**Chapter 14 가상 메모리**

***14-1*** **연속 메모리 할당**

**연속 메모리 할당**: 프로세스에 연속적인 메모리 공간을 할당하는 방식

**<스와핑>**

**스와핑**: 메모리에 적재된 프로세스들 중 현재 실행되지 않는 프로세스를 임시로 보조기억장치 일부 영역으로 쫓아내고, 그렇게 해서 생긴 메모리상의 빈 공간에 또 다른 프로세스를 적재하여 실행하는 방식

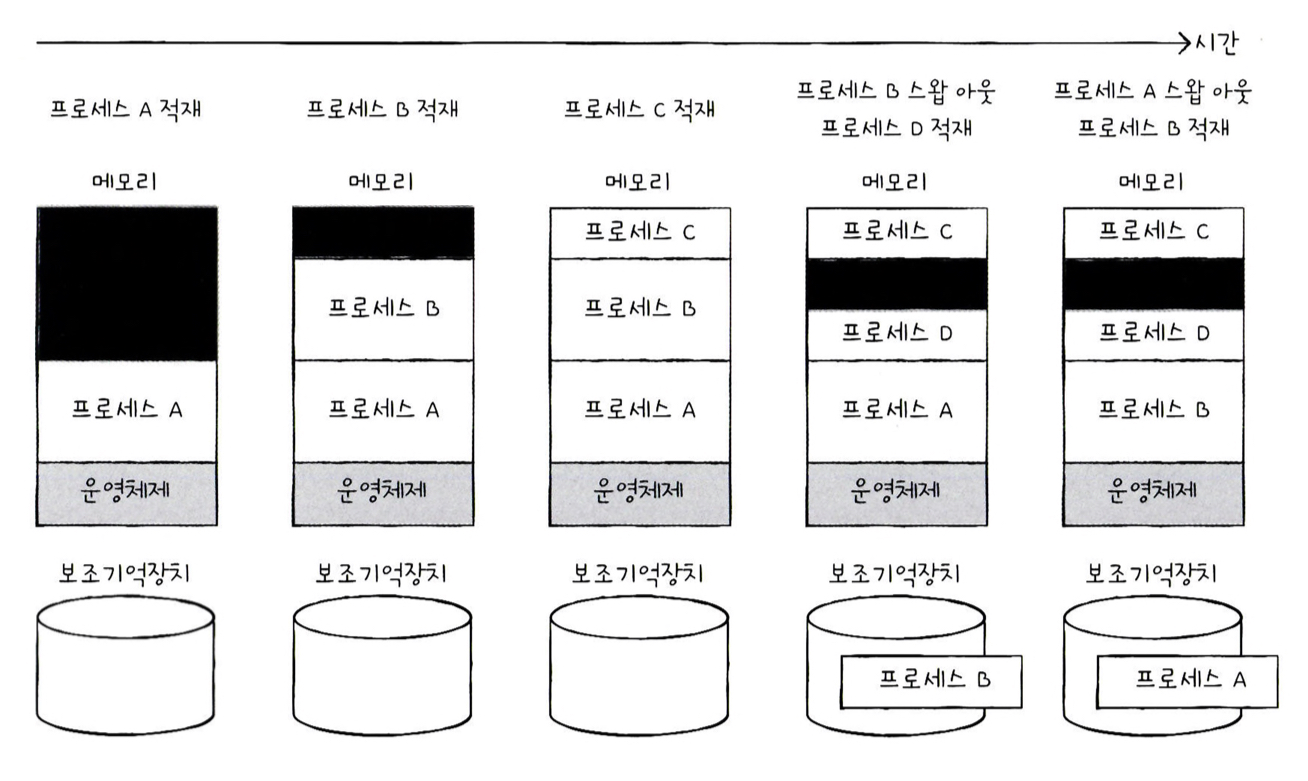
* 프로세스들이 쫓겨나는 보조기억장치의 일부 영역: **스왑영역**

**스왑 아웃**: 현재 실행되지 않는 프로세스가 메모리에서 스왑영역으로 옮겨지는 것

**스왑 인**: 스왑 영역에 있던 프로세스가 다시 메모리로 옮겨오는 것

* 스왑 아웃-> 스왑 인 시 물리 주소는 스왑 아웃되기 전의 물리 주소와는 다른 주소에 적재될 수 있음

스와핑의 **장점**: 프로세스들이 요구하는 메모리 주소 공간의 크기가 실제 메모리 크기보다 큰 경우에도 프로세스들을 동시 실행할 수 있음



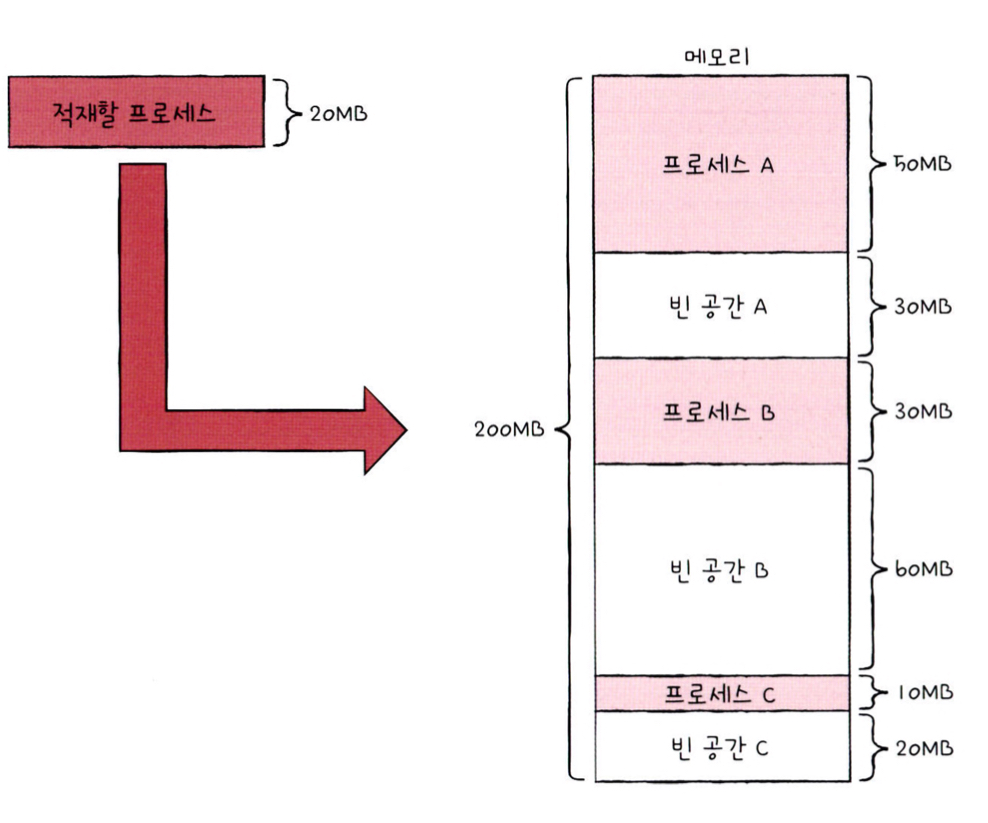
**<메모리 할당>**

비어 있는 메모리 공간에 프로세스를 연속적으로 할당하는 방식에는 **최초 적합, 최적 적합, 최악 적합**의 세 가지 방식이 존재

1. **최초 적합**:

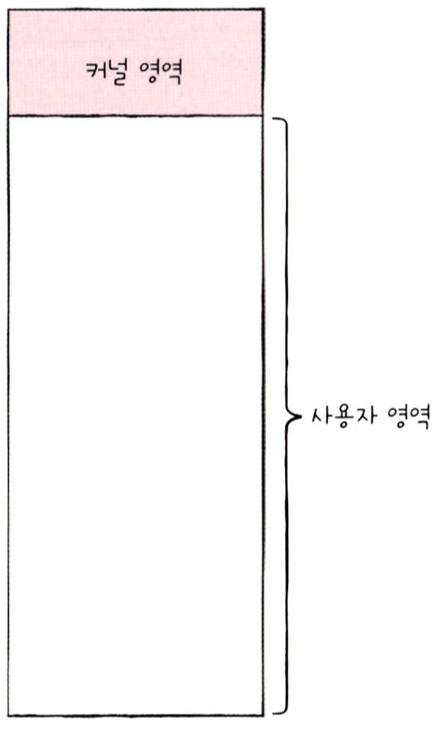
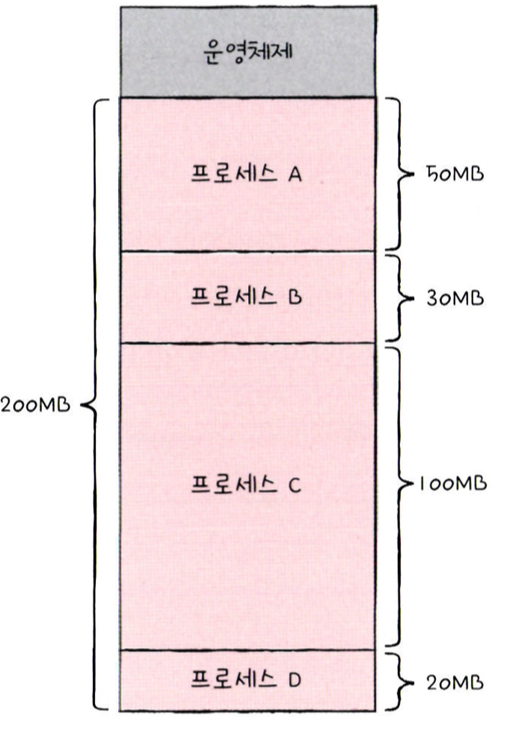
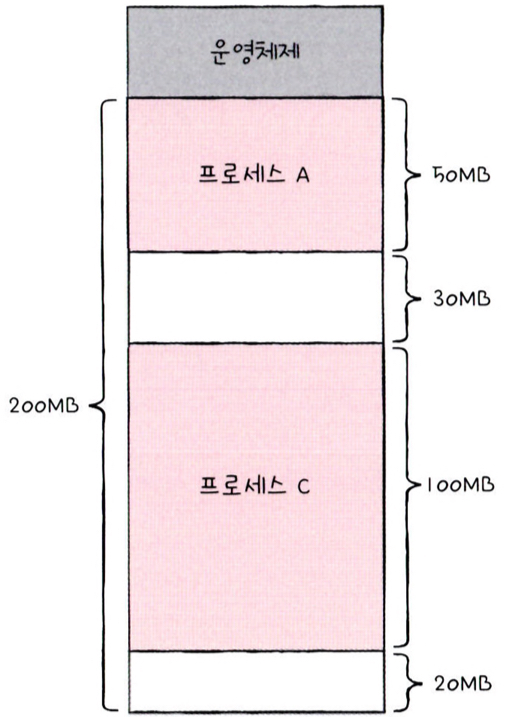
* 운영체제가 메모리 내의 빈 공간을 순서대로 검색하다가 적재할 수 있는 공간을 발견하면 그 공간에 프로세스를 배치하는 방식
* 검색을 최소화할 수 있고 빠른 할당이 가능

1. **최적 적합**: 운영체제가 빈 공간을 모두 검색해 본 후, 프로세스가 적재될 수 있는 공간 중 가장 작은 공간에 프로세스를 배치하는 방식
2. **최악 적합**: 운영체제가 빈 공간을 모두 검색해 본 후, 프로세스가 적재될 수 있는 공간 중 가장 큰 공간에 프로세스를 배치하는 방식



**<외부 단편화>**

연속 메모리 할당은 효율적인 방법이 아님 -> **외부 단편화**라는 문제를 내포하기 때문

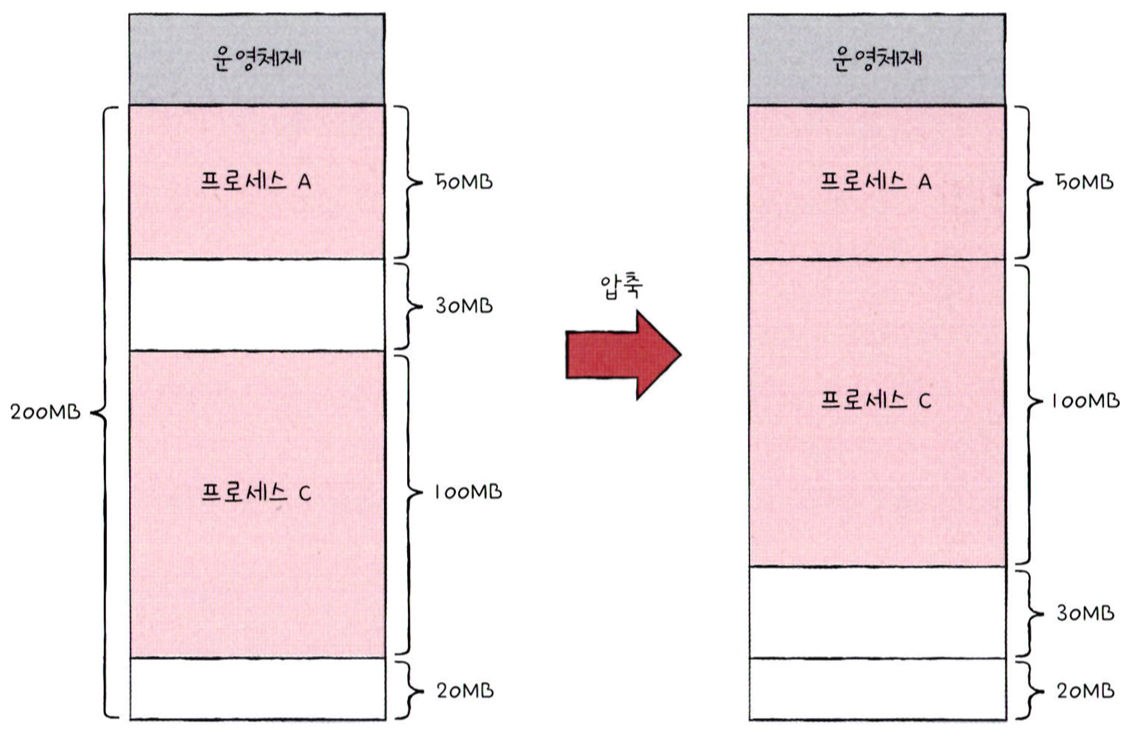
  

**외부 단편화**: 프로세스들이 실행되고 종료되기를 반복하며 메모리 사이 사이에 빈 공간들이 생기는데 이 빈 공간에는 그 보다 큰 프로세스를 적재하기 어려움 -> 메모리 낭비

메모리 **압축**(메모리 조각 모음): 외부 단편화를 해결할 수 있는 대표적인 방안으로 여기저기 흩어져 있는 빈 공간들을 하나로 모으는 방식. 메모리 내에 저장된 프로세스를 적당히 재배치 시켜 여기저기 흩어진 작은 빈 공간들을 하나의 큰 빈공간으로 만드는 방법

메모리 압축의 **단점**

* 빈 공간들을 모으는 동안 시스템은 하던일을 중지해야 함
* 메모리에 있는 내용을 옮기는 작업은 많은 오버헤드를 야기
* 어떤 프로세스를 어떻게 움직여야 오버헤드를 최소화하며 압축 가능한 지에 대한 명확한 방법을 결정하기 어려움



**메모리 압축**

외부 단편화를 없앨 수 있는 또 다른 해결 방안으로 **가상 메모리 기법**이 등장했고 대표적으로 **페이징 기법**이 있다

***14-2*** **페이징을 통한 가상 메모리 관리**

**가상 메모리**: 실행하고자 하는 프로그램을 일부만 메모리에 적재하여 실제 물리 메모리 크기보다 더 큰 프로세스를 실행할 수 있게 하는 기술

**<페이징이란>**

연속 메모리 할당 방식에서 외부 단편화가 생긴 근본적인 이유는 **각기 다른 크기**의 프로세스가 메모리에 **연속적**으로 할당되었기 때문 -> 메모리와 프로세스를 **일정한 단위**로 자르고 메모리에 **불연속적**으로 할당할 수만 있다면 외부 단편화는 발생하지 않음

**페이징**: 프로세스의 논리 주소 공간을 **페이지**라는 일정한 단위로 자르고, 메모리 물리 주소 공간을 **프레임**이라는 페이지와 동일한 크기의 일정한 단위로 자른 뒤 페이지를 프레임에 할당하는 가상 메모리 관리 기법

페이징에서도 스와핑을 사용 가능: 페이지 단위로 스왑 아웃/스왑 인

메모리에 적재될 필요가 없는 페이지들은 보조기억장치로 스왑 아웃(**페이지 아웃**)

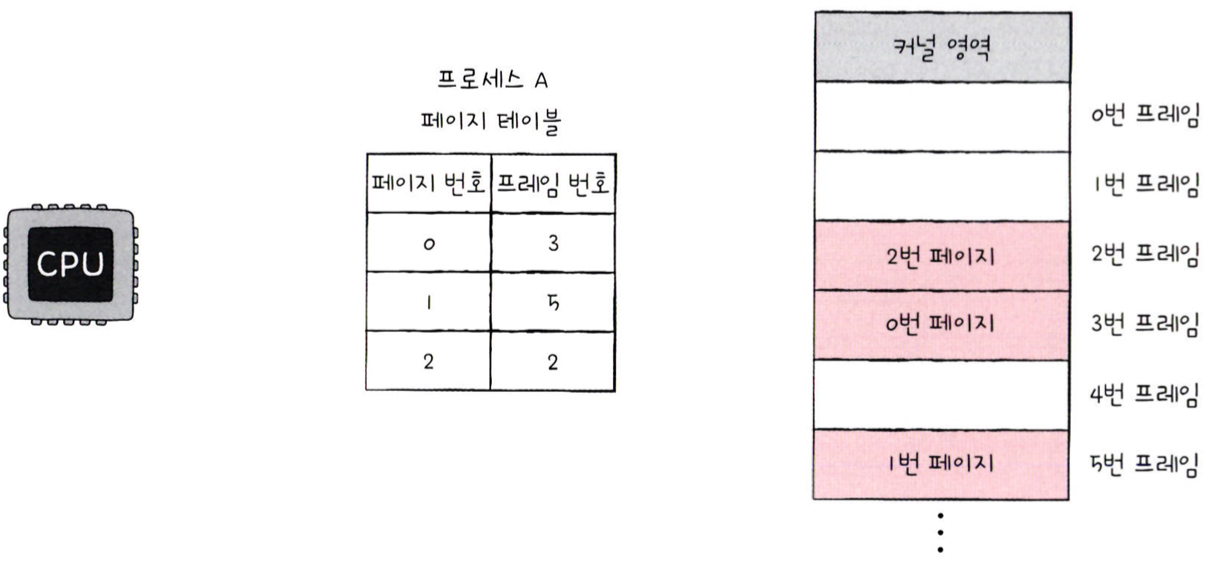
실행에 필요한 페이지들은 메모리로 스왑 인(**페이지 인**)

-> 물리 메모리보다 더 큰 프로세스를 실행 가능

**<페이지 테이블>**

프로세스가 메모리에 불연속적으로 배치되어있다면 CPU입장에서는 이를 순차적으로 실행할 수가 없음 -> 이를 해결하기 위해 페이지 테이블을 이용

**페이지 테이블**: 페이지 번호와 프레임 번호를 짝지어 주는 일종의 이정표. 실제 메모리 내의 물리주소가 불연속적으로 배치되어있더라도 CPU가 바라보는 논리 주소에는 연속적으로 배치되도록 하여 CPU가 페이지 번호만 보고 해당 페이지가 적재된 프레임을 찾을 수 있게 해준다.



* 프로세스마다 각자의 프로세스 테이블 존재
* 각 프로세스의 페이지 테이블들은 메모리에 적재
* CPU 내의 페이지 테이블 베이스 레지스터(PTBR)는 각 프로세스의 페이지 테이블이 적재된 주소를 가리킴
* 페이지 테이블을 메모리에 두면 메모리 접근 시간이 두 배로 늘어나는 문제

-> 페이지 테이블을 보기 위해 한 번, 프레임에 접근하기 위해 한 번

**TLB**

* 메모리에 두 번 접근하는 문제를 해결하기 위해 CPU곁에 페이지 두는 테이블 캐시 메모리
* 페이지 테이블의 일부 내용을 저장
* 참조 지역성에 근거해 주로 최근에 사용된 페이지 위주로 가져와 저장
* CPU가 발생한 논리 주소에 대한 페이지 번호가 TLB에 있을 경우 **TLB 히트**

-> 페이지 테이블을 보기 위해 메모리에 접근할 필요 없음

* 페이지 번호가 TLB에 없을 경우 페이지 테이블을 보기 위해 메모리에 접근 해야함 -> **TLB 미스**

**<페이징에서의 주소 변환>**

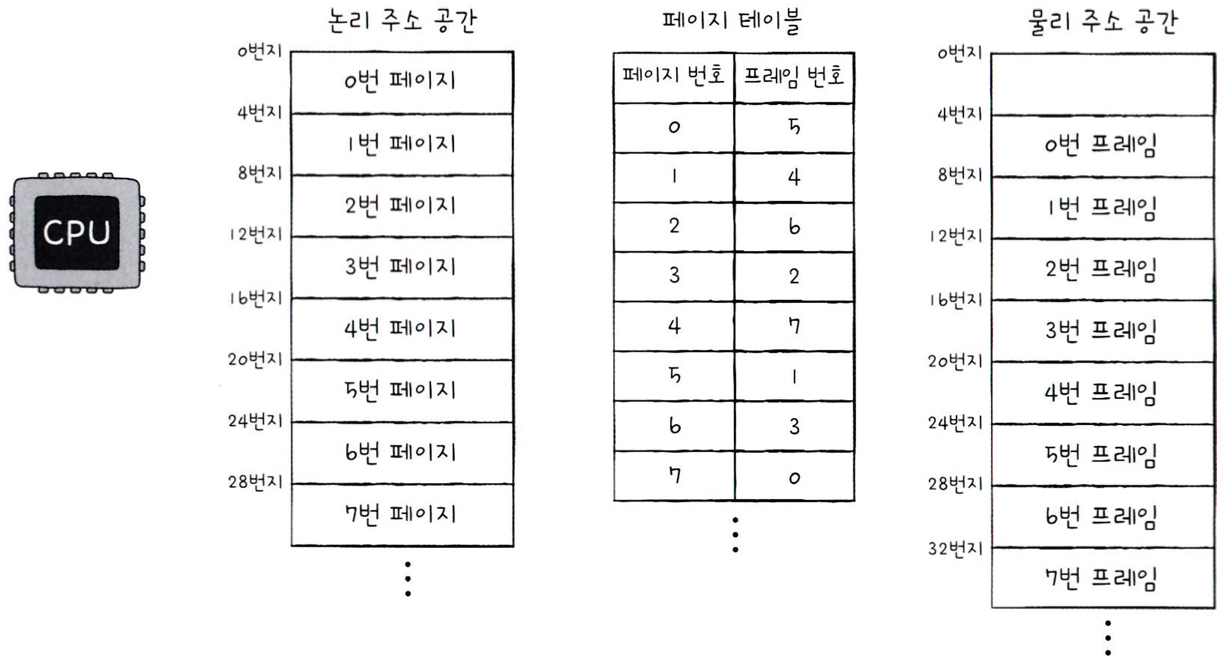
특정 주소에 접근하려고 할 때 필요한 두 가지 정보

* 어떤 페이지 혹은 프레임에 접근하고 싶은지
* 접근하려는 주소가 그 페이지 혹은 프레임으로부터 얼마나 떨어져 있는지

페이징 시스템에서는 모든 논리 주소가 기본적으로 **페이지번호**와 **변위**로 이루어짐

**페이지 번호**: 접근하고자 하는 페이지 번호. 페이지 번호를 찾으면 페이지가 어떤 프레임에 할당되었는지 알수 있음

**변위**(offset): 접근하려는 주소가 프레임의 시작 번지로부터 얼만큼 떨어져 있는지 알기 위한 정보



페이지 번호:5 변위:2

CPU가 접근하게 될 물리 주소는?

**<페이지 테이블 엔트리>**

**페이지 테이블 엔트리**

* 페이지 테이블의 각각의 행
* 페이지 번호, 프레임 번호 이외에도 다른 중요한 정보들이 담김

-> 대표적으로 유효 비트, 보호 비트, 참조 비트, 수정 비트가 있음

**유효 비트**

* 현재 해당 페이지에 접근 가능한지 여부를 알려줌
* 현재 페이지가 메모리에 적재되어 있는지 보조기억장치에 있는지 알려주는 비트
* 페이지가 메모리에 적재되어 있다면 -> 유효 비트가 1

페이지가 보조기억장치에 있다면 -> 유효 비트가 0

* CPU가 유효 비트가 0인 페이지에 접근하려고 하면 **페이지 폴트**라는 예외 발생
* 페이지 폴트의 처리 과정은 하드웨어 인터럽트 처리 과정과 유사

1. CPU는 기존의 작업 내역을 백업
2. 페이지 폴트 처리 루틴 실행
3. 페이지 처리 루틴은 원하는 페이지를 메모리에 가져온 뒤 유효 비트를 1로 변경
4. 페이지 폴트 처리 후 CPU는 해당 페이지에 접근 가능

**보호 비트**

* 페이지 보호 기능을 위해 존재하는 비트
* 해당 페이지가 읽고 쓰기가 모두 가능한지, 읽기만 가능한지 나타냄
* 비트가 0인 경우 -> 읽기만 가능

비트가 1인 경우 -> 읽고 쓰기 모두 가능

* 읽기 전용인 페이지에 쓰기를 시도하면 운영체제가 막아줌으로써 페이지를 보호
* 세 개의 비트로 조금 더 복잡하게 구현 가능

r(읽기 Read), w(쓰기 Write), x(실행 eXecute)

ex) 보호비트가 111인 경우 읽기, 쓰기, 실행 모두 가능

**참조 비트**

* CPU가 이 페이지에 접근한 적이 있는지 여부를 나타냄
* 적재 이후 CPU가 읽거나 쓴 페이지 -> 참조 비트 1

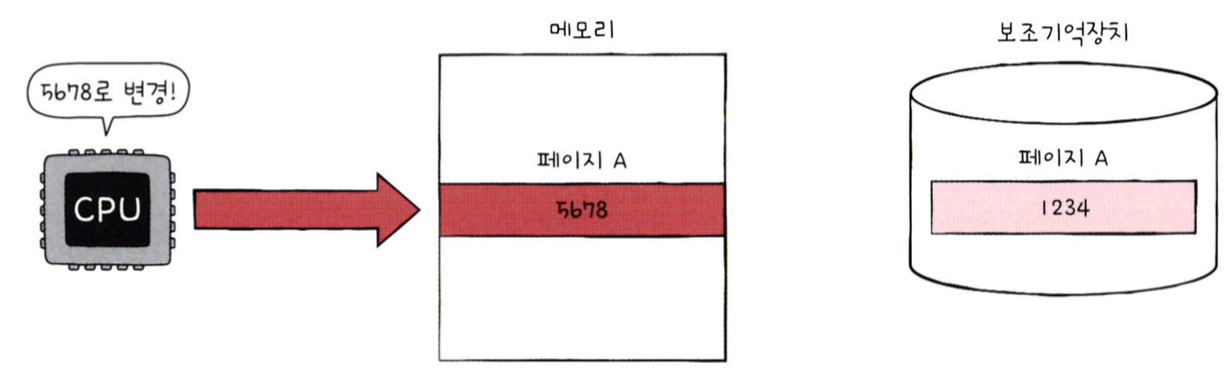
적재 이후 한 번도 읽거나 쓴 적이 없는 페이지 -> 참조 비트 0

**수정 비트(더티 비트)**

* 해당 페이지에 데이터를 쓴 적이 있는지 없는지 수정 여부를 알려줌
* 비트가 1인 경우 -> 변경된 적이 있는 페이지

비트가 0인 경우 -> 변경된 적이 없는 페이지(접근한 적 없거나, 읽기만 했던 페이지)

* 페이지가 메모리에서 사라질 때 보조기억장치에 쓰기 작업을 해야하는지, 할 필요가 없는지 판단하기 위해 존재
* 한 번도 수정된 적이 없는 페이지가 스왑 아웃 될 경우 추가 작업 없이 새로 적재된 페이지로 덮어쓰기만 하면 됨 -> 어차피 똑같은 페이지가 보조기억장치에 저장되어 있기 때문
* 수정된 적이 있는 페이지의 경우 보조기억장치에 저장된 페이지의 내용과 메모리에 저장된 페이지의 내용은 서로 다른 값을 갖게 됨 -> 페이지 스왑 아웃 시 변경된 값을 보조기억장치에 기록하는 작업이 추가 되어야 함



수정 비트가 1인 경우

페이징은 외부 단편화 문제를 해결할 수 있지만 **내부 단편화** 문제를 야기할 수 있음

-> 일정한 크기로 자른 페이지에 모든 프로세스들이 딱 들어맞는 것은 아님

ex) 페이지 크기: 10KB, 프로세스 크기: 108KB -> 2KB의 페이지가 남음(메모리 낭비)

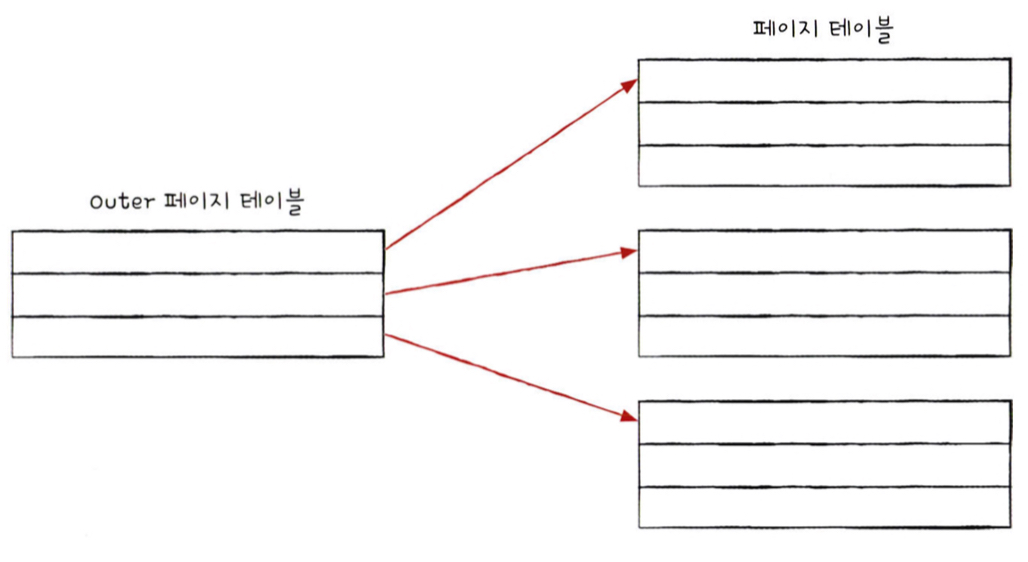
* 내부 단편화는 하나의 페이지 크기보다 작은 크기로 발생
* 하나의 페이지 크기가 작다면 발생하는 내부 단편화의 크기는 작아질 것
* 하지만 하나의 페이지 크기를 너무 작게 설정하면 -> 페이지 테이블의 크기는 커지고 페이지 테이블이 차지하는 공간이 낭비

페이징의 이점 - 쓰기 시 복사

* 프로세스 간에 페이지 공유 가능
* 부모 프로세스에서 자식 프로세스 생성 시 동일한 프레임 가리킴 -> 굳이 복사하지 않아도 동일한 코드 및 데이터 영역을 가리킬 수 있음
* 부모 프로세스, 자식 프로세스 둘 중 하나가 페이지에 쓰기 작업을 하면 그 순간 해당 페이지가 별도의 공간으로 복제(쓰기 시 복사) -> 프로세스 생성 시간 줄일 수 있고 메모리 공간 절약도 가능

계층적 페이징(다단계 페이지 테이블)

* 페이지 테이블을 페이징하여 여러 단계의 페이지를 두는 방식
* 프로세스의 페이지 테이블을 여러 개의 페이지로 자르고, 바깥쪽에 페이지 테이블을 하나 더 두어 잘린 페이지 테이블의 페이지들을 가리키는 방식



* 모든 페이지 테이블을 항상 메모리에 유지할 필요가 없음. 일부는 보조기억장치에 있어도 무방하며 추후 해당 페이지 테이블을 참조해야 할 때 메모리에 적재

-> 막대한 크기의 페이지 테이블로 인해 낭비되는 공간을 줄일 수 있음

* 계층적 페이징을 사용하는 경우 CPU가 발생하는 논리주소도 달라짐

1. 바깥 페이지 번호를 통해 페이지 테이블의 페이지를 찾기
2. 페이지 테이블의 페이지를 통해 프레임 번호를 찾고 변위를 더함으로서 물리 주소 얻기

* 페이지 테이블의 계층이 늘어날수록 페이지 폴트가 발생했을 경우 메모리 참조 횟수가 많아지므로 계층이 많다고 해서 반드시 좋다고 볼 수는 없음

***14-3*** **페이지 교체와 프레임 할당**

**<요구 페이징>**

**요구 페이징**: 프로세스를 메모리에 적재할 때 처음부터 모든 페이지를 적재하지 않고 필요한 페이지만을 메모리에 적재하는 기법

요구 페이징의 기본적인 양상

1. CPU가 특정 페이지에 접근하는 명령어를 실행
2. 해당 페이지가 현재 메모리에 있을 경우(유효 비트가 1) CPU는 페이지가 적재된 프레임에 접근
3. 해당 페이지가 현재 메모리에 없을 경우(유효 비트가 0) 페이지 폴트 발생
4. 페이지 폴트 처리 루틴은 해당 페이지를 메모리로 적재하고 유효비트를 1로 설정
5. 다시 1번을 수행

**순수 요구 페이징**: 아무런 페이지도 메모리에 적재하지 않은 채 무작정 실행하여 프로세스의 첫 명령어를 실행하는 순간부터 페이지 폴트가 계속 발생하게 되고 실행에 필요한 페이지가 어느 정도 적재된 이후부터는 페이지 폴트 발생 빈도가 떨어지는 기법

요구 페이징 시스템이 안정적으로 작동하려면 **페이지 교체**와 **프레임 할당**을 해결해야 함

요구 페이징 기법으로 페이지들을 적재하다 보면 언젠가 메모리가 가득 참 -> 당장 실행에 필요한 페이지를 적재하기 위해 메모리에 적재된 페이지를 보조기억장치로 내보내야 함

* 어떤 페이지를 내보내는 것이 최선일까? 이를 결정하는 방법이 **페이지 교체 알고리즘**

**<페이지 교체 알고리즘>**

좋은 페이지 교체 알고리즘이란? 페이지 폴트를 가장 적게 일으키는 알고리즘

* 페이지 교체 알고리즘을 제대로 이해하려면 **페이지 폴트 횟수**를 알 수 있어야 함
* 페이지 폴트 횟수는 **페이지 참조열**을 통해 알 수 있음
* 페이지 참조열: CPU가 참조하는 페이지들 중 연속된 페이지를 생략한 페이지열
* CPU가 아래와 같은 순서로 페이지에 접근 했다면



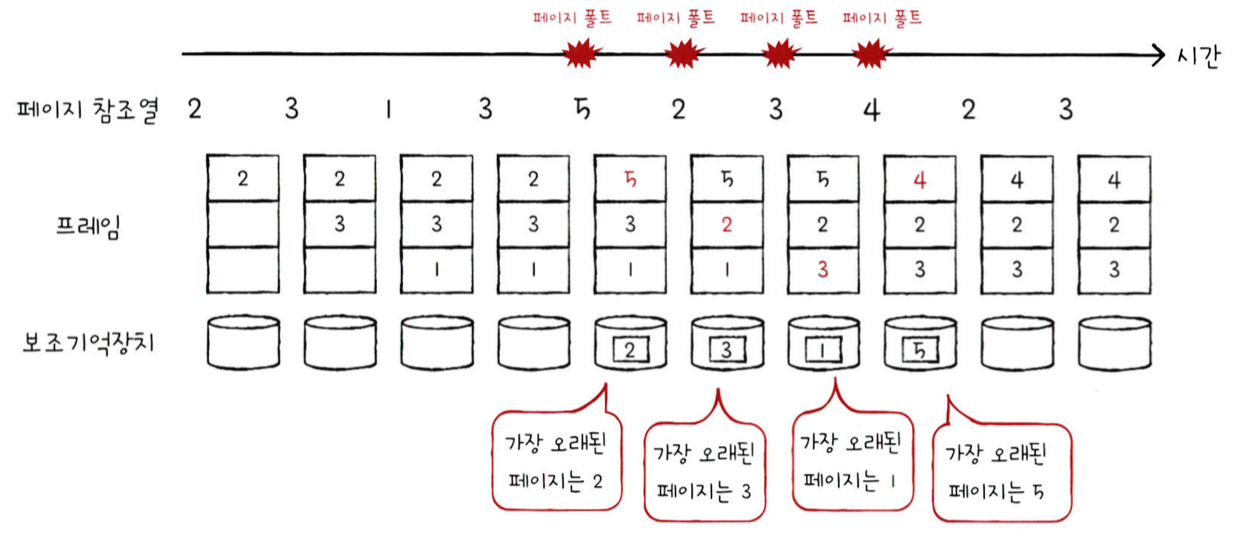
* 연속된 페이지를 생략한 페이지열, 페이지 참조열은 아래와 같다



* 연속된 페이지를 생략하는 이유? 중복된 페이지를 참조하는 행위는 페이지 폴트를 발생시키지 않기 때문

**FIFO 페이지 교체 알고리즘**

* 메모리에 가장 먼저 올라온 페이지부터 내쫓는 방식
* 아이디어와 구현이 간단하지만 실행 초기에 적재된 페이지 속에는 프로그램 실행 내내 사용될 내용을 포함하고 있을 수도 있음 -> 이런 페이지는 내쫓아서는 안됨

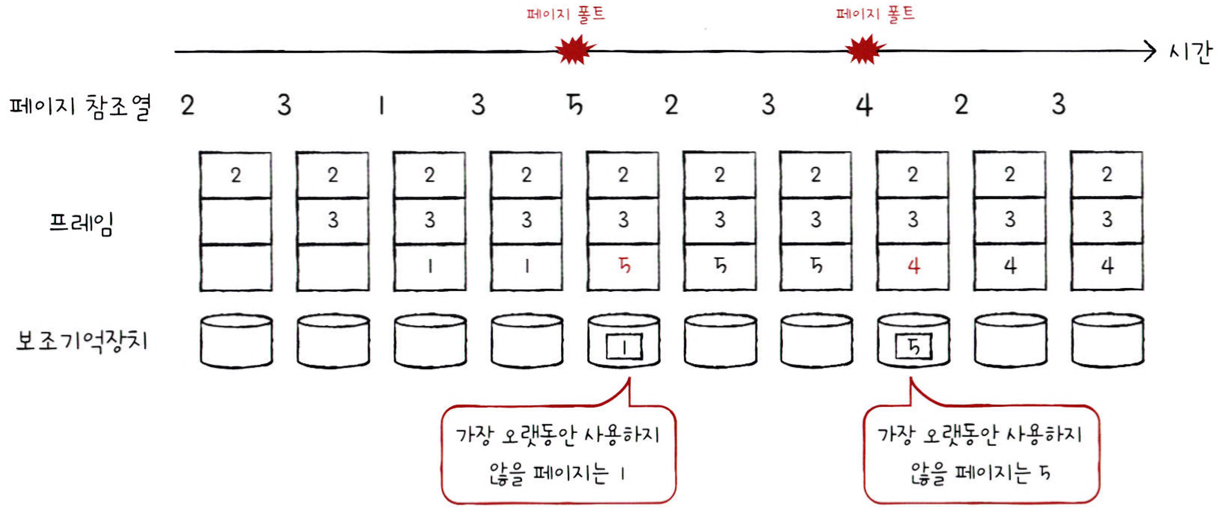


**2차 기회 페이지 교체 알고리즘**

* FIFO 페이지 교체 알고리즘의 부작용을 개선한 알고리즘
* 만일 페이지의 참조 비트가 1일 경우 당장 내쫓지 않고 참조 비트를 0으로 만든 뒤 현재 시간을 적재 시간으로 설정
* 참조 비트가 0일 경우 이 페이지는 가장 오래된 페이지이면서 동시에 사용되지 않는 페이지라고 볼 수 있으므로 보조기억장치로 내보냄

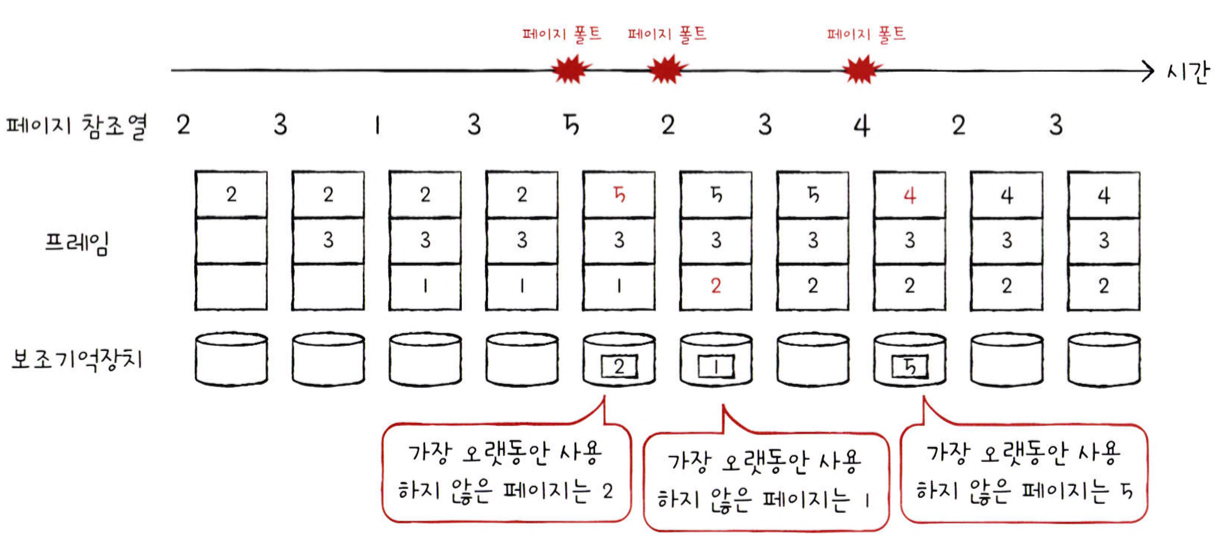
**최적 페이지 교체 알고리즘**

* CPU에 의해 참조되는 횟수를 고려하는 페이지 교체 알고리즘
* 앞으로의 사용 빈도가 가장 낮은 페이지를 교체하는 알고리즘
* 가장 낮은 페이지 폴트율을 보장하는 알고리즘
* 실제 구현이 어려움 -> 앞으로 오랫동안 사용되지 않을 페이지를 예측하기 어려움
* 주로 다른 페이지 교체 알고리즘의 이론상 성능을 평가하기 위한 목적으로 사용



**LRU 페이지 교체 알고리즘**

* 가장 오랫동안 사용되지 않은 페이지를 교체하는 알고리즘
* 최근에 사용되지 않은 페이지는 앞으로도 사용되지 않을 것이라는 아이디어를 토대로 만들어진 알고리즘



**<스래싱과 프레임 할당>**

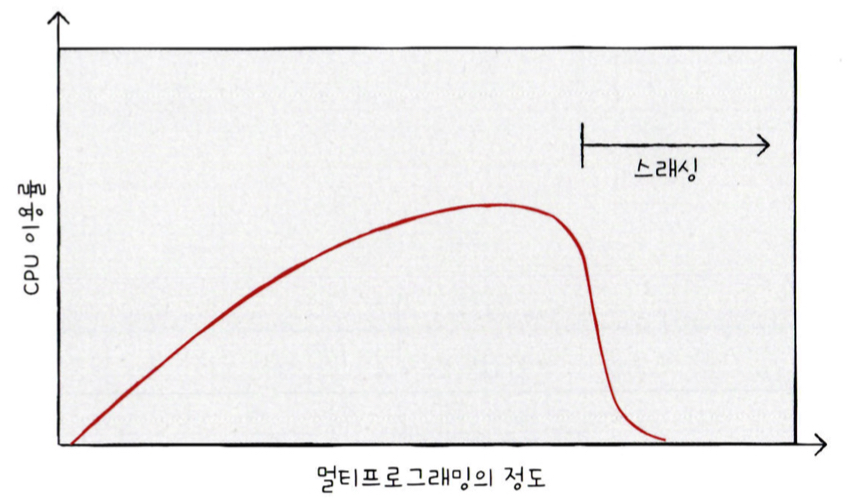
프로세스가 사용할 수 있는 프레임 수가 적어도 페이지 폴트는 자주 발생

프레임이 부족하면 CPU가 쉴새 없이 프로세스를 실행해야 컴퓨터 전체의 생산성이 올라감 -> 페이지 교체를 끊임없이 하게 되면서 페이지 교체에 너무 많은 시간을 쏟으면 성능에도 악영향

**스래싱**: 프로세스가 실제 실행되는 시간보다 페이징에 더 많은 시간을 소요하여 성능이 저해되는 문제

**멀티프로그래밍의 정도**: 메모리에서 동시 실행되는 프로세스의 수

* 멀티프로그래밍의 정도가 **높다면** 현재 메모리에는 **많은 프로세스**가 동시에 실행 중
* 멀티프로그래밍의 정도가 **낮다면** 현재 메모리에는 **적은 프로세스**가 동시에 실행 중



* 멀티프로그래밍의 정도를 늘린다고 해서 CPU 이용률이 그에 비례해서 증가하는 것은 아님
* 필요 이상으로 늘리면 각 프로세스들이 사용할 수 있는 프레임 수가 적어지기 때문에 페이지 폴트가 지나치게 빈번히 발생하고 CPU 이용률이 떨어져 전체적인 성능이 저해
* 스레싱이 발생하는 근본적인 원인은 각 프로세스가 필요로 하는 최소한의 프레임 수가 보장되지 않았기 때문

-> 운영체제는 각 프로세스들이 무리 없이 실행하기 위한 최소한의 프레임 수를 파악하고 프로세스들에 적절한 수만큼 프레임을 할당해 줄 수 있어야 함

**프레임 할당 방식**

1. 균등 할당

* 모든 프로세스에 균등하게 프레임을 제공하는 방식
* 프로세스들의 크기는 각기 다른데 천편일률적으로 동일한 프레임 개수를 할당하는 것은 비합리적

1. 비례 할당

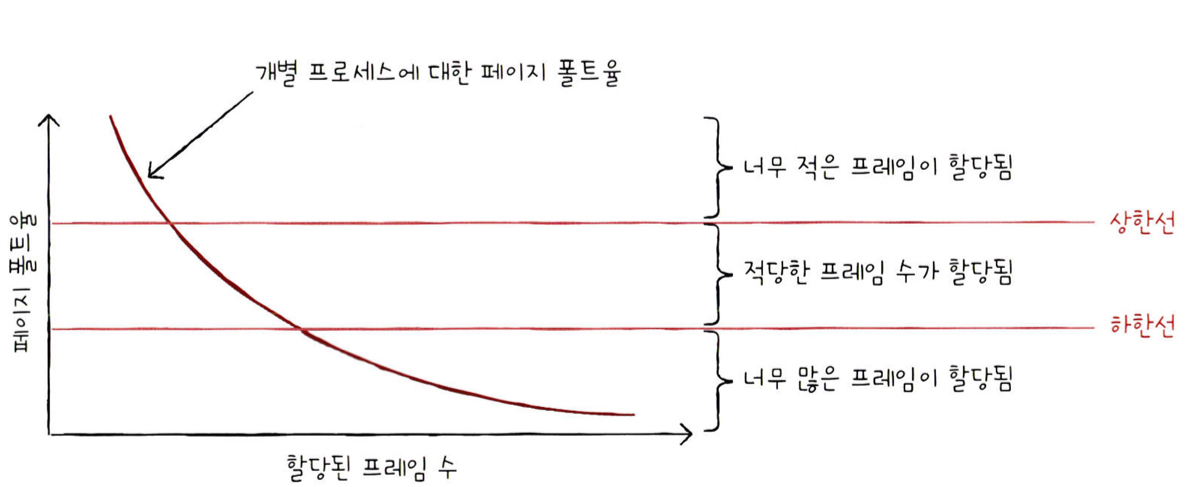
* 프로세스의 크기가 크면 프레임을 많이 할당하고 프로세스 크기가 작으면 프레임을 적게 할당하는 방식
* 하지만 프로세스의 크기가 크더라도 막상 실행해 보면 많은 프레임을 필요로 하지 않는 경우와 프로세스의 크기가 작아도 실행해 보니 많은 프레임을 필요로 하는 경우가 존재

1. 작업 집합 모델 기반 할당

* 프로세스가 일정 기간 동안 참조한 페이지 집합을 기억하여 빈번한 페이지 교체를 방지
* CPU가 특정 시간 동안 주로 참조한 페이지 개수만큼만 프레임을 할당하면 페이지 교체는 빈번하게 발생하지 않음
* **작업 집합**: 실행 중인 프로세스가 일정 시간 동안 참조한 페이지의 집합

1. 페이지 폴트 빈도 기반 할당

* 페이지 폴트율이 너무 높으면 그 프로세스는 너무 적은 프레임을 갖고 있다
* 페이지 폴트율이 너무 낮으면 그 프로세스는 너무 많은 프레임을 갖고 있다
* 페이지 폴트율에 상한선과 하한선을 정하고, 이 범위 안에서만 프레임을 할당하는 방식



* 페이지 폴트율이 상한선보다 높을 때 -> 프레임을 더 할당
* 페이지 폴트율이 하한선보다 더 낮을 때 -> 다른 프로세스에 할당하기 위해 프레임을 회수