# OSTEP
# Memory Virtualization

# Memory API &
# Address Translation

# Memory API

# 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

# 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`.

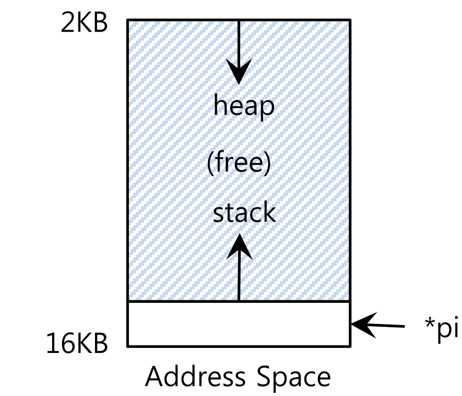
- Argument

  `void *ptr` : a pointer to a memory block allocated with `malloc`

- Return

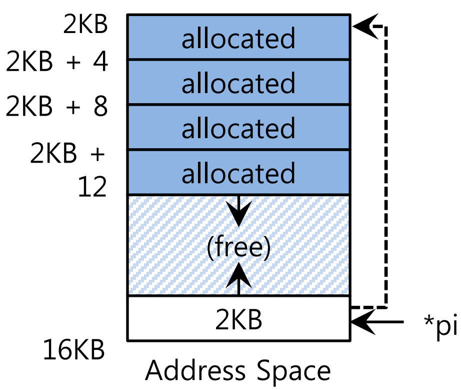
  none

# Memory Allocating



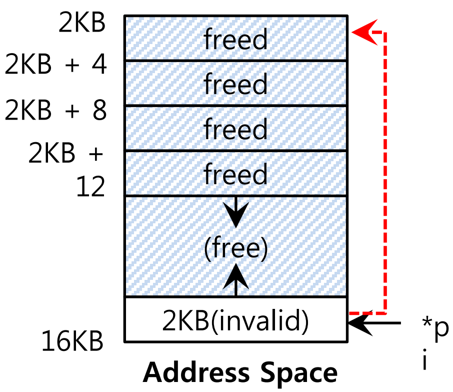

pointer

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

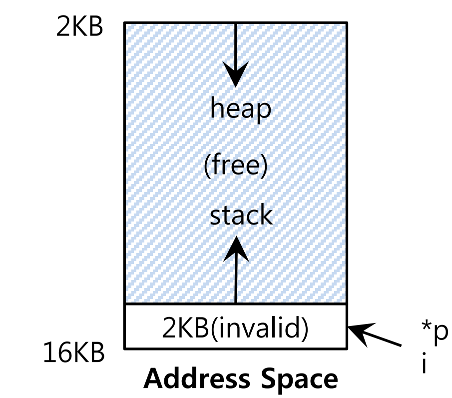



```
pi = (int *)malloc(sizeof(int)* 4);
```

# Memory Freeing



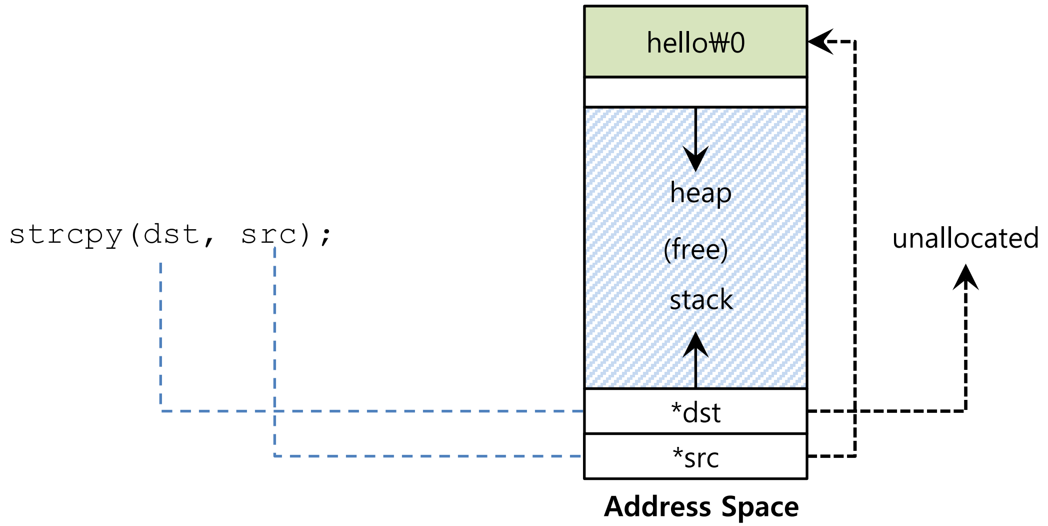

```
free(pi);
```



# Forgetting To Allocate Memory

## Incorrect code

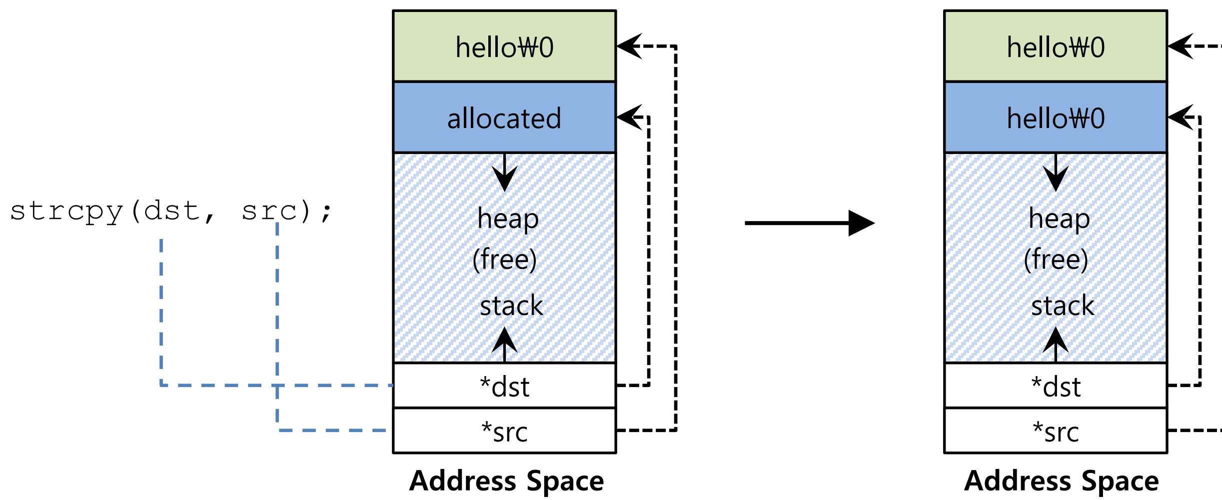
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



# Forgetting To Allocate Memory(Cont.)

## Correct code

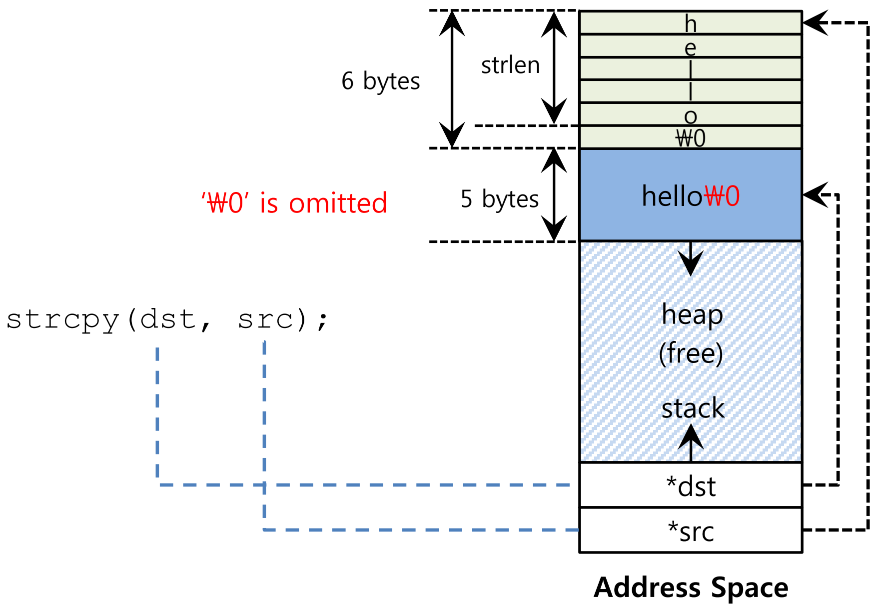
```
char *src = "hello";        //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);   //work properly
```



# Not Allocating Enough Memory

## Incorrect code, but work properly

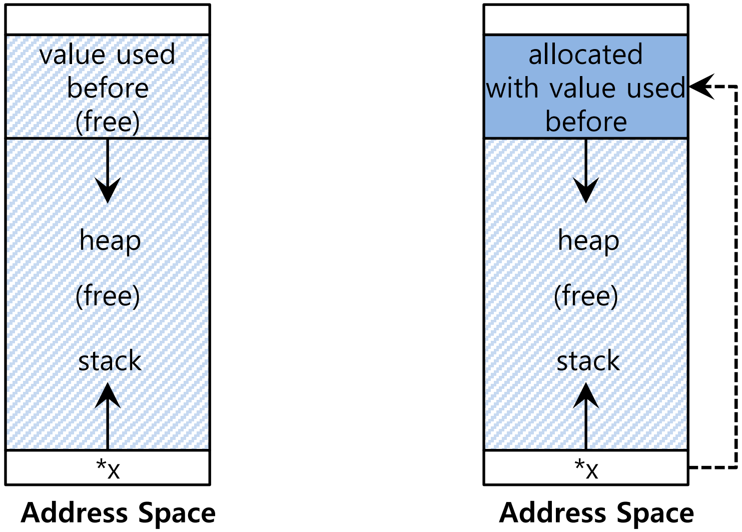
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);     //work properly
```



# Forgetting to Initialize

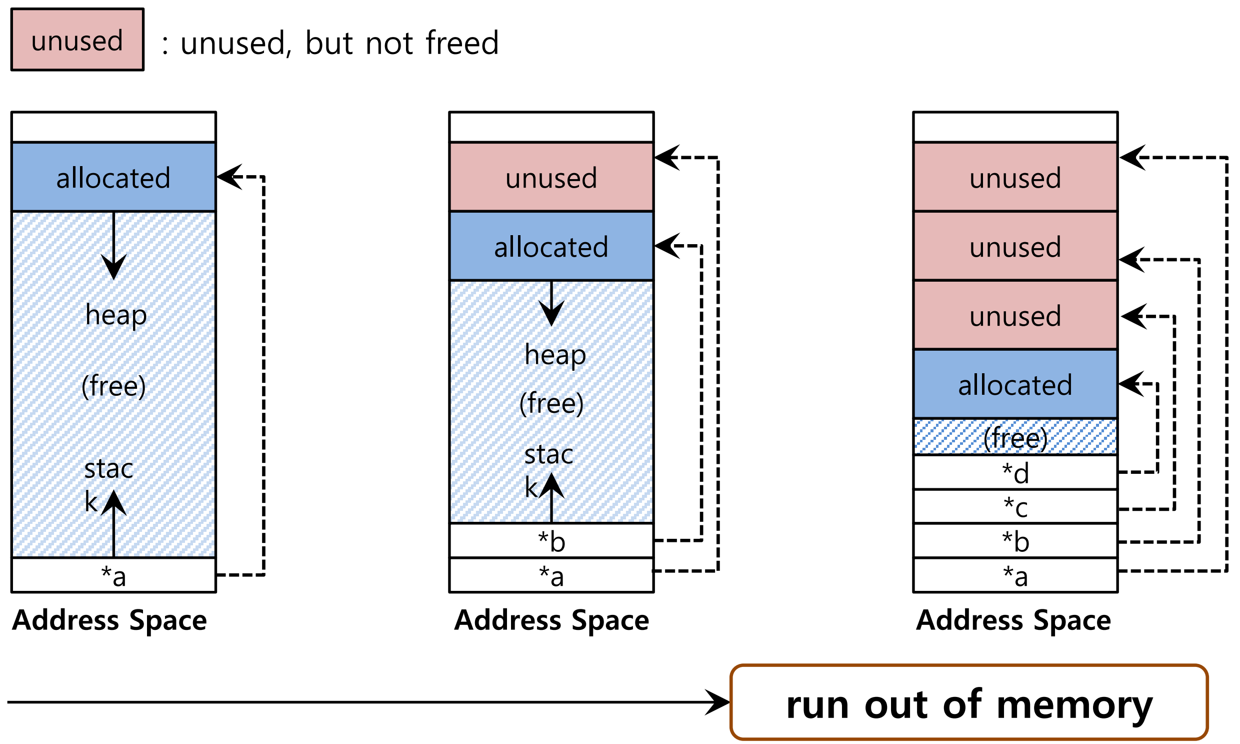
## Encounter an uninitialized read

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



# Memory Leak

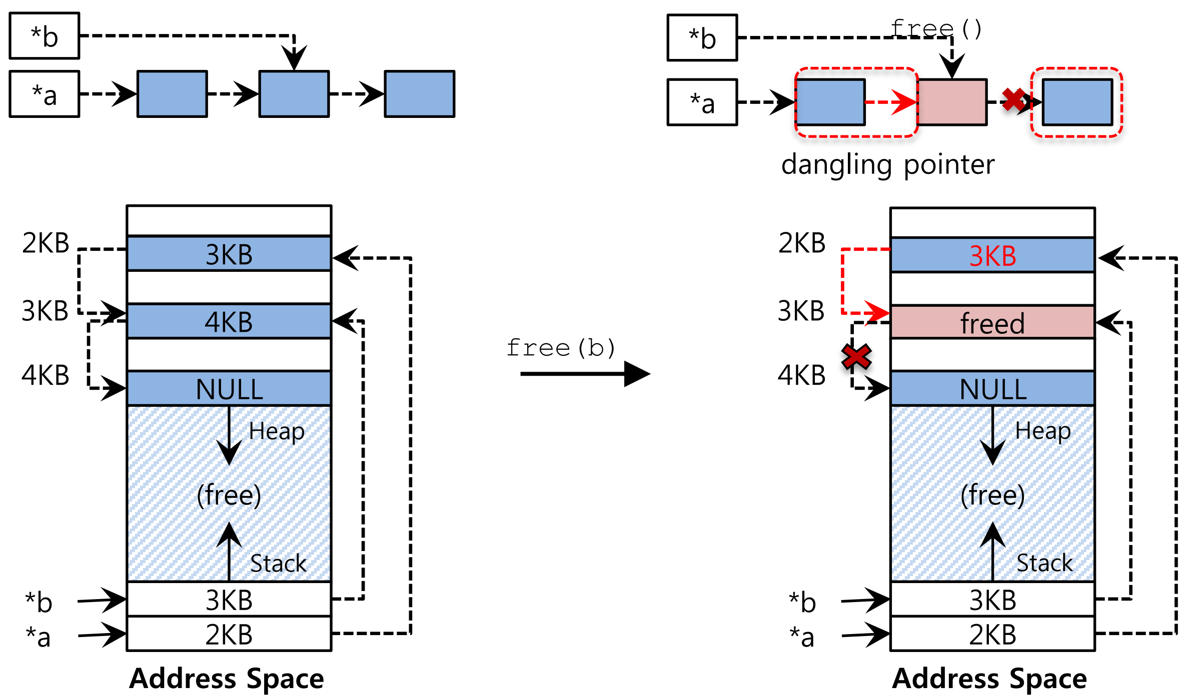
A program runs out of memory and eventually dies



# 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)
```

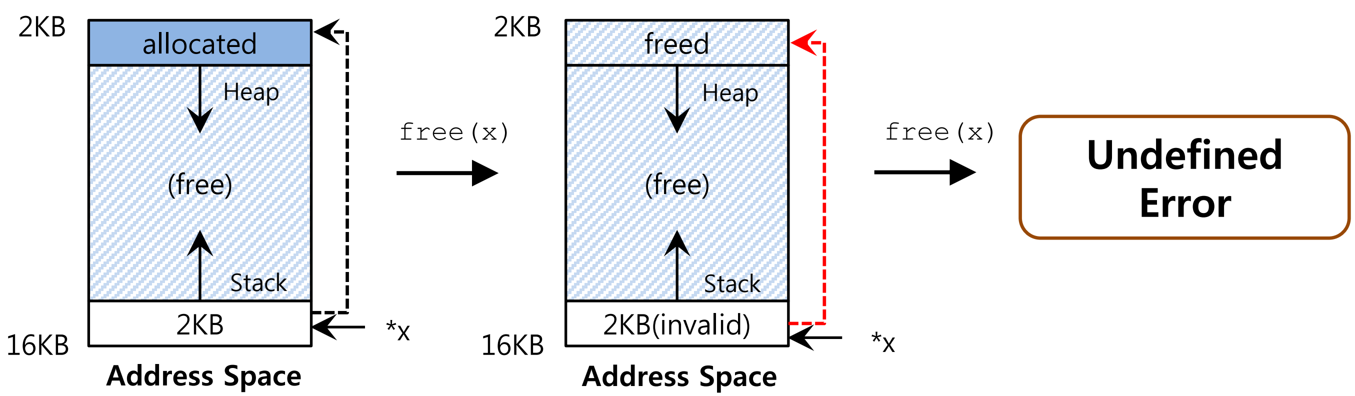
Allocate memory on the heap and zeroes it before returning.

- 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()

```c
#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

```
#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(Cont.)

```
#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.

# Address Translation

# 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

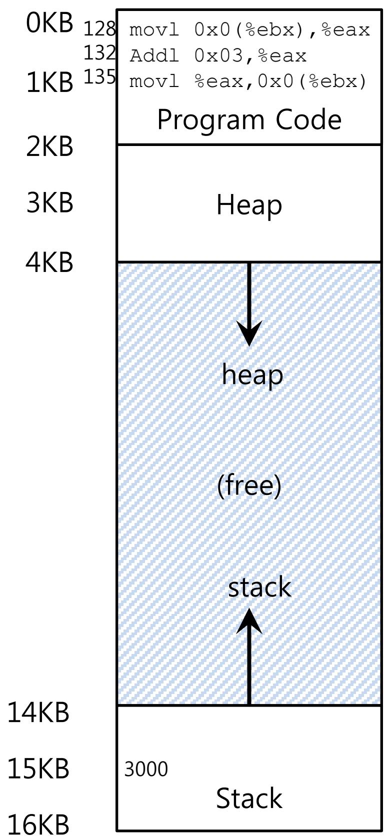
# Example: Address Translation(Cont.)

## 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(Cont.)



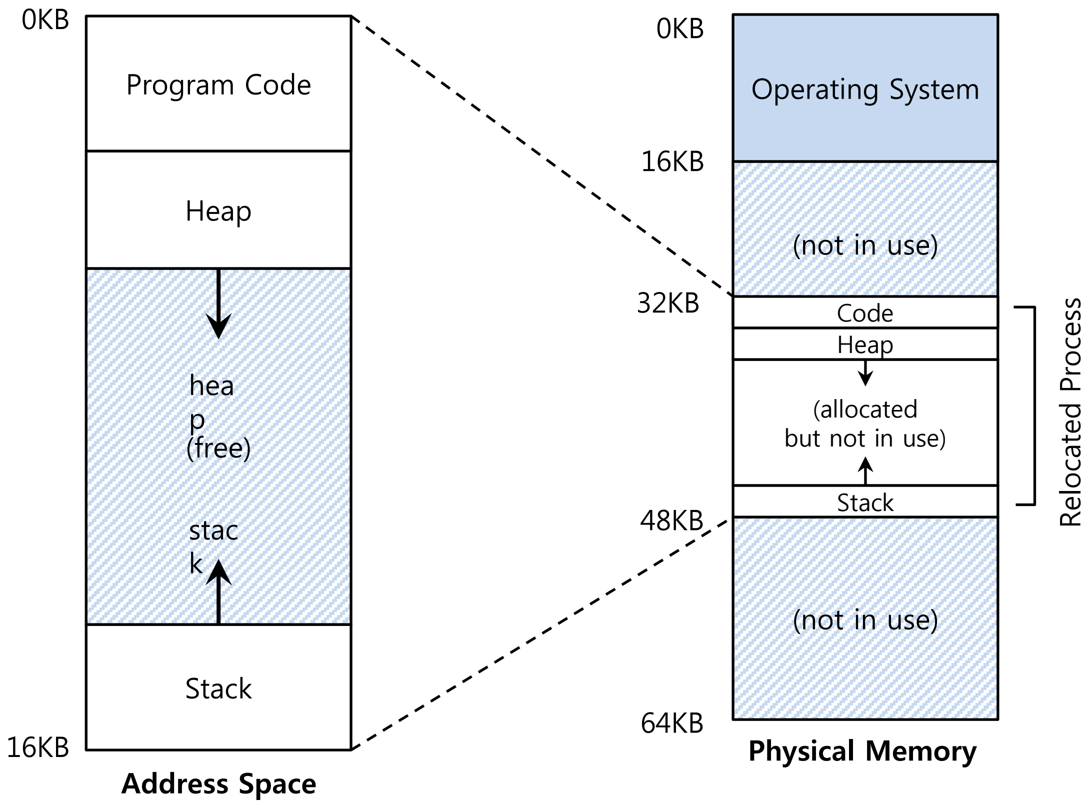

- Fetch instruction at address 128
- Execute this instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute this instruction (no memory reference)
- Fetch the instruction at address 135
- Execute this instruction (Store to address 15 KB)

# Relocation Address Space

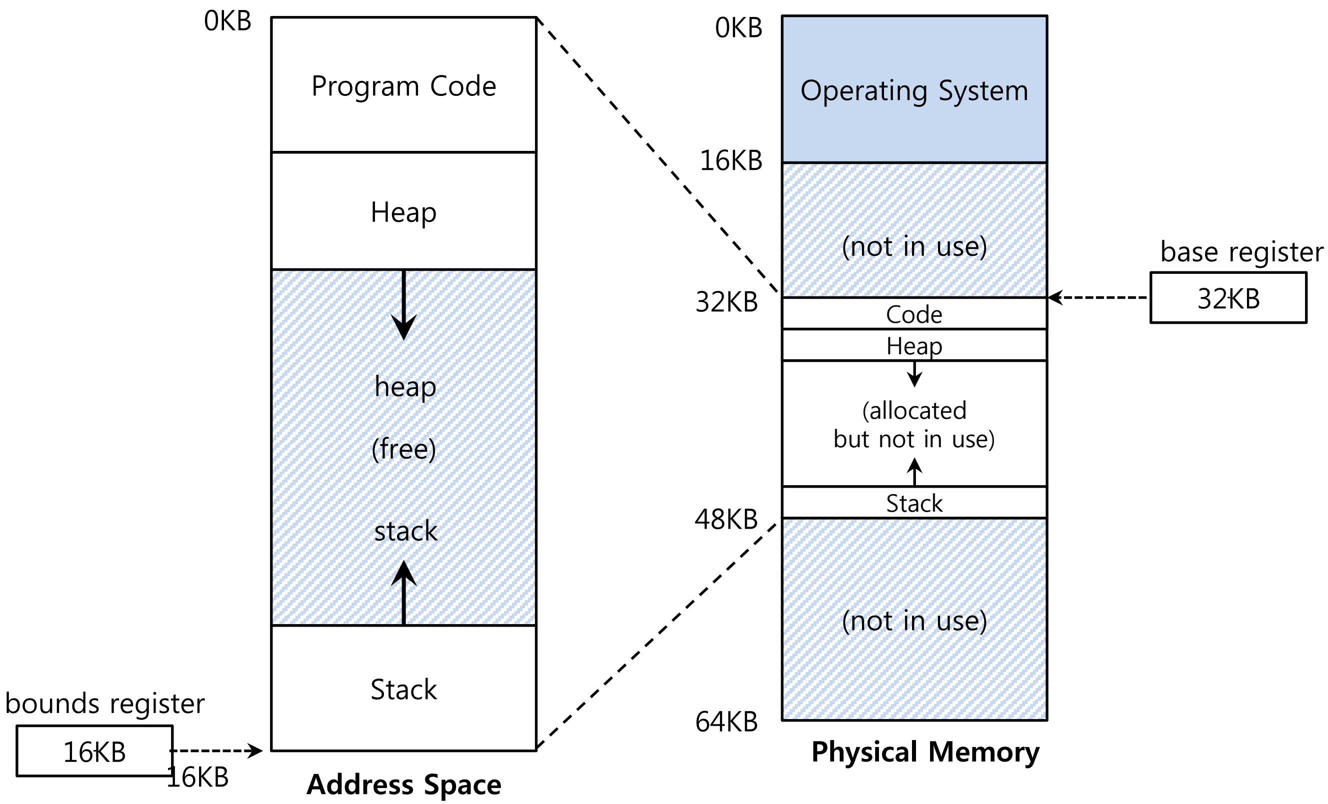
The OS wants to place the process **somewhere else** in physical memory, not at address 0.

- 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

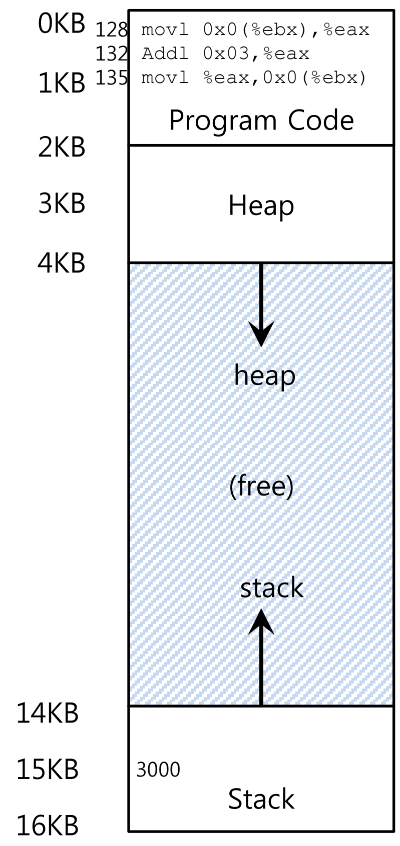
```
128 : movl 0x0(%ebx), %eax
```

- **Fetch** instruction at address 128

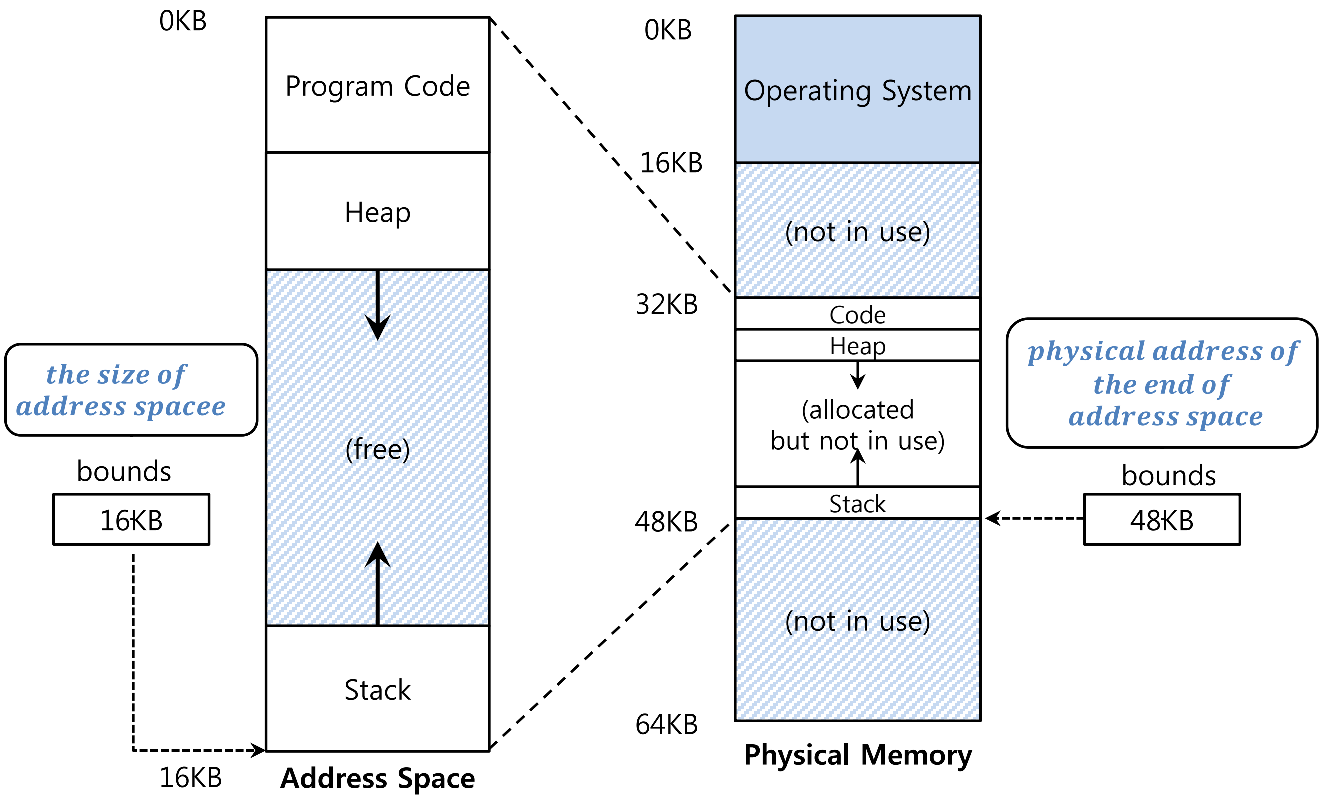
$$32896 = 128 + 32KB(base)$$

- **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

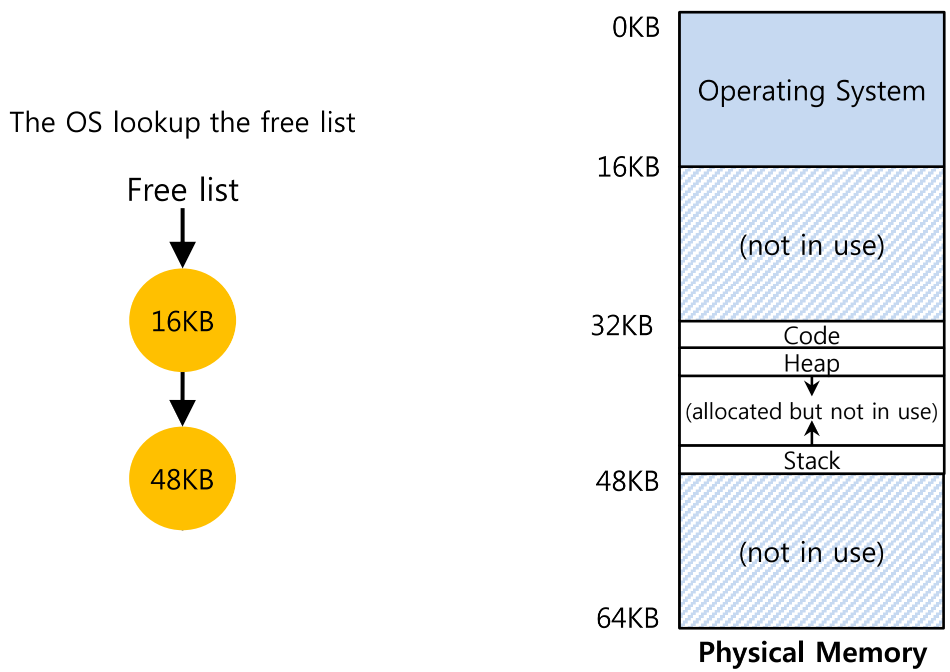
Three critical junctures:

- 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

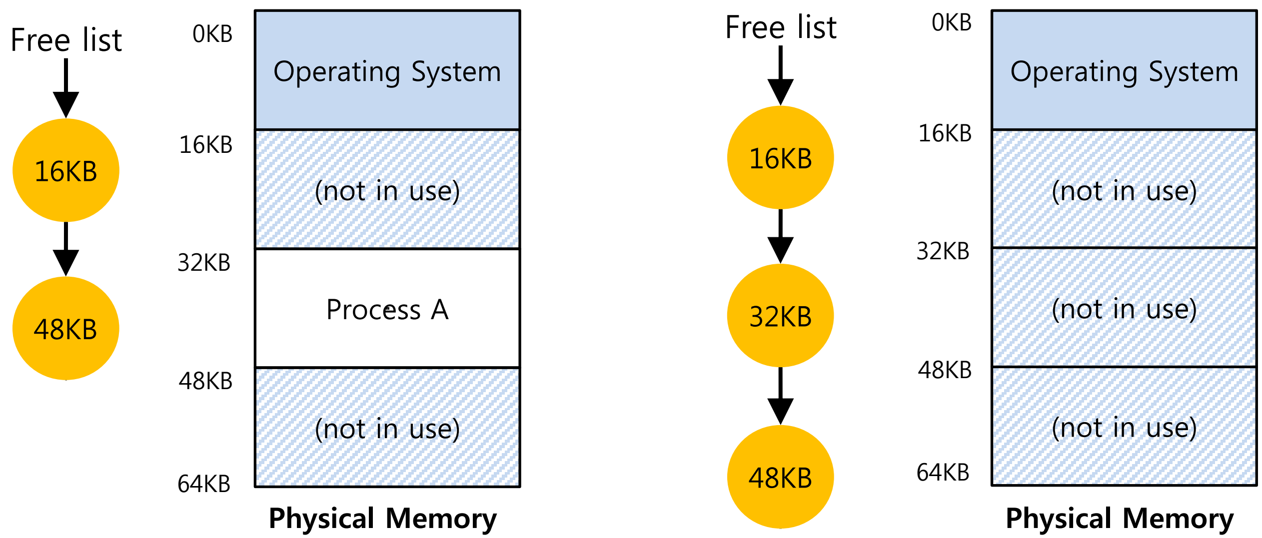
The OS must **find a room** for a new address space.

- 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

The OS must **save and restore** the base-and-bounds pair.

- In **process structure** or **process control block(**PCB)