14. Memory API

Operating System: Three Easy Pieces

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 (defined in C standard).
  - 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
```

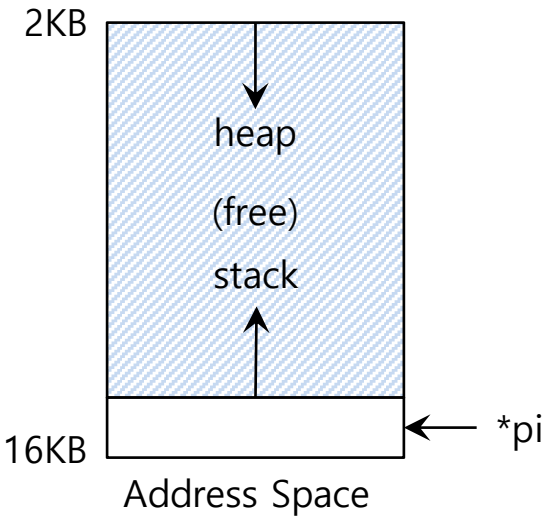
Memory API: free()

```
#include <stdlib.h>

void free(void* ptr)
```

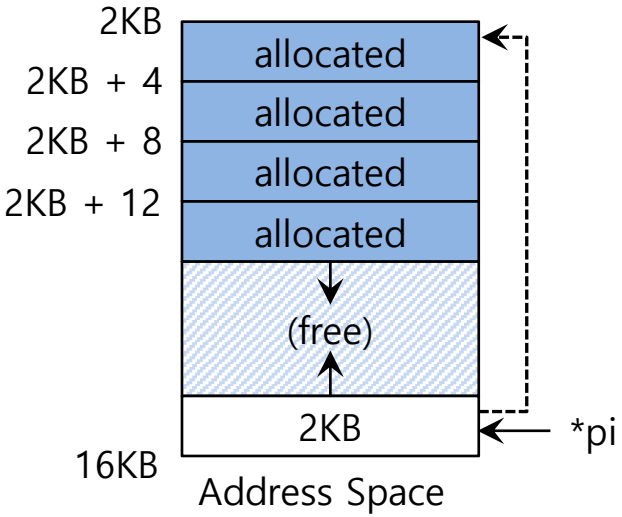
- Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

Memory Allocating



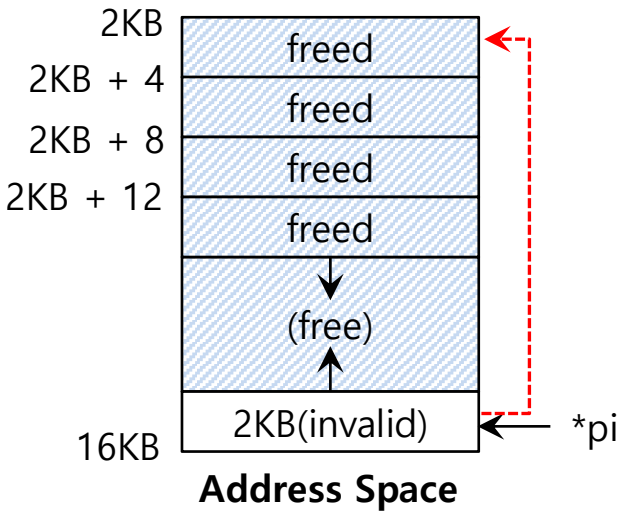
pointer

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

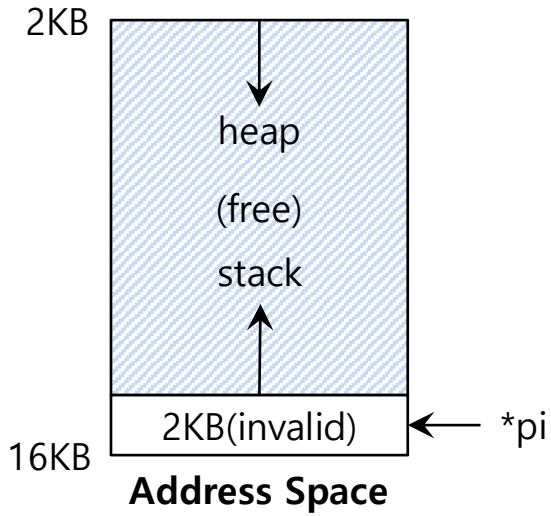



```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing



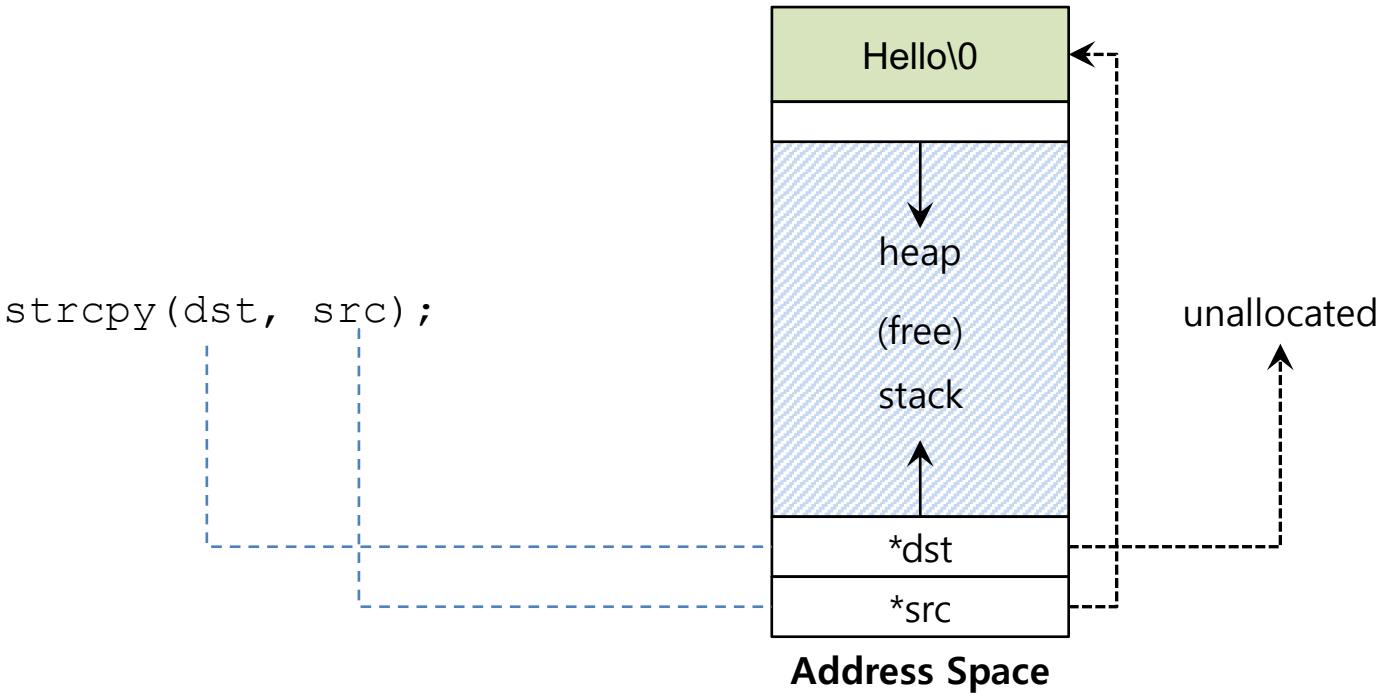

```
free(pi);
```



Forgetting To Allocate Memory

- Incorrect code

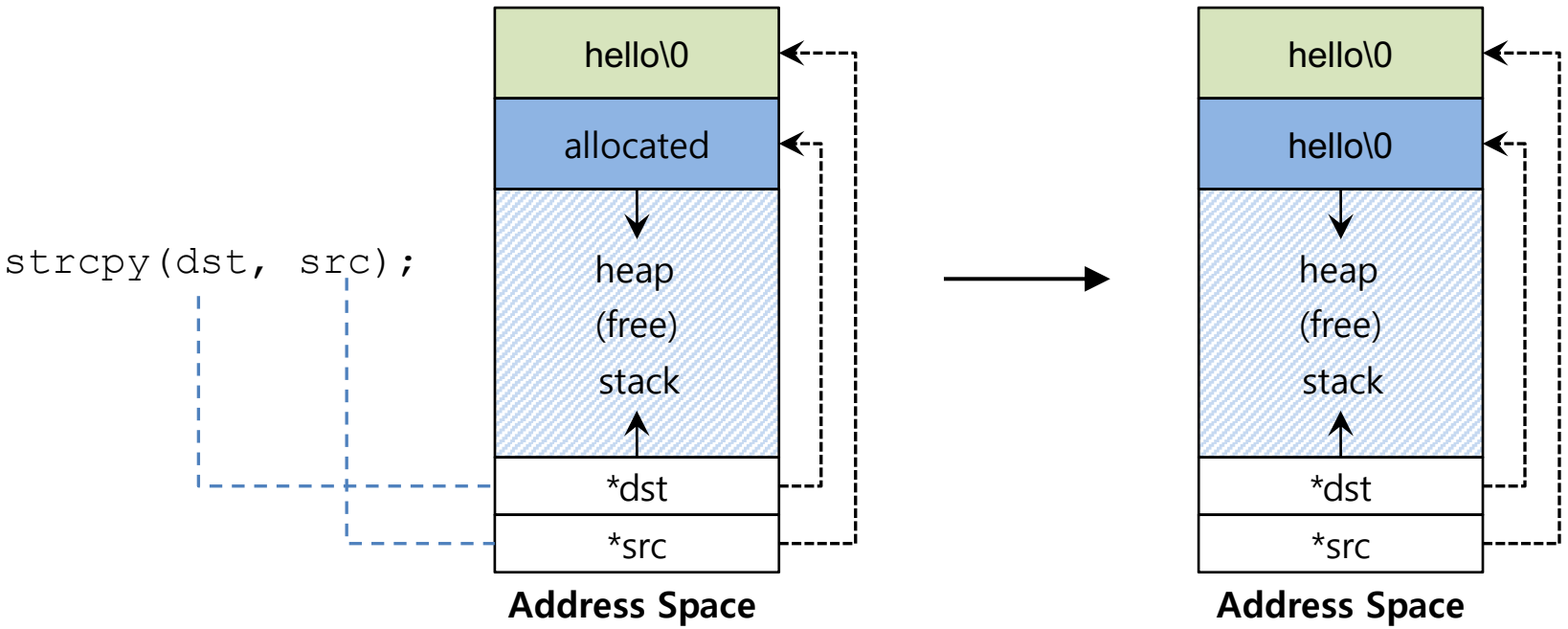
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory(Cont.)

- Correct code

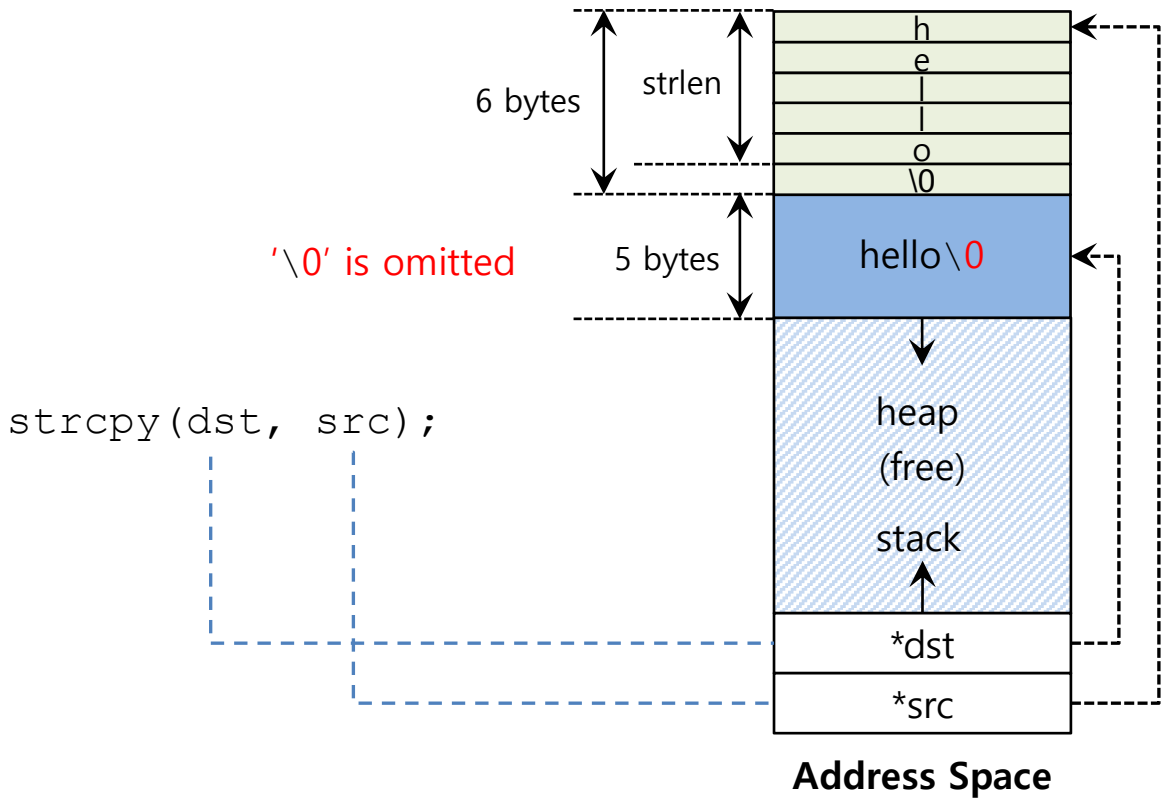
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory

- Incorrect code, but work properly

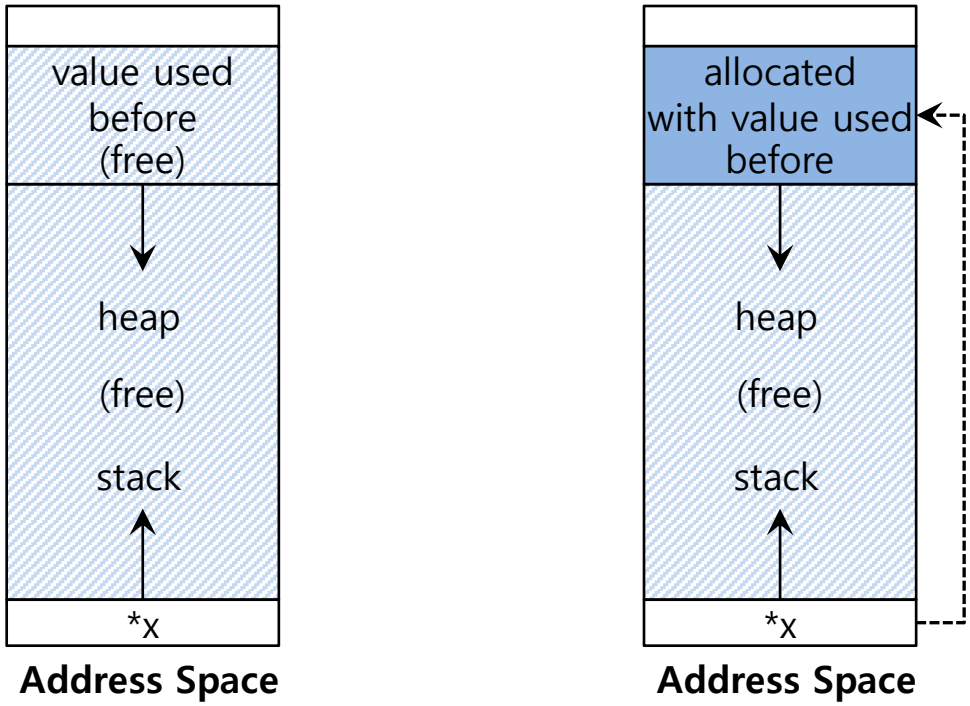
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src); //work properly
```



Forgetting to Initialize

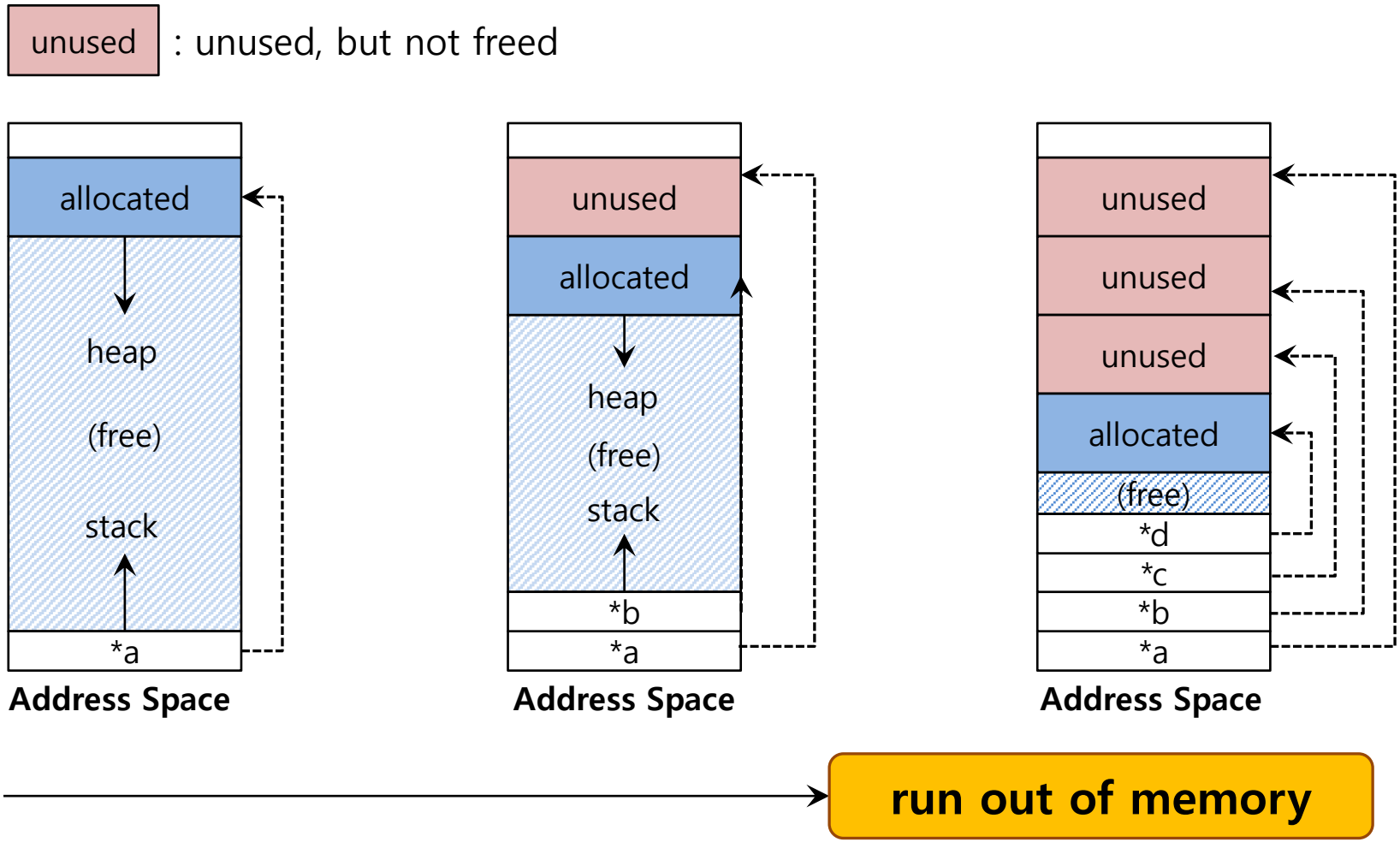
- Encounter an uninitialized read

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



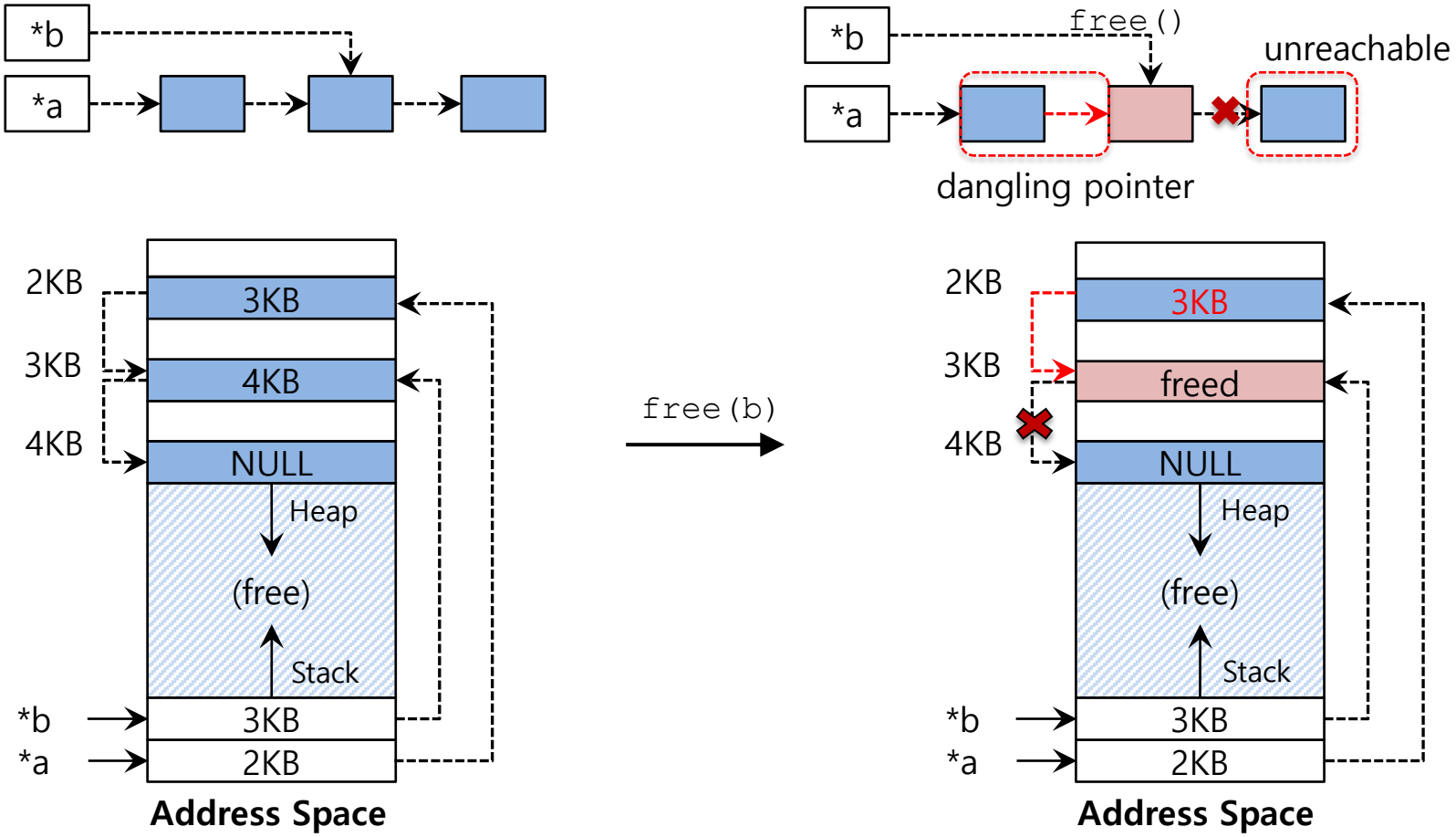
Memory Leak

- A program runs out of memory and eventually dies.



Dangling Pointer

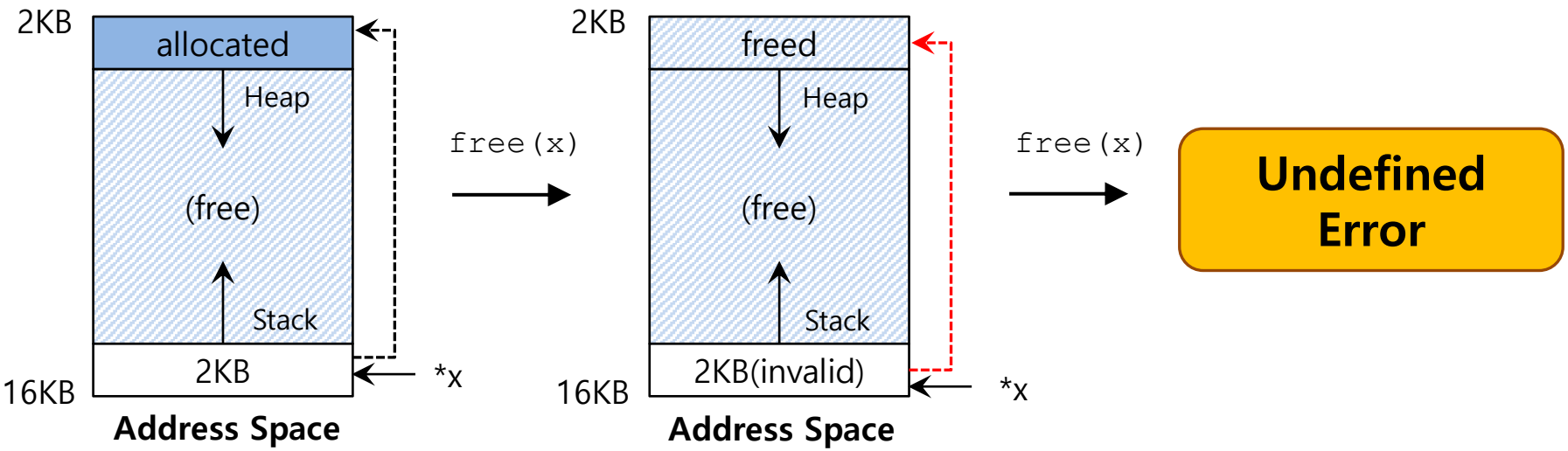
- Freeing memory before it is finished using
  - A program accesses to memory with an invalid pointer



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: 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

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

```
#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
  - Handled like heap

- Maps files/devices into memory (memory mapped I/O)
  - E.g. use: shared access to files across processes/threads
  - E.g. use: open really small files
  - …

Memory API nuances

- **Manual** memory handling is prone to (hard to find) bugs. ~~Unmanaged~~ Unsafe languages?

- Managed languages are painfully slow
  - An OS written in java?

- Unmanaged languages with "**managed" code**
  - C++ with std::ptr
  - **Rust**

- Rust is be the future?
  - (+) Linux modules support for rust
  - (-) https://news.ycombinator.com/item?id=31432908

- This lecture slide set is used in AOS course at University of Cantabria. 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 Arpaci-Dusseau (at University of Wisconsin)