# 21CS2109AA

Operating Systems

Session 10

Memory Virtualization : Address Spaces, Memory API

# Memory Virtualization

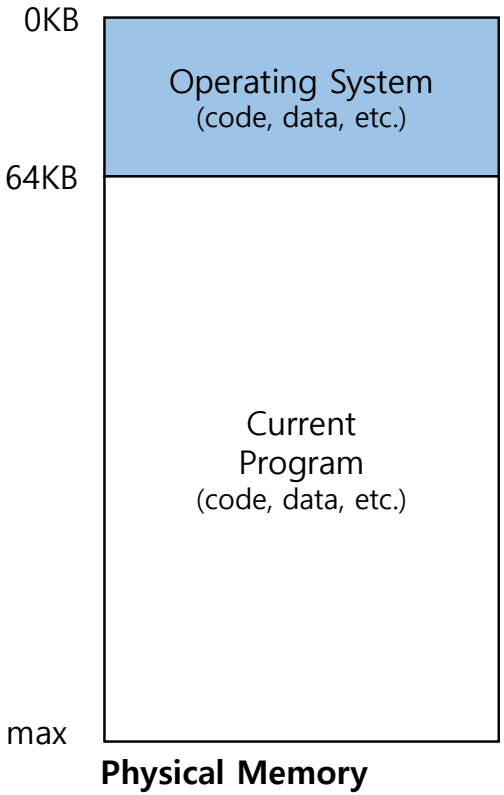
- What is **memory virtualization**?
  - OS virtualizes its physical memory.
  - OS provides an illusion memory space per each process.
  - It seems to be seen like each process uses the whole memory .

# 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

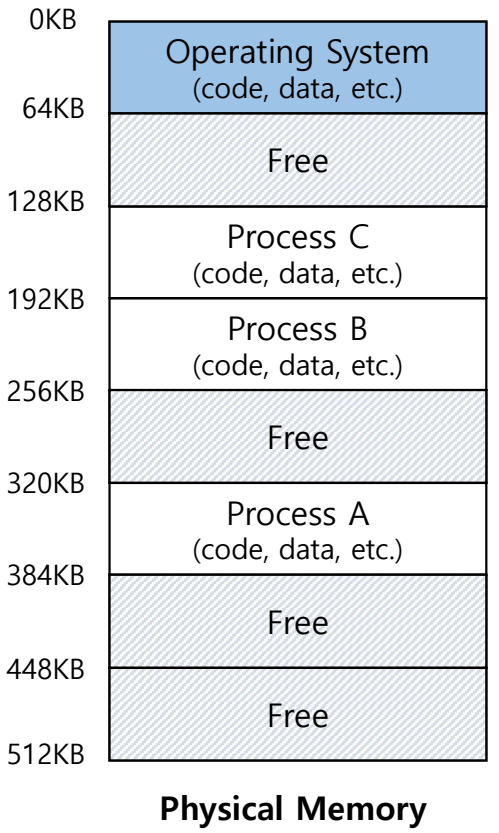
- Load only one process in memory.
  - Poor utilization and efficiency



# 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.

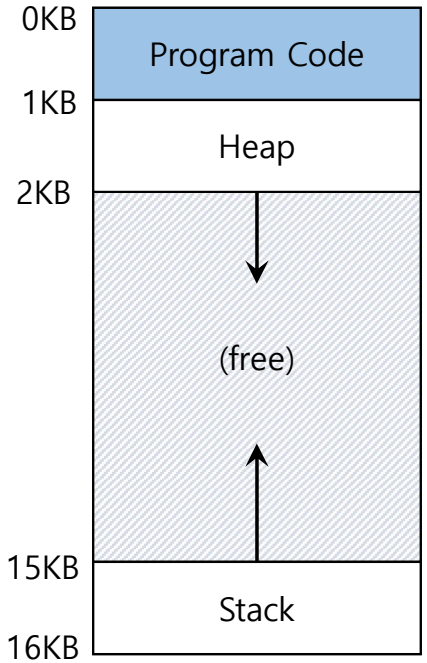
- Cause an important **protection issue**.
  - Errant memory accesses from other processes




**Physical Memory**

# Address Space

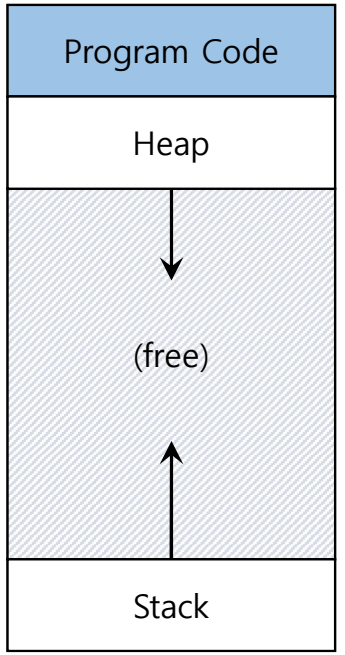
- OS creates an **abstraction** of physical memory.
  - The address space contains all about a running process.
  - That is consist of program code, heap, stack and etc.



**Address Space**

# Address Space(Cont.)

- 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

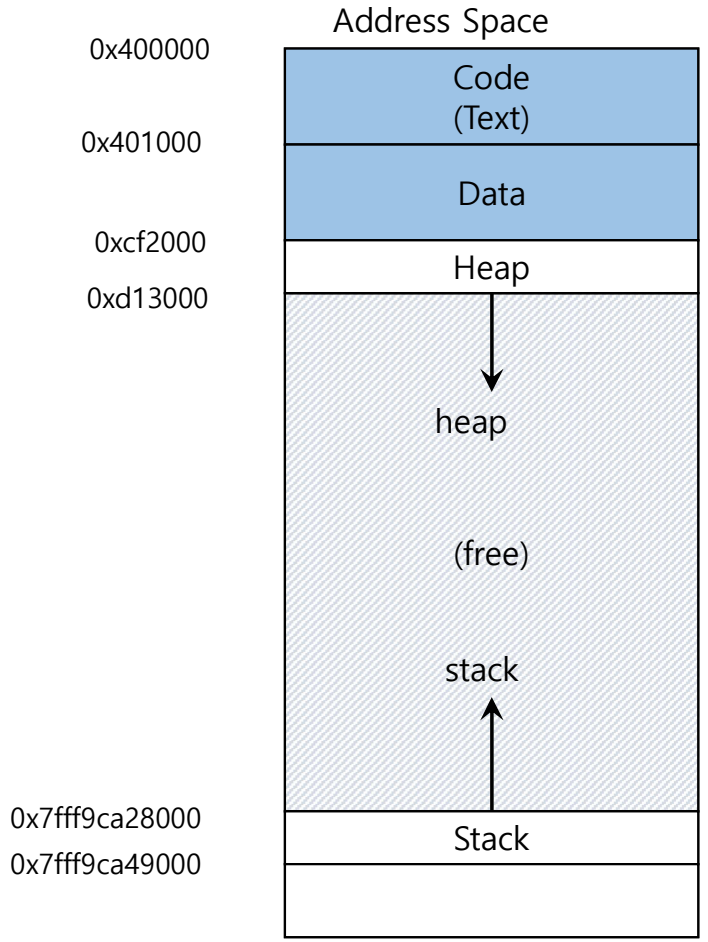
```
#include <stdio.h>
#include <stdlib.h>

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(Cont.)

- 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

# 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));
```

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

```
4
```

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

```
40
```

- When *sizeof()* is used with the data types such as int, float, char… etc it simply returns the amount of memory is allocated to that data types.

```
#include <stdio.h>
int main()
{
   printf("%lu\n", sizeof(char));
   printf("%lu\n", sizeof(int));
   printf("%lu\n", sizeof(float));
   printf("%lu", sizeof(double));
   return 0;
}
```

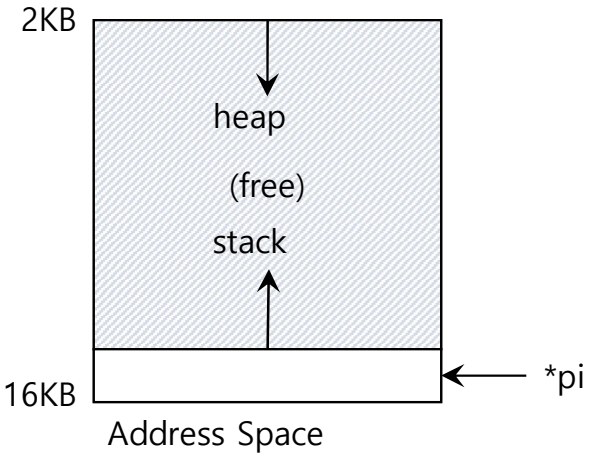
**Output:**1 4 4 8

# Memory API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

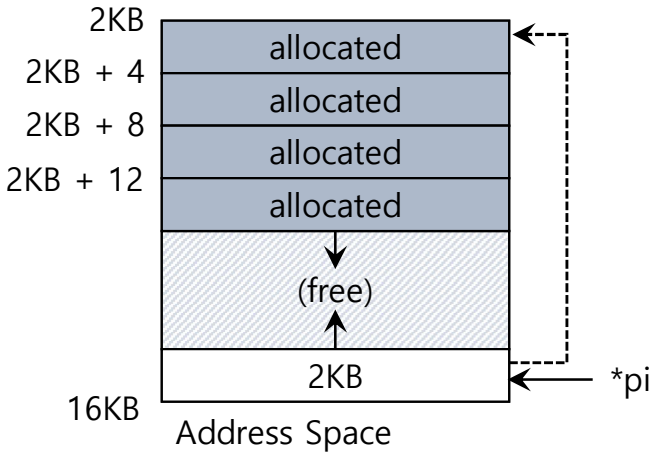
- Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

# Memory Allocating



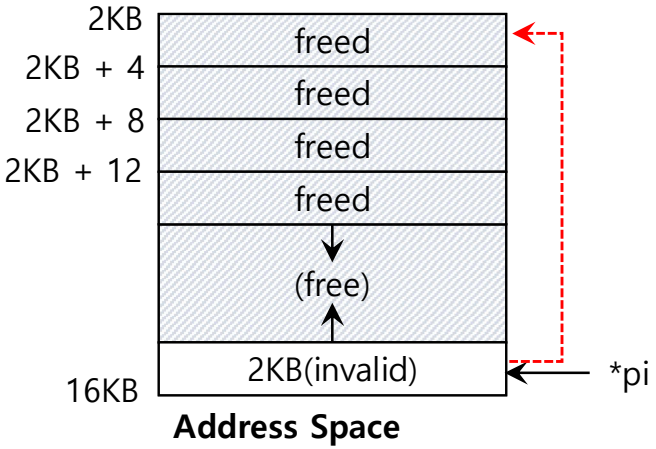

pointer

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

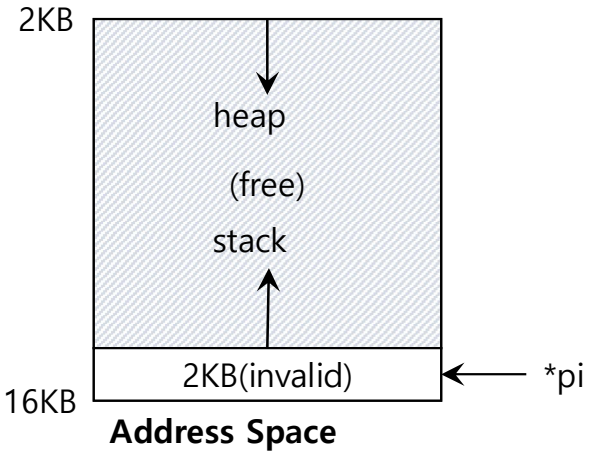



```
pi = (int *)malloc(sizeof(int)* 4);
```

# Memory Freeing



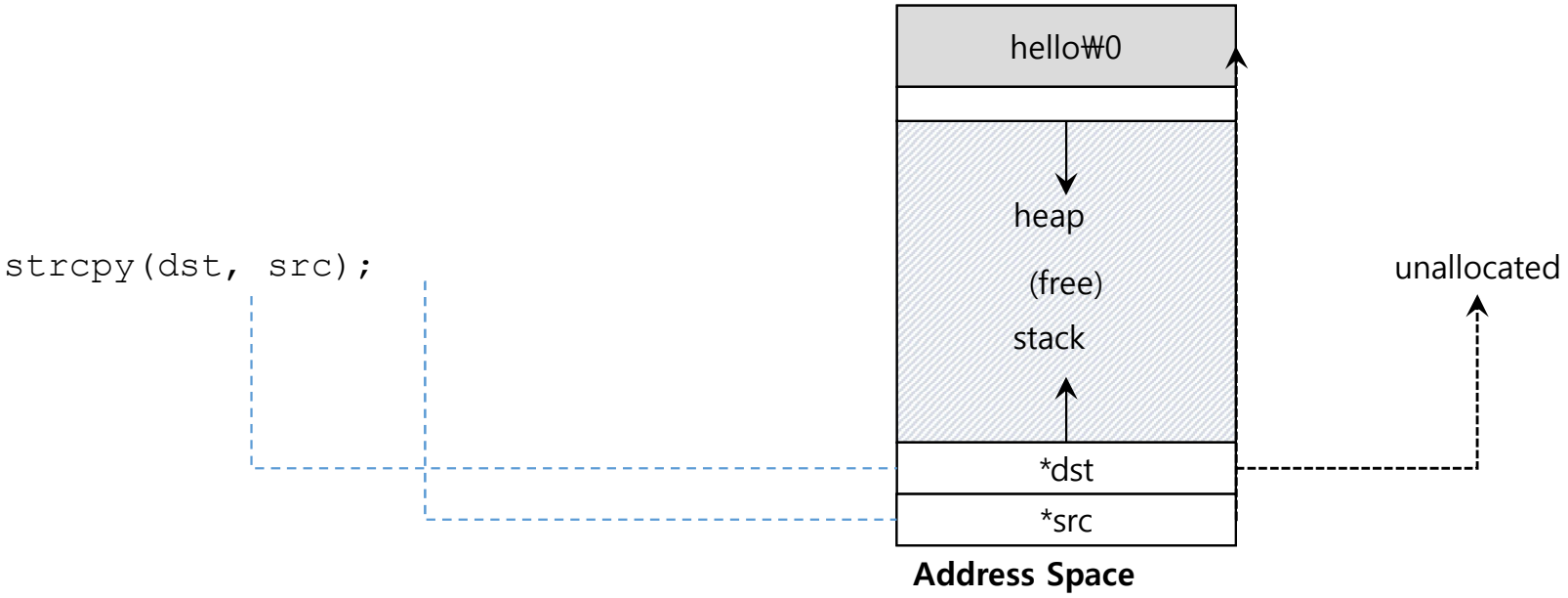

```
free(pi);
```



# Forgetting To Allocate Memory

- Incorrect code

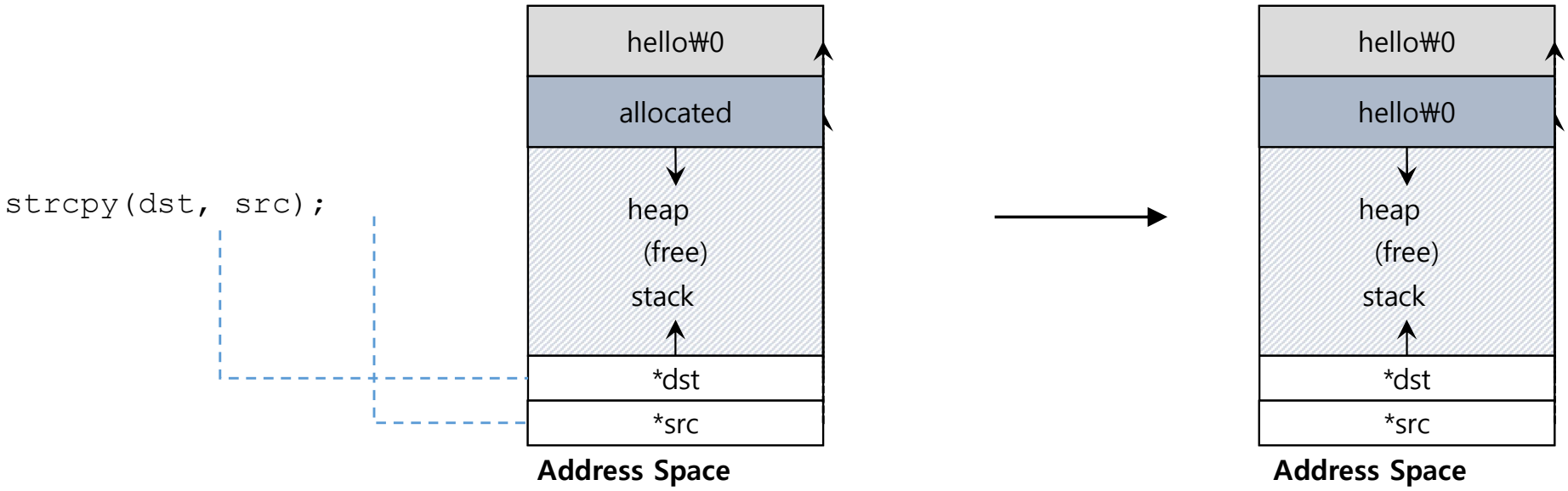
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



# Forgetting To Allocate Memory(Cont.)

- Correct code

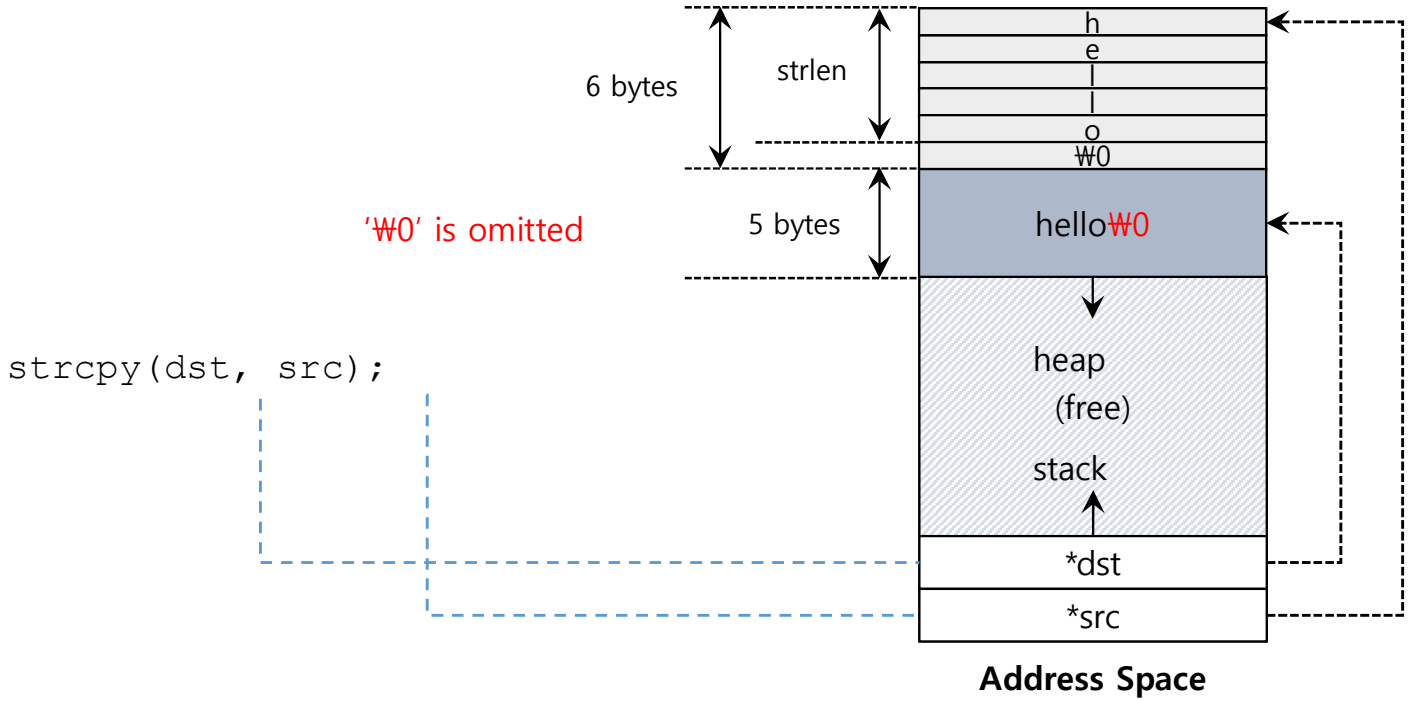
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



# Not Allocating Enough Memory

- Incorrect code, but work properly

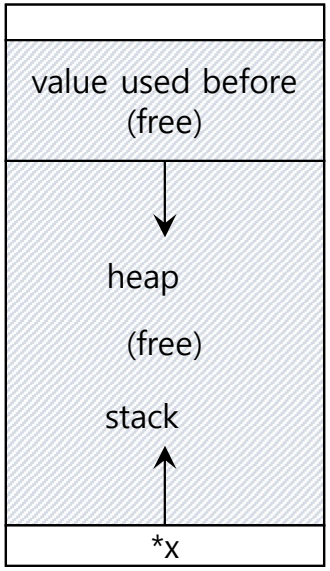
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```



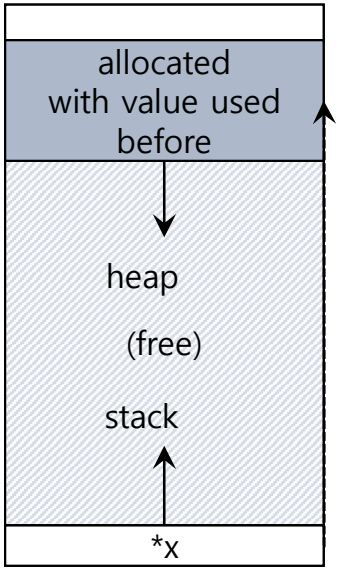
# Forgetting to Initialize

- Encounter an uninitialized read

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




**Address Space**




**Address Space**

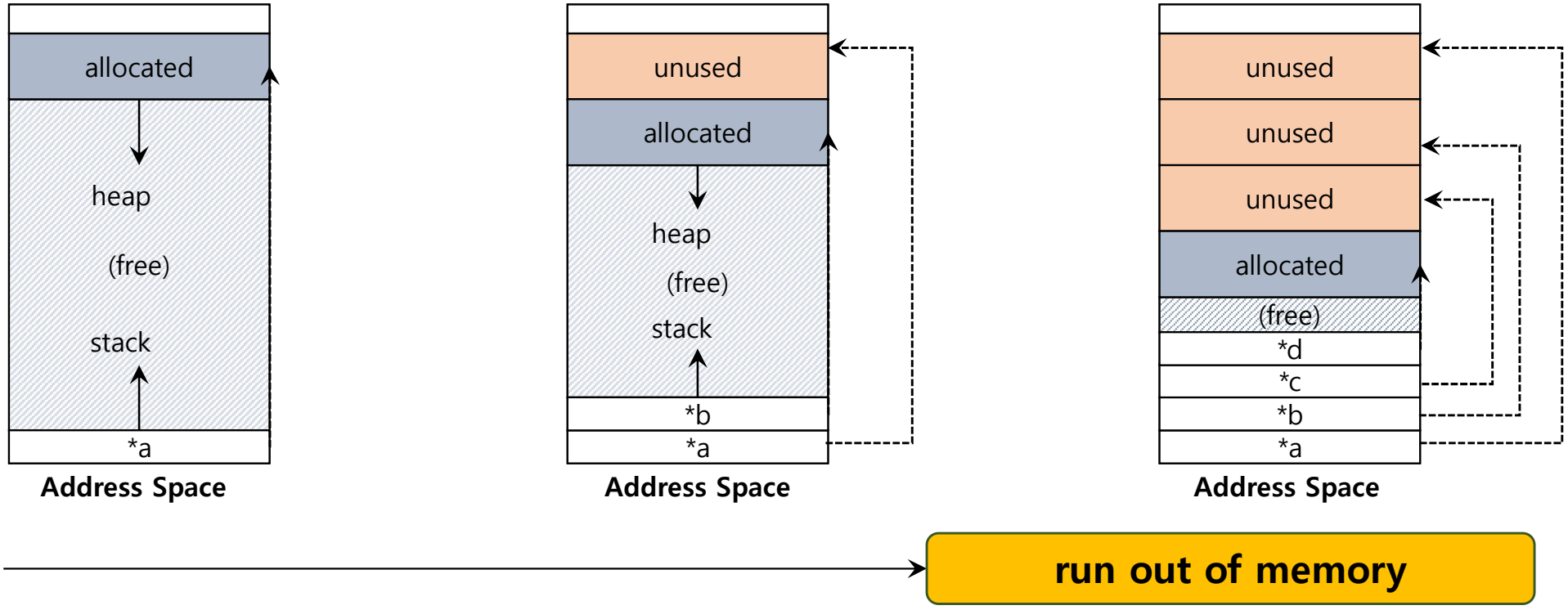
# Forgetting to Initialize


- With this error, you call malloc() properly, but forget to fill in some values into your newly-allocated data type.
- Don’t do this! If you do forget, your program will eventually encounter an **uninitialized read, where it** reads from the heap some data of unknown value.

# Memory Leak

- A program runs out of memory and eventually dies.

unused : unused, but not freed

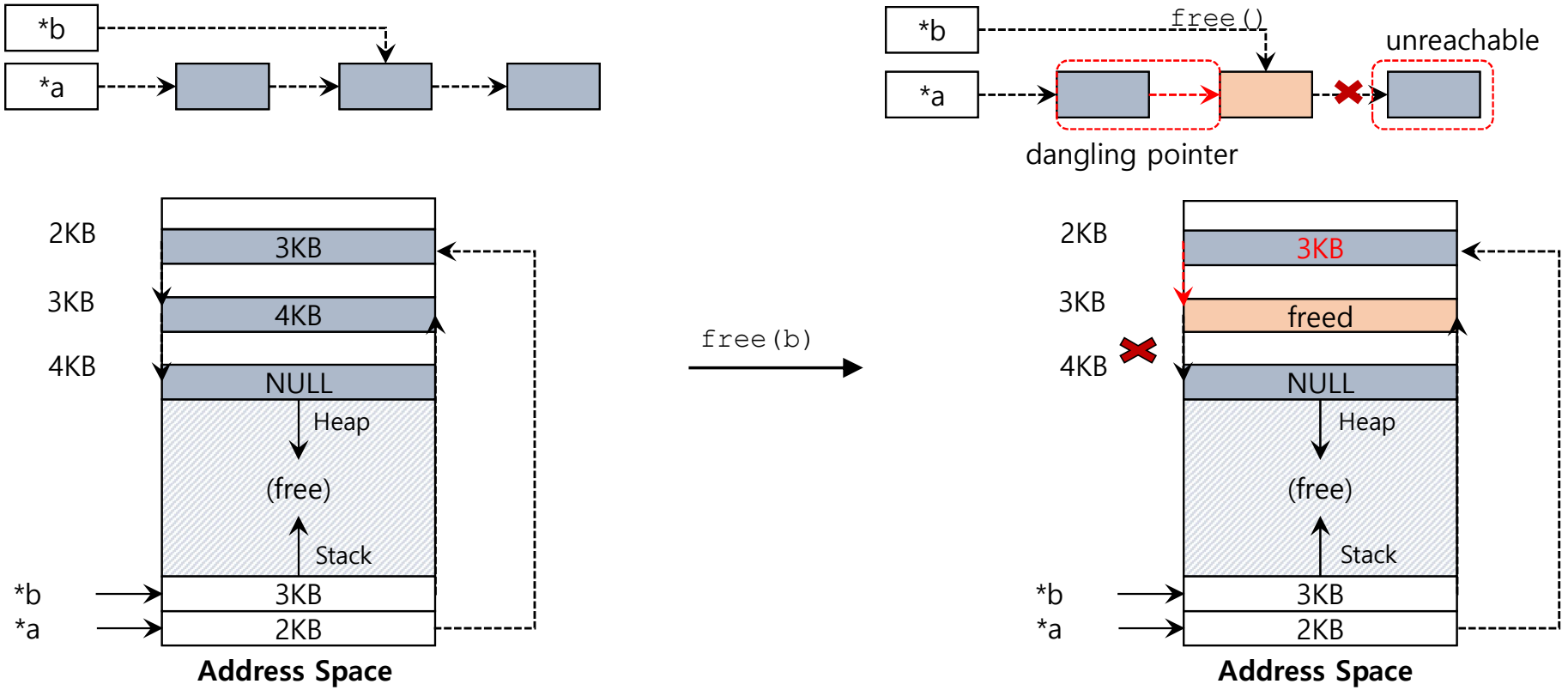


# Memory Leak


- Another common error is known as a **memory leak, and it occurs when** you forget to free memory.
- In long-running applications or systems (such as the OS itself), this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required

# Dangling Pointer

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



# Dangling Pointer


- Sometimes a program will free memory before it is finished using it; such a mistake is called a **dangling pointer, and it, as you can guess, is also a** bad thing.
- The subsequent use can crash the program, or overwrite valid memory.
- e.g., you called free(), but then called malloc() again to allocate something else, which then recycles the errantly-freed memory.

# Other Memory APIs: calloc()

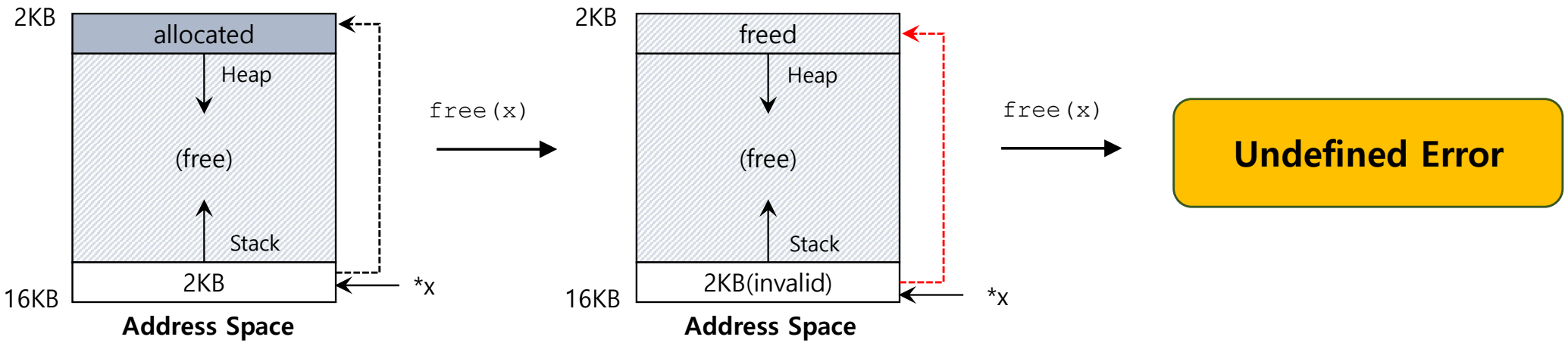
```
#include <stdlib.h>

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

- calloc() allocates the memory and also initializes the allocates memory block to zero. If we try to access the content of these blocks then we'll get 0 whereas if u want to access the content of the block which was allocated by malloc(), it will return garbage values.
- 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
```



# Double Free


- Programs also sometimes free memory more than once; this is known as the **double free.**
- **The result of doing so is undefined.**
- As you can imagine**,** the memory-allocation library might get confused and do all sorts of weird things; crashes are a common outcome.

# Other Memory APIs: realloc()

```
#include <stdlib.h>

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

- The **realloc**() function is used to resize allocated memory without losing old data.
- 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**() and **sbrk**() change the location of the *program break*, which defines the end of the process's data segment
  - **brk**() sets the end of the data segment to the value specified by *addr*, **sbrk**() increments the program's data space by *increment* bytes.
  - 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.