CDA0017: Operating Systems

Donghyun Kang (donghyun@changwon.ac.kr)

NOSLab (https://noslab.github.io)

Changwon National University

Paging의 문제

- 주소 변환이 느림
  - 주소 변환을 위해 2번의 페이지 접근이 수행됨
    - 페이지 테이블 접근
    - 명령어 및 데이터 접근
  - 멀티 단계의 페이지 테이블의 경우 더 심각함
    - 다음 시간에..
- 최적화 필요
  - 효율적으로 명령어 및 데이터를 빠르게 접근하기 위한 방법이 필요함

변환-색인 버퍼 (translation-lookaside buffer, TLB)

- 변환-색인 버퍼
  - 페이징의 성능을 향상시키기 위해 하드웨어의 도움이 필요함
  - MMU (memory management unit) 칩에 내장되어 있음
  - 자주 참조되는 가상 주소-물리주소 변환 정보에 대한 캐시 역할을 함
    - 즉, 주소-변환 캐시 (address-translation cache)

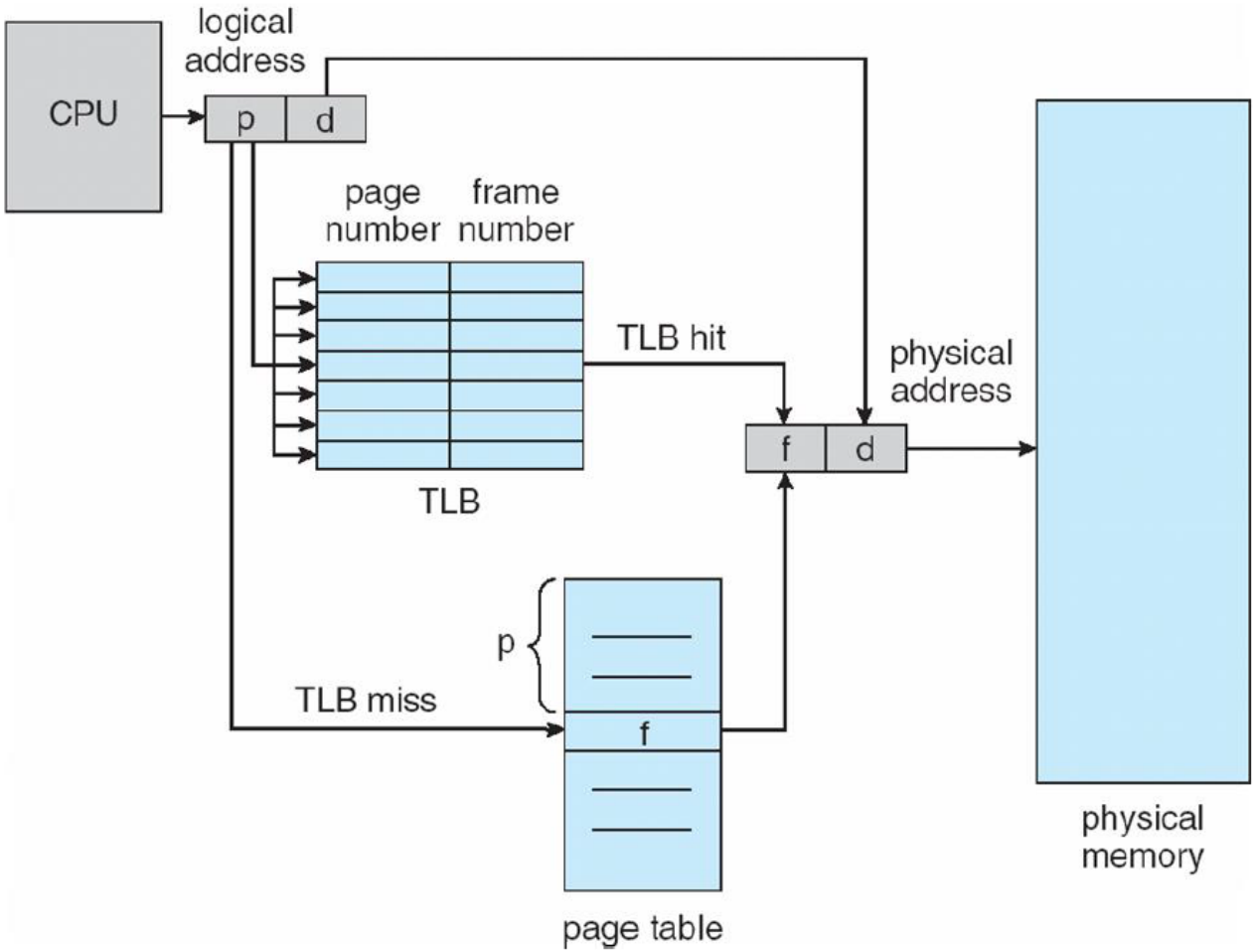
- 매핑 변환 단계
  - 가상 주소 -> TLB 캐시 확인
  - cache hit: TLB에서 물리 주소 변환 후 리턴
  - cache miss: 기존과 동일하게 메모리 접근 후 리턴

변환-색인 버퍼 (translation-lookaside buffer, TLB)

- 변환-색인 버퍼는 하드웨어로 구현되어 있음
  - 16 ~ 256개의 엔트리 포함
  - 접근 방식
    - Full associative
    - Set associative
  - 교체 방식
    - LRU 방식 사용
  - TLB는 PFN이 아닌 PTE을 캐싱함

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

변환-색인 버퍼 기반 주소 변환



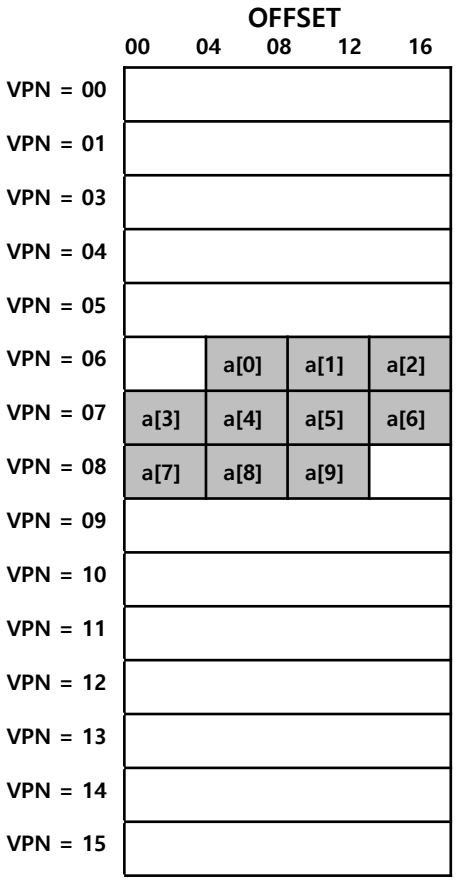
변환-색인 버퍼 알고리즘

```
1: VPN = (VirtualAddress & VPN_MASK ) >> SHIFT
2: (Success , TlbEntry) = TLB_Lookup(VPN)
3:    if(Success == Ture){ // 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)
```

변환-색인 버퍼 알고리즘

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

예제: 배열 접근



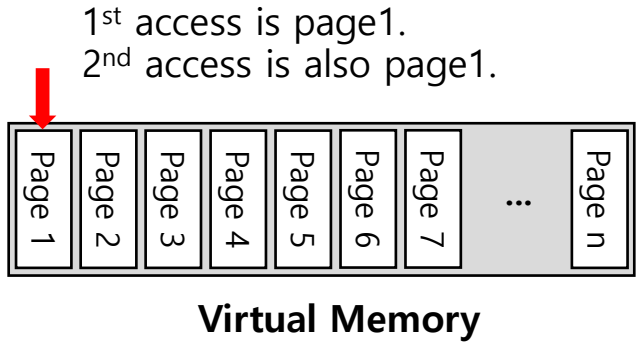

```
0:      int sum = 0 ;
1:      for( i=0; i<10; i++){
2:              sum+=a[i];
3:      }
```

The TLB improves performance due to spatial locality

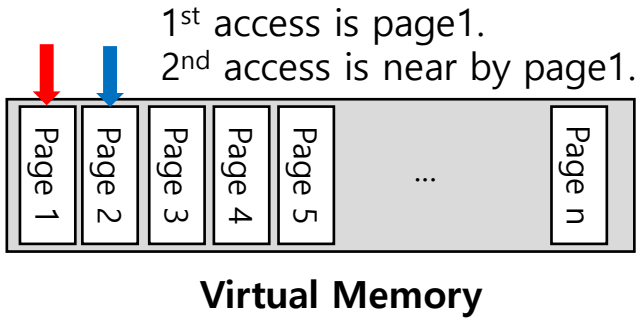
3 misses and 7 hits.
Thus TLB hit rate is 70%.

지역성

- 시간 지역성
  - 최근에 접근된 명령어 및 데이터가 다시 접근될 가능성이 높음



- 공간 지역성
  - 명령어 및 데이터가 순차적으로 접근될 가능성이 높음



TLB Cache Miss 처리

- 하드웨어 기반 Cache miss
  - CISC (complex-instruction set computers) 구조에서 사용
  - 하드웨어가 페이지 테이블에 대한 정보를 가지고 있음
    - 즉, page-table base register에 대한 위치 및 형식을 하드웨어가 파악하고 있음

- Cache miss 처리 방법
  - 페이지 테이블 엔트리 검색
  - 필요한 변환 정보 추출
  - TLB 갱신
  - TLB miss 명령어 재 실행

TLB Cache Miss 처리

- 소프트웨어 기반 Cache miss
  - RISC (reduced instruction set computing) 구조에서 사용
  - 소프트웨어 관리 TLB라고도 불림

- Cache miss 처리 방법

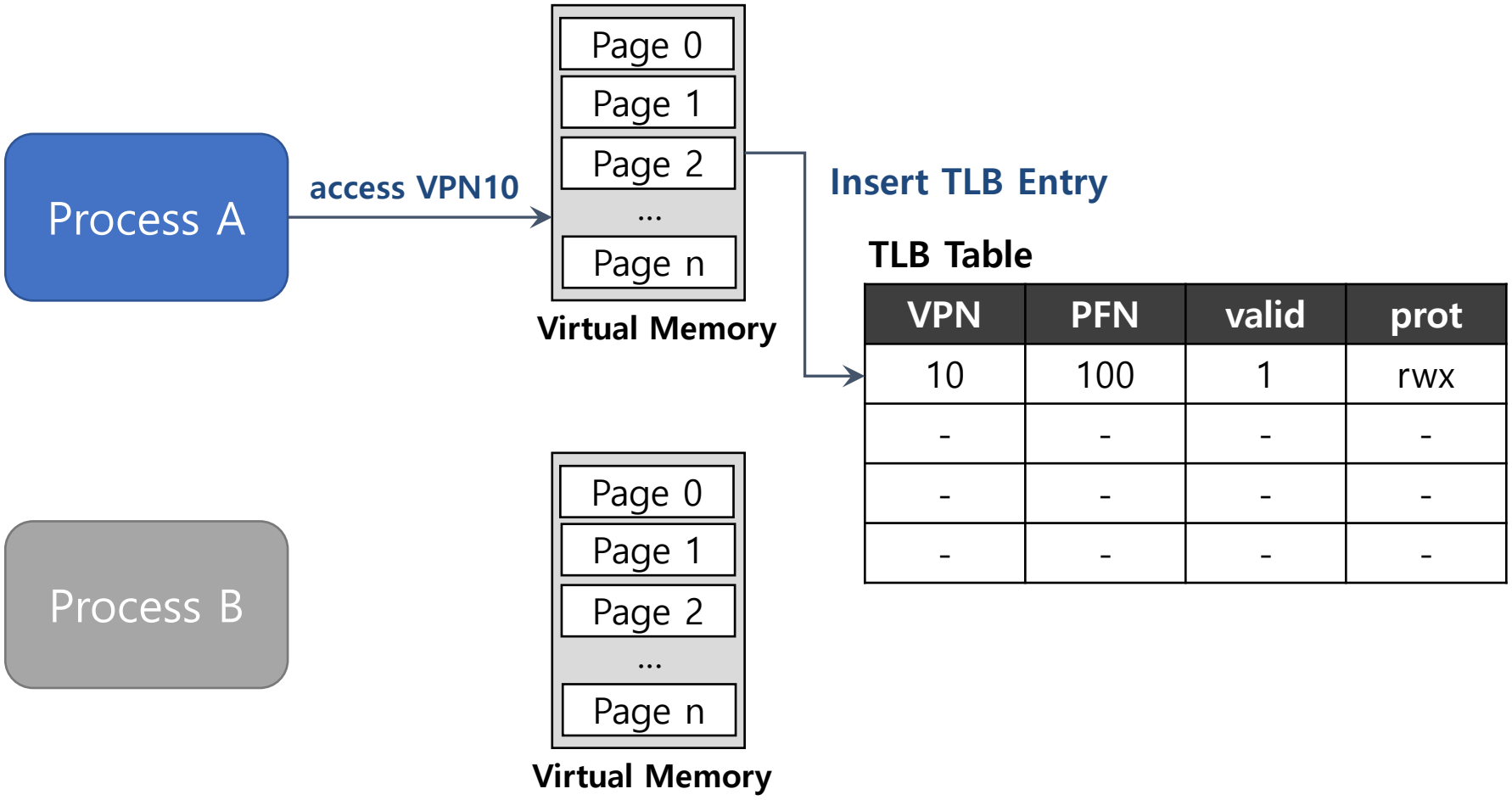
```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)      // TLB 히트
4      if (CanAccess(TlbEntry.ProtectBits) == True)
5          Offset   = VirtualAddress & OFFSET_MASK
6          PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7          Register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

소프트웨어 TLB 트랩 핸들러의 중요 사항

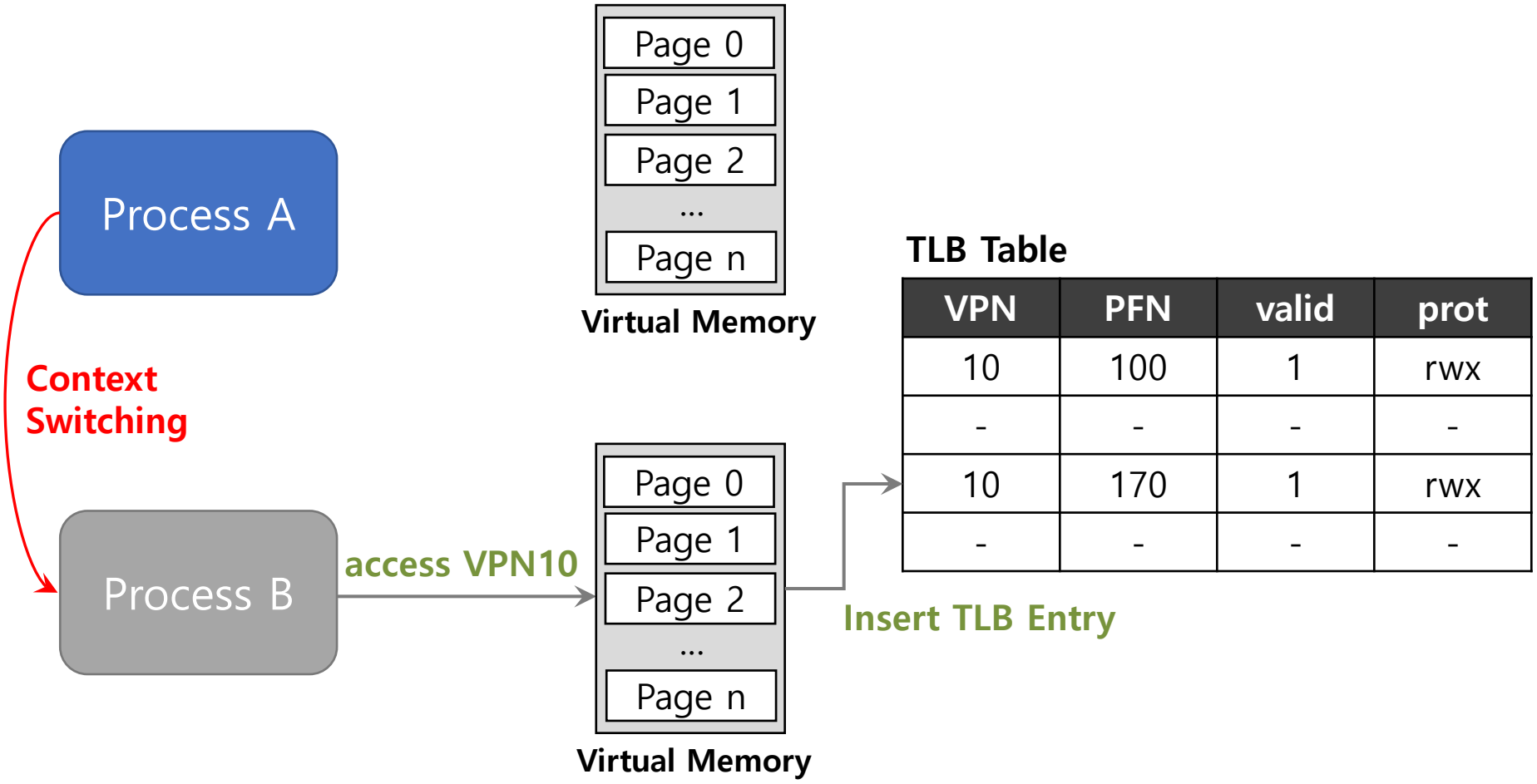
- TLB 트랩 핸들러는 시스템 콜 호출의 트랩 핸들러와 다름
  - 시스템 콜: 트랩 핸들러 호출 후 "다음 (**next**)" 명령어 실행
  - TLB: 트랩 핸들러 호출 후 "현재 (**current**)" 명령어 재 실행

- TLB miss가 무한 반복되지 않도록 설계 해야함
  - 해결 방안
    - TLB 트랩 핸들러를 물리 주소로 직접 할당
      - 즉, 주소 변환이 필요 없음
    - TLB 트랩 핸들러의 일부를 핸들러 코드 주소에 영구히 저장
      - 즉, TLB 핸들러는 항상 TLB에서 cache hit 발생

TLB의 문제점: 문맥 교환



TLB의 문제점: 문맥 교환



TLB의 문제점: 해결 방안 1

- 문맥 교환을 수행할 때 기존 TLB 내용을 clear 함
  - 소프트웨어 기반 TLB: 특권을 갖는 하드웨어 명령어를 사용하여 변경
  - 하드웨어 기반 TLB: 페이지 테이블 베이스 레지스터가 변경될 때 변경

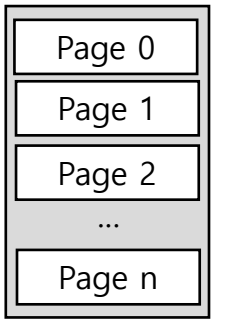
- 문제점은??

TLB의 문제점: 해결 방안 2

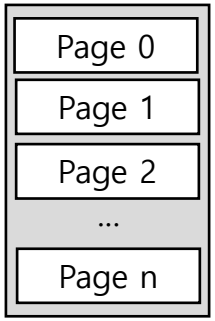
- TLB 내에 주소 공간 식별자 (address space identifier, ASID)을 추가함
  - PID와 유사한 개념으로 TLB의 프로세스 ID을 식별함

Process A

Process B



Virtual Memory



Virtual Memory

TLB Table

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

TLB의 페이지 공유

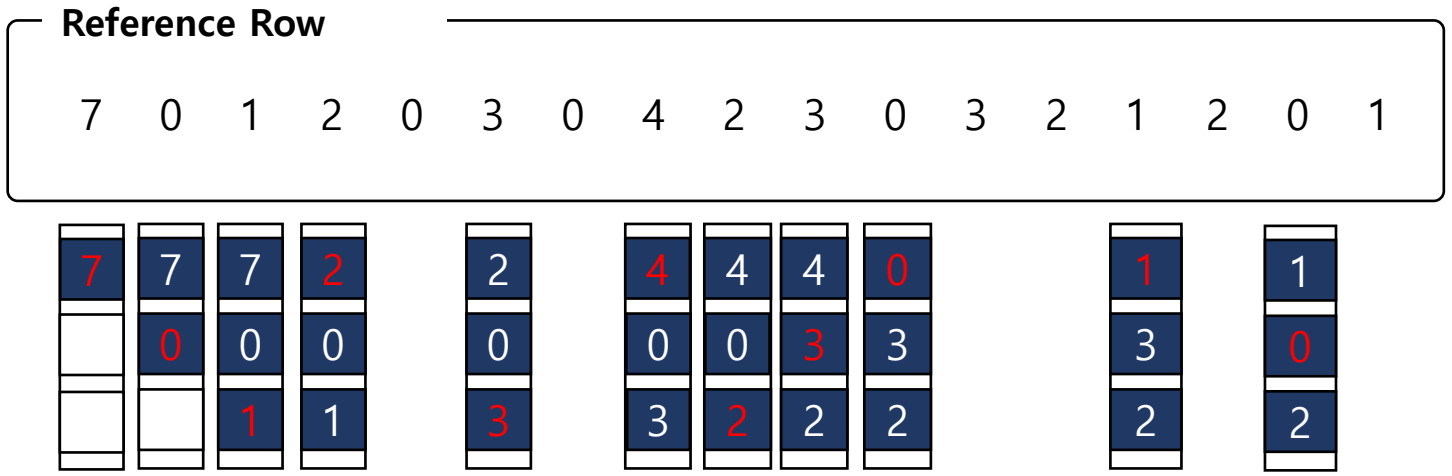
- 두개의 프로세스가 하나의 페이지를 공유하는 경우
  - 서로 다른 가상 주소가 하나의 물리 주소를 포인팅 함

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 페이지 교체 정책

- LRU(Least Recently Used)



Total 11 TLB miss

Q&A