

# Operating Systems

Prof. Yongkun Li

中国科大-计算机学院 教授

<http://staff.ustc.edu.cn/~ykli>

Ch7, part1

## Memory Management from a Programmer's Perspective

# Why we need memory management

- The running program code requires memory
  - Because the CPU needs to fetch the instructions from the memory for execution
- We must keep several processes in memory
  - Improve both CPU utilization and responsiveness
  - Multiprogramming

It is required to efficiently manage the memory

# Topics in Ch7

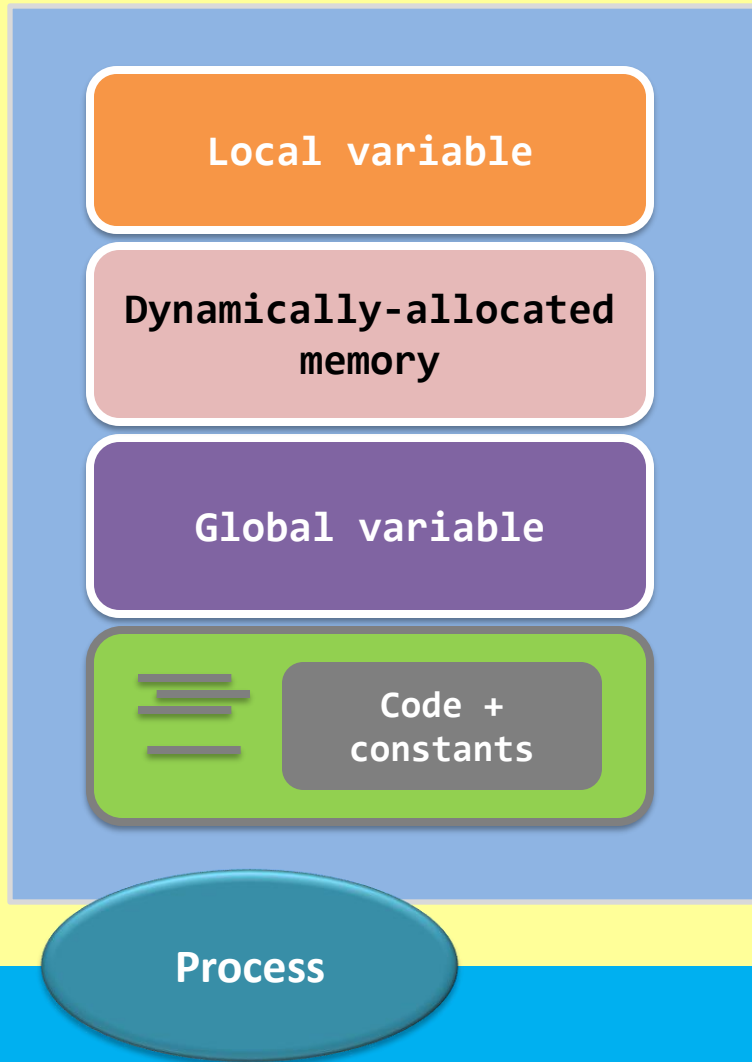
## From a programmer's perspective: user-space memory management

- What is the address space of a process?
- How are the program code and data stored in memory?
- How to allocate/free memory (malloc() + free())?
- How much memory can be used in a program?
- What are segmentation and segmentation fault?

## From the kernel's perspective: How to manage the memory

- What is virtual memory?
- How to realize address mapping (paging)?
- How to support very large programs (demand paging)?
- How to do page replacement?
- What is TLB?
- What is memory-mapped file?

# Part 1: User-space memory



Do you remember this?

- Content of a process (in user-space memory)

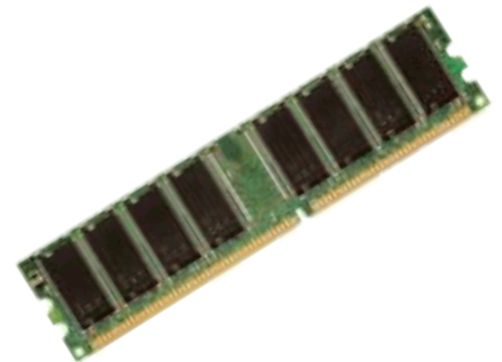
How does each part use the memory?

- From a programmer's perspective

Let's forget about the kernel for a moment. We are going to explore the **user-space memory** first.

# User-space memory management

- **Address space;**
- Code & constants;
- Data segment;
- Stack;
- Heap;
- Segmentation fault;



# Address space

How does a programmer look at the memory space?

- An array of bytes?
- Memory of a process is divided into segments
- This way of arranging memory is called **segmentation**

Stack - Local variables

Heap - Dynamically allocated memory

Data Segment & BSS - Global and static variables

Code + Constant

# Address space

```
int main(void) {  
    int *malloc_ptr = malloc(4);  
    char *constant_ptr = "hello";  
  
    printf("Local variable = %15p\n", &malloc_ptr);  
    printf("malloc() space = %15p\n", malloc_ptr);  
    printf("Global variable = %15p\n", &global_int);  
    printf("Code & constant = %15p\n", constant_ptr);  
  
    return 0;  
}
```

\$ ./addr

Local variable = 0xbfa8938c

malloc() space = 0x915c008

Global variable = 0x804a020

Code & constant = 0x8048550

\$ \_

## Note

The addresses are not necessarily the same in different processes

What is the process address space?

Increasing  
address

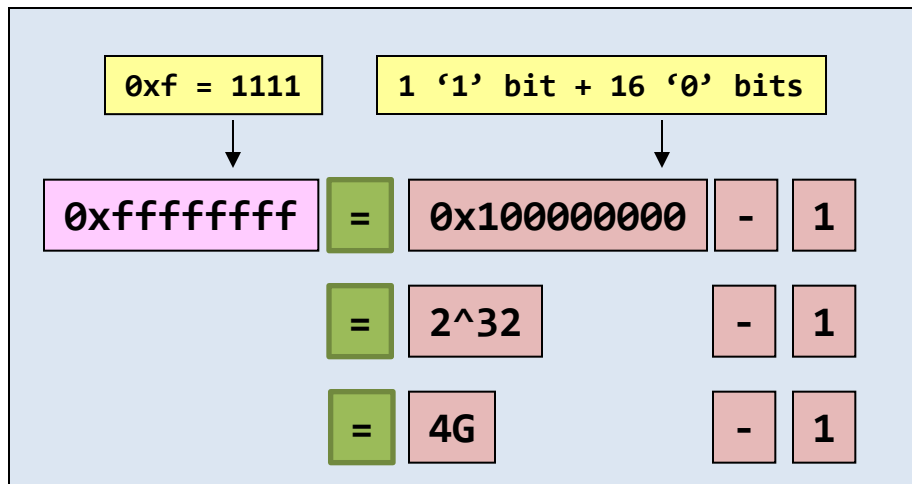
Stack - Local variables

Heap - Dynamically  
allocated memory

Data Segment & BSS -  
Global and static  
variables

Code + Constant

# Address space



In a 32-bit system,

- One address maps to one byte.
- The maximum amount of memory in a process is **4GB**.

Note

- This is the so called logical address space
- **Each process has its own address space**, and it can reside in any part of the physical memory

How large is the address space?

Increasing address

Stack - Local variables

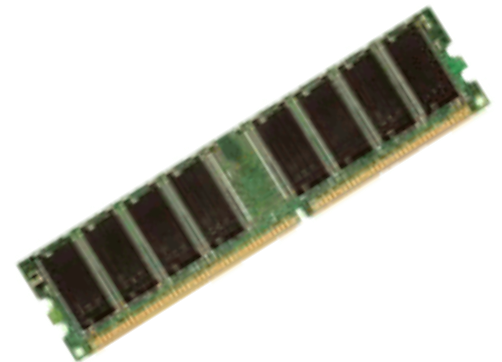
Heap - Dynamically allocated memory

Data Segment & BSS - Global and static variables

Code + Constant

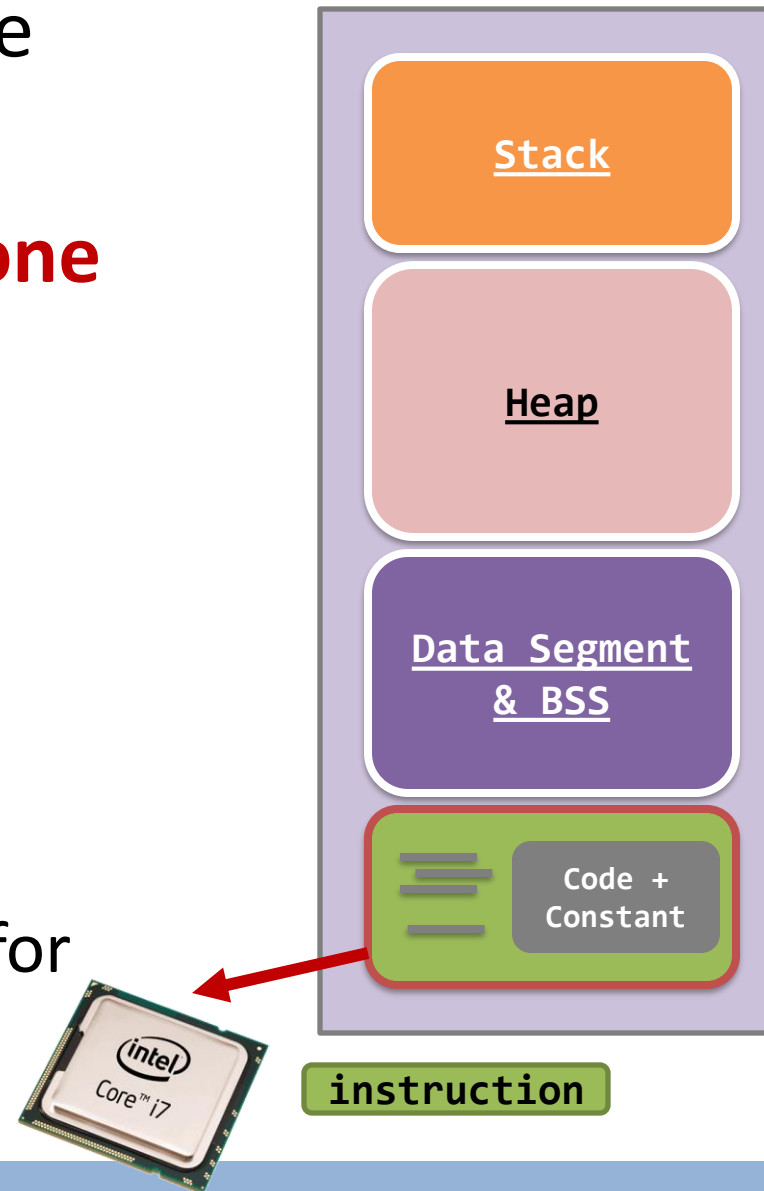
# User-space memory management

- Address space;
- **Code & constants;**
- Data segment;
- Stack;
- Heap;
- Segmentation fault;



# Program code & constants

- A program is an executable file
- A process is **not bounded to one program code.**
  - Remember `exec*()` family?
- The program code requires memory space because...
  - The CPU needs to fetch the instructions from the memory for execution.



# Program code & constants

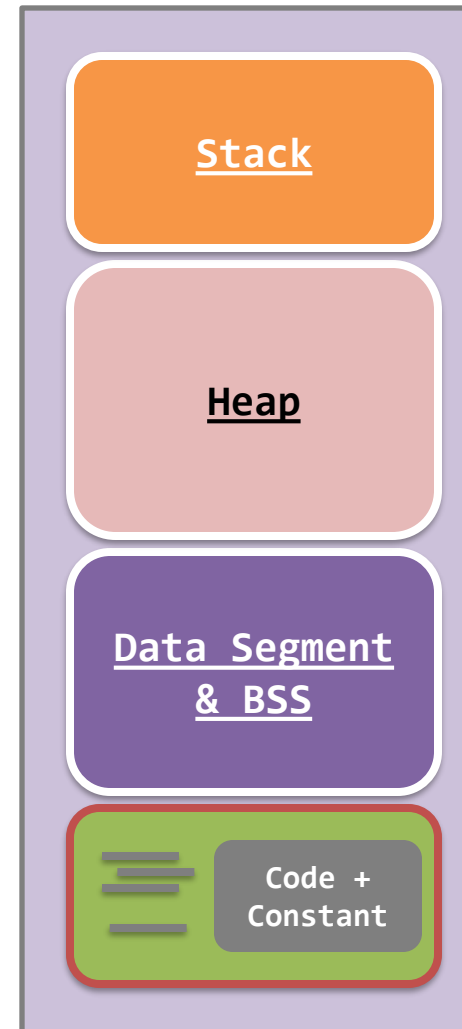
```
1 int main(void) {
2     char *string = "hello";
3     printf("\"hello\"      = %p\n", "hello");
4     printf("String pointer = %p\n", string);
5     string[4] = '\0';
6     printf("Go to %s\n", string);
7     return 0;
8 }
```

- **Question #1.** What are the printouts from Line 3 & 4?

```
"hello"      = 0x8048520
String pointer = 0x8048520
```

- **Question #2.** What is the printout from Line 6?

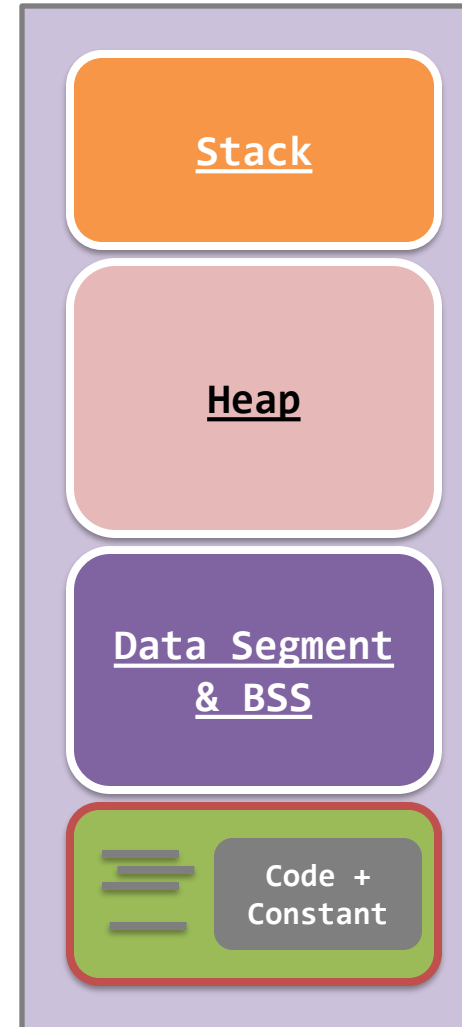
```
Segmentation fault
```



# Program code & constants

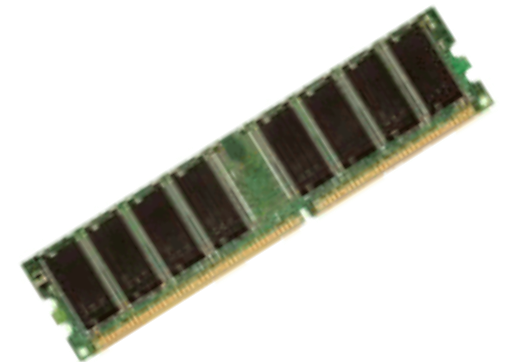
```
1 int main(void) {
2     char *string = "hello";
3     printf("\"hello\" = %p\n", "hello");
4     printf("String pointer = %p\n", string);
5     string[4] = '\0';
6     printf("Go to %s\n", string);
7     return 0;
8 }
```

- Constants are stored in code segment.
  - Accessing of constants are done using addresses (or pointers).
- Codes and constants are both **read-only**.



# User-space memory management

- Address space;
- Code & constants;
- **Data segment;**
- Stack;
- Heap;
- Segmentation fault;



# Data Segment & BSS – properties

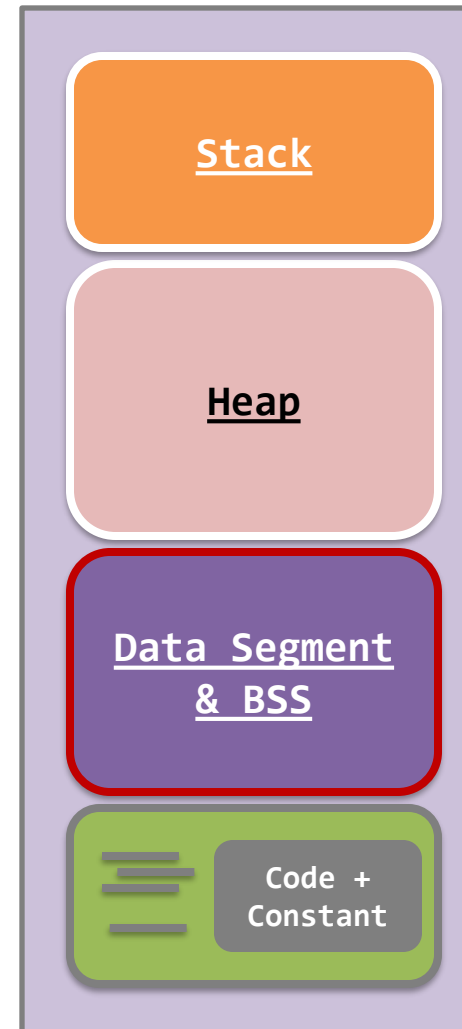
```
int global_int = 10;
int main(void) {
    int local_int = 10;
    static int static_int = 10;
    printf("local_int  addr = %p\n", &local_int );
    printf("static_int addr = %p\n", &static_int );
    printf("global_int addr = %p\n", &global_int );
    return 0;
}
```

```
$ ./global_vs_static
local_int  addr = 0xbf8bb8ac
static_int addr = 0x804a018
global_int addr = 0x804a014
$_
```

They are stored next to each other.

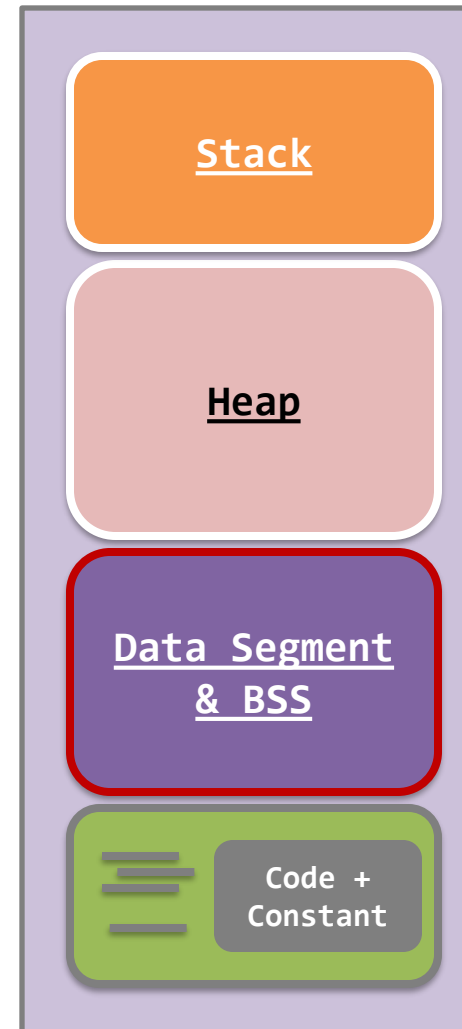
This implies that they are **in the same segment!**

Note: A static variable is treated as the same as a global variable!



# Data Segment & BSS – properties

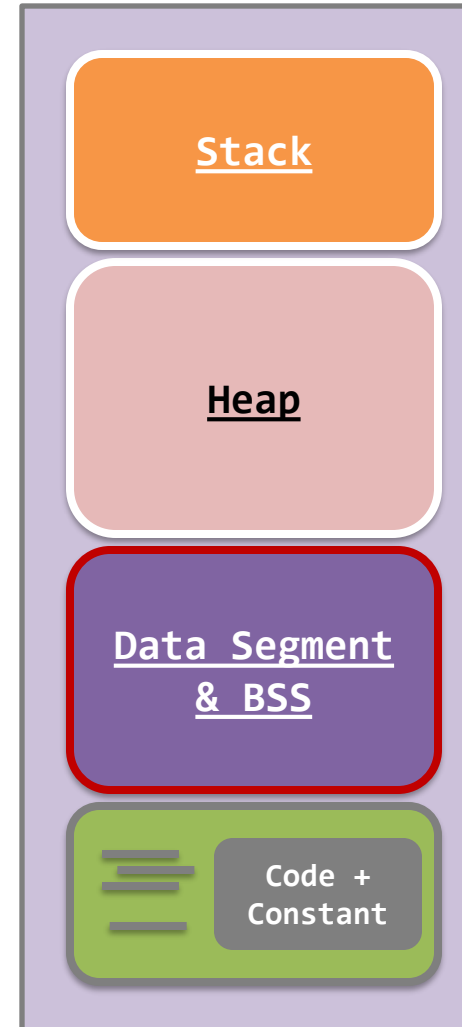
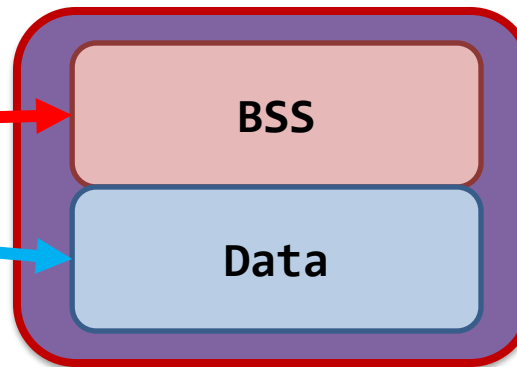
- Data
  - Containing **initialized** global and static variables.
- BSS (**Block Started by Symbol)**
- Containing **uninitialized** global and static variables.



# Data Segment & BSS – locations

```
1 int global_bss;  
2 int global_data = 10;  
3 int main(void) {  
4     static int static_bss;  
5     static int static_data = 10;  
6     printf("global bss = %p\n", &global_bss );  
7     printf("static bss = %p\n", &static_bss );  
8     printf("global data = %p\n", &global_data );  
9     printf("static data = %p\n", &static_data );  
10 }
```

```
$ ./data_vs_bss  
global bss = 0x804a028  
static bss = 0x804a024  
global data = 0x804a014  
static data = 0x804a018  
$_
```



# Data Segment & BSS – sizes

```
char a[1000000] = {10};  
  
int main(void) {  
    return 0;  
}
```

Program: data\_large.c

```
char a[100] = {10};  
  
int main(void) {  
    return 0;  
}
```

Program: data\_small.c

No optimization.

```
$ gcc -O0 -o data_large data_large.c  
$ gcc -O0 -o data_small data_small.c  
  
$ ls -l data_small data_large
```

Guess! Which one is large?

What is the difference between data and BSS?

# Data Segment & BSS – sizes

```
char a[1000000] = {10};

int main(void) {
    return 0;
}
```

Program: data\_large.c

```
char a[100] = {10};

int main(void) {
    return 0;
}
```

Program: data\_small.c

```
$ gcc -O0 -o data_large data_large.c
$ gcc -O0 -o data_small data_small.c

$ ls -l data_small  data_large
-rwxr-xr-x ... 1004816 ... data_large
-rwxr-xr-x ... 4916 ... data_small
$_
```

Wow!

The data segment has the required space already allocated.

# Data Segment & BSS – sizes

```
char a[1000000];
```

```
int main(void) {  
    return 0;  
}
```

Program: bss\_large.c

```
char a[100];
```

```
int main(void) {  
    return 0;  
}
```

Program: bss\_small.c

```
$ gcc -O0 -o bss_large bss_large.c  
$ gcc -O0 -o bss_small bss_small.c
```

```
$ ls -l bss_small  bss_large  
-rwxr-xr-x ... 4775... bss_large  
-rwxr-xr-x ... 4775... bss_small  
$ _
```

Same size!

To the program, BSS is just a bunch of symbols.  
The space is not yet allocated.

The space will be allocated to the process once it starts executing.

This is why BSS is called “*Block Started by Symbol*”.

# Data Segment & BSS – limits

How large is the data segment?

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size      (kbytes, -d) unlimited
.....
$ _
```

In Linux, “**ulimit**” is a built-in command in “/bin/bash”.

It sets or gets the system limitations in the current shell.

Does the “unlimited” mean that you can define a global array with large enough size?

# Data Segment & BSS – limits

```
#define ONE_MEG (1024 * 1024)
```

```
char a[1024 * ONE_MEG];
```

```
int main(void) {  
    memset(a, 0, sizeof(a));  
    printf("1GB OK\n");  
}
```

```
#define ONE_MEG (1024 * 1024)
```

```
char a[2048 * ONE_MEG];
```

```
int main(void) {  
    memset(a, 0, sizeof(a));  
    printf("2GB OK\n");  
}
```

```
$ gcc -Wall -O0 global_2gb.c -o global_2gb  
global_2gb.c:6: warning: integer overflow in expression  
global_2gb.c:6: error: size of array 'a' is negative  
$ _
```

The size of an array is a 32-bit **signed integer**, no matter 32-bit or 64-bit systems.  
Therefore...

# Data Segment & BSS – limits

```
#define ONE_MEG (1024 * 1024)

char a[1024 * ONE_MEG];
char b[1024 * ONE_MEG];
char c[1024 * ONE_MEG];
char d[1024 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf("1GB OK\n");
    memset(b, 0, sizeof(b));
    printf("2GB OK\n");
    memset(c, 0, sizeof(c));
    printf("3GB OK\n");
    memset(d, 0, sizeof(d));
    printf("4GB OK\n");
}
```

Program: global\_4gb.c

Segmentation fault  
why?

On a **32-bit** Linux system, the user-space **addressing space is around 3GB**.

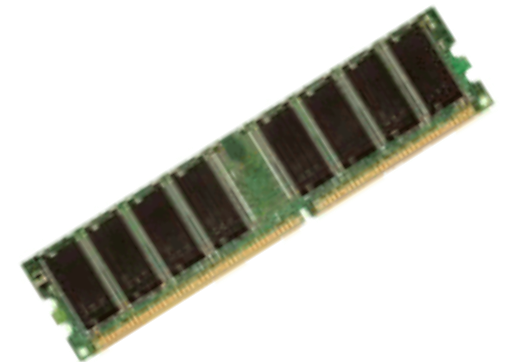
The kernel reserves 1GB addressing space.

# Data Segment & BSS – summary

- Remember, “**global variable == static variables**”.
  - Only the **compiler cares** about the difference!
- Everything in a computer has a limit!
  - Different systems have different limits: 32-bit VS 64-bit.
  - Your job is to adapt to such limits.
  - On a **32-bit** Linux system, the user-space **addressing space is around 3GB**.

# User-space memory management

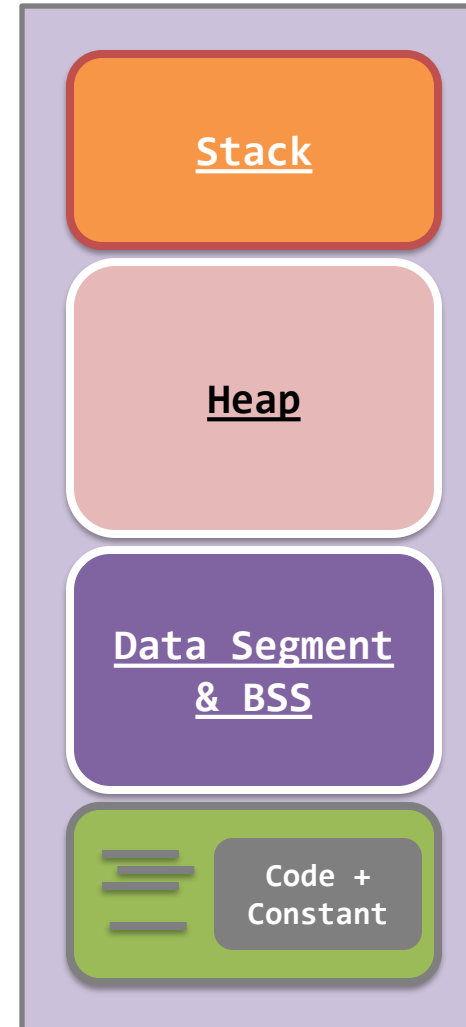
- Address space;
- Code & constants;
- Data segment;
- **Stack;**
- Heap;
- Segmentation fault;



# Stack – properties

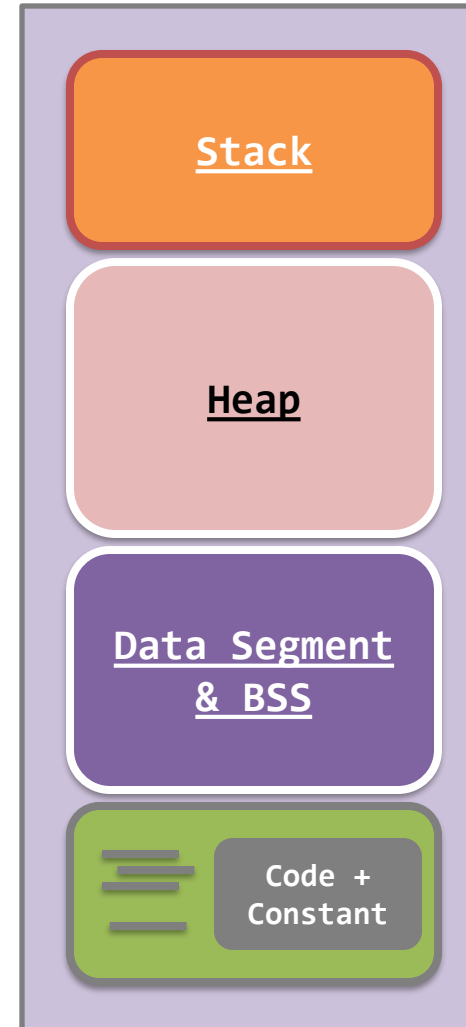
- The stack contains:
  - all the local variables,
  - all function parameters,
  - program arguments, and
  - environment variables.

How are the data stored and what is the size limit?

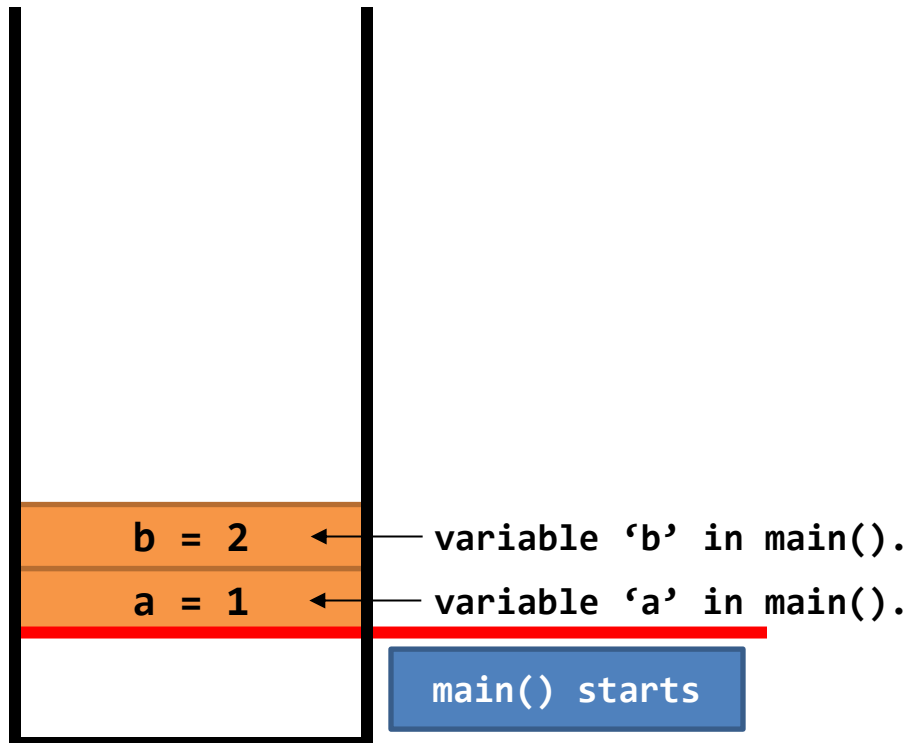


# Stack – properties

- Stack: FILO
- When a function is called, the local variables are allocated in the stack.
- When a function returns, the local variables are deallocated from the stack.



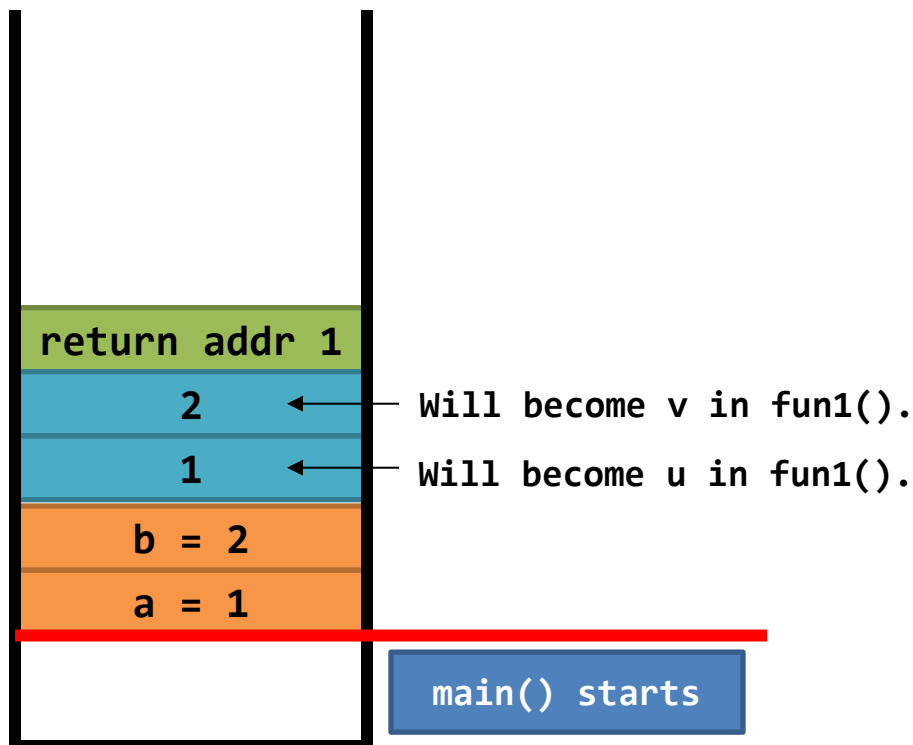
# Stack – push & pop mechanisms



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}  
  
int fun1(int u, int v) {  
    return fun2(v, u);  
}  
  
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

# Stack – push & pop mechanisms

Calling function “**fun1()**” starts.  
It is the beginning of the call, and the CPU has not switched to **fun1()** yet.

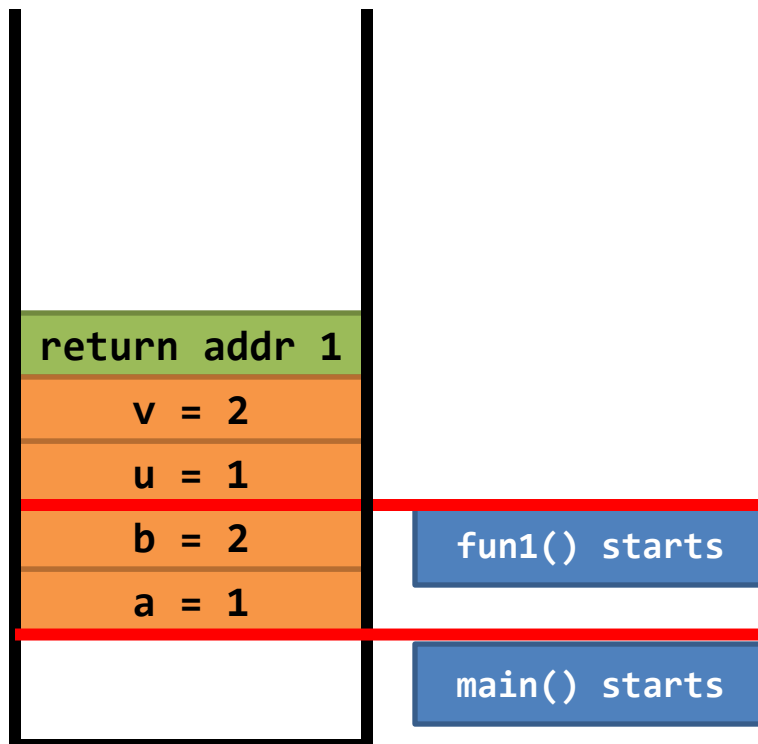


```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}  
  
int fun1(int u, int v) {  
    return fun2(v, u);  
}  
  
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

“return addr 1”  
is approx. here.

# Stack – push & pop mechanisms

Calling function “**fun1()**” takes place. The CPU has switched to **fun1()** .



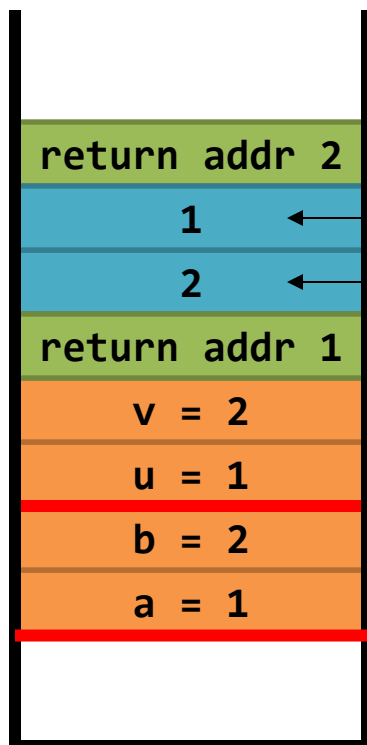
```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}
```

```
int fun1(int u, int v) {  
    return fun2(v, u);  
}
```

```
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

# Stack – push & pop mechanisms

Calling function “**fun2()**” starts.  
It is the beginning of the call, and the CPU has not switched to **fun2()** yet.



Will become y in fun2().  
Will become x in fun2().

return addr 2 is approx. here.

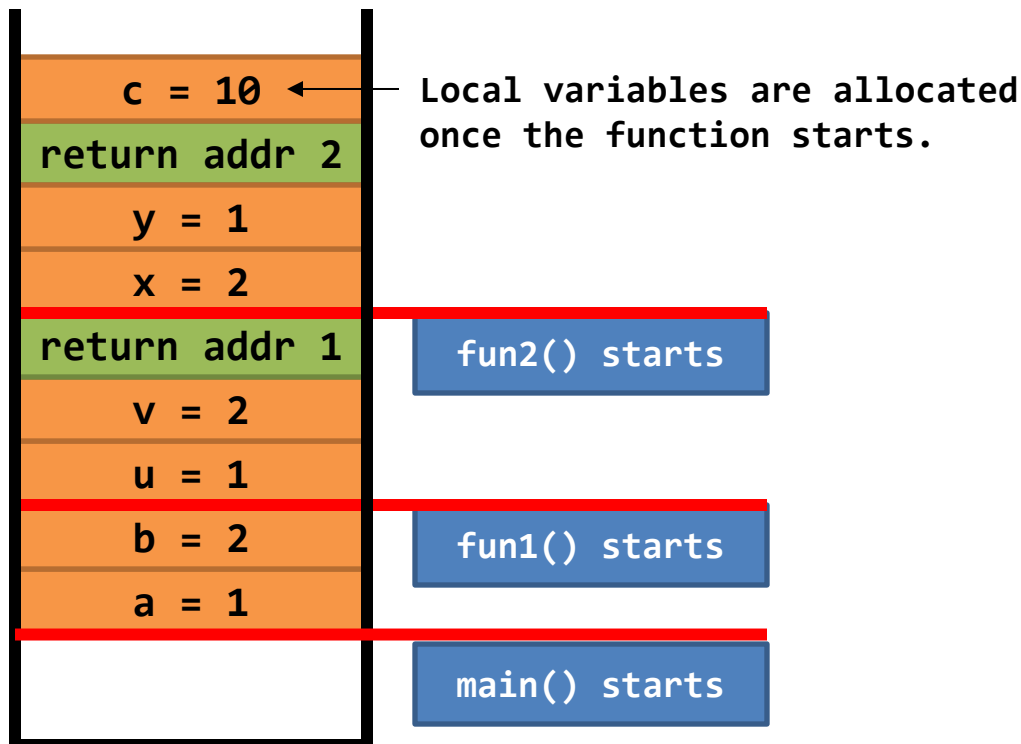
```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

# Stack – push & pop mechanisms

Calling function “**fun2()**” takes place. The CPU has switched to **fun2()** .



```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

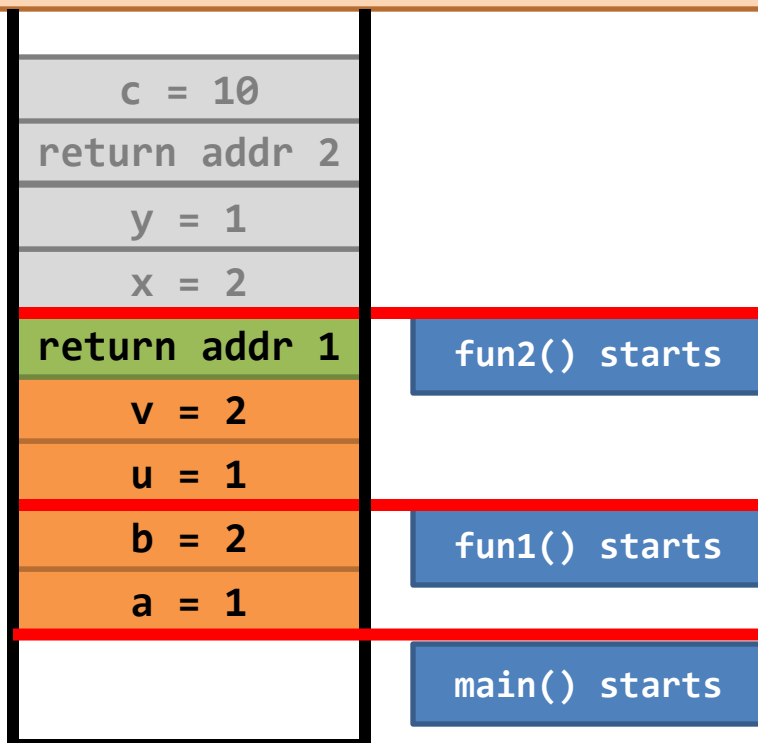
int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

# Stack – push & pop mechanisms

“Return” takes place.

- (1) Return value is written to the EAX register.
- (2) Stack **shrinks**.
- (3) CPU jumps back to **fun1()**.



```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}  
  
int fun1(int u, int v) {  
    return fun2(v, u);  
}  
  
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```



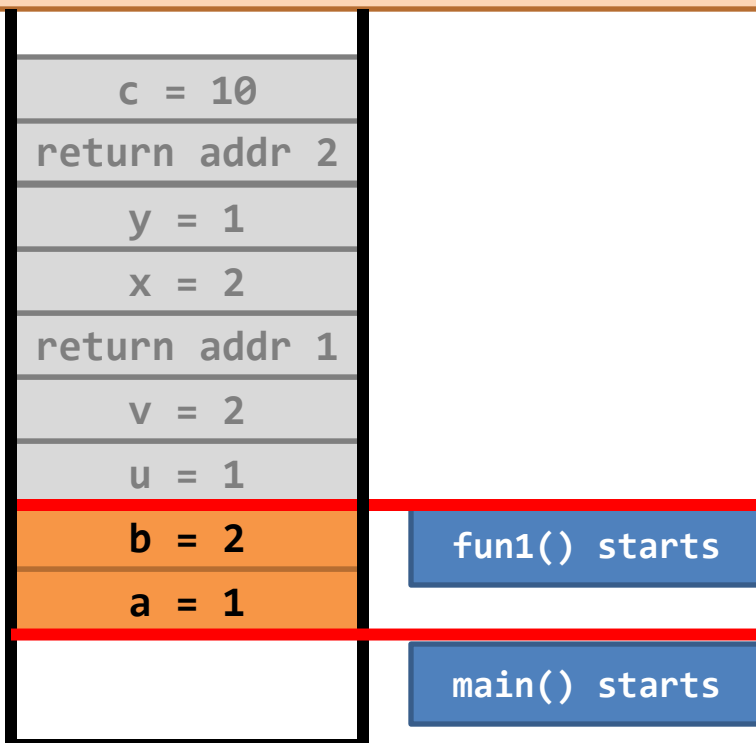
EAX: 13



# Stack – push & pop mechanisms

“Return” takes place.

- (1) Return value is written to the EAX register.
- (2) Stack **shrinks**.
- (3) CPU jumps back to `main()`.

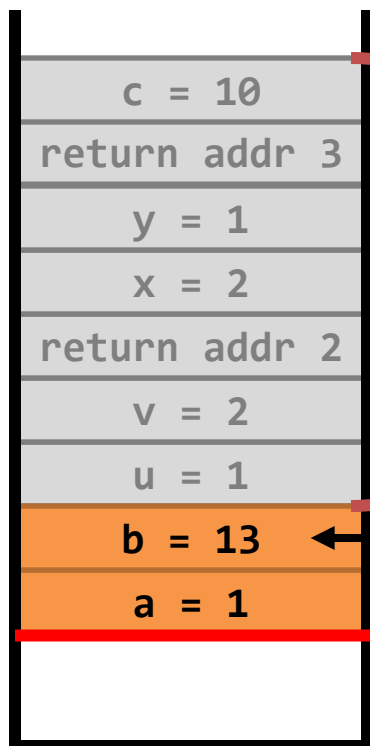


```
int fun2(int x, int y) {  
    int c = 10;  
    return (x + y + c);  
}  
  
int fun1(int u, int v) {  
    return fun2(v, u);  
}  
  
int main(void) {  
    int a = 1, b = 2;  
    b = fun1(a, b);  
    return 0;  
}
```

EAX: 13



# Stack – push & pop mechanisms



Those memory is NOT returned to the OS!!

Those memory will be re-used when you call functions again.

Upon "return", the value of EAX is then copied to "b"

main() starts

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

EAX: 13



# Stack – push & pop mechanisms

c = 10
return addr 3
y = 1
x = 2
return addr 2
v = 2
u = 1
b = 13
a = 1

Eventually, the main function reaches “return 0”.

This takes the CPU pointing to the C library.

Inside the C library, we will eventually reach the system call **exit()**.

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

EAX: 0



# Stack – limits

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size      (kbytes, -d) unlimited
.....
stack size         (kbytes, -s) 8192
.....
$ _
```

So, the limit is:  
 $8192 \times 1024 = 8\text{MB}$ .

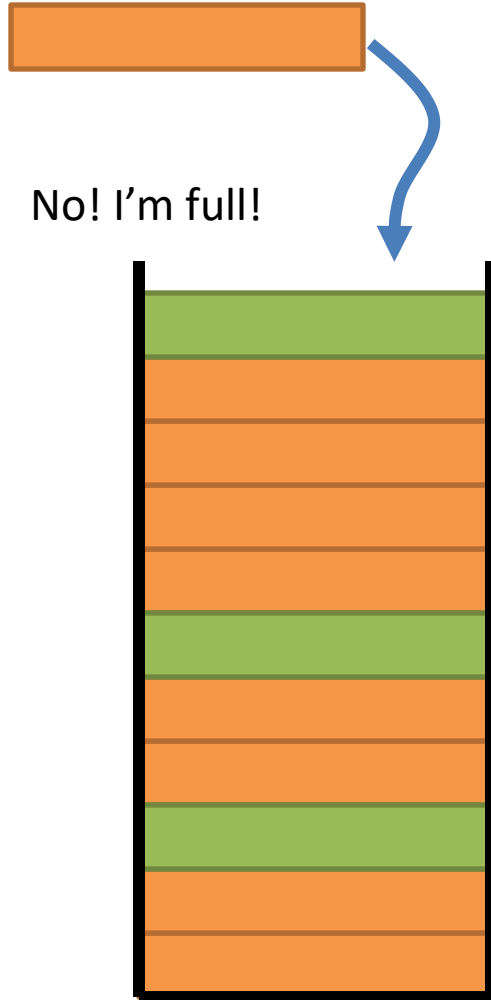
Can you define a local array larger than the limit?

**Segmentation  
fault**

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size      (kbytes, -d) unlimited
.....
stack size         (kbytes, -s) 8192
.....
$ ulimit -s 81920
```

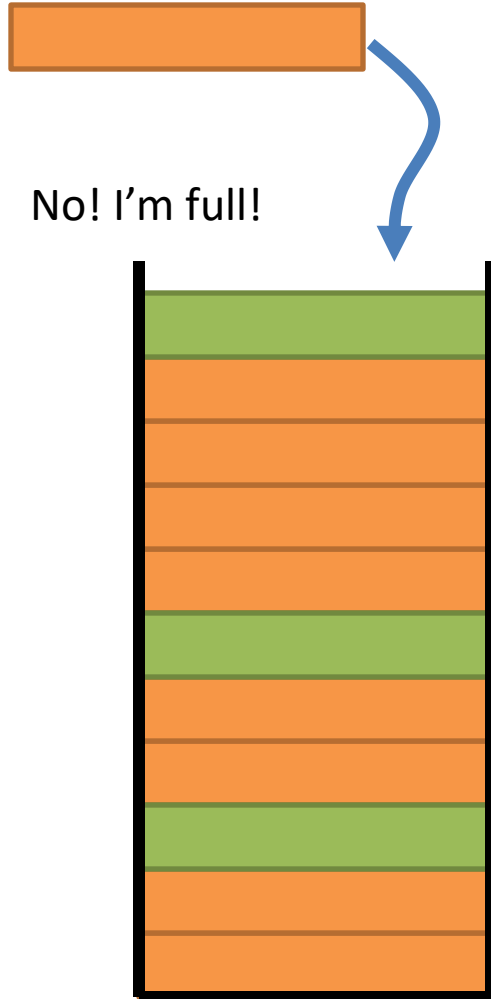
Now, the limit is:  
 $81920 \times 1024 = 80\text{MB}$ .

# Stack – summary



- What if it is a chain of endless recursive function calls?
- What will happen?
  - **Exception caught by the CPU!**
    - **Stack overflow exception!**
  - **Program terminated!**

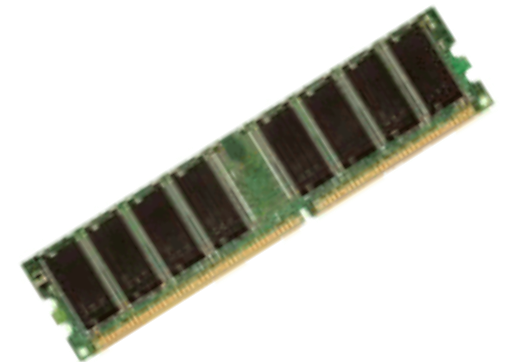
# Stack – summary



- *“I really need to play with recursions.”* Any workaround?
  - Minimize the number of arguments
  - Minimize the number of local variables
  - Minimize the number of calls
  - Use global variables
- Note: A function can ask the CPU **to read and to write anywhere in the stack**, not just the “zone” belonging to the running function!
  - Isn't it horrible (profitable and fun)?

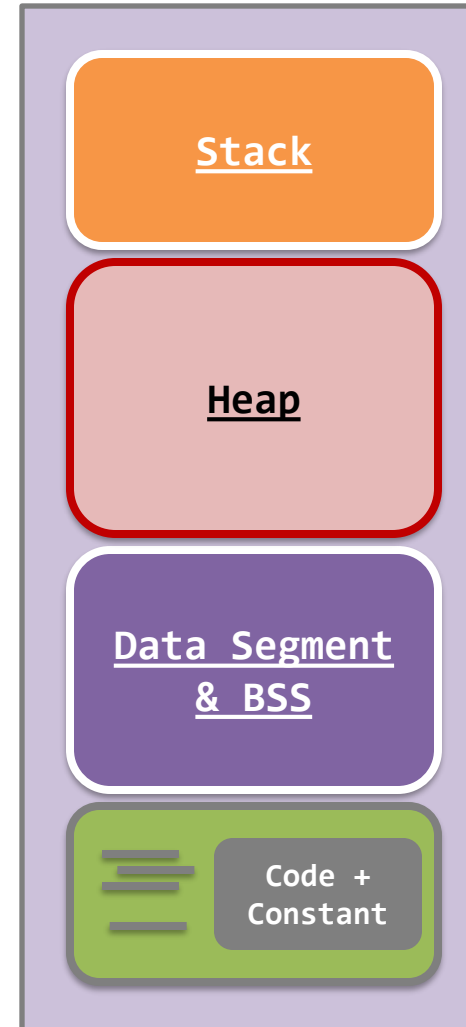
# User-space memory management

- Address space;
- Code & constants;
- Data segment;
- Stack;
- **Heap;**
- Segmentation fault;



# Dynamically allocated memory – properties

- Its name tells you its nature:
  - The dynamically allocated memory is called the **heap**.
    - Don't mix it up with the binary heap;
    - It has nothing to do with the binary heap.
  - **Dynamic**: not defined at compile time.
  - **Allocation**: only when you ask for memory, you would be allocated the memory.

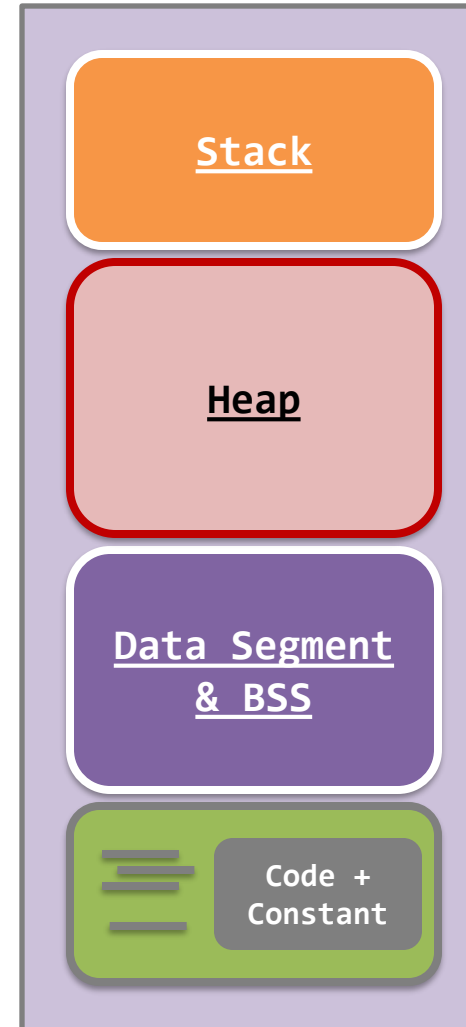


# Dynamically allocated memory – properties

- Lecturers of a programming course would tell you the following:

- *“**malloc()**” is a function that allocates memory for you.*
- *“**free()**” is a function that gives up a piece of memory that is produced by previous “**malloc()**” call.*

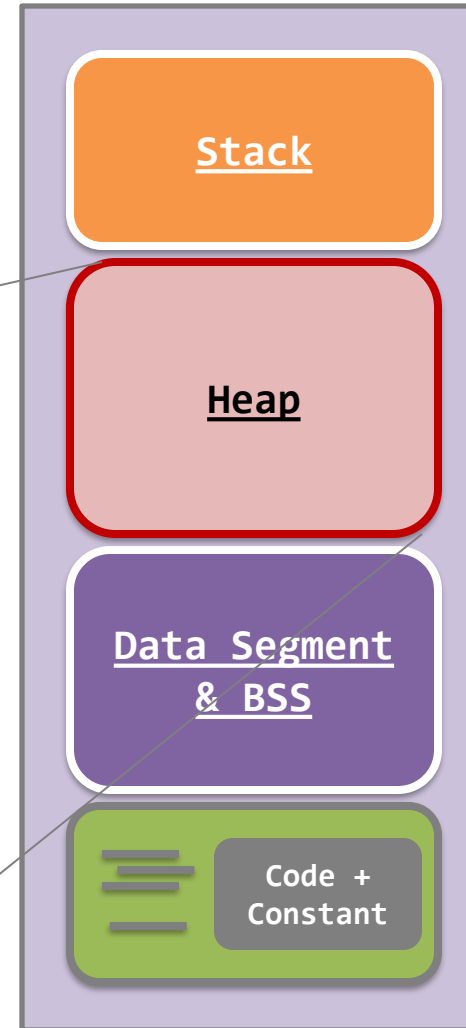
- The lecturer of the OS course is **to define and to defy** what you know about the **malloc()** and **free()** library functions.



# malloc()

When a program just starts running, the entire heap space is unallocated, or empty.

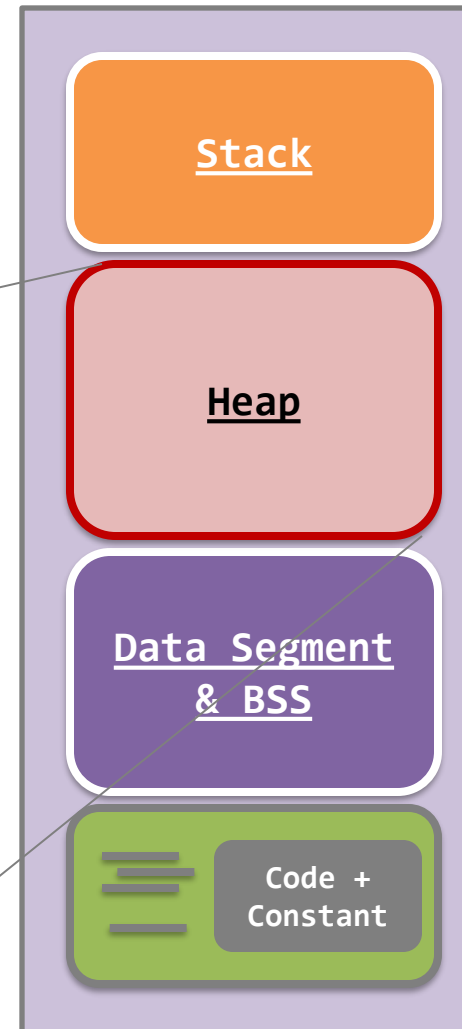
An empty heap.



# malloc()

When “**malloc()**” is called, the “**brk()**” system call is invoked accordingly.

“**brk()**” allocates the space required by “**malloc()**”. But, it doesn’t care how “**malloc()**” uses the space.

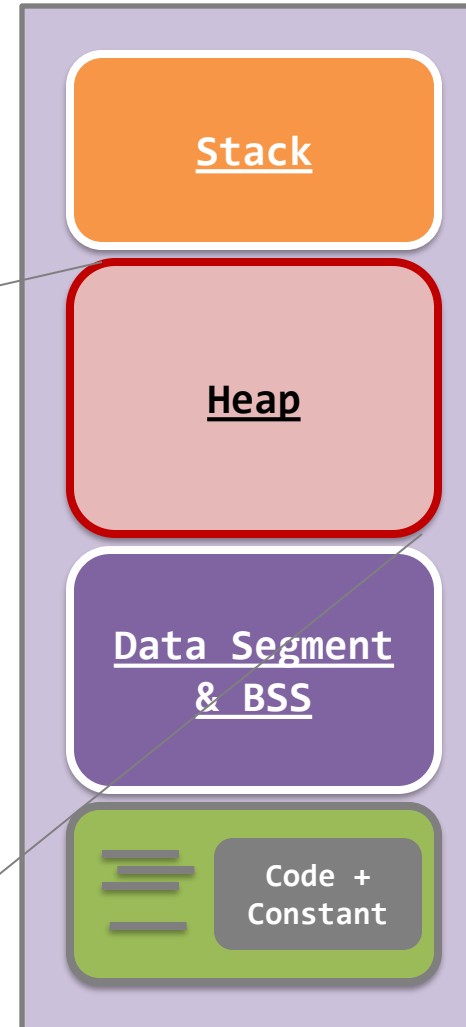
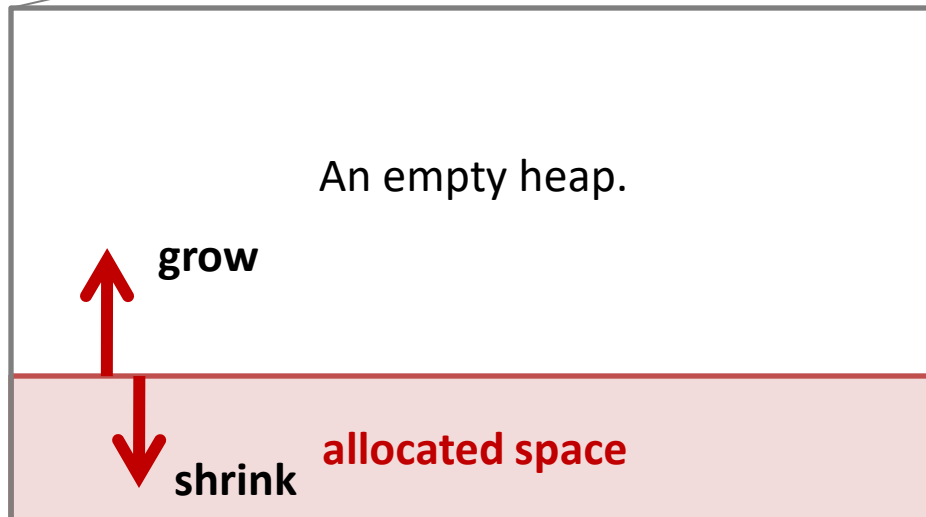


# malloc()

The allocated space growing or shrinking depends on the further actions of the process. That means the “**brk()**” system call can **grow or shrink** the allocated area.

In **malloc()**, the library call just invoke **brk()** for growing the heap space.

The **free()** call may shrink the heap space.



# malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = (char *)malloc(16);  
    ptr2 = (char *)malloc(16);  
  
    printf("Distance between ptr1 and  
           ptr2 - ptr1);  
    return 0;  
}
```

The return value of **malloc()** is of type “**void \***”, which means it is just a memory address only, and can be of any data types.

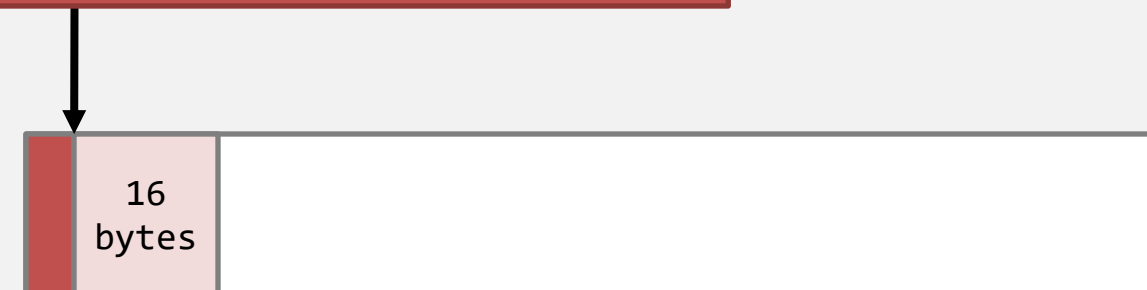
Such a memory address is the starting address of a piece of memory of 16 bytes (“16” is the request of **malloc()** call).



# malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = (char *)malloc(16);  
    ptr2 = (char *)malloc(16);  
  
    printf("Distance between ptr1 and ptr2: %d bytes\n",  
          ptr2 - ptr1);  
    return 0;  
}
```

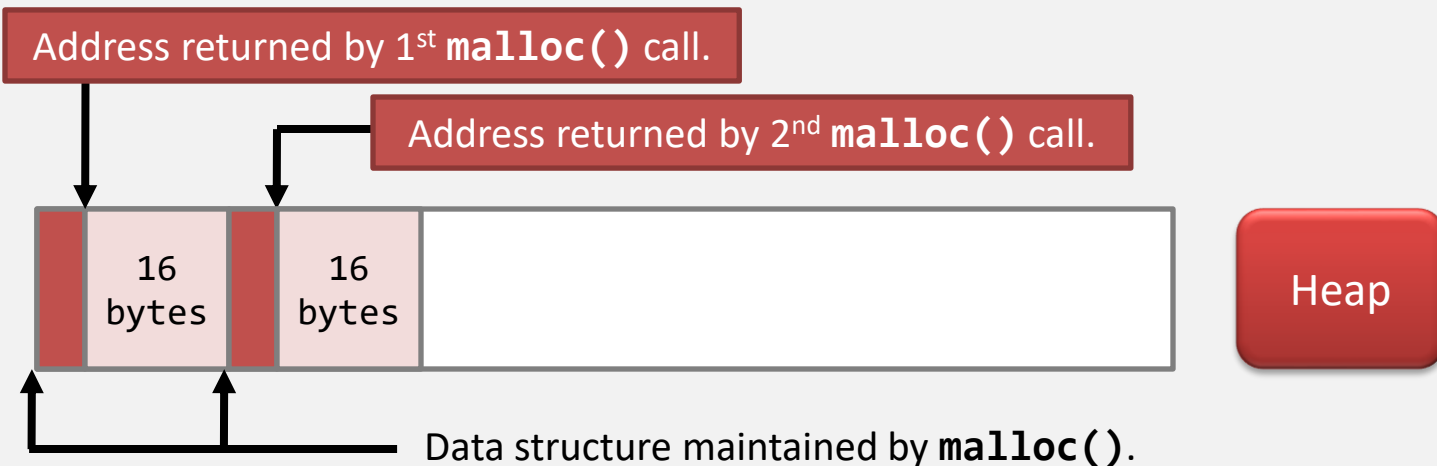
Address returned by 1<sup>st</sup> `malloc()` call.



Data structure maintained by `malloc()`.

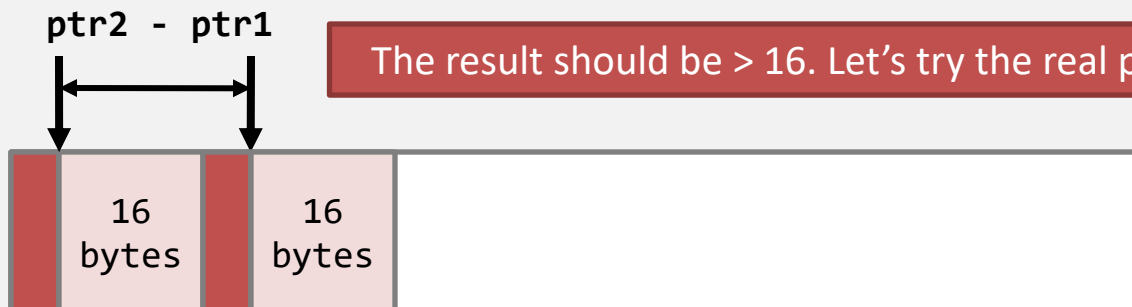
# malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = (char *)malloc(16);  
    ptr2 = (char *)malloc(16);  
  
    printf("Distance between ptr1 and ptr2: %d bytes\n",  
          ptr2 - ptr1);  
    return 0;  
}
```



# malloc()

```
int main(void) {  
    char *ptr1, *ptr2;  
    ptr1 = malloc(16);  
    ptr2 = malloc(16);  
  
    printf("Distance between ptr1 and ptr2: %d bytes\n",  
          ptr2 - ptr1);  
    return 0;  
}
```

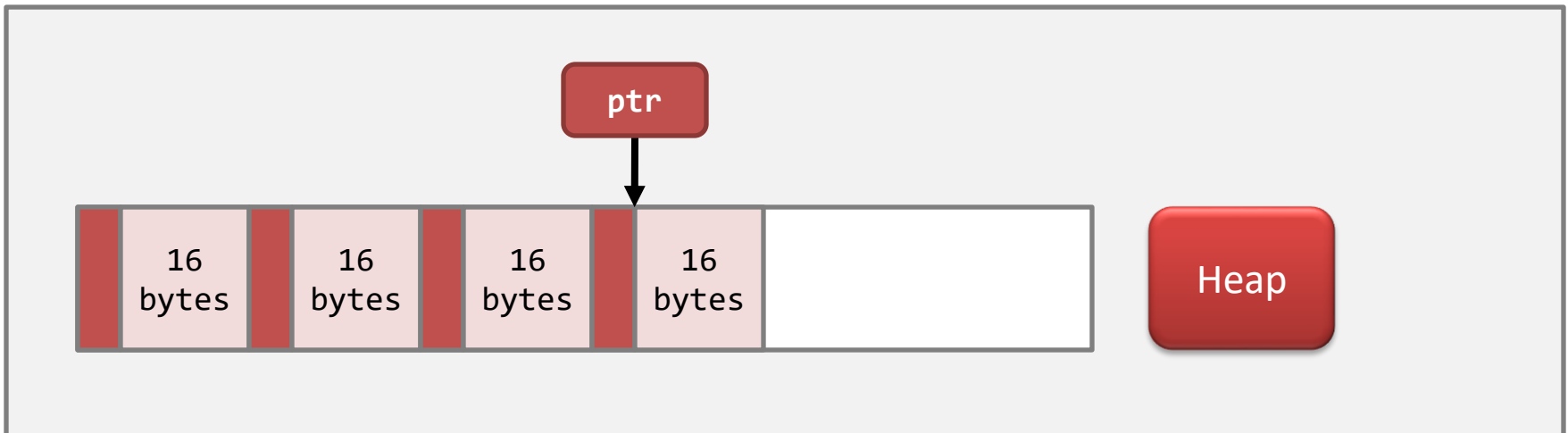


The result should be  $> 16$ . Let's try the real program!

Heap

# free()

- “**free()**” *seems to* be the opposite to “**malloc()**”:
  - It de-allocates any allocated memory.
  - When a program calls “**free(ptr)**”, then the address “**ptr**” must be the start of a piece of memory obtained by a previous “**malloc()**” call.



# free() – case #1

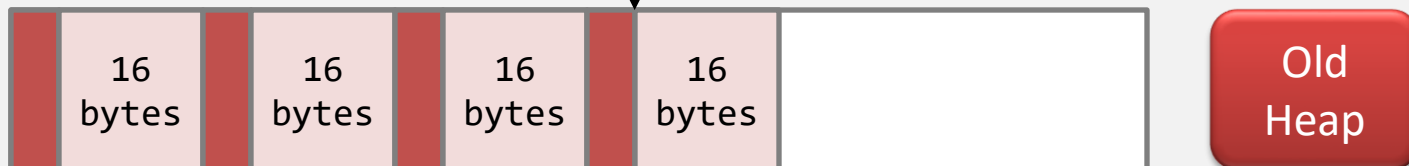
- Case #1: de-allocating the last block.

This is accomplished by calling `brk()` system call. This heap has become smaller.



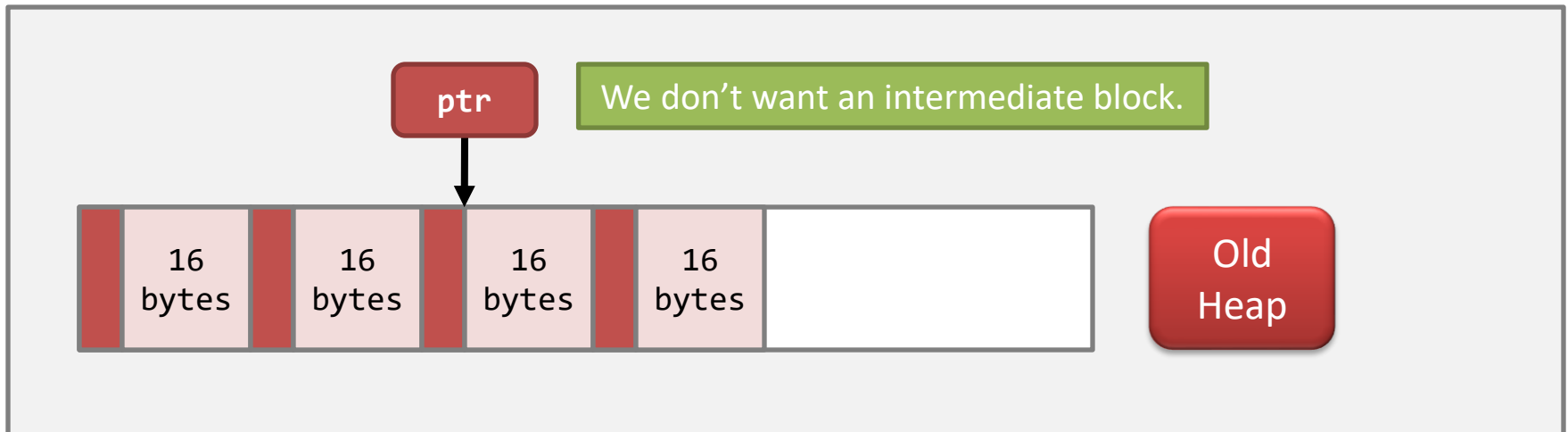
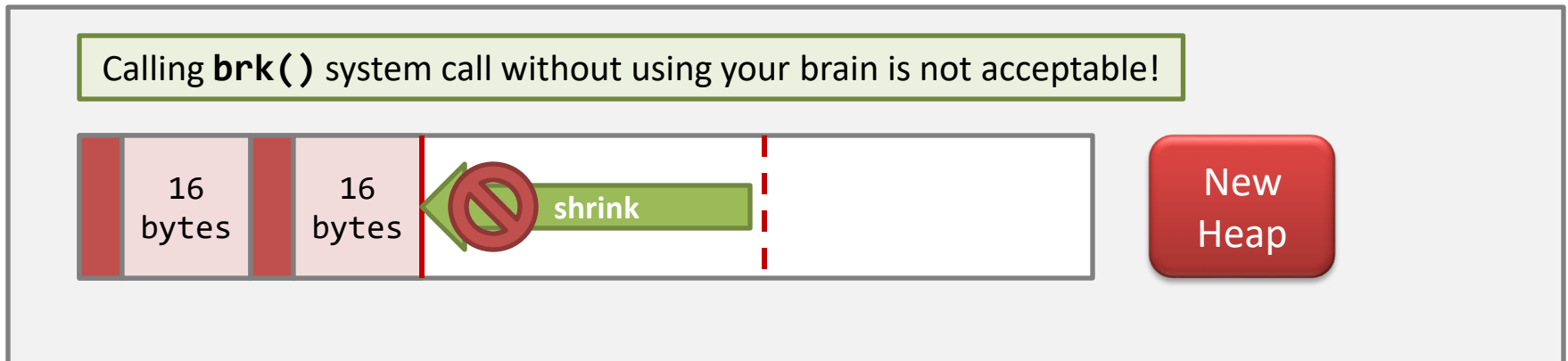
ptr

The last block is not needed.



# free() – case #2

- Case #2: de-allocating an intermediate block.



# free() – case #2

- Case #2: de-allocating an intermediate block.

Here comes the role of the data structure created by `malloc()`!

This pointer is used for creating a linked list of de-allocated block.

This size record the size of de-allocated block.

NULL

size

address

32-bit system:  $4+4 = 8$  bytes  
64-bit system:  $8+8 = 16$  bytes

16  
bytes

16  
bytes

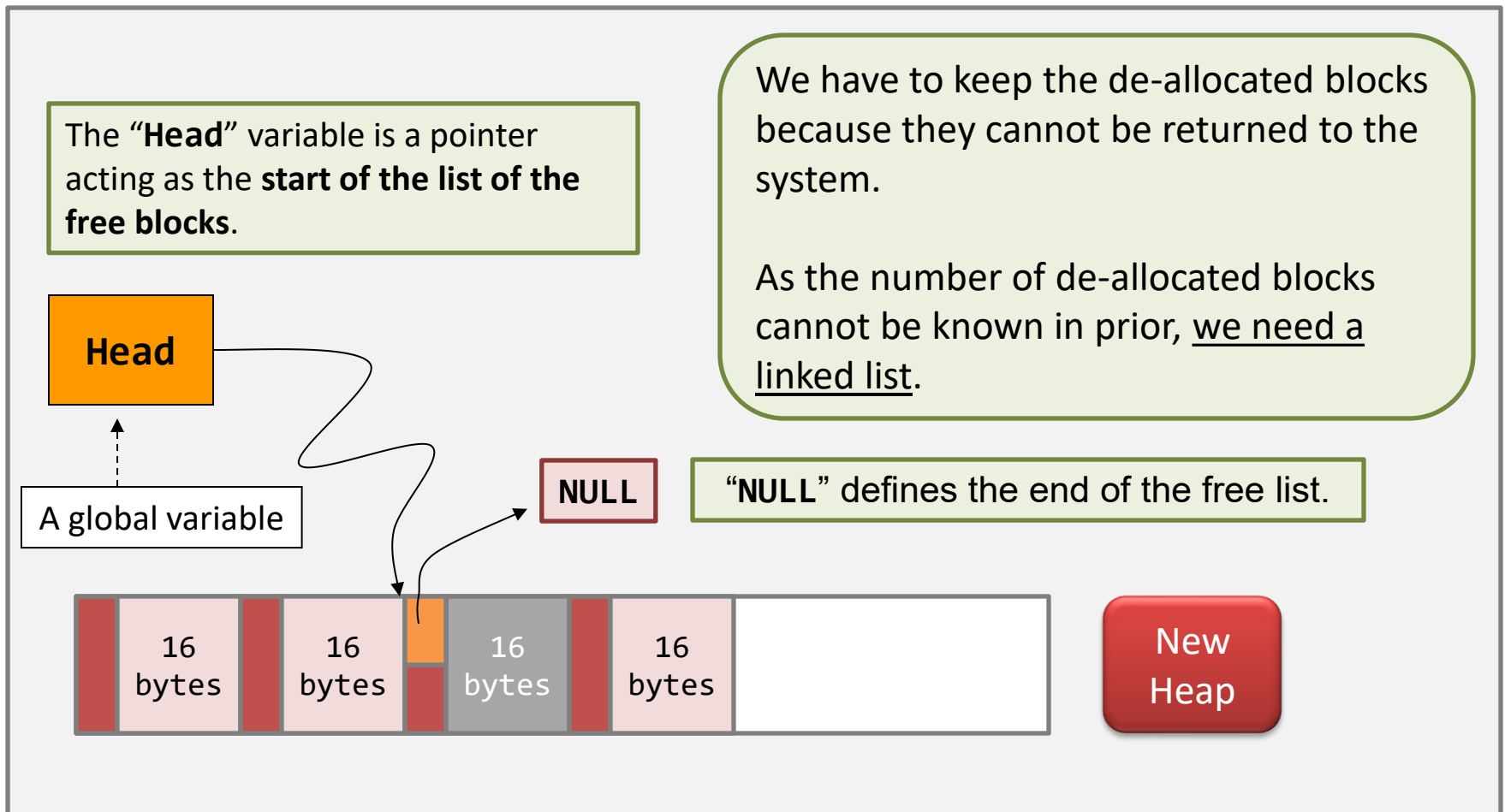
16  
bytes

16  
bytes

Heap

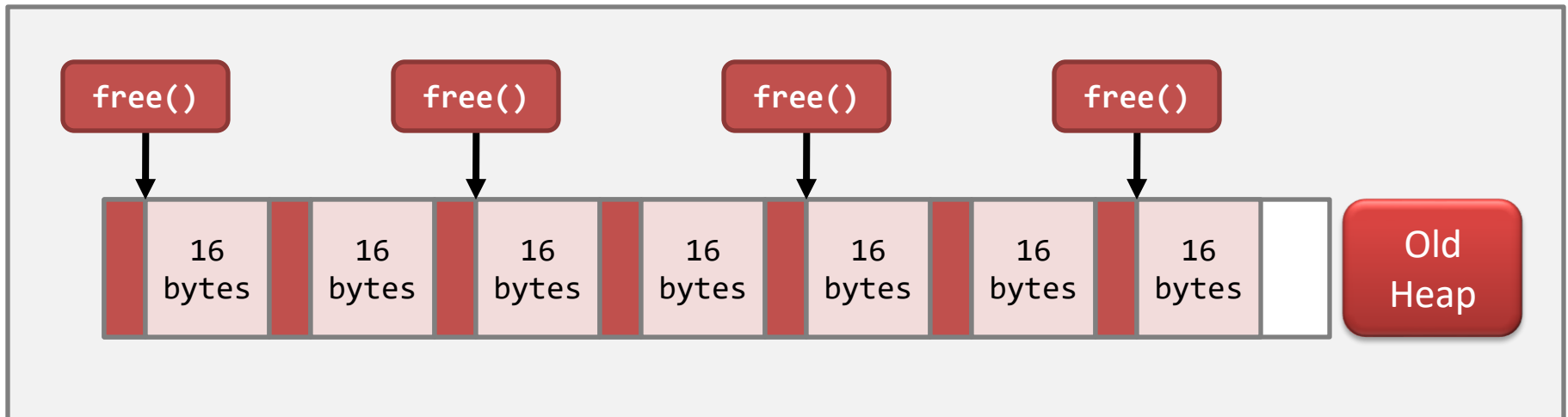
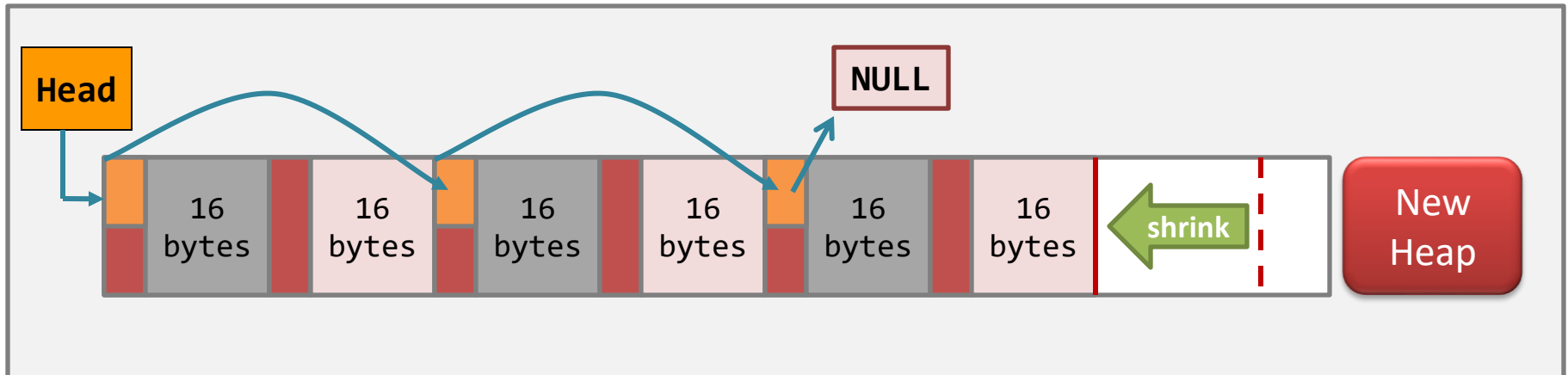
# free() – case #2

- Case #2: de-allocating an intermediate block.



# free() – case #2

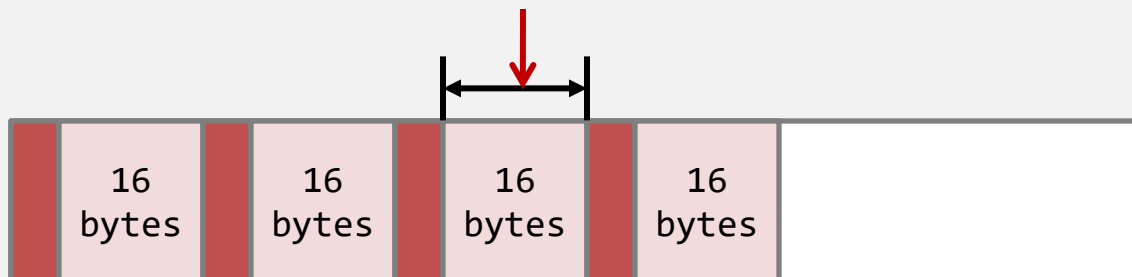
- Case #2: another example.



# free() – cautions

- The calling program is **assumed** to be carefully written.
  - After **malloc()** has been invoked, the program should read and write inside the requested area only.
  - Now, you know why you'd **have troubles** when you write data outside the allocated space.

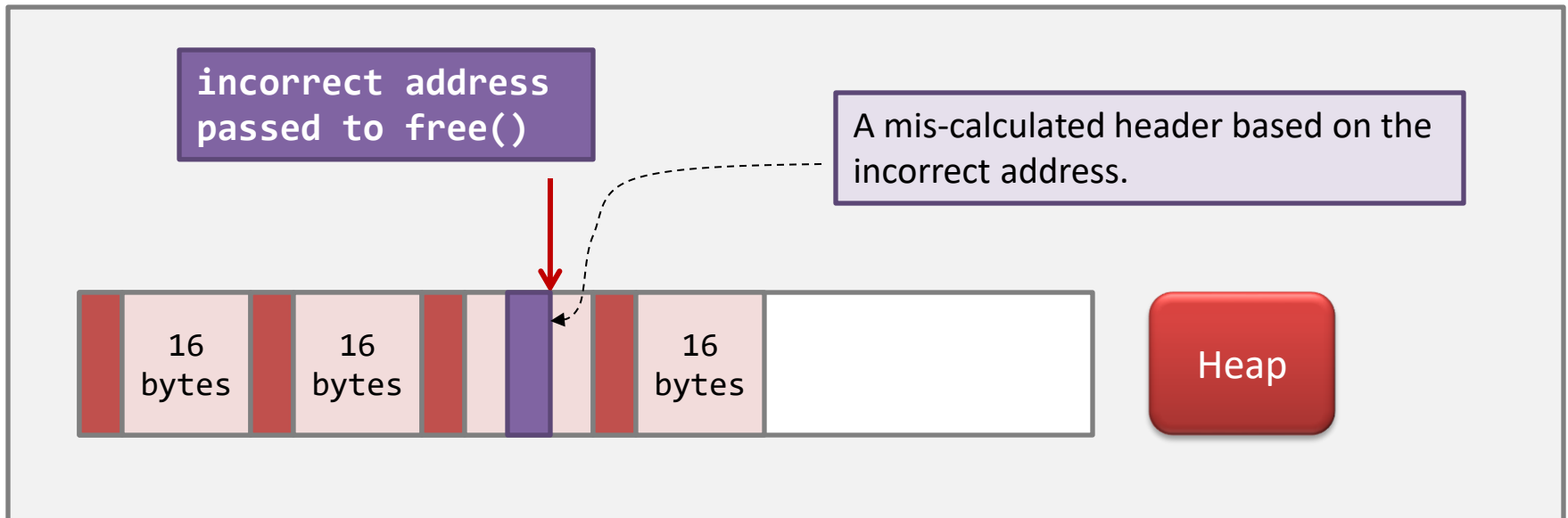
You can only play within this zone. Please behave!  
Note: be careful of the **consequences** of misbehaves...



Heap

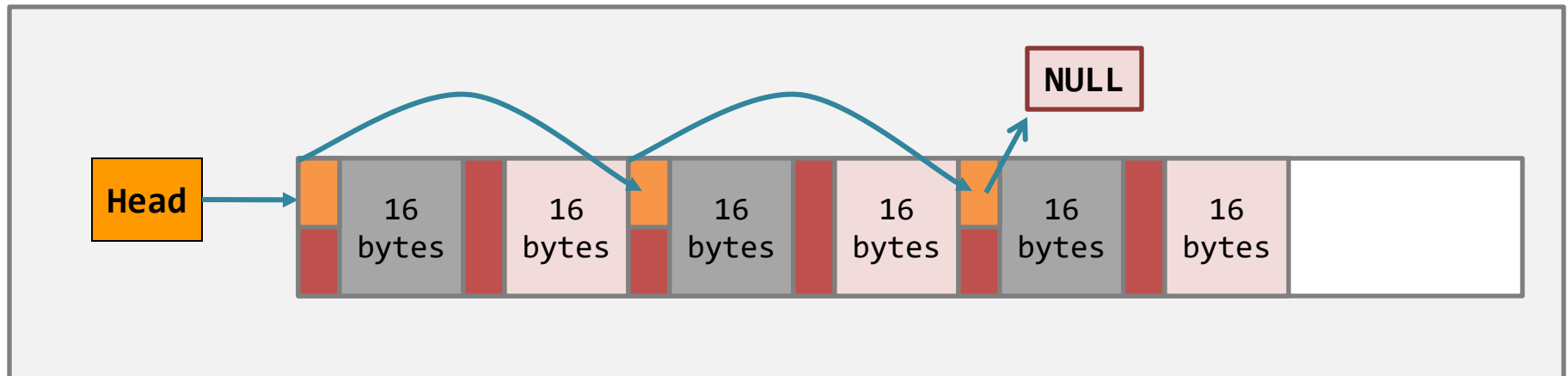
# free() – cautions

- The calling program is **assumed** to be carefully written.
  - When **free()** is called, the program should provide **free()** with the correct address...
    - i.e., the address previously returned by a **malloc()** call.



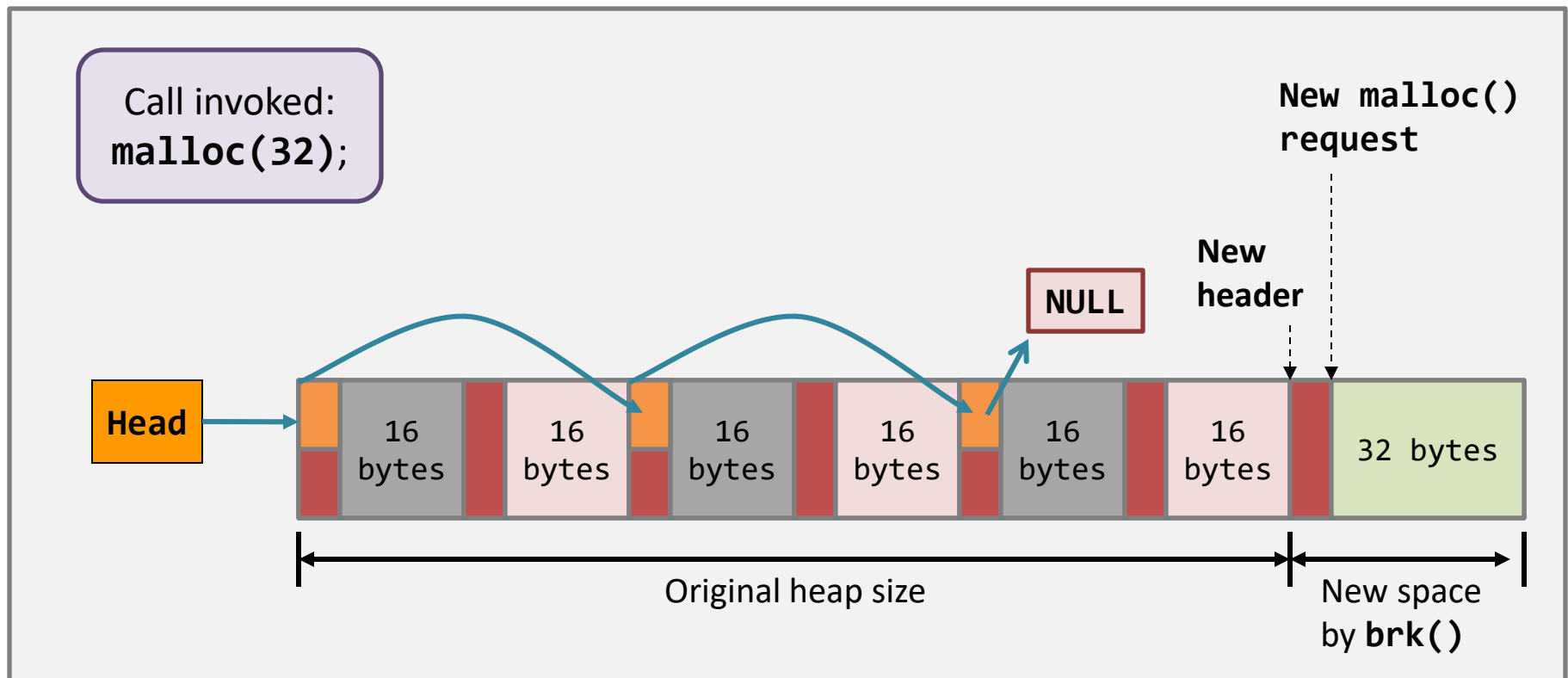
# When `malloc()` meets free blocks...

- Problem: whether to use the free blocks or not?
  - *Is there any free block that is **large enough** to satisfy the need of the `malloc()` call?*



# When `malloc()` meets free blocks...case #1

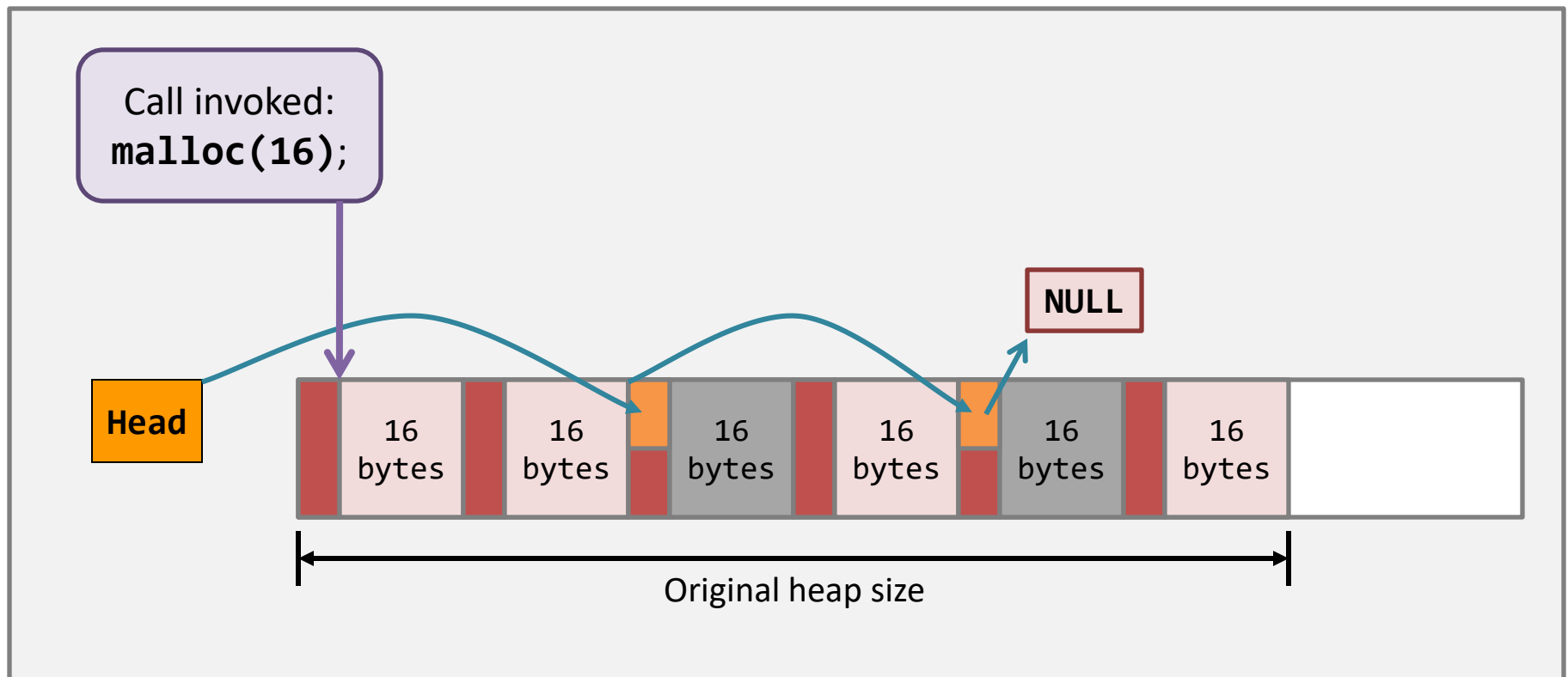
- Case #1: if there is **no suitable free block**...
  - then, the `malloc()` function should call `brk()` system call...in order to claim more heap space.





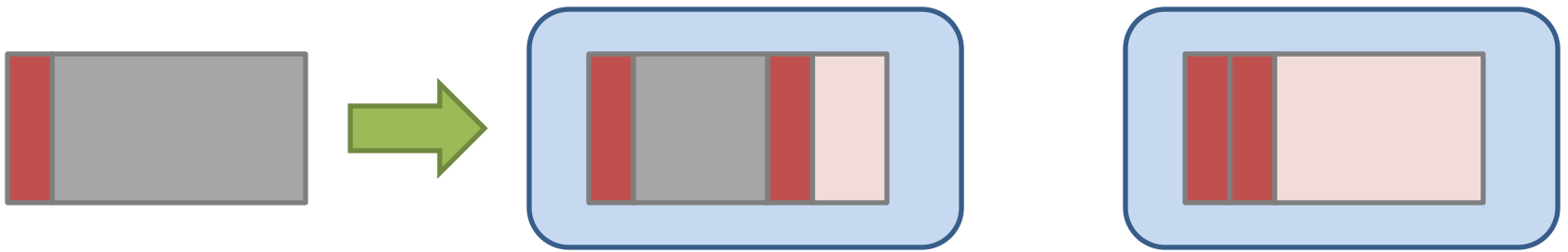
# When `malloc()` meets free blocks...case #2

- Case #2: if there is a suitable free block
  - the `malloc()` function should reuse that free block.



# When `malloc()` meets free blocks...

- There can be other cases:
  - A `malloc()` request that takes a partial block;
  - A `malloc()` request that takes a partial block, but leaving no space in the previously free block.



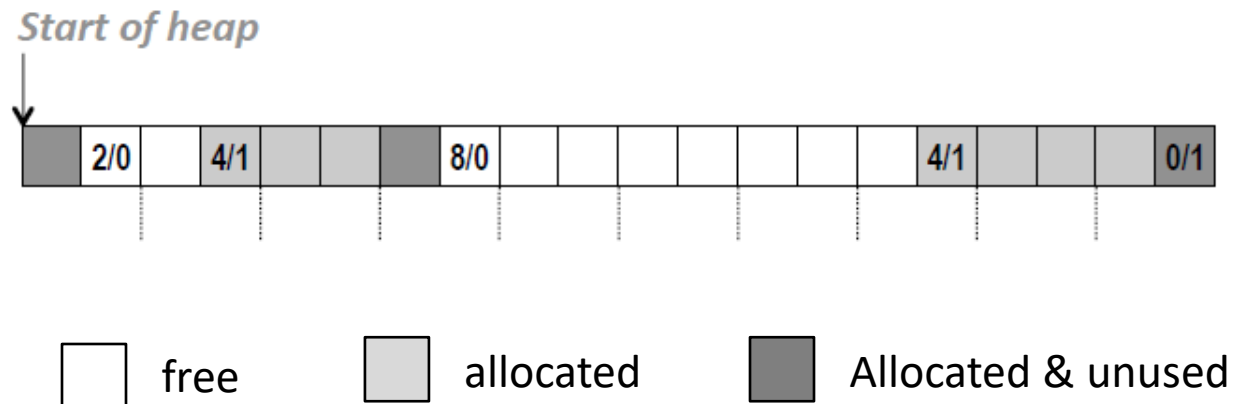
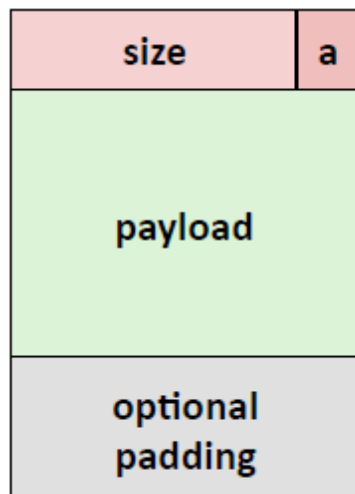
- We will skip those subtle cases...
  - It boils to implementation only...
  - You already have the **big picture** about `malloc()` and `free()`.

# When `malloc()` meets free blocks...

- Now, let us look at some implementations...

# Implicit free list

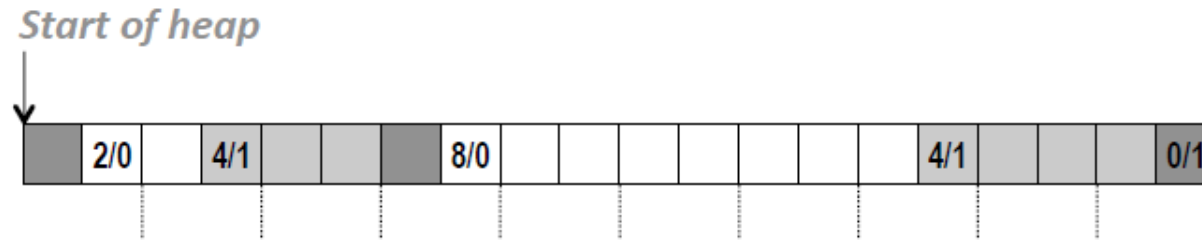
- Needs two information for each block
  - size & is\_allocated



How about memory allocation and free?

# Implicit free list

- **Contiguous Allocation:** May need linear time search



**First fit:** allocate the first hole that is big enough (fast)

**Next fit:** similar to first fit, but start where previous search finishes

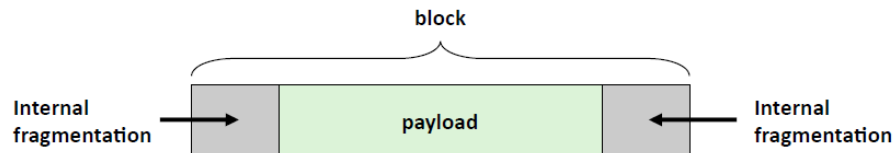
**Best fit:** allocate the smallest hole that is big enough (helps fragmentation, larger search time)

**Worst fit:** allocate the largest hole

– Allocate the whole block or splitting

# Fragmentation

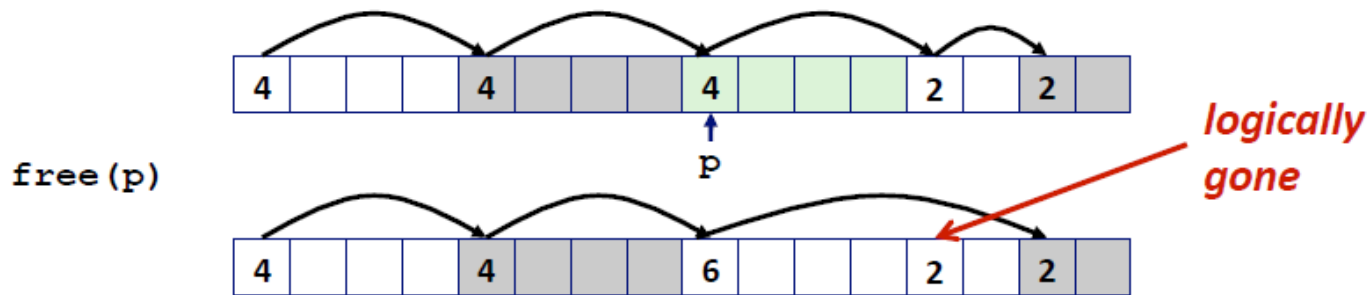
- External fragmentation
  - The heap memory looks like a map with many holes
  - It is the source of inefficiency because of the **unavoidable search for suitable space**
  - Sol: Compaction (need to move data to merge free mem)
- Internal fragmentation
  - Payload is smaller than allocated block size



# Implicit free list

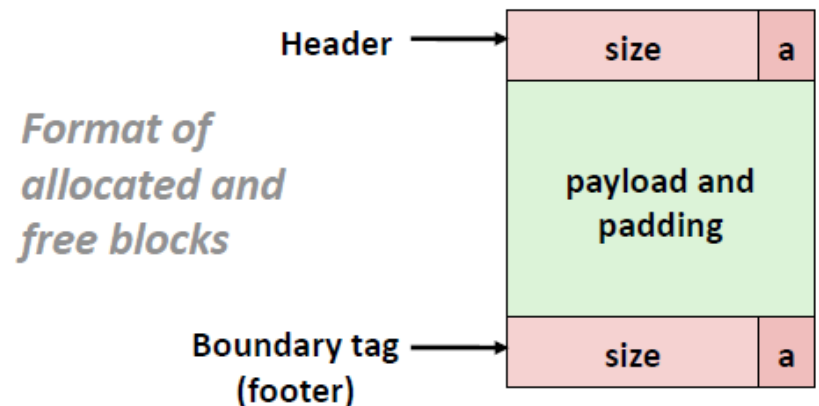
- Free: Coalescing

- Coalescing with next block: easy



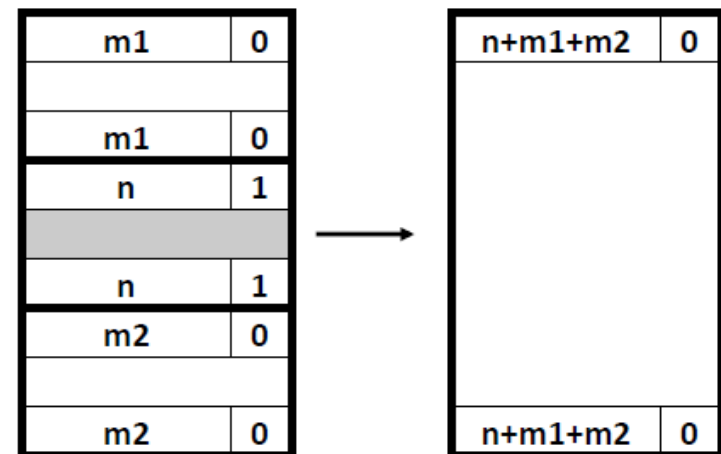
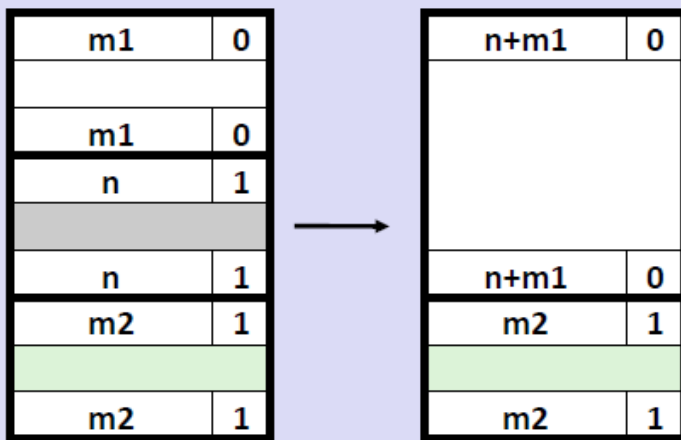
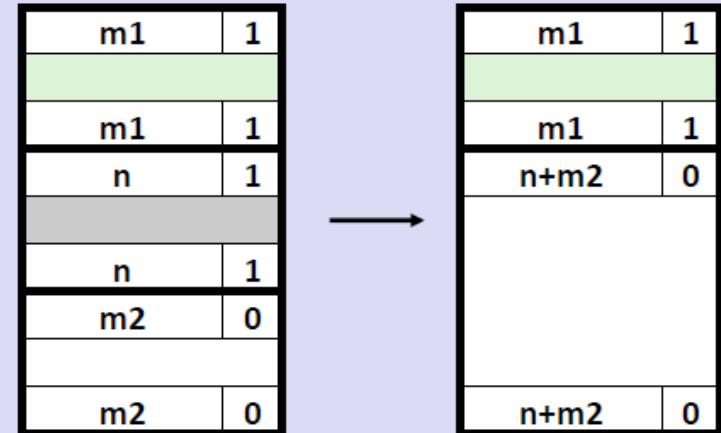
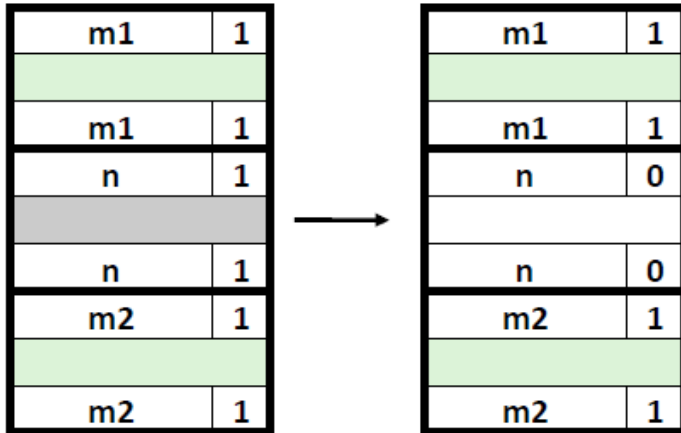
- How about coalescing with previous block?

- [Knuth 73] Add a boundary tag in the footer



# Implicit free list

- Constant time coalescing w/ boundary tag (4 cases)

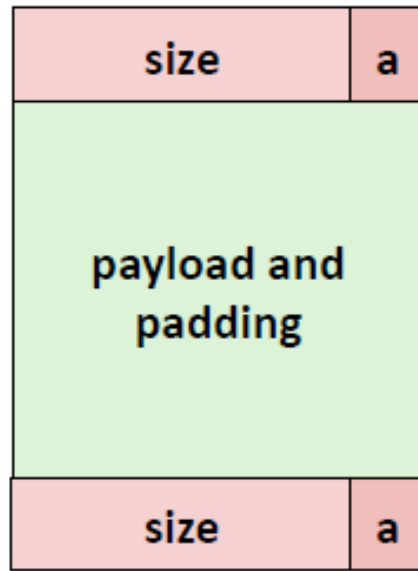


# Implicit free list: summary

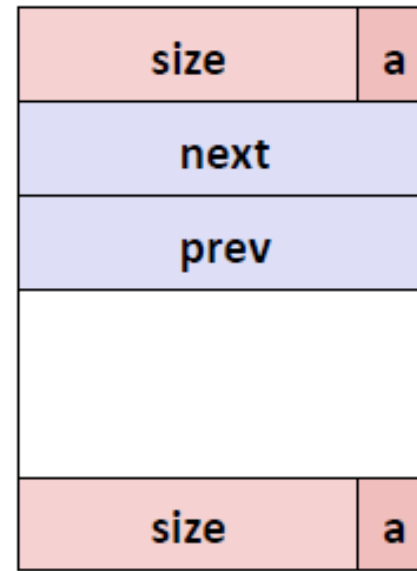
- May not be used in practical malloc() and free() implementations
  - High memory allocation cost
- Some ideas are still useful and important
  - Splitting available blocks
  - Boundary tag

# Explicit free list

Allocated (as before)



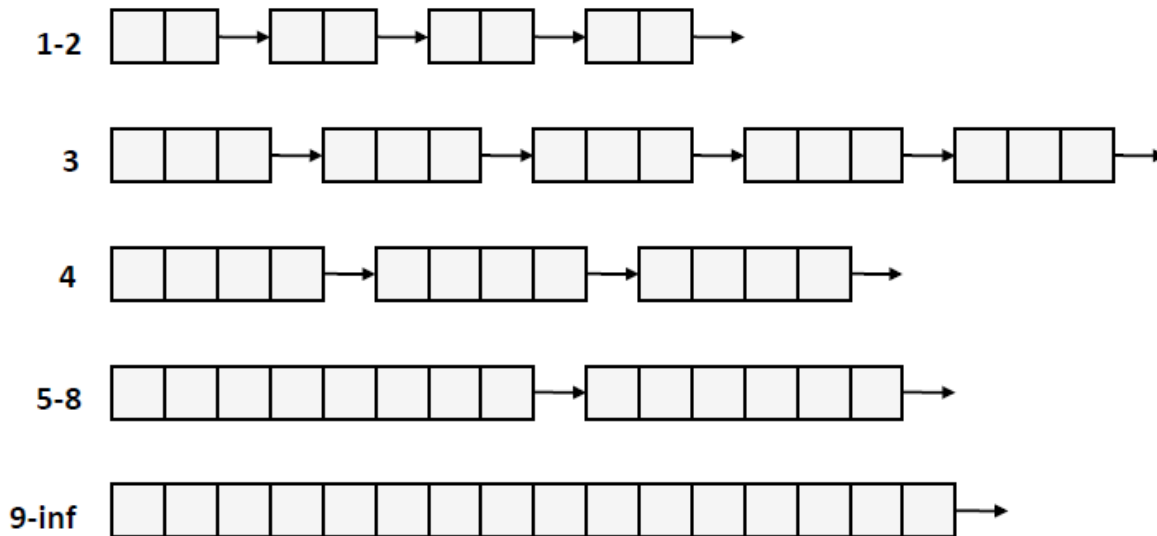
Free



- Track only free blocks (LIFO or address-ordered)
- Block splitting is useful in allocation
- Boundary tag is still useful in coalescing

# Segregated free list

- Segregated free list (分离空闲链表)
  - Different free lists for different size classes



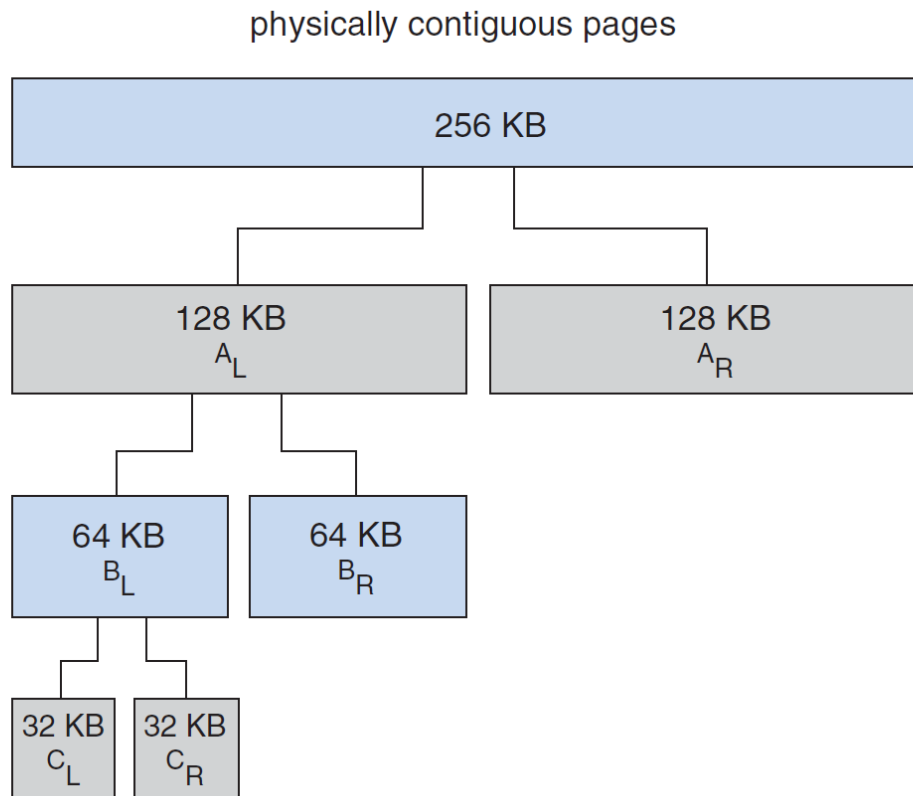
## – Allocation

- Search appropriate list (larger size)
- Found and split
- Not found: search next

Approximates best -fit

# Segregated free list

- Special example
  - Buddy system (power-of-two block size)



# Issues raised by malloc() and free()

- The kernel knows how much memory should be given to the heap.
  - When you call **brk()**, the kernel tries to find the memory for you.
- Then...one natural question...
  - Is it possible to **run out of memory (OOM)**?

# Out of memory?

- Try this!

```
#define ONE_MEG 1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

Is it safe to run this program on a 32-bit machine?

What is the output?

```
Allocated 3052 MB
Allocated 3053 MB
Allocated 3054 MB
Allocated 3055 MB
linux2:/uac/rshr/ykli>
```

# Out of memory?

- On 32-bit Linux, why does the OOM generator stop at around 3055MB?
- Still remember what we said when we are talking about data segment?
  - Every 32-bit Linux system has an **addressable memory space** of 4G-1 bytes.
  - The kernel reserves 1GB addressing space.

# Out of memory?

- Try this! Yet another OOM Generator!

```
#define ONE_MEG  1024 * 1024
char global[1024 * ONE_MEG];
int main(void) {
    void *ptr;
    int counter = 0;
    char local[8000 * 1024];
    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

Yet, what is the output?

```
Allocated 3044 MB
Allocated 3045 MB
Allocated 3046 MB
Allocated 3047 MB
linux2:/uac/rshr/ykli>
```

# Real OOM!

Explanation is in Part 2.

```
#define ONE_MEG 1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        memset(ptr, 0, ONE_MEG);
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

**Warning #1.** Don't run this program on any department's machines.

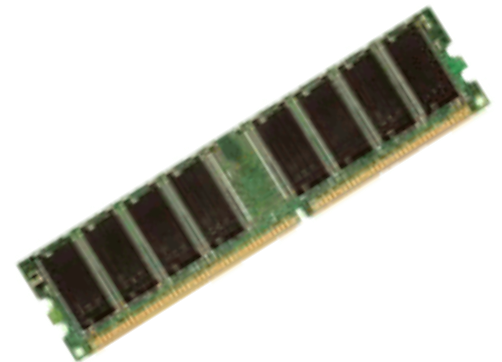
**Warning #2.** Don't run this program when you have important tasks running at the same time.

**Lazy allocation**

That is why previous programs run very fast.

# User-space memory management

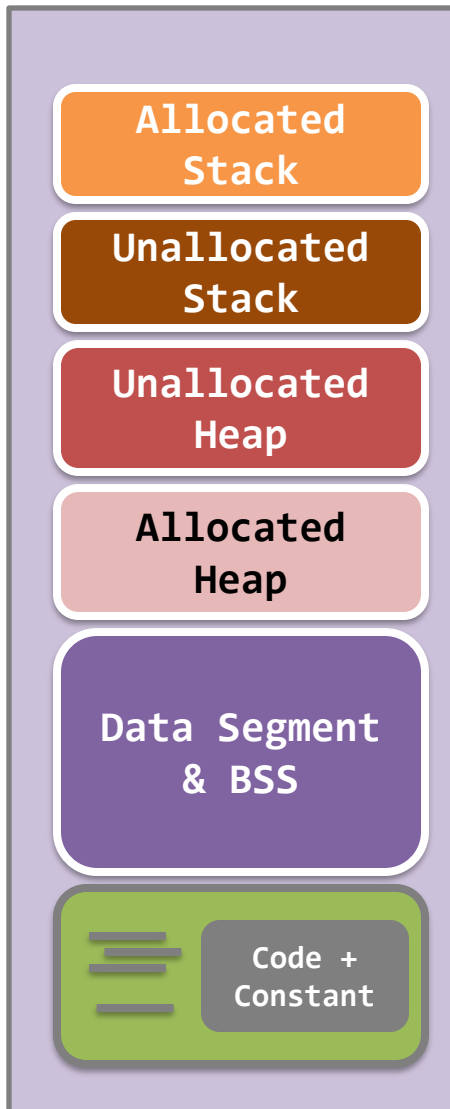
- Address space;
- Code & constants;
- Data segment;
- Stack;
- Heap;
- **Segmentation fault;**



# What is segmentation fault?

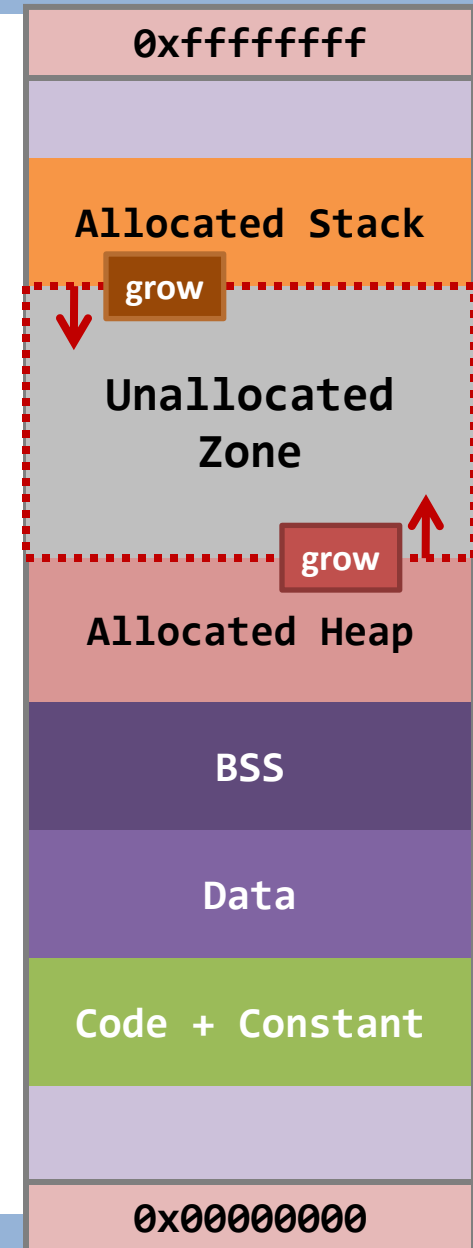
- Someone must have told you:
  - When you are accessing a piece of memory that is not allowed to be accessed, then the OS returns you an error called – segmentation fault.
- As a matter of fact, how many ways are there to generate a segmentation fault?

# What is segmentation fault?



Forget about the illustration, the memory in a process is separated into **segments**.

So, when you visit a segment in an illegal way, then...**segmentation fault**.



# How to “segmentation fault”?

Read	0xffffffff	Write
YES	Unusable	YES
NO	Allocated Stack	NO
YES	Unallocated Zone	YES
NO	Allocated Heap	NO
NO	BSS	NO
NO	Data	NO
NO	Code + Constant	YES
YES	Unusable	YES
	0x00000000	

# How to “segmentation fault”?

Read	0xffffffff	Write
YES	Unusable	YES
NO	Allocated Stack	NO
YES	Unallocated Zone	YES
NO	Allocated Heap	NO
NO	BSS	NO
NO	Data	NO
NO	Code + Constant	YES
YES	Unusable	YES
	0x00000000	

Now, we can understand:

```
char *ptr = NULL;
char c = *ptr;
```

will generate

**Segmentation fault**

NULL = 0x00000000



\*ptr is reading



# Summary of segmentation fault

- When you have a **so-called address** (maybe it is just a random sequence of 4 bytes), one of the following cases happens:

See if you have luck...

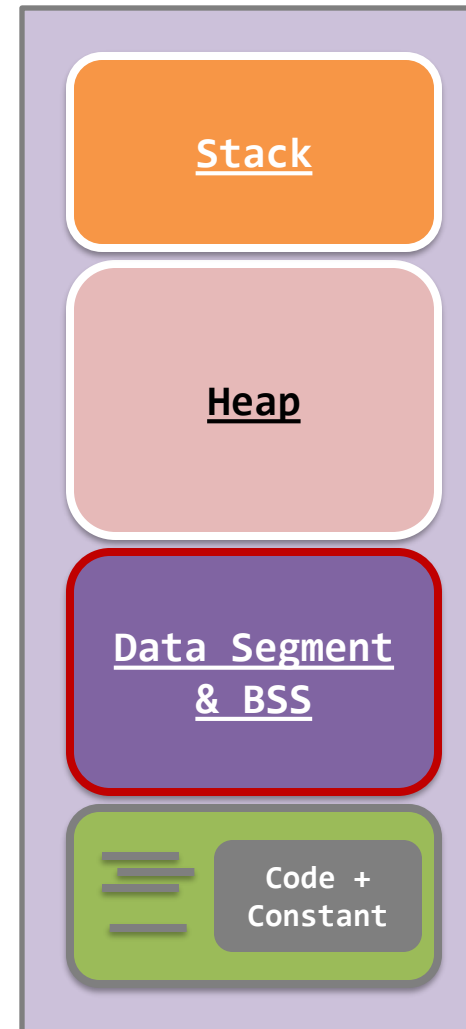
	Read-only segments	Allocated segments	Unused or unallocated segments
Reading	No problem	No problem	<b>Segmentation fault</b>
Writing	<b>Segmentation fault</b>	No problem	<b>Segmentation fault</b>

# Summary of segmentation fault

- Now, you know what is a segmentation fault, and the cause is always **carelessness!**
  - Now, you know why “**free()**” sometimes give you segmentation fault...
    - because **you corrupt the list of free blocks!**
  - Now, you know why “**malloc()**”-ing a space that is smaller than required is ok...
    - because **you are overwriting the neighboring blocks!**

# Summary of part 1

- Memory of a process is divided into segments (**segmentation**):
  - codes and constants;
  - global and static variables;
  - allocated memory (or heap);
  - local variables (or stack);
- When you access a memory that is not allowed, then the OS returns you **segmentation fault**
- **Every process' segments are independent and distinct.**



# Summary of part 1

- The dynamically allocated memory is not as simple as you learned before.
  - Allocating large memory blocks is not efficient; instead, **allocating small memory blocks** can make use of the **holes** in the heap memory efficiently.
  - Keep calling **malloc()** without calling **free()** is dangerous...
    - because there is no garbage collector in C or the OS...
    - **OOM error awaits you!**

**End of part 1**