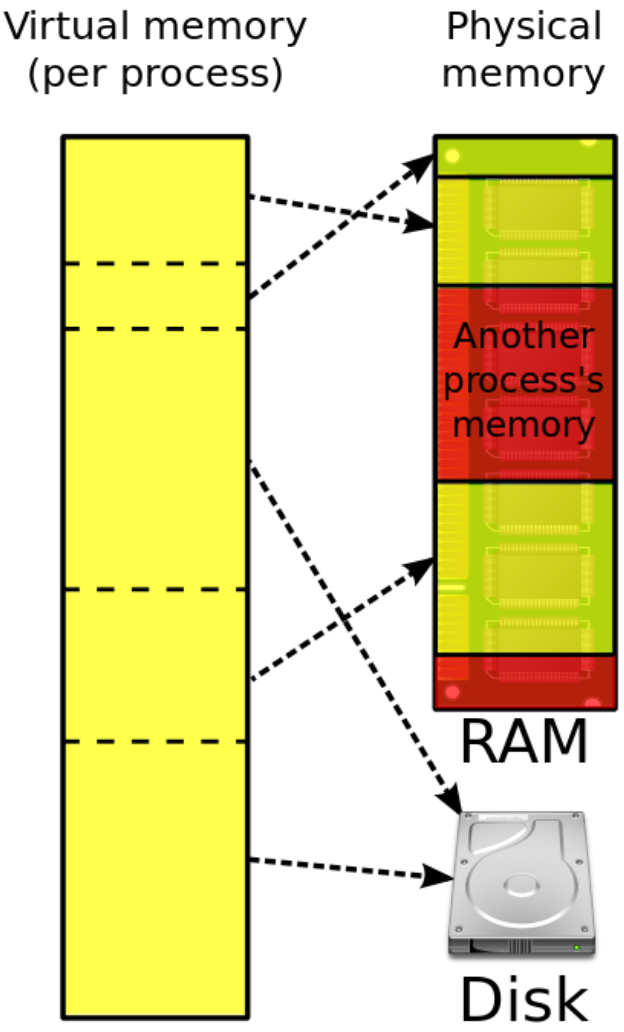
Virtual Memory

태그	OS
주차	6주차

목차

1. 가상 메모리



- 메모리 관리 기법 중 하나
- 컴퓨터 시스템에 실제로 이용 가능한 기억 자원을 이론적으로 추상화하여 사용자에게 큰 메모리로 보이게 만듦
- 프로그램의 실제 메모리가 아닌 가상의 메모리 주소를 주는 방식

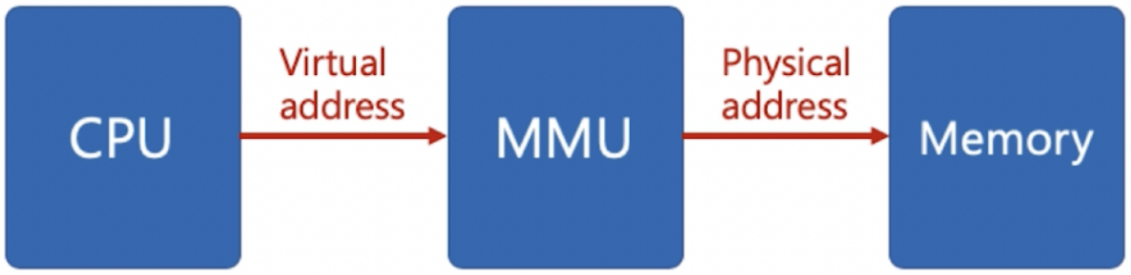
1-1. 동작

- 프로세스는 가상 주소를 사용
- 실제 주소에서 데이터를 읽고 쓸 때만 물리주소로 바꿈

가상 주소와 물리 주소

- 가상 주소: 프로세스가 참조하는 주소
- 물리 주소: 실제 메모리 RAM 주소

1-2. MMU(Memory Management Unit)



- 가상 메모리 주소와 물리 메모리 주소를 관리하는 장치
- 실제 물리주소를 접근할 때 가상 주소를 매칭된 물리 주소로 변환
- 하드웨어 장치
  - 소프트 웨어보다 하드웨어가 주소 변환이 더 빠르기 때문

2. 페이징 시스템(Paging System)

- 크기가 동일한 페이지로 가상 주소 공간과 매칭하는 물리 주소 공간을 관리
  - 프로세스가 실행될 때 필요한 공간만큼만 메모리에 적재시켜 실행
  - 이 때의 공간을 Page 단위로 나누어 적재
- 페이징 시스템은 하드웨어의 지원이 필요
- 페이지 단위
  - Intel x86(32-bit)CPU(하드웨어)
    - 4KB, 2MB, 1GB
  - LINUX(소프트웨어)
    - 4KB

2-1. 페이징 시스템 구조

1) 페이지 프레임(Page Frame)

- 고정된 크기의 페이지 블록(4KB)
- 구조
  - 페이지 번호 p: 가상 메모리 페이지
  - 변위(오프셋) d: p 안에서 참조하는 위치

> ☝🏻 **(LINUX) 프로세스가 4GB를 사용하는 이유**
>
> 리눅스에서는 프로세스 크기를 4GB로 고정하는데, 이는 32-bit 시스템에서는 2^{32}-bit, 즉 4GB의 시스템에서 모두 표현할 수 있기 때문

2) 페이지 테이블(Page Table)

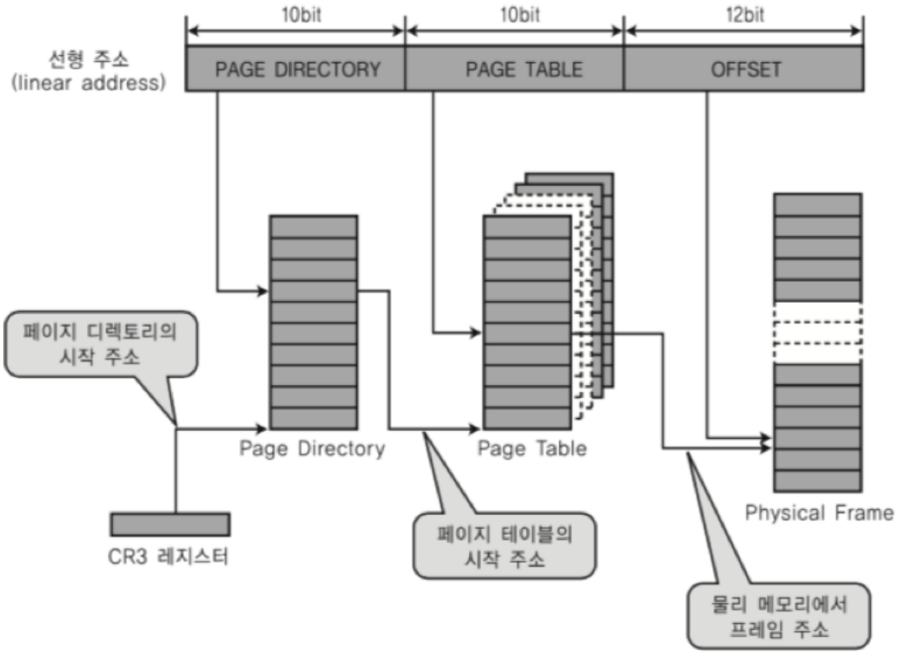
- PA에 있는 페이지 번호와 해당 페이지의 첫 PA 정보를 매핑한 표
- 프로세스에서 특정 VA로 접근하기 위해 다음 과정 거침
  1. 해당 프로세스의 PT에서 해당 VA가 포함된 페이지 번호가 있는지 확인
  2. 페이지 번호가 있다면, 이 페이지가 매핑된 첫 PA 알아냄 (p')
  3. p' + 오프셋(d)가 실제 PA가 됨
- 프로세스의 페이지는 항상 물리메모리에 적재될 필요가 없기 때문에 적재되지 않은 페이지는 PT상에 `valid/invalid bit`로 적재 여부 표시

2-2. 페이징 시스템과 MMU

- CPU는 가상 주소 접근 시 MMU 장치를 통해 물리 메모리 접근
- 프로세스가 생성되면 페이지 테이블 정보가 생성
  1. PCB 등에서 해당 페이지 테이블 접근 가능 & 관련 정보는 물리메모리에 적재

2. 프로세스 구동 시 해당 PT의 base 주소가 별도 레지스터인 CR3에 저장
3. CPU가 가상 주소를 접근하면 MMU가 PT의 base 주소에 접근해 PA 가져옴

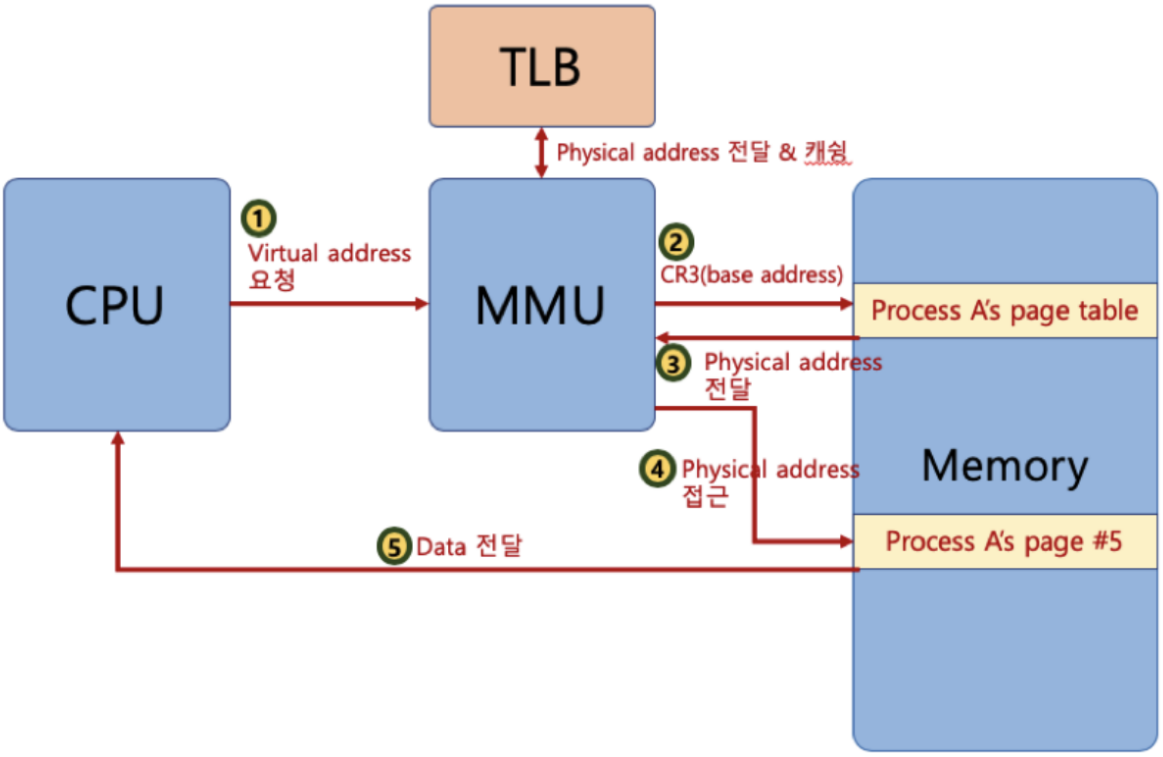
2-3. 다중 단계 페이징 시스템



- 실제 프로세스는 4GB를 넘는 경우가 거의 없음
- 따라서 페이지 정보를 단계별로 나눠서 생성
  - 필요 없는 페이지를 생성하지 않아 공간 절약
- 상위 10bit: Page Directory
- 다음 10bit: Page Table
- 마지막 12bit: Offset

> ☝🏻 **최근 리눅스는 4단계까지 나눔**

2-4. TLB(Translation Lookaside Buffer)

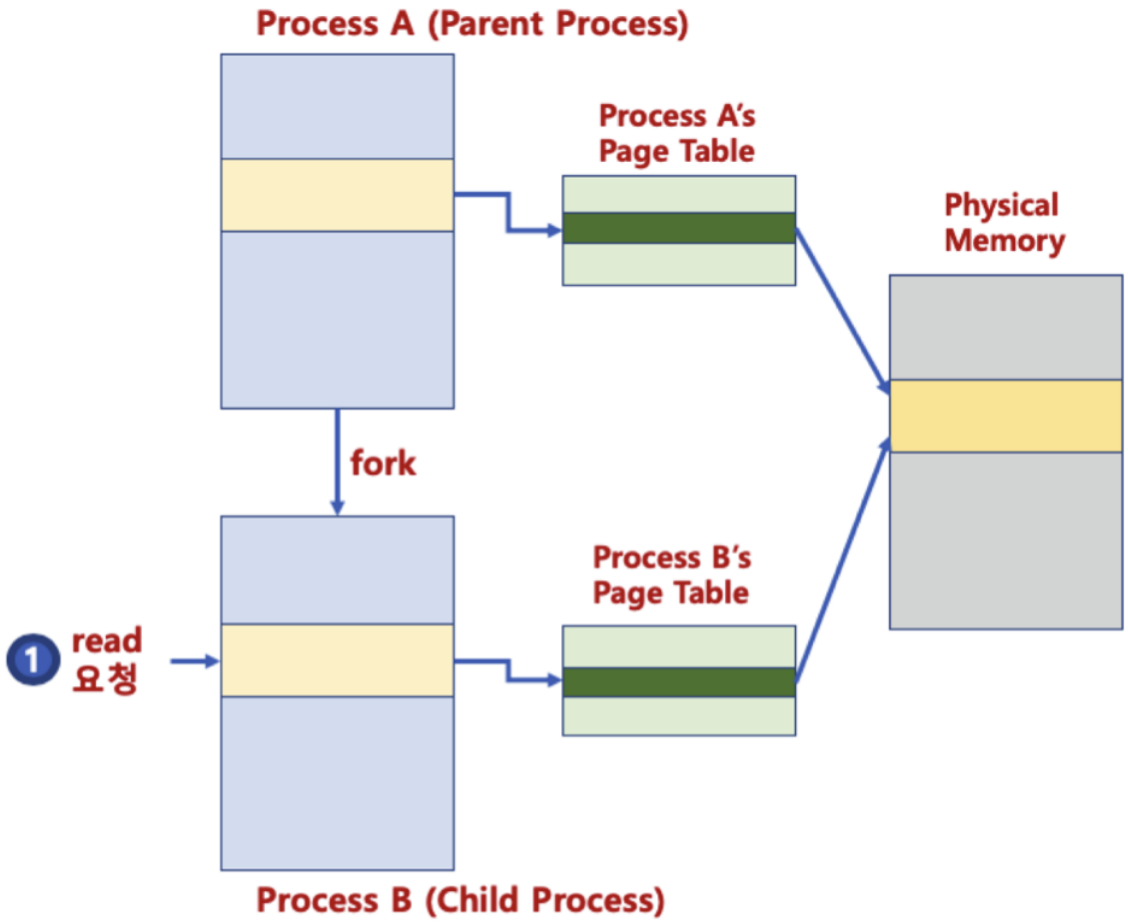


- 실제 물리 주소를 접근할 때 MMU 뿐 아니라 추가적인 하드웨어 보조장치인 TLB 활용
- 데이터에 접근할 때 마다 레지스터에 캐싱해 접근 속도 향상

1) 동작

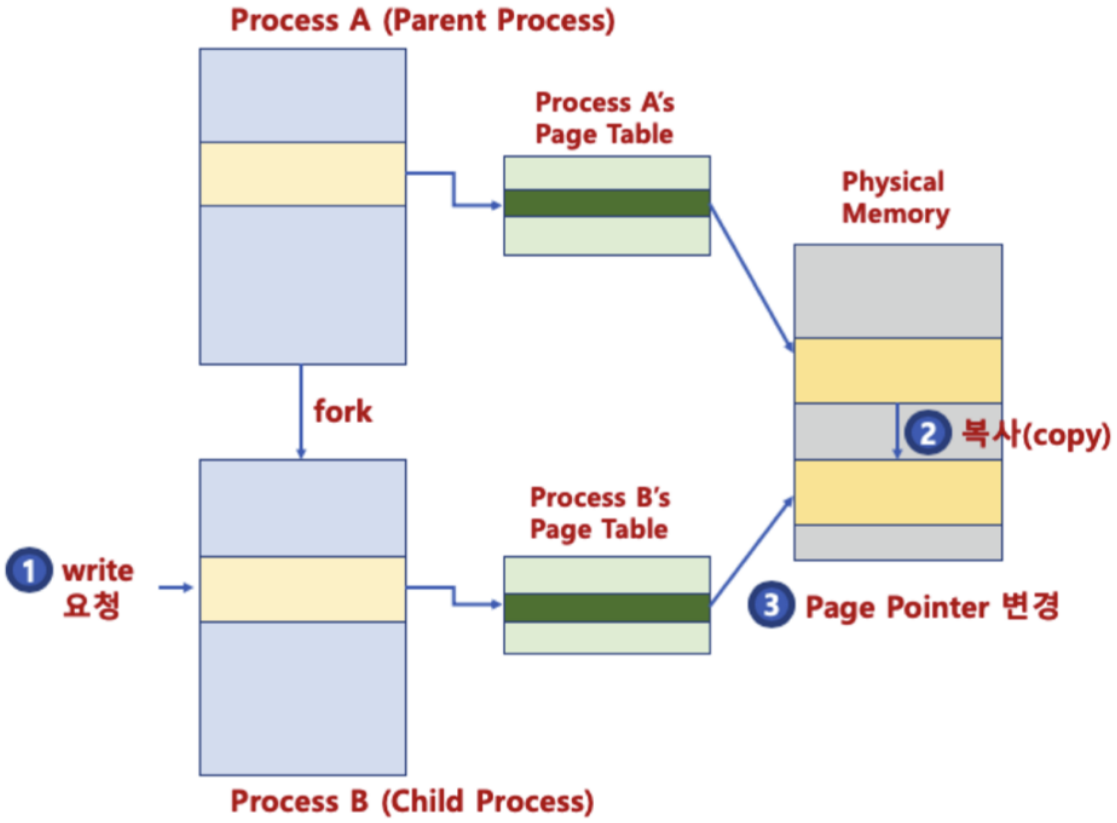
1. 가상 주소 요청 → MMU
2. MMU는 TLB에서 최근에 가상 주소에서 물리 주소로 변환한 정보가 있는지 확인
    a. 있다면 해당 정보 활용
    b. 없다면 CR3에 등록된 base 주소에 접근해 메인 메모리에서 데이터 찾아 물리주소 반환
3. 해당 정보를 TLB에 캐싱 후 해당 물리 주소를 기반으로 메인 메모리에 접근
4. 메인 메모리는 해당 데이터를 CPU에 적재하여 프로세스 수행

2-5. 페이징 시스템과 공유 메모리



- 프로세스는 동일한 물리 주소를 가리킬 수 있음
  - 실행중인 프로세스들이 동일한 로직을 수행한다면 동일한 공간에서 프로세스 수행
- fork() 사용 후 read만 수행 시 부모와 자식 프로세스는 물리메모리를 공유 메모리로 실행

1) COW(Copy-on-write)



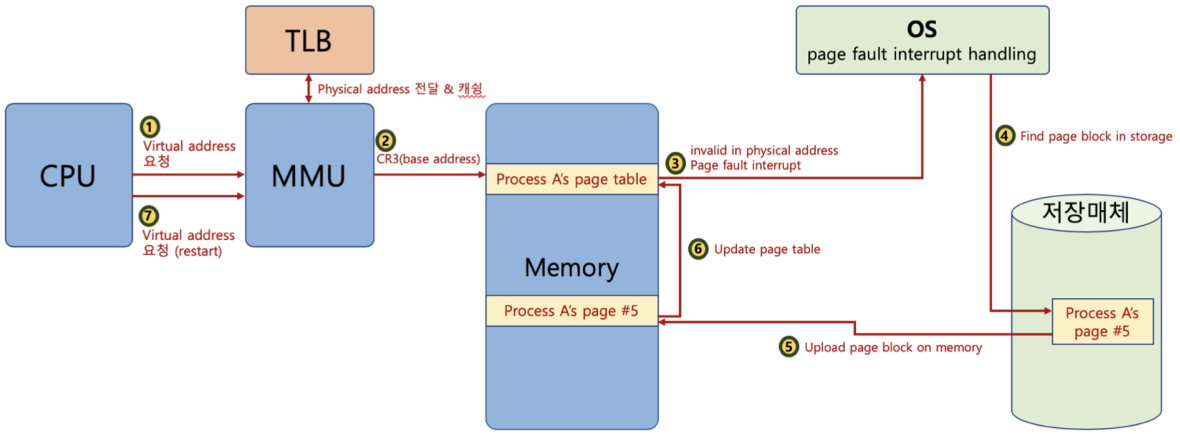
- 그러나 write 실행 시, 즉 데이터가 변경될 경우 부모와 자식은 서로 다른 로직을 수행
- 이 때 물리주소를 복사해 필요할 때만 공간 할당을 수행

2-6. 요구 페이징(Demand Paging)

- 가상 메모리는 프로세스의 모든 데이터를 적재하지 않고 필요한 시점에만 적재
- 필요하지 않은 페이지 프레임은 다시 저장매체에 저장해야 함
- 이 때 이를 위한 페이징 교체 알고리즘이 필요

> ☝🏻 **선행 페이징(Anticipatory Paging)**
>
> 미리 프로세스와 관련된 모든 데이터를 메모리에 올려놓고 실행하는 방식

1) 페이지 폴트(Page Fault)



어떤 페이지가 실제 물리 메모리에 없을 때 일어나는 인터럽트. page fault 발생 시 해당 페이지를 물리메모리에 적재

- Page Fault 발생 시 OS에게 Trap을 요청하고 이를 저장매체에서 불러와 메모리에 적재
- 그러나 메인 메모리 접근은 오버헤드가 크므로 향후 실행 및 참조될 코드와 데이터를 미리 물리 메모리에 올릴 수 있어야 함
- 이를 위한 페이지 교체 정책 필요

2) 페이지 교체 정책

- OS가 특정 페이지를 물리 메모리에 올릴 때 물리 메모리가 다 차면 다음과 같은 과정을 거침
  1. 기존 페이지 중 하나를 물리메모리에서 저장매체로 내림
  2. 새로운 페이지를 해당 공간에 올림
- 여기서 어떤 메모리를 내릴 것인지 결정하는 것이 페이지 교체 알고리즘
- 페이지 교체 알고리즘
  - OPT (Optimal Replacement)
    - 최적 교체 알고리즘
    - 가장 오랫동안 사용하지 않을 페이지를 내림
    - 실제로 구현은 불가
  - FIFO (First In First Out)
    - 처음에 들어온 페이지를 내리고 새 페이지를 올림
  - LRU (Least Recently Used)

        - 가장 최근에 사용한 페이지를 교체
        - 한번 사용한 페이지는 다시 사용하지 않을 것이라 가정
    - LFU (Least Frequently Used)
        - 가장 적게 사용한 페이지를 교체
        - 처음에 초기화한 중요 데이터가 있는 페이지를 교체할 가능성이 있음
    - NUR (Not Used Recently) = NRU (Not Recently Used)

NUR		read	read	read	read	read	write	write	read	write	read
Requested Page		1	2	3	2	1	3	1	2	1	4
Physical Memory		1	1	1	1	1	1	1	1	1	1
			2	2	2	2	2	2	2	2	4
				3	3	3	3	3	3	3	3

Page	참조비트	수정비트
1	1	1
2	1	0
3	1	1

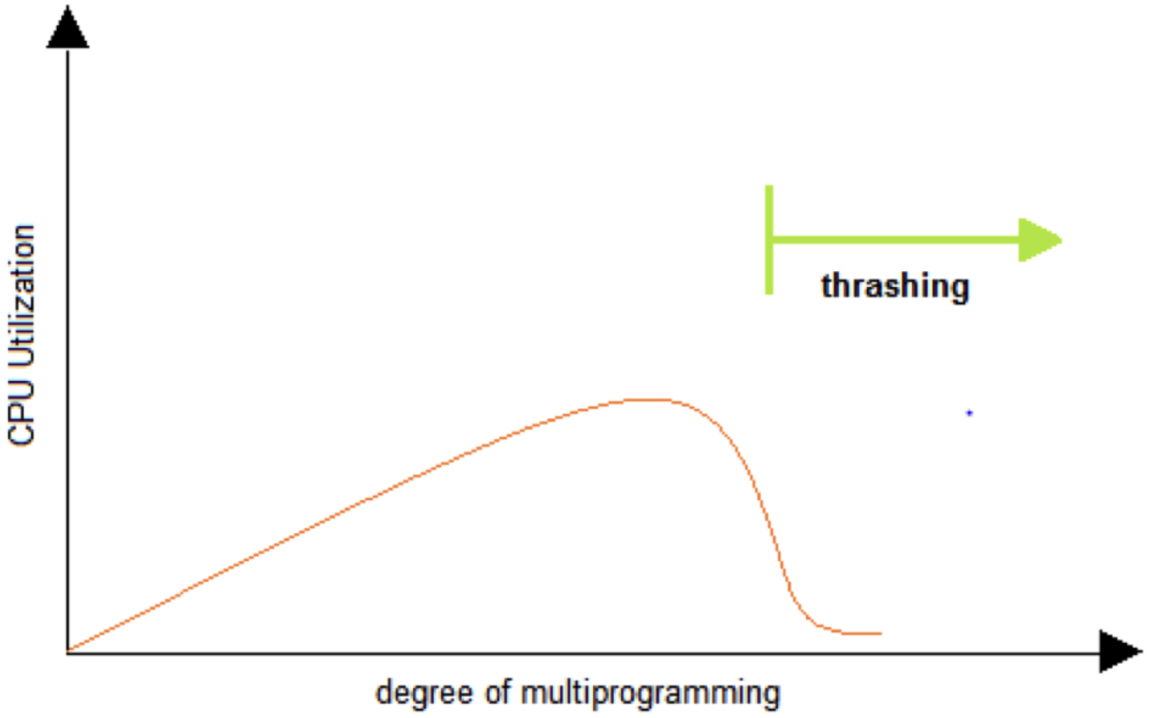
        - 각 페이지마다 참조비트 `R`, 수정 비트 `M`을 쌍으로 묶어 관리 참조 여부 → 수정 여부 순으로 페이지 교체

3) 데이터 지역성 (Data Locality)

- 캐시 메모리는 공간이 작아 어떤 데이터를 유지시킬지에 대한 논리가 필요
- **시간적 지역성(Temporal Locality)**
    - 한번 참조된 데이터는 잠시 후에 또 참조될 가능성이 높음
    - ex) for/while과 같은 반복문에 사용하는 조건 변수
- **공간적 지역성 (Spatial Locality)**
    - 같은 배열을 연속적으로 접근할 때 참조된 데이터 근처에 있는 데이터가 잠시 후에 사용될 가능성이 높음

☝🏻 **LRU 알고리즘은 공간적 지역성 특성을 잘 활용하여 교체 알고리즘으로 자주 사용됨**

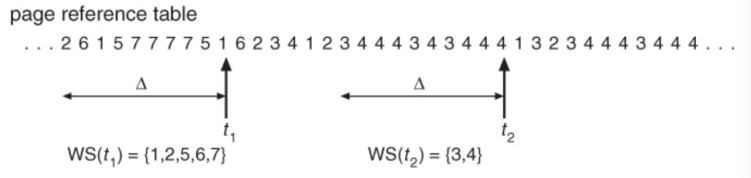
4) Thrashing

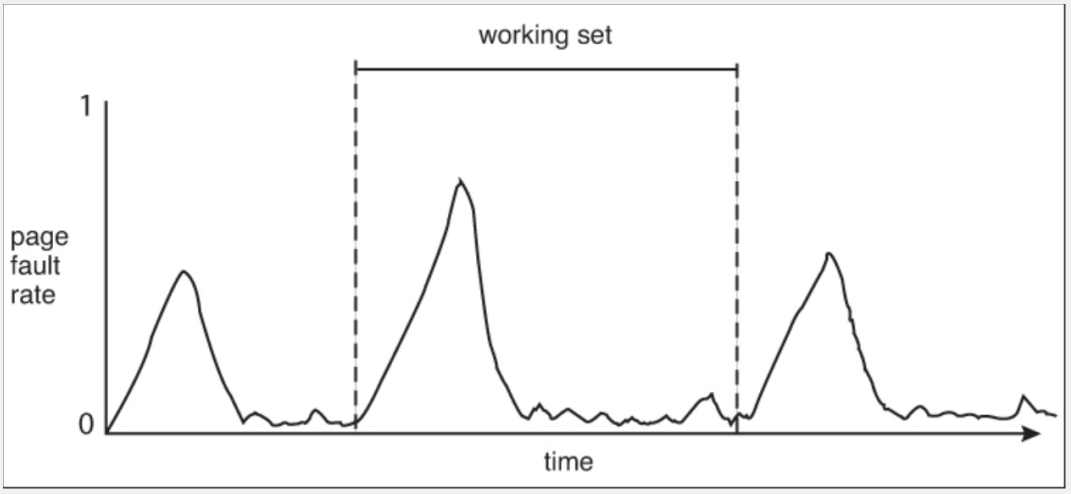


- 메모리 영역에 접근할 때 메모리에 page fault율이 높은 것
- 반복적으로 페이지 폴트가 발생해 과도하게 페이지 교체 작업이 일어나 실제로는 아무 일도 하지 못하는 상황
- 각 프로세스가 필요로 하는 최소한의 프레임 갯수를 보장해줘야 함

작업 공간 모델 (Working Set Model)

- 운영체제가 메모리를 관리할 때 지역성을 활용하는 모델
- 현재 시점으로부터 이전에 실행된 일정한 메모리 참조만을 working set으로 구별해 메모리에 할당
- 여기서 working set 구간을 **working set window**라고 함
- working set에 따라 메모리 프레임 할당 시 page fault가 균일하지 않고 특정 부분에서만 발생
    - 한동안은 지역성 때문에 page fault가 발생하지 않음





3. 세그멘테이션(Segmentation) 기법

- 페이징 시스템과 비교되는 기법
    - 실제로는 페이징 시스템을 더 많이 사용
- 가상 메모리를 서로 다른 논리적 단위인 세그먼트로 분할
    - 같은 크기의 block으로 분할하는 페이징 기법과 다름

3-1. 페이징 vs 세그멘테이션

- 페이징 시스템과 세그멘테이션 사이에는 Trade-off가 존재함

1) 내부 단편화 (Internal Fragmentation)

- 페이징에서 발생
- 페이지가 블록만큼 데이터가 딱 맞지 않는 경우 공간 낭비 발생
  - ex) 4KB 페이지 중 프로그램이 3KB만 사용하는 경우

2) 외부 단편화 (External Fragmentation)

- 세그멘테이션에서 발생
- 물리 메모리가 원하는 연속된 크기의 메모리를 제공하지 못하게 되는 경우
- 메모리에 남는 잉여 데이터때문에 공간이 충분히 있음에도 적재를 할 수 없음

세그멘테이션, 페이징 모두 하드웨어 지원이 필요

다양한 컴퓨터 시스템에 이식성을 중시하는 리눅스는 페이징 기법 기반으로 구현