Memory and Address Space

Memory Virtualization

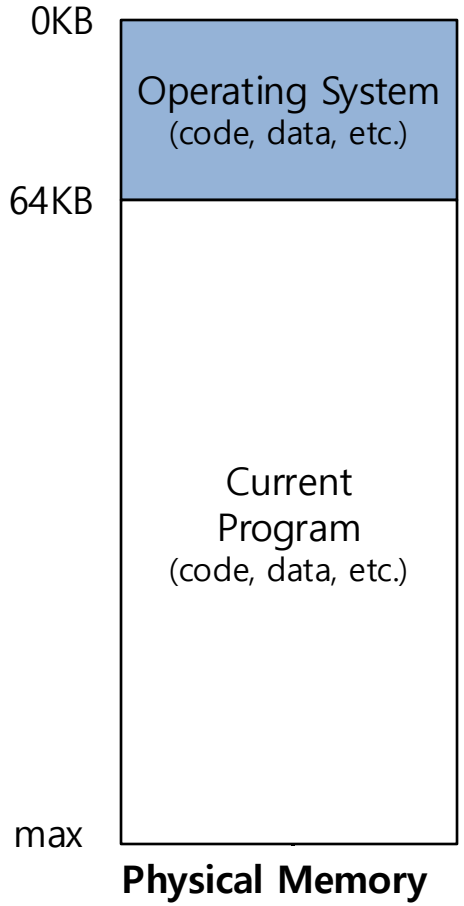
- What is memory virtualization?
  - OS virtualizes its physical memory
  - OS provides an illusion memory space per each process
  - It seems to be seen like each process uses the whole memory

Benefit of Memory Virtualization

- Ease of use in programming
- Memory efficiency in terms of times and space
- The guarantee of isolation for processes as well as OS
  - Protection from errant accesses of other processes

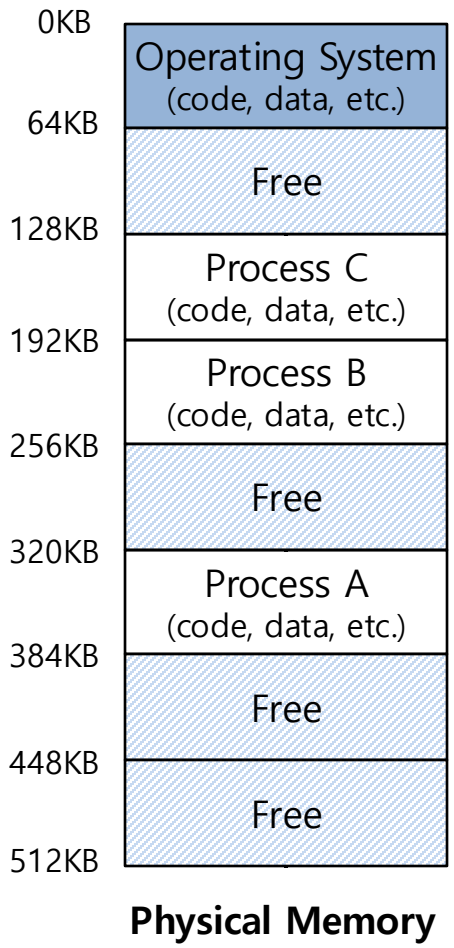
OS in The Early System

- Load only one process in memory
  - Poor utilization and efficiency



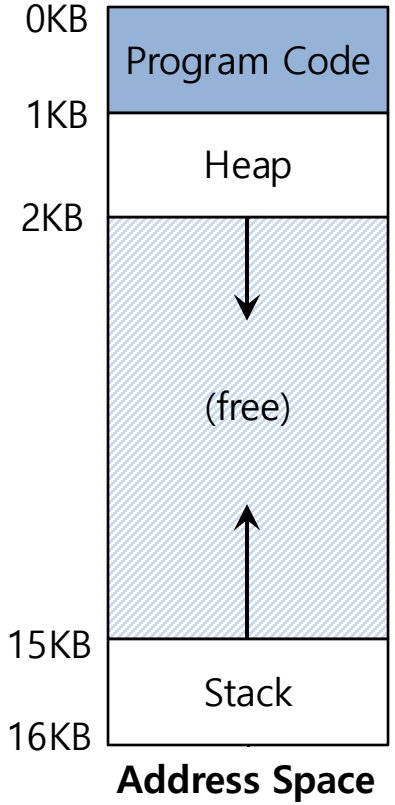
Multiprogramming and Time Sharing

- **Load multiple processes** in memory
  - Execute one for a short while
  - Switch processes between them in memory
  - Increase utilization and efficiency
- Cause an important **protection issue**
  - Errant memory accesses from other processes



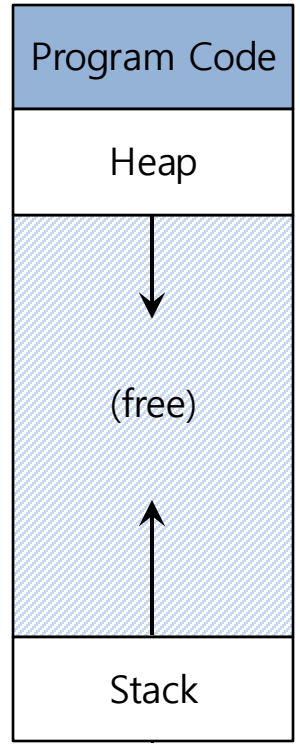
Address Space

- OS creates an abstraction of physical memory.
  - The address space contains all about a running process.
  - That is consist of program code, heap, stack and etc.



Address Space

- Code
  - Where instructions live
- Heap
  - Dynamically allocate memory.
    - malloc in C language
    - new in object-oriented language
- Stack
  - Store return addresses or values.
  - Contain local variables arguments to routines.



Address Space

Virtual Address

- **Every address** in a running program is virtual
  - OS translates the virtual address to physical address

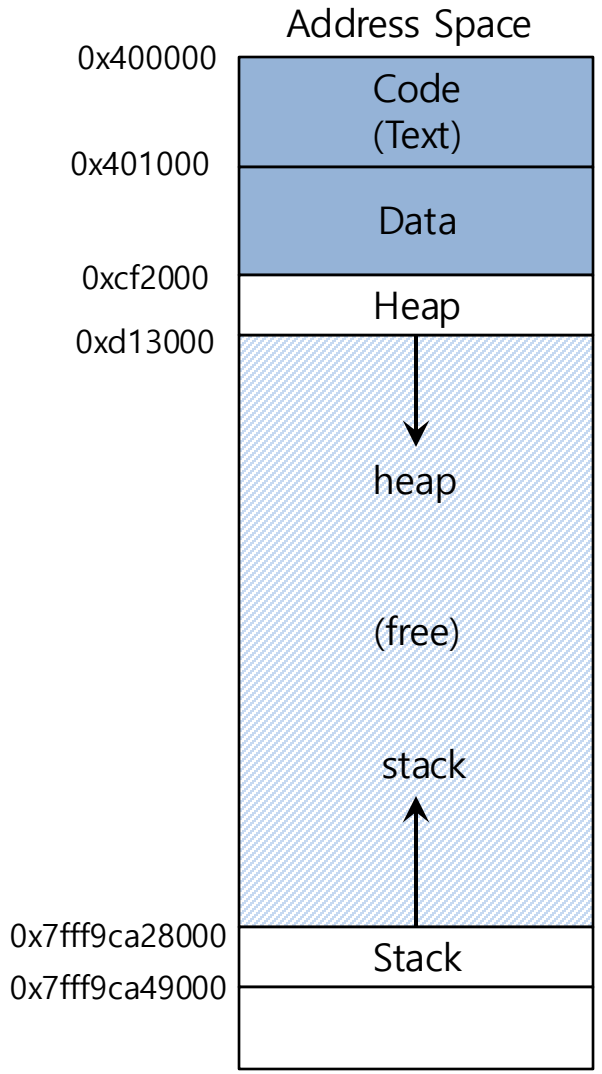
```c
#include <stdio.h>
#include <stdlib.h>

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;
}
```

A simple program that prints out addresses

Virtual Address

- The output in 64-bit Linux machine

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



Memory API: *malloc()*

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap
  - Argument
    - `size_t size` : size of the memory block(in bytes)
    - `size_t` is an unsigned integer type.
  - Return
    - Success : a void type pointer to the memory block allocated by `malloc`
    - Fail : a null pointer

Memory API: *sizeof()*

- Routines and macros are utilized for `size` in `malloc` instead typing in a number directly
- Two types of results of `sizeof` with variables
  - The actual size of 'x' is known at run-time

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

  - The actual size of 'x' is known at compile-time

```
int x[10];
printf("%d\n", sizeof(x));
```

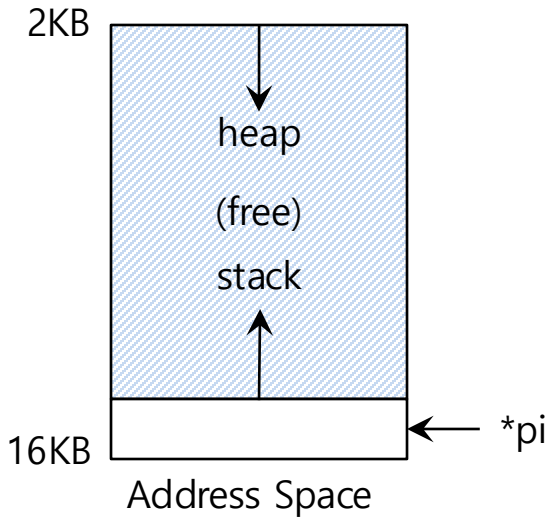
```
40
```

Memory API: *free()*

```
#include <stdlib.h>

void free(void* ptr)
```

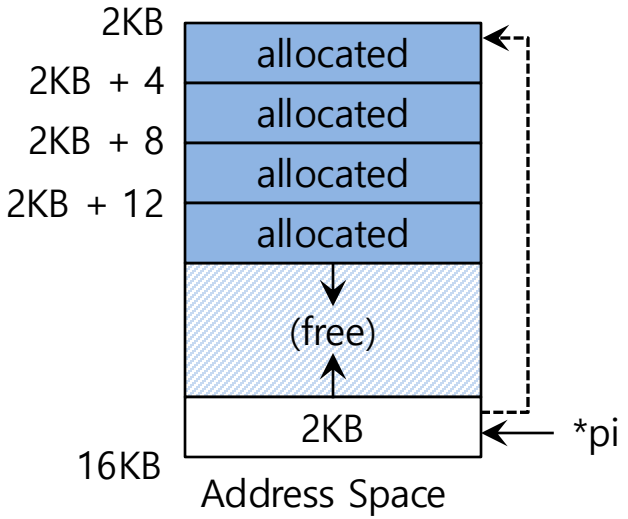
- Free a memory region allocated by a call to `malloc`
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

Memory Allocating



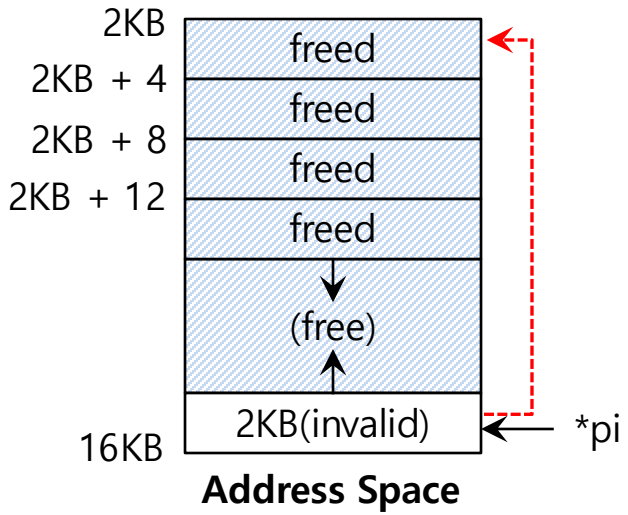
pointer

```
int *pi; // local variable
```

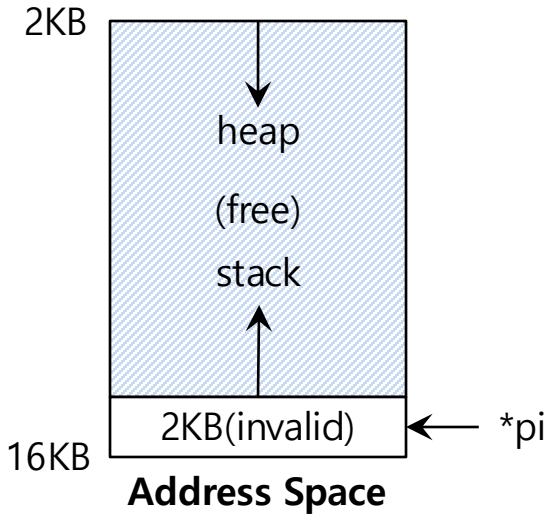



```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing



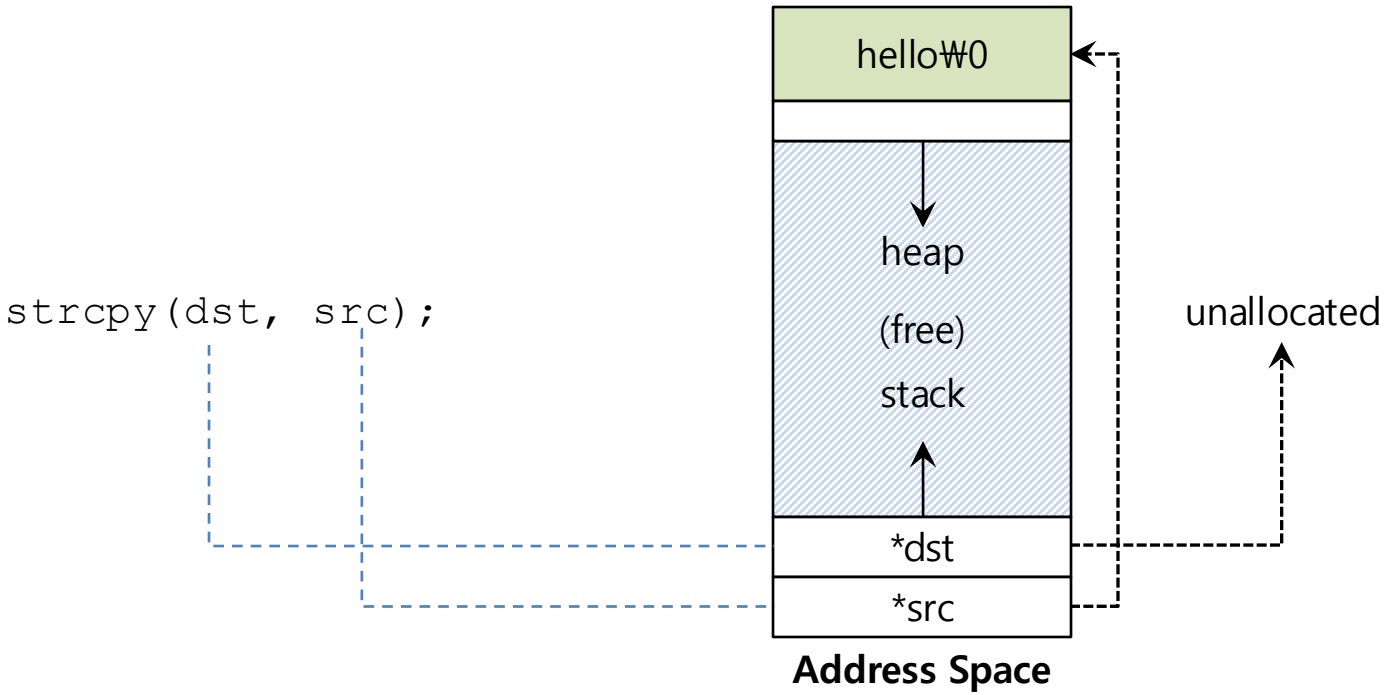

```
free(pi);
```



Forgetting To Allocate Memory

- Incorrect code

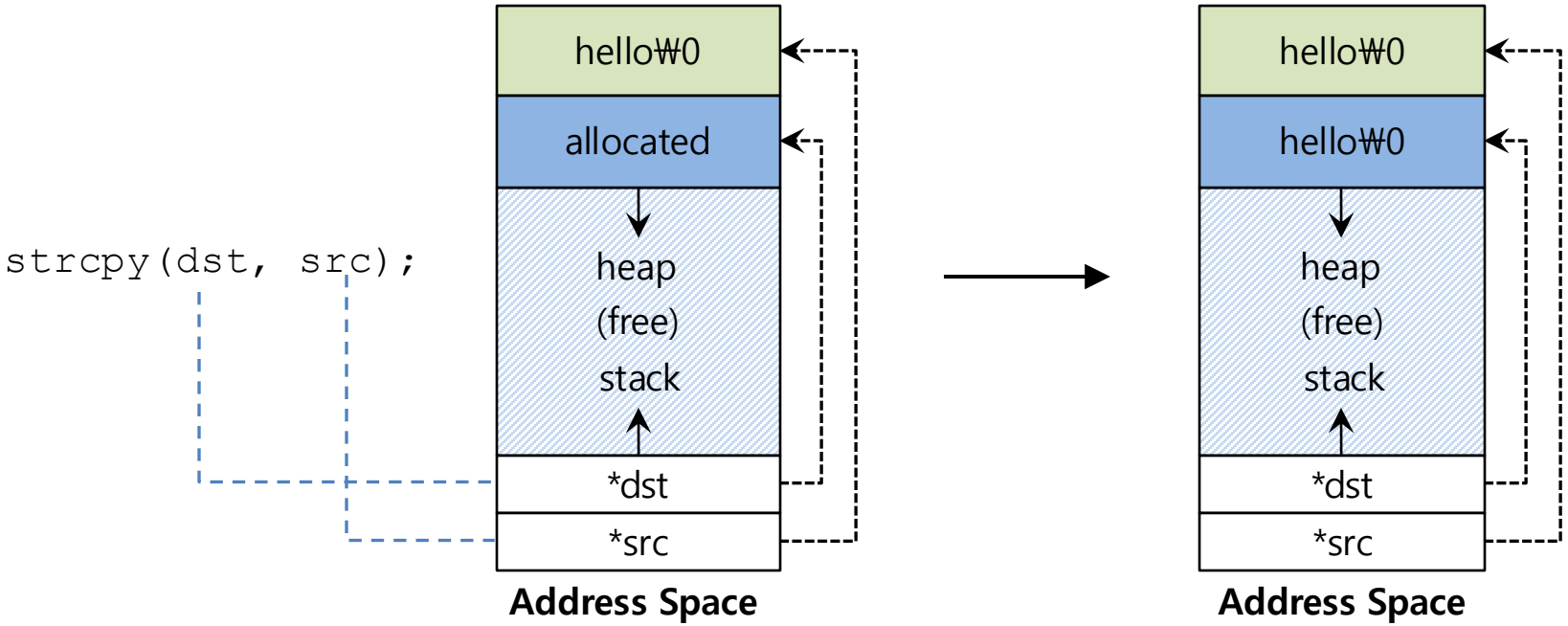
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory

- Correct code

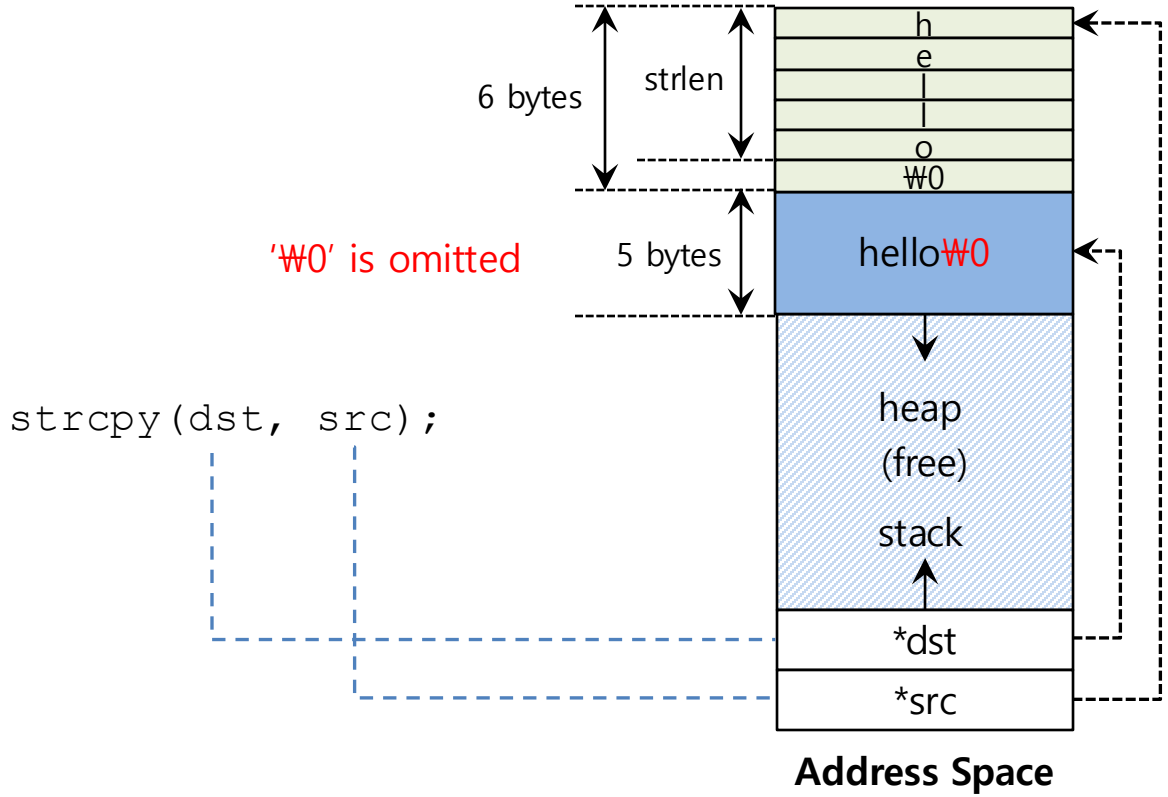
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory

- Incorrect code, but work properly

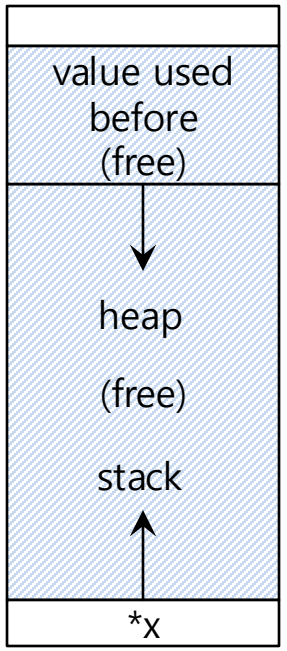
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```



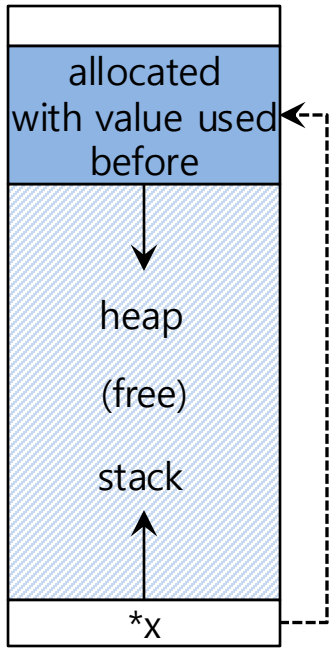
Forgetting to Initialize

- Encounter an uninitialized read

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```



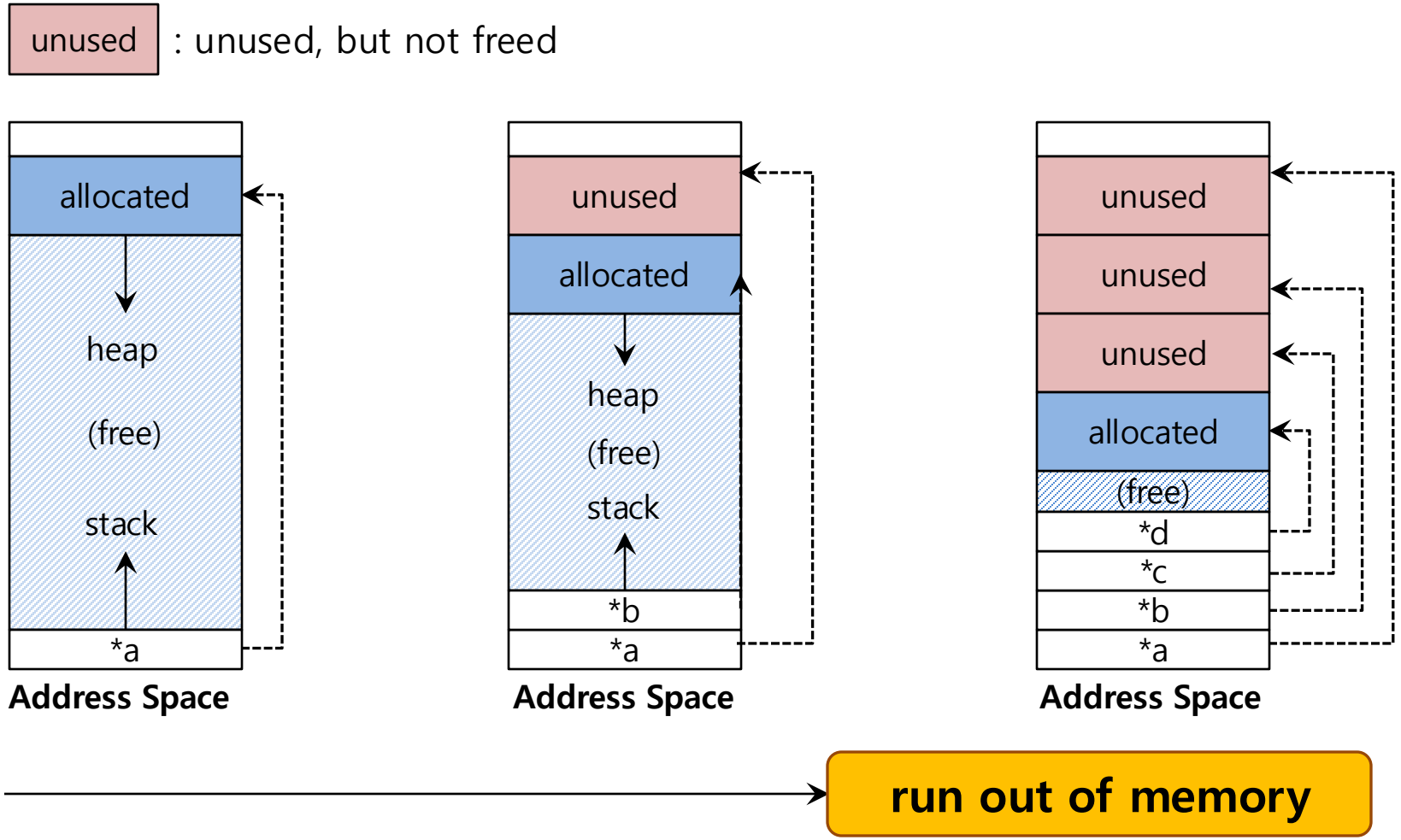
Address Space



Address Space

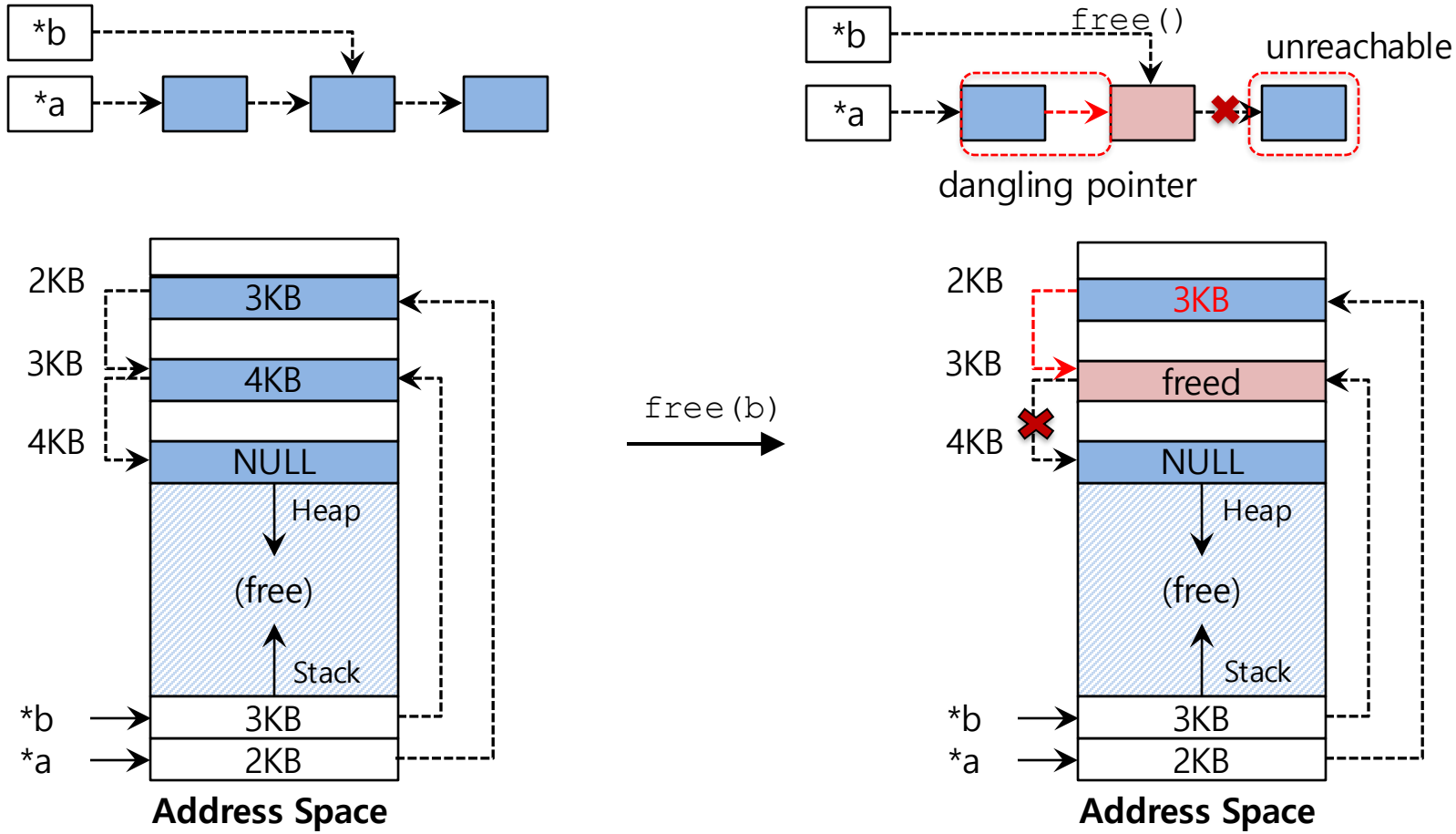
Memory Leak

- A program runs out of memory and eventually dies



Dangling Pointer

- Freeing memory before it is finished using
  - A program accesses to memory with an invalid pointer



Other Memory APIs: calloc()

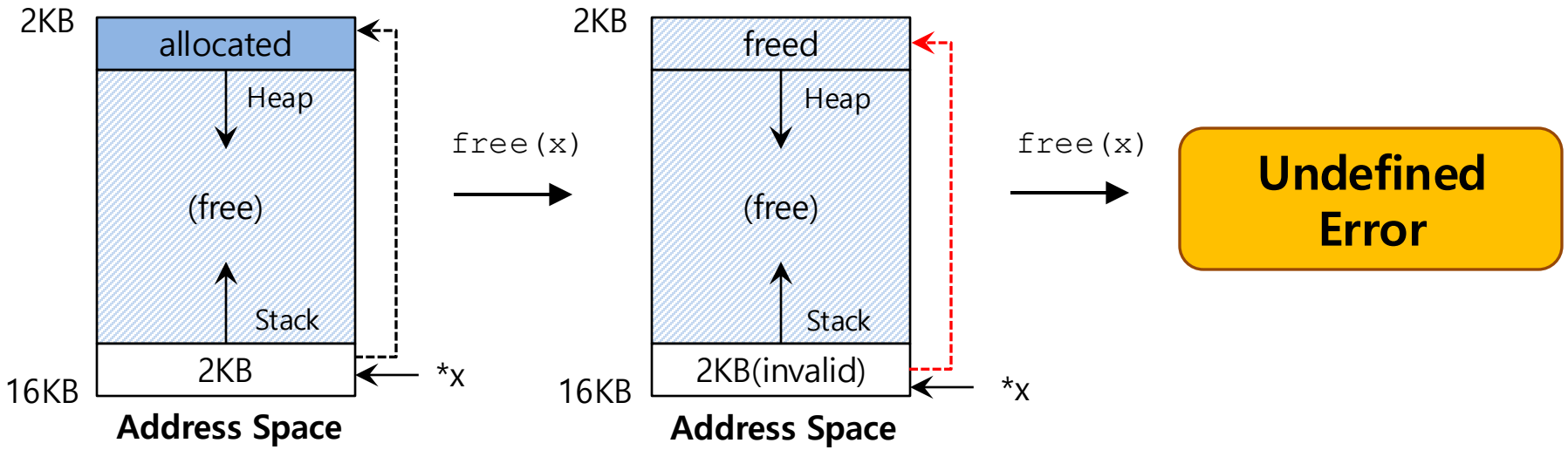
```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate memory on the heap and zeroes it before returning
  - Argument
    - `size_t num` : number of blocks to allocate
    - `size_t size` : size of each block(in bytes)
  - Return
    - Success : a void type pointer to the memory block allocated by `calloc`
    - Fail : a null pointer

Double Free

- Free memory that was freed already

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory
free(x); // free repeatedly
```



Other Memory APIs: realloc()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Change the size of memory block
  - A pointer returned by `realloc` may be either the same as `ptr` or a new
  - Argument
    - `void *ptr`: Pointer to memory block allocated with `malloc`, `calloc` or `realloc`
    - `size_t size`: New size for the memory block(in bytes)
  - Return
    - Success: Void type pointer to the memory block
    - Fail : Null pointer

System Calls for Allocation

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

- `malloc` library call use `brk` system call
  - `brk` is called to expand the program's *break*
    - *break*: The location of **the end of the heap** in address space
  - `sbrk` is an additional call similar with `brk`
  - Programmers **should never directly call** either `brk` or `sbrk`

System Calls for Memory Mapping

```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags,
int fd, off_t offset)
```

- `mmap` system call can create **an anonymous** memory region

Memory Virtualizing with Efficiency and Control

- Memory virtualizing takes a similar strategy known as **limited direct execution(LDE)** for efficiency and control
- In memory virtualizing, efficiency and control are attained by hardware support
  - e.g., registers, TLB(Translation Look-aside Buffer)s, page-table

Address Translation

- Hardware transforms a **virtual address** to a **physical address**
  - The desired information is actually stored in a physical address

- The OS must get involved at key points to set up the hardware
  - The OS must manage memory to judiciously intervene

Example: Address Translation

- C - Language code

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

- **Load** a value from memory
- **Increment** it by three
- **Store** the value back into memory

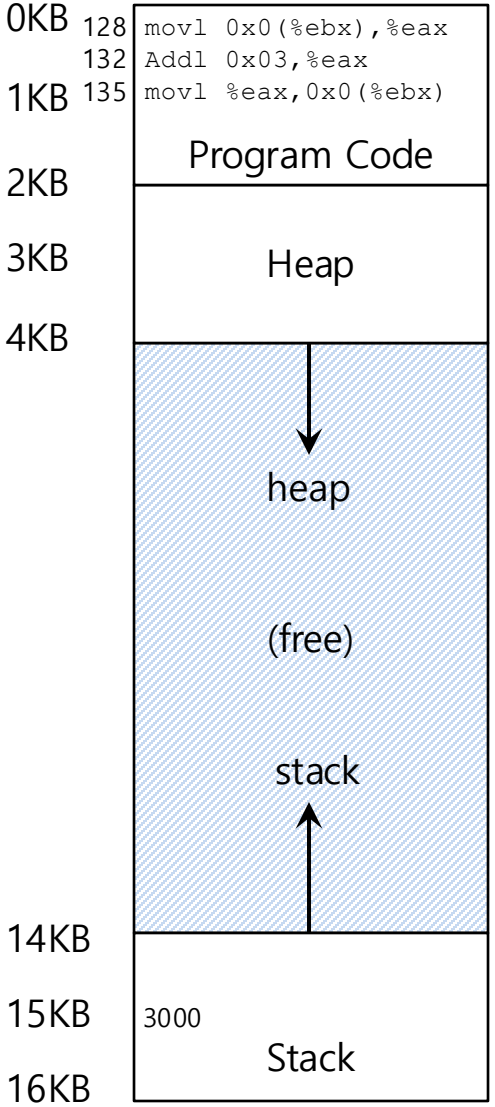
Example: Address Translation

- Assembly

```
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
```

- Presume that the address of 'x' has been place in ebx register.
- **Load** the value at that address into eax register.
- **Add** 3 to eax register.
- **Store** the value in eax back into memory.

Example: Address Translation

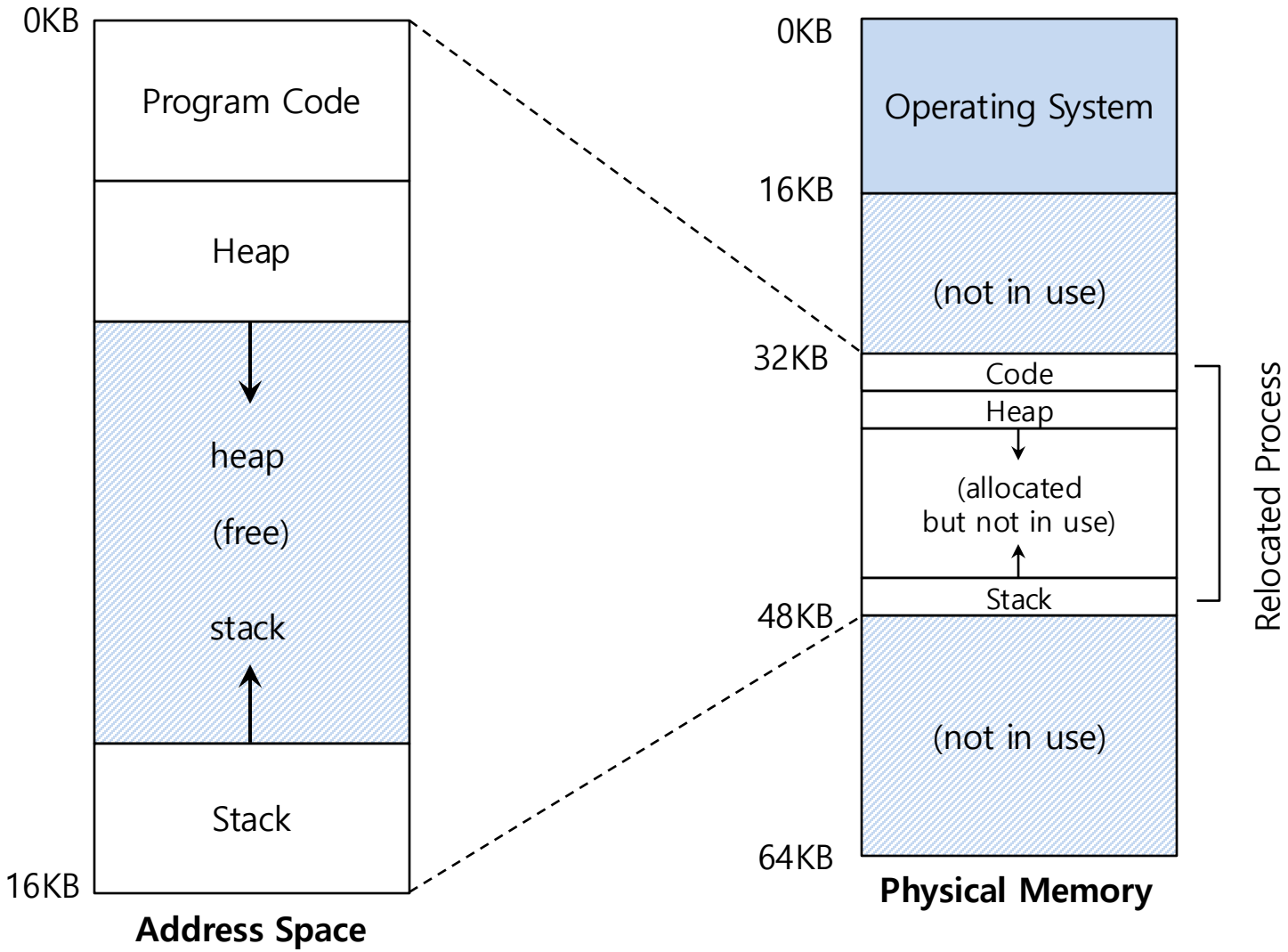


- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (store to address 15 KB)

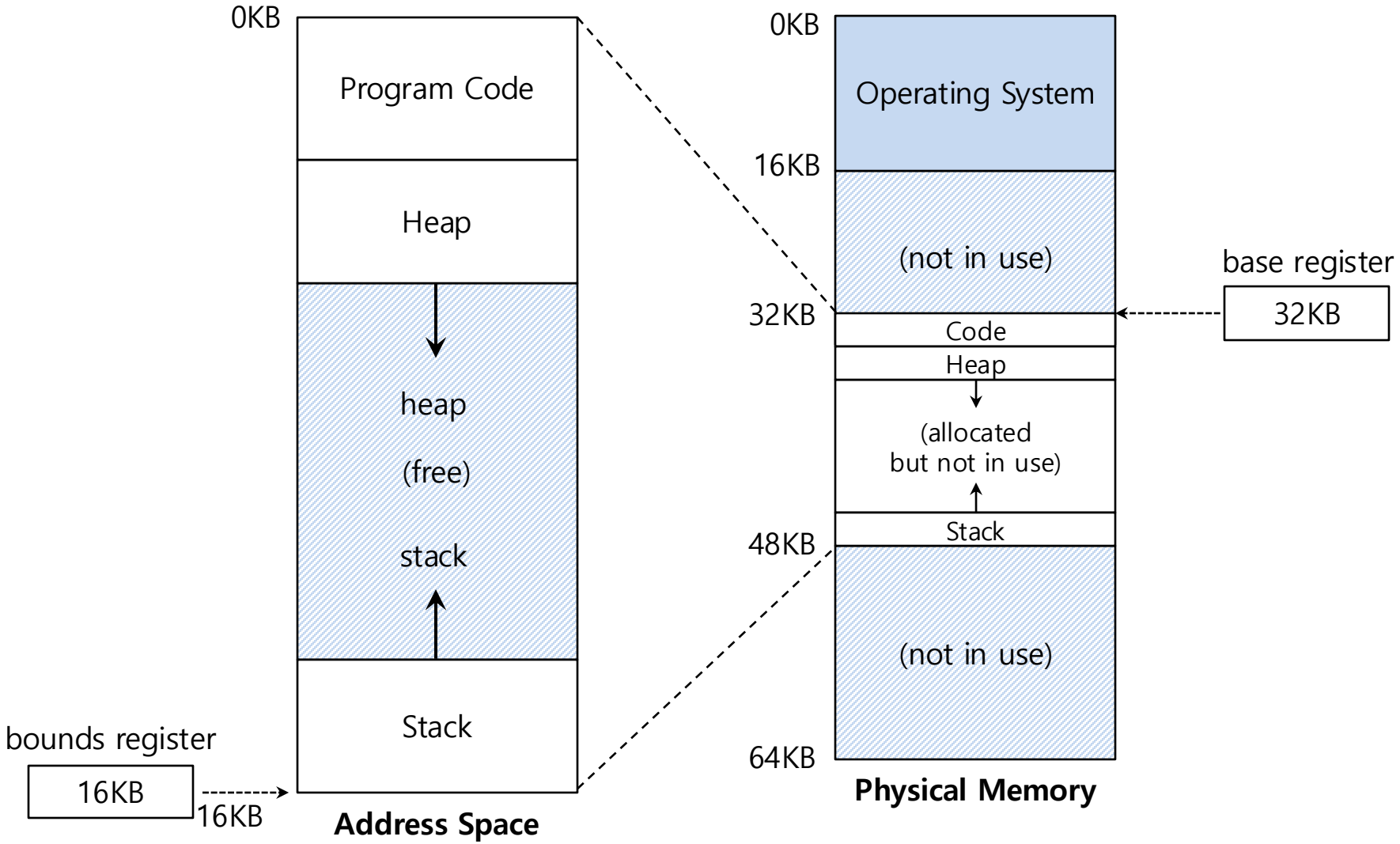
Relocation Address Space

- The OS wants to place the process **somewhere else** in physical memory, not at address 0
  - The address space start at address 0

A Single Relocated Process



Base and Bounds Register



Dynamic(Hardware base) Relocation

- When a program starts running, the OS decides **where** in physical memory a process should be **loaded**
  - Set the **base** register a value

$$phycal\ address = virtual\ address + base$$

  - Every virtual address must **not be greater than bound** and **negative**

$$0 \leq\ virtual\ addressvirtual\ address < bounds$$

Relocation and Address Translation

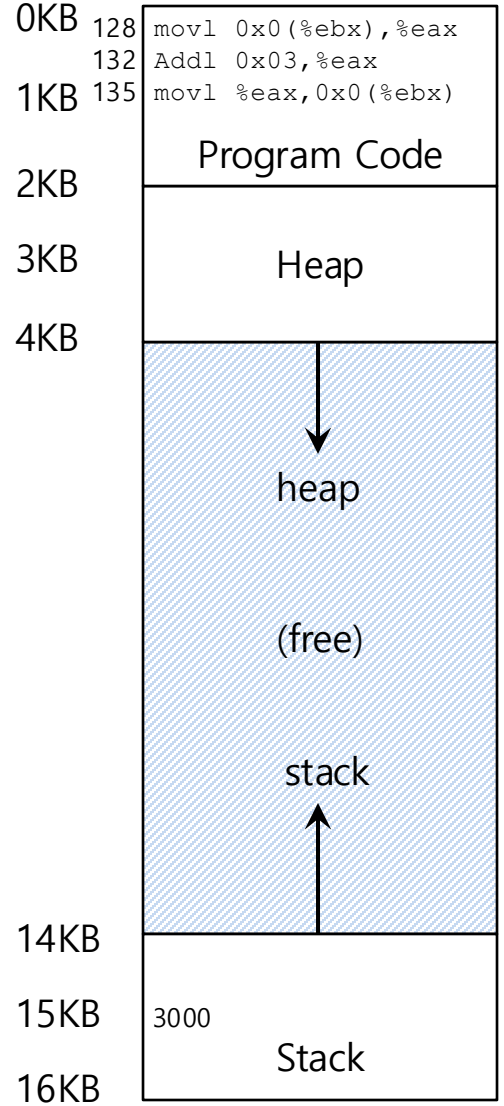
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

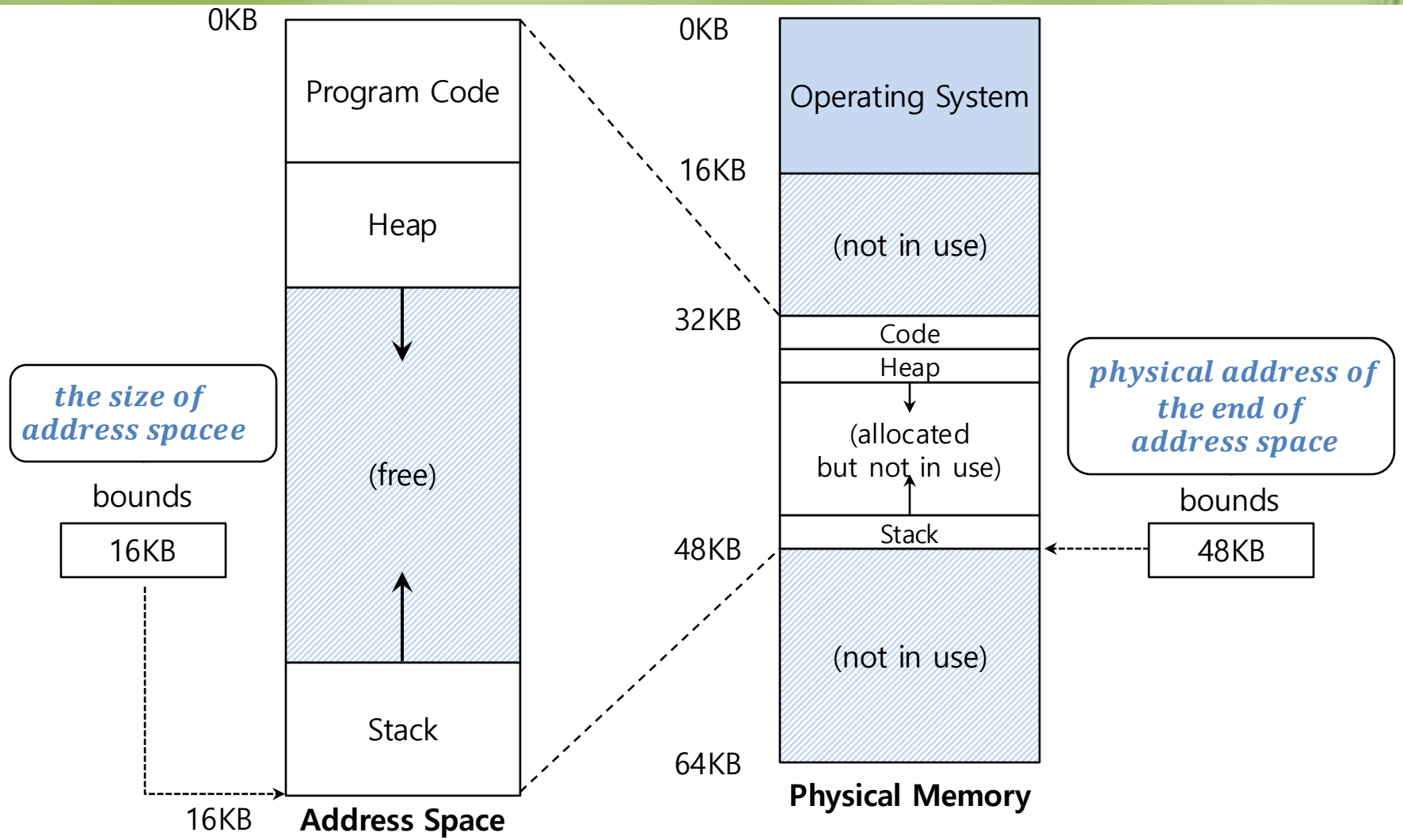
$$32896 = 128 + 32KB(base)$$

- **Execute** this instruction
  - Load from address 15KB

$$47KB = 15KB + 32KB(base)$$



Two ways of Bounds Register

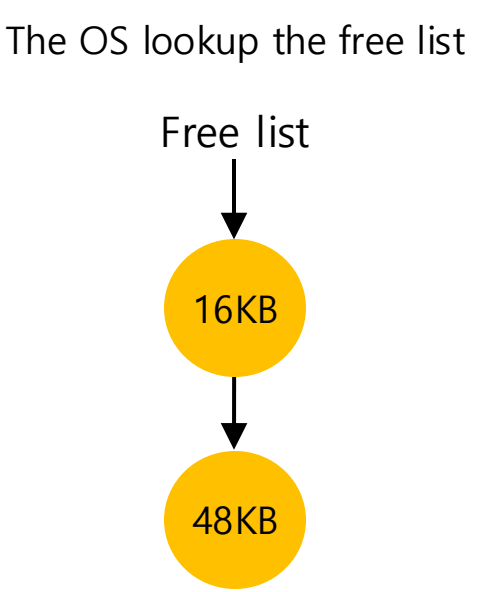


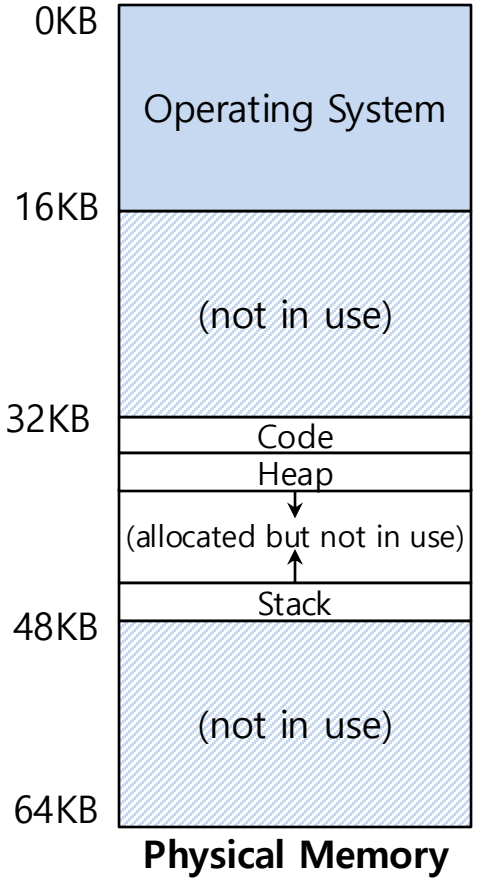
OS Issues for Memory Virtualizing

- The OS must **take action** to implement **base-and-bounds** approach
- Three critical junctures:
  - When a process **starts running:**
    - Finding space for address space in physical memory
  - When a process is **terminated:**
    - Reclaiming the memory for use
  - When context **switch occurs:**
    - Saving and storing the base-and-bounds pair

OS Issues: When a Process Starts Running

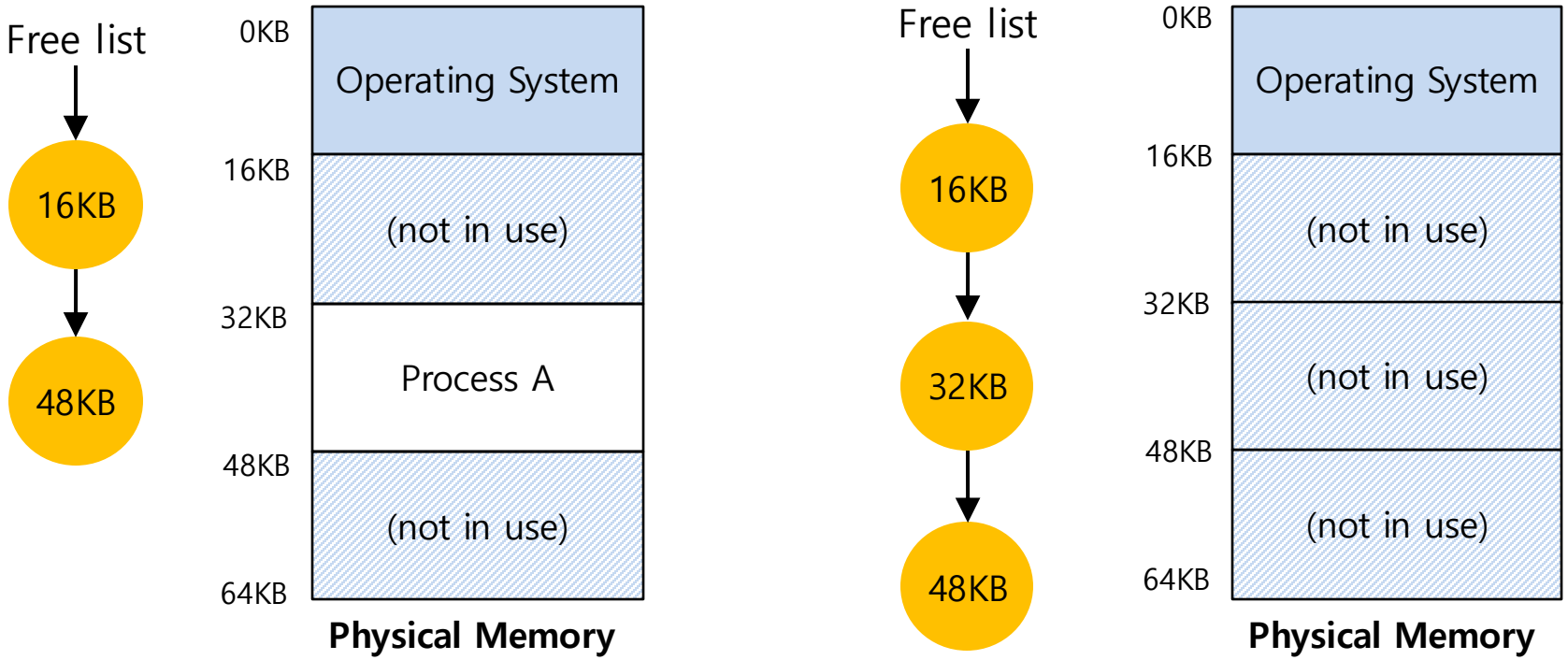
- The OS must **find a room** for a new address space
  - free list : A list of the range of the physical memory which are not in use





OS Issues: When a Process Is Terminated

- The OS must **put the memory back** on the free list



OS Issues: When Context Switch Occurs

- OS must **save and restore** the base-and-bounds pair
  - In **process structure** or **process control block(**PCB)