3주차_메모리 관리1

상태	진행 중
강의	CS 스터디
작성일	@2024년 2월 15일

메모리 관리

Logical vs. Physical Address

- Logical address (= virtual address) : 논리적 주소
  - 프로세스마다 독립적으로 가지는 주소 공간
  - 각 프로세스마다 0번지부터 시작
  - CPU가 보는 주소는 logical address임
    → 메모리 접근을 할때마가 주소 변환을 해서 봐야함

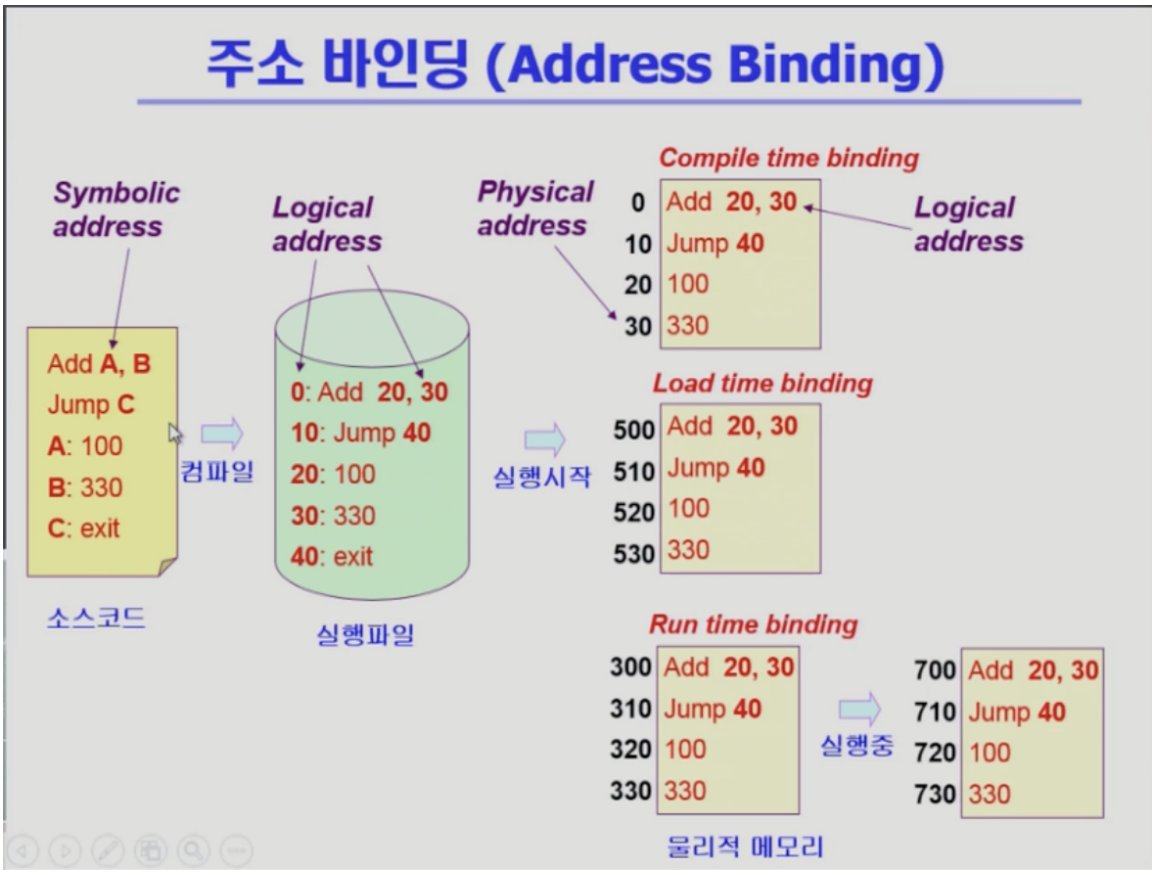
- Physical address : 물리적 주소
  - 메모리에 실제 올라가는 위치
- 주소 바인딩 : 주소를 결정하는 것
  Symbolic Address → Logical Address —(이 시점이 언제인가? (next page))→ Physical address

주소 바인딩(Address Binding)

- Compile time binding
  - 물리적 메모리 주소(physical address)가 컴파일 시 알려짐
  - 시작 위치 변경시 재컴파일
  - 컴파일러는 절대 코드(absolute code) 생성
- Load time binding
  - Loader의 책임하에 물리적 메모리 주소 부여
  - 컴파일러가 재배치가능코드(relocatable code)를 생성한 경우 가능

- Execution time binding(=Run time binding)
  - 수행이 시작된 이후에도 프로세스의 메모리 상 위치를 옮길 수 있음
  - CPU가 주소를 참조할 때마다 binding을 점검 (address mapping table)
    CPU는 논리적 주소를 보기 때문
  - 하드웨어적인 지원이 필요

    (e.g., base and limit registers, MMU).



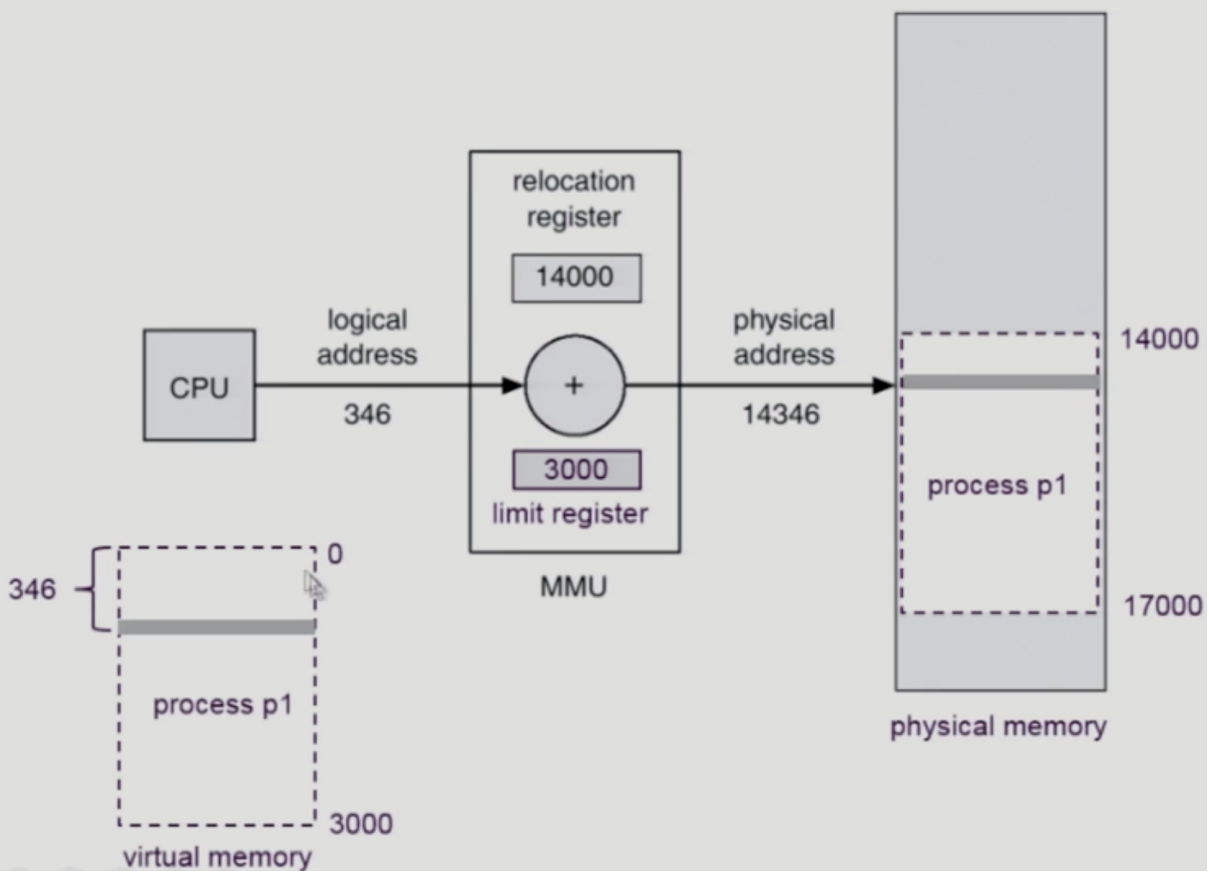
  - 물리적 주소는 0번지부터 올라가야한다.
  - Load time binding와 Run time binding은 실행 시에 물리적인 주소가 부여됨
    Load time binding → 메모리상 위치 변경 불가
    Run time binding → 메모리상 위치 변경 가능

Memory-Management Unit(MMU)

- MMU(Memory-Management Unit)
  - Logical address를 physical address로 매핑해 주는 Hardware device
- MMU scheme

    - 사용자 프로세스가 CPU에 수행되며 생성해내는 모든 주소값에 대해 base register(=relocation register)의 값을 더한다.
- user program
    - logical address만을 다룬다.
    - 실제 physical address를 볼 수 없으며 알 필요가 없다.

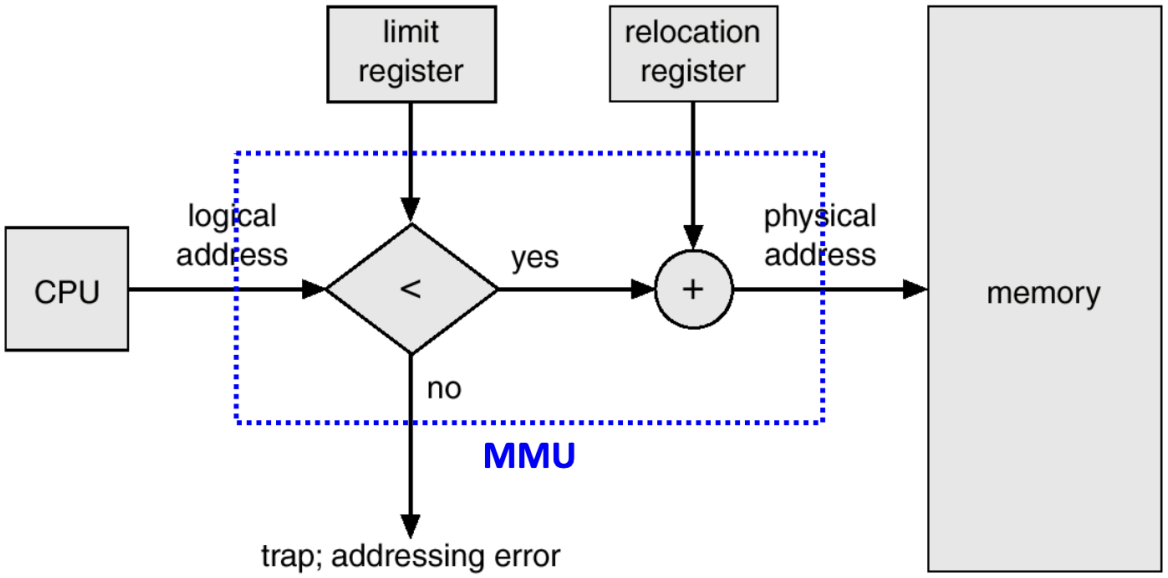
Dynamic Relocation



relocation register는 물리적 메모리의 시작점 위치를 담고있음

limit register는 프로그램의 길이를 담고 있어 → 본인의 주소공간이 아닌 곳을 요청한다면 악의적인 시도일 수 있으니 사전에 차단할 수 있음

Hardware Support for Address Translation



운영체제 및 사용자 프로세스 간의 메모리 보호를 위해 사용하는 레지스터

- Relocation register : 접근할 수 있는 물리적 메모리 주소의 최소값
  (=base register)
- Limit register: 논리적 주소의 범위

Dynamic Loading

- 프로세스 전체를 메모리에 미리 다 올리는 것이 아니라 해당 루틴이 불려질 때 메모리에 load하는 것
- memory utilization의 향상
- 가끔씩 사용되는 많은 양의 코드의 경우 유용
  - 예: 오류 처리 루틴
- 운영체제의 특별한 지원 없이 프로그램 자체에서 구현 가능(OS는 라이브러리를 통해 지원 가능)
  현대적인 운영체제에서는 Dynamic Loading 지원 되기도 함
- Loading: 메모리로 올리는 것

Overlays

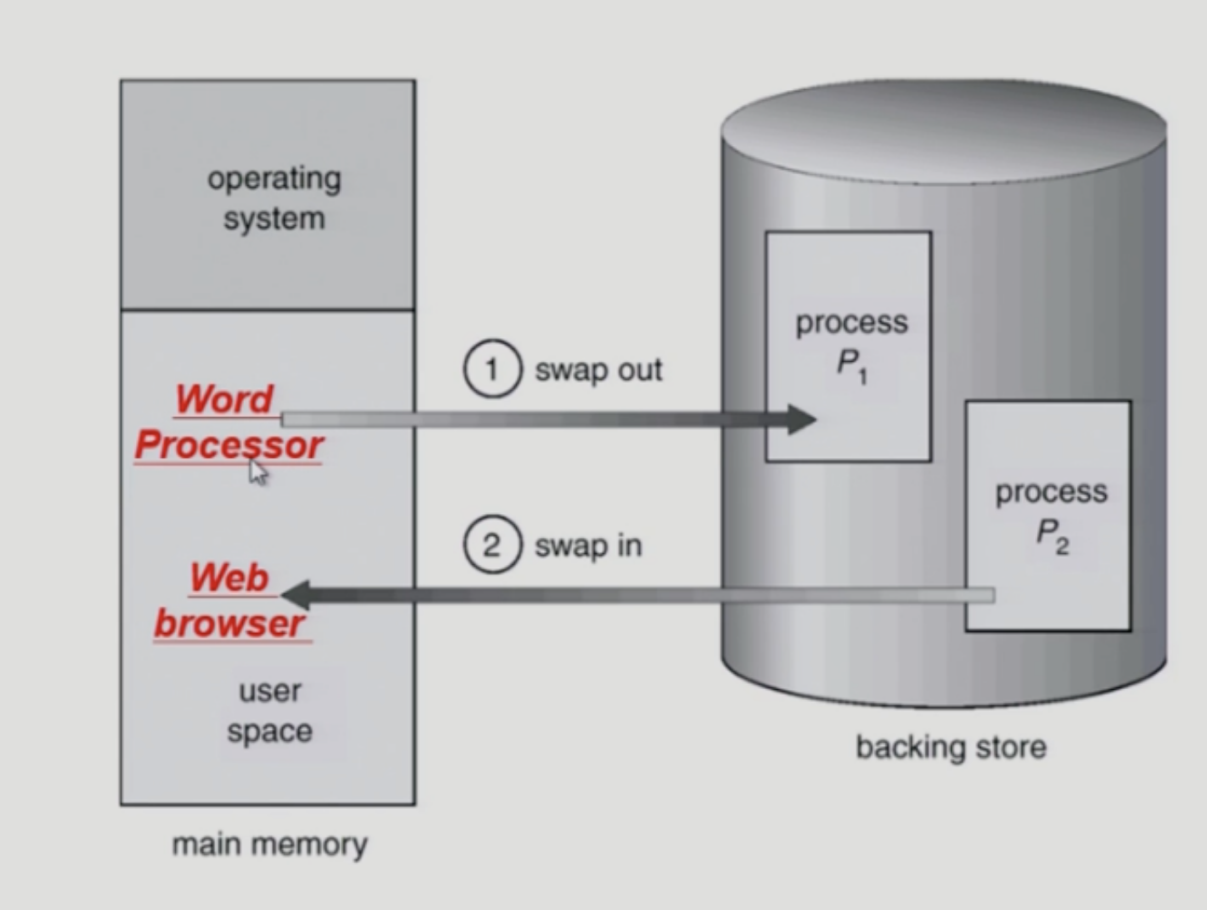
- 메모리에 프로세스의 부분 중 실제 필요한 정보만을 올림

- 프로세스의 크기가 메모리보다 클 때 유요
- 운영체제의 지원 없이 사용자에 의해 구현
- 작은 공간의 메모리를 사용하던 초창기 시스템에서 수작업으로 프로그래머가 구축
    - “Manual Overlay”
    - 프로그래밍이 매우 복잡

Swapping

- swapping (여기서는 통째로 쫓겨나는 것을 말하고 있음)
    - 프로세스를 일시적으로 메모리에서 backing store로 쫓아내는 것
- Backing store (=swap area)
    - 디스크
        - 많은 사용자의 프로세스 이미지를 담을 만큼 충분히 빠르고 큰 저장 공간
- Swap in / Swap out
    - 일반적으로 중기 스케줄러(swapper)에 의해 swap out 시킬 프로세스 선정
    - priority-based CPU scheduling algorithm
        - priority가 낮은 프로세스를 swapped out 시킴
        - priority가 높은 프로세스를 메모리에 올려 놓음
    - Compile time 혹은 load time binding에서는 원래 메모리 위치로 swap in 해야 함
    - Execution time binding레서는 추후 빈 메모리 영역 아무 곳에나 올릴 수 있음
    - swap time은 대부분 transfer time (swap되는 양에 비례하는 시간)임

Schematic View of Swapping



Dynamic Linking

- Linking을 실행 시간(execution time)까지 미루는 기법
- Static linking (← static library)
  - 라이브러리가 프로그램의 실행 파일 코드에 포함됨
  - 실행 파일의 크기가 커짐
  - 동일한 라이브러리를 각각의 프로세스가 메모리에 올리므로 메모리 낭비
    (e.g. Printf 함수의 라이브러리 코드)
- Dynamic linking (← shared library) .so .dll
  - 라이브러리가 실행시 연결(link)됨
  - 라이브러리 호출 부분에 라이브러리 루틴의 위치를 찾기 위한 stub이라는 작은 코드를 둠
  - 라이브러리가 이미 메모리에 있으면 그 루틴의 주소로 가고 없으면 디스크에서 읽어옴

- 운영체제의 도움이 필요

Allocation of Physical Memory

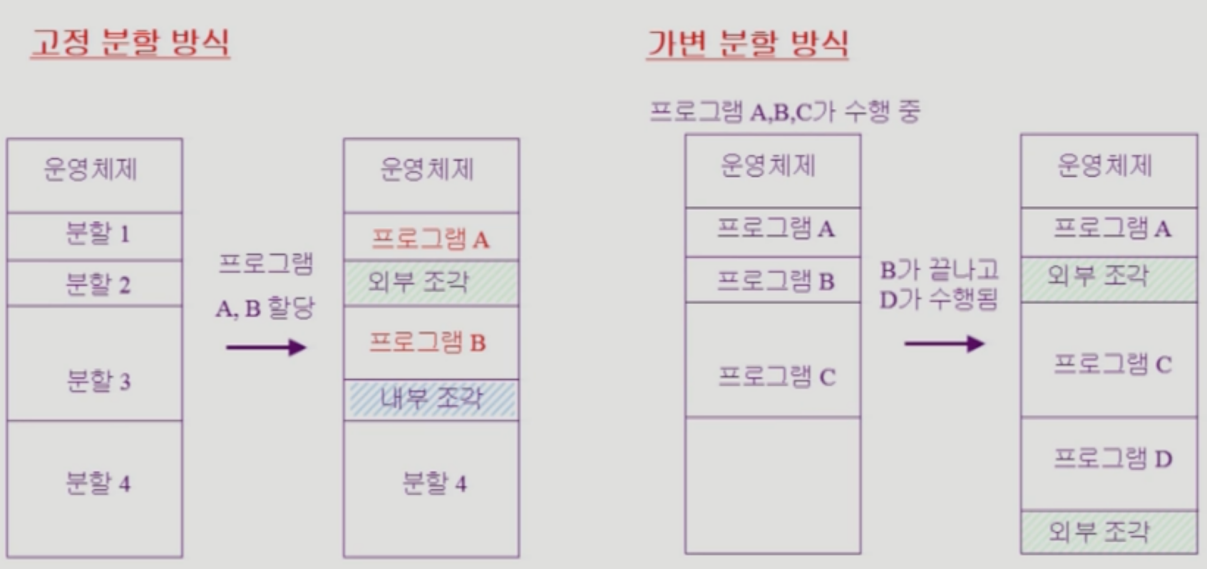
- 메모리는 일반적으로 두 영역으로 나뉘어 사용
  - OS 상주 영역
    - interrupt vector와 함께 낮은 주소 영역 사용
  - 사용자 프로세스 영역
    - 높은 주소 영역 사용
- 사용자 프로세스 영역의 할당 방법
  - Contiguous Allocation (연속 할당)

    :각각의 프로세스가 메모리의 연속적인 공간에 적재되도록 하는 것
    - Fixed partition allocation
    - Variable partition allocation
  - Noncontiguous allocation (불연속 할당)
    : 하나의 프로세스가 메모리의 여러 영역에 분산되어 올라갈 수 있음
    - Paging
    - Segmentation
    - Paged Segmentation

Continuous Allocation

- 고정분할 (Fixed partition) 방식
  - 물리적 메모리를 몇개의 영구적 분할(partition)로 나눔
  - 분할의 크기가 모두 동일한 방식과 서로 다른 방식이 존재
  - 분할당 하나의 프로그램 적재
  - 융통성이 없음
    - 동시에 메모리에 load되는 프로그램의 수가 고정됨
    - 최대 수행 가능 프로그램 크기 제한

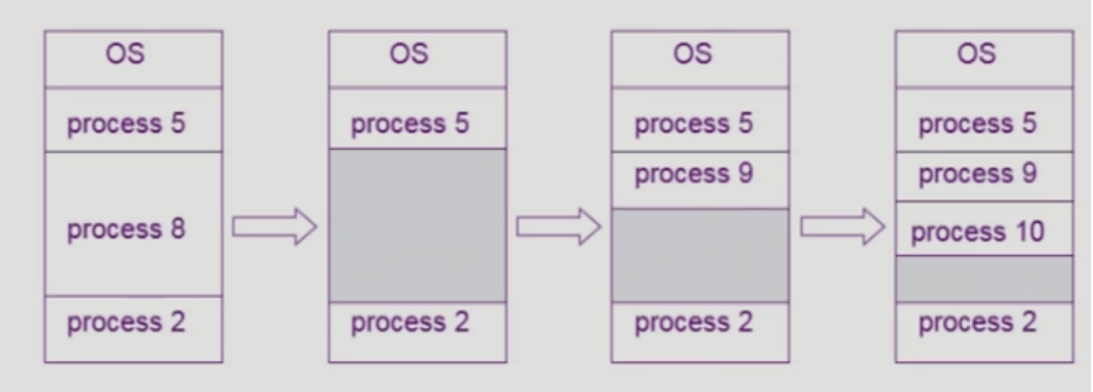
    - Internal fragmentation(내부 조각) 발생 (external fragmentation(외부 조각)도 발생)
- 가변분할 (Variable partition) 방식
    - 프로그램의 크기를 고려해서 할당
    - 분할의 크기, 개수가 동적으로 변함
    - 기술적 관리 기법 필요
    - External fragmentation 발생 (내부 조각은 발생 하지 않음)



- External fragmentation(외부조각)
    - 프로그램 크기보다 분할의 크기가 작은 경우
    - 아무 프로그램에도 배정되지 않은 빈 곳인데도 프로그램이 올라갈 수 없는 작은 분할
- Internal fragmentation(내부조각)
    - 프로그램 크기보다 분할의 크기가 큰 경우
    - 하나의 분할 내부에서 발생하는 사용되지 않는 메모리 조각
    - 특정 프로그램에 배정되었지만 사용되지 않는 공간

- Hole
    - 가용 메모리 공간
    - 다양한 크기의 hole들이 메모리 여러 곳에 흩어져 있음

- 프로세스가 도착하면 수용가능한 hole을 할당
- 운영체제는 다음의 정보를 유지
  a) 할당 공간 b) 가용 공간 (hole)



- Dynamic Storage-Allocation Problem
  : 가변 분할 방식에서 size n인 요청을 만족하는 가장 적절한 hole을 찾는 문제
  - First-fit
    - Size가 n 이상인 것 중 최초로 찾아지는 hole에 할당
  - Best-fit
    - Size가 n 이상이 가장 작은 hole을 찾아서 할당
    - Hole들의 리스트가 크기순으로 정렬되지 않은 경우 모든 hole의 리스트를 탐색해야 함
    - 많은 수의 아주 작은 hole들이 생성됨
  - Worst-fit
    - 가장 큰 hole에 할당
    - 역시 모든 리스트를 탐색해야 함
    - 상대적으로 아주 큰 hole들이 생성됨
  - First-fit과 best-fit이 worst-fit보다 속도와 공간 이용률 측면에서 효과적인 것으로 알려짐 (실험적인 결과)

- compaction (Run time binding에서만 가능)

- external fragmentation 문제를 해결하는 한 가지 방법
- 사용중인 메모리 영역을 한군데로 몰고 hole들을 다른 한 곳으로 몰아 큰 block을 만드는 것
- 매우 비용이 많이 드는 방법임
- 최소한의 메모리 이동으로 compaction하는 방법 (매우 복잡한 문제)
- Compaction은 프로세스의 주소가 실행 시간에 동적으로 재배치 가능한 경우에만 수행될 수 있다.