10 가상메모리

Demand paging

: 실제로 *필요할 때 page를 메모리에 올리는 것*

얻는 효과

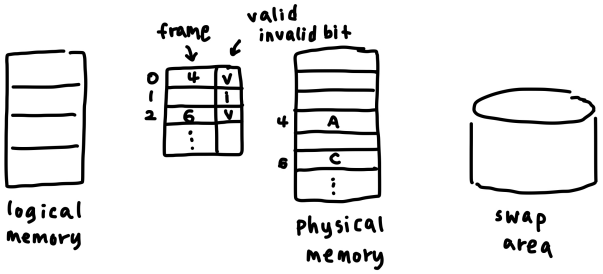
- I/O 양의 감소
- memory 사용량 감소
- 빠른 응답 시간
- 더 많은 사용자 수용

valid/invalid bit 사용

- invalid의 의미
  - 사용되지 않는 주소 영역인 경우
  - 페이지가 물리적 메모리에 없는 경우
- 처음에는 모든 page entry가 invalid로 초기화
- address translation 시에 invalid bit이 set되어 있으면 **page fault**

  : 요청한 페이지가 메모리에 없는 경우

  ⇒ 이러한 상황 발생하면 자동으로 *CPU는 운영체제로 넘어감*



→ 물리적 메모리에 올라와있으면 valid, frame 번호 존재

page fault

- invalid page 접근하면 MMU가 trap을 발생시킴 (*page fault trap*)
- kernel mode로 들어가서 page fault handler가 invoke됨
- 다음과 같은 순서로 page fault를 처리

1. invalid reference ? > abort process
2. 비어있는 page frame을 얻음 (없으면 뺏어옴 = replace)
3. 해당 페이지를 disk에서 memory로 읽어옴
    a. disk I/O가 끝나기까지 이 프로세스는 CPU를 preempt당함 (block)
    b. disk read가 끝나면 page tables entry 기록, valid-invalid bit = valid
    c. ready queue에 process를 insert ⇒ dispatch later
4. 이 프로세스가 CPU 잡고 다시 running
5. 아까 중단되었던 instruction 재개

performance of demand paging

- *page fault rate*

  if p = 0, no page fault → 다 메모리 안에 존재

  if p = 1, every refernce is a fault

- effective access time = (1-p) x memory access + p

  (OS & HW page fault overhead + swap page out if needed + swap page in + OS & HW restart overhead)

free frame이 없는 경우

- *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 최소화*하는 것이 목표
    - 알고리즘의 평가

      → 주어진 page reference string에 대해 page fault를 얼마나 내는지 조사

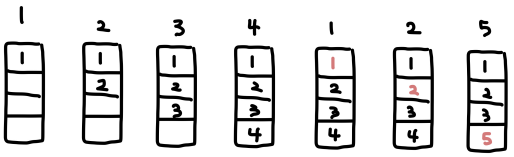
      ⇒ 시간 순서에 따라 page별로 번호를 붙이고 page들이 참조된 순서 나열

> 💡 메모리 위에 올라와서 write가 발생되면 쫓아낼 때 변경된 내용 backing store에도 써줘야 함

Optimal algorithm

→ page fault를 가장 적게 하는 알고리즘

MIN(OPT) : 가장 먼 미래에 참조되는 page를 replace



- 미래의 참조 어떻게 아는가? → offline algorithm

  ⇒ 미리 알고있다는 가정 하에 움직임 - 실제 online에서 사용 못함
- 다른 알고리즘의 성능에 대한 upper bound 제공

  → belady's optimal algorithm, MIN, OPT 등으로 불림

FIFO (First-in First-Out algorithm

FIFO : *먼저 들어온 것을 내쫓음*

→ 메모리 더 늘리면 page fault가 더 늘어나는 상황이 발생할 수 있음

- FIFO anomaly (Belady's anomaly)

  → more frames ⇒ less page faults

LRU (Least Recently Used) algorithm

LRU : *가장 오래 전에 참조된 것*을 지움

→ LRU는 오래 되어도 재사용되면 다시 최근 참조

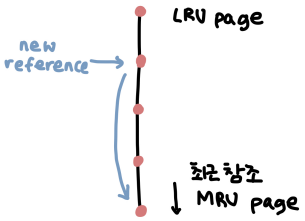
LFU (Least Frequently Used) algorithm

LFU : *참조횟수(reference count)가 가장 적은 페이지*를 지움

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

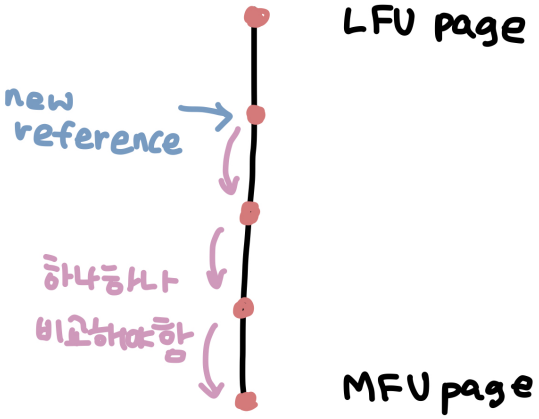
LRU와 LFU 알고리즘의 구현

- LRU
- LFU



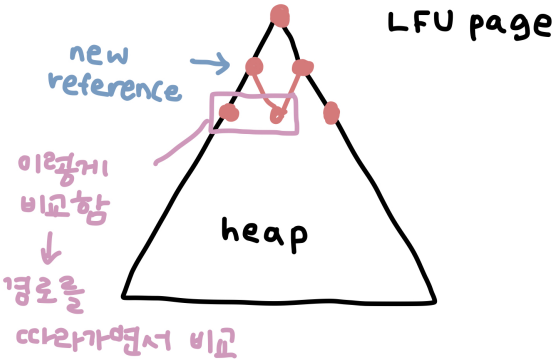
→ 페이지 참조시간에 따라 줄 세우기

시간복잡도 : **O(1)**



→ 한 줄로 줄 세우기 불가능

시간복잡도 : **O(n)**



시간복잡도 : $O(logn)$

다양한 캐싱 환경

- 캐싱 기법

  → *어떤걸 메모리에 올리고 어떤걸 내쫓을지*

  - 한정된 빠른 공간(=캐쉬)에 요청된 데이터를 저장해 두었다가 후속 요청 시 캐쉬로부터 직접 서비스하는 방식
  - paging system 외에도 cache memory, buffer caching, web caching 등 다양한 분야에서 사용
- 캐쉬 운영의 시간 제약

  → n개 다 살펴보고 무엇을 쫓아낼지 정하는건 매우 비효율적

  - 교체 알고리즘에서 삭제할 항목을 결정하는 일에 지나치게 많은 시간이 걸리는 경우 실제 시스템에서 사용할 수 없음
  - *buffer caching이나 web caching인 경우*

    → $O(1)$에서 $O(logn)$ 정도까지 허용
  - *paging system인 경우*

page fault인 경우에만 OS가 관여함

페이지가 이미 메모리에 존재하는 경우 참조시각 등의 정보를 OS가 알 수 없음

→ $O(1)$인 LRU인 list조각조차 불가능

paging system에서 LRU, LFU 가능한가?

이미 물리적 메모리에 내용이 이미 올라와있으면 CPU가 그냥 가져가기만 하면 되니까 **OS 개입 X**

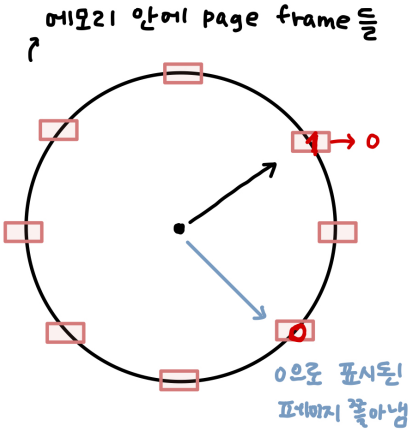
LRU 사용하려면 페이지 사용 시점을 다 알아야 함 → 이미 물리적 메모리에 있는 페이지면 OS개입하지 않아서 줄 세우기 불가능

→ page fault로 disk에 접근했을 때만 알 수 있음

Clock algorithm

- *LRU의 근사 알고리즘*
- 여러 명칭으로 불림 → second chance algorithm, NUR (Not Used recently), NRU (Noot Recently Used)
- **reference bit**을 사용해서 *교체 대상 페이지 선정* (circular list)
- reference bit가 0인 것을 찾을 때까지 포인터를 하나씩 앞으로 이동
- 포인터가 이동하는 중에 reference bit 1은 모두 0으로 바뀜
- refernce bit이 0인 것을 찾으면 그 페이지를 교체
- 한 바퀴 되돌아와서도 (=second chance) 0이면 그때는 replace 당함
- 자주 사용되는 페이지라면 second chance가 올 때 1

  → 한 번 turn동안 적어도 한 번은 사용됨



Clock algorithm의 개선

- reference bit과 **modified bit (dirty bit)**을 함께 사용
- *reference bit* = 1 : 최근에 참조된 페이지 → read 발생했을 때
- *modified bit* = 1 : 최근에 반영된 페이지 (I/O를 동반하는 페이지) → write 발생했을 때

→ modified = 1, reference = 1

⇒ 수정 발생하면 쫓겨날 때 수정된 내용 디스크에 쓰려고

> 💡 refernce bit 0 → 1 : 페이지 참조 ⇒ **하드웨어**가 바꿔놓음
> refernce bit 1 → 0 ⇒
> **운영체제**가 바꿔놓음

page frame의 allocation

원래 LRU는 어떤 프로세스의 page인지 고려하지 않고 쫓아냄 → 프로세스에 page를 미리 할당하겠다

allocation problem : 각 프로세스에 얼마만큼의 page fault를 할당할 것인가?

allocation의 필요성

- 메모리 참조 명령어 수행 시 명령어, 데이터를 여러 페이지 동시 참조

  → 명령어 수행을 위해 최소한 할당되어야 하는 frame의 수가 있음
- loop을 구성하는 page들은 한꺼번에 allocate되는 것이 유리함

  → 최소한의 allocation이 없으면 매 loop마다 page fault

allocation scheme

- **equal allocation** : 모든 프로세스에 똑같은 개수 할당

↓ 프로세스마다 크기 다 다를 수 있음

- **proportional allocation** : 프로세스 크기에 비례하여 할당

↓ CPU를 바로 사용할 것인가

- **priority allocation** : 프로세스의 priority에 따라 다르게 할당

Global vs. local replacement

- **global replacement**

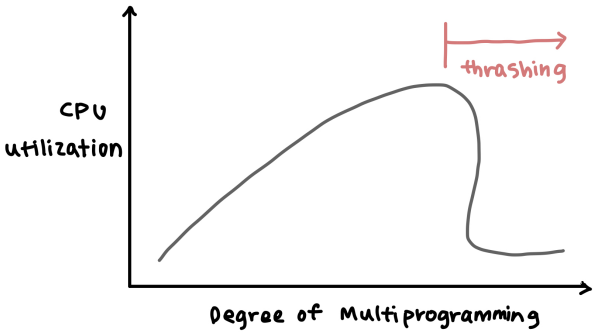
  → 쫓겨난게 어떤 process의 page인지 신경 안씀

  ⇒ 메모리 많이 사용하는 프로그램이 page를 많이 할당받음 → 특정 프로그램이 메모리 독식할 수 있음
  - replace시 다른 process에 할당된 frame을 빼앗아 올 수 있다
  - process별 할당량을 조절하는 또 다른 방법임
  - FIFO, LRU, LFU 알고리즘을 global replacement로 사용시에 해당
  - working set, PFF 알고리즘 사용
- **local replacement**

  → 일단 할당 후 자신(프로세스)에 해당하는 페이지 중 하나 쫓아냄 (이때 알고리즘 사용)

- 자신에게 할당된 frame 내에서만 replacement
- FIFO, LRU, LFU 등의 알고리즘을 process 별로 운영시

Thrashing algorithm



→ 메모리에 하나만 올라가 있을 경우 I/O 하는 동안 CPU는 놀기 때문에 메모리 이용률이 낮음

Thrashing

: 메모리에 너무 많은 프로그램이 올라와서 *프로그램이 원활하게 실행되기 위해서 필요한 최소한의 메모리도 얻지 못한 상황*

→ 계속 page fault가 일어남 (너무 메모리를 적게 가지고 있어서)

Thrashing

프로세스의 원활한 수행에 필요한 최소한의 page frame 수를 할당받지 못한 경우 발생

- page fault rate이 매우 높아짐
- CPU utilization이 낮아짐
- OS는 MPO(Multiprogramming Degree)를 높여야 한다고 판단
- 또 다른 프로세스가 시스템에 추가됨 (higher MPO)
- 프로세스 당 할당된 frame의 수가 더욱 감소
- 프로세스는 page의 swap in/swap out으로 매우 바쁨
- 대부분의 시간에 CPU는 한가함
- low throughput

↓ 프로세스가 필요한 최소한의 메모리는 보장

Working-set model

- locality of reference
  - 프로세스는 특정 시간동안 일정 장소만을 집중적으로 참조한다

    - 집중적으로 참조되는 해당 page들의 집합을 **locality set**이라 함
- working set model
    - locality에 기반하여 프로세스가 일정시간 동안 원활하게 수행되기 위해 한꺼번에 메모리에 올라와 있어야 하는 page들의 집합을 **working set**이라 정의함
    - working set model에서는 process의 working set 전체가 메모리에 올라와 있어야 수행되고 그렇지 않을 경우 모든 frame을 반납한 후 swap out (suspend)

      → 메모리 한개도 주지 않고 *suspend*
    - thrashing을 방지함
    - multiprogramming degree를 결정함

working-set algorithm

- working set의 결정
    - *working set window*를 통해 알아냄 → 과거를 통해 추정함
    - window size가 Δ인 경우
        - 시각 t_i에서의 woking set $WS(t_i)$

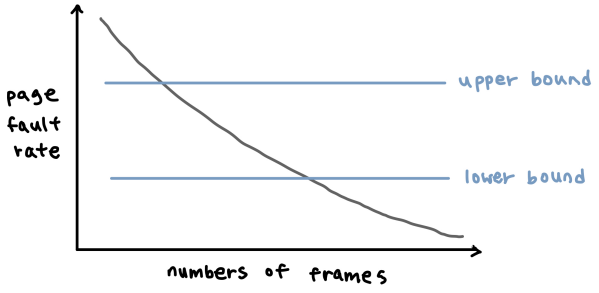
          → time interval $|t_i - \Delta_i t_i|$ 사이에 참조된 서로 다른 페이지들의 집합
        - working set에 속한 page는 메모리에 유지, 속하지 않은 것은 버림

          (즉, 참조된 후 Δ시간 동안 해당 page를 메모리에 유지한 후 버림)

→ global replacement

PFF (Page-fault frequency) scheme

→ *page fault rate보고 page 더 줄지 말지 결정*



- page fault rate의 상한값과 하한값을 둔다
    - page fault rate이 상한값을 넘으면 *frame을 더 할당*한다
    - page fault rate이 하한값 이하면 *할당 frame 수를 줄인다*
- 빈 frame이 없으면 일부 프로세스를 swap out

page size의 결정

*page size를 감*소시키면

- 페이지 수 증가
- 페이지 테이블 크기 증가
- internal fragmentation 감소
- disk transfer의 효율성 감소
    - seek / rotation vs. transfer
- 필요한 정보만 메모리에 올라와 메모리 이용이 효율적
    - locality의 활용 측면에서는 좋지 않음

trend

→ 요즘 메모리 크기가 커지면서 larger page size