10장. 가상 메모리 (Virtual Memory)

순천향대학교 컴퓨터공학과
이 상 정

강의 목표 및 내용

- 목표
  - 가상 메모리 시스템의 이점
  - 요구 페이징과 페이지 교체 정책, 물리 페이지 할당이라는 개념
  - 작업 집합(working-set) 모델에 대해 논의
- 내용
  - 배경
  - 요구 페이징
  - 쓰기 시 복사
  - 페이지 교체
  - 프레임의 할당
  - 스레싱
  - 메모리 사상 파일
  - 커널 메모리의 할당

배경 (Background) (1)

- 프로그램 코드는 실행 시에 메모리에 적재되어 있어야 하지만, 전체 프로그램의 사용은 빈번하지 않음
  - 에러 코드, 비정상적인 루틴, 대구모 데이터 구조
- 전체 프로그램 코드가 동시에 필요하지 않음

- 프로그램의 일부만 메모리에 적재하고 실행 시 이점
  - 프로그램이 더 이상 물리 메모리의 크기 제약을 받지 않음
  - 각 프로그램의 실행 시 메모리를 덜 차지함
    - 동시에 더 많은 프로그램의 실행이 가능
    - 응답시간이 증가시키지 않으면서 CPU 활용도와 효율성 증대
  - 프로그램을 메모리로 적재하거나 스왑 시에 더 적은 I/O 필요
    - 각 사용자 프로그램이 더 빨리 실행

배경 (2)

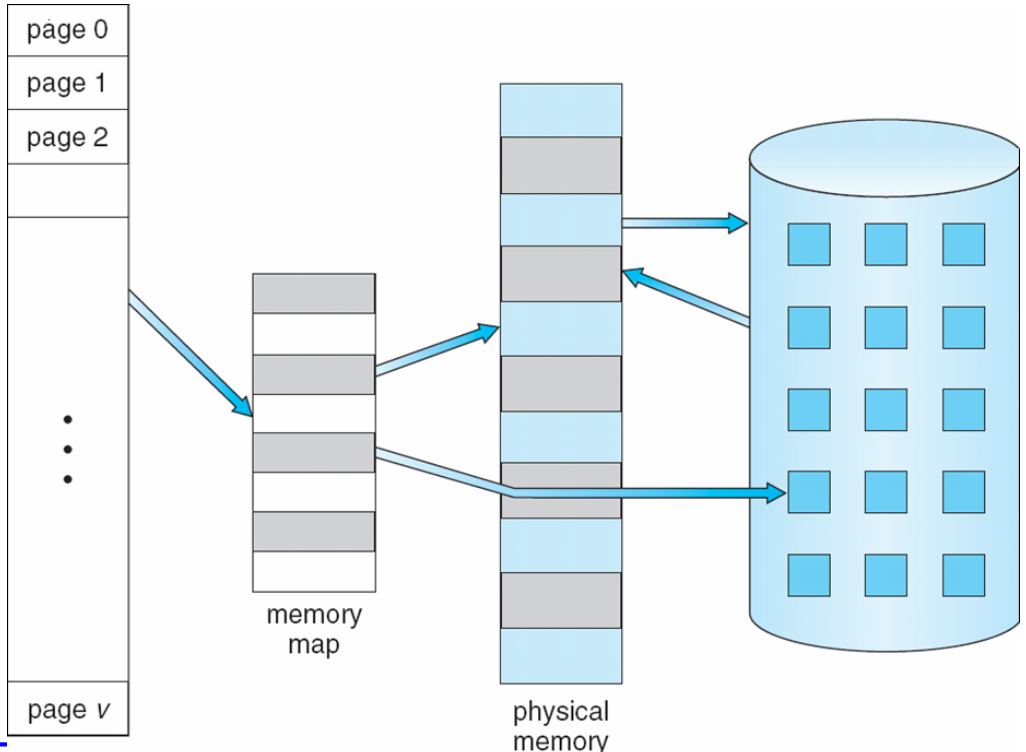
- 가상 메모리(virtual memory)는 실제의 물리 메모리 개념과 사용자의 논리 메모리 개념을 분리
  - 프로그램의 일부분만 메모리에 올려 놓고 실행
    - 현재 실행되고 있는 프로세스가 반드시 물리 메모리에 놓여 있어야 된다는 조건을 완화
  - 물리 주소 공간보다 훨씬 큰 논리 주소 공간(가상 주소 공간)을 제공
    - 작은 메모리를 가지고도 얼마든지 큰 가상 주소 공간을 프로그래머에게 제공
  - 여러 프로세스들에게 공유되는 주소 공간을 허용
  - 효율적인 프로세스 생성이 가능
  - 더 많은 프로그램들이 병행 실행 될 수 있음
  - 프로세스 적재 및 스왑 시에 더 적은 I/O(입출력 과정) 필요

배경 (3)

- 가상 메모리 공간(virtual memory space)는 프로세스가 메모리에 저장되는 논리적인 관점
  - 0번지 주소부터 시작되는 연속적인 주소 공간
  - 반면, 물리 메모리는 페이지 프레임들로 구성
    - 프로그램이 분할되고, 비연속적으로 물리 메모리에 배치
  - 논리주소를 물리주소로 변환하는 MMU(Memory Management Unit)이 필요

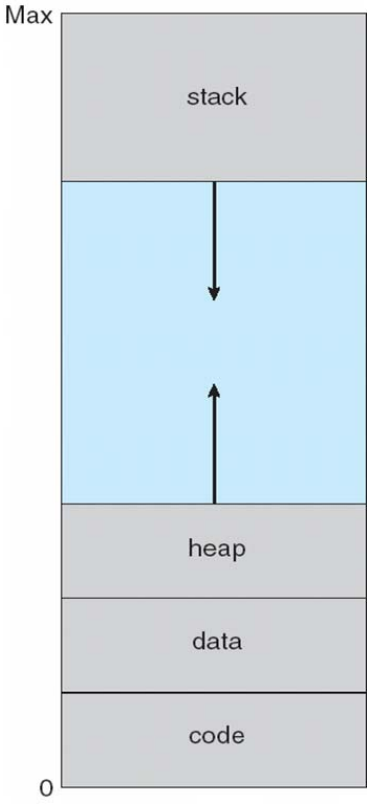
- 가상 메모리 구현
  - 요구 페이징 (demand paging)
  - 요구 세그멘테이션 (demand segmentation)

물리 메모리보다 큰 가상 메모리

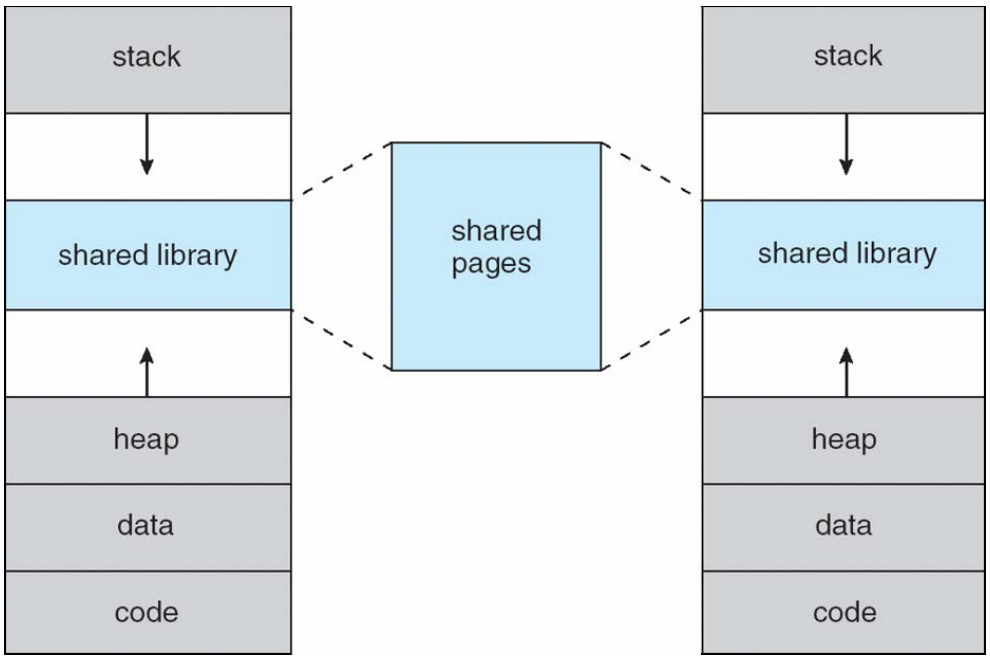


프로세스의 가상 주소 공간

- 일반적으로 논리 주소공간의 최대 주소에서 스택이 시작되어 낮은 주소로 확장되고, 힙은 높은 주소 방향으로 확장
  - 두 영역 사이에 사용되지 않은 주소 공간은 빈 영역(hole)
    - 힙이나 스택이 새로운 페이지로 확장되기 전에는 물리 메모리가 필요하지 않음
  - 확장될 빈 영역의 성긴 주소 공간 (sparse address space)에 동적 연결 라이브러리 등이 배치
  - 시스템 라이브러리 등은 가상 주소 공간을 매핑하여 공유

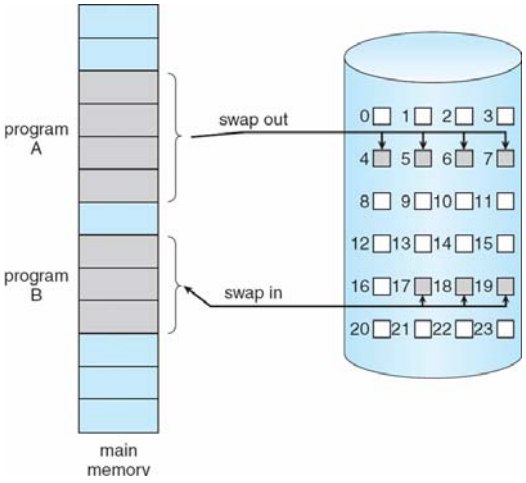


가상 메모리를 사용할 때의 공유 라이브러리



요구 페이징 (Demand Paging)

- 프로세스 전체를 메모리에 적재하지 않고, 필요한 페이지만을 물리 메모리에 적재
  - 사용되지 않을 페이지를 메모리로 가져오지 않음으로써 시간 낭비와 메모리 공간 낭비를 줄임
  - 입출력 과정 감소
  - 적은 메모리 사용
  - 더 빠른 응답
  - 더 많은 사용자 허용
- 스와핑을 갖는 페이징 시스템과 유사 (오른쪽 그림)
- 필요한 페이지를 참조
  - 잘못된 참조 => 취소
  - 물리 메모리에 없으면 => 메모리로 가져옴
- 게으른 스와퍼(lazy swapper)
  - 페이지가 필요하지 않으면 메모리에 적재하지 않음
  - 페이지를 관리하는 스와퍼를 페이저(pager)라 함

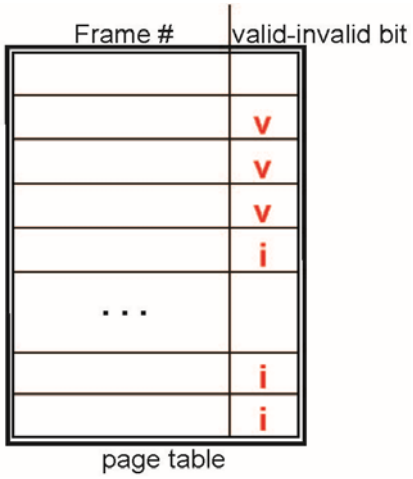


요구 페이징 기본 개념

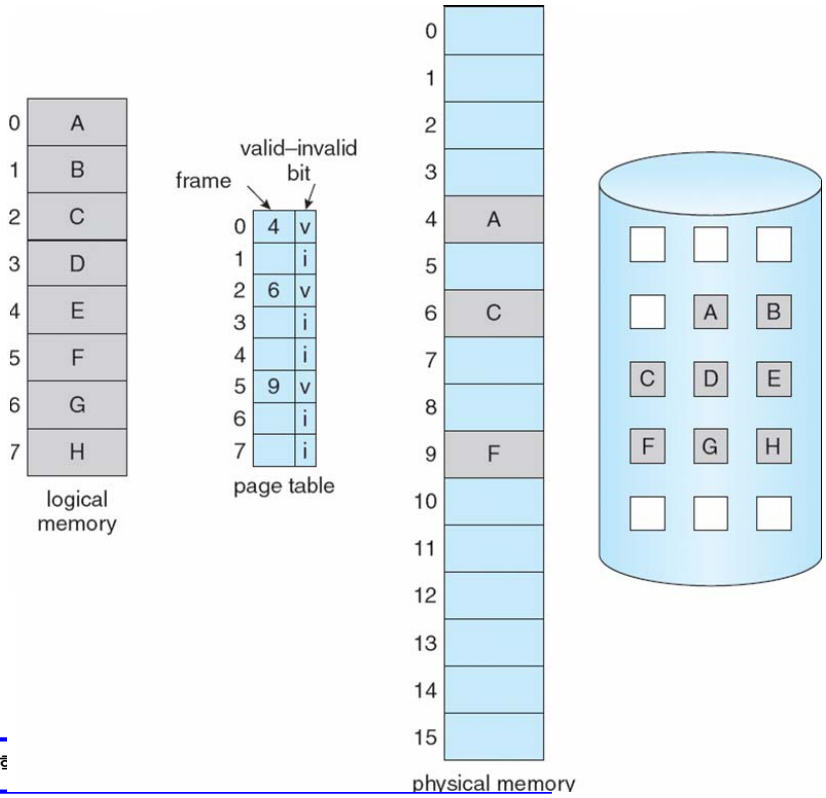
- 페이저가 필요한 페이지들을 메모리에 적재
- 메모리에 적재될 페이지들을 어떻게 결정하는가?
  - 요구 페이징을 위해 새로운 MMU 기능이 필요
- 필요한 페이지들이 이미 메모리에 있으면
  - 요구 페이징이 아닌 시스템과 다를게 없음
- 필요한 페이지들이 메모리에 없으면
  - 이를 감지하여 저장장치에서 메모리로 페이지를 적재
    - 프로그램 동작 변경 없음
    - 프로그래머가 코드를 변경할 필요 없음

유효-무효 비트 (Valid-Invalid Bit)

- 페이지가 메모리에 올라와 있는지 구별을 위해 페이지 테이블에 유효-무효 비트 표시
  - 유효비트(v, valid)로 세트되면 페이지가 메모리에 있음 표시
  - 무효비트(i, invalid)로 세트되면 페이지가 메모리에 없음을 표시
  - 초기에는 i으로 지정
- 주소 변환 중 무효비트 i로 세트되어 있으면 페이지 부재(page fault) 발생



일부 페이지들이 메인 메모리 없는 경우 페이지 테이블

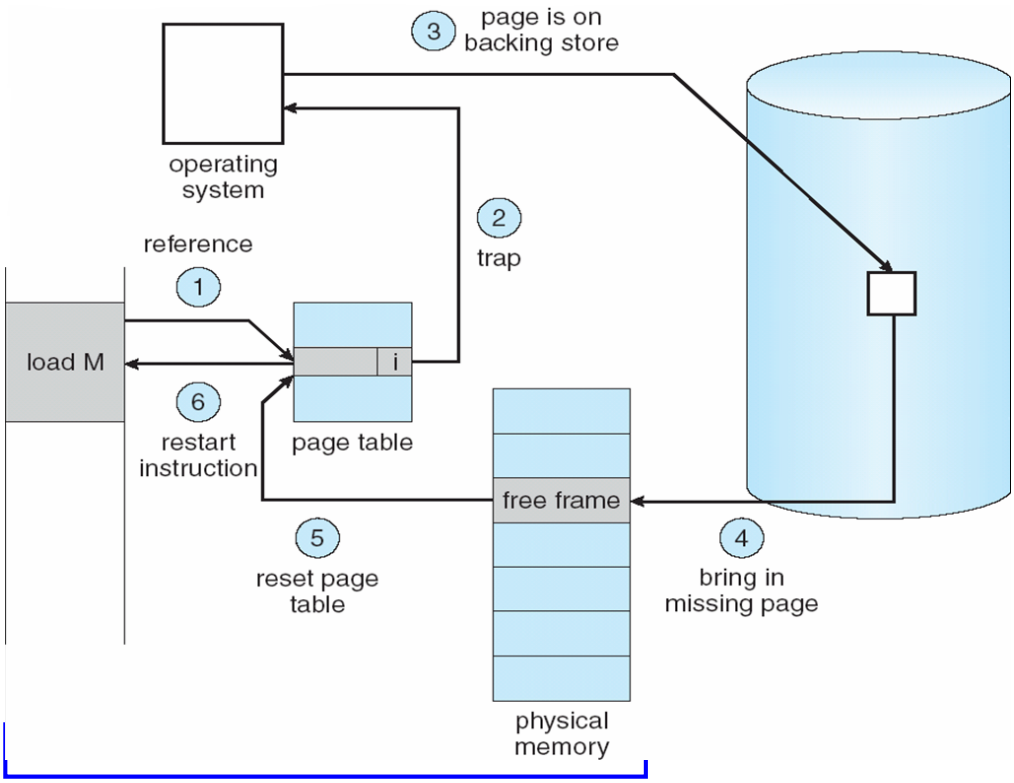


페이지 부재 (Page Fault)

- 프로세스가 메모리에 올라와 있지 않는 페이지를 접근하는 경우 페이지 부재 (page-fault) 트랩(trap)이 발생

- 페이지 부재 처리 과정
  1. 운영체제는 내부 테이블(PCB와 함께 유지)을 검사하여 메모리 참조가 유효인지 무효인지를 조사
     - 무효한 참조 -> 프로세스 중단
     - 메모리에는 없는 유효한 참조
  2. 빈 공간 즉, 자유 프레임(free frame)을 검색
  3. 새로이 할당된 프레임으로 해당 페이지를 읽어 들이도록 요청
  4. 페이지 테이블을 갱신 (유효 비트 세트)
  5. 페이비 부재 트랩에 의해 중단되었던 명령어를 다시 수행

페이지 부재 처리 과정



요구 페이징의 성능 (Performance of Demand Paging)

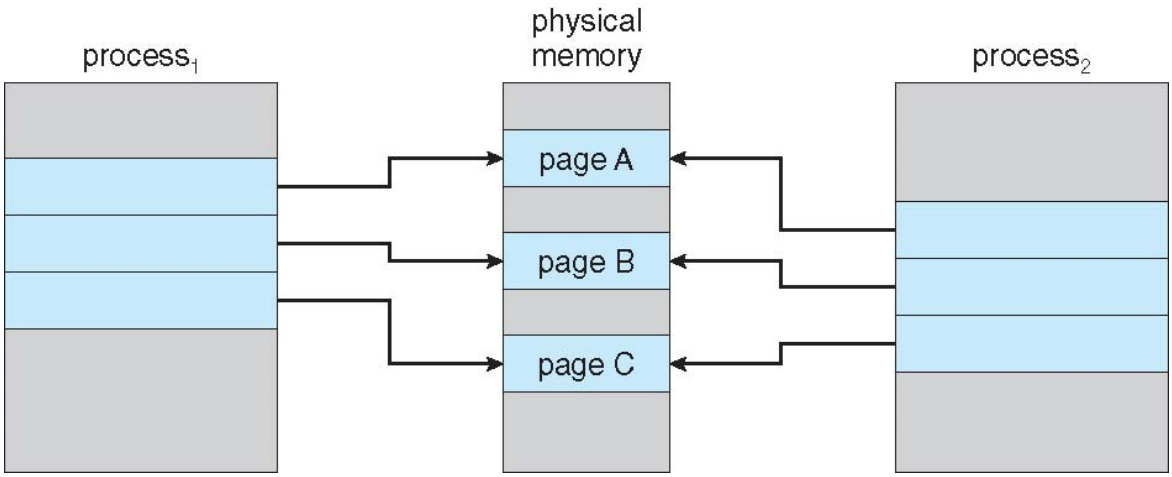
- 유효 접근 시간(effective access time)
  = (1 - p) × ma + p × 페이지 부재 처리 시간
  - p: 페이지의 부재 확률, 0 ≤ p ≤ 1.0
  - ma: 메모리 접근 시간(memory access time)
  - 페이지 부재 처리 시간
    - 인터럽트 처리, 페이지 읽기, 프로세스 재시작
  - 예
    - 메모리 접근 시간: 200ns
    - 페이지 부재 처리 시간: 8ms
    - 실질 접근 시간 = (1-p) x 200 ns + p x 8,000,000ns
      = 200 + 7,999,800 x p
    - 1000번에 한 번 접근 시 페이지 부재가 발생한다면 유효 접근 시간은 8.2us
      => 요구 페이징으로 인해 40배 느려짐

쓰기 시 복사 (Copy-on-Write)

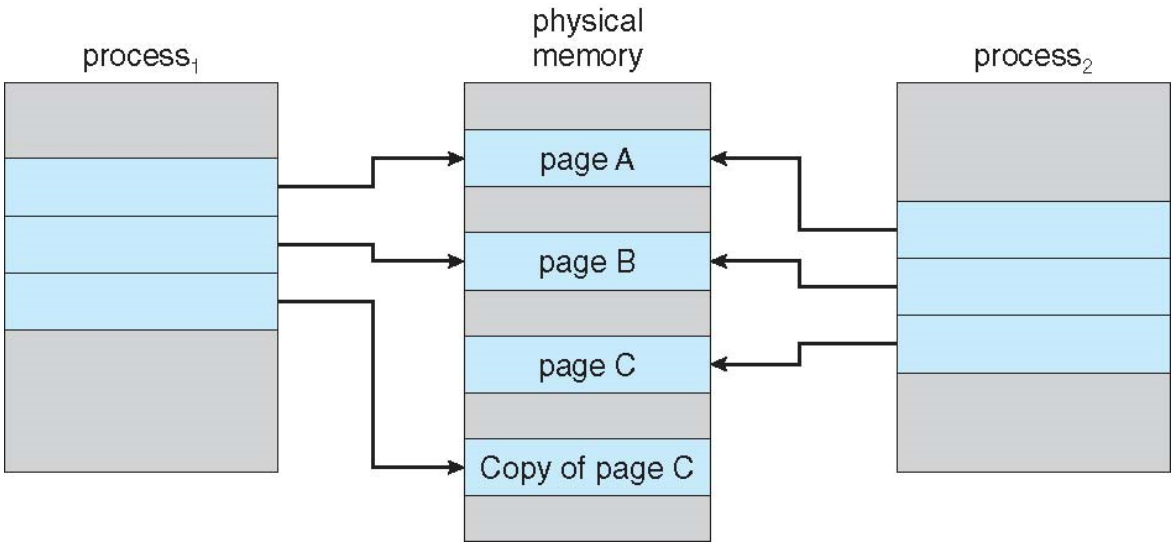
- 가상 메모리는 프로세스 생성(fork()) 시 페이지를 공유하여 생성 시간을 줄임
  - 쓰기 시 복사 적용

- 쓰기 시 복사 (copy-on-write)
  - 자식 프로세스가 시작할 때 부모의 페이지를 당분간 함께 공유하여 사용
  - 둘 중 한 프로세스가 공유중인 페이지에 쓸 때 페이지의 복사본이 생성
  - 수정되는 페이지만 복사본을 만들어 효율적인 프로세스 생성

프로세스 1이 페이지 C를 수정하기 전



프로세스 1이 페이지 C를 수정한 후

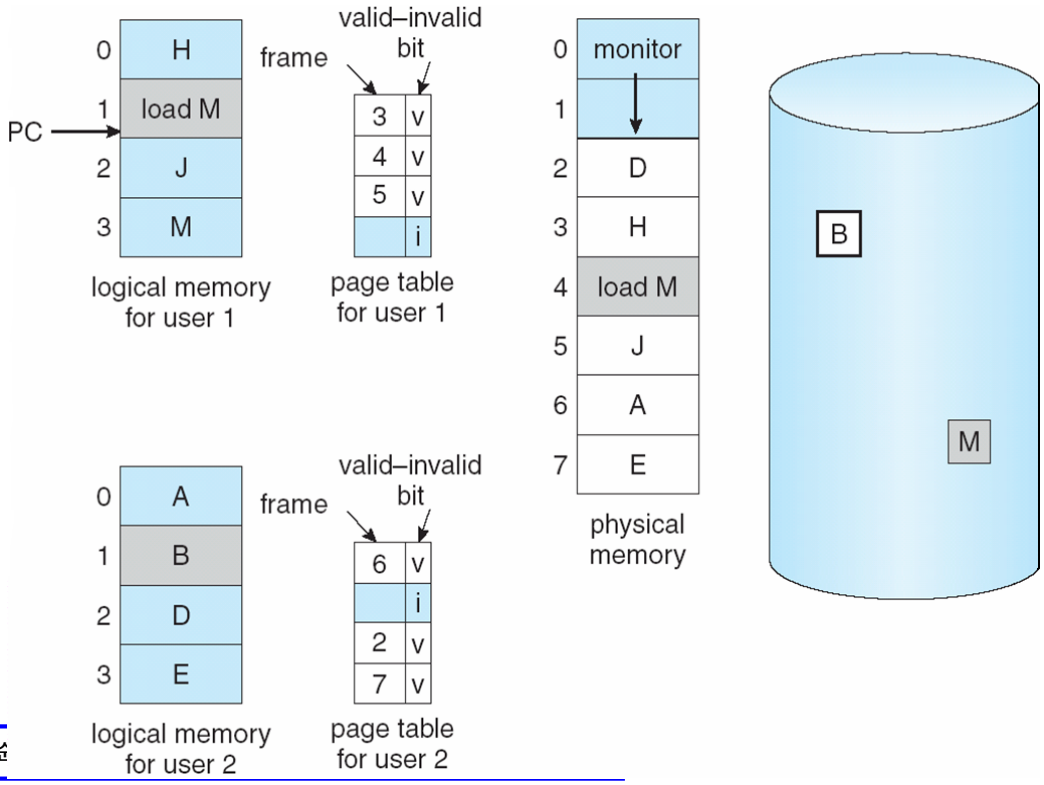


페이지 교체 (Page Replacement)

- 모든 메모리가 사용 중이어서 빈 프레임이 없는 경우는?
  - 페이지 교체 (page replacement)

- 페이지 교체 알고리즘(page replacement algorithm)
  - 사용중인 페이지 중 하나를 선택하여 교체(replacement)
  - 새로운 페이지는 메모리로 가져옴
  - 교체되는 페이지(victim)는 디스크에 저장
    - 수정된 페이지만을 디스크에 저장하여 페이지 전송 오버헤드 줄임
    - 변경비트(modify bit, dirty bit) 사용
  - 성능
    - 페이지 부재가 최소가 되도록 기대
    - 일부 페이지는 자주 교체되어 메모리에 여러 번 가져오는 경우도 발생

페이지 교체 필요성



기본적인 페이지 교체

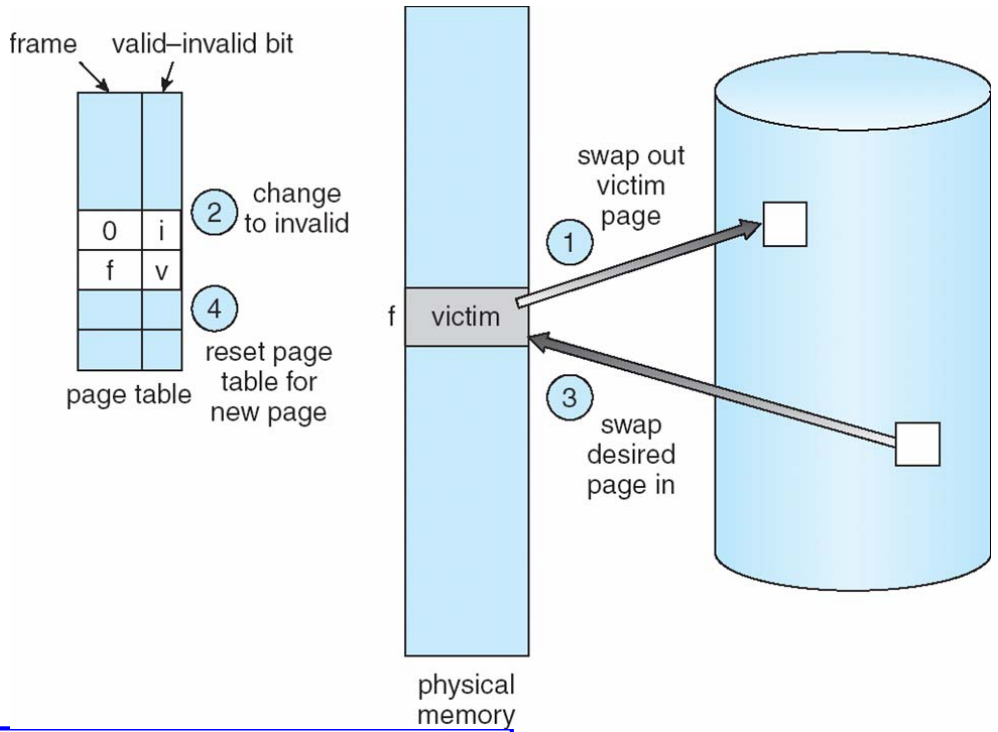
1. 디스크에서 필요한 페이지의 위치를 알아냄

2. 빈 페이지 프레임을 찾음
   a. 빈 프레임이 있다면 그것을 사용
   b. 없다면 희생될(victim)프레임을 선정하기 위하여 페이지 교체 알고리즘을 가동
   c. 희생될 페이지를 디스크에 기록하고, 관련 테이블을 수정

3. 새롭게 비워진 프레임에 새 페이지를 읽어오고 테이블을 수정

4. 사용자 프로세스를 재시작

페이지 교체

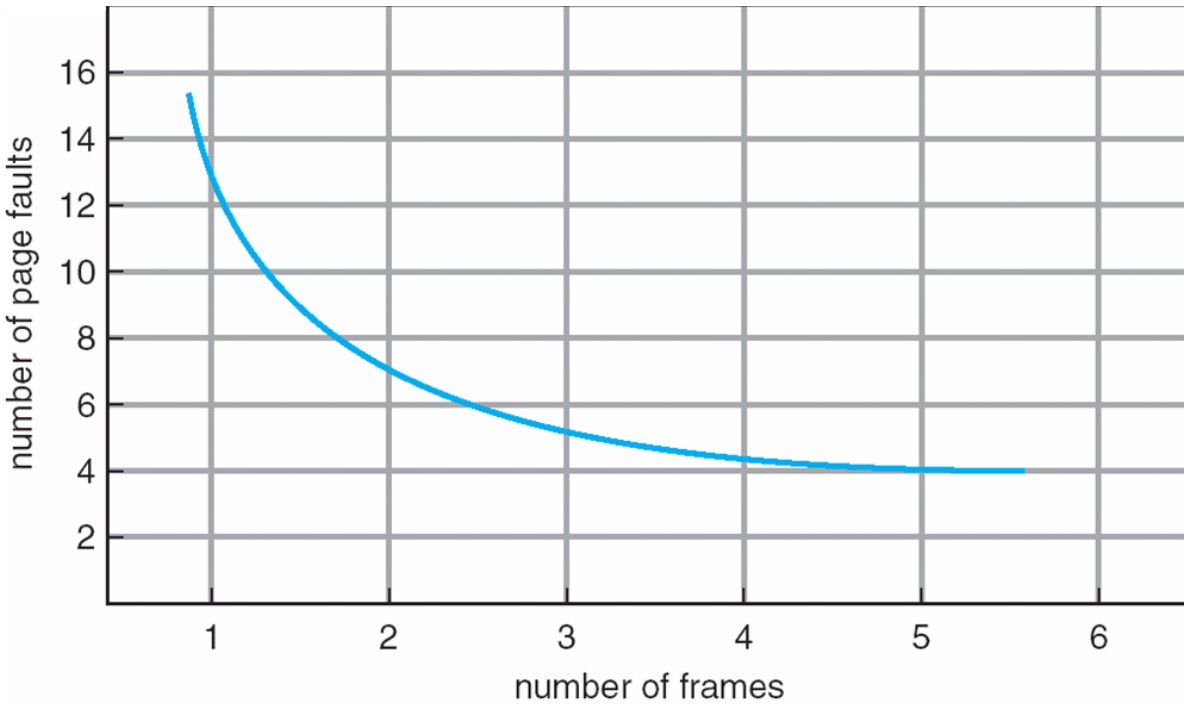


페이지 교체 알고리즘 (Page Replacement Algorithm)

- 페이지 부재율(page-fault rate)이 가장 낮은 것을 선정

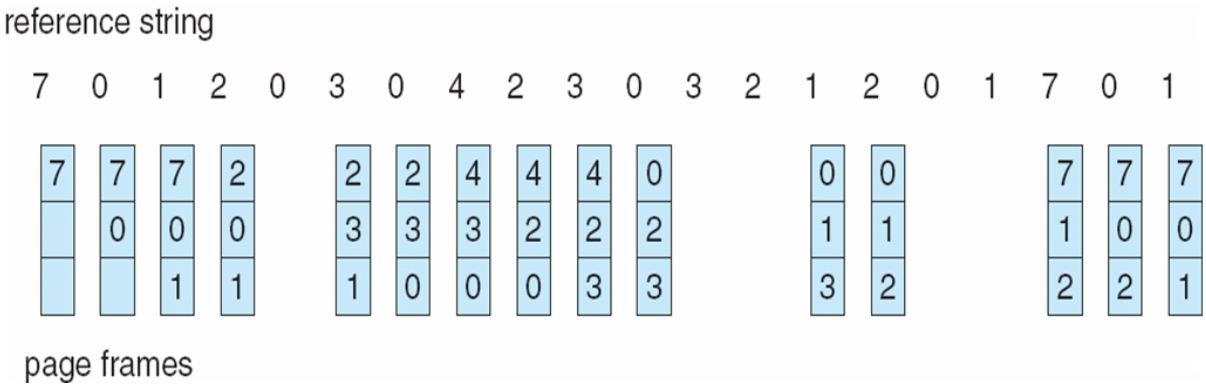
- 페이지 교체 알고리즘의 성능
  - 특정 메모리 참조 나열에 대해 알고리즘을 적용하여 페이지 부재 발생 횟수를 계산하여 평가
  - 메모리 주소의 나열을 참조열(reference string)이라고 함
- 참조열 예
  - 페이지 크기 100 바이트 가정
  - 주소열
    0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
  - 참조열
    1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

페이지 부재 횟수 대 프레임 수의 그래프



FIFO 페이지 교체 (FIFO Page Replacement)

- 가장 간단한 페이지 교체 알고리즘
- 예
  - 3개의 프레임
  - 15개의 페이지 부재 발생



Belady의 모순(Belady's Anomaly)

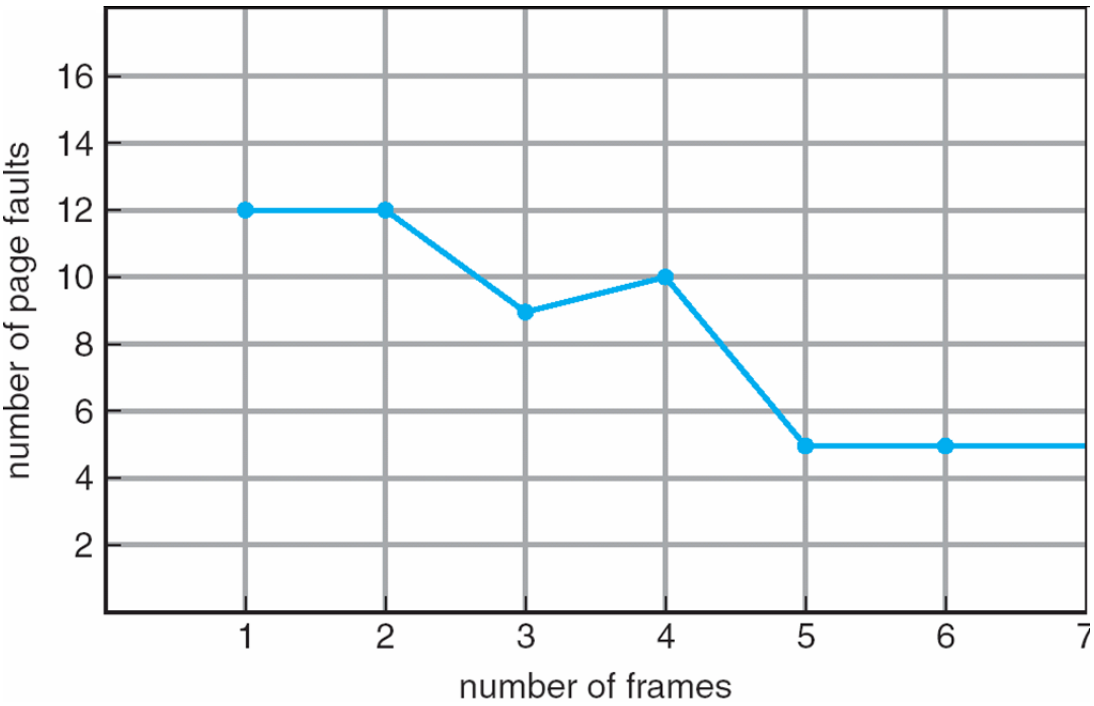
- FIFO는 Belady의 모순(Belady's anomaly) 발생
  - 프레임을 더 주었는데 오히려 페이지 부재율은 더 증가하는 현상
- 예: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3개 프레임

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

  - 4개 프레임

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

Belady 모순을 보여주는 FIFO



최적 페이지 교체 (Optimal Page Replacement)

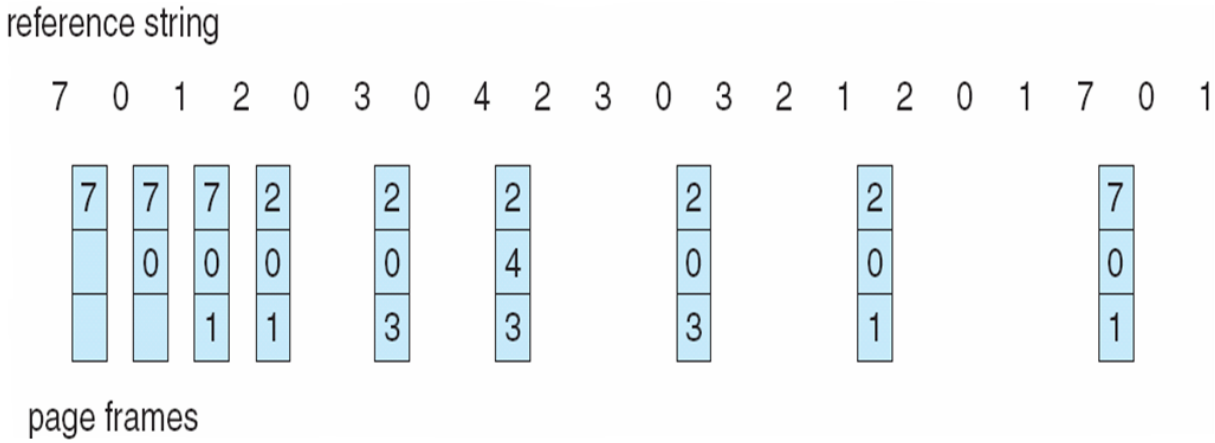
- 앞으로 가장 오랜 동안 사용되지 않을 페이지를 찾아 교체
  - 4 프레임 예
    1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

6 page faults

- 앞으로 가장 오랜 동안 사용되지 않을 페이지를 어떻게 알 수 있는가?

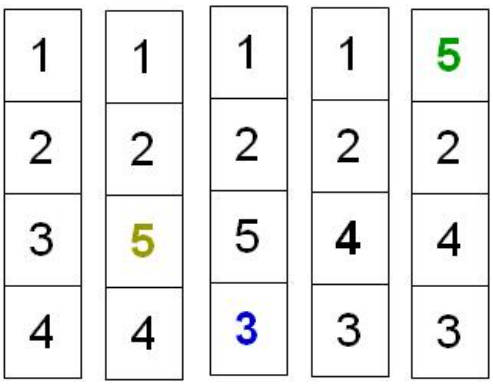
최적 페이지 교체 예



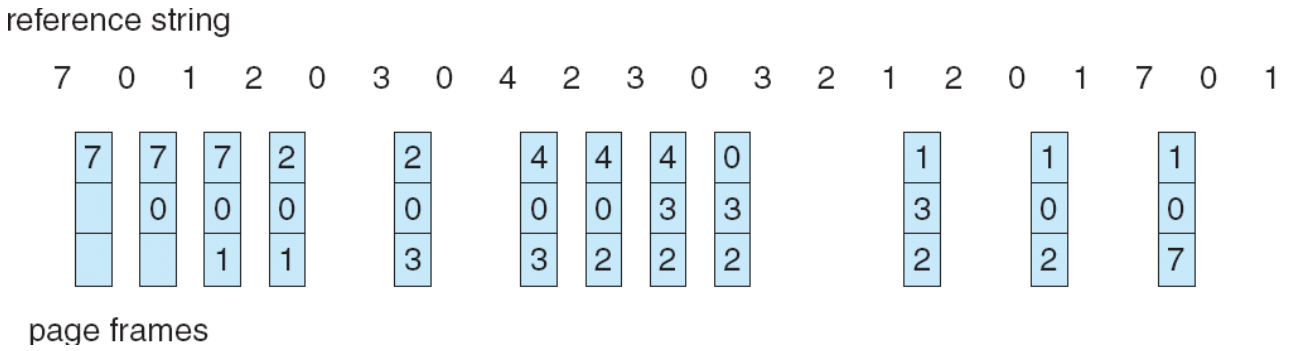
- 9개 페이지 부재 발생

LRU 페이지 교체 (LRU Page Replacement) 알고리즘

- 최근의 과거를 가까운 미래의 근사치로 추정
  - 가장 오랜 기간 동안 사용되지 않은 페이지를 교체
  - LRU (Least Recently Used)
- 참조열: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  => 8개 페이지 부재



LRU 페이지 교체 예



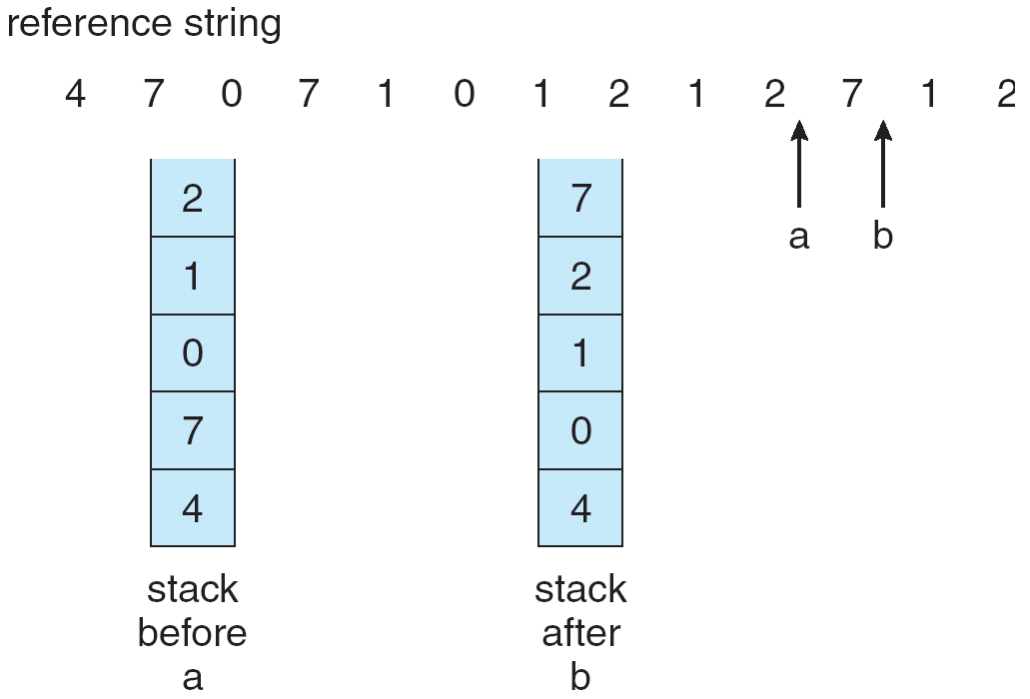
❑ 12개 페이지 부재 발생

LRU 알고리즘 구현

❑ 계수기(counter)
- 각 페이지 항목마다 계수기 추가
- 페이지 참조마다 계수기 증가
- 페이지 교체 시 가장 작은 값의 계수기의 페이지를 교체
  - LRU 페이지를 찾기 위해 페이지 테이블 검색
  - 메모리 참조마다 계수기 갱신을 위해 메모리 쓰기

❑ 스택(Stack)
- 페이지 번호의 스택을 유지
  - 스택은 보통 이중 연결 리스트로 구현
- 페이지가 참조될 때마다 페이지 번호는 스택 중간에서 제거되어 스택 꼭대기(top)에 놓임
  - 페이지 테이블 검색 필요 없음
  - 포인터 값 변경 오버헤드

LRU 스택 예



LRU 근사 페이지 교체 (LRU Approximation Page Replacement)

- LRU 교체 알고리즘은 하드웨어 오버헤드가 큼

- 근사 LRU 페이지 교체
  - 페이지 항목에 1비트 참조 비트 (reference bit) 사용
  - 참조 비트는 0으로 초기화
  - 페이지가 참조되면 참조 비트가 1로 세트
  - 참조 비트가 0인 페이지를 교체
  - 페이지가 사용된 순서는 모름

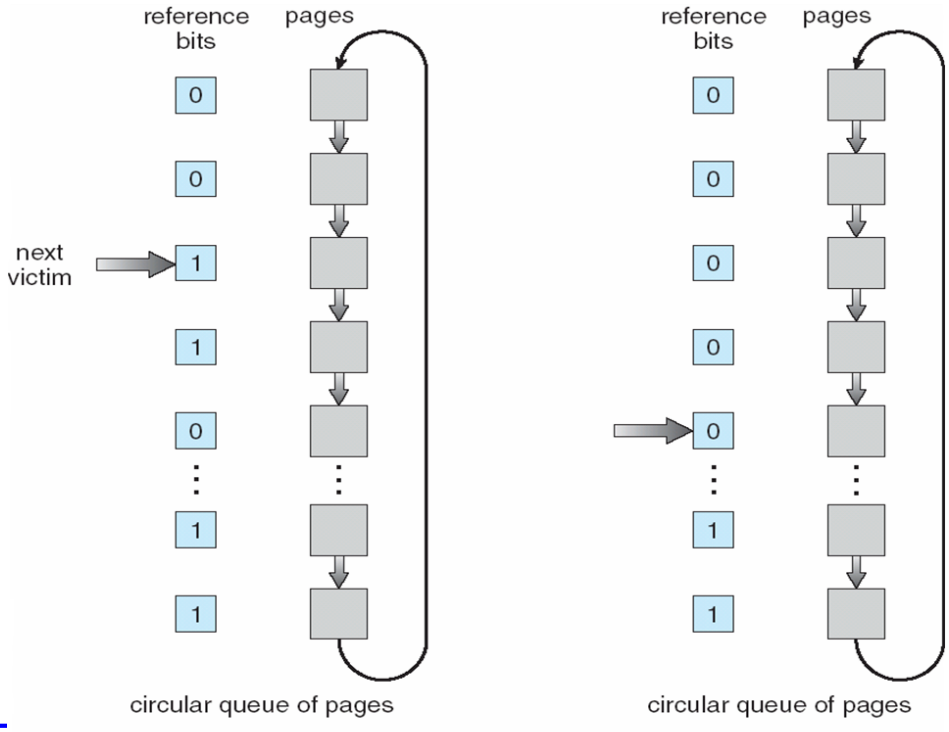
부가적 참조 비트 알고리즘 (Additional-Reference Bit Algorithm)

- 일정한 간격마다 참조 비트들을 기록함으로써 추가적인 선후 관계 정보를 얻음
- 각 페이지에 대해 8 비트의 시프터 레지스터(shift register)할당
- 일정한 시간 간격마다(예를 들면 매 100 밀리초) 타이머 인터럽트(timer interrupt)
  - 참조 비트는 8 비트 정보의 최상위 비트로 이동
  - 나머지 비트들은 하나씩 오른쪽으로 이동
  - 8 비트 시프트 레지스터는 가장 최근 8 구간 동안의 그 페이지의 사용 기록을 저장
  - 예
    - 00000000, 페이지를 한 번도 사용하지 않음
    - 11000100인 페이지는 01110111인 페이지보다 더 최근에 사용

2차 기회 알고리즘 (Second-Chance Algorithm)

- 참조 비트 사용하고 클록 교체(clock replacement)라고도 함
- 교체될 페이지를 순서대로 조사
  - 참조 비트가 1이면 0으로 세트하고 한 번 더 기회를 줌
  - 참조 비트가 0이면 해당 페이지를 교체
- 순환 큐(circular queue)를 사용하여 구현
  - 포인터(시계의 침)가 다음에 교체될 페이지를 가리킴

2차 기회 알고리즘 예



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

- (참조비트, 변경비트) 두 개의 비트를 조합하여 등급을 설정
  - (0, 0) 최근에 사용되지도 변경되지도 않은 경우
    => 교체하기 가장 좋은 페이지
  - (0, 1) 최근에 사용되지는 않았지만 변경은 된 경우
    => 이 페이지가 교체되면 디스크에 내용을 기록해야 하기 때문에 교체에 적당하지 않음
  - (1, 0) 최근에 사용은 되었으나 변경은 되지 않은 경우
    => 이 페이지는 곧 다시 사용될 가능성이 높음
  - (1, 1) 최근에 사용도 되었고 변경도 된 경우
    => 아마 곧 다시 사용될 것이며 교체되면 역시 디스크에 그 내용을 먼저 기록해야 함
- 페이지 교체가 필요할 때 클록 알고리즘과 같은 방법을 사용
  - 가장 낮은 등급을 가지면서 처음 만난 페이지를 교체

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

- 각 페이지를 참조할 때마다 계수(count)

- LFU (Least Frequently Used) 알고리즘
  - 참조 횟수가 가장 작은 페이지를 교체
  - 프로세스가 초기 단계에서는 한 페이지를 집중적으로 사용 후 다시 사용하지 않는 경우에 판단 틀림
    - 참조 횟수를 일정한 시간마다 하나씩 오른쪽으로 이동해서 지수적으로 그 영향력을 감소시켜서 해결

- MFU(Most Frequently Used) 알고리즘
  - 가장 작은 참조 횟수를 가진 페이지가 가장 최근 참조된 것이고 앞으로 사용될 것이라는 판단에 근거

페이지-버퍼링 알고리즘 (Page-Buffering Algorithms)

- 페이지 교체 알고리즘과 병행하여 적용
- 가용 프레임 여러 개를 풀(pool)에 저장 (버퍼링)
  - 페이지 부재 시 새로운 페이지를 풀로 먼저 읽음 (교체될 프레임 디스크 쓰기 전)
  - 교체될 프레임(victim)을 가용 프레임 풀에 추가
  - 편리한 시점에 교체될 프레임을 디스크에 저장

- 가용 프레임 풀은 유지하면서 프레임의 임자 페이지를 기억할 수도 있음
  - 이 프레임을 재 참조 시 디스크를 읽을 필요 없음
  - 희생될 페이지를 잘 못 선정한 경우 유용

응용과 페이지 교체

- 앞의 모든 알고리즘들은 페이지의 향 후 사용을 예측하여 동작
- 일부 응용들은 메모리 사용 동작에 대해 많은 지식을 가짐
  - 데이터베이스 응용 등
- 메모리를 많이 사용하는 범용 응용들은 이 중으로 버퍼링
  - 운영체제가 입출력 버퍼 등과 같은 메모리에 페이지를 복사
  - 응용은 자신의 작업을 위해 페이지를 메모리에 유지
- 특수한 응용들을 위해 운영체제는 디스크에 직접 접근하는 수단을 제공
  - raw disk mode
  - 파일 시스템의 요구 페이징, 파일 잠금, 버퍼링 등의 서비스를 우회 (bypass)

프레임의 할당 (Allocation of Frame)

- 각 프로세스에게 할당해야 하는 프레임 수 결정
  - 각 프로세스 당 필요한 최소의 프레임 수는 아키텍처에 의해 결정
  - 각 프로세스 당 최대 할당 프레임 수는 가용물리 메모리에 의해 결정
- 할당 알고리즘 (allocation algorithm)
  - 균등 할당 (equal allocation)
    - 모든 프로세스에게 똑같이 할당
    - 예: 5개의 프로세스, 93개 프레임 가정
      - 프로세스 당 18개 프레임 균등 할당
      - 3개는 자유 프레임 버퍼 풀
  - 비례 할당 (proportional allocation)
    - 각 프로세스의 크기 비율에 맞추어 할당

비례 할당 예와 우선순위 할당

❑ 비례 할당 예

s_i = 프로세스 p_i의 크기

$S = \sum s_i$

m = 가용 프레임의 전체 개수

a_i = 프로세스 p_i의 할당 $= \frac{s_i}{S} \times m$

$m = 62$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

❑ 우선순위 할당 (priority allocation)

- 비례 할당 방법을 사용하면서 프레임 비율을 프로세스의 크기가 아닌 우선순위를 사용하여 또는 크기와 우선순위의 조합을 사용하여 결정

전역 대 지역 할당 (Global Versus Local Allocation)

❑ 전역 교체(global replacement)

- 교체할 프레임을 다른 프로세스에 속한 프레임을 포함한 모든 프레임을 대상으로 찾는 경우
- 프로세스 실행 시간이 크게 변함
- 높은 메모리 이용율

❑ 지역 교체(local replacement)

- 각 프로세스가 자기에게 할당된 프레임들 중에서만 교체될 희생자를 선택
- 일관된 프로세스의 성능
- 낮은 메모리 이용율

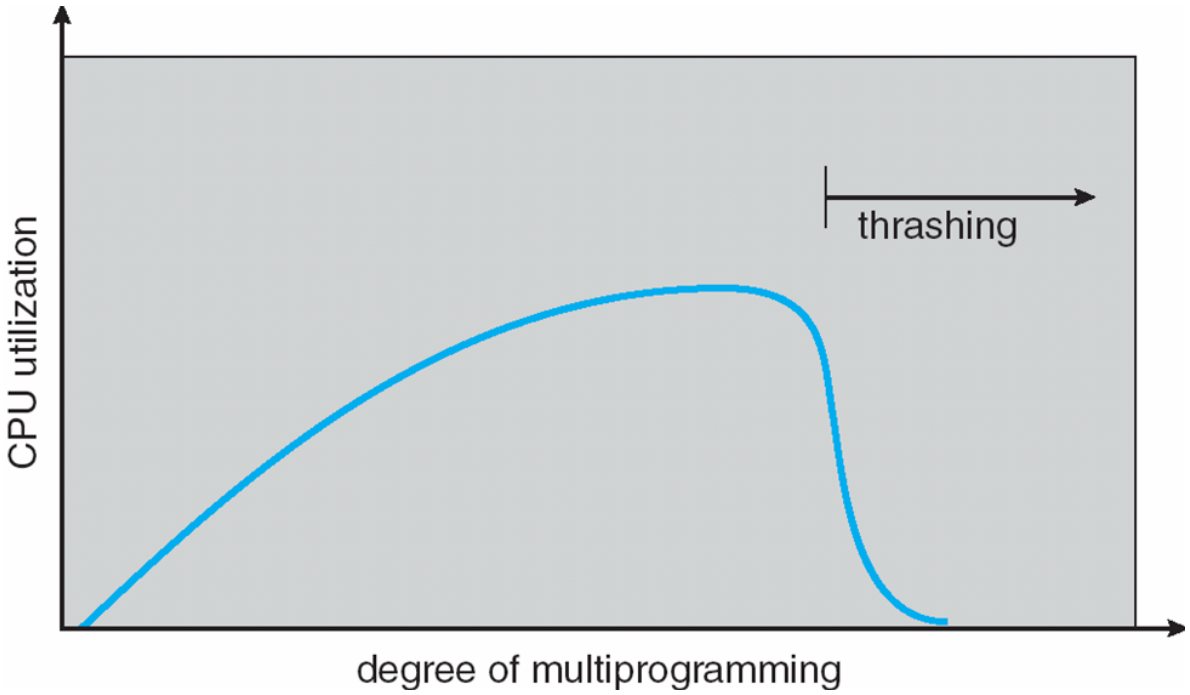
비균등 메모리 접근 (Non-Uniform Memory Access)

- 지금까지는 모든 주 메모리에 동등하게 접근한다는 가정
- 메모리 접근 시간이 차이가 나는 NUMA(Non-Uniform Memory Access) 시스템
  - CPU와 메모리가 같은 보드인 경우 접근
  - 시스템 버스로 상호 연결된 메모리에 접근
- 프로세스(스레드)가 실행 중인 CPU에 가장 가까운 메모리 프레임에 할당하는 것이 최적의 성능
  - 스케줄러가 가능하면 같은 시스템 보드 상에 스레드를 스케줄
  - Solaris는 lgroup 이라는 개체를 생성하여 해결
    - CPU/메모리 지연이 작은 그룹들을 추적
    - 프로세스의 모든 스레드와 메모리 할당을 lgroup 단위로 처리

스레싱(Thrashing) (1)

- "충분한" 프레임을 할당 받지 못한 프로세스는 빈번한 페이지 부재가 발생하여 다음 상황 초래
  - 낮은 CPU 이용률 (CPU utilization)
  - OS는 새로운 프로세스를 시스템에 더 추가(스왑 인 스왑 아웃)해서 다중 프로그래밍의 정도(degree of multiprogramming)를 높임
  - 다른 프로세스가 시스템에 추가되어 페이지 부재 빈도는 더 높아지고 CPU 이용률은 더 악화
  - 다시 프로세스를 추가 반복(스왑 인 스왑 아웃 반복)
- 스레싱 (thrashing)
  - 교체된 페이지가 얼마 지나지 않아 다시 사용되는 반복적인 페이지 부재가 발생하는 상황

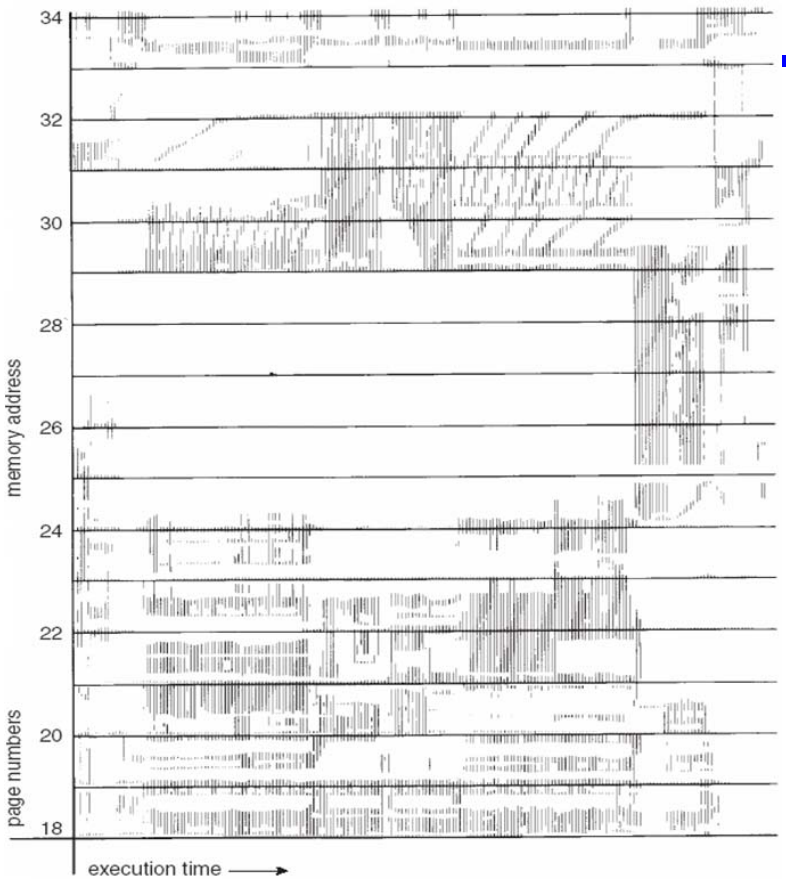
스레싱 (2)



요구 페이징과 스레싱

- 요구 페이징(demand paging)은 지역성 모델(locality model)에 기반하여 동작
  - 지역성 모델이란 프로세스가 실행될 때에는 항상 어떤 특정한 지역에서만 메모리를 집중적으로 참조함을 의미
  - 지역성(locality)이란, 집중적으로 함께 참조되는 페이지들의 집합을 의미
- 스레싱 발생 원인
  - 어떤 프로세스가 새로운 지역으로 진입할 때 필요한 크기 보다 적은 프레임을 할당하게 되면 페이지 부재가 자주 발생하여 스레싱 유발
  - Σ size of locality > total memory size
- 스레싱 방지
  - 프로세스가 필요하는 최소한의 프레임의 수 보장
  - 작업 집합(working set) 방법 사용하여 프로세스가 사용하는 프레임 수 조사

메모리 참조 패턴의 지역성

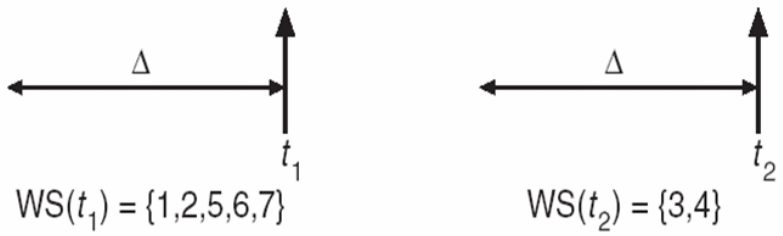


작업 집합 모델 (Working-Set Model) (1)

- 작업 집합 창(working set window), Δ
  - Δ ≡ 고정된 페이지 참조 횟수
  - 최근 Δ 만큼의 페이지 참조를 관찰
  - 한 프로세스가 최근 Δ번 페이지를 참조했다면 그 안에 들어 있는 서로 다른 페이지들의 집합을 작업 집합(working set, WS) 이라고 함
  - 예: Δ = 10

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

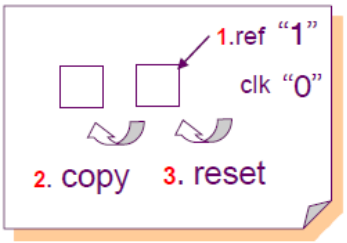


작업 집합 모델 (2)

- 작업 집합의 정확도
  - Δ 값이 너무 작으면 전체 지역성을 포함하지 못함
  - Δ 값이 너무 크면 여러 지역성을 과도하게 수용
  - Δ가 무한히 크면 프로세스가 실행중에 만난 모든 페이지의 집합 수용
- 모든 프로세스의 전체 프레임 요구량, *D*
  - $D = \Sigma WSS_i$
  - WSS_i : 프로세스 P_i의 작업 집합의 크기
- D>m 이면 스레싱
  - 요구량 D가 시스템이 보유한 총 메모리 크기 m보다 커지면 스레싱
  - m은 가용한 총 프레임 수
- D>m이면 프로세스 중 하나를 중지하고 그 프로세스의 페이지들을 다른 프로세스에게 할당하여 스레싱 예방

작업 집합 추적

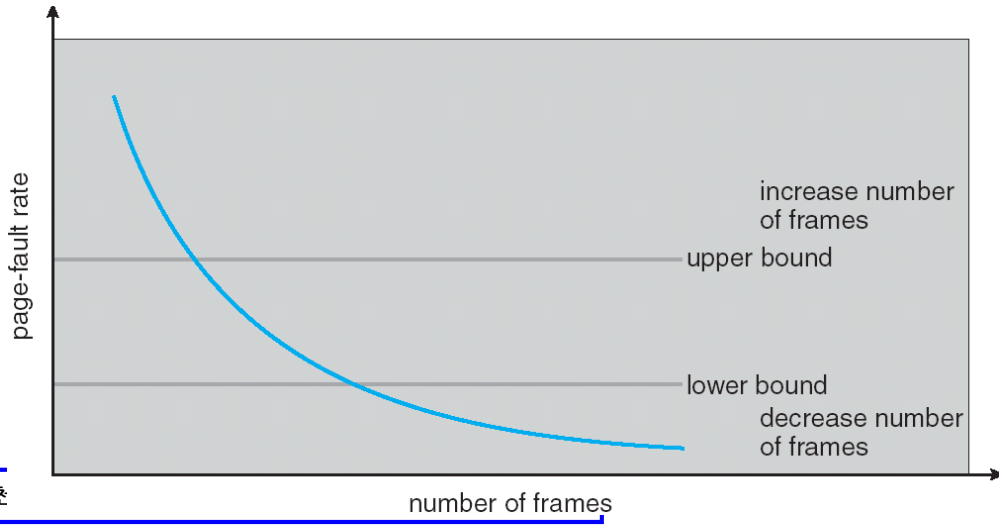
- 일정 간격 타이머 인터럽트와 참조 비트를 사용하여 근사화



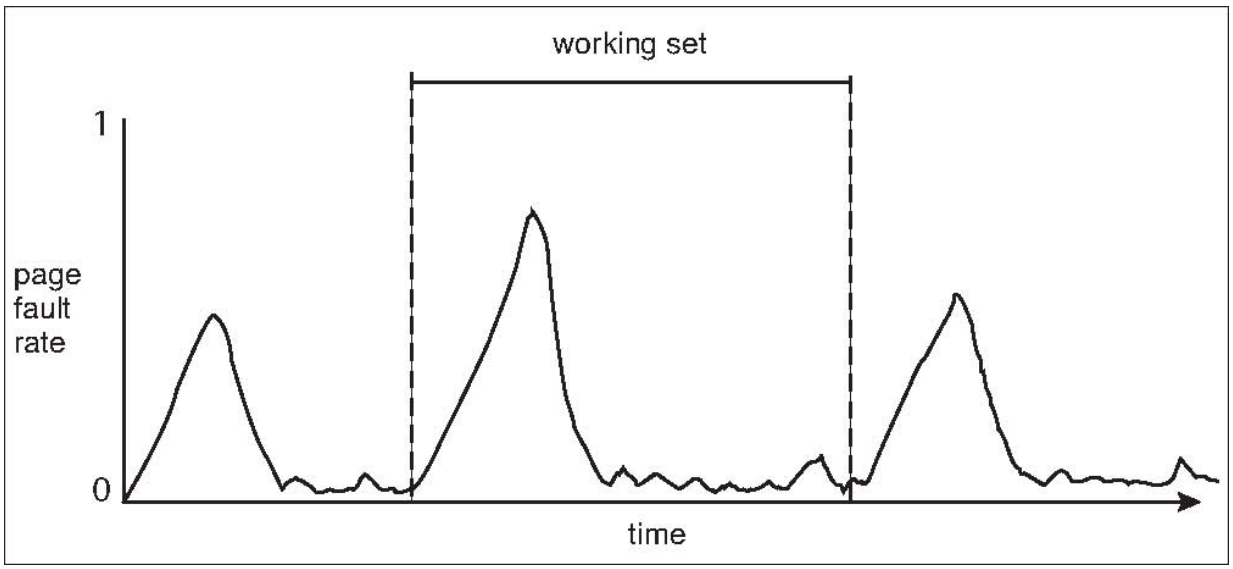
- 예: Δ = 10,000
  - 매 5,000 번의 참조마다 타이머 인터럽트
  - 각 페이지 마다 2비트 참조 비트 추가
  - 페이지 참조 시 하위 비트 1로 세트
  - 타이머 인터럽트 발생 시 모든 참조비트 하위 비트 0으로 세트하고 이전 하위 비트 값은 상위 비트에 복사(왼쪽 시프트)
  - 두 비트 중 하나가 1인 비트는 작업 집합에 속함
  - 정확도를 높이기 위해선 비트 수를 늘리고 타이머 인터럽트 빈도 수 높임
    - 10비트 참조 비트
    - 매 100번 참조 마다 인터럽트

페이지 부재 빈도 (Page-Fault Frequency)

- 페이지 부재 빈도 방식은 보다 더 직접적으로 스레싱을 조절
- 수용 가능한 페이지 부재율(page-fault rate) 설정
  - 페이지 부재율이 상한을 넘으면 그 프로세스에게 프레임을 더 할당
  - 페이지 부재율이 하한보다 낮아지면 그 프로세스의 프레임 수를 줄임



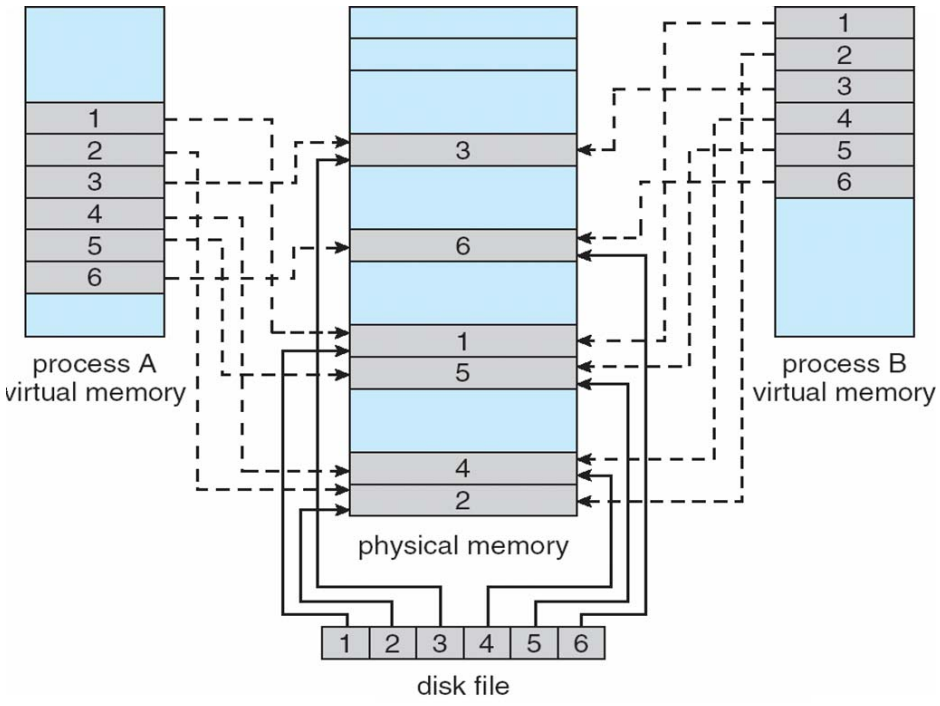
작업 집합과 페이지 부재율



메모리 사상 파일 (Memory-Mapped Files)

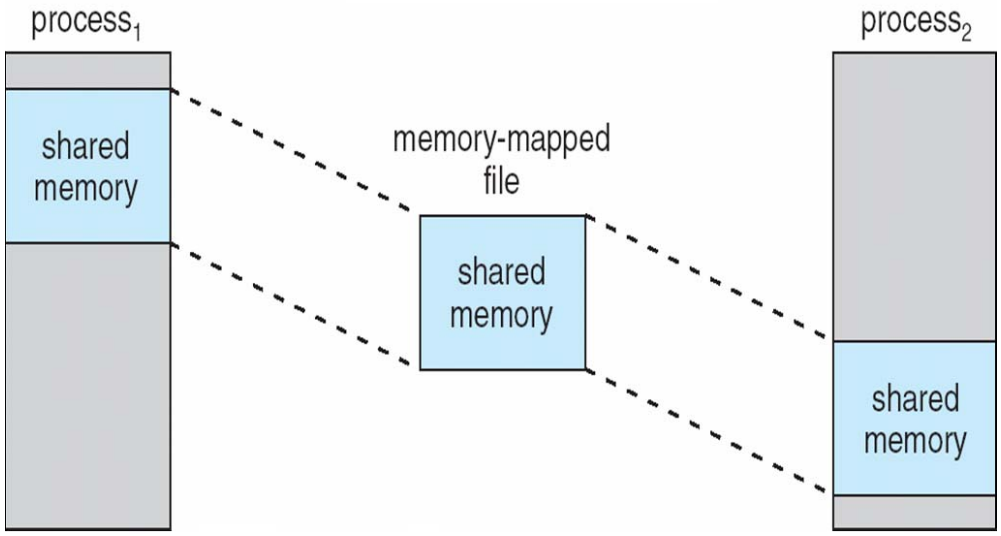
- 메모리 사상 파일 입출력은 디스크의 블록을 메모리 참조로 변환
  - 메모리 내의 페이지와 관련시켜 사상(mapping)
  - 처음 파일 접근 시 요구 페이징으로 페이지 부재가 발생하여 페이지 크기의 파일 부분이 메모리의 물리 페이지로 적재되고, 이 후 파일 읽기/쓰기는 일반 메모리 접근으로 처리
- 파일 read()/write() 시스템 호출 대신 메모리를 통해 파일 입출력을 하기 때문에 파일 접근과 사용을 단순화
- 메모리 내의 페이지들이 공유되어 여러 프로세스들이 같은 파일에 사상될 수 있음

메모리 사상 파일



메모리 사상 파일을 이용한 공유 메모리

- 윈도우 시스템에서는 공유 메모리를 메모리 사상 파일을 이용하여 구현



커널 메모리의 할당

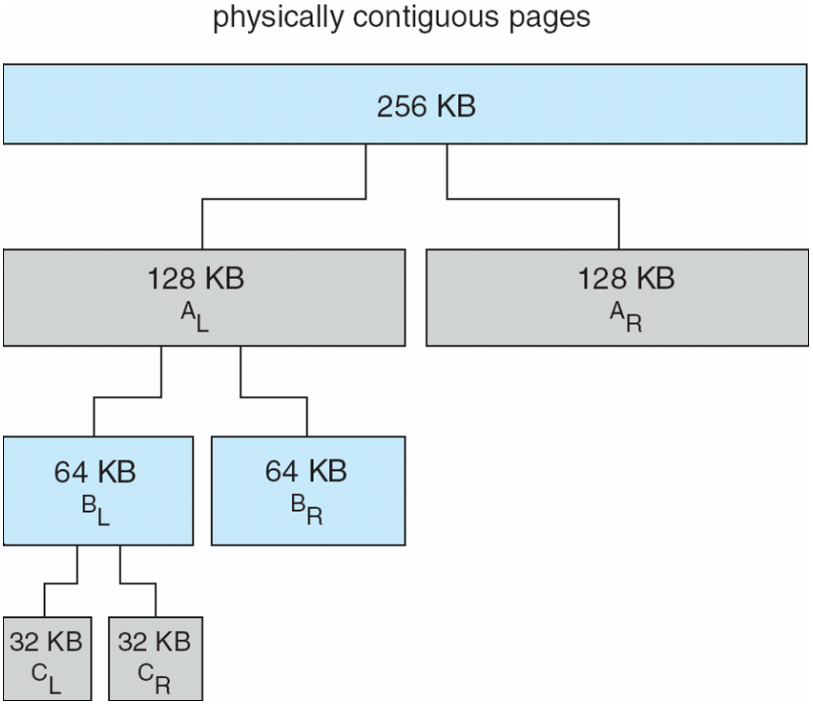
- 커널 메모리는 사용자 프로세스의 메모리와 다르게 관리
- 커널 메모리는 별도의 메모리 풀(memory pool)에서 할당 받음
  - 커널은 다양한 크기의 자료구조를 위한 메모리를 요청
    - 페이지 보다 작은 크기의 자료구조 사용
    - 많은 운영체제들이 커널 코드나 데이터에 페이징을 사용하지 않음
  - 일부 커널 메모리는 연속적인 할당이 필요
    - 물리 메모리에 직접 접근하는 하드웨어 장치

버디 시스템 (Buddy System)

- 물리적으로 연속된 페이지들로 구성된 고정된 크기의 세그먼트로부터 메모리를 할당
  - 세그먼트에서 2의 멱승(거듭제곱)의 크기 단위로 할당
  - 2의 멱승이 아닌 메모리 요구는 요구 크기 보다 큰 가장 가까운 2의 멱승 크기로 할당
  - 내부 단편화 가능성

- 예: 256KB 세그먼트에서 커널이 21KB 메모리를 요구
  - 세그먼트는 128KB 크기의 2개의 버디(buddy)로 분할
  - 128KB 버디 중 하나가 다시 2개의 64KB 버디로 분할
  - 64KB 버디 중 하나가 2개의 32KB 버디로 분할
  - 32KB 버디 중 하나가 21KB 요청을 위해 할당

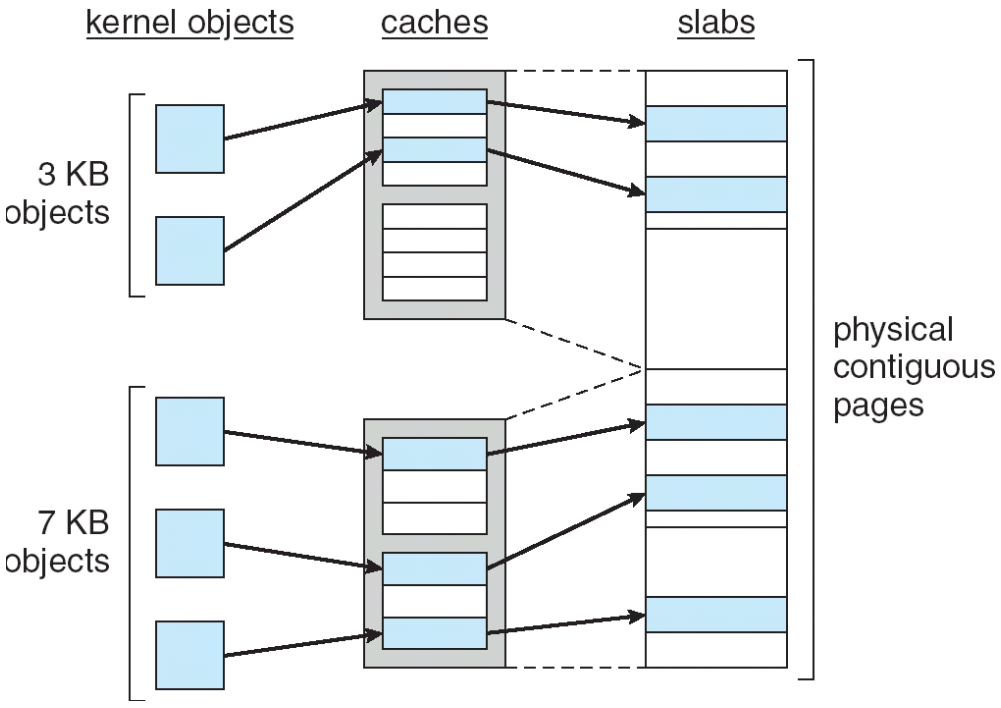
버디 시스템 할당 예



슬랩 할당 (Slab Allocation)

- 슬랩(slab)은 하나 이상의 연속된 페이지들로 구성
- 캐시(cache)는 하나 이상의 슬랩들로 구성
  - 각 커널 자료구조 마다 하나의 캐시를 가짐
    - 예를 들어 프로세스 디스크립터, 파일 객체, 세마포를 위한 캐시
  - 각 캐시는 커널 자료 구조의 인스턴스로 구성된 객체(object)가 할당
- 슬랩 할당
  - 캐시가 생성되면 초기에 free라고 표시된 객체들이 캐시에 할당
  - 커널 자료구조를 위한 객체가 필요하면 캐시 내의 free 객체 중 하나를 할당하고 used로 표시
  - 슬랩들이 모두 할당되어 사용 중이면 새로운 슬랩들을 할당
- 단편화에 의해 메모리 낭비가 없고 메모리 요청이 빠르게 처리

슬랩 할당 예



과제

❑ 연습문제

- 연습문제 10.1
- 연습문제 10.5
- 연습문제 10.9