**운영체제 8장 가상메모리**

2015253039 권진우

8장

배경 / 개념

가상(Virtual)메모리 : DISK <--> Memory

프로그램의 사용주소 : 논리주소 실제 메모리 : 물리주소

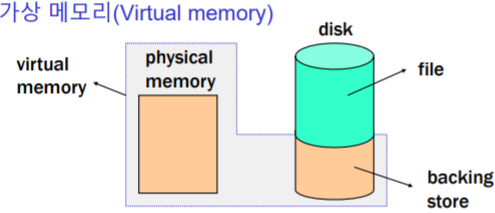
**가상 메모리란 : 주 메모리가 아닌 것을 메모리처럼 사용**

**현재 수행되는 실행 명령어는 물리메모리에 적재**되어 있어야 함

- 프로그램 전체 논리주소 공간(프로그램 전체)을 물리 메모리가 적재(1:1) 후 실행

- 동적 적재(수행하는데 필요한 부분만 동적을 나눠서 적재)

- 가상 메모리 사용 : 실행에 필요한 부분만 물리메모리(주 메모리)에 적재



주 메모리 디스크의 가상 메모리공간

\*가상 메모리 : 주 메모리 + 디스크의 가상 메모리

\*주 메모리에 실린 데이터는 현재 많이 사용되는 데이터부분, 디스크의 가상메모리공간은 **페이지 단위**로 주 메모리와 **Swapping**이 일어남

**가상 메모리**

- 프로그램의 **논리 메모리와 물리 메모리를 분리**

- **프로세스 전체가 메모리에 적재되지 않아도 실행**가능

- 따라서, 물리적 메모리(실제 메모리) 크기보다 큰 프로그램 실행 가능(몇몇 페이지씩 적재+Swap)

- Disk와 주 메모리(RAM) 사이에 **페이지 단위로 Swap in/out**-> **입출력 크기 감소** = 성능 향상

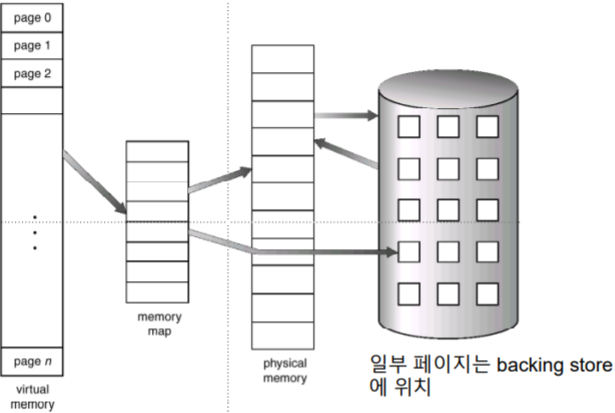
-> 프로그램 최초 실행 시 프로세스의 **부분적인 데이터만 적재**해도 되므로 **실행** 속도 향상 + 페이지 단위(크기가 작음)로 Swap 되므로 성능 향상

가상 메모리 구현 방법

-**요구 페이징(Demand Paging)** : 현대 OS들이 많이 사용, 필요 시 요청

-요구 세그먼테이션 : 프로세스를 논리적 단위로 나누기 때문에 가변크기 때문에 복잡하여 사용X

물리적 메모리보다 큰 가상 메모리



\* **일부** 페이지는 **디스크**의 가상 공간에 있고 **일부** 페이지는 **물리적 메모리** 공간에 있다.

\* 페이지 단위 스와핑을 통해 필요한 페이지를 가져와서 수행

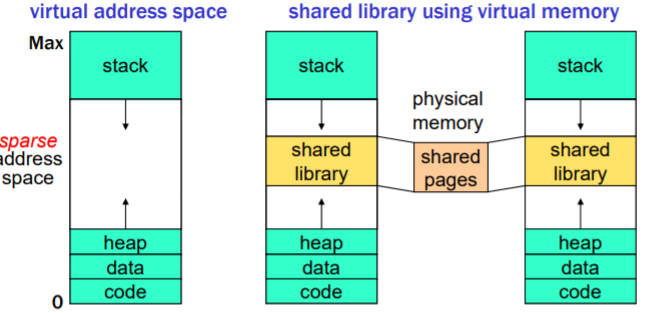
**가상 메모리를 통해 메모리 공유가 쉬워짐**

- 가상 메모리는 프로세스들이 메모리와 파일 공유를 가능케 함

-공유 라이브러리

-공유 메모리

-fork()에 의한 프로세스 생성 동안 **부모와 자식프로세스가 페이지 공유**



\* **처음에는** 주 메모리에 공유 라이브러리가 적재되어 있지 않아서 **Disk에서 공유 라이브러리를 가져와서 주 메모리에 적재**하고 난 **후부터는 물리 메모리(주 메모리)의 라이브러리를 공유**하게 된다.

\* **위에서는 가상 메모리 공간에 공유 라이브러리가 있고 공유 라이브러리가 물리적 메모리의 페이지를 공유하여 사용함으로써 물리 메모리 자원을 절약**할 수 있다.

**요구 페이징**

- 해당 Page가 필요할 때 Page를 **Swap in 하거나 Swap out + Swap in**하여 필요한 Page만 적재

- 요구 페이징을 수행하는 프로그램을 **Lazy swapper(요구할때까지 기다림)혹은 Pager**라고 부른

다.

요구 페이징의 **장점**

1. **입출력 시간 감소** : **페이지 단위(전송 단위小) 입출력**을 통한 Swap시간 감소

2. **메모리 공간 절약** : 필요 페이지만 올림

3. **빠른 응답 시간** : 프로그램 **최초 실행 시 모두 적재하지 않고 필요 페이지들만 적재**하여 실행하기 때문에 빠른 응답 \*프로세스의 데이터에 대한 *액세스 속도가 빠르다는 것이 아님*

4. **향상된 멀티 프로그래밍** : 더 많은 프로세스 동시 적재 가능 = **더 많은 사용자 허용**

\*예를 들어 프로그램을 실행하는데 예외처리 코드와 같은 예외적인 부분은 프로세스 실행 내내 사용되지 않을 수 있음. 따라서 요구 페이징을 사용하면 메모리 공간을 절약할 수 있음

요구 페이징 **구현**하기

- **Valid bit 사용**한 Page table, 보조기억장치(HDD, SSD) 필요

- 각 **Page table마다 Valid bit**(1비트)씩 추가

- **'1'** = 액세스하려는 데이터가 메모리에 **있음**

- **'0'** = 액세스하려는 데이터가 메모리에 **없음**

- Valid bit **초기값은 '0'** -> 페이지가 메모리에 적재 될 때 Page table 갱신하며 '1'로 바꿈

-> Page table을 사용한 주소 변환 과정에서

-> **Valid 비트가 1이면 메모리 참조**

-> **Valid 비트가 0이면 메모리에 없음** -> Page fault trap발생 -> trap handler가 처리

-> 메모리에 없지만 **Disk에 있으면** -> 해당 페이지를 **메모리에 적재 후 실행**

-> 메모리에 없고 **Disk에도 없으면** -> 잘못된 참조(invalid reference) -> 프로세스 중단(**Abort**)

\***Trap handler(ISR)는 Page fault trap이 발생했을 때** Disk에서 해당 페이지를 찾고 있으면 메모리에 적재, 없으면 Abort를 수행하는 **OS**의 인터럽트 서비스 루틴 중 하나이다.

**\*최종적으로 Page fault는 가상 메모리 공간(= DISK)에서 발생한다.**

**Page fault**의 **처리과정**

**1. Page table의 해당 Page entry를 탐색**하여 invalid 원인(V비트가 '0'일 때)을 알아냄

-> Disk에도 없으면 Invalid reference(잘못된 참조) -> **Abort(중지)** -> 프로세스 종료(끝)

-> 메모리에 없고 Disk에 있으면 -> **메모리로 해당 페이지 적재**

**2.** 주 메모리 공간에 **빈 Page frame을 찾음** = RAM 빈 공간 찾기

**3.** **Disk의 Page를 메모리의 Page frame으로 Swap in(적재)** <- Disk 읽기가 필요함(그동안 wait <I/O bound>)

**4.** **Disk read가 끝나면 Page table 갱신** -> 적재된 **Page table entry의 Valid bit를 '1'로 바꿈**

**5**. Page fault trap의 ISR(Page handler)이 끝났으므로 **중지된 부분부터 명령어 수행 재시작**

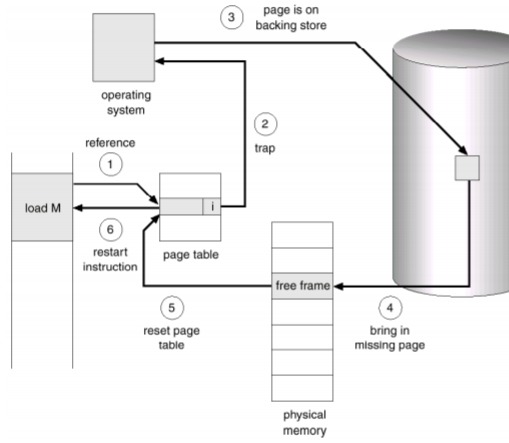
**순수 요구 페이징**

- 요구 받기 전에는 Page를 적재하지 않기 때문에 최초 프로세스를 실행 할 때는 적재되는 **모든 Page frame에 대해 Page fault**가 발생 -> 모두 Page fault trap을 통해 Disk에서 메모리로 필요한 Page들을 적재

**메모리 참조의 지역성**

- 페이지 적재 시에 **연속된 몇 개의 페이지들을 미리 같이 적재**해둠-> 액세스 할 때 **Hit율 증가**

- Page fault가 일어났을 때 처리과정 그림



Valid bit = 0(Page table에 없음)

Page fault를 처리는 알았고 Page fault 처리 후 instruction(명령어) 재실행 방법

- 예시 2가지

- 처리하기 쉬운 예 : 3주소 명령어 -> add A,B,C

과정 : 명령어(ADD) 인출 - A인출 - B인출 - 덧셈 - C저장

\*총 Read 3번, Write 1번 -> 총 4번 액세스

\***Page fault 발생시 처리 후 중지되었던 과정에서 재실행**

- 처리하기 어려운 예 : **Block move 명령어**(블록을 복사해서 옮기는 명령어)

-> Page **fault trap이 발생하면 메모리에 trap 발생 이전 메모리 내용을 다시 원상태로 복구하고 다시 시작**해야함 - But, **Overwrite 후에는 복원이 불가능**

요구 페이징의 **성능관계**

Page 부재율을 최대한 낮추는 것이 -> **Disk를 찾아가서 Disk액세스하는 횟수(시간)이 줄어들기 때문에 Page 부재율이 낮아야 좋다.** = 메모리 상에서 액세스할 Page Hit ratio가 높아야 좋다.

Page miss가 0.00025%일 때 성능이 10%정도 저하된다. = 캐시-메모리간 미스보다 **메모리-Disk간 미스가 성능에 훨씬 크게 작용**한다.

쓰기 시 복사

**Copy on Write**

- **fork()를 사용하여 프로세스를 생성**하면 자식 프로세스는 부모 프로세스의 Page를 공유하며 만들어진다. (**Page table 내용(여러 Page frame을 가르킴) 복사**= mapping이 같음, **Page는 복사 X)**

-> Page를 복사하지 않는 이유 : **exec()함수를 통해 자식 프로세스를 변경** 할 때 복사한 Page가 무효화되는 문제를 해결

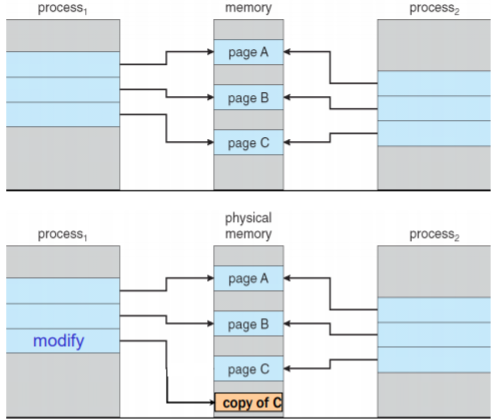
- **공유 페이지를 수정**하려면, 수정하려는 **페이지만 복사해서 다른 물리적 메모리에 적재한 뒤 수정(Write)**

- 수정된 페이지만 복사되므로 효율적 프로세스 생성

**Zero fill on demand**

- Page frame을 할당 할때 모든 내용을 0으로 채워서 이전내용을 지운 후 할당(깔끔)

- Stack 또는 Heap이 확장되거나, Copy on Write를 수행 할때 free page frame 할당시에 사용



page 변경

Page table

**\* 공유 Page 수정 시 Copy하여 새로운 물리적 메모리 공간을 할당 받아 적재하고 수정(Write)**

페이지 교체(OS가 관리)

- **free page frame가 없는 경우** = 메모리의 여유 공간이 없는 경우

-> 해결책 : 멀티 프로그래밍 정도를 낮춤 => 프로세스 종료

-> 또 다른 해결책 **: Page 교체** -> 잘 쓰지 않는 페이지를 Disk로 보내고 새 page를 받아옴

- free frame이 없으면 메모리의 사용 중이지 않은(실행중이거나 I/O대기 page frame은 불가) **victim page frame을 선택하여 새 page frame 과 Swap** => 요구 페이징의 기본

-논리 메모리와 물리적 메모리의 분리의 완성

**페이지 교체 알고리즘** : 희생 page frame을 선택하는 알고리즘

-> 가장 Page fault가 적게 일어나는 알고리즘 사용 - Page fault(Disk 액세스)가 일어나면 성능 저하 요인

페이지 교체 과정 **(DISK <-------> Memory)**

**1.** 요구한 Page가 메모리상에 없어서 Page fault trap handler를 통해 Disk로 감

**2.** Disk에서 필요한 Page 위치 찾기

**3**. 메모리의 free frame 찾기

a) **빈 공간 있으면 사용**

b) **빈 공간 없으면, page 교체 알고리즘을 사용**하여 Victim(희생) Page선택

-> **희생 page는 Swap out / 새 page는 Swap in**

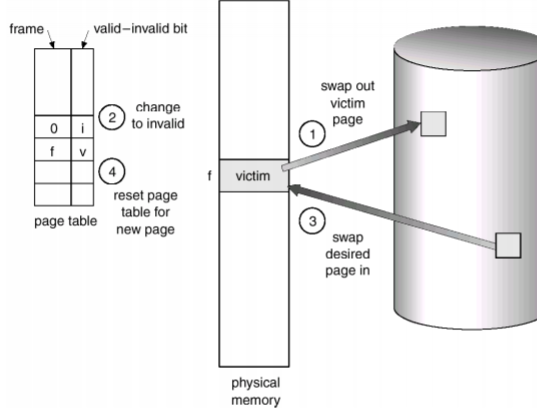
\*이 때 , Page table과 Page frame table의 수정(**Swap in/out** ***2번***)이 일어남

**4.** Page fault가 발생했던 지점부터 명령어 **재시작**

-빈 Page frame이 없다면 2번의 page 전송 필요(Swap in, Swap out)

-Modify bit(Dirty bit) : 변경된(수정된) 비트

-> **Dirty bit를 이용**하여 **페이지 수정을 표시**함으로써 페이지 **교체 시** Dirty bit를 확인하여 **수정된 페이지만 Disk로 쓰기 수행**(수정되지 않은 것(변경X)은 Disk로 다시 쓰기작업을 할 필요가 없음) = **Swap out<Memory -> Disk> 불필요**



\* 순서도 잘보기 : Swap out -> Page table 수정 -> Swap in -> Page table 수정

**요구 페이징에 필요한 알고리즘 : frame 할당 알고리즘, page 교체 알고리즘**

- **frame 할당 알고리즘은 각 프로세스마다 몇개의 page frame을 할당 할것인지 결정**

- **page** **교체 알고리즘은 어떤 Victim page를 선택할 것인지 + Page Swap out/in 수행**

**페이지 교체 알고리즘**

- 페이지 부재율이 낮은 것으로 선정

- Page 교체 알고리즘 5가지

1. **FIFO :** 가장 오래된 페이지 교체

2**. LRU :** 최근에 사용되지 않은(가장 옛날에 사용된)페이지 교체

3. **LRU approximation :** LRU방식이나 LRU는 정렬이 필요한데, 그 비용이 더 커서 오버헤드가 발 생하기 때문에 LRU 비슷한 방식(완전한 LRU는 아님)이지만 정렬이 필요없는 알고리즘

4. **Counting based** : LFU(least frequently used), MFU(Most frequently used)

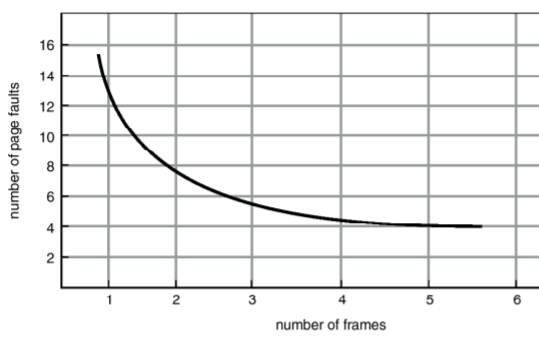
: Page table 마다 Counting 변수 필요

5. **Page buffering :** 메모리의 Page frame 여유(가용 frame pool유지)를 두고 사용(비상금 개념) 및 지연된 Swap out

\* **Page buffering은 위의 알고리즘과 Exclusive가 아닌 같이 사용됨**

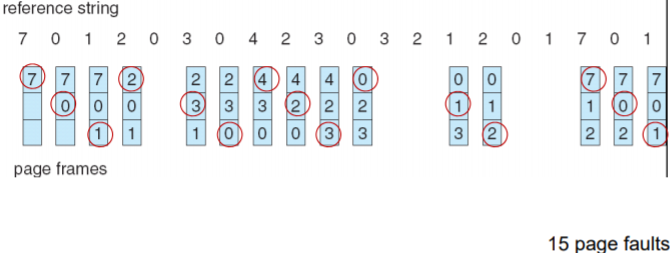
**\*컴구 캐시- 메모리랑 비슷하게 생각**

가용 frame 수(프로세스마다 할당해주는 page frame수)와 page fault의 관계



\*가용 frame 수가 늘어날수록 Page fault 가 줄어들지만 어느 순간부터는 유지된다.

FIFO 방식 교체 : 가장 오래된 Page frame 교체

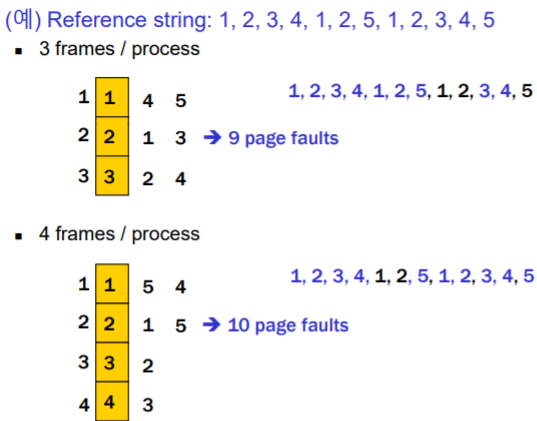


참조할 page frame번호

프로세스에 3개 page frame 제공

**FIFO 교체 알고리즘은 Belady 모순이 발생할 수 있다.**

- **Belady모순** : 프로세스당 더 많은 Page frame을 할당했지만 더 많은 Page fault가 발생하는 경우도 있음



**Optimal 알고리즘** : 가장 오랫동안 사용되지 않을 Page 교체

- 미래적 관점 : 실제로 구현 불가

- 알고리즘 비교 목적으로 사용

**LRU 알고리즘** : Least Recently Used

- 가장 덜 최근에 사용된(가장 옛날에 참조된) Page 교체

LRU 알고리즘 구현 <- **H/W 의 지원 필요**

**- Counter :** Page table entry 에 Counter를 추가

- 참조마다 Counter 증가

- 페이지 참조마다 Clock을 Counter로 복사

- Counter값이 최소(최근에 가장 안쓰임)인 Page 교체

**- Stack** : 페이지 번호 저장용 Stack 유지

- 페이지 참조마다 참조된 페이지 번호를 Stack top으로 이동

- Stack bottom(최근에 가장 안쓰임)의 Page를 교체

->> 실제로 LRU알고리즘을 사용하면 **교체할 Page 선택을 위한 알고리즘을 수행하는데 시간이 더 걸림** = 오버헤드 , 따라서 **LRU근사(approximation) 알고리즘 사용**

**LRU approximation(근사) 알고리즘**

- 참조비트

- 각 page entry에 reference bit 추가 및 '0'으로 초기화

- Page가 참조되면 '1'로 변경

- Page 사용 순서는 모름 But, 사용되지 않은 page는 알 수 있음

- 참조되지 않은 비트 중에 Random으로 선택해서 교체

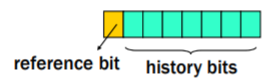
- 부가적 참조비트

- Page entry에 참조 비트가 아닌 참조 바이트를 추가 함으로써 가장 왼쪽 1비트를 참조비트,

오른쪽 7비트를 History 비트로 정의

- 일정 시간마다 참조바이트(8비트)를 Shift Right -> 8비트 중에서 가장 값이 작은 비트가 교

체 대상 = 가장 오랫동안 쓰이지 않은 Page

 \*Ex. 00000001 < 01101011 < 10000000 <왼쪽꺼 먼저 교체>

- 2차 기회 알고리즘 = Clock 교체 알고리즘 이라고도 함

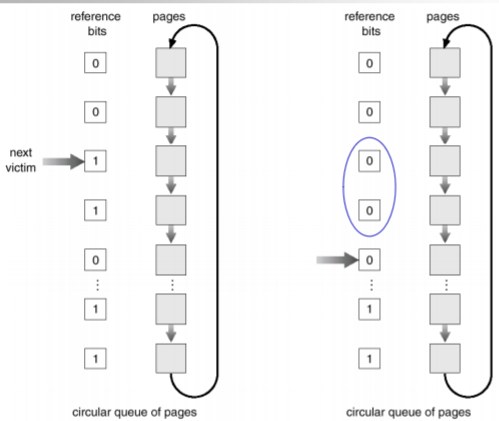
- 참조비트만 사용

- Page들을 Circular queue로 구현

- 교체될 Page 탐색

- 참조 비트가 1이면 -> 0으로 바꿈(기회를 한번 주는 격)

- 참조 비트가 0이면 -> 해당 페이지 교체(Victom page)

 - 2차 기회 알고리즘(LRU 근사)

1->0 으로 바뀜->

현재 rotation 탐색 위치

희생 page->

- 개선된 LRU근사 알고리즘

- **참조비트 + Modify(Dirty)비트** <- 수정된 비트 표시 사용

- 4가지 조합 : (0,0 < 0,1 < 1,0 < 1,1) 0,0이 가장 먼저 교체대상 페이지

- 현실적으로 사용하는 방식

\*Modify bit가 1이면 이전에 참조 및 수정된 적이 있다는 뜻

\*Circular queue를 사용해서 reference(참조)비트는 최근 사용 비트를 나타내는 것?

Page buffering 알고리즘 : 가용 page frame pool을 유지하고 먼저 free frame으로 읽어와서 Idle 시점에 희생 Page를 디스크로 Swap out(지연된 Swap out)

수정된 Page buffering 알고리즘 : Modify bit를 이용하여 수정된 Page만 Idle시점에 Disk에 저장

수정되지 않은 Victim Page를 저장할 필요X (0,0 1,0)

**프로세스에게 Page frame 수 할당**

- 프로세스에게 할당하는 Page frame의 수

**- 최소 frame : 아키텍처에 의해 결정**

**- 최대 frame : 가용 물리 메모리 크기에 의해 결정**

\* Minimum보다는 같거나 크게 할당 되어야 함

Ex. IBM370의 SS MOVE 명령어는 6Page이상 필요

할당 알고리즘

**-Fixed allocation**(고정된 갯수(크기)의 frame 할당)

-**균등할당** : **모든 프로세스가 균등**하게 할당(100page,4프로세스면 25page씩 할당)

-**비례할당** : 프로세스 **특성(크기)에 따라 할당**

-EX. 메모리 가용 page frame 공간이 100이고 P1이 150, P2가 50 크기일 때

100/(150+50) = 0.5 --> P1 = 150 x 0.5=75 P2 = 50 x 0.5=25

**-Priority allocation**(우선순위 할당)

- 프로세스 크기 대신 우선순위 또는 크기 및 우선순위 조합을 사용하여 할당

Page **교체** 시 방식 **2가지**

- **전역 교체** : **모든 Frame 집합에서 교체할 Page frame 선택**, 타 프로세스로부터 frame 가져올

수 있다, 사용중이 아닌 Page frame 교체 (할당 받는 Page frame 크기가 가변적)

- **지역 교체** : **각 프로세스는 자신에게 할당된 frame 중에서 교체할 Page frame 선택**

\*전역 교체(Global replacement)가 단위 시간당 처리속도가 더 좋아서 더 많이 사용

**쓰레싱(Thrashing)**

- 프로세스가 충분한 Page를 할당 받지 못하면 **Page fault가 너무 자주 발생**하여 Disk를 계속 액세스해야 하는 **I/O 인터럽트 및 작업이 많이 일어나서 문제**발생

1. 프로세스가 **Swap out/Swap in에 시간 소비**(오버헤드) = I/O burst 시간이 길어짐

2. **CPU 이용률이 낮아짐**

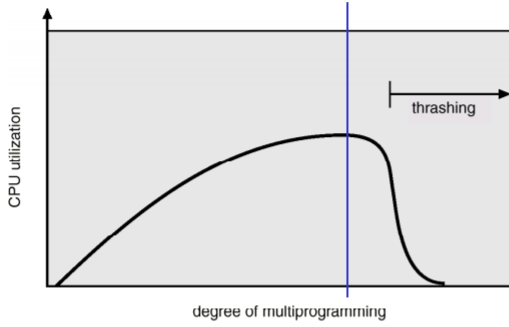
3. **OS**는 CPU이용률이 낮아서 **멀티프로그래밍 정도를 증가**시키도록 판단

4. **다른 프로세스**가 시스템(메모리)에 **추가**

5. 메모리가 부족해서 Page frame을 별로 할당 받지 못해서 일어난 문제인데 새 프로세스가 추가됨으로써 **Page fault가 더 자주 발생**

6. CPU이용률 더욱 저하 ---------------> ***악순환***

Thrashing : 빈번한 page교체로 프로세스가 반복적으로 Swap in/out 하느라 바쁜 상황



**지역성 모델** : Thrashing을 방지하기 위한 모델

- 각 프로세스에게 일정량 필요한 만큼의 frame을 제공하도록 해야함

**지역성 : 집중적으로 함께 사용되는 페이지들의 집합**

프로그램은 몇 개의 다른 지역성(몇개의 집중적으로 사용되는 부분)들로 구성됨

프로세스가 실행됨에 따라 지역성은 이주

지역성은 중첩될 수 있음

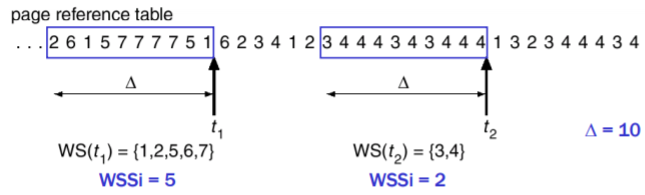
Thrashing 발생 **원인**

- **집중적으로 사용되는 페이지들의 집합** > **할당된 메모리(Page frame수) 사이즈** : Thrashing 발생

- 함께 집중적으로 사용되는 페이지를 모두 메모리에 적재할 수 없을 때 발생

**Working set 모델** : 고정된 횟수의 페이지 참조(고정된 수의 페이지 참조)

- 일정시간 간격으로 페이지 참조



\*WSSi는 서로 다른 참조 페이지 갯수(범위의 페이지 참조에 들어있는)

**Working Set model**

- 전체 frame요구량 D>M(가용 page frame) 일 때 발생

- M를 늘려야 하므로 한 프로세스를 중지하여 M크기를 늘린다.

Working set 추적방법 : 고정 시간간격 타이머와 2비트 참조 비트 사용

**Page fault Frequency(PFF) Scheme**

**Thrashing 제어**방식

- 어떤 프로세스를 기준으로 Page fault rate를 **상대적**으로 참고하여 Page frame의 수 조절

**- Page fault rate가 너무 낮을 때 -> page frame 수 감소시킴**

**- Page fault rate가 너무 높을 때(할당받은 frame 수 부족) -> page frame 수 증가시킴**

\* page fault rate가 너무 낮아도 메모리를 낭비하고 있는 것이므로 frame수를 줄임

\* 가장 많이 사용되는 방식

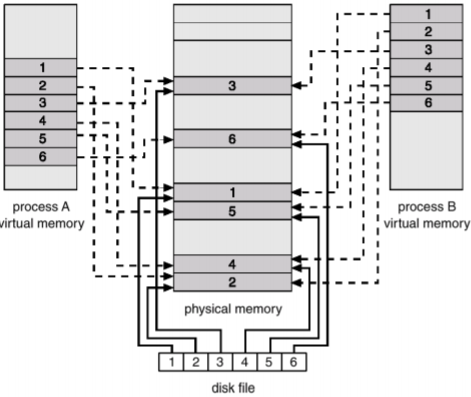
Memory Mapped File

- 디스크 블록을 메모리에 있는 Page에 맵핑

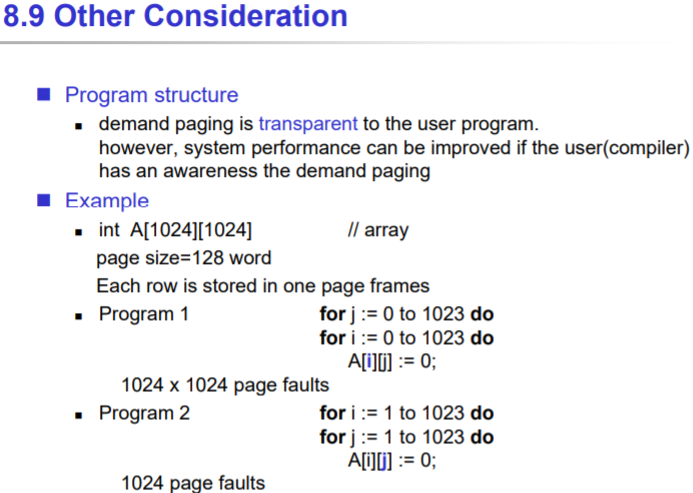
- mmap() system call 사용(in UNIX)

-> 여러 프로세스가 페이지를 공유하여서 같은 파일에 맵핑 가능

**Disk가 가상주소 공간으로 맵핑되고 실제로 프로그램이 실행되서 사용되려면 물리적인 메모리로 복사가 필요함(Read : Disk(가상 주소공간) -> 물리적Memory || Write : 물리적M -> Disk)**



\* Disk는 가상 주소 공간에 프로세스를 Mapping하고 물리적 메모리로 적재하고 프로세스의 가상 논리 주소를 물리적 메모리로 복사(프로세스마다 서로 다른 논리 주소가 같은 물리적 메모리 주소를 공유할 수 있음)한다.



1Page는 세로방향으로 1라인 액세스하여 Page frame에 적재하는데, 열우선(P1)의 경우 j가 매번 바뀌므로 매번 Page fault 발생

C언어는 행우선 액세스

**할당되는 page frame이 1024보다 작을 때**

행 i

열 j

- 프로그래머는 자신이 짜는 프로그램이 모두 메모리의 연속적인 공간에 적재된다고 생각하고 프로그램을 짠다. -> 프로그래머 입장에서는 페이지 단위로 분할되어 적재되는 것을 알 수 없다.

-> 따라서 **요구 페이징은 '투명하다'(Transparent) 라고 한다**.(프로그래머에게 보이지 않음)

-> 시스템의 동작 원리를 잘 알고 있을 수록 메모리를 좀더 효율적으로 사용하도록 프로그램을 설계할 수 있다.

\***1024\*1024** 배열 => **1K\*1K = 1M개**의 배열 ==> **1M**개 배열 x Word(32bit=**4byte**) = 4MByte

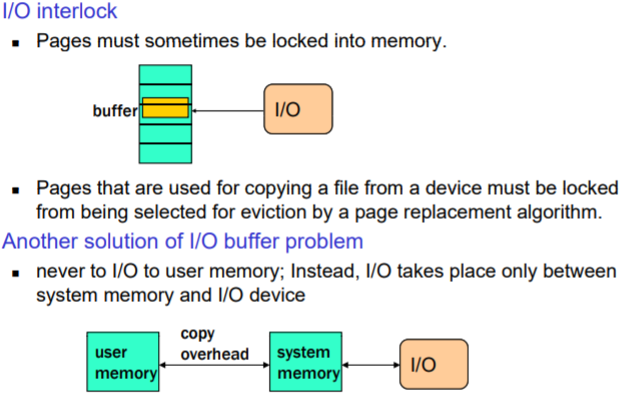
\***1페이지 크기는 4KB**이다. 따라서 **1페이지당 배열 1라인을 적재**할 수 있다.

\*i는 행(Row) / j는 열(Column)을 가르킴

**\*program 1(열우선)은 가로방향-->으로 먼저 액세스 / program 2(행우선)는 으로 먼저 액세스**

\*최초 액세스 시에는 1~1024까지는 모두 Disk에 저장되어 있으므로 Page fault가 모두 발생(1024)

\*Page fault 횟수가 차이가 많이 남 -> 성능에 영향



\*페이지는 가끔 메모리에 고정적으로 적재되어 있어야 한다.(Ex. I/O 대기중)

= **교체 대상에서 제외**

\*메모리와 I/O사이에 시스템 메모리를 두어 **안전하게 전송**

= but, **오버헤드 발생**

8장끝