프로그래머가 알아야 하는 메모리 관리 기법

Sunny Kwak

(sunnykwak@daum.net)

목 차

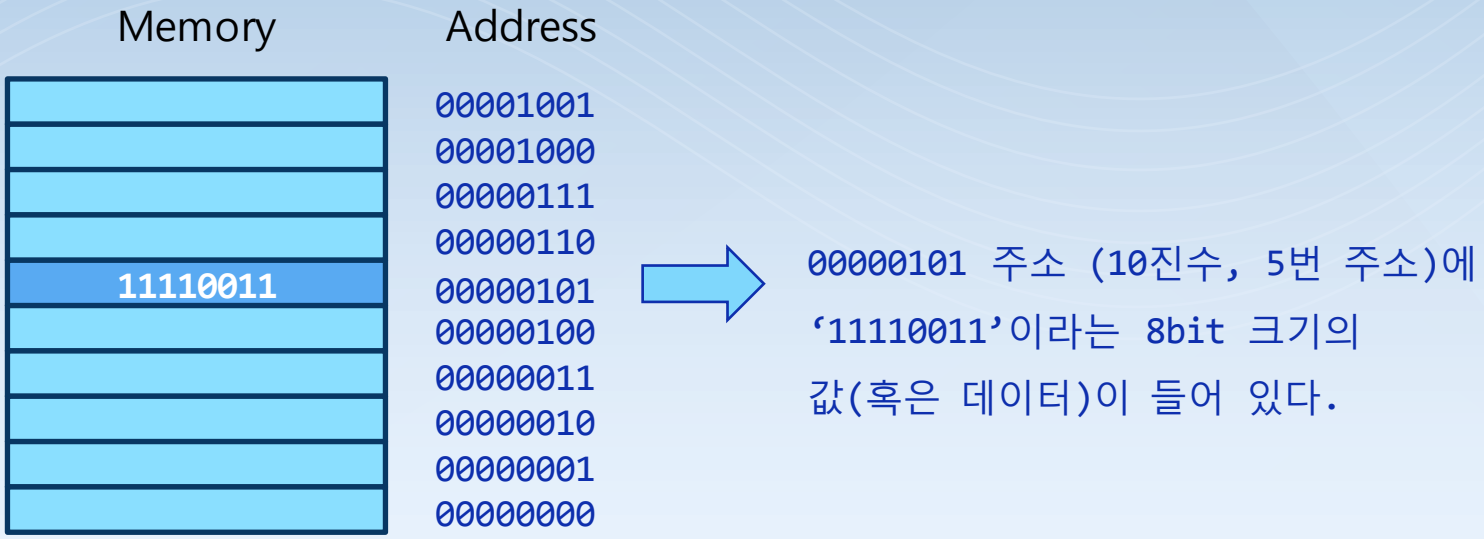
주소 지정 개념 이해

주소 지정(ADDRESSABILITY)

주소 지정

- **메모리 위치(memory location) 및 메모리 주소(memory address)**
  - 모든 메모리 위치는 주소를 가져야 한다.
  - 일반적으로 각각의 메모리 위치는 8 bit의 데이터를 저장한다.
  - 각각의 주소는 특정 메모리 위치에 대한 고유한 식별자(identifire)이다.

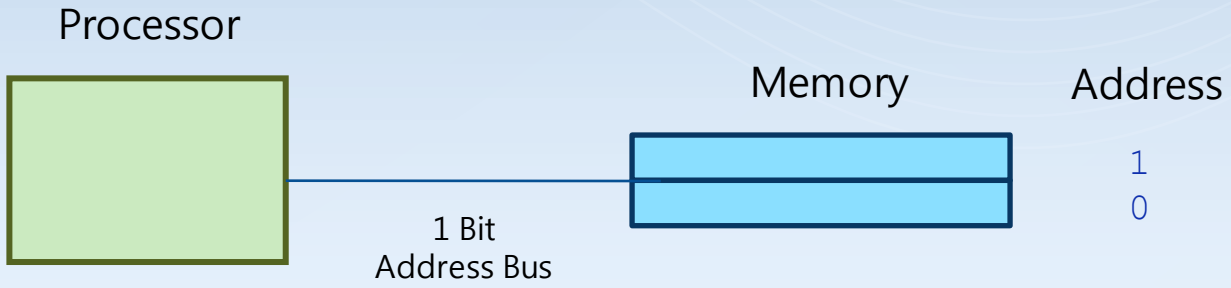
- **주소 지정 (Addressability)**
  - 컴퓨터가 메모리 위치를 식별하는 방식(the way)을 '주소 지정'이라 한다.



메모리 위치

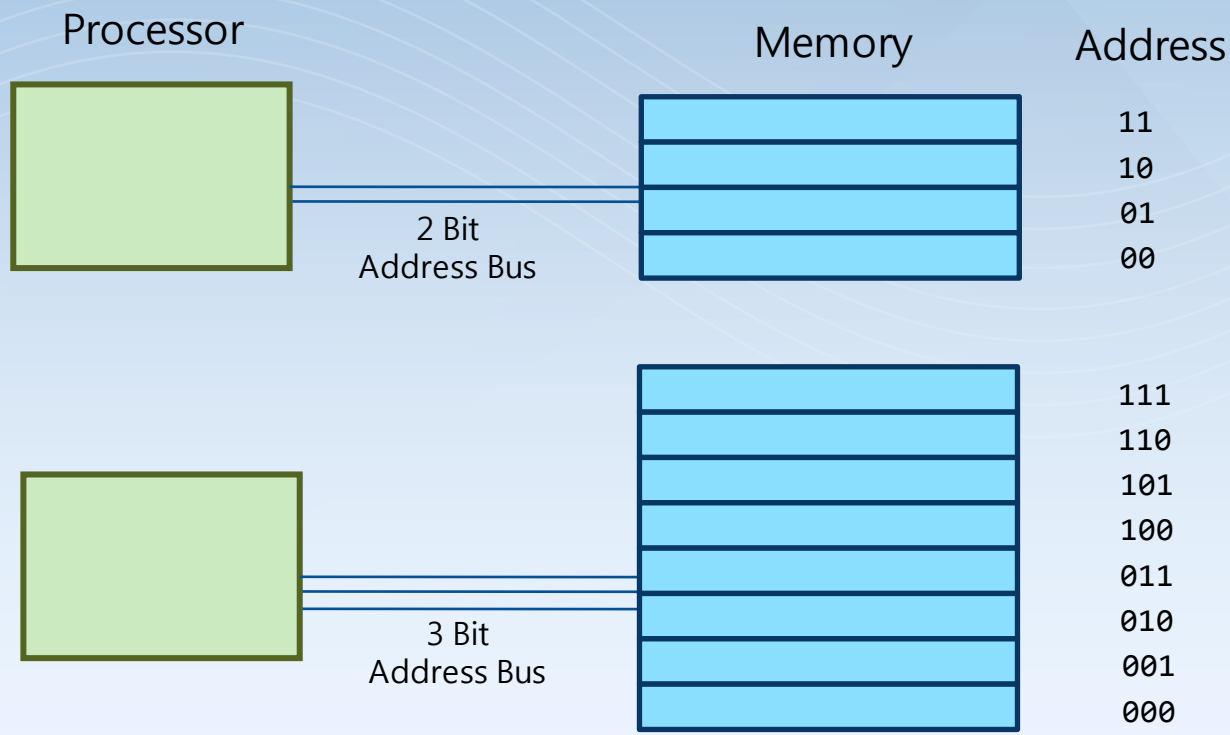
- **메모리 위치(memory location)와 주소 버스(address bus)**
  - 주소 버스(address bus)의 폭(width)이 주소 지정할 수 있는 메모리 위치의 최대 크기(갯수)를 결정한다.
  - 1 bit 의 주소 버스는 2개의 메모리 위치에 접근할 수 있다.

- **버스(bus)**
  - '버스'는 컴퓨터 용어로, 컴퓨터의 내부나 외부의 각 장치와의 정보나 신호를 주고 받는 데 사용되도록 구성된 전기적 통로를 말한다. (위키피디아 인용)
  - '주소 버스'는 주소 값을 주고 받을 수 있는 '신호 통로'를 의미하며, 1 bit 주소 버스로는 1 bit의 주소 값을 주고 받을 수 있다.



N bit 주소 버스

- **주소 버스 크기와 주소 지정**
  - 컴퓨터가 사용할 수 있는 '주소 지정' 가능한 범위는 '주소 버스'의 크기에 좌우된다.
  - 2 비트 주소 버스는 4개, 3 비트는 8개의 주소를 지정할 수 있다.
  - 일반적으로 32 bit 컴퓨터라는 말의 의미는 32 bit 주소 버스를 가진 (혹은 그런 구조로 설계된) 컴퓨터이다. 또한, 64 bit 컴퓨터라고 해서 물리적으로 64 bit address bus 를 사용하는 것은 아니다. 논리적 개념으로 이해해야 한다.

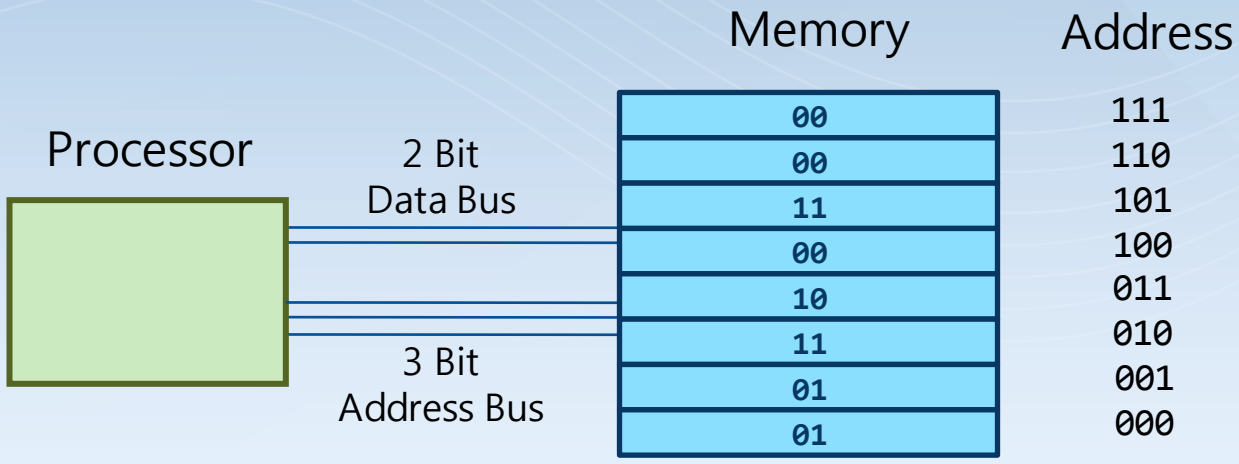


주소 지정 범위 = 2^n

- **8 bit 주소 버스의 주소 지정 범위**
  - 2^8 = 256 개의 메모리 위치
- **16 bit 주소 버스의 주소 지정 범위**
  - 2^{16} = 65,536 개의 메모리 위치
- **32 bit 주소 버스의 주소 지정 범위**
  - 2^{32} = 4,294,967,296 개의 메모리 위치 = 4 Giga Byte
- **64 bit 주소 버스의 주소 지정 범위**
  - 2^{64} = 18,446,744,073,709,551,616 개의 메모리 위치 = 대략 많다...!!!

사용 가능한 메모리 용량

- **사용 가능한 메모리 용량 계산**
  - 컴퓨터가 3 bit의 주소 버스를 가지고 있다면, 8개의 메모리 위치를 사용할 수 있다. 그런데, 얼마나 많은 메모리를 사용할 수 있을까?
  - 질문에 대답하기 위해서는, 각각의 메모리 위치에 얼마나 많은 비트가 저장될 수 있는지를 알아야 하고, 메모리 위치에 저장되는 데이터의 크기는 '데이터 버스'의 크기에 의해 결정된다.
  - 만일 데이터 버스의 크기가 2 bit 라면, 사용 가능한 메모리 용량은 8 * 2 = 16 bit = 2 byte 가 된다.



☞ **address bus 와 data bus의 크기를 이용해 사용 가능한 메모리의 최대 용량을 산정하는 것은 논리적인 용량 계산으로 이해하는 것이 옳다. CPU 설계에 따라 물리적인 bus 설계가 다를 수 있기 때문이다.**

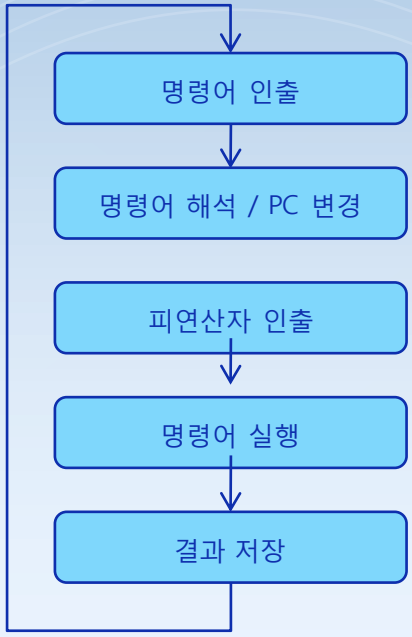
메인 메모리는 어떻게 관리되는가?

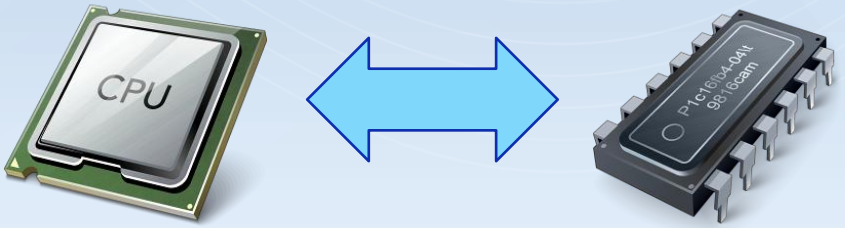
주 메모리(MAIN MEMORY)

배경 (Background)

- **메인 메모리**
  - 메모리는 각각 주소가 할당된 바이트 단위 데이터 공간으로 구성된다.

- **CPU의 동작 방식**
  - CPU는 PC(Program counter) 레지스터가 가리키는 메모리 위치에 있는 다음 명령어를 가져온다.
  - 그 이후 피연산자가 필요할 경우 메모리에서 또 가져오고 명령어를 실행한다.
  - 결과를 레지스터 혹은 메모리 저장하고, 다음 명령을 가져온다.





배경 (Background)

- **Main Memory, Register**
  - CPU가 직접 접근 가능한 데이터 저장 장치
  - 개별 레지스터는 고유 명칭과 용도가 지정되어 있는 단일 데이터 저장소
  - 주 메모리(main memory)는 복수의 데이터를 저장할 수 있으며, 주소 지정을 통해 개별 데이터를 읽고 쓸 수 있다.

- **실행(연산)을 위한 준비**
  - 모든 명령어와 자료들을 수행하기 위해선 CPU가 가져와야 한다.
  - 만약 보조기억장치나 다른 곳에 있다면 먼저 메모리로 옮긴 이후에 수행한다.

- 데이터 접근 속도
  - 레지스터는 일반적으로 CPU 1 clock cycle 만에 접근 가능하다.
  - 메모리는 일반적으로 CPU 2clock cycle 이상의 시간이 필요하다.
    - 버스를 통한 전송 시간, 정확한 위치 계산 등이 필요함.
    - 데이터를 가져오는 동한 CPU가 아무 일도 못하는 현상도 발생함.
    - 캐시 메모리(cache memory) 등을 이용해 처리 속도를 높이는 기법도 적용.

배경 (Background)

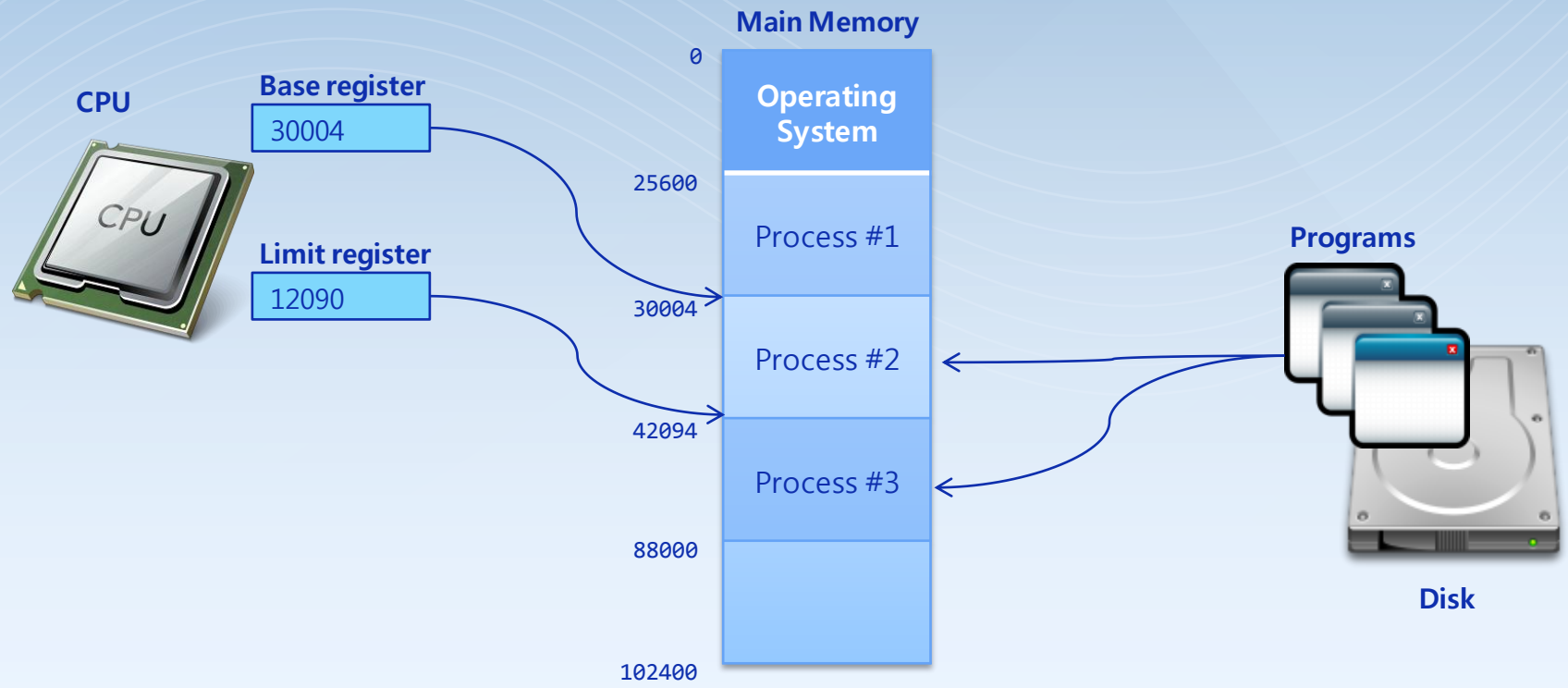
- **멀티 태스킹(multitasking)**
  - 엄밀히 말해 폰 노이만 방식의 컴퓨터에서 CPU가 한번에 처리할 수 있는 명령은 하나에 불과하다.
  - CPU가 짧은 시간에 여러 프로세스를 조금씩 나누어 작업을 마치 동시에 여러가지 작업을 수행하는 것처럼 보이는 것이 '다중 작업' 혹은 '멀티 태스킹'이다.
  - 멀티 태스킹의 효과
    - 각 프로세스(혹은 어플리케이션)의 특성에 따라 하나의 프로세스가 거의 작업을 하지 않을 때 다른 프로세스를 실행해 CPU의 실행 효율을 높일 수 있다.
    - 사용자에게 다양한 작업이 동시에 실행되는 것으로 느끼게끔 하여, 컴퓨터의 활용도를 높인다.
    - 다른 작업을 하던 중이라도 사용자의 이벤트(키보드/마우스 입력 등) 발생 시, 사용자의 반응을 처리하는 프로세스로 복귀하여 사용자가 체감하는 반응 속도를 높이는 효과가 있다.
      (멀티 태스킹을 안하면, 사용자는 특정 작업이 끝날 때까지 컴퓨터가 멈추는 것으로 느낄 수 있다.)

- **프로세스 별 메모리 공간 분리**
  - 멀티 태스킹을 수행하기 위해서는 메모리 상에 복수의 프로세스가 준비되고 함께 실행되어야 한다.
  - 운영 체제와 각 어플리케이션 프로세스는 분리된 공간에 적재(load) 된다.
  - Base register 와 limit register는 프로세스 주소 공간을 결정하는 데 사용된다.

프로그램, 메모리, CPU

- 실행 프로그램
  - 실행 프로그램은 '영구 저장장치'인 하드 디스크 등에 저장되어 있음.
  - 실행 시에 실행 프로그램의 코드와 데이터를 '메인 메모리'로 복사

- 메인 메모리와 CPU
  - 운영체제를 포함한 복수의 프로그램이 메인 메모리에 적재(load) 됨.
  - CPU는 여러 레지스터(register)를 이용해 실행 중인 명령 위치를 지정하고 프로그램을 실행.



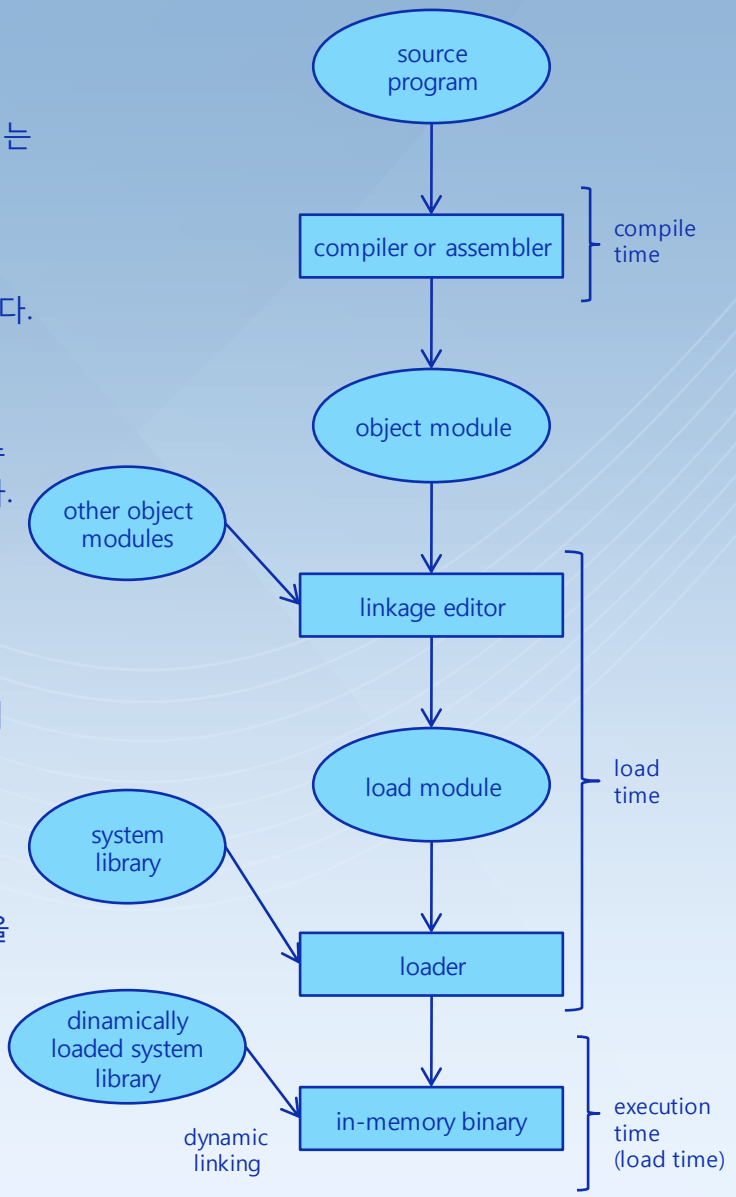
주소 지정 (Address Binding)

- **주소 결합 혹은 주소 지정**
  - 프로그램이 보조기억장치(혹은 보조저장장치)에 기록되어 있는 동안에는 메인 메모리에 어느 위치에 적재(load)될 지 알 수가 없다.
  - 따라서 프로그램이 메모리로 옮겨진 후에 명령과 데이터들의 주소가 확정(지정)되는 절차를 '주소 결합(address binding)'이라고 한다.
  - 컴파일 시점 지정, 적재 시점 지정, 실행 시점 지정 등 3가지 방식이 있다.

- **컴파일 시점 지정 (Compile Time Binding)**
  - 프로그램이 메모리의 어느 위치에 적재될 지 컴파일 시점에 미리 알 수 있다면 메모리 절대 주소(absolute memory address)를 지정할 수 있다.
  - 운영체제의 boot loader (운영체제 시작 프로그램)는 절대 주소 지정 방식을 사용한다.

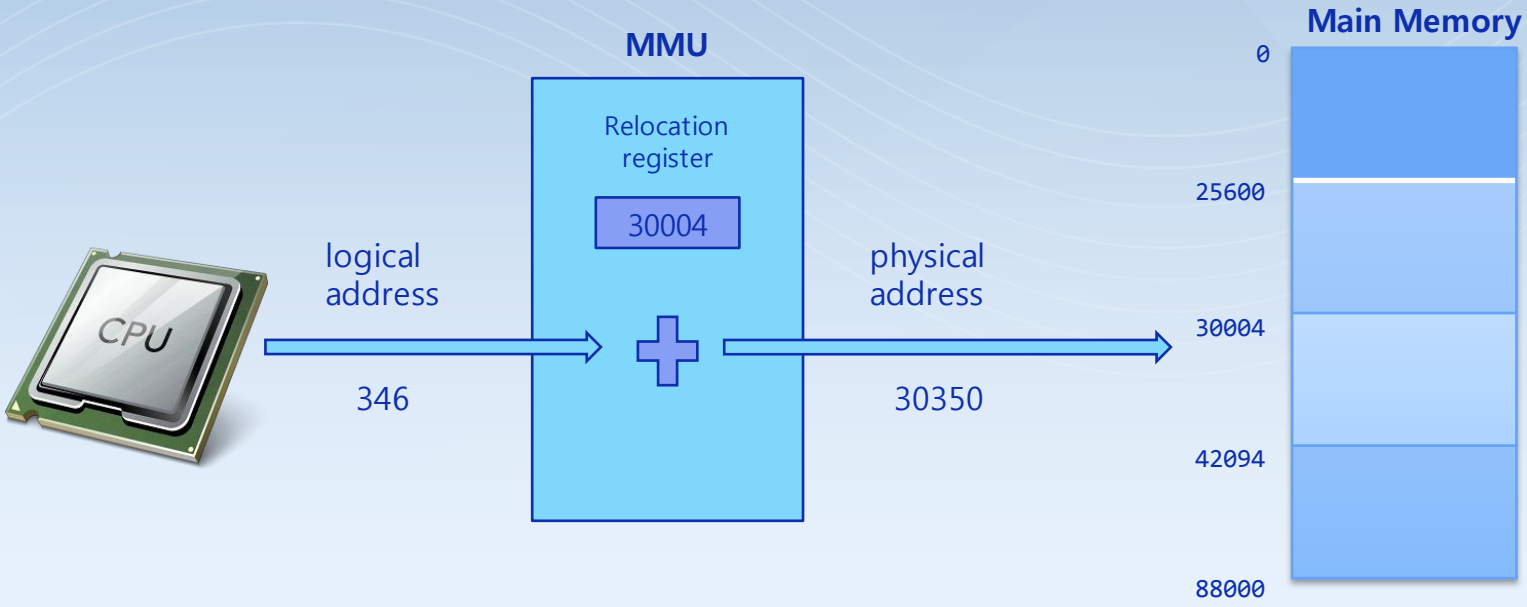
- **적재 시점 지정 (Load Time Binding)**
  - 메모리 적재 위치를 컴파일 시간에 알 수 없다면, 프로그램이 메모리에 적재되는 시점에 주소 재배치 코드를 실행한다.
  - 어플리케이션 프로그램(운영체제를 제외한)은 메인 메모리의 어느 위치에 적재될 지 알 수 없으므로 프로그램의 첫번째 명령을 기준으로 상대 주소(relative address)를 컴파일 시점에 설정한다.
  - 프로그램을 실행하면, 프로그램을 메모리에 적재하면서 절대 주소 값을 재계산한다.

- **실행 시점 지정 (Excution time Binding)**
  - 프로그램 실행 중에 메모리 주소가 변경되는 방식이다.
  - 하드웨어의 지원이 필요하다. (하드웨어가 주소를 변경해 주는 기능)



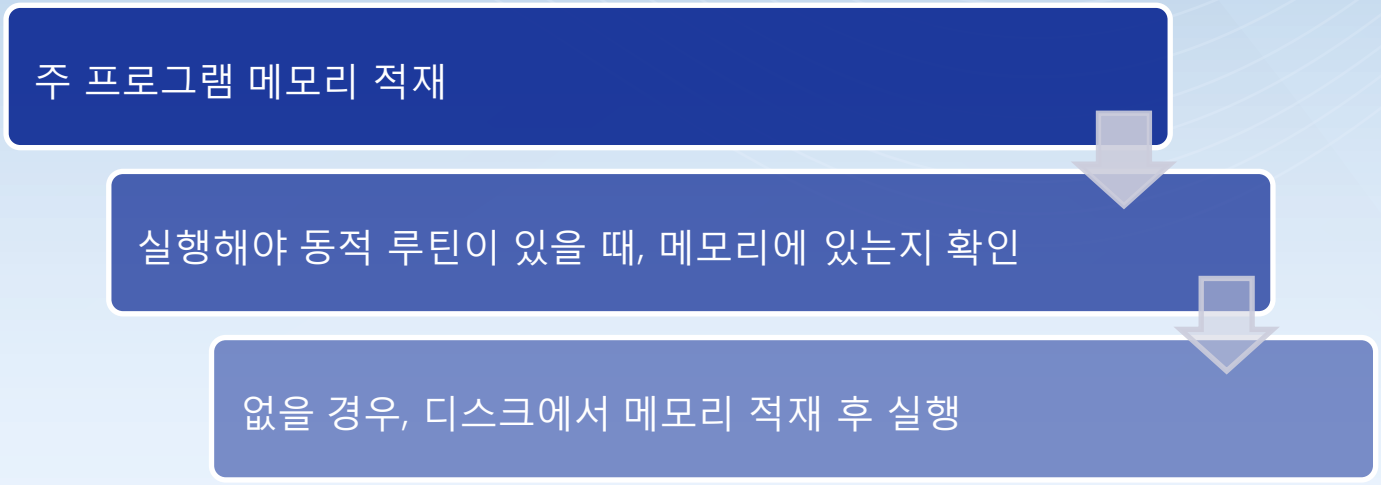
MMU (Memory Management Unit)

- **MMU (Memory Management Unit)**
  - 논리 주소(logical address)를 물리 주소(physical address)로 변환(mapping)하는 하드웨어.
  - CPU 에서 특정 논리 주소 위치에 저장된 데이터를 요청할 때, MMU 가 자동으로 주소를 변환한다.
  - 소프트웨어 프로그래머는 논리 주소 만으로 작업한다.
    즉, 실제 (물리적) 주소 위치는 모른다.



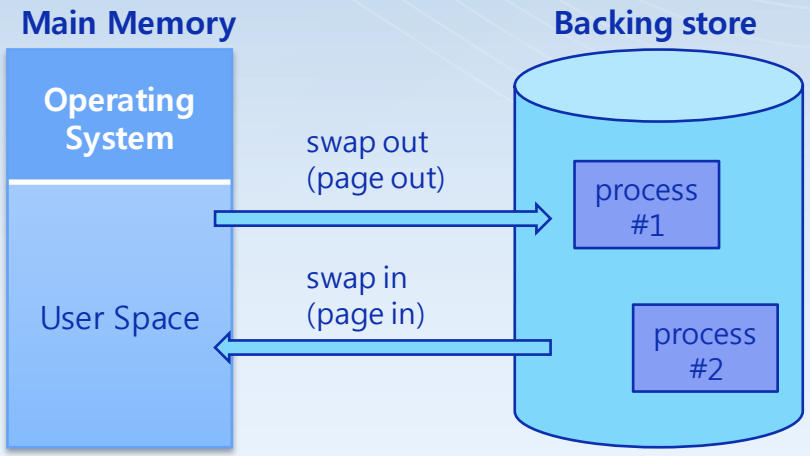
동적 로딩 (dynamic loading)

- **동적 로딩 (dynamic loading)**
  - 프로세스가 시작될 때 그 프로세스의 주소 공간 전체를 메모리에 올려 놓는 것이 아니라 메모리를 좀 더 효율적으로 사용하기 위해 필요한 루틴(rouine)이 호출될 때 해당 루틴을 메모리에 적재하는 방식
  - 사용되지 않을 많은 코드가 메모리에 올라가는 것을 막아 메모리의 낭비를 막는 기법이다.
  - 동적 로딩은 운영체제의 특별한 지원 없이 프로그램 자체에서 구현이 가능하며, 운영체제가 라이브러리를 이용해 지원할 수도 있다.



스와핑 (Swapping)

- **스와핑 (Swapping)**
  - 메모리에 적재된 프로세스의 메모리 공간 전체를 디스크의 스왑 영역(swap area)에 일시적으로 내려 놓는 것을 말한다.
  - 이 때, 스왑 영역을 backing store 라고도 부르며, 디스크 내의 파일 시스템과는 별도로 존재하는 영역이다.
  - 스왑 영역은 프로세스가 수행 중인 동안에만 디스크에 저장되는 공간이므로 저장 기간이 상대적으로 짧은 저장 공간이다.
  - Swap In (Page In) : 디스크에서 메모리로 프로세스를 실행하기 위해 메모리로 옮기는 작업
  - Swap Out (Page Out) : 메모리에서 프로세스를 일시적으로 저장 장소로 옮기는 작업

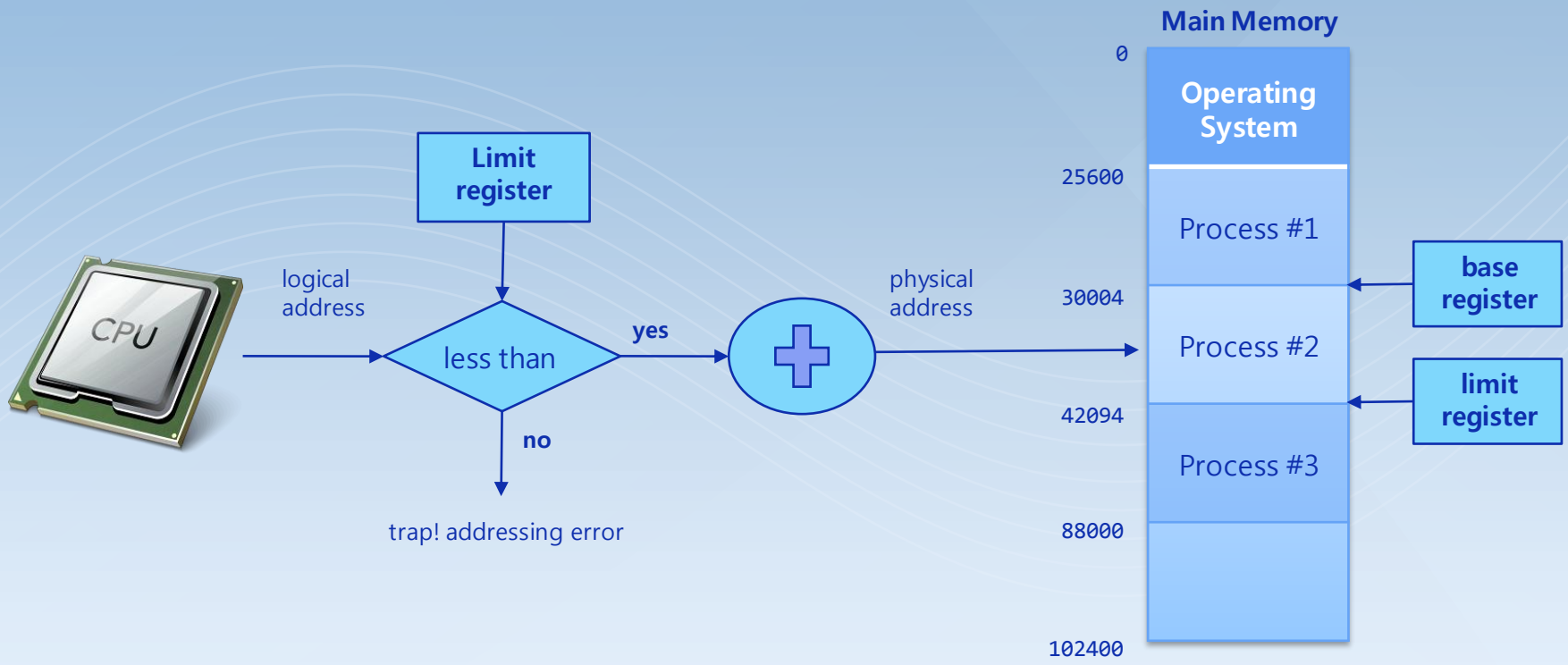


연속 할당 (Continuous Allocation)

- **운영체제 vs. 사용자 프로세스**
  - Main Memory 는 두 영역으로 나뉜다.
  - 운영 체제 (Operating System)
    - 인터럽트 백터(Interrupt Vector)와 함께 하위 메모리 영역에 위치한다.
  - 사용자 프로세스 (User Process)
    - 운영체제 보다 상위 메모리에 위치한다.

- **연속 할당 (Continuous Allocation)**
  - 각 프로세스는 메모리의 연속적인 한 블록(block) 에 적재된다.
  - Base register 방식으로 사용자 프로세스 간에 간섭(침해)하지 못하도록 한다.
    - Base register는 가장 작은 물리 주소값 포함
    - Limit register는 논리 주소의 크기 포함
    - Limit regisater 의 값보다 큰 위치의 주소를 참조(접근)할 수 없다.

연속 할당 (Continuous Allocation)



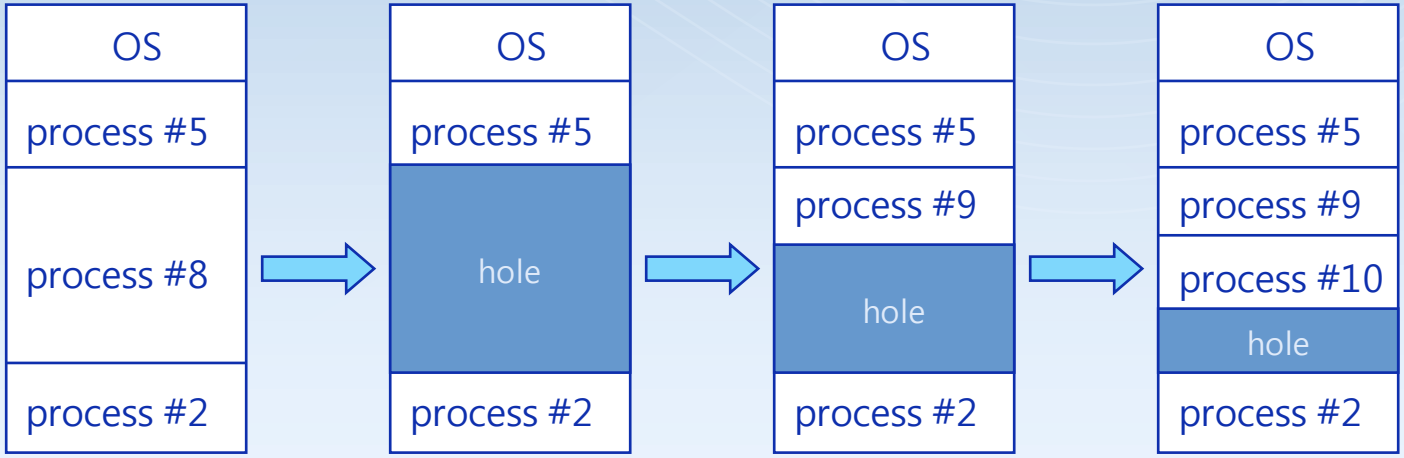
☞ 주소 연산 수행 시, 연속 할당된 범위를 벗어난 주소에 접근할 경우, 오류가 발생한다.

구멍과 할당 (Hole and Allocation)

- **구멍 (Hole)**
  - 할당 가능한 사용 중이 아닌 메모리 영역
  - 전체 메모리 중에서 사용자 프로세스(어플리케이션)가 할당되지 않은 영역

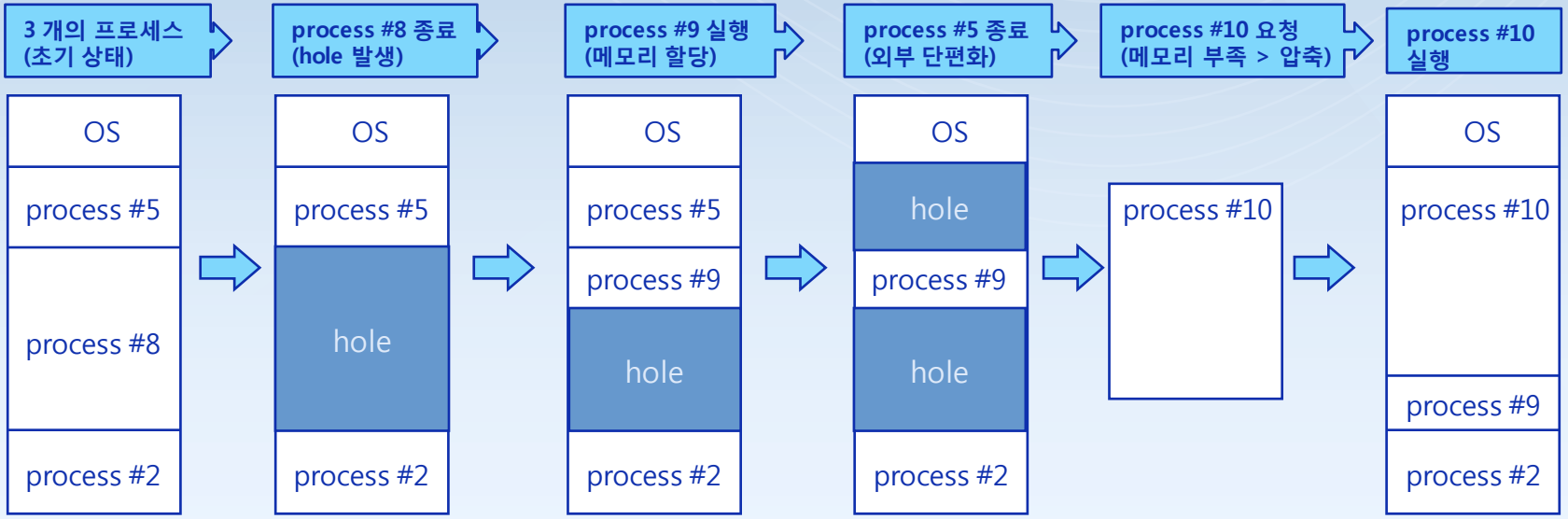
- **할당 영역 vs 자유 영역**
  - 운영체제는 실행 중인 프로세스를 위해 할당된 부분(allocated partition)과 사용되고 있지 않은 자유 영역(free partitions = hole)을 관리한다.

- **프로세스 메모리 할당**
  - 프로세스 실행 시, 해당 프로세스에게 필요한 메모리 영역보다 큰 구멍을 찾아서 프로세스가 사용할 수 있도록 제공(배정)한다.



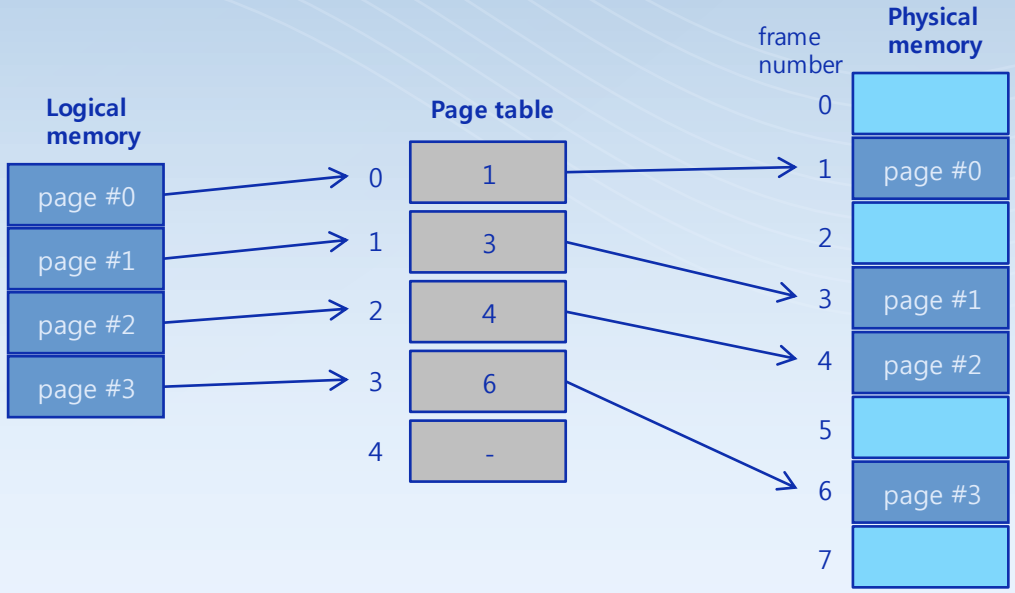
단편화(Fragmentation)

- **단편화 (Fragmentation)**
  - 프로세스의 반복적인 실행과 종료로 인해 복수의 구멍(hole)이 메모리 상에 생겨나는 현상
- **외부 단편화 (External Fragmentation)**
  - 전체 여유 메모리 공간이 하나의 할당 요청을 만족시키기에 충분하지만 연속적이지 않은 경우
- **내부 단편화 (Internal Fragmentation)**
  - 할당된 메모리가 요청한 메모리보다 약간 더 큰 경우에 생김
  - 최소 할당 블록의 정수배 메모리를 할당하므로, 이로 인해 남는 공간을 의미
    (최소 할당 블록 크기가 32 byte인 경우, 50 byte 할당을 요청하면 64 byte의 메모리 공간을 제공하고, 14 byte의 내부 단편이 발생한다.)
- **압축 (Compaction)**
  - 외부 단편화 영역을 합치는 것, 모든 여유(free) 메모리 공간을 모아 큰 영역으로 만드는 작업
  - 재배치(relocation)이 동적, 즉 실행 시간에 이루어질 수 있는 경우에만 가능하다.



페이징 (Paging)

- **페이징 (Paging)**
  - 프로세스의 주소 공간을 동일한 크기의 페이지 단위로 나누어 불연속적인 메모리 공간에 저장하는 방식을 말한다.
  - 각 프로세스의 메모리 영역 전체를 물리적 공간에 적재할 필요가 없으며, 일부는 backing store (disk 영역 등)에 일시적으로 보관하는 것이 가능하다.
  - 연속 할당에서 동적 메모리 할당 시 단편화로 인해 '압축'을 빈번하게 실행해야 하는 문제가 발생하지 않는다. 또한 외부 단편화가 발생하지 않지만, 내부 단편화는 발생한다.
  - 논리적 메모리 영역을 물리적 메모리 위치로 매핑(mapping) 하기 위해 페이지 테이블(page table)을 사용한다.



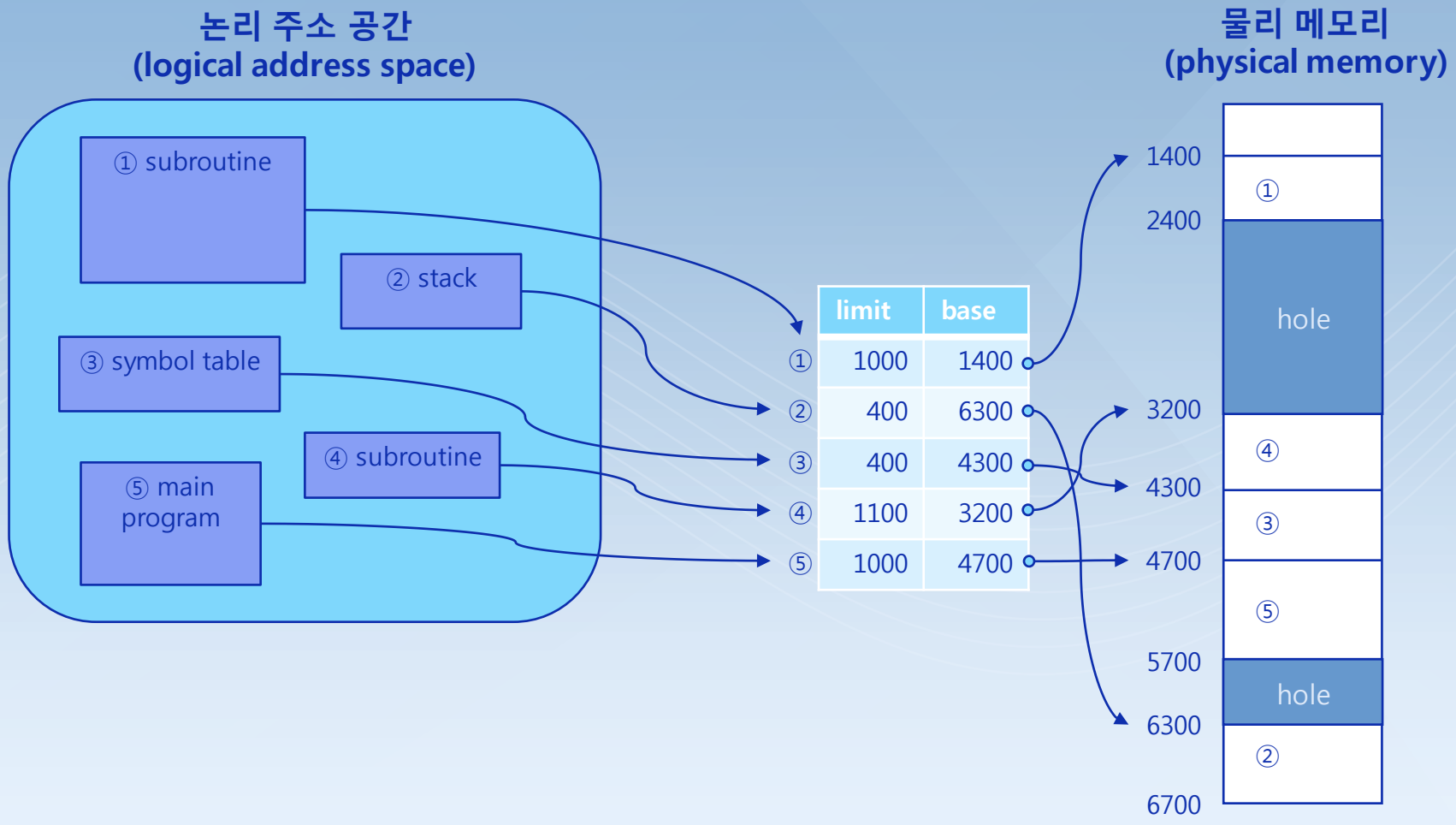
세그멘테이션 (Segmentation)

- **세그멘테이션(Segmentation) 기법**
  - 페이징 기법에서는 가상 메모리를 같은 크기의 블록으로 분할했으나 세그멘테이션 기법에서는 가상 메모리를 서로 크기가 다른 논리적 단위인 세그먼트(segment)로 분할하고 메모리를 할당하며 주소 변환을 하게 된다.
  - 세그먼트들의 크기가 서로 다르기 때문에 메모리를 페이징 기법처럼 미리 분할해 둘 수 없고, 메모리에 적재될 때 빈 공간을 찾아 할당하는 사용자 관점의 가상 메모리 관리 기법이다.

- **세그먼트(Segment)**
  - 각 메모리 블록의 크기가 같은 페이지와는 달리 세그먼트는 논리적 의미에 부합하도록 서로의 크기가 다르며 각각의 세그먼트들은 연속적인 공간에 저장되어 있다.
  - 세그멘테이션 기법의 '세그먼트 가상 주소'는 v = (s,d)로 표현되며 s는 세그먼트 번호를, d는 블록 내 세그먼트의 변위(상대적 위치)를 나타낸다.
  - 각 세그먼트는 쪼개지지 않고, 하나의 블록으로 보조기억장치에서 메모리의 연속적인 가용 공간에 적재된다.

- **세그먼트 테이블(segment table)**
  - 페이징 기법에는 페이지 테이블이 있는 것처럼, 세그멘테이션 기법에는 세그먼트 테이블이 사용된다. 세그먼트 테이블은 사용자가 정의한 주소를 실제 주소로 매핑(mapping)하는 정보를 저장하고 있으며 각 세그먼트 항목별 Base(세그먼트 시작 주소)/Limit(세그먼트 길이)의 정보를 가지고 있다.

세그먼트 매핑(segment mapping)



세그멘테이션 기법 적용

- **컴파일러에 의한 세그먼트 생성**
  - 일반적으로 컴파일러에 의해 소스 프로그램 컴파일 수행 시, 자동적으로 실행 프로그램 내에 세그먼트를 구축한다
  - 컴파일러는 다음과 같은 요소들을 위한 세그먼트들을 생성한다 :
    - 프로그램 실행 코드
    - 전역 변수들
    - 힙(heap), 메모리가 할당된 폼
    - 각 쓰레드(thread) 및 함수에 의해 사용되는 스택(stack)
    - 각종 라이브러리

- **라이브러리와 로더**
  - 라이브러리(library)들은 분리된 세그먼트에 할당될 수 있다.
  - 로더(loader)는 실행 파일이 저장된 세그먼트들을 취한 후, 메모리 로딩 시 세그먼트 번호를 할당한다.

- **malloc()**
  - malloc() 함수 동작 메커니즘 = segmentation.

- **segmentation fault error**
  - 프로그램 실행 중 발생하는 'segmentation fault' 메시지는 '지정된 세그먼트의 허용 범위를 벗어난 주소'에 접근했을 때 발생하는 오류이다.

추가 학습 가이드

- **MMU (Memory Management Unit)**
  - MMU 관련 TLB(Translation Look aside Buffer) 개념도 중요하다.
  - 참조 : http://recipes.egloos.com/5232056

- **페이징 기법**
  - page fault(major fault/minor fault) 개념도 중요하다.

- **세그멘테이션과 페이징 (segmentation with paging)**
  - 현대 운영체제(operating system)에서는 가상 메모리(virtual memory) 관리를 위해 페이징 기법과 세그멘테이션 기법을 함께 사용한다.
  - 참조 : http://www.cs.nyu.edu/courses/spring09/V22.0202-002/lectures/lecture-22.html

참고 자료

- 친절한 '임베디드 시스템 개발자 되기' 강좌
  - http://recipes.egloos.com/
- Accessibility
  - http://www.slideshare.net/cunniman/addressability
- Main memory
  - http://www.slideshare.net/younggunna794/chapter-8-18403737
- SNOW 위키
  - http://snowwiki.fuzewire.com/wiki/applied_sciences/computer_science/infor_science/read.html?psno=*D30ACDF27AEAFC44D3AF426EF24F0DA54E784B89