**Chapter 9: Virtual Memory**

**9-1 배경**

프로그램에는 잘 발생하지 않는 오류 상황을 처리하는 코드가 종종 존재하는 데, 이러한 오류들은 실질적으로 거의 발생하지 않으므로, 이 코드들은 거의 실행되지 않음.

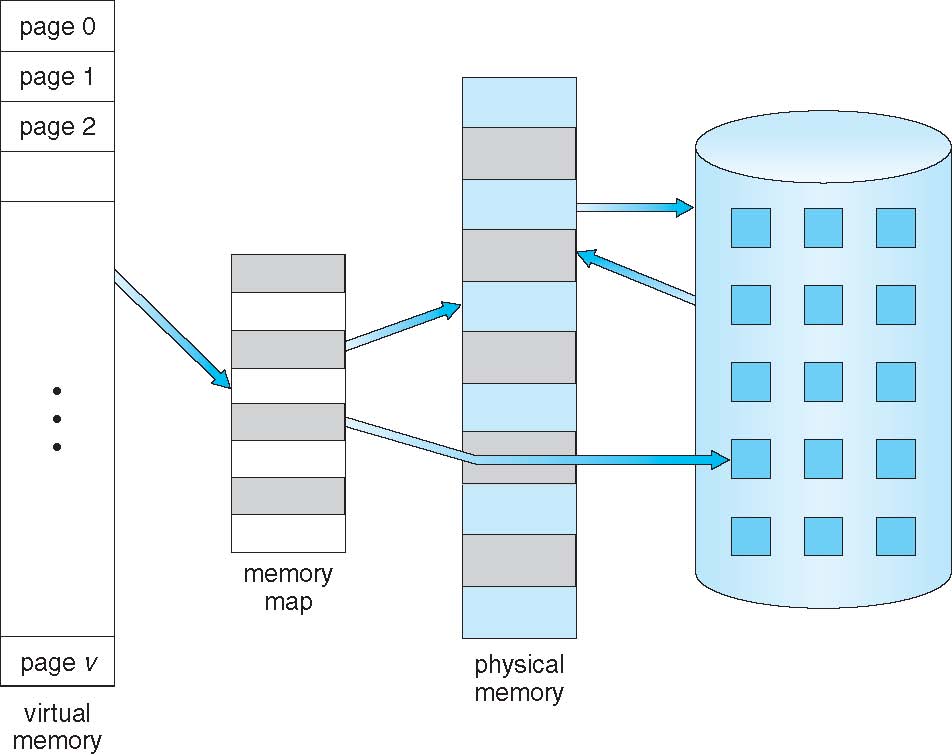
배열, 리스트, 테이블 등은 필요 이상으로 많은 공간을 점유하는 수가 있음 => 예를 들어 프로그램에서 사용하는 심볼의 수가 200개 정도임에도 불구하고 3000개까지의 심볼을 수용하도록 기억 공간을 할당받았을 수 있음

프로그램내의 어떤 옵션이나 기능들은 거의 사용되지 않음

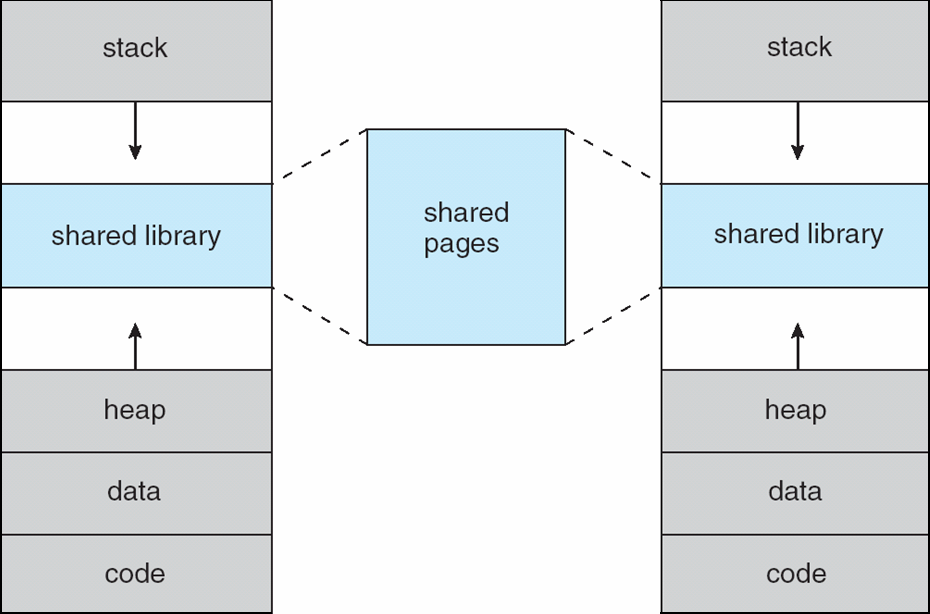
* 그래서, 부분적으로 메모리에 로드하여 실행할 수 있는 기능을 고려
* 프로그램 일부분만 메모리에 올려놓고 실행할 수 있다면 다음과 같은 많은 이점이 있음

1. 물리 메모리 크기에 의해 더 이상 제약받지 않게 됨
2. 각 사용자 프로그램이 더 작은 메모리를 차지하므로 더 많은 프로그램을 동시에 수행할 수 있게 됨.
3. 프로그램을 메모리에 올리고 스왑하는데 필요한 입출력 횟수가 줄어들기 때문에 프로그램들이 보다 빨리 실행됨

가상 메모리 – 실제의 물리 메모리 개념과 사용자의 논리 메모리 개념(가상 메모리)을 분리한 것

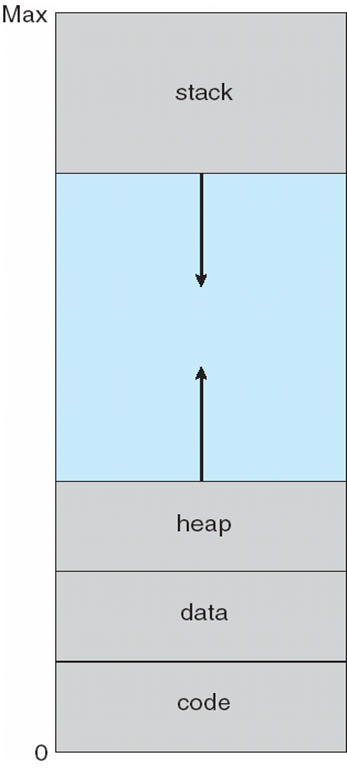


* 작은 메모리를 가지고도 얼마든지 큰 가상 주소 공간을 프로그래머에게 제공할 수 있는 점이 장점
* 또한, 가상 메모리는 수요 페이징이나 수요 분할에 의해 구현될 수 있음
  + 수요 페이징 : 메모리 관리 메커니즘(Memory management unit 메커니즘)을 사용해서 여러 프로세스가 시스템의 메모리를 효율적으로 공유할 수 있도록 하는 기술
  + 수요 분할 : 하나의 세그먼트가 다시 여러 페이지로 나뉘는 페이지된 세그먼트 기법
* 가상 메모리는 페이지 공유를 통해 파일이나 메모리가 둘 또는 그 이상의 프로세스들에 의해 공유되는 것을 가능케 함
  + 다음과 같은 장점이 존재
  + 시스템 라이브러리가 여러 프로세스들에게 공유될 수 있음
  + 프로세스들이 메모리를 공유할 수 있음
  + 페이지는 fork() 시스템 호출을 통한 프로세스 생성 과정 중에 공유될 수 있기 때문에 프로세스 생성 속도를 높일 수 있다.



가장 주소 공간 – 프로세스가 메모리에 저장되는 논리적인 모습

* 보통 주소 0에서 시작, 공간의 끝까지 연속 주소
* Stack과 heap사이의 사용되지 않은 공간 즉, 공백을 포함하는 가상 주소 공간을 성긴(sparse) 주소 공간이라고 함
* 성긴 주소 공간은 힙 또는 스택이 지정된 새 페이지로 각각 확장 될 때까지 필요한 실제 메모리가 없음



**9-2 Demand Paging**

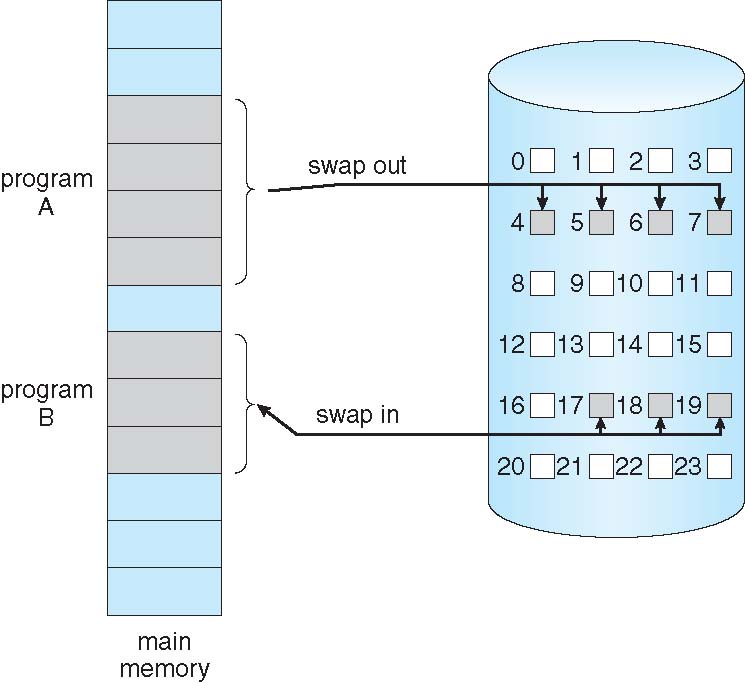
필요한 프로그램만 메모리에 적재 하는 방법

장점 => 적은 I / O 필요, 불필요한 I / O 없음

적은 메모리 필요

빠른 응답

스와핑을 사용하는 페이징 시스템과 유사



페이지가 필요하면 참조를 수행하는데

* 잘못된 참조의 경우 -> 중단
* 메모리에 부재시 -> 메모리로 가져다 줌

또한, 프로세스를 실행하고 싶으면 메모리로 읽어 들이는데(Swap in) 이 때 전체 프로세스를 읽어오지 않고 게으른 스왑퍼(lazy swapper)를 사용한다.

게으른 스왑퍼(lazy swapper) : 페이지가 필요하지 않는 한 그 페이지를 메모리에 적재하지 않는 것

* 참고로, 요구 페이징은 프로세스 내의 개별 페이지를 관리하는 것이므로 Swapper보다는 Pager라는 용어가 더 적절함.

**9-2-1 기본 개념(Basic Concepts)**

Swapping시 Pager는 프로세스가 다시 swap out 되기 전에 실제로 사용될 페이지들이 어떤 것인지 추측함.

대신, Pager는 실제 필요한 페이지들만 메모리로 읽어 옴

* 사용되지 않을 페이지를 메모리로 가져오지 않음으로써 시간 낭비와 메모리 공간 낭비를 줄일 수 있음

이를 위해선 어느 페이지가 메모리에 올라와 있는지 구별할 수 있어야 함

* 따라서, 유효.무효 비트(Valid-Invalid Bit) 기법을 사용함

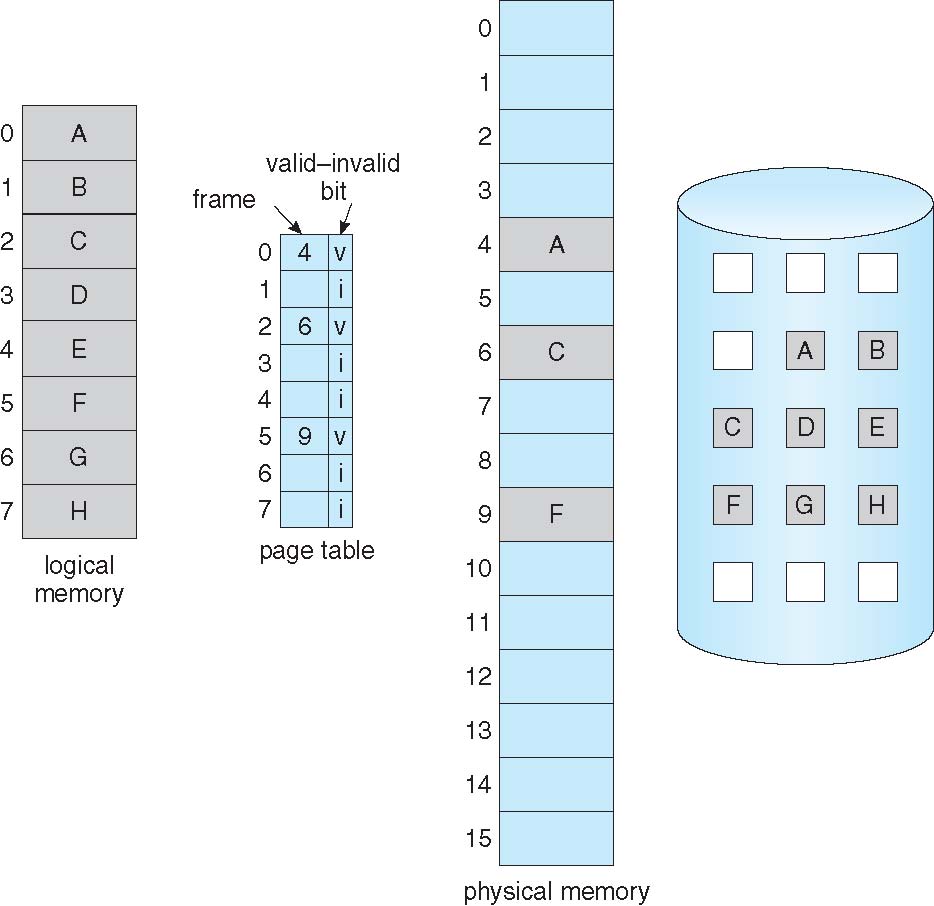
Valid Bit(유효 비트) – 해당 페이지가 메모리에 있다는 의미

* V : in-memory – **memory resident**

Invalid Bit(무효 비트) – 해당 페이지가 유효하지 않거나 ( 가상 주소 공간상에 정의되지 않았거나 ) 유효하지만 디스크에만 존재한다는 것을 의미

* I : not-in-memory

처음에는 모든 항목에서 Valid-Invalid Bit가 I로 설정 됨.



Valid-Invalid Bit 사용 예시

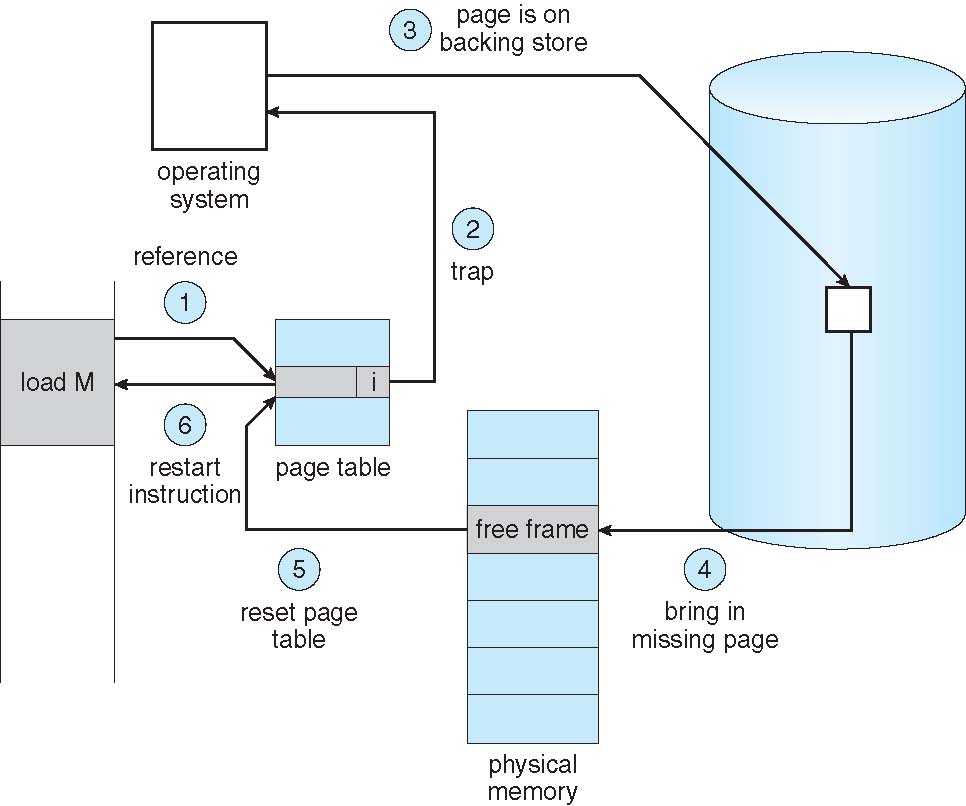
프로세스가 메모리에 올라와 있지 않는 페이지를 접근하려고 하면 어떠한 일이 발생할까? => 페이지 테이블 항목이 무효로 설정되어 있으므로 page-fault trap(페이지 부재 트랩)을 발생 시킴

Page-fault(페이지 부재)를 처리하는 과정

1. 프로세스에 대한 내부 테이블을 검사하여 그 메모리 참조가 Valid인지 Invalid인지를 알아냄
2. Invalid에 대한 참조 => 프로세스 중단

Valid에 대한 참조 (메모리에만 부재인 경우) => 디스크로부터 가져와 메모리에 올림

1. Free Frame을 찾음 (Valid 인 경우)
2. 디스크에 새롭게 할당된 Frame으로 해당 페이지를 읽어 들이도록 요청
3. 메모리에 페이지가 표시되도록 테이블 재설정, 유효 비트 설정 = V
4. 트랩에 의해 중단되었던 명령어를 다시 수행. 앞으로 그 페이지가 항상 메모리에 있었던 것처럼 해당 페이지에 접근할 수 있음



Page-fault 처리 과정

Demand Paging 측면

극단적인 경우 – 메모리가 페이지에 하나도 안 올라와 있는 상태에서 프로세스 실행

운영체제에서 명령 포인터의 값을 프로세스의 첫 명령으로 설정하는 순간, 이 명령이 메모리에 존재하지 않는 페이지에 있으므로 페이지 부재 발생. 페이지가 적재되고 나면 프로세스는 수행을 계속하는데 프로세스가 사용하는 모든 페이지가 메모리에 올라올 때까지 필요할 때마다 페이지 부재가 발생한다. 필요한 모든 페이지가 적재되고 나면 더 이상 부재 오류가 발생하지는 않음. 이것이 순수 요구 페이징(pure demand paging). 즉, 어떤 페이지가 필요해지기 전에는 결코 그 페이지를 메모리로 적재하지 않는 방법

프로그램은 한 명령어에 대해서도 여러 개의 페이지 부재를 일으킬 수 있음

* 명령문을 읽기 위해서 한 페이지, 데이터마다 한 페이지씩
* 하지만, 참조의 지역성(Locality of reference)라는 성질이 있기 때문에 Demand Paging은 만족할 만한 성능을 보임

Demang Paging에 필요한 하드웨어

1. 유효/무효 비트가 있는 페이지 테이블
2. 보조 기억 장치(secondary memory) : 주 메모리에 없는 모든 페이지들을 가지고 있음

- 스왑 장치라고도 하며, 이 목적을 위해 사용하는 디스크 영역을 스왑 공간(Swap Space)라고 함

3. 페이지 부재 오류 처리 후에 명령어 처리를 다시 시작할 수 있어야 함

**9.2.2 요구 페이징의 성능(Performance of Demand Paging)**

실질 접근 시간(effective access time)을 계산하기 위함

페이지 부재 확률 = p라 가정

그에 대한 범위는 0 <= p <= 1

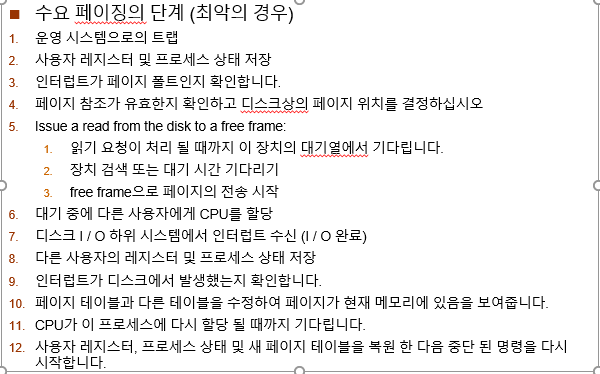
P = 0 => 페이지 부재(page fault)가 없음

P = 1 => 모든 참조가 페이지 부재(page fault)

실질 접근 시간(effective access time) = ( 1 – p ) \* memory access + p \* 페이지 부재 시간

EAT를 위해 페이지 부재를 처리하는 데 얼마나 많은 시간이 걸리는지 알아야 함

우선, 페이지 부재는 아래와 같은 순서로 처리(참고만 할 것)



또한, 페이지 부재를 처리하는 시간은 다음 3개의 큰 구성 요소로 이루어져 있음

1. 인터럽트의 처리
2. 페이지 읽기
3. 프로세스 재시작

EAT(실질 접근 시간) 예시

평균 페이지 부재 처리 시간 = 8 밀리초 => 나노초로 바꾸면 8000000 나노초

메모리 접근시간 = 200 나노초

EAT = (1 – p) \* (200) + p \* (8000000)

= 200 + 7,999,800 \* p

1,000 개 중 하나의 액세스로 인해 페이지 오류가 발생하면

P = 1 / 1000 => 대입시 EAT = 약, 8200 나노초

이는 8.2 마이크로초와 같음

EAT = 8.2 microseconds.

메모리 접근 시간인 200나노초는 0.2마이크로초 즉, demand Paging 때문에 40배나 느려짐

만약, 성능저하를 10% 이내로 낮추고 싶다면 메모리 접근 시간에서 10%만큼을 더한 220 > 200 + 7,990,800 \* p

P > 0.0000025

즉, 약 400000번 중에 1번 이하의 페이지 부재가 발생해야 가능함

Demand Paging 최적화

Demand paging의 또 다른 특성 중 하나는 Swap space 관리

* 파일 시스템 I/O보다 스왑 공간에서의 디스크 입출력 훨 빠름
* 파일 시스템보다 더 큰 블록을 사용하기 때문

**9.3 쓰기 시 복사(Copy-on-write)**

**COW (Copy-On-Write)**는 부모 프로세스와 자식 프로세스가 모두 메모리의 동일한 페이지를 처음에 공유 할 수 있게 하는 것

이때 공유되는 페이지를 쓰기 시 복사(Copy-On-Write) 페이지라고 표시

COW를 사용하면 수정 된 페이지 만 복사되므로 보다 효율적인 프로세스 생성이 가능

운영체제는 보통 zero-fill-on-demand기법으로 빈 페이지 집합(pool)의 페이지를 할당받음

즉, 페이지를 할당할 때 그 내용을 다 0으로 채워 이전 내용을 다 지움

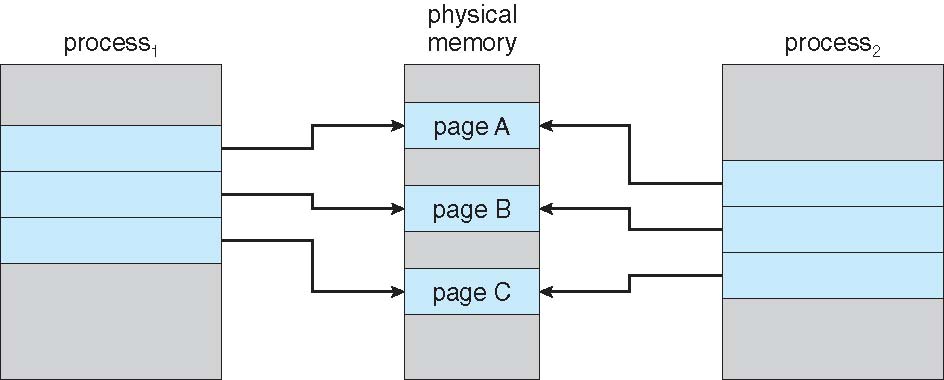
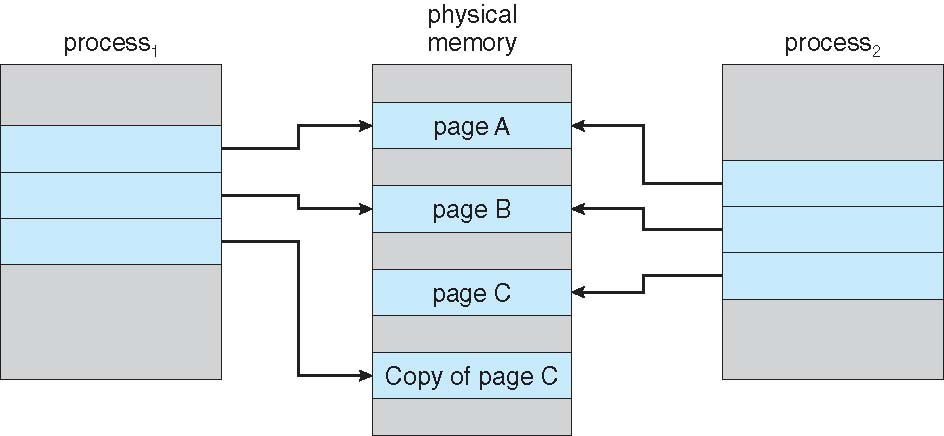
Solaris와 Linux를 포함한 몇 가지 UNIX 변종들은 vfork()(virtual memory fork)라는 시스템 호출을 제공하고 있음

vfork() 시 부모 프로세스는 보류되고 자식이 부모의 주소공간을 사용하게 됨

vfork()는 자식이 만들어지자마자 exec()를 하는 경우를 위해 마련된 것

또한, 페이지가 전혀 복사되지 않으므로 매우 효율적이다.

**프로세스 1이 페이지 C를 수정하기 전 프로세스 1이 페이지 C를 수정 한 후**

**9.4 페이지 교체(Page Replacement)**

페이지 교체를 포함하도록 페이지 오류 서비스 루틴을 수정하여 메모리 초과 할당 방지

페이지 전송의 오버 헤드를 줄이기 위해 수정 (더티) 비트 사용 - 수정 된 페이지 만 디스크에 기록됨

페이지를 교체하면 논리 메모리와 실제 메모리 간의 분리가 완료됩니다. - 큰 가상 메모리는 더 작은 실제 메모리에 제공 될 수 있습니다

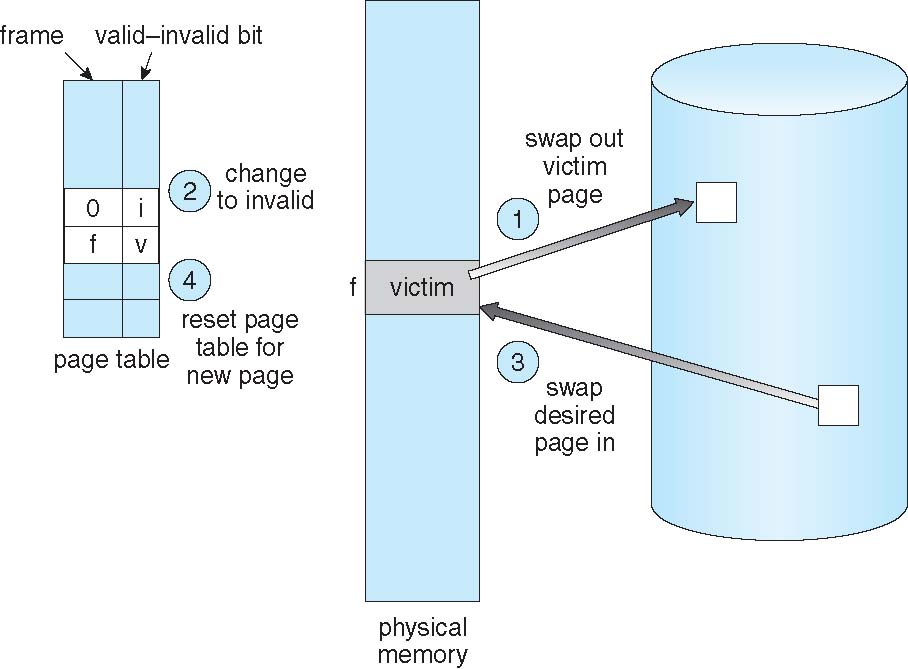
**9.4.1 기본적인 페이지 교체(Page Replacement)**

교체 과정

1. 디스크에서 필요한 페이지의 위치를 ​​찾음
2. Free Frame(빈 페이지 프레임)을 찾음 :  
    - 빈 페이지 프레임이 있다면 그것을 사용  
    - 없다면 희생될(victim) 프레임을 선정하기 위하여 페이지 교체 알고리즘을 가동 시킴

**-** Dirty인 경우 희생될 페이지를 디스크에 기록하고, 관련 테이블을 수정한다.

1. 빼앗은 프레임에 새 페이지를 읽어오고 테이블을 수정함
2. 페이지 부재가 발생한 지점에서부터 사용자 프로세스를 계속한다.



페이지 교체 과정 1-2-3-4

변경 비트(modify bit 또는 dirty bit) : CPU가 페이지 내의 어떤 바이트라도 쓰게 되면 페이지가 변경되었음을 나타내기 위해 설정되는 비트

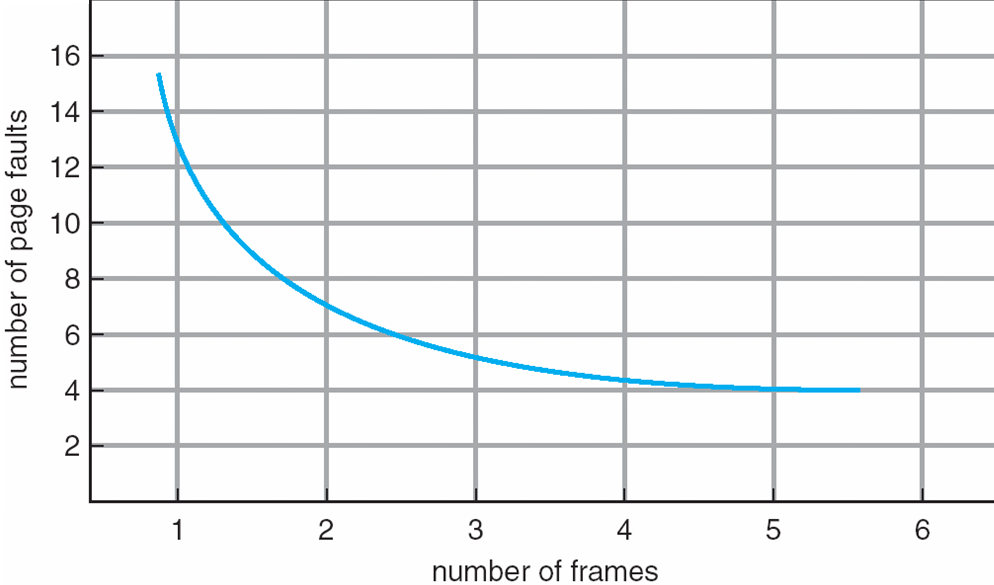
프레임 할당(frame-allocation) 알고리즘 : 여러 프로세스가 존재할 때 각 프로세스에 얼마나 많은 프레임을 할당해야 할지 결정하기 위한 알고리즘

페이지 교체(page-replacement) 알고리즘 : 페이지 교체가 필요할 때마다 어떤 페이지를 교체해야 할지 결정하기 위한 알고리즘

알고리즘을 선택할 때 일반적으로 페이지 부재율(page-fault rate)이 가장 낮은 것을 선정

참조열(reference string) : 페이지 교체 알고리즘의 성능을 평가할 때 특정 메모리 참조 나열을 하는데 이러한 메모리 주소 나열들을 말함

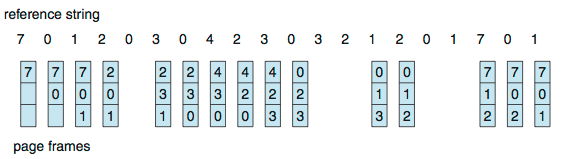
**페이지 부재 횟수 VS 프레임 수**



**9.4.2 FIFO 페이지 교체**

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1라 가정

프레임 3개(프로세스마다 한번에 3페이지씩 메모리에 저장 가능)

****

1. age faults (새로운 숫자를 참조할때마다 페이지 부재 한번 씩 발생)

Belady’s Anomaly(Belady의 모순) : 프로세스에게 프레임을 더 주었는데 오히려 페이지 부재율이 더 증가하는 현상

예시 : 1,2,3,4,1,2,5,1,2,3,4,5 <= 이 경우 모순이 일어남 레알 일어남

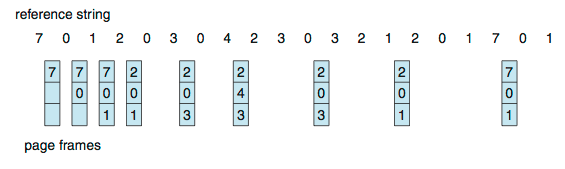
9_13.pdf

결론 : 많은 프레임을 할당하면 성능이 좋아질 것으로 생각하였지만 Belady의 모순을 통해 이 가정이 항상 옳은 것이 아님을 발견

**9.4.3 최적 페이지 교체(Optimal Page Replacement // OPT라고도 부름)**

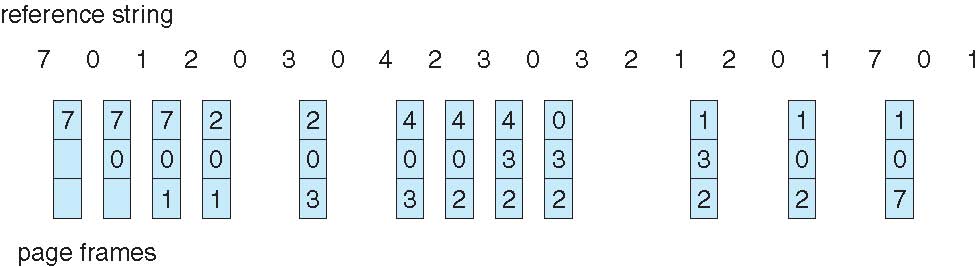
앞으로 가장 오랫 동안 사용되지 않을 페이지를 찾아 교체하는 것

* 이 알고리즘은 할당된 프레임 수가 고정된 경우 가장 낮은 페이지 부재율을 보장
* 하지만 미래를 알 수 없기에 구현은 어려움



**9.4.4 LRU 페이지 교체(LRU Page Replacement)**

Least-recently-used(LRU) 알고리즘 : 페이지 교체 시 가장 오랫 동안 사용되지 않은 페이지를 선택

****

12 faults – better than FIFO but worse than OPT

LRU 알고리즘 구현 방법

계수기(counters) : 페이지가 참조될 때마다 시간을 카운터에 복사

매 메모리 접근시마다 시간은 증가하므로 시간이 가장 작은 페이지가 교체 됨

스택(STACK) : 페이지가 참조 될 때마다 페이지 번호는 스택 중간에서 제거되어 스택 꼭대기에 놓이게 됨

따라서, 맨 위는 항상 가장 최근에 사용된 페이지이고, 밑바닥은 가장 오랫동안 이용되지 않은 페이지이다.

LRU와 OPT는 Belady’s Anomaly가 없는 스택 알고리즘 사례

스택 알고리즘(Stack algorithm) : Belday’s Anomaly가 나타나지 않는 알고리즘

9_16.pdf

**9.4.4 LRU 근사 페이지 교체(LRU Approximation Page Replacement)**

Reference bit(참조 비트)

처음에 모든 참조 비트는 운영체제에 의해 0으로 채워짐

프로세스가 실행되면서 참조되는 페이지의 비트는 하드웨어가 1로 세트

얼마가 지나면 페이지 사용의 순서는 모르지만 어떤 페이지가 사용되었고 한번도 사용되지 않았는지에 대해 알 수 있음

**9.4.5 2차 기회 알고리즘(Second-Chance Algorithm)**

기본 FIFO 교체 알고리즘 + 페이지가 선택될 때마다 참조 비트를 확인

참초 비트가 0 => 페이지 교체

1 => 다시 한 번 기회를 주고 다음 FIFO 페이지로 넘어감

( 기회를 받게 되면 참조 비트는 해제되고 도착 시간이 현재 시간으로 재설정 => 이에 따라, 그 페이지는 다른 모든 페이지들이 교체되거나 기회를 받을 때 까지 교체되지 않음 )

9_17.pdf

**9.4.5 개선된 2차 기회 알고리즘(Enhanced Second-Chance Algorithm)**

두 개의 비트를 조합하여 사용하면 다음 네 가지 등급이 가능

* Take ordered pair (참조, 수정)

1. (0,0)최근에 사용되지 않았으며 수정되지도 않음 – 교체하기 가장 좋은 페이지
2. (0,1)최근에 사용되지 않았지만 수정됨– 교체하려면 디스크에 내용을 기록해야 하기 때문에 교체 하기에 좋지 않음
3. (1,0)최근 사용하였으나 수정은 되지 않은 경우 – 이 페이지는 곧 다시 사용될 가능성이 높음
4. (1, 1) 최근 사용되었으며 수정됨 – 곧 다시 사용될 예정이며 교체 전에 디스크에 내용을 먼저 기록해야 함

**9.4.6 계수-기반 페이지 교체(Counting-Based Page Replacement)**

Least Frequently Used (LFU) Algorithm : 참조 회수가 가장 작은 페이지를 교체하는 방법

Most Frequently Used (MFU) Algorithm: 가장 작은 참조 회수를 가진 페이지가 가장 최근 참조된 것이고 앞으로 사용될 것이라는 판단에 근거한 것

둘 다 일반적으로 잘 쓰이진 않음

**9.4.7 페이지-버퍼링 알고리즘(Page-Buffering Replacement)**

1. 시스템들이 가용 프레임(Free Freme) 여러 개를 풀(pool)로 갖고 있다가, 페이지 부재가 발생하면 교체될 페이지를 찾지만, 교체될 페이지의 내용을 디스크에 기록하기 전에 가용 프레임(Free Frame)에 새로운 페이지를 먼저 읽어 들이는 방법.

* 이 개념의 확장으로 변경된 페이지 리스트를 유지하는 방법이 있다. 디스크가 아무런 일도 없게 되면 그 때마다 변경된 페이지들을 차례로 디스크에 쓴 후에, 페이지의 변경 비트를 0으로 되돌려 놓는다.(reset)

2. Free Frame Pool을 유지하지만 그 Pool속 각 프레임의 원래 임자 페이지가 누구였었는지를 기억해 놓는 것

**9.4.8 응용(applications)과 페이지 교체**

몇몇 경우에는 운영체제의 가상 메모리를 통해 데이터를 접근하는 응용이 운영체제가 전혀 버퍼링 기능을 제공하지 않는 경우에 비해 오히려 안 좋은 성능을 보일 때가 존재

(EX 데이터베이스 )

몇몇 운영체제는 특별한 프로그램들에는 디스크 파티션을 파일 시스템 구조가 아닌 단순한 논리적인 블록들의 순차적인 배열로서 사용할 수 있게 해주는 기능을 갖추고 있음

이 배열은 종종 raw disk라고 불리며 여기에 대한 입출력은 raw입출력이라는 용어를 사용

Raw 입출력은 모든 파일 시스템 서비스를 거치지 않음

**9.5 프레임의 할당(Allocation of Frames)**

각 프로세스에는 최소 프레임 수가 필요

**9.5.1 최소로 할당해야 할 프레임의 수(Minimum Number of Frames)**

프로세스당 필요한 최소 프레임 수는 명령어 집합 아키텍처(instruction set architecture)에 의해 결정됨

EX) 저장 공간으로부터 저장 공간으로의 이동 (IBM 370의 MVC 명령어)

저장 공간으로부터 저장 공간으로의 명령이기에 6바이트를 차지, 2페이지에 걸칠 수 있음

이동하려는 문자 블록과 그것을 저장할 공간 역시 2페이지(각각)에 걸칠 수 있음

* 이런 상황은 6프레임을 필요로 함

두 개의 주요 할당 체계가 존재

고정 할당(Fixed Allocation)

우선 할당(priority allocation)

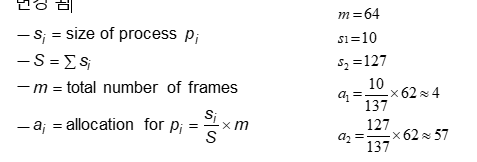
**9.5.2 할당 알고리즘(Allocation Algorithms)**

균등 할당 : 모든 프로세스에게 똑같이 프레임을 할당 하는 것

93개의 Frame과 5개의 Process가 있는 경우 // 각 프로세스는 18개씩 프레임을 받음

나머지 3개를 가용 프레임 버퍼 저장소(free frame buffer pool)로 사용

비례 할당 방식(proportional allocation) : 프로세스 크기에 따라 할당



m = 62개 <= 잘못 나옴

우선 할당(priority allocation)

크기 대신 우선 순위를 사용하는 proportional allocation scheme(비례 할당 체계) 사용

프로세스 P에서 페이지 부재 발생 경우,

프레임 중 하나를 교체하기 위해 선택

우선 순위 번호가 낮은 프로세스에서 프레임 교체를 선택

**9.5.3 전역 대 지역 할당(Global Versus Local Allocation)**

전역 교체(Global replacement) : 프로세스가 모든 프레임 집합에서 교체 프레임을 선택

* 전역 교체가 지역 교체 알고리즘보다 더 좋은 시스템 성능을 나타냄

지역 교체(Local replacement) : 각, 프로세스가 오로지 할당된 프레임 집합에서만 선택

* 프로세스별 성능의 일관성 향상
* 하지만, 활용도가 낮은 메모리일 가능성이 있음

**9.5.4 비균등 메모리 접근(Non-Uniform Memory Access)**

비균등 메모리 접근(Non-Uniform Memory Accesss – NUMA) : 메모리 접근 시간이 현저하게 차이가 나는 시스템

최적의 성능을 위해선 프로세스가 실행 중인 CPU에 가능한 가장 “가까운” 메모리 프레임이 할당되도록 하는 것

“가까운” = “최소 지연시간을 가진”

실행 중인 많은 스레드를 가진 프로세스의 스레드들이 여러 시스템 보드로 스케줄 된다면 메모리는 어떻게 할당되어야 하겠는가?

Solaris는 커널 내에 Igroup이라는 개체를 만들어서 이 문제를 해결하였음

Igroup들은 가까운 CPU와 메모리를 모음.

그룹 간의 지연 시간을 고려하면 Igroup들은 계층 구조를 형성하게 됨

* 따라서, Solaris는 모든 스레드와 메모리 할당을 Igroup 단위로 처리하는데

이것이 불가능하다면 필요한 나머지 자원에 대해서는 가까운 Igroup을 선택

하여 할당

**9.6 스레싱(Thrashing)**

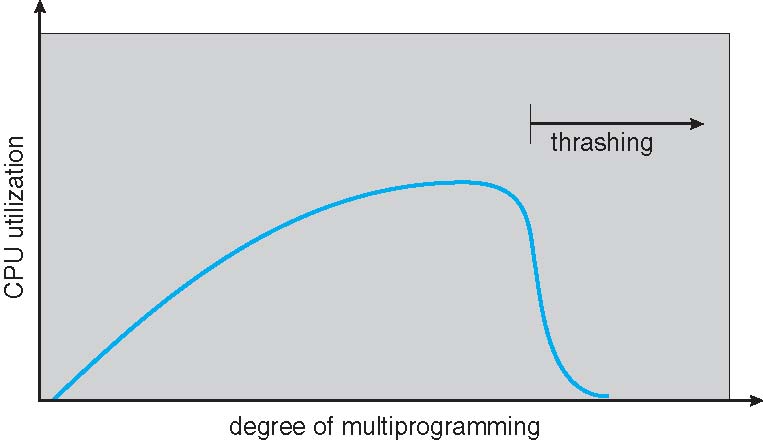
스레싱(Thrashing) : 어떤 프로세스가 실제 실행보다 더 많은 시간을 페이징에 사용하고 있는 경우

**9.6.1 스레싱의 원인(Cause of Thrashing)**

운영체제는 CPU의 이용률(utilization)을 감시

만약, CPU 이용률이 너무 낮아지면 새로운 프로세스를 시스템에 더 추가해서 다중 프로그래밍의 정도(degree of multi-programming)를 높임

프로세스들이 페이징 장치를 기다리는 동안 CPU 이용률이 떨어지기 때문에 새로운 프로세스를 추가하여 다중 프로그래밍의 정도를 더 높임. 새로 시작하는 프로세스는 실행 중인 프로세스들로부터 프레임을 가져오고자 하며 더 많은 페이지 부재와 더 긴 페이징 장치 대기 시간을 야기. 결국, 스레싱이 일어나게 됨 즉, 실질 메모리 접근 시간은 증가하고 프로세스들은 페이징하는 데 시간을 다 소비하게 되어 아무런 일도 할 수 없게 됨



스레싱은 지역 교환 알고리즘(local)이나 우선순위 교환 알고리즘을 사용하면 제한할 수 있음

지역 교환 알고리즘 하에서는 한 프로세스가 스레싱을 유발하더라도 다른 프로세스로부터 프레임을 뺏어 올 수 없으므로, 다른 프로세스는 스레싱으로부터 자유로울 수 있음

스레싱 현상을 방지하기 위해서는 각 프로세스가 필요로 하는 최소한의 프레임 개수를 보장해야 함

* 9.6.2절 작업 집합 방법을 통해 알 수 있음
* 이 방법은 프로세스 실행의 **지역성 모델(locality model)**을 기반으로 함

스레싱이 발생하는 이유 ?  
Σ size of locality > total memory size

필요로 하는 지역성의 크기보다 적은 프레임을 할당하게 되면, 그 프로세스는 접근해야 하는 모든 페이지를 메모리에 유지할 수 없기 때문에 지속적으로 페이지 부재를 발생시키게 됨

**9.6.2 작업 집합 모델(Working-Set Model)**

9_19.pdf

메모리 참조 패턴의 지역성(locality)

한 번도 안 올라온 페이지는 메모리에 안 올라옴.

Δ ≡ working-set window ≡ 고정 페이지 참조 수

WSSi( working set of Process *Pi* ) = 가장 최근의 Δ에서 참조된 총 페이지 수(시간에 따라 다름)

작업 집합의 정확도는 Δ의 선택에 따라 좌우

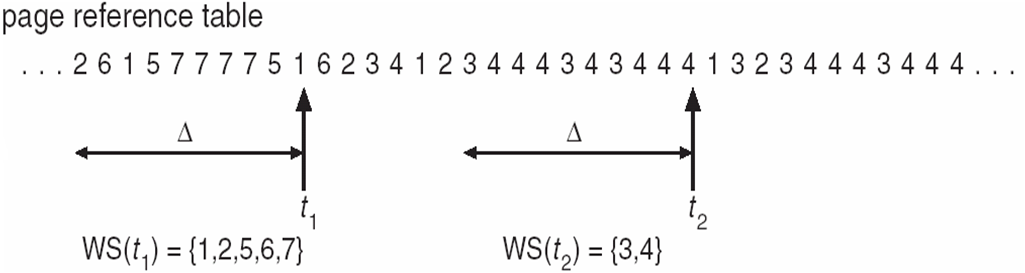
Δ 값이 너무 작으면 전체 지역을 포함하지 못함

Δ 값이 너무 크면 여러 지역성을 과도하게 수용

Δ 값이 무한히 크면 이 집합은 프로세스가 실행 중에 만난 모든 페이지의 집합

*D* = Σ *WSSi* ≡ 총 수요 프레임

if *D* > *m(총 메모리 크기)* ⇒ Thrashing 유발



Δ가 10인 경우 => 10개 참조

Policy if *D* > m, 프로세스 중 하나 일시 중단 또는 Swap out

이 모델의 어려운 점은 작업 집합을 추적하는 일

Example: Δ = 10,000

1. 5000번의 참조마다 타이머 인터럽트를 건다고 가정

2. 타이머 인터럽트가 걸릴 때 마다 각 페이지의 현재 참조 비트의 값을 저장해 두고 참조 비트는 0으로 재설정

- 따라서, 페이지 부재가 발생하면 바로 앞 10000 ~ 15000번의 참조 사이에 그 페이지가 사용됬는지 알기 위해 현재 참조 비트와 앞서 저장된 2개의 참조 비트 값을 검사할 수 있음

3. 셋 중 어느 한 비트라도 1인 페이지는 작업 집합에 속함

하지만, 이 방법은 완전히 정확한 것은 아님

향상시키기 위해선 과거 기억 비트와 타이머 인터럽트 빈도를 높이면 불확실성을 줄일 수 있음

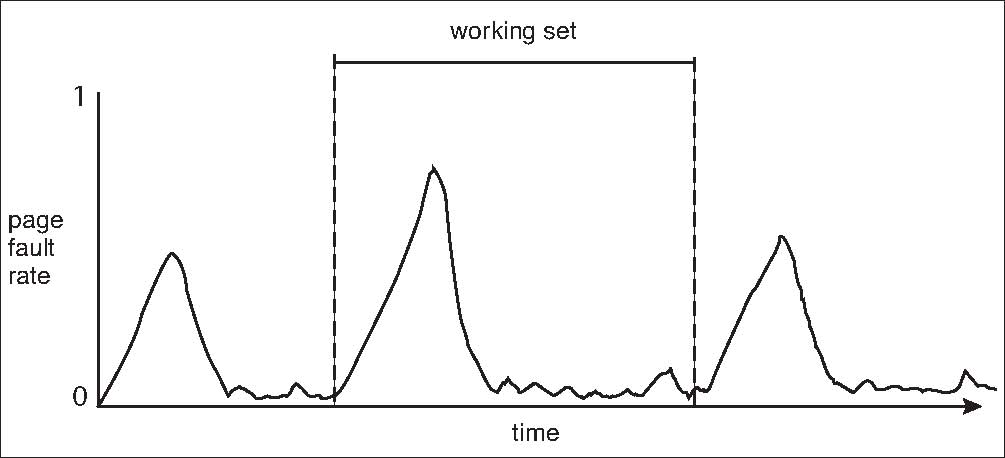
페이지 부재 빈도

9_21.pdf

페이지 부재율이 낮으면 프로세스는 프레임을 줄임

페이지 부재율이 높으면 프로세스는 프레임을 늘림

작업 집합과 페이지 부재율



데이터와 코드에 대한 참조는 시간의 흐름에 따라 한 지역에서 다른 지역으로 옮겨감. 프로세스가 스레싱 중이 아니면 페이지 부재율은 시간이 지남에 따라 고점과 저점 사이를 오르내리게 됨.

페이지 부재율은 고점(高点)은 새로운 지역으로 들어가 Demand paging이 시작되는 경우에 발생 => 그 후, 새로운 지역의 작업 집합이 메모리에 올라오고 나면 부재율은 낮아짐

어떤 고점으로부터 다음 고점까지의 시간 간격은 한 작업 집합에서 다른 작업 집합으로의 전이를 나타냄

**9.7 메모리 사상 파일(Memory-Mapped Files)**

메모리 사상(memory-mapping) : 프로세스의 가상 주소 공간 중 일부를 관련된 파일에 할애하는 것

**9.7.1 기본 기법**

9_22.pdf

메모리 사상 파일

9_23.pdf

메모리 사상 입출력을 사용한 공유 메모리

**9.7.2 Windows API에서 공유 메모리**

먼저 사상(Mapping) 할 파일에 대한 file mapping을 생성함

* 그 후, 사상된 파일의 프로세스 가상 주소 공간상의 뷰(view)를 만듬

생산자 / 소비자 프로세스를 고려

생산자 프로세스는 메모리 사상 기능을 사용하여 공유 메모리 객체를 생성

CreateFile() 함수를 호출하여 사상될 파일을 열고 그에 대한 HANDLE을 반환받음

CreateFileMapping() 함수를 사용하여 이 파일 HANDLE의 사상을 생성

CreateFileMapping()을 호출하여 named 공유 메모리 객체(named shared-memory object)

를 생성

MapViewOfFile() 함수를 이용하여 사상된 파일의 가상 주소 공간 상의 뷰를 생성

운영체제 책 P495

**9.7.3 메모리 사상 입출력**

레지스터와 시스템 메모리 간의 데이터 전송을 위해서는 대개 특별한 입출력 명령어가 사용되는데 이 때 좀 더 편리하게 접근할 수 있도록, 많은 컴퓨터 아키텍처는 메모리 사상 입출력 기능을 제공하고 있다.

Programmed 입출력(PIO) : CPU가 제어 비트 값을 확인하기 위해 폴링(polling)방식을 사용 하는 것

폴링(polling) : 계속적으로 루프를 돌면서 장치가 준비 상태로 들어가는지 확인하는 것

CPU가 폴링하지 않고 장치가 준비되었을 때 인터럽트를 받는 경우는 interrupt driven이라고 부름

**9.8 커널 메모리의 할당(Allocating Kernel Memory)**

사용자 메모리와는 다르게 처리 됨

커널 메모리는 별도의 free-memory pool에서 할당 됨 // 왜 ?

1. 커널은 메모리를 조심스럽게 사용하여야 하며 단편화에 의한 낭비를 최소화하고자 함.

* 단편화 : 기억 장치의 빈 공간 또는 자료가 여러 개의 조각으로 나뉘는 현상을 말함

1. 물리 메모리에 직접 접근하는 특정 하드웨어 장치는 물리적으로 연속적인 메모리를 필요로 하는 경우가 있음

**9.8.1 버디 시스템**

물리적으로 연속된 페이지들로 이루어진 고정된 크기의 세그먼트로부터 메모리를 할당하는 시스템

메모리는 2의 거듭제곱 단위로 할당 ( 4 KB, 8 KB, 16 KB 등등..)

2의 거듭제곱 크기가 아닌 메모리 요구는 가장 가까운 2의 거듭제곱 크기로 올림 됨

EX) 메모리 세그먼트의 크기는 초기에 256 KB라 가정하고 커널이 21 KB의 메모리를 요구했다고 가정

1. 세그먼트는 Al, Ar로 표시하는 128 KB 크기의 두 개의 **버디(Buddies)**로 나누어짐
2. 그 중 하나는 다시 두 개의 64 KB 크기의 버디(Bl, Br)로 나누어 짐. 2
3. 21 KB에 가장 가까운 2의 거듭제곱은 32 KB이므로 Bl 또는 Br은 다시 두개의 32 KB Cl, Cr들로 나뉨

9_26.pdf

장점 : **합병(coalescing)**이라고 부르는 과정을 통해 서로 인접한 버디들이 손쉽게 하나의 큰 세그먼트로 합쳐질 수 잇다는 점

Ex ) 21 KB 요청을 위해 할당 된 Cl을 해제하면 Cl 과 Cr을 합칠 수 있고 결국, 다시 원래의 256 KB 세그먼트로 돌아가게 됨

단점 : 2의 거듭제곱으로의 올림이 할당된 세그먼트 내의 단편화를 가져온다는 것

EX) 33 KB의 요청

**9.8.2 슬랩 할당(Slab Allocation)**

슬랩(Slab)은 하나 이상의 물리적으로 연속된 페이지

캐시(cache)는 하나 혹은 그 이상의 슬랩(Slab)들로 구성됨

각, 커널 자료 구조마다 하나의 캐시가 존재함

캐시가 생성되면 초기에는 free라고 표시된 몇 개의 객체들이 캐시에 할당 됨

초기에는 캐시 내의 모든 객체가 free로 표시 됨.

객체가 필요하면 free 객체들 중 하나를 캐시로부터 할당하며 할당된 객체는 used라고 표시

9_27.pdf

**슬랩 할당기**

Slab Allocator in Linux

struct task\_struct 자료 구조로 표현 약, 1.7 KB 정도를 요구

새로운 프로세스를 생성할 때 캐시로부터 struct task\_struct 객체를 위한 메모리 할당을 요청 => 이 때, 캐시는 요청을 처리하기 위해 이미 슬랩에 할당되어 있던 free로 표시된 struct task\_struct 객체를 사용.

1. Full : All used
2. Empty : All Free
3. 부분적 : mix of free and used

슬랩 할당기는 먼저 Partial 슬랩의 free 객체를 이용해 요청을 처리하려고 시도.

Partial 슬랩이 없으면 Empty 슬랩으로부터 Free 객체를 할당

Empty 슬랩도 없는 경우에는 새로운 슬랩이 연속된 물리 메모리에서 할당되어 캐시에 주어짐

장점 : 단편화에 의해 낭비되는 메모리가 없음

메모리 요청이 빠르게 처리됨

**9.9 기타 고려 사항(Other Considerations)**

**9.9.1 프리페이징(Prepaging)**

프로세스 시작 시 발생하는 많은 페이지 오류 수를 줄이기 위함

관련된 모든 페이지를 사전에 한꺼번에 메모리 내로 가져오는 기법

하지만, 사전 준비된 페이지를 사용하지 않으면 I/O 및 메모리가 낭비됨

s개의 페이지를 미리 올려놓았는데 그중 a부분만이 실제로 사용되었다고 가정.

( 0 <= a <= 1) 문제는 s\*a 만큼 줄어든 페이지 부재 투자가 s \* ( 1 – a ) 만큼의 불필요한 페이지를 미리 올려놓은 것보다 더 좋은가의 여부.

a가 1에 가까울수록 프리페이징이 좋은 것

**9.9.2 페이지 크기(Page Size)**

시스템을 개발할 때는 페이지 크기를 결정해야 함

항상 최적일 수 없는 페이지 크기를 어떻게 결정 ?

1. 단편화 고려
2. 테이블 사이즈
3. 정밀도
4. 입출력 시간
5. 페이지 부재 수
6. 지역성
7. TLB Size 와 효율성

페이지 크기는 항상 2의 제곱이며 일반적으로 2^12 (4,096 바이트) ~ 2^22 (4,194,304 바이트) 범위

**9.9.3 TLB Reach**

TLB reach : TLB로부터 액세스할 수 있는 메모리 공간의 크기를 의미

TLB reach = TLB size(TLB에 있는 항목 수) \* Page Size

한 프로세스의 작업 집합이 TLB에 다 들어올 수 있으면 가장 이상적

TLB reach를 늘릴 수 있는 다른 방안

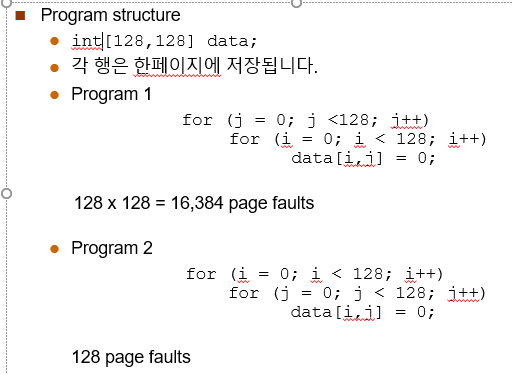
1. 페이지 크기를 늘림

* 큰 크기의 페이지를 필요로 하지 않는 프로그램에서는 단편화가 증가하여 문제가 발생할 수도 있음

1. 운영체제가 여러 개의 페이지 크기를 허용하는 방법이 있을 수 있음

* 여러 개의 크기를 가짐에 따라 단편화 문제를 해결 할 수 있음

**9.9.3 프로그램 구조(Program Structure)**



**9.9.3 입출력 상호 잠금(I/O Interlock)과 페이지 잠금(locking)**

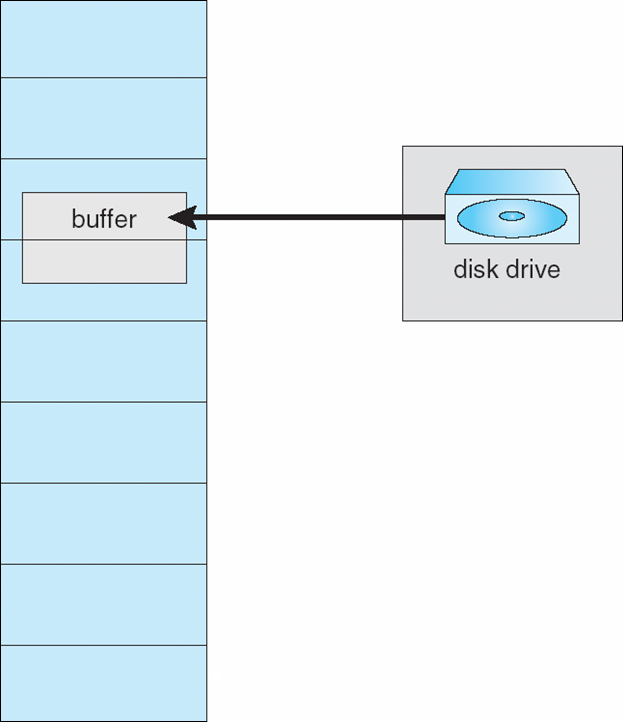
페이지의 일부는 메모리에 고정시키는 것이 필요한 경우가 있음

(EX -> 사용자 입출력 요구 시 프로세스가 페이지 부재를 일으키고 전역 교체 알고리즘을 사용하여 대기 중인 프로세서의 버퍼 메모리를 포함한 페이지를 교체함. 그 페이지는 페이지 아웃되고 얼마 후 입출력 요청이 큐의 맨 앞으로 이동하고 입출력 작업이 처리됨. 그러나, 원래 버퍼가 있는 프레임은 이미 다른 프로세스에 의해 사용중임

위의 문제를 해결하려면

1. 사용자 공간에는 입출력을 하지 않는 것
2. 페이지를 메모리에서 잠금

* 잠금 비트를 각 프레임마다 두고 프레임이 잠기면 교체 고려 대상에서 제외시킴



이러한 페이지 **고정(pinning)**은 널리 사용 됨

**9.9.3 운영체제의 예(Operating-System Example)**

Windows

Solaris

**9.10.1 Windows**

Windows는 클러스터링(clustering) 방식으로 요구 페이징 가상 메모리를 구현한다.

클러스터링에서는 페이지 부재가 발생하면 그 페이지뿐만 아니라 그 페이지 다음의 몇 페이지들도 함께 가져옴

프로세스는 처음 생성될 때 작업 집합의 상한치와 하한치가 설정됨

작업 집합의 하한치 : 시스템이 보장하는 최소한의 메모리 공간 크기

프로세스는 작업 집합의 상한치까지 페이지를 할당할 수 있음

시스템 전체적으로 가용 공간이 임계값보다 낮아지면 시스템은 **작업 집합 자동 조절 정책(automatic trimming)**을 써서 임계값에 해당하는 가용공간 확보

이 정책은 하한치보다 넘게 할당 된 페이지가 있으면 FIFO 알고리즘을 통해 하한치까지 내려갈 때 까지 페이지를 회수함

**9.10.2 Solaris ( 이런 게 있다고만 알아놓음 될 듯)**

가용 페이지 리스트에는 lotsfree(paging을 시작하는 임계값 – 사용가능한 메모리 양) 파라미터가 지정

가용 공간의 크기가 lotsfree 보다 작으면 **pageout**이라는 프로세스를 가동 시킴

**Desfree** – 페이징 증가에 대한 임계값 매개 변수

**Minfree** – 스왑 중인 임계값 매개 변수

우선 순위 페이징은 코드 페이지를 처리하는 데 우선 순위를 부여합니다.

9_29.pdf