운영체제: 강의노트 11

A. Silberschatz, P.B. Galvin, G. Gagne
Operating System Concepts,
Sixth Edition, John Wiley & Sons, 2003.

Part III. Storage Management

10 가상기억장치

10.1 배경

- 가상기억장치의 등장 배경
  - 프로그램을 실행하는 동안 종종 프로그램 전체를 모두 사용하지 않는다.
    - 오류를 처리하는 루틴은 오류가 발생하지 않으면 사용되지 않는다.
    - 프로그램에서 사용되는 배열, 연결 리스트, 테이블과 같은 구조는 일반적으로 실제로 사용되는 공간보다 많은 공간을 할당하여 사용한다.
    - 프로그램의 일부 기능과 특징은 거의 사용되지 않는 경우가 많다.
  - 프로그램을 실행할 때 한 순간에 프로그램 전체를 필요로 하지 않는다.
- 프로그램의 일부만 주기억장치에 있어도 실행이 가능하도록 하면 얻을 수 있는 장점
  - 프로그램은 더 이상 물리적 주기억장치 크기에 제한을 받지 않는다.
  - 각 프로그램이 적은 공간을 차지하므로 동시에 여러 프로그램을 주기억장치에 적재할 수 있다.
  - 프로그램을 적재하는데 걸리는 입출력 비용을 줄일 수 있다.
- 가상기억장치는 **요구 페이징(demand paging)** 기법을 이용하여 구현한다. 이 때 세그먼테이션과 결합하여 사용할 수 있다.

10.2 요구 페이징

- 요구 페이징은 스왑핑과 페이징을 결합하여 사용한다.
- 요구 페이징에서는 프로세스 전체를 스왑하지 않고, **지연 스왑퍼(lazy swapper)**를 사용한다. 이 스왑퍼는 페이지가 필요할 경우에만 주기억장치에 적재한다. 페이징에서는 스왑퍼라는 용어 대신에 **페이저(pager)**라는 용어를 사용한다.

10.2.1 기본 개념

- 프로세스가 스왑인될 때 페이저는 어떤 페이지들을 사용할지 추측한 다음에 그 페이지들만 스왑인한다.
- 이를 위해 주기억장치에 있는 페이지와 디스크에 있는 페이지를 구분할 수 있어야 한다. 이 구분은 보통 유효비트를 이용한다.
  - 유효비트가 1이면 이 페이지는 주기억장치에 적재되어 있는 페이지임을 의미한다.
  - 유효비트가 0이면 이 페이지는 디스크에 있거나 프로세스의 가장 주소 공간에 포함되지 않는 페이지임을 의미한다.
- 프로세스가 주기억장치에 적재되어 있지 않는 페이지를 접근하면 다음과 같은 절차에 따라 그 페이지를 주기억장치에 적재한다.
  - 단계 1. 접근하는 페이지에 대해 페이지 테이블을 참조하면 유효비트가 현재 0이므로 **페이지 결함 트랩(page fault trap)**을 발생한다.
  - 단계 2. 트랩이 발생하면 프로세스의 PCB를 검사하여 참조의 유효성을 검사한다. 즉, 디스크에 있는 패이지에 대한 참조인지 프로세스의 가상 주소 공간을 벗어난 참조인지를 검사한다. 만약 후자이면 프로세스는 종료된다.
  - 단계 3. 빈 프레임을 찾는다.
  - 단계 4. 디스크 입출력을 이용하여 페이지를 프레임에 적재한다.
  - 단계 5. 디스크 입출력이 종료되면 프로세스의 내부 테이블과 페이지 테이블을 수정한다.
  - 단계 6. 페이지 결함 트랩을 발생시킨 명령어를 다시 수행한다.
- **순수 요구 페이징**: 프로세스가 처음 실행될 때 주기억장치에 전혀 페이지가 없는 상태로 시작하는 방식.
  - 초기에는 페이지 결함이 자주 발생한다.
- 한 명령어를 실행한 결과로 여러 페이지 결함이 발생할 수 있다. 하지만 참조의 지역성(locality of reference) 원리 때문에 이런 경우는 극히 드물다.
- 요구 페이징을 위한 하드웨어 지원
  - 페이징 테이블에 대한 지원: TLB, 유효 비트, 보호 비트
  - 보조 기억장치: 디스크의 일부분을 스왑 공간으로 사용한다.
- 페이징에서는 페이지 결함이 발생하면 그 페이지를 디스크에서 주기억장치로 옮기고 페이지 결함을 일으킨 명령어를 다시 실행해야 한다.

  - 일반적인 경우에는 페이지 결함을 일으킨 명령어를 다시 실행하는 것은 큰 문제가 되지 않는다.
  - 한 명령어가 여러 위치의 내용을 변경하는 경우에는 간단하지 않다. 특히 한 명령어가 두 개 이상의 다른 페이지를 접근하는 경우에는 명령어를 간단히 다시 실행할 수 없을 수 있다. 이것에 대한 두 가지 해결책이 있다.
    - 방법 1. 명령어 수행에 필요한 모든 페이지를 미리 검사하여 이들을 주기억장치에 적재한 다음에 명령어를 수행한다.
    - 방법 2. 임시 레지스터을 이용하여 일시적으로 바꾼 위치들의 기존 값을 보관한다. 페이지 결함이 발생하면 기존 값을 복원한 다음에 다시 명령어를 수행한다.

10.2.2 요구 페이징의 성능

- 요구 페이징에서 평균 접근 시간

$$(1-p) \times ma + p \times \text{pft}$$

  여기서 $p(0 \leq p \leq 1)$는 페이지 결함이 일어날 확률이고, ma는 주기억장치 접근시간이며, pft는 페이지 결함을 처리하는데 소요되는 시간이다. 보통 ma는 10에서 200 ns 정도 소요된다.

- 페이지 결함을 처리하는 과정은 크게 다음 세 가지 요소로 구성된다.
  - 페이지 결함 인터럽트의 처리: μs 단위이다.
  - 디스크에 있는 페이지를 주기억장치로 적재: 디스크 입출력에 소요되는 시간은 주기억장치 접근 시간에 비해 상대적으로 매우 크다. 대략 25 ms 정도 소요된다.
  - 중단된 프로세스의 재개: 디스크 입출력 때문에 이 프로세스는 대기큐로 옮겨진다. 이 때 CPU는 다른 프로세스를 대신 스케줄하여 실행한다. 그러므로 실제 언제 재개될지는 예측하기 힘들다.

  대기큐에 기다린 시간을 제외하면 페이지 결함을 처리하는 전체 시간은 디스크 입출력에 비례한다. 따라서 페이지 결함률이 0.1%만 되어도 페이지 결함이 없을 때에 비해 250 배 정도 느려진다고 볼 수 있다.

- 스왑 공간에 대한 입출력은 디스크 내에 다른 공간으로부터의 입출력보다 상대적으로 빠르다. 따라서 프로그램을 실행할 때 프로그램을 모두 스왑 공간으로 옮긴 다음에 스왑 공간에서 페이징이 일어나도록 하면 성능을 향상시킬 수 있다.

10.3 프로세스 생성

10.3.1 Copy-on-Write

- 프로그램을 처음 실행하면 디스크에 있는 실행 이미지를 이용하여 프로세스를 생성한다.
- 실행중인 프로세스가 fork() 명령을 사용하여 자식 프로세스를 생성하면 디스크 입출력이 필요없다. fork()에 의한 자식 프로세스의 생성에 대해서는 다음과 같은 방법으로 성능을 향상시킬 수 있다.
  - 보통 자식 프로세스는 바로 exec() 시스템 호출하는 경우가 많다. 이 경우에 부모 프로세스의 주소 공간을 복사하는 것은 낭비이다.
  - 이 때 copy-on-write 기법을 사용할 수 있다. 초기에 자식과 부모 프로세스는 페이지를 모두 공유한다. 한 프로세스가 페이지의 내용을 변경하면 그 때 복사가 이루어진다. 따라서 변경하는 페이지만 복사되고 나머지는 모두 공유된다.

10.3.2 기억장치-사상 파일

- 디스크에 있는 파일을 접근할 때마다 디스크 입출력이 필요하다. 이런 파일 작업을 빠르게 하기 위해 파일을 처음 접근할 때에는 페이지 크기만큼의 데이터가 가상기억장치로 적재되고, 그 다음부터는 주기억장치 접근을 통해 파일을 접근하게 된다. 이런 방식을 기억장치-사상 파일 기법이라 한다.

10.4 페이지 교체

- 10 페이지 크기의 프로세스가 실제 5 페이지만 사용하면 요구 페이징 기법은 디스크 입출력을 줄일 수 있고, 다중 프로그래밍의 정도를 높일 수 있다.
- 시스템의 주기억장치가 40 프레임으로 구성되어 있으면 이와 같은 프로세스를 8개를 병행으로 수행할 수 있다.
- 다중 프로그래밍의 정도를 높이면 프레임 공간이 부족할 수 있다. 또한 주기억장치의 모든 프레임을 사용자 프로세스에게 할당해줄 수 없다.
- 페이지 결함이 발생하였을 때 페이지를 수용할 프레임이 없으면 운영체제는 다음 중 한 가지 방법을 선택하여 이를 해결해야 한다.
  - 프로세스 종료: 페이징은 사용자에게 투명하게 제공되어야 하므로 이것은 선택 가능한 방법이 아니다.
  - 프로세스 스왑아웃: 프로세스의 모든 페이지를 디스크로 스왑아웃하여 다중 프로그래밍의 정도를 줄인다. 때로는 필요한 방법이다.
  - **페이지 교체(page replacement)**: 적재되어 있는 기존 페이지를 새 페이지로 교체한다.

10.4.1 기본 방법

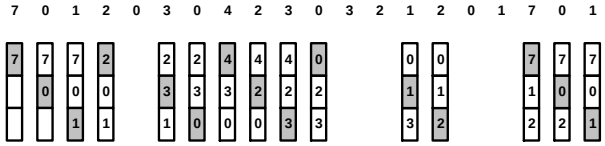
- 페이지 결함이 발생할 때 페이지 교체 절차
  - 단계 1. 디스크에서 페이지의 위치를 찾는다.
  - 단계 2. 빈 프레임을 찾는다.
    - 있으면, 거기에 페이지를 적재한다.
    - 없으면, 페이지 교체 알고리즘을 이용하여 희생 프레임을 선택한다. 그 다음 희생 프레임에 있는 페이지를 디스크에 쓰고, 프레임에 새 페이지를 적재한다.
  - 단계 3. 페이지 테이블과 프레임 테이블을 변경한다.
  - 단계 4. 프로세스를 재개한다.
- 빈 프레임이 없으면 항상 두 번의 디스크 입출력이 필요하다. 디스크 입출력은 **변경 비트(modify bit, dirty bit)**를 이용하여 줄일 수 있다. 이것은 하드웨어적으로 구현되며, 변경이 없는 페이지가 희생자로 선택되면 이 페이지를 디스크에 쓸 필요가 없다. 단, 디스크에 이 페이지의 복사본이 없으면 변경 여부와 상관없이 디스크에 써야 한다. 또한 실행 전용 페이지는 무조건 디스크로 다시 옮길 필요가 없다.
- 제공되어야 하는 두 가지 알고리즘
  - **프레임 할당 알고리즘(frame-allocation algorithm)**: 한 프로세스에게 할당할 프레임의 수를 결정하는 알고리즘
  - **페이지 교체 알고리즘(page-replacement algorithm)**: 빈 프레임이 없을 때 희생자를 선택하는 알고리즘
- 기본적으로 가장 낮은 페이지 결함률을 제공하는 페이지 교체 알고리즘을 사용해야 한다.
- 알고리즘을 평가할 때 일련의 주기억장치 참조에 대해 페이지 결함 수를 계산하여 비교한다. 일련의 주기억장치 참조를 **참조 문자열(reference string)**이라 한다.
- 참조 문자열의 예

  $$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$

  여기서 특정 수는 참조된 페이지 번호이다.
- 페이지 결함 수를 계산하기 위해서는 프로세스에게 할당된 프레임 수를 알아야 한다. 프레임 수가 많을수록 결함 발생수는 적어진다.

10.4.2 FIFO 페이지 교체 알고리즘

- **FIFO (First-In-First-Out)**
- 가장 오래전에 들어온 페이지가 희생자로 선택된다.
- 예) 페이지 프레임 수 = 3

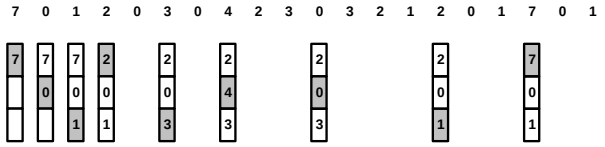


<그림 10.1> FIFO 페이지 교체 알고리즘의 예

- 가장 이해하기 쉽고, 구현하기 쉬운 방법이다.
- 가장 오래전에 들어온 페이지가 초기화 루틴으로 처음에만 사용되고 더 이상 사용되지 않는 것일 수 있지만 초기에 초기화된 자주 사용된 변수가 있는 페이지일 수 있다.
- Belady의 모순: 프레임 수가 증가하면 페이지 결함 발생 수는 감소해야 정상이다. 그러나 FIFO 방식에서는 증가하는 경우도 있다.

10.4.3 OPT 페이지 교체 알고리즘

- **OPT (OPTimal)**
- 가장 오래 동안 사용되지 않을 페이지가 희생자로 선택된다.
- 예) 페이지 프레임 수 = 3

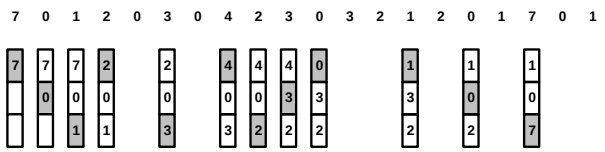


<그림 10.2> OPT 페이지 교체 알고리즘의 예

- 페이지 결함 수를 이 이상 줄일 수 없다.
- 문제점: 미래의 참조를 예측하기가 어렵다.

10.4.4 LRU 페이지 교체 알고리즘

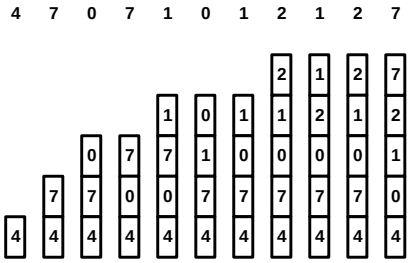
- **LRU (Least-Recently Used)**
- 가장 오래전에 참조한 페이지가 희생자로 선택된다.
- 예) 페이지 프레임 수 = 3



<그림 10.3> LRU 페이지 교체 알고리즘의 예

- 가장 널리 사용되는 기법이다.
- LRU 구현 방법

- 카운터: 각 페이지 테이블의 항마다 하나의 필드를 추가하여 마지막으로 접근된 시간을 나타낸다. 보통은 논리 클럭이나 카운터를 사용한다. 이 방법은 교체가 필요할 때마다 테이블 전체를 검색해야 하며, 페이지가 접근될 때마다 페이지 테이블을 갱신해야 한다.
- 스택: 페이지 번호로 구성된 스택을 이용한다. 스택 톱에는 가장 최근에 접근한 페이지를 유지하고, 스택 바닥에는 교체할 페이지를 유지한다. 이를 위해서는 스택 중간에 있는 페이지를 스택 톱으로 옮겨야 하므로 이중 연결 리스트를 이용하여 구현한다. 이렇게 하면 갱신 비용이 카운터보다 더 많이 소요되지만 검색할 필요가 없다. 참조. 그림 10.4

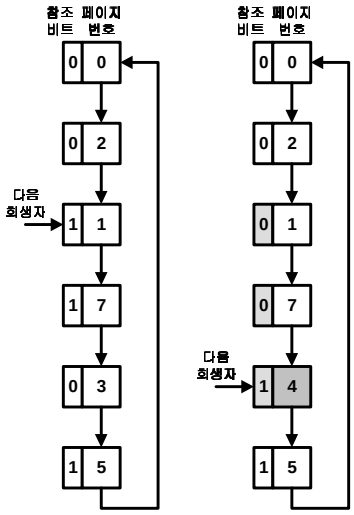


<그림 10.4> 스택을 이용한 LRU 구현

- OPT와 LRU에서는 Belady의 모순 현상이 발생하지 않는다.
- 하드웨어 지원이 반드시 필요하다. 소프트웨어로 테이블의 갱신을 하면 주기억장치 접근마다 많은 시간이 소요된다.

10.4.5 근접 LRU 페이지 교체 알고리즘

- LRU를 제공하기 위한 충분한 하드웨어 지원이 없는 경우에 사용하는 기법이다.
- 대부분의 컴퓨터는 **참조 비트(reference bit)**를 제공한다. 어떤 페이지가 참조되면 자동으로 해당 페이지 테이블을 갱신하여 준다. 이것을 이용하여 LRU와 근접한 페이지 교체 알고리즘을 제공할 수 있다.
- **추가 참조 비트 알고리즘**: 각 페이지마다 참조 비트외에 추가로 8 비트를 이용하고, 이것을 주기적으로 오른쪽 논리 이동을 하여, 지난 8주기 동안 각 페이지의 접근 여부를 유지하는 방법이다. 예를 들어 접근 역사 비트들의 값이 0000 0000이면 8주기동안 한번도 접근이 안된 페이지를 나타내며, 반대로 이 값이 1111 1111이면 지난 8주기 동안 계속 접근이 되었다는 것을 나타낸다. 또한 값이 1100 0100인 페이지는 0111 0111인 페이지보다 더 최근에 접근되었음을 나타낸다.
- 접근 역사를 유지하는 비트의 수는 다양할 수 있다. 극단적인 경우 이 수를 0까지 줄일 수 있다.
- **기회를 한번 더 주는 알고리즘**: 이 알고리즘은 접근 역사를 유지하는 비트를 하나도 사용하지 않고 참조 비트만 사용한다. 다른 말로 클럭 알고리즘이라 한다.
  - 처음에 페이지를 접근하면 참조 비트가 1이 되고, 주기마다 이것을 다시 0으로 바꾸어 준다.
  - 이 알고리즘은 순환 버퍼를 이용하여 구현할 수 있다.
  - 교체할 페이지는 순환 버퍼에서 비트 값이 0인 것이 선택된다. 이 때 찾을 때까지 비트 값이 1인 것은 0으로 바꾼다.
  - 새 페이지는 순환 버퍼에 교체되는 페이지 위치에 삽입한다.
  - 순환 버퍼에 있는 모든 페이지의 비트 값이 1이면 이 알고리즘은 FIFO와 같다.



(a) 검색 전 (b) 검색 후

<그림 10.5> 클럭 알고리즘

- **확장된 클럭 알고리즘**: 이 알고리즘은 참조 비트와 변경 비트를 함께 사용한다.
  - 이 알고리즘은 페이지를 네 개의 클래스로 분류한다.
    - 분류 1. (0,0): 최근에 사용되지도 않고 변경도 되지 않은 페이지
    - 분류 2. (0,1): 최근에 사용되지도 않았지만 변경되어 있는 페이지, 교체를 하면 디스크에 써야 한다.
    - 분류 3. (1,0): 최근에 사용되었지만 변경되지 않은 페이지
    - 분류 4. (1,1): 최근에 사용되었고 변경된 페이지
  - 낮은 분류에 속한 페이지를 먼저 교체한다.
  - 검색시간이 길어진다.

10.4.6 계수 기반 페이지 교체 알고리즘

- 참조된 회수를 기반하는 알고리즘이다.
- LFU(Least-Frequently-Used) 페이지 교체 알고리즘: 참조가 가장 적은 페이지를 교체한다.
  - 문제점
    - 초기에 많이 사용되었지만 더 이상 사용되지 않은 페이지는 계속 교체되지 않는다.
    - 처음으로 교체되어 들어온 페이지는 바로 교체될 수 있다.
  - 첫번째 문제점은 주기적으로 회수를 감소시켜 극복할 수 있다.
- MFU(Most-Frequently-Used) 페이지 교체 알고리즘: 참조가 가장 많은 페이지를 교체한다. 참조가 적은 것일 수록 최근에 들어온 것으로 간주하는 방법이다.
- 두 방법 모두 거의 사용되지 않는다.

10.4.7 페이지 버퍼링 알고리즘

- 페이지 교체 알고리즘과 함께 사용되는 여러 기법이 있다.
- 빈 프레임 풀의 유지
  - 시스템은 보통 고정된 수의 빈 프레임을 항상 유지한다.
  - 페이지 결함이 발생하면 교체 알고리즘에 따라 희생자를 선택한다.
  - 새 페이지를 희생자 프레임에 적재하지 않고 우선 빈 프레임 풀에 속한 프레임에 적재한다. 이것은 페이지 결함 발생 이후 보다 빠르게 프로세스를 재수행하기 위한 것이다.
  - 희생자 프레임이 디스크에 쓰여지면 이 프레임을 빈 프레임 풀에 추가한다.
  - 또한 빈 프레임 풀에 속한 각 프레임에 어떤 페이지가 적재되어 있는지를 유지하면 페이지 교체 때 이것을 활용할 수 있다.
- 변경 페이지 목록 유지
  - 변경된 페이지의 목록을 유지하고, 페이징 장치가 유휴상태이면 변경 페이지를 하나 선택하여 디스크에 쓰고, 변경 비트를 재설정한다.
  - 이 방법을 사용하면 교체할 페이지가 항상 변경되지 않은 깨끗한 페이지일 확률을 높여준다.

10.5 프레임의 할당

- 운영체제가 필요한 만큼을 제외한 나머지 프레임은 모두 사용자 프로세스에게 할당된다. 이 중 일부는 항상 빈 프레임이 되도록 유지할 수 있다.

10.5.1 최저 프레임 수

- 프로세스가 필요한 최소한의 프레임 수 이상을 각 프로세스에게 할당해주어야 한다.
- 최저 프레임 수는 명령어 집합 구조(instruction set architecture)에 의해 정의된다. 하나의 명령어가 참조 가능한 모든 페이지를 수용할 수 있을 만큼의 프레임은 최소한 제공되어야 한다.
- 즉, 명령어가 위치한 페이지와 그 명령어가 참조하는 페이지는 동시에 수용할 수 있어야 한다. 이것은 페이지 결함이 발생하면 그 페이지를 주기억장치에 적재한 다음에 명령어를 다시 실행해야 하기 때문이다. 또한 간접 주소지정 모드를 사용하면 페이지 0도 항상 할당되어 있어야 한다.

10.5.2 할당 알고리즘

- 프로세스의 수: m
- 프레임의 수: n (운영체제와 빈 프레임 풀에 할당된 프레임들은 제외한 수)
- **균등 할당**(equal allocation) **알고리즘**: 각 프로세스에게 n/m 개의 프레임을 할당하는 방법
- **비례 할당**(proportional allocation) **알고리즘**: 각 프로세스의 크기에 비례하여 할당하는 방법
  - s_i: 프로세스 P_i의 크기
  - $S = \sum s_i$
  - 각 프로세스 P_i에게 $s_i/S \times n$ 개의 프레임을 할당한다. 여기서 n은 남은 프레임 수이다.
  - 물론 각 프로세스에게 최저 프레임 수가 할당되도록 조절해야 한다.
  - 예) 62 개의 프레임이 남아 있을 때 10 페이지와 127 페이지로 구성된 두 개의 프로세스가 있으면 각각에 4와 57 개의 프레임을 할당해준다.
- 두 할당 알고리즘 모두 다중 프로그래밍의 정도에 따라 각 프로세스에게 할당되는 프레임의 수는 변한다.
- **우선순위 기반 알고리즘**: 각 프로세스의 우선순위에 비례하여 할당하는 방법으로, 우선순위가 높은 프로세스는 보다 많은 프레임을 할당해주어 빠르게 수행될 수 있도록 해준다.

10.5.3 광역과 지역 할당

- 다중 프로세스가 존재할 때 페이지 교체 알고리즘의 분류
  - **광역 교체 알고리즘(global replacement algorithm)**: 다른 프로세스의 페이지가 희생자로 선택될 수 있는 알고리즘
  - **지역 교체 알고리즘(local replacement algorithm)**: 자신의 페이지만 희생자로 선택될 수 있는 알고리즘

- 광역 교체는 우선순위가 높은 프로세스에게 작은 프로세스를 희생하면서 할당된 프레임 수를 늘려줄 수 있다.
- 광역 교체의 문제는 프로세스가 자신의 페이지 결함률을 제어할 수 없다는 것이다. 즉, 프로세스의 성능이 외부 상황에 따라 변할 수 있다.
- 현재는 대부분 광역 교체 알고리즘을 사용한다.

10.6 스레싱

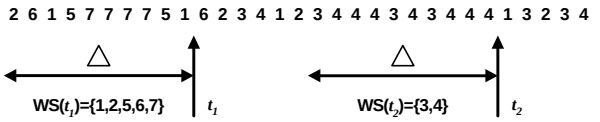
- 계속하여 페이지 결함이 발생하는 현상을 **스레싱(thrashing)**이라 한다. 실행하는 시간보다 페이지 결함을 처리하는 시간이 더 많아지면 이 현상이 발생하였다고 한다.
- 프로세스에 할당된 프레임 수가 필요한 최소의 프레임 수 이하로 내려가면 스레싱이 발생한다.

10.6.1 스레싱의 원인

- 다중 프로그래밍의 정도가 낮으면 CPU 사용 효율이 떨어진다. 이를 극복하기 위해 프로세스의 수를 늘리면 사용 효율은 증가한다. 그러나 어느 정도를 넘으면 스레싱 발생하여 오히려 사용 효율이 떨어진다. 이 경우에는 다시 다중 프로그래밍의 정도를 줄여야 한다.
- 지역 교체 알고리즘을 사용하면 스레싱의 효과를 줄일 수 있다. 이것은 하나의 프로세스의 스레싱이 다른 프로세스에게 영향을 주지 않기 때문이다. 그러나 스레싱이 발생한 프로세스는 대부분의 시간을 페이징 장치의 큐에서 보내게 되므로 다른 프로세스도 기다리는 시간이 길어진다.
- 스레싱를 예방하기 위해서는 프로세스에게 필요한 만큼의 프레임을 할당해주어야 한다. 그러나 프로세스가 필요한 프레임의 수를 알기가 쉽지 않다.
- **지역성 모델(locality model)**: 이 모델에 의하면 프로세스는 실행되는 동안 한 지역에서 다른 지역으로 이동한다. 여기서 지역이란 현재 활동적으로 사용하고 있는 페이지들의 집합을 말한다. 현재 프로세스의 지역을 수용할 수 있는 만큼의 프레임을 할당해주면 스레싱은 발생하지 않는다.

10.6.2 작업 집합 모델

- 작업 집합 모델(working-set model)은 지역성 가정에 기반한 모델이다.
- 이 모델에서는 Δ로 표기되는 작업 집합 창을 정의한다.
- 작업 집합이란 가장 최근에 참조한 Δ 페이지의 집합을 말한다.
- 예) $\Delta = 10$



<그림 10.6> 작업 집합 모델

- 작업 집합 모델의 정확성은 Δ의 선택에 의해 결정된다. 작으면 현재의 지역을 충분히 나타내지 못하며, 크면 다른 지역과 중첩될 수 있다.
- 작업 집합 모델의 구현
  - 운영체제는 각 프로세스의 작업 집합을 감시하여 각 프로세스의 현재 작업 집합을 수용할 수 있는 충분한 수의 프레임을 각 프로세스에게 할당해준다.
  - 새 프로세스를 수용할 만큼의 빈 프레임이 존재하면 새 프로세스를 실행해주며, 반대로 각 프로세스의 작업 집합 크기의 총합이 사용가능한 프레임 수를 초과하면 하나의 프로세스를 일시 중단한다.
  - 이 방법의 가장 큰 어려움은 각 프로세스의 작업 집합을 유지하는 것이다. 참조 역사를 이용하여 유지할 수 있다.

10.6.3 페이지 결함 빈도

- 페이지 결함 빈도에 따라 할당되는 프레임 수를 조절하는 방법이다.
- 페이지 결함률의 상한과 하한을 정의하여, 프로세스의 결함률이 상한보다 높으면 프레임을 추가로 할당해주고, 반대로 하한보다 낮으면 프레임을 제거한다.
- 페이지 결함률이 상한을 넘었지만 더 이상 할당해줄 빈 프레임이 없으면 프로세스를 일시 중단해야 한다.

10.7 윈도우 NT

- 윈도우 NT는 요구 페이징과 클러스터링(clustering)을 이용하여 가상기억장치를 구현한다.
- 클러스터링은 결함된 페이지뿐만 아니라 주변 페이지까지 함께 적재한다.
- 프로세스가 시작되면 작업 집합 최저와 최대를 할당한다.
- 작업 집합 최저란 프로세스에게 항상 보장해주는 프레임의 수를 말한다.
- 작업 집합 최대란 그 프로세스에게 할당할 수 있는 최대 프레임 수를 말한다.
- 가상기억장치 관리자는 빈 프레임의 목록을 유지한다. 또한 빈 프레임 수가 충분히 있는지 여부를 알기 위해 임계값(threshold)을 사용한다.

- 작업 집합 최대 이하의 프레임이 할당된 프로세스에게 페이지 결함이 발생하면 빈 프레임 목록에서 하나의 프레임을 프로세스에게 할당해준다.
- 남은 빈 프레임의 수가 임계값 이하로 내려가면 자동 작업 집합 조절(automatic working-set trimming) 기법을 이용한다. 이 기법은 프로세스에게 할당된 프레임 수를 검사하여 작업 집합 최저 이상으로 할당된 프레임을 회수한다.

10.8 다른 고려

10.8.1 선페이징

- 순수 요구 페이징의 문제점은 초기에 많은 페이지 결함이 발생한다는 것이다.
- 또한 스왑아웃된 프로세스를 다시 실행할 때에도 같은 현상이 발생한다.
- **선페이징(prepaging)**은 초기에 높은 페이지 결함률을 줄이기 위한 방법이다.
- 작업 집합 모델을 사용할 경우에는 스왑아웃할 때 작업 집합 모델에 포함된 페이지 목록을 보관하였다고 그 프로세스가 스왑인될 때 이 페이지를 모두 적재한 다음에 프로세스를 재개한다.
- 선페이징의 비용이 이들에 대한 페이지 결함을 처리하는 비용보다 적어야 효과가 있다. 선페이징을 통해 적재된 페이지들이 사용되지 않으면 효과가 없다.
- s개의 페이지를 선페이징하였을 때 이중 α 비율만 사용되었다고 하자. 그러면 $s \times \alpha$ 페이지 결함을 처리하는 비용보다 $s \times (1-\alpha)$ 페이지를 선페이징하는 비용이 적어야 효과가 있다.

10.8.2 페이지의 크기

- 기존 시스템의 페이지 크기는 변경할 수 없다. 하지만 새 시스템을 설계할 때에는 페이지 크기를 결정해야 한다.
- 페이지 크기가 클수록 페이지 테이블의 크기는 작아진다. 이 측면에서는 페이지 크기가 클수록 좋다.
- 페이지 크기가 클수록 마지막 페이지에 존재하는 내부 단편화가 크다.
- 입출력 시간을 최소화하기 위해서는 페이지 크기가 클수록 좋다. 적은 양을 옮기나 많은 양을 옮기나 둘 다 같은 접근 시간이 소요되며, 전송 시간은 접근 시간보다 상대적으로 적은 시간이다.
- 입출력의 전체 크기는 페이지 크기가 작을 수록 좋다. 이것은 페이지 크기가 작을 수록 지역성이 향상되어 불필요한 것이 주기억장치에 적재될 확률이 낮아지기 때문이다.
- 페이지 결함률은 페이지가 클수록 낮다. 페이지 결함이 낮을수록 페이지 결함을 처리하는 오버헤드가 적어진다.
- 큰 페이지를 사용하는 것이 추세다.

10.8.3 TLB 범위

- TLB의 적중률(hit ratio)은 페이지 테이블 대신에 TLB에서 가상주소 번역이 이루어지는 비율을 말한다. TLB의 적중률은 TLB의 항 수를 증가시키면 높아지지만 비용이 비싸다.
- TLB 범위(reach)란 TLB를 통해 접근 가능한 주기억장치 공간을 말하며, TLB의 항 수에 페이지 크기를 곱한 값이다. TLB의 항수가 많을 수록 범위가 넓어진다.
- 최소한 프로세스의 작업 집합은 TLB에 저장되어 있어야 한다.
- 항 수를 변경하지 않아도 페이지 크기를 늘리면 TLB 범위는 넓어진다. 그러나 내부 단편화가 증가할 수 있다. 따라서 프로세스마다 다른 페이지 크기를 사용할 수 있도록 해주는 시스템도 있다. 그러나 다양한 페이지 크기를 제공하면 주소 번역이 복잡해진다.

10.8.4 프로그램의 구조

- 요구 페이징은 사용자에게 투명하게 제공되지만 사용자가 이 사실을 알고 이것을 프로그래밍에 적용하면 보다 효율적인 프로그램을 작성할 수 있다.
- 예) 페이지 크기가 128 워드일 때 다음과 같은 프로그램이 있다고 하자.

```
int A[128][128], i, j;
for(j=0; j < 128; j++)
    for(i=0; i<128; i++)
        A[i][j]=0;
```

배열은 행 중심으로 저장되기 때문에 이와 같이 하면 $128 \times 128 = 16,384$ 번의 페이지 결함이 발생한다. 이것을 다음과 같이 변경하면

```
for(i=0; i < 128; i++)
    for(j=0; j < 128; j++)
        A[i][j]=0;
```

한 행을 처리한 다음에 페이지 결함이 발생하므로 총 128번의 페이지 결함만 발생한다. 이 처럼 프로그래머가 페이징을 인식하고 프로그램을 작성하면 보다 효율적으로 프로그래밍을 할 수 있다.

- 페이징에서 효율적으로 사용할 수 있는 데이터 구조가 있고, 그렇지 않은 구조가 있다. 예를 들어 스택은 항상 스택 톱을 이용하여 조작하므로 페이징에 효율적인 구조이다. 반면에 해쉬 테이블은 접근위치가 널리 퍼져 있으므로 그렇지 않다.

- 컴파일러도 페이징을 고려하여 코드를 구성하면 보다 효율적인 프로그램을 만들 수 있다. 예를 들어 함께 자주 사용되는 루틴은 하나의 페이지에 포함하면 페이지 결함 수를 줄일 수 있다.

10.8.5 페이지 잠금

- 요구 페이징을 사용할 때 어떤 페이지들은 교체될 수 없도록 해야 하는 경우가 있다.
- 예) 입출력 요청한 프로세스가 입출력할 내용을 버퍼에 옮기고 입출력 대기 큐로 이동하였을 때 광역 교체 알고리즘을 사용하면 큐로 이동한 프로세스의 버퍼 페이지를 교체할 수 있다. 이 경우 엉뚱한 데이터가 입출력될 수 있다.
- 해결책
  - 입출력은 항상 시스템에게 할당된 주기억장치 영역에서 이루어지도록 한다. 추가 복사가 필요하다는 것이 이 해결책의 문제점이다.
  - 페이지 잠금 기법: 각 프레임마다 **잠금 비트(lock bit)**를 연관하여 교체의 희생자로 선택될 수 없도록 하는 방법이다.
- 입출력에 사용될 버퍼나 운영체제 커널에게 할당된 페이지는 잠금 비트를 설정하여 교체될 수 없도록 한다.
- 잠금 기법은 주의있게 사용해야 한다. 사용자 페이지를 잠금 다음에 다시 풀지 않으면 사용할 수 있는 프레임 수가 줄어들게 든다.