메모리
관리

메모리 구역 나누기

메모리 구역 나누기

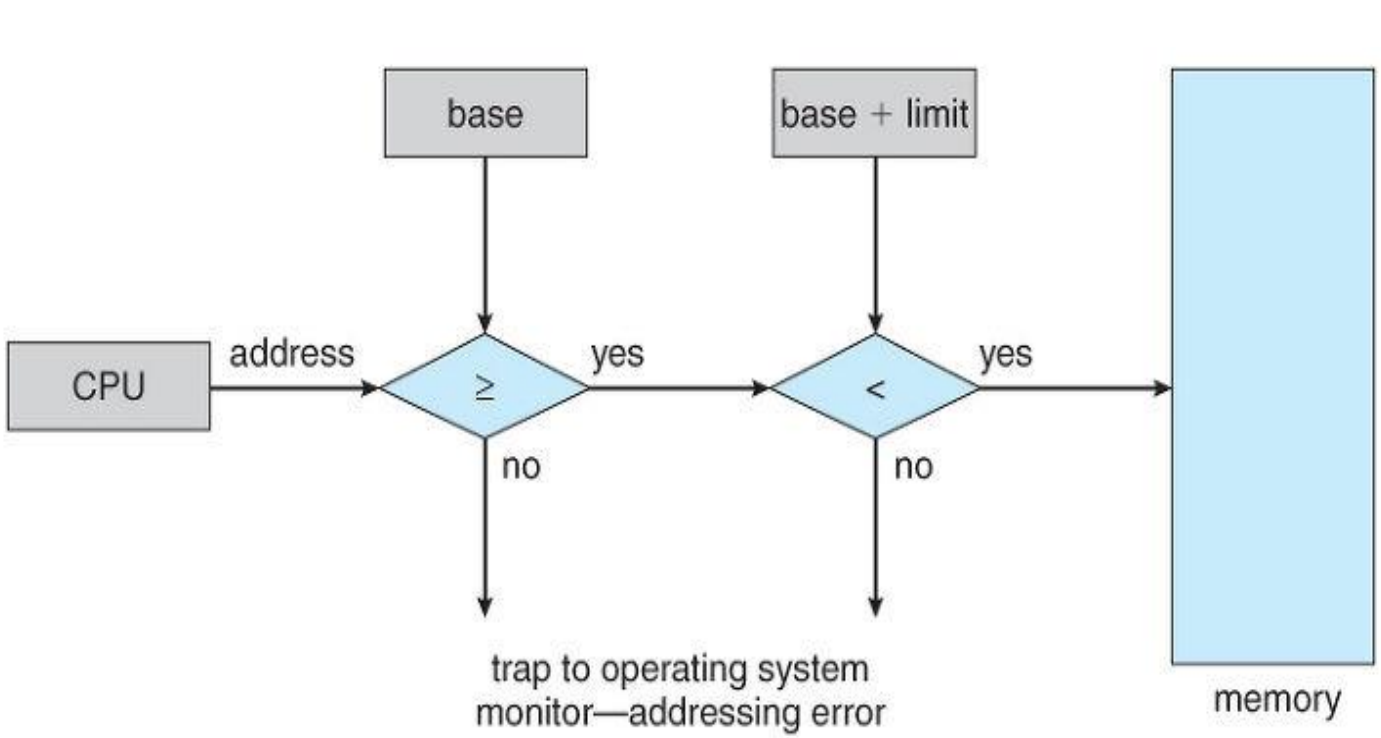
여러가지 프로세스가 실행되는
환경에서 다른 프로세스의 메모리
공간을 침범하지 않도록 해야함.

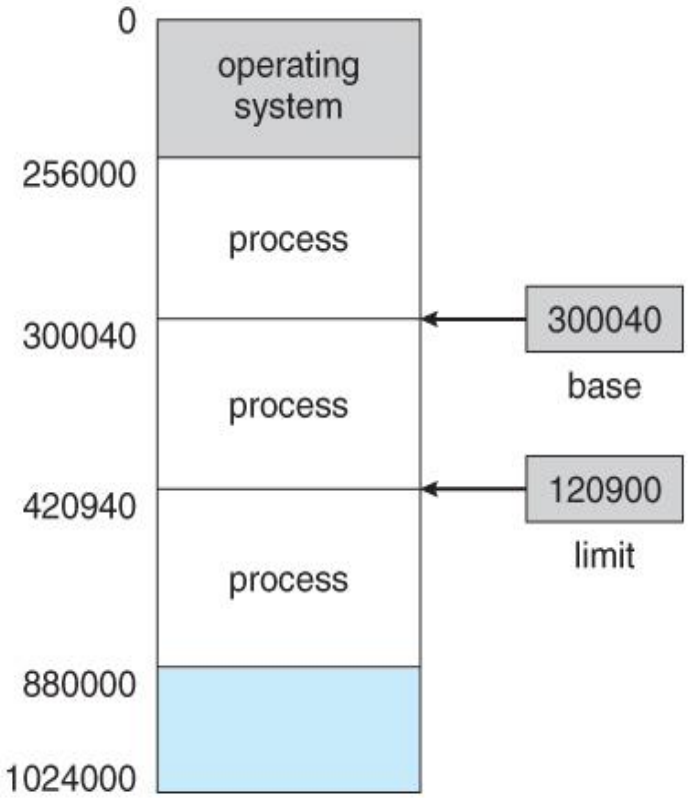
보통 속도를 위해 하드웨어적으로
구현함

-> base 레지스터, limit 레지스터 사용
(운영체제에 의해서만 로드 가능)

다른 메모리 침범하면 운영체제에서
trap을 발생하도록 함

메모리 구역 나누기


base
base + limit
CPU
address
≥
yes
<
yes
no
no
trap to operating system
monitor—addressing error
memory


0
operating system
256000
process
300040
process
420940
process
880000
1024000
300040
base
120900
limit

가상주소를 실제주소로 바꾸기

가상주소를 실제주소로 바꾸기

- 컴파일 시 컴파일러는 변수들을 재할당 가능한 주소로 바꾼다

  ※ 재할당 가능한 주소 : 상대주소

- 링크가능한 에디터는 이 상대주소를 절대주소로 바꾼다.
- “바인딩 " 이라고 함

가상주소를 실제주소로 바꾸기
- 바인딩 하는 시간 -

컴파일 시간에 주소 계산

- - 컴파일 시간에 프로세스가 어디 메모리에 있는지 알면, alsolute code 생성
- - 메모리 주소 하드 코딩
- - 시작주소가 바뀌면 컴파일 다시 해야함

가상주소를 실제주소로 바꾸기
- 바인딩 하는 시간 -

로드타임에 시간 계산

- 컴파일 시간에 프로세스가 어디 메모리에 있을지 모르면
  - -> relocatable code 생성
- 바인딩을 로드시간까지 미룸
- 시작주소 바뀌면 re-loading

가상주소를 실제주소로 바꾸기
- 바인딩 하는 시간 -

실행시간

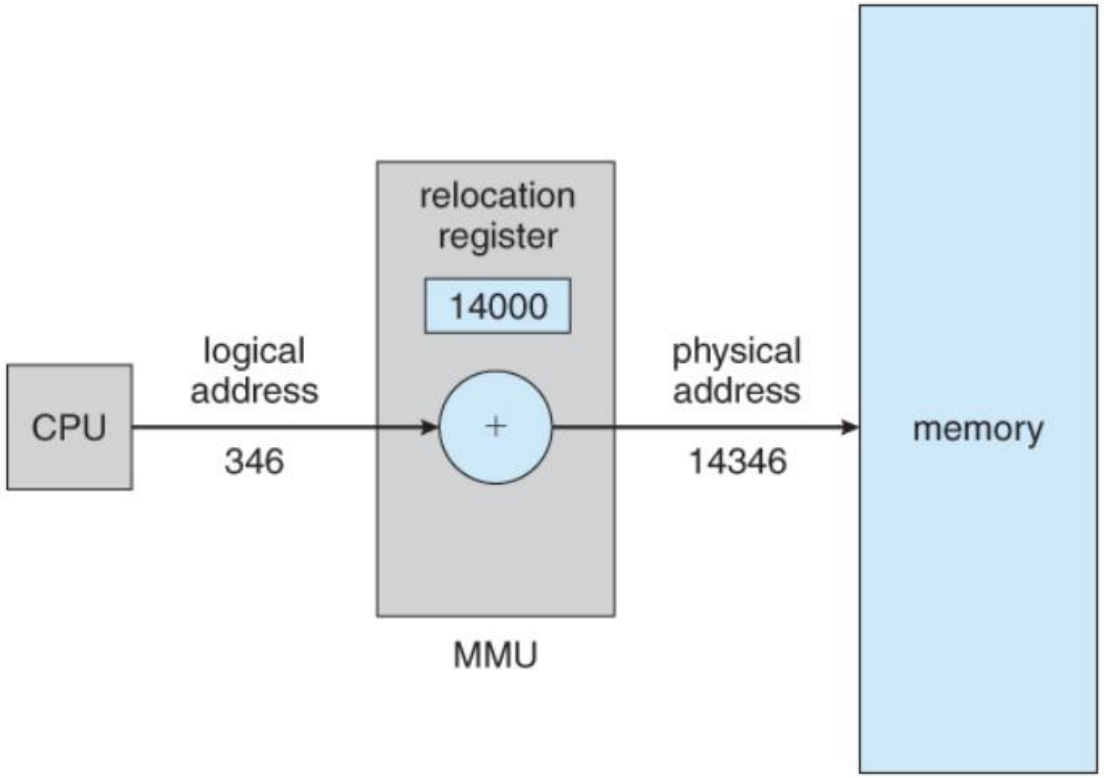
- 프로세스가 다른 메모리 세그먼트로 갈 수 있으면 실행할 때 까지 바인딩 미룸
- STORE 같은 명령어 실행 할 때마다 메모리주소 새로 결정
- base / limit 레지스터의 도움 필요

논리 메모리주소 vs 물리 메모리 주소

메모리주소 vs 물리 메모리 주소

- 논리 메모리 주소(virtual address)는 CPU에 의해 생성됨
- 물리 메모리 주소는 메모리 유닛에 의해 보이는 주소, 메모리 주소 레지스터에 로드되는 주소
- MMU는 가상메모리주소 <-> 실제메모리 주소 매핑시킴

메모리주소 vs 물리 메모리 주소



- logical 주소 == 실제주소 - 컴파일시간, loadtime 바인딩
- logical 주소 != 실제주소 - 런타임(MMU가 메모리 주소 결정)
- **유저는 실제메모리 주소를 볼 수 없다.**

다이나믹 로딩

다이나믹 로딩

- 프로그램이 실행되기 위해서는 메모리에 적재되어 있는 상태여야 함.
- 프로그램이 너무 커서 메모리에 모두 올라가지 못하면?

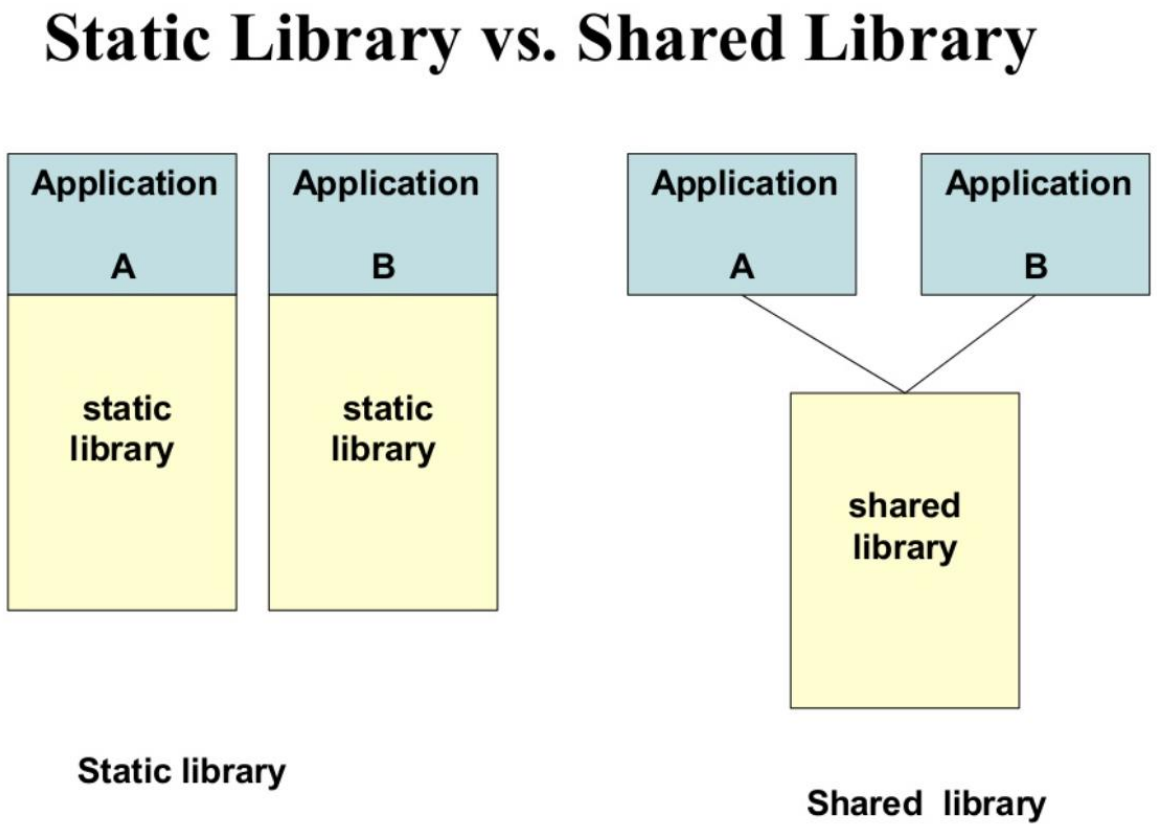
-> 프로그램 일부만 메모리에 올림, 실행되지 않은 루틴은 메모리에 올리지 않음

다이나믹 링킹 & 공유 라이브러리

다이나믹 링크 라이브러리?

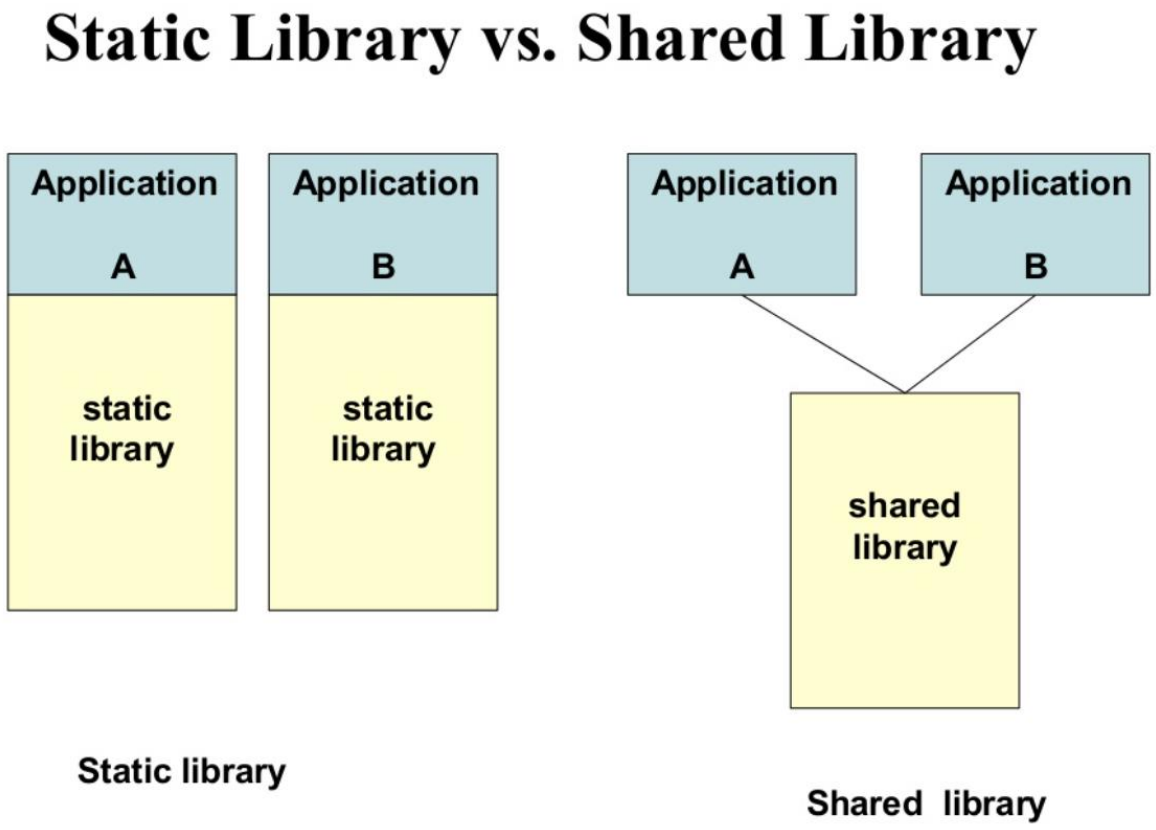
- 유저 프로그램이 실행될 때 링킹 되는 시스템 라이브러리이다.
- 리눅스 : *.so, 윈도우 : *.dll

스태틱 링킹



- 스태틱 링킹 != 다이나믹 링킹
- os가 라이브러리 obj로 취급하여 로드 할 때 프로세스 이미지와 합침, 중복된 코드 들어감, 메모리 낭비

다이나믹 링킹

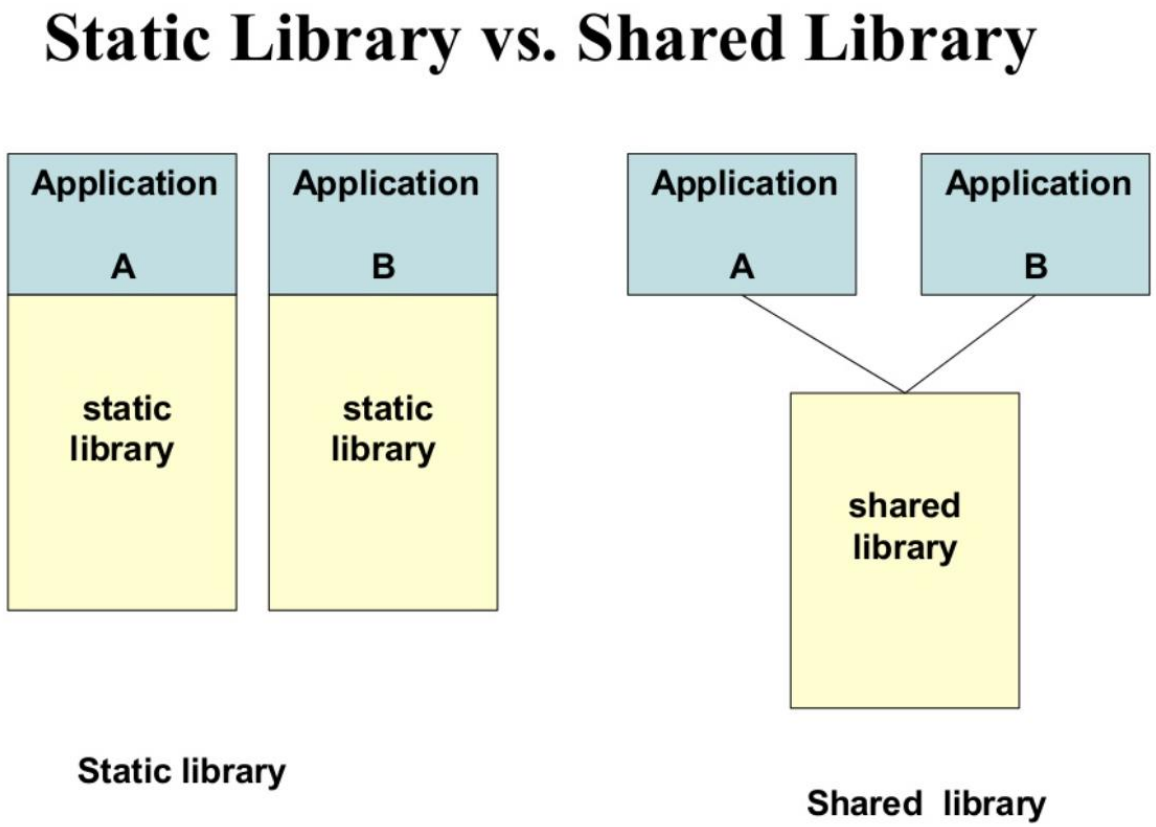


- 시스템 라이브러리를 메모리에 딱 한번만 올리고 다른 프로세스들이 공유함, 링킹이 실행시간 까지 연기됨
- 다이나믹 링킹에서는 stub이 각 프로세스 이미지에 들어감
- stub은 라이브러리를 어떻게 메모리에 위치시킬지,

로드 안되었으면 어떻게 로드 할 지 알려줌

- 라이브러리가 업데이트 되었다면 라이브러리를 프로세스들은 단지 새 라이브러리를 "참조"하기만 하면 됨

다이나믹 링킹 단점

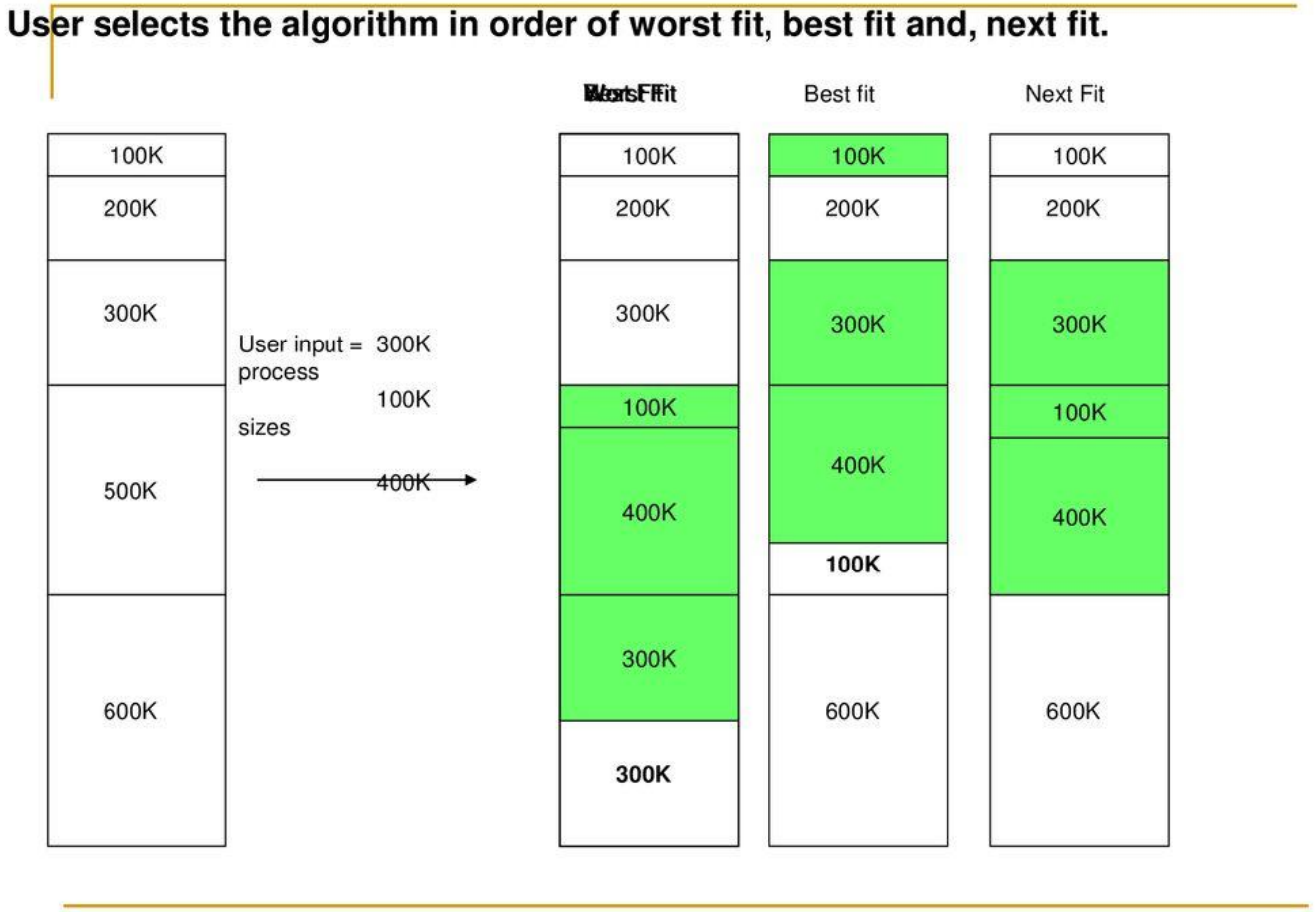


- 비표준 라이브러리를 제공하지 않는 os라면 실행이 안 될 수도 있음
- 운영체제에서 이 다이나믹 링킹을 지원해야 함
- 메모리 보호때문에 라이브러리 접근이 안 될 때가 있음

메모리 할당

메모리 할당

1. First Fit
2. Best Fit
3. Worst Fit



1. **First Fit**
   - 가장 먼저 발견한 필요한 메모리보다 큰 공간 할당
2. **Best Fit**
   - 충분히 큰 메모리 중 가장 작은 것, 메모리 전체 탐색함
3. **Worst Fit**
   - 가장 메모리에서 가장 큰 공간 할당, 메모리 전체 탐색함

단편화 문제

단편화 문제

First Fit, Best Fit 만족하는 메모리 할당 정책의 문제

- 외부단편화(메모리가 충분해도 메모리 할당이 불가능) 문제 있음
- 내부 단편화 문제도 있음.

단편화 문제

- 외부단편화를 해결하기 위해 compaction(메모리를 한곳으로 쭉 미는 작업) 수행
- 항상 compaction이 가능한 것은 아님, 변수가 static이거나 로딩 중일 때는 불가능
- 그리고 미는 데 걸리는 cost 가 비쌈(시간 오래 걸림)

외부단편화를 해결하기 위한 방법

1. compaction(메모리를 한곳으로 쭉 미는 작업)
   - 항상 compaction이 가능한 것은 아님, 변수가 static이거나 로딩 중일 때는 불가능
   - 그리고 미는 데 걸리는 cost 가 비쌈(시간 오래 걸림)
2. 논리주소공간을 비연속적이게 함 (세그멘테이션, 페이징)
   - 빈 메모리가 어디에 있든지 할당 가능