Slide 0

3부 저장장치 관리
(10장. 가상기억장치)

배경

Slide 1

- 가상기억장치 (virtual memory) 등장 배경
  - 9장 기억장치 관리 방법
    - 목적: 다중 프로그래밍 (여러 프로세스를 동시에 수행)
    - 관리: 모든 프로세스의 프로그램 전체를 기억장치에 유지
  - 프로그램은 실행 중 모든 부분이 필요하지는 않음
    - 오류조건처리: 자주 발생하지 않음으로 대부분의 경우 필요없음
    - 배열, 리스트, 테이블: 대부분 부분적으로만 사용됨
    - 프로그램 특정 부분: 거의 사용되지 않는 부분이 있음
  - 프로그램의 전체가 한 순간에 다 필요한 것은 아님
    - 프로그램은 순차적으로 수행됨
    - 어떤 순간에 모든 부분이 필요한 것은 아님

배경 (Cont'd)

Slide 2

➥ 주기억 장치에 프로그램의 일부만 있어도 실행가능하게 함 (virtural memory)

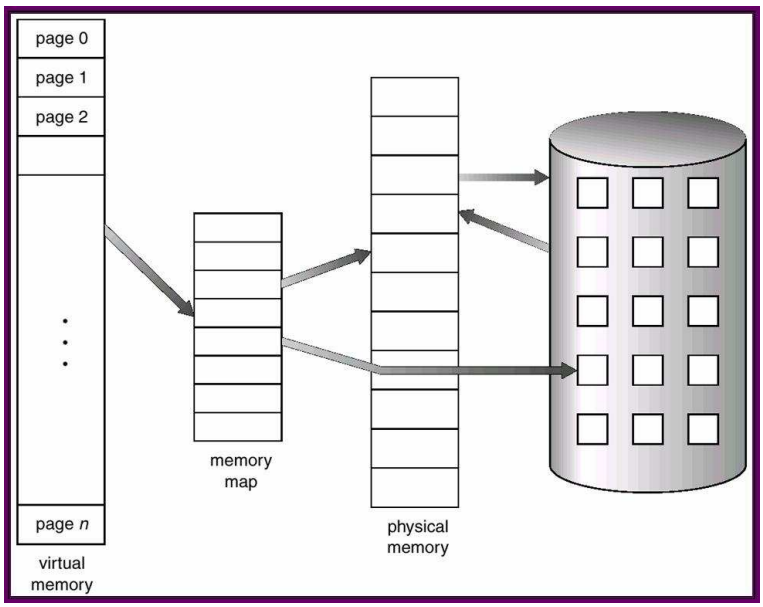
* 장점
  · 물리적 주기억장치 크기에 제한받지 않음
  · 동시에 많은 프로그램을 수행시킬 수 있음
  · 프로그램의 적재나 스왑시 입출력 비용이 적음

– 가상기억장치의 구현

* 요구 페이징 (demand paging)
* 요구 세그먼테이션 (demand segmentation)
* 페이지된 세그먼테이션 기법도 가능함 (세그먼트가 페이지로 나뉨)

배경 (Cont'd)

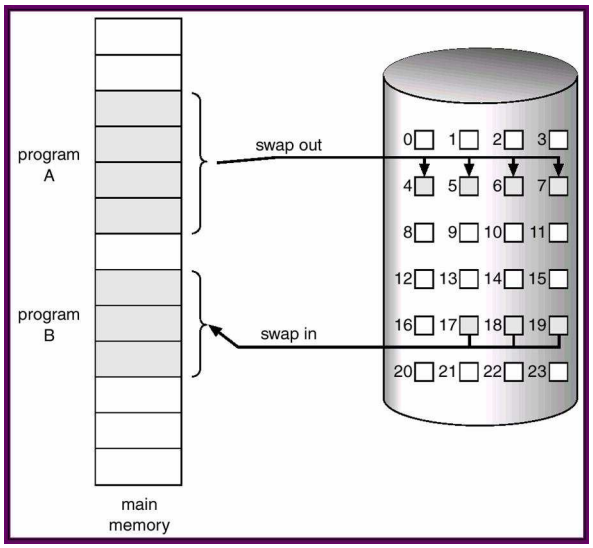
Slide 3



요구 페이징(Demand paging)

Slide 4

- 요구 페이징이란?
  - 스왑핑 기법을 사용하는 페이징 시스템



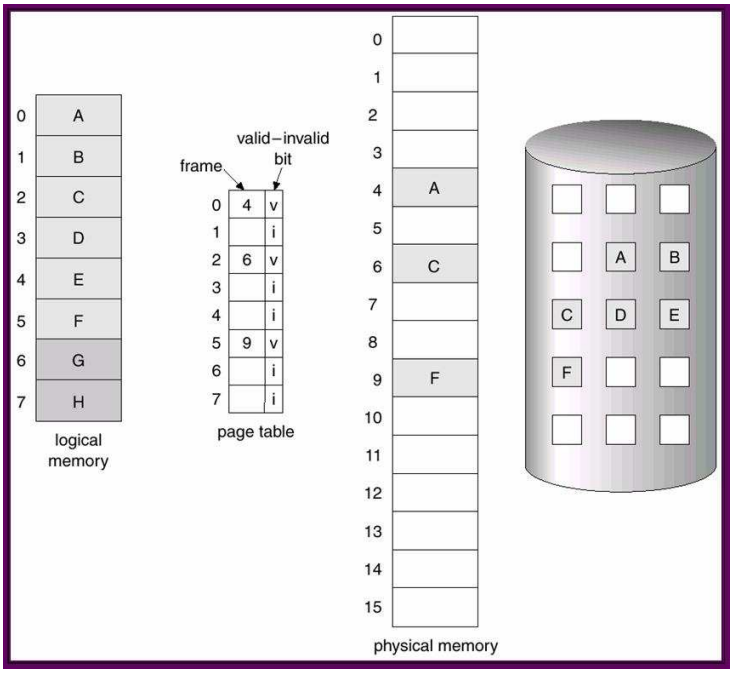
요구 페이징(Demand paging) (Cont'd)

Slide 5

- 스왑핑이 프로세스 전체를 스왑하는데 비하여 페이지 요구가 있을때 필요한 페이지만 스왑
- 지연 스왑퍼 (lazy swapper)
  - 페이지 요구가 있을때 필요한 페이지만 스왑해 줌
  - 스왑퍼(swapper)라는 용어 대신에 페이저(pager)라는 용어를 사용
- 장점
  - 사용되지 않는 페이지를 기억장치로 가져오지 않음
  - 스왑시간의 단축
  - 필요한 물리메모리 총량이 줄어듬
- 유효/무효비트 사용
  - 유효 (valid)
    - 기억장치 상에 있는 페이지와 디스크 상에 있는 페이지가 동일함
  - 무효 (invalid)
    - 그 프로세스의 논리주소 공간에 존재하지 않던지
    - 기억장치에 없고 디스크 상에 만 있을 경우

요구 페이징(Demand paging) (Cont'd)

Slide 6



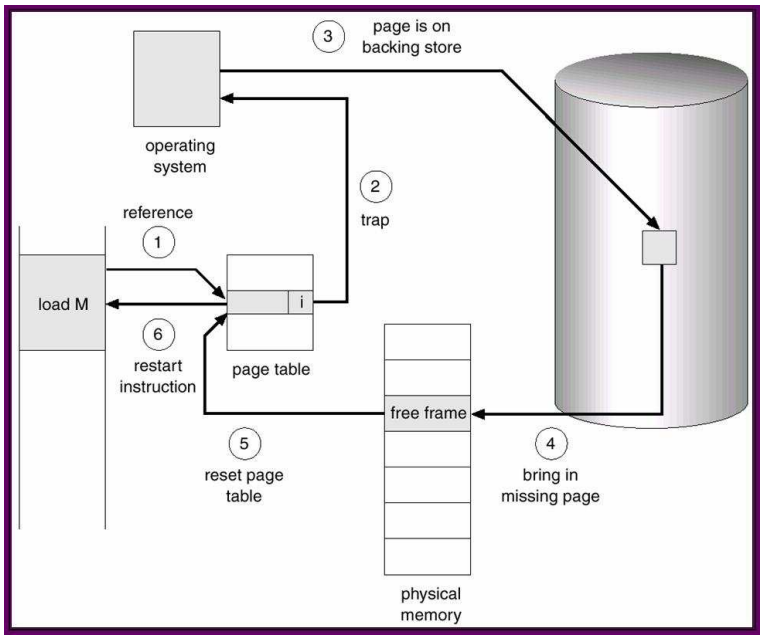
요구 페이징(Demand paging) (Cont'd)

Slide 7

- 페이지 틀림(page fault) 처리
  - 프로세스가 무효페이지를 액세스할 경우 page fault 발생
  - 페이지 틀림 트랩 (page fault trap)이 발생됨
  - 처리 순서
    - 프로세스의 가상주소 공간내의 참조인지를 점검
      (보통 PCB 내의 internal table을 검사함)
      (주소공간을 벗어난 참조인 경우 프로세스 종료함)
    - 빈 프레임(free frame)을 찾음
    - 디스크 입출력을 통하여 페이지를 프레임에 적재
    - 디스크 입출력이 완료되면 내부테이블과 페이지테이블을 수정
    - 페이지 틀림 트랩을 발생시킨 명령어를 다시 시작

요구 페이징(Demand paging) (Cont'd)

Slide 8



요구 페이징(Demand paging) (Cont'd)

Slide 9

- 순수 요구 페이징 (pure demand paging)
  - 프로세스가 처음 시작할 때 주기억장치에 유효페이지가 없는 상태로 시작
  - 초기에는 페이지 틀림이 자주 발생함
  - 참조의 지역성으로 큰 문제가 되지는 않음
- 페이지 틀림 트랩 후 명령어 재시작 문제
  - 명령어 수행 중 페이지 틀림 트랩 발생

  ➔ 페이지 틀림 트랩 처리 후 해당 명령어 재시작

  ➔ 대부분의 경우 명령어 재시작에 문제 없음
  - 명령어 수행 재시작이 원래의 명령어 수행과 다른 결과를 가져오는 경우

    ➥문제 발생
    * 예)
    * 다량의 문자이동 명령어수행 중 트랩 발생
    * 명령어 수행으로 source block 의 내용이 변경 (destination block 과 겹쳐서)
    * 재시작하면 변경된 source block을 사용하므로 문제 발생

요구 페이징(Demand paging) (Cont'd)

Slide 10

- 해결 방법
  - 방법 1
    - 마이크로코드에서 이동을 하기 전에
    - 모든 필요한 페이지를 접속해서 미리 페이지 틀림을 발생하게 함
    - 실제 이동이 일어날 때는 필요한 모든 페이지가 메모리에 있게 함
  - 방법 2
    - 임시 레지스터를 사용
    - 명령어 수행 중 변경되는 모든 메모리 값을 저장하고 있다가
    - 페이지 틀림 트랩이 일어나면 원래의 메모리 값으로 복귀 (명령어 수행 전의 메모리 상태로)
    - 트랩 처리후 명령어 재시작

요구 페이징(Demand paging) (Cont'd)

Slide 11

- 요구페이징의 성능
  - 평균 접근 시간 (effective access time): $(1-p) \times ma + p \times pft$
    - p: 페이지 틀림 확률
    - ma: 주기억장치 접근시간 (보통 10 ~200 ns)
    - pft(page fault time): 페이지 틀림 처리 총 소요시간
  - pft
    - 주요 필요한 작업
      - 페이지 틀림 인터럽트 처리 (μs 단위)
      - 디스크에 있는 페이지를 주기억장치로 적재하는데 시간 (25ms 정도)
      - 중단된 프로세스의 재개 (디스크 입출력을 위해 대기중임, 다시 실행될 시간 예측은 힘듦)
    - 대기 시간을 제외하면 pft는 페이지 틀림 확률에 비례함
      - p=0.001 인 경우 페이지 틀림이 없을때 보다 250배 정도 느려짐
      - p를 가능한한 낮추어야함

프로세스 생성

Slide 12

- Copy-on-Write
  - fork() 에 의하여 새로운 프로세스 생성시
  - 부모 프로세스의 주소공간이 자식 프로세스에게 복사 (copy) 됨
  - Copy-on-Write(COW) 방법
    * 이때 자식 프로세서 생성시 바로 복사하게 하지 않고
    * 초기에는 부모 프로세서의 주소공간을 공유하게 함
    * 두 프로세스 (부모, 자식) 중 어느 하나가 페이지를 변경하면 (Write)
    * 그 때 복사(Copy) 함
  - 장점
    * 빠른 프로세스 생성이 가능
    * 자식 프로세스가 생성후 바로 exec 를 호출하면 주소공간 복사는 낭비
    * 데이터가 아닌 코드는 변경하지 않음으로 공유가능 (효율증가)
  - 적용 운영체제: 윈도우2000, 리눅스, 솔라리스 2

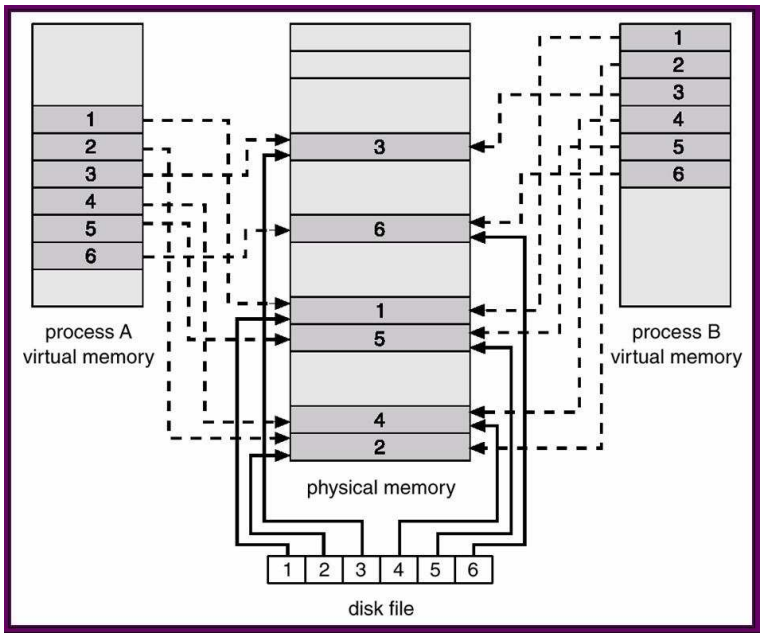
프로세스 생성 (Cont'd)

Slide 13

- 기억장치-사상 파일
  - 디스크에 있는 파일 접속은 open(), read(), write()와 같은 시스템 호출을 통함
  - 기억장치-사상 파일이란
    * 파일 작업을 빠르게 하기 위해 파일을 기억장치에 사상하여 다루는 방법
    * 처음에는 요구페이징으로 기억장치로 읽어들이고
    * 이후에는 기억장치 접근을 통해 파일을 다룸
    * 파일 내용 변경도 기억장치 상에서 이루어짐
    * 파일을 close하면 디스크로 쓰여짐
  - 여러 프로세스간 공유
    * 각 프로세스의 가상주소를 같은 물리 기억장치 프레임으로 매핑하여 구현
    * 읽기전용은 공유에 문제가 없음
    * 읽기쓰기라도 COW 방법을 사용하여 실제로 변경될 때 새로 copy하는 방법이 사용

프로세스 생성 (Cont'd)

Slide 14



페이지 교체

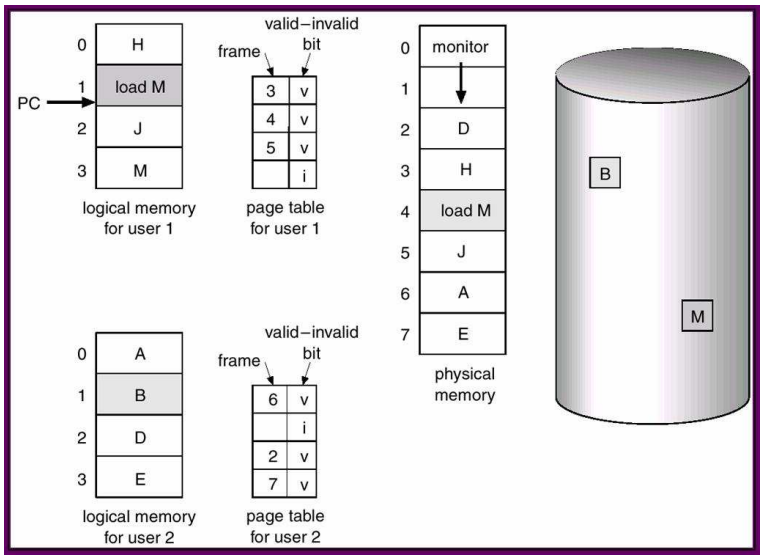
Slide 15

- 페이지 교체가 필요한 이유
  - 요구페이징에서 각 프로세스는 필요한 페이지를
  - 페이지 틀림 처리를 통하여 디스크에서 물리 기억장치로 가져 옮
  - 기억장치의 모든 프레임이 유효한 페이지로 할당된 후
  - 새로운 프로세스가 시작되면
  - 이 프로세스가 필요로 하는 페이지를 저장하기 위한 빈 프레임이 없음
  - 이를 처리하기 위한 가능한 방법
    - 해당 프로세스를 시작하지 못하게 하거나 종료함 (좋지 않은 방법)
    - 한 프로세스를 선정하여 스왑아웃 (때로는 필요한 방법)
    - 유효한 페이지 중에서 선정하여 새 페이지로 교체
  - 페이지 교체 알고리즘은 페이지 중에서 적당한 페이지를 선정함

페이지 교체 (Cont'd)

Slide 16

- 페이지 교체의 필요성



페이지 교체 (Cont'd)

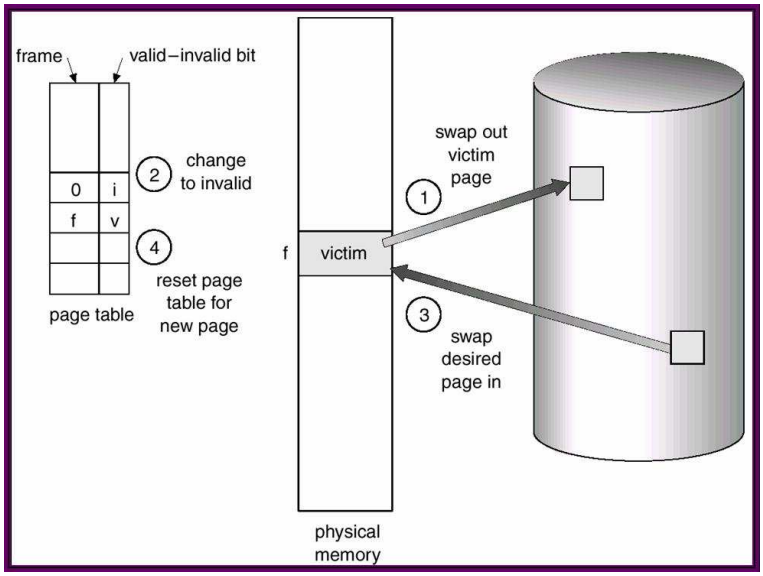
Slide 17

- 페이지 교체가 있는 페이지 틀림 처리
  - * 디스크에서 요구한 페이지 위치를 알아냄
  - * 빈 프레임을 찾음
    - · 있으면 사용
    - · 없으면 페이지 교체 알고리즘으로 희생 프레임을 선정
    - · 희생 프레임은 디스크에 기록 후 페이지와 프레임 테이블 수정
  - * 희생 프레임에 새 프레임을 읽어들이고 페이지와 프레임 테이블 수정
  - * 프로세스 재시작

페이지 교체 (Cont'd)

Slide 18

– 페이지 교체



페이지 교체 (Cont'd)

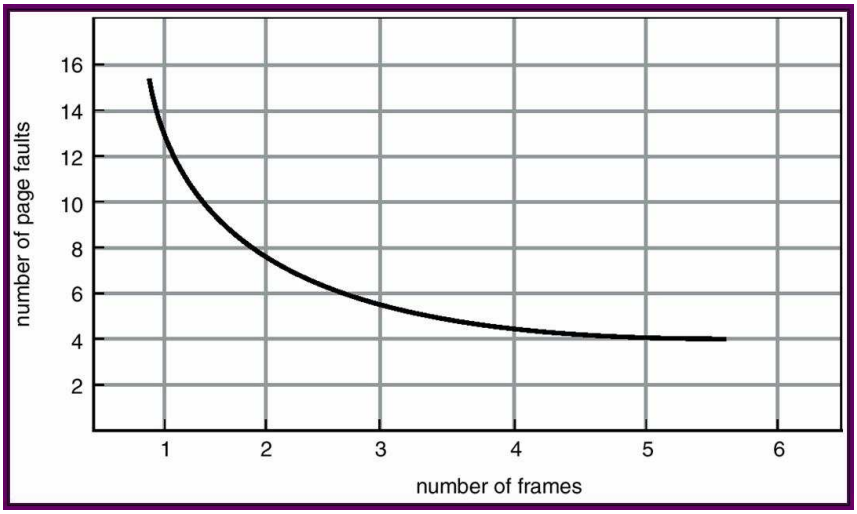
Slide 19

- 효과적인 페이지 교체
  - 희생 프레임 디스크 기록 + 새 프레임 기억장치로 적재
  - 2번의 디스크 동작으로 페이지 틀림 처리 시간이 길어짐
  - ➥ 변경비트 (modify bit or dirty bit)를 사용하여 문제를 해결
  - 변경비트가 설정되지 않은 희생 프레임은 디스크에 기록하지 않아도 됨
  - 읽기전용의 경우는 항상 디스크에 기록할 필요가 없음
- 두가지 필요 알고리즘
  - 프레임 할당 알고리즘(frame-allocation algorithm)
    - 한 프로세스에게 할당할 프레임의 수를 결정하는 알고리즘
  - 페이지 교체 알고리즘(page-replacement algorithm)
    - 빈 프레임이 없을 때 희생 프레임을 선택하는 알고리즘

페이지 교체 (Cont'd)

Slide 20

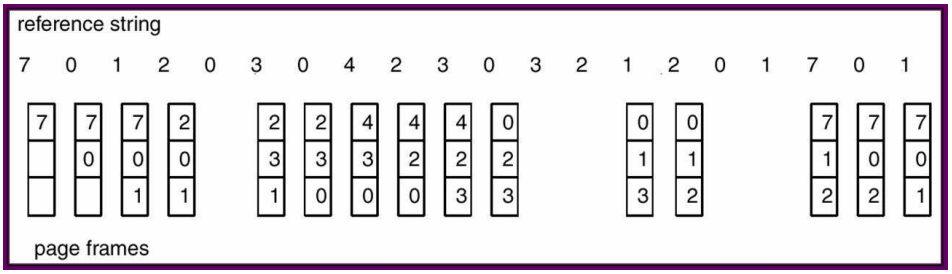
– 프레임 수에 따른 페이지 틀림 횟수



페이지 교체 알고리즘

Slide 21

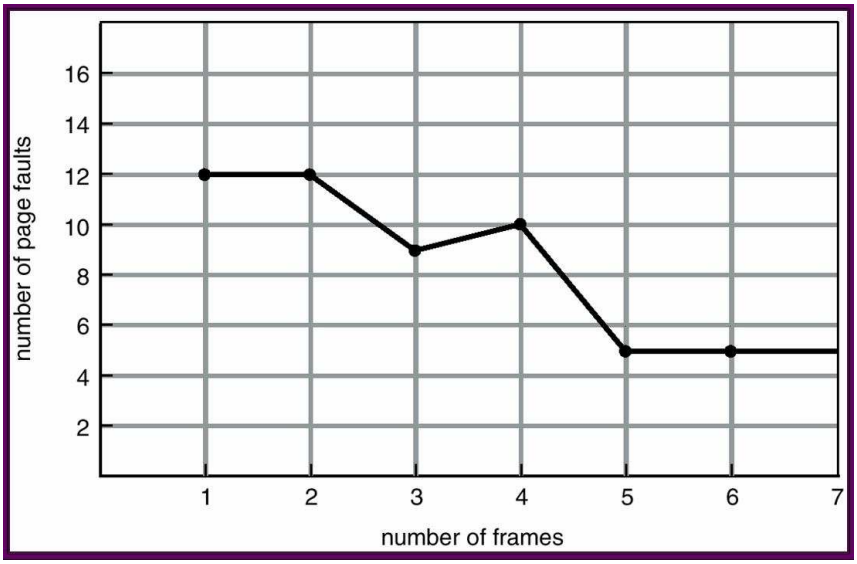
- 설명
  - 페이지 틀림 발생 확률이 가장 낮은 알고리즘을 선정
  - 기억장치에 대한 참조열을 이용하여 알고리즘 평가
    (참조열은 임의로 만들거나 실제 시스템을 추적하여 만듦)
- 선입선출 (FIFO) 페이지 교체 알고리즘
  - 가장 먼저들어온 것을 먼저 교체
  - 들어온 시간을 기록하지 않고 FIFO 큐 사용해도 됨
  - 가장 간단하고 구현하기 쉬운 방법



페이지 교체 알고리즘 (Cont'd)

Slide 22

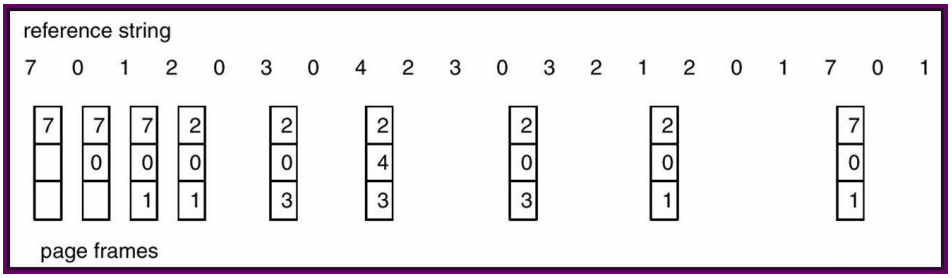
- Belady 이상현상: 프레임 개수가 커져도 페이지 틀림 횟수가 증가하는 현상



페이지 교체 알고리즘 (Cont'd)

Slide 23

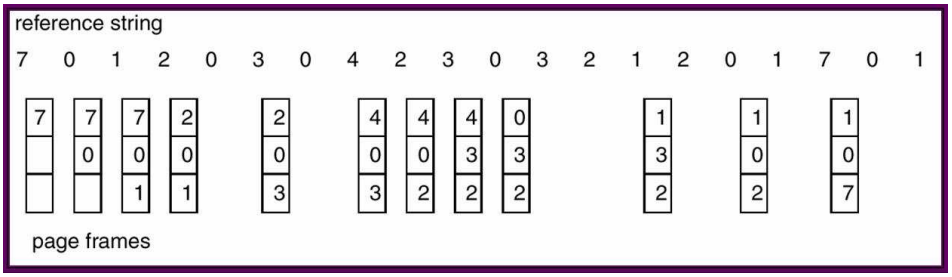
- 최적(Optimal) 페이지 교체 알고리즘
  - 가장 오랫동안 사용되지 않을 페이지를 먼저 교체
  - 페이지 틀림 확률이 가장 낮음 (최적의 알고리즘임)
  - 문제점
    - 미래의 참조를 예측하기가 어려움
    - 주로 최적 값을 계산할때 만 사용



페이지 교체 알고리즘 (Cont'd)

Slide 24

- LRU 페이지 교체 알고리즘
  - LRU(Least Recently Used) 가장 오랫동안 사용되지 않은 페이지를 먼저 교체
  - 가장 널리 사용되는 방법
  - 하드웨어 지원이 필수적임
    (소프트웨어로 하면 10배정도 기억장치 참조가 느려짐)



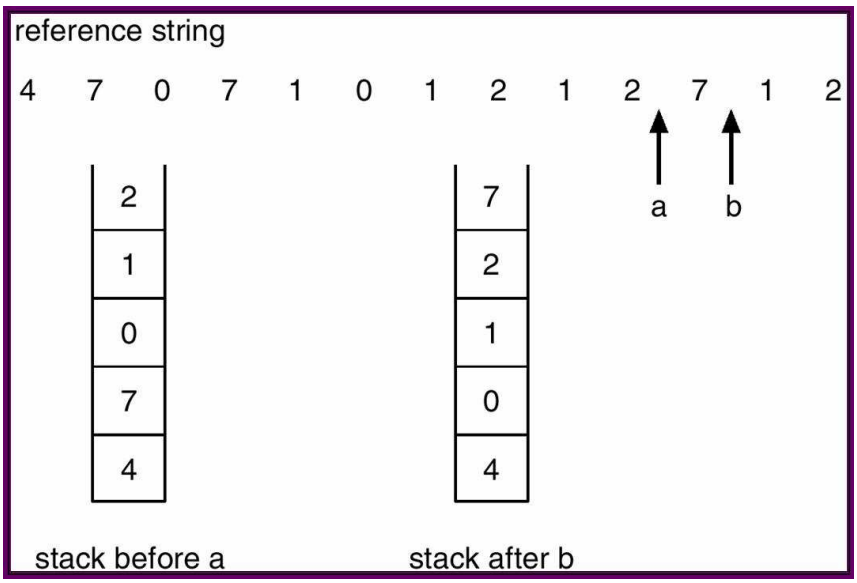
페이지 교체 알고리즘 (Cont'd)

Slide 25

- 구현 방법
  * 카운터사용
    · 페이지 참조 시마다 카운터 값 증가
    · 참조되는 페이지의 페이지 테이블 항목에 카운터 값을 저장
    · 이 값이 가장 적은 것을 선택하면 됨
    · 문제점
      - 가장 적은 것을 찾기 위해 페이지 테이블을 검색해야함
      - 기억장치 접근 때마다 페이지테이블에 기록
      - 카운터 클럭의 오버플로우도 고려해야함
  * 스택
    · 페이지 참조가 된 페이지는 스택에서 제거되어 top 에 놓임
    · 스택의 가장 밑에 있는 것이 가장 오래동안 사용되지 않은 것임
    · 스택 중간에 있는 페이지가 참조되면 중간에서 제거후 top 에 놓임
    (이중 연결 리스트로 구현해야함)
    · 최악의 경우 리스트의 6개의 포인터를 변경해야하나
    · 탐색시간이 필요하지 않는 장점이 있음

페이지 교체 알고리즘 (Cont'd)

Slide 26



페이지 교체 알고리즘 (Cont'd)

Slide 27

- LRU 근접 페이지 교체 알고리즘
  - LRU 를 제공하기에 충분한 하드웨어 지원이 없을 경우 사용
  - 참조비트
    - 참조된 페이지의 참조비트를 1로 설정
    - 참조비트로 참조된 페이지와 참조되지 않은 페이지로 구분 가능 (단, 참조된 순서는 모름)
  - 추가 참조 비트 알고리즘
    - 참조비트 + 8비트 추가 참조 비트
    - 정기적으로 운영체제가 제어를 획득 (보통 100ms 마다)
    - 이 때마다 참조비트를 추가참조비트의 msb로 추가참조비트는 오른쪽으로 시프트
    - 추가 참조 비트 값이 가장 작은 것 ➙가장 오래된 것
    - 단, 같은 값이 여러 개 존재 가능
      - 같은 값 모두를 교체하거나 그들 가운데서 FIFO 적용

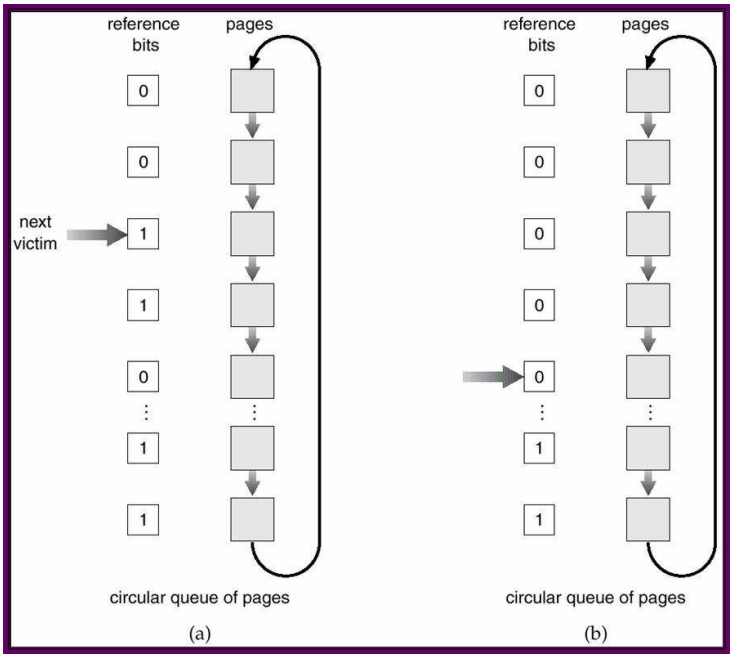
페이지 교체 알고리즘 (Cont'd)

Slide 28

- 2차 기회 알고리즘
  - 기본적으로 FIFO 교체 개념
  - 참조비트 조사
    - 만약 0이면 그 페이지 교체
    - 만약 1이면 2차 기회를 주고 0으로 바꾸고 계속 검색
    - 참조비트는 주기적으로 0으로 바꿈

페이지 교체 알고리즘 (Cont'd)

Slide 29



페이지 교체 알고리즘 (Cont'd)

Slide 30

- 2차 기회 개선 알고리즘
  - 참조비트와 더불어 수정비트 사용
  - 가능한 조합
    - 1등급 ➙(0,0): 참조X, 수정X
    - 2등급 ➙(0,1): 참조X, 수정O(교체 전 기록되어야함 →교체 적당X)
    - 3등급 ➙(1,0): 참조O, 수정X(곧 다시 사용될 것임)
    - 4등급 ➙(1,1): 참조O, 수정O(곧 다시 사용될 것이고 교체 전 기록되어야함)
  - 가장 낮은 등급 중 처음 만난 페이지 교체
  - 단점: 검색시간이 길어짐

페이지 교체 알고리즘 (Cont'd)

Slide 31

- 계수기반 페이지 교체 알고리즘
  - 참조된 횟수를 기반으로 하는 알고리즘
  - LFU (least Frequently Used)
    - 참조가 가장 적은 페이지를 교체
    - 문제점
      - 초기에 많이 사용되었으나 더이상 사용되지 않는 페이지가 계속 남음 (주기적으로 횟수를 감소시켜서 해결 가능)
      - 처음으로 교체되어 들어온 페이지는 바로 교체 될 수 있음
  - MFU (Most Frequently Used)
    - 참조가 많은 페이지를 교체
    - 참조가 많은 것이 오래되었음을 가정
  - 두 방법 모두 효과적이지 못하며 잘 사용되지 않음

페이지 교체 알고리즘 (Cont'd)

Slide 32

- 페이지 버퍼링 알고리즘
  - 페이지 교체 알고리즘과 함께 사용되는 기법
  - 빈 프레임 풀의 유지
    * 고정된 수의 빈 프레임을 유지
    * 페이지 틀림 발생
    * 희생자 프레임 선택
    * 새 페이지를 희생자 프레임이 아닌 빈 프레임에 적재
      (페이지 틀림 발생 프로세스의 처리가 빨라짐)
      (희생자 프레임이 변경되었을 경우 디스크로 적재하는 과정이 줄어들어서)
    * 후에 희생자 프레임을 디스크로 쓰고 이 프레임을 빈 프레임으로 유지
  - 변경 페이지 목록 유지
    * 변경 페이지 목록을 유지하고 있다가
    * 페이징 장치가 유휴상태에 변경 페이지를 디스크에 씀 (변경 비트 조정)
    * 교체할 페이지가 변경비트 설정으로 디스크에 써야할 확률을 줄여줌
      ➥페이지 틀림 처리 속도를 향상 시킴

프레임의 할당

Slide 33

- 프로세스에게 프레임을 할당하는 방법
  - 보통 운영체제가 일부를 사용하고 나머지는 사용자 프로세스에게 할당
  - 변형된 형태로 빈 프레임을 특정 개수만큼 유지하면서 프레임을 할당
- 최소 프레임 수
  - 한 프로세스에게 할당해야하는 최소의 프레임 수
  - 해당 프로세서의 명령어 집합 구조 (instruction set architecture)에 의존함
  - 즉, 하나의 명령이 수행될 때 참조 가능한 페이지 수만큼은 할당되어야함
  - 예)
    * 명령어와 직접주소데이터: 명령어 + 데이터 (최소 2개)
    * 명령어와 간접주소데이터: 명령어 + 간접주소 + 데이터 (최소 3개)
    * 명령어와 데이터가 페이지에 걸쳐있을 경우: 3개 × 2 (최소 6개)
    * 다단계 간접 참조시는 더 늘어남 (다 단계 간접 참조의 최대 수를 제한)
  - 최대 프레임 수는 유효 실제 기억장소의 양으로 정의
  - 최소와 최대 사이에 적당한 개수로 프레임을 할당해야함

프레임의 할당 (Cont'd)

Slide 34

- 할당 알고리즘
  - 균등 할당 (equal allocation)
    - m개의 프레임을 n개의 프로세스에게 할당시
    - 각 프로세스에게 m/n 만큼 할당
  - 비례 할당 (proportional allocation)
    - 각 프로세스의 크기에 비례하여 할당
    - 프로세스 P_i에게 a_i 만큼 할당

      $a_i = s_i/S \times m$

      단,

      s_i 는 P_i의 크기

      $S = \sum s_i$

      m 은 가용 프레임 총수
  - 다중 프로그래밍의 정도에 따라 바뀜
  - 우선순위에 기반하여 할당하는 방법도 있음

프레임의 할당 (Cont'd)

Slide 35

- 전역할당 과 지역할당
  - 다중 프로세스가 존재할 때
  - 지역할당
    - 페이지 틀림시 해당 프로세스에게 할당된 프레임 내에서 페이지 교체를 함
    - 한 프로세스가 점유하는 페이지 총수에 변함이 없음
    - 외부적 환경요인(현재 실행중인 프로세스의 개수등)에 영향을 받지 않음
    - 한 프로세스에서 자주 사용되지 않는 페이지를 다른 프로세스에게 할당할 수 없는 큰 문제가 있음
  - 전역할당
    - 페이지 틀림시 다른 프로세스에게 할당된 프레임도 페이지 교체 대상이 됨
    - 우선순위가 있을 경우 우선순위가 낮은 프로세스의 페이지를 교체 대상으로 할 수 있음
    - 각 프로세스에게 할당된 페이지의 개수가 동적임
      (즉 외부적 환경요인에 영향을 받음)
    - 프로세스별 페이지틀림 확률을 제어할 수 없음
  - 지역교체의 큰 문제로 인하여 전역교체가 보다 일반적으로 사용됨

스레싱(Thrashing)

Slide 36

- 스레싱이란?
  - 프로세스에게 할당된 프레임의 수가 적어서 계속적으로 페이지 틀림이 발생하는 현상
  - 보통 프로세스의 실행시간 < 페이지 틀림 처리시간일때 발생하였다함
  - 스레싱이 발생하면 심각한 성능저하가 일어남
- 발생원인
  - CPU 이용률의 향상을 위해 다중 프로그래밍 정도를 높임

  ➟ 새 프로세스의 페이지 틀림 증가 ➟다른 프로세스의 페이지로 교체

  ➟ 다른 프로세스의 페이지 틀림 증가 ➟페이지 교체로 입출력 대기 증가

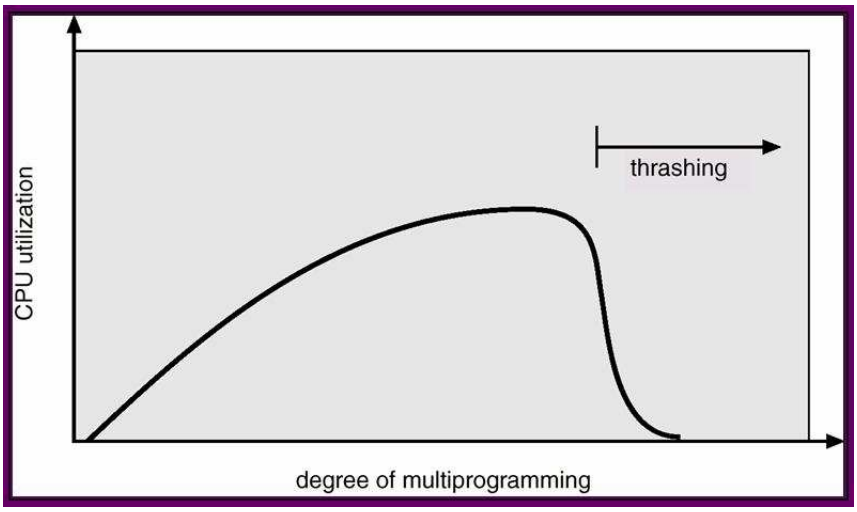
  ➟ 준비 큐 프로세스 수 감소 ➟CPU 이용률 하락

  ➟ 다중 프로그래밍 정도를 높임 ➟점점더 페이지 틀림 증가

  ➥ 스레싱 발생

스레싱(Thrashing) (Cont'd)

Slide 37



스레싱(Thrashing) (Cont'd)

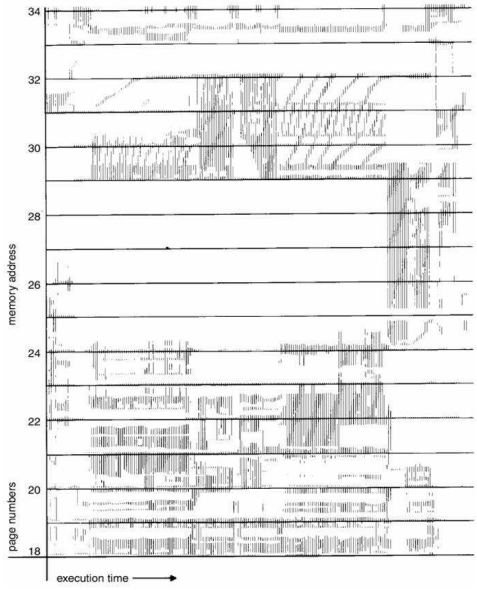
Slide 38

- 해결 방법
  - 지역 교체 알고리즘을 사용하면 스레싱 효과를 줄임
    (한 프로세스의 스레싱이 다른 프로세스로 전파되지 않음)
  - 스레싱 발생을 예방
    - 이를 위해 각 프로세스가 필요로하는 만큼 프레임을 할당해야함
    - 프로세스가 필요한 프레임의 수를 어떻게 아는가가 문제
    - 지역성 모델 (locality model)
      - 프로세스가 실행되는 동안 필요로하는 페이지는 지역적 특성을 보임
      - 지역이란 현재 활동적으로 사용하고 있는 페이지의 집합을 말함
    - 프로세스의 지역을 수용할 수 있을 정도로 프레임을 할당해야함

스레싱(Thrashing) (Cont'd)

Slide 39

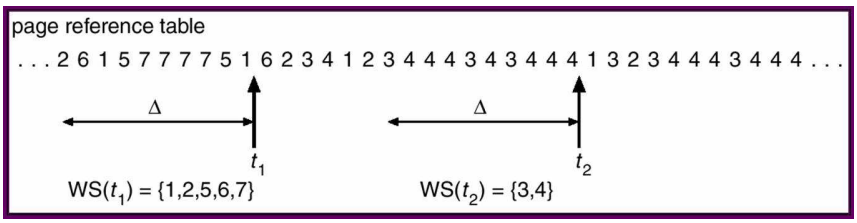
- 기억장치 참조의 지역성



스레싱(Thrashing) (Cont'd)

Slide 40

- 작업집합 모델(working set model)
  - △: 가장 최근에 참조한 페이지의 갯수
  - WS(): working set으로 △ 기간에 참조한 페이지의 집합
  - 예) $\triangle = 10$

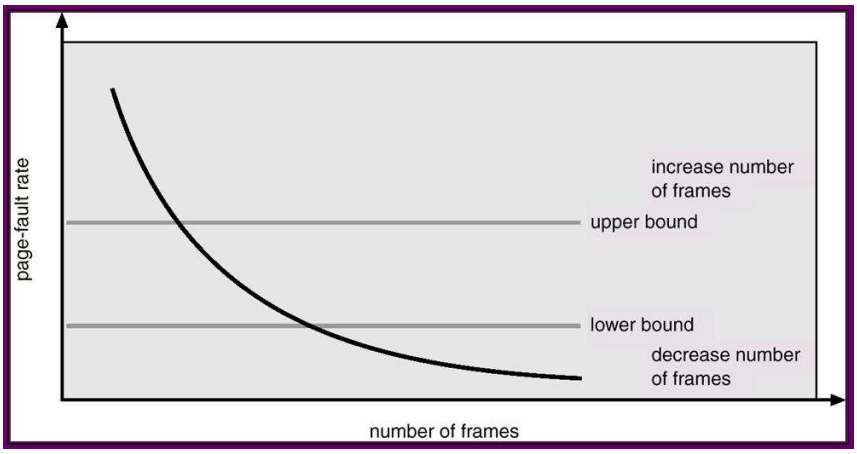


  - 전체 요구 프레임 $D = \sum WSS_i$, WSS_i는 P_i의 working set
    $D > m$ ➟스레싱 발생 (이 경우 하나의 프로세스를 중단 시킴)
  - 프로세스별 작업집합을 유지해야하는 어려움이 있음

스레싱(Thrashing) (Cont'd)

Slide 41

- 페이지 틀림 빈도
  - 프로세스별로 페이지 틀림 빈도를 측정하여 프레임 수를 조절하는 방법
  - 페이지 틀림 상한과 하한을 정의하여 조절
  - 빈 프레임이 없는 경우에는 프로세스를 선택해 중지함



운영체제 예

Slide 42

- 윈도우 NT
  - 요구페이징 + 클러스터링 (clustering)
  - 클러스터링: 페이지 틀림 페이지와 더불어 주위에 있는 페이지도 적재
  - 프로세스가 시작되면 최소, 최대 작업집합이 할당됨
    - 최소 작업집합: 프로세스에게 보장해주는 최소 프레임의 수
    - 최대 작업집합: 프로세스에게 할당할 수 있는 최대 프레임의 수
  - 빈 프레임 목록을 유지
    - 최대 작업집합보다 작게 할당된 프로세스에 페이지 틀림이 발생하면 ➙빈 프레임 목록에서 새로 할당
    - 최대 작업집합과 같게 할당된 프로세스에 페이지 틀림이 발생하면 ➙지역 페이지 교체정책으로 페이지 하나를 선택하여 교체
    - 빈 프레임이 임계값 이하로 내려가면
      - 자동 작업집합 조절 (automatic working-set trimming) 동작
      - 최소 작업집합보다 많게 할당된 프로세스에게서 프레임을 제거 (최소 작업집합이 될 때까지)
      - 프레임 제거 방법은 프로세서별로 다른 방법 사용

기타 고려사항

Slide 43

- 선 페이징(prepaging)
  - 필요성
    - 순수 요구 페이징은
      - 프로세스 시작시
      - 스왑아웃된 페이지가 다시들어올때 많은 페이지 틀림이 발생
  - 선 페이징 방법: 사용 가능성이 있는 페이지를 모두 가져 옮
  - 예)
    - 작업집합을 사용하는 시스템에서
    - 해당 프로세스가 대기로 스왑아웃 되었다가 다시 시작될 때
    - 작업집합을 모두 복구 시킴
  - 비용
    - s개가 선 페이징됨
    - s개 중 α $(0 \leq \alpha \leq 1)$비율만 사용된다면
    - $(1-\alpha)$ 의 선 페이징 비용 $< \alpha$ 개의 페이지 틀림 처리 비용이면 사용

기타 고려사항 (Cont'd)

Slide 44

- 페이지의 크기
  - 보통 512 ~16K 바이트나 워드 크기를 가짐
  - 작을 수록 좋은 이유
    * 마지막 페이지의 내부 단편화가 줄어듬 (평균적으로 페이지 크기의 절반)
    * 입출력 전체의 크기는 줄어듦 (필요한 것만 세부적으로 적재 가능함으로)
  - 클 수록 좋은 이유
    * 페이지 테이블의 크기가 작아짐
    * 입출력 시간의 단축
      (전송시간보다 주로 탐색시간 및 회전지연에 시간이 많이 걸림)
    * 페이지 틀림 확률이 줄어듦
    * 페이지 틀림 처리 비용이 예전보다 증가
  - 전반적으로 크게하는 것이 전체적인 추세임

기타 고려사항 (Cont'd)

Slide 45

- 프로그램 구조
  - 요구페이징은 프로그램의 동작 자체에는 영향을 미치지 않음
  - 단, 프로그램 구조에 따라 성능은 좋아질 수 있음 (아래 예)

```
int A[][] = new int[1024][1024];
for (j = 0; j < A.length; j++)
    for (i = 0; i < A.length; i++)
        A[i,j] = 0;
```

1024 x 1024 page faults

```
int A[][] = new int[1024][1024];
for (i = 0; i < A.length; i++)
    for (j = 0; j < A.length; j++)
        A[i,j] = 0;
```

1024 page faults

기타 고려사항 (Cont'd)

Slide 46

- 입출력 상호잠금 (I/O Interlock)
  - 문제상황
    - 프로세스가 입출력 요구 후 입출력 장치 큐에 삽입
    - 다른 프로세스가 수행되고 수행 중 페이지 틀림 발생
    - 전역 교체 알고리즘으로 입출력 프로세스의 버퍼 페이지를 교체
    - 입출력 서비스 수행됨 (이때 버퍼 페이지는 다른 프로세스가 사용중)
  - 해결방법
    - 사용자 기억장치에 직접 입출력을 하지 않음
      - 시스템 기억장치를 이용하여 입출력
        (시스템 기억장치는 교체되지 않음으로...)
      - 단, 사용자 기억장치와 시스템 기억장치사이에 복사가 필요
        (성능에 문제)
    - 잠금비트(lock-bit)를 사용
      - 프레임마다 잠금비트를 사용
      - 잠금비트가 1인 것은 교체되지 않음
      - 입출력 버퍼나 운영체제 커널에 할당된 페이지는 잠금비트 설정
      - 단, 주위깊게 해제하지 않으면 가용 프레임 수가 줄어 듦