

9. Virtual Memory

▼ 목차

1. Demand paging
2. page fault
3. page replacement
4. page replacement algorithm
 - 4-1. Optimal Algorithm
 - 4-2. FIFO (First In First Out) Algorithm
 - 4-3. LRU (Least Recently Used) Algorithm
 - 4-4. LFU (Least Frequently Used) Algorithm
 - 4-5. LRU LFU 알고리즘 구현
5. 다양한 캐싱 환경
6. paging system에서 LRU LFU는 가능한가?
7. Clock Algorithm
8. page frame의 allocation
9. global replacement & local replacement
10. Thrashing
 - 10-1. Working-Set Model
 - 10-2. PFF(Page-Fault Frequency) Scheme

1. Demand paging

- 실제로 필요할 때 page를 메모리에 올리는 것
 - IO 양의 감소
 - Memory 사용량 감소
 - 빠른 응답 시간
 - 더 많은 사용자 수용

▼ Demand paging 예시

1. 텍스트 편집기: 사용자가 텍스트 파일을 편집하는 텍스트 편집기를 생각해봅시다. 텍스트 파일은 일반적으로 큰 크기를 가지며, 모든 내용을 한 번에 메모리에 로드하는 것은 비효율적입니다. Demand paging을 사용하면 사용자가 현재 편집 중인 부분에만 해당하는 페이지들이 메모리에 적재됩니다. 사용자가 스크롤하거나 특정 위치로 이동할 때 필요한 페이지만 디스크에서 가져와 메모리에 로드되므로, 실제로 필요한 데이터만을 처리하게 됩니다.

2. 웹 브라우저: 웹 브라우저를 실행하여 여러 웹 페이지를 동시에 탐색하는 경우를 생각해봅시다. 각 웹 페이지는 HTML, CSS, JavaScript 등으로 구성되어 있으며, 모든 웹 페이지를 동시에 메모리에 로드하는 것은 메모리 낭비입니다. Demand paging을 사용하면 사용자가 현재 보고 있는 웹 페이지에 해당하는 페이지들만 메모리에 적재됩니다. 사용자가 다른 페이지로 이동할 때 해당 페이지의 페이지들이 메모리에 로드되고, 이전에 로드된 페이지는 필요 없어지면 디스크로 다시 내보내어 메모리 공간을 확보합니다.

- Valid Invalid bit 사용

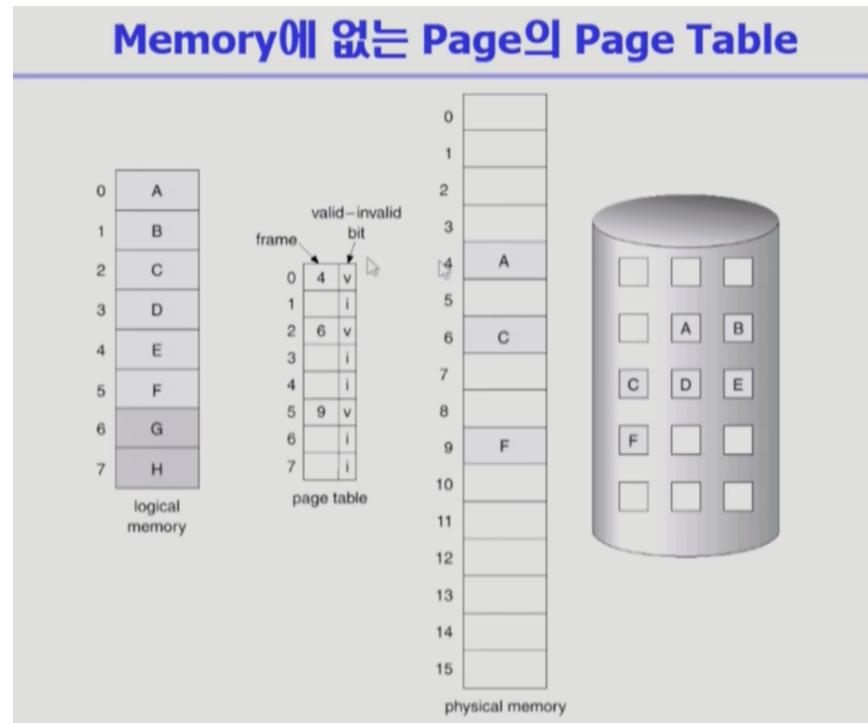
- invalid

사용 되지 않는 주소 영역인 경우

페이지가 물리적 메모리에 없는 경우

- 처음에는 모든 page entry가 invalid로 초기화

- address translation 시에 invalid bit이 set되어 있으면 → **page fault**
page fault : 요청한 페이지가 메모리에 없는 경우



2. page fault

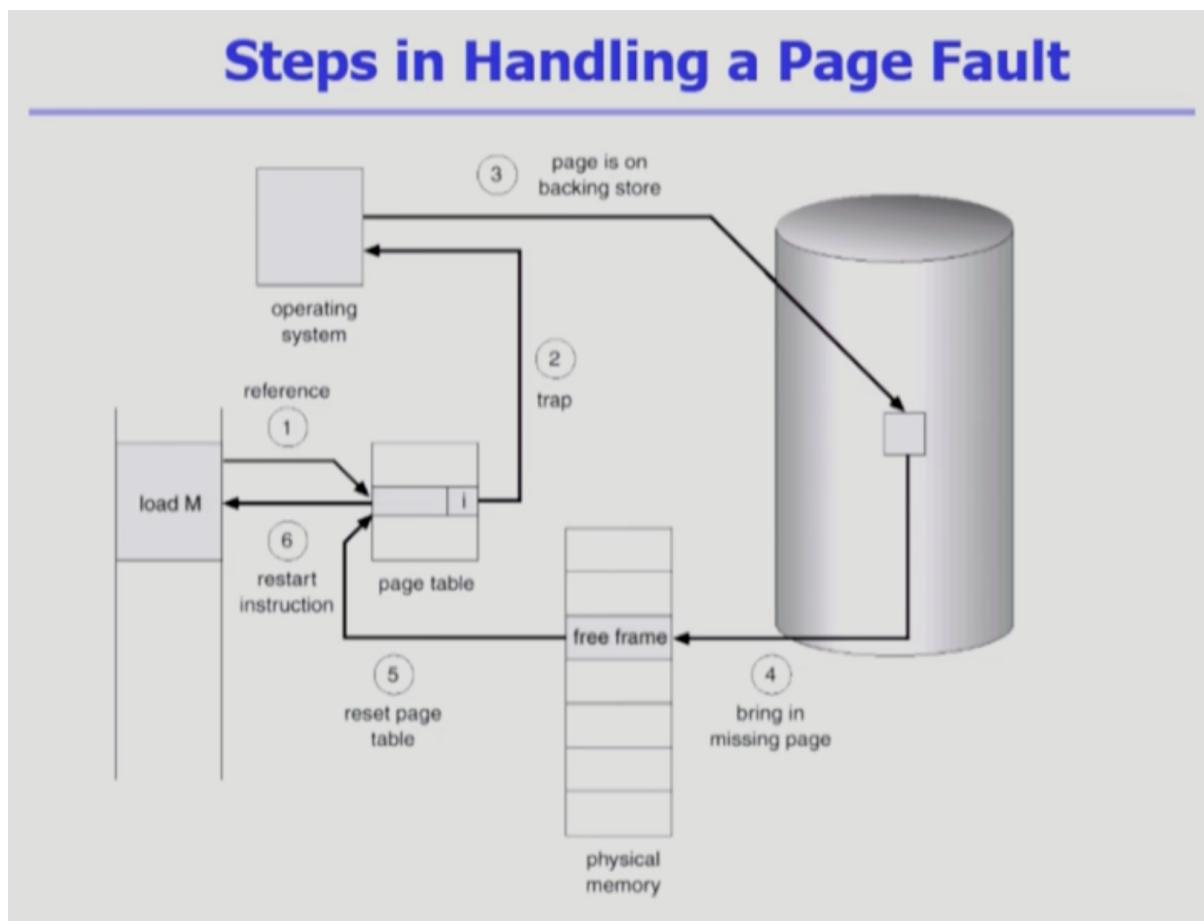
- **요청한 페이지가 메모리에 없는 경우**

cpu는 프로세스에서 운영체제로 넘어가고 (trap 발생) backing store의 데이터를 물리 메모리로 가져옴

invalid page를 접근하면 MMU가 trap을 발생시킴

kernel mode로 들어가며 page fault handler가 불러짐

1. 운영체제가 잘못된 요청이 아닌지를 확인 (주소가 잘못되었거나 ..) 하고 잘못된 요청일 경우 중단
2. 정상적인 메모리 요청이라면 backing store에서 메모리로 페이지를 가져와야 하므로, 메모리의 공간을 확보해야 한다. (만약 공간이 꽉 차 있을 경우라면 공간을 뺏어서 확보 한다)
3. 공간이 확보 되었다면 디스크에서 메모리로 페이지를 읽어온다
 - a. disk io가 끝나기까지 이 프로세스는 CPU를 선점 당함 (block)
 - b. disk read가 끝나면 page tables entry 기록, valid 로 처리
 - c. ready queue에 다시 해당 프로세스를 삽입 → 추후에 다시 실행
4. 해당 프로세스가 다시 cpu를 잡게 된다면 아까의 MMU 요청을 처리하고, 다음 프로세스로 넘어감



위 그림은 page fault를 해결하는 과정

하지만 페이지 폴트가 발생해서 디스크에 접근하는 것은 대단히 올래 걸리는 작업이다
따라서 page fault를 최대한 발생하지 않게 해야 한다.

3. page replacement

page fault가 발생 해서 backing store의 데이터를 가져와야 하는데, 빈 프레임(free frame)이 없는 경우

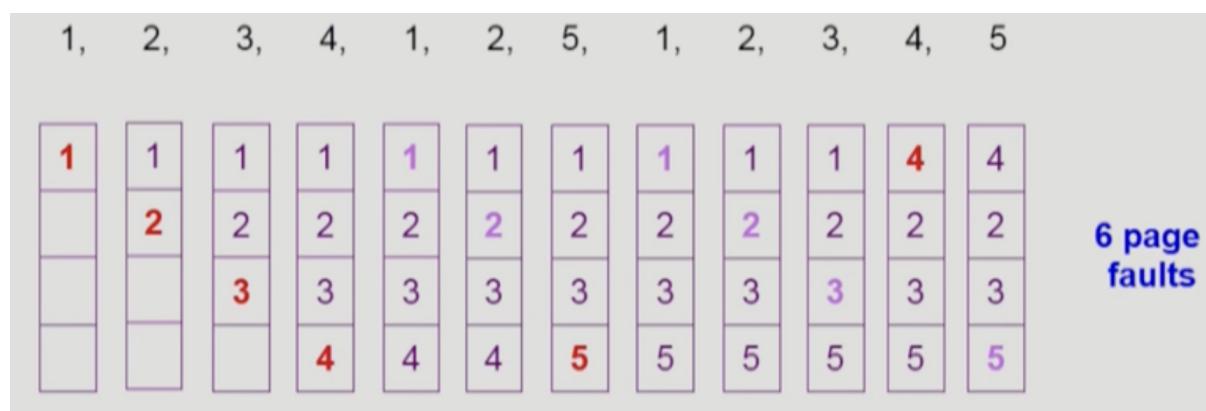
- page replacement
 - 어떤 frame을 빼앗아올지 결정해야 함
 - 곧바로 사용되지 않을 page를 쫓아내는 것이 좋다
 - 동일한 페이지가 여러 번 메모리에서 쫓겨났다가 다시 들어올 수 있다
- replacement Algorithm
 - page fault rate을 최소화 하는 것이 목표
 - 알고리즘 평가
 - 주어진 page reference string에 대해 page fault를 얼마나 내는지 조사

4. page replacement algorithm

4-1. Optimal Algorithm

page fault를 최소화 하는 알고리즘

가장 먼 미래에 참조되는 page를 replace

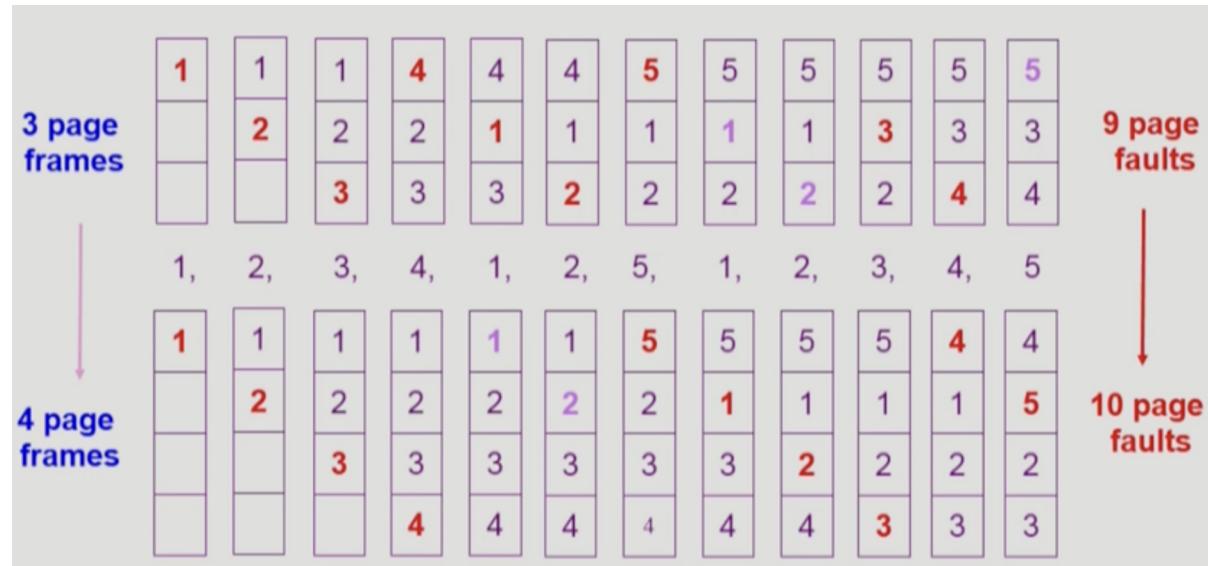


- 실제에서는 미래의 참조를 알고 있다는 것은 말이 안되지만 우선 가정을 하고 성능을 측정한다

- 이러한 가정 때문에 가장 좋은 성능을 보여주어서 다른 알고리즘의 성능에 대한 upper bound를 제공한다

4-2. FIFO (First In First Out) Algorithm

먼저 들어온 것을 먼저 내쫓는 알고리즘

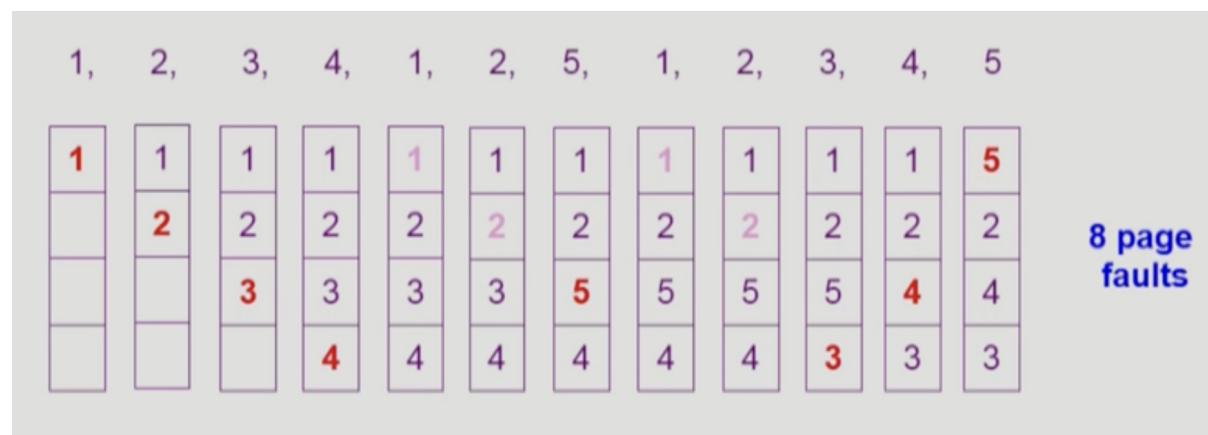


FIFO Anomaly

frame의 수를 늘렸을 때 더 많은 page fault 가 발생하는 현상

4-3. LRU (Least Recently Used) Algorithm

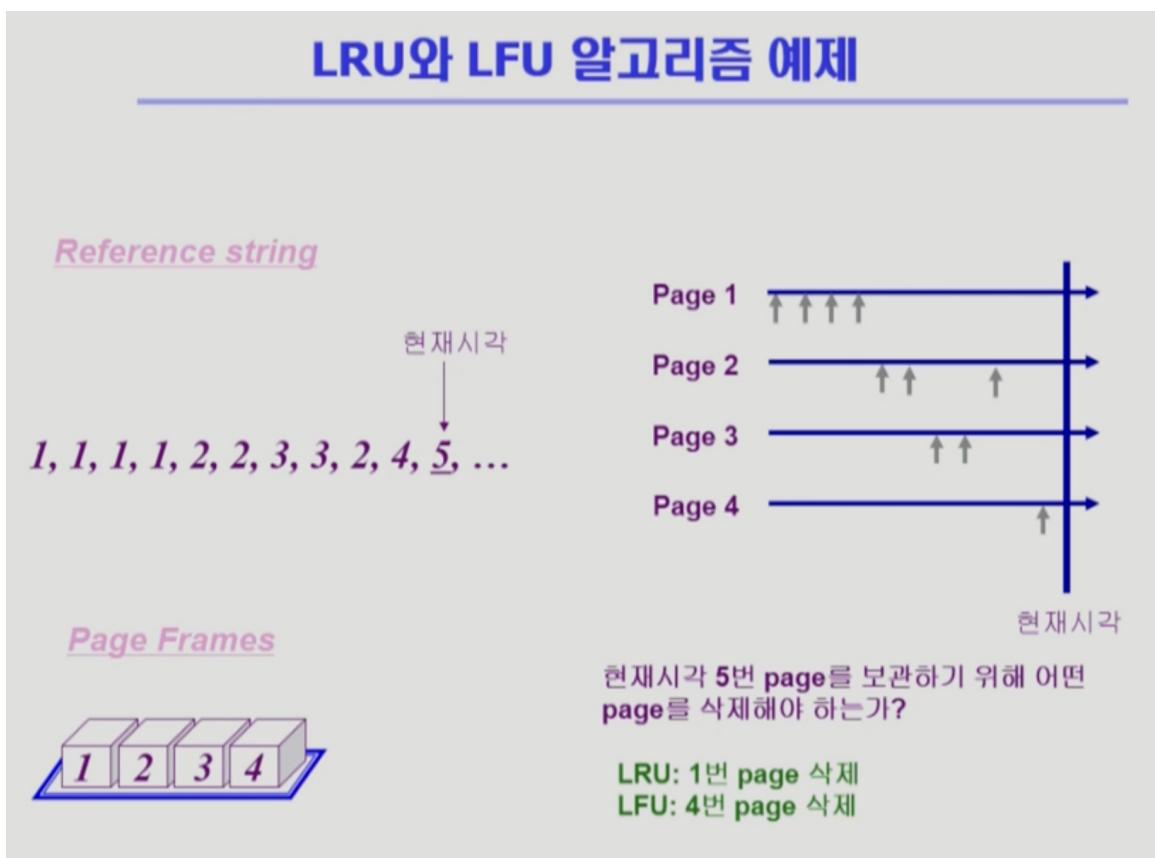
가장 오래 전에 참조된 것을 지움



4-4. LFU (Least Frequently Used) Algorithm

참조 횟수가 가장 적은 페이지를 지움

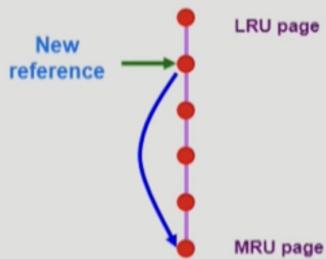
- 최저 참조 횟수인 page가 여럿 있는 경우
 - LFU 알고리즘 자체에서는 여러 page 중 임의로 선정
 - 성능 향상을 위해 가장 오래 전에 참조된 page 를 지우게 구현할 수도 있다
- 장단점
 - LRU처럼 직전 참조 시점만 보는 것이 아니라 정기적인 시간 규모를 보기 때문에 page의 인기도를 좀 더 정확히 반영할 수 있음
 - 참조 시점의 최근성을 반영하지 못함
 - LRU보다 구현이 복잡



4-5. LRU LFU 알고리즘 구현

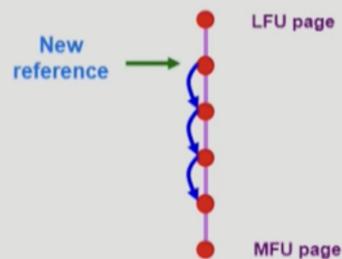
LRU와 LFU 알고리즘의 구현

↳ LRU



$O(1)$ complexity

↳ LFU



$O(n)$ complexity

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

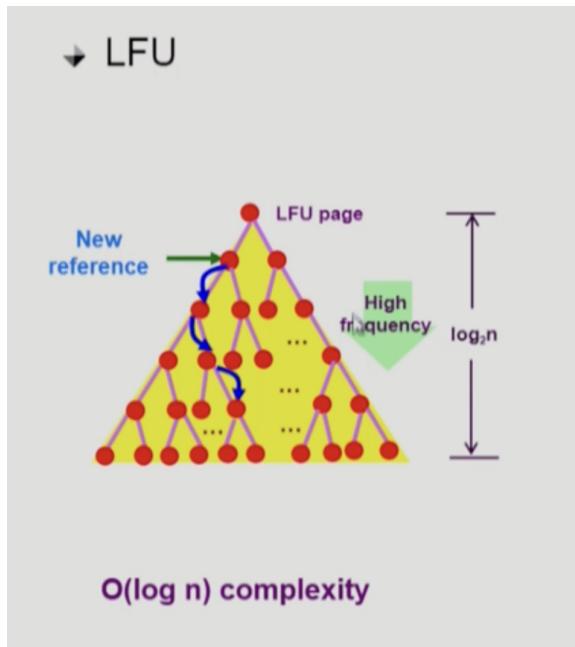
이 알고리즘은 참조의 횟수에 따라 위치가 지정 됨

시간 복잡도 : $O(n)$

최악의 경우 참조 횟수가 1증가 했는데 최하단까지 비교를 해야할 경우 시간 복잡도가 크다

_ $O(n)$

그렇기 때문에 이런식으로 구현을 하지 않고 heap을 사용해서 구현한다
heap을 사용하게 되면 $O(\log n)$ 의 시간 복잡도를 갖는다

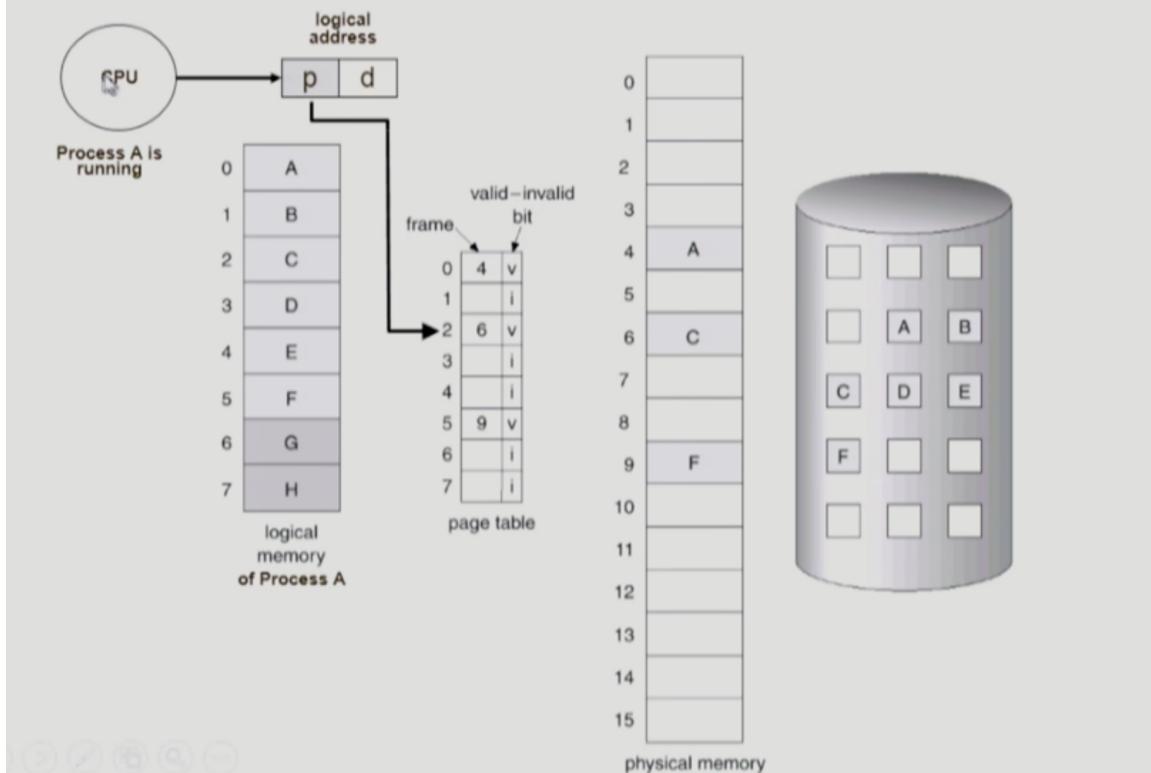


5. 다양한 캐싱 환경

- 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 caching이나 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

6. paging system에서 LRU LFU는 가능한가?

Paging System에서 LRU, LFU 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

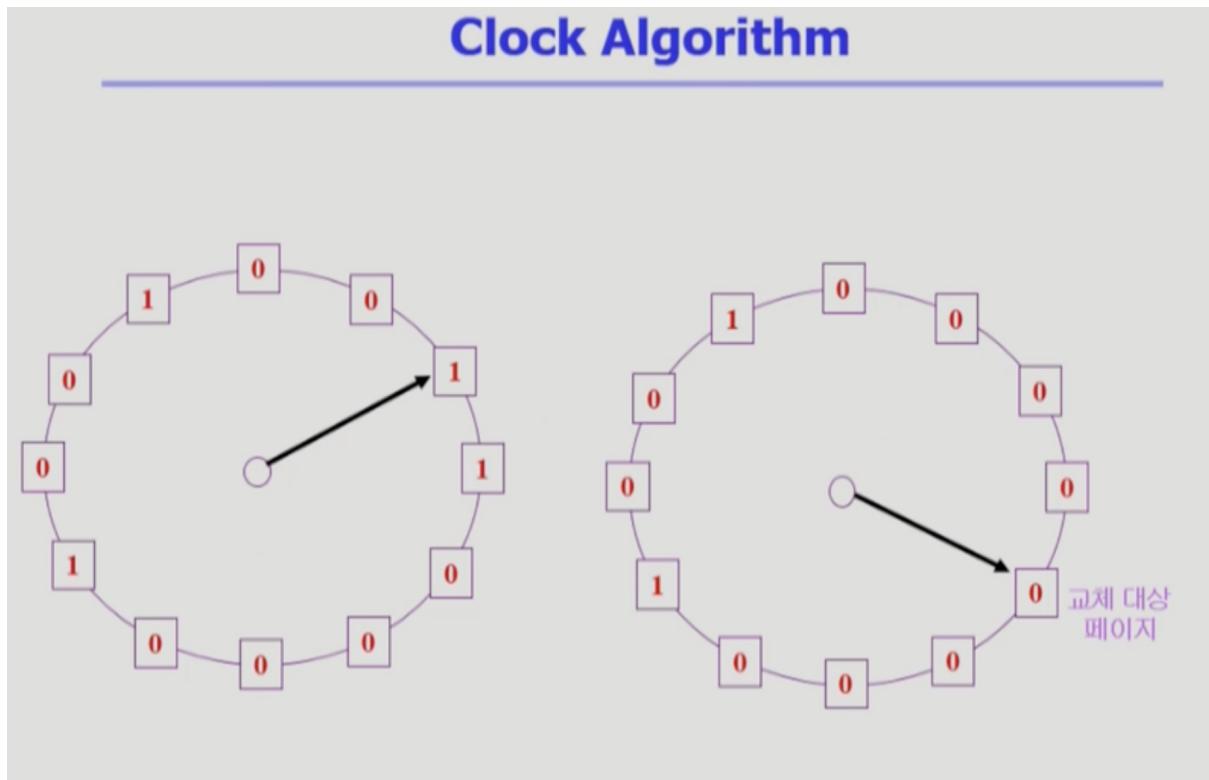
그렇기 때문에 LRU LFU는 paging system에서 사용될 수 없다

7. Clock Algorithm

paging system에서는 LFU LRU 알고리즘을 사용할 수 없기 때문에 이 알고리즘을 사용한다

- LRU와 근사한 알고리즘
- Reference bit를 사용해서 교체 대상 페이지 선정
- Reference bit가 0인 것을 찾을 때까지 포인터를 하나씩 앞으로 이동
- 포인터가 이동하는 중에 Reference bit 1은 모두 0으로 바꿈

- Reference bit이 0인 것을 찾으면 그 페이지를 교체
- 한 바퀴 되돌아와서도(second chance) 0이면 그때에는 replace 당함
- 자주 사용되는 페이지라면 second chance가 올 때 1



8. page frame의 allocation

각 프로세스에 얼마만큼의 page frame을 할당할 것인가?

- 메모리 참조 명령어 수행시 명령어, 데이터 등 여러 페이지 동시 참조
 - 명령어 수행을 위해 최소한 할당되어야 하는 frame 의 수가 있다
- loop 를 구성하는 page 들은 한꺼번에 allocate되는 것이 유리하다
 - 최소한의 allocation이 없으면 매 loop 마다 page fault

Equal allocation : 모든 프로세스에 똑같은 갯수 할당

Proportional allocation : 프로세스 크기에 비례해서 할당

Priority allocation : cpu 우선순위가 높은 프로세스에게 프레임을 많이 할당

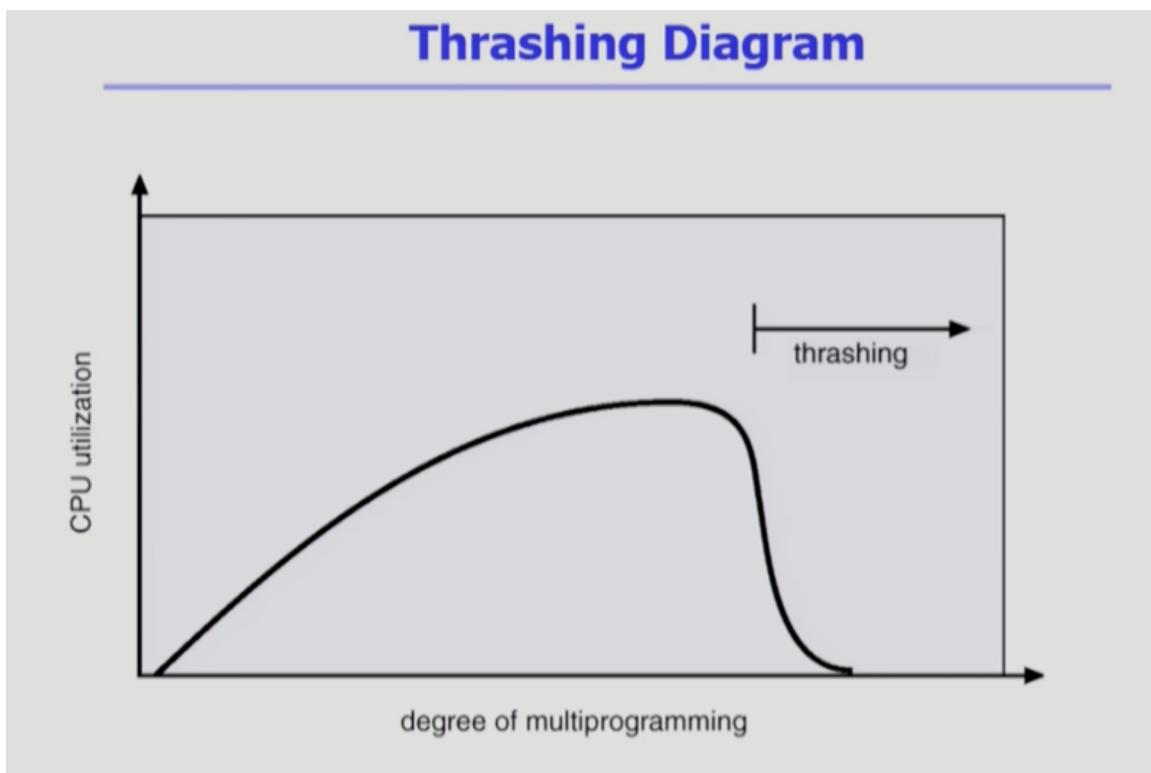
9. global replacement & local replacement

Global Replacement은 Replace 할 때 다른 프로세스에 할당된 frame을 빼앗아올 수 있다. 프로세스별로 frame 할당량을 조절하는 또 다른 방법이 될 수 있지만, 자신의 page fault rate를 조절할 수 없다.

일반적으로 더 좋은 처리량을 가지므로 가장 흔하게 사용되는 방법이다.

Local Replacement은 자신에게 할당된 frame 내에서만 교체하는 방법이다. 알고리즘을 프로세스마다 독자적으로 운영하는 경우 가능하다. 쉬고 있는 메모리를 사용할 수 없기 때문에 비교적 비효율적이다.

10. Thrashing



프로세스가 원활한 수행에 필요한 최소한의 page frame을 할당받지 못해서, 실행보다 Swapping 하는데 더 많은 시간을 소모하는 현상이다.

1. 메모리에 프로세스를 추가할 수록 cpu의 활용은 증가하게 된다
2. 하지만 프로세스가 과도하게 많아지게 되면 각 프로세스에 할당되는 frame의 크기도 줄어들게 되어서 page fault가 빈번히 일어나게 된다
3. page fault가 높아지면 io작업이 진행되는 동안 cpu가 다른 프로세스에게 넘어가는데, 거기에서 또 page fault가 발생하게 되면 또 cpu는 다른 프로세스로 넘어감

4. 이런 상황이 반복되다 보면 cpu의 활용은 줄어들게 되고, OS는 이 현상을 cpu의 사용량이 줄어드는 것으로 인식해 MPD(multiprogramming degree)를 높혀야 한다고 판단한다
5. 그래서 다른 프로세스가 추가 되면 각 프로세스에 할당되는 프레임이 줄어들게 되어 악순환이 이어진다
6. 모든 프로세스들이 io작업만 진행하고 있어서 cpu가 할 일이 없어진다

⇒ 이러한 문제를 해결하기 위해서는 프로세스에게 frame을 필요한 만큼 많이 제공

Working Set Algorithm & Page Fault Frequency

10-1. Working-Set Model

가능한 최대 Multiprogramming Degree를 유지하면서 Thrashing을 막는 방법이다.

Locality of reference(참조 지역성의 원리)

프로세스가 특정 시간 동안 일정 장소를 집중적으로 참조하는 성질을 말한다.

집중적으로 참조되는 해당 page들의 집합을 locality set이라고 한다

이 Locality에 기반하여 프로세스가 일정 시간 동안 원활히 수행되기 위해 한꺼번에 메모리에 올라와있어야 하는 page들의 집합을 Working set이라고 한다.

Working set은 Working set window라는 고정된 page 참조 횟수(시간)로 구한다.

10-2. PFF(Page-Fault Frequency) Scheme

page fault의 상한 값과 하한 값을 두고, page fault rate가 상한 값을 넘으면 frame을 더 할당하고, 하한 값보다 낮아지면 할당된 frame 수를 줄이는 방법이다.

