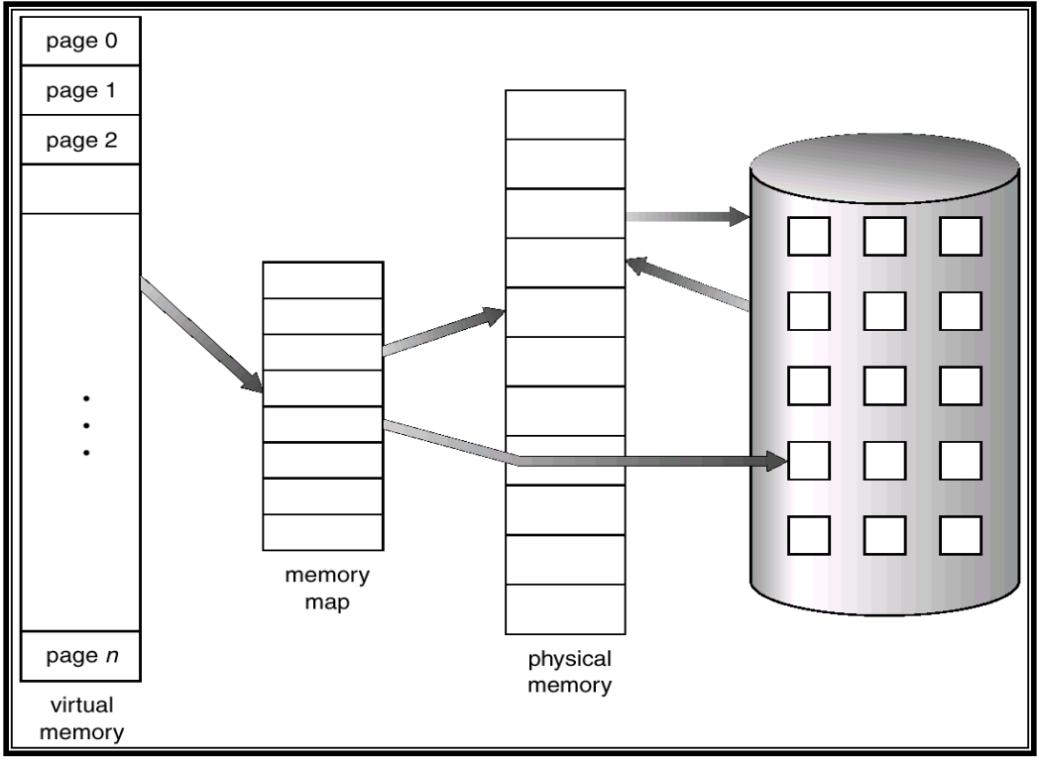
**Ch 9) virtual memory management**

* User memory와 physical memory의 분리하는 것이 핵심.
* 8장에서는 virtual address translation에 대해 설명했다면 9장에서는 마치 무한대의 메모리가 있는 것처럼 운영체제가 어떻게 서비스 하는지에 대해 좀 더 자세히 알아본다.

1. Demand Paging

* 메모리에 올려 놓은 프로세스를 모두 사용하지 않음. 사용하지 않는 프로세스는 secondary-storage에 내리고 필요한 것만 메모리에 올려서 사용.

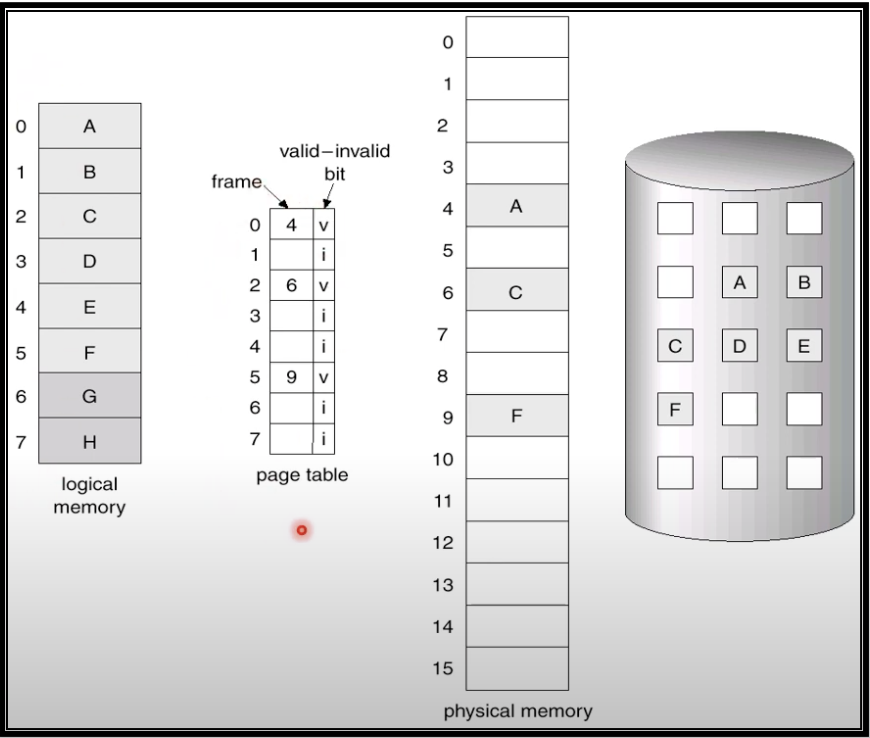


* Locality

1. Temporal locality: 최근에 사용된 것이 다시 사용될 확률이 높음. 가장 사용되지 않는 것을 하드디스크에 내리고 가장 최근에 사용된 것을 메모리에 보존하는 방식.
2. Spatial locality: 해당 page 근처에 있는 것이 다시 사용될 확률이 높음. 근처에 있는 page를 메모리에 보존.
3. Valid-invalid Bit

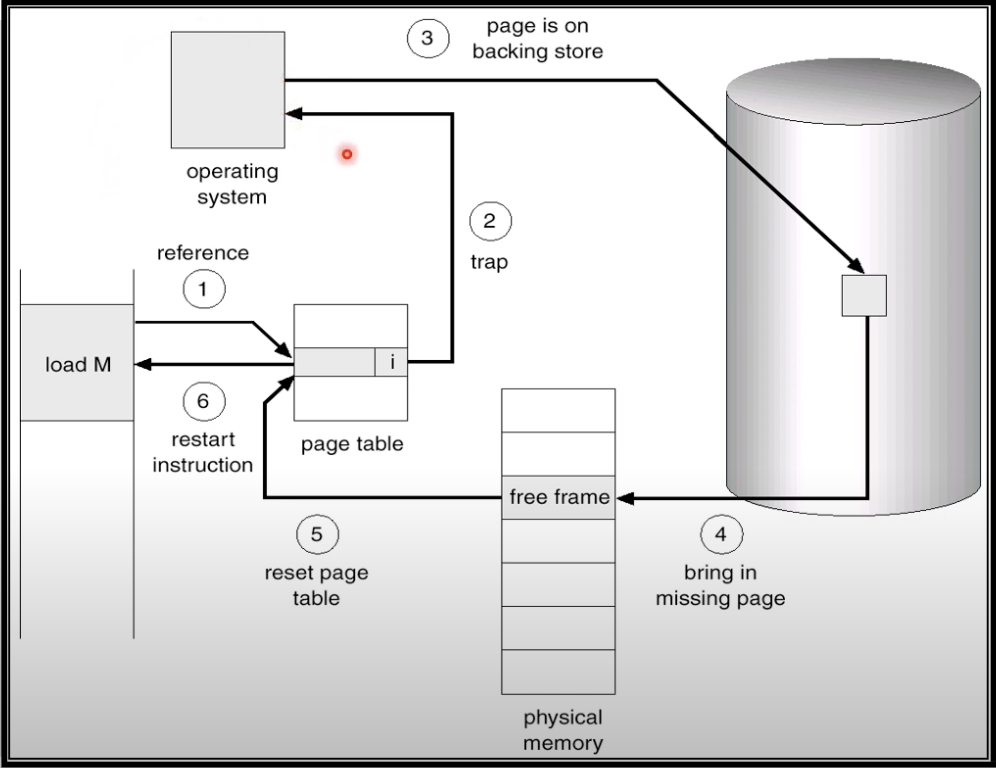
* 1(valid) -> in-memory, 0(invalid) -> not in memory
* Invalid한 경우는 두가지 경우로 나뉜다.

1. Page fault: 실제로는 valid하지만 main memory에 있는 것이 아닌 hard-disk에 있는 상태
2. Protection fault: 정말 invalid한 메모리에 access한 경우
3. Page table when some pages are not in main memory (예시)



* + A, C, F -> valid
  + B, D, E -> invalid(page fault)
  + G, H -> invalid(protection fault)
  + G, H(6, 7) 같은 경우는 logical memory에만 존재하고 hard-disk와 physical memory에는 존재하지 않음. 따라서 불러올 수 없는 invalid한 메모리에 access한 경우(memory protection fault)에 속한다.

1. Page Fault Handling 과정

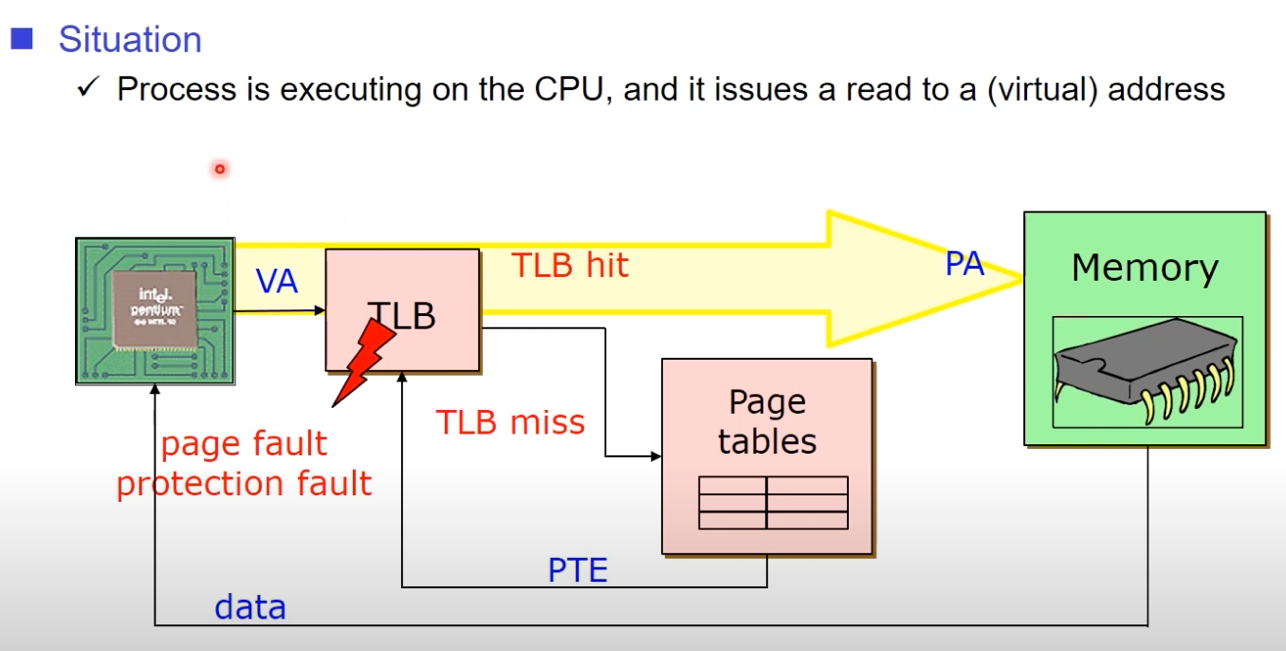


1. Reference: 그림에는 없지만 TLB를 방문하는 과정이 포함. TLB miss(<-> TLB hit) 되어 page table access. 하지만 invalid인 상태
2. Trap: CPU에게 trap을 건 후 운영체제에 요청
3. Page is on backing store: page가 메모리에 있지 않고 hard-disk에 있는 상태.
4. Bring in missing page: hard-disk에 있는 page를 physical memory에 올린다.
5. Reset page table: page table을 업데이트 해준다. Page정보를 page table에 올리고 invalid 상태를 valid상태로 변경.
6. Restart instruction

\*\* 이렇게 요구하는 page를 올려놓는 기법을 demand paging이라고 한다.

\*\* 만약 여기서 Free frame(빈 공간)이 없다면? -> 내릴 것(앞으로 가장 안 쓰일 것)을 선택하여 hard-disk로 내린다.

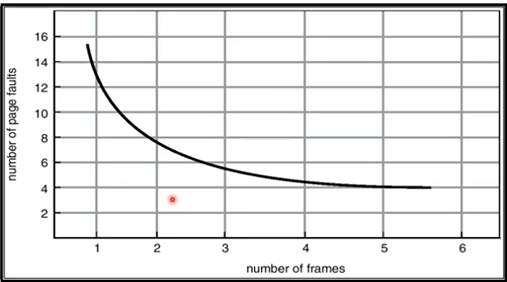
1. Memory Reference



* VA: virtual address, PA: physical address
* 99%는 TLB hit라고 함.

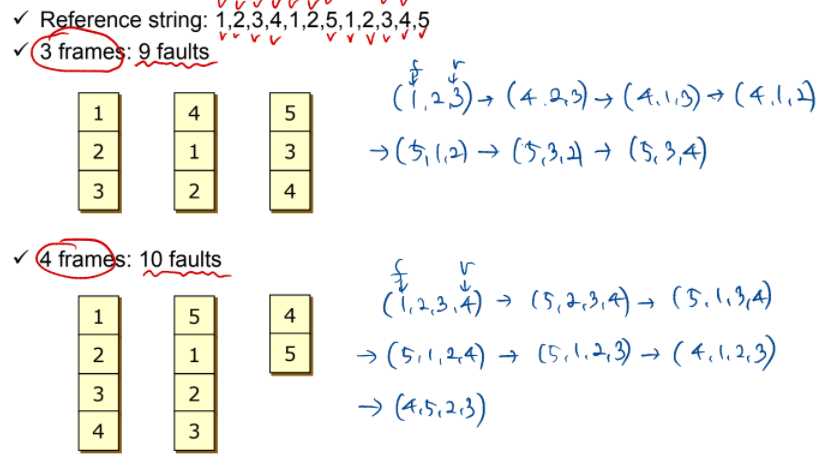
1. \*\*\* Page Replacement Algorithms(중요)

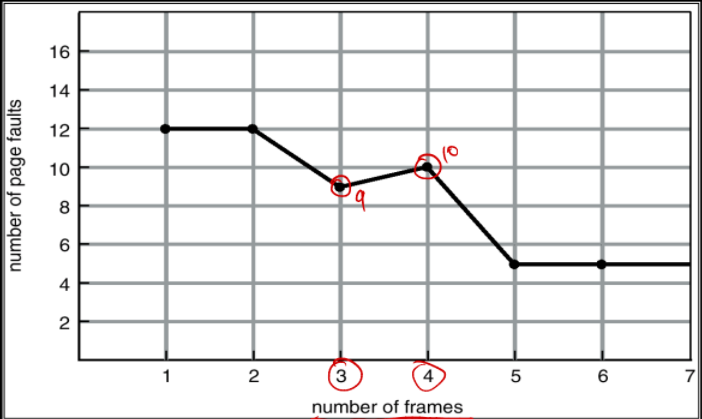
* Page Replacement: page fault가 일어났을 때 Page를 교체하는 작업
* 앞으로 안 쓸 것을 하드 디스크로 내리고 자주 쓰는 것을 메모리에 남겨놓음
* Page fault 횟수를 줄여서 hard-disk에서 page를 불러오는 I/O작업을 최소화 하는 것이 핵심



* Frame(free frame)의 수와 page fault수의 관계
* frame수가 늘어나면 page fault수가 줄어드는 현상이 나타난다 왜 그럴까?
* 메모리에 넣을 수 있는 frame수가 늘어나면 최근에 사용한 것들을 많이 가지고 있을 가능성 있다.

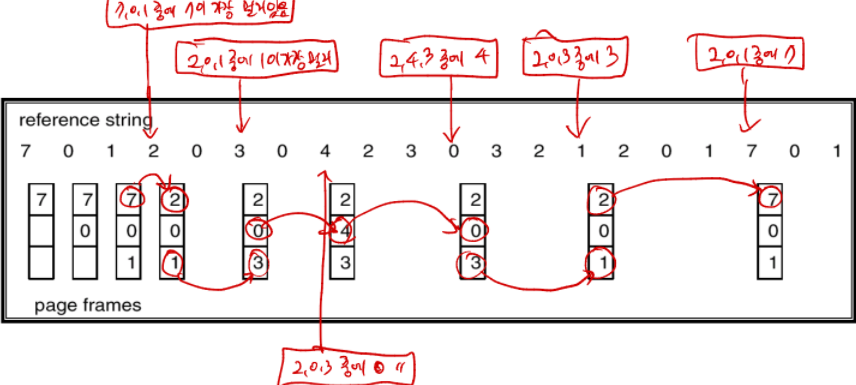
1. FIFO (first-come-first-out)
   * 가장 먼저 들어온 데이터가 가장 오래된 데이터이고 오래된 데이터는 앞으로는 잘 사용되지 않을 것 이라고 예측하여 이 데이터를 하드 디스크로 내려서 replace하는 방식





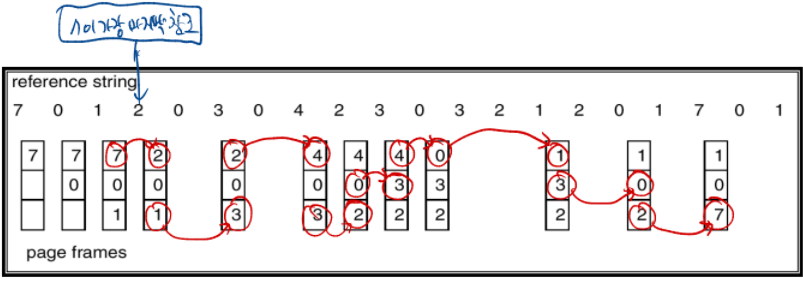
* + frame수가 증가하면 page fault수가 줄어들어야 하지만 3-4구간에서 오히려 늘어나는 것을 볼 수 있다.
  + 오래된 것을 꼭 자주 사용하지 않는 것이라고 할 수는 없음
  + FIFO 알고리즘은 그렇게 좋은 알고리즘이 아님

1. Optimal Algorithm
   * 교체 되어야 할 frame number가 미래에 얼마나 멀리 떨어져 있는가를 확인 후 교체
   * 가까이 있는 것은 재사용될 확률이 높다, 멀리 있는 것을 빼주자.



* + 앞서 배운 SJF 알고리즘처럼 미래에 어떤 frame이 들어올 지 모르기 때문에 사실상 실현하기 어려운 알고리즘이다.(참조 순서를 미리 알 수 없음)

1. Least Recently used(LRU) Algorithm
   * 과거의 데이터를 보고 가장 최근에 사용한 것을 보존하자
   * 최근에 참조한 것을 보존, 가장 늦게 참조한 것 교체



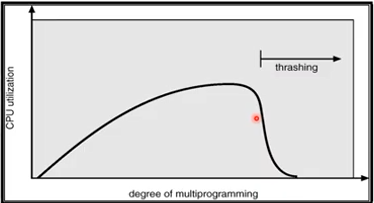
* + 가장 합리적인 알고리즘이며, 실제로 운영체제에서 사용하는 방법임

1. Second Chance
   * FIFO를 베이스
2. Not Recently Used(NRU)
   * LRU와 비슷
3. Counting Algorithms
   * 이전 데이터의 참조 횟수를 보고 어떤 것을 내릴 지 결정
4. LFU Algorithm – 가장 적게 참조한 데이터 교체
5. MFU Algorithm
6. Allocation of Frames
7. Fixed allocation – RR과 비슷
8. Priority allocation – priority scheduling과 비슷, 우선 순위에 따라 할당
9. Global vs Local Allocation
10. Global replacement – 전체 physical memory 내에서 victim frame을 정함(모든 process 내에서)
11. Local replacement – 특정 process내에서 victim frame을 정함.

* 현재 운영체제에서는 local replacement 사용.

1. Thrashing

* Page-fault가 자주 일어나서 이를 처리하면서 성능이 저하되는 문제



* Page-fault가 증가 -> hard-disk에서 데이터를 읽어서 memory에 올려놓는 작업 시간 증가(I/O 작업 시간 증가) -> 작업 지연
* CPU가 Thrashing 문제를 해결하느라 다른 process의 수행이 불가능하다. 따라서 프로세스를 하나씩 죽여야 한다.
* 하지만 최근에는 SSD 사용 증가로 thrashing 현상이 감소하는 추세이다.(SSD는 전기적 신호로 데이터를 쓰고 읽는 방식)

9. Other Considerations(또 다른 메모리 관리 기법)

Prepaging – page를 올려놓을 때 주변에 있는 것 까지 같이 올려놓음(spatial locality에 기반)

-Page size selection – page 사이즈를 설정.

-TLB Reach