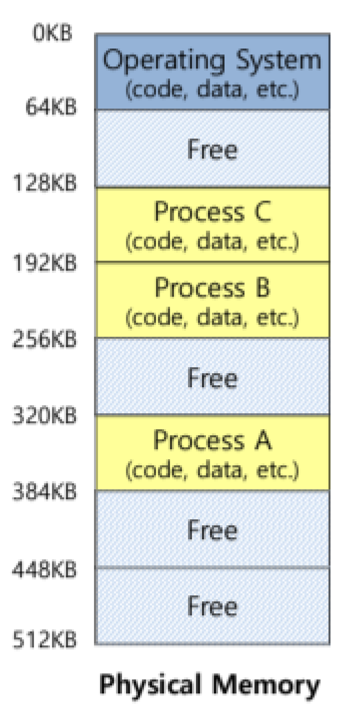
in the Early System...

- 메모리에 하나의 프로세스만 load
- poor utilization, poor effieciency

Multiprogramming and time sharing

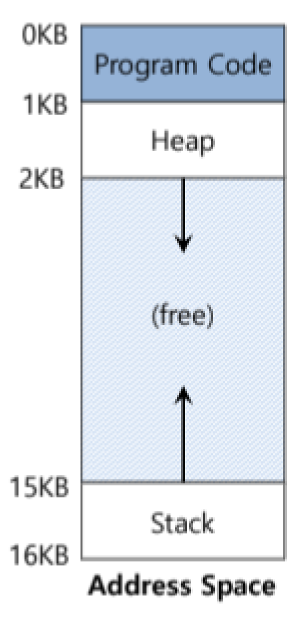
- 옛날엔 메모리에 여러 프로세스를 적재



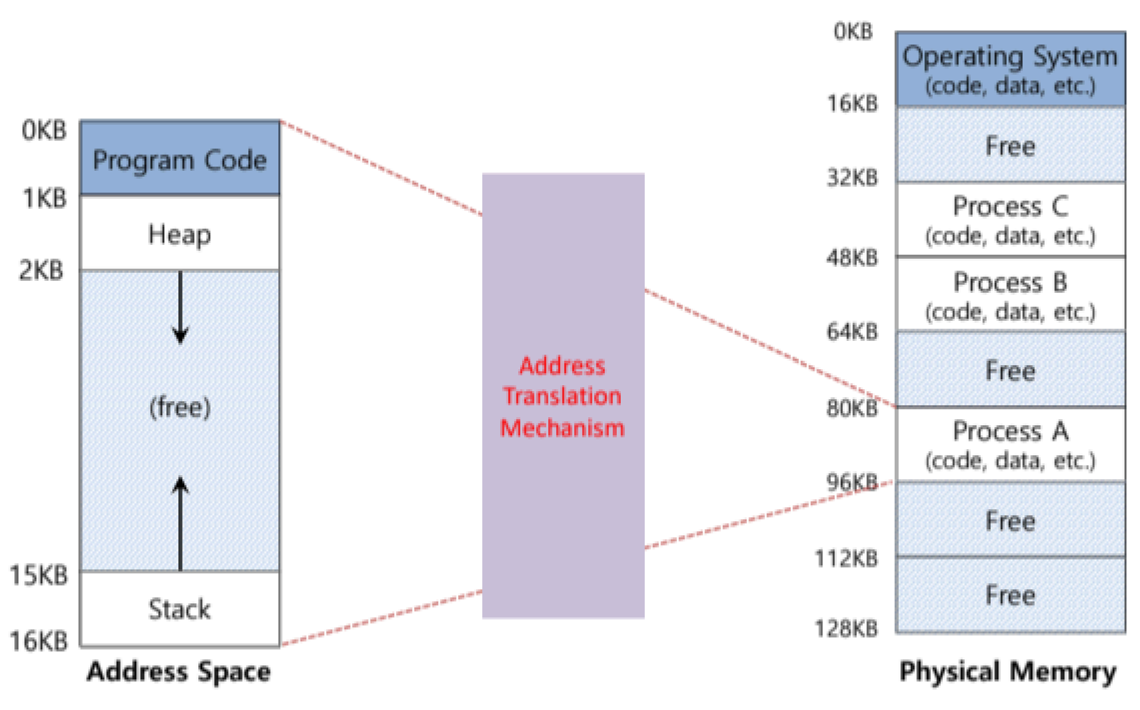
  - 짧은 시간 동안 하나만 execute
  - 메모리에서 각 프로세스들을 switching
  - good utilization, efficiency
- protection issue 발생 가능성
  - errant한 memory access

Address Space

- OS가 physical memory를 추상화
- address space가 현재 실행 중인 process에 대한 모든 걸 소유
  - address space contains various components such as program code, heap, and stack



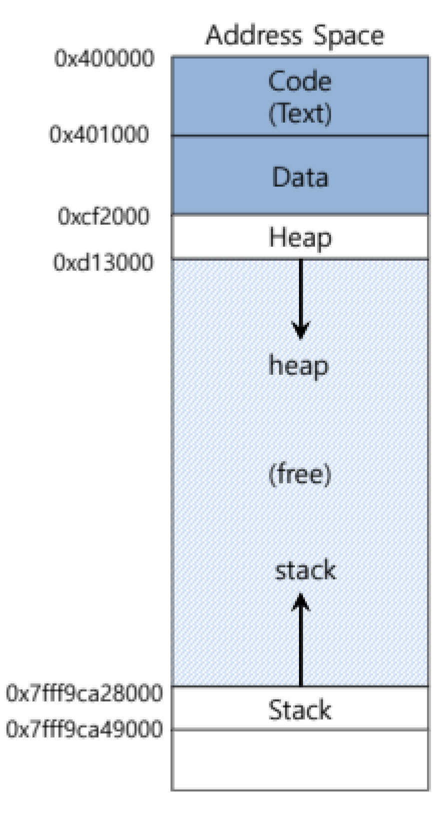
- program code : instructions이 존재하는 곳
- heap : 동적으로 메모리가 할당되는 곳
- stack : return addr이나 value들 저장, local variable이랑 argument도 저장



- physical memory의 주소를 address space로 virtualization

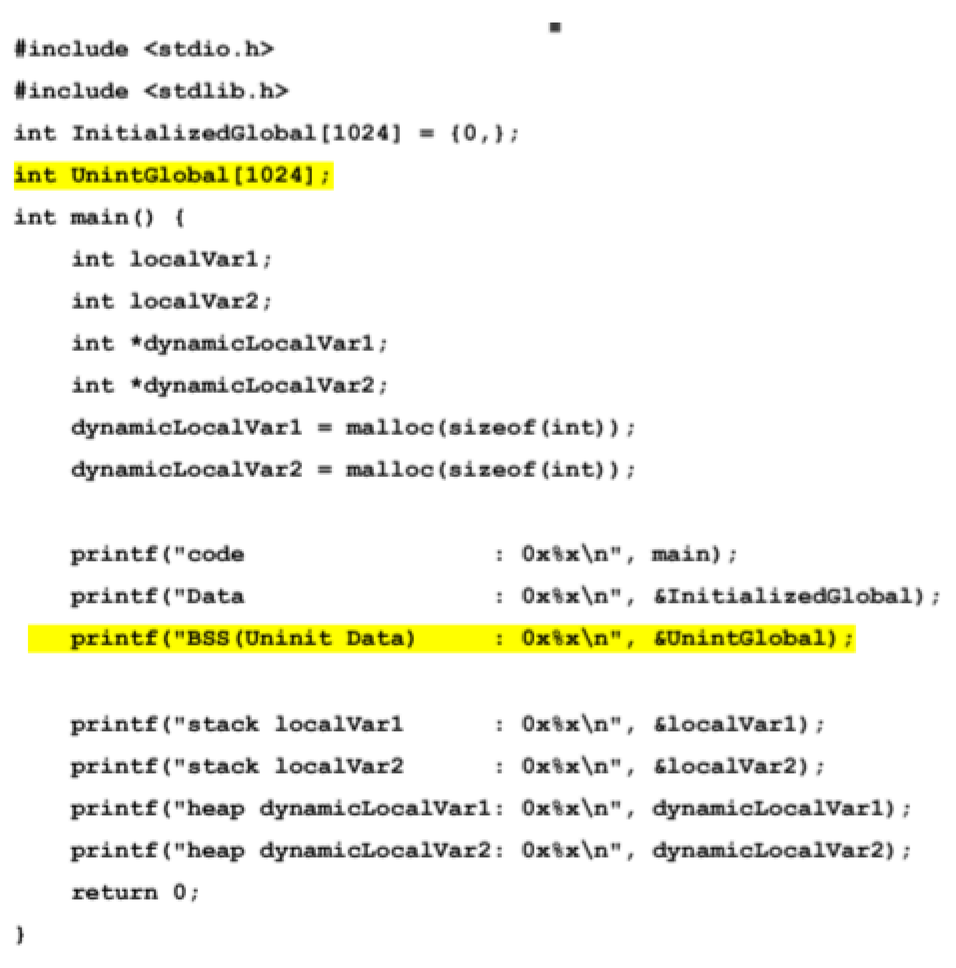
virtual address

- 실행 중인 프로그램에서 사용 중인 address는 모두 virtual
- OS가 virtual address를 physical address로 변환



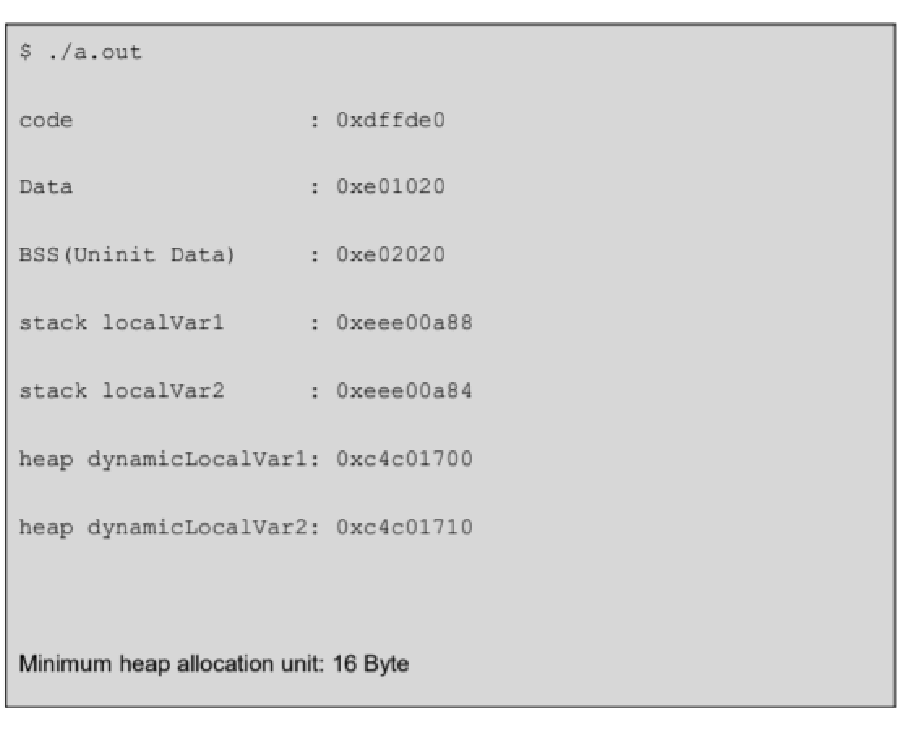
- 1바이트마다 주소 0x1 늘어남
- 1페이지가 4KB인데 이는 0x1000 (2^12)이므로 0x01에서 1 페이지 뒤 주소는 0x1001

virtual address space에서 주소 확인해보기

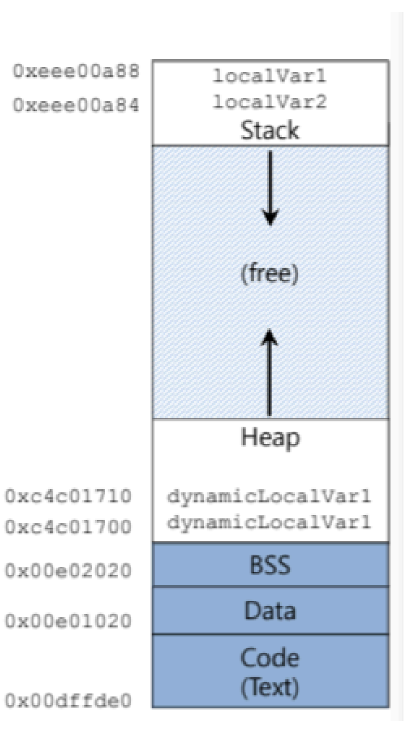


```
#include <stdio.h>
#include <stdlib.h>
int InitializedGlobal[1024] = {0,};
int UnintGlobal[1024];
int main() {
    int localVar1;
    int localVar2;
    int *dynamicLocalVar1;
    int *dynamicLocalVar2;
    dynamicLocalVar1 = malloc(sizeof(int));
    dynamicLocalVar2 = malloc(sizeof(int));

    printf("code                : 0x%x\n", main);
    printf("Data                : 0x%x\n", &InitializedGlobal);
    printf("BSS(Uninit Data)    : 0x%x\n", &UnintGlobal);

    printf("stack localVar1     : 0x%x\n", &localVar1);
    printf("stack localVar2     : 0x%x\n", &localVar2);
    printf("heap dynamicLocalVar1: 0x%x\n", dynamicLocalVar1);
    printf("heap dynamicLocalVar2: 0x%x\n", dynamicLocalVar2);
    return 0;
}
```

- InitializedGlobal : 초기화된 글로벌 변수
- UnintGlobal : 선언만 한 글로벌 변수



```
$ ./a.out

code                 : 0xdffde0

Data                 : 0xe01020

BSS(Uninit Data)     : 0xe02020

stack localVar1      : 0xeee00a88

stack localVar2      : 0xeee00a84

heap dynamicLocalVar1: 0xc4c01700

heap dynamicLocalVar2: 0xc4c01710


Minimum heap allocation unit: 16 Byte
```



- code :
  - main 함수가 0x00dffde0에 시작
  - 그 위로 코드들이 자리를 차지
- Data : 코드 다음엔 글로벌 변수들이 자리를 차지
- BSS : 그 후 선언만 한 글로벌 변수들이 차지 (BSS : Block Started by Symbol)
- stack : 스택엔 지역 변수들이 저장
  - int이므로 0x04 차이 (4바이트 차이)
- heap : 스택엔 포인터들이 저장
  - 포인터는 32비트 시스템에선 4바이트 64비트 시스템에선 8바이트 (여긴 32비트)
  - int* 이므로 0x10차이 (16바이트 차이, minimum heap allocation unit)

Memory Virtualization

- OS는 physical memory를 virtualize
- virtualization으로 인해 각 프로세스가 memory space를 가진 듯한 illusion
- 마치 각 프로세스가 메모리 전체를 쓰는 것 처럼 보임

왜 가상화를 하는가? 가상화는 어떤 목표를 추구해야?

- Transparency
  - 프로세스는 메모리가 공유된다는 걸 모르는 게 좋음
  - 프로그래밍하는 데 편리한 abstraction 제공 (i.e. a large, contiguous memory space)
- Efficiency
  - 파편화 최소화 (space)
  - hardware의 support를 받기 (time)
- Protection
  - 프로세스와 OS를 다른 프로세스로부터 protect
  - Isolation : 프로세스가 다른 프로세스를 건드리지 않고 fail
  - 동시에 다른 프로세스와 메모리 일부를 공유

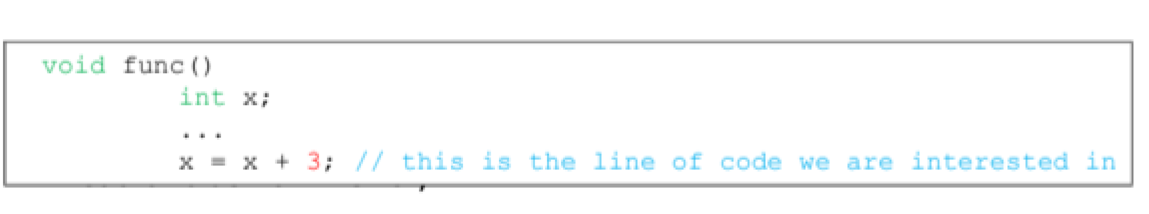
Logical vs. Physical Address Space

- logical address space가 physical address space는 분리되어있지만 서로 연결(bound)되어있다는 개념은 적절한 memory management를 위해 central(중요)하다
  - Logical address : CPU에 의해 생성 (virtual address 라고도 함)
  - physical address : 실제 memory 주소
- Logical address와 physical address는 실제로 compile-time과 load-time에서 address binding schemes 중에도 같음
  - load time ? 프로그램을 메모리에 적재하는 시점
- 하지만 execution-time address binding scheme 때는 달라짐

Address Translation

- 하드웨어가 virtual address를 physical address로 변환
- OS는 하드웨어를 설정하는 중요한 순간에 관여를 해줘야 함
  - 현명하게 개입하기 위해 메모리를 관리해야 함
  - 어느 위치가 비어있고 사용 중인지 계속 파악해야 함
- Assumption
  - user의 address space는 physical memory에서 contiguously하게 배치되어야 함
  - address space의 크기는 physical memory보다 작아야 함
  - 각각의 address space는 정확히 같은 크기어야 함

address가 변환되는 간단한 예시

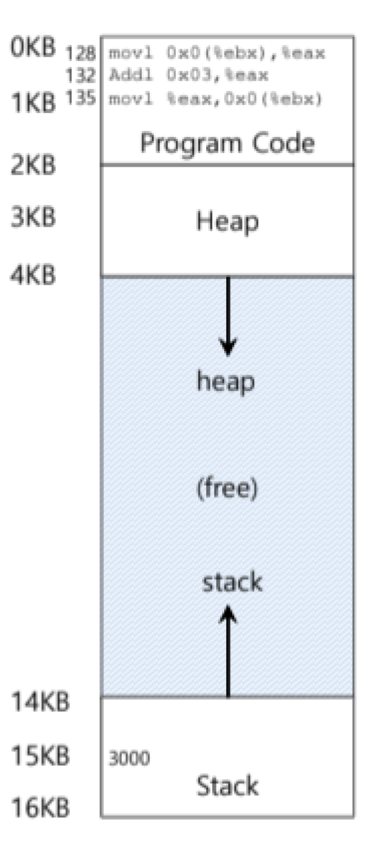


```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

- x에 3을 더하는 간단한 예시

```
128 : movl 0x0(%ebx), %eax          ; load 0+ebx into eax
132 : addl $0x03, %eax              ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)          ; store eax back to mem
```

- 실제 메모리 주소 사용 중



- logical과 physical 주소를 넘나들며 사용

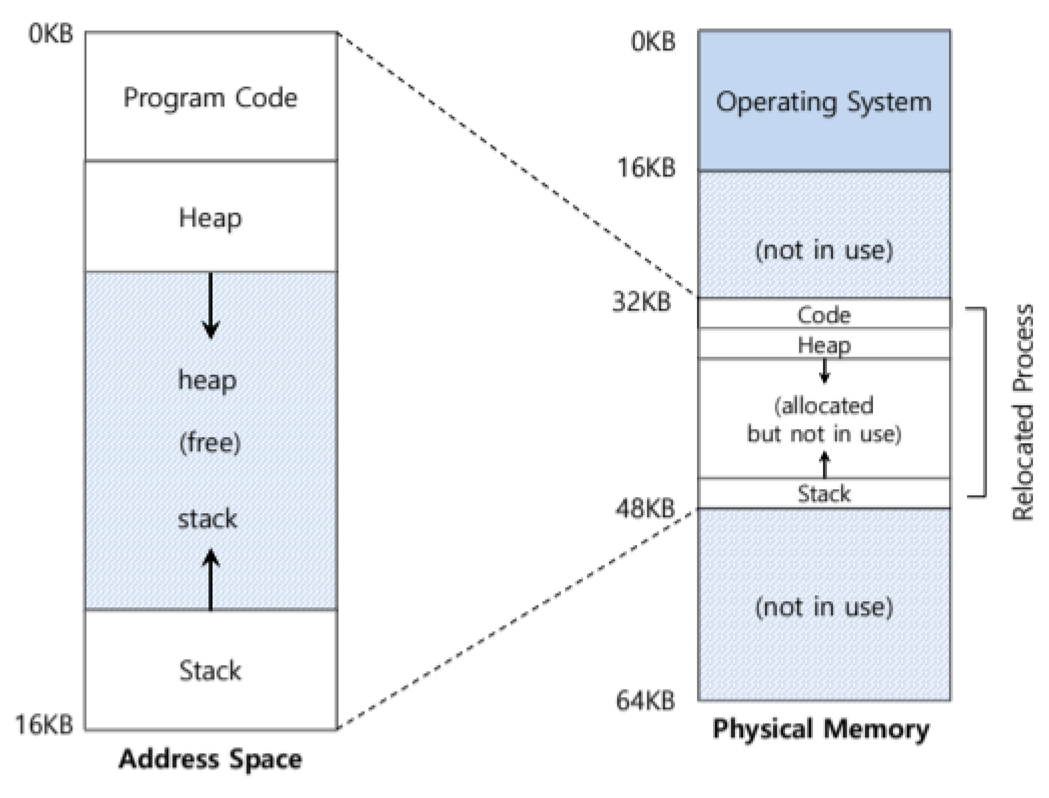
Address Binding

- instruction과 data의 메모리 주소를 binding하는 작업은 3단계로 나뉨

1. compile time
    - 메모리 주소를 미리 (priori) 알고 있다면 absolute code가 compile time에 생성됨
    - absolute code : 절대 주소를 사용하는 코드
    - 시작 위치가 바뀌면 다시 컴파일 해줘야 함
2. load time
    - 메모리 주소를 compile time에 모른다면 반드시 relocatable code를 생성해야 함
    - relocatable code : 코드가 메모리 내에서 위치가 바뀌어도 제대로 작동하는 코드
3. execution time
    - run time 전 까진 binding이 지연됨
    - 주소 매핑을 해주는 hardware support가 필요함 (ex : base, limit register)

Relocation

A Single Relocated Process

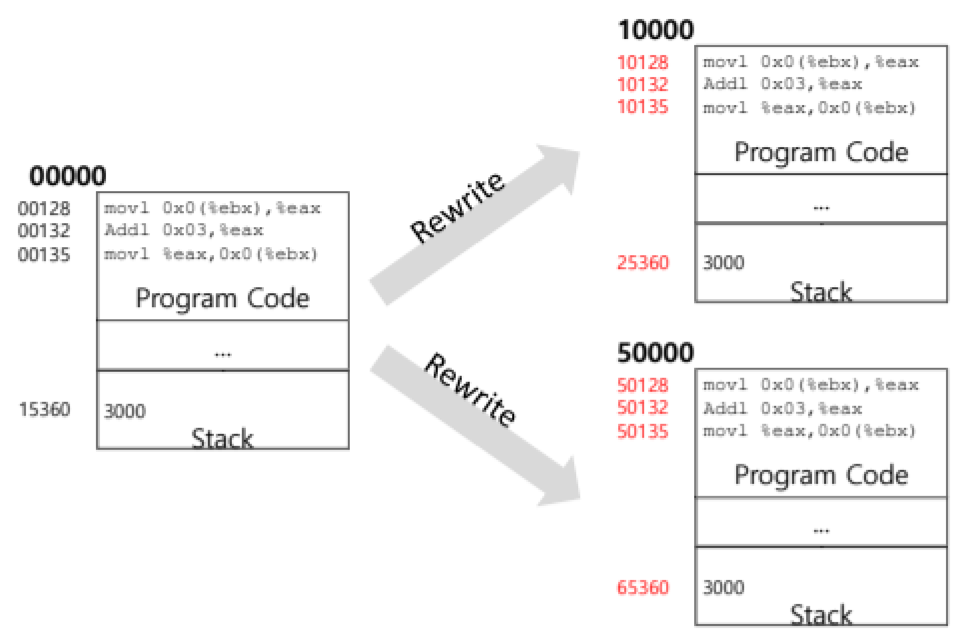


relocated process란?

- 프로그램 실행 중 어디서든 실행될 수 있도록 설계된 프로세스
- 프로세스는 고정된 주소가 아닌 메모리 내 다른 위치에 로드됨
- 프로세스가 실행되는 동안 주소가 변할 수 있음
- relocation은 컴파일러나 OS가 담당
- 상대 주소를 사용하여 어디에 로드되든지 문제 없도록 함

Static Relocation

- Software-based relocation
  - OS가 각 프로그램을 메모리에 적재하기 전에 rewrite
  - static data와 function의 주소를 change



부산대학교

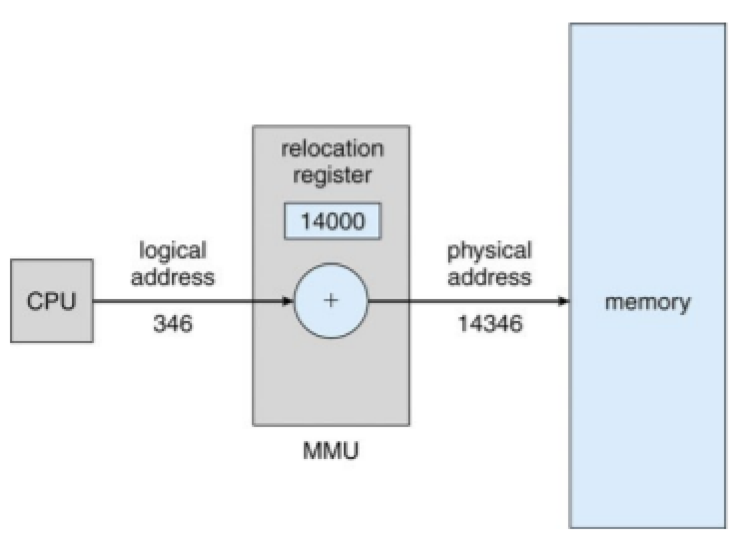
- 장점 : hardware support가 필요 없음
- 단점
    - No protection
        - 프로세스가 OS나 다른 프로세스의 메모리 영역을 건들 수 있음
        - No privacy : 어떤 메모리 주소든지 읽힐 수 있음
    - address space가 배치된 후 move 불가능
        - 파편화 때문에 새 프로세스를 할당할 수 없을 가능성도 있음

Dynamic Relocation

- Hardware-based relocation
- MMU (Memory Management Unit)이 모든 memory referece instruction에 메모리 변환 해줌
- Hardware가 Protection도 해줌
    - 만약 virtual address가 invalid하면 MMU가 exception을 raise
- OS가 현재 프로세스의 valid한 address space 정보를 MMU에게 넘겨줌

MMU (Memory Management Unit)

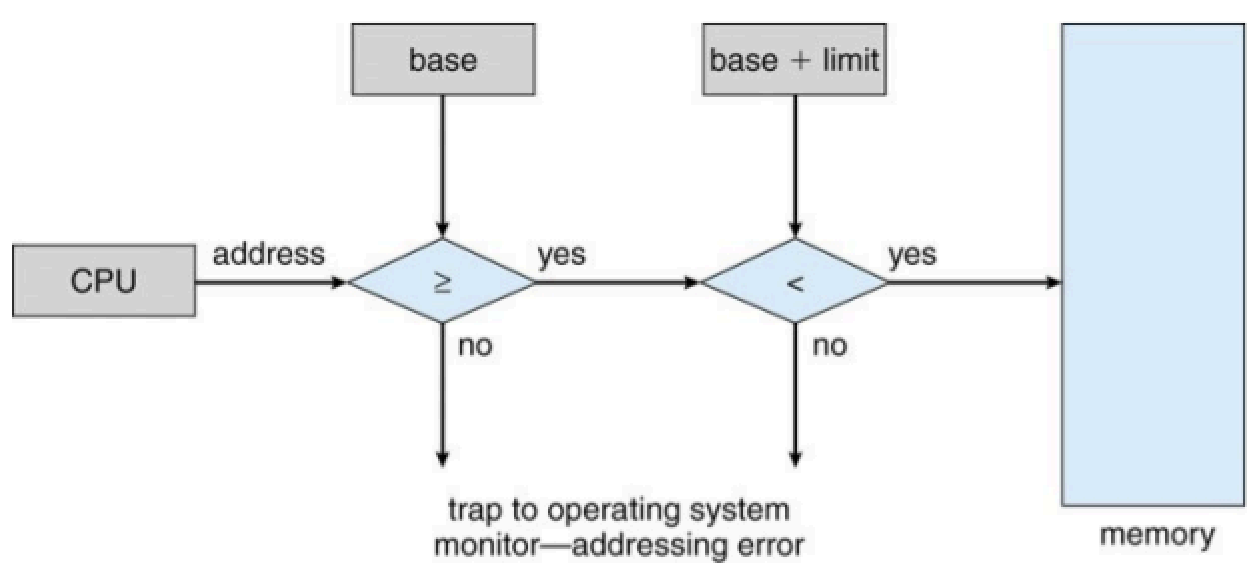
- virtual address를 physical address로 매핑해주는 HW 장치



- user process가 메모리로 적재될 때 process가 생성한 모든 address에 relocation register 값이 더해짐
- base register가 relocation register 역할
- user program은 virutal address만 다루고 절대 physical address를 알 수 없음
    - 메모리 내 주소에 reference가 이루어지면 execution-time binding 발생
    - execution-time binding이 MMU가 매핑하는 과정

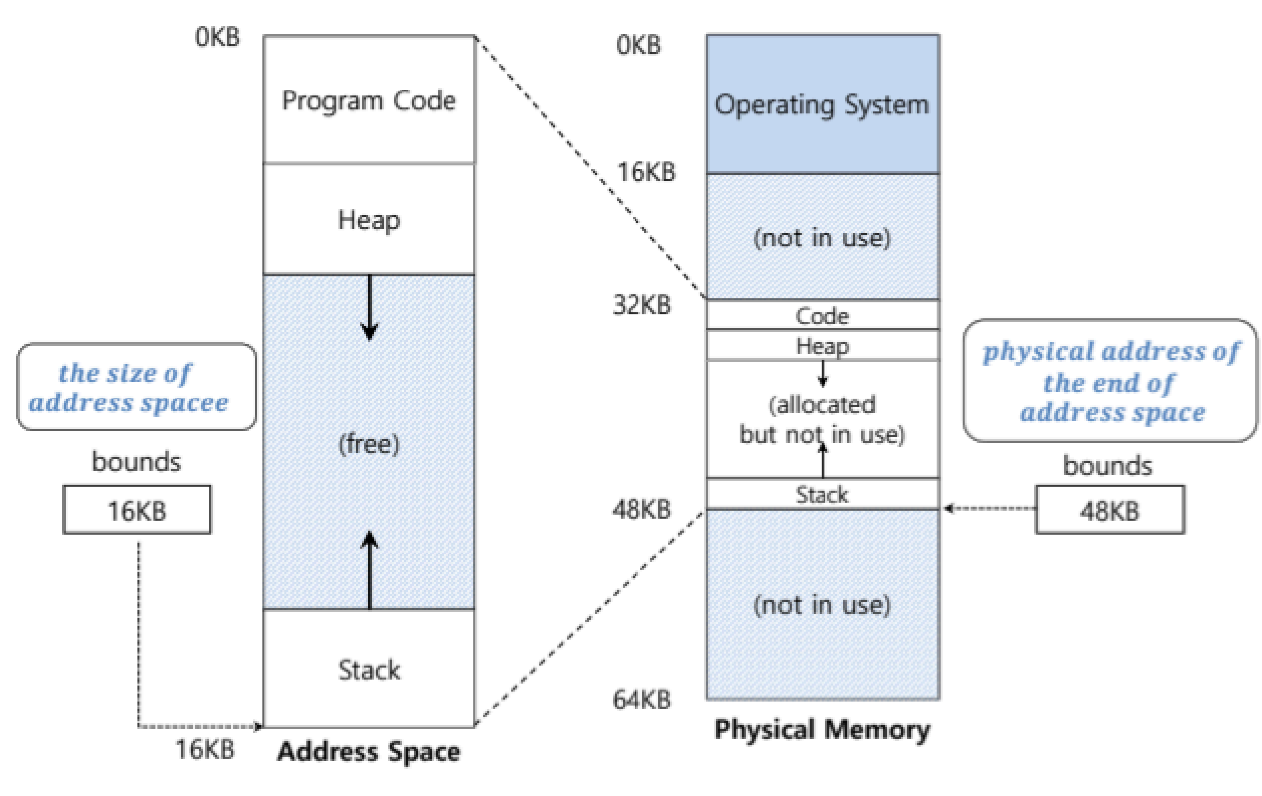
Background

- base register : 프로세스 첫 주소를 가르키는 레지스터
- limite register : 프로세스의 크기를 저장하는 레지스터



It is guaranteed that each process has a separate memory space

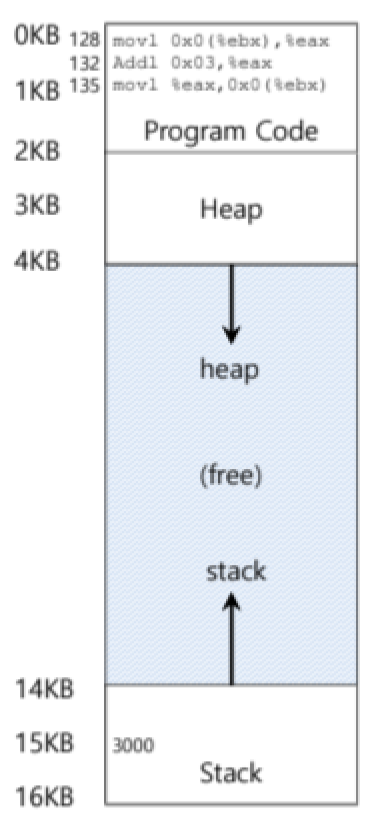
- 주소값이 base 이상, base + limit 미만임을 검사하는 과정이 필요



- bound register의 두 가지 역할

Dynamic Relocation

- hardware based
- 프로그램이 시작되면 OS는 프로세스를 어느 physical memory에 배치할지 결정



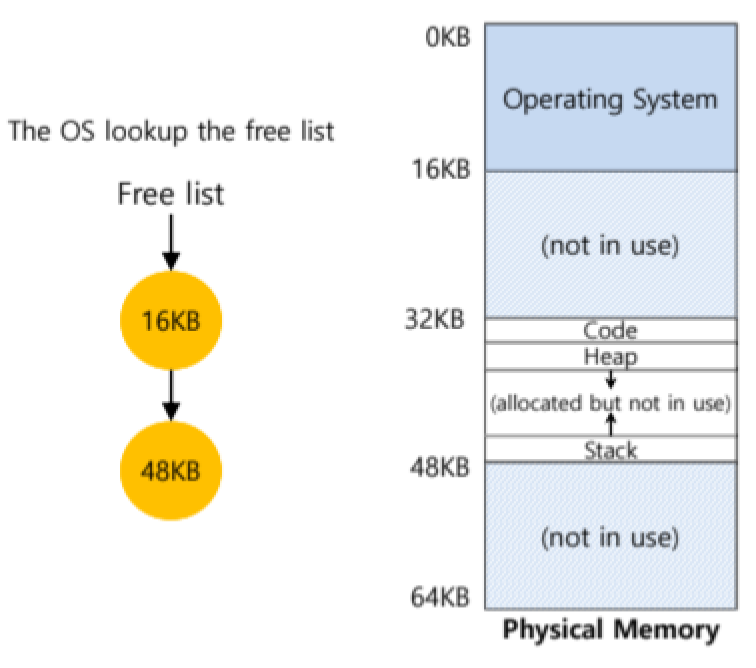
- 그림엔 안 보이지만 base register는 32KB
- 32KB는 32,768
- 128 : movl 0x0(%ebx), %eax
    - address space 기준 주소가 128이므로 실제 physical address는 128 + 32768 = 32896
- stack에 있는 3000 값에 접근하려면 address space에서 15KB이므로 physical address는 47KB

OS Issue for Memory Virtualizing

- base and bound approach를 구현하기 위해 OS가 해야할 일들
- Three critical junctures (시점)
    1. 프로세스가 시작할 때
        - physical memory에서 address space를 위한 공간 탐색
    2. 중단될 때
        - 사용할 메모리 reclaiming (회수)
    3. context switching 시
        - base와 bound를 저장

OS Issues : When a Process Starts Running

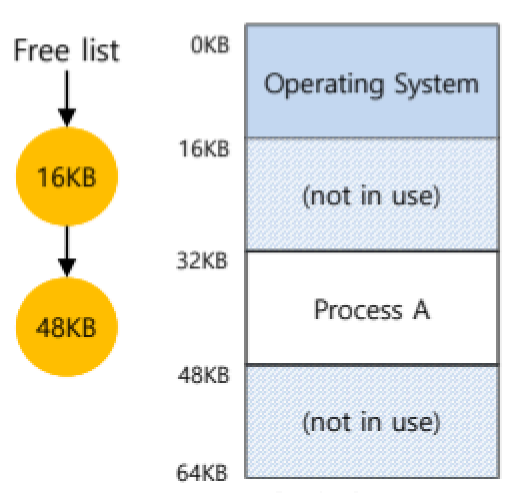
- OS는 new address space를 위한 공간을 찾아야 함
- free list : 사용 중이지 않은 physical memory의 범위가 적힌 list

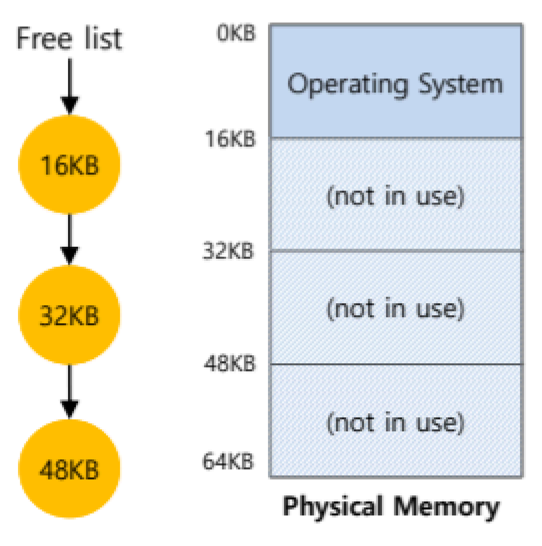


- OS는 free list를 참조

OS Issues : When a Process is Terminated

- OS는 free list에 메모리를 되돌려 놔야 함





OS Issues : When Context Switch Occurs

- OS는 base와 bounds를 PCB나 Process structure에 저장해놔야 함