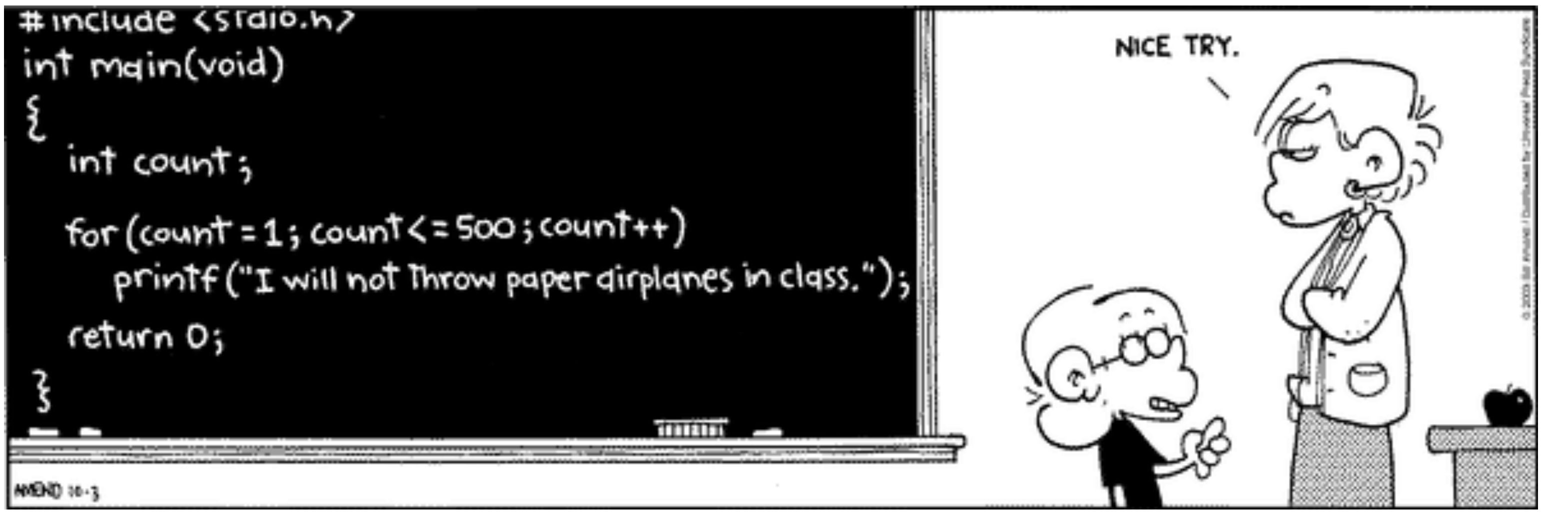
Admin

- Halfway there!
- A deep appreciation for printf -- and you'll appreciate having it!
- Project proposals due Saturday, October 24th



Today: Thanks for the memory!

Linker memory map, address space layout

Loading, how an executable file becomes a running program

Heap allocation, malloc and free

Combining Multiple Modules (`.o`) into a Single Executable (`.elf`)

`memmap`

```
// memmap
MEMORY
{
    ram : ORIGIN = 0x8000,
          LENGTH = 0x8000000
}
.text : {
    start.o (.text)
    *(.text*)
} > ram

// Why must start.o go first?
```

`_start must be located at #0x8000!!`

Magic constant that’s part of Raspberry Pi boot sequence.

```
$ arm-none-eabi-nm -n clock.elf
00008000 T _start
0000800c t hang
00008010 T square
0000801c T blink
00008070 T main
0000808c T timer_init
00008090 T timer_get_ticks
00008098 T timer_delay_us
000080a4 T timer_delay_ms
000080c0 T timer_delay
000080e0 T gpio_init
000080e4 T gpio_set_function
000080e8 T gpio_get_function
000080f0 T gpio_set_input
000080f4 T gpio_set_output
000080f8 T gpio_write
000080fc T gpio_read
00008104 T _cstart
00008154 T __bss_start__
00008158 T __bss_end__
```

```
# size reports the size of the text
% arm-none-eabi-size main.elf
   text    data     bss     dec     hex filename
    216       0       0     216      d8 main.elf

% arm-none-eabi-size *.o
   text    data     bss     dec     hex filename
      8       0       0       8       8 clock.o
     80       0       0      80      50 cstart.o
     20       0       0      20      14 gpio.o
     20       0       0      20      14 main.o
     12       0       0      12       c start.o
     76       0       0      76      4c timer.o

# 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, #0x8000000
    mov fp, #0
    bl _cstart
hang:
    b hang
```

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

0000000 <_start>:
   0:   mov sp, #0x8000000
   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 clock.elf.list

00008000 <_start>:
    8000: mov      sp, #134217728  ; 0x8000000
    8004: bl       8088 <_cstart>

00008008 <hang>:
    8008: b        8008 <hang>

00008088 <_cstart>:
    8088:   push    {r3, lr}

// Note: the address of _cstart is #8088
// Now _start knows where _cstart is!
```

data/

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

// initialized global and static variables
int k = 1;
int l = 0;
static int m = 2;

// initialized global and static const
variables
const int n = 3;
static const int o = 4;
```

```
% arm-none-eabi-nm -S tricky.o
00000004 00000004 C i
00000000 00000004 b j
00000000 00000004 D k
00000004 00000004 B l
00000004 00000004 d m
00000000 00000004 R n
00000004 00000004 r o
00000000 000000d8 T tricky

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

# If you compile with -Og, some variables
# are optimized out -- which?
```

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 (instead of B)

lower-case letter means `static`

Data Symbols

Types

- global vs static
- read-only data vs data
- initialized vs uninitialized data
- common (shared data)

Sections

Instructions go in `.text`

Data goes in `.data`

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

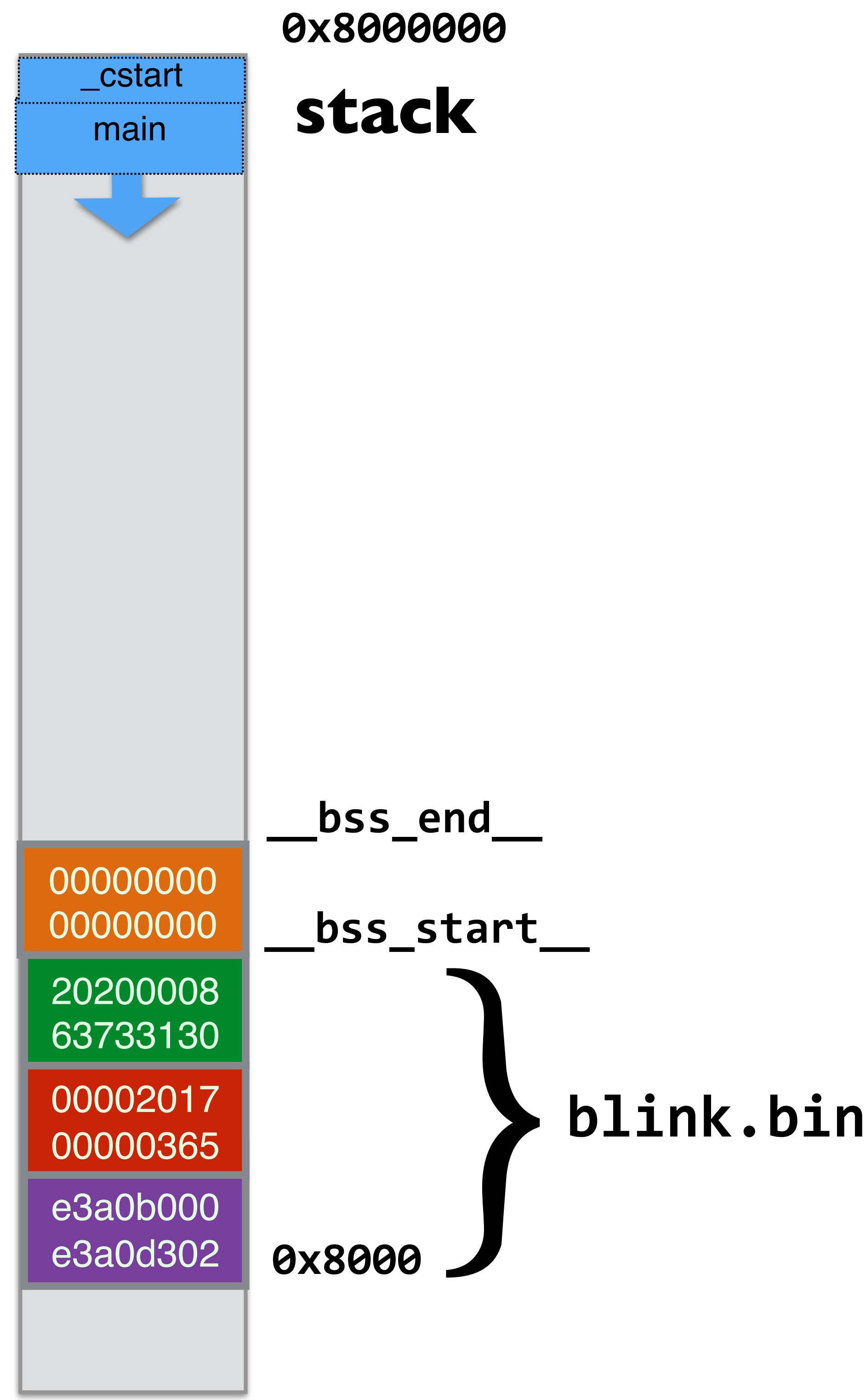
Uninitialized data goes in `.bss`

+ other information about the program

- symbols, relocation, debugging, …

```
SECTIONS
{
  .text 0x8000 :  { start.o(.text*)
                    *(.text*) }
  .data :         { *(.data*) }
  .rodata :       { *(.rodata*) }

  __bss_start__ = .;
  .bss :          { *(.bss*)
                    *(COMMON) }
  __bss_end__ = ALIGN(8);
}
```



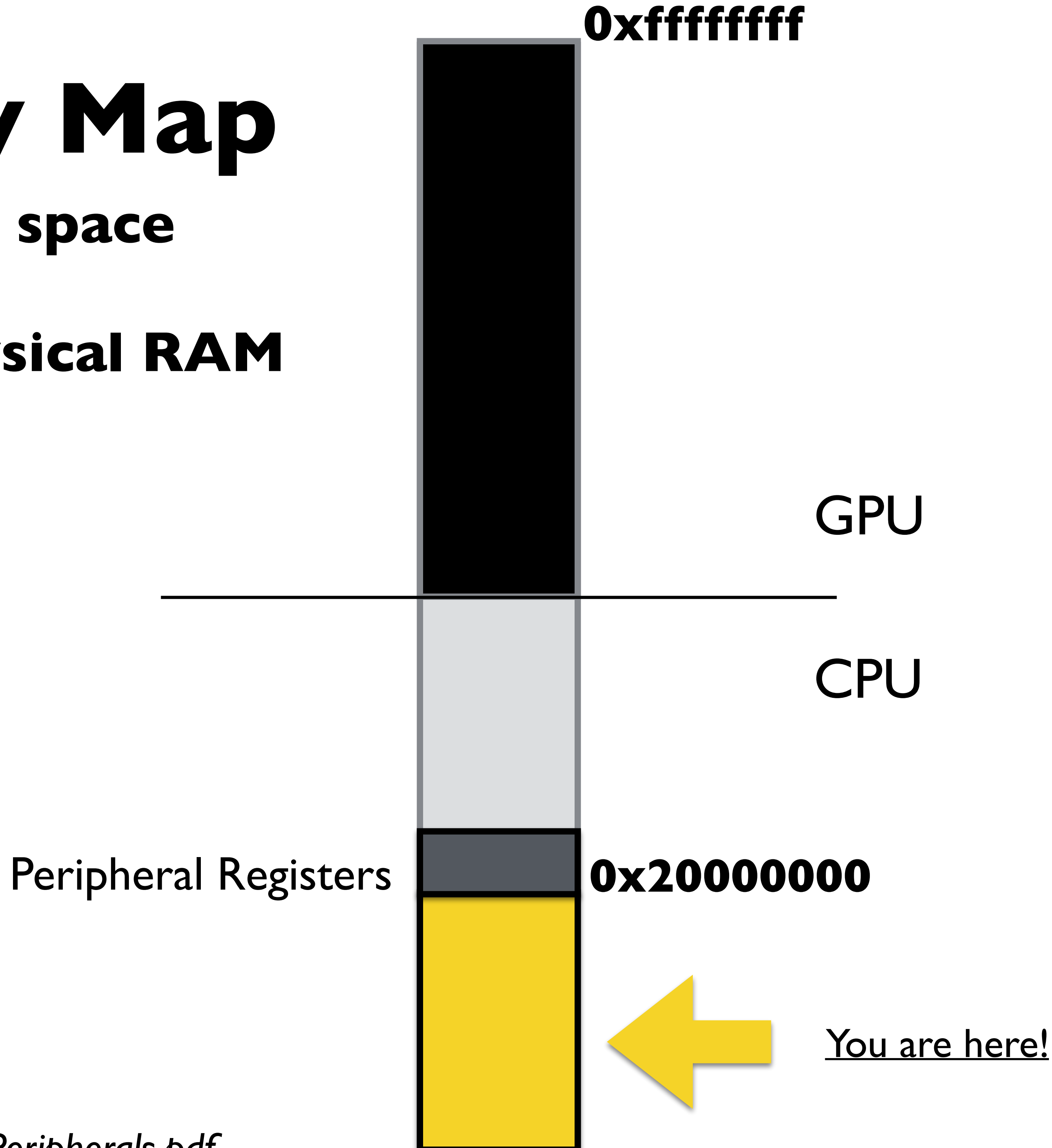
```
$ cd ../data
$ arm-none-eabi-nm -S -n main.elf
00008000 T _start
0000800c t hang
00008010 00000038 T main
00008048 00000040 T tricky
00008088 00000058 T _cstart
000080e0 00000004 D k
000080e4 00000004 R n
000080e8 R __bss_start__
000080e8 00000004 b j
000080ec 00000004 B l
000080f0 00000004 B i
000080f8 B __bss_end__
```

```
SECTIONS
{
    .text 0x8000 : {
        start.o(.text*)
        *(.text*)
    }
    .data : { *(.data*) }
    .rodata : { *(.rodata*) }
    __bss_start__ = .;
    .bss : { *(.bss*)  *(COMMON) }
    __bss_end__ = ALIGN(8);
}
```

Memory Map

32-bit address space

512 MB of physical RAM



Ref: BCM2835-ARM-Peripherals.pdf

```
SECTIONS
{
   .text 0x8000 : { start.o(.text*)
                          *(.text*)}
   .data :        { *(.data*) }
   .rodata :      { *(.rodata*) }

  __bss_start__ = .;
   .bss :         { *(.bss*)
                    *(COMMON) }
  __bss_end__ = ALIGN(8);
}
```

Use this memory for heap☞

(zeroed data) .bss

(read-only data) .rodata

(initialized data) .data

.text

0x8000000

```
_start:
   mov sp, #0x8000000
   mov fp, #0
   bl _cstart
```

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

__bss_end__

__bss_start__

blink.bin

0x8000

Global allocation

+ **Convenient**
  Fixed location, shared across entire program
+ **Fast, plentiful**
  No explicit allocation/deallocation
  But have to send over serial to bootloader (can be slow)

- **Size fixed at declaration, no option to resize**

+/- **Scope and lifetime is global**
  No encapsulation, hard to track use/dependencies
  One shared namespace, have to manually manage conflicts
  Static variables can address some issues
  Frowned upon stylistically (advanced systems reasons)

Stack allocation

+ **Convenient**

  Automatic alloc/dealloc on function entry/exit

+ **Fast**

  Fast to allocate/deallocate, good locality

- Usually don't allocate large chunks (megabytes)

- **Size fixed at declaration, no option to resize**

+/- **Scope/lifetime dictated by control flow**

  Private to stack frame

  Does not persist after function exits

- **Memory bug can corrupt execution**

Heap allocation

+ **Moderately efficient**
  Have to search for available space, update record-keeping
+ **Very plentiful**
  Heap enlarges on demand to limits of address space
+ **Versatile, under programmer control**
  Can precisely determine scope, lifetime
  Can be resized

- **Low type safety (can't access by value)**
  Interface is raw void *, number of bytes
- **Lots of opportunity for error**
  (allocate wrong size, use after free, double free)
- **Leaks**
- **Hard to track down sources of corruption**

Heap interface

void *__malloc__ (size_t nbytes);
void __free__ (void *ptr);
void *__realloc__ (void *ptr, size_t nbytes);

void* pointer

"Generic" pointer, a memory adddress

Type of pointee is not specified, unknown

What you can do with a void*

Pass to/from function, pointer assignment

What you cannot do with a void*

Cannot dereference (must cast first)

Cannot do pointer arithmetic (cast to char * to manually control scaling)

Why do we need a heap?

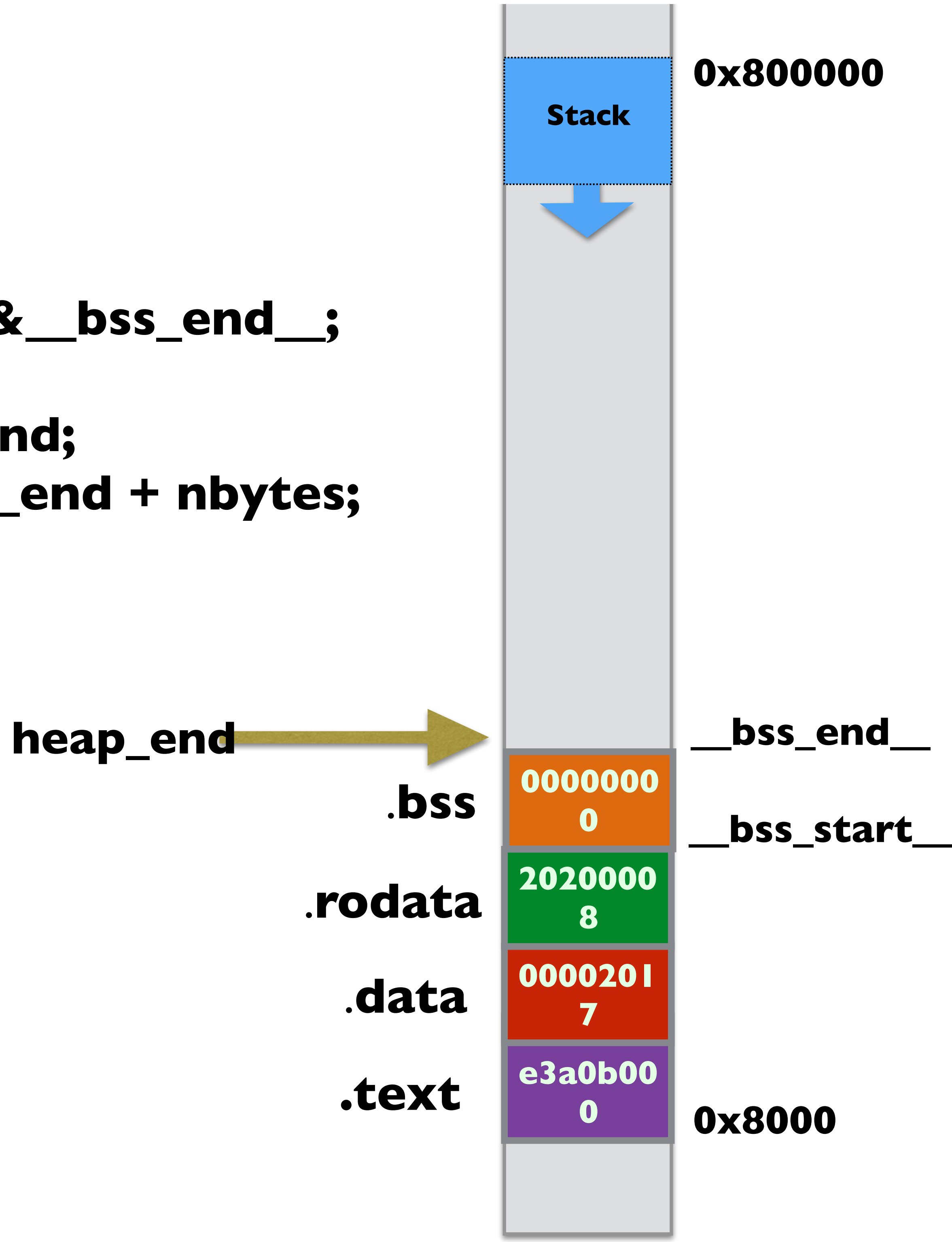
Let's see an example!

code/heap/names.c

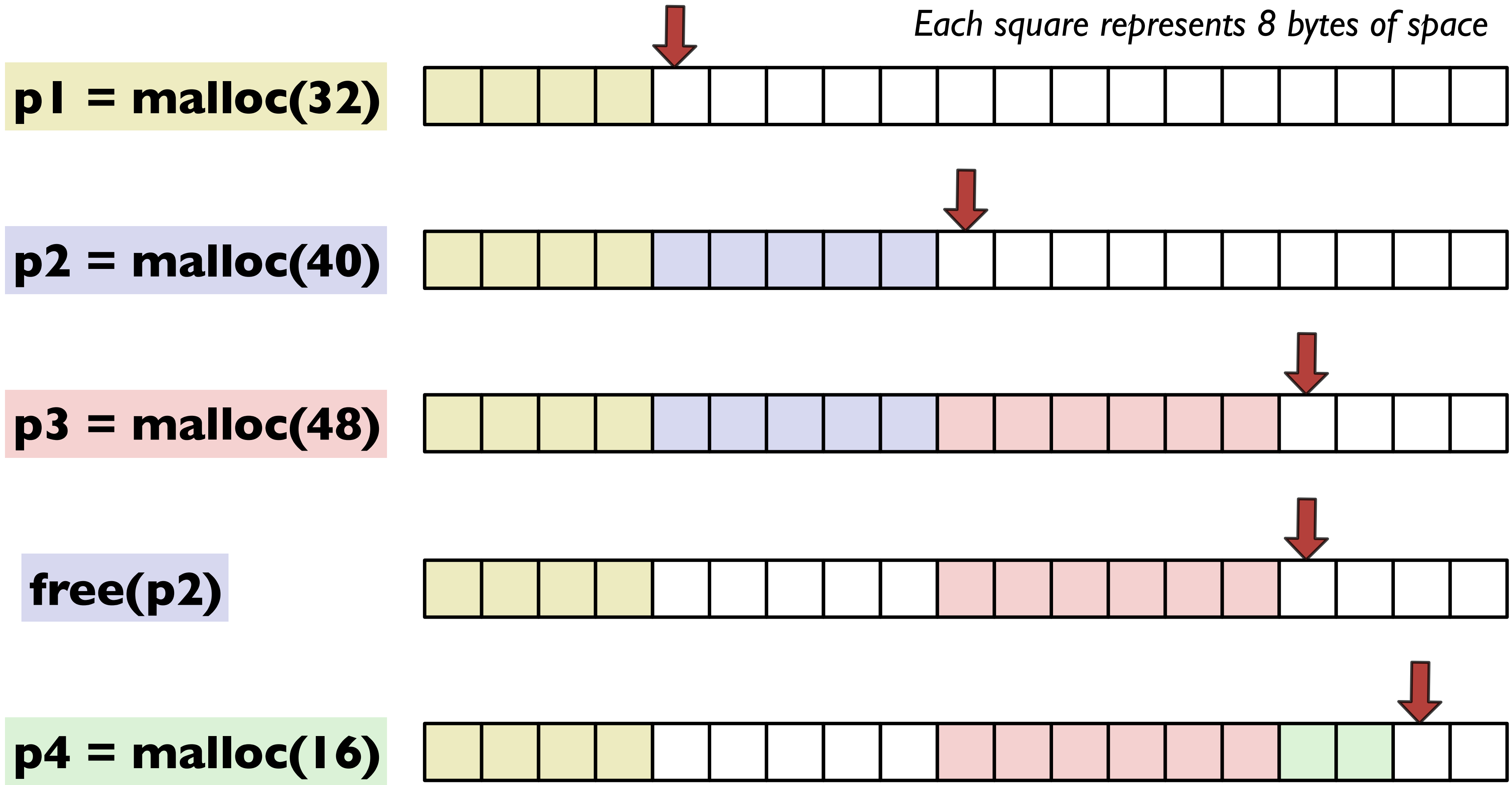
How to implement a heap

```
void *sbrk(int nbytes)
{
    static void *heap_end = &__bss_end__;

    void *prev_end = heap_end;
    heap_end = (char *)heap_end + nbytes;
    return prev_end;
}
```



Tracing the bump allocator



Bump Memory Allocator

code/heap/malloc.c

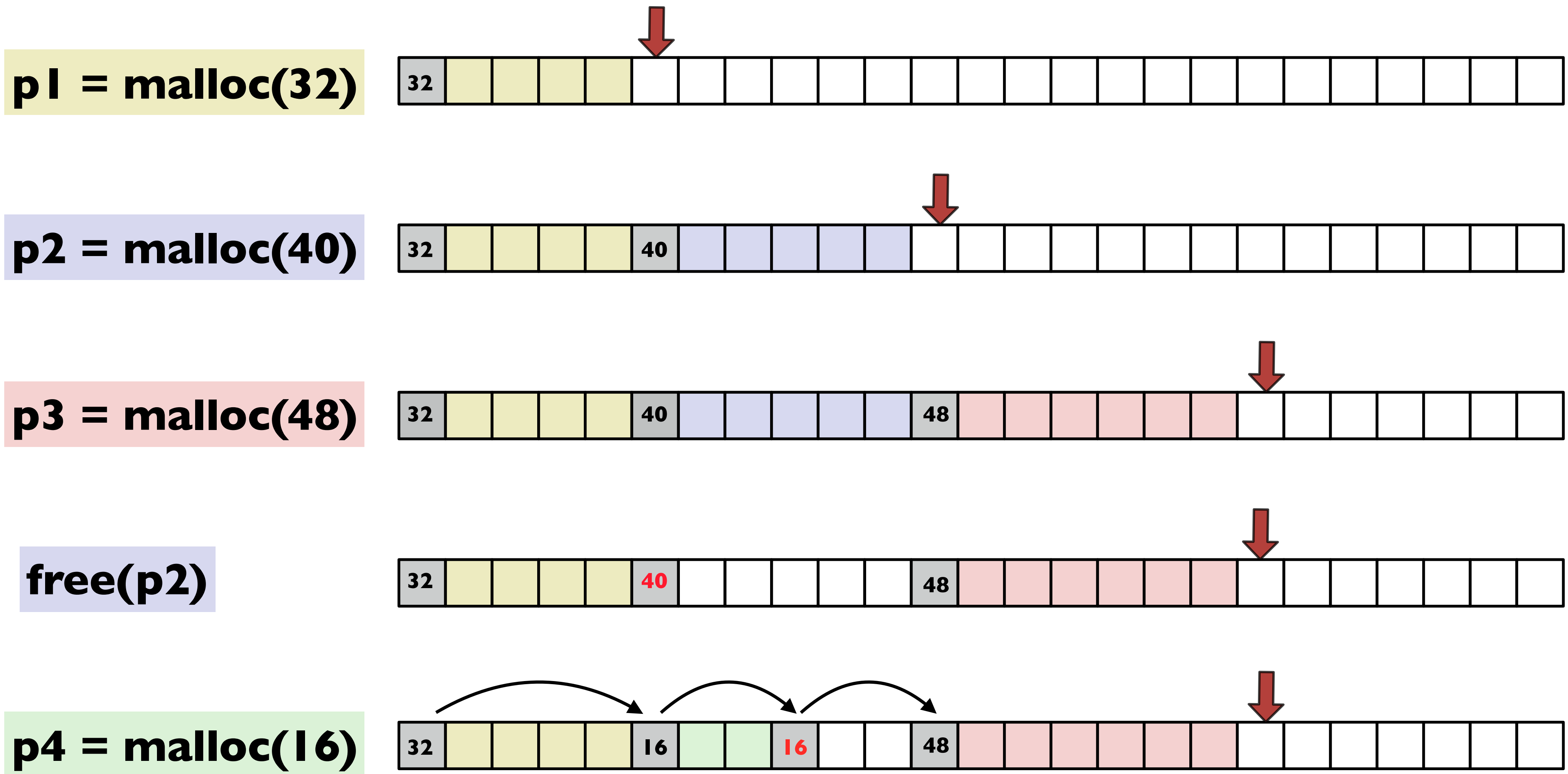
Evaluate bump allocator

+ Operations super-fast
+ Very simple code, easy to verify, test, debug

- No recycling/re-use
    (in what situations will this be problematic?)
- Sad consequences when sbrk() advances into stack
    (what can we do about that?)

Pre-block header, implicit list

Each square represents 8 bytes of space, size recorded as total byte count



Header struct

```
struct header {
    unsigned int size;
    unsigned int status;
};                              // sizeof(struct header) = 8 bytes

enum { IN_USE = 0, FREE = 1};


void *malloc(size_t nbytes)
{
    nbytes = roundup(nbytes, 8);
    size_t total_bytes = nbytes + sizeof(struct header);

    struct header *hdr = sbrk(total_bytes);
    hdr->size = nbytes;
    hdr->status = IN_USE;
    return hdr + 1;     // return address at start of payload
}
```

Challenges for malloc client

- **Correct allocation (size in bytes)**
- **Correct access to block (within bounds, not freed)**
- **Correct free (once and only once, at correct time)**

What happens if you…

— forget to free a block after you are done using it?
— access a memory block after you freed it?
— free a block twice?
— free a pointer you didn't malloc?
— access outside the bounds of a heap-allocated block?

Challenges for malloc implementor

just **malloc** is easy 😎
malloc with **free** is hard 🤔
Efficient **malloc** with **free** ….Yikes! [?]

Complex code (pointer math, typecasts)
Thorough testing is challenge (more so than usual)
Critical system component
- correctness is non-negotiable, ideally fast and compact

Survival strategies:
- draw pictures
- printf (you’ve earned it!!)
- early tests use examples small enough to trace by hand if need be
- build up to bigger, more complex tests