11. Virtual Memory 2

Multi-Programming 환경에서, 각 Process에 몇 개의 Frame을 할당해야 할 지에 대한 문제가 생긴다.

각 Process가 실행되기 위해서는 **최소한의 Page**를 가져야 한다.

- 이것이 보장되지 않으면, page fault가 자주 발생한다.

하드웨어적인 관점

- IBM 370 SSMOVE 명령어를 처리하기 위해서는 최소한 6개의 Page가 필요하다.
  - 한 명령어가 6 byte라 2개의 Page가 필요하다.
  - 이동시킬 데이터가 이전에 저장되어 있는 공간 2 Page, 데이터를 이동시켜 저장할 공간 2 Page가 필요하다.
  - 위 명령어를 수행하는 데 최소한 6개의 page가 할당되어 있어야 Page fault가 발생되지 않는다.

소프트웨어적인 관점

- Loop 내의 모든 Page는 한 번에 allocation 되는 것이 유리하다.
  - 그렇지 않다면 매 Loop마다 page fault가 발생해 **CPU/Disk load 간의 심한 불균형**이 발생한다.
    - Disk I/O는 증가하고, CPU는 idle해지는 경우가 많다.
  - 즉, 이 Program의 locality에 포함되는 데이터나 명령어가 메모리에 안정되게 올라가 있을 공간이 필요하다.

즉, Page fault가 너무 자주 발생하지 않으면서, program이 정상적으로 동작하기 위해서는 최소한의 개수의 page frame 할당이 보장되어야 한다.

이러한 문제를 해결하기 위해 Page frame allocation 방법에 대해 살펴보자

Allocation scheme

Fixed Allocation

1. Equal allocation : **모든 process에 동일한 size의 page를 할당한다.**
    - EX) 100개의 frame에 5개의 process라면, 각 process에 20개의 frame을 할당한다.

2. Propotional allocation : **각 Process의 size에 비례하여 할당**

— s_i = size of process p_i
— $S = \sum s_i$
— m = total number of frames
— a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

- 이와 같이, 각 process의 사이즈에 비례하여 할당한다.
- 그러나 이는 동적 할당 등의 경우에 부정확하다.

Priority Allocation

- Process의 size 대신, Priority 기준으로 frame을 propotional allocation

Priority가 크면 Page fault가 적어 I/O가 적고, **process가 waiting하는 경우가 적어 빨리 끝날 수 있다.**

Prioirity가 작으면 Page fault가 많아 I/O가 늘어나고 Waiting을 많이 해야 한다. 이에 따라 **process는 느리게 끝나게 된다.**

이를 지원하기 위해 Priority가 큰 process에 더 많은 Frame을 할당한다.

Process에서 page fault가 발생한 경우

- 자신의 Frame 중 하나를 victim으로 선정한다.
- Lower priority process의 frame 중 하나의 frame을 victim으로 선정한다.

그러나, 처음에 priority만을 가지고, Frame 할당을 사전에 결정하는 방식이 정확할까?

- `Global replacement` 방법이 이 문제를 해결할 수 있다.
- Priority가 높은 것에 너무 많은 Frame을 할당하고, Priority가 작은 것에 너무 적은 Frame을 할당한 경우, Priority가 낮은 process에서 너무 많은 page fault가 발생하는데 Priority가 높은 process에서는 frame이 놀게 된다.

Replacement 방법으로는 아래 두 가지가 있다.

1. `Global` **: 다른 process의 frame도 victim으로 선정할 수 있도록 한다.**
   - Priority가 낮은 process에서 Frame을 **넉넉하게 할당 받은 process의 frame을 victim으로 선정**할 수 있도록 하자
     - 위 문제를 해결할 수 있다.
   - `Second-chance algorithm`에서 각 process마다 작은 Circular queue를 사용하는 대신에, 하나의 큰 Circular queue를 사용하는 방식으로 구현할 수 있다.
   - 일반적인 경우에 Local replacement보다 더 많이 사용된다.

2. `Local` **: 각 Process에 할당된 범위 내에서만 VIctim을 선정할 수 있도록 한다.**
   - 특정 System이 돌아가는 메커니즘이 뻔하거나, program이 특수한 목적만 수행하는 경우에는 Local replacement + Fixed allocation을 사용해도 무방하다.

Thrashing

- 아무리 좋은 Replacement algorithm을 가진다고 하더라도, **각 process에 부여된 Page frame 자체가 부족해지는 순간부터 답이 없다.**
  - 하나의 Process에 할당된 Frame의 개수가 너무 적은 경우
  - Page fault rate가 급격히 증가한다.

- 이는 아래 상황들로 이어진다.
  - `Low CPU Utilization` : page fault가 너무 많이 발생해 waiting time이 증가하면 CPU utilization이 감소
  - OS는 CPU utilization이 낮아지는 상황에, Multiprogramming degree(MPD)를 높여야 한다고 판단한다.
    - Long-term scheduler가 새로운 process를 ready queue(시스템)에 추가하게 된다.
  - 새로운 Process가 실행되기 위해선 기존 Process의 Frame 중 일부를 뺐어야 한다. 이는 Page fault의 증가와 CPU Utilization의 감소로 이어진다.

위 같은 상황이 이어지면, Process는 Page swap in / out 만 하게 되고, CPU는 idle 상태에서 계속 있게 된다.

- 이처럼 **process가 page fault가 너무 많이 발생하여, Swap in / out만 하고 실질적인 실행은 하지 못하는 상태를** `Thrashing` 이라고 한다.
- Throughput이 감소하고 CPU Utilization이 감소한다.

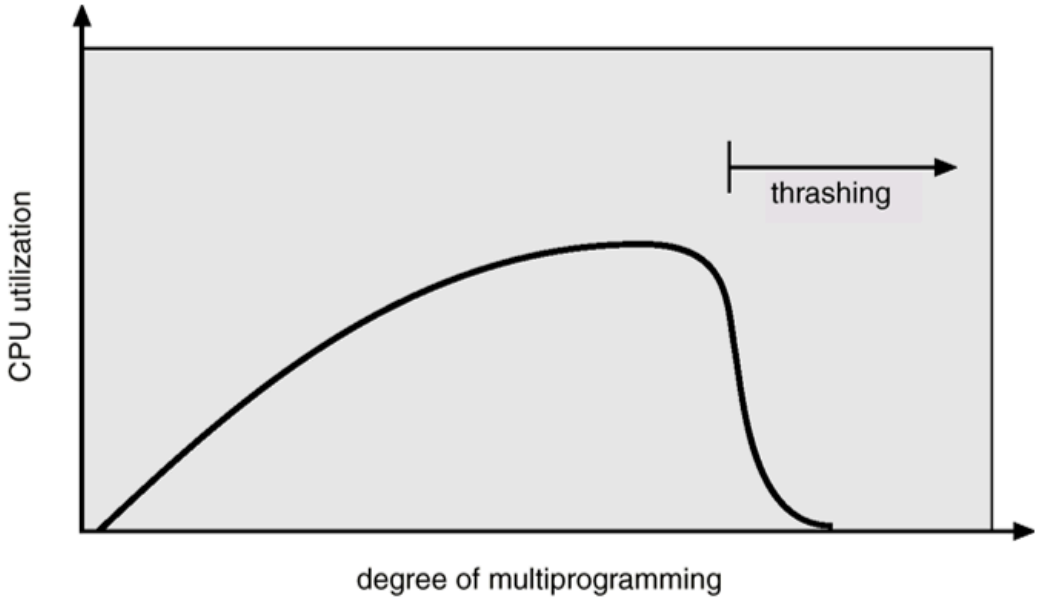
예시는 아래와 같다.

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

A
main()
X
B

- 이 경우, `main()` , `A` , `B` , `X` 를 위한 Page가 있고 `I` 를 위한 page frame은 존재하지 않는다.
- Frame 딱 하나가 부족해서, Loop 한 번 돌 때마다 page fault가 발생한다.

Thrashing Diagram



- 초반에는 MGD가 커질수록 CPU Utilization이 커진다.
  - 초반에는 CPU idle 시간이 줄고, context switching이 효율적으로 CPU를 활용하게 한다.
- **특정 시점에 Process가 딱 한 개만 더 늘어나도 CPU Utilization은 급격히 감소**한다.
  - 이는 `Thrashing` 때문이다.

위에서 확인한 것처럼 Thrashing은 굉장히 위험해서 절대 발생하면 안된다.

- 그러나 OS는 Thrasning을 예방하기 위해서 동시에 실행되는 Process의 개수를 제한하는 등의 행동을 하지 않는다.
  - OS가 process의 개수를 제한하는 순간 욕을 먹기 때문이다.
- Thrashing은 온전히 User가 감당하는 것이기 때문에 User가 한 번에 너무 많은 Program을 실행하지 않아야 한다.

Thrashing을 process의 개수를 제한하지 않고 예방하는 방법은?

- 각 Process가 자주 접근하는 Page들이 Memory에 다 올라와 있도록 보장해야 한다.
- 이를 위해선 OS가 locality를 알아야 하는데, OS가 locality를 정확히 알 수도 없고 Locality가 고정된 것도 아니기 때문에 쉬운 일이 아니다.

Locality를 정확하게 알 수 없고, 직접적으로 막을 수 있는 방법도 따로 없다면, 우리는 Thrashing이 왜 발생하는지 근본적인 원인만 알면 된다.

- **(전체 process에서의 locality size의 합) > (Total allocated memory size)**
- 언제 Thrashing이 발생하는 지를 정확히 이해하여 예방 및 대비

우리는 Locailty를 직접적으로 알 수는 없어도, 간접적으로 확인할 수 있어야 한다.

Locality에 대해 먼저 알아보자

Locality

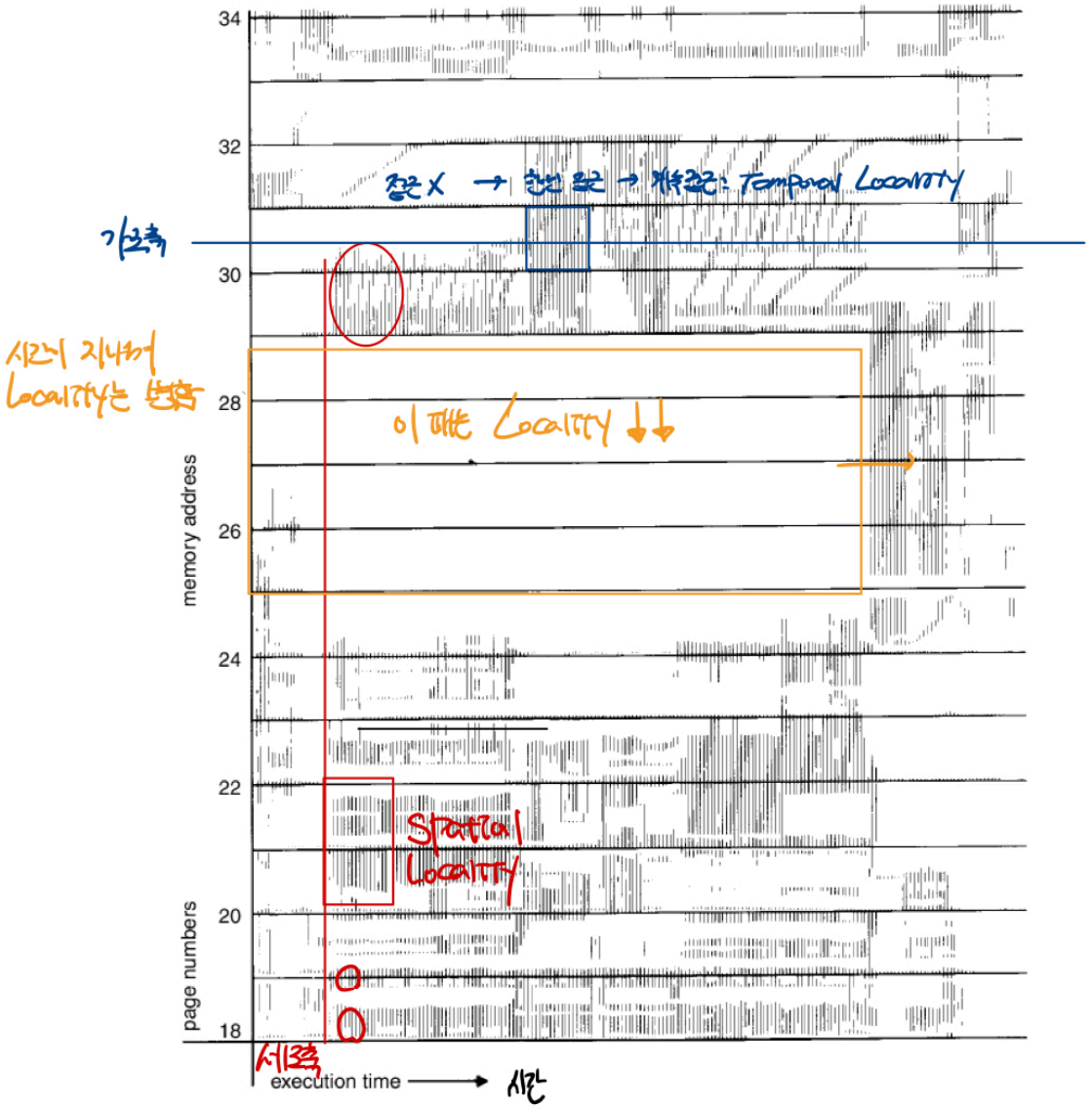
- Program의 메모리 참조는 고도의 Locality를 갖는다.
- **임의의 시간 간격 동안 program의 일부분을 집중적으로 참조한다.**
- 시간 지역성 : 현재 참조된 메모리가 가까운 미래에도 참조
- 공간 지역성 : 하나의 메모리가 참조되면 주변의 메모리가 계속 참조될 가능성이 있다.

Locality는 paging이 가능하게 해준다.

- 만약 program이 locality를 갖지 않고, 모든 부분을 참조한다면, DRAM은 부족하다.

Locality는 겹칠 수 있다.

- Program이 한 번에 하나의 부분만 집중적으로 참조하는 것이 아니라 다른 부분도 같이 집중적으로 참조할 수 있다.



- Locality를 시각적으로 확인할 수 있다.
- **시간에 따라서 Locality는 변한다**

Working-Set Model

Working Set : Program이 일정 시간 동안 접근한 Page들의 집합

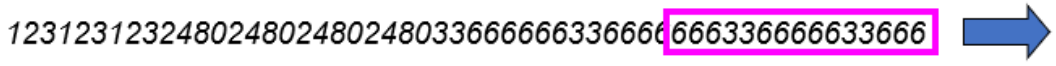
- Memory에 올라와 있고, 일을 하는 Page에 집합
- 일을 한다 = CPU가 접근 한다.

Working Set window : page reference의 fixed number

WS (Working Set) 은 아래와 같이 정의 된다.

$WS(t_i) = \{ \text{pages referenced in } [t_i, \ t_i - \Delta] \}$

- if Δ too small, will not encompass entire locality
- if Δ too large, will encompass several localities
- if $\Delta = \infty \Rightarrow$ will encompass entire program



- 위의 예시에서 WS(T_i) = {3, 6}

Window의 값이 굉장히 중요하다.

- Window가 너무 작은 경우
  - 큰 Loop와 같이 Locality가 굉장히 큰 경우, Loop의 전체 Page를 포함하지 못 할 수도 있다.
- Window가 너무 큰 경우
  - 현 시점에서는 Locality가 아닌 것까지 포함하게 된다.

Working set model **은 Working set을 조정하여, 현재 시점에서의 locality를 파악해 Thrashing을 예방하는 모델이다.**

WSS_i = Working set size for process P_i

$D = \Sigma\ WSS_i \equiv$ total demand frames

If $D > m \Rightarrow$ *Thrashing* (m is total number of available frames)

Thrashing을 어떻게 탐지?

- **D > m** 인 경우에 thrashing이 발생한다.

Working set model은 아래와 같이 동작한다.

가정)

1. **Process 전체가 각각의 WSS 만큼의 Frame을 얻어야 정상 동작할** 수 있다고 가정하자.
2. Process는 **정상 동작하거나 Suspend 되는 두 가지 경우**만 있다고 가정하자.

동작)

1. **만약 D > m이라면 기존 process 중 하나를 suspend 한다.**
   - Thrashing이 발생할 것 같은 경우를 아예 방지한다.
   - D > m이 해결될 때까지 하나씩 Suspend 한다.
2. **만약 특정 process가 자신의 WSS 만큼의 Frame을 할당받지 않은 채로 동작한다면, Suspend 한다.**
   - 모든 Process가 자신의 Locality 만큼의 frame을 할당 받음을 보장하기 위함

Process 하나를 Suspend하면 MGD가 1씩 줄어든다.

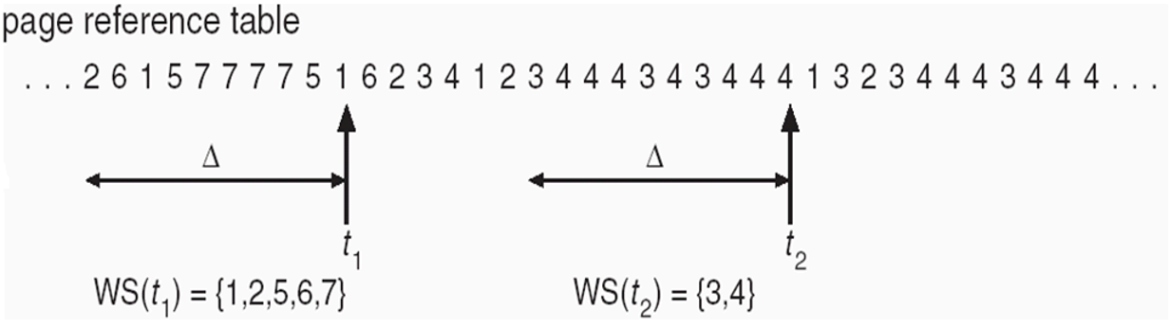
- Suspend 하는 이유는 Kill보다 Suspend를 하게 되면, 특정 작업을 마친 process가 메모리에서 빠져나갈 때, Suspend 된 것을 다시 실행시킬 수 있기 때문이다.

- 전체적인 Throughput 관점에서, **Thrashing이 발생하는 것보다 일부 Process를 Suspend하는 것은 전체 Process와 Suspend된 process 모두의 관점에서 효율적이다.**
    - 전체적인, 그리고 본인의 효율성을 위해 Suspend 되는 Process가 양보한다고 생각하면 된다.

Working set model이 정확하게 동작하기 위해서는 결국 Working set을 어떻게 찾는 지가 가장 중요하다.

- 현재 시점의 Locality를 알기 위해 필요하다.
- Locality가 높은 Frame은 유지될 수 있도록 충분한 Frame을 할당
- **해당 Frame은 절대 Victim으로 선정되지 않도록 해야 한다.**

Example)



- 시간이 변하면 **Working set size도 달라지고 Working set도 달라진다.**

Working-Set Model은 **특정 Page P가 WS에 속해 있다면 메모리에 유지하고, WS에 속해 있지 않다면 메모리에서 내쫓는다.**

Working set model을 Allocation, Replacement에 이용할 수 있다.

1. Global replacement
    - WS에 없는 frame을 전체에서 찾아 Victim으로 선정

2. Local replacement
    - 현재 process의 frame에서 WS에 포함되지 않은 frame을 찾아서 Victim으로 선정

Allocation은 WSS에 따라서 결정하게 된다.

WS(t_i)에 존재하는 Page가 전부 메모리에 올라와 있는 것이 보장되어야만 실행하고 아니면 Suspend 한다.

Working Set Implementation

이론을 그대로 구현: **각 Memory reference 마다 각 Page들의 최근 Refence time을 ws-window와 비교하는 방법**

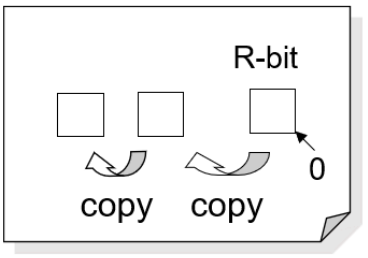
- Space overhead : 각 page마다 Ref-count를 적는 time field가 필요하다.
- Time overhead : 전체 window와 time field를 비교하는 데에 시간이 소요된다.

그래도 구현하지 말고, Working set model을 Approximation 해서 사용하자

Approximate with interval timer + a reference bit

Example: Δ = 10K,

- Timer interrupts: every 5K time units.
- Keep in memory 2 bits for each page.
- timer interrupts → copy and resets all reference bits
- If one of the bits in memory = 1 ⇒ page belongs to the working set



<u>만약 Window = 10K라면, 1/2배인 5K마다 Timer interrupt를 보내 R-bit를 0으로 만들고 기존 bit를 left shift 한다.</u>

- 각 page에 2 bit만이 추가로 필요하다.
- Reference bit는 기존 swapping 때의 구현과 동일하다.

- Timer interrupt 시 R-bit를 0으로 초기화 하고, Left bit에 기존 값 copy
- 위의 **bit 중 하나라도 1이라면 WS에 포함됨**을 확인할 수 있다.

그러나 이 방법은 정확하지 않다.

방금 막 Copy 된 경우 (Timer interrupt가 온 경우)에는 additional 2 bit가 10K (window) 까지의 범위를 정확하게 나타내지만, **다음 Timer interrupt 직전에는 가장 왼쪽에 있는 비트에는 14999까지의 reference 정보**가 담겨져 있다.

- 이는 **기존 Working set size보다 150% 초과하며, 50%의 에러가 발생**한다.

해결법)

- **Timer interrput를 1K에 한 번 오도록 하고, Additional bit를 10개로 늘린다.**
- 이 경우에는 오차가 10%로 감소한다.

- 그러나 이 경우. **Page 마다 관리하는 bit 수가 증가해** `Space overhead` 가 발생하고, **Left shift와 각 page에서 10개의 bit를 탐색하는 데에** `Time overhead` 가 발생한다.

결국에는 **Overhead와 정확도간의 Trade-off 가 발생**한다

- 딜레마

그렇다면 working set window의 사이즈는 어떻게 결정해야 할까?

- Process마다 WSS도 다르고, 성격도 다르다.
- **이것이 실제 구현이 어려운 가장 큰 이유이다.**

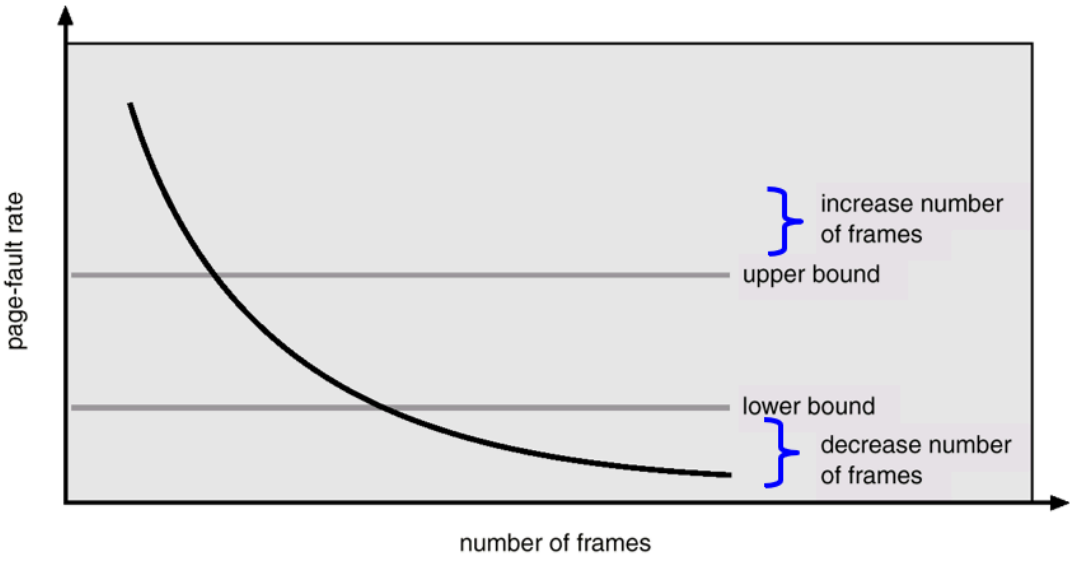
Working-set model은 구현이 너무 어렵다. 그렇다면 **Page fault가 많이 발생하는 것을 막을 다른 방법**은 없을까?

Page-Fault Frequency Scheme

Working set model을 구현하는 것도 어렵고, Locality을 정확하게 파악하는 것도 너무 어렵다.

그렇다면 **Process를 blackbox처럼 사용해서 현재 이 Process가 Memory가 부족한지 여부를 확인할 수 있는 Signal을 추가하자.**

- Process에서 **Page fault가 얼마나 자주 발생하는 지를 체크**해 Page fault가 많이 발생하면 Page가 부족함을 인지한다.
- Acceptable page-fault rate



- 기본적으로 Page frame의 개수가 늘어날 때마다 Page fault rate는 감소한다.
- **Page fault rate가 큰 process는 page가 부족하다는 것을 의미하고, Page fault rate가 작은 process는 page가 넉넉하다는 것을 의미**한다.
- 이를 이용하여, **Page fault rate가 낮은 process가 Page fault rate가 큰 process에 page를 줄 수 있도록 하자.**
- 즉, **Page fault 발생 시, Page set 을 수정하게 된다.**

이 방법은 **추가적인 Replacement algorithm이 있다는 전제 하에 사용**한다.

- 별도의 Replacement algorithm이 없다면, **Page fault rate가 낮은 process에서 어떤 Page를 replacement 해야 할 지 알기 위해서 또 Locality가 필요**하다.
  - 이런 경우에는 위 방법을 사용하나 마나
  - Fault의 개수를 알기 위한 용도로만 사용하게 된다.

Overhead

1. **Working set model**
   - Page reference 마다 window를 이동하고 Working set을 수정해야 한다.
   - Overhead가 크다.

2. **Page fault frequency scheme**
   - Page fault 발생 시에만 처리하면 된다.
   - **Page fault 발생 횟수는 Page reference 횟수에 비해 작다.**
   - Overhead가 비교적 작다.

VM의 다른 장점

- Process 생성 시의 장점을 살펴보자

Copy-on-write

기존 fork()는 **parent address space를 child address space에 copy**하는 방식이다.

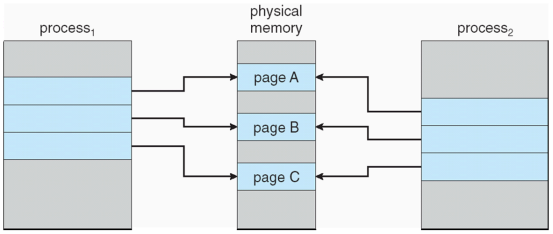
- Overhead가 크다.
- VM이 없다면 모든 Process가 각각의 Address space를 가져야 하기 때문이다.

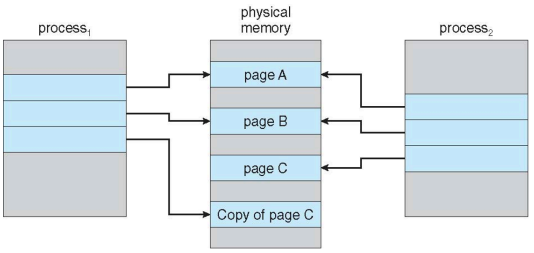
VM을 사용한다면, Process는 logical address만 확인하기 때문에 Parent process의 데이터를 child process에 굳이 복사할 필요가 없다.

→ **Parent process의 address space를 child process가 공유하도록 page table을 복사**한다.

Copy-on-write

- exec() 호출 등 값을 직접 write 하기 전까지는 **address space를 공유하다가 write 시도할 때 새로운 Copy를 생성해서 Copy에 write할 수 있도록 하는 방법**





첫 번째 사진은 fork() 직후의 상황을 나타낸다.

두 번째 사진에서, **Process 1이 Page C에 write를 하고자 하면, 이때 새로운 page를 할당하고, Page C의 데이터를 copy한다.**

- 이후 Process 1은 Copy본에 write할 수 있다.
- Mapping table을 수정해야 한다.

Code (text section / Read-Only)와 같이 공유해야 하는 부분에 대해서는 공유하면 된다.

두 Process가 완전히 독립적인 Address space를 갖기 위해서는 두 Process가 모든 Page에 대해 write 요청을 해야 한다.

Memory - Mapped Files

- **여러 Process가 하나의 File을 공유하고자 하는 경우에, Memory에 하나의 Copy file을 등록하면 된다.**

`Memory-mapped file I/O`

- **file I/O를 메모리 접근처럼 처리**할 수 있도록 **process가 접근하고자 하는 file을 process의 logical address에 매핑한다.**
- EX) 특정 파일의 몇 번째 btye = logical address의 몇 번지

기존에는 file 접근 시, `read(), write()` system call을 호출했다.

- 하지만 Memory-mapped file 사용 시, **logical address가 file에 매핑되어 있으므로 memory access로 접근한다.**
- 때문에 첫 번째 파일을 읽을 시에는 `demand paging`
- **file이 Page-size로 쪼개져서 읽어지고, memory에 저장**되어야 한다.
- 연속적인 file read/write는 oridnary memory access 처럼 다루어진다.
- 기존 file system은 오래 걸렸는데, 이 방법을 사용하면 file system을 거치지 않고 `Load, store` 명령어를 사용하여 효율적이다.

`Worst case` : 전체 파일을 전부 다 사용해 scan 하는 경우

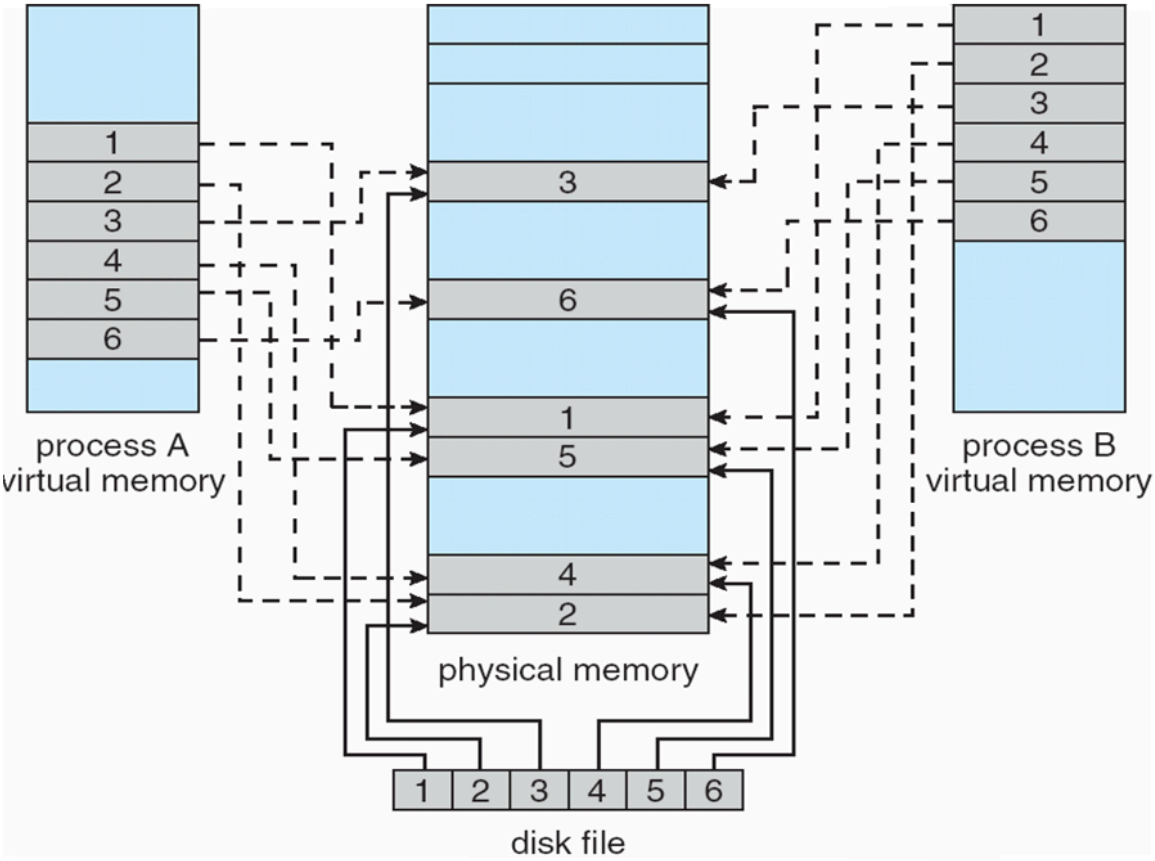
- 전체 File이 한 번씩 올라오는만큼 Page fault가 발생
- fscanf() 시 1 byte를 읽어오는데, 1 byte마다 disk I/O 발생

`Best case` : 어떤 File의 첫 번째 byte만 읽고 close

- 한 번의 Page fault로 종료된다.
- 한 Page만 읽는 경우도 동일

VM을 사용하지 않을 시, Memory - mapped file을 사용하기 위해선, 파일의 모든 page가 memory에 올라가야 해서 오히려 비효율적이다.

같은 file을 공유하는 경우 **memory에 있는 같은 page를 다른 process가 매핑**하게 만들면 된다.



- 만약 read-only file 이라면, 메모리에 하나의 copy만 있어도 된다.

VM과 상관없는 다른 이슈

Prepaging

- process의 시작 단계에서 발생하하는 page fault를 줄일 수 는 없을까?
- 어차피 코드는 시작하므로 Text section의 **처음 몇 개의 page는 memory에** 올리자
- **즉, reference되기 전에 process가 사용할 만한 page를 미리 올려놓자.**

처음에 올릴 page가 **너무 적으면 똑같이 처음에 page fault가 발생**해서 문제이고, **너무 많다면 쓸데없는 I/O와 memory 공간 낭비**가 발생해서 문제이다.

Assume *s* pages are prepaged and *α* of the pages is used

- Is cost of *s* * *α* saved pages faults > or < than the cost of prepaging *s* * (*1*-*α*) unnecessary pages*?*
- *α* near zero ⇒ prepaging loses

- `s`: 미리 올린 page의 개수, `a` = 미리 올린 page 중 실제 사용한 비율
- Page fault를 save한 개수는 s*a, page fault를 막지 못한 개수는 s*(1-a) 이다.
  - s*a > s * (1-a) 이어야 의미가 있다.
- a = 0이라면 **prepaging loss**

Page size

Page size를 키웠다고 가정해보자

장점)

- Page 개수가 적어진다.
- Page table size가 작아진다.
- Locality가 더 적은 수의 page가 들어갈 수 있다.
- 한 번의 Page fault가 더 많은 Page를 가져와 I/O 호출 횟수가 줄어든다.

단점)

- Internal fragmentation의 증가
- 되게 작은 것을 저장하기 위해 더 큰 page를 할당해야 한다.

TLB Reach

- **TLB를 통해 주소 변환이 가능한 주소의 범위**
- **TLB Hit를 통해 얼마나 많은 Page에 접근할 수 있는 지를 나타낸다.**

TLB hit가 증가하면 page table을 확인할 필요가 없어 memory access time이 줄어든다.

`TLB Reach` = (`TLB Size`) * (`Page size`)

- TLB size는 하드웨어적으로 고정이 되어 있어 바꾸기 어렵다
- TLB reach를 키우기 위해선 Page size를 키워야 한다.
  - Internal fragmentation의 위험이 있긴 하다.

이상적으로는, 각 Process의 working set에 해당하는 page가 TLB에 저장되어 있으면 좋다.

- 그렇지 않으면 TLB miss가 과하게 발생할 것이기 때문이다.

Page size를 키우면 Internel fragmentation의 위험이 있다.

→ `Multiple page size` : OS가 다양한 사이즈의 Page를 제공

- Internal fragmentation의 위험이 있을 때에는 작은 size의 page를 할당한다.

Program structure

VM과 Demand paging을 사용할 때, **코딩을 잘하면 Page fault를 줄일 수 있다.**

각 row가 한 page에 저장될 수 있고, page의 개수는 128개보다 작아 충분하지 않은 경우를 생각해보자.

- `int data[128][128];`
- Each row is stored in one page
- Assume that the # of free frames for data < 128

Program 1 은 **Column을 고정하고 row먼저 바꾸는 방식**이다.

Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

128 x 128 = 16,384 page faults

- 이 경우, i 변수를 사용하는 반복문 실행마다 page fault가 발생한다.
- LRU, FIFO를 사용한다고 가정

Program 2 는 **Row을 고정하고 column 먼저 바꾸는 방식**이다.

Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

128 page faults (even when there is only one page frame for data)

- 이 경우에는, j 변수를 사용하는 반복문에서는 page fault가 발생하지 않게 된다.

OS가 row base로 배열을 저장하는지, Column base로 배열을 저장하는지에 따라 코딩 방식을 다르게 해야할 필요가 있다.

I/O Interlock

어떤 Page는 victim으로 선정되면 안 된다고 선언하는 것

EX) Disk to USB copy

1. Disk에서 파일을 읽어 Memory buffer 에 올린다.
2. Mmory buffer에서 USB로 복사
    - (1) → (2) 과정에서 replacement algorithm에 의해 교체되면 다시 Disk에서 load 해야 한다.
    - USB에 copy 할 때까지 해당 page는 victim으로 선정되면 안 된다.
    - 따라서 `Pin (Page interlock`)으로 표시한다.