11. Virtual Memory

Virtual memory: process의 logical memory와 physical memory를 분리

⇒ 이렇게 함으로써 메모리 효율성을 챙길 수 있었어

근데 이거 성능 더 좋게할 수 없을까? 에서 나온 아이디어

1. Demand paging
2. Demand segmentation

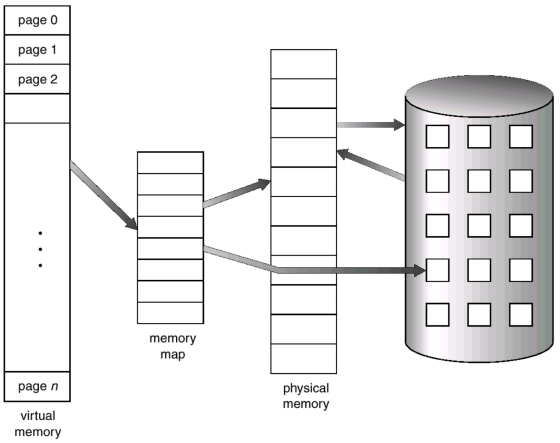
지금은 이런식으로 실제로 필요할 때마다 page나 segment를 메모리로 올리는데 이 발상을 바꿔보는거지

애초에 CPU한테 프로세스의 모든 데이터가 메모리에 올라왔다고 속이고 진행하면 되지 않나?

(단 OS는 일부만 올라간 것을 알고 있어)

그렇다면 CPU가 memory에 올라와 있지 않은 영역에 접근할 때 어떻게 처리할 것인지?

이거만 잘 처리해준다면 잘 쓸 수 있지 않을까?



이런식으로 메모리에 있다면 OS가 memory에 mapping 시켜주고

만약 메모리에 없다면 OS가 disk에서 memory로 올려다 주고

여기서 CPU는 기본적으로 다 메모리 영역으로 올라왔다고 생각

Demand Paging

아 그러니까 이제 CPU가 메모리에 올라왔다고 생각하는 건 알겠어

근데 이게 실제로 올라왔는지 없는지 어떻게 구분할건데?

전에 valid/invalid bit 사용해서 저장하는 방식을 그대로 사용할 거야

CPU가 요청하는 page가 실제로 없다면? 그 때 메모리에 올려주는 방식

Valid-Invalid Bit

Valid는 뭐 그냥 메모리에 올라온 상태를 말해서 그냥 넘어감

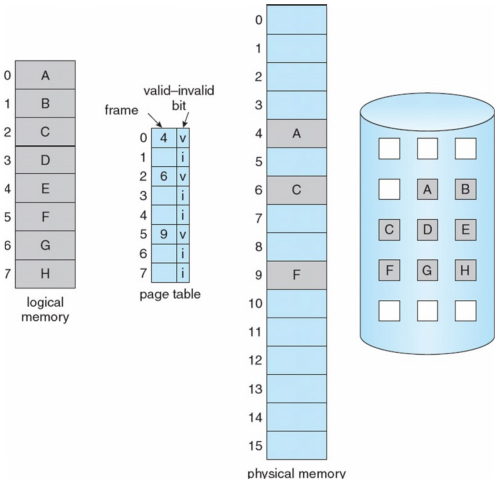
Invalid는 뜻이 2가진데

1. illegal
2. not-in-memory

전에 valid invalid bit 배울 때는 실제 접근하는 영역이 맞는지 판별하는 용도로 쓰였는데

이제는 page가 memory에 존재하는지 파악하는 용도로도 사용

⇒ 즉 현재 메모리에 데이터가 있어도 invalid인 경우, 메모리에 있는 데이터가 갱신되지 않은 데이터



그래서 특정 시점에 대한 snapshot을 찍어서 찾아보면 memory에 올라와있는

A, C, F 는 valid 처리가 되어 있고 나머지는 invalid

Page Fault

CPU가 page에 접근하려 하는데 해당 page가 실제로 memory에 없는 경우

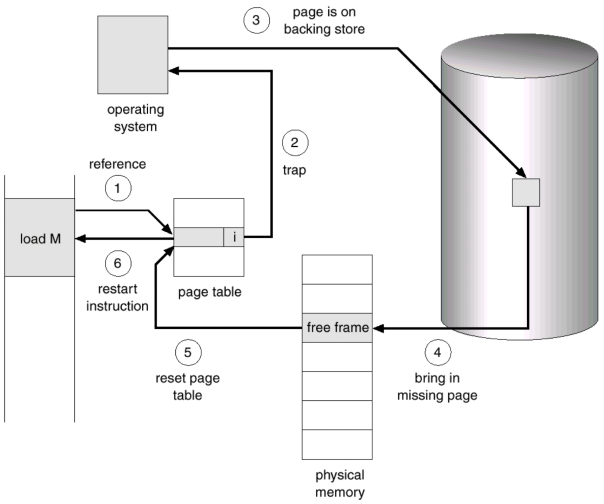
(invalid인 page에 접근하고자 할 때)

그래서 page fault가 발생하면 다음 절차를 따름

1. page fault가 뭐 때문에 발생했는지 확인
    - illegal reference ⇒ abort process
    - not in memory? ⇒ 일단 진행
2. free frame 확보
    - 만약 빈 공간이 없으면 기존의 frame과 교체
3. free frame에 page 등록
    - 근데 프로세스 입장에서 생각해보면 이 page I/O는 프로세스가 발생시켰을까?

      ⇒ ㄴㄴ OS가 발생시킨거
4. page fault 처리가 끝나면 다시 CPU가 process를 잡고
5. trap이 발생했던 지점부터 다시 이어서 진행

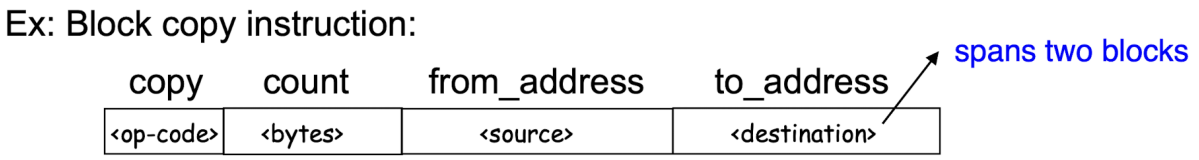
Page Fault Handling



만약 page table의 entry가 valid bit면?

⇒ free frame을 만드는게 아니라 physical memory에서 frame 찾아서 그대로 프로세스로 불러들임

Difficulties in actual HW design



이거는 from address에 있는 데이터를 to address로 옮기는 명령인데

만약 from 과 to에 각각 2개의 page가 걸쳐져 있다면 어떻게 될까?

(source에 page 2개, destination에 page 2개)

상황을 좀 자세하게 보면

1. src에서 하나의 페이지는 복사가 잘 됨
2. 근데 두번째에서 page fault 발생

그러면 이 때는 어떻게 처리해야하나?

⇒ undo 로 바꿔줘야 해, 명령을 반만 따르는 건 말이 안되니까

Performancee of Demand Paging

page fault rage: p (0이상 1이하)

$$EAT = (1-p) \times memory\ access\ +\ p(page\ fault\ overhead + [swap\ page\ out\ if\ neeeded]\ +\ swap\ page\ in\ +\ restart\ overhead)$$

page fault가 발생하지 않으면 그대로 memory access해서 page 찾고

만약 page fault가 발생하면 page fault의 원인을 찾기 위한 overhead, free frame이 부족하다면 page swap out, 요구하는 page를 불러들이고 프로세스를 재시작하는 overhead

(page fault가 발생하는 순간 wait 상태로 진입)

근데 Demand Paging을 하는게 EAT 계산상으로 40배 느려짐 ⇒ 그럼 쓰면 안되지 않냐?

다행히도 locality of reference 덕분에 성능저하가 크게 안온다

(cache hit 같은 거 많이 발생하기 때문)

What happens if there is no free frame?

Page replacement

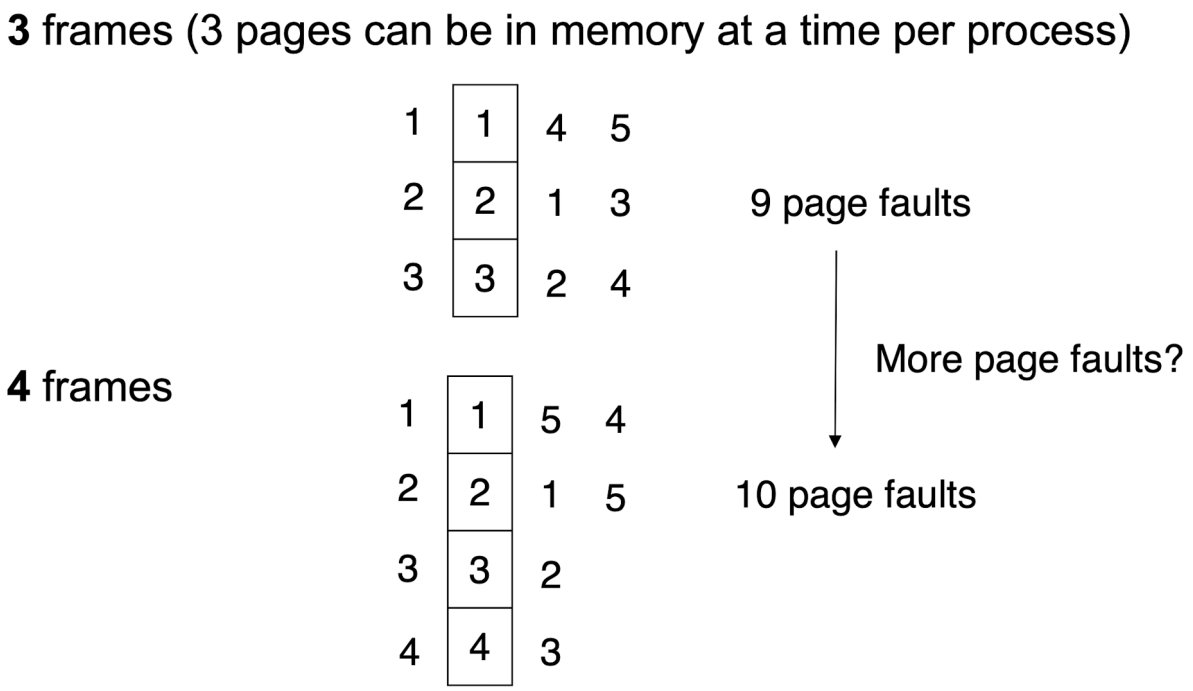
(아까 위에서 page 교체 해야한다고 하긴 했어)

그래서 이제부터는 어떤 page replacement 알고리즘을 선택해야 page fault를 적게 발생시키는지 알아볼 예정

이제부터 기본적으로 접근하는 page 번호 sequence는

> 💡 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

First-In-First-Out(FIFO) Algorithm



기본적으로 frame을 여러개 쓰면 page fault가 적게 일어날 것을 기대

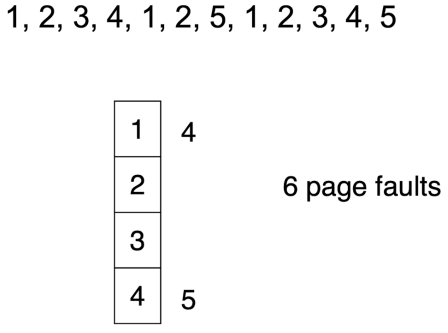
그래서 frame을 3개 쓸 때를 보면 page fault가 9번 발생

근데 frame을 4개 쓸 때 page fault가 10번 발생

왜 이런일이 발생했을까? ⇒ 몰라

그래서 이걸 Belady's Anomaly 라고 함

Optimal Algorithm

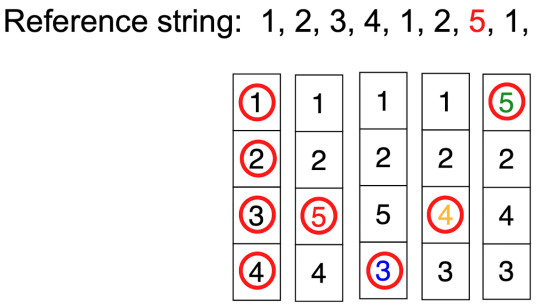


이건 일단 신이 된 입장에서 page를 끼워넣는 거

그래서 앞으로 자주 쓰일 page를 메모리에 남기고 안 쓰일 페이지를 내려보내

근데 이걸 어떻게 알아? ⇒ 모름;;

Least Recently Used (LRU) Algorithm



가장 이전에 쓰인 page부터 내려보내

⇒ 이거하니까 Optimal과 유사한 성능을 내

근데 가장 이전에 쓰인 page를 알아야 하니까 각 page 별로 Timestamp가 필요해

timestamp가 가장 작은 page를 찾는 알고리즘도 필요

근데 이거 쓸라다보니 space/time overhead가 커

그래서 실제 LRU는 쓰지 않고 Approximation model을 사용

LRU Implementation Algorithm

1. Counter Implementation

   ⇒ 정확한 timestamp는 필요없고 상대적인 순서만 알면 되지 않을까?

   그래서 CPU의 counter를 사용

   CPU가 메모리에 접근할 때마다 해당 페이지에 현재 Counter를 기록

   그래서 다음 page evict 시킬 때 counter값이 가장 작은 page를 내려보내
2. Stack Implementation

   stack 사용

   대충 double linked list 사용한다고 생각해도 됨

   그래서 지속적으로 사용되는 페이지를 맨 뒤 or 맨 앞으로 보냄

   (맨 뒤로 보냈다면 맨 앞에 있는 page가 evict, 맨 앞으로 보냈다면 맨 뒤에 있는 page가 evict)

   대신에 page를 옮길 때마다 pointer가 총 6개 사용되는게 overhead

   (장점으로는 evict page를 찾는게 빠르겠지)

LRU Approximation Algorithms

1. Reference bit

   bit를 둬서 접근하면 1로 바꿔줌

   일정 주기마다 bit를 0으로 변경

   ⇒ 즉 bit 값이 1이라는 이야기는 최근에 접근됐다는 소리
2. Additional-Reference-Bits Algorithms

   이거는 reference bit 쓰는건 동일한데, bit를 8개 사용

   그래서 이전에는 하나의 주기 내에서는 어떤 페이지가 더 최근에 접근됐는지 파악하기가 어려운데

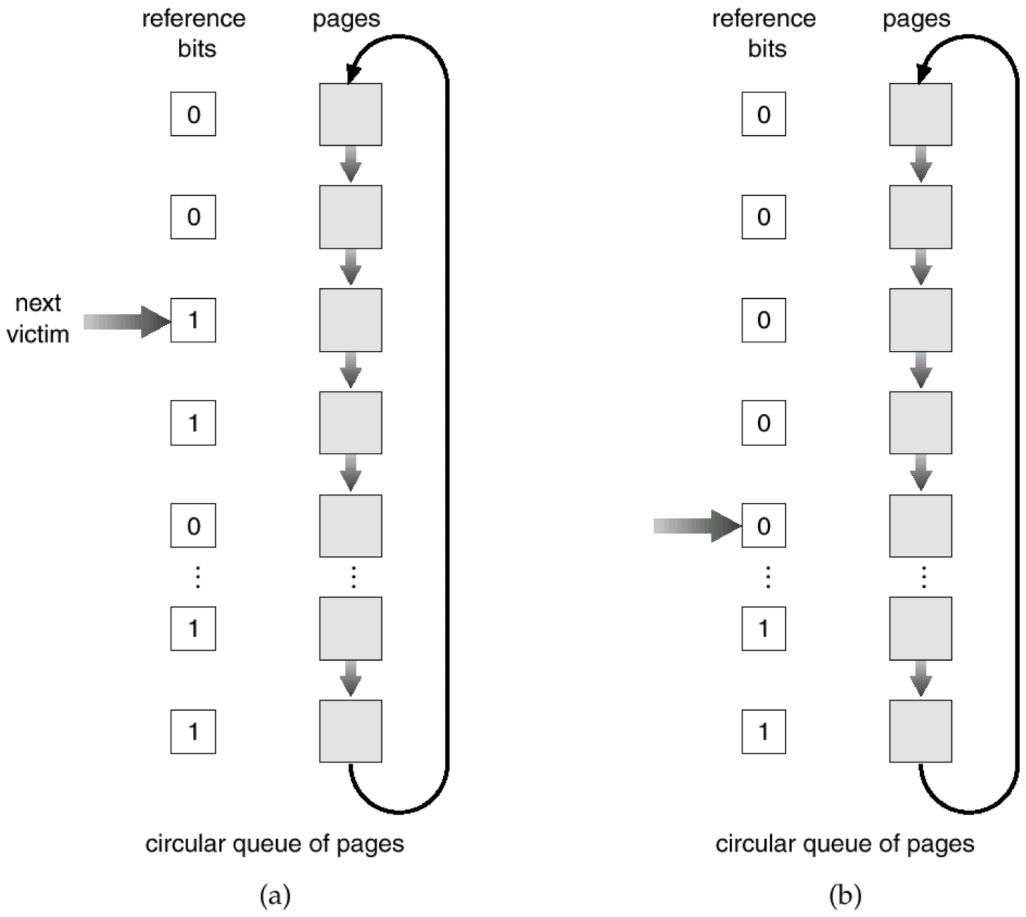
   추가적인 bit를 씀으로써 하나의 주기 내에서도 누가 더 최근에 참조 되었는지 확인 가능

   값으로 비교하기 때문에 MSB에다가 접근하면 bit 1을 넣고, 접근하지 않으면 bit 0을 넣음

   그러면 값이 더 큰 page가 비교적 최근에 접근한 page가 되겠지

   (8bit를 안써도 되는데, 결국 이거도 overhead라서...)
3. Second chance (clock) algorithm

   원형 queue로 관리



pointer를 옮겨가면서 0을 발견하면 그 page를 evict

bit가 1이면 0으로 바꿔주고 pointer 옮김

이 때 모든 bit가 1이라면 FIFO가 되버림(근데 그럴일이 얼마나 될까...)

LFU(Least Frequently Used) MFU(Most Frequently Used) 알고리즘이 있는데 이런게 있구나 정도?

Allocation of Frames

process에 얼마만큼의 page를 할당하는게 좋을까?

⇒ 최소로 필요로 하는 만큼

Fixed Allocation

일단 먼저 말하자면 안 씀 ㅇㅇ...

방법이 2가지가 있는데

1. Equal Allocation

   모든 프로세스가 균등하게 전체 frame을 나눠가짐

   -ex) 100 frames, 5 processes ⇒ 20 pages each
2. Proportional allocation

   프로세스의 크기를 고려해서 비율만큼 나눠가짐

Priority Allocation

이제는 우선순위를 기준으로 page 할당

그래서 중요한 작업에 page를 많이 할당해서 해당 프로세스가 wait 상태로 진입하지 않고 그대로 끝내는게 목적

근데 이거 문제는 process가 필요한 page의 개수가 진행되면서 달라질 수 있어

(일정하다는 보장이 없으니까)

그래서 애초에 모든 프로세스를 통으로 묶어서 한번에 처리하면 어떨까 하는 아이디어가 나옴

Thrashing

만약 충분한 페이지가 없다면 page fault rate이 너무 높아짐

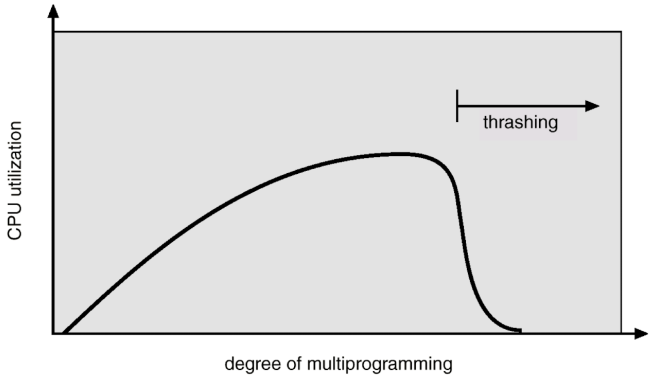
⇒ 그러면 page 교체하느라 process 진행은 못해서 느려지는 거

```
main()
{ for (I=1, 100000)  { A = B + X }}
```

A
main()
X
B

이런 코드에서 I에 대해서는 page가 할당되어 있지 않아서

모든 for문에서 page fault 발생(5개 중에 하나는 무조건 없으니까)



딱 저기 꺾이는 지점부터 thrashing 발생

Locality

thrashing 같은 문제를 피하기 위해서 locality를 이용해서 page 개수를 정할 필요가 있어

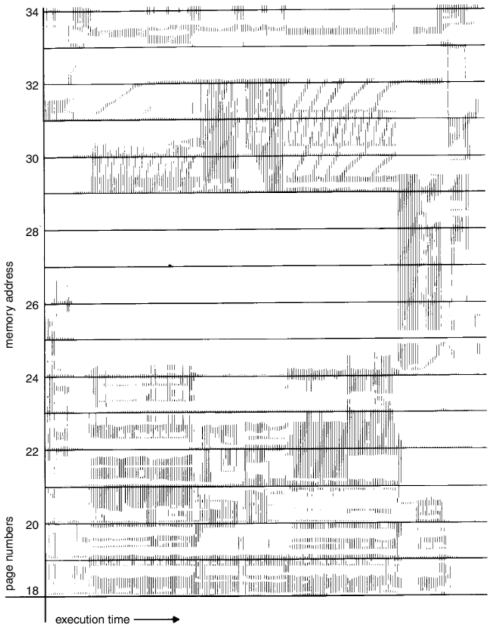
그래서 locality는 2가지가 있는덴

1. 시간 지역성

   현재 참조된 메모리가 다시 참조될 가능성이 높아 (반복문)

2. 공간 지역성

   하나의 메모리가 참조되면 주변의 메모리가 계속 참조될 가능성이 높아(배열 순회)



가로축이 시간, 세로축이 공간

실제로 하나의 시간을 고정해놓고 공간을 보면 접근이 많이된 것을 볼 수 있음

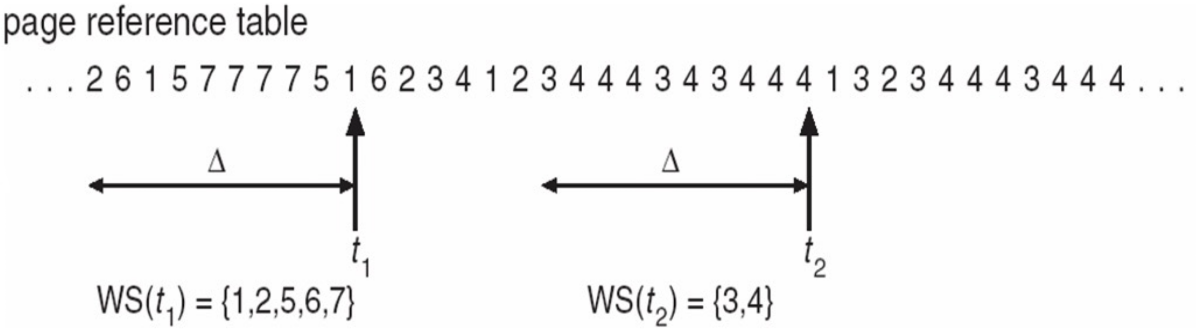
반대로 공간을 고정해놓고 시간을 보면 미래에도 계속 접근되는 것을 볼 수 있음

Working-Set Model

window를 설정해서 해당 window내에 존재하는 page들을 저장

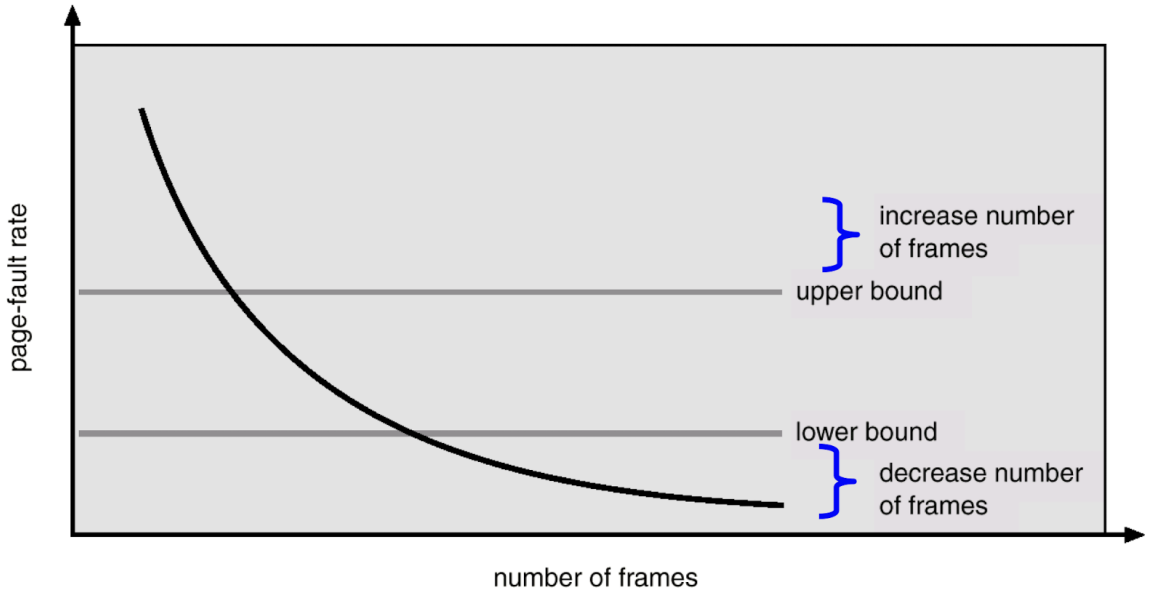
약간 슬라이딩 윈도우 생각하면 될듯?

그래서 window 내에 있는 working set의 page보다 많이 사용해야 Thrashing 방지할 수 있어



그래서 옮기면서 working set에 여전히 존재하면 메모리에 남기고 없으면 disk로 쫓아내고

Page-Fault Frequency Scheme



상한보다 많이 쓰는 process들은 frame이 더 필요하니까 하한보다 안쓰는 애들한테서 가져와

반대로 하한보다 적게 쓰는 process들은 상한보다 많이 쓰는 process들한테 줌

이거도 하나의 model인데

Working set 기법은 page 참조 시마다 페이지 집합을 수정한다면

Page-Fault Frequency는 page fault 발생 시에만 페이지 집합을 수정