



# 가상 메모리



## 목차

 가상 메모리란?

[가상 메모리](#)



작동 매커니즘

[MMU \(Memory Management Unit\)](#)



페이징 시스템

[구조](#)

[페이지 테이블](#)

[MMU와 TLB](#)

[페이징 시스템과 공유 메모리](#)

[Demand Paging](#)

[Page Fault](#)

[페이지 교체 정책](#)

[FIFO](#)

[최적 교체 알고리즘](#)

[LRU 알고리즘](#)

[LFU 알고리즘](#)

[NUR 알고리즘](#)

[스레싱 \(Threshing\)](#)



세그멘테이션 기법

[내부 단편화 문제](#)

[외부 단편화 문제](#)



[참고](#)



## 가상 메모리란?

- 폰 노이만 아키텍처 기반으로 된 코드들은 반드시 메모리(RAM)에 적재되어 실행되어야 한다
- 실제로 각 프로세스마다 메모리를 모두 할당하기에는 메모리 크기에 한계가 존재
  - 그러나 우리는 유튜브를 검색하면서 음악을 듣고 게임을 한다

- 이걸 가능하게 하는 것이 바로 **가상 메모리 (Virtual Memory) 시스템**

## 가상 메모리

- 적은 메모리로 여러 프로세스를 실행시키기 위한 시스템
  - 메모리가 실제 메모리보다 많아 보이게 하는 기술
  - 프로세스 크기 자체는 크지만, 실제로 프로세스가 사용하는 메모리는 작다는 점에서 착안
  - 또한, 프로세스간 공간 분리로 프로세스 이슈가 전체 시스템에 영향을 주지 않게 할 수 있음

## 작동 매커니즘

- 프로세스는 가상 주소를 사용해서 실제 해당 주소에서 데이터를 읽고 쓸 때만 물리 주소로 바꿔준다



### 가상 주소와 물리 주소

- 가상 주소 : 프로세스가 참조하는 주소
- 물리 주소 : 실제 메모리(RAM)에 있는 주소

## MMU (Memory Management Unit)



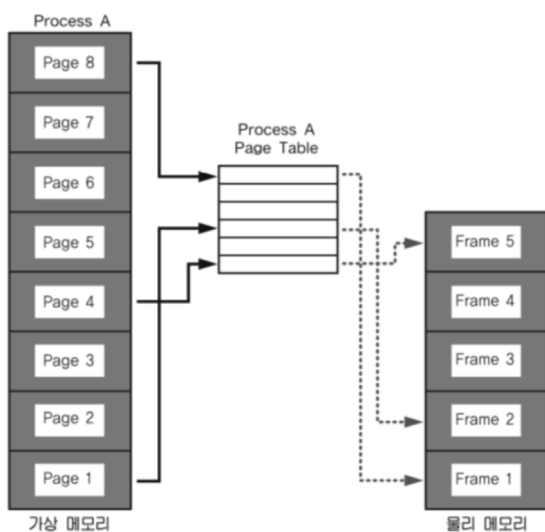
- 가상 메모리 시스템을 작동하기 위해 MMU에서 가상 메모리 주소와 물리 주소를 관리
  - CPU는 가상 메모리 주소를 다루고, 실제 물리 주소를 접근할 때는 MMU를 통해 가상 주소와 매핑된 물리 주소를 접근

- 소프트웨어 방식보단 하드웨어 장치가 주소 변환이 더 빠르기 때문에 MMU는 하드웨어 장치로 되어있다.



## 페이징 시스템

- 페이징 시스템은 크기가 동일한 페이지로 가상 주소 공간과 매칭하는 물리 주소 공간을 관리
  - 프로세스가 실행될 때 그때마다 필요한 공간만 적재시켜서 실행하면 됨
  - 이 필요한 공간을 페이지 단위로 나뉘어서 메모리를 적재, 프로세스를 실행
  - 이 페이지를 관리하는 시스템을 **페이징 시스템**이라고 한다



- 프로세스의 PCB에 페이지 테이블 구조체를 가리키는 주소가 들어있음
- page table에 가상 주소와 물리 주소 간 매핑 정보가 존재

## 구조

- **paging system**은 **page** 또는 **page frame**이라고 불리는 고정된 크기의 block으로 프로세스를 나눈다
  - block이 고정된 크기의 4kb라고 할 때, 만약 9kb의 데이터가 있다고 하면 page1에 4kb, page2에 4kb, page3에 1kb에 남는 3kb까지 함께 넣는다

가상 주소(Virtual Address)  $v = (p, d)$

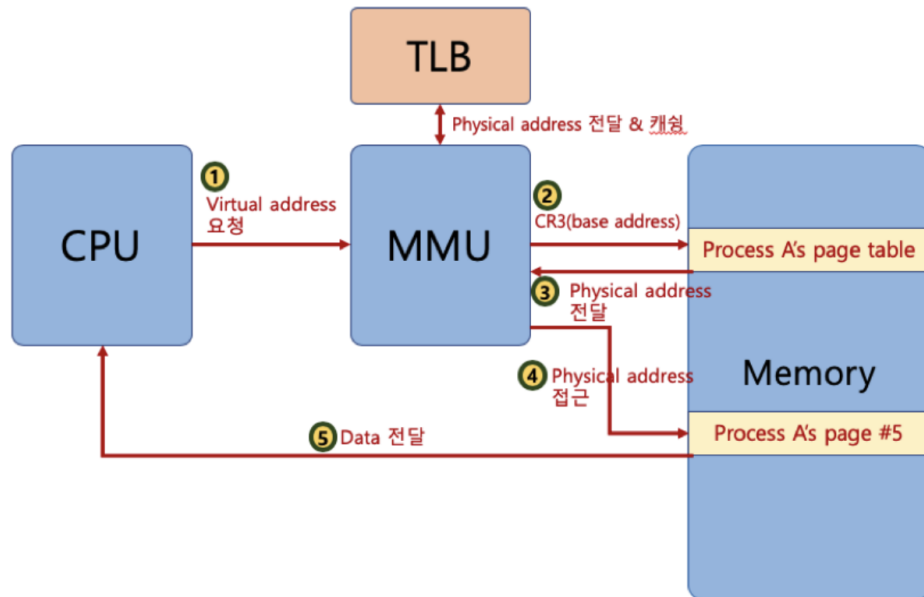
페이지 번호 $p$	변위(오프셋) $d$
------------	-------------

- **paging system**
  - 가상 주소는 페이지 번호(p)와 페이지 내에서의 오프셋(d)로 나타낼 수 있음
  - **page** 크기가 4kb일 때를 예시로 들면 32bit에서 12~31bit는 페이지 번호가 될 수 있음
  - 0~11bit는 오프셋(변위)를 나타냄
  - 이 page 번호를 그에 해당하는 물리 주소로 매핑
  - 이렇게 매핑된 물리 페이지 번호에 오프셋을 더하여 실제 물리 주소를 알아낼 수 있다

## 페이지 테이블

- 물리 주소에 있는 페이지 번호와 해당 페이지의 base 물리 주소 정보를 매핑한 표
- 이것을 이용해서 물리 주소에 접근하기 위해서는
  1. 해당 프로세스의 페이지 테이블에 해당 가상 주소가 포함된 페이지 번호가 있는지 확인
  2. 페이지 번호가 있다면 이 페이지에 매핑된 base 물리 주소를 알아낼 수 있다
  3. base 물리 주소에 변위를 더한 값이 실제 물리 주소가 됨
- 프로세스의 페이지는 항상 물리 메모리에 적재될 필요가 없다
  - 따라서, 적재되지 않은 페이지는 페이지 테이블에 valid/invalid bit를 이용해 적재 여부 정보를 판단

## MMU와 TLB



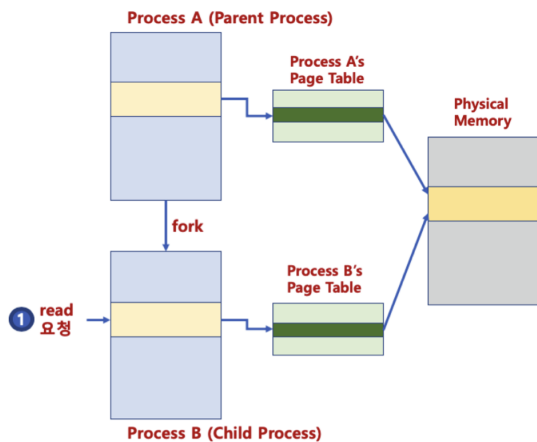
- TLB는 Translation Lookaside Buffer
- MMU를 활용해서 실제 물리 주소를 접근할 때 추가적인 하드웨어 보조장치를 활용해서 접근
  - 과거에 해당 정보를 이용한 적이 있다면 이를 따로 레지스터에 캐싱한 후 접근 속도를 높이는 보조장치가 바로 **TLB**이다

## 메커니즘 with TLB

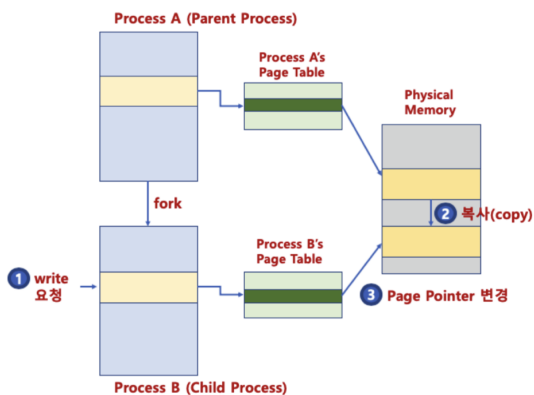
1. MMU에 가상 주소 요청
2. MMU는 TLB에서 최근에 가상 주소에서 물리 주소로 변환한 정보가 있는지 확인, 있다면 해당 정보 활용
3. TLB에 없다면 base 주소로 접근해 메인 메모리에서 데이터를 찾음
4. 찾았다면 메인 메모리는 MMU에 물리 주소를 반환
5. MMU는 이 정보를 TLB에 캐싱하고 이 물리 주소를 기반으로 메인 메모리에 접근
6. 메인 메모리는 해당 데이터를 CPU에 적재시켜서 프로세스를 수행

## 페이징 시스템과 공유 메모리

- 프로세스는 동일한 물리 주소를 가리킬 수 있다
- 실행 중인 프로세스들이 동일한 로직을 수행한다면 굳이 따로 메모리에 적재될 필요가 없다
  - 결과적으로 메모리 절약이 됨



- `fork()` 함수는 실행 중인 프로세스에게 동일한 자식 프로세스를 만들어 준다
- 동일하니까 로직도 같으니 `read` 만 수행한다면 따로 메모리 공간을 할당시킬 필요가 없음
- 공간 절약과 동시에 메모리 할당 시간도 절약

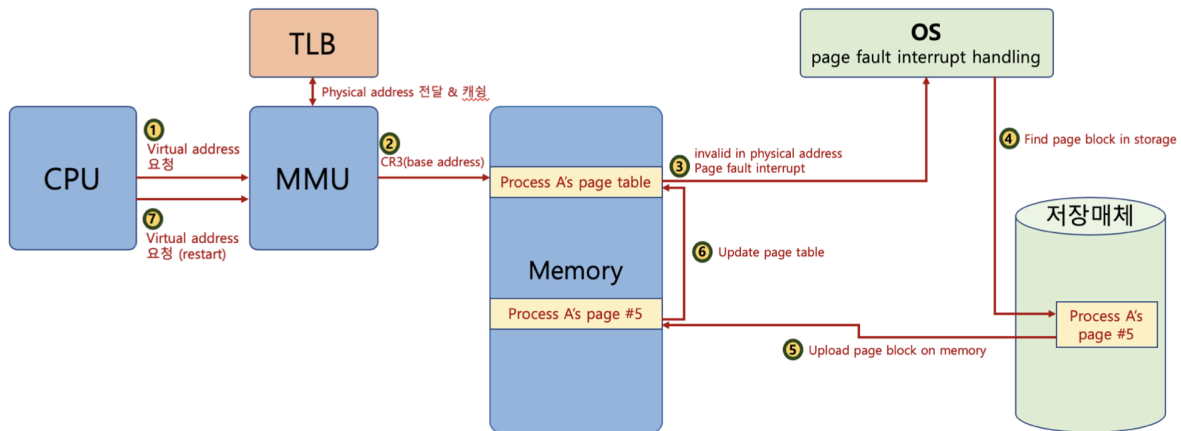


- 단, 특정 프로세스에서 `write` 를 요청해 물리 주소에서 데이터가 변경되면 메모리 공간이 같으면 안된다
- 이렇게 공간 할당이 필요할 때만 물리 주소 복사를 수행
  - 이를 `copy-on-write` 라고 한다

## Demand Paging

- 가상 메모리는 실행 중에 필요한 시점에만 프로세스를 적재
- 그렇다면 더 이상 필요하지 않은 페이지 프레임은 다시 저장 매체에 저장을 해야 한다
  - 이를 위한 **페이지 교체 알고리즘**이 필요

## Page Fault



- 어떤 페이지가 실제 물리 메모리에 없을 때 일어나는 interrupt
- 운영체제가 **page fault** 를 발생시키면 해당 페이지를 물리 메모리에 올리는 방식
  - 그러나 메인 메모리에 접근하는 시간은 큰 비용이 든다
  - 따라서, 적절한 페이지를 미리 물리 메모리에 올릴 수 있어야 한다

## 페이지 교체 정책

- 운영 체제가 특정 페이지를 물리 메모리에 올리려 할 때, 물리 메모리가 다 차게 되면
  - 기존 페이지 중 하나를 물리 메모리 → 저장 매체로 내린다
  - 새로운 페이지를 해당 공간에 올린다
- 이 때 어떤 메모리를 내릴 것인가 정하는 것이 바로 **페이지 교체 알고리즘**이다

## FIFO

- 가장 처음에 들어온 페이지를 내리고 새 페이지를 해당 공간에 올린다

## 최적 교체 알고리즘

- 앞으로 가장 오랫동안 사용하지 않을 페이지를 내린다
- 그렇지만 이를 예측하는 것은 쉽지 않기 때문에 완벽하게 구현할 수 없는 알고리즘

## LRU 알고리즘

- 가장 최근에 사용한 페이지를 교체
- 한번 사용했으니 또 사용할 일은 없을 것이라는 전제 하에 수행하는 알고리즘

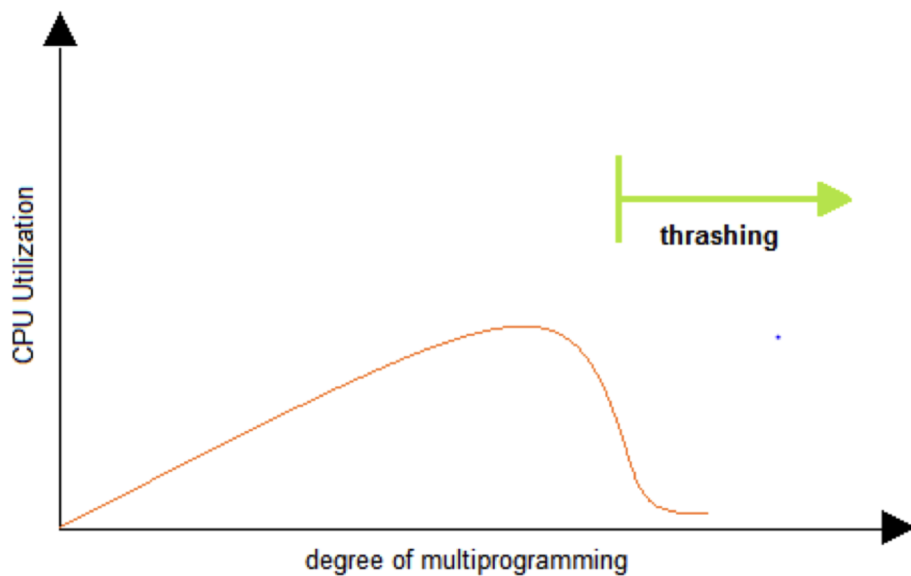
## LFU 알고리즘

- 가장 적게 사용한 페이지를 교체
- 그러나, 처음에 초기화를 위한 중요 데이터가 있는 페이지를 교체할 가능성이 존재한다

## NUR 알고리즘

- 각 페이지마다 참조 비트 **R** 과 수정 비트 **M** 을 쌍으로 묶어서 정보를 유지
  - 참조 여부를 기준으로 페이지를 우선적으로 교체
  - 참조 여부가 같은 페이지는 수정 여부를 기준으로 교체

## 스레싱 (Thrashing)



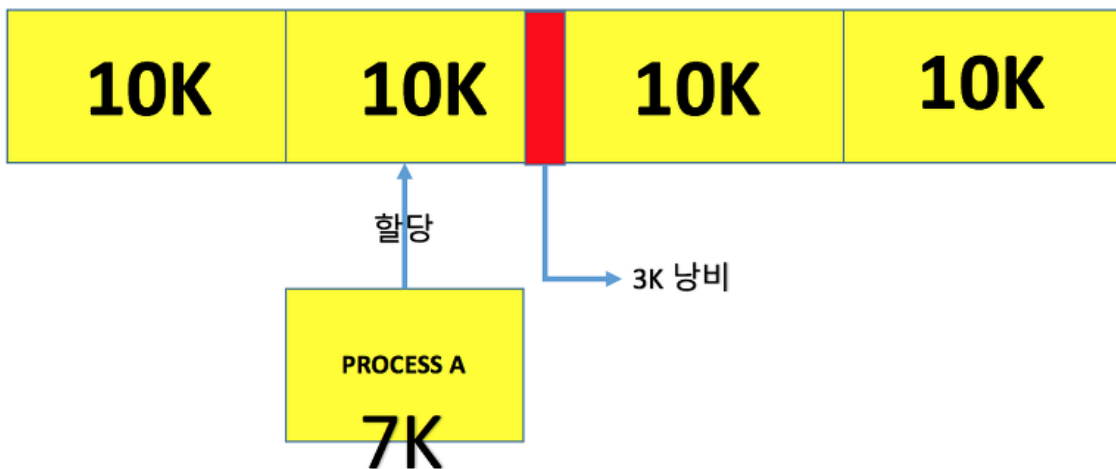
- 반복적으로 페이지 폴트가 발생할 때 생기는 문제
- 멀티프로그래밍의 정도는 높아지지만 오히려 CPU 이용률은 줄어드는 현상을 의미

## 🍌 세그멘테이션 기법

- 페이징 시스템과 비교되는 기법 중 하나
- 가상 메모리를 서로 크기가 다른 논리적인 내용을 기준으로 **Segment** 로 분할
  - 같은 크기의 **block** 으로 분할하는 페이징 기법과 다소 상이

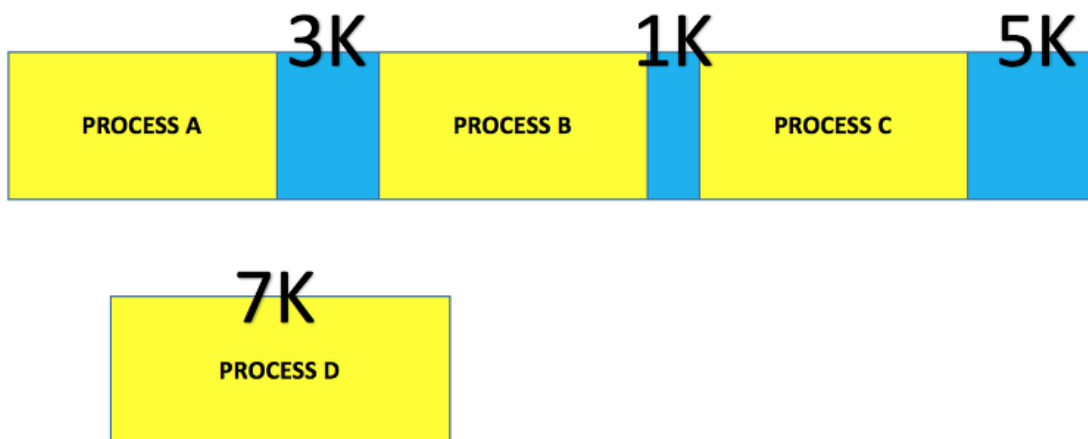
## 내부 단편화 문제





- 페이징 기법의 경우 페이지를 같은 크기의 block으로 나눈다
  - 따라서 process의 크기와 딱 나누어 떨어지지 않을 경우 공간 낭비가 발생
  - 이를 내부 단편화 문제라고 한다

## 외부 단편화 문제



- 물리 메모리가 원하는 연속된 크기의 메모리를 제공하지 못하게 되는 경우
  - 할당된 메모리 사이 사이에 사용하지 않는 작은 메모리가 생김
  - 작은 메모리들의 합인 총 메모리 공간은 충분하지만, 실제 연속적으로 할당할 수 있는 공간은 부족해 메모리를 할당할 수 없는 상황


- 세그멘테이션 기법의 경우 찌꺼기 데이터 때문에 공간이 충분히 있음에도 적재를 할 수 없는 상황이 되는 문제 발생

## 참고

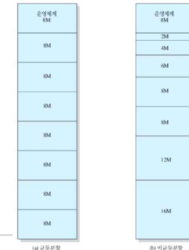
### ▼ 링크

#### 페이징(Paging)과 세그먼테이션(Segmentation)

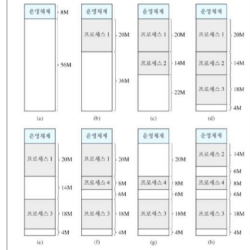
이 기법을 쓰는 이유는 다중 프로그래밍 시스템에서 다수의 프로세스를 수용하기 위해 주기억장치를 동적으로 분할하는 메모리 관리 작업이 필요함 즉, 단일 프로그램만 쓰는 것이 아니기 때문에 이

 [https://velog.io/@gang\\_shik/페이징Paging과-세그먼테이션Segmentation](https://velog.io/@gang_shik/페이징Paging과-세그먼테이션Segmentation)

3) 고정 분할 예시 (64MB 메모리)



4) 동적 분할 예시 (64MB 메모리)



#### [운영체제] 가상 메모리

가상 메모리 내용 짹짹 채워서 정리해 봤어요(편을 나누는게 나았나...).

 <https://velog.io/@redgem92/운영체제-가상-메모리>

가상 메모리

#### [운영체제(OS)] 13. 페이징


1. 페이징(Paging) 외부 단편화로 인한 메모리 낭비는 매우 심하다는 것을 살펴보았다. Compaction을 사용하면 외부 단편화는 해결할 수 있지만, 그로 인해 발생하는 오버헤드와 비효율적인 성

 <https://velog.io/@codemcd/운영체제OS-13.-페이징>

velog

#### 가상메모리와 페이징 시스템이란 무엇인가?

운영체제의 대표적인 일 중 하나가 컴퓨터 내의 한정된 메모리를 극한으로 활용하는 메모리 관리이다. 가상 메모리는 메모리 관리 기법의 하나로 컴퓨터가 실제로 이용 가능한 자원을 추상화하여

 <https://velog.io/@hoon-devlog/가상메모리와-페이징-시스템이란-무엇인가>

velog