Operating Systems

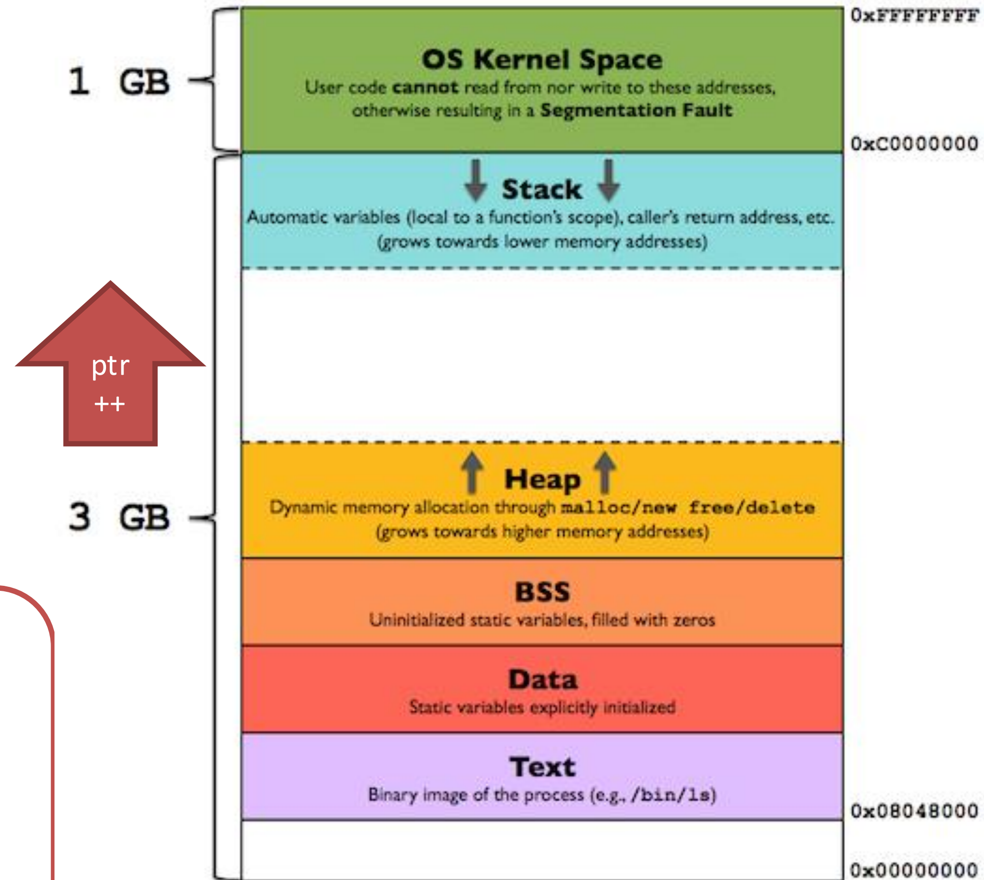
Eric Lo

6 – Memory Management

[Most contents assume 32-bit unsecure version]

Address space and Segment

```
$ ./addr
Local variable   = 0xbfa8938c
malloc() space   =  0x915c008
Global variable =  0x804a020
Code & constant =  0x8048550
$ _
```



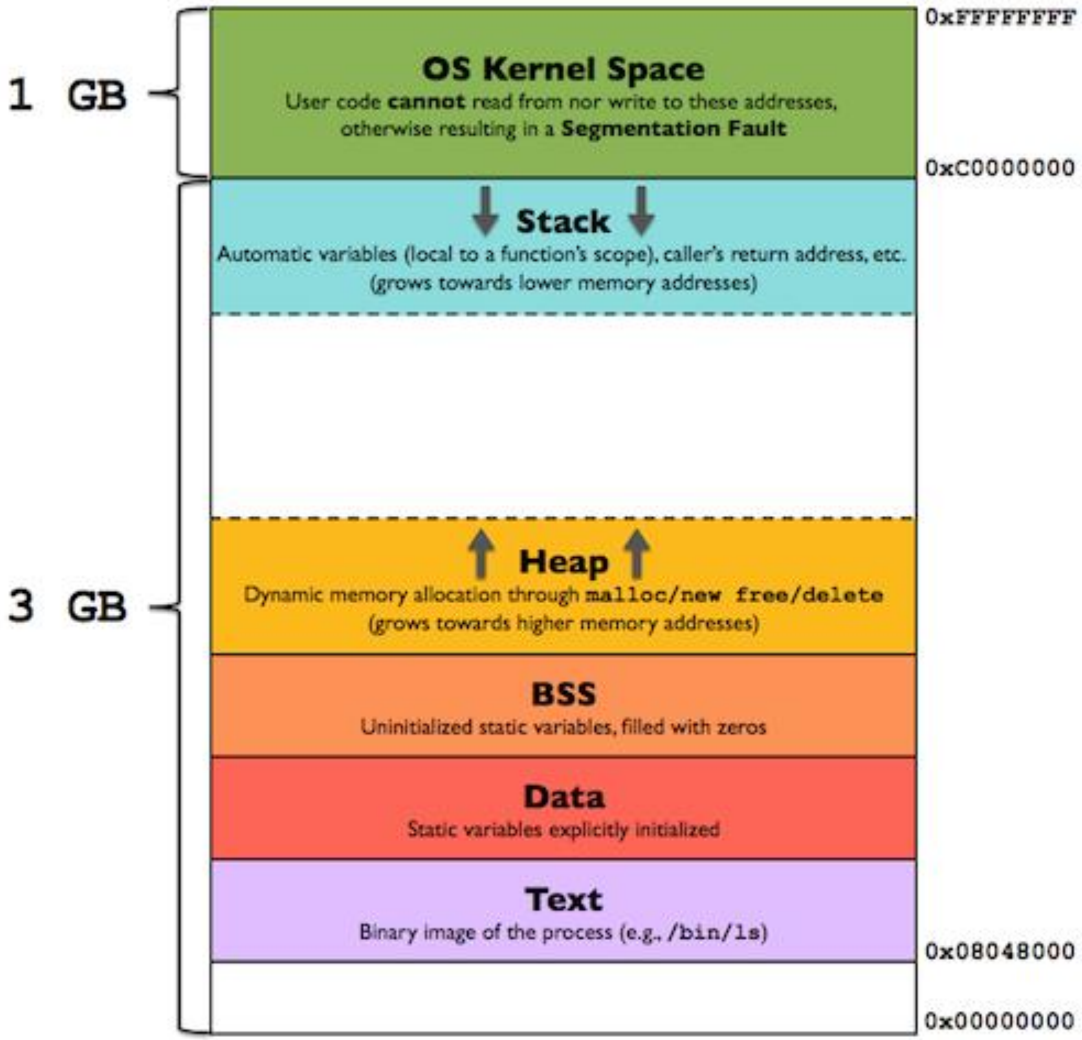
One address maps to one byte.
On a 32-bit system,
- The maximum amount of memory in a process is 2^{32} bytes = **4GB**.

Then, how about a 64-bit system?
= **16EB (256TB)**

```
[examples@3150] cat addr.c
```

… KB (2^{10}), MB (2^{20}), GB (2^{30}), TB (2^{40}), PB (2^{50}), <u>**EB**</u> (2^{60}), ZB

Virtual address space

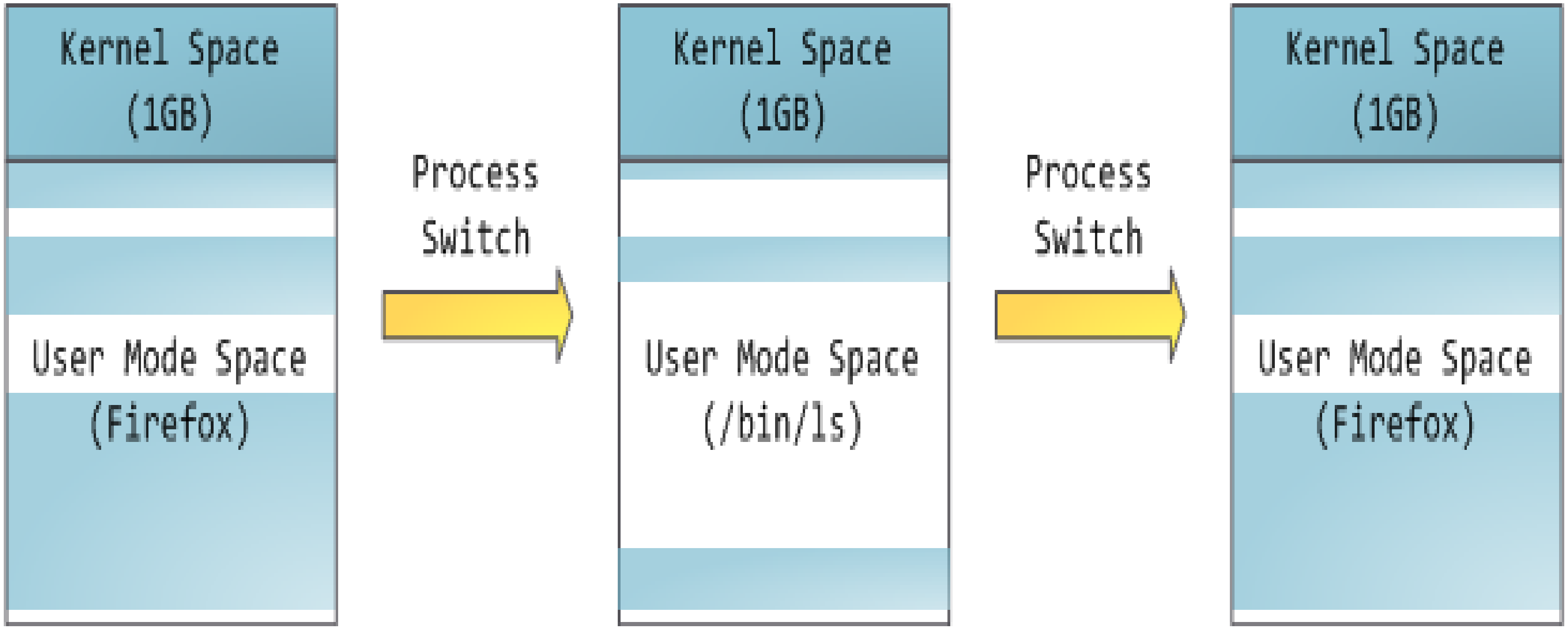


Each process has its own user (address) space

Each process thinks it has 2^{32} (256TB) byte memory

https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

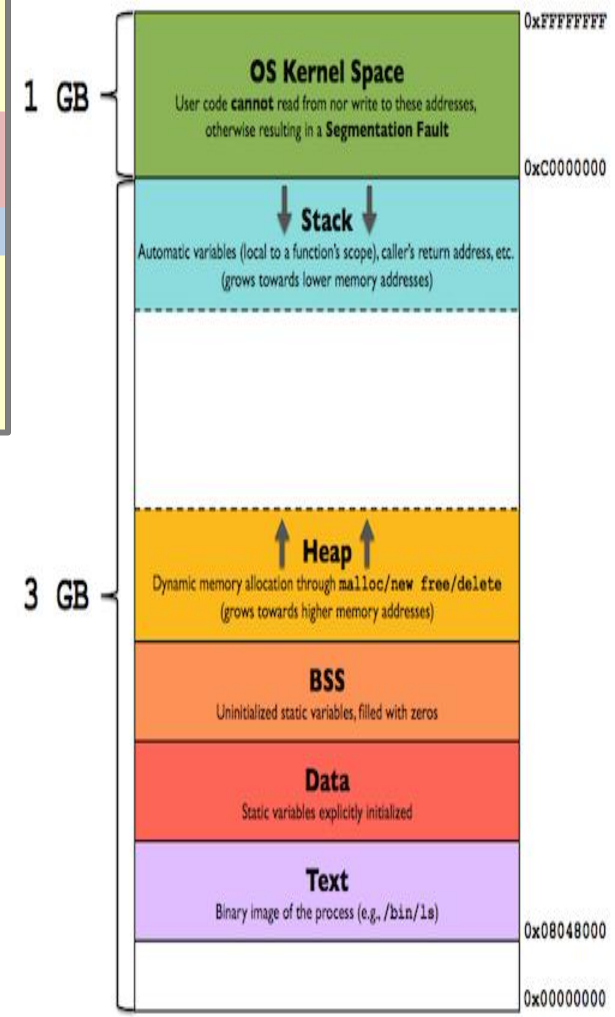
Context Switch



http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

Program code & constants: Text Segment

```
1   int main(void) {
2       char *string = "hello"; //string constant
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   }
```

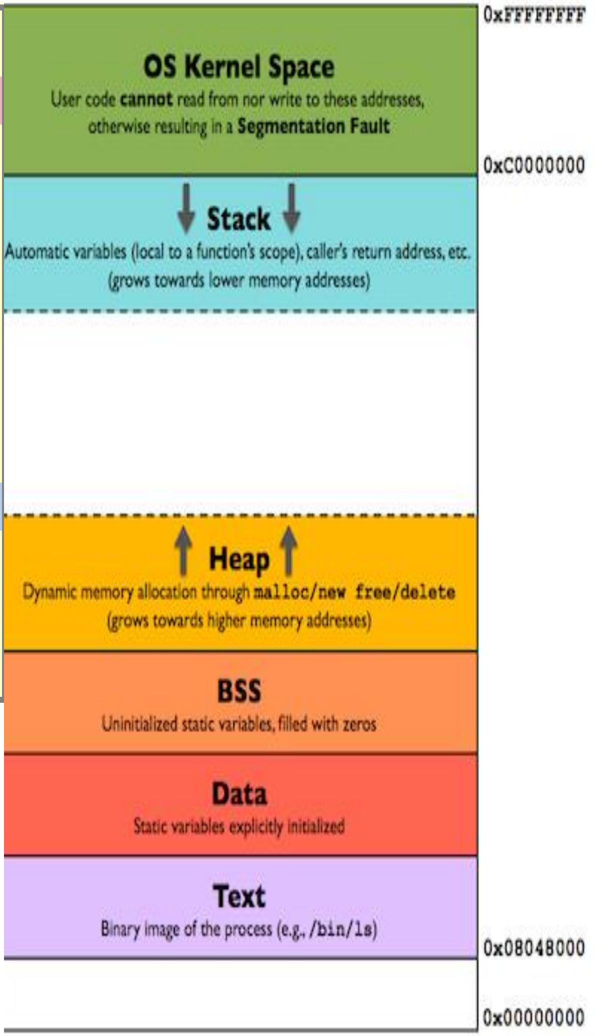


- **Lines 3&4: compiler is smart at**
  - saving space
- **Line 5: segmentation fault!**
  - Updating the constant!

```
[examples@3150] cat constant_ptr.c constant_array.c
```

Program code & constants: Text Segment

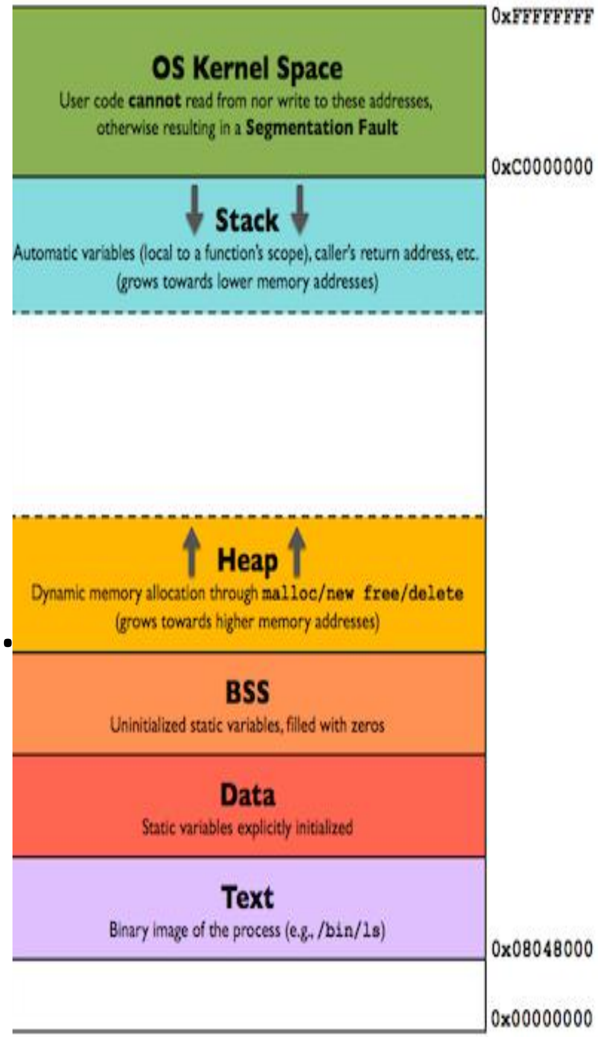
```c
int main(void) {
 void *ptr = main;  //ptr to any type
 unsigned char c = *((unsigned char *) ptr);
     //casting ptr to be a char pointer
     //deference it; so the content of c is
     //the first byte from ptr
 printf("Read  : 0x%x\n", c);
      //read the 1st byte content of the main code
 printf("Write : ");//modify the main at runtime
 *((unsigned char *) ptr) = 0xff;
 printf("done\n");
 return 0;
}
```




```
[examples@3150] cat access_code.c
```

Program code & constants

- Summary:
  - Codes and constants are both **read-only**.
    - You cannot change the values of the codes nor the constants during runtime.
  - Constants are stored in code segment.
    - Only store the **unique constants**.



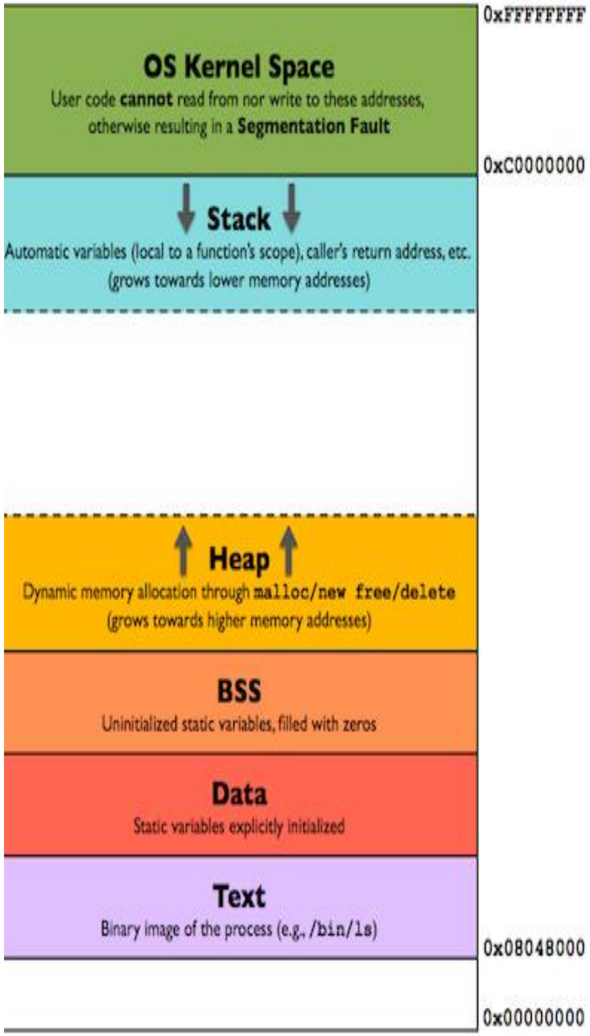
User-space memory management

- Addressing and Segment;
- Code & constants;
- **Data segment;**

Data Segment & BSS – properties

- For global and static variables
  - Data Segment
    - For **initialized**
  - BSS (**B**lock **S**tarted by **S**ymbol)
    - For **uninitialized**

"Better Save Space"



Global vs static: differ mostly in programming/compile time (scope, same variable name in different files, etc.)
https://stackoverflow.com/questions/7837190/c-c-global-vs-static-global

They become similar when in real-time (OS view)

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!**

```
[examples@3150] cat global_vs_static.c
```

Data Segment & BSS – locations

```c
int global_bss;
int global_data = 10;
int main(void) {
    static int static_bss;
    static int static_data = 10;
    printf("global  bss = %p\n", &global_bss  );
    printf("static  bss = %p\n", &static_bss  );
    printf("global data = %p\n", &global_data );
    printf("static data = %p\n", &static_data );
}
```

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

`[examples@3150] cat data_vs_bss.c`

Data Segment & BSS – sizes

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

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

Program: data_large.c

```
char a[555] = {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 ... 1007174 ... data_large
-rwxr-xr-x ...    7240 ... data_small
$_
```

a[] → global variable → initialized → data segment
→ since all things are known at compile time
→ the complier pre-allocates the space in the compiled code

```
[examples@3150] cat data_small.c data_large.c
```

Data Segment & BSS – sizes

```
char a[1000000];

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

Program: bss_large.c

```
char a[555];

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 ... 7130 ... bss_large
-rwxr-xr-x ... 7130 ... bss_small
$_
```

Same size!

a[] → global variable → uninitialized → BSS

Binary object code (at *compile-time*)

- Just keep the length value (e.g., "555") there in the BSS

a[555] (at *run-time*)

- will get instantiated to all 0's by exec()

```
[examples@3150] cat bss_small.c bss_large.c
```

Data Segment & BSS – sizes

The “**`size`**” program allows you to see the size of the TEXT segment, DATA segment, and the BSS.

```
$ size bss_small bss_large data_small data_large

   text     data      bss      dec      hex   filename
    836      260      132     1228      4cc   bss_small
    836      260  1000032  1001128    f46a8   bss_large
    836      384        8     1228      4cc   data_small
    836  1000284        8  1001128    f46a8   data_large

$
```

Just the size in different bases.

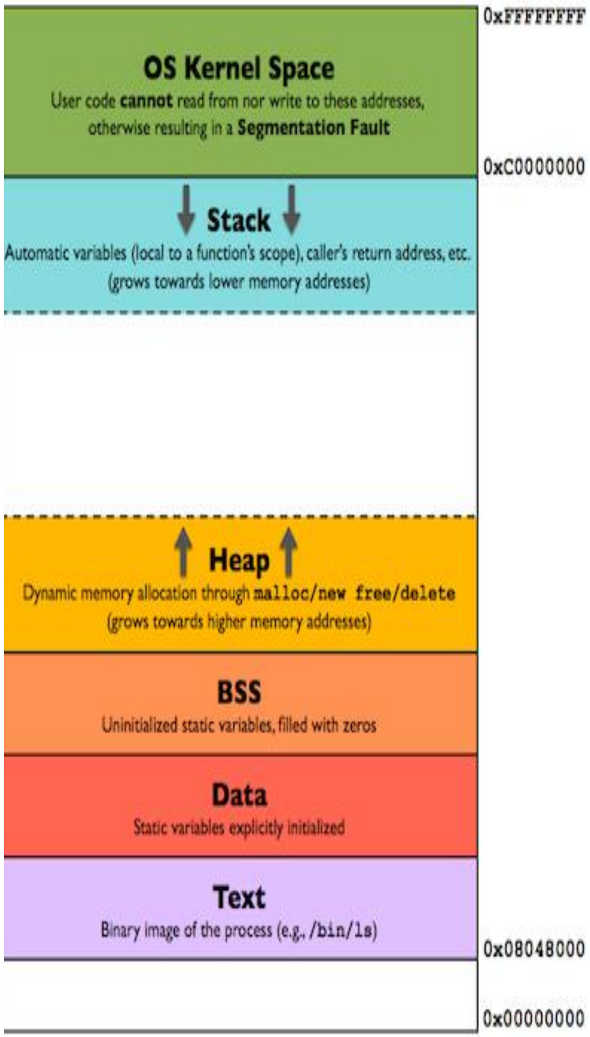
User-space memory management

- Addressing and Segment;
- Code & constants;
- Data segment;
- **Stack;**

Stack – properties

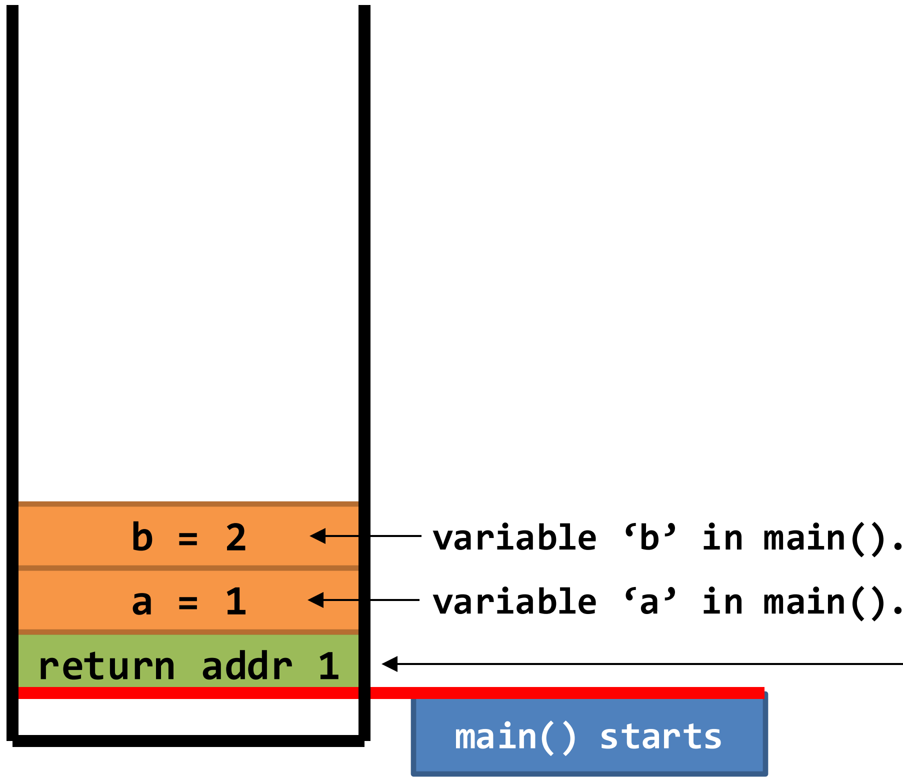
- The stack contains **per function**:
  - local variables,
  - function parameters, and
  - environment variables.

```
int main(int argc, char **argv, char **envp) {
    ......
}
```



Stack frame of 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;
}
```

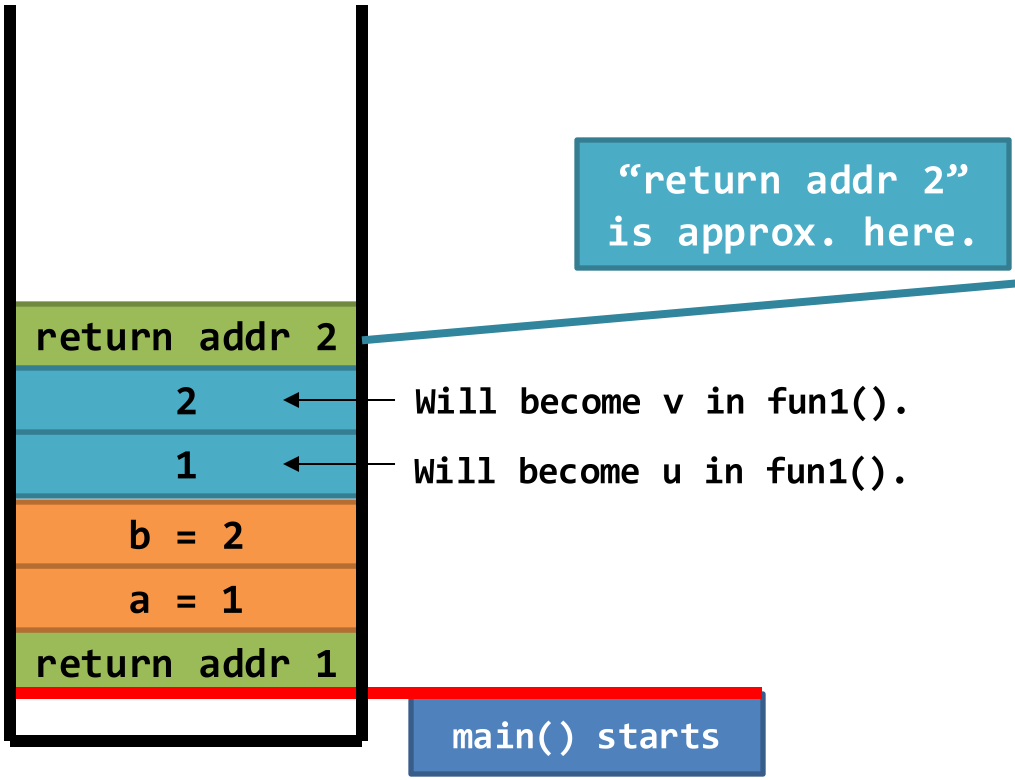


This address tells the program **where to return** (in the code segment) when main() returns.

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

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

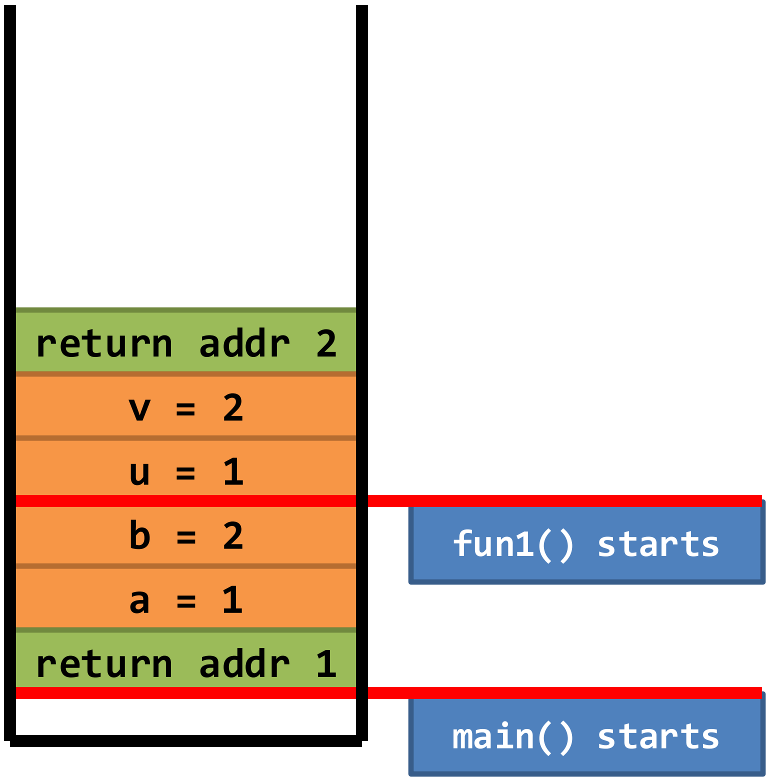



```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

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

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

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

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



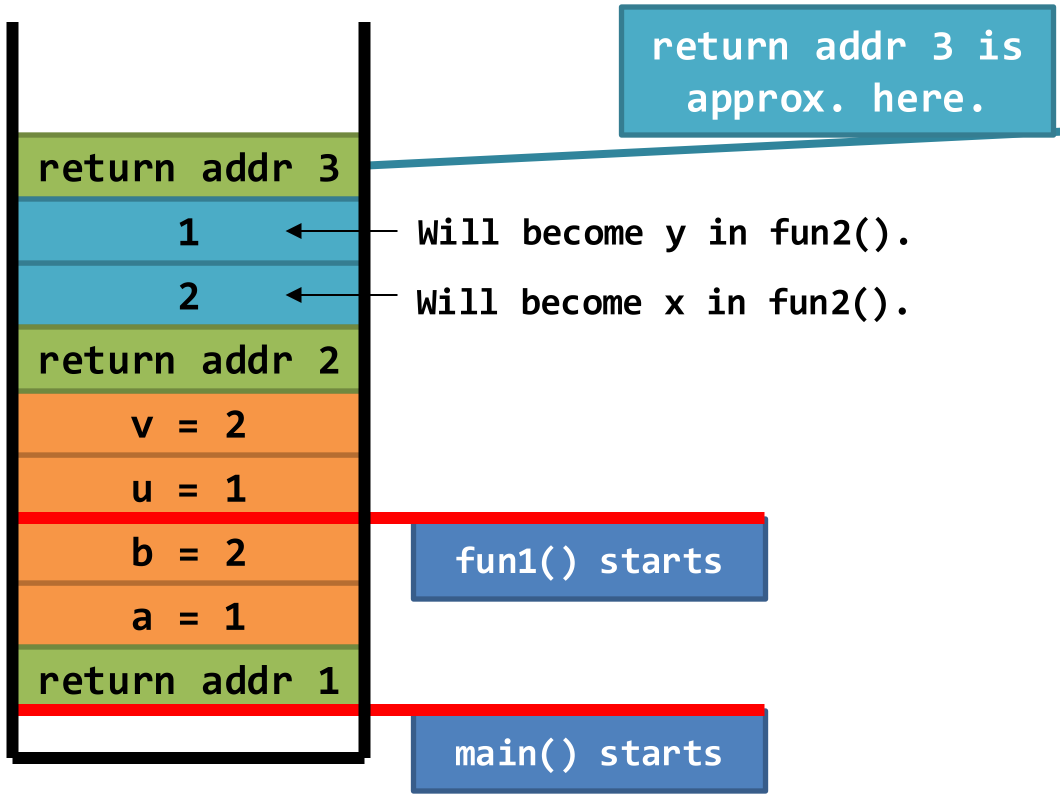

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

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

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

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

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

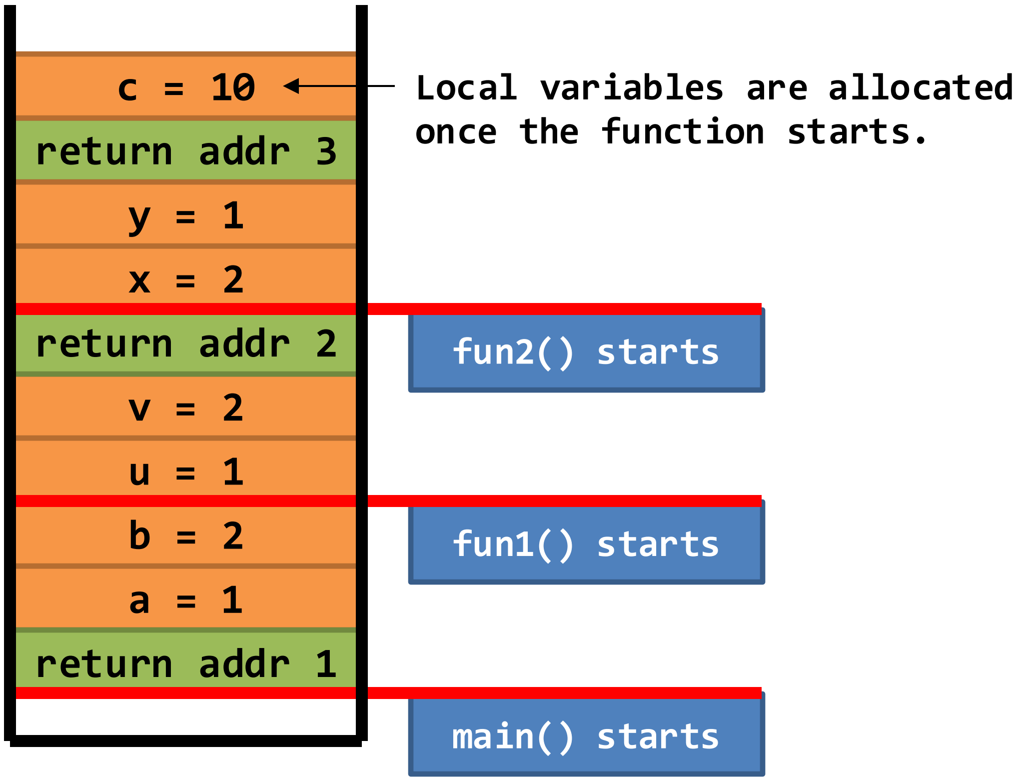
```c
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

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

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




```
[examples@3150] cat stack.c
```

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;
}
```

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

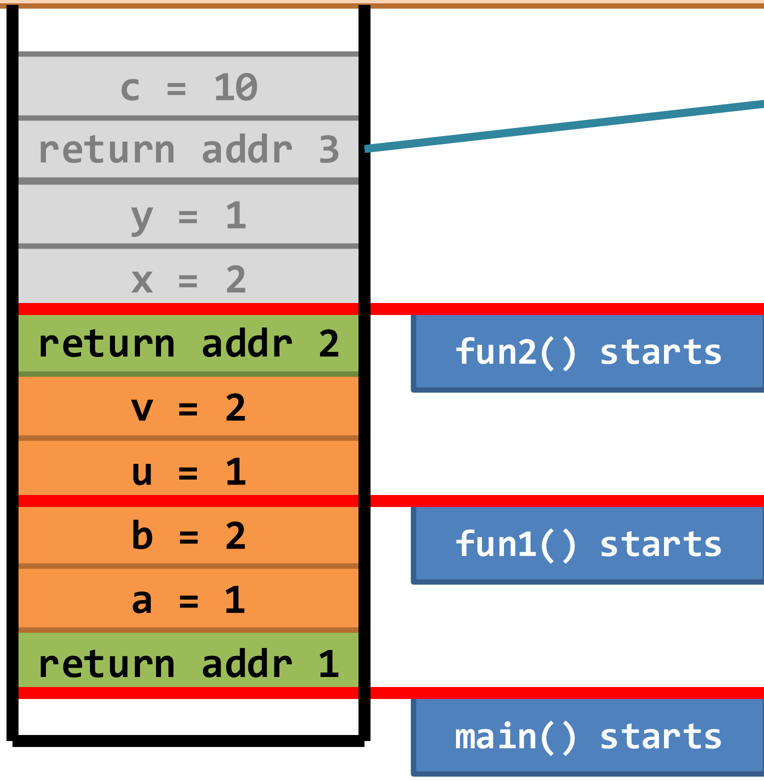
"Return" takes place.

(1) Return value is written to the EAX register.

(2) Stack **shrinks**.

(3) CPU jumps back to **`fun1()`**.

```c
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


```
[examples@3150] cat stack.c
```

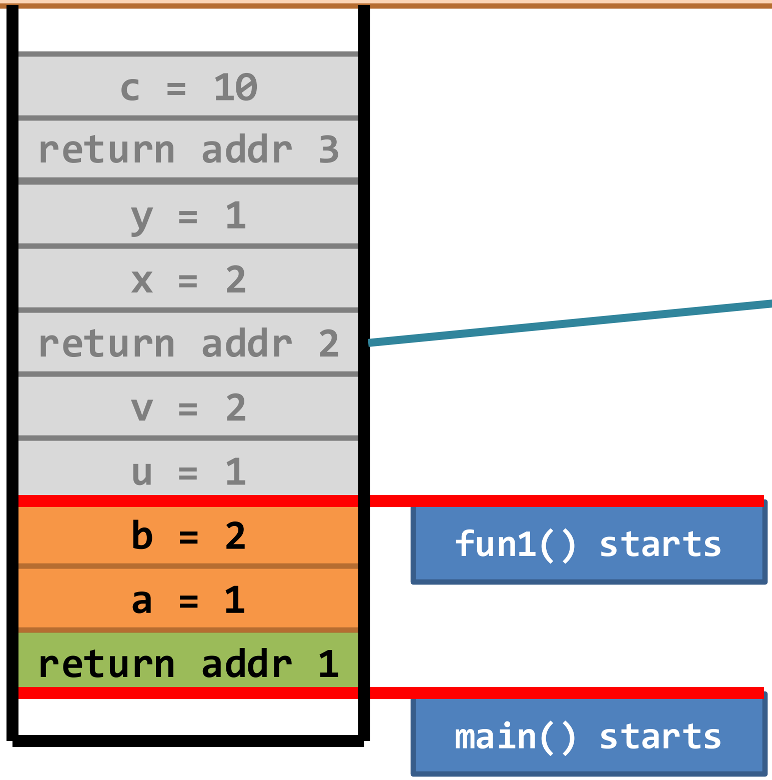
Stack – push & pop mechanisms

“Return” takes place.

(1) Return value is written to the EAX register.

(2) Stack **shrinks**.

(3) CPU jumps back to **`main()`**.

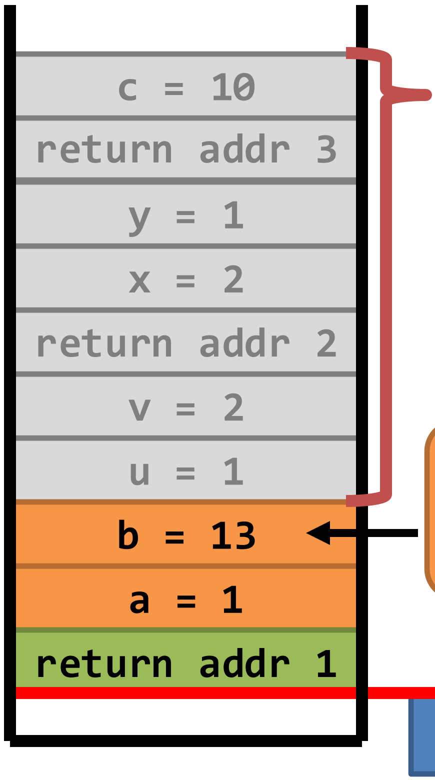



```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

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

[examples@3150] cat stack.c

Stack – push & pop mechanisms



<u>Very Important.</u>

Those memory **<u>is NOT returned to the OS</u>**!!

Those memory will be re-used in case you call the functions again.

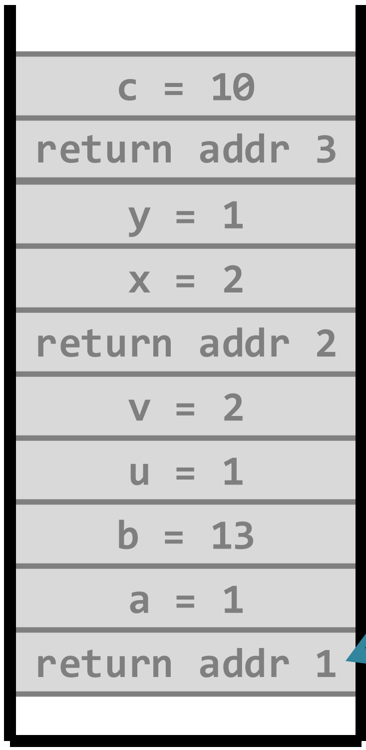
Upon "return", the value of EAX is then copied to "b"

```
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


```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms



Eventually, the main function reaches "**`return 0`**".

This takes the CPU returning to C library (e.g., exec()).

Inside the exec(), it will eventually call system call **`exit()`** for you.

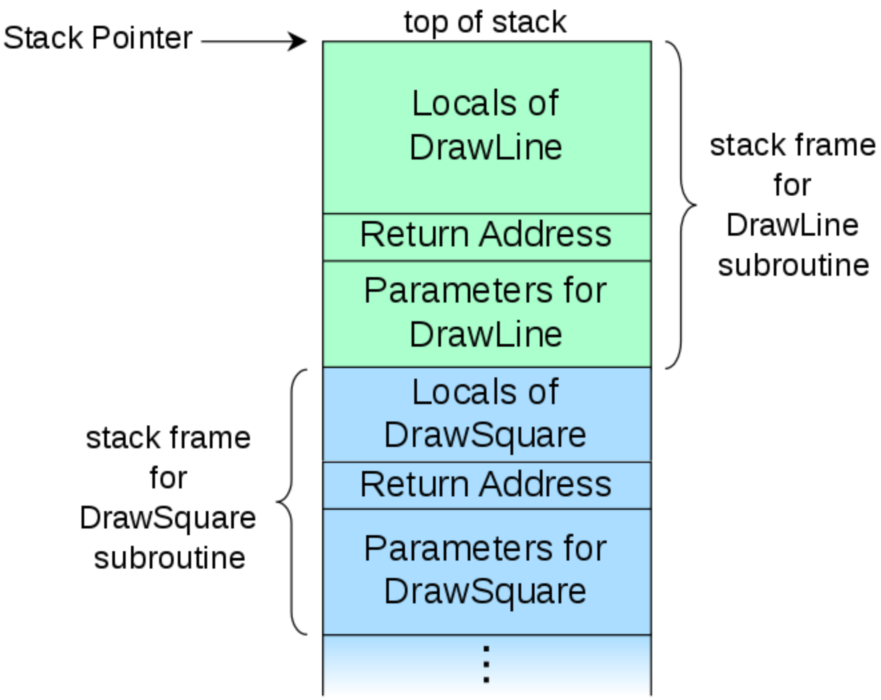
```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

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


```
[examples@3150] cat stack.c
```

Stack in-depth

- When a function is called
  - Push a **stack frame**.
- When a function returns
  - Pop the stack frame.
    - Set stackptr = *frameptr
    - *frameptr stores the previous stack ptr
- The compiler **hardcodes this mechanism** into your program
  - This is not done by the kernel*



*passing function parameters via registers vs. stack

Stack size

- The compiler cannot estimate the stack's size in compile time.
  - Because the number of active function calls depends on the program state, the user inputs, etc.
  - The kernel can only reserve certain space for the stack.
- Implication 1:
  - Stack overflow
- Implication 2:
  - A function has permission to **read and write anywhere in the stack**,
  - **not restricted to its own stack frame**

Stack overflow – limits

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

So, the limit is:
8192KB = 8MB.

```
int main(void) {
    char a[9000 * 1024]; //a local var
    memset(a, 0, sizeof(a));
    printf("OK!\n");
    return 0;
}
```

Can you see "OK"?

```
[examples@3150] cat max_stack.c
```

Stack overflow – limits

```
$ 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 x 1024 = 80MB.

```c
int main(void) {
    char a[9000 * 1024]; //a local var
    memset(a, 0, sizeof(a));
    printf("OK!\n");
    return 0;
}
```

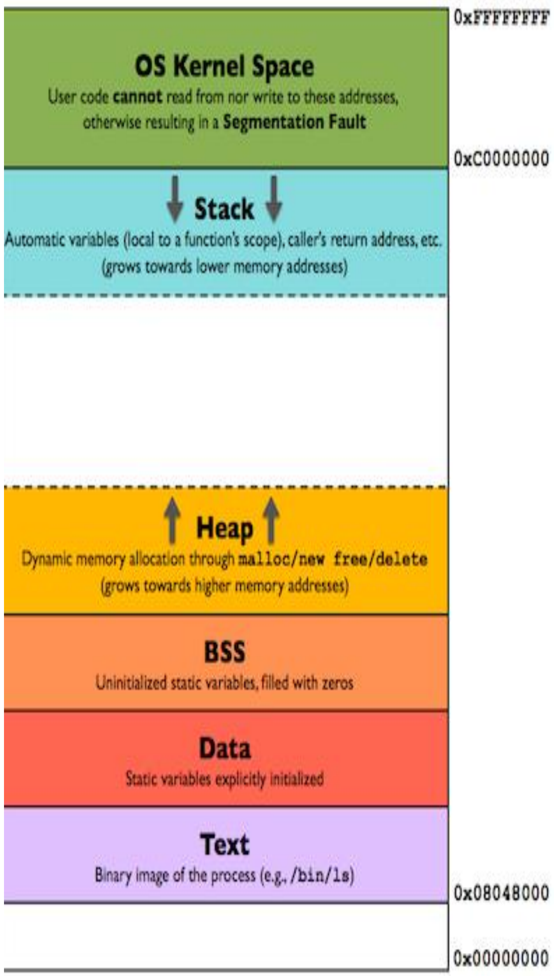
Can you see "OK"?

```
[examples@3150] cat max_stack.c
```

Stack overflow

"I really need to play with recursions." Any workaround?

- Minimizing the number of function arguments.
- Minimizing the number of local variables.
- Minimizing the number of calls (if you can).
- Use global variables!
- Use malloc instead
- **<u>Or: tail recursion</u>**



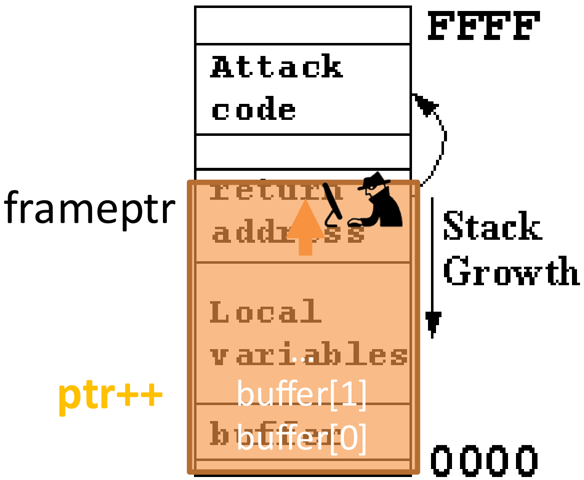
Anything wrong about this code?

```
#include <stdio.h>
#define BUFFER SIZE 256
int main(int argc, char *argv[])
{
   char buffer[BUFFER SIZE];
   if (argc < 2)
       return -1;
   else {
       strcpy(buffer,argv[1]);
       return 0;
   }
}
```

Security leak!

- We can feed an input that goes beyond the buffer!

Stack smashing attack by buffer overflow



exec is syscall

But libc has wrapped it to become different exec* family members

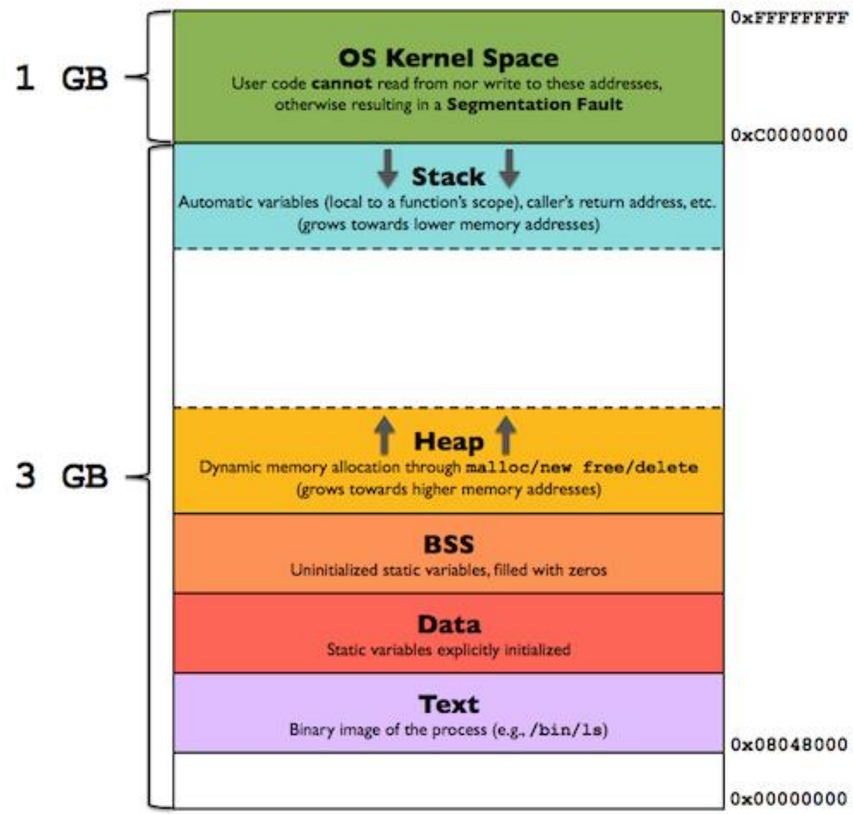
- Input a string that overflows the buffer so that the **return address** is overwritten
- "Attack code" can be as short as "`execvp("/bin/sh")`"
  - That short piece of attack code can be placed elsewhere in the memory by you overflowing **another** buffer in the same program
  - `execvp` should be in the same address space because `libc` is the default library executing with every program
- If you attack a program which was launched by root (e.g., `netd`)
  - Then `netd` suddenly `exec("/bin/sh")`
  - Then you get a shell with root privilege

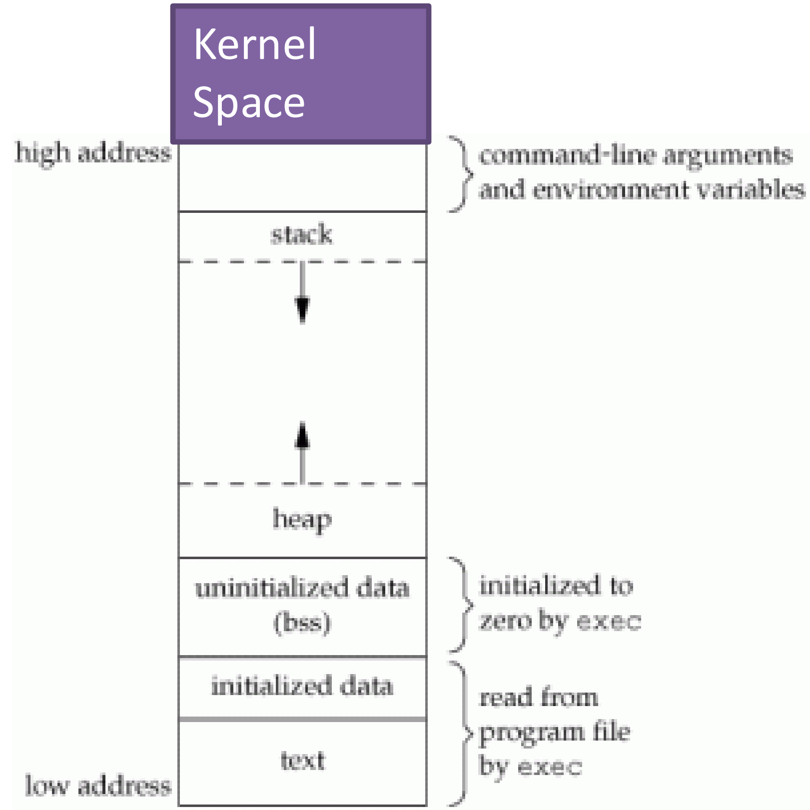
User-space memory management

- Addressing and Segment;
- Code & constants;
- Data segment;
- Stack;
- **Heap;**

Heap

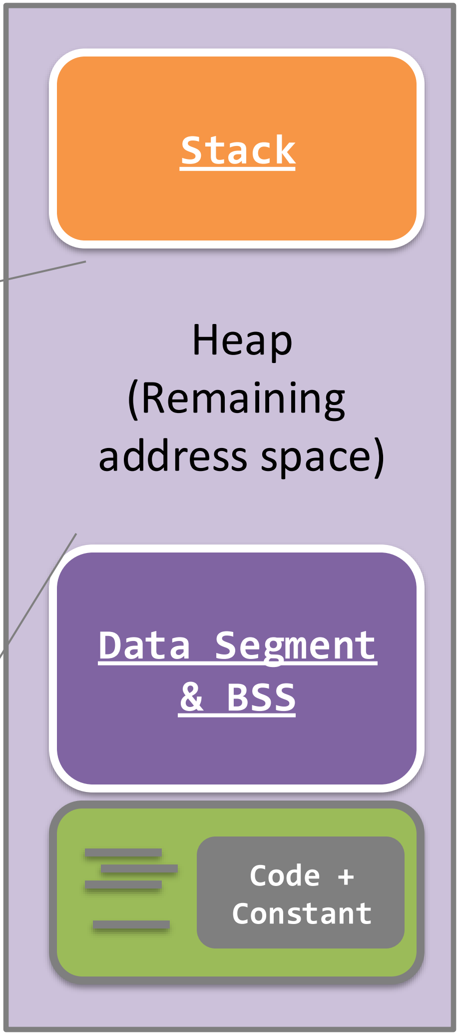
English Dictionary
Heap: an untidy collection of objects





De-mystifying `malloc()`

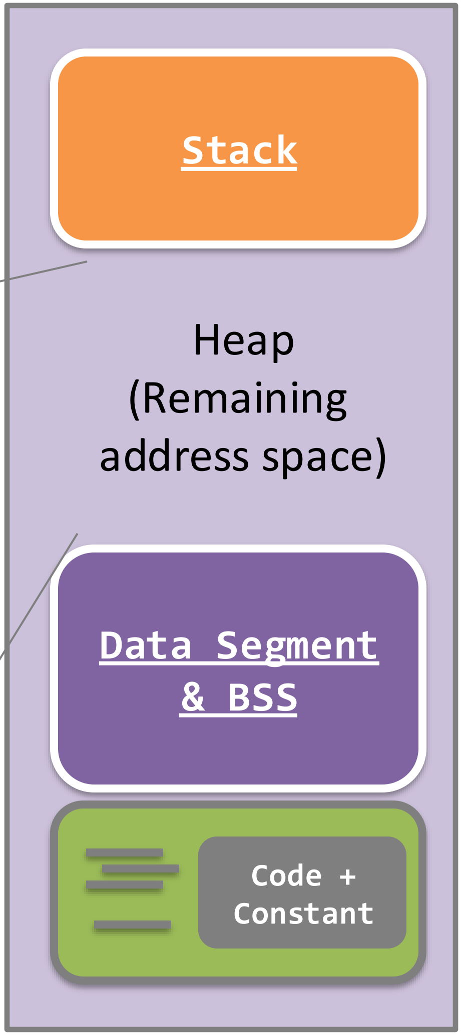
When a program just starts running, the entire heap space is unallocated, or empty.



De-mystifying `malloc()`

When "**`malloc()`**" is called, it may call the "**`brk()`**" system call

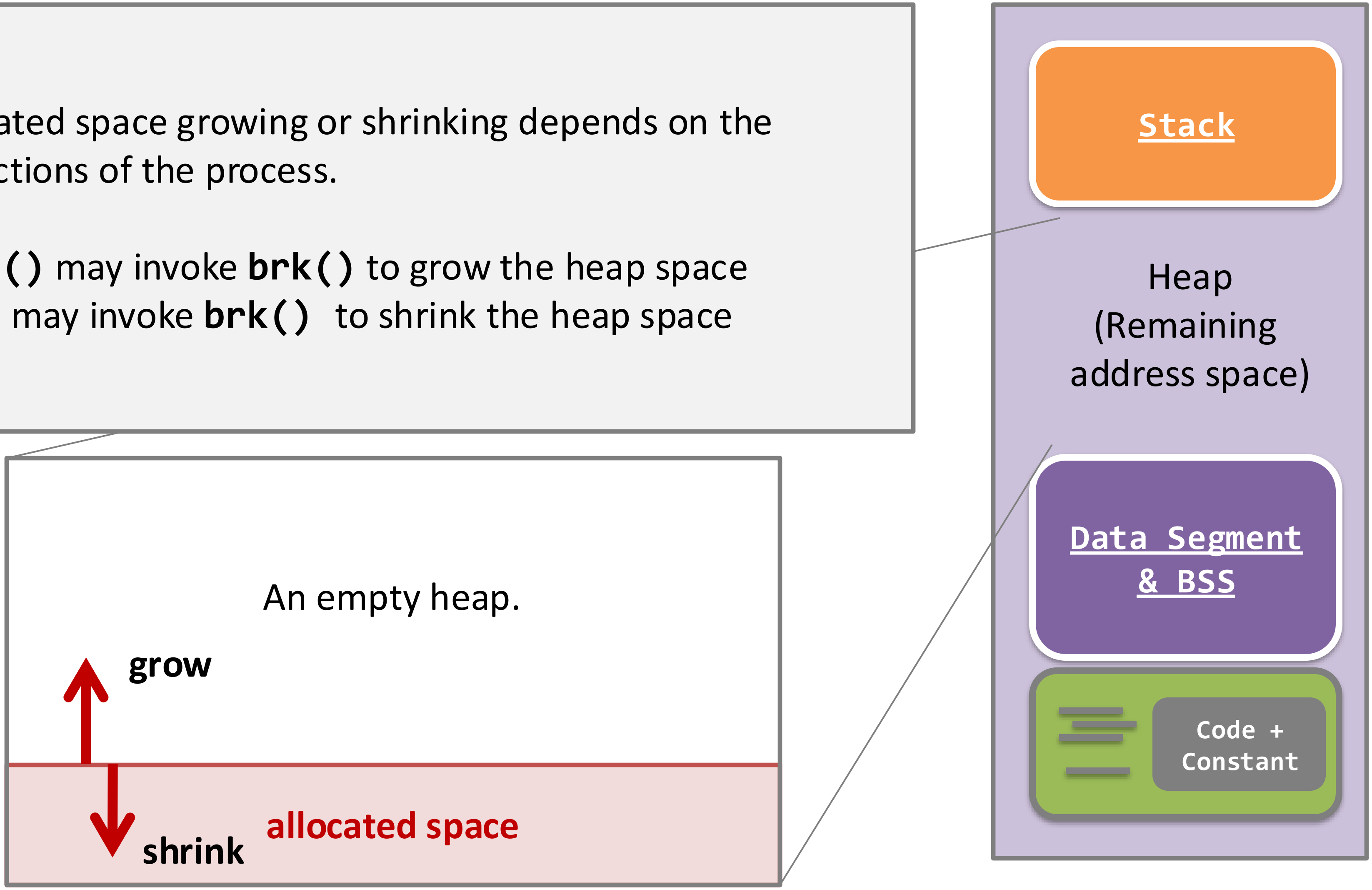
The "**`brk()`**" syscall sets the ending address of the heap



De-mystifying `malloc()`

The allocated space growing or shrinking depends on the further actions of the process.

- **`malloc()`** may invoke **`brk()`** to grow the heap space
- **`free()`** may invoke **`brk()`** to shrink the heap space



De-mystifying `malloc()`

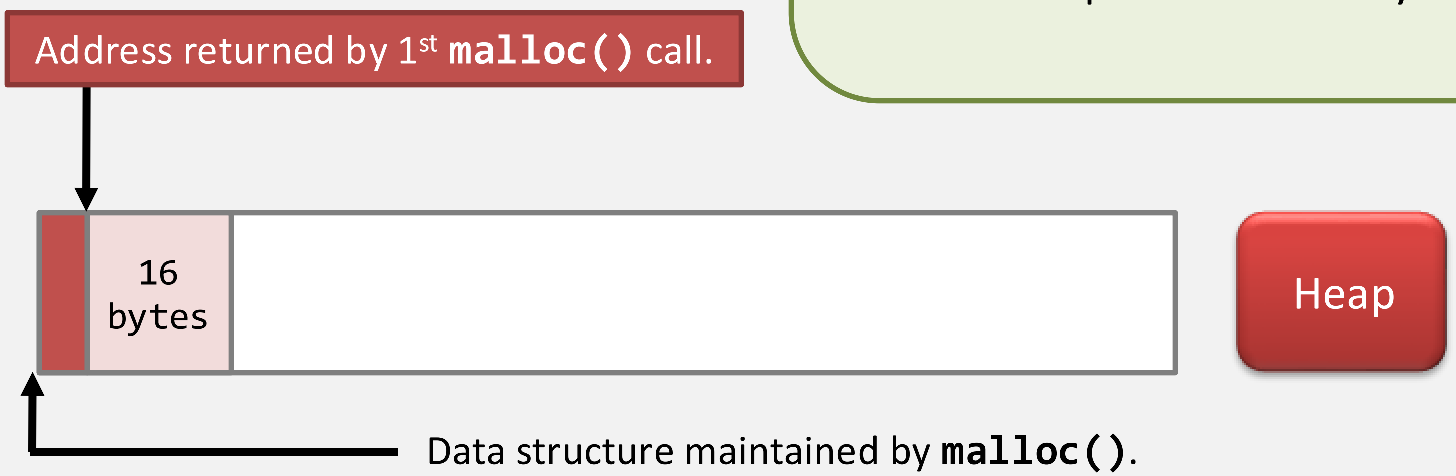
```
int main(void) {
  char *ptr1, *ptr2;
  ptr1 = malloc(16);
  ptr2 = malloc(16);

  printf("Distance between ptr1 and ptr2: %d bytes\n",
         ptr2 - ptr1);
  return 0;
}
```

De-mystifying **malloc()**

```
int main(void) {
  char *ptr1, *ptr2;
  ptr1 = malloc(16);
  ptr2 = malloc(16);

  printf("Distance between ptr1 an
         ptr2 - ptr1);
  return 0;
}
```

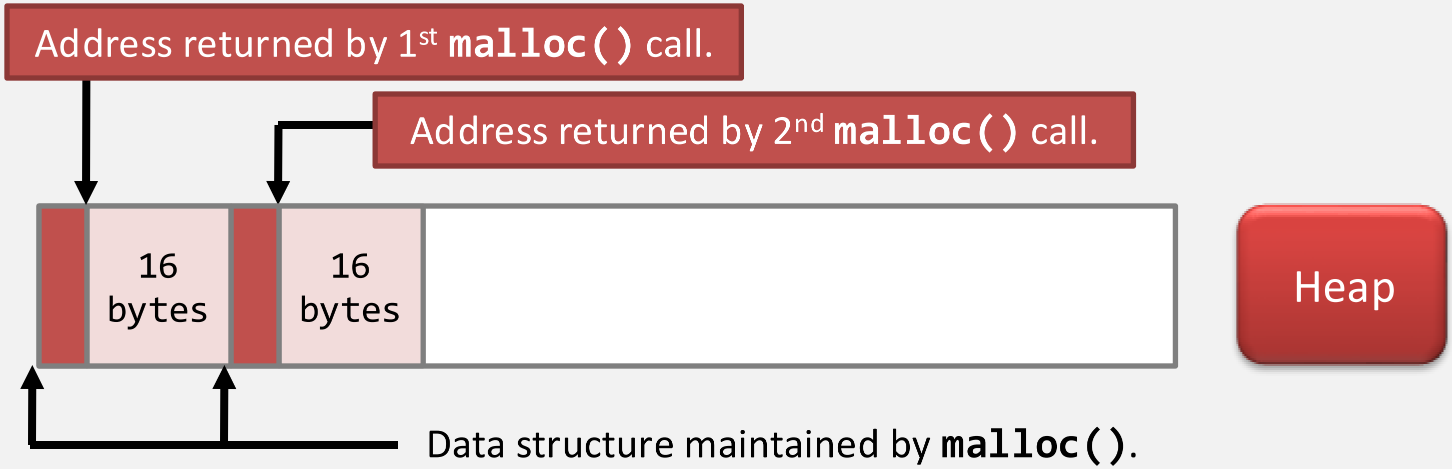
The return value of **malloc()** is of type "**void ***", which means it is just a memory address only, and can be of any data types.

Such a memory address is the starting address of a piece of memory.



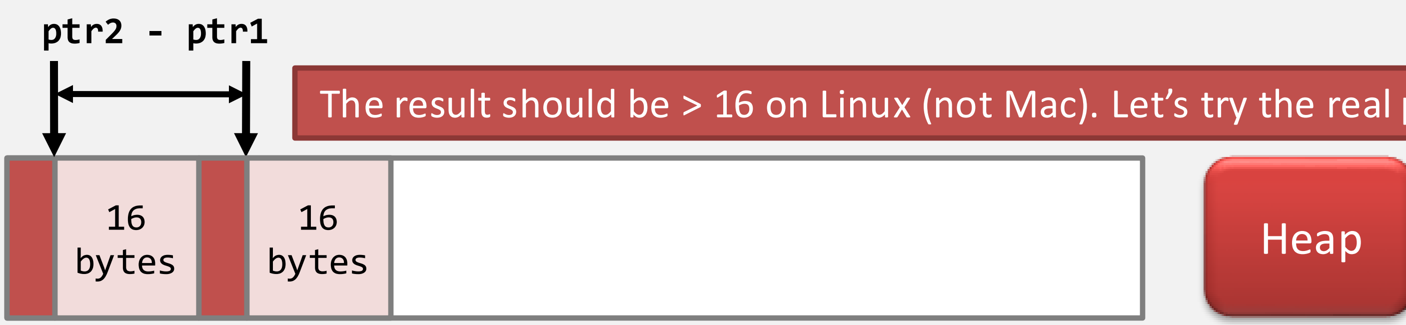
De-mystifying `malloc()`

```
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;
}
```



De-mystifying `malloc()`

```c
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;
}
```

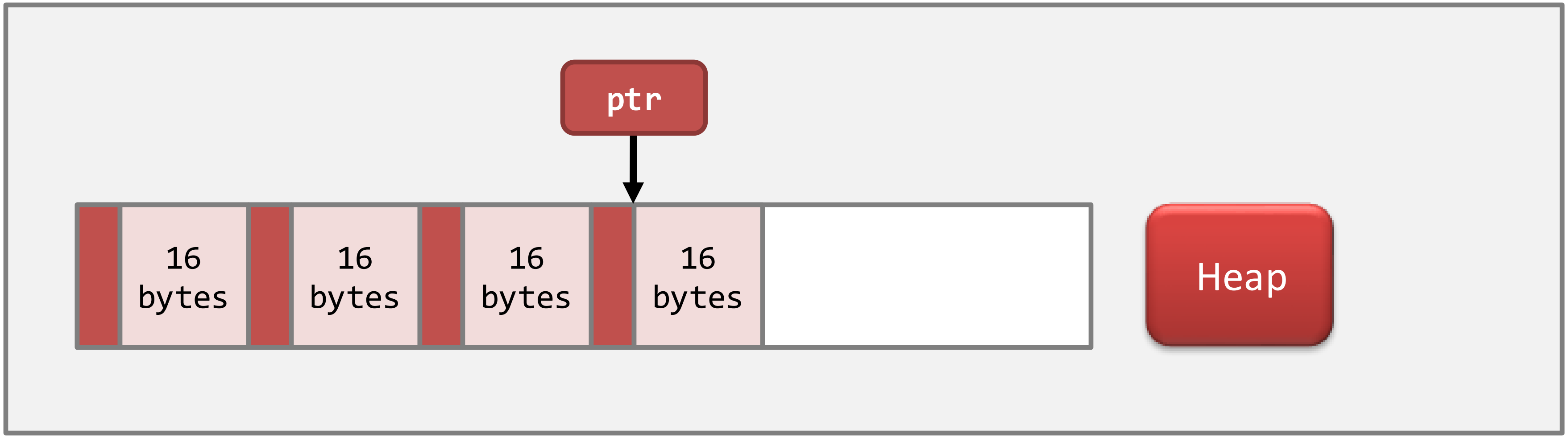


The result should be > 16 on Linux (not Mac). Let’s try the real program!

```
[examples@3150] cat malloc_distance.c # 32-bit & 64-bit machines yield different results
```

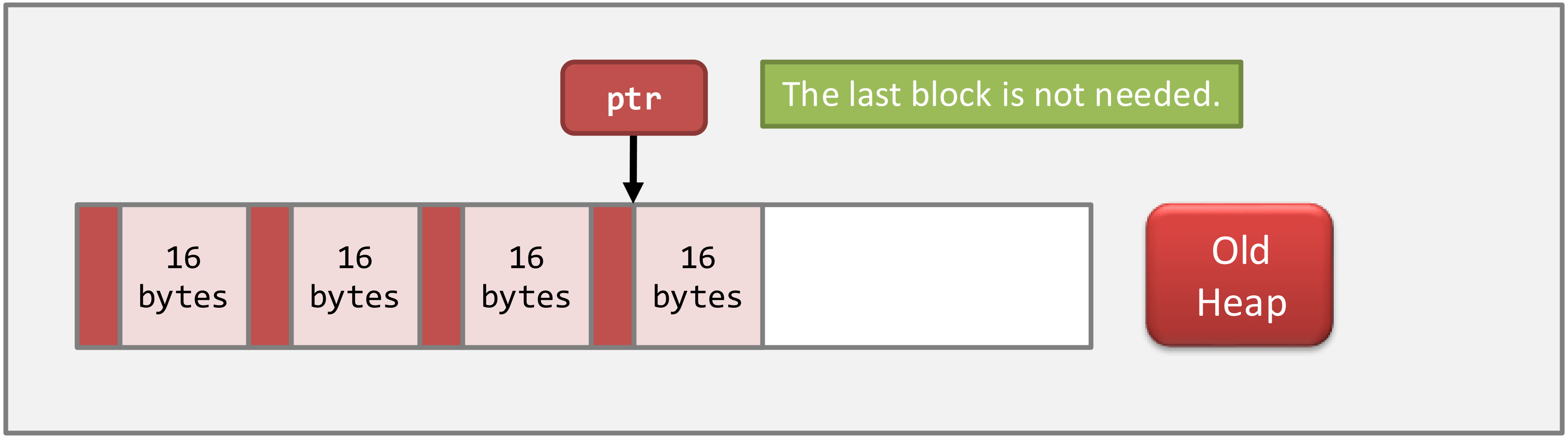
De-mystifying `free()`

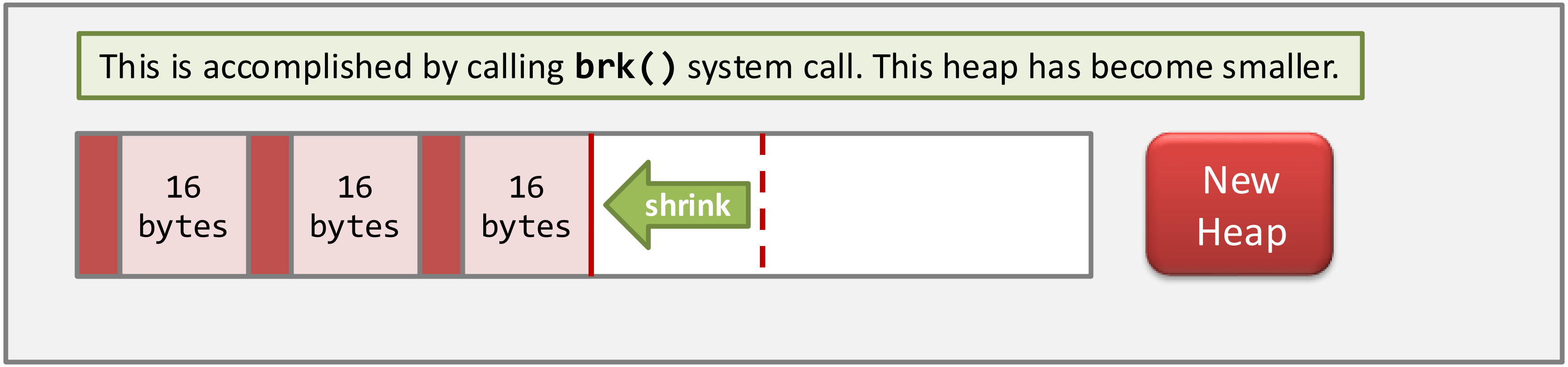
- "**`free()`**" *seems to* be the opposite to "**`malloc()`**":
  - It de-allocates any allocated memory.
  - When a program calls "**`free(ptr)`**", then the address "**`ptr`**" must be the start of a piece of memory obtained by a previous "**malloc()**" call.



De-mystifying **free()** – case #1

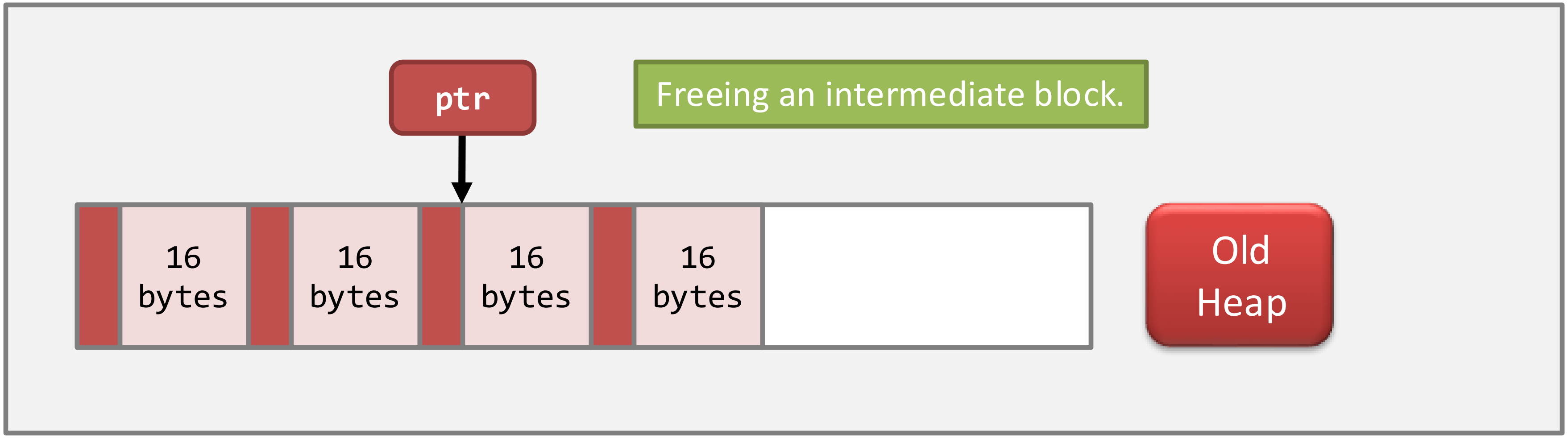
- Case #1: de-allocating the last block.

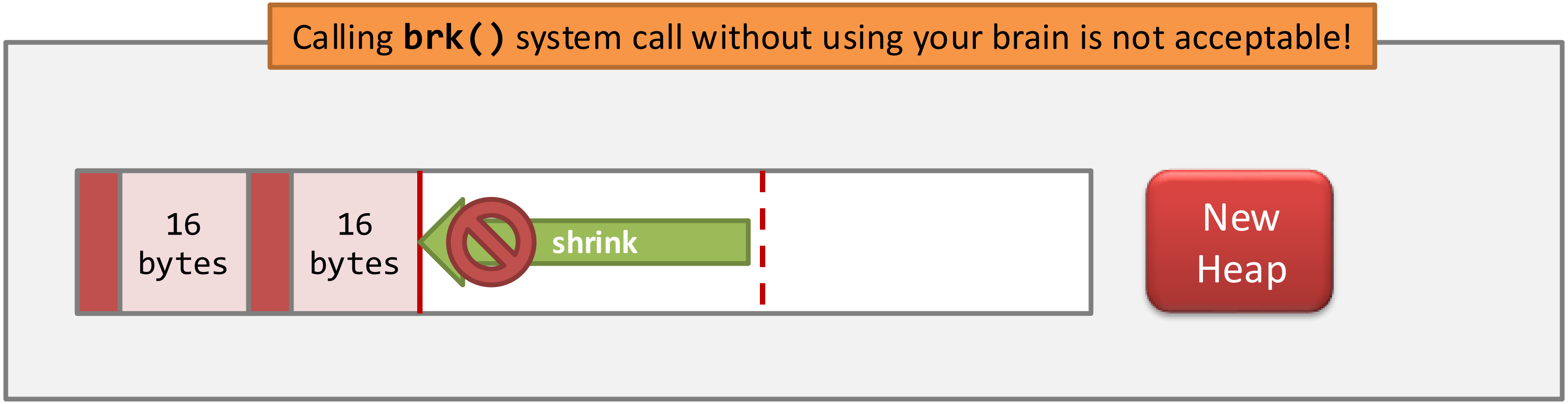




De-mystifying **free()** – case #2

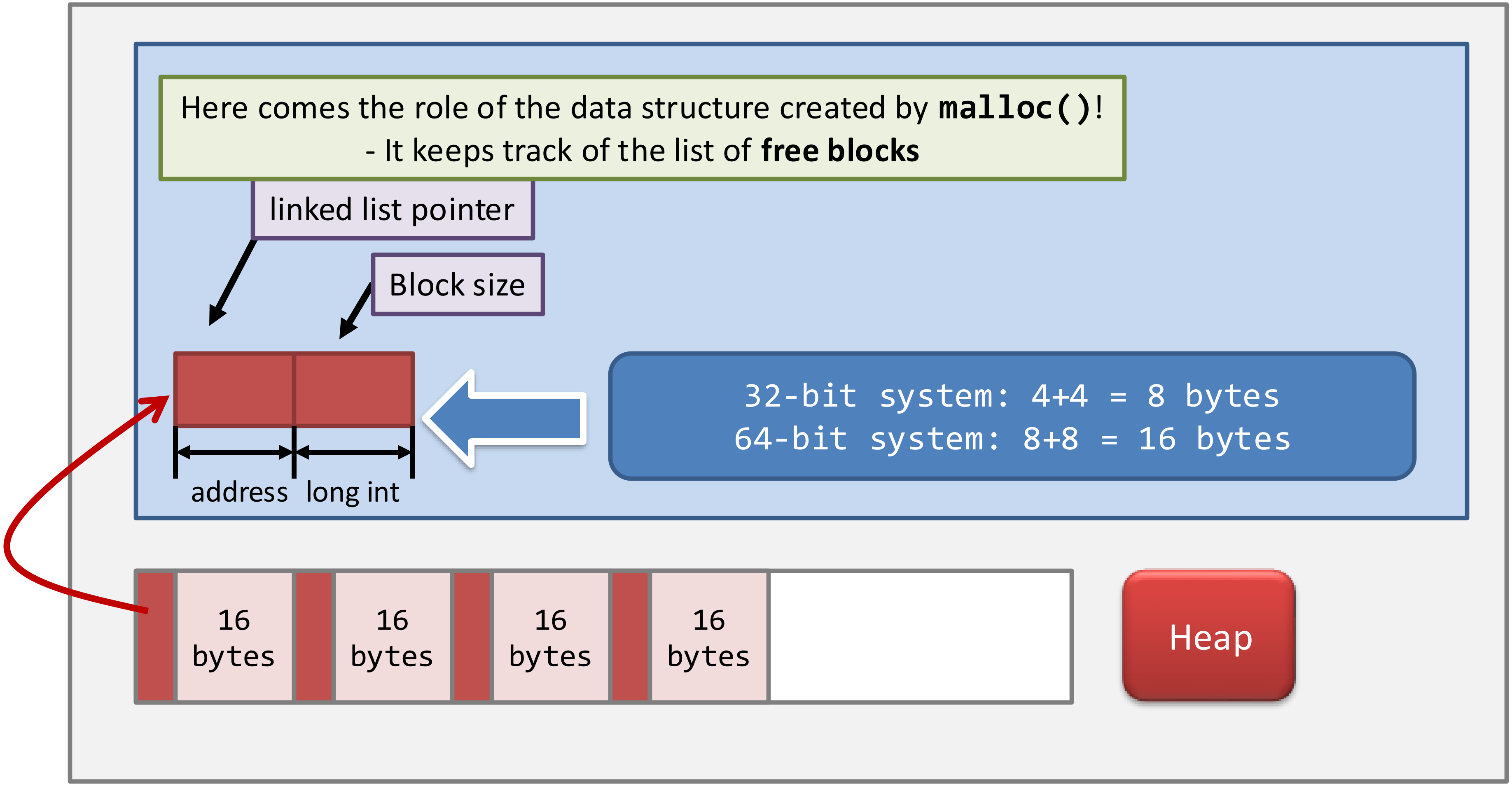
- Case #2: de-allocating an intermediate block.





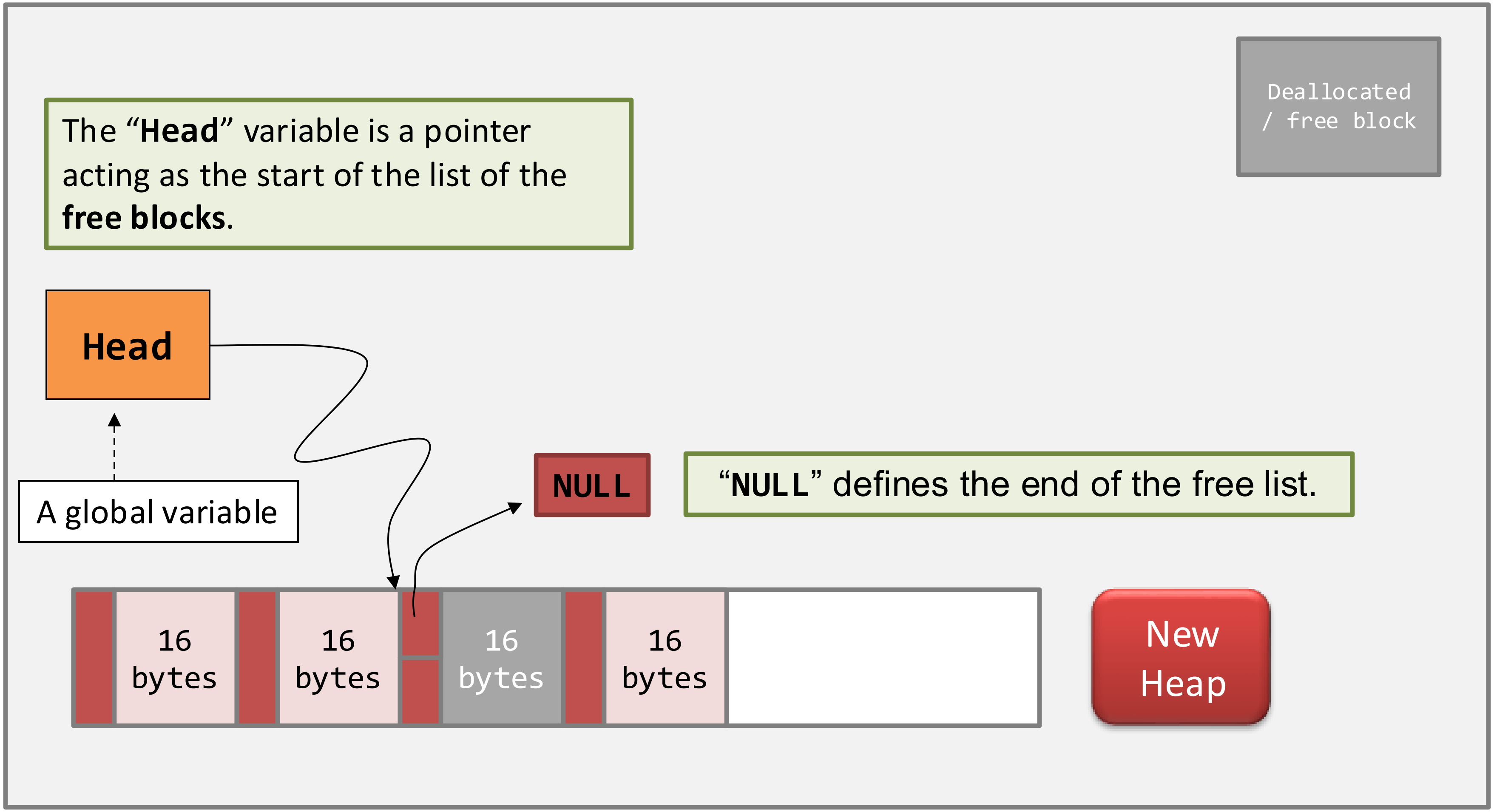
De-mystifying **`free()`** – case #2

- Case #2: free space management

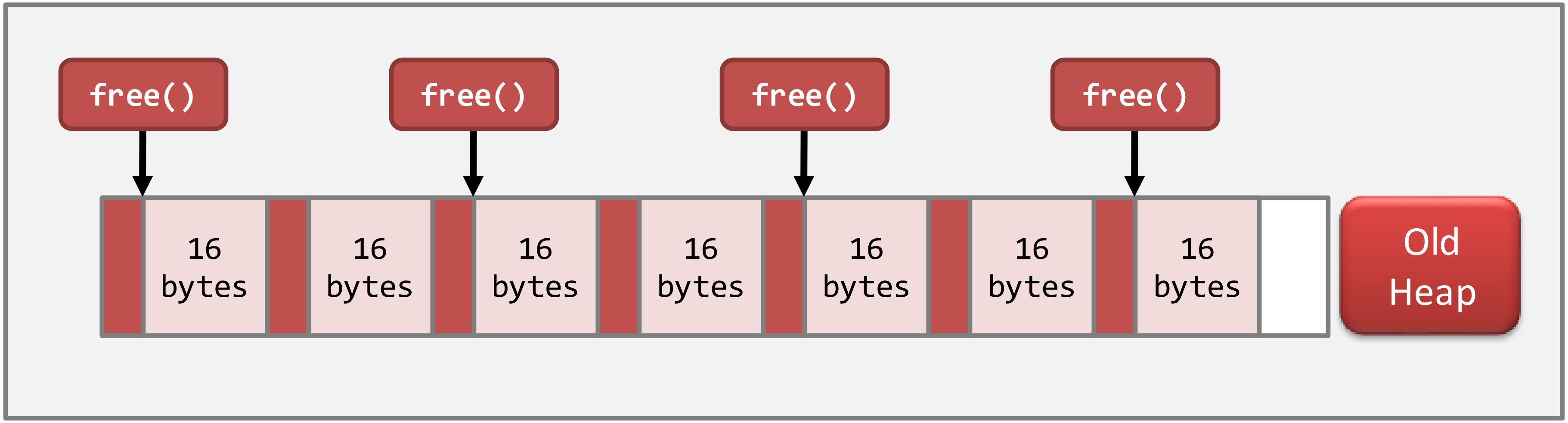


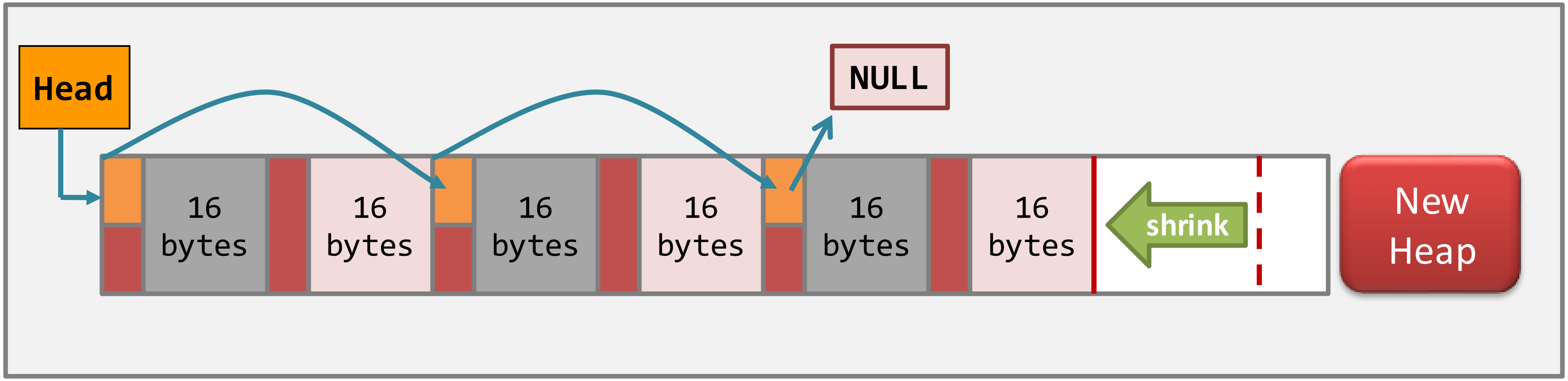
De-mystifying `free()` – case #2

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



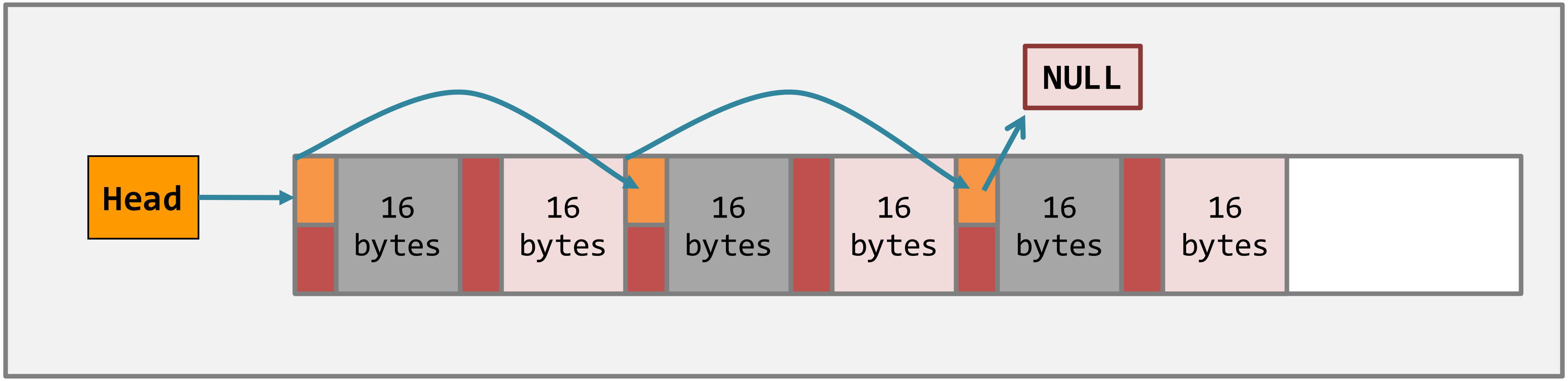
De-mystifying **free()**





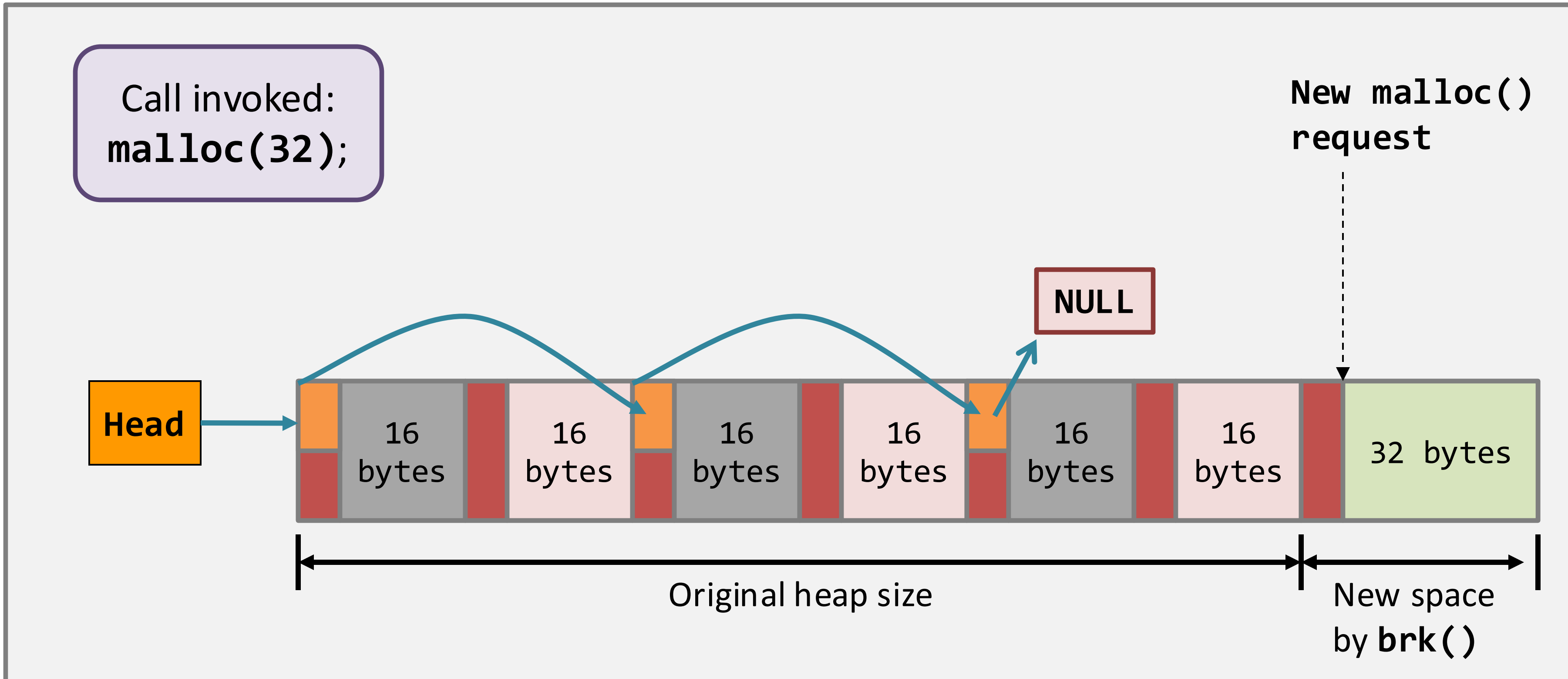
When `malloc()` meets free blocks...

- 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?*



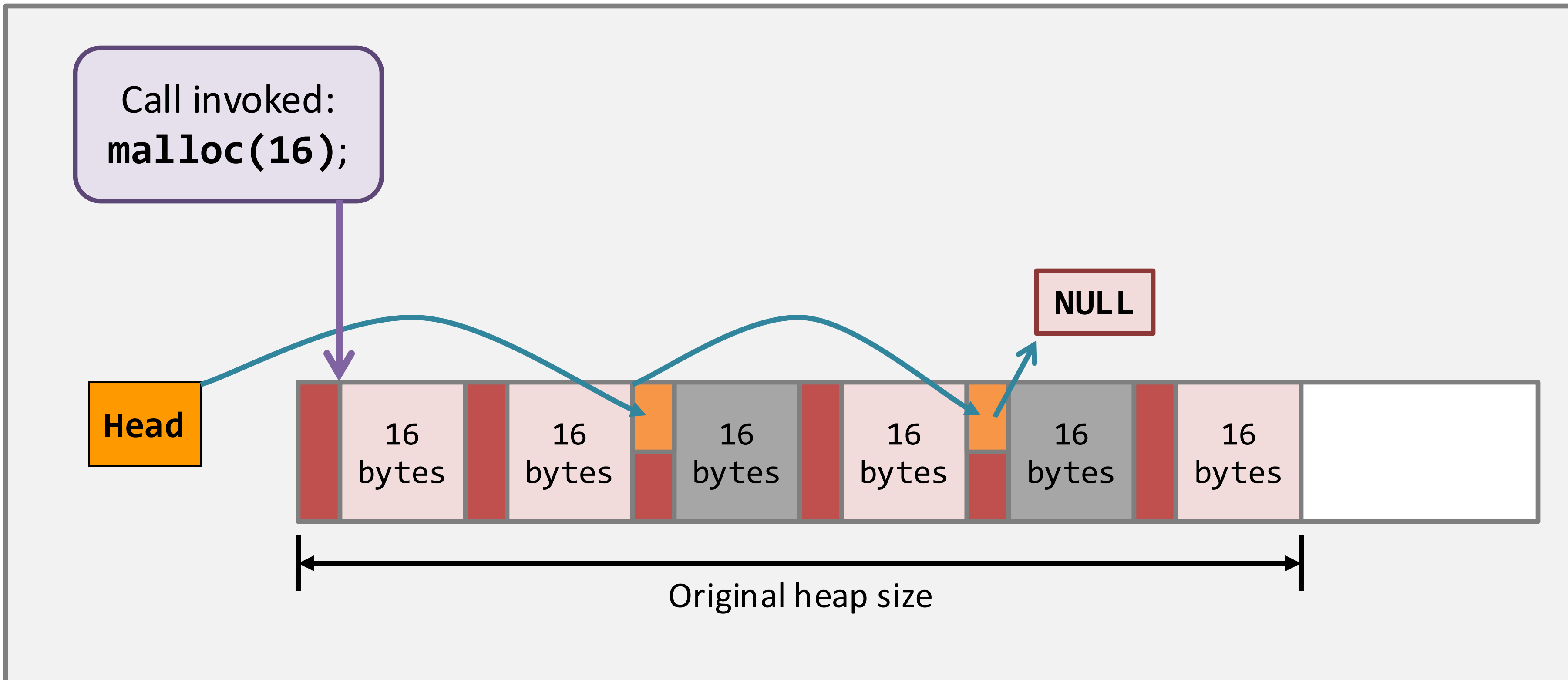
When **`malloc()`** meets free blocks…case 1

- 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.



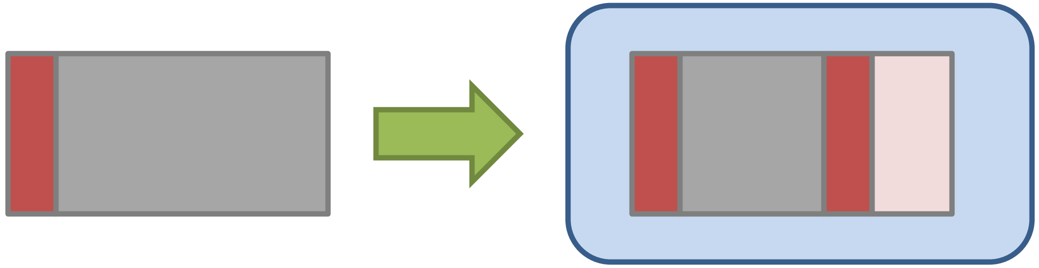
When `malloc()` meets free blocks…case 2

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



When **`malloc()`** meets free blocks…

- There can be other cases:
  - E.g., A **`malloc()`** request that takes a partial block;



Check out: https://en.wikipedia.org/wiki/C_dynamic_memory_allocation#Implementations

Out of memory?

- Try this! An OOM Generator!

```
#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;
}
```

```
[examples@3150] cat oom_v1.c
```

Out of memory?

- On 32-bit Linux, why does the OOM generator always stop at around 3055MB?
  - The kernel reserves 1GB address 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;
}
```

Out of memory earlier

<u>malloc: Cannot allocate memory</u>

```
[examples@3150] cat oom_v2.c
```

Out of memory?

- oom_v1 and oom_v2 just make you run out of memory **<u>addresses</u>**
- They have not consumed your memory yet

A Real OOM!

```
#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;
}
```

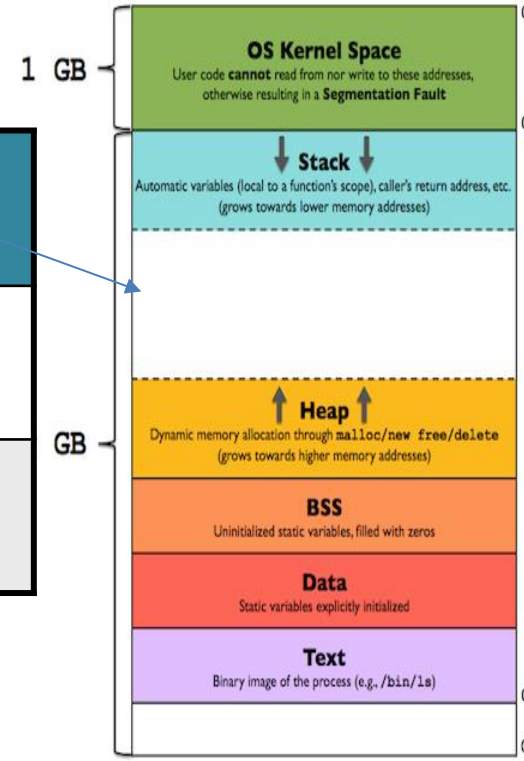
http://stackoverflow.com/questions/10575544/difference-between-declaration-and-malloc

User-space memory management

- Addressing and Segment;
- Code & constants;
- Data segment;
- Stack;
- Heap;
- **Segmentation fault;**

Segmentation fault

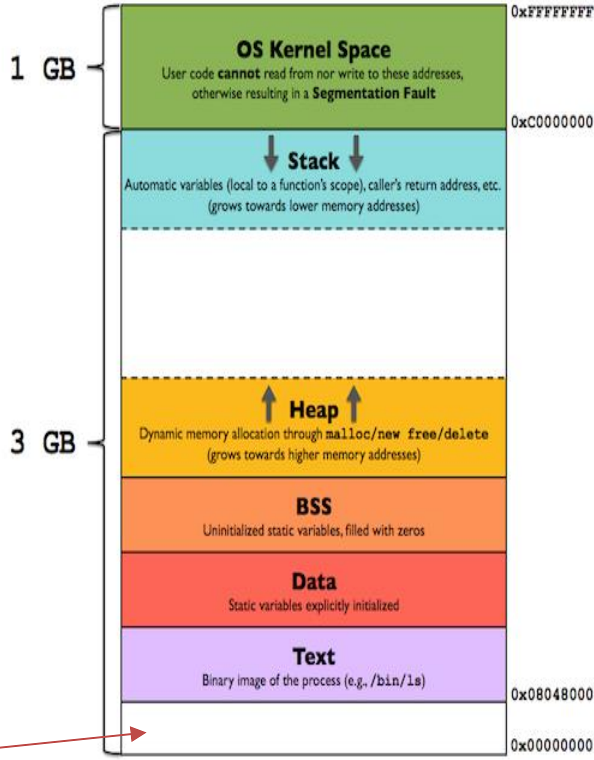
	Read-only segments	Allocated segments	Unallocated segments
Reading	No problem	No problem	**Segmentation fault**
Writing	**Segmentation fault**	No problem	**Segmentation fault**



https://stackoverflow.com/questions/7187981/whats-the-memory-before-0x08048000-used-for-in-32-bit-machine

Going down to the unusable segment

```
char *ptr = NULL;

void handler(int sig) {
    printf("TEXT segment starts at %p\n", ptr);
    exit(0);
}

int main(void) {
    char c;
    signal(SIGSEGV, handler);
    ptr = (char *) main;
    while(1) {
        ptr--;
        c = *ptr;    /* wanna generate SIGSEGV */
    }
}
```

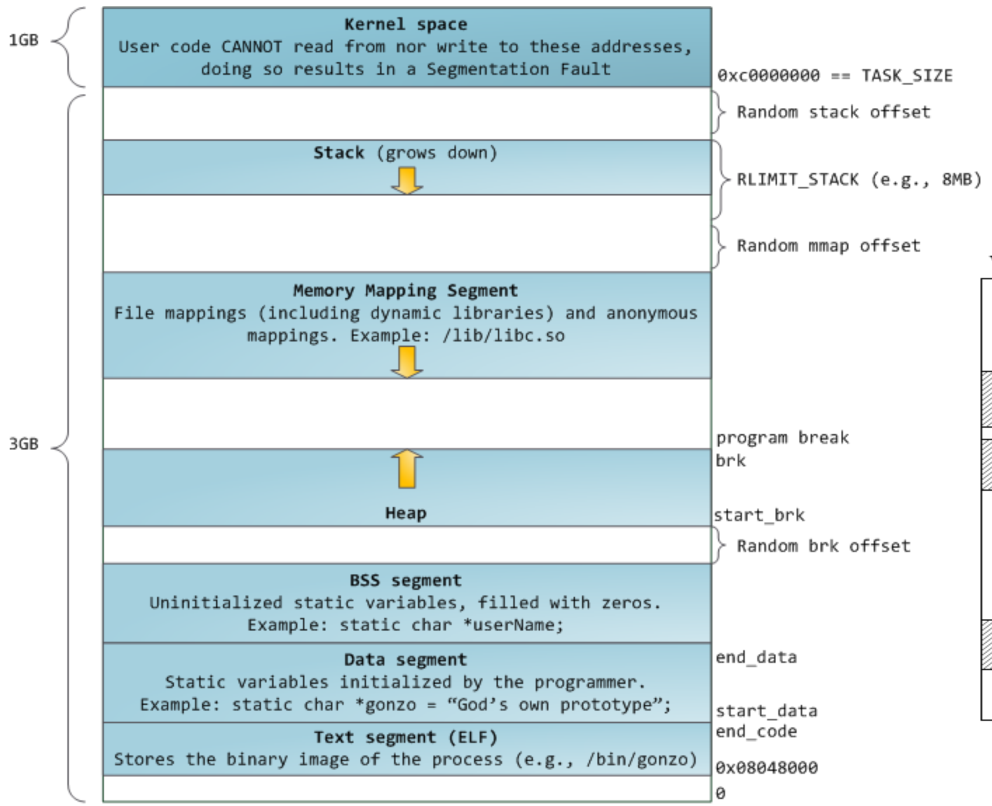


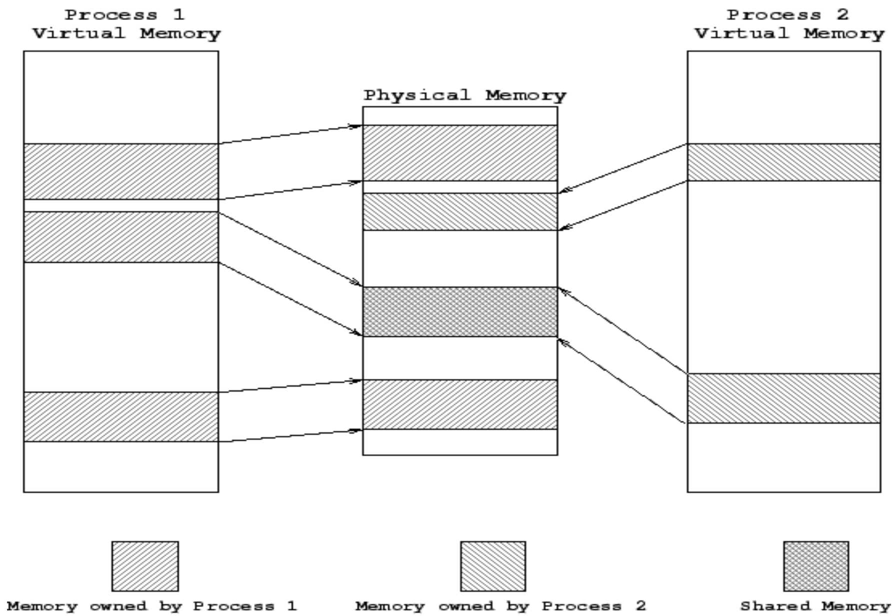

```
[examples@3150] cat code_start.c
```

https://www.quora.com/On-Linux-why-does-the-text-segment-start-at-0x08048000-What-is-stored-below-that-address

Memory Mapping and Virtual Memory

- Memory Mapping
  - I/O (e.g., driver, file)
  - Shared Memory
- **Randomization**





http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/