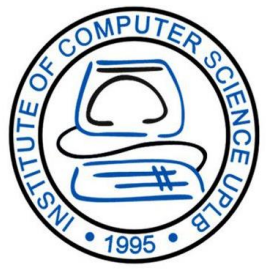
CMSC 125: Operating Systems

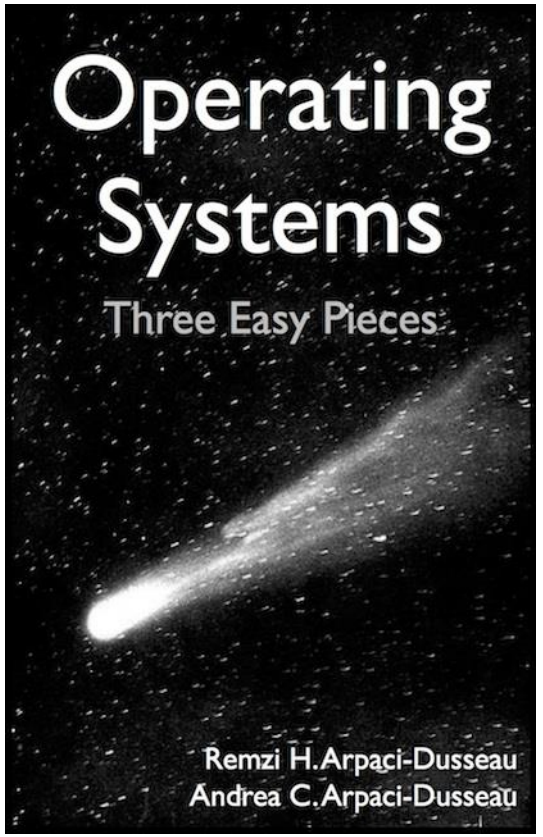
- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org



Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

14. Memory API

Operating System: Three Easy Pieces

Process Memory Usage

- We know that at runtime a process is allocated memory regions(pages) for Code, Data, Stack, Heap
- Code and Data normally do not change – pre-allocated at load time, no allocations and deallocations
- Stack changes during function calls – short-lived allocations
  - Used to store function parameters, local variables, and return address
  - Memory in this region are allocated-deallocated by manipulating the Stack Pointer(SP) register using PUSH and POP instructions
  - Implicit management - the compiler generates the PUSHes and POPs for a C program
- Heap – long-lived
  - Explicit management - allocations and deallocations handled by programmer via some API
  - Challenging to both systems and programmers

```
void func() {
   int *x = (int *) malloc(sizeof(int));
   ...
}
```

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`

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
```

- Be careful with strings: `malloc(strlen(s) + 1);`

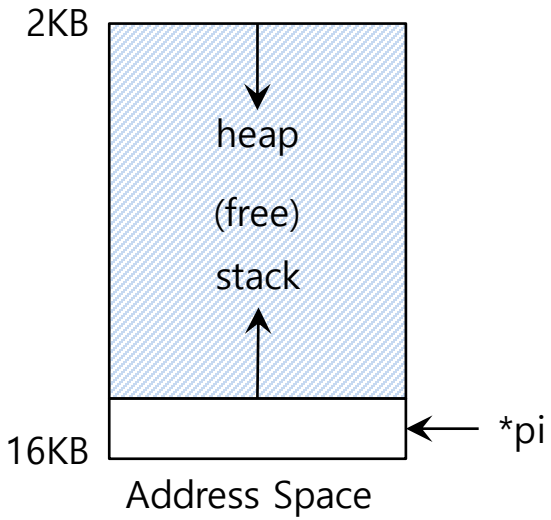
Memory API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

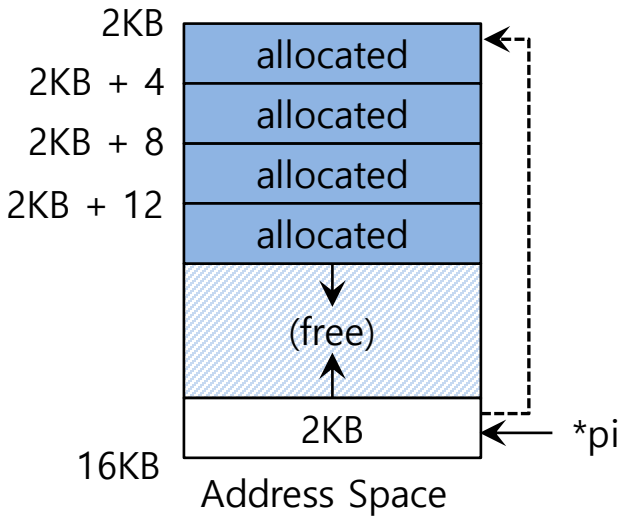
- Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

Allocating Memory



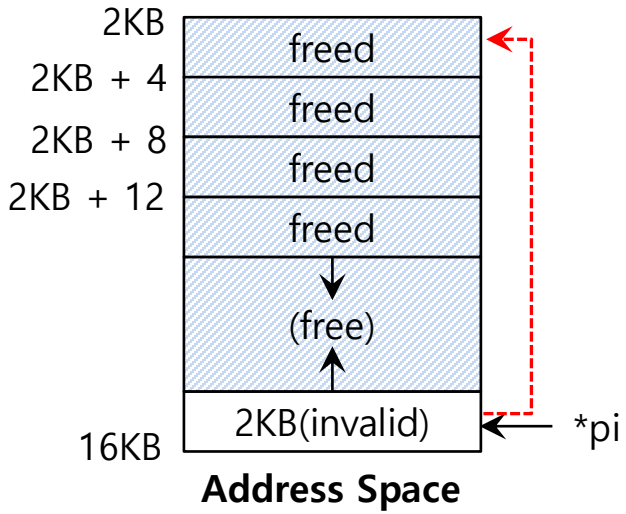
pointer

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

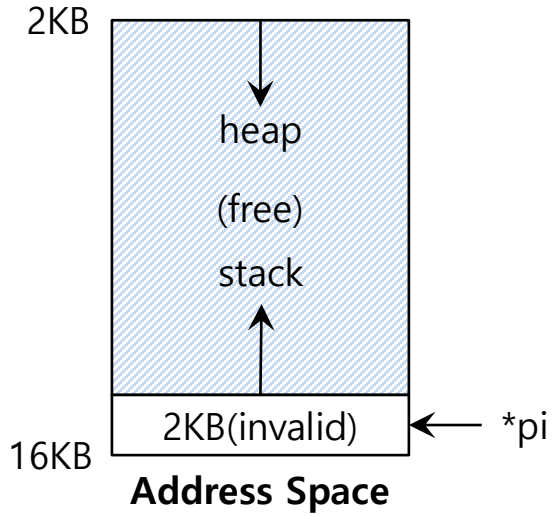



```
pi = (int *)malloc(sizeof(int)* 4);
```

Freeing Memory



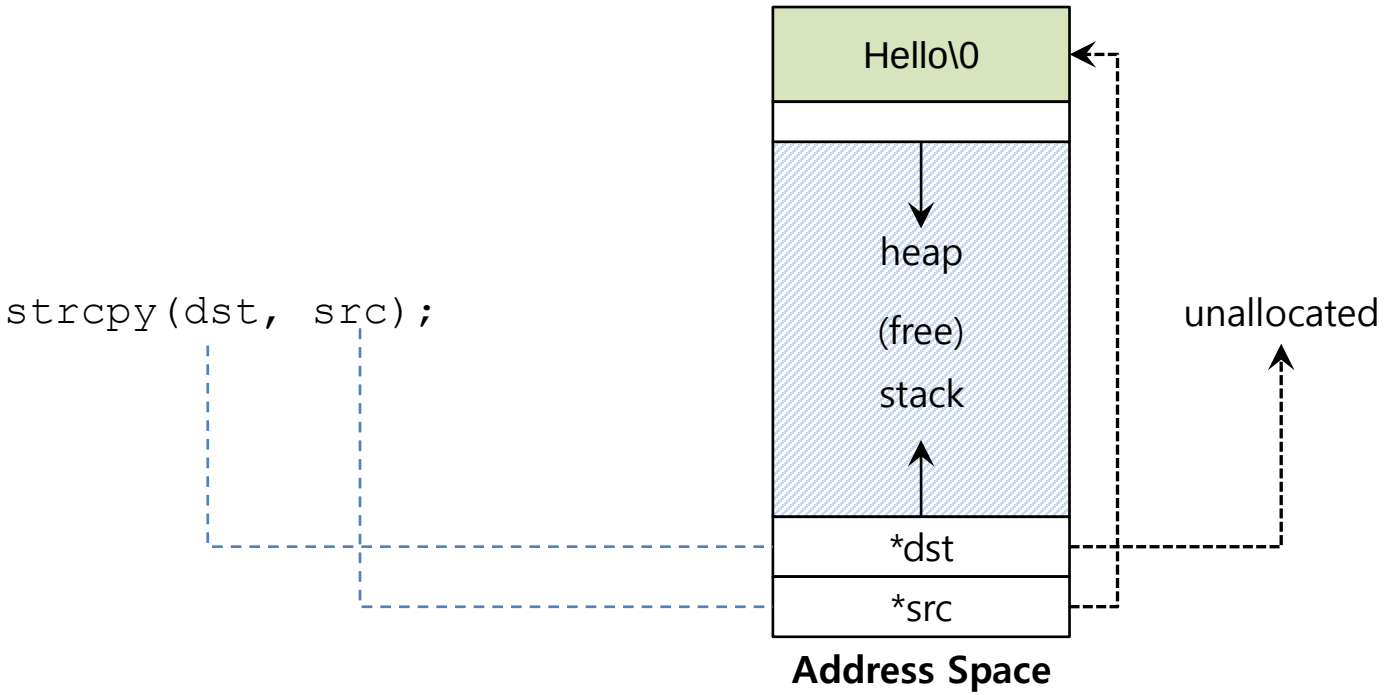

```
free(pi);
```



Common Error: Forgetting To Allocate Memory

- Incorrect code

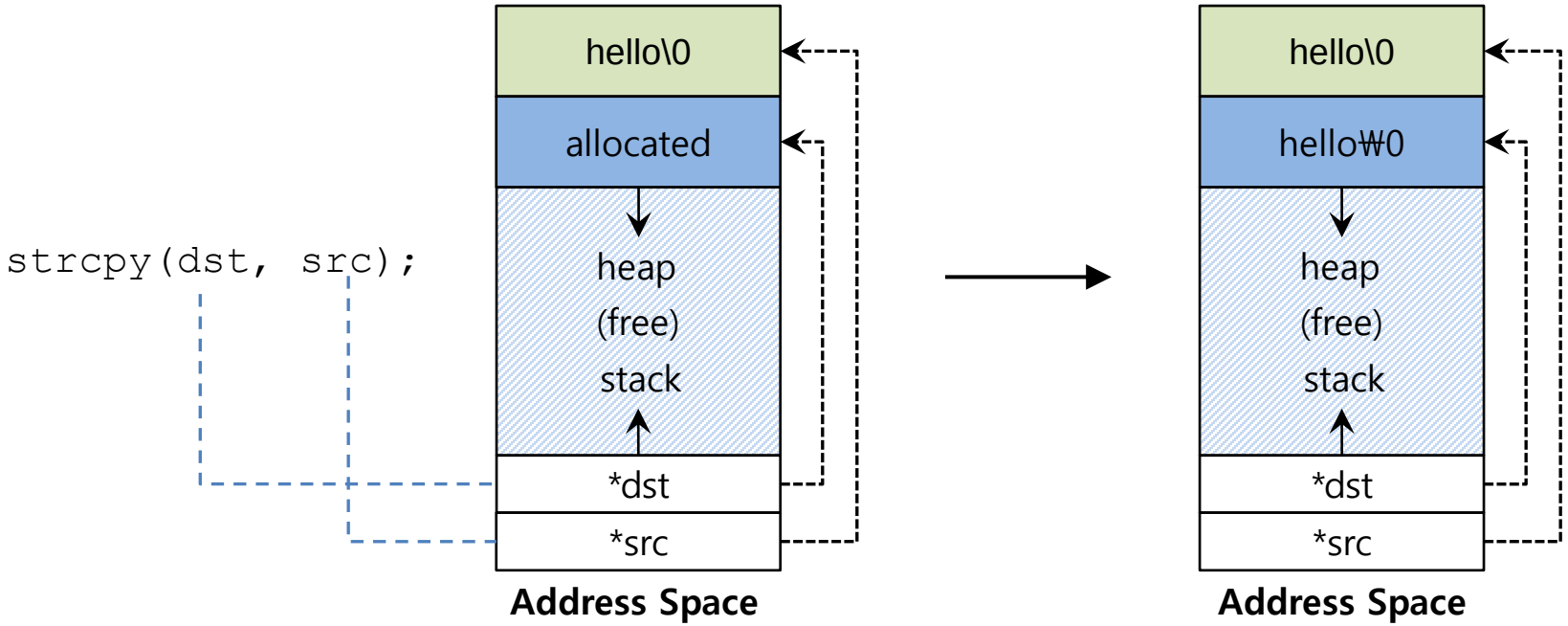
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Common Error: Forgetting To Allocate Memory(Cont.)

- Correct code

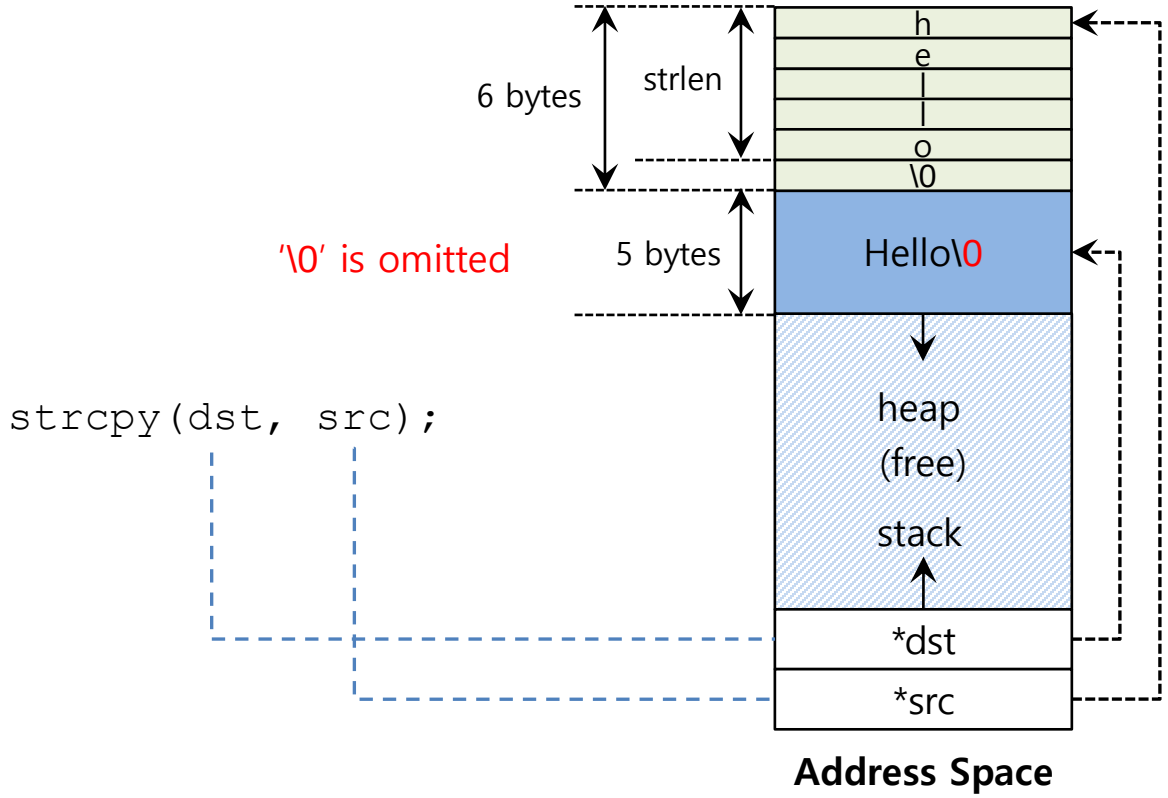
```
char *src = "hello";   //character string constant
char *dst = (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Common Error: Not Allocating Enough Memory

- Incorrect code, but executes properly (aka **buffer overflow**)

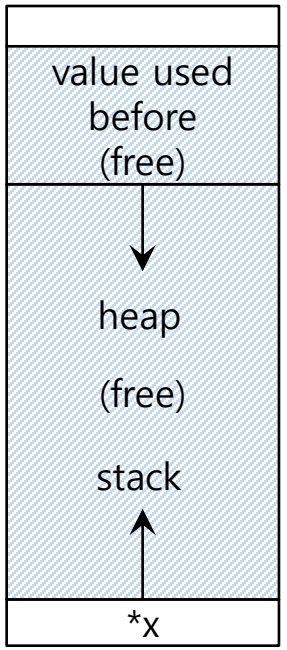
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```



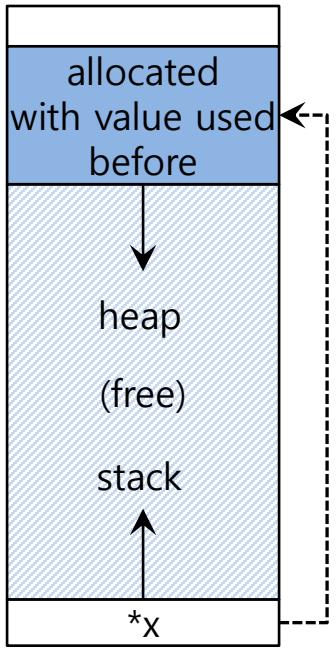
Common Error: Forgetting to Initialize

- Encounter an **uninitialized read**

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



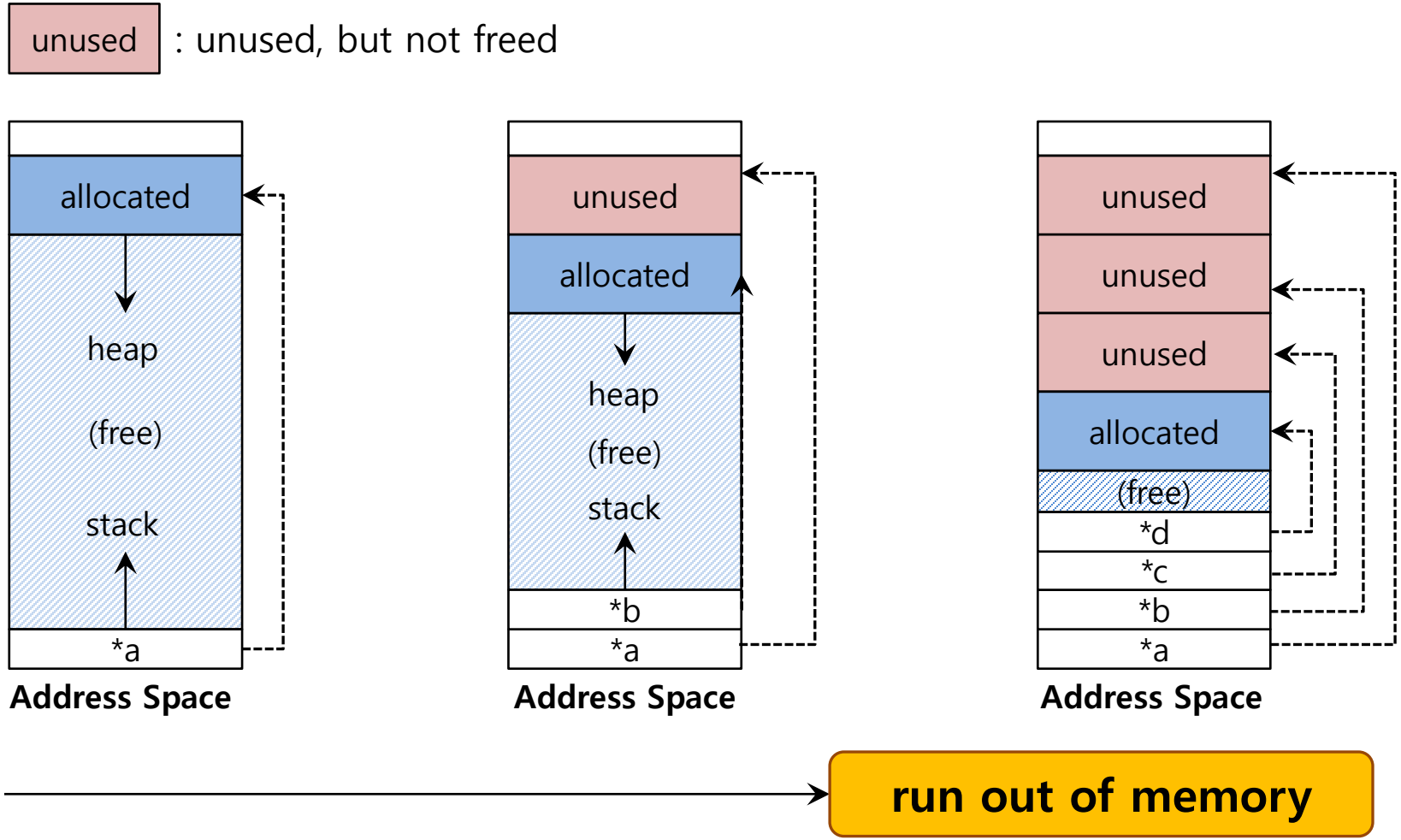
Address Space



Address Space

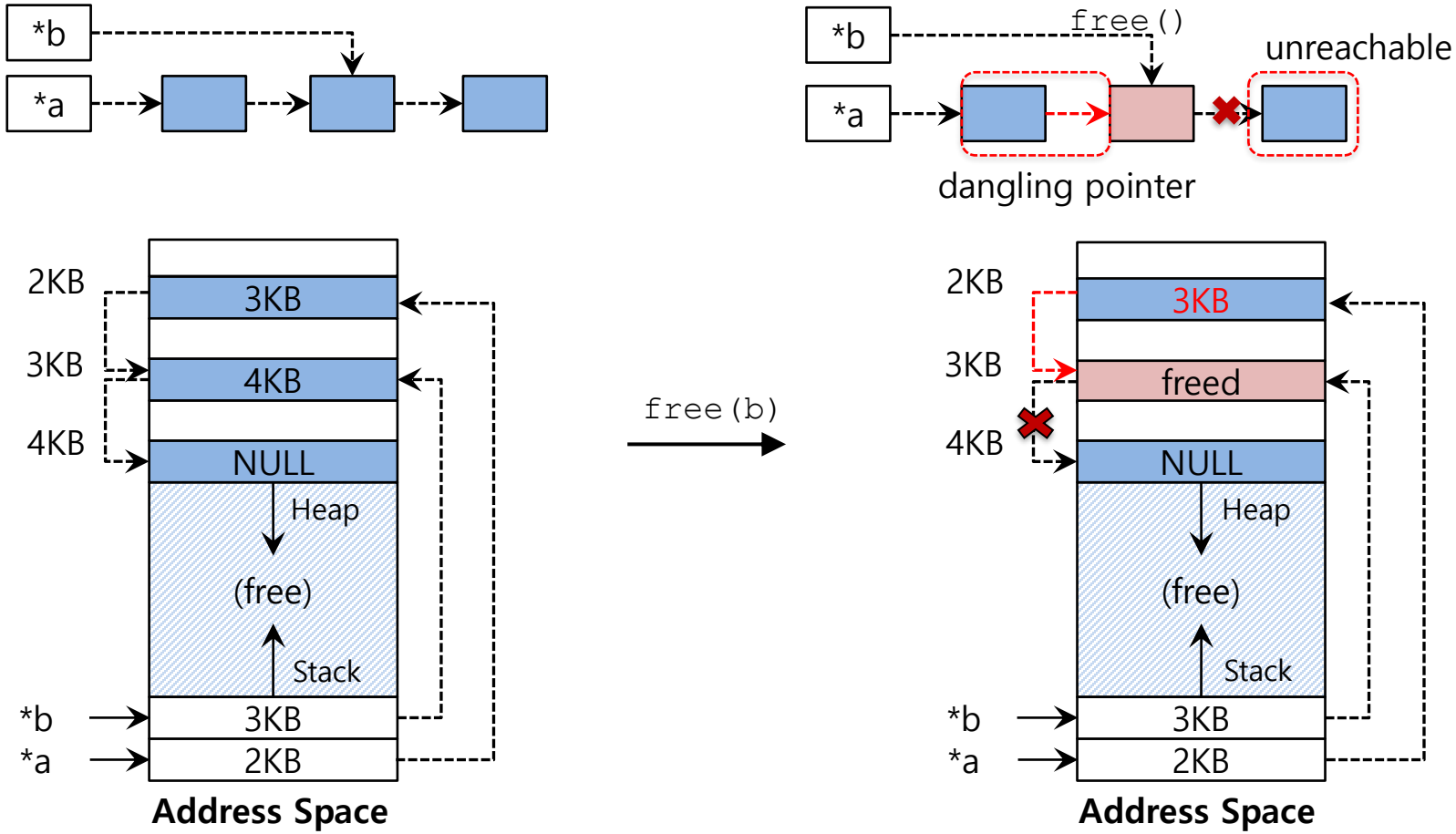
Common Error: Memory Leak

- Forgetting to free until a process runs out of memory and eventually dies



Common Error: Dangling Pointer

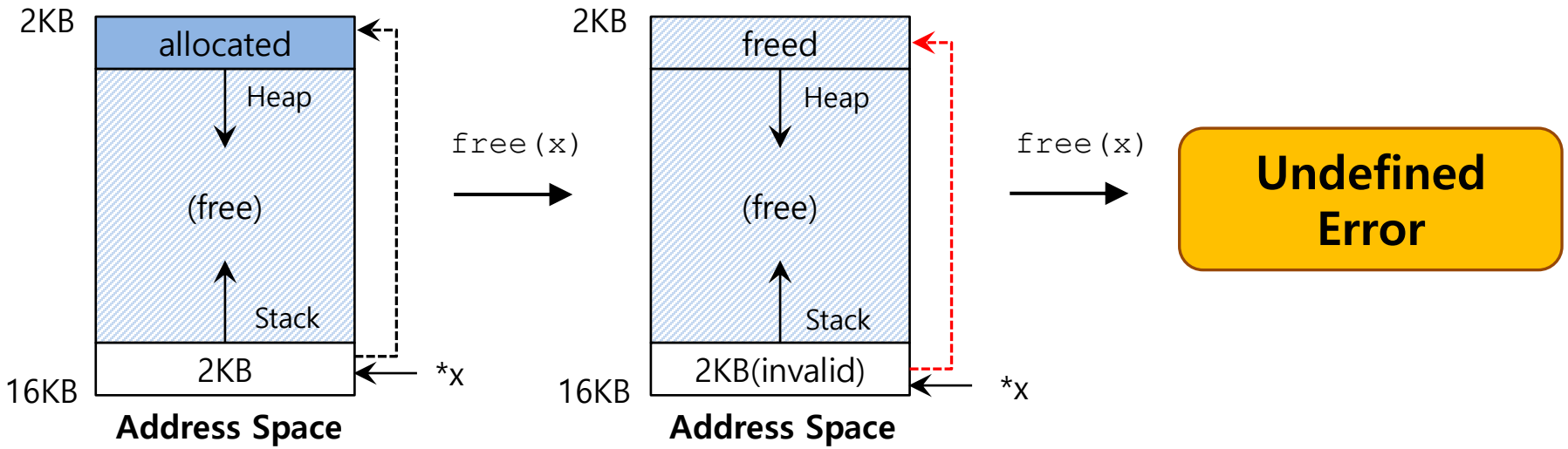
- Freeing memory before you are done with it
  - A program accesses to memory with an invalid pointer



Common Error: Double Free

- Free memory that was freed already

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



Common Error: Invalid Frees

- Calling free() incorrectly is dangerous
  - It expects a pointer as parameter that returned by malloc()

Tools for Checking Memory Errors

- Purify
- Valgrind

System calls related to memory allocation

```
#include <unistd.h>

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

- `malloc` library call use `brk` system call - old approach
  - `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 related to memory allocation(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 – used in modern malloc() implementations

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 : NULL

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