[Chapter 10]

경영학과/2018100233/김정민

I. Virtual Memory

A. Virtual memory의 개념

1. 실제 피지컬 메모리보다 더 많은 공간을 메모리로 사용하는 것처럼 느끼게 해준다.
2. Secondary storage(disk)의 일정 부분을 메모리로 간주해서 사용한다.
3. 프로세스 시작할 때 메모리에 하나도 안 올리고 시작한다. (디스크를 메모리로 간주하므로)
4. Page table에 valid/invalid bit를 포함한다.
   - CPU가 페이지 테이블에 접근했는데 invalid일 경우에 disk에서 메모리로 올려준다. (Swap)

B. Demand Paging

1. 필요한 것만 피지컬 메모리에 할당한다.
   - swapping 개념 활용
   - 디스크에서 필요한 것들만 피지컬 메모리로 할당
2. 프로세스 단위가 아니라 '페이지 단위'로 swap이 일어나는 paging system
   - 필요한 페이지만 디스크에서 피지컬 메모리로 할당 (swap in)
   - 피지컬 메모리가 꽉 찬 상태로 다른 페이지가 할당되어야 할 때, 안 쓰는 메모리를 디스크로 swap out
3. OS는 메인 메모리를 프로세스에 할당된 모든 데이터의 캐시로 활용한다.
   - 처음엔 페이지들이 피지컬 메모리 프레임들로부터 할당된다.
   - 피지컬 메모리가 채워지면, 새로운 페이지를 할당하려면 기존에 존재하던 프레임을 victim으로 만들어야 한다.
4. Victim이 된 페이지들은 disk로 swap out된다.
   - dirty 페이지(메모리 내용이 변경된 페이지)는 기존 디스크의 내용과 달라졌으므로 디스크에 덮어써야 한다.
   - 이러한 페이지의 swap 과정은 OS에 의해 관리된다. 이것도 I/O 작업이다.
   - Transparent to the application
     - 애플리케이션은 피지컬 메모리, 디스크에 있는지 알 수 없다.
     - OS가 abstraction을 통해 관리해주기 때문에, 겉으로 보기에는 알 수 없다.

5. Locality : 페이징이 어느 부분에 집중된다.

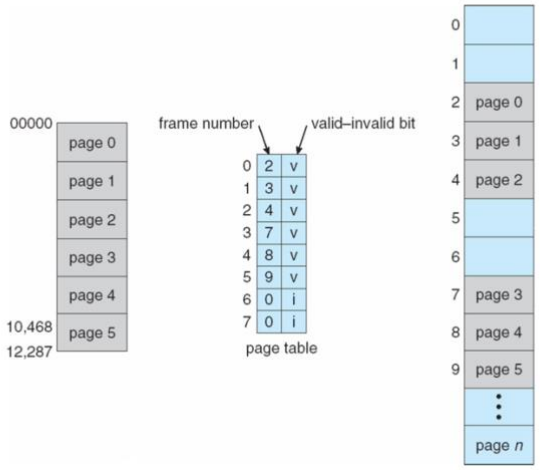
- 캐시의 개념 -> 자주 쓰이는 것을 빠른 매체에 두어 더 느린 매체를 통할 필요가 없게끔 한다.
- Temporal locality : 최근에 참조된 메모리는 또 참조될 가능성이 높다.
- Spatial locality : 최근에 참조된 메모리의 근처 메모리들은 조만간 참조될 가능성이 높다.
- 한 번 페이지를 피지컬 메모리에 올리면, 해당 프레임은 많이 사용될 가능성이 높다.
- 평균적으로, 이미 page in 된 메모리를 사용한다.
- 여러 가지 요소가 영향을 끼친다.
  - locality 정도
  - page replacement policy
  - 피지컬 메모리의 크기
  - 메모리 참조 패턴
  - 한 프로세스가 피지컬 메모리에서 얼마나 차지하고 있는지

6. "Demand" paging인 이유?

- 프로세스가 처음 시작될 때, 모든 엔트리(PTE)가 invalid 상태인 페이지 테이블을 가진다.
- 피지컬 메모리에 아무것도 할당되지 않은 상태
- 실행하면 처음에는 무조건 인스트럭션은 코드나 데이터 페이지에서 Fault 발생한다.

- 모든 필요한 페이지가 메모리에 들어가면 fault 발생하지 않는다.
- 프로세스 실행에 필요한 코드나 데이터는 피지컬 메모리에 로드되어야 한다.
- 필요한 페이지는 계속해서 바뀐다.

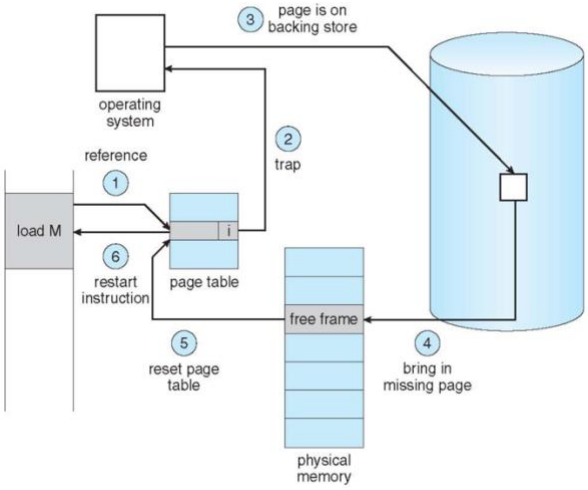
7. Valid-invalid bit in demand paging
   - demand paging이 아닌 그냥 paging인 경우
     - if valid-invalid bit == i
       - ✓ Protection fault



     -
     - valid bit가 invalid인 경우에 protection fault가 발생한다.
     - 해당 메모리에 접근할 수 없다. (protection을 위해)
   - demand paging인 경우
     - valid-invalid bit가 v인지 i인지에 따라서 MMU가 맞는 처리를 해준다.
       - v -> MMU가 address translation 수행
     - i인 경우 (MMU가 스스로 인터럽트 발생시킨다.)
       - protection fault (not-in-memory) : 접근하면 안되는 메모리일 때
       - page fault : 아직 디스크에서 피지컬 메모리로 로드가 되지 않은 상태
       - 두 경우에 MMU가 스스로 인터럽트를 발생시켜서 OS가 핸들러를 수행한다.
       - 하드웨어적으로는(MMU) 두 경우를 구분할 수 없기 때문에, OS가 판단해준다.

8. Page fault
   - 페이지 테이블에서 invalid인 엔트리의 로지컬 주소를 통해 접근할 때, 아직 로드가 안된 경우다. -> page fault
   - 페이지가 피지컬 메모리에서 내려갔을 때, OS는 해당 PTE를 invalid로 바꾸고, swap file에 해당 위치를 저장한다.
   - 프로세스가 invalid PTE에 접근하면, exception을 날린다.
   - OS는 page fault 핸들러를 수행한다.
     - 핸들러는 invalid PTE를 swap file에 위치시킨다.
     - 핸들러는 페이지를 피지컬 메모리 프레임에 넣고, PTE를 valid로 바꾼다.
     - 이후에 핸들러는 실패한 프로세스를 다시 수행한다. (다시 읽어와야 한다.)
   - 접근된 페이지는 어느 프레임으로 들어가는가?
     - 한 프레임은 희생당해야 한다.
     - OS는 free page 목록을 관리하려고 노력한다. (프레임을 최대한 희생시키지 않기 위해)

9. Steps in Handling a Page Fault



-
- 1. invalid PTE에 접근
- 2. trap (SW 인터럽트) 발생
- 3. (OS 개입) 페이지가 디스크에 있는지 확인
- 4. 해당 페이지를 피지컬 메모리로 가져온다. (free frame이 있다면 그 위치로)
- 5. 페이지 테이블 수정 (PTE -> valid)
- 6. 인스트럭션을 다시 수행

C. Memory Reference

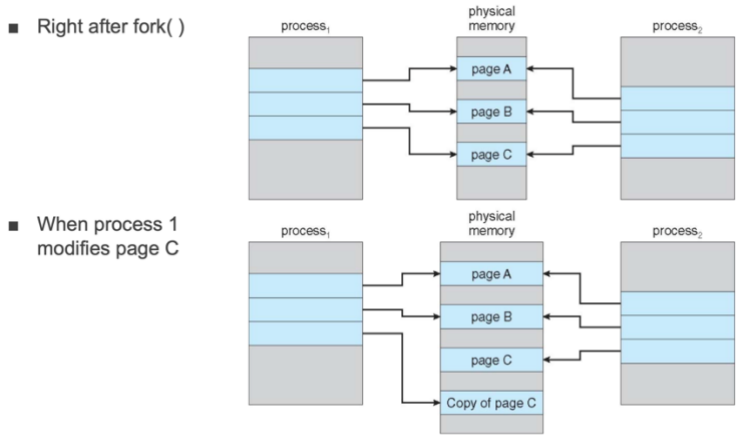
1. 프로세스가 CPU 위에서 연산을 수행하고 있을 때 + virtual address(VA)를 통해 메모리에 접근할 때
2. 일반적인 경우
   - MMU 안에 있는 TLB에 접근해서 VA의 PN에 해당하는 페이지가 있는지 확인한다.
   - PN이 있다면 PTE를 리턴한다.
   - TLB는 PTE protection을 확인해서 읽을 수 있는지 검증한다.
   - PTE는 어떤 프레임(피지컬 메모리)이 해당 페이지를 담고 있는지 알려준다.
   - MMU는 피지컬 프레임과 오프셋을 조합해서 피지컬 주소를 계산한다.
   - MMU는 해당 피지컬 주소로부터 읽어서 그 값을 CPU로 전달해준다.
3. TLB misses (특이한 2가지 케이스)
   - 페이지 테이블에 탐색하는 valid PTE가 있다는 가정.
   - 1. MMU가 페이지 테이블의 PTE를 TLB에 로드한다. (HW)
     - 페이지 테이블에 대한 정보를 디스크로 내리지 않게끔
     - TLB는 하드웨어적으로 관리되고, OS는 이 과정에서 관여하지 않는다.
     - OS는 이미 페이지 테이블을 관리하고 있으므로, 하드웨어적으로 페이지 테이블에 직접적으로 접근해서 정보를 가져온다.
   - 2. OS로 트랩(SW 인터럽트)을 날린다.
     - 소프트웨어적으로 TLB를 관리하는 경우
     - OS가 예외 처리를 해준다.
     - OS는 페이지 테이블에서 PTE를 찾아서 TLB에 로드한다.
     - OS는 예외처리를 끝내고 인스트럭션을 다시 수행할 수 있도록 TLB를 다시 진행시킨다.
   - 페이지 테이블에서 탐색할 때 페이지 테이블이 paged out(디스크에)이 되어 있다면 재귀적인 fault가 발생할 수 있다.
     - 페이지 테이블이 TLB에 없고, 페이지 테이블 자체도 피지컬 메모리에 없는 상황
     - 페이지 테이블이 디스크에 있으면, 페이지 테이블을 탐색할 때마다 TLB miss가 발생한다.

- 페이지 테이블이 피지컬 메모리 안에 있으면 문제가 되지 않는다.

- TLB가 PTE를 가지고 있다면, address translation을 다시 시작한다.
  - 일반적인 경우는 PTE가 메모리 안에 있는 valid 페이지를 가리킨다.
  - 일반적이지 않은 경우는 PTE protection bits, valid bit 등 때문에 PTE에 접근하지 못하고 TLB miss가 발생할 수 있다.

4. Page faults

- PTE는 protection fault를 나타낼 수 있다.
  - read/write/execute : 페이지에 대해 해당 작업을 수행할 권한이 있는지?
  - invalid : 해당 페이지가 피지컬 메모리에 없거나(paged out), virtual page가 할당되지 않은 상태
- TLB가 OS로 트랩을 날린다. (SW)
  - read/write/execute : OS는 보통 프로세스로 다시 fault를 보내 알려주거나, 트릭을 사용한다. (copy on write, mapped files)
  - invalid (Not allocated) : VA에 해당하는 페이지가 아직 할당되지 않았으면, OS는 해당 프로세스로 오류를 알리기 위해 트랩을 보낸다. (segmentation fault)
    - PTE가 만들어지지 않았거나, 페이지가 디스크에도 올라가있지 않은 경우
  - invalid (Not in physical memory) : VA에 해당하는 페이지가 피지컬 메모리에 존재하지 않는다면, OS는 해당 페이지를 디스크에서 읽어와서 PTE를 업데이트하고 피지컬 프레임과 연결한다.
    - 페이지가 디스크에는 올라가있는 경우
- copy on write



  - fork() 직후에는 모든 데이터, 코드는 두 프로세스가 완전히 같다. 메모리 따로 할당할 필요 없다. (Demand paging)
  - PCB는 각각 가진다.
  - 만약 process 1이 page C에 대한 내용을 수정했으면, page C를 복사하고 복사본에 대해서 수정한다.

II. Page Replacement

A. Victim을 선정해야 할 때가 있다.

1. Page fault가 발생하면, OS는 실패한 페이지를 디스크로부터 피지컬 메모리 프레임에 할당해준다.
2. 어느 순간에는 프로세스가 피지컬 메모리 프레임들을 모두 사용할 수도 있다.
3. 이렇게 되면 OS는 실패하는 페이지와 이미 올라가있는 페이지 하나(victim)를 바꿔야 한다.

B. Page replacement algorithm

1. Best victim을 선정하는 것이 중요한 이유

- Fault rate을 낮출 수 있다.
- 한 번도 참조되지 않은 페이지가 가장 희생되기에 좋다.

- 최대한 적게 참조된 페이지가 현실적으로 좋다.
    - Belady's proof : 가장 긴 시간 동안 사용되지 않을 페이지를 희생하는 것이 page fault 수를 줄인다.

2. Page replacement가 발생할 때 OS가 하는 일의 순서
    - 1. Victim 페이지를 swap out
    - 2. 페이지 테이블에서 해당하는 페이지를 invalid로 바꾼다.
    - 3. 접근하고 싶은 디스크 페이지를 피지컬 메모리로 swap in
    - 4. 페이지 테이블 변경 정보를 갱신 (TLB도 갱신)
3. Page replacement 개념이 포함된 Demand Paging 성능 (EAT)
    - Page Fault Rate p, $0 \le p \le 1$
    - Effective Access Time (EAT)

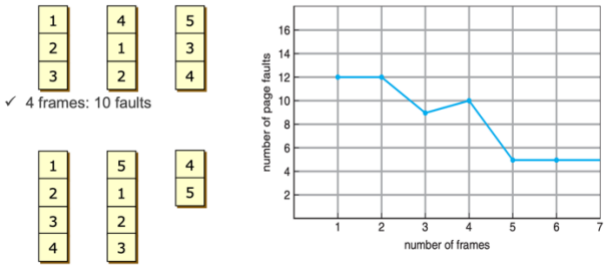
$$\begin{aligned} EAT = {} & (1 - p) \times \text{memory access} \\ & + p \times (\text{page fault overhead} \\ & + [\text{swap page out}] \\ & + \text{swap page in} \\ & + \text{restart overhead}) \end{aligned}$$

    -
    - (1-p) x memory access : page fault 없으면 TLB를 통해 Memory access 가능
    - page fault overhead : 페이지 폴트가 발생하면 생기는 오버헤드 (페이지 테이블 업데이트 등)
    - [swap out] : 상황에 따라 다르다. (페이지를 디스크로 보내는 I/O 시간)
    - swap in : 디스크에서 가져오는 I/O 시간
    - restart overhead : 처리 후에 다시 인스트럭션을 실행하는 데 생기는 오버헤드
4. 알고리즘의 목표 : page-fault rate을 최대한 작도록
    - 프레임의 수를 무작정 늘린다고 효과적이지 않다.
        - 어느 수준 이상으로 가면 프레임 수를 늘려도 page fault가 개선되는 정도가 적다.
        - 반복구조가 상당히 많기 때문에 썼던 메모리에 계속 접근할 확률이 높다.

C. FIFO

1. 구현하기가 가장 간단하다.
    - 페이지가 피지컬 메모리에 로드된 순서(리스트)를 유지해야 한다.
    - 가장 오래전에 로드된 페이지를 희생시킨다.
2. 장점
    - 가장 오래전에 로드된 페이지는 아마도 안 쓰이고 있을 것이다
3. 단점
    - 가장 오래된 페이지를 반복해서 사용할 수도 있는데, 이런 정보를 알 수 없다.
    - optimal하지 않다.
4. Belady's Anomaly
    - 알고리즘이 더 많은 피지컬 프레임을 늘려줘도 fault rate이 증가할 수도 있다.

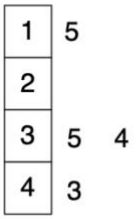
- Example: Belady's anomaly
  - ✓ Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - ✓ 3 frames: 9 faults



-

D. Least Recently Used(LRU) Algorithm

1. Optimal Algorithm : 가장 오랫동안 사용되지 않을 페이지를 대체해야 한다.
   - 미래는 알 수 없다. 과거를 이용해보자.
   - 위에 나온 4 frames example로 알고리즘 성능 측정 가능
2. 더 좋은 대체를 하기 위해 참조해야 할 정보가 있다.
   - 개념 : 과거의 경험을 통해 미래를 추측한다.
   - '과거'에 가장 오랫동안 쓰이지 않은 페이지를 대체한다.
   - LRU는 과거를 보고 결정을 내린다.
3. 4 frames example
   - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



-
   - 총 4+3+1 = 8번의 page fault 발생

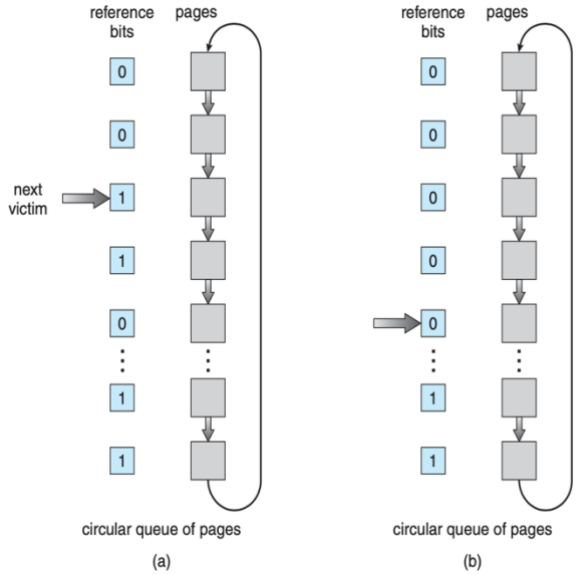
E. LRU 구현 방법

1. Timestamp 구현
   - 모든 페이지 엔트리는 counter를 가진다.
   - 모든 타임 페이지는 해당 엔트리를 참조한다. (카운터에 시간 정보를 복사한다.)
   - 페이지가 변경되어야 하면, 카운터(timestamp)를 보고 가장 오래된 페이지를 내린다.
2. Stack 구현
   - PN에 대한 스택을 유지한다.
   - 스택에 있는 페이지가 접근되면, 해당 페이지 넘버를 스택의 맨 위로 올린다.
   - 어떤 페이지를 내려야할지 검색할 필요가 없다.
3. Approximation
   - 완벽하게 replacement를 하기 위해서는 모든 참조마다 해당 PTE에 정확한 timestamp 정보를 저장하거나, 스택을 유지관리해야 한다.
   - 비용이 너무 비싸다.
   - 따라서 approximation을 이용한다.
4. LRU Approximation Algorithms
   - Reference bit
     - 각 페이지마다 비트를 하나 추가

- 처음엔 0, 한 번 참조되면 1로 업데이트
- reference bit가 0인 페이지 중 랜덤하게 대체한다.
- 그런데 reference bit가 0인 페이지들 중에서도 오래된 페이지, 방금 0이 된 페이지 등의 구분이 있는데, 이걸 고려 못한다. -> second chance (or LRU clok) 활용

5. Second chance (or LRU clock)

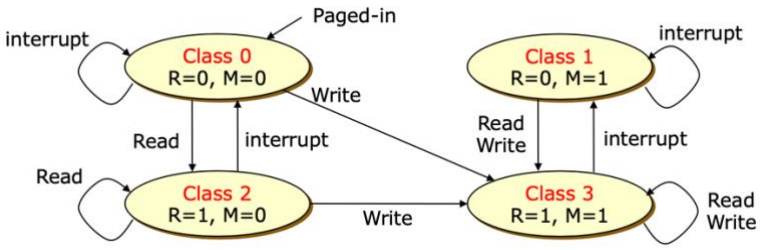
- 마찬가지로 reference bit 필요하다.
- clock replacement (clock order)
- clock order에 따라서 대체되어야 할 페이지의 reference bit가 1이라면,
  - reference bit를 0으로 바꾸고
  - 해당 페이지는 메모리에 두고, clock order 상의 다음 페이지를 대체한다.
  - 한 번 더 기회를 주는 것이다.
- clock order 순으로 페이지를 정렬할 circular queue를 유지해야 한다.



- circular queue 탐색하는 과정도 결국 오버헤드가 크다.

6. NRU (or enhanced second chance)

- R (reference), M (modify) bits (비트 1개 추가)
- 주기적으로 R 비트는 초기화된다.
  - R비트가 1인 페이지 중에서 최근에 참조되지 않은 페이지를 구별하기 위해
- LRU보다는 성능이 낮다. (optimal하지 않다.)
  - 그런데 approximation은 구현 관점에서 쓰기가 어렵다.
- 상태처리도

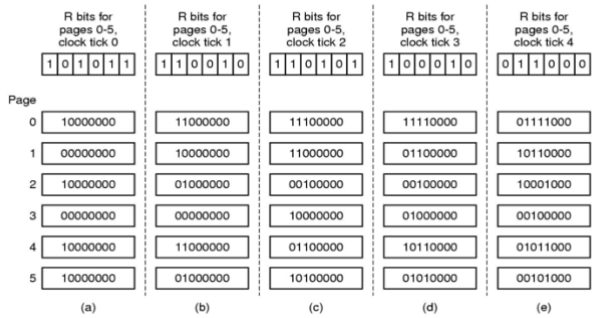


- class 0 : 우선순위 제일 낮음, Read, Write이 일어나지 않은 상태
- class 1 : Read, Write이 모두 일어난 class 3에서 주기적으로 인터럽트를 통해서 R bit가 0이 된 상태
  - 한 번 write하면, 다시 접근해서 read할 확률이 높다. (locality) 따라서 class 0보다 높은 우선순위를 가진다.

- class 2 : class 0으로부터 참조가 수행된 상태
- class 3 : Read, write이 모두 최근에 일어난 상태
- R 비트는 인터럽트를 통해 주기적으로 1-> 0으로 초기화되는 반면에, M 비트는 1 -> 0이 되는 케이스가 없다.

- 낮은 클래스에 해당하는 페이지부터 랜덤으로 디스크로 내린다.
- M=1, R=0인 class 1(적어도 한 클럭 동안 참조되지 않은 페이지)을 지우는 것이 R=1, M=0인 class 2(수정사항 없지만 참조가 자주 되는 페이지)를 지우는 것보다 좋다.
- 장점
  - 이해하기 쉽다.
  - 구현에 있어서 꽤 효율적이다.
  - 완전한 optimal은 아니지만 괜찮은 성능을 보인다.

7. Least Frequently Used (LFU)
   - Counting-based page replacement
   - 소프트웨어 카운터는 각 페이지마다 존재한다.
   - 클럭 인터럽트마다(타이머 인터럽트와 비슷) 각 페이지는 R bit를 카운터에 더한다. (최근에 참조되었으면 1 더하기 연산)
     - 카운터는 얼마나 자주 페이지가 참조되는지를 나타낸다.
   - 가장 작은 카운터 값을 가진 페이지가 대체된다.
   - Most Frequently Used (MFU)도 있다.
     - 가장 큰 카운터 값을 가진 페이지가 대체된다.
   - 참조 이력을 가지고 있다.
     - 페이지는 초기에 많이 쓰이고, 나중에 안 쓰일 수도 있다. -> 이런 것까지 고려할 수 있다.
   - Aging
     - 이진수로 구성된 카운터 값(초기값은 0)은 R비트가 맨 왼쪽에 추가되기 전에 오른쪽으로 1자리씩 shift한다.
     -



III. Allocation of Frames

A. Allocation algorithms

1. Equal allocation : 프로세스 10개, 프레임 100개 -> 프로세스당 10개씩 균등분배
   - IDLE 프로세서 같은 busy-waiting이 주인 프로세스의 경우에는 이만큼 필요없을 수도 있다.
2. Proportional allocation : 프로세스의 크기에 따라서 차등분배
   - 꼭 프로세스의 크기가 크다고 프레임이 많이 필요한 게 아닐 수도 있다.
3. Priority-based allocation : 우선순위 기반으로 프레임 할당
4. 보통 2번과, 3번을 조합해서 사용한다.

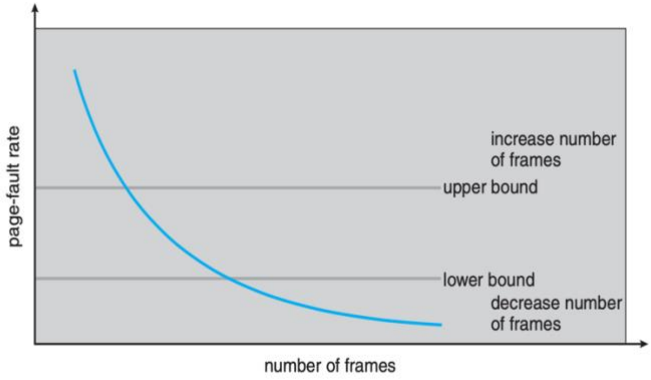
B. Page replacement

1. Global replacement : 페이지를 교체할 때 다른 프로세스의 페이지를 내릴 수도 있다.

2. Local replacement : 프로세스 단위로 페이지 교체가 이루어진다.
   - 페이지 테이블이 이미 존재하기 때문에 프레임 교체가 비교적 쉽다.
   - 다른 프로세스에서 victim을 찾지 않는다.

C. Page-Fault Frequency Scheme

1. 프레임을 상수값으로 분배하는 것은 별로다.
   - 처음에 특정 값으로 설정해보고, page fault가 0에 가까우면 프레임 개수를 하나씩 줄인다.



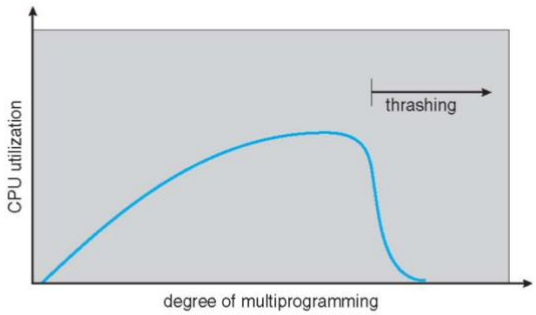
- Establish "acceptable" page-fault rate
  - ✓ If actual rate too low, process loses frame
  - ✓ If actual rate too high, process gains frame

2.
   - page-fault rate의 상한선과 하한선을 설정하고, 그에 맞게 프레임의 수를 조절해준다.
   - fault-rate가 너무 낮으면, 프레임이 과하게 할당된 것이므로, 감소시킨다.
   - fault-rate가 너무 높으면, 프레임이 부족하게 할당된 것이므로, 증가시킨다.

D. Thrashing

1. 프로세스가 페이지를 swap in/out하느라 바쁜 상태
2. Thrashing 발생 이유?
   - 프로세스마다 필요한 locality의 총량 (시스템 전체의 locality)가 피지컬 메모리 크기보다 클 때 발생한다.

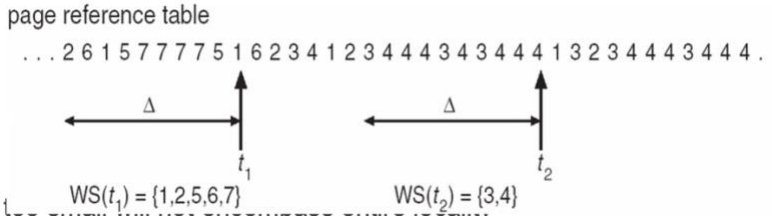
- Why does thrashing occur?
  - ✓ Σ size of locality > total memory size



   -
   - 프로세스가 증가하면, locality 총량이 피지컬 메모리보다 커지게 된다.
   - 그러면 page replacement(I/O)가 계속 발생하므로 CPU utilization이 뚝 떨어진다.

E. Working-Set Model

1. Locality 총량을 계산하기 위한 모델
2. Locality D = 모든 프로세스의 Working set size의 합 = 필요한 프레임의 수
   - WSS of Process Pi : 가장 최근 단위시간 동안 참조된 페이지의 개수
   - 단위시간 (델타) : working-set window = 고정된 참조 횟수



- Δ t[illegible]
- D 집합의 크기가 메모리 사이즈보다 크면, Thrashing이 발생한다.
- 그러면 page replacement(I/O)가 계속 발생하므로 CPU utilization이 뚝 떨어진다.

IV. Other Considerations

A. Prepaging

1. 최근에 접근된 페이지 근처 페이지까지 미리 가져온다.
2. 프로세스가 최초로 시작될 때 사용하면 좋다.

B. Page size selection

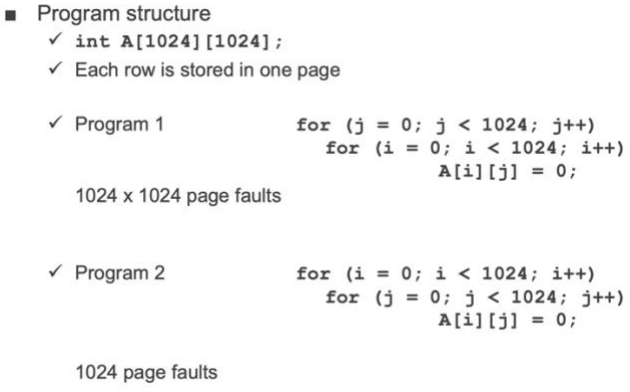
1. (Internal) Fragmentation
   - Fragmentation이 크면 -> 페이지 사이즈 증가 -> 페이지 엔트리 수 감소
2. Page table size
   - 페이지 테이블 사이즈가 크면 -> 페이지 테이블 엔트리 수 증가
3. I/O overhead
   - 페이지 사이즈를 너무 작게 설정하면, page replacement가 증가
4. Locality
   - 페이지 사이즈를 작게 설정할수록 locality 특성이 감소한다.
5. 협의점 : page size 4KB

C. TLB Reach

1. TLB hit을 통해 접근할 수 있는 메모리의 양
2. TLB Reach = TLB Size * Page Size
   - TLB Reach를 증가시키려면
     - TLB의 크기 증가
     - page entry 하나의 사이즈를 늘리면 커버할 수 있는 메모리의 총량이 늘어난다.
     - cf. multiple size page -> multiple size frame
3. 각 프로세스의 working set(최근 단위시간 동안 참조된 페이지)가 TLB에 저장된다.
   - 그렇지 않으면 page fault가 높은 확률로 발생한다.

D. Program structure

1. 이중 for문을 통해 2차원 배열을 탐색할 때, arr[행][열]로 하는 이유? -> page fault, replacement를 감소시키기 위해
2. arr[열][행]으로 해도 모든 배열을 탐색하는 것은 가능하다.
   - 그런데 한 행이 하나의 페이지를 저장한다고 할 때, page fault가 제곱으로 발생한다.



-

E. I/O Interlock

1. DMA (Direct Memory Access) : CPU를 거치지 않고 I/O 장치에서 바로 메모리에 데이터를 쓴다.
   - I/O 장치는 로지컬 메모리를 모른다.
2. 이렇게 디스크에서 피지컬 메모리로 올라온 페이지는 올라오자마자 page replacement되면 안된다.
3. 그래서 DMA하는 동안 Buffer로 묶어서 victim이 되지 않게 페이지에 Lock을 걸어놓는 것이다.