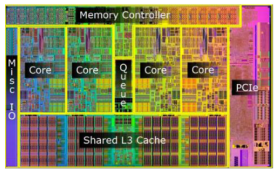
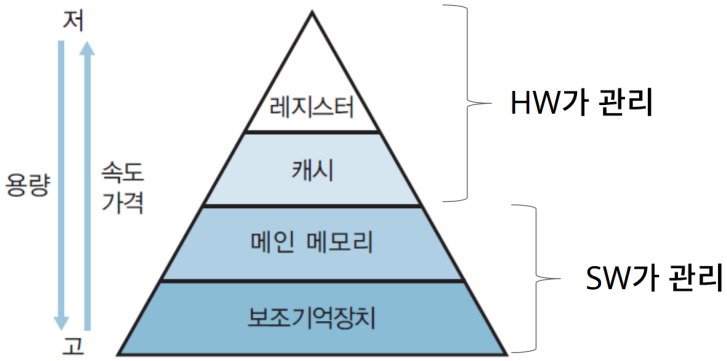
08. Memory Management

Background

메모리의(기억장치)의 종류



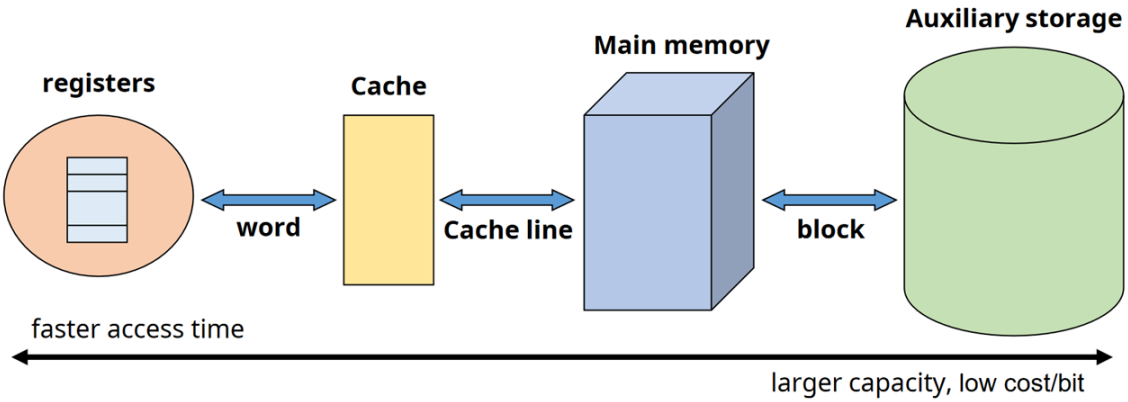


[그림출처] 운영체제, 한빛미디어

이 그림에서 HW는 CPU, SW는 OS를 뜻함

- **시스템에서 다루는 메모리 종류**
  1. 레지스터
  2. 캐시
  3. 메인 메모리
  4. 보조 기억 장치
- **메모리가 계층 구조를 갖게 된 원인**
  - I/O Bottleneck을 해소하기 위해

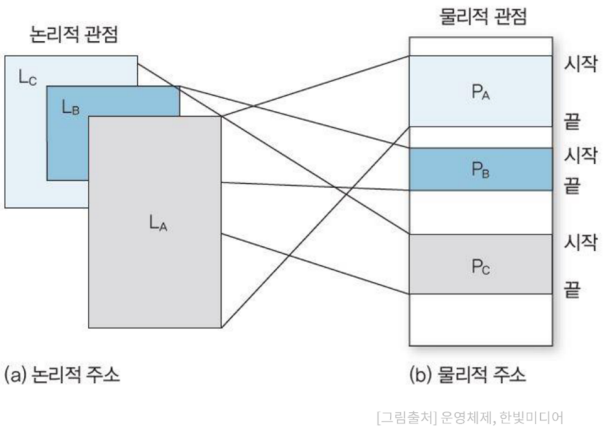
메모리(기억장치) 계층구조



- **Block**
  - 보조 기억 장치와 주기억 장치 사이 데이터 전송 단위 (OS 따라 다르지만 보통 1~4kb)
- **Word**
  - 주기억 장치와 레지스터 사이 데이터 전송 단위 (16~64bits)
  - 레지스터의 크기가 Word 크기와 동일하고, 이만큼의 데이터를 한 번에 읽어옴
    - 32bit, 64bit 컴퓨터 얘기할 때의 기준이 Word 단위라고 말해도 틀린 말은 아님

Address Binding

> 프로그램의 논리 주소를 실제 메모리의 물리 주소로 매핑(mapping)하는 작업

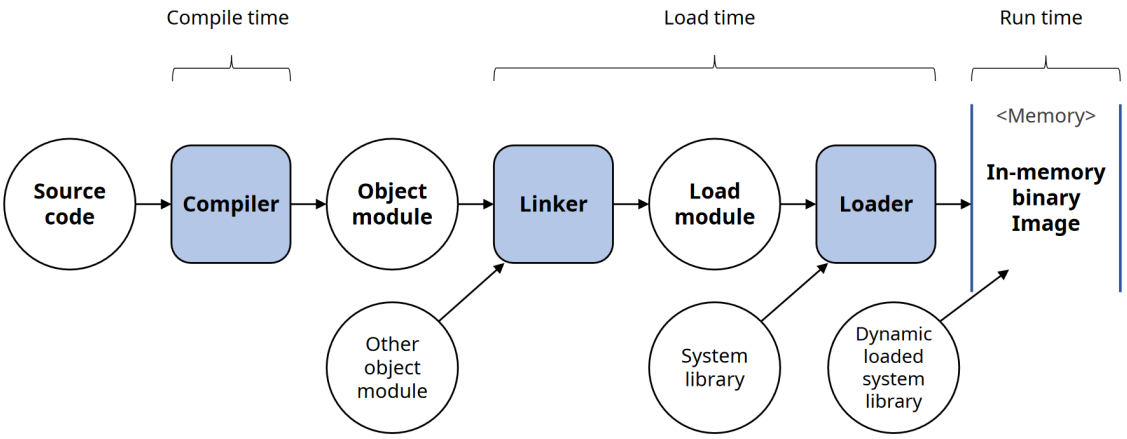


[그림출처] 운영체제, 한빛미디어

- **Binding 시점에 따른 구분**
  - Compile Time Binding

- Load Time Binding
- Run Time Binding

User Program Processing Steps



소스 코드를 메모리에 올려 실행하기까지의 과정

[Compile Time]

1. **Compile**
   - Compiler가 소스 코드를 object module로 변환

[Load Time]

2. **Linking**
   - Linker가 compile 과정에서 생성된 object module을 라이브러리 등 다른 object module과 묶어 실행 가능한 load module (`.exe` 등) 생성
3. **Loading**
   - load module을 실행하면, Loader가 load module을 메모리에 올려주는 작업 수행

[Run Time]

- 프로세스가 메모리에 올라가 실행됨

Compile Time Binding

> **소스 코드를 컴파일할 때** address binding 진행

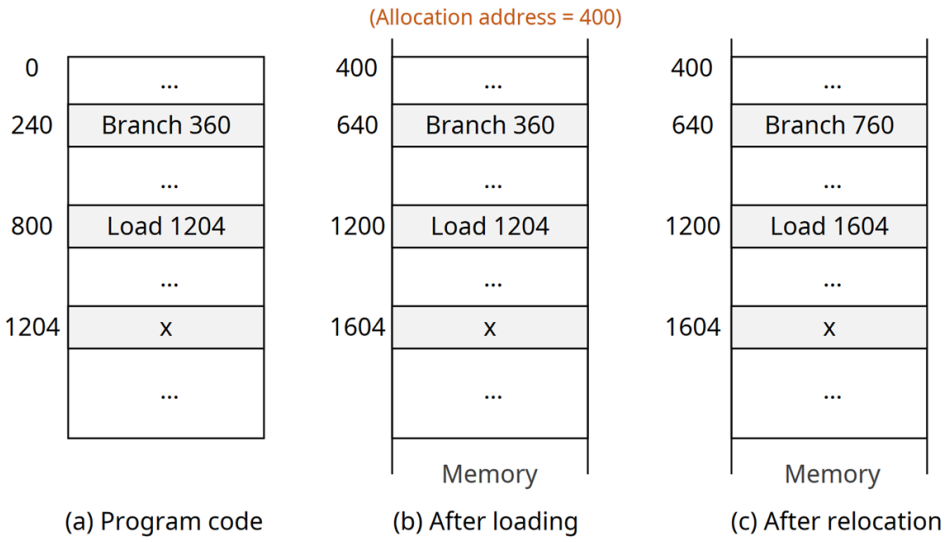
- **한계**
  - 컴파일러가 프로세스를 메모리의 어느 위치에 적재할지 미리 알고 있어야 함

        - 프로세스와 매핑된 물리 주소를 변경할 수 없음
    - 프로그램 전체가 메모리에 올라가야 함 (위치가 이미 정해져 있기 때문)

Load Time Binding

> **프로그램을 로드할 때** address binding 진행

[Load Time Binding 과정]



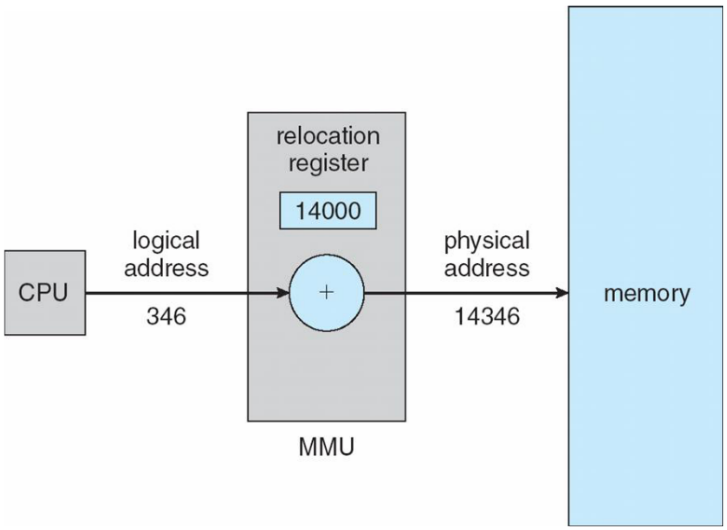
a. 컴파일 시점에 메모리 적재 위치를 모른다면, 대체 가능한 상대 주소를 생성
b. 프로그램을 메모리에 로드
c. 실제 적재 시점(load time)의 시작 주소를 반영해 매핑 재설정(relocation)

[Load Time Binding 특징]

- 한계
    - 프로그램 전체가 메모리에 올라가야 함 (상대 주소로 미리 위치 정해둠)

Runtime Binding

> **실행 시간에** address binding 진행
>
> - 실행 시간: 프로세스가 READY ➡️ RUNNING으로 상태 전이할 때



[그림출처] 운영체제, 한빛미디어

- 프로세스가 수행 도중 다른 메모리 위치로 이동할 수 있음
  - READY ⇒ RUNNING 상태 전이할 때마다 새롭게 address binding 진행
- HW의 도움이 필요: MMU (Memory Management Unit)
- 대부분의 OS가 사용하고 있는 Address Binding 방식

Dynamic Binding

> **프로그램 실행 중, 루틴(function)의 호출 시점에** 메모리 적재 및 address binding 진행

[Dynamic Binding 과정]

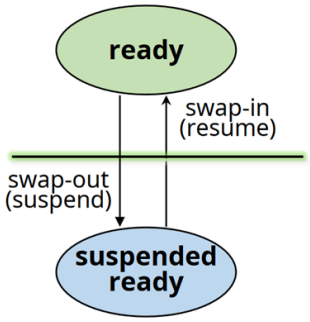
- 모든 루틴(function)을 교체 가능한 형태로 디스크에 저장
- 실제 호출 전까지는 메모리에 루틴을 적재하지 않음
  - 메인 프로그램만 메모리에 적재해 수행하다가 루틴의 호출 시점에 Address Binding 수행

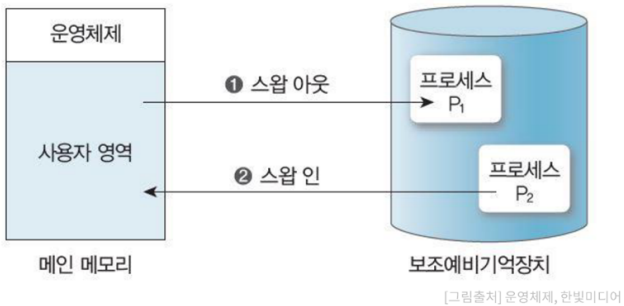
[Dynamic Binding 특징]

- 장점
  - 메모리 공간의 효율적 사용

Swapping

> 메인 메모리에서 실행되고 있는 프로세스를 보조 기억 장치에 저장된 프로세스와 일시적으로 교체해 메인 메모리 사용률을 높이는 메모리 관리 기법





[그림출처] 운영체제, 한빛미디어

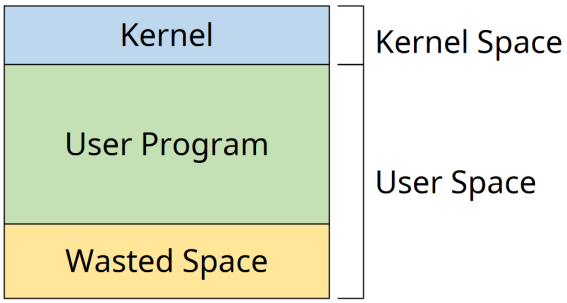
- **Swap Out**
  - READY/BLOCKED 상태에 있다가 프로세서를 뺏긴 프로세스를 swap device로 보냄
  - 이 때, 메모리 image를 swap device에 저장
- **Swap In**
  - swap device에 있던 프로세스를 다시 메모리에 적재

Continuous Memory Allocation

- 프로세스(context)를 하나의 연속된 메모리 공간에 할당하는 정책
  - 프로그램, 데이터, 스택 등
- **Continuous Memory Allocation 메모리 구성 정책에서 고려해야 할 점**
  - Multiprogramming Degree: 메모리에 동시에 올라갈 수 있는 프로세스 수
  - 각 프로세스에 할당되는 메모리 공간 크기
  - 메모리 분할 방법

Uni-Programming

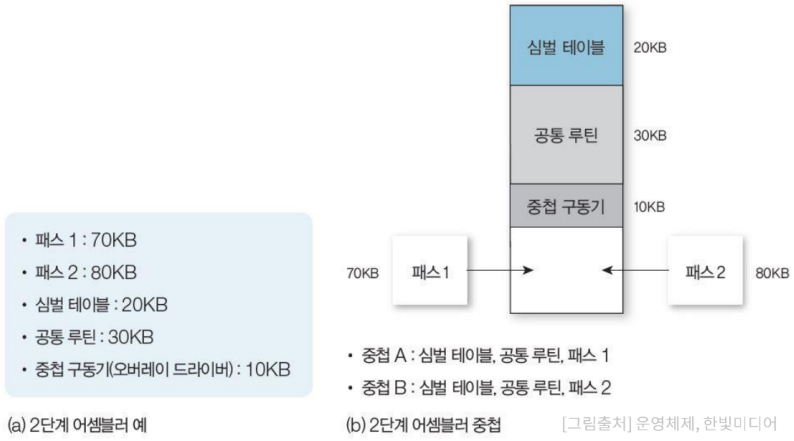
> 하나의 프로세스만 메모리 상에 존재하는 경우. 가장 간단한 메모리 관리 기법



Uni-Programming의 경우, 프로세스를 User Program 메모리 공간에 적재하면 됨

[Uni-Programming 문제점 ➡️ 해결 방안]

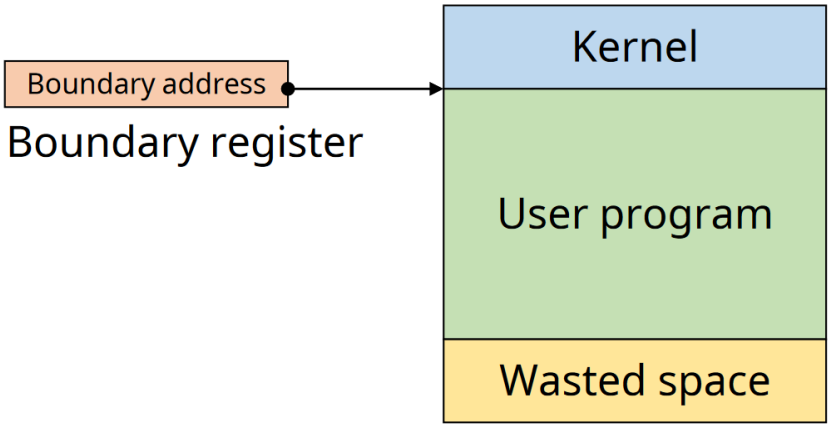
1. 프로그램의 크기 > 메모리의 크기 ➡️ **Overlay Structure**



[그림출처] 운영체제, 한빛미디어

- 메모리에 현재 필요한 영역만 적재
  - 공통된 부분은 계속 유지해두고, 그때그때 필요한 부분을 메모리에 적재
- 사용자가 프로그램의 흐름 및 자료구조를 모두 알고 있어야 가능

2. 커널 보호 ➡️ 경계 레지스터 사용

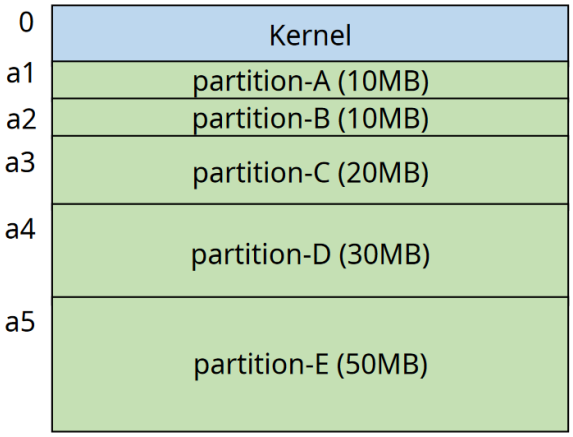


- 프로세스를 메모리에 적재할 때, 커널을 침범할 수 있음
- 경계 레지스터 안에 경계 주소를 미리 저장, 프로세스가 적재되지 않도록 함

[Uni-Programming의 한계]

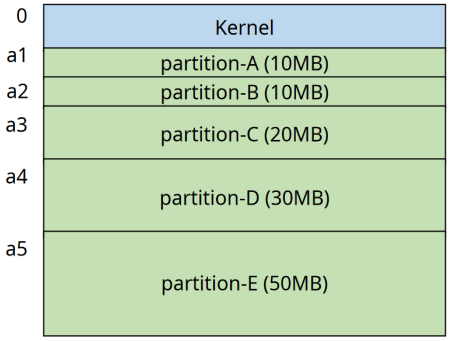
- 메모리에 프로세스 하나만 적재하기 때문에 아래 문제 발생
  - Low system resource utilization
  - Low system performance
- ➡️ Multi-Programming

Fixed Partition Multi-Programming (FPM)



- 메모리 공간을 고정된 크기로 분할
  - 프로세스 적재 전 미리 분할
- 각 프로세스를 하나의 partition(분할)에 적재
  - Partition의 수 = Mutiprogramming Degree

[Fixed Partition Multi-Programming 자료구조 예시]



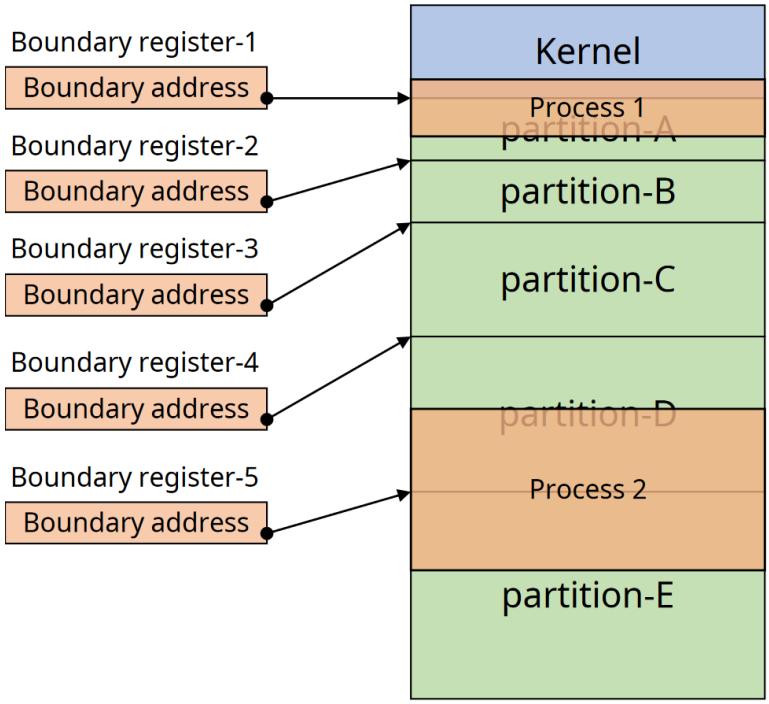
partition	start address	size	current process ID	other fields
A	a1	10 MB	-	···
B	a2	10 MB	-	···
C	a3	20 MB	-	···
D	a4	30 MB	-	···
E	a5	50 MB	-	···

<Partition table or State table>

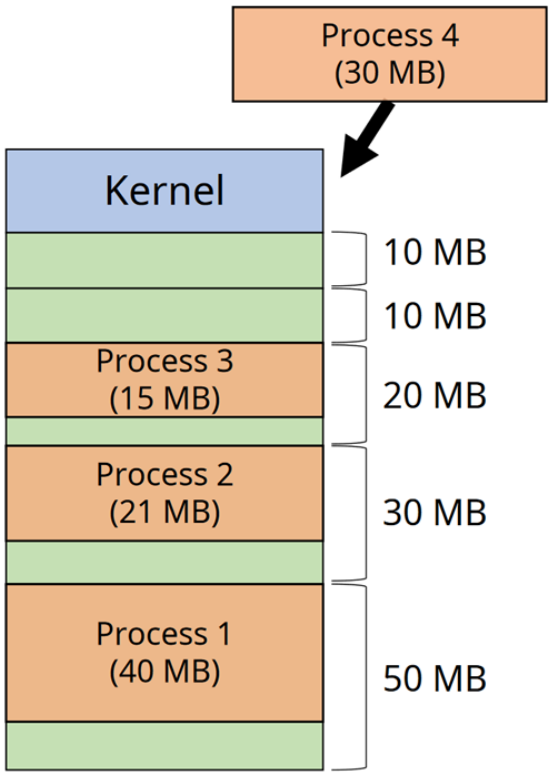
- Partition Table을 사용해 각 Partition에 어떤 프로세스가 적재되어 있는지 관리

[Fixed Partition Multi-Programming 문제점 ➡️ 해결 방안]

1. 커널 및 사용자 영역 보호 ➡️ 경계 레지스터 여러 개 사용



2. **Fragmentation (단편화)**



- Internal Fragmentation (내부 단편화)
  - Partition 크기 > Process 크기인 경우, Partition 내 낭비되는 공간이 발생함
- External Fragmentation (외부 단편화)
  - 남은 메모리 크기 > Process 크기지만, 연속된 공간이 아니라 프로세스 적재 불가 ⇒ 메모리 낭비

[Fixed Partition Multi-Programming 특징]

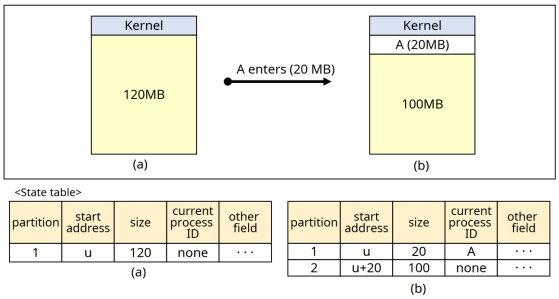
- 장점
  - 고정된 크기로 메모리가 미리 분할되어 있기 때문에 메모리 관리 간편 ⇒ Low overhead
- 단점
  - 시스템 자원이 낭비될 수 있음 ⇒ Internal/External fragmentation
    - ➡️ Variable Partition Multi-Programming

Variable Partition Multi-Programming (VPM)

- 초기에는 메모리 전체가 하나의 영역
- 프로세스 요청이 들어올 때 메모리 공간을 동적으로 분할

- Internal Fragmentation 발생 X

[Variable Partition Multi-Programming 예시]



<State table>

partition	start address	size	current process ID	other field
1	u	120	none	···

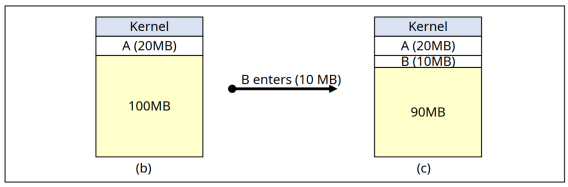
(a)

partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	100	none	···

(b)

a. 프로세스 A가 메모리 20MB 요청

b. 프로세스 A 적재, partition table 갱신



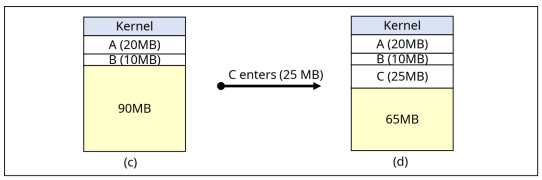
partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	100	none	···

(b)

partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	B	···
3	u+30	90	none	···

(c)

c. 프로세스 B(10MB) 적재



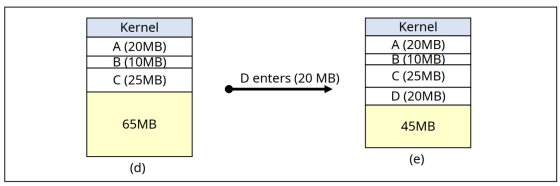
partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	B	···
3	u+30	90	none	···

(c)

partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	B	···
3	u+30	25	C	···
4	u+55	65	none	···

(d)

d. 프로세스 C(25MB) 적재



partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	B	···
3	u+30	25	C	···
4	u+55	65	none	···

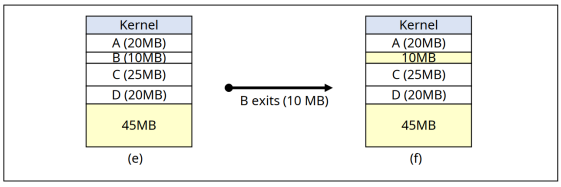
(d)

partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	B	···
3	u+30	25	C	···
4	u+55	20	D	···
5	u+75	45	none	···

(e)

e. 프로세스 D(20MB) 적재

- Internal Fragmentation 발생 X



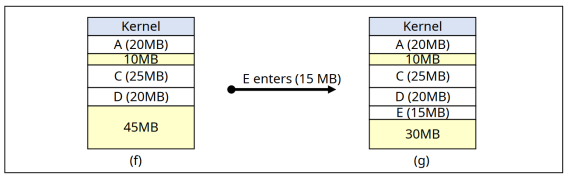
partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	B	···
3	u+30	25	C	···
4	u+55	20	D	···
5	u+75	45	none	···

(e)

partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	none	···
3	u+30	25	C	···
4	u+55	20	D	···
5	u+75	45	none	···

(f)

f. 프로세스 B가 메모리 반납



partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	none	···
3	u+30	25	C	···
4	u+55	20	D	···
5	u+75	45	none	···

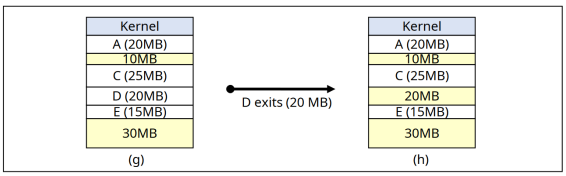
(f)

partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	none	···
3	u+30	25	C	···
4	u+55	20	D	···
5	u+75	15	E	···
6	u+90	30	none	···

(g)

g. 프로세스 E(15MB) 적재

- 분할된 채로 비어있는 메모리 공간 발생



partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	none	···
3	u+30	25	C	···
4	u+55	20	D	···
5	u+75	15	E	···
6	u+90	30	none	···

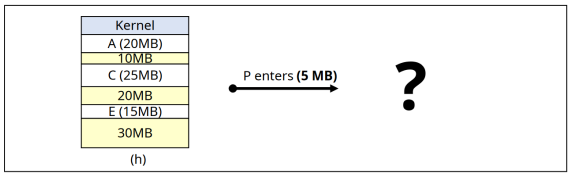
(g)

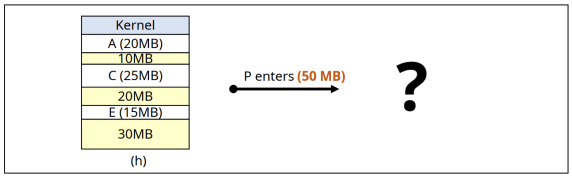
partition	start address	size	current process ID	other field
1	u	20	A	···
2	u+20	10	none	···
3	u+30	25	C	···
4	u+55	20	none	···
5	u+75	15	E	···
6	u+90	30	none	···

(h)

h. 프로세스 D가 메모리 반납

- 기존 프로세스 B 있던 공간에 적재 불가



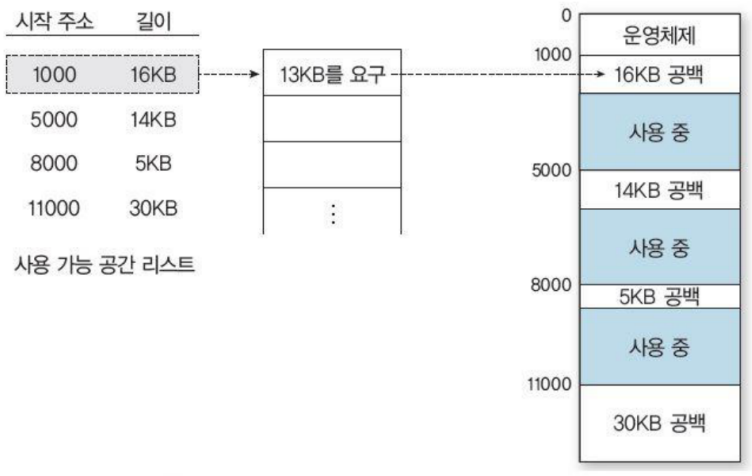


- 새로운 프로세스가 메모리 요청한다면, 어디에 적재해야 하는가?
  - Internal Fragmentation ➡️ **배치 전략**
  - External Fragmentation

    ➡️ **Coalescing holes, Storage compaction**

Placement Strategies

[First-fit (최초 적합)]



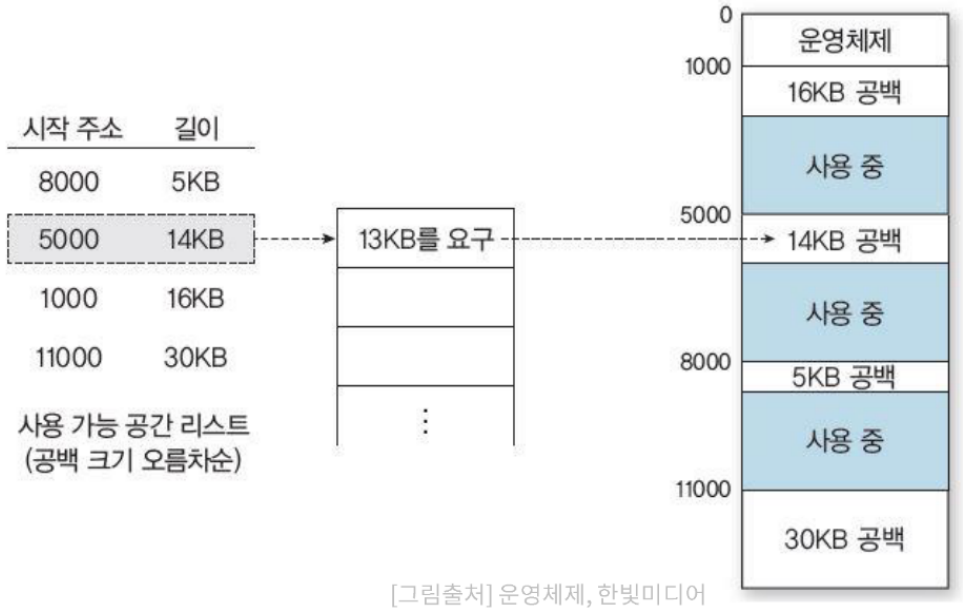
[그림출처] 운영체제, 한빛미디어

- **정의**

- 충분한 크기를 가진 첫번째 partition 선택

- **장점**
  - 단순하고 오버헤드 적음
- **단점**
  - 상황에 따라 공간 활용률이 떨어질 수 있음

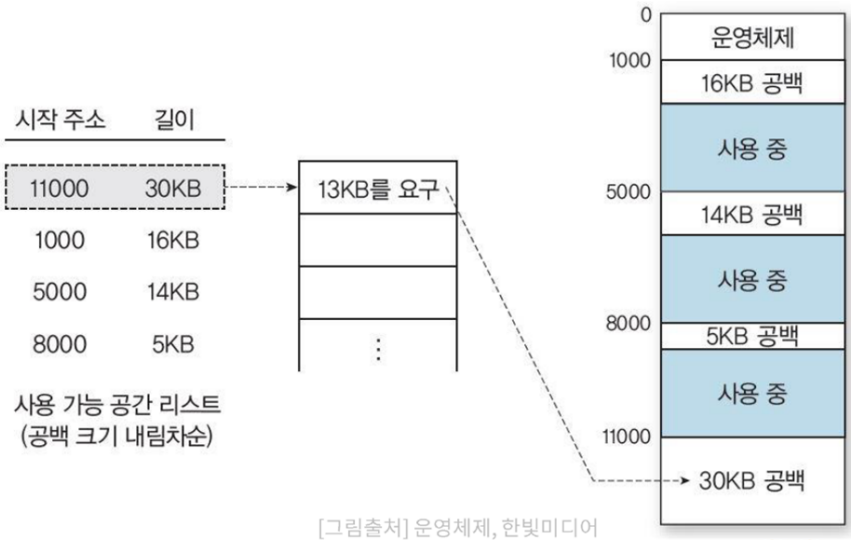
[Best-fit (최적 적합)]



[그림출처] 운영체제, 한빛미디어

- **정의**
  - 프로세스가 들어갈 수 있는 partition 중 가장 작은 곳 선택
- **장점**
  - 크기가 큰 partition을 유지할 수 있음
- **단점**
  - 모든 partition을 살펴봐야 하기 때문에 탐색 시간이 오래 걸림. 오버헤드 높음
  - 활용하기 너무 작은 크기의 partition이 많이 발생할 수 있음

[Worst-fit (최악 적합)]



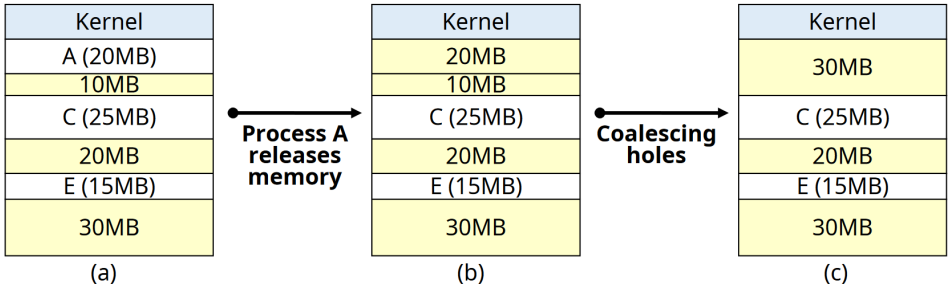
[그림출처] 운영체제, 한빛미디어

- **정의**
  - 프로세스가 들어갈 수 있는 partition 중 가장 큰 곳 선택
- **장점**
  - 작은 크기의 partition 발생을 줄일 수 있음
- **단점**
  - 모든 partition을 살펴봐야 하기 때문에 탐색 시간이 오래 걸림. 오버헤드 높음
  - 큰 프로세스에게 필요한 큰 크기의 partition 확보가 어려움

[Next-fit (순차 최초 적합)]

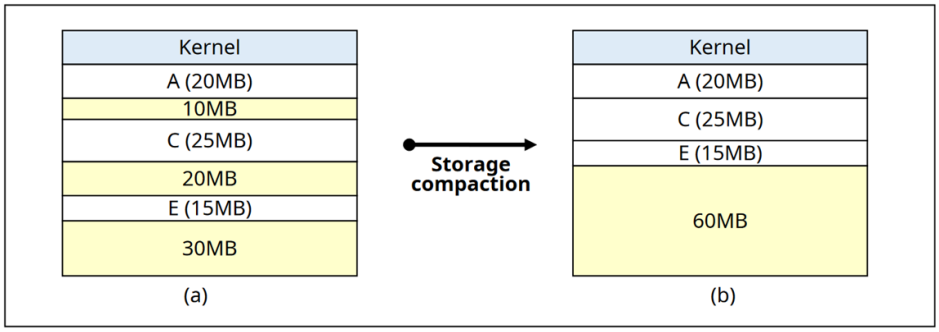
- **정의**
  - 최초 적합 전략과 유사하나, state table에서 마지막으로 탐색한 위치부터 시작해 발견한 충분한 크기의 첫번째 partition 선택
- **장점**
  - 단순하고 오버헤드 적음
  - 메모리 영역의 사용 빈도 균등화
- **단점**
  - 상황에 따라 공간 활용률이 떨어질 수 있음

Coalescing Holes (공간 통합)



- 인접한 빈 영역을 하나의 partition으로 통합
  - 프로세스가 메모리를 반납하고 나갈 때 수행
  - 인접 영역을 통합하기 때문에 오버헤드 낮음

Storage Compaction (메모리 압축)



partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	C
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	20	A
2	u+20	25	C
3	u+45	15	E
4	u+60	60	none

- 모든 빈 공간을 하나로 통합
  - 프로세스 처리에 필요한 적재 공간 확보가 필요할 때 수행 (자주 수행 X)
  - 모든 프로세스를 중지한 후 재배치하는 과정을 거쳐야 하기 때문에 오버헤드 높음