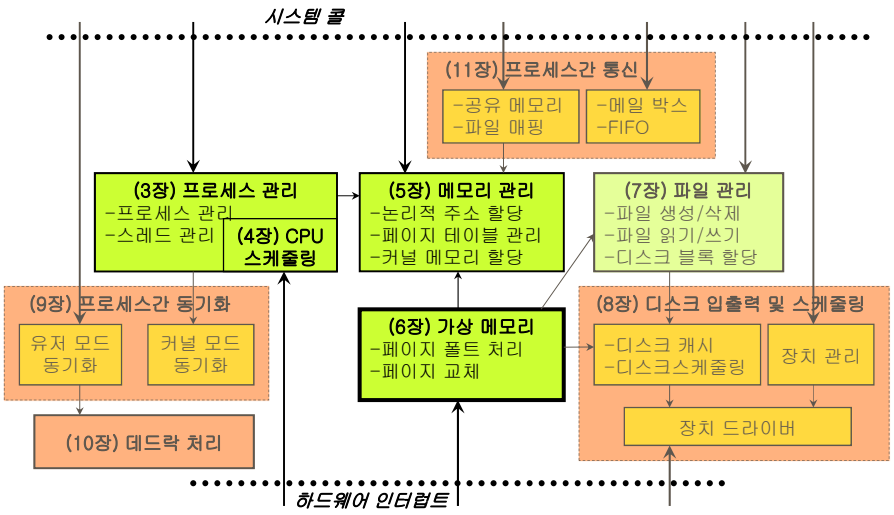
가상 메모리

- ✓메모리 용량 한계의 극복
- ✓디멘드 페이징
- ✓스왑 디바이스
- ✓페이지 테이블과 페이지 폴트
- ✓페이지 폴트의 처리
- ✓디멘드 페이징과 평균 메모리 접근 시간
- ✓페이지 교체 알고리즘
- ✓디멘드 페이징에서 고려해야 할 사항들
- ✓디멘드 세그먼테이션
- ✓페이지 폴트 처리의 구현 예

관련 운영체제 구성 모듈



이해하고 넘어가야 할 내용들

- ✓ 가상메모리의 기본 개념 및 디멘드 페이징에 대한 이해
- ✓ 페이지 폴트 발생 및 처리의 구현 방법에 대한 이해
- ✓ 페이지 교체 알고리즘 및 구현 방법에 대한 이해
- ✓ 실제 운영체제들에서 적용하는 페이지 교체 알고리즘들의 비교

메모리 용량 한계의 극복

- ✓ **메모리 사용량 절약**
  - ✓ 실행시점 동적 링킹, 공유 라이브러리
- ✓ **전체 메모리 용량을 초과하면**
- ✓ **가상메모리 (virtual memory)**
  - ✓ 스왑(swap) 디바이스를 메모리의 일부처럼 활용
  - ✓ 당장 실행하지 않아도 되는 부분은 스왑 디바이스에 보관 (swap-out)
  - ✓ 실행해야 하는 시점이 되면 스왑 디바이스로부터 읽어 들임 (swap-in)
  - ✓ 스왑 인/아웃의 단위: 페이지/세그먼트

디멘드 페이징

- ✓ **프리페이징 (pre-paging)**
  - ✓ 실행에 필요할 것으로 예측되는 페이지들을 미리 적재
  - ✓ 예측이 어려움 → 디멘드 페이지의 보완 기능으로 활용
  - ✓ 예측: 실행되는 페이지의 인접 페이지 들
- ✓ **디멘드 페이징 (demand paging)**
  - ✓ 당장 실행에 필요한 페이지들을 스왑 디바이스에서 메모리로 적재
  - ➔ 각 프로세스는 일부분만 메모리에 적재됨
- ✓ **디멘드 페이징의 구현에 필요한 것들**
  - ✓ 1. 스왑 디바이스에 페이지들을 저장하는 방식
  - ✓ 2. 프로세스가 실행 도중에 필요한 페이지를 결정하고 적재하는 방식 ➔ 페이지 폴트 처리
  - ✓ 3. 스왑 아웃 대상의 페이지들을 선택하는 규칙 ➔ 페이지 교체 알고리즘

스왑 디바이스

- ✓ **2가지 구현 방안**
  - ✓ 스왑 파티션: 별도의 디스크 파티션을 스왑 디바이스 용으로 할애
  - ✓ 스왑 파일: 일반 파티션에 큰 크기의 파일을 생성하고 이것을 스왑 디바이스처럼 사용
- ✓ **장단점**
  - ✓ 디스크 입출력 속도: 스왑 파일은 파일시스템의 블록위치 결정 등의 오버헤드 수반
  - ✓ 디바이스의 용량: 스왑 파티션은 디스크 초기화 과정에서 크기가 고정됨
- ✓ **2가지를 상호 보완하여 사용**
  - ✓ 정상 상태에서는 스왑 파티션 사용
  - ✓ 스왑 파티션의 용량이 부족해 지면 스왑 파일을 보조 수단으로 사용

스왑 디바이스 블록 할당

- ✓ **2가지 방안**
  - ✓ 1. 프로세스 별로 연속된 블록들을 할당
  - ✓ 2. 매 페이지 별로 한 블록씩 할당
- ✓ **코드 영역**
  - ✓ 필요한 페이지는 실행파일로부터 직접 읽어 들임
  - ← 프로세스 실행중에 변경되지 않음
- ✓ **데이터 영역**
  - ✓ 수시로 변경되므로 스왑 디바이스에 블록 할당
  - ✓ 처음 요청되는 시점에는 실행파일로부터 직접 읽어올 수 있음
- ✓ **스택 영역**
  - ✓ 수시로 변경되므로 스왑 디바이스에 블록 할당
  - ✓ 처음 요청되는 시점에는 스왑인 작업 생략 (메모리만 할당)
- ✓ **스왑 아웃 작업의 생략**
  - ✓ 메모리에 적재된 후에 전혀 수정이 없으면 스왑 아웃 작업 생략
  - ✓ 수정 여부는 페이지/세그먼트 테이블의 Dirty 비트로 검사 (MMU에 의해 설정)

페이지 테이블과 페이지 폴트

- ✓ **페이지 폴트 (page fault)**
  - ✓ 프로세스의 정상적인 주소지만 메모리에 적재되지 않아서 주소오류가 발생하는 것
  - ← 페이지 테이블에 invalid 로 설정
  - ✓ 인터럽트 처리과정에서 필요한 페이지를 스왑 인
- ✓ **페이지 테이블 항목**
  - ✓ 페이지 프레임 번호
  - ✓ Valid 비트
  - ✓ Write 비트
  - ✓ Access (또는 Reference) 비트
  - ✓ Dirty (또는 Modify) 비트
  - ✓ User Mode 비트 등 포함

논리적 주소

p	d(12bit)

p : 페이지 번호
d : 페이지 내의 오프셋 (displacement)

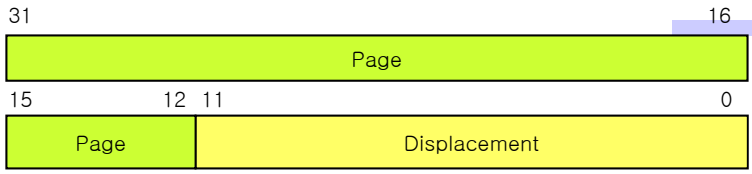
페이지 테이블 항목

f	?(7) U(1) D(1) A(1) W(1) V(1)

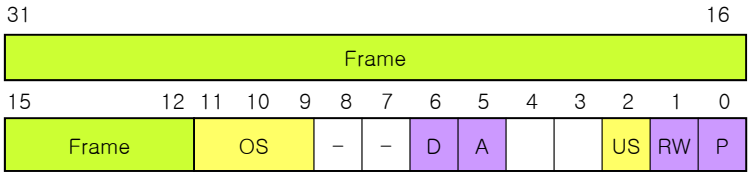
f : 페이지 프레임 번호
U(user mode)
D(dirty) A(access) W(write) V(valid)

x86 프로세서

논리적 주소

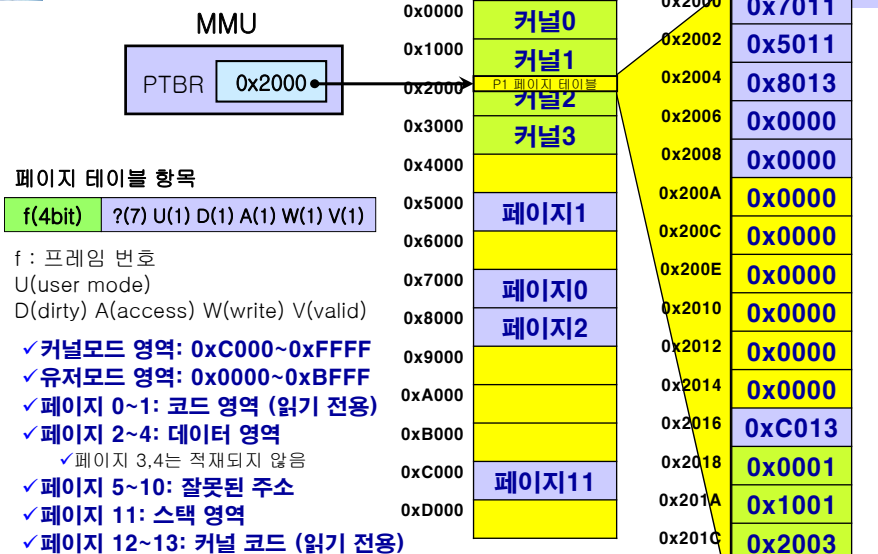


페이지 테이블 항목



Frame: 프레임 번호 (20bit)
OS: 용도 (page aging)
D: dirty
A: accessed
US: user-supervisor(1) / supervisor-only(0)
RW: read-write(1) / read-only(0)
P: present(1) / invalid(0)

페이지 테이블 예



페이지 테이블 항목

f(4bit)	?(7) U(1) D(1) A(1) W(1) V(1)

f : 프레임 번호
U(user mode)
D(dirty) A(access) W(write) V(valid)

- ✓커널모드 영역: 0xC000~0xFFFF
- ✓유저모드 영역: 0x0000~0xBFFF
- ✓페이지 0~1: 코드 영역 (읽기 전용)
- ✓페이지 2~4: 데이터 영역
  - ✓페이지 3,4는 적재되지 않음
- ✓페이지 5~10: 잘못된 주소
- ✓페이지 11: 스택 영역
- ✓페이지 12~13: 커널 코드 (읽기 전용)
- ✓페에지 14~15: 커널 데이터

페이지 폴트의 처리

- ✓ **주소오류 인터럽트가 발생하는 원인은 2가지**
- ✓ 1. 정상적인 주소지만 메모리에 적재되지 않은 상태 (Page Fault)
  - ✓ 해당 페이지를 적재하는 작업 실시 (swap in) 후
  - ✓ 해당 프로세스를 주소오류 인터럽트 시점부터 실행 재개
- ✓ 2. 프로그램 오류에 의한 잘못된 주소
  - ✓ 해당 프로세스를 강제로 종료

페이폴트 처리 시나리오 예

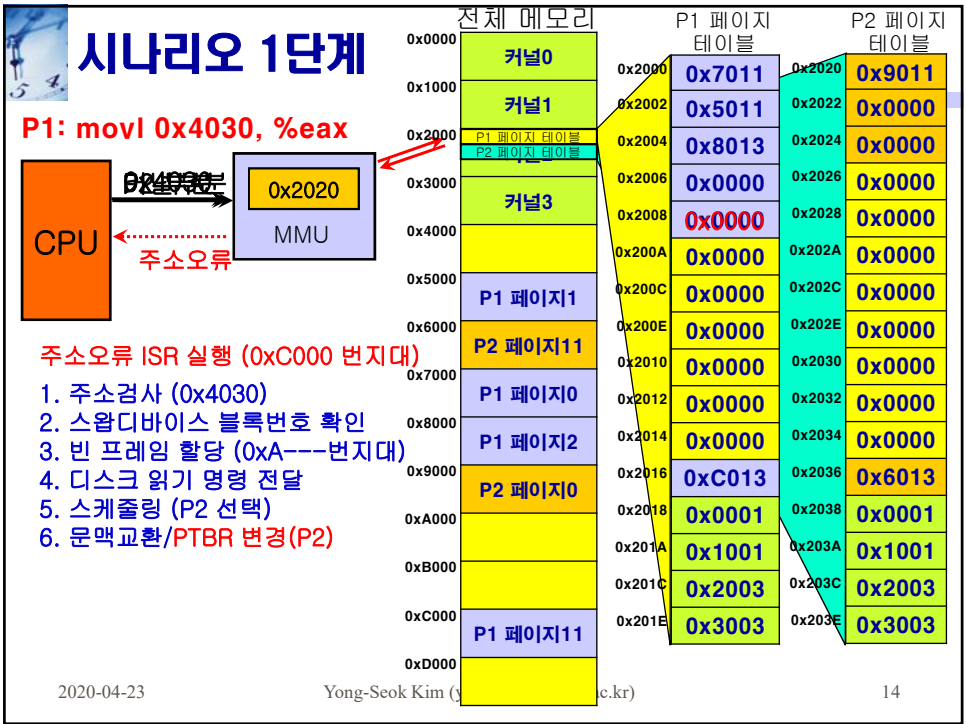
- ✓ **논리적 주소공간**
  - ✓ 사용자 모드 공간 : 0x0000 ~ 0xBFFF
  - ✓ 커널 모드 공간 : 0xC000 ~ 0xFFFF
- ✓ **커널**
  - ✓ 4페이지이며 0x0000 ~ 0x3FFFF 까지 적재
  - ✓ 프로세스 P1 페이지 테이블 주소: 0x2000 ~ 0x201F
  - ✓ 프로세스 P2 페이지 테이블 주소: 0x2020 ~ 0x203F
- ✓ **프로세스 P1 논리적 주소공간**
  - ✓ 0x0000 ~ 0x1FFF: 코드영역 (2페이지)
  - ✓ 0x2000 ~ 0x4FFF: 데이터영역 (3페이지)
  - ✓ 0xB000 ~ 0xBFFF: 스택영역 (1페이지)
- ✓ **프로세스 P2 논리적 주소공간**
  - ✓ 0x0000 ~ 0x0FFF: 코드영역 (1페이지)
  - ✓ 0x1000 ~ 0x2FFF: 데이터영역 (2페이지)
  - ✓ 0xB000 ~ 0xBFFF: 스택영역 (1페이지)
- ✓ **P1 이 다음의 명령어 실행**
  - ✓ `movl 0x4030, %eax`

페이지 폴트 처리 시나리오

1. **(프로세스 P1)** CPU는 P1의 전역변수 영역인 0x4030번지 읽기 시도 (예: 'movl 0x4030, %eax')
2. **(MMU)** 4번 페이지의 페이지 테이블 항목이 invalid ➔ 주소 오류 인터럽트
3. **(커널)** 운영체제의 주소오류 인터럽트 처리 루틴 ➔ 오류 발생 주소가 P1의 정상적인 주소인지를 검사
4. **(커널)** 정상적인 데이터 영역의 주소이므로 비어있는 A번 프레임 (0xA000번지)을 찾아 P1에게 할당
5. **(커널)** 스왑 디바이스에서 P1의 4번 페이지에 해당하는 블록 검사 ➔ 디스크 제어기에게 이를 읽어서 0xA000번지부터 채우도록 명령을 전달
6. **(커널)** P1을 대기 상태로 전환하고 스케줄링 실행 ➔ 다른 프로세스를 선택하고 문맥교환
7. **(프로세스 P2)** 스케줄링에 의해 선택된 P2 실행
8. **(디스크 제어기)** 디스크 읽기 작업을 실행하고 ➔ P1의 4번 페이지 읽기가 완료되면 디스크 인터럽트를 발생
9. **(커널)** 운영체제의 디스크 인터럽트 처리 루틴 ➔ P1의 페이지 테이블의 페이지 4의 항목을 0xA013으로 수정
10. **(커널)** P1를 실행 가능 상태로 변경하고 스케줄링 실행 ➔ 적절한 프로세스 (P1)로 문맥교환
11. **(프로세스 P1)** CPU는 0x4030번지 읽기 다시 실행
12. **(MMU)** 논리적 주소 0x4030을 물리적 주소 0xA030번지로 변경하여 메모리로 전송
13. **(프로세스 P1)** CPU는 물리적 주소 0xA030번지의 내용 읽기 완료

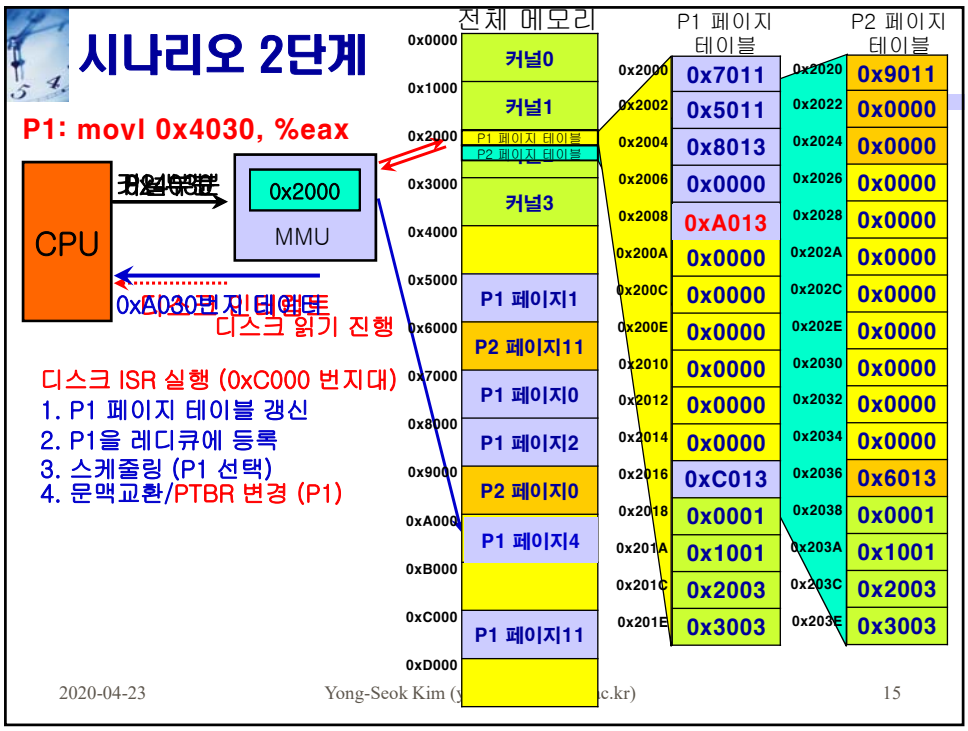
시나리오 1단계

P1: movl 0x4030, %eax



주소오류 ISR 실행 (0xC000 번지대)

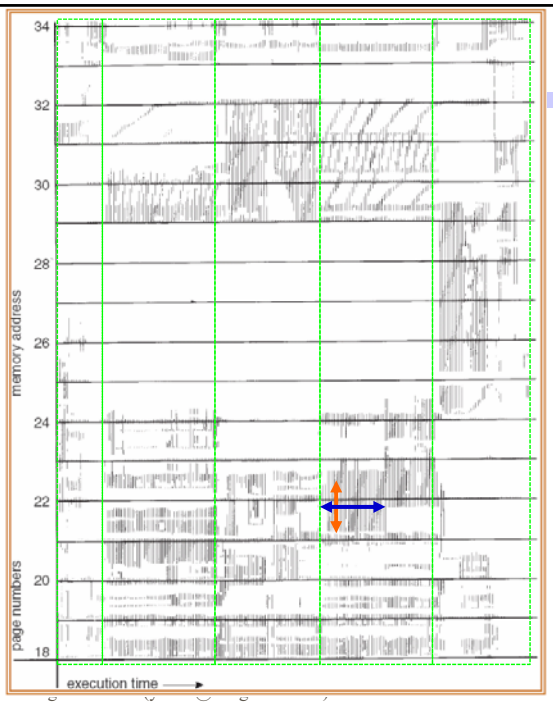
1. 주소검사 (0x4030)
2. 스왑디바이스 블록번호 확인
3. 빈 프레임 할당 (0xA---번지대)
4. 디스크 읽기 명령 전달
5. 스케줄링 (P2 선택)
6. 문맥교환/PTBR 변경(P2)



평균 메모리 접근 시간

- ✓ **정상적인 실행시 메모리 접근 시간**
  - ✓ 정상적인 메모리 접근시간
- ✓ **페이지 폴트 발생시 메모리 접근시간**
  - ✓ 페이지 폴트 발생 오버헤드
  - ✓ (한 페이지 스왑 아웃 시간)
  - ✓ 한 페이지 스왑 인 시간
  - ✓ 프로세스 실행 재개 오버헤드
  - ✓ 정상적인 메모리 접근 시간
- ✓ **평균 메모리 접근 시간 (페이지 폴트 발생 확률: p)**
  - ✓ EAT = (1−p) x (메모리 접근시간) + p x (페이지 폴트시의 시간)
  - ✓ 가정: 메모리 접근시간 114ns, 페이지 폴트 발생 및 실행재개 오버헤드 10us, 스왑 인/아웃 시간 50ms
  - ✓ EAT = 114 + p x 50,010,000 ns
  - ✓ 10% 이하의 속도지연 ➔ $p < 2.28 \times 10^{-7}$
  - ✓ 메모리 사용의 지역성으로 인해 이 정도의 확률이 가능함

메모리 접근의 지역성



공간적 지역성
시간적 지역성

페이지 교체 알고리즘

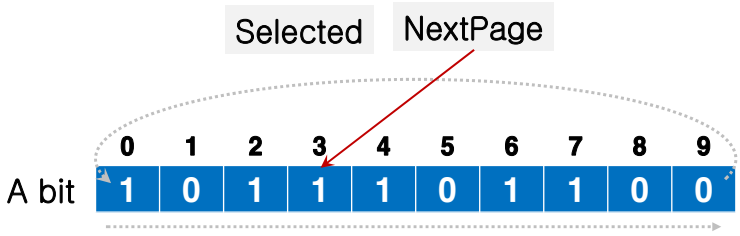
- ✓ **페이지 교체 (page replacement)**
  - ✓ 빈 메모리 프레임을 확보하기 위해 적절한 대상 페이지를 선택하여 스왑 아웃
- ✓ **페이저 (pager)**
  - ✓ 페이지 교체 작업을 실행하는 주체
  - ✓ 보통 커널 모드에서 실행되는 스레드로 구현
  - ✓ 빈 메모리 프레임 개수가 일정수준 이하로 줄어들 때 실행
- ✓ **페이지 교체 알고리즘**
  - ✓ 스왑 아웃 대상 페이지를 선택하는 알고리즘
  - ✓ 평가기준: 페이지 폴트 발생을 최소화하면서 오버헤드가 적어야
  - ✓ 일반적으로 메모리 사용의 지역성에 근거하여 당분간 사용하지 않을 페이지들을 선택

대표적인 페이지 교체 알고리즘

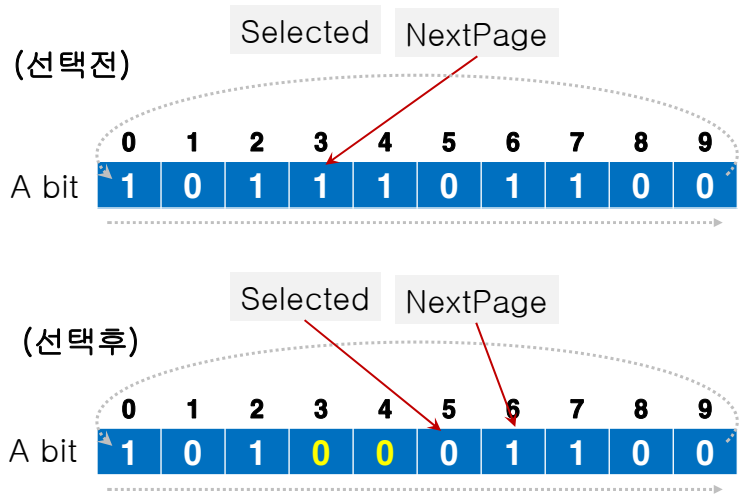
- ✓ **최적 (Optimal 또는 OPT) 알고리즘**
  - ✓ 전체 프로세스들을 완료할 때까지 페이지 폴트가 최소한으로 발생하는 알고리즘
  - ✓ 이론상으로만 존재, 여러 가지 알고리즘들을 평가할 때 평가 기준으로 활용함
- ✓ **LRU (Least Recently Used) 알고리즘**
  - ✓ 사용된 지가 오래된 페이지 프레임 선택
  - ✓ 당분간 사용되지 않을 페이지를 예측하는 데에 메모리 사용의 지역성 특성 활용
- ✓ **에이징 (Aging) 알고리즘**
  - ✓ LRU를 현실적으로 사용가능 하게 단순화 시킴
  - ✓ 주기적으로 페이지 프레임 별로 사용된 시간 증가 (Access 비트가 설정되지 않은 페이지들은 age 증가, A ← 0) ➔ age 가 가장 큰 페이지 선택
- ✓ **NRU (Not Recently Used) 알고리즘**
  - ✓ 에이징을 더 단순화 시켜서 별도의 사용/비사용으로만 처리
  - ✓ Access 비트가 설정되지 않은 페이지들 중에 임의로 선택
  - ✓ A 비트 clear 처리는 적절히 (주기적으로 또는 모두 설정되었을 때)
- ✓ **이차기회 (Second Chance) 알고리즘 (또는 Clock Algorithm)**
- ✓ **워킹 셋 (Working Set) 알고리즘**
  - ✓ 지정한 age 이내 것들 (워킹 셋)과 초과한 것들로 구분
  - ✓ 초과한 것들 중 임의로 선택

이차기회 알고리즘

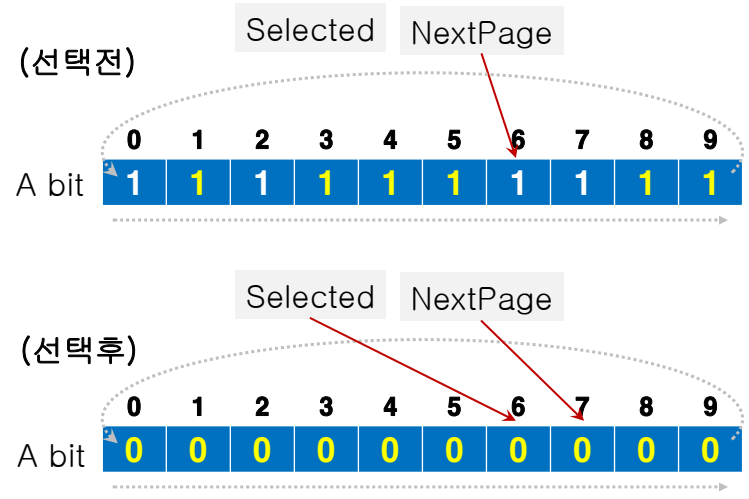
- ✓ **모든 페이지 프레임들을 원형의 목록으로 관리**
- ✓ **차례대로 검사하면서 Access 비트가 0인 것을 선택, 0 이 아니면 0 으로 수정하고 다음 프레임 검사**
- ✓ **모든 페이지 프레임들이 Access 비트가 1 이더라도 2차 검사과정에는 반드시 Access 비트가 0인 프레임이 선택됨**
- ✓ **Access 비트**



이차기회 알고리즘 (현재)



이차기회 알고리즘 (다음검사)



이차기회 알고리즘 예시

```
frameEntT *vmemNextPage;          // 다음 번에 검사할 첫 페이지
frameEntT *vmemAllocFrame()
{  frameEntT *selected = (frameEntT *)0;

    while (selected == (frameEntT *)0) {
      // if access bit is not 0
      if (*(vmemNextPage->MmuEnt) & PAGE_ACCESS_BIT) {
          // clear the access bit
          *(vmemNextPage->pageMmuEnt) &= ~PAGE_ACCESS_BIT;
      // if the access bit is 0
      } else {
          // select the page
          selected = vmemNextPage;
      }
      // move to the next page
      vmemNextPage = vmemNextPage->Next;
    }
    return selected;
}
```


이차기회 알고리즘의 장단점

- ✓ **장점**
  - ✓ 단순하고 효율적임
  - ✓ (비교) 워킹 셋 알고리즘에서는 주기적으로 페이지들의 age를 갱신해야 함
- ✓ **단점**
  - ✓ 우선순위가 낮은 프로세스의 페이지가 상대적으로 더 많이 선택됨

  ←우선순위가 낮은 프로세스는 실행할 기회가 작으므로
  소속 페이지들이 사용되지 않은 것으로 유지되고
  스왑 아웃 대상으로 선정될 가능성 높음

워킹셋 알고리즘

- ✓ **워킹셋 (working set)**
  - ✓ 프로세스 별로 최근 일정한 실행 시간 구간 내에 사용된 페이지 들의 집합
  - ✓ ㈜ 실제 시간이 아니고 프로세스가 실행되는 시간에 따라 증가하는 프로세스 별 가상 시간 적용
- ✓ **워킹셋 알고리즘**
  - ✓ 워킹셋에 포함되지 않은 페이지 프레임들 중에서 임의로 선택
  - ← 메모리 사용의 지역성 특성 활용
  - ✓ 시간 구간의 크기는 운영체제의 실행 환경에 따라 적절하게 설정
- ✓ **위킹셋의 관리**
  - ✓ 에이징 알고리즘과 유사하게 프로세스 별로 페이지 프레임들의 age를 관리
  - ✓ age가 시간 구간을 초과하는 페이지는 워킹셋에서 제외

UNIX / LINUX / Windows

- ✓ **버전 별로 약간씩의 변화가 있음**
- ✓ **UNIX/Windows**
  - ✓ 이차기회 알고리즘을 적용하는 예가 많음
  - ✓ UNIX: 4.4 BSD, System V
  - ✓ Windows: 프로세스 별로 이차기회 알고리즘 적용, 프로세스 별 할당 페이지 프레임 수는 수시로 조절
- ✓ **Linux 2.2/2.4/2.6**
  - ✓ 경험적인 결과를 바탕으로 버전마다 차이가 있음
  - ✓ 커널 2.2: 이차기회 알고리즘에 근거하지만 메모리 프레임을 많이 가진 프로세스부터 검사
  - ✓ 커널 2.4/2.6: 모든 페이지 프레임들을 active 목록 (일종의 워킹셋)과 inactive 목록으로 구분, inactive 목록의 페이지들에 이차기회와 유사한 알고리즘 적용

고려해야 할 사항들

- ✓ **스왑아웃 대상 페이지 범위**
  - ✓ 1. 모든 페이지 프레임들 중에서 선택
  - ✓ 2. 페이지 폴트가 난 프로세스의 소유 프레임들 중에서 선택
- ✓ **프로세스 개수 (장기 스케줄러)**
  - ✓ 프로세스 개수가 많으면 계산 작업과 입출력 작업이 섞여서 CPU의 활용률 증가
  - ✓ 그러나, 프로세스 개수가 늘수록 프로세스당 할당되는 프레임 수 감소 ➔ 페이지 폴트 발생 확률 증가
  - ✓ 트레싱 (trashing) 현상: 페이지 폴트 증가 ➔ 페이지 입출력을 위해 대기하는 시간 증가 ➔ 프로세스 추가 생성 ➔ 페이지 폴트 더욱 증가
- ✓ **페이지 크기가**
  - ✓ 클수록 페이지 테이블 크기 감소
  - ✓ 작을수록 꼭 필요한 부분만 메모리에 적재
- ✓ **입출력 인터락 (interlock) 문제 해결**
  - ✓ 1. DMAC에 의해 사용되고 있는 페이지는 스왑아웃 대상에서 제외
  - ✓ 2. 모든 입출력은 커널 부분의 버퍼사용

응용 프로그램의 작성 기법

- ✓ **응용 프로그램 작성시 페이지 폴트 발생도 고려해야**
- ✓ **배열 A[1024][1024]를 초기화 하는 예제**

```
(1) for (i=0; i<1024; i++)
        for (j=0; j<1024; j++)
            A[i][j] = 1;

(2) for (j=0; j<1024; j++)
        for (i=0; i<1024; i++)
            A[i][j] = 1;
```

➔ **(2)는 (1)에 비해 월등히 많은 페이지 폴트 발생**

- ✓ (1)이 메모리 사용의 지역성이 높음

디멘드 세그먼테이션

- ✓ 세그먼트 단위로 스왑 인/아웃하는 점만 다를 뿐 기본적으로 디멘드 페이징에서의 처리방법과 유사
- ✓ 스왑 디바이스의 블록 할당 시 세그먼트 별로 연속된 영역 할당
- ✓ x86 프로세서
  - ✓ 페이징 기능과 결합된 세그먼테이션 기능 제공
  - ✓ 그러나, 리눅스에서는 페이징 기능만 활용하고 있음

페이지 폴트 처리의 구현 예

- ✓ 스왑 디바이스 블록 번호 기록
  - ✓ 페이지 테이블에서 invalid 인 항목은 나머지 비트들에 블록 번호 기록

프레임 번호를 기록하고 있을 때

f(4bit)	?(8) D(1) A(1) W(1)	1

f :프레임 번호, D: dirty, A: access, W: write, V=1(*valid*)

스왑 디바이스의 블록 번호를 기록하고 있을 때

Block (15bit)	0

Block: 스왑 디바이스 블록 번호, V=0(*invalid*)

- ✓ 페이지 폴트의 처리 예 : 프로그램 6.2 참조

```
void addrErrorIsr(void)
{
   unsigned short addr, paddr, operation;
   int page, block;
   unsigned short protect;              // protection mode
   frameEntT *selected;

①  addr = vmemFaultAddr();              // get the fault address
   operation = vmemFaultOperation();    // get the fault operation

②  page = (addr >> PAGE_OFFSET_BITS);

③  // read memory operation on valid code area
   if (addr >= RunningProcess->CodeAddr
     && addr < RunningProcess->CodeAddr + RunningProcess->CodeSize
     && operation == MEM_OP_READ)
   {
      protect = PAGE_VALID_BIT;

   // read/write memory operation on valid data area
   } else if (addr >= RunningProcess->DataAddr
     && addr < RunningProcess->DataAddr + RunningProcess->DataSize)
   {
      protect = PAGE_VALID_BIT | PAGE_WRITE_BIT;
```



```
   // read/write memory operation on valid stack area
   } else if (addr >= RunningThread->StackAddr
      && addr < RunningThread->StackAddr + RunningThread->StackSize)
   {
      protect = PAGE_VALID_BIT | PAGE_WRITE_BIT;

   // invalid address or write memory operation on code area
   } else {
      ProcessExit();
   }
④  // allocate a memory frame
   selected = vmemAllocFrame();                  // (프로그램 6.1)
   // get the physical memory address of the selected page
   paddr = selected->Paddr;

   if (selected->MmuEnt != (unsigned short *)0) {
⑤    // if it is dirty page, write to the swap device
     block = selected->Block;
     if (*(selected->MmuEnt) & PAGE_DIRTY_BIT) {

        // allocate a block on the swap device
        if(block == 0) {
           block = vmemAllocSwapBlock();
           selected->block = block;
         }
        // write the selected page to the swap device (swap-out)
        diskOperation(DISK_WRITE, block, paddr);  // (프로그램 8.1)
     }
```

```
⑥     *(selected->MmuEnt) = (block << 1);
      // now, PAGE_VALID_BIT is cleared
    }
⑦   selected->MmuEnt = &RunningProcess->PageTable[page];

⑧   // the block number of the wanted page on the swap device
    block = (*(selected->MmuEnt) >> 1);
    // set the block number of swap device
    selected->Block = block;
    if (block == 0) {
        clearPageFrame(paddr);
    } else {
        // read the wanted page from the swap device (swap-in)
        diskOperation(DISK_READ, block, paddr); // (프로그램 8.1)
    }

⑨   // set the page table entry as valid
    *(selected->MmuEnt) = paddr | protect;

⑩    return;
}
```