Ch09 Main Memory

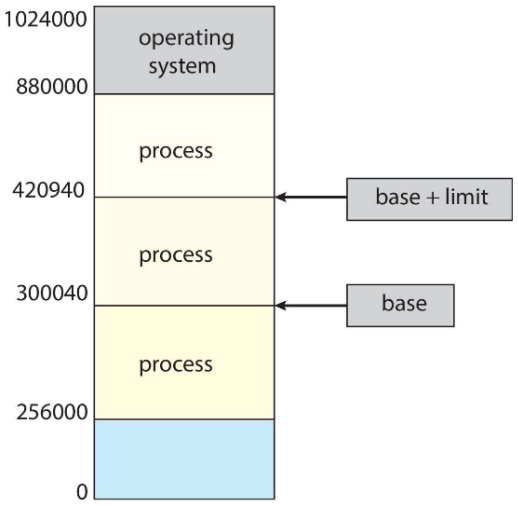
Objectives

- 논리 주소와 물리 주소의 차이점과 주소를 변환할 때 MMU(메모리 관리 장치)의 역할을 설명한다.
- 메모리를 연속적으로 할당하기 위해 최초, 최적 및 최악 접합 전략을 적용한다.
- 내부 및 외부 단편화의 차이점을 설명한다.
- TLB(Translaction look-aside buffer)가 포함된 페이징 시스템에서 논리 주소를 물리 주소로 변환한다.
- 계층적 페이징, 해시 페이징 및 역 페이지 테이블을 설명한다.

Background

- 프로그램을 실행하기 위해서는 (디스크에서) 메모리로 가져와 프로세스 내에 배치해야 한다.
- 메인 메모리와 레지스터들은 CPU가 직접적으로 접근할 수 있는 유일한 저장장치이다.
  - 레지스터 access는 1 CPU 클락만큼, 혹은 그 안에 모두 끝난다.
  - 메인 메모리의 접근을 완료하기 위해서는 많은 CPU 클록 틱 사이클이 소요되며, 이 경우 CPU가 필요한 데이터가 없어서 명령어를 수행하지 못하고 지연되는(stall) 현상이 발생하게 된다.
- 메모리 유닛은 오직 다음의 stream들만 볼 수 있다.
  - addresses + read requests 혹은
  - addresses + data and write requests
- 캐시는 메인 메모리와 CPU 레지스터 사이에 있다.
- 올바른 작동을 보장하기 위해 메모리 보호가 필요하다.

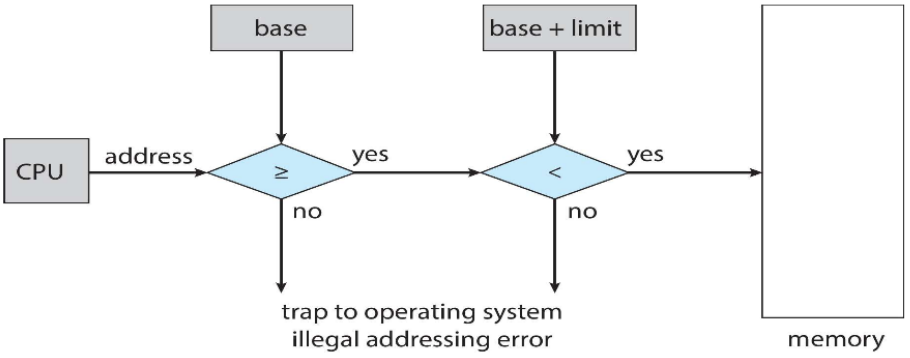
Protection



- 프로세스가 해당 주소 공간에 있는 해당 주소에만 액세스할 수 있는지 검열해야 한다.
- 한 쌍의 기준(base) 레지스터와 상한(limit) 레지스터를 사용하여 프로세스의 논리적 주소 공간을 정의함으로써 이 보호를 제공할 수 있다.

Hardware Address Protection

- CPU는 사용자 모드에서 생성된 모든 메모리 액세스가 해당 사용자에 대한 기준 레지스터와 상한 레지스터 사이인지 확인해야 한다.



- 기준 레지스터와 상한 레지스터를 로딩하는 명령어는 특권 명령어이다.

데이터와 명령어의 Binding

> 💬 Binding이란, 컴퓨터 프로그래밍에서 각종 값들이 확정되어 더 이상 변경할 수 없는 구속(bind) 상태가 되는 것.프로그램 내에서 변수, 배열, 라벨, 절차 등의 명칭, 즉 식별자(identifier)가 그 대상인 메모리 주소, 데이터형 또는 실제 값으로 배정되는 것이 이에 해당되며, 원시 프로그램의 컴파일링 또는 링크 시에 확정되는 바인딩을 정적 바인딩(static binding)이라 하고, 프로그램의 실행되는 과정에서 바인딩 되는 것을 동적 바인딩(dynamic binding)이라고 한다.

- 만약 프로그래머가 코딩하는 동안 정확한 메모리 위치를 모두 알고 있다면 바인딩은 필요하지 않다.

Address Binding Schemes

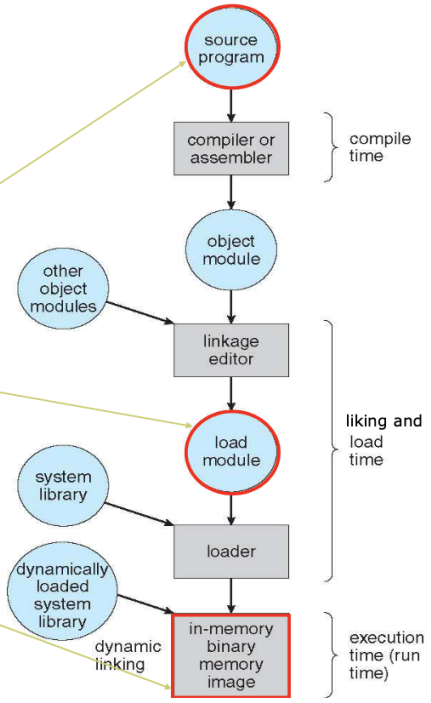
- 전통적으로 메모리 주소 공간에서 명령어와 데이터의 바인딩은 그 바인딩이 이루어지는 시점에 따라 다음과 같이 구분된다.
    - **컴파일 시간 바인딩** : 만일 프로세스가 메모리 내에 들어갈 위치를 컴파일 시간에 미리 알 수 있으면 컴파일러는 절대 코드를 생성할 수 있다. 그러나 만일 이 위치가 변경되어야 한다면 이 코드는 다시 컴파일되어야 한다.
    - **적재 시간 바인딩** : 만일 프로세스가 메모리 내 어디로 올라오게 될지를 컴파일 시점에 알지 못하면 컴파일러는 일단 이진 코드를 재배치 가능 코드로 만들어야 한다. 이 경우에 심볼과 진짜 번지수와의 바인딩은 프로그램이 메인 메모리로 실제로 적재되는 시간에 이루어지게 된다. 이렇게 만들어진 재배치 가능 코드는 시작 주소가 변경되면 아무 때나 사용자 코드를 다시 적재하기만 하면 된다.
    - **실행 시간 바인딩 :** 만약 프로세스가 실행하는 중간에 메모리 내의 한 세그먼트로부터 다른 세그먼트로 옮겨질 수 있다면 우리는 바인딩 실행 시간까지 허용되었다고 이야기 한다.

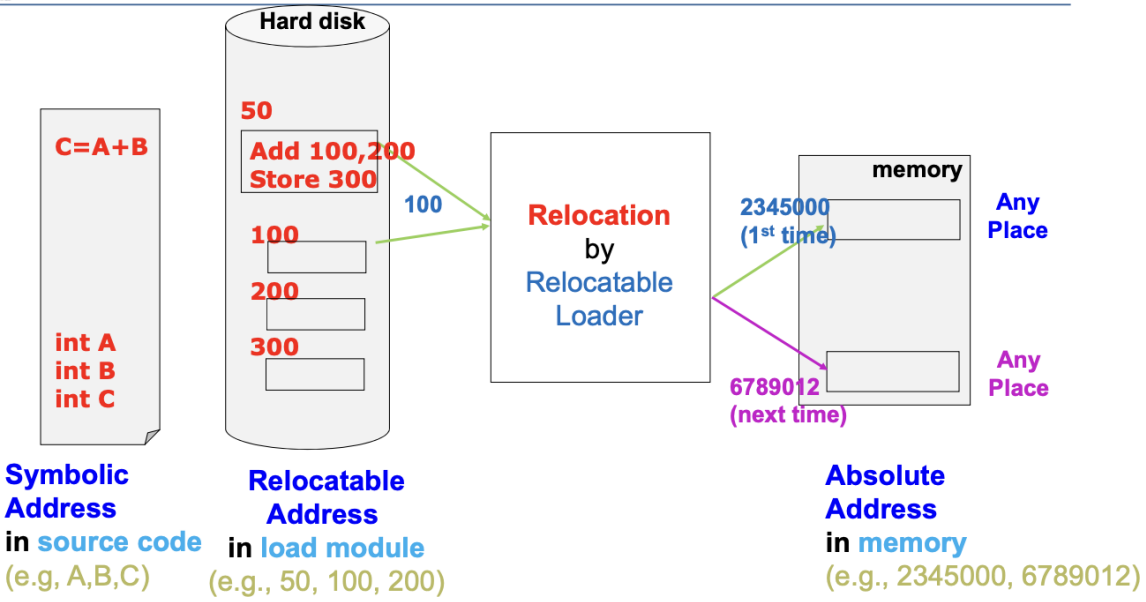
Address Binding

- 프로세스가 실행되면 메모리에서 명령 및 데이터에 액세스 한다. 결국 프로세스가 종료되고 다른 프로세스에서 사용하기 위해 메모리가 회수된다. 대부분의 시스템은 사용자 프로세스가 메모리 내 어느 부분으로도 올라올 수 있도록 지원하고 있다. 사용자 프로세스의 주소가 00000번지부터 시작된다고 해서 이 프로그램이 메모리의 00000부터 올라와야 할 필요는 없다.
- 원시 프로그램에서 주소는 숫자가 아닌 (변수 count와 같이) 심볼 형태로 표현된다. 컴파일러는 이 심볼 주소를 재배치 가능 주소(예를 들면 “이 모듈의 첫번째 바이트로부터 열네 번째 바이트 주소”)로 바인딩시키고, 다음에 링커(linker)나 로더(loader)가 재배치 가능 주소를 절대 주소(예를 들면 74014번지)로 바인딩시킨다. 각각의 바인딩 과정은 한 주소 공간에서 다른 주소 공간으로 맵핑하는 것이다.

Multistep Processing of a User Program

- User programs go through several steps before being run on memory
  - Compile, Linking, Loading
- Address may be represented in different ways during these steps
  - Symbolic address - variables in source code
  - Relocatable address - address from beginning of the load module
  - Absolute address - physical address in memory





Logical vs. Physical 주소 공간

- 별도의 물리적 주소 공간에 바인딩된 논리적 주소 공간의 개념은 적절한 메모리 관리의 핵심이다.
  - 논리 주소 : CPU에 의해 생성, 가상 주소라고도 불림

    - 물리 주소 : 메모리 장치에서 볼 수 있는 주소(즉, 메모리 주소 레지스터(MAR)에 주어지는 주소)
- 주소 공간과 주소 바인딩 스키마
    - 컴파일 또는 적재 시에 주소를 바인딩하면 논리 주소와 물리 주소가 같다.
    - 실행 시간 바인딩 기법에서는 논리 주소와 물리 주소가 다르다.

메모리 관리 요구사항

Relocation

- 프로그래머는 프로그램이 실행될 때 프로그램이 메모리의 어디에 위치할지 알 수 없다.
- 프로세스는 메모리 어느 지점에도 로드가 가능해야 한다.
- 따라서 재배치가능한 주소(load 모듈의 시작 주소)는 load 모듈에서 사용된다.
- 코드의 메모리 참조(심볼 주소, 이후 재배치 가능한 주소)는 실제 물리적 메모리 주소로 변환되어야 한다.

Protection

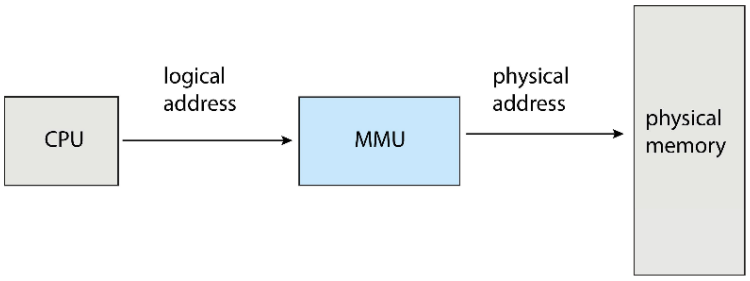
- 프로세스는 다른 프로세스의 메모리 위치를 허가 없이 참조할 수 없어야 한다.
- 컴파일 시 절대 주소를 확인할 수 없다.
- 실행 시간 중 체크되어야 한다.
- 메모리 보호 요구 사항은 OS(소프트웨어)가 아닌 프로세서(하드웨어)가 충족해야 한다.

Sharing

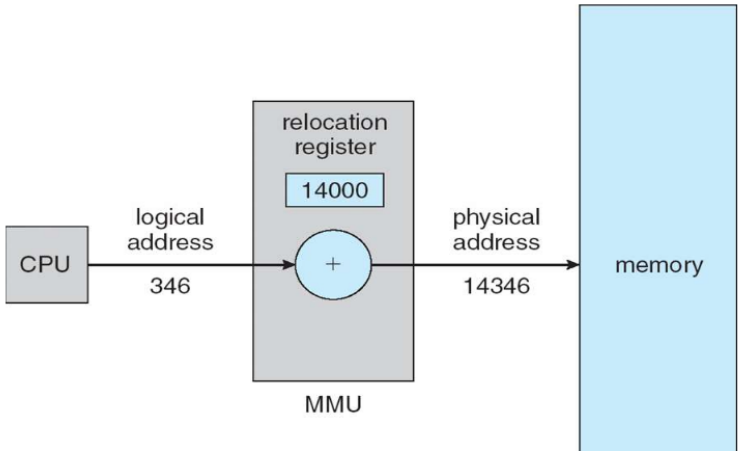
- 여러 프로세스가 동일한 메모리 부분에 액세스할 수 있도록 허용한다.
- 각 프로세스가 자신의 개별 복사본을 갖는 것보다 프로그램의 동일한 복사본에 액세스할 수 있도록 허용하는 것이 좋다.

Memory-Management Unit(MMU)

메모리 관리 장치는 하드웨어 장치이며, `프로그램 실행 중 가상 주소를 물리 주소로 바꾸어주는 변환(mapping) 작업을 수행` 한다.



- 사용자 프로그램은 논리 주소를 다루며, 물리 주소를 절대 볼 수 없다.
- 실행 시간 바인딩은 메모리 위치를 참조할 때 발생한다. 논리 주소가 물리 주소로 바인딩 된다.



재배치 레지스터를 이용한 동적 재배치

- 해당 그림은 기준 레지스터(base register) 기법을 일반화시킨 아주 단순한 MMU 기법에 따른 변환이다. 기준 레지스터를 재배치(relocation) 레지스터라고 부른다.
- 재배치 레지스터 속에 들어있는 값은 주소가 메모리로 보내질 때마다 그 모든 주소에 더해진다. 예를 들어, 재배치 레지스터 값이 14000이라면 프로세스가 346번지에 액세스할 때, 실은 메인 메모리의 14346번지에 액세스하게 된다.
- `사용자 프로그램은 절대 실제적인 물리 주소에 접근하지 않는다는 것`을 주의해야 한다.

Dynamic Loading

정적 적재(Static Loading)

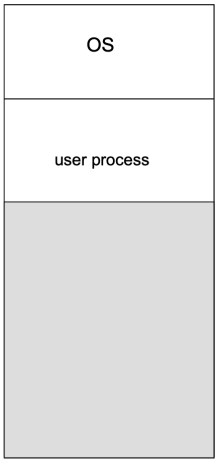
- 프로세스를 실행하기 위해 전체 프로그램 및 프로세스의 모든 데이터가 로드된다.
- 따라서 프로세스의 크기는 물리적 메모리의 크기로 제한된다.

동적 적재(Dynamic Loading)

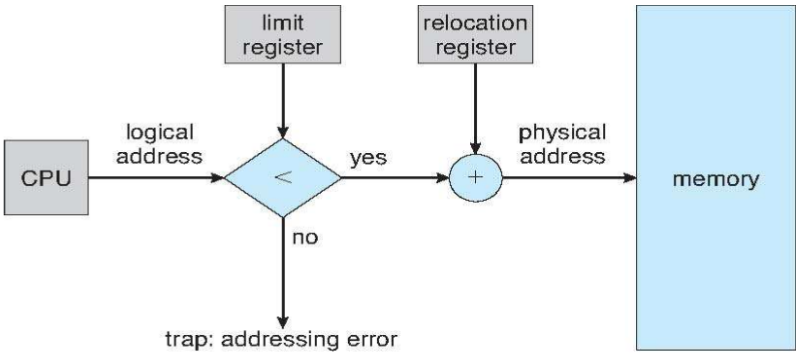
- 각 루틴은 실제 호출되기 전까지는 메모리에 올라오지 않고 재배치 가능한 상태로 디스크에서 대기하고 있다. 먼저 main 프로그램이 메모리에 올라와 실행된다. 이 루틴이 다른 루틴을 호출하게 되면 호출된 루틴이 이미 메모리에 적재되었는지를 조사한다. 만약 적재되어 있지 않다면, 재배치 가능 연결 적재기(relocatable linking loader)가 불려 요구된 루틴을 메모리로 가져오고, 이러한 변화를 테이블에 기록해 둔다. 그 후 CPU 제어는 중단되었던 루틴으로 보내진다.
- 동적 적재의 장점은 루틴이 필요한 경우에만 적재된다는 것이다. 이러한 구조는 오류 처리 루틴과 같이 아주 간혹 발생하면서도 실행할 코드가 많은 경우에 특히 유용하다. 이러한 상황에서 전체 프로그램 크기가 클 수 있지만 사용되는 부분이 훨씬 작을 수 있다.
- 동적 적재는 운영체제로부터 특별한 지원이 필요없다. 사용자 자신이 프로그램의 설계를 책임져야 한다. 운영체제는 동적 적재를 구현하는 라이브러리 루틴을 제공해 줄 수는 있다.

Contiguous Allocation(연속 할당)

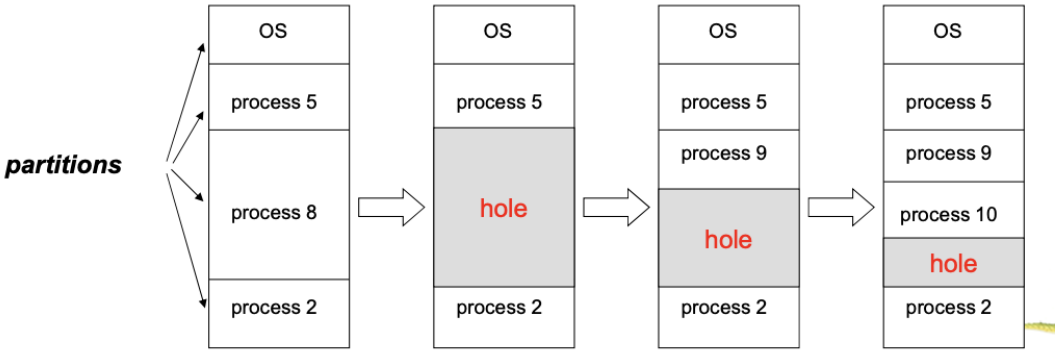
- 메인 메모리는 OS와 사용자 프로세스를 모두 지원해야 한다. 그리고 각 영역은 각각 목적에 맞도록 효율적으로 관리되어야 한다.
- 연속할당은 고전 방법이다.



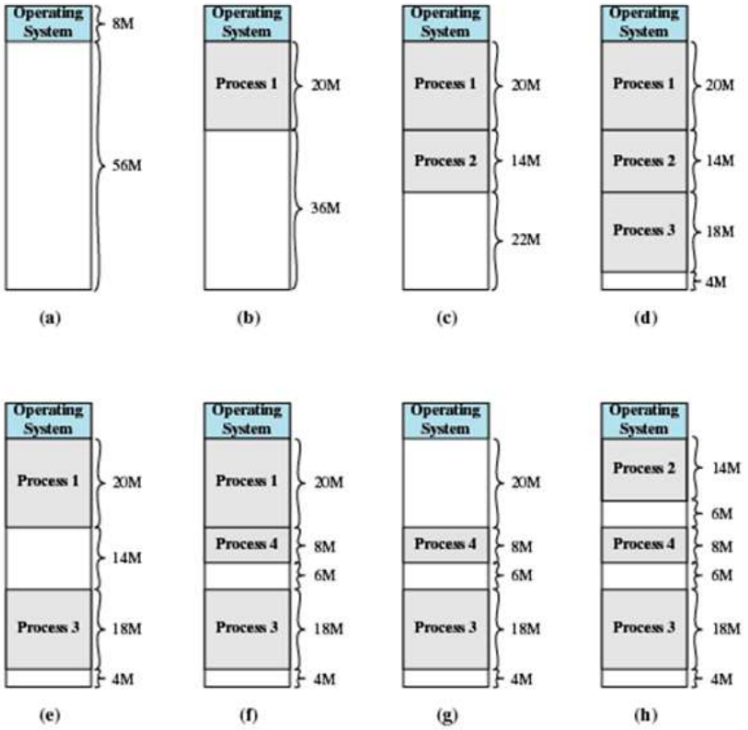
- 메인 메모리는 보통 2 부분으로 나뉜다. 하나는 운영체제를 위한 것이며 다른 하나는 사용자 프로세스를 위한 것이다.
    - 운영체제를 낮은 메모리 주소 또는 높은 메모리 주소에 배치할 수 있다.
    - 일반적으로 여러 사용자 프로세스가 동시에 메모리에 상주하기를 원한다. 따라서 메모리에 적재되기를 기다리는 프로세스에 사용 가능한 메모리를 할당하는 방법을 고려해야 한다. 연속적인 메모리 할당에서 각 프로세스는 다음 프로세스가 적재된 영역과 인접한 하나의 메모리 영역에 적재된다.



- 할당된 레지스터는 사용자 프로세스로부터 OS 코드와 데이터가 변경되지 않도록 보호하는 데 사용된다.
  - 기준(base) 레지스터는 물리 주소의 가장 작은 값을 가진다.
  - 한계(limit) 레지스터는 논리 주소의 범위를 포함한다 - 각각의 논리 주소는 한계 레지스터보다 작아야 한다.
  - MMU는 논리 주소를 동적으로 매핑한다.
- Multiple Partition Allocation
  - 메모리를 할당하는 가장 간단한 방법 중 하나는 프로세스를 멤리의 가변 크기 파티션에 할당하는 것이다. 각 파티션에는 정확히 하나의 프로세스만 적재될 수 있다. 이 가변 파티션 기법에서 운영체제는 사용 가능한 메모리 부분과 사용 중인 부분을 나타내는 테이블을 유지한다. 처음에는 모든 메모리가 사용자 프로세스에 사용 가능하며, 하나의 큰 사용 가능한 메모리 블록인 hole로 간주한다.



  - 처음에는 프로세스 5,8 및 2가 적재되어 있고 메모리가 완전히 활용 중이다. 프로세스 8이 종료된 후 하나의 연속된 hole이 생긴다. 나중에 프로세스 9가 도착하고 메모리가 할당된다. 그런 프로세스 5가 종료되면 두 개의 연속되지 않은 hole이 생긴다.



동적 메모리 할당 문제(Dynamic Storage-Allocation Problem)

- 일련의 가용 공간-리스트로부터 크기 n-바이트 블록을 요구하는 것을 어떻게 만족시켜 줄 것이냐를 결정하는 문제

✔ 알고리즘

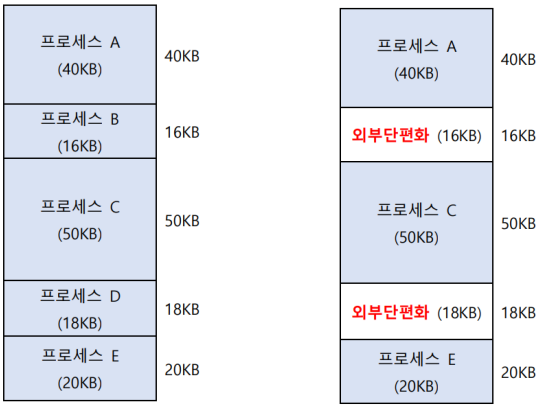
- 최초 적합(First-Fit)
  - 첫 번째 사용 가능한 가용 공간을 할당한다. 검색은 집합의 시작에서부터 하거나 지난번 검색이 끝났던 곳에서 시작될 수 있다. 충분히 큰 가용 공간을 찾았을 때 검색을 끝낼 수 있다.
- 최적 적합(Best-Fit)
  - 사용 가능한 공간 중에서 가장 작은 것을 택한다. 리스트가 크기 순으로 되어 있지 않다면 전 리스트를 검색해야만 한다. 이 방법은 아주 작은 가용 공간을 만들어 낸다.
- 최악 적합(Worst-Fit)
  - 가장 큰 가용 공간을 택한다. 이 방식에서 할당해 주고 남게 되는 가용 공간은 충분히 커서 다른 프로세스들을 위하여 유용하게 사용될 수 있다. 이때 가용 공간들이 크기 순으로 정렬되어 있지 않으면 전 리스트를 다 검색해야만 한다.

단편화(Fragmentation)

내부 단편화

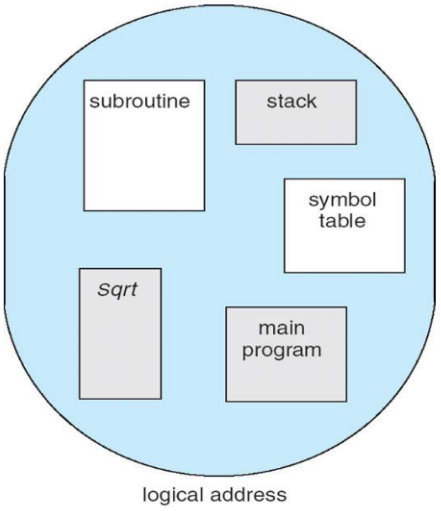
- 일반적으로 메모리를 먼저 아주 작은 공간들로 분할하고 프로세스가 요청하면 할당을 항상 이 분할된 크기의 정수배로만 해준다.
- 18,464 B 크기의 가용 공간을 생각해 보고, 어느 한 프로세스가 18,462 B를 요구한다고 가정한다. 요구한 블록을 정확히 할당하면 2B 크기의 가용 공간이 남는다. 이 공간이 내부 단편화이다.

외부 단편화



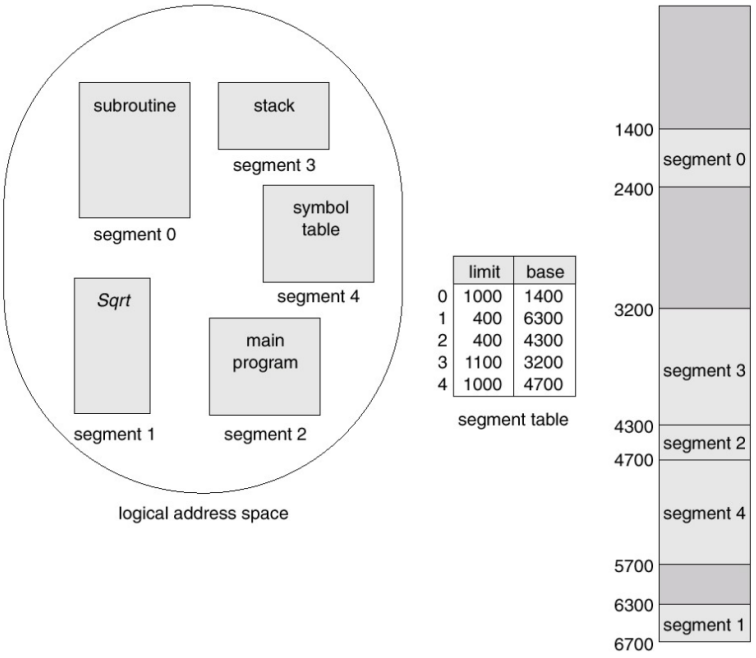
- 유휴 공간들을 모두 합치면 충분한 공간이 되지만 그것들이 너무 작은 조각들로 여러 곳에 분산되어 있을 때 발생한다. 즉, 메모리가 너무 많은 수의 매우 작은 조각들로 단편화되어 있는 것이다.
- 외부 단편화를 해결하는 방법으로는 압축(compaction)이 있다. 이 방법은 메모리 모든 내용을 한군데로 몰고 모든 가용 공간을 다른 한군데로 몰아서 큰 블록을 만드는 것이다. 그러나 압축이 항상 가능한 것은 아니다. 재배치가 어셈블 또는 적재 시간에 정적으로 행해진다면, 압축은 실행될 수 없다.

Segmentation



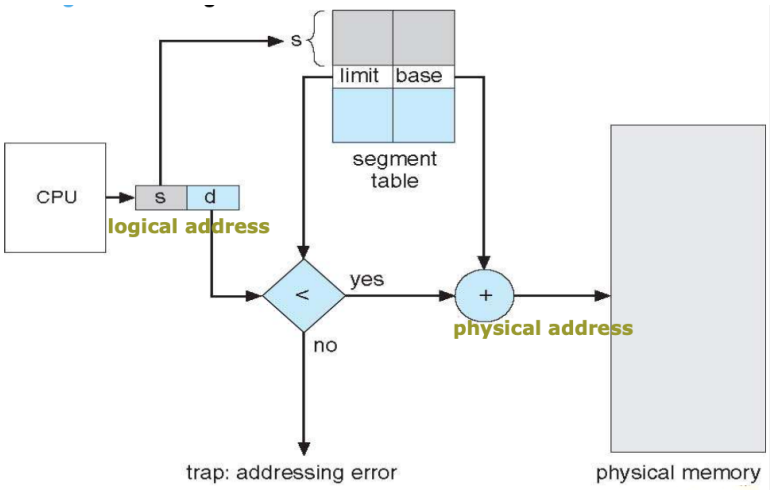
💬 프로세스를 논리적 내용을 기반으로 나눠서 메모리에 배치하는 것

- 프로그램은 세그먼트의 모음이다.
- 세그먼트는 (main 프로그램, 함수, 객체, 변수, 스택 ...)의 논리적 유닛이다.



- 세그먼테이션은 세그먼트 테이블을 가지고 있다. 논리주소는 <segment-number, offset>로 이루어져 있다. 다만 세그먼트의 크기는 일정하지 않기 때문에, 테이블에 limit 정보가 추가로 들어있다. 만약 세그먼트의 크기를 초과하는 주소가 들어오면 인터럽트가 발생해 해당 프로세스는 강제로 종료된다.
- base : 세그먼트가 메모리에 있는 시작 물리적 주소를 포함

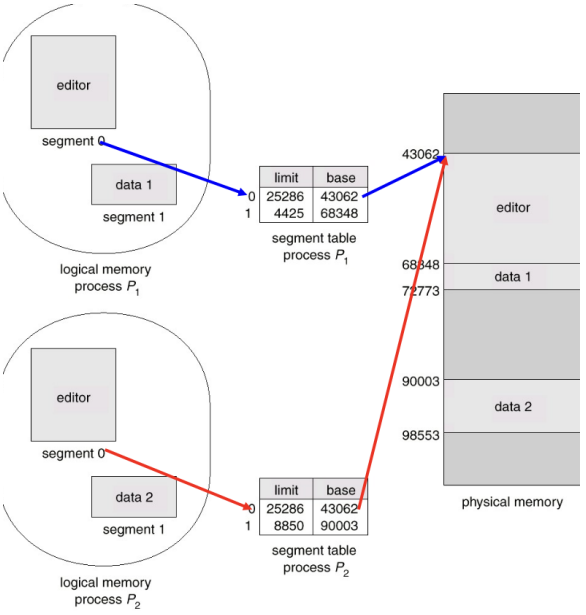
- limit : 세그먼트의 길이를 정의



- Segment-table base register(STBR)은 메모리의 세그먼트 테이블 위치를 가리킨다.
- Segment-table length register(STLR)은 프로그램에 의해 사용되는 세게ㅡ먼트 번호를 가리킨다.

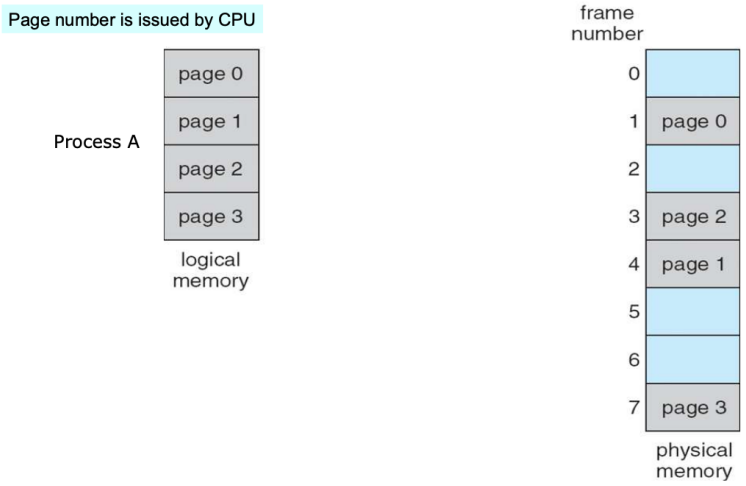
$$\text{segment number } s \text{ is legal if } s < \text{STLR}$$

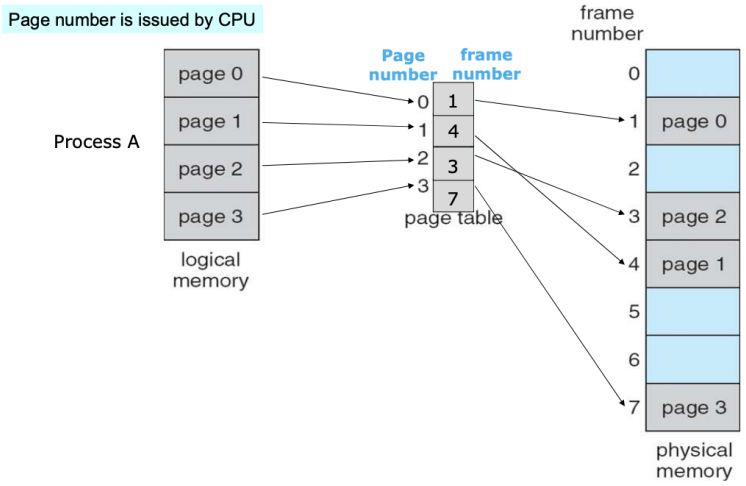
- 보호
  - 세그먼트는 validate 하지 않은 세그먼트를 `validation bit = 0` 으로 처리한다.
  - 세그먼트는 read/write/execute 실행 권한을 가진다.
- 공유 - 세그먼트 레벨에서 코드 공유가 발생한다.

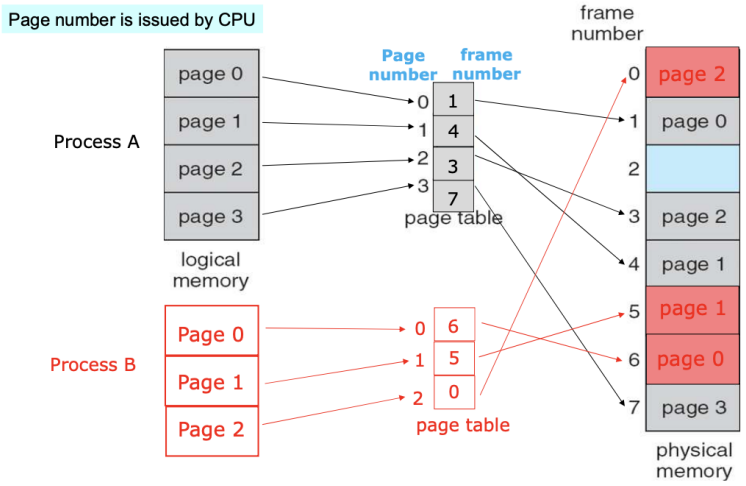


Paging

- 프로세스의 물리 주소 공간이 연속되지 않아도 되는 메모리 관리 기법
- 연속메모리 할당을 괴롭히는 두 가지 문제인 외부 단편화와 관련 압축의 필요성을 피한다.
- 물리 메모리는 프레임(frame)이라 불리는 같은 크기 블록으로 나누어진다. 논리 메모리는 페이지(page)라 불리는 같은 크기의 블록으로 나누어진다. 프로세스가 실행될 때 그 프로세스의 페이지는 파일 시스템 또는 예비 저장장치로부터 가용한 메인 메모리 프레임으로 적재된다. 예비 저장장치는 메모리 프레임 혹은 프레임의 묶음인 클러스터와 동일한 크기의 고정 크기 블록으로 나누어진다.
- 페이지의 크기는 2의 거듭제곱으로, 512 bytes ~ 16 Mbytes
- N page를 실행시키기 위해 N 프레임이 필요하다.
- 논리 주소를 물리 주소로 변경하기 위해 페이지 테이블을 세팅해야 한다.
- 여전히 내부 단편화는 존재한다.



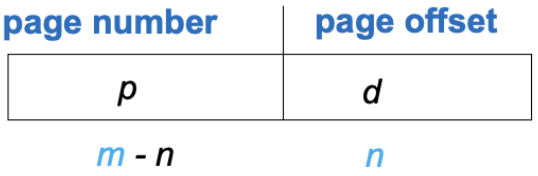




주소 변환 스키마

- CPU로부터 생성되는 주소는 다음처럼 나뉜다.

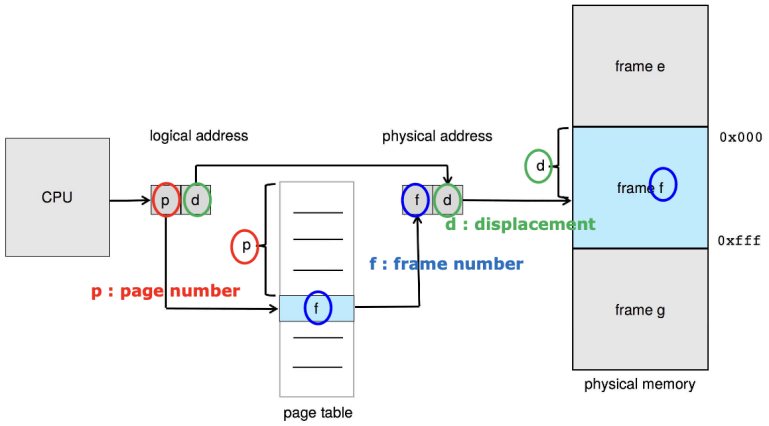
- page number(p) : 물리적 메모리의 각 페이지의 기본 주소를 포함하는 페이지 테이블의 인덱스로 사용
- page offset(d) : 기본 주소와 결합하여 메모리 장치로 전송되는 물리적 메모리 주소를 정의



logical address space 2^m and page size 2^n

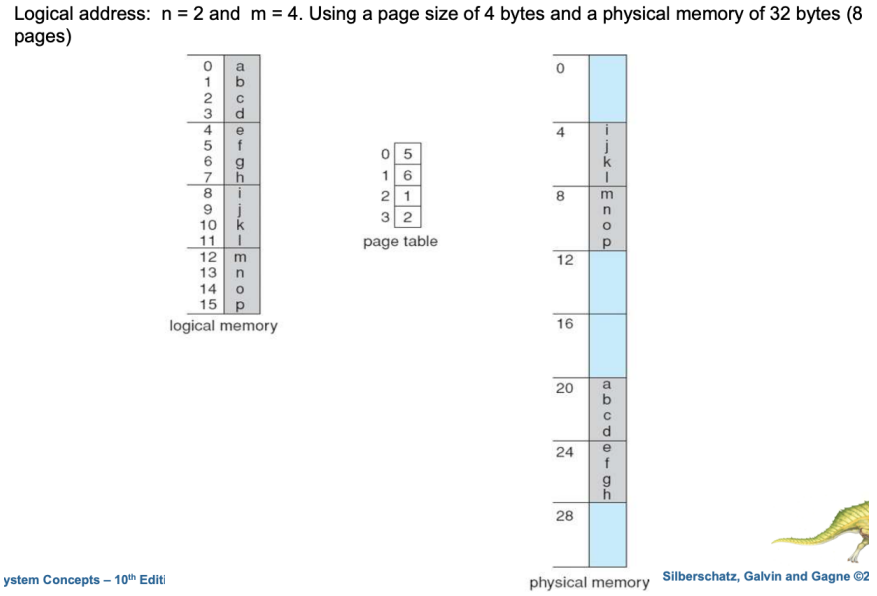
- 논리 주소 공간의 크기가 2의 m제곱이고 페이지 크기가 2의 n제곱 바이트인 경우 논리 주소의 상위 m-n 비트는 페이지 번호를 지정하고 n 하위 비트는 페이지 오프셋을 지정한다.

Paging Hardware



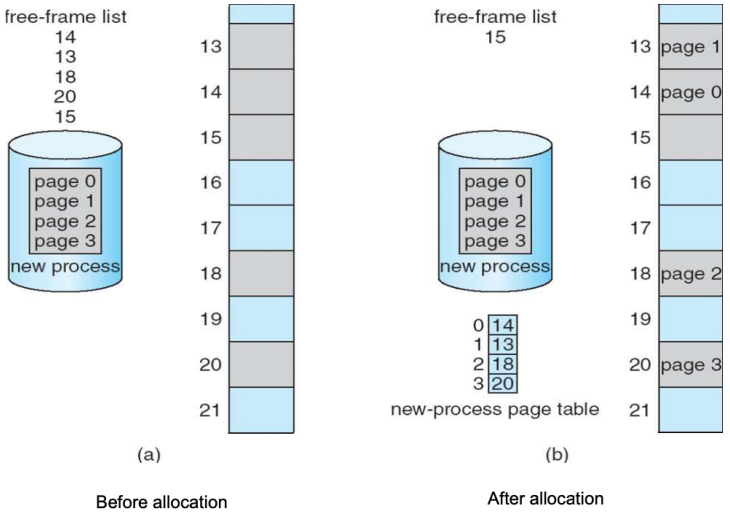
1. 페이지 번호 p를 추출하여 페이지 테이블의 인덱스로 사용한다.
2. 페이지 테이블에서 해당 프레임 번호 f를 추출한다.
3. 논리 주소의 페이지 번호 p를 프레임 번호 f로 바꾼다.

Paging Example



- 논리 주소에서 n=2, m=4이다. 4B의 페이지 크기와 32B의 물리 메모리(8페이지)를 사용하여 프로그래머가 보는 메모리가 물리 메모리로 사상되는 예를 보자. 논리 주소 0은 페이지 0, 오프셋 0이다. 페이지 테이블을 색인으로 찾아서 페이지 0이 프레임 5에 있다는 것을 알아낸다. 그래서 논리 주소 0은 실제 주소 20[=(5X4)+0]으로 사상된다. 논리 주소 3(페이지 0, 오프셋 3)은 실제 주소 23[=(5X4)+3]으로 사상된다. 논리 주소 4는 페이지 테이블에 의해 페이지 1, 오프셋 0이고, 페이지 1은 프레임 6으로 사상한다. 그래서 실제 주소 24[=(6X4)+0)으로 사상한다. 논리 주소 13은 실제 주소 9로 사상된다.

Free Frames



Page Table의 구현

- 페이지 테이블은 메인 메모리에 저장된다.

- Page-table base register(PTBR)은 페이지 테이블을 가리킨다.
- Page-table length register(PTLR)은 페이지 테이블의 크기를 가리킨다.
- 이 스키마에서 모든 데이터와 명령어는 두 가지의 메모리 액세스가 필요하다. (페이지 테이블에 대한 액세스, 데이터와 명령어에 대한 액세스)
- 2개의 메모리 액세스 문제는 연관 메모리 또는 번역 look-aside buffers(TLBs)(연관 메모리라고도 함)라고 하는 특수한 fast-lookup hardware 캐시를 사용함으로써 해결될 수 있다.
- TLB hit인 경우 페이지 프레임 번호에 빠르게 접근이 가능하다.
- TLB miss인 경우 TLB에 내용이 load된다. 교체 policies는 반드시 고려되어야 한다.

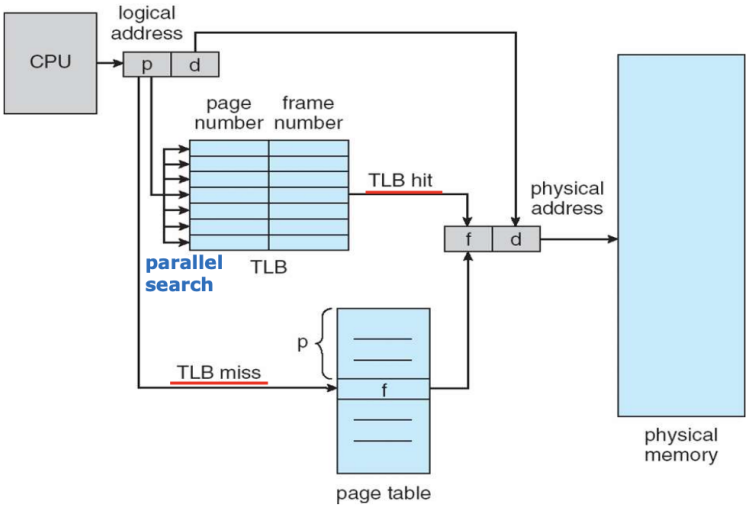
연관 메모리 (Associative Memory)

Parallel search

page number	frame number

주소 변환(p,d)

- 만약 p가 연관 레지스터 안에 있다면 TLB hit! 프레임 번호를 얻는다.
- 그렇지 않으면, TLB miss, 메모리의 페이지 테이블에서 프레임 번호를 얻는다.



실질 메모리 접근 시간(Effective Memory Access Time)

- HIT ratio : TLB에서 찾을 수 있는 페이지 번호의 시간적 퍼센트
- 적중률 80%는 TLB에서 원하는 페이지 번호를 80%로 찾는 것을 의미한다.
- 메모리 접근에 10 나노초가 걸린다고 가정하자.
    - 만약 우리가 TLB에서 요구된 페이지를 찾을 수 있다면 mapped-memory access는 10 나노초가 걸릴 것이다.
    - 그렇지 않다면 우리는 20나노초가 걸릴 것이다. (두 개의 메모리 접근에)
- Effective Access Time(EAT)

    `EAT = 0.80 x 10 + 0.20 x 20 = 12 ns`

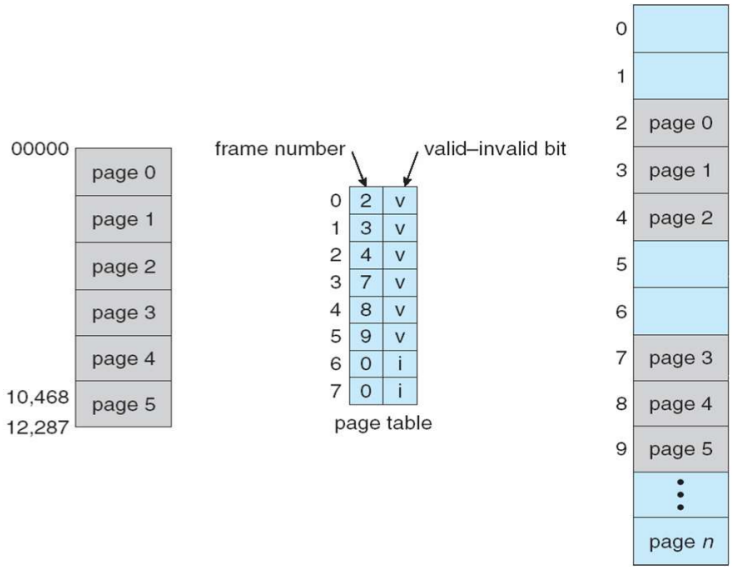
    - 접근 시간이 20% 단축된다.
- 보다 현실적인 명중률 99%를 고려해 보면

    `EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns`

    - 접근 시간이 1% 단축된다.

메모리 보호

- 각 프레임에 보호 비트를 연결하여 읽기 전용 액세스 또는 읽기-쓰기 액세스가 허용되는지 표시함으로써 구현된 메모리 보호
    - 페이지 실행 전용 등을 나타내기 위해 비트를 더 추가할 수도 있다.
- 페이지 테이블의 각 항목에 유효하지 않은 비트가 첨부:
    - "valid"는 관련 페이지가 프로세스의 논리 주소 공간에 있으므로 합법적인 페이지임을 나타냄
    - "invalid"는 페이지가 프로세스의 논리 주소 공간에 없음을 나타냄



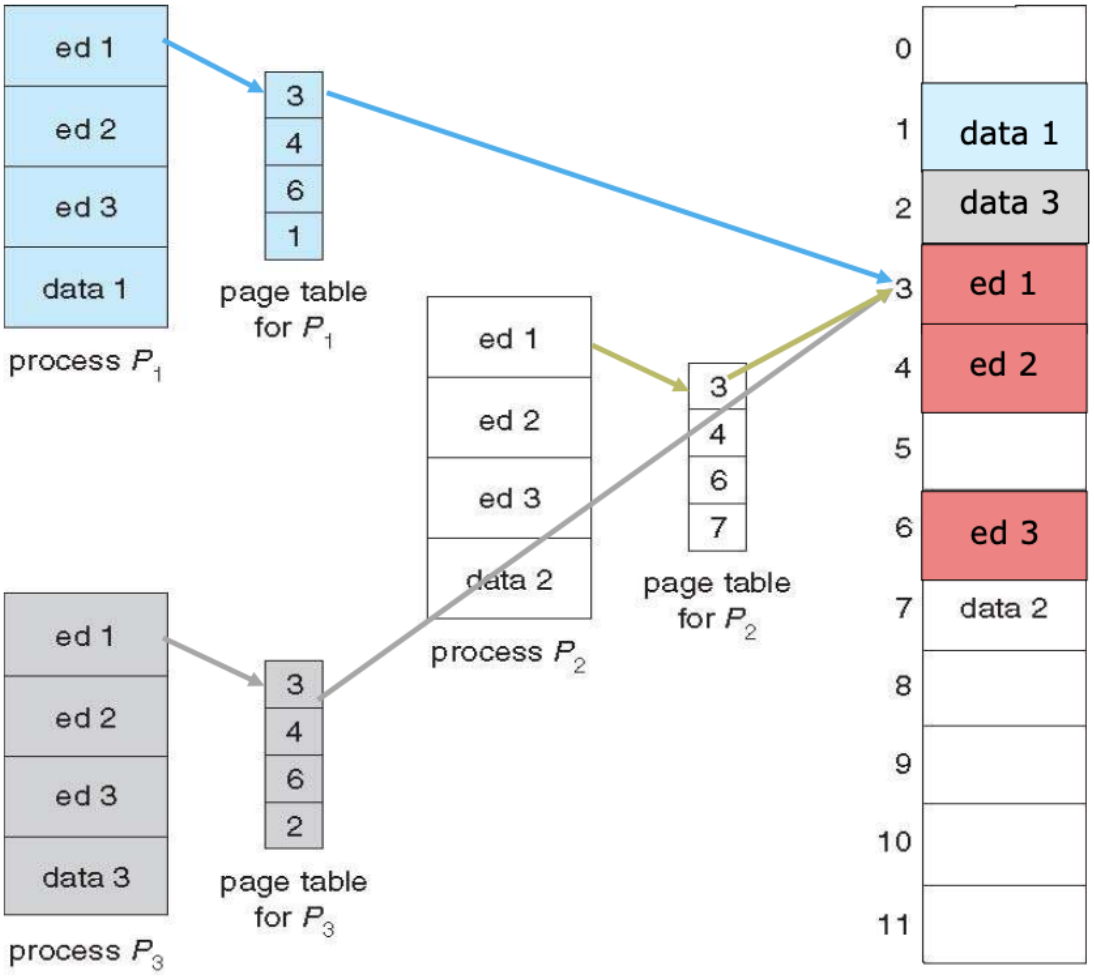
Shared Pages

공유 코드

- 재입력 코드는 자체 수정이 불가능한 코드이며 실행 중에는 변경되지 않음
- 프로세스 간에 공유되는 읽기 전용(재입력) 코드의 한 복사본(즉, 텍스트 편집기, 컴파일러, 윈도우 시스템)
- 여러 스레드가 동일한 프로세스 공간을 공유하는 것과 유사함
- 읽기-쓰기 페이지의 공유가 허용되는 경우 프로세스 간 통신에도 유용

Private Code와 Data

- 각 프로세스는 코드와 데이터의 개별 복사본을 보관
- 개인 코드 및 데이터의 페이지는 논리 주소 공간의 어느 곳에나 나타날 수 있다.


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1
2
3
4
5
6
7
8
9
10
11
data 1
data 3
ed 1
ed 2
ed 3
data 2