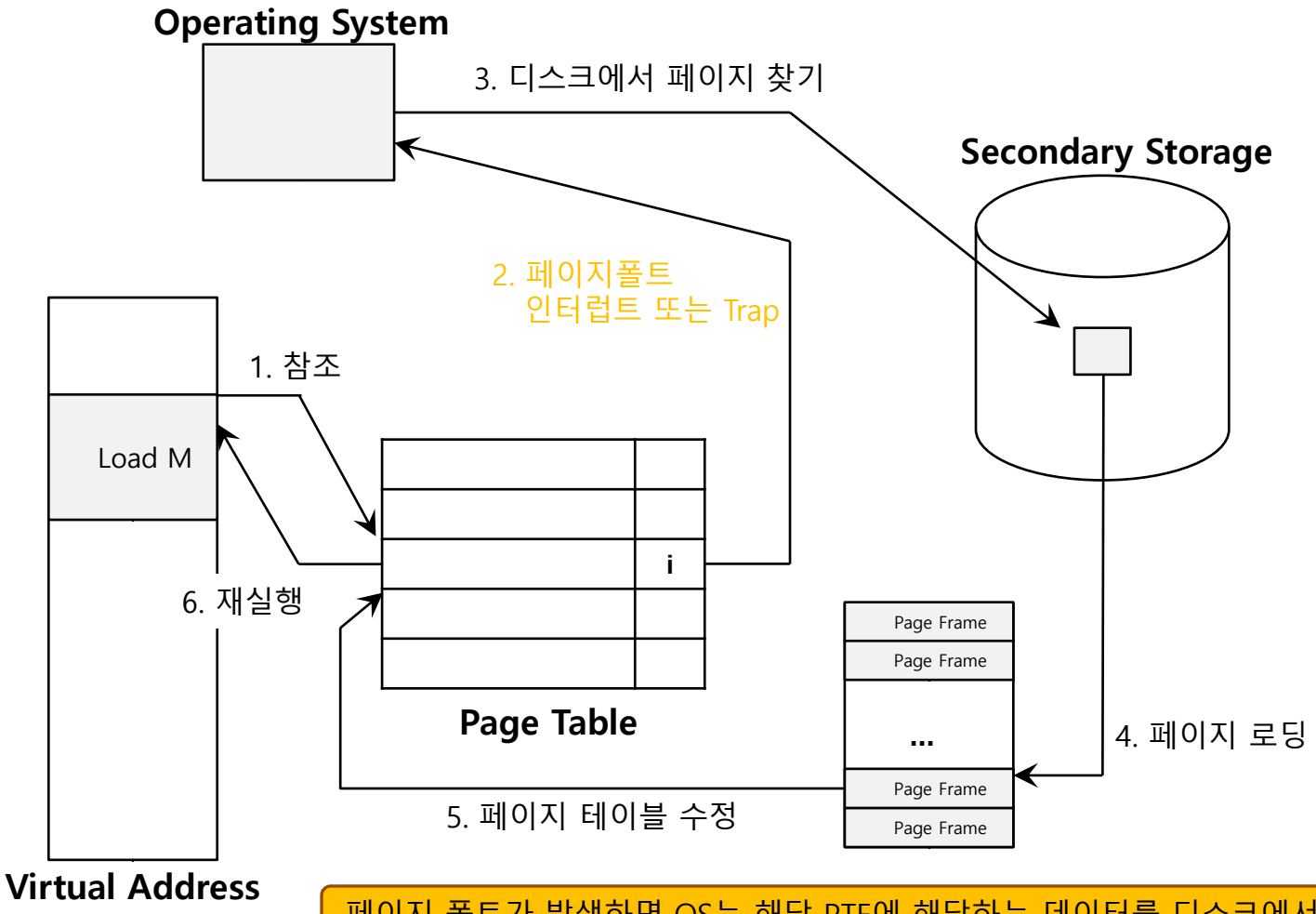
- OS는 실제 메모리에 올라와 있는 페이지들을 디스크로 내려보내야 한다.
  - 어떤 페이지를 내려보내는 가는 페이지 교체 정책 (page-replacement policy)으로 결정한다.

페이지 폴트

- 실제 메모리에 없는 페이지에 접근 하는 것.
  - 만일 그 페이지가 디스크에 스와핑되어 있다면 운영체제는 그 페이지를 메모리로 불러와서 문제를 해결해야 한다.

페이지 폴트 처리

- PFN에 디스크 상에 페이지위치를 넣을 수도 있다.



페이지 폴트가 발생하면 OS는 해당 PTE에 해당하는 데이터를 디스크에서 로딩해야 한다.

페이지 폴트 처리 - HW

```
1: VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2: (Success, TlbEntry) = TLB_Lookup(VPN)
3: if (Success == True) // TLB Hit
4:       if (CanAccess(TlbEntry.ProtectBits) == True)
5:             Offset = VirtualAddress & OFFSET_MASK
6:             PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7:             Register = AccessMemory(PhysAddr)
8:       else RaiseException(PROTECTION_FAULT)
```

페이지 폴트 처리 - HW

```
9: else // TLB Miss
10:       PTEAddr = PTBR + (VPN * sizeof(PTE))
11:       PTE = AccessMemory(PTEAddr)
12:       if (PTE.Valid == False)
13:                 RaiseException(SEGMENTATION_FAULT)
14:       else
15:       if (CanAccess(PTE.ProtectBits) == False)
16:                 RaiseException(PROTECTION_FAULT)
17:       else if (PTE.Present == True)
18:       // assuming hardware-managed TLB
19:                 TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
20:                 RetryInstruction()
21:       else if (PTE.Present == False)
22:                 RaiseException(PAGE_FAULT)
```

페이지 폴트 처리 - OS

```
1:        PFN = FindFreePhysicalPage()
2:        if (PFN == -1) // no free page found
3:                PFN = EvictPage() // run replacement algorithm
4:        DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
5:        PTE.present = True // update page table with present
6:        PTE.PFN = PFN // bit and translation (PFN)
7:        RetryInstruction() // retry instruction
```

- OS는 읽어올 페이지를 위한 빈자리 프레임을 찾는다..
- 빈 프레임이 없으면, 교체 알고리즘을 수행해서 몇 개의 페이지를 디스크로 쫓아낸다.
- 4번에서 실제로는 블럭상태가 되어 DiskRead가 끝날때 까지 대기
  - 사실 모든 디스크 접근은 프로세스의 상태를 기록하고 대기시킨다.

교체는 실제 언제 일어나는가?

- 메모리가 꽉 차면 교체를 시작한다.
  - 조금 실제적이 않다. 운영체제는 원활한 운영과 비상시를 위한 빈 공간을 항상 남겨 두어야 한다.

- 스왑데몬(Swap Daemon), 페이지데몬(Page Daemon)
  - 최소값(LW)보다 여유 페이지가 적으면 페이지를 최대값 (HW)에 도달할 때 까지 디스크로 옮긴다.

22. 물리메모리 크기의 극복 : 정책

실제 메모리 그 이상: 정책

- 메모리 압박은 OS가 페이지를 쫓아 내도록 한다.
- 어떤 페이지를 방출할 것인지 결정하는 것이 OS의 교체 정책이다.

캐시 관리

- 교체 정책의 목적은 캐시 미스를 줄이는 것이다.
- 캐시 미스확률을 통해 평균 메모리 접근시간(*average memory access time, AMAT)*을 얻을 수 있다.

$$AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D)$$

Arguement	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{Hit}	The probability of finding the data item in the cache(a hit)
P_{Miss}	The probability of not finding the data in the cache(a miss)

최적 교체 정책

- 가장 적은 교체회수를 가짐
  - 앞으로 재사용될 시간이 가장 나중인 페이지를 교체
  - 최소의 캐시 미스
- 비교를 위해서 사용, 다른 방법들이 얼마나 완벽한지 알기 위한 비교 용

Tracing the Optimal Policy

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

Future is not known.

간단한 정책: FIFO

- 메모리에 올라온 순서대로 페이지 내보내기.
- 교체가 발생하면 큐의 앞에 있는 페이지가 방출된다.(“처음 들어온“ 페이지)
  - 구현이 간단하지만, 페이지의 중요성이 고려되어 있지 않다.

FIFO 정책의 흐름

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{36.4\%}$

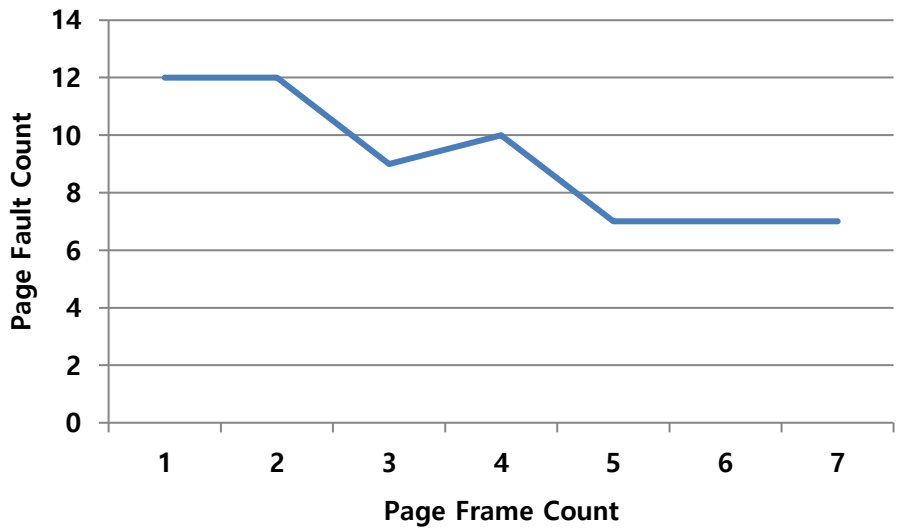
Even though page 0 had been accessed a number of times, FIFO still kicks it out.

BELADY의 역설

- 할당된 물리프레임수가 커질수록 페이지 적중률이 올라갈 것 같지만, FIFO에서 오히려 떨어지는 현상.

Reference Row

1 2 3 4 1 2 5 1 2 3 4 5



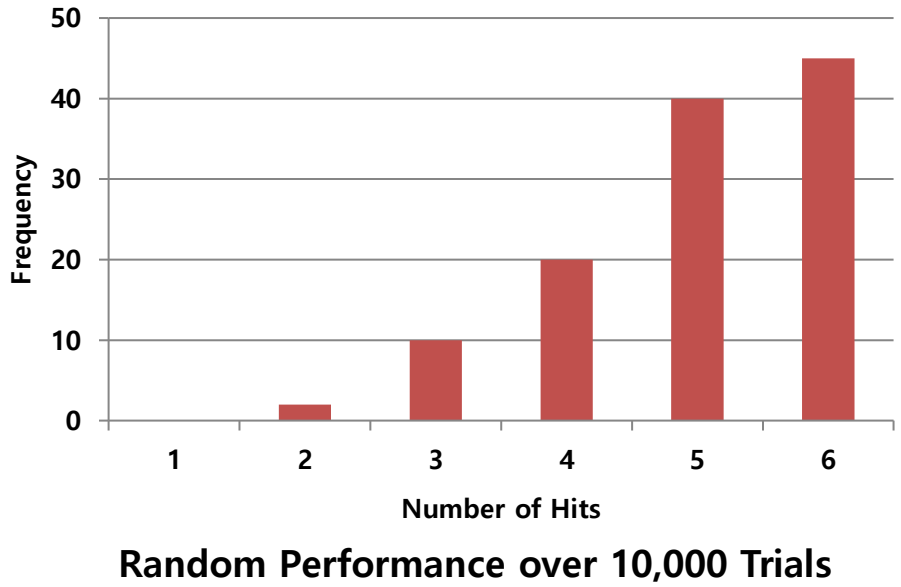
또 다른 간단한 정책 : 무작위

- 메모리 압박시 랜덤한 페이지를 내보낸다.
  - 성능도 운에 따른다.
  - FIFO보다는 좋고 최적보다는 나쁘다.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

무작위 선택의 성능

- 가끔, 최적과 비슷한 성능을 보인다. (6개의 적중)



과거 정보의 사용

- 과거를 살펴보고 역사를 활용한다.
  - 두 종류의 과거 정보

Historical Information	Meaning	Algorithms
최근성(recency)	최근에 접근한 페이지에 또 접근할 확률이 높다.	LRU
빈도수 (frequency)	많이 접근한 페이지일수록 가치가 높아 또 접근할 확률이 높다.	LFU

2022년 중간고사

- 2022년 10월 25일 화요일

- 시험 범위는 : 가상화 메모리까지
  - 병행성 전까지

과거 정보 활용: LRU

- 최소 최근 접근(least-recently-used) 페이지를 교체

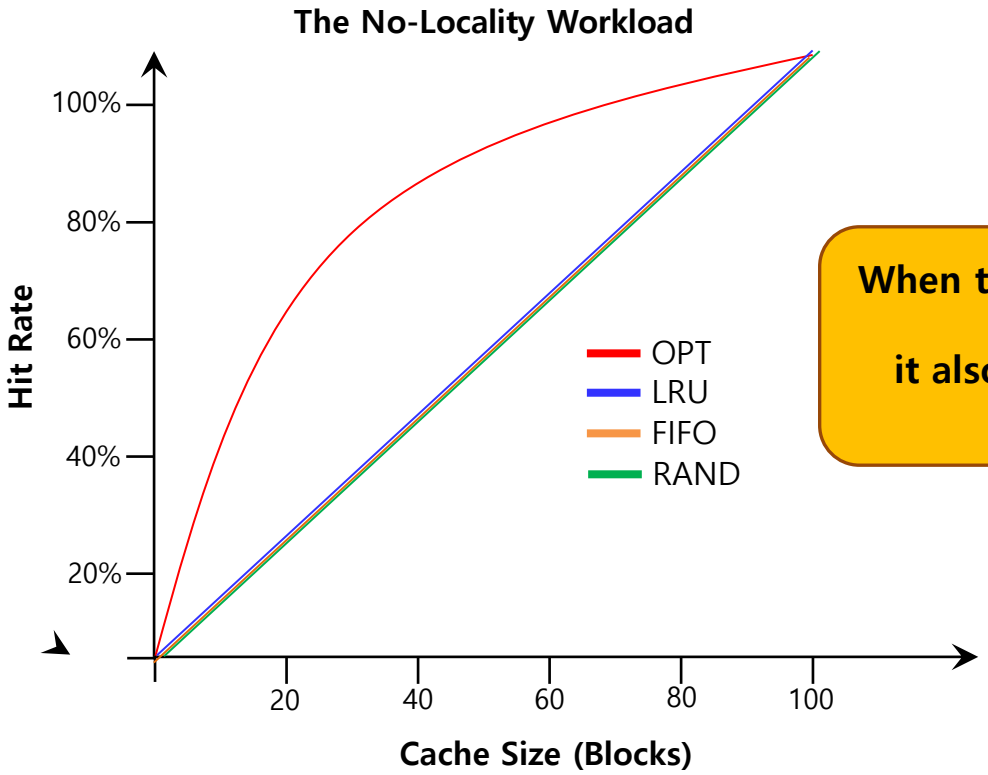
Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

워크로드에 따른 성능 비교

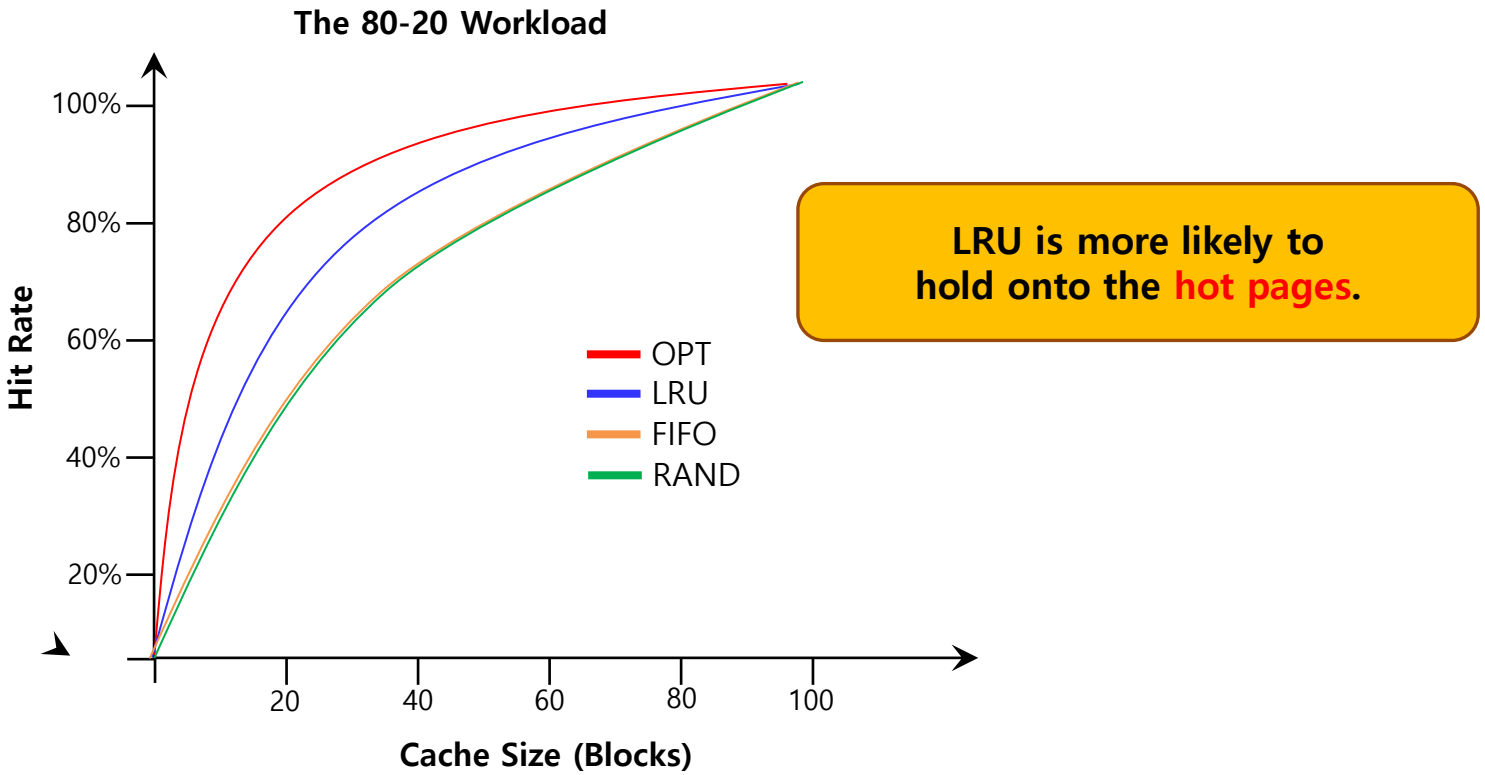
- 지역성이 없는 랜덤한 페이지 접근
  - 100개의 페이지에 총 10000번 접근



When the cache is large enough to fit the entire workload, it also doesn't matter which policy you use.

워크로드 예: 80-20 워크로드

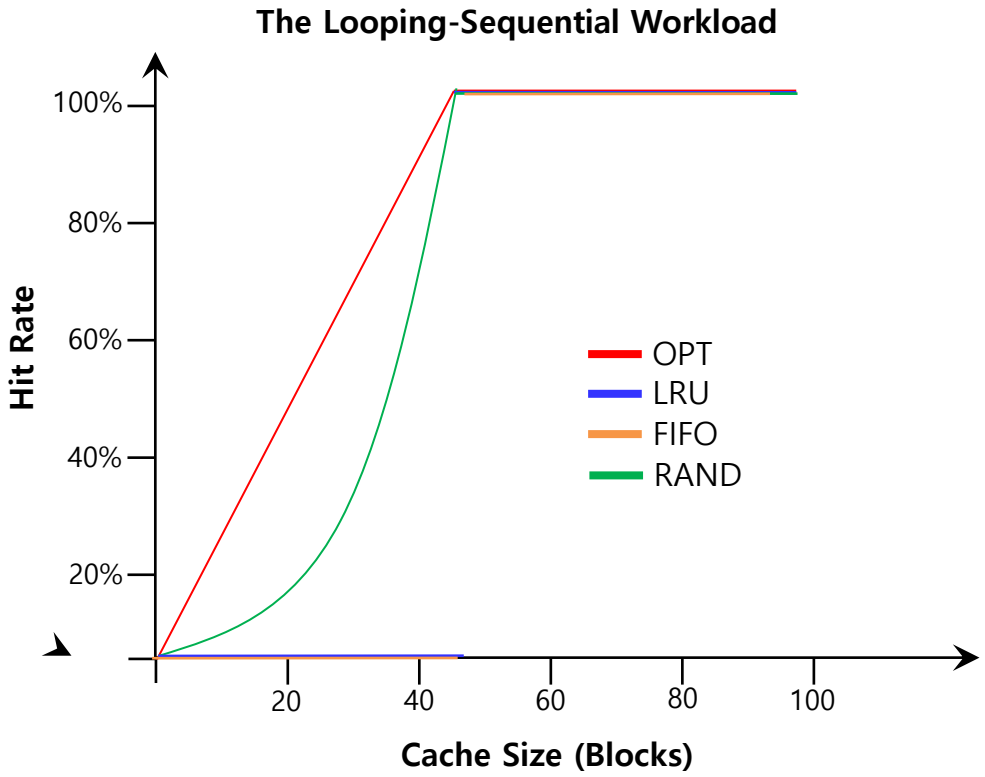
- 지역성 고려 : 총 접근의 80%가 전체 페이지의 20%에 존재.
- 나머지 20%의 접근이 나머지 80%의 페이지에서 발생.



LRU is more likely to hold onto the hot pages.

워크로드 예 : 순환형

- 50페이지를 순차적으로 접근
  - 0, 1, … 49까지 순차 접근 후, 이를 반복한다.



과거 이력 기반 알고리즘의 구현

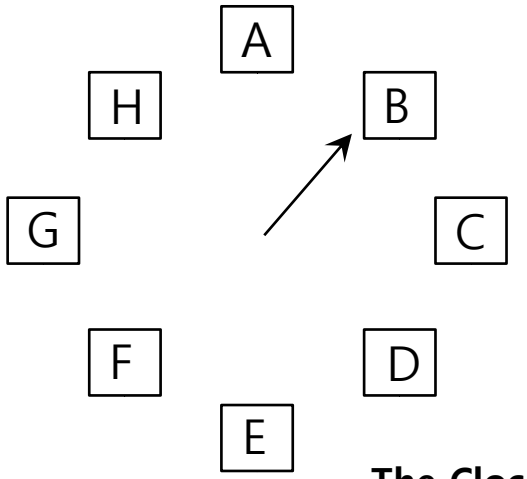
- LRU방식은 메모리접근 정보를 기록해야 한다.
  - 약간의 하드웨어가 필요하다.

LRU 근사하기

- **use bit**형식의 하드웨어 지원이 필요다. 페이지가 참조되면 use bit는 HW에 의해 1이 된다.
  - HW는 그 비트를 초기화 하지 않는다. OS가 필요할 때 초기화 한다.
- 시계 알고리즘
  - 모든 페이지를 원형 리스트에 배치한다.
  - 처음에는 시계바늘이 시작페이지를 가르킨다.

시계 알고리즘

- 시계바늘은 use bit가 0인 페이지를 만날 때까지 이동한다.



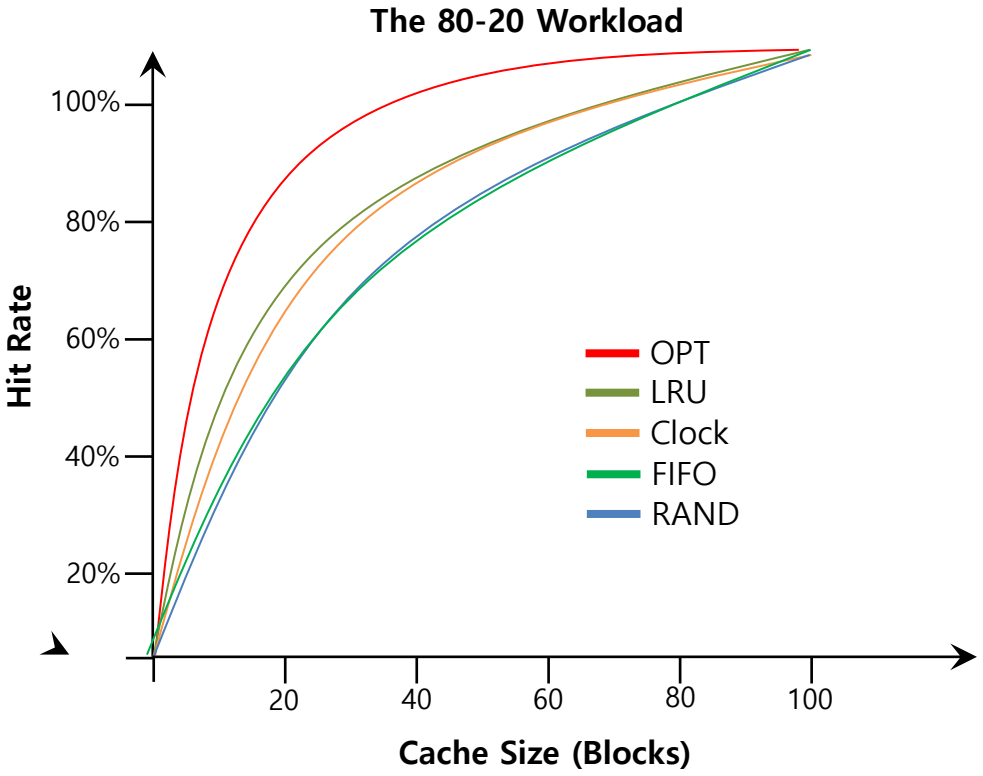
Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

The Clock page replacement algorithm

페이지 폴트가 발생하면 시계바늘이 가리키는 페이지 부터 검사한다.
Use bit에 따라 교체가 진행된다.

시계 알고리즘 워크로드

- LRU보다는 못하지만, 과거 정보를 활용하지 않는 다른 방법보다는 났다.



갱신된 페이지(Dirty Pages) 고려

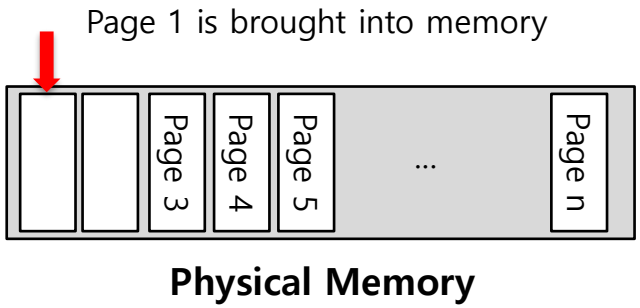
- HW로 구현된 변경 비트(**modified bit**, 또는 **dirty bit**)
  - 페이지가 변경되었다면 방출될 때 디스크에 내용을 기록해야 한다.
  - 변경되지 않았다면, 방출 비용은 없다.
- 갱신되지 않고 사용 되지도 않은 페이지를 우선 방출하고, 없으면 변경되고 사용되지 않은 페이지를 방출한다.

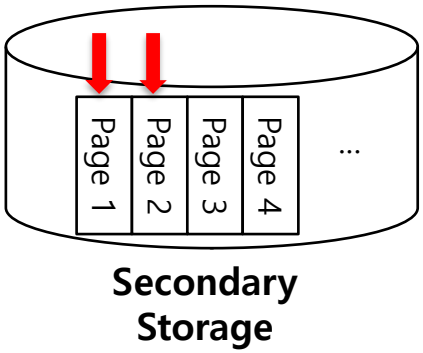
다른 VM정책들

- 페이지들을 언제 메모리에 로딩할 것인가?
- 여러가지 옵션이 있다.

선반입(Prefetching)

- 사용될 페이지를 추측해서 미리 불러오는 것

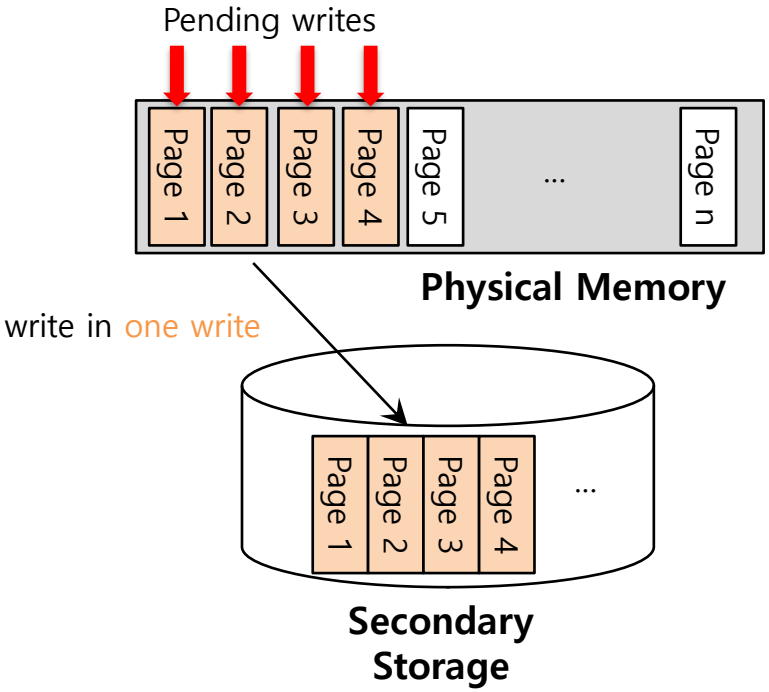




Page 2 likely soon be accessed and
thus should be brought into memory too

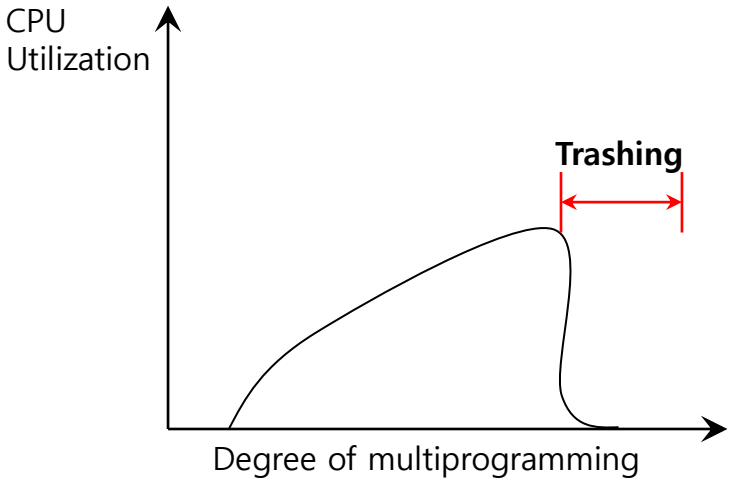
클러스터링(Clustering), 모으기(Grouping)

- 디스크에 써야 하는 페이지들을 모아 놓았다가 한번에 쓰는 방식
  - 여러 번의 작은 디스크 전송보다. 한번의 큰 디스크 전송이 훨씬 효율적이다.



쓰래싱(Thrashing)

- 메모리 사용 요구가 감당할 수 없을 만큼 많고 실행 중인 프로세스가 요구하는 메모리가 가용 물리 메모리 크기를 초과하는 경우.
  - 일부 프로세스의 실행을 중단.
  - **워킹 셋(working set)**이 물리 메모리 크기에 맞도록 조정.



과 제 7

- OPT, FIFO, LRU, RANDOM 시계알고리즘의 성능을 비교 하라.
- 입력 :
  - 첨부한 프로젝트에 있는 page_data.txt 사용
  - 참조한 Page Number가 순서대로 적혀 있음
  - -1로 입력의 종료를 표시
- 조건 :
  - 전체 사용 가능한 물리 메모리의 크기는 128프레임이다.
  - 가상 주소의 크기는 512 페이지이다.
- 결과물 :
  - 각 알고리즘의 Page 적중률 (Hit Ration)
- 첨부한 프로젝트에서 FIFO의 구현을 참고 할 것.
- 제출 (실행화일X, replacement.cpp화일, 실행결과)