Virtual Memory

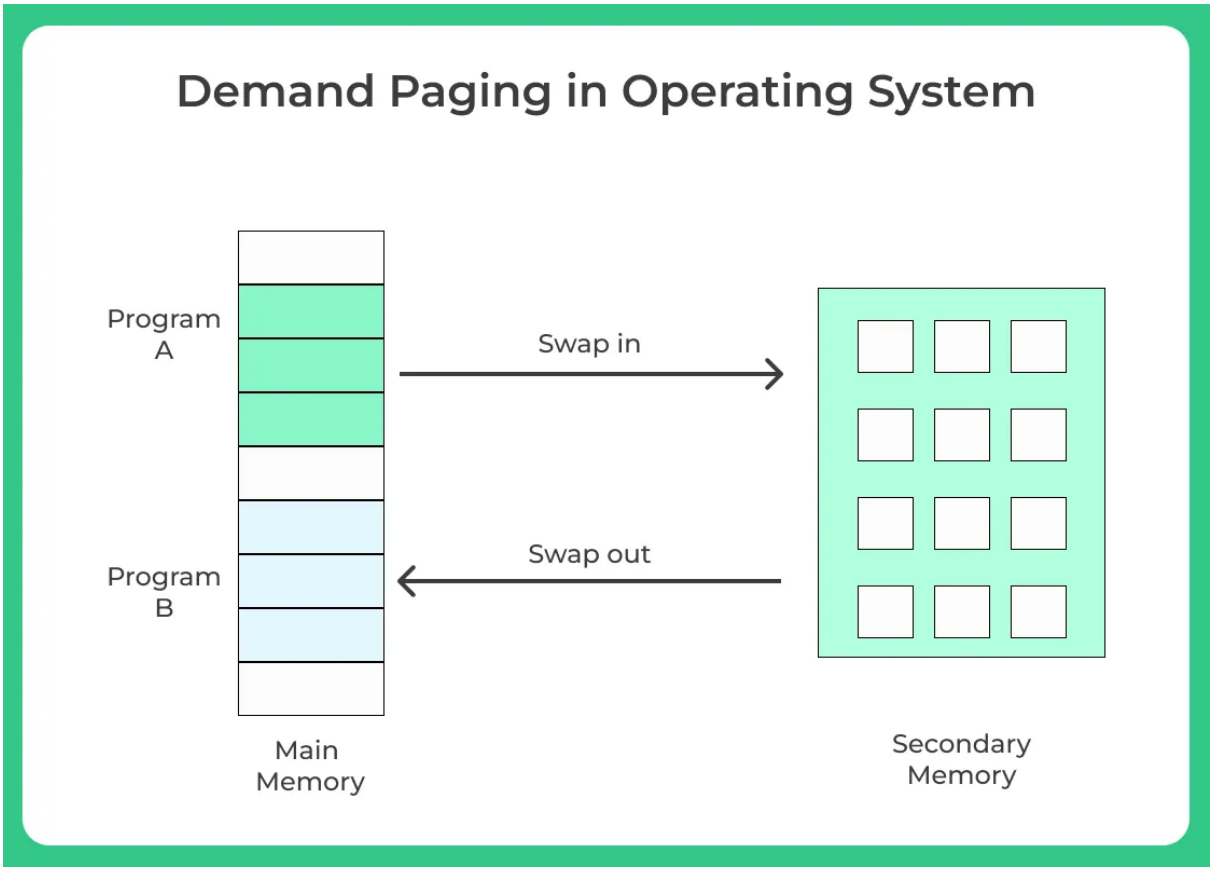
가상 메모리

- 장점
  - 메인 메모리를 하드 디스크의 캐시로 처리하여 더 넓은 메모리 공간을 제공한다. 즉, 임시 데이터를 하드 디스크 드라이브에 저장하고 필요에 따라 기본 메모리로 가져올 수도 있다
  - 메모리 크기의 제약으로부터 자유로워졌고 사용자 프로그램이 실행에 필요한 최소 크기의 메모리만 올라가져 있기 때문에 더 많은 프로그램을 동시 수행 가능해졌다.
  - 추가 권한 비트를 사용하여 각 프로세스에 메모리에 대한 제한된 권한을 부여하여 메모리를 보호한다. OS는 권한 비트를 확인하여 하나의 프로세스가 다른 프로세스의 메모리 공간을 손상시키지 않도록 할 수 있다.
- 단점
  - 프로그램이 현재 메모리에 없는 페이지에 엑세스하면 디스크에서 페이지를 가져오는 동안 지연이 발생하게 된다.
  - 디스크 입출력이 많아지면 시스템의 전반적인 반응 속도가 느려지는 현상이 발생할 수 있다.

Demand Paging

OS에 실행 중인 모든 프로그램을 저장하는 데에 300MB의 메모리가 필요하다고 가정하고, 현재 RAM에 저장된 사용 가능한 물리적 메모리는 50MB라고 가정하자.

- OS에 250MB의 가상 메모리를 설정하고 VMM(Virtual Memory Manager)를 사용하여 250MB를 관리한다.
- 이 경우 VMM은 필요한 추가 메모리를 저장하기 위해 250MB 크기의 파일을 디스크에 생성한ㄴ다.
- 이제 OS는 RAM에 저장된 300MB의 실제 메모리를 고려하므로 50MB의 공간만 사용할 수 있더라도 메모리 주소 지정을 계속한다.
- 50MB의 실제 메모리 공간만 사용할 수 있더라도 300MB의 메모리를 관리하는 것이 VMM의 역할이다.



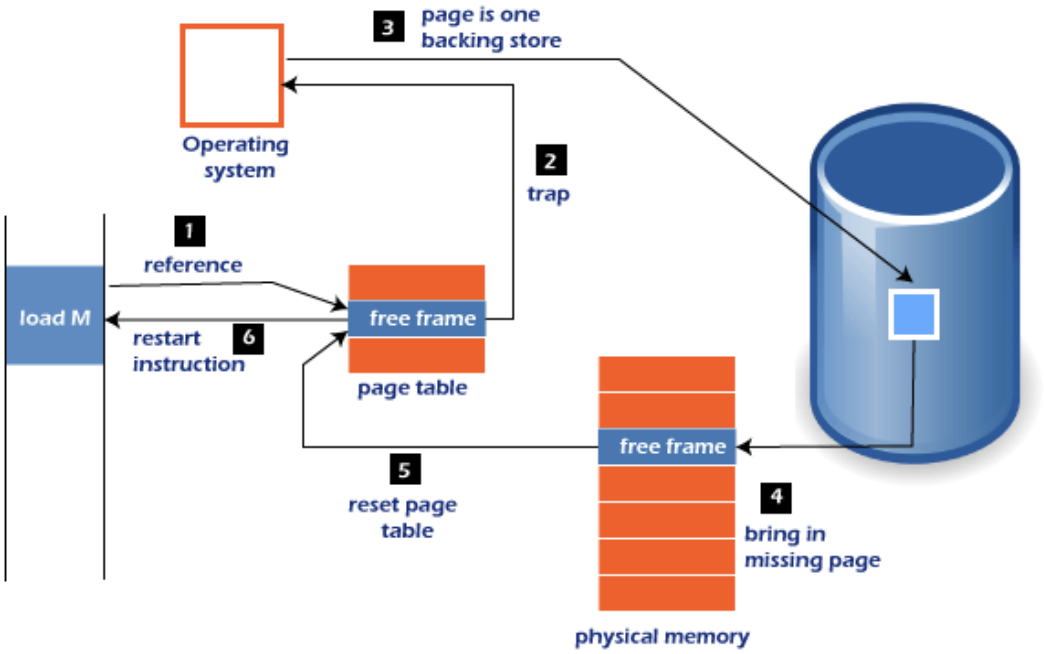
Demand Paging은 페이지 테이블에서 해당 페이지가 메모리에 있는지를 나타내는 유/무효 비트를 사용한다.

비트가 무효인 경우 페이지가 물리적 메모리에 없다는 것으로 페이지 부재가 발생한다.

페이지 부재 처리 과정

- CPU가 특정 페이지를 접근하여 페이지 테이블에서 유/무효 상태를 확인한다.
- 페이지가 무효 상태일 때 MMU에서 페이지 부재 트랩이 발생한다. (page miss)
- 디스크에서 해당 페이지를 빈 프레임에 적재하고 페이지 테이블을 업데이트 합니다. (무효 → 유효)
- 트랩에 의해 중단되었던 명령을 다시 수행한다.

MMU(Memory Management Unit) : 가상 주소를 물리 메모리 주소로 변환해주는 하드웨어 장치



여기서 어떤 페이지를 탈락시키고 어떤 페이지를 프레임에 올릴지를 결정하는 알고리즘에 따라 성능이 결정된다. 때문에 적절한 알고리즘을 사용하는 것이 중요하다.

페이지 교체 알고리즘

페이지 부재가 발생하면 요청된 페이지를 디스크에서 메모리로 가져온다. 이 때, 물리적 메모리에 공간이 부족할 수 있다. 그럴 경우 메모리에 올라와 있는 디스크를 옮겨서 메모리 공간을 확보해야 하는데 이를 페이지 교체라고 한다.

어떤 페이지를 교체할 것이냐를 결정하는 알고리즘을 통해 페이징 부재를 줄여 쓰레싱을 방지하는 것을 목표로 한다.

💡 **Thrashing(쓰레싱)**
쓰레싱이란 메모리 영역에 접근하게 될 때, 메모리에 페이지 부재율이 높은 것을 의미하고, 이는 심각한 성능 저하를 초래한다.

쓰레싱의 원인

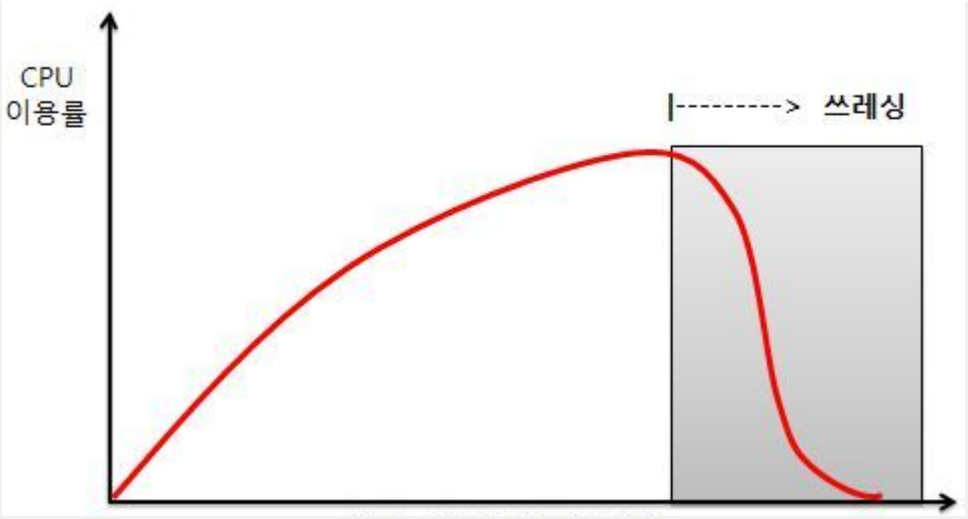
- 다중 프로그래밍 정도가 높아짐에 따라 CPU이용률이 높아지고, CPU 이용률이 최대값에 도달했을 때, 다중 프로그래밍의 정도가 그 이상으로 커지게 되면 쓰레싱이 일어나게 되고 CPU이용률은 급격히 떨어진다.
- 운영체제는 CPU이용률을 감시하며, CPU이용률이 너무 낮아지면 새로운 프로세스를 시스템에 추가하여 다중 프로그래밍의 정도를 높이게 된다. 이 때 전역 페이지 교체 알고리즘을 사용하여 어떤 프로세스의 페이지인지에 대한 고려없이 교체를 수행하게 된다.

  그런데, 이때 교체된 페이지들이 해당 프로세스에서 필요로 하는 것이었다면 그 프로세스 역시 페이지 폴트를 발생시키고 또 다른 프로세스에게서 프레임을 뺏어온다.(다른 프로세스가 필요로 하는 페이지를 탈락시키고 자신의 페이지로 채워넣는다.)

  이러한 프로세스들이 페이지 스왑 인, 스왑 아웃을 위해 페이징 장치를 사용해야 하는데, 이 장치에 대한 Queueing이 진행되며 Ready - Queue는 비게 된다.

  프로세스들이 페이징 장치를 기다리는 동안 CPU이용률을 떨어지게 되고, CPU스케줄러는 이용률이 떨어지는 것을 보고 높이기 위하여 새로운 프로세스를 추가하여 다중 프로그래밍의 정도를 더 높인다.

  새로 시작하는 프로세스는 실행 중인 프로세스들로부터 프레임을 가져오고자 하며, 더 많은 페이지 폴트와 더 긴 페이징 대기시간을 야기하게 된다.



쓰레싱은 지역교환 알고리즘이나 우선순위 교환 알고리즘을 사용해 제한할 수 있다.

쓰레싱을 해결하는 방법은?

- 각 프로세스가 필요로 하는 최소한의 프레임 갯수를 보장한다.
- locality를 이용한다.
- 다중 프로그래밍 정도를 낮춘다. → Working Set을 이용하여 다중 프로그래밍 정도를 낮출 수 있다.

Working Set

프로세스는 일정 시간 동안 특정 주소를 집중적으로 참조하는 경향이 있다.

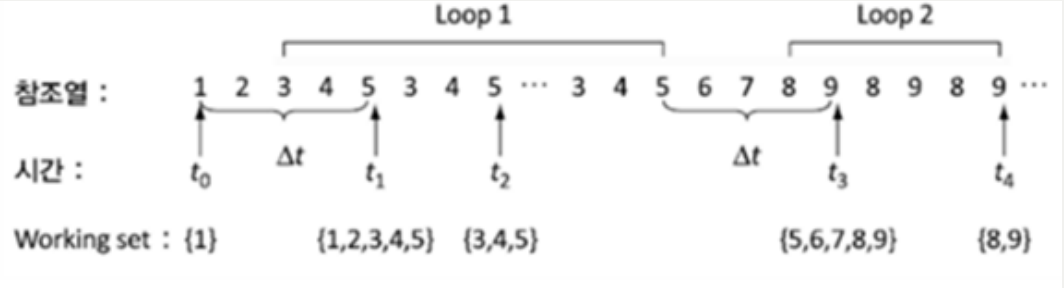
때문에 일정 시간 동안 참조한 페이지들을 메모리에 적재함으로써 페이지 부재를 최소화한다.

Working Set, W(t, Δt)는 프로세스가 [t – Δt ,t] 시간 동안 참조한 페이지들의 집합이다. Δt는 일정 크기의 시간으로 Window Size라 부르고 시스템에 의해 적절한 값으로 정해진다.

Window Size를 너무 크게 잡으면 지역성에 포함되는 페이지들 외에 다른 페이지들도 Working Set에 포함될 수 있게 되어 메모리의 낭비와 적절한 다중 프로그래밍 정도를 유지하기 힘들다.

너무 작게 잡으면 지역성에 포함되는 페이지들조차 Working Set에서 빠져 쓰레싱을 겪게 될 것이므로 시스템의 다양한 상황을 반영하여 결정해야 한다.

예시를 보자.



1. 프로그램의 실행이 시작된 후 t0부터 처음 참조되는 페이지들은 부재를 겪으며 점차 Working Set을 키워 가다가 t1에서 최대 크기인 5가 된다.
2. 이후부터 지역성을 보이면서 t2에서와 같이 (세 개의 페이지로 Loop 1을 구성하는 동안에는) 일정한 크기(=3)을 유지됨
3. 실행이 진행되면 다음 지역성을 보이는 Loop 2 부분으로 넘어가면서 t3에서 5개로 증가했던 Working Set은 점점 줄어들어 t4에서와 같이 페이지 두 개로 구성되는 강한 지역성을 보여줌

지역성을 표현하는 Working Set을 메모리에 유지함으로서 스레싱을 방지한다. 크기에 변동이 있으므로 Working Set이 작아지면 프레임을 회수하고 커지게 되면 그만큼 프레임을 더 할당해 주는 가변 할당이 필요하다는 것이다.

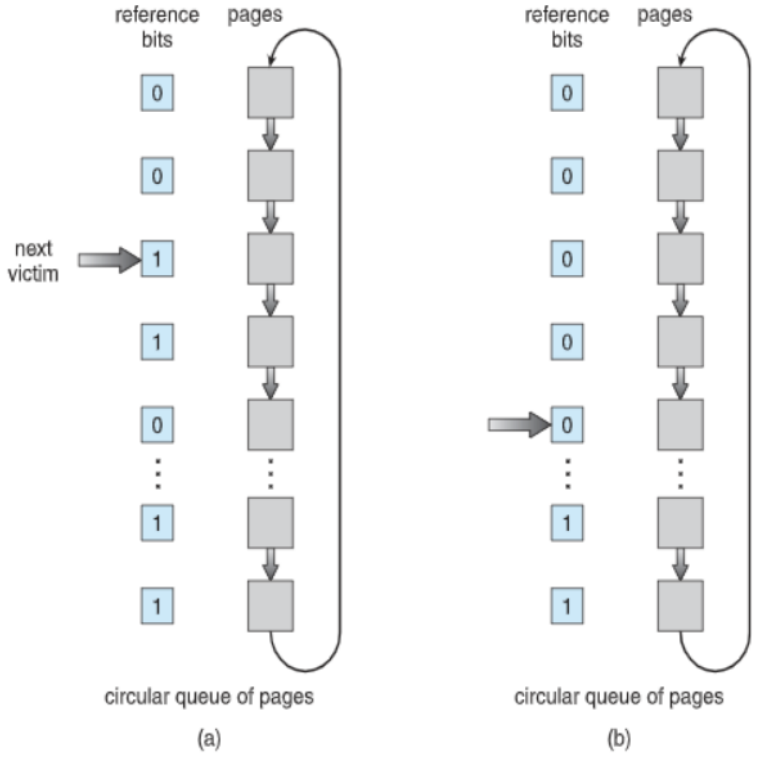
다음과 같은 알고리즘들이 있다.

- **FIFO**(First In First Out)
  - 가장 먼저 들어온 페이지를 교체한다.
- **LRU**(Least Recently Used)
  - 가장 오랫동안 사용되지 않은 페이지를 교체한다.
  - 이 알고리즘을 구현하기 위해서는 하드웨어의 지원이 필요하고, 아래 두 가지 방법이 가능하다.
    - 계수기(Counters) : 각 페이지 항목마다 최근 사용된 시간을 넣는다.

    - 스택(Stack) : 페이지 번호에 대한 스택을 유지하며, top이 가장 최근, bottom이 가장 오래된 페이지가 된다.

- **LRU Approximation**(LRU Approximation Page Replacement)
    - LRU page replacement를 충분히 지원할 수 있는 하드웨어는 거의 없다.
    - 그러나, 많은 시스템들이 reference bit의 형태로 어느 정도의 지원을 하려고 한다.
    - 처음에 모든 참조 비트는 0으로 초기화되고, 프로세스가 실행되면서 참조되는 페이지의 비트는 1로 바뀐다.

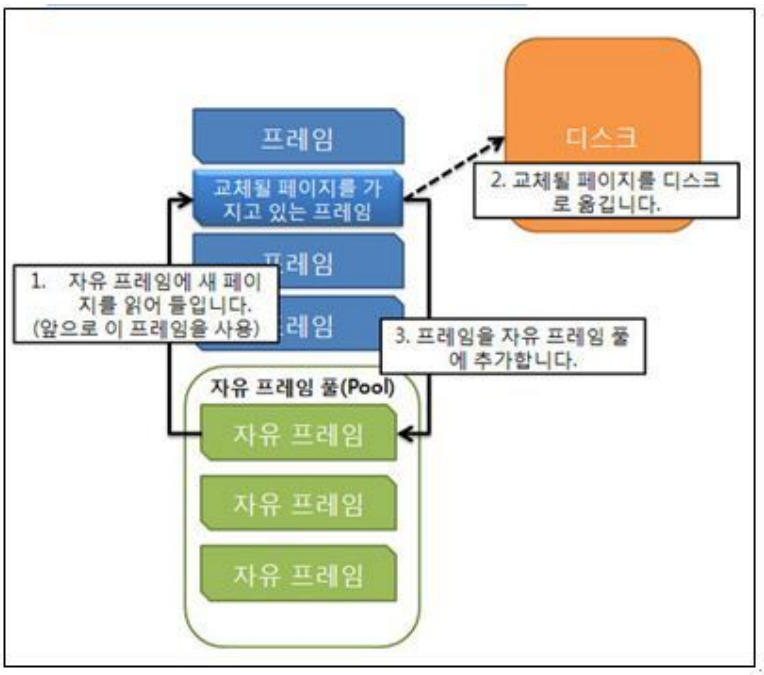
    1. 부가적 참조 비트 알고리즘(Additional - Reference Bits Algorithm)
        a. 일정한 간격마다 참조 비트를 기록함으로써 추가적인 선 후 관계 정보를 얻을 수 있다.
        b. 각 페이지에 대해 8비트의 참조 비트를 기록하고, 가장 최근 8구간 동안의 해당 페이지의 사용 기록을 담는다.
    2. 2차 기회 알고리즘(Second - Chance Algorithm)



        1. 2차 기회 알고리즘의 기본은 FIFO 알고리즘이다. 이와 함께 페이지가 선택될 때마다 참조 비트를 확인한다.
        2. 참조 비트가 0이면 페이지를 교체하고, 1이면 다시 한 번 기회를 준 뒤 FIFO로 넘어간다.

3. 개선된 2차 기회 알고리즘 (Enhanced Second-chance algorithm)
    a. Enhanced Second-chance algorithm이 아래에서 소개하는 **NUR**이다.

- **LFU**(Least Frequently Used)
    - 참조 횟수가 가장 작은 페이지를 교체한다.
        - 단점 : 가장 최근에 불러온 페이지가 교체될 수 있다. 이는 큰 오버헤드를 초래할 수 있다.
- **MFU**(Most Frequently Used)
    - 참조 횟수가 가장 많은 페이지를 교체한다.
        - 가장 많이 사용된 페이지가 앞으로는 사용되지 않을 것이라고 가정하는 것이다.
- **OPT**(Optimal)
    - 앞으로 가장 오랫동안 사용하지 않을 페이지를 내린다는 최적 알고리즘이다. 가장 이상적인 알고리즘이지만 미래를 예측하는 것을 기반으로 하기 때문에 불가능하고 성능 비교 목적으로 사용된다.
- **NUR**(Not Used Recently)
    - 추가 비트 2개(참조 비트, 변경 비트)만 사용하여 미래를 추정한다.
    - 다음과 같은 우선 순위로 교체될 페이지를 선정한다.
        1. (0, 0) : 최근에 사용되지도 변경되지도 않은 경우 (교체하기 가장 좋은 페이지)
        2. (0, 1) : 최근에 사용되지는 않았지만, 변경은 된 경우 (교체에 적당하지 않은 페이지)
        3. (1, 0) : 최근에 사용은 되었으나, 변경은 되지 않은 경우 (다시 사용될 가능성이 높은 페이지)
        4. (1, 1) : 최근에 사용도 되었고, 변경도 된 경우 (다시 사용될 가능성이 높으며 교체에 적당하지 않은 페이지)
    - 만약 모든 페이지가 (1, 1)이 되면 모든 페이지 비트를 (0, 0)으로 초기화한다.
- **Page-Buffering Algorithm → 정확히 이해가 안됨**
    - 여러 개의 자유 프레임을 pool에 넣어 가지고 있다가 페이지 교체가 필요할 때 자유 프레임에 새로운 페이지를 읽어 들이고, 희생된 프레임을 찾은 뒤 프레임을 디스크에 기록하고 자유 프레임 풀에 추가한다.



https://www.youtube.com/watch?v=TVpBx63trhI

프레임의 할당(Allocation of Frames)

여러 개의 프로세스들에 대해 제한된 메모리를 어떻게 할당할 것인가에 대한 문제이다.

최소로 할당해야 할 프레임의 수(Minimum Number of Frames)

- 페이지 공유가 없다면, available frame 수보다 더 많이 할당할 수는 없지만, 너무 작게 할당해서는 안 된다.
- 각 프로세스에 할당되는 frame 수가 줄어들면 page fault rate는 증가하고, 프로세스 실행은 늦어지게 된다.

할당 알고리즘(Allocation of Algorithms)

- 가장 쉬운 할당 방법은 모든 프로세스에게 똑같이 할당해 주는 방법이다.
- 예를 들어 93개의 frame과 5개의 프로세스가 있을 경우, 각 프로세스는 18개의 frame을 할당받는다.
- 나머지 3개의 frame은 free frame buffer pool로 활용한다.
- 이런 방법을 균등 할당(Equal Allocation)이라고 한다.

- 그러나, 10KB와 132KB의 프로세스가 있을 경우, 균등 할당은 좋은 방법이 아니다

- 이에, 각 프로세스의 크기 비율에 맞춰 frame을 할당하는 비례 할당 방식(Proportional allocation)을 사용할 수 있다.
- 균등 할당과 비례 할당은 모두 프로세스의 우선순위를 고려하지 않는 방식이다.

전역 대 지역 할당(Global vs Local Allocation)

- 다수의 프로세스가 frame할당을 위해 경쟁하는 환경에서 페이지 교체 알고리즘은 크게 두 가지 범주로 나뉜다.
- (1) **전역 교체** (Global Replacement) (2) **지역 교체** (Local Replacement)

- **전역 교체**는 프로세스가 교체할 frame을 다른 프로세스에 속한 frame을 포함한 모든 프레임을 대상으로 찾는 경우이다.
- **지역 교체**는 각 프로세스가 자기에게 할당된 frame들 중에서만 교체될 victim을 선택할 수 있는 경우이다.

- **전역 교체**에서는 한 프로세스에 할당된 프레임의 수는 바뀔 수 있다.
- **지역 교체**에서는 프로세스에 할당된 프레임의 수는 변하지 않는다.

- **전역 교체** 알고리즘에서의 한 가지 문제점은 한 프로세스가 자신의 page fault rate를 조절할 수 없다는 것이다.
  - 한 프로세스의 page fault rate는 그 프로세스가 어떤 프로세스들과 함께 실행되느냐에 영향을 받는다.
- **지역 교체** 알고리즘의 경우 다른 프로세스에게 영향을 받지 않는데, 유일하게 그 프로세스의 페이징 형태에만 영향을 받기 때문이다.

- 일반적으로 **전역 교체**가 **지역 교체** 알고리즘보다 더 좋은 성능을 보이고, 더 자주 쓰인다.

비균등 메모리 접근(Non-Uniform Memory Access)

- 메모리 접근 시간이 현저하게 차이가 나는 시스템을 모두 비균등 메모리 접근(NUMA)라고 한다.

- 이러한 시스템에서 메모리를 동등하게 대하면, NUMA 구조를 고려한 메모리 할당 알고리즘을 사용하는 시스템에서보다 CPU가 메모리를 접근할 때 대기 시간이 매우 길어지게 된다.

커널 메모리의 할당

- 커널 메모리는 보통 사용자 모드 프로세스에게 할당해 주기 위한 페이지 리스트와는 별도의 메모리 풀에서 할당받는다.
- 그 이유는 아래 2가지와 같다.
  1. 커널은 다양한 크기의 자료 구조를 위해 메모리를 할당받는다. 이 자료구조들은 페이지 크기보다 작은 크기를 갖기도 한다. 때문에 커널은 메모리를 조심스럽게 사용해야 하고, **단편화에 의한 낭비를 줄여야 한다**.
  2. 사용자 모드 프로세스에 할당되는 페이지는 물리 메모리 상에서 꼭 연속적일 필요가 없다. 하지만, 특정 하드웨어 장치는 물리적으로 연속적인 메모리를 요구할 수 있다.
- Buddy System Allocator와 Slab Allocator 두 가지 방식이 존재한다.

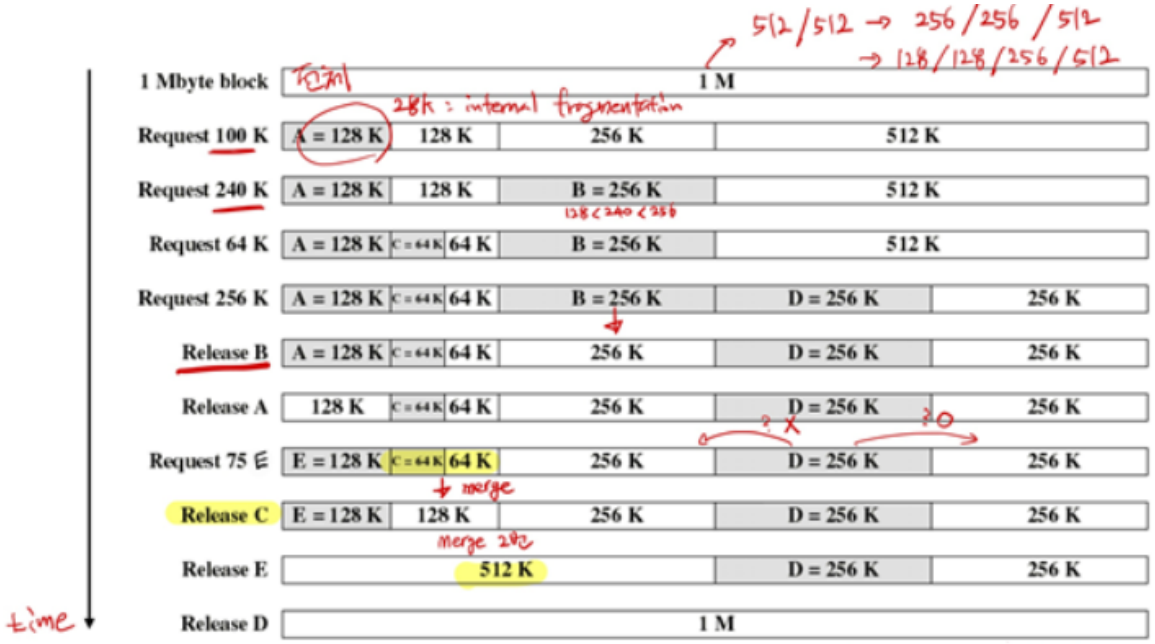
버디 시스템(Buddy System Allocator)

- 버디 시스템은 Fixed, Dynamic Partitioning 기법의 fragmentation 발생 이슈를 보완하기 위한 방법이다. 이 방법은 실제로 Linux OS에서 사용한다.

> 1. 메인 메모리는 항상 2^N의 크기로 할당한다.
> 2. 사용할 수 있는 가장 큰 메모리부터 시작해서 Binary로 절반씩 쪼개나가면서 아래 조건을 만족시키는 공간을 찾는다.
> 3. 만약 프로세스 메모리 크기가 K라고 하면, 2^(U-1) < K ≤ 2^N을 만족시키는 U를 찾고, 2^U 만큼 공간에 프로세스를 할당한다. 예를 들어, 프로세스 크기가 1000B이면, 512B < 1000B < 1024B이기 때문에 1024B 사이즈의 메모리에 할당하는 것이다.
> 4. 프로세스가 종료되고, 만약 같은 Parent를 갖는 Buddy 공간이 비어있다면 Merge한다.

위 과정을 조금 더 디테일하게 들여다보면, 메모리를 할당할 때는 가장 하위 메모리부터 탐색을 해가면서, 메모리를 할당할 수 있는 가장 작은 크기의 Block을 선택해서 쪼개거나 할당

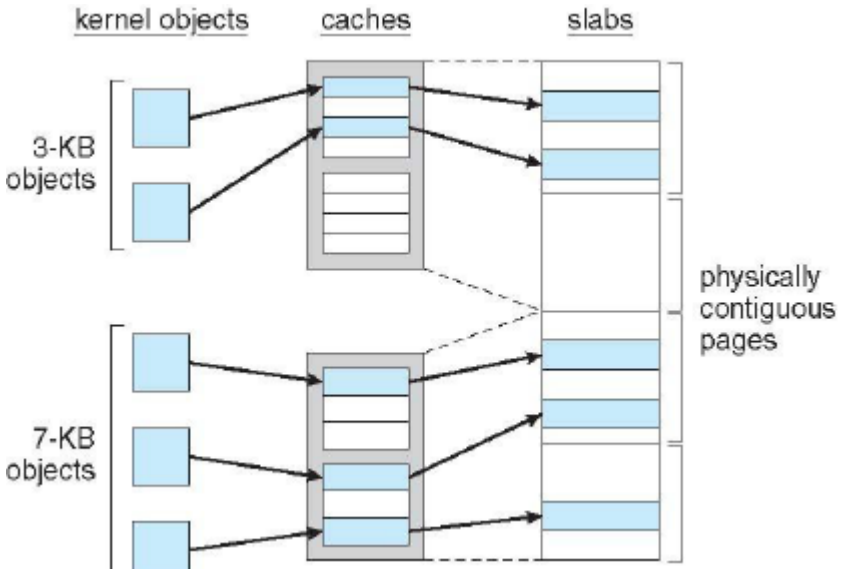
한다. 아래 그림을 통해 이해하자.



버디 시스템을 쓰게 될 때 장단점은 무엇일까? 버디 시스템은 Dynamic Partitioning에 비해 External Fragmentation이 적고, 하위 메모리로부터 공간을 사용하기 때문에 자동적으로 Compaction에 대한 오버헤드가 적다. 하지만, 2^N 크기의 메모리를 무조건 할당해야 하기 때문에 **Internal Fragmentation이 필연적으로 존재**할 수밖에 없다.

- Compaction : 작은 공간들을 모아서 연속적인 공간으로 만드는 작업이다.

Slab Allocator → 이해가 잘 안됨



- Kernel은 여러 종류의 Data Structure를 사용하며 이를 통틀어서 Object라고 한다.
- Object는 종류가 많으므로 크기가 제각각인데 같은 크기의 Object에 대해서 Cache에서 저장하여 값을 가지고 있다.
- Cache에서 각 크기마다 Object를 가지고 있게 되어서 Internal Fragmentation을 없앨 수 있다.
- 물리적인 연속된 Page들의 묶음을 Slab이라고 하며 같은 Cache에도 2개의 Slab이 저장될 수 있다.
- Slab 공간이 부족해지면 Buddy Allocator로 그 공간을 늘릴 수 있다.
- Slab Allocator는 미리 할당하는 방법으로 Cache에 저장해 두었다가, 사용되면 Used로 표시하고, 사용하지 않으면, Free로 Object에 표시한다.
- bit 전환만으로 사용, 비사용으로 관리하기 때문에 속도가 빠르다.
- 주소 번역이 필요 없다.

기타 고려 사항

프리 페이징(Prepaging)

- Spatial Locality에 기반하여 참조되는 페이지 외에도 주변 페이지들이 참조될 가능성이 높기에 미리 로드해 놓는 것이다.
- 요즘에는 많이 사용되지 않는다. 대신 prepaging과 비슷한 기법이 있다.

> 💡 **Request Paging**
>
> 논리적으로 연관성이 있는 페이지들을 미리 메인 메모리로 로드하는 기법

페이지 크기(Page Size)

- 페이지 테이블의 크기를 작게 유지하기 위해서는 큰 크기의 페이지가 좋다.
- 할당해 준 메모리 사용 효율을 높이기 위해서는 작은 크기의 페이지가 좋다.(내부 단편화를 줄임)

TLB Reach

- TLB Reach란 TLB로부터 엑세스할 수 있는 메모리 공간의 크기이다.
  - TLB(translation-lookaside buffer)는 MMU안에 있는 특수 소형의 캐시이다.
- TLB Reach = (TLB에 들어갈 수 있는 페이지 개수) x (page size)
- 여러 크기의 페이지를 수용하려면 TLB는 운영체제가 관리를 해야 한다. 그러면 소프트웨어적인 오버헤드가 발생하는데 hit ratio와 TLB reach의 향상으로 상쇄될 수 있다.

역 페이지 테이블(Inverted Page Table)

- 8.6.3절에서 역 페이지 테이블의 목적은 페이지 테이블이 차지하는 메모리 공간을 줄이기 위함이었다.
- 이 테이블은 <process-id, page-number>에 의해 index 되며, 물리 메모리마다 한 항목을 갖는다.
- 각 페이지 프레임에 어떤 가상 메모리 페이지가 저장되어 있는지의 정보만 유지하면 되기 때문에 필요한 물리 메모리 양을 줄인다.

프로그램 구조(Program Structure)

```
for (i = 0; i < 128; i++)
  for (j = 0; j < 128; j++)
    data[i,j] = 0;
```

```
for (i = 0; i < 128; i++)
  for (j = 0; j < 128; j++)
    data[j,i] = 0;
```

- 왼쪽의 구조가 더 좋다.

이 외에도 입출력 상호 잠금과 페이지 잠금 등을 고려해야 한다.

대표적인 OS의 예시들

Linux

Windows

Solaris

Reference :

https://velog.io/@wngud4950/CS운영체제OS-가상-메모리-Virtual-Memory

https://jwprogramming.tistory.com/56

https://velog.io/@fldfls/Trashing-working-set

https://itdexter.tistory.com/410

https://will-behappy.tistory.com/25

https://will-behappy.tistory.com/25

https://studyandwrite.tistory.com/18

https://blog.naver.com/kseo712/220883220425