메모리 & 캐시

메모리(RAM)

RAM(Random Access Memory)는 애플리케이션에 단기적으로 데이터 저장 및 액세스 공간을 제공하는 컴퓨터의 임시 스토리지

프로그램 실행 시 필요한 주소, 정보들을 저장하고 가져다 사용할 수 있게 만드는 공간

RAM의 용량이 크면 실시간으로 더 많은 데이터를 액세스하고 읽을 수 있다.

**** 메모리(RAM)를 하나의 큰 배열로 생각할 수 있다. 주소를 통해 해당 위치의 데이터에 접근하는 방식이 배열에서 인덱스를 사용하여 각 요소에 접근하는 것과 유사**

물리주소 vs 논리주소

- 물리주소: 실제 RAM에 매겨진 주소로 하드웨어에 의해 고정된 메모리 주소이다.
- 논리주소: 각 프로세스 마다 독립적으로 가지고 있는 주소 값. 실제 메모리의 주소가 아닌 프로세스 마다 가지는 가상 주소. 프로그램 마다 중복되는 논리 주소를 가질 수 있다.
- CPU는 프로세스의 논리주소만 읽고 메모리는 물리주소를 읽음 → (**CPU는 현재 실행중인 프로세스의 내부 주소만 알면 되기 때문. 어떤 프로세스인지 알 필요가 없음**)

Q: 직접 물리주소로 접근하면 되지 왜 논리 주소가 필요할까?

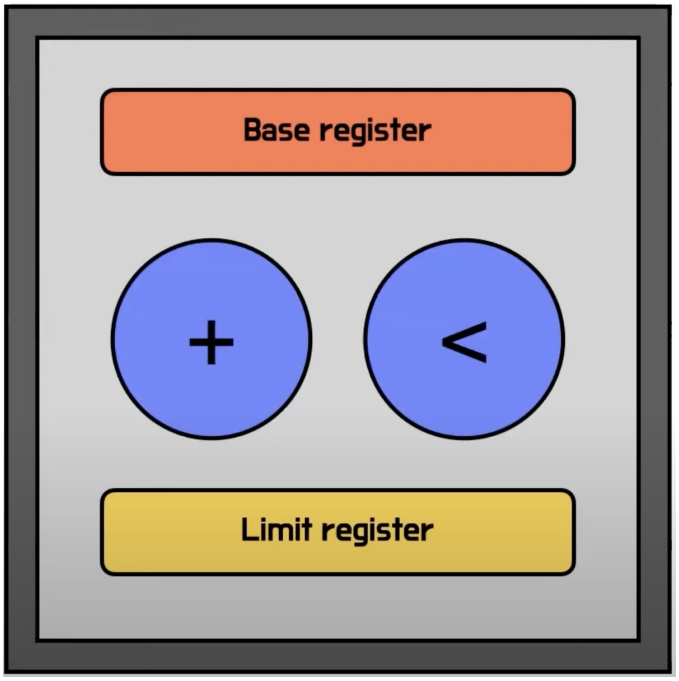
1. **안정성 측면 :** 프로그램들이 동일한 물리적 메모리 공간(동일한 물리주소)에 접근할 경우, 데이터 손상이나 충돌이 발생할 수 있다.프로그램들이 동일한 논리 주소에 접근하더라도, 물리주소로 변환하는 과정을 통해 동일한 물리주소로 접근하는 상황을 막을 수 있다.

ex) 각 프로그램이 물리주소 10번지에 접근 → 충돌

각 프로그램이 논리주소 10번지에 접근 → 각각 물리주소 10번지, 20번지로 변환되어 충돌 방지

2. **보안적 측면 : 논리 주소 → 물리 주소 변환하는 과정을 통해 프로세스가 요청하는 논리주소가 올바른지 확인 가능**
3. **효율적인 메모리 사용, 유연성**

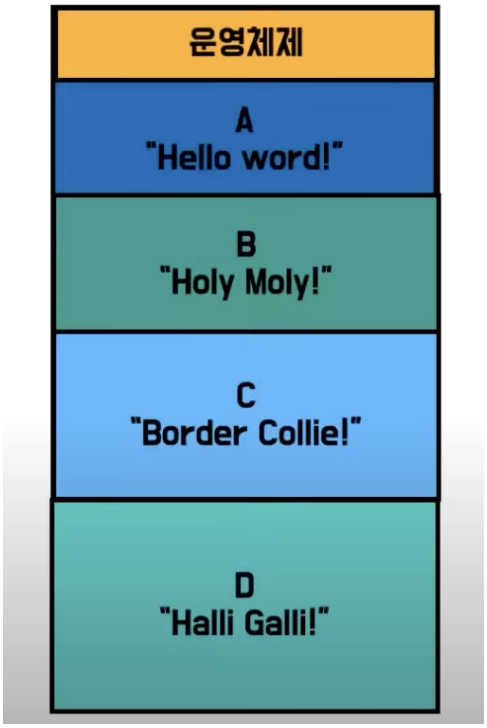
MMU(Memory Management Unit)



- 소프트웨어가 아닌 하드웨어
- MMU는 프로세스의 논리주소에 Base register에 들어있는 시작 주소를 더해 실제 물리주소로 변환.
- 이때, Limit register에 들어있는 마지막 주소로 프로세스가 요청하는 논리주소가 올바른지 검증을 거친다. (자신의 논리적 주소가 아닌 다른 주소를 통해 다른 프로그램의 정보를 요청하는 경우) → 올바른 요청이 아니면 CPU권한을 OS에 넘기고, OS는 이 프로세스를 확인

즉 MMU를 통해 우선 검증을 하고, 검증된 프로세스의 논리주소를 실제 물리주소로 변환하는 과정을 거쳐 CPU가 물리주소에 있는 정보를 읽어올 수 있다.

메모리 할당, Swapping



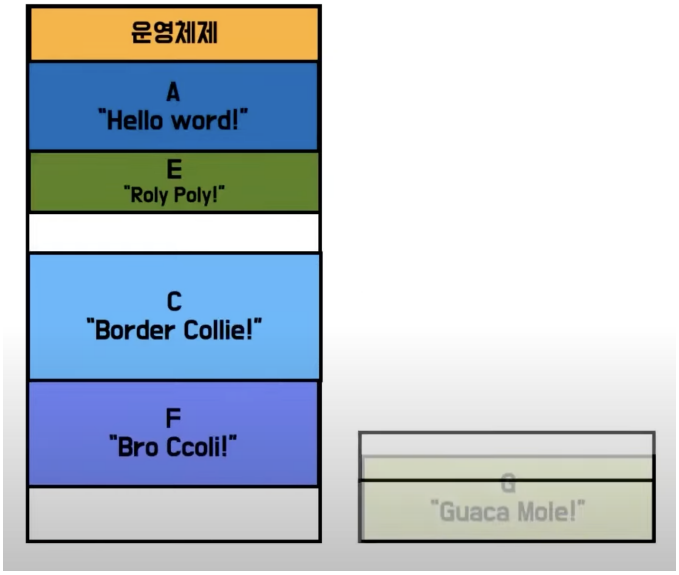
**운영체제도 메모리에 올라와 할당되는 프로세스중 하나!

위 그림과 같이 메모리에는 프로세스들이 차례로 할당 됨

메모리에 프로세스를 실행시킬 공간이 부족할 때 다른 프로세스를 **하드디스크**에 있는 SWAP 공간으로 보내는 과정이 필요 → **Swapping 기법**

메모리에는 프로세스들이 차곡차곡 이쁘게 쌓이지 않는다!

아래 그림과 같이 남는 공간을 합치면 하나의 프로세스를 실행시킬 수 있지만 이를 이용할 수 없는 상황 발생



- 공통 문제점
  - 내보낼 프로세스를 선정하기 위한 기준 선정의 어려움. 이를 위해 전체를 탐색해야 함.
  - 프로세스를 내보낸 후 메모리의 공간을 활용하기가 어려움
  - Swapping 자체의 오버헤드! (디스크 I/O)

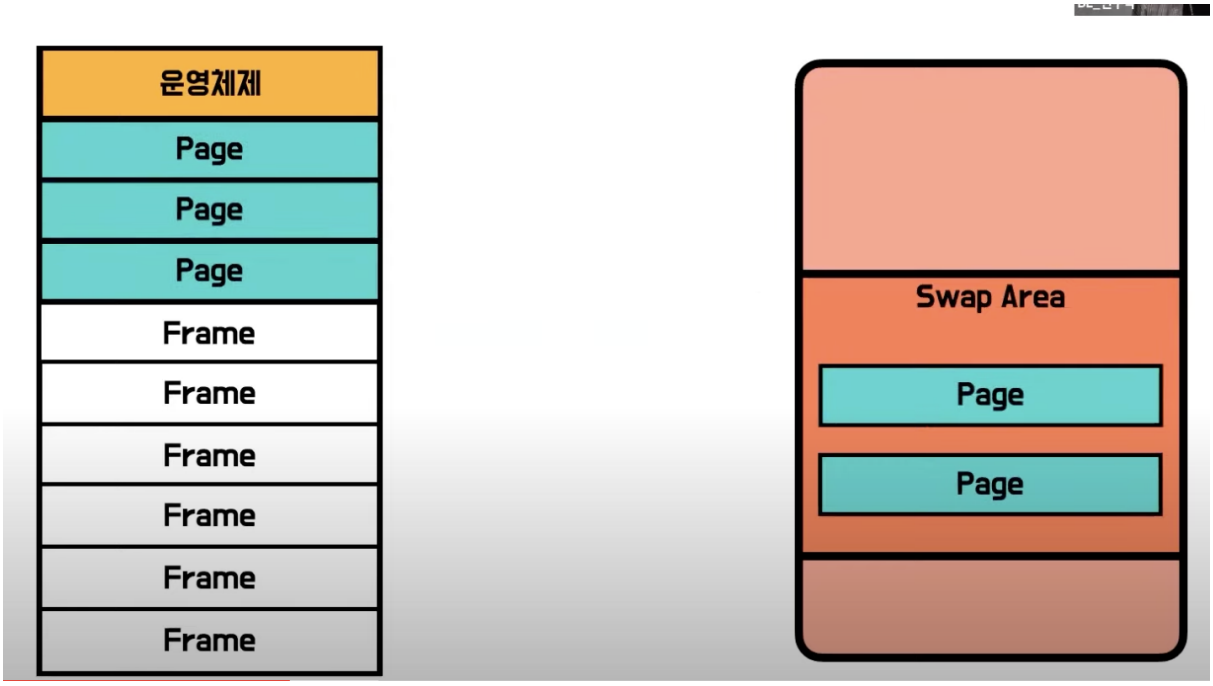
공간 활용 기법 3가지

- First-fit : 메모리의 시작부터 탐색을 시작하여, 프로세스를 수용할 수 있는 첫 번째 충분히 큰 공간에 프로세스를 할당 → 빠르지만 여전히 공간의 낭비 발생 가능성 높음.
- Best-fit : 프로세스를 수용할 수 있는 가장 작은 공간에 프로세스를 할당 → 탐색이 오래걸림, 매우 작은 남은 공간이 여러 개 발생하여 여전히 낭비 발생 가능성 높음.
- Worst-fit : 프로세스를 수용할 수 있는 가장 큰 공간에 프로세스를 할당 → 빠르게 공간 소진, 탐색 오래걸림, 공간 낭비

메모리를 사용하기 위해 Swapping 기법은 반드시 필요하고 이를 위한 여러 방법이 등장했지만 근본적인 문제들을 해결하지는 못함

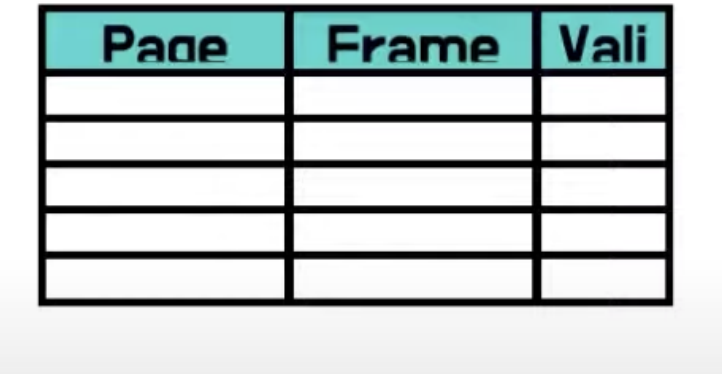
메모리 공간을 일정하게 잘라두고, 그에 맞춰서 프로그램을 조금씩 잘라서 올리자! → Paging 기법 등장

Paging

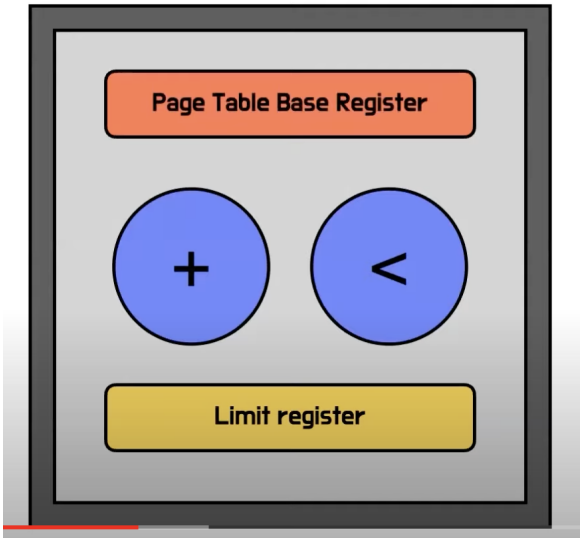


메모리 공간을 일정 크기로 자르고(Frame) 프로세스도 이 프레임 크기로 잘라 각각 메모리에 할당하는 방식

당장 프로세스 실행에 필요한 page만 메모리에 올리고, 나머지는 swap 공간에 저장

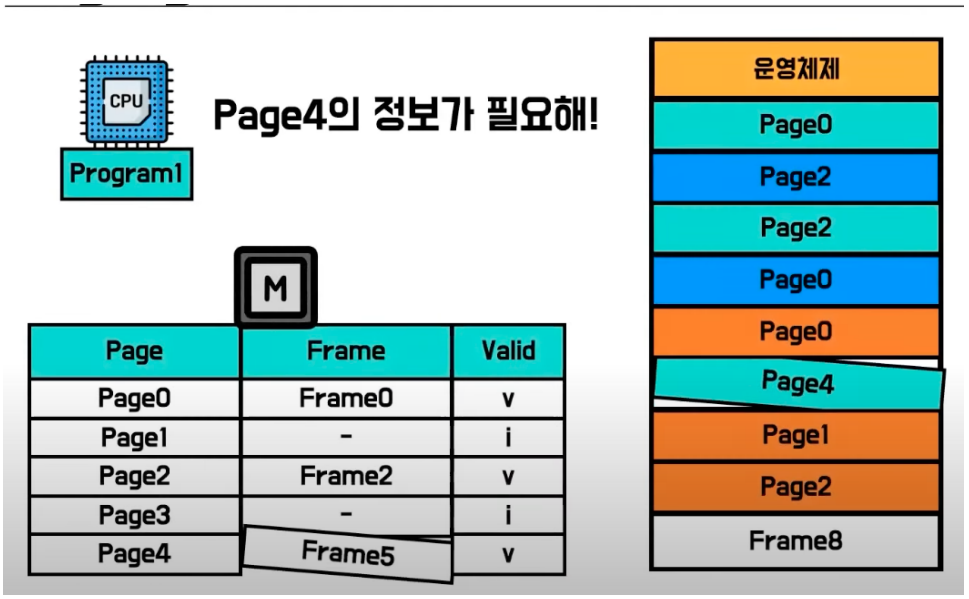


페이징 기법을 사용하면 위와 같이 한 프로그램의 페이지가 메모리의 여러 공간에 분리되어 순서도 보장이 안되고 MMU 계산도 복잡해짐. → 논리주소와 물리주소 간의 매핑 정보를 저장하는 Paging Table 등장



페이징 테이블이 등장하면서 MMU도 이에 맞게 역할이 바뀜.

Base register는 페이징 테이블의 시작주소를 더해주는 역할을 하고, Limit register는 페이지 테이블의 크기를 검증하는 역할을 한다.

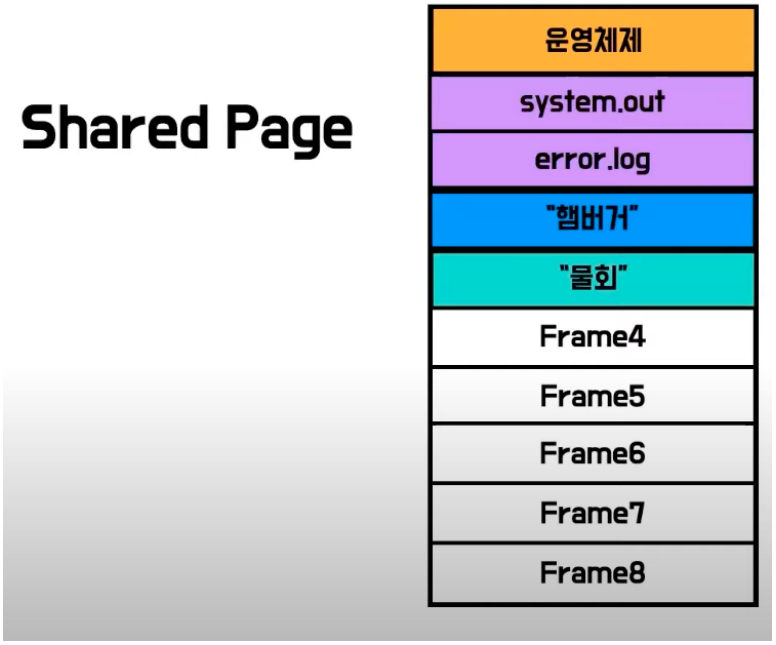


**Valid는 메모리에 있는지 Swap 공간에 있는지 빠르게 조회하기 위함

페이징 테이블도 마찬가지로 메모리에 저장 → 페이징 테이블의 크기도 크기 때문에 메모리 공간 부족 문제

→ 프로세스들이 공통으로 사용하는 Shared Page를 만들어 공간을 확보

반드시 Read-only, 프로세스 마다 동일한 논리주소를 가져서 탐색 시간을 없애야함. → 페이징 테이블에 Read-only 표시를 위한 칼럼이 하나 더 추가 되었지만, 훨씬 더 많은 공간을 아낌.



여기까지의 문제점..!

CPU가 정보를 요청할 때 마다 PageTable에 접근하기 위해 메모리 참조, pageTable에서 얻은 정보를 통해 Page에 접근하기 위해 메모리를 참조하여 총 두 번의 접근이 필요함 → 명령 수행 시간 문제 발생

→ 캐시(TLB)를 통해 접근 횟수를 줄이는 방법으로 해결

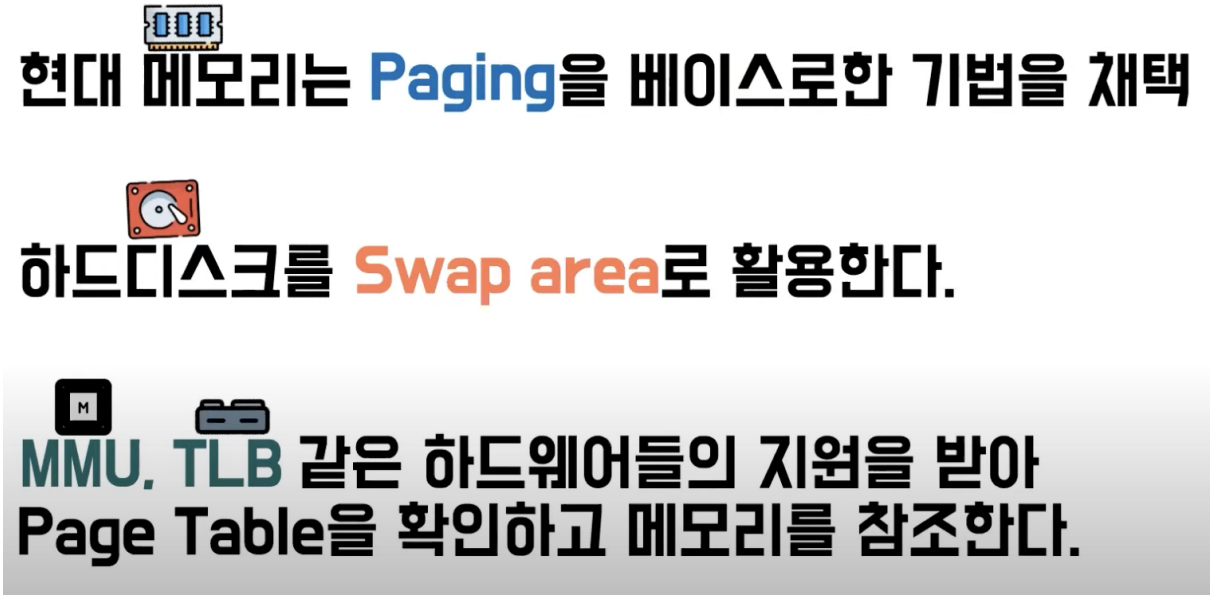
캐시(TLB-Translation Look-aside Buffers)

Page Table에 접근하기 이전에 우선적으로 TLB를 확인하여 매칭되는 주소가 있으면 Page Table에 접근하지 않고 바로 메모리에서 해당 page를 가져온다! → Page Table을 참조하느라 메모리에 2번 접근하던 것을 1번으로 줄이기 위한 방법!

대부분의 프로세스들은 한번 참조했던 곳을 다시 참조하는 경우가 많기 때문에 성공률이 높아져서 메모리 접근에 대한 문제가 많이 개선이 됨

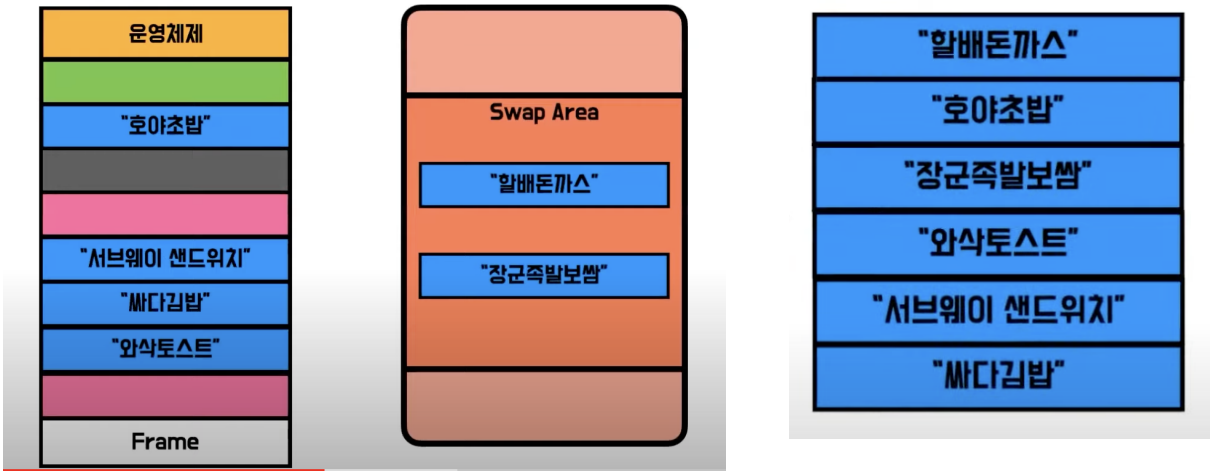
동작방식

- 프로세스가 메모리에 접근할 때, MMU는 해당 주소를 TLB에 검색
- TLB에 매핑 정보가 존재하면, 물리 주소로의 변환이 바로 수행됩니다.
- 만약 TLB에 해당 정보가 없다면, 페이지 테이블을 검색하여 물리 주소를 찾습니다. 그 결과를 TLB에 저장하고, 이후에 사용할 때 사용할 수 있게 된다.



그렇다면 메모리 관리를 위한 OS의 역할은 뭐지...?

운영체제(리눅스)의 메모리 관리 기법



- **가상 메모리로 사용자 프로세스 속이기**

실행중인 프로세스는 자신이 온전한 형태로 메모리에 올라가있다고 생각하기 때문에 에러를 내지 않고 정상적으로 작동한다. 그러나 실제 프로세스는 필요한 부분만 메모리에 있고 나머지 부분은 swap공간에 위치

메모리 공간과 swap공간에 있는 프로세스 조각을 전부 합쳐 만들어낸 가짜 메모리를 **가상 메모리**라 한다.

이를 통해 실제 메모리의 크기보다 더 많은 양의 메모리를 사용하는 것 처럼 구현을 할 수 있다!

CPU를 통해 요구하던 논리주소는 이 가상 메모리 주소.

- **Page fault**

프로세스가 CPU를 점유하고 작업을 하던 중 TLB와 메모리에 올라와있지 않은 page를 요구할 경우 (이를 **Page fault**라고 한다) swap 공간에서 page를 가져오고 TLB에 등록해야 한다. 이 작업을 운영체제가 진행하며 디스크 I/O 작업이기 때문에 시간이 상당히 오래걸린다. Page fault확률이 곧 성능!!! → 다만 컴퓨터 프로그램 특성 상 같은 곳을 계속 참조하기 때문에, TLB를 참조하여 Page fault가 발생할 확률이 낮다.

- Replacement

swap공간에서 page를 가져오다가 frame이 가득 찼을 경우 메모리의 page 하나를 쫓아내야한다. 이를 replacement라 하고 어떤 페이지를 replacement하는지는 운영체제가 결정한다. 이 때 가장 오랫동안 참조되지 않은 페이지를 쫓아내면 좋겠지만(LRU) 운영체제에는 이에 대한 정보가 없음...

→ Clock Algorithm을 사용하는데.... 원리는 생략하고 결과만 말하면, 가장 최근에 참조된 페이지는 피하는 방법이다!

페이지를 내쫓을 때도 변경사항이 있으면 하드디스크에도 변경을 기록해야 하기 때문에 Dirty 비트가 생기고, 이와 관련된 작업을 전부 OS가 실행

Page	Frame	Valid	Auth	Ref	Dirty
호야초밥	1	v	R/W	0	0
와삭토스트	6	v	R/W	0	0
할배돈까스	8	v	R/W	1	0
장군족발보쌈		i			
서브웨이 샌드위치	4	v	R/W	1	0
싸다김밥		i		0	

- **Trashing**

메모리에 프로세스 종류가 많아질 수록, 각 프로세스가 사용할 수 있는 frame의 개수는 줄어들고 page가 적게 올라온 프로세스들은 해당 페이지만큼만 명령을 수행할 수 있고, 다음

작업을 수행하려다 page fault가 자주 발생하여 replacement를 요청하게 된다!

프로세스가 replacement를 진행하는 동안 다른 프로세스로 CPU 점유가 넘어가지만 그 프로세스 또한 replacement를 하게되는 상황 발생.

replacement를 하는 동안 CPU는 할 일이 없음. 운영체제는 이를 보고 프로세스의 개수를 늘림 → 악순환 반복

이 현상을 Trashing이라 한다.

운영체제는 Workin-set, page Frequeny 알고리즘 사용하여 해소

- working-set

대부분 프로세스가 일정 page만 집중적으로 이용한다는 성격 이용!

만약 frame 3개를 자주 이용할 때 최소 3개의 frame이 확보되어야 메모리에 이 프로세스를 올리고 replacement도 마찬가지로 3개의 단위로 page를 쫓아냄

- page Frequeny

pagefault 비율의 상한선 하한선을 두는 기법

상한선을 넘으면 프로세스마다 지급하는 Frame을 늘리고 (프로세스 개수를 줄임)

하한선보다 작으면 프로세스 마다 지급하는 Frame 개수를 줄임 (프로세스 개수를 늘림)

메모리 고갈 상황과 CPU 사용률을 체크하는 이유

"메모리가 고갈되면 어떤 현상이 발생할까요?"

프로세스들의 Swap이 활발해지면서 CPU 사용률 하락
운영체제가 프로세스 추가. 쓰레싱 발생
쓰레싱이 해소되지 않을 경우 Out of Memory 상태로 판단
중요도가 낮은 프로세스를 찾아 강제 종료

"CPU 사용률을 계속해서 체크하는 이유"

특정 시점만 체크한 경우 CPU 사용률이 높아보일 수 있음
연속 체크시 CPU 사용률이 급격하게 떨어지는 구간 발견 가능성
메모리 적재량을 함께 체크하면서 쓰레싱 유무 확인
추가적인 서버자원을 배치하는 등 해결방안을 마련