가상메모리

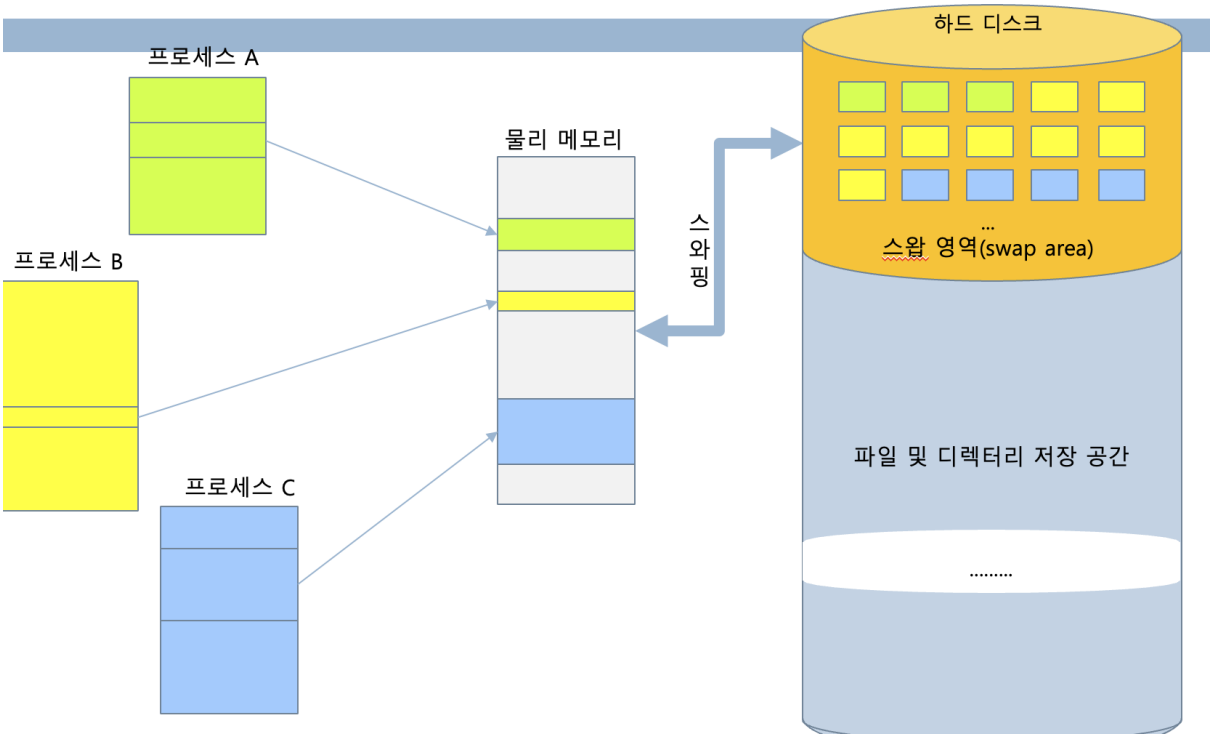
물리 메모리의 한계

설치된 물리 메모리보다 큰 프로세스를 실행시킬 수 있을까?
프로세스들을 합친 크기가 설치된 물리 메모리보다 클 때 이들을 실행시킬 수 있을까?

→ 프로세스 전체가 물리 메모리에 꼭 적재되어야 하는가?
→ 실행에 당장 필요한 프로세스의 일부 메모리만 적재한 채로 실행 시킬 수는 없나?

가상 메모리란?

- 위 질문과 같은 물리 메모리의 한계를 극복하기 위한 해결책
- 가상 메모리 기법
  - 물리 메모리를 디스크 공간으로 확장
    - 물리 메모리 ~ 하드디스크로 메모리 연장
    - 프로세스를 물리 메모리와 하드 디스크에 나누어서 저장
    - 사용자나 실행되는 프로세스 입장에서는 실행하기에 충분히 큰 메모리가 존재한다고 착각하게 만드는 메모리 관리 기술임
  - 스외핑
    - 메모리가 부족할 때, 실행에 필요하지 않는 부분은 하드 디스크로 이동
    - 실행에 필요한 경우 하드디스크로부터 물리 메모리로 이동 시킨다.
    - 물리 메모리를 확장하여 사용하는 디스크 영영을 스왑영역이라고 함
      - swap out : 물리메모리 → 스왑 영역
      - swap in : 스왑 영역 → 물리 메모리
- 가상 메모리는 운영체제마다 구현 방법이 다름



가상 메모리 구현 방법

- demand paging (요구 페이징)
  - 페이징 기법을 토대로 프로세스 일부 페이지들만 메모리에 할당,
  - 페에지가 필요할 때 메모리를 할당받고 페이지를 적재시키는 메모리 관리 기법
  - 페이징 + 스와핑
- demand segmentation
  - 세그먼테이션 + 세그먼트 스와핑

가상 메모리에 대한 문제

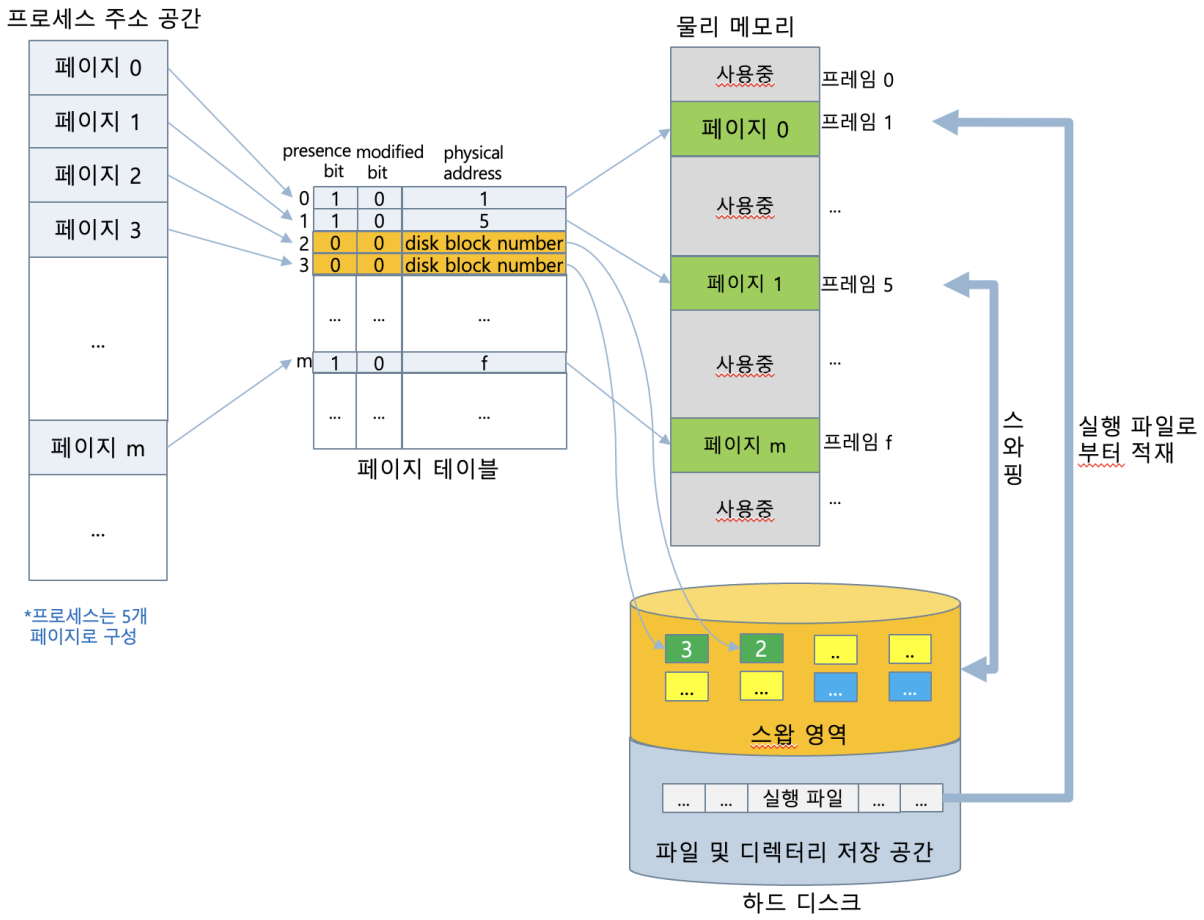
- 스레싱 문제 : 물리 메모리 - 디스크의 스왑영역 사이에 입출력이 너무 빈번하지 않나?
- 페이지 테이블 : 페이지 테이블은 어떻게 구성?
- 페이지 폴트 : 가상 주소를 물리주소로 변환할 때 페이지가 프레임에 없는 경우는?

- 페이지 할당 : 프로세스의 어떤 페이지를 물리 메모리에 두고 어떤 페이지를 하드 디스크에 둘건지?
- 스왑영역 : 스왑영역의 크기는?
- 작업 직합 : 프로세스는 일정 시간 범위에서 실행 중에 몇 개의 프레임을 실제로 사용하고 있는지?
- 페이지 교체 알고리즘 : 필요한 페이지를 읽어 오기 위해 프레임 중 하나를 비울때 어떤 프레임을 비울지

요구 페이징

- 요구 페이징
  - 현재 실행에 필요한 일부 페이지만 메모리에 적재하고 나머지는 하드 디스크에 두며 페이지가 필요할 때 메모리에 적재하는 방식
- 요구 페이징 구현
  - 첫 페이지만 물리 메모리에 적재
  - 실행 중 다음 페이지가 필요하면 그때 적재시키는 방법
- 스왑 영역
  - 메모리가 부족할 때, 메모리를 비우고 페이지를 저장해두는 하드 디스크 영역
    - 리눅스 : 스왑 파티션에 구성
    - 윈도우 : C:pagefile.sys
- 페이지 테이블 항목
  - presence/ valid bit : 해당 페이지가 물리 메모리에 있는지?
    - 비트 1이면 해당 페이지 프레임 번호의 메모리에 존재
    - 비트 0이면 해당 페이지 디스크에 존재
  - modified/dirty bit : 해당 페이지 수정 여부
    - 비트 1이면 페이지가 프레임에 적재 후 수정됨
    - 비트 0아면 페이지 수정X
  - physical address
    - presence bit가 1이면 해당 페이지가 적재되어 있는 프레임 번호

        - presence bit가 0이면 해당 페이지가 있는 디스크 블록 번호
- 페이지 폴트
    - CPU가 access 하려는 페이지가 물리 메모리에 없는 경우 발생
    - 페이지 폴트 발생 시 빈 프레임 할당 후 스왑 영역이나 실행 파일로부터 페이지 적재한다.



요구 페이징

페이징 폴트

- CPU가 발생시킨 가상 주소 페이지가 메모리 프레임에 없는 상황
- MMU가 가상 주소를 물리 주소로 바꾸는 과정에서 발생한다.
- 처리
    - 테이블 확인 → invalid reference → abort process

    - 비어있는 frame 찾기
    - 해당 페이지 disk에서 memory로 읽어옴
    - 프로세스가 cpu 잡고 다시 재개
    - 중단된 instruction 재개

비어있는 프레임이 없는 경우

- page replacement
    - 어떤 프레임을 빼앗을지 결정
    - 곧바로 사용되지 않을 page를 쫓아내는 것이 좋다
- replacement algorithm
    - 페이지 폴트 최소화를 목표로 프로세스 작업 집합에 포함될 페이지들을 수용할만한 개수의 프레임을 할당한다.

페이지 교체 알고리즘

- 최적 교체 알고리즘 : 가장 먼 미래에 사용될 페이지를 교체 대상으로 결정
- FIFO : 가장 오래전에 적재된 페이지 선택
- LRU : 가장 최근에 사용되지 않았던 페이지 선택
- Clock : LRU 단순화

Optimal page Replacement

- 가장 먼 미래에 사용될 페이지를 교체 대상으로 결정함
- 미래 페이지 사용 패턴을 알 수없어 비현실적이다.

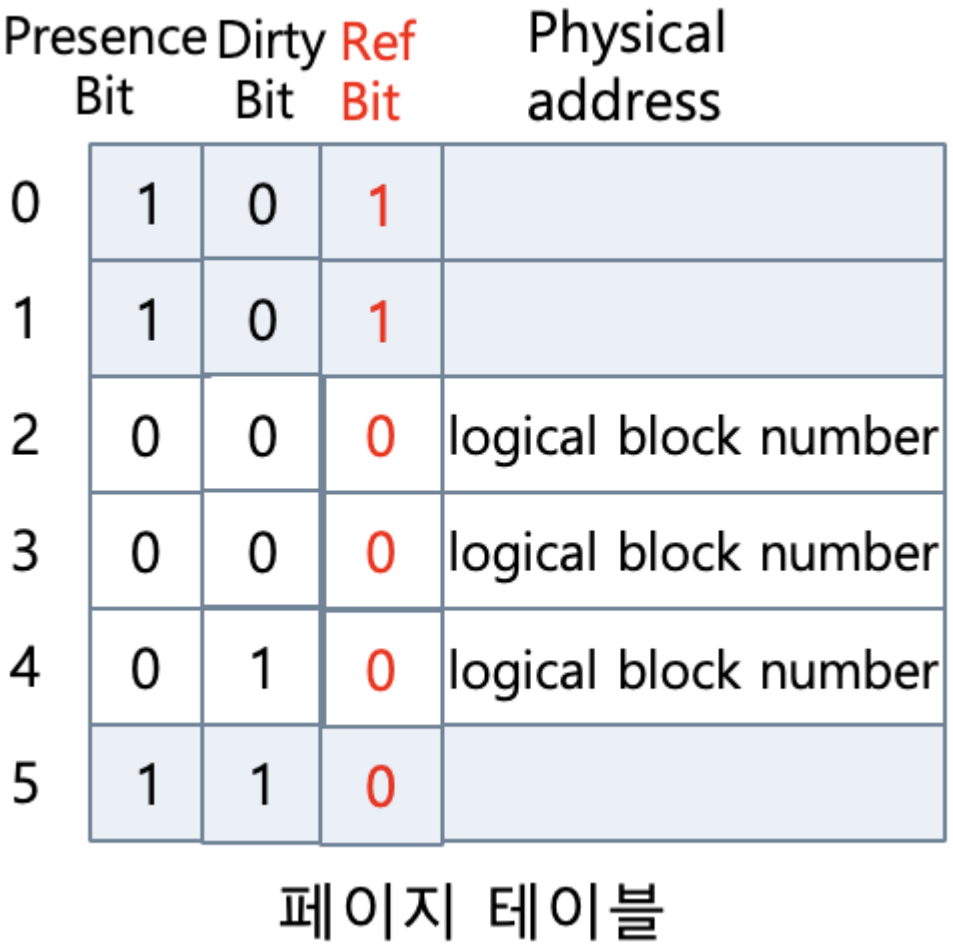
FIFO

- 페이지가 적재된 시간을 저장하여 가장 오래전에 적재된 페이지를 선택한다.
- 이해가 쉽고 구현이 단순하다.

- 다만 오래된 페이지에도 자주 사용되는 변수나 코드가 있을 수 있기에 성능면에서 떨어진다.

LRU

- 가장 최근에 사용되지 않는 페이지 선택함
- 좋은 알고리즘으로 평가받으며 많이 사용됨
- 구현 방식
  - 타임 스탬프 이용
    - 모든 프레임에 참조 시간 기록할 수 있는 비트 추가
    - cpu가 페이지 참조 시 마다 참조 시간을 기록한다.
  - 하드웨어 이용, 참조 비트 사용
    - 페이지 테이블 항목에 1개의 참조 비트 추가함 (Ref 비트)
      - ref 비트 1이면 최근 참조
      - 0이면 최근 참조 X
    - cpu가 페이지 참조 시 H/W로 참조 비트를 1로 설정



LFU

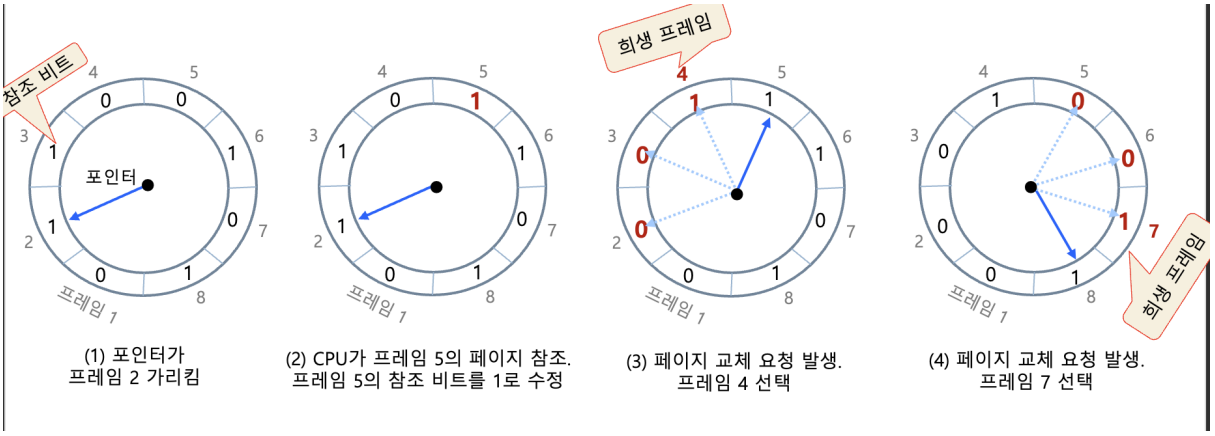
- 참조 횟수가 가장 적은 페이지를 선택
- 장점
  - LRU와 비교하여 참조시점 뿐만 아닌 장기적인 시간을 두고 보기에 보다 정확함
- 단점
  - LRU보다 구현 복잡함

Clock(second chance)

- 프레임당 1비트 참조비트 사용
  - 프레임을 원형 큐로 연결하여 관리함

- 페이지 참조 시 참조비트 1로 셋

- 희생 프레임 선택
  - 원형 큐에서 검색 시작하는 프레임 위치를 포인터라고 부름
  - 포인터에서 시작하여 시계방향으로 원형 큐를 따라 이동한다.
    - 참조 비트 0이면 해당 프레임 선택
    - 참조 비트 1이면 0으로 바꾸고 다음 프레임 이동
  - 한바퀴돌면 처음 프레임 선택



(1) 포인터가 프레임 2 가리킴
(2) CPU가 프레임 5의 페이지 참조. 프레임 5의 참조 비트를 1로 수정
(3) 페이지 교체 요청 발생. 프레임 4 선택
(4) 페이지 교체 요청 발생. 프레임 7 선택

프레임 할당

- 프로세스에게 작업집합에 포함된 페이지들을 적재할 충분한 메모리 할당
- 페이지 폴트를 줄이고 스레싱을 예방하는 것이 목적
- 방법
  - equal allocation
    - 균등할당
    - 프로세스에게 동일한 개수의 프레임 할당
    - 단순하지만 작은 프로세스에게는 프레임 낭비이며 큰 프로세스는 페이지 폴트 빈번한 발생가능성
  - proportional allocation
    - 비례할당
    - 프로세스 크기에 비례하여 프레임 할당

        - 장점은 페이지 폴트 줄임
        - 단점은 프로세스 크기를 어떻게 아는지, 실행 도중 작업 집합 판단해야함
    - priority allocation
        - 순위 할당
        - 우선순위에 따라 다르게 할당

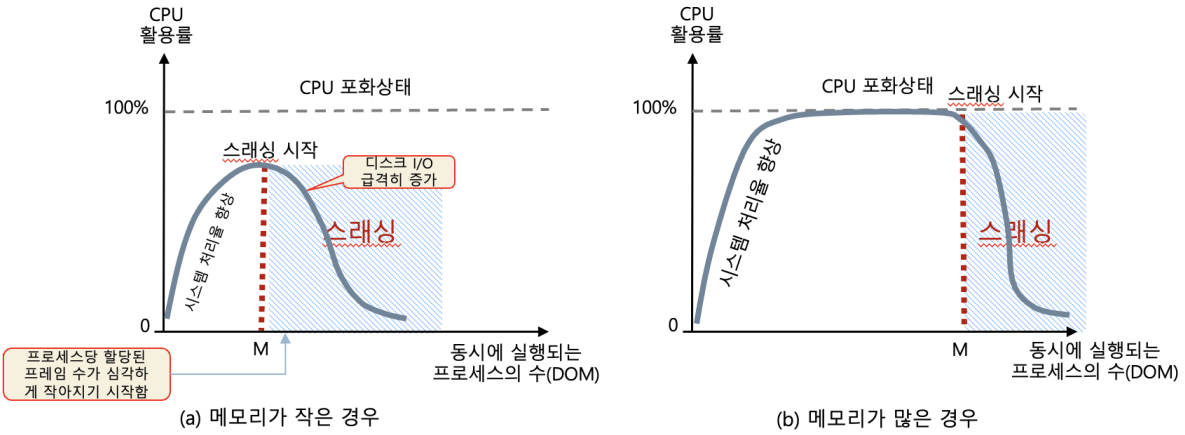
> 💡 고민해볼 사항
>
> 페이지 폴트 빈번하게 발생하면 시스템 성능 저하 ?
> → 스레싱이라고 부름, 프로그램 실행 초기에는 페이지 폴트 발생하지만 이후에는 필요한 페이지들이 메모리에 올라와서 적게 발생한다.
>
> 처음부터 한번에 로딩하면 안되나?
> → 어떤 페이지가 참조될지 모르는데 어떻게 올리나
>
> 프로세스에게 할당할 수 있는 메모리 프레임의 제한은?
> → 한 프로세스에게 할당되는 최대 메모리 프레임의 개수는 제한됨. 프로세스가 필요로 하는 모든 페이지에 대해 메모리 프레임을 할당할 수는 없다.
>
> 프레임수와 페이지 폴트의 관계?
> → 프레임의 개수가 많을수록 페이지 폴트 횟수는 줄어듬

스래싱

- 스레싱
    - 페이지 폴트가 계속 발생하여 메모리 프레임에 페이지가 반복적으로 교체되고 디스크 입출력이 급하게 증가하는 현상

      → CPU 활용률이 감소
    - 페이지 폴트 → 스래싱과정
        - 페이지 폴트 발생하면 스레드 실행 중단 후 디스크 입출력을 동반한 페이지 교체 발생

        - 페이지 교체는 메모리의 페이지 아웃, 아웃된 페이지 접근 시 다시 페이지 폴트 유발
        - 시스템은 계속 페이지 폴트 처리하느라 정신없고 스레드는 계속 대기
- 원인
    - 다중 프로그래밍 정도가 너무 과한경우
    - 잘못된 메모리 할당, 페이지 교체 알고리즘 적용
    - 메모리양이 너무 작으면
    - 갑자기 너무 많은 프로세스 실행
- 발생 시점
    - 다중 프로그램 정도(DOM)이 높아지면 CPU 활용률 증가하는데
    - 임계점을 넘어가면 스래싱이 발생한다.
- 해결 방법
    - 윈도우 : process explorer
    - 리눅스 : top, htop, vmstat



Working Set(작업 집합)

- 일정 시간 범위 내 프로세스가 참조한 페이지들의 집합
    - 작업 집합에 포함된 페이지들이 모두 메모리에 적재되어 있는 것이 프로세스 실행 최고 성능
- 페이지 폴트 : 작업 집합을 메모리에 적재하는 과정이다.

    - 참조의 지역성으로 인해 일정 시간 내 작업 집합 형성
- 참조의 지역성 → 페이지 폴트 → 메모리에 작업 집합 형성

> 💡 reference of locality (참조의 지역성)
>
> - cpu가 짧은 시간 내 일정 구간의 메모리 영역을 반복적으로 참조하는 현상
> - 현재 프로세스 실행 패턴을 관찰하면, 가까운 미래에 프로세스 코드와 데이터 사용을 합리적으로 예측하여 메모리 할당과 페이지 교체 전략에 활용한다.
>
> - 시간의 지역성 : ex) 반복문
> - 공간 지역성 : ex) 순차 읽기, 쓰기

- 작업 집합 이동
    - 프로세스가 실행되는 동안 작업집합 이동
    - 시간이 지나면 새로운 작업 집합 형성