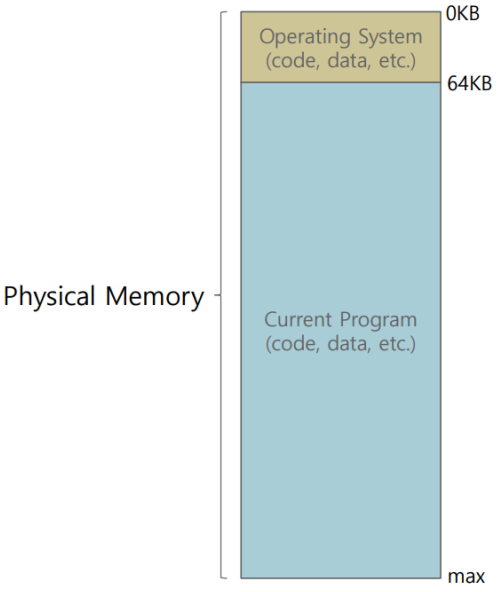
Address space : 각 process들이 자신만의 공간이 있다고 착각
↳ code, data 포함
↳ virtual address :
Address Translation : 가상주소를 physical address로 바꿔줌.

07. Address Spaces → memory virtualization

▼ Address Spaces

▼ Early Systems



1. 추상화 제공하지 않음 → physical memory에 현재 실행 중인 program 모두 저장
2. 계산만 열심히 함 → CPU bound

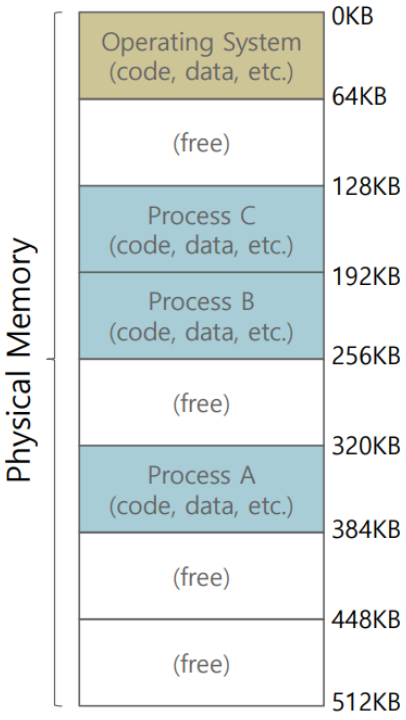
- early system 구조 ⇒전체 memory를 하나의 process를 위해서만 실행시킬수 있음.
  - i/o 작업을 하는 시스템 등장 → 여러 응용 SW 등장 → CPU 활용도 낮음

⇒ 성능 BAD

▼ Multiprogramming and Time Sharing

- i/o 작업을 하는 시스템 등장 → 여러 응용 sw 등장
  - early system 구조 : cpu 활용도가 낮아 여러 응용 sw에 대해 좋은 성능 X

⇒ 다른 구조가 필요함!



⇒ 고정된 크기로 미리 슬롯 나눠 놓음(각 프로세스 당 한 slot 차지)

- **Multiprogramming** : 여러 개의 program이 동시에 memory에 적재되어서 실행되는 환경을 뜻함.
  - multiprocess : 주어진 시간 동안 실행될 준비 → OS가 번갈아 실행

    ⇒ CPU 효용성 증가
- **Time Sharing** 한정된 CPU 자원을 여러 process에게 나눠주기 위해 time sharing.
  - context switch ⇒ 기존 구조에서는 원래 실행 중이던 process의 memory를 모두 disk에 저장했어야 함. (sharing X)

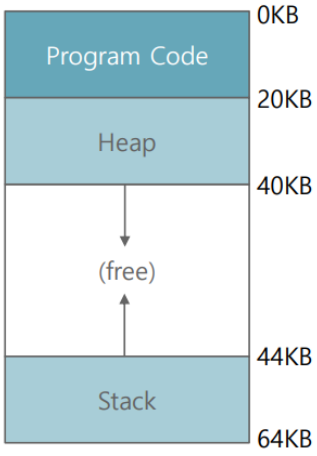
    ⇒ context switch overhead ↑
  - memory 영역 나누면 disk 접근하지 않아도 됨 ⇒ 성능향상

  But 임베디드 시스템에서는 사용하지만 현재 general purpose에서는 사용x
    - 각 process가 사용할 물리적인 메모리 제한
- Protection issue ☆
  - 어떤 process가 작동하고 있을 때 다른 process의 memory r/w 막을 수 있어야 함

⇒ address space 사용!

▼ Address Spaces

physical memory의 abstraction 사용하기 편리함



- system 내부의 실행 중인 program memory view → process마다 해당 공간이 존재한다고 생각하고 실행 가능
  - 실행 중인 program에 대한 memory state 모두 저장 (code, stack, heap)
- abstraction
  - ①응용 프로그램이 physical memory에 얼마나 차지하는지, 어디에 있는지 상관없이 가상 공간 제공 (0~64KB memory)
    - 항상 주소가 존재한다고 생각하고 실행 가능 ⇒ transparency
  - ②protection 기능 제공 → 0~64KB 이외의 영역에는 접근 X
- example
  - 64KB address space ⇒ 0에서 시작
  - program : physical memory의 0부터 64KB 전에 실제로 존재하는 것은 아님

▼ Virtual Memory

- How to
  1. private한가?
  2. 꽤 넉넉한가?
  3. 1,2 만족하면서 제한적인 물리 메모리에서도 돌아갈 수 있게 OS가 일하는가?

▼ virtual memory

- running program은 어떻게 생각?
  - particular address 0번 memory에 load된다고 생각
  - 엄청 큰 virtual address space(2^{32} byte or 2^{64} byte) 가진다고 생각 (각 process마다)

1. transparency : 자기 자신의 private physical memory 독점 (각 process마다 address space) (독점)

⇒ 두 가지를 고려해야 함. 2. efficiency

a. Time : 너무 느리지 않게 program running

   i. 소프트웨어적으로 환상 제공 → 오버헤드 발생 → 어떻게 구현?

b. Space : virtualizaton 지원하는데 필요한 구조에 너무 많은 메모리 사용 x

   i. kernel 내부에 존재해야 하는 자료구조 → 어떻게 공간적인 효율성 구현?

3. protection

a. process 사이에 isolation (침범 x)

b. OS : process들이 다른 공간(OS, 다른 process)에 맘대로 침입 못 하게 보호해줘야 함

▼ Virtual Address

↳ 확인 어떻게?? ⇒ 간단한 예제로 확인해보자!

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
  printf("location of code : %p\n", (void *) main);
  printf("location of heap : %p\n", (void *) malloc(1));
  int x = 3;
  printf("location of stack : %p\n", (void *) &x);

  return x;
}
```

↳ main 구현되어 있는 시작 부분 주소 return → code 주소 return

↳ heap 위 부분 출력

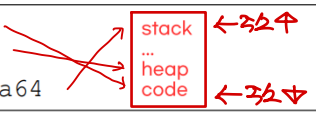
↳ 1byte 할당 → heap 주소 return.

↳ stack top 주소

↳ 고정변수 → stack에 저장 → stack 주소 return

(또 local variable이 stack에 저장되는건 아님. register에도 저장됨. ⇒ 주소 확인하려면 stack)

```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack : 0x7fff691aea64
```



↑주소수

↓주소수

▼ Memory API

▼ Type of Memory

- **Stack**: complier가 숨어서 할당, 반환 관리함. (함수 호출 or local variable 할당시에 사용)

  → automatic memory라고도 함.

  - 응용 SW : 명시적으로 stack을 조작하도록 하는 건 없음

- **Heap** : programmer가 명시적으로 할당, 반환 관리함
  - 응용 SW : 명시적으로 조작하고 메모리 할당
  - user, system 모두에게 더 많은 과제를 제시함.

▼ Memory API ⇒ library level에서 memory 할당과 반환이 진행됨.

- heap memory pull

```
void *malloc(size_t size) : a library call → not system call
```

  - heap에 memory 할당
  - process 생성 시에 고정적으로 생성? → 낭비, 부족의 여지가 있어 비효율적
    - 실제로 메모리 할당 받기 위한 system call은 내부에 존재함
    - malloc 안에서 해당 system call(brk, sbrk)을 호출하여 필요할 때마다 요청 → library 함수이기에 내 입맛대로 내부 수정 가능
      - *brk* : heap의 끝 위치를 변경(program's break)
      - *sbrk*
- heap memory free

```
void free(void *)
```

- common errors → OS가 protection 얼마나 잘 하는지 check!
  - forgetting to allocate memory
    - segmentation fault

```
//포인터 변수 초기화 x
char *src = "hello";
char *dst; //oops! unallocated -> 초기화 되지 않음, garbage
strcpy(dst, src); //segfault and die
//데이터를 작성할 수 없는 상태 or
//다른 버퍼에 저장되어 있던 내용이 지워짐
```

    - 올바르게 수정한 버전 : 초기화 → null 문자까지 포함하기 위해 +1

```
char *src = "hello";
char *dst = (char *)malloc(strlen(src) + 1 );
strcpy(dst, src); //work properly
```

  - Not allocating enough memory
    - buffer overflow → nul 문자 포함하지 않았을 경우

```
char *src = "hello";
char *dst = (char *)malloc(strlen(src)); //too small!
strcpy(dst, src); //work properly
```

응용 프로그램을 개발하는 과정에서는 항상 0으로 초기화하는 습관. 어느정도 구현은 되어 있지만 혹시 모른다!

① Forgetting to initialize allocated memory

② Forgetting to free memory
- Memory leak — 큰 프로그램, 실행 시간이 긴 프로그램 내에서는 더더욱 free가 필요함.

③ Freeing memory before you are done with it
- Dangling pointer — 일이 일어나기 전에 free하면 문제 -> 다른 프로그램에서 쓰려고 할 수 있음

④ Freeing memory repeatedly
- Double free — 같은 영역에 대해 메모리 반복적으로 프리 -> 프로그램이 die

⑤ Calling free() incorrectly
- Invalid free — 시작 주소가 아닌 할당되지 않은 정확하지 않은 주소에 있는 메모리에 대해 free 해버림 -> 프로세스가 die

▼ Address Translation

▼ Address space

program : address space 사용..

→ 자기 자신만의 code와 데이터 존재하는 private memory가 있다는 착각에 빠짐 (이런것처럼 OS가 제공)

→ 많은 programs : memory 실제로 동시에 공유함 → CPU : context switch

(하나의 memory를..)

⇒ 꽤 복잡함

▼ Address translation

- instruction에 의해 제공된 virtual address → 원하는 information이 실제로 위치한 physical address로 변경하는 작업
  - virtual address 이외에 실제 physical address 필요 ⇒ OS가 관리 (translation을)
  - fetch, load, store → program을 실행하기 위해서 필요한 기본적인 기능
- example
  - ① fetch : 기계어를 CPU가 가져옴
  - ② Load : memory → register
  - ③ store : register → memory

```
void func() {
        int x = 3000;
        x = x + 3;
…
}
```
*C code

```
128 movl 0x0(%ebx), %eax
132 addl $0x03, %eax
135 movl %eax, 0x0(%ebx)
```
*assembly code (memory / register / x=x+3 / register / memory)

① Fetch the instruction at address 128 (memory 위치)
Execute this instruction (load from address 15KB)
② Fetch the instruction at address 132
Execute this instruction (no memory reference) → 상속 더하기 (memory 접근 X)
③ Fetch the instruction at address 135
Execute this instruction (store to address 15KB)

- Hardware-based Address Translation
  - memory 접근 할 때마다 h/w가 address translation 일어나야 함. ☆

- but, h/w 혼자서는 virtualize memory에 접근할 수 없음
- OS : h/w 설정하기 위해 도움 필요 ⇒ 가구 만들기
  - cpu가 실행, OS가 자료구조/register 세팅 ⇒ 도구 세팅

◦ memory 효율적이고 유연하게 virtualization 하는 방법 → 다양한 memory 영역에 대해 하나의 address space가 잘 지원해야 함.

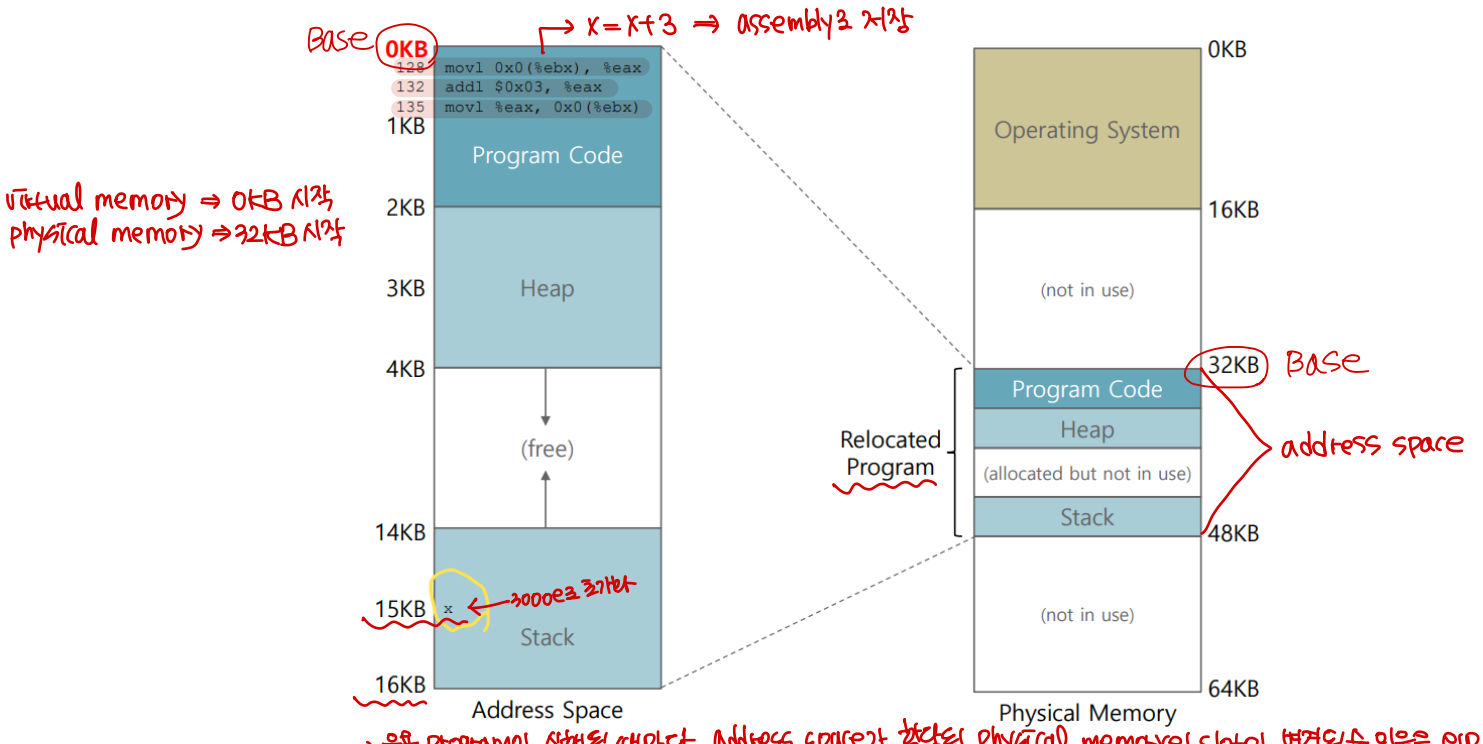
1. application이 필요한 flexibility 어떻게 제공?
   - 그들이 원하는 방식대로 program의 address space 사용 가능
2. application이 접근하는 memory location에 대해 어떻게 제어 유지?
   - 자신의 process 빼고 어떤 응용 program도 접근할 수 없도록 함 → protection 수
3. 여러가지 자료구조 등 하드웨어를 제공할 때 효율적으로 해야 함.

▼ Memory Relocation ⇒ address space를 제공할 수 있는 방법 중 가장 간단한 놈!

▼ assumtions for simple Memory Virtualization → 간단하게 보기 위해 실제와 다르게 가정

1. user의 address space : physical memory에 연속적으로 배치된 상태 (공간X)
2. address space의 크기가 너무 크지 않음 ⇒ physical memory보다 작음
3. 각 address space ⇒ 정확히 같은 사이즈

▼ Memory Relocation → virtual하게 0~16KB 존재하는 address space를 physical memory의 어느 위치에 둘 것인지 결정하는 것



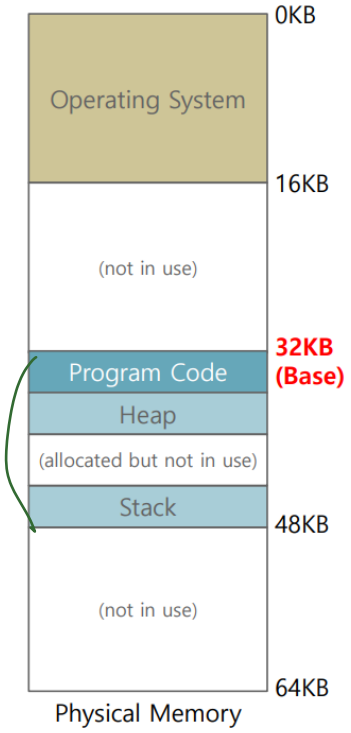
→ 응용 program이 실행될 때마다 address space가 할당된 physical memory 상의 어디에 배정될 수 있음을 의미

▼ Dynamic (Hardware-based) Relocation ⇒ 앞에서에서 dynamic, protection 지원해보자!

- **Base and Bounding** ⇒ 필요한 hw support

- *base* register : physical memory에 어디에나 address space 배치 가능 (시작 지점 명시)
- *bounds* register : process가 자기 자신 address space만 접근 가능 (끝 지점 명시)

- **Address translation** ⇒ base 가지고 할수 있음.



⇒ physical base + virtual = physical address

  - physical address = *Base* + Virtual address
    - virtual address 상에서의 offset
    - physical address의 시작 주소
  - example

virtual address

```
128: movl 0x0(%ebx), %eax
```

base 가상 주소 물리주소

PC: 128 -> 32KB(32768B) + 128B = 32896B

fetch 완료!

x: 15KB -> 32KB + 15KB = 47KB

base virtual physical

- dynamic!
  - 실행될 때마다 relocation(runtime 내에, process 시작한 후에도)
  - 다른 slot에 할당될 수 있음 ⇒ base 변화 가능
  - virtual 변화는 x

⇒ physical address 변화 가능

- protection ⇒ bound 가지고 할수 있음.

- memory reference가 bound 내에 있는지 check
- 만약 virtual address가 bound보다 크거나 음수라면 → CPU : exception ⇒ protection violation
  - ex) bounds : 16KB or 48KB

↳여기서부터 강의 다시 들자 ..

▼ h/w support → address space 제공하기 위해!!

	Hardware Requirements	Notes
①	Privileged mode (kernel mode일 때 실행 가능한 instructions)	Needed to prevent user-mode processes from executing privileged operations ⇒ OS가 쓰고 있는건 protection 가능 (↳user mode일 때 실행X)
②	base/bounds registers (exception 발생 가능한 것 -> OS가 가지고 있어야 ! 하는 기능)	Need pair of registers per CPU to support address translation and bounds checks ⇒ base : 시작 bound : 어디까지 유효?
③	Ability to translate virtual addresses and check if within bounds	Circuitry to do translations and check limits (in this case, quite simple) ⇒ 주소 바꿔줄 수 있어야 함 + bound 확인도 가능 (↳virtual address가 valid한지!!)
④	Privileged instruction(s) to update base/bounds (context switch -> base 주소 바뀜 -> user mode에는 실행되면 안 됨)	OS must be able to set these values before letting a user program run ⇒ base/bound 수정할 수 있는 privileged instruction 존재해야 함.
⑤	Privileged instruction(s) to register exception handlers	OS must be able to tell hardware what code to run if exception occurs (ex. 실행 중이던 process 종료 -> kernel에서만 실행 가능하도록!)
⑥	Ability to raise exceptions (exception 실행 능력)	When processes try to access privileged instructions or out-of-bounds memory

↳exception handler.

▼ OS issues (h/w가 관리 못한 부분은 OS가 관리)

- physical memory 잘 배분 → process 하나 생성될 때마다 relocation
- h/w가 지원해주는 기본 기능 ⇒ OS가 잘 사용해야 함

OS Requirements	Notes
Memory management (memory 관리 ↳relocation을 위해 memory에 대한 free 정보를 관리해야 함.)	Need to allocate memory for new processes; Reclaim memory from terminated processes; Generally manage memory via *free list* *)
base/bounds management (↳base/bound 저장, update)	Must set base/bounds properly upon context switch or moving a process's address space
Exception handling (↳exception handler 내부 구현)	Code to run when exceptions arise; likely action is to terminate offending process

*) Variable-sized address spaces and Larger address spaces than the size of physical memory are more difficult to handle ⇒ 가변적이고 physical memory보다 크면 다루기 어려움. *

* Summary

① Address space → program마다 자신만의 private memory 가지고 있다는 착각
② Address Translation → physical address ↔ virtual address

→ H/W support
- Base/Bound registers
- privileged instruction ⇒ update base/bounds
- bound 내에 있는지 check, translate 해.
- register exception

OS가 해야할 일
- OS Issues
- ⋮