9. Virtual Memory

2023.07.11 **김하림, 신석철, 박영서**

Index

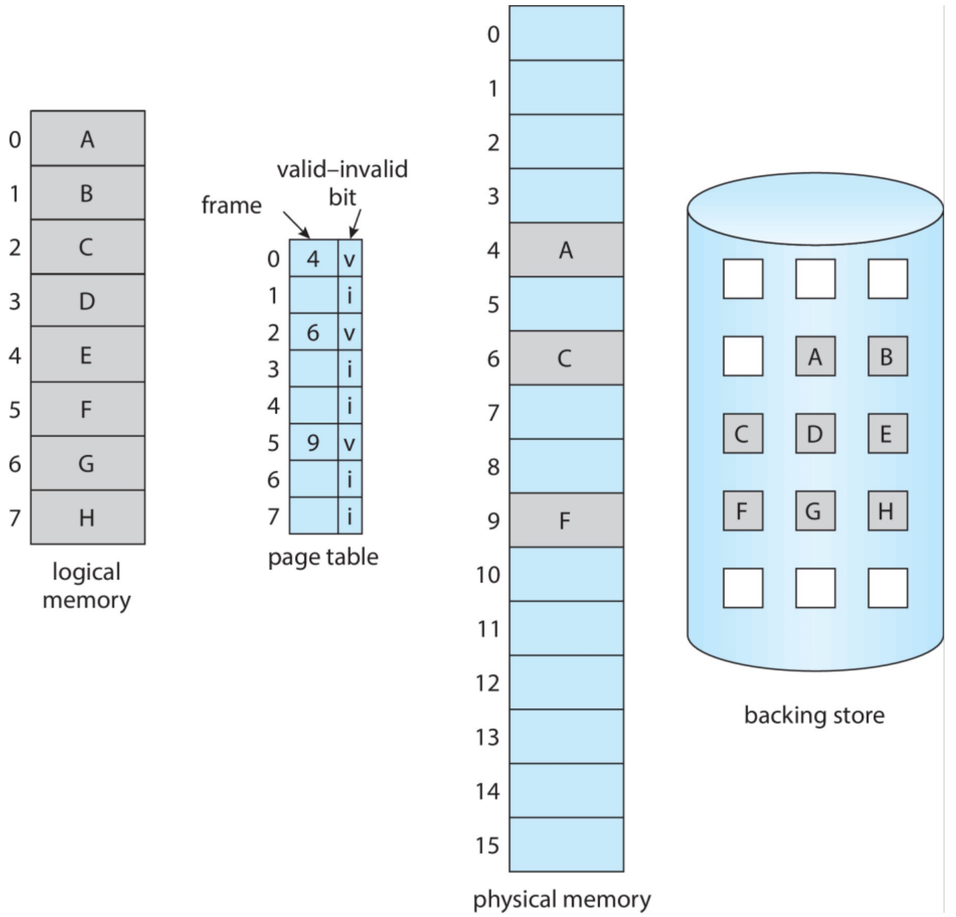
01 Demand Paging

- 현대 대부분의 시스템은 Paging 사용중
- Demand Paging : 실제로 필요할 때 page를 메모리에 올리는 것

- **Demand Paging의 장점**
  1. Disk I/O 감소
  2. Memory 사용량 감소
  3. 빠른 응답 시간 → 한정된 메모리 공간을 더 잘 쓰기 때문, Disk I/O 감소 등의 영향
  4. 더 많은 프로그램, 사용자 수용

01 Demand Paging

- Valid/Invalid bit 사용
  - Invalid
    1. 사용되지 않는 주소 영역
    2. 페이지가 물리적 메모리에 없는 경우
  - 초기에는 모든 Page entry가 invalid로 초기화
  - **Page fault** : address translation했는데 invalid인 경우



01 Page Fault

- Invalid page에 접근하면 MMU가 **page fault trap** 발생시킴
- Kernel mode로 들어가서 page fault handler가 invoke됨

- Page fault 처리 순서
    1. Invalid reference? (eg. bad address, protection violation) → abort process
    2. 빈 page frame을 가져온다. 없으면 다른 page frame을 뺏어온다. (page replacement - Swap out)
    3. 해당 페이지를 디스크에서 메모리로 읽어온다 (Disk I/O - Swap in)
        1. 디스크 I/O가 끝나기까지 이 프로세스는 CPU를 preempt당함 (process state: blocked)
        2. Disk read가 끝나면 PTE에 frame number 기록 후 valid로 set
        3. ready queue에 프로세스를 추가 → dispatch later
    4. 프로세스가 CPU를 잡고 다시 running

01 Page Fault

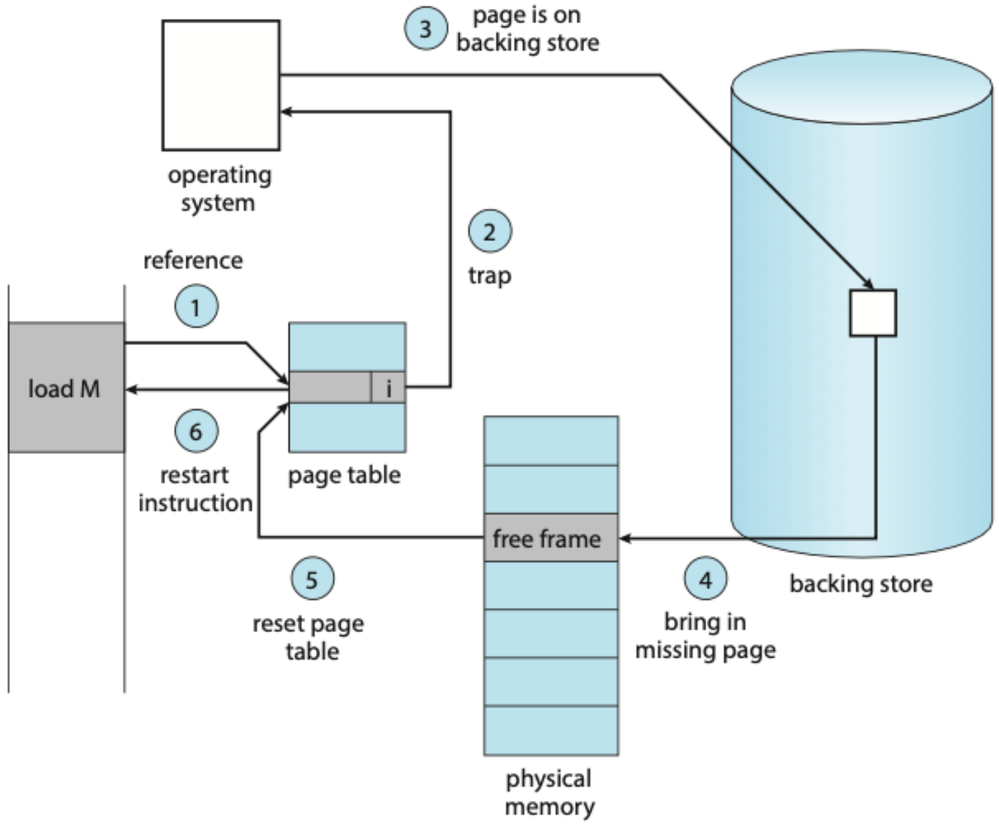


Figure 10.5 Steps in handling a page fault.

01 Performance of Demand Paging

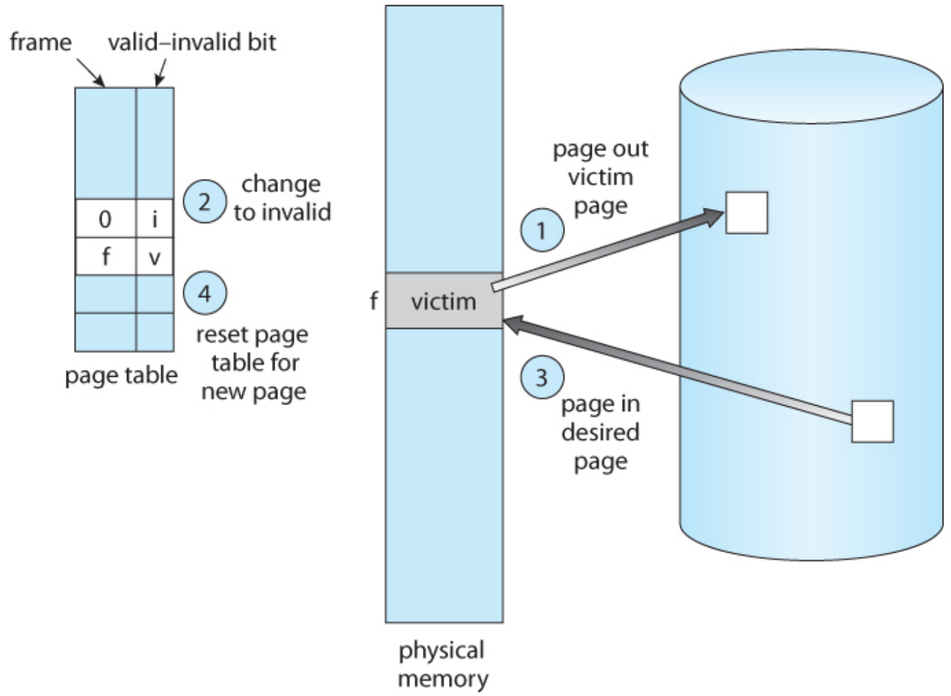
- Page fault rate $0 \leq p \leq 1$
  - p = 0, no page faults
  - p = 1, every reference is a fault

- Effective Access Time = $(1 - p) \times$ memory access
  $+ p \times$ (OS & HW page fault overhead +
  [swap page out if needed] +
  swap page in + OS & HW restart overhead)

02 Replacement Algorithm

- Page replacement
  - 어떤 frame을 빼앗아올지 결정해야 함
  - 곧바로 사용되지 않을 page를 쫓아내는 것이 좋음
  - 동일한 페이지가 여러 번 메모리에서 쫓겨났다가 다시 들어올 수 있음

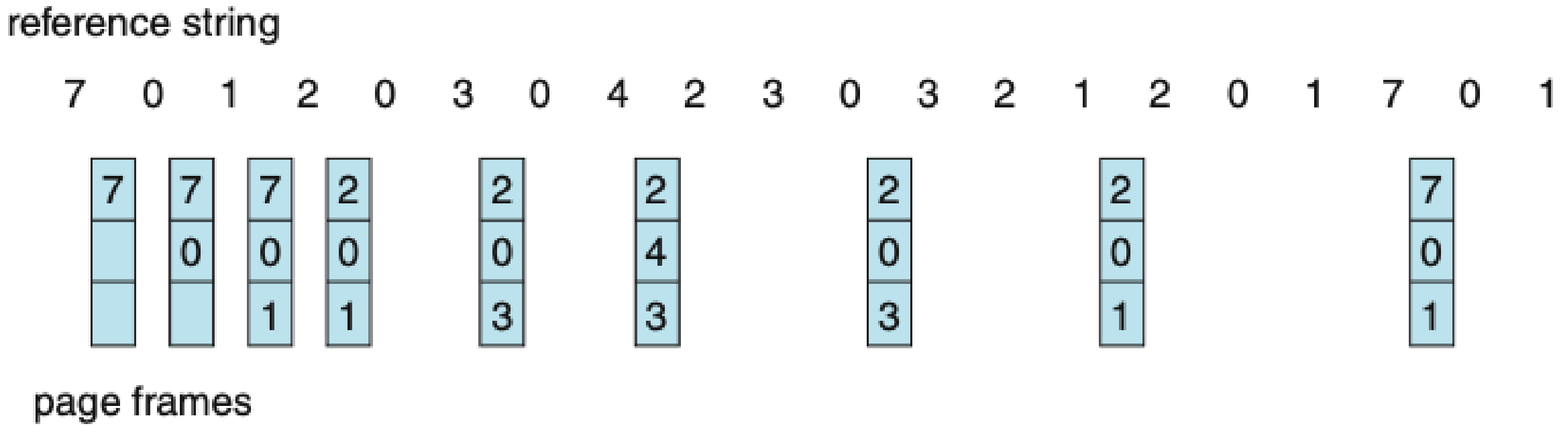
- Replacement Algorithm
  - page-fault rate을 최소화하는 것이 목표
  - 주어진 page reference string에 대해 page fault를 얼마나 내는지
  - reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3



02 Optimal Algorithm

- MIN(OPT): **가장 먼 미래에 참조되는 page**를 replace
- 다른 알고리즘의 성능에 대한 upper bound 제공
- Belady's optimal algorithm, MIN, OPT 등으로 불림
- Offline algorithm: 미래의 참조를 아는 알고리즘

- 4 frames example



02 Advanced Question

- 석철: page fault 과정에서 cpu의 제어권이 변경 되는 과정이 궁금합니다. page fault로 인해 디스크에서 메모리로 데이터를 옮겨야 한다면, 그 긴 시간 동안 cpu의 제어권을 여전히 os가 가지고 있는 것인지 궁금합니다. 만약 그렇다면 데이터가 이동하는 시간 동안 cpu의 제어권을 다시 프로세스가 잡게 가능한지 또 그 방법이 더 효율적일지도 궁금합니다?

- page fault로 인한 디스크 I/O는 시간이 오래 걸림. 따라서 OS는 해당 프로세스(page fault가 일어난)의 CPU 제어권을 회수해서(preempt) ready queue에 있는 다음 프로세스에게 CPU 제어권을 부여함. Disk I/O 시간동안 해당 프로세스에게 계속해서 CPU 제어권을 주는 것은 전체적인 CPU 사용률을 떨어뜨리므로 다른 프로세스가 잡게 하는 것이 더 효율적임

02 Advanced Question

- 영서: Free frame이 없는 경우, 현재 메모리 상의 페이지는 swap out, backing storage에 있는 페이지는 swap in 해야 하기 때문에 data transfer가 두 번 일어나야 한다. 이를 더 개선시킬 수 있는 방안이 무엇인지 한 가지 제시하고 설명하시오.

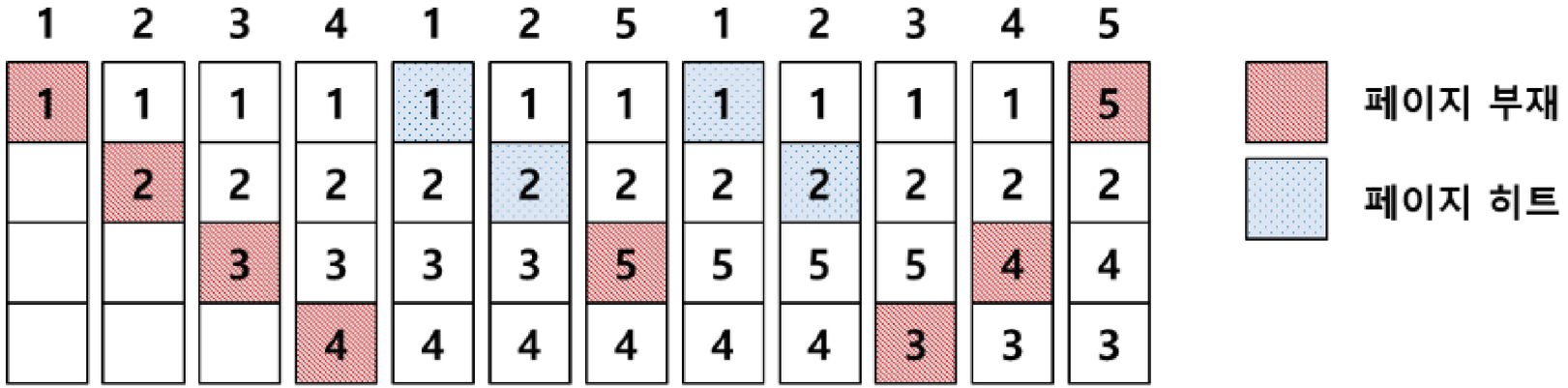
- modified bit = 0일 경우 disk I/O 없이 memory에서만 삭제하면 됨(내용이 동일하므로)

03 FIFO (First In First Out) Algorithm

- 먼저 들어온 것을 먼저 내쫓는 알고리즘

FIFO Anomaly : frame의 수를 늘렸을 때 더 많은 page fault 가 발생하는 현상

03 LRU (Least Recently Used) Algorithm) Algorithm



- 가장 오래 전에 참조된 것을 지움

page가 언제 사용되었는지에 대한 시간 정보가 있어야 하고, 이에 대한 구현 방법으로는 counter와 stack 이있다

03 Q.영서

Belady’s anomaly가 일어나지 않는 page-replacement algorithm에는 stack algorithm이 있다. 이에 대해 설명하시오. (Operating System Concepts pp. 408 - 409에 있음)

-> Belady’s anomaly : 사용하는 page frame이 많아질수록 가지고 있는 page의 개수가 많기 때문에 사용하려는 페이지가 없는 page fault 문제가 덜 발생할 것이라고 예측하지만 그렇지 않은 경우가 있다

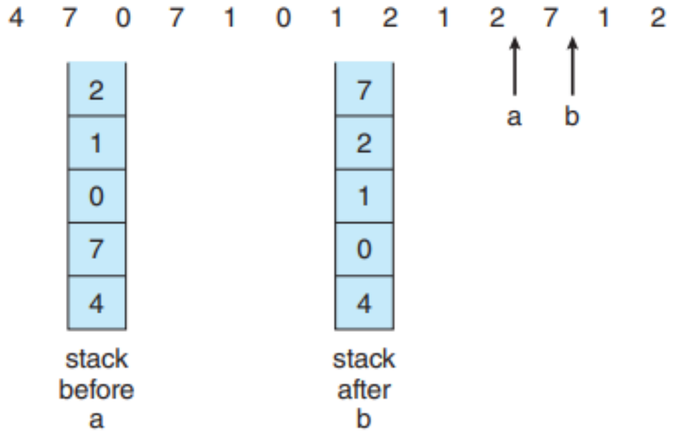
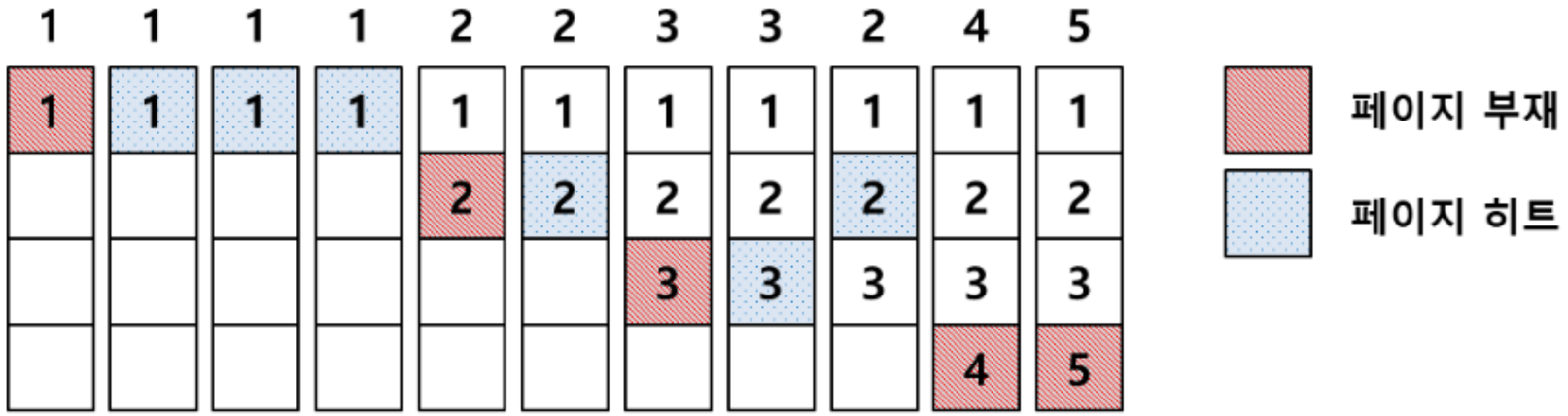


Figure 10.16 Use of a stack to record the most recent page references.

- 들어오는 페이지들을 stack에 쌓아두다가, stack에 있는 페이지가 들어오면 기존에 있던걸 빼고 새로 위에 쌓는다.

- stack이 다 찼는데 새로운 페이지가 들어오면, 가장 밑에 있는 페이지를 빼고 쌓는다.

03 LFU (Least Frequently Used) Algorithm

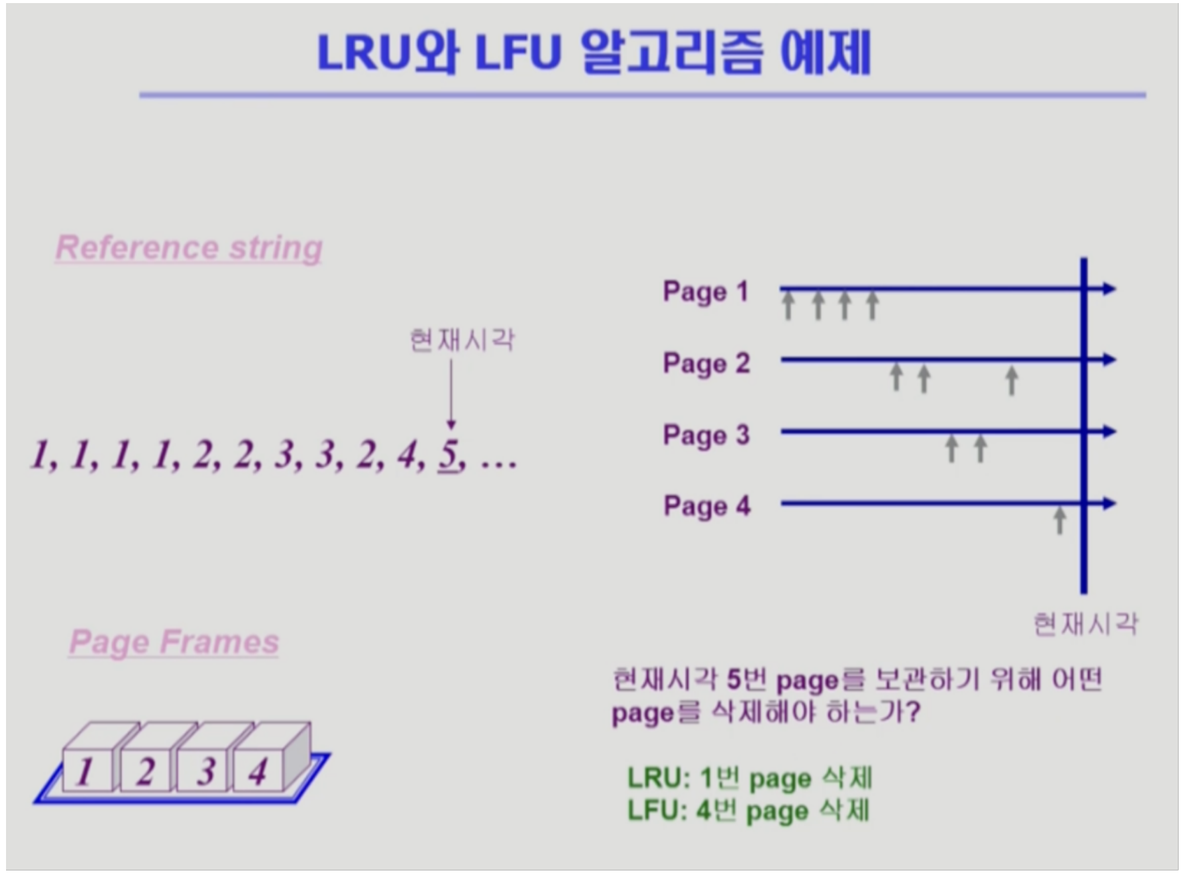


- 참조 횟수가 가장 적은 페이지를 지움

-최저 참조 횟수인 page가 여럿 있는 경우

- LFU 알고리즘 자체에서는 여러 page 중 임의로 선정
- 성능 향상을 위해 가장 오래 전에 참조된 page 를 지우게 구현할 수도 있다

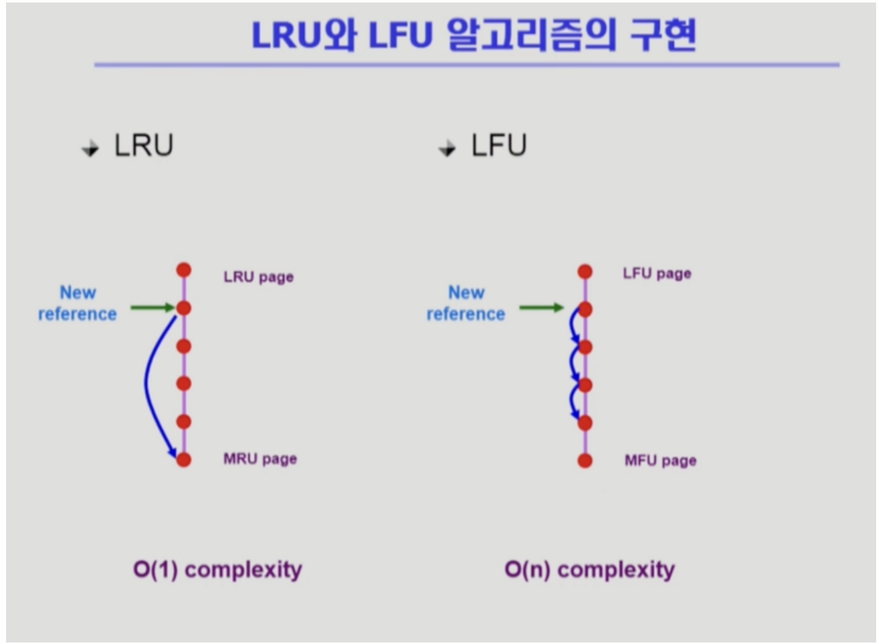
03 LRU와 LFU 알고리즘 예제



5번 page를 보관하기 위해 삭제하는 page

- LRU : 1번 페이지 삭제 (가장 먼저 참조된 페이지)
- LFU : 4번 페이지 삭제 (가장 적게 참조된 페이지)

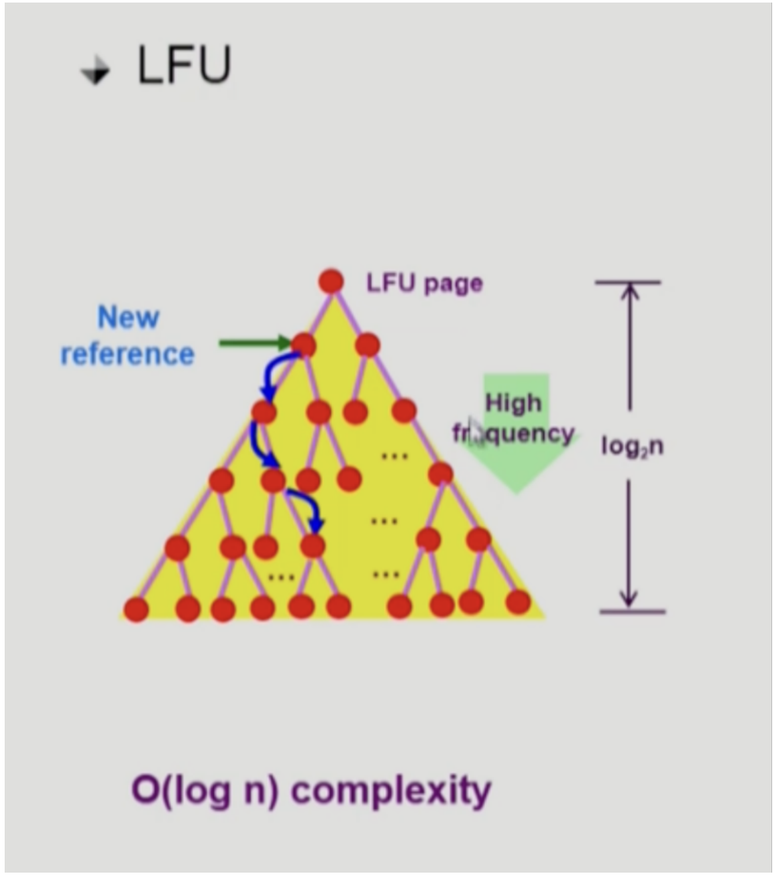
03 LRU LFU 알고리즘 구현



LRU 알고리즘
참조 순서에 대해 줄 세우기를 함 (링크드 리스트)
시간 복잡도 : O(1) 비교가 필요 없음
다시 참조가 된다면 기존의 값을 줄의 가장 끝으로 보내면 되고
자리가 없는 상태에서 새롭게 참조되는 값이 있다면 제일 상단에
있는 값을 지우고 줄 가장 끝에 새로운 값을 추가

LFU 알고리즘
이 알고리즘은 참조의 횟수에 따라 위치가 지정 됨
시간 복잡도 : O(n)
최악의 경우 참조 횟수가 1증가 했는데 최하단까지 비교를 해야
힐 경우 시간 복잡도가 크다 _ O(n)

03 LFU 알고리즘 heap 사용



‘Q.하림 : LRU와 LFU 알고리즘의 구현’ 파트에서 각 알고리즘의 시간 복잡도가 왜 O(1), O(n), O(log n)인지 구체적으로 설명해 주세요

앞서 말한 것처럼 최악의 경우 LFU는 최상단에서 최하단까지 비교를 해야할 경우가 발생할 수 있다 _ O(n)

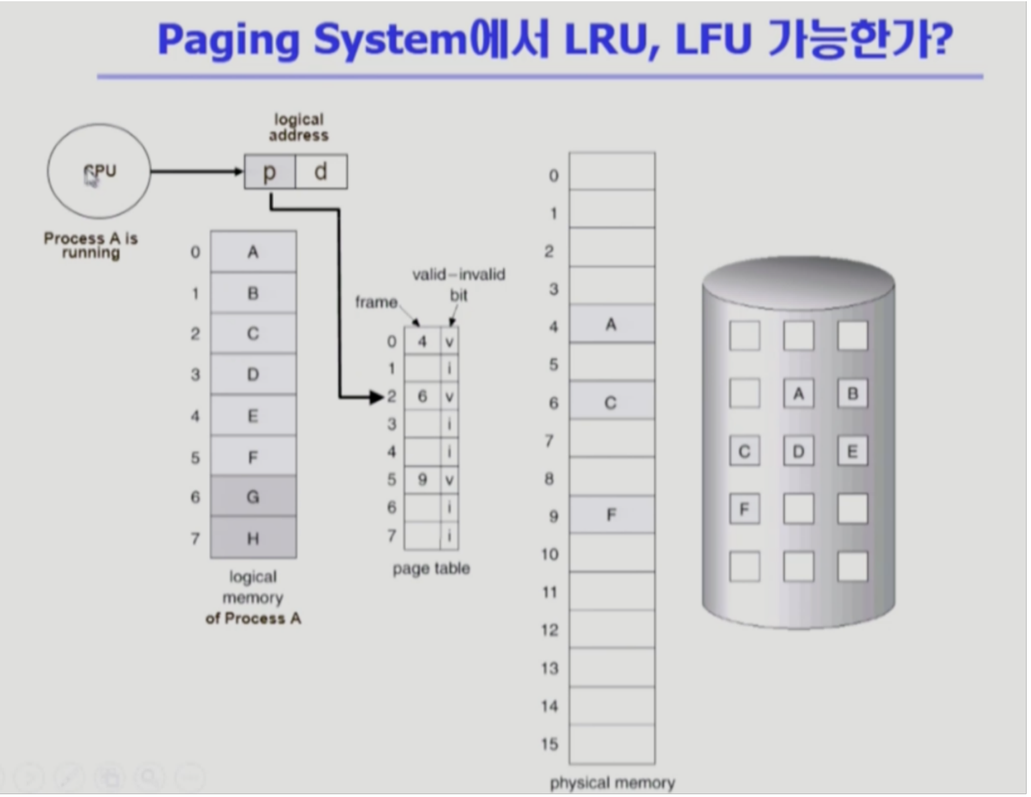
그렇기 때문에 이런식으로 구현하지 않고 heap을 사용한다
heap을 사용할 경우 시간 복잡도를 O(log n)까지 줄일 수 있다

03 다양한 캐싱 환경

- 캐싱 기법
  - 캐쉬에 요청된 데이터를 저장해 두었다가 후속 요청시 캐쉬로부터 직접 서비스하는 방식
  - paging system 외에도 cache memory, buffer caching, Web caching 등 다양한 분야에서 사용

- 캐쉬 운영의 시간 제약
  - 교체 알고리즘에서 삭제할 항목을 결정하는 일에 지나치게 많은 시간이 걸리는 경우 실제 시스템에서 사용할 수 없다
  - buffer caching 이나 web caching의 경우
    O(1)에서 O(log n)정도까지 허용
  - paging system인 경우
    page fault인 경우에만 OS가 관여
    페이지가 이미 메모리에 존재하는 경우 참조시각 등의 정보를 os가 알 수 없음
    O(1) 인 LRU의 list 조작조차 불가능

03 paging system에서 LRU LFU는 가능한가?



page fault가 발생해서 LRU나 LFU 알고리즘을 활용해야 한다고 했을 때 과연 운영체제는 참조된 시점 혹은 참조된 횟수에 대해서 알 수 있는가?

운영체제는 알 수 없다
만약 메모리에 페이지가 없는 경우에는 cpu의 제어권이 운영체제에게 넘어와서 알 수 있지만
이미 메모리에 있고 재참조 되는 경우에는 운영체제는 관련 정보를 알 수 없다
그렇기 때문에 LRU LFU는 paging system에서 사용될 수 없다

04 LRU-Approximation Page Replacement

- *LFU나 LRU 등의 알고리즘을 실제 페이징 시스템에서 사용할 수는 없다. (다른 하드웨어를 지원하지 않는 경우)*
- 따라서 많은 시스템들은 reference bit을 사용해 **LRU와 같은 효과**를 만들어낸다.
  - reference bit : page의 어떤 바이트가 read 되거나 write될 때 하드웨어에 의해 set되는 비트.

LRU의 근사 알고리즘, LRU-Approximation Page Replacement

- 모든 ref bit은 OS에 의해 0으로 set되어 있다.
- 프로세스가 실행되면, 참조되는 각 페이지에 붙은 bit가 하드웨어에 의해 1로 set 된다.
- 얼마간의 시간이 지나면, 우리들은 각 페이지가 어떤 순서로 참조되었는지는 알 수 없지만, 적어도 어떤 페이지가 최근에 참조되거나 참조되지 않았는지를 알 수 있게 된다.
- 이 정보가 바로 많은 LRU-근사 알고리즘들을 basis가 된다.

04 LRU-Approximation : Additional Ref. Bits Algo.

Additional-Refernece-Bits Algorithm

- 앞서와 달리 각 페이지에 여러 개의 ref bit들을 두어 순서 정보를 얻을 수도 있다.
- 8-bit의 ref bit들을 두었다고 하자.
- 일정 시간마다 timer interrupt가 발생해 OS로 CPU 권한이 넘어가면, 각 비트를 high-order로 shift하자.
  - 만약 8번의 interval동안 계속해서 참조된 페이지가 있다면 그 ref bits는 1111_1111
  - 만약 8번의 interval동안 한 번도 참조된 적이 없는 페이지라면 그 ref bits는 0000_0000
- 이 ref bits를 unsigned integer과 같이 생각한다면, 가장 작은 숫자를 가지는 페이지를 내쫓으면 된다.
- 단, 이 ref bits들이 unique하지는 않다. 그 경우에는 모두 swap out 하는 등의 방법을 사용할 수 있다.

- 얼마나 많은 수의 ref bit들을 사용할 건지는 달라질 수 있지만, update를 최대한 빠르게 할 수 있도록 정해진다.

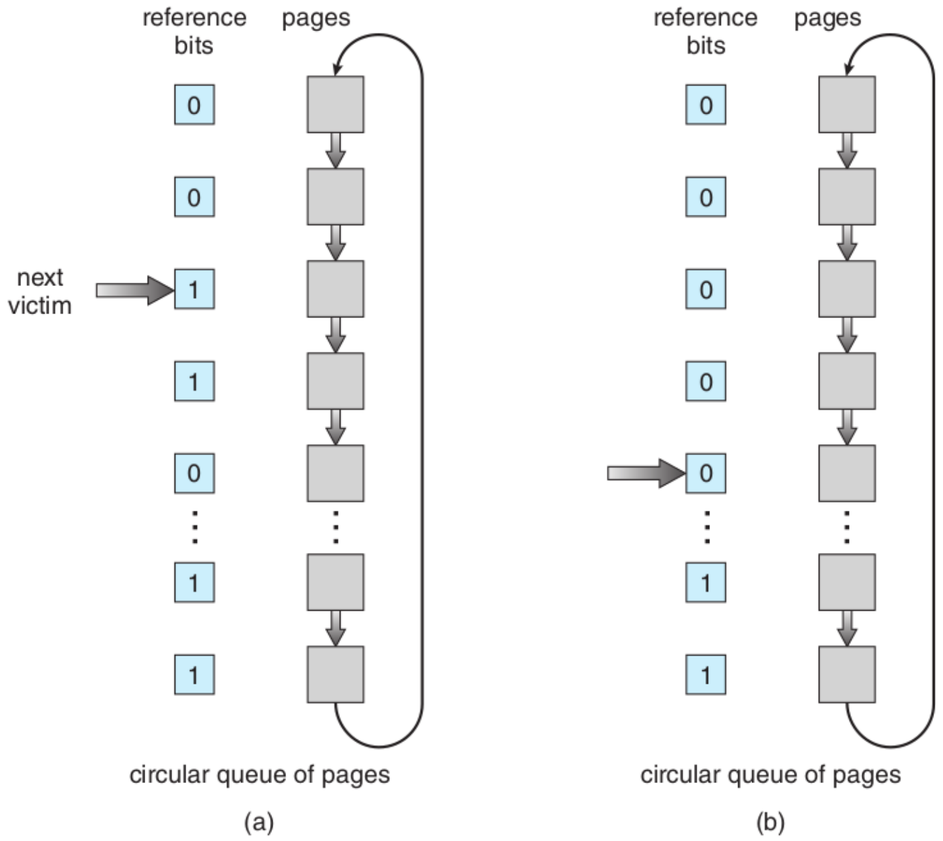
(강의에는 없는데 책에는 있는 내용. 원하면 박 무지렁이)

04 LRU-Approximation : Second-Chance Algorithm

Second-Chance Algorithm

- 기본적으로는 FIFO 알고리즘이라고 생각할 수 있다.
- 페이지가 선택되면 해당 페이지의 ref bit을 보자
  - 만약 ref bit가 0이면 swap out한다.
  - 1이면 한 번의 기회를 더 주고 다음으로 넘어간다. (ref bit은 초기화, 다시 FIFO에 넣기)
- 돌고 돌아 다시 해당 페이지의 차례가 되었는데, 그동안 이 페이지가 참조되지 않아 ref bit가 0이면 replace
  - 만약 자주 참조되는 페이지라면 ref bit가 1로 계속 변해 replace되지 않을 것.

04 LRU-Approximation : Second-Chance Algorithm



Clock Algorithm은 Second-Chance Algorithm을 구현하는 한 방법이다.

- Circular queue를 이용한 구현.
- 최악의 경우, 모든 ref bit이 1이면, 한 바퀴를 돌면서 모두 0으로 바꾸게 된다.
  - 이 경우에는 결국 FIFO와 같아지게 됨

04 LRU-Approximation : Enhanced Second-Chance Algo.

Reference Bit 외에도 Modify Bit을 사용해서 Second-Chance Algorithm의 성능을 향상시킬 수도 있다.

- ref bit와 modify bit을 하나의 페어로 생각하자. (ref, modify)

  (0, 0) : best page to replace

  (0, 1) : 페이지가 replace되기 전에 secondary storage에 write 돼야 하므로 가장 좋지는 않다.

  (1, 0) : 아마 곧 다시 쓰일 수도 있다

  (1, 1) : 아마 곧 다시 쓰일 수도 있다. secondary storage에도 write 되어야 한다.

- (0, 0)을 만나게 되면 우선적으로 replace한다.

04 Global vs. Local Replacement

Page-Replacement Algorithm는 다음과 같이 분류할 수 있다.

Global Replacement

- Replace 시 다른 프로세스에 할당된 frame을 빼앗아 process 별 할당량을 조절
- FIFO, LRU, LFU 등의 알고리즘을 global replacement로 사용할 경우에 해당. 할당 효과는 없음
- Working set, PFF 알고리즘 사용. 할당 효과가 있음
- System throughput이 높아 더 많이 쓰임

Local Replacement

- 자신에게 할당된 frame 내에서만 replacement
- Replacement algorithm을 각 프로세스 별로 운영하는 경우
- 한 프로세스를 돌리는 데에도 frame 변경이 잦아 throughput은 낮지만, frame은 적게 사용

05 Thrashing

프로세스의 원활한 수행에 필요한 최소한의 page frame 수를 할당받지 못한 경우에 발생하는 문제다.

- page fault rate가 매우 높아짐
- CPU utilization이 낮아짐
- 낮은 CPU util.로 인해 OS는 다른 프로세스도 돌릴 수 있겠다 판단, MPD를 높이려 함
  - MPD(Multiprogramming degree) : 메모리에 한 번에 올라온 process의 수
- 다른 프로세스가 시스템에 추가됨
- 프로세스 당 할당된 frame 수는 더욱 감소
- 프로세스는 page swap in / out으로 매우 바쁘지만, 대부분 시간 CPU는 한가함
- throughput이 낮아짐

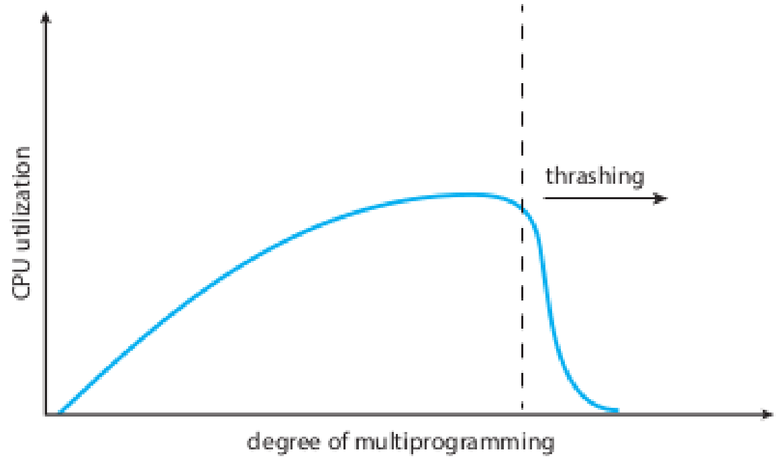


Figure 10.20 Thrashing.

05 Working-Set Model

Working-Set Algorithm

- 프로세스는 특정 시간 동안 일정 장소만을 집중적으로 참조한다.
- 이 집중적으로 참조되는 해당 page들의 집합을 locality set이라 함

Working-Set Model

- Locality에 기반, 프로세스가 일정 시간 동안 원활히 수행되기 위해 한꺼번에 메모리에 올라와 있어야 하는 page들의 집합
- Working-set Model에서는 process의 working set 전체가 메모리에 올라와 있어야 수행되고,
  그렇지 않을 경우 모든 frame을 반납하고 swap out
- Thrashing 방지
- MPD 결정

05 Working-Set Model

Working-Set Algorithm

- Working Set 또한 미리 알아낼 수는 없고, 과거를 보고 추정할 수 밖에 없다.
- Working Set Window를 통해 알아낸다.
- Window의 사이즈가 Δ인 경우,
  - 시각 t_i에서의 working set WS(t_i)
    - time interval [t_i - Δ, t_i] 사이에 **참조된** 서로 다른 페이지들의 집합
  - Working set에 속한 page는 메모리에 유지하고, 그렇지 않은 것들은 버린다.
    - 참조된 후 Δ 시간 동안 해당 page를 메모리에 유지한 후 버린다.
- Working Set의 정확도는 Δ를 정하는 데에 있다.
  - 너무 작거나 큰 경우에는 locality를 제대로 반영하지 못할 것이기 때문에

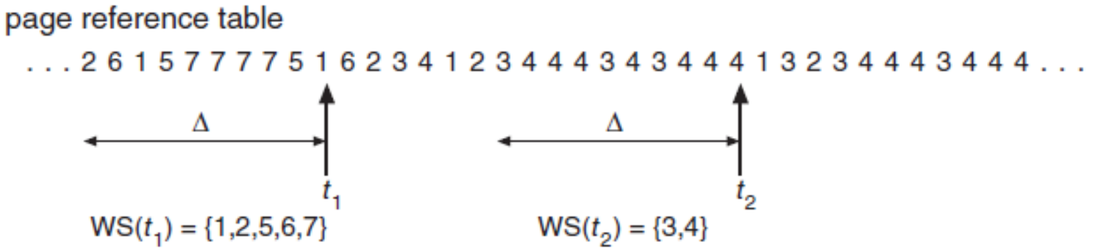


Figure 9.20 Working-set model.

05 Working-Set Model

흐아림이의 질문:

working-set model도 page fault인 경우에만 OS가 관여할 것 같은데 누가 page reference table을 관리하고 working set window를 움직이는지 잘 모르겠습니다. 단순히 코드를 보고 어떤 페이지가 쓰일지 page reference table에 쭉 나열하는 것인지?

05 Working-Set Model 

Operating System Concepts p.424, https://www.youtube.com/watch?v=GQ7BjDIuRcg 참고.
아래는 뇌피셜이 가미되었으니 같이 의논해보아요

매 reference 마다 interrupt를 일으켜 OS가 관리하도록 할 수도 있겠지만 비효율적이다.
고정된 간격의 timer interrupt와 reference bit를 이용해 working set을 근사할 수 있다.
예를 들어 Δ가 10,000, timer interrupt는 5,000 reference마다 일어난다고 하자.
각 페이지마다 2-bit의 bit을 두고, 이는 메모리에 저장한다. (뇌: 10,000 / 5,000, 전 interval과 전전 interval 기록)
OS는 timer interrupt를 받으면 각 페이지의 reference bit을 복사하고 0으로 초기화한다.
(뇌: 그런데 아마 shift한 후 복사하는 것 같다.)
page fault가 일어나면, 현재 interval, 전 interval, 전전 interval에서 참조가 되었는지를 확인할 수 있다.
(현재 reference bit과 기록된 2-bit 중 하나라도 1인 경우, working set에 들어있다고 할 수 있다)

=> 그러나 정확히 언제 참조되었는지는 알기 힘들다. 이 때는 Δ와 타이머 간격을 줄여서 개선할 수 있지만, 성능 상으로는 불리.

05 Page-Fault Frequency(PFF) Scheme

Page-fault rate의 상하한을 두고 모니터링 한다.

- 상한을 넘으면 프로세스에 frame을 더 할당하고
- 하한 이하면 할당 frame 수를 줄인다.

빈 frame이 없어 더 줄 메모리가 없으면 특정 프로세스를 골라 swap out하고 해당 프레임은 높은 page-fault rate를 가지는 프로세스에 분배한다.

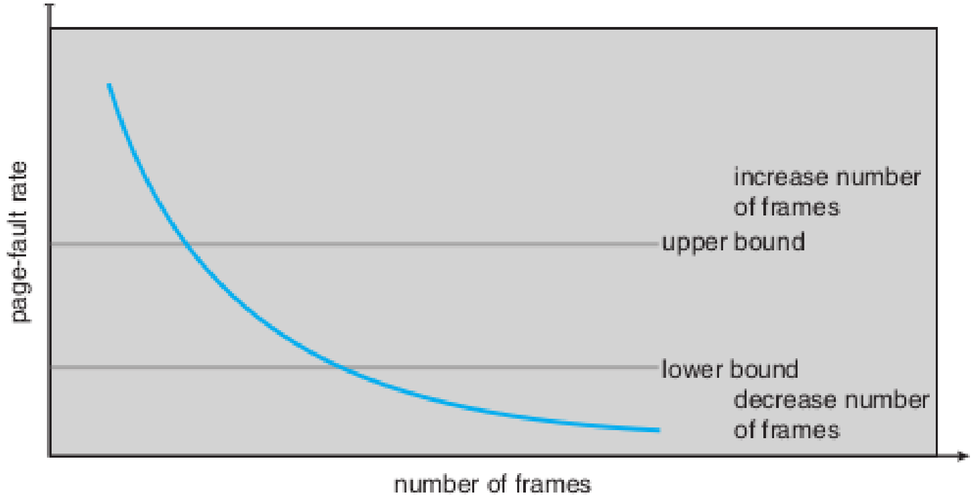


Figure 10.23 Page-fault frequency.

(그런데 강의에서는 "빈 frame이 없어 더 줄 메모리가 없으면, 그 메모리를 통째로 swap out해 남아 있는 프로세스들이라도 잘 돌아가게 한다."라고 했는데, 이것이 무슨 말인지는???)

05 Page-Fault Frequency(PFF) Scheme

철썩이의 질문: 

PFF에서 상한값과 하한값을 정하는 대략적인 기준과, 빈 frame이 없을 경우 일부 프로세스를 swap out 시키는데 이때 옮겨질 프로세스는 어떠한 기준으로 정하는지 궁금합니다?

지 선생님의 말씀:

1) 시스템의 특성 및 성능 요구 사항에 따라 다르게 설정되며, 경험적으로 혹은 성능 테스트를 통해 결정된다고 합니다.

2) 최근에 사용되지 않은 프로세스, 우선 순위가 낮은 프로세스 등이 선정될 수 있다고 합니다.

환경과 구현에 따라 다른 듯합니다.

06 Page Size

- Page size를 감소시키면
  - 페이지 수 증가
  - 페이지 테이블 크기 증가
  - Internal Fragmentation 감소
  - Disk transfer 효율성 감소
    - Seek/rotation vs transfer
    - 디스크는 디스크 헤드가 이동해서 찾는데 이 시간이 오래 걸림
    - 따라서 웬만하면 한 번에 많이 찾아서 많이 올려 놓는 게 좋음
  - 필요한 정보만 올라오므로 메모리 이용이 효율적
    - locality 활용 측면에서는 좋지 않음
- 트렌드는 큰 page size를 이용하는 것