Memory Management

Linking/Relocation
Loading
Starting
Heap

Baremetal Section

Processor and memory architecture

Peripherals: GPIO, timers, UART

Assembly language and machine code

From C to assembly language

Functions and stack frames

Serial communication and strings

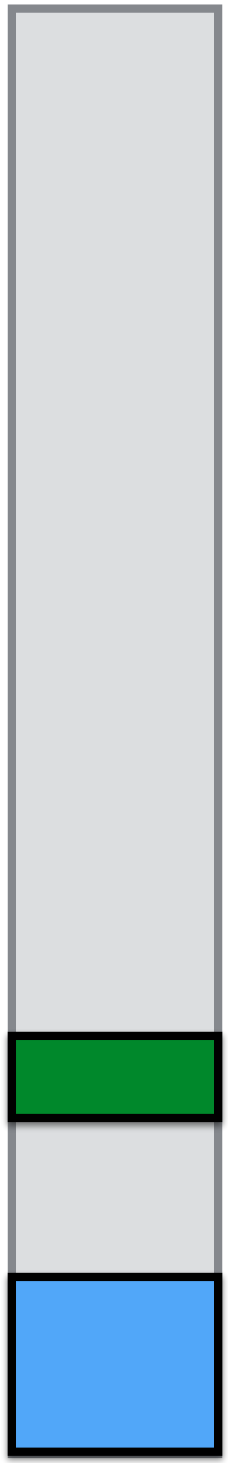
Modules and libraries: Linking

Memory Map: Linking, starting, and the heap

gpio
timer
uart
printf
malloc
keyboard
mailbox
fb
gl
console
monitor

The Memory Map

Memory Map



100000000_{16}

4 GB

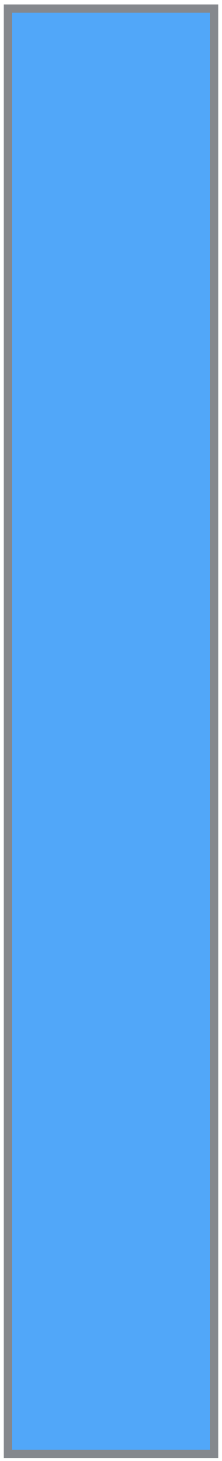
Peripheral Registers

020000000_{16}

010000000_{16}

Ref: BCM2835-ARM-Peripherals.pdf

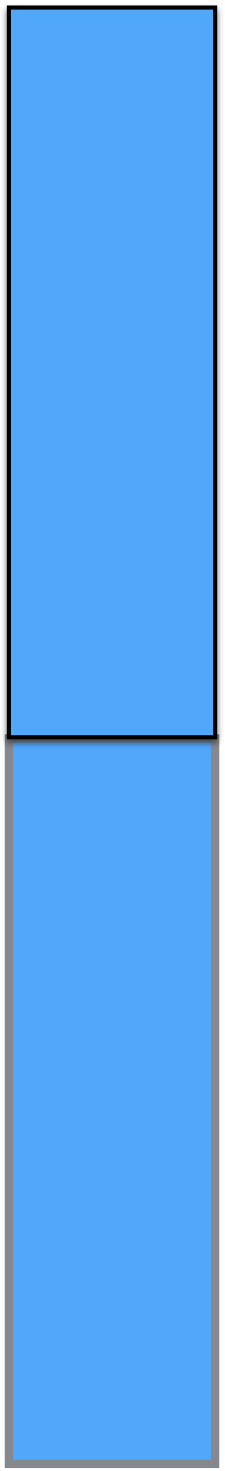
Memory Map



01000000_{16}

256 MB

Memory Map



10000000_{16}

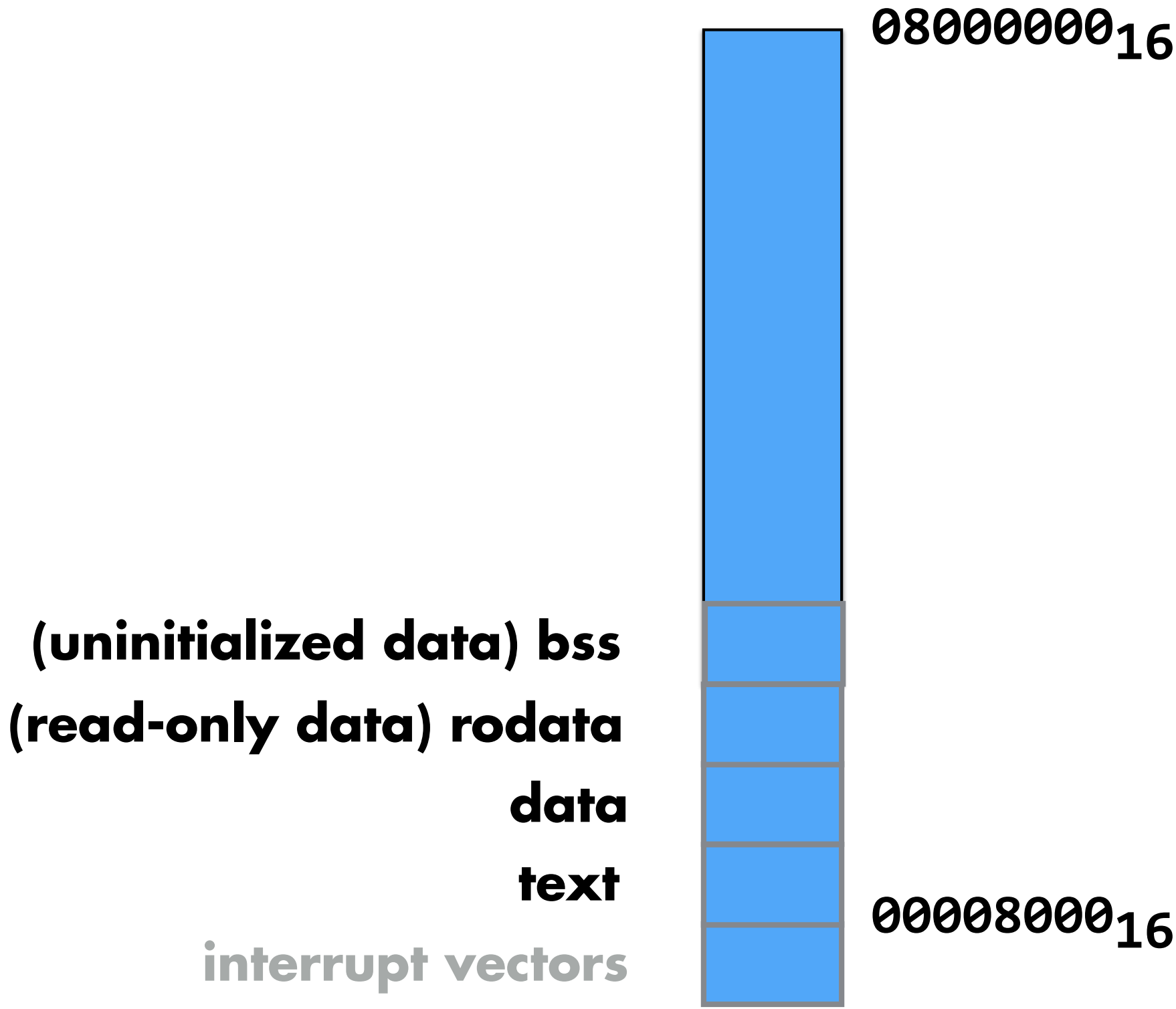
256 MB

GPU

08000000_{16}

128 MB

CPU


08000000_{16}
(uninitialized data) bss
(read-only data) rodata
data
text
interrupt vectors
00008000_{16}

linking/

```
// uninitialized global and static
int i;
static int j;

// initialized global and static
int k = 1;
static int l = 2;

// initialized global and static const
const int m = 3;
static const int n = 4;

// extern variable
extern int p;
```

```
% make tricky.o
% arm-none-eabi-nm -S -n tricky.o
                 U p
00000000 00000004 b j
00000000 00000004 D k
00000000 00000004 R m
00000000 00000044 T tricky
00000004 00000004 C i
00000004 00000004 d l

# The global uninitialized variable i
# is in common (C).

# The static const variable n
# has been optimized out.
```

Symbols

Types

- extern (undefined)
- global vs static
- initialized vs uninitialized
- const vs non-const

Guide to Symbols

T/t - text

D/d - (read-write) data

R/r - (read-only) data

B/b - bss (*Block Started by Symbol*)

C - common (used instead of B)

lower-case letter means `static`

Sections

Instructions go in `.text`

Data goes in `.data`

`const` data (read-only) goes in `.rodata`

Uninitialized data goes in `.bss`

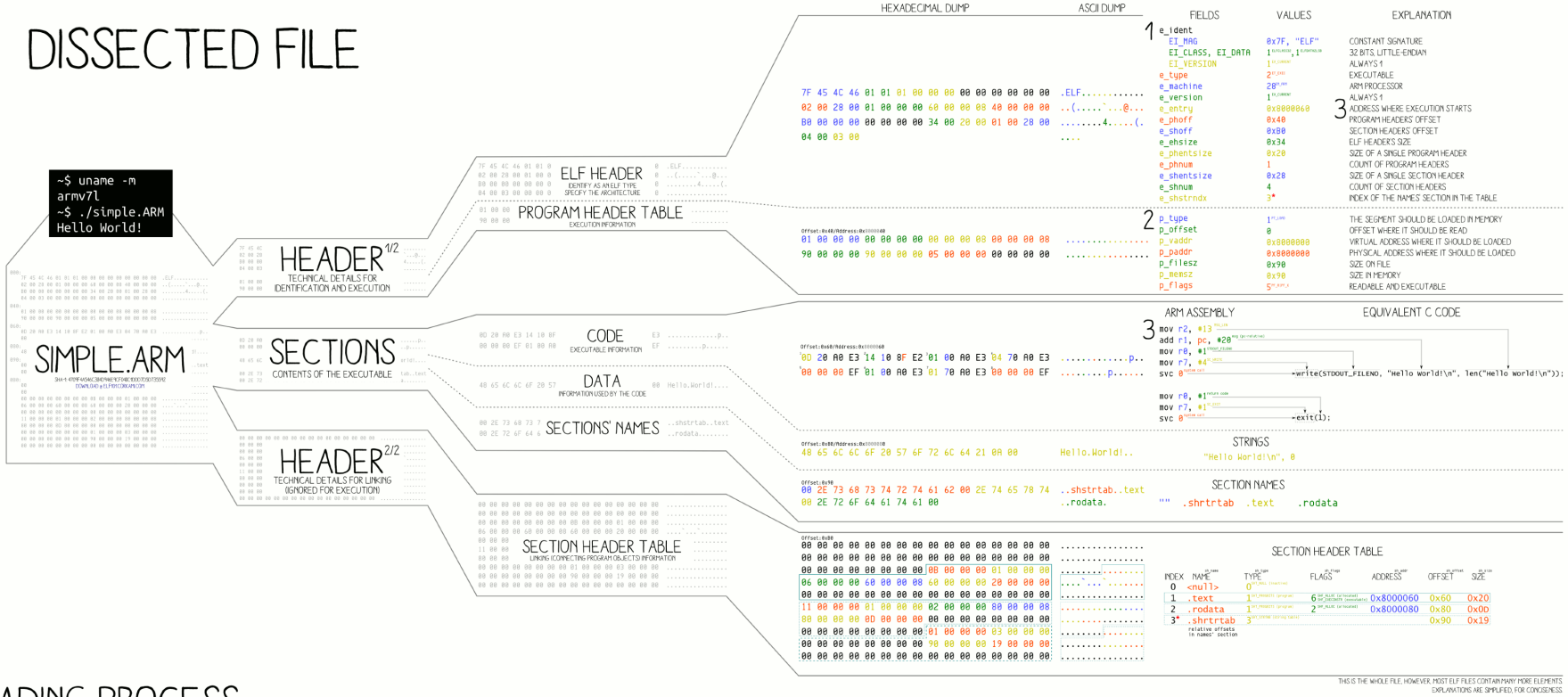
+ other information about the program

- **symbols, relocation, debugging, ...**

ELF[101] a Linux executable walk-through

ANGE ALBERTINI
CORKAMI.COM

DISSECTED FILE



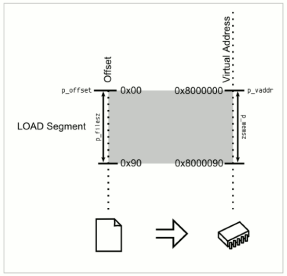
LOADING PROCESS

1 HEADER

THE ELF HEADER IS PARSED
THE PROGRAM HEADER IS PARSED
(SECTIONS ARE NOT USED)

2 MAPPING

THE FILE IS MAPPED IN MEMORY
ACCORDING TO ITS SEGMENT(S)



3 EXECUTION

ENTRY IS CALLED
SYSCALLS[GLOSS] ARE ACCESSED VIA:
- SYSCALL NUMBER IN THE R7 REGISTER[ARM]
- CALLING INSTRUCTION SVC

TRIVIA

THE ELF WAS FIRST SPECIFIED BY U.S.L. AND U.I.[SYSTEM INTERNATIONAL]
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:
- LINUX, ANDROID, *BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSES MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS

VERSION 1.0A
2013/12/06

memmap

```
.text : {
    start.o (.text)
    *(.text*)
    . = ALIGN(8);
} > ram
.data : { *(.data*) } > ram
.rodata : { *(.rodata*) } > ram
.bss : {
    __bss_start__ = .;
    *(.bss)
    *(COMMON)
    . = ALIGN(8);
    __bss_end__ = .;
} > ram
```

_start must be at #0x8000

```
% arm-none-eabi-nm -S main.exe | sort
00008000           T _start          # text
0000800c           t hang
00008010 00000038 T _cstart
00008048 00000048 T tricky
00008090 00000044 T main
000080d4 00000004 D k               # data
000080d8 00000004 d l
000080dc 00000004 D p
000080e0 00000004 R m               # rodata
000080e4           B __bss_start__  # bss
000080e4 00000004 b j
000080e8 00000004 B i
000080ec           B __bss_end__
```

```
% arm-none-eabi-size main.exe
   text    data     bss     dec     hex filename
    212      12       8     232      e8 main.exe

% a-none-eabi-size *.o
   text    data     bss     dec     hex filename
     56       0       0      56      38 cstart.o
     68       4       0      72      48 main.o
     12       0       0      12       c start.o
     76       8       4      88      58 tricky.o

# size reports the size of the text

# Note that the sum of the sizes of the .o's
# is equal to the size of the main.exe
```

Relocation

```
// start.s

.globl _start
start:
    mov sp, #0x8000
    mov fp, #0
    bl _cstart
hang:
    b hang
```

```
// Disassembly of start.o (start.list)

0000000 <_start>:
   0:   mov sp, #0x8000
   4:   mov fp, #0
   8:   bl  0 <_cstart>

0000000c <hang>:
   c:   b   c <hang>

// Note: the address of _cstart is 0
// Why?
//   _start doesn't know where c_start is!

// Note it does know the address of hang
```

```
// Disassembly of main.exe.list

00008000 <_start>:
    8000:   mov sp, #0x8000
    8004:   mov fp, #0
    8008:   bl  8010 <_cstart>

0000800c <hang>:
    800c:   b   800c <hang>

00008010 <_cstart>:
    8010:   push    {r3, lr}

// Note: the address of _cstart is #8010
// Now _start knows where _cstart is!
```

Loading and Starting

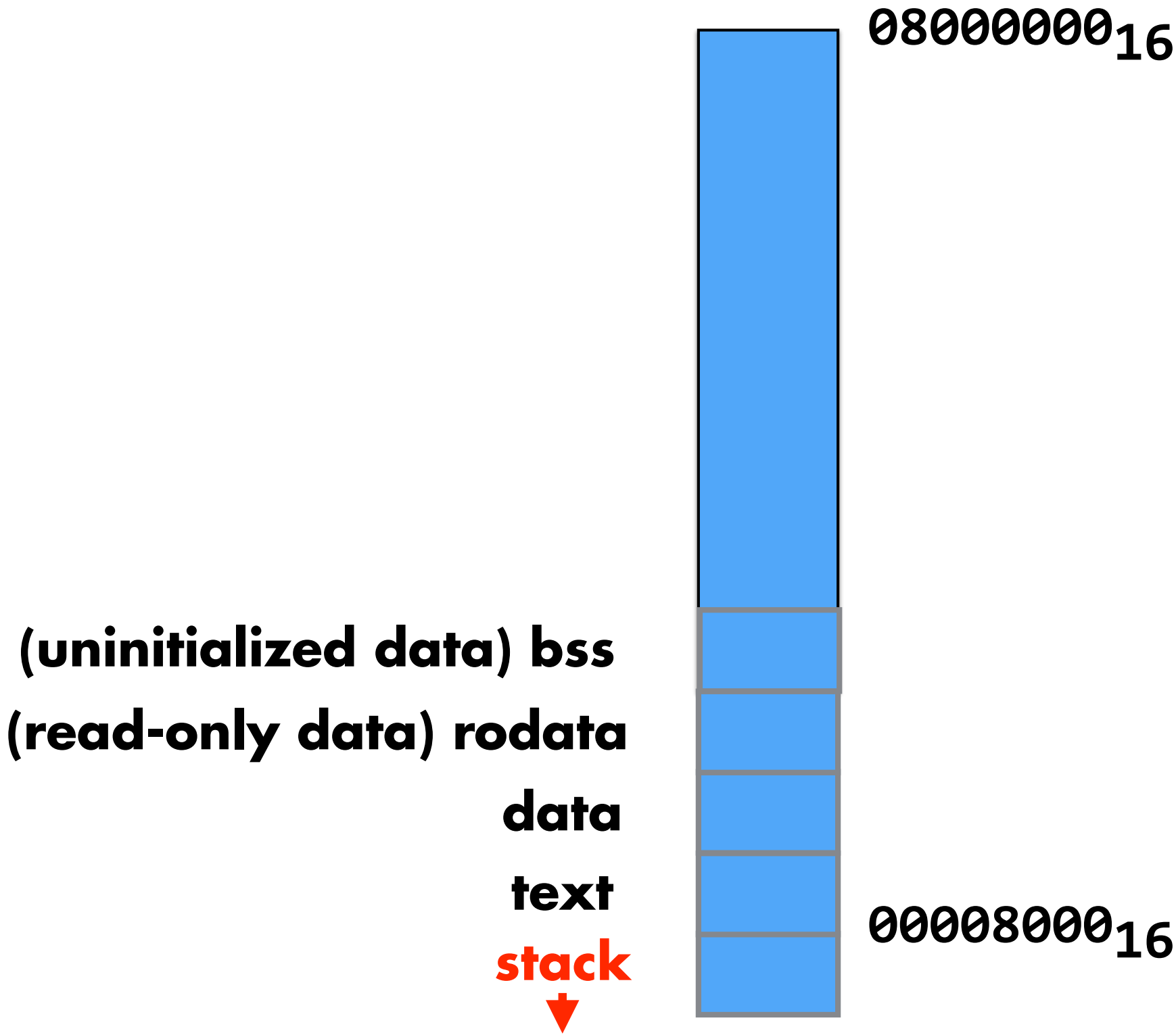
Loading and Starting

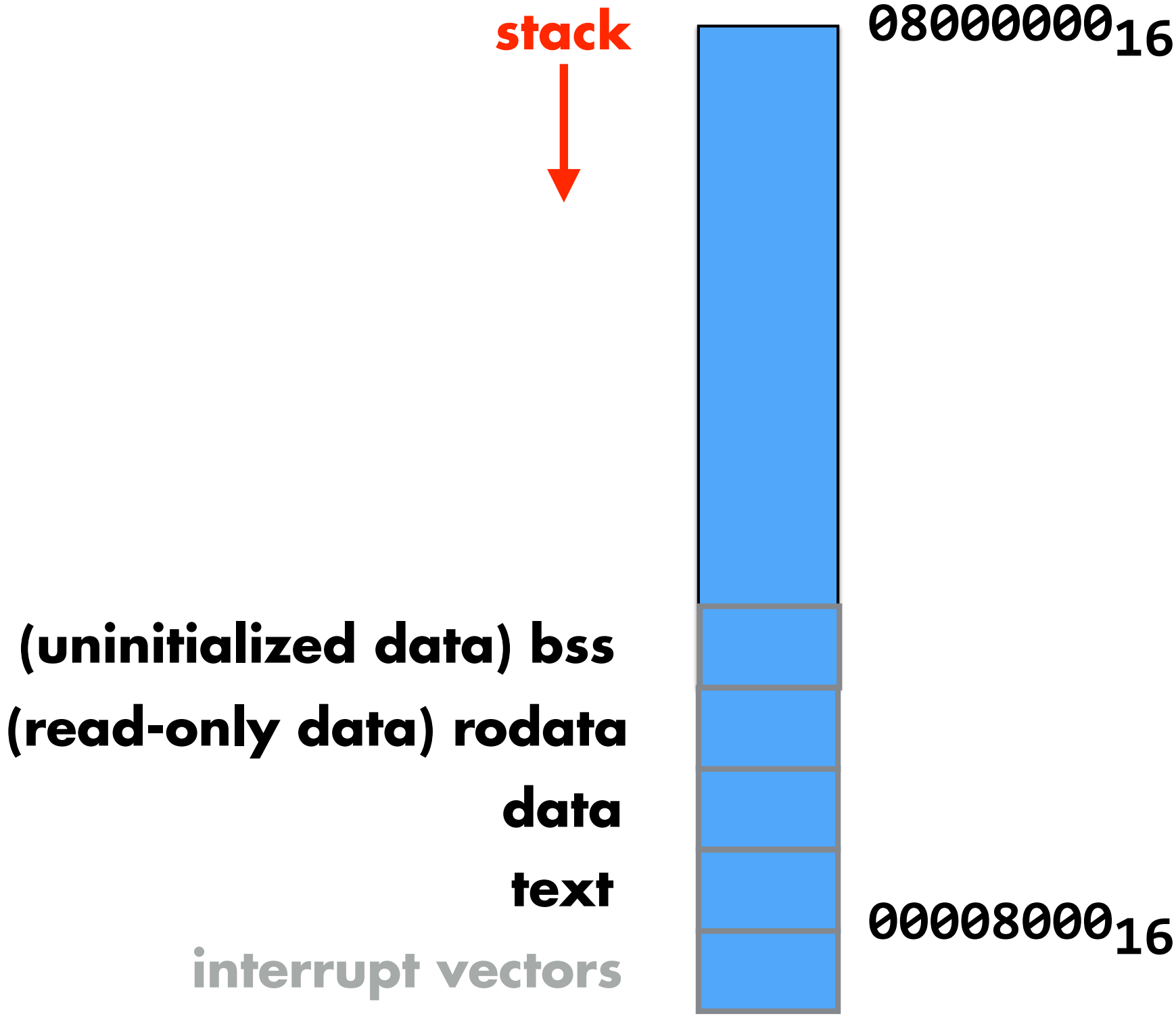
The boot loader copies the binary (`*.bin`) to memory starting at location `#0x8000`

Then the loader begins execution of the program by branching to `#0x8000`

Make sure that `_start` is at location `#0x8000`

`_start` needs to setup the stack


08000000_{16}
(uninitialized data) bss
(read-only data) rodata
data
text
stack
00008000_{16}


stack
08000000_{16}
(uninitialized data) bss
(read-only data) rodata
data
text
interrupt vectors
00008000_{16}

```
// better start.s

.globl _start
start:
    mov sp, #0x08000000
    mov fp, #0
    bl _cstart
hang:
    b hang
```

```
// cstart.c - initializes bss to 0

extern int __bss_start__;
extern int __bss_end__;
void main();

void _cstart() {
    int* bss = &__bss_start__;
    int* bss_end = &__bss_end__;
    while( bss < bss_end )
        *bss++ = 0;
    main();
}
```

Memory Allocation and the Heap

Memory Allocation?

Compile-time memory allocation

Run-time memory allocation

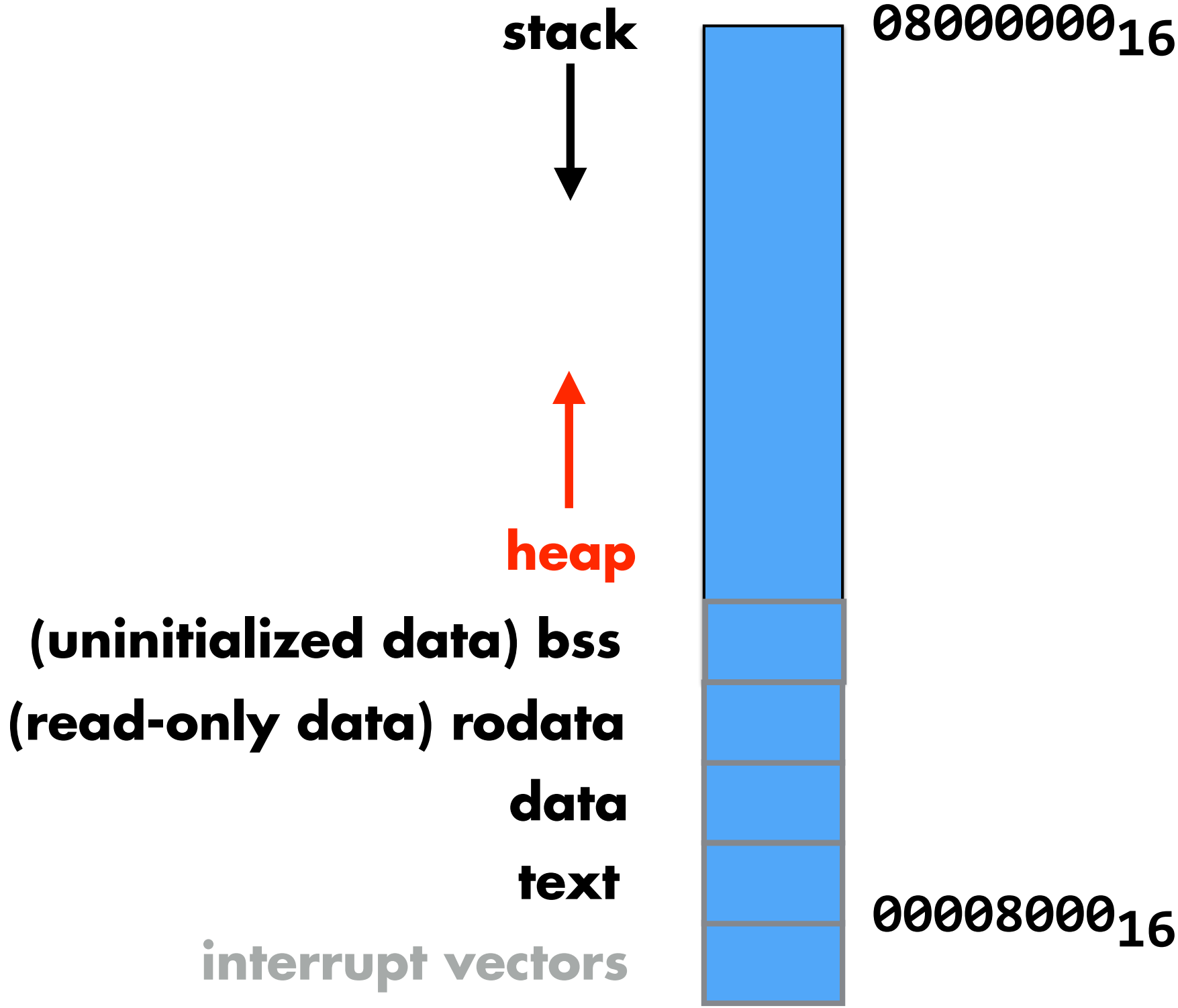
Why at run-time?

1. Don't know the size of an array when compiling

2. Dynamic data structures such as lists and trees

API

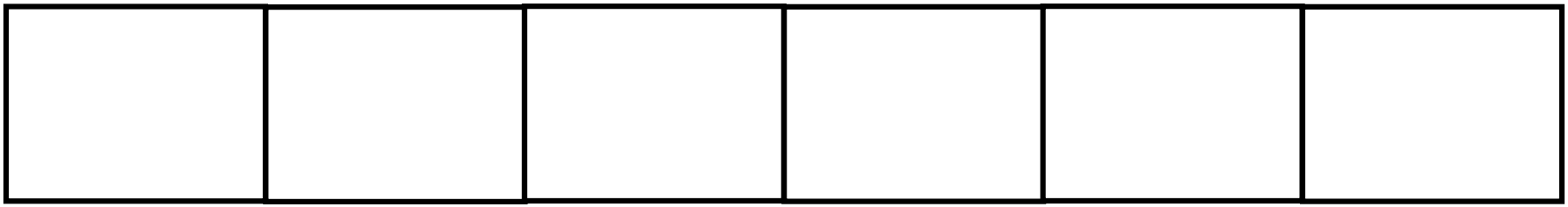
```
void *malloc( size_t size );
void free( void *pointer );

// Note that void* is a generic pointer
```


stack
08000000_{16}
heap
(uninitialized data) bss
(read-only data) rodata
data
text
interrupt vectors
00008000_{16}

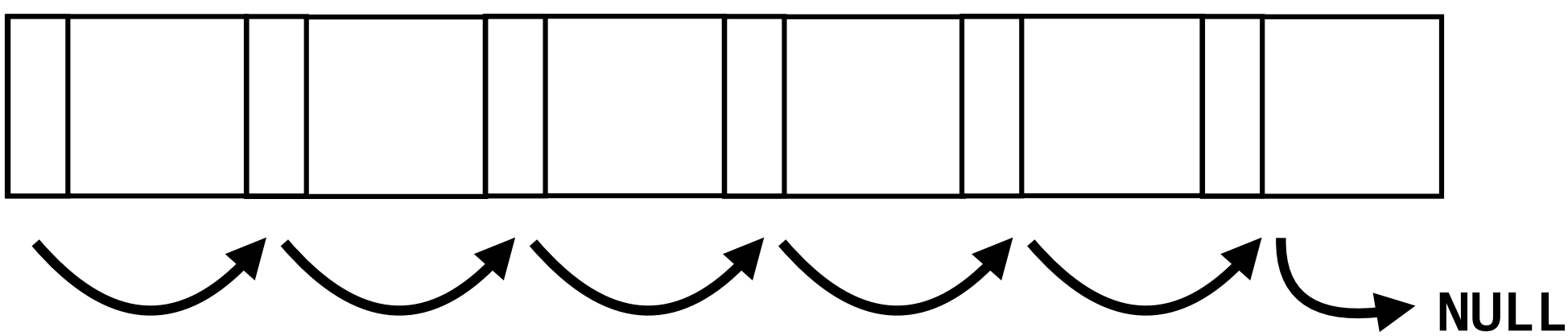
sbrk

list.c and block.c

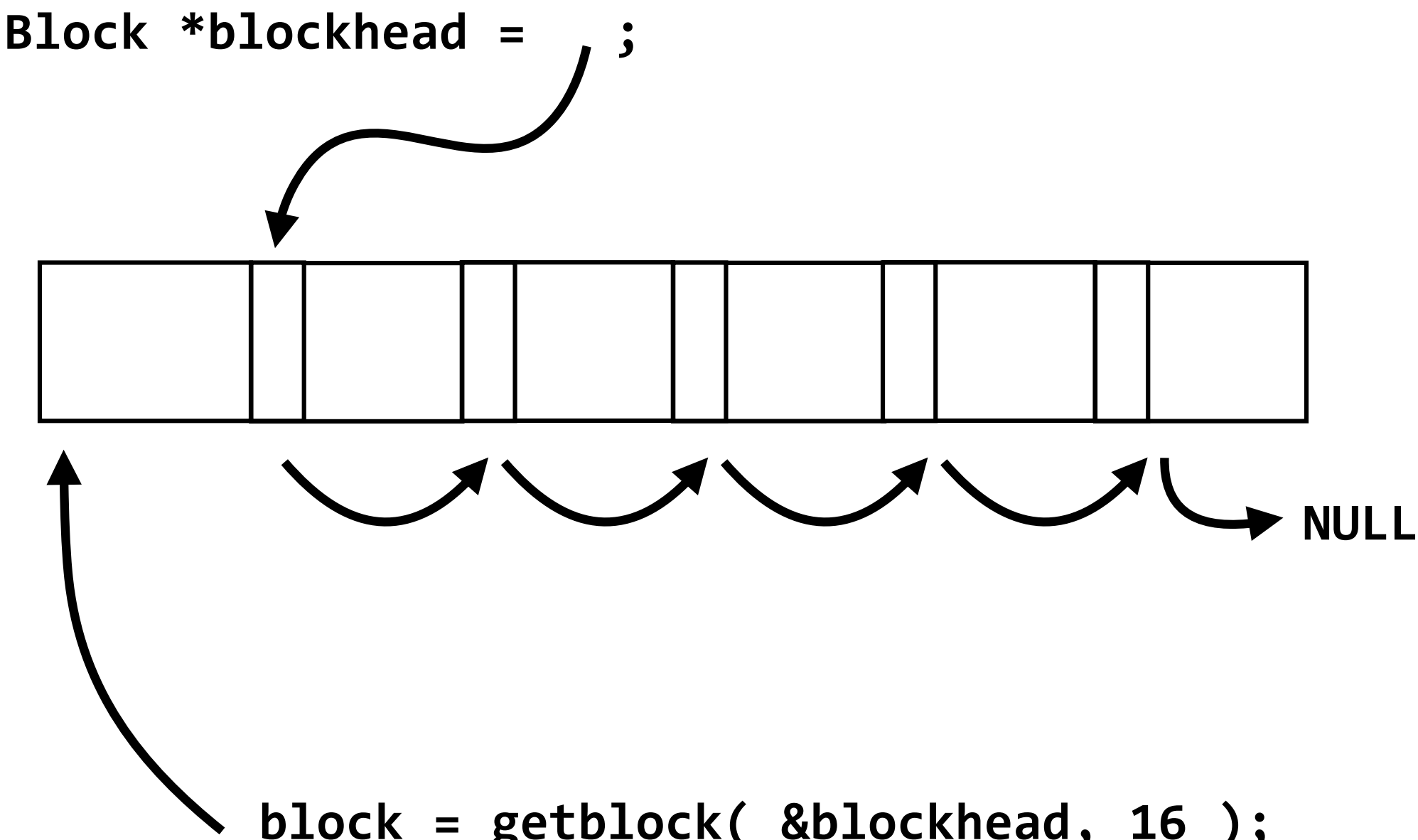
```
newblock( nelements=6, nsize=16 );
```



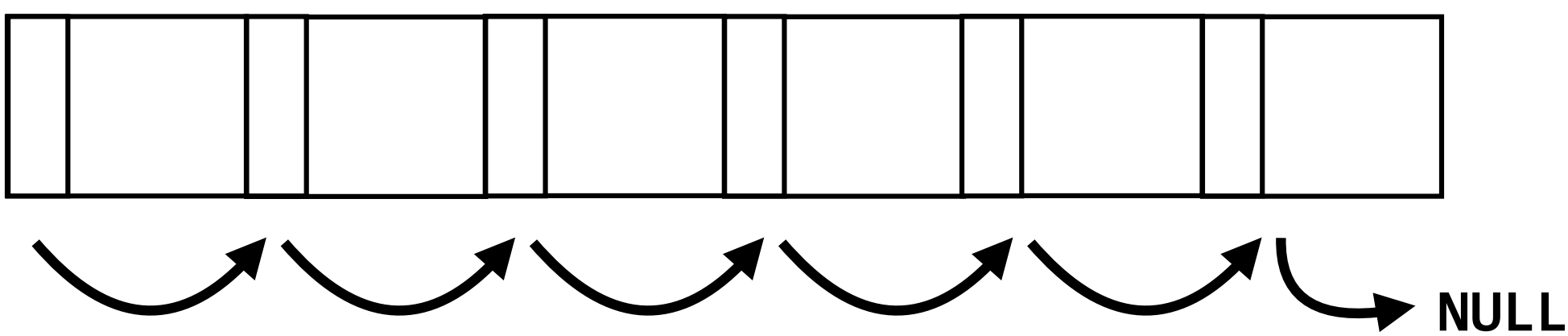
```
Block *blockhead =    ;
```




```
typedef struct b {
    struct b *next;
} Block;
```


Block *blockhead = ;
NULL
block = getblock(&blockhead, 16);

```
Block *blockhead =    ;
```




```
getblock( &blockhead, block );
```

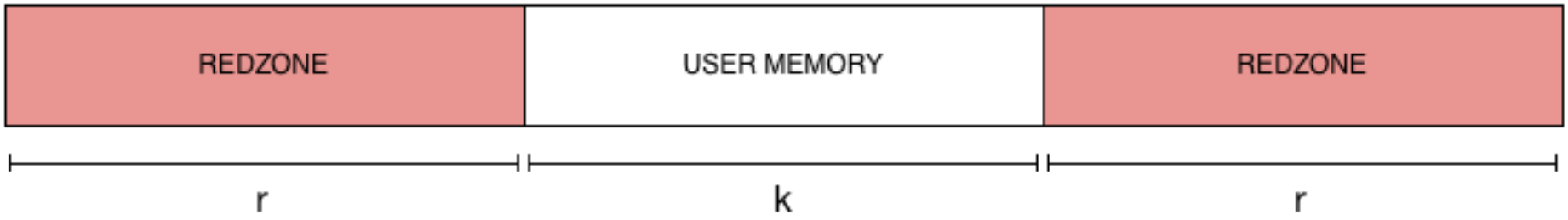
Variable Size `malloc/free`

just `malloc` is easy 🙂

`malloc` with `free` is hard 🙁

- **`free` returns blocks that can be re-allocated**
- **`malloc` should search to see if there is a block of sufficient size. Which block should it choose (best-fit, first-fit, largest)?**
- **`malloc` may use only some of the block. It splits the block into two sub-blocks of smaller sizes**
- **splitting blocks causes fragmentation**

Memory Corruption



Write special value (`#0xDEADBEEF`) to red zone
Look for unintended writes to the red zones

`red zone malloc`
`Assignment 4`