14.가상_메모리

14.가상_메모리

14.1. 연속 메모리 할당

- 프로세스들을 메모리에 연속적으로 할당할 때 고려해야 하는 것들과 잠재적 문제들에 대해 알아본다.

14.1.1. 스와핑

- 대기 상태의 프로세스, 오랫동안 사용되지 않은 프로세스와 현재 실행되지 않는 프로세스들을 임시로 보조기억장치 일부 영역으로 내보낸 후 다른 프로세스를 적재하는 방식
- 스왑 영역(swap space)
  - 이때, 쫓겨난 프로세스들이 들어가는 보조기억장치의 일부 영역
- 스왑 아웃(swap-out)
  - 실행되지 않는 프로세스가 메모리에서 스왑 영역으로 옮겨지는 것
- 스왑 인(swap-in)
  - 스왑 영역의 프로세스가 메모리로 옮겨오는 것
  - 이때 적재되는 물리 주소는 이전과 다른 주소일 수 있다.
- 스와핑을 이용해 프로세스들이 요구하는 메모리 주소 공간의 크기가 실제 메모리 크기보다 크더라도 프로세스들을 동시에 실행할 수 있다.

14.1.2. 메모리 할당

- 비어 있는 메모리 공간에 프로세스를 연속적으로 할당하는 방식으로 최초 적합, 최적 적합, 최악 적합 세 가지 방식이 있다.
- 최초 적합 first fit:
  - OS가 메모리 내 빈 공간을 탐색하다 적재할 수 있는 공간을 발견하면 프로세스를 배치하는 방식
  - 빠른 할당이 가능
- 최적 적합 best fit:
  - 빈 공간을 모두 탐색 후 프로세스가 적재될 수 있는 가장 작은 공간에 프로세스를 배치.
- 최악 적합 worst fit:
  - 빈 공간을 모두 탐색 후 프로세스가 적재될 수 있는 가장 큰 공간에 배치.

14.1.3. 외부 단편화

- external fragmentation
- 연속 메모리 할당으로 발생할 수 있는 문제
- 프로세스의 실행/종료 과정에서, 비어있는 메모리의 총 합에 비해 적재할 수 있는 프로세스의 크기가 제한될 수 있다.
- 즉, 비어있는 메모리가 총 50MB이지만 5MB씩 분할되어 있다면 적재할 수 있는 프로세스의 최대 크기는 5MB가 되는 셈이다.
- 요약하자면, 프로세스 할당이 어려울 만큼 작은 메모리 공간들로 메모리가 낭비되는 현상을 의미한다.
- 압축 compaction:
  - 외부 단편화를 해결할 수 있는 대표적 방안
  - 메모리 조각 모음이라고도 부름
  - 메모리 내 저장된 프로세스를 적당히 배치시켜 흩어져 있는 빈 공간들을 하나로 모으는 방식.
  - 이 과정에서 시스템은 다른 모든 작업을 중지시켜야 하며, 내용을 옮기는 작업에서 오버헤드가 발생한다.

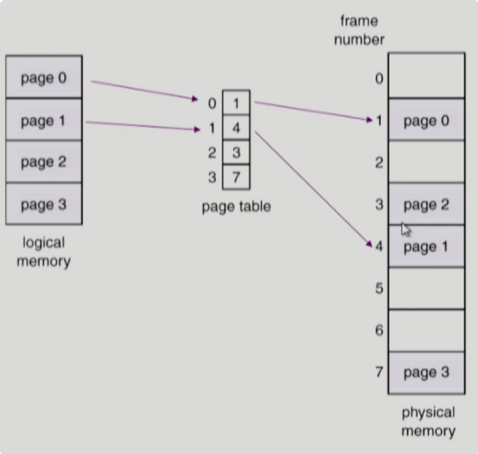
14.2. 페이징

- 연속 할당 방식의 두 가지 문제, 외부 단편화와 물리 메모리보다 큰 프로세스를 실행할 수 없는 문제를 해결하기 위해 사용하는 방법
- 가상 메모리 virtual memory:
  - 실행하려는 프로그램 일부만 메모리에 적재해 실제 물리 메모리보다 큰 프로세스를 실행할 수 있게 하는 기술
  - 현대 대부분의 운영체제가 채택하는 페이징 기법 외에 세그멘테이션 기법도 있다.

14.2.1. 페이징?

- 프로세스의 논리 주소 공간을 페이지 단위로 자르고, 물리 주소 공간을 프레임(페이지와 동일한 크기의 단위) 단위로 자른 뒤 페이지를 프레임에 할당하는 가상 메모리 관리 기법.[1]
- 프로세스 전체가 아닌 페이지 단위로 스왑 아웃/인 되며, 이를 페이징 시스템에서는 페이지 아웃/인 이라 부르기도 한다.
- 하나의 프로세스를 실행하기 위해 프로세스의 전체가 적재될 필요가 없다.

14.2.2. 페이지 테이블



- 불연속적으로 배치된 페이지들에서 '다음 실행할 명령어'를 찾기 위해 고안된 방식
- 물리주소에 불연속적으로 배치되더라도 논리 주소에 연속적으로 배치할 수 있다.
- 페이지 번호와 프레임 번호를 짝지어 주는 일종의 이정표로써의 역할.
- 프로세스 마다 테이블이 있으며, 이들은 메모리에 적재된다.[2]
- CPU 내의 페이지 테이블 베이스 레지스터(PTBR: Page Table Base Register)는 각 프로세스의 페이지 테이블이 적재된 주소를 가리킨다.
- TLB (translation lookaside buffer):
  - 메모리에 페이지 테이블을 두면 메모리 접근 시간이 두 배로 든다.
  - 이를 해결하기 위해 CPU 곁에 (MMU 내에) TLB라는 페이지 테이블의 캐시 메모리를 둔다.
  - 이곳에 페이지 테이블의 일부 내용을 저장하고, 참조 지역성에 근거해 주로 최근에 사용한 페이지 위주로 가져와 저장한다.
- TLB Hit
  - CPU가 발생한 페이지 번호가 TLB에 있을 경우
  - 메모리 접근 한 번으로 프레임에 바로 접근한다.
- TLB miss
  - TLB에 찾는 페이지 번호가 없는 경우 메모리 내의 페이지 테이블에 접근한다.

14.2.3. 페이징 주소 변환

- 하나의 페이지(프레임)은 여러 주소를 포괄하기에 어떤 페이지(프레임)에 접근하는지, 접근 주소가 페이지(프레임)에서 얼마나 떨어져 있는지 확인해야 한다.
- 때문에 논리 주소는 페이지 번호와 변위로 이루어진다. 즉, CPU가 32비트를 내보냈다면 N비트는 페이지 번호, 32-N 비트는 변위로 구성된다.
- 논리주소와 물리 주소의 변위는 같다.

14.2.4. 페이지 테이블 엔트리

- 페이지 테이블의 각 행
- 유효비트 valid bit
  - 해당 페이지에 접근 가능 여부
  - 현재 페이지의 위치, 즉 메모리 혹은 보조기억장치에 있는지를 알려준다.
  - CPU가 메모리에 적재되지 않은 페이지에 접근하려 하면 페이지 폴트(page fault) 예외를 발생시키며, 다음과 같이 처리한다.
    1. CPU는 기존 작업을 백업
    2. 페이지 폴트 처리 루틴 실행
    3. 원하는 페이지를 메모리에 가져온 후 유효비트를 1로 변경

    4. 이후 CPU에 할당
- 보호 비트 protection bit
  - 페이지의 읽기/쓰기 가능 여부
  - 0일 경우 읽기만 가능, 1일 경우 읽기/쓰기 가능
  - r(read), w(write), x(execute) 등 세 개의 비트 `000 ~ 111` 로도 표현 가능.
- 참조 비트 reference bit
  - CPU가 페이지에 접근한 적이 있는지 여부
  - 한 번이라도 접근한 적이 있다면 1, 없다면 0
- 수정 비트 modified bit
  - 데이터 수정 여부
  - dirty bit라고도 함
  - 0이라면 한 번도 접근한 적 없거나 읽기만 한 페이지임을 나타냄
  - 페이지가 메모리에서 사라질 때 보조기억장치에 쓰기 작업 여부를 확인하기 위해 필요하다.

14.2.5. 참고

- Copy on write
  - fork에서는 자식 프로세스가 부모 프로세스의 모든 자원을 복사했지만, 쓰기 시 복사에서는 페이지 테이블을 공유한 후 데이터를 수정할 경우에만 새로운 페이지에 복사본을 만들어 수정하기 때문에 메모리 및 프로세스 생성 시간을 절약할 수 있다.
- 계층적 페이지
  - hierarchical paging (multilevel page table)
  - 프로세스 크기가 커짐에 따라 프로세스 테이블이 증가되었고, 이때 모든 페이지 테이블 엔트리를 메모리에 적재하는 것은 비효율적이다.
  - 페이지 테이블을 페이징 하여 여러 단계의 페이지를 두는 방식
  - 즉, 페이지 테이블을 가리키는 페이지 테이블을 만든다.
  - 이 경우 논리 주소는 `바깥 페이지 번호 | 안쪽 페이지 번호 | 변위` 로 구성된다.
  - 주소 변환은 1. 바깥 페이지 번호로 페이지 테이블이 저장된 페이지를 찾고, 2. 페이지 테이블의 페이지에서 프레임 번호를 찾아 변위를 더해 물리 주소에 접근한다.
  - 계층이 많아질 수록 페이지 폴트에서 메모리 참조 횟수가 많아지는 오버헤드가 발생할 수 있다.

14.3. 페이지 교체와 프레임 할당

14.3.1. 요구 페이징

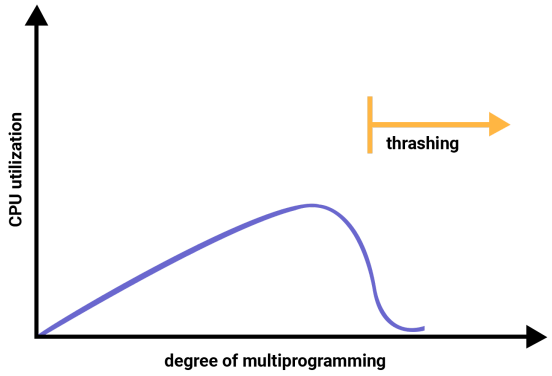
- demand paging
  - 실행에 요구되는 페이지만 적재
  1. CPU가 특정 페이지에 접근하는 명령어를 실행
  2. 해당 페이지가 메모리에 있을 경우 CPU는 페이지가 적재된 프레임에 접근
  3. 메모리에 없을 경우 페이지 폴트
  4. 폴트 처리 루틴은 해당 페이지를 메모리에 적재하고 유효 비트를 1로 변환
  5. 다시 `1.` 을 수행
- 순수 요구 페이징 pure demand paging
  - 아무런 페이지도 적재하지 않은 채 실행시켜, 실행 시작부터 페이지 폴트가 발생하고 필요한 페이지가 어느 정도 적재된 이후부터 페이지 폴트 발생 빈도가 낮아지는 방식.

14.3.2. 페이지 교체 알고리즘

- 쫓아 낼 페이지를 결정하는 알고리즘
- 페이지 폴트를 최소화 하기 위한 방법
- 이를 위해서는 페이지 참조열을 통해 페이지 폴트 횟수를 알아야 한다.
- 페이지 참조열
  - 참조하는 페이지 중 연속된 페이지를 생략한 페이지열.
  - CPU가 `2 2 2 3 5 5 5 3 3 7` 순서로 페이지에 접근했다면, 순서열은 `2 3 5 3 7` 이 된다.
  - 연속된 페이지 생략하는 이유는 중복된 페이지를 참조하는 것은 페이지 폴트를 발생시키지 않기 때문이다.
- FIFO
  - 메모리에 가장 먼저 올라온 페이지부터 내쫓는 방식
  - 구현은 간단하지만, 프로그램 생명 주기 중 계속 사용하는 내용이 있다면 지속적으로 폴트가 발생한다.

    - 이를 개선한 2차 기회 페이지 교체 알고리즘(second chance page replacement algorithm)은 참조 비트가 0이면서 가장 오래된 페이지를 내쫓고, 가장 오래됐지만 참조 비트가 1인 페이지는 참조 비트를 0으로 변경하기만 한다.
- 최적 페이지 교체
    - optimal page replacement algorithm
    - 참조 횟수를 고려하여 가장 오랫동안 사용되지 않을 페이지를 예측하여 쫓아내는 방식.
    - 단, 구현이 불가능에 가깝기 때문에, 페이지 폴트의 하한선으로 간주하고 다른 알고리즘을 평가할 때 사용한다.
- LRU
    - Least Recently Used Page Replacement Algorithm
    - 가장 오랫동안 사용하지 않은 페이지를 내쫓는 방식.
    - 페이지마다 마지막으로 사용한 시간을 토대로 가장 사용이 적었던 페이지를 교체.

14.3.3. 스래싱과 프레임 할당



- 페이지 폴트에서, 페이지 교체 알고리즘 보다 프레임의 절대적 부족이 더 큰 원인이라 할 수 있다.
- 폴트가 자주 발생하면 CPU 이용률이 저하되며, 생산성이 떨어지고 이런 문제를 스래싱 (thrashing)이라 한다.
- 그림 처럼 프로세스의 수를 늘린다 해서 CPU 이용률이 비례적으로 증가하지는 않으며, 일정 수준 이상 늘리면 이용률 감소로 성능이 저하된다.
- 스래싱의 근본 원인은 각 프로세스가 요구하는 최소한의 프레임 수가 보장되지 않았기 때문이며, 이를 해결하기 위해서는 적절히 프레임을 할당해야 한다.
- 균등 할당 equal allocation
    - 가장 단순한 방식으로, 프로세스 갯수만큼 프레임을 1/N 하여 할당한다.
    - 정적 할당 방식[3]
- 비례 할당 proportional allocation
    - 프로세스의 크기가 다양한 만큼 프레임 요구량도 다르며, 더 큰 프로세스에 더 많은 프레임을 할당하는 방식.
    - 정적 할당 방식

---

1. 외부 단편화는 해결했지만, 내부 단편화를 야기할 수 있다. 모든 프로세스가 페이지 단위에 맞게 잘리는 것(프로세스는 페이지의 배수)은 아니며, 만약 페이지가 10kb 이고 프로세스가 108kb 라면 마지막 페이지는 2kb가 남는 현상이 발생한다.↩
2. 페이지 테이블 정보들은 각 프로세스의 PCB에 기록된다.↩
3. 프로세스의 크기와 물리 메모리의 크기만 고려한 방식. 이외에 프로세스의 실행 과정을 고려하는 동적 할당 방식(Dynamic memory allocation)이 있다. 이 방식은 유연하며 효율적 메모리 사용이 가능하지만, 할당/해제 과정에서 오버헤드, 메모리 누수, 포인터 역참조(주소를 통해 값에 접근) 오류(메모리에 남는 공간이 없는 상) 등의 문제가 있다.↩