실제 메모리 구성과 관리

10th Week

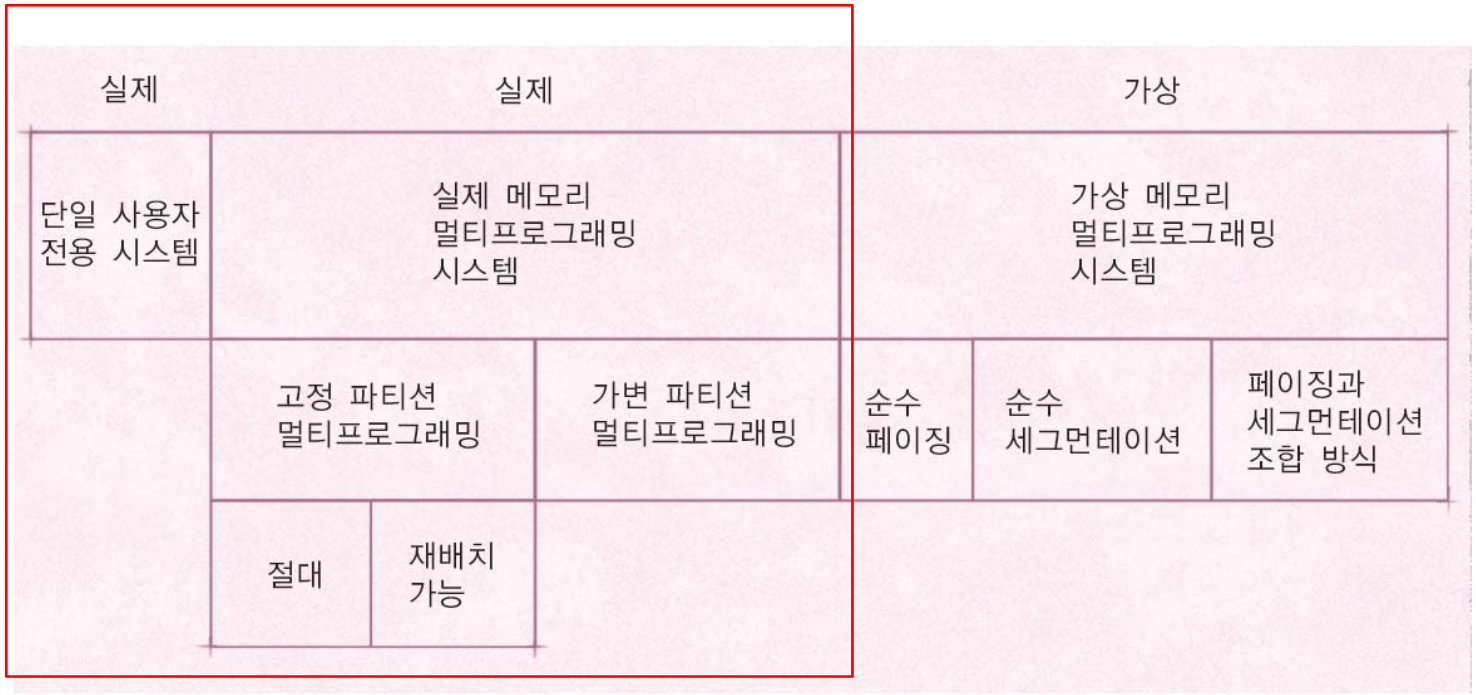
Kim, Eui-Jik

Contents

- 소개
- 메모리 구성
- 메모리 관리
- 메모리 계층
- 메모리 관리 전략
- 연속/불연속 메모리 할당
- 단일 사용자 연속 메모리 할당
- 고정 파티션 멀티 프로그래밍
- 가변 파티션 멀티 프로그래밍

소개

- Memory management



[그림 10-1] 메모리 구성의 진화

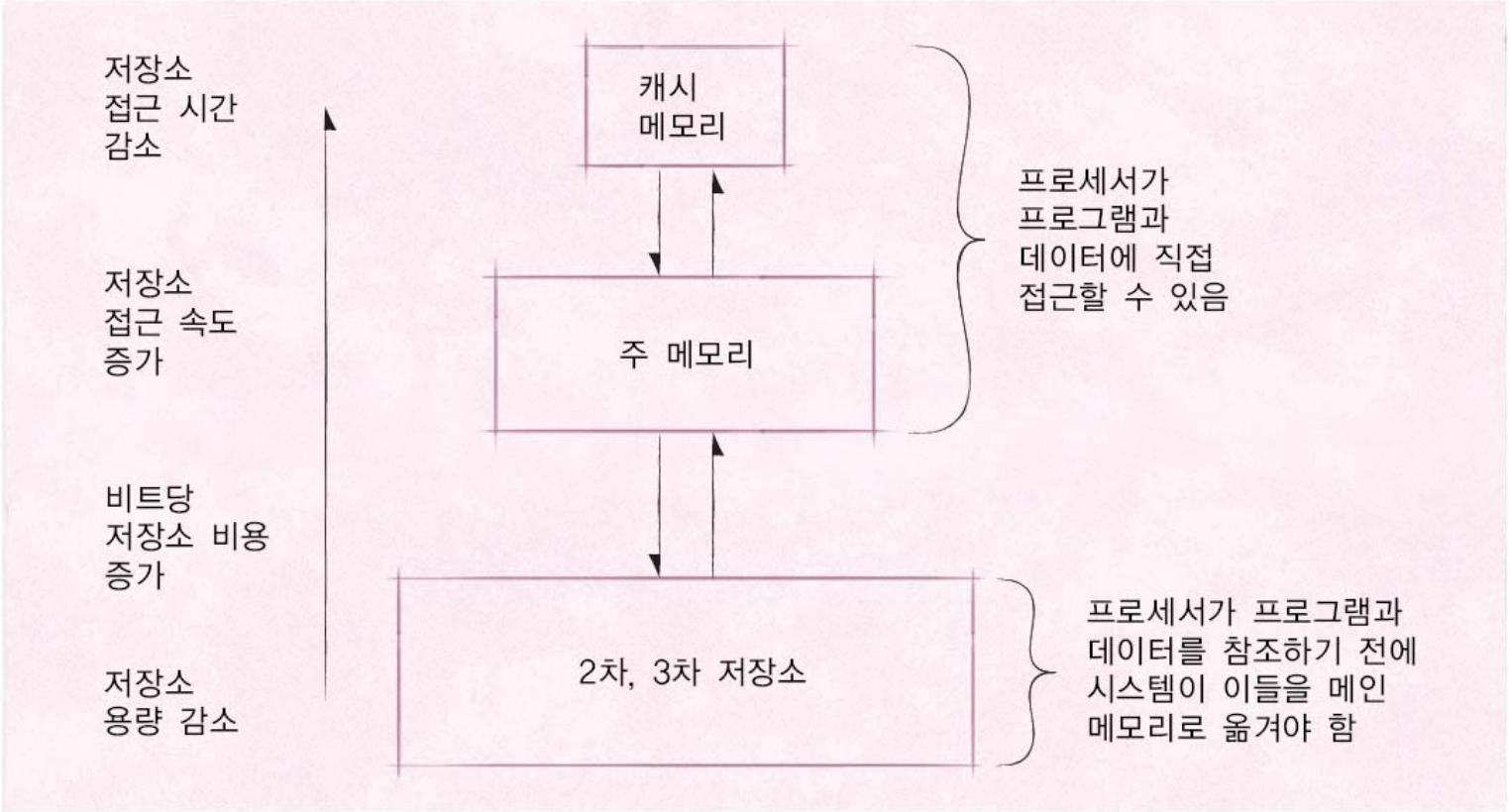
메모리 관리

- 최적의 메모리 성능을 내기 위한 전략 결정
  - 메모리 관리자(memory manager)
    - 시스템의 메모리 구성과 관리 전략을 담당하는 운영체제 구성 요소
    - 어떤 프로세스를 메모리에 머무르게 할 것인가?
    - 얼마만큼의 메모리를 각 프로세스에 할당할 것인가?
    - 각 프로세스를 메모리의 어디에 둘 것인가?

메모리 계층

- 메인 메모리(main memory)
  - 시스템이 당장 필요한 프로그램과 데이터
- 2차 저장소(secondary storage)
  - 시스템에 곧장 필요하지 않은 프로그램이나 데이터
- 캐시 메모리(cache memory)
  - 매우 빠름
  - 보통 각 프로세서에 위치
  - 시간적 지역성(temporal locality)

메모리 계층



[그림 9-1] 계층적 메모리 구성

메모리 관리 전략

- 메모리 관리 전략 구분
  - 페치 전략(fetch strategies)
    - 2차 저장소에 있는 프로그램이나 데이터를 메인 메모리로 옮길 시점 결정
    - 요구 페치(demand fetch strategies)

      - 운영체제나 시스템 프로그램, 사용자 프로그램 등의 참조요구에 따라 메인 메모리에 적재하는 방법
    - 예측 페치(anticipatory strategies)

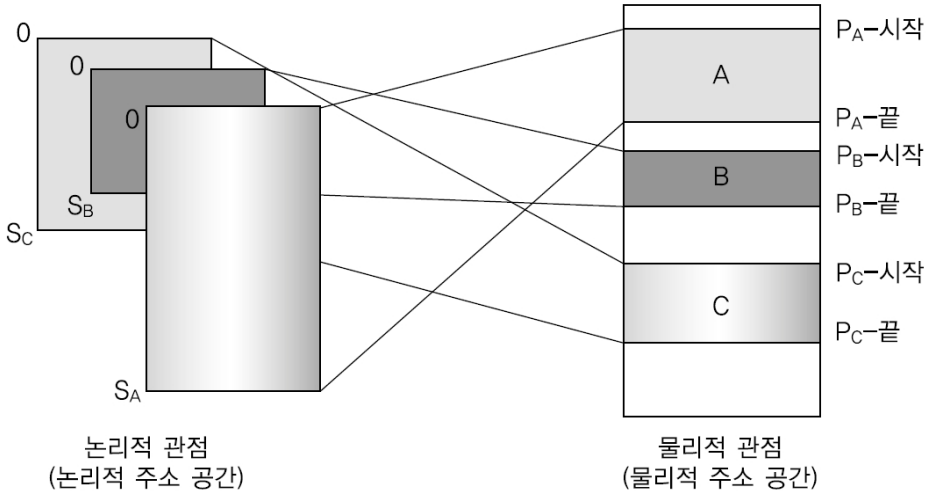
      - 시스템의 요구를 예측하여 메모리에 미리 적재하는 방법
  - 배치 전략(placement strategies)
    - 새로 로드하는 프로그램이나 데이터의 부분을 놓을 메인 메모리 위치 결정
    - 최초 적합(first-fit)
    - 최적합(best-fit)
    - 최악 적합(worst-fit)
  - 교체 전략(replacement strategies)
    - 메모리가 부족할 경우, 어떤 부분을 제거해 새로운 데이터를 메모리에 배치할지 결정

연속/불연속 메모리 할당

- 프로그램을 메모리에 할당하는 방법
  - 연속 메모리 할당
    - 연속적인 메인 메모리 공간에 할당
    - 프로그램이 여유 메모리보다 큰 경우의 문제
    - 낮은 오버헤드
  - 불연속 메모리 할당
    - 프로그램을 블록 또는 세그먼트(segment)로 나눔
    - 각 세그먼트는 메모리에 불연속적으로 배치
    - 메모리 홀 활용
    - 큰 오버헤드
    - 멀티프로그래밍 수준 증가

[참조] 메모리 관리 개념

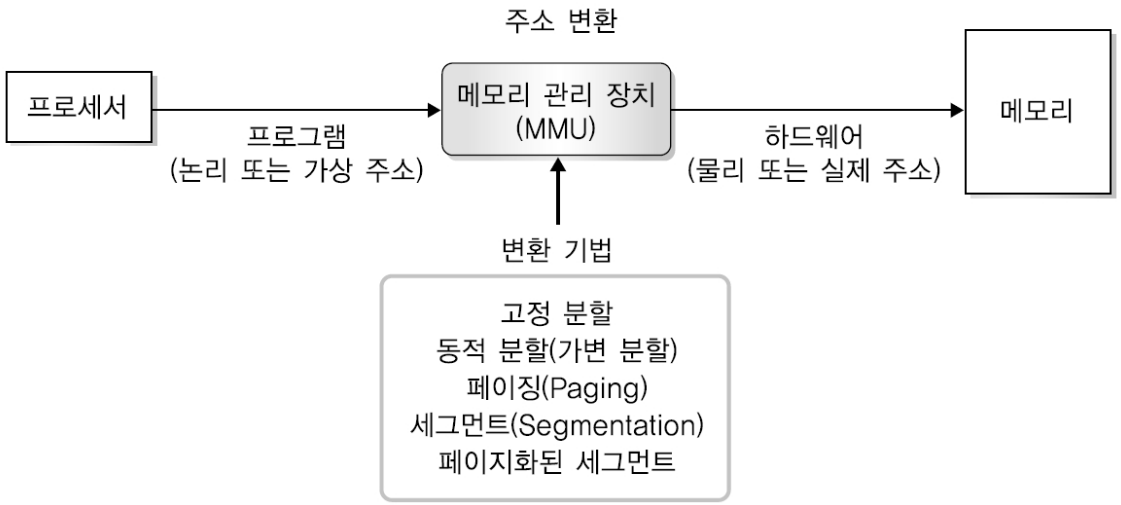
- 메모리 해석에 대한 두 가지 관점
  - 물리적 공간(물리적 주소)
    - 실제 데이터나 프로그램이 저장되는 공간
    - 메모리 칩(Chip) 또는 디스크 공간으로 생성. 사용되는 단위는 바이트(Byte)
    - 논리적 주소보다 크거나, 작거나, 같을 수 있음
  - 논리적 공간(논리적 주소)
    - 프로그래머가 프로그래밍에 사용하는 공간
    - 목적코드(Object Code)가 저장된 공간과 프로그램에서 사용하는 자료 구조 등이 해당됨
    - 논리적 메모리 크기는 각 시스템에서 정의한 워드의 길이에 따라 다름



메모리 해석에 대한 두 가지 관점

[참조] 메모리 관리 개념

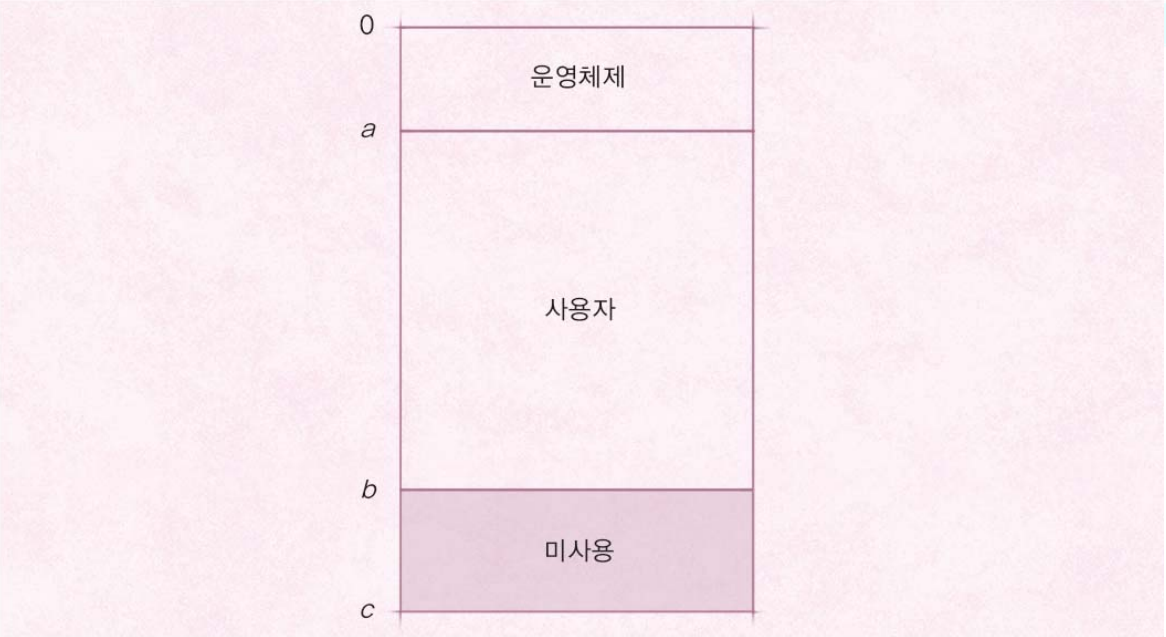
- 메모리 매핑(Memory Mapping)
  - 논리적 주소와 물리적 주소의 연결
  - 메모리 관리 장치(MMU, Memory Management Unit)인 하드웨어에서 실행
  - 메모리 관리 방식에 따라 여러 방식으로 구분됨
    - 고정 분할
    - 동적 분할(가변 분할)
    - 페이징(Paging)
    - 세그먼트(Segment)
    - 페이지화된 세그먼트 방식 등



메모리 관리 장치에 의한 메모리 매핑

단일 사용자 연속 메모리 할당

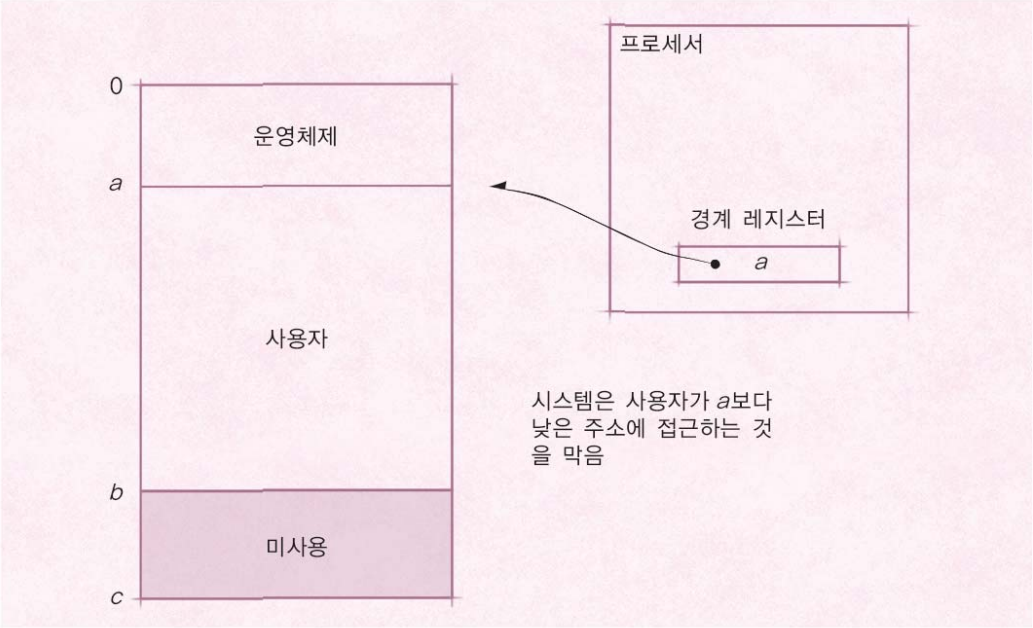
- 한 번에 한 사용자만 시스템 사용
  - 자원 공유 불필요
  - 운영체제 없음
    - 프로그래머가 모든 코드 구현



[그림 9-2] 단일 사용자 연속 메모리 할당

단일 사용자 연속 메모리 할당

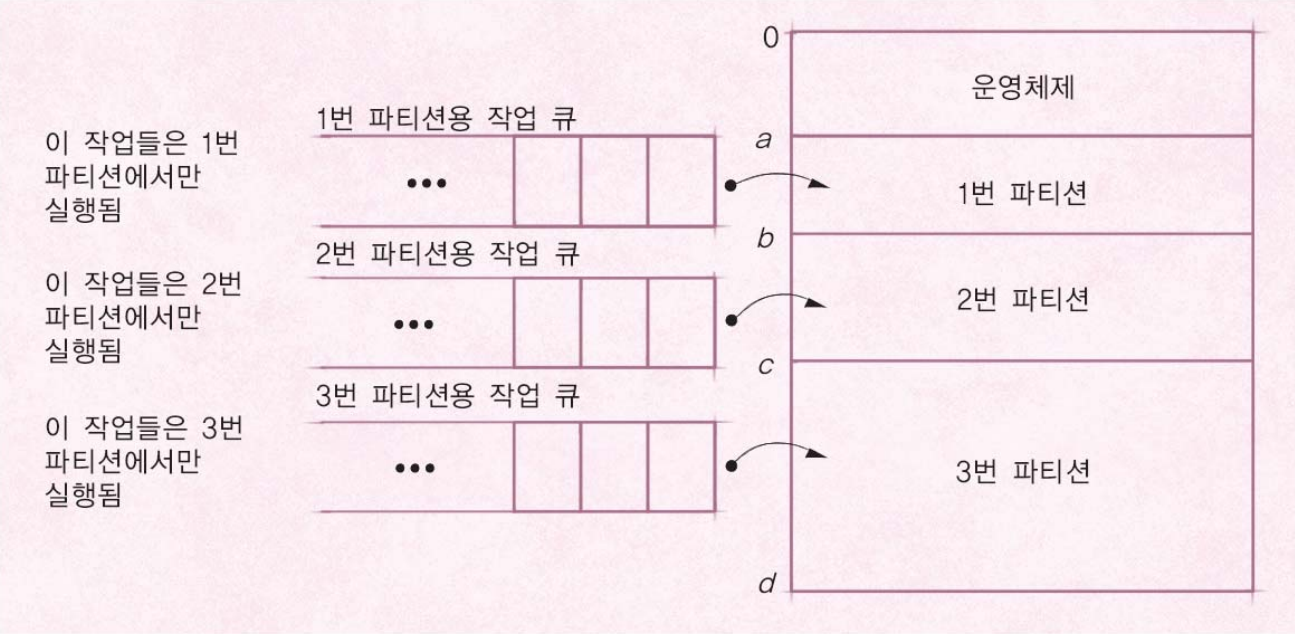
- 단일 사용자 시스템에서의 보호
  - 프로세스들로부터 운영체제 보호
  - 경계 레지스터 (boundary register)
    - 사용자 프로그램의 시작 메모리 주소 정보 포함
    - 접근 불가 메모리 접근 시 거부
      - 사용자 프로그램이 메모리 주소를 참조할 때마다 경계 레지스터를 검사한 후 실행됨



[그림 9-4] 단일 사용자 연속 메모리 할당을 사용한 메모리 보호

고정 파티션 멀티프로그래밍

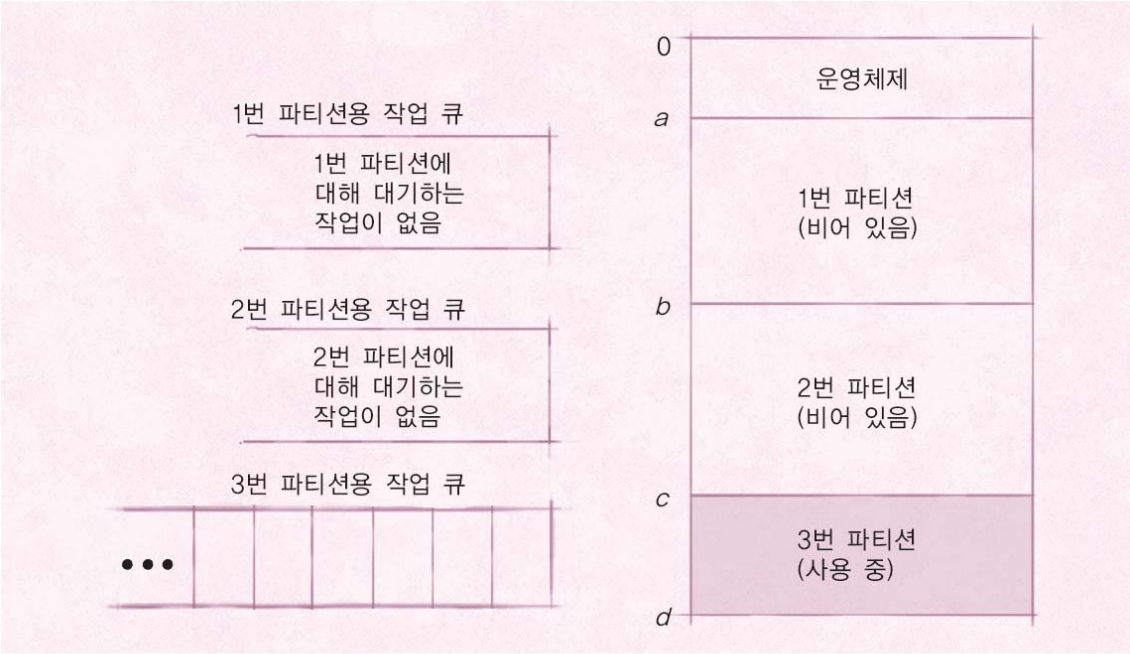
- 고정 파티션 멀티프로그래밍
  - 메인 메모리를 고정 크기 파티션으로 나눔
  - 각 파티션 한 작업만 보유
  - 작업간 프로세서 전환
  - 많은 제약조건
    - 메모리 낭비



[그림 9-6] 절대 변환과 로딩 방식을 지닌 고정 파티션 멀티프로그래밍

고정 파티션 멀티프로그래밍

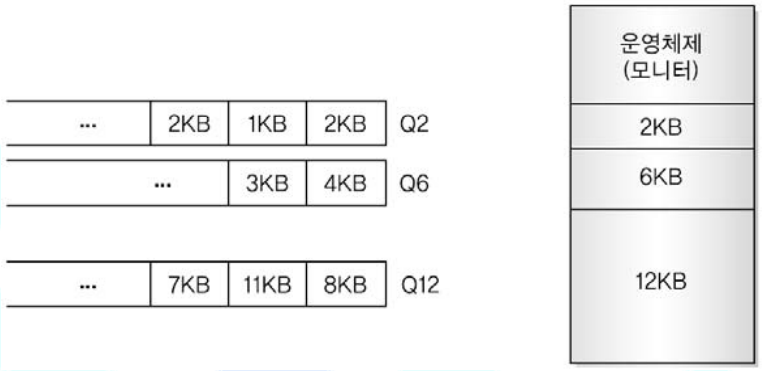
- 고정 파티션의 단점
  - 작업 시작 전 메모리의 위치 결정, 지정된 파티션에서만 실행
    - 메모리 낭비
  - 파티션이 이미 사용되면, 대기



[그림 9-7] 절대 변환과 로딩 방식을 지닌 고정 파티션 멀티프로그래밍의 메모리 낭비 예

[참고] 고정 파티션 멀티프로그래밍

- 고정 분할 시스템 예
  - 프로세스 큐 하나의 크기가 2KB인 Q2, 6KB인 Q6, 12KB인 Q12가 있는 경우
  - 작업량이 2KB 미만은 Q2로, 6KB 미만은 Q6, 12KB 미만은 Q12로 보냄
  - 큐에 할당하는 과정은 자동적으로 처리되며, 최대 메모리 요구량을 큐에 표시함
  - 각 큐는 자신의 기억 영역을 가지므로 큐 사이에 경쟁은 없음

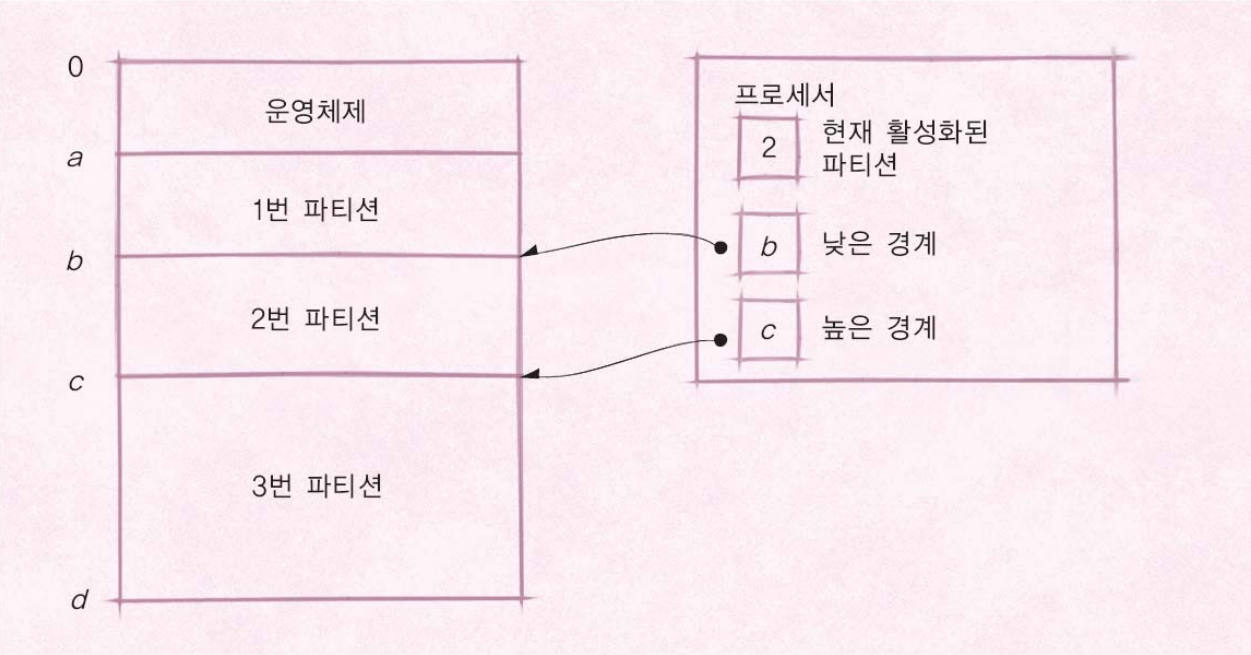


각 영역에 독립된 큐를 갖는 고정 분할 시스템

  - 문제점
    - 크기에 따라 각 분할 영역을 담당하는 큐가 있어 Q12 큐가 이미 다 차있는 경우, 다른 큐(Q2, Q6)가 비어있더라도 큐를 이용할 수 없음

고정 파티션 멀티프로그래밍

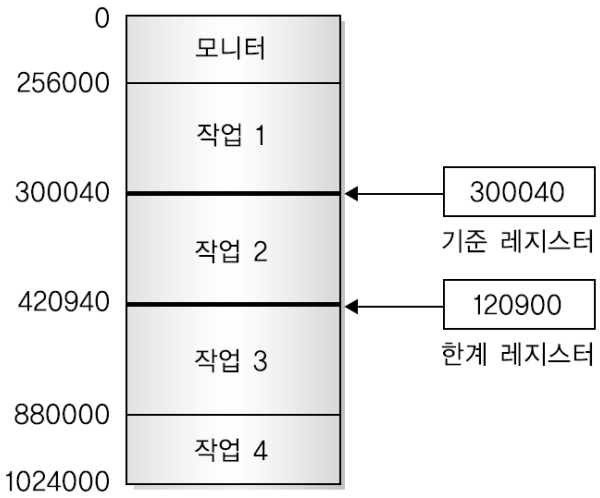
- 보호 메커니즘
  - 여러 경계 레지스터 사용
    - 기준(base) 및 상한(limit) 레지스터
    - 요청된 주소
      - 기준 보다 높음
      - 상한 보다 낮음



[그림 9-9] 연속 할당 멀티프로그래밍 시스템의 메모리 보호

[참고] 고정 파티션에서의 memory protection

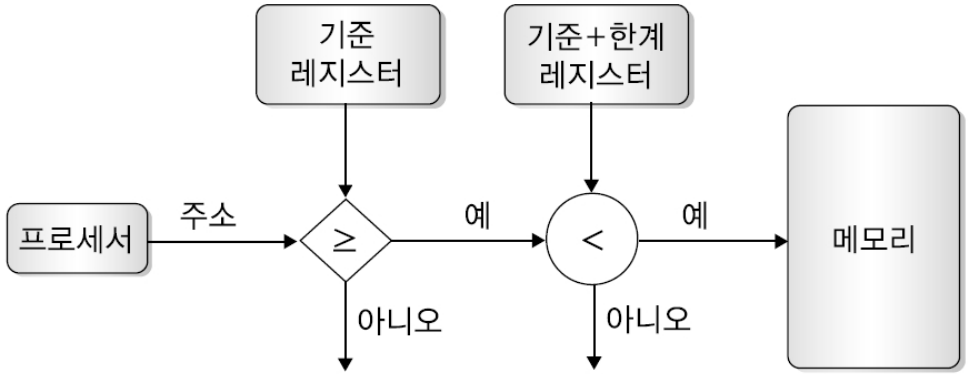
- 고정분할에서의 메모리 보호
  - 두 개의 레지스터(기준(base) 레지스터와 한계(limit) 레지스터)를 사용하여 분할된 영역 보호
    - 기준(base: 낮은 경계) 레지스터 : 가장 작은 합법적인 물리 메모리 주소(300040)를 저장
    - 한계(limit: 높은 경계) 레지스터 : 프로그램 영역이 저장되어 있는 범위의 크기(120900)를 저장



고정분할에서의 메모리 보호

[참고] 고정 파티션에서의 memory protection

- 고정분할에서의 메모리 보호 예
  - 앞페이지 그림에서, 작업 2는 가장 작은 물리적 주소의 하한값이 300040이고, 크기가 120900인 프로세스임
    - 기준 레지스터의 가장 작은 물리적 주소는 300040이며, 한계 레지스터는 논리 주소 120900임
    - 사용자 주소 범위는 0~120900으로, 논리 주소는 한계 레지스터 주소보다 작아야 함
    - 프로세스의 상한값은 420940(=기준 레지스터(300040)+한계 레지스터(120900))이 되므로, 사용자 주소 범위는 상한값과 하한값 사이가 됨
  - 프로세서에 의해 생성된 모든 주소는 레지스터와 함께 검사되므로 다른 사용자의 프로그램과 데이터를 보호할 수 있음

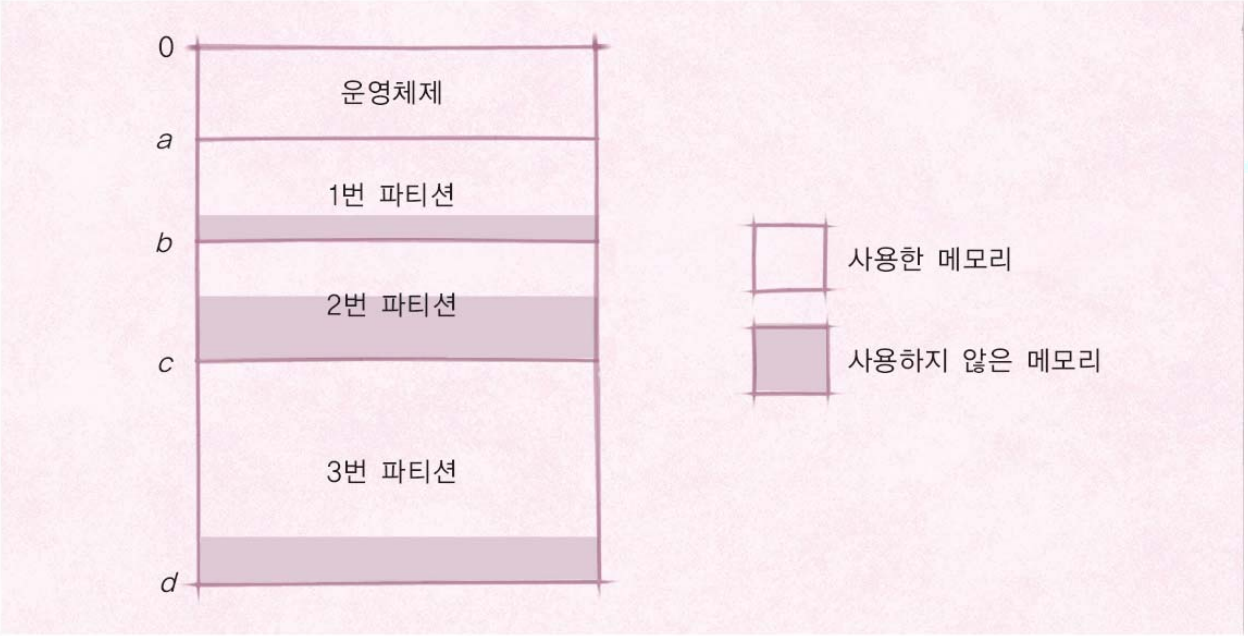


트랩: 주소 지정 오류

기준 레지스터와 한계 레지스터를 이용한 프로세스 보호

고정 파티션 멀티프로그래밍

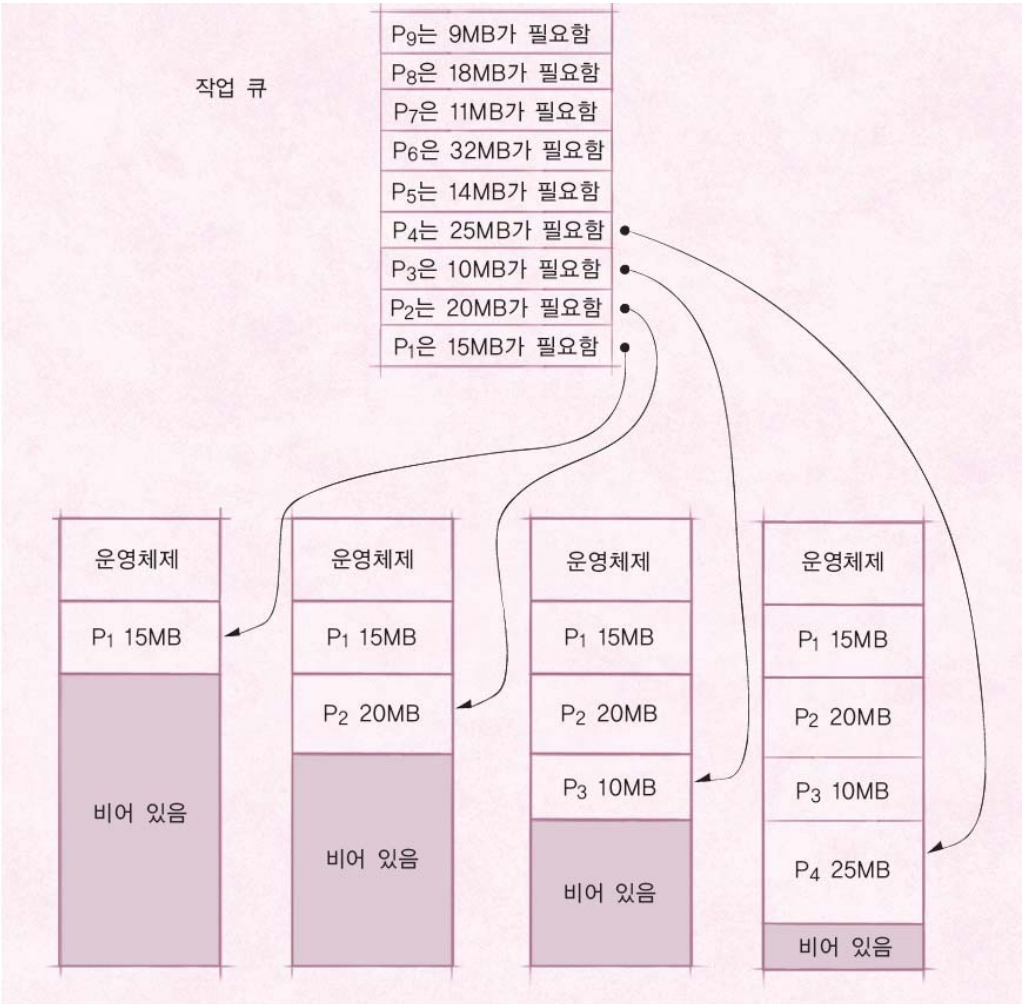
- 고정 파티션의 단점
  - 내부 단편화(internal fragmentation)
    - 프로세스의 메모리와 데이터 크기가 프로세스가 실행되는 파티션보다 작을 때



[그림 9-10] 고정 파티션 멀티프로그래밍 시스템의 내부 단편화

가변 파티션 멀티프로그래밍

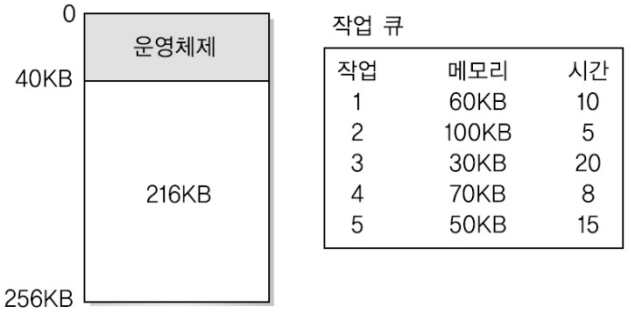
- 고정 파티션 멀티프로그래밍의 제약사항
  - 시스템 활용도 저하
  - 내부단편화



[그림 9-11] 가변 파티션 프로그래밍에서 초기 파티션 할당

[참고] 가변 파티션 멀티프로그래밍

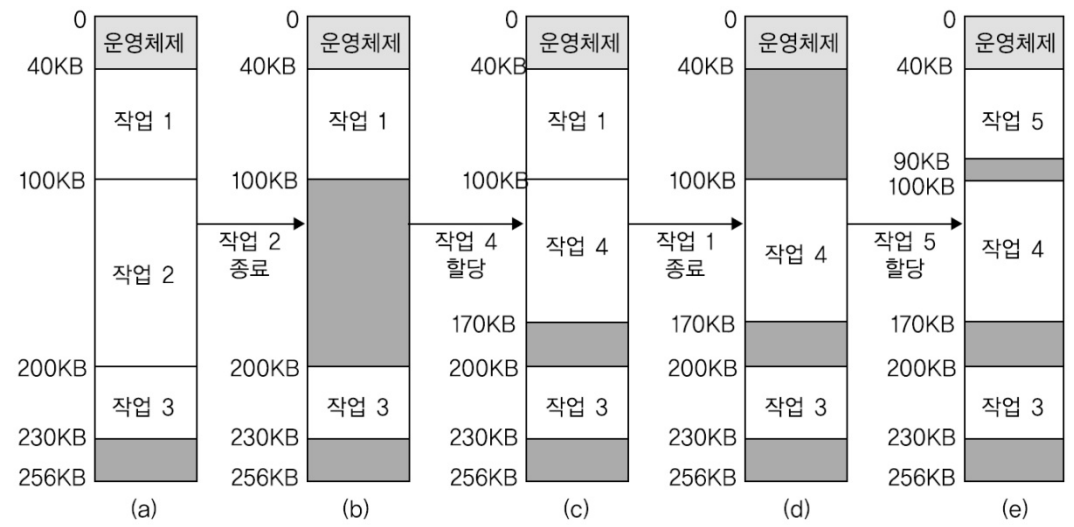
- 운영체제는 메모리의 사용 내역을 확인할 수 있는 테이블을 유지해야 함
  - 256KB의 이용할 수 있는 메모리와 40KB 크기의 운영체제 그리고 작업 큐에 그림과 같이 작업을 가진다 가정함
  - 가변 분할(동적 메모리 할당)은 요구된 크기 n을 사용가능 공간에 어떻게 할당하느냐의 문제임
  - 예: 최초 적합(First-Fit), 최상 적합(Best-Fit), 최악 적합(Worst-Fit) 등



작업 큐

작업	메모리	시간
1	60KB	10
2	100KB	5
3	30KB	20
4	70KB	8
5	50KB	15

메모리와 작업 큐



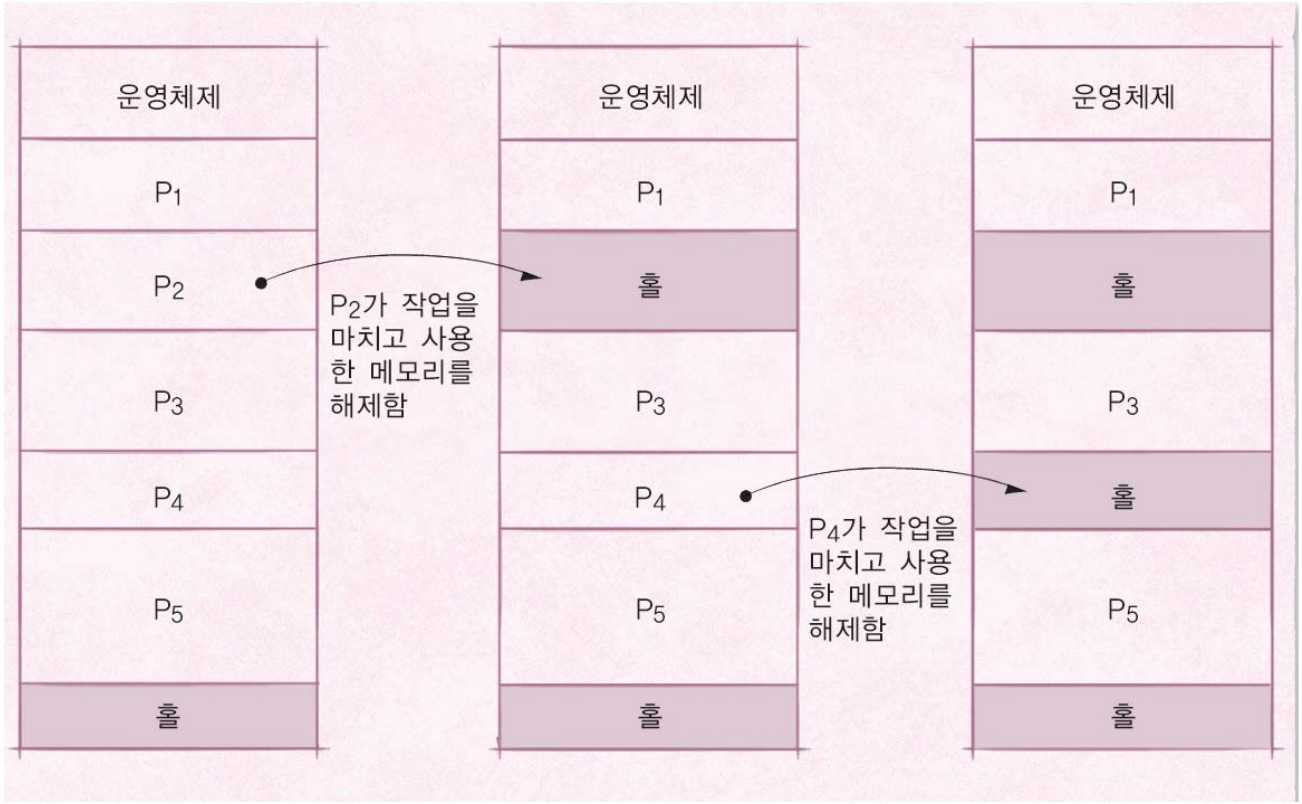
메모리 할당과 스케줄

가변 파티션 멀티프로그래밍

- 가변 파티션의 특징
  - 프로세스들이 필요한 공간만 차지
    - 자원 낭비 없음
    - 내부 단편화 문제 없음
      - 정확히 프로세스의 크기에 따라 파티션 할당
  - 외부 단편화
    - 프로세스 제거 시 발생
    - 계속되는 배치에 의한 홀의 생성
    - 프로세스를 수용하지 못하는 개별 홀

가변 파티션 멀티프로그래밍

- 가변 파티션의 특징



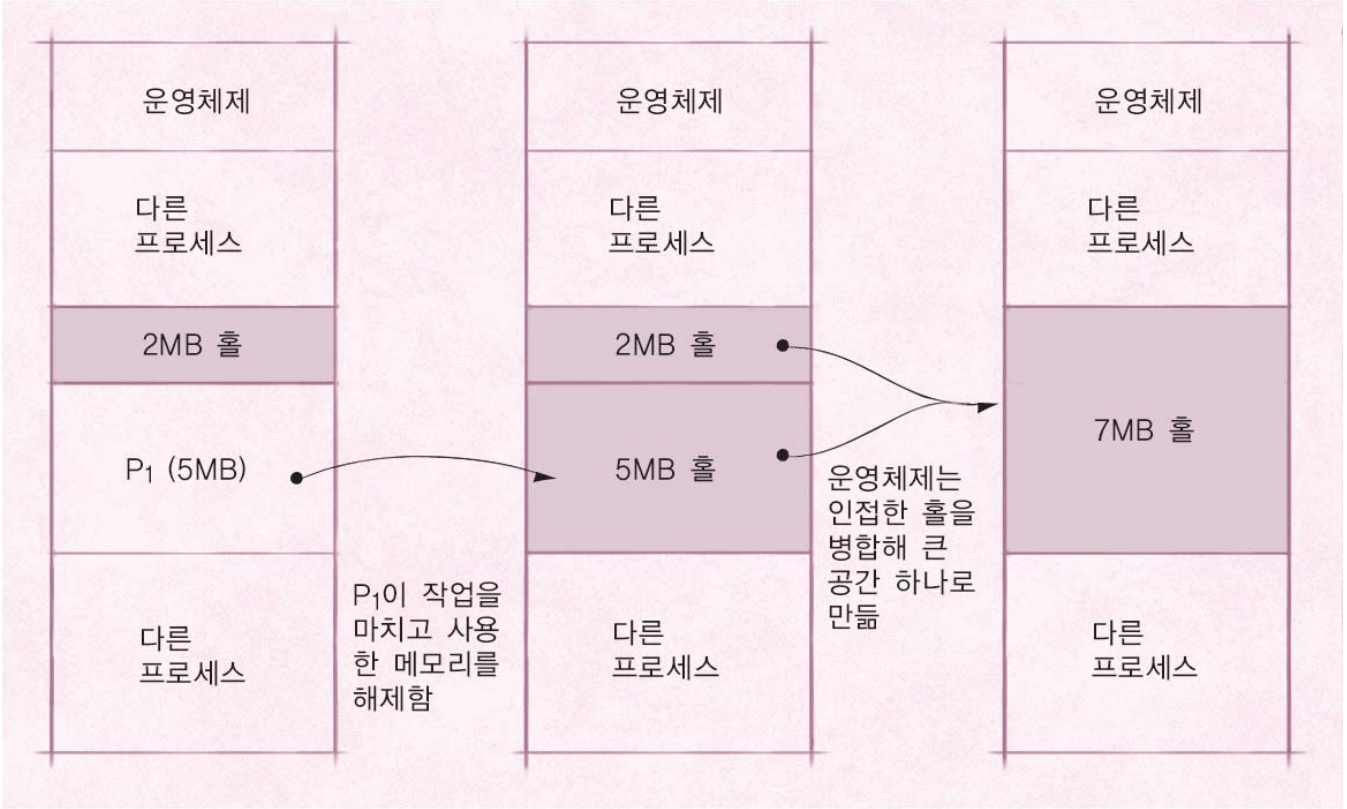
[그림 9-12] 가변 파티션 멀티프로그래밍에서 메모리 '홀'

가변 파티션 멀티프로그래밍

- 외부 단편화를 줄이기 위한 조치
  - 병합(coalescing)
    - 인접한 홀을 병합해 한 큰 홀 생성
    - 메인 메모리 전반에 걸쳐 흩어진 홀들
  - 메모리 압축(memory compaction)
    - 메모리 트림시키기(burping the memory) 혹은 가비지 컬렉션(garbage collection)
    - 메모리 중 사용되는 모든 공간은 메인 메모리의 끝으로 재배치
    - 오버헤드 발생

가변 파티션 멀티프로그래밍

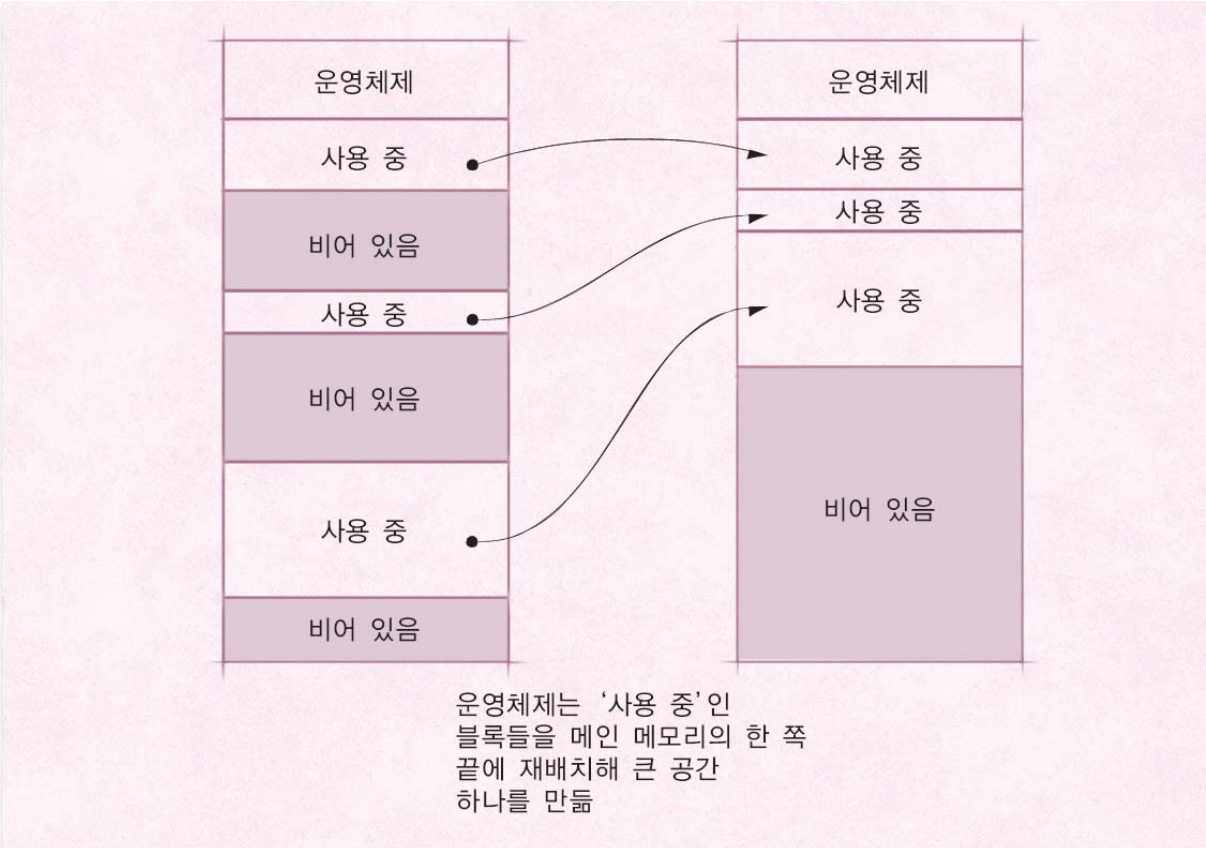
- 병합(coalescing)



[그림 9-13] 가변 파티션 멀티프로그래밍에서 메모리 '홀' 병합

가변 파티션 멀티프로그래밍

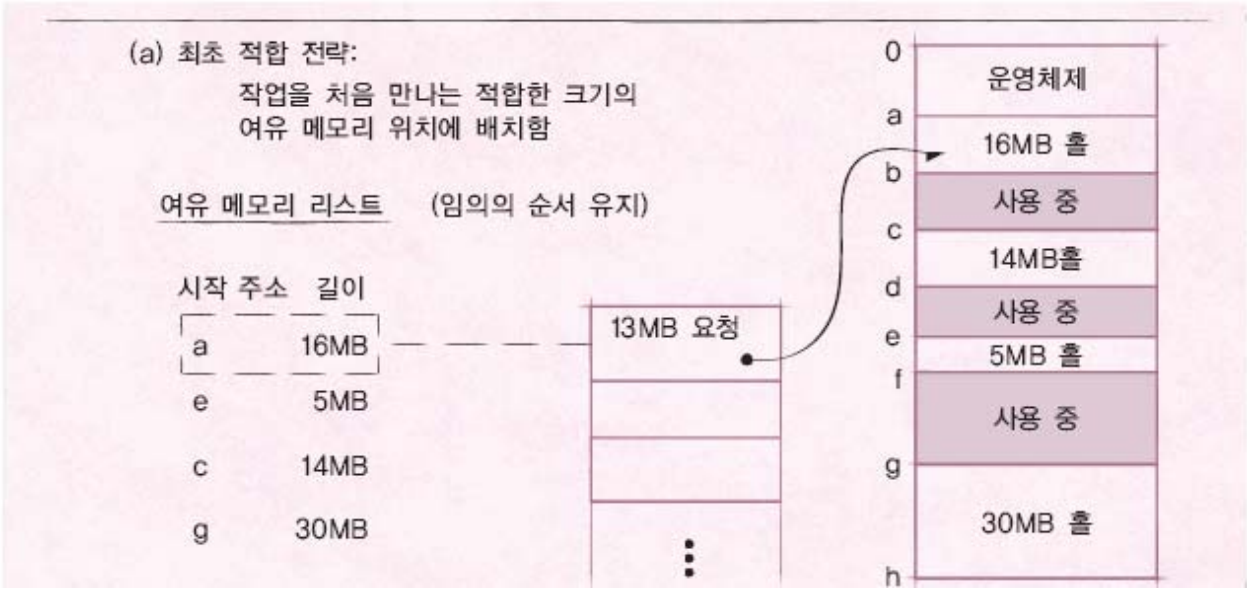
- 메모리 압축(memory compaction)



[그림 9-14] 가변 파티션 멀티프로그래밍에서 메모리 압축

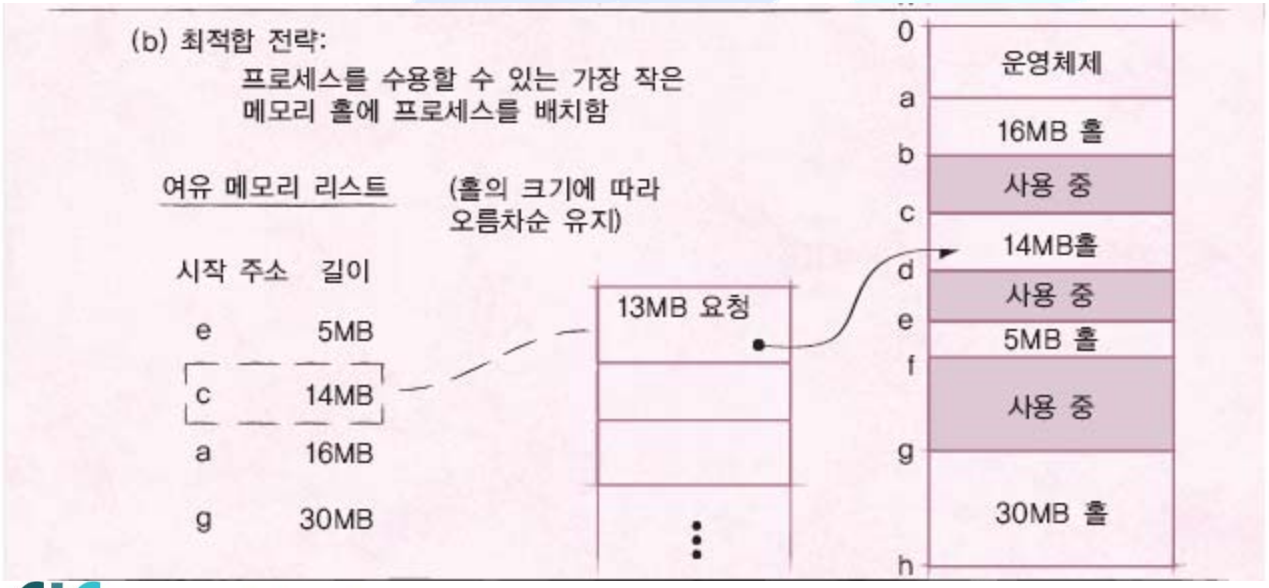
가변 파티션 멀티프로그래밍

- 메모리 배치 전략
  - 프로세스를 어떤 메모리 홀에 배치
  - 최초 적합 전략(first-fit strategy)
    - 처음 발견된 넉넉한 공간에 배치
    - 배치할 위치를 빨리 찾을 수 있음(검색은 빠르나 공간 활용률이 떨어짐)



가변 파티션 멀티프로그래밍

- 최적합 전략(best-fit strategy)
  - 가장 잘 맞는 부분에 배치
    - 프로세스가 들어갈 수 있는 충분히 큰 사용가능공간 중에서 가장 작은 크기의 사용 공간에 작업을 할당함
  - 사용하지 못할 작은 홀들을 최소화
  - 모든 여유 홀 조사
    - 사용가능공간에 대한 지속적인 정렬과정이 필요하여 비효율적임
  - 사용하지 않는 작은 홀 생성



가변 파티션 멀티프로그래밍

- 최악 적합 전략(worst-fit strategy)
  - 가능한 큰 공간에 배치(작업을 가장 큰 사용가능공간에 할당함)
  - 오버헤드
    - 공간이 크기 순서로 정렬되어 있지 않을 시 전 리스트를 검색해야 함
  - 사용하지 않는 작은 홀 생성
  - 최적합보다 메모리 활용면에서 더 유용함