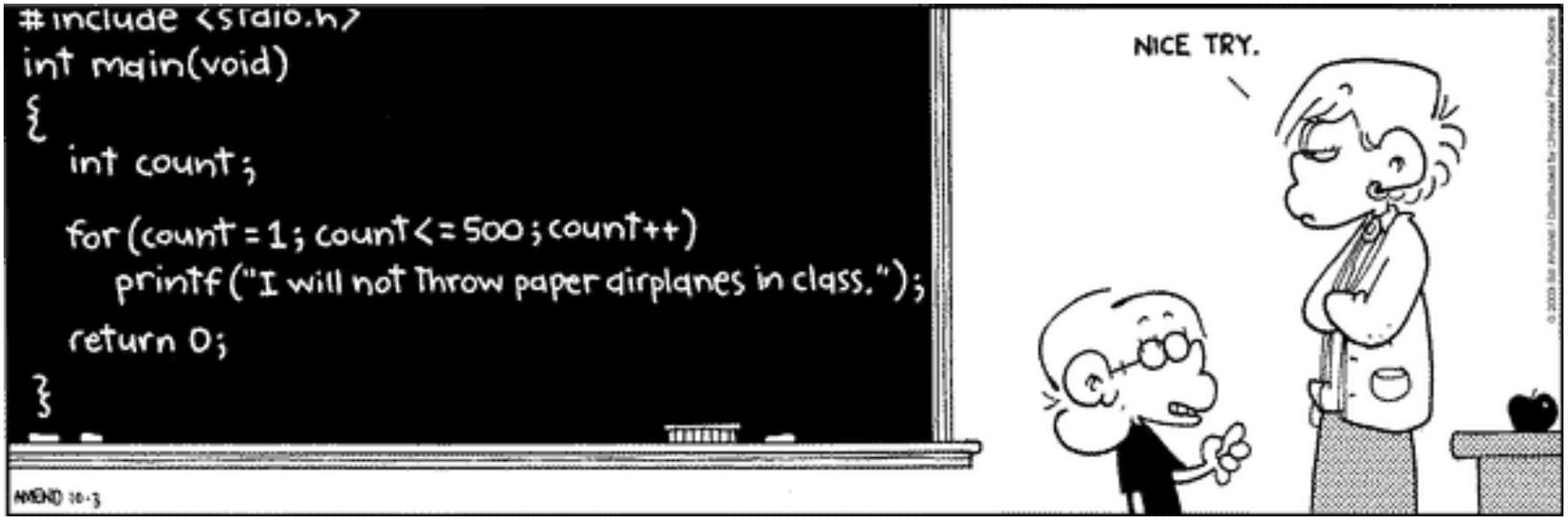
Admin

- Assign 2 issues posted, open for fix and resubmit
- printf perseverance and pride!!



Today: Thanks for the memory!

Linker memory map, address space layout

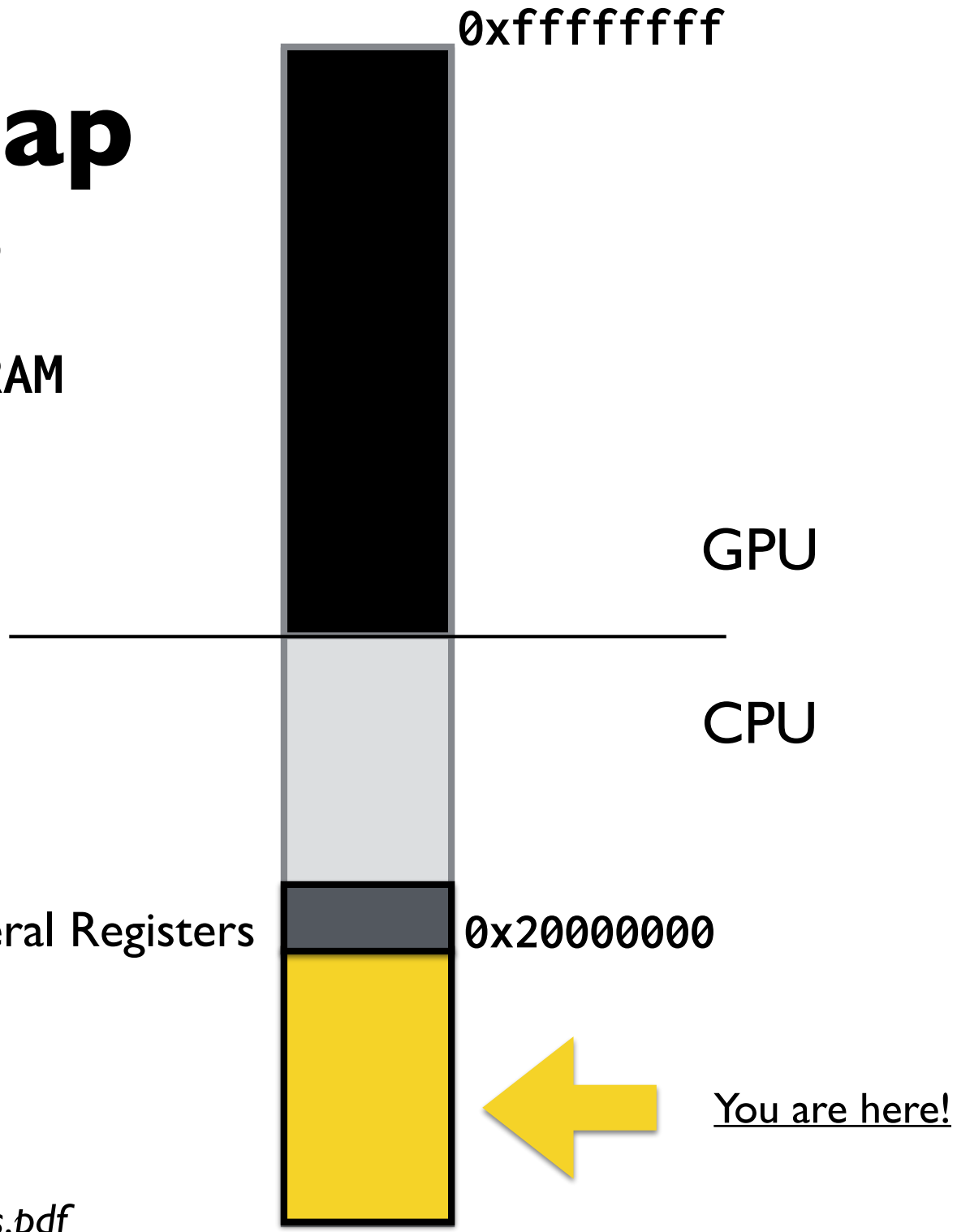
Loading, how an executable file becomes a running program

Heap allocation, malloc and free

Memory Map

32-bit address space

512 MB of physical RAM



Ref: BCM2835-ARM-Peripherals.pdf

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

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

Use this memory for heap ☞

(zeroed data) .bss

(initialized data) .data

(read-only data) .rodata

.text

0x8000000

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

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

__bss_end__

__bss_start__

0x8000

blink.bin

Global allocation

+ **Convenient**
  Fixed location, shared across entire program
+ **Fairly efficient, plentiful**
  No explicit allocation/deallocation
  Heavy cost to send over serial line to bootloader
+ **Reasonable type safety**
- **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
  Frowned upon stylistically

Stack allocation

+ **Convenient**
  Automatic alloc/dealloc on function entry/exit
+ **Efficient, fairly plentiful**
  Fast to allocate/deallocate, low consequence if large size
+ **Reasonable type safety**
- **Size fixed at declaration, no option to resize**
+/- **Scope/lifetime dictated by control flow**
  Private to stack frame
  Does not persist after function exits

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**
  Interface is raw void *, number of bytes
- **Much opportunity for error**
  (allocate wrong size, use after free, double free)
- **Leaks** (less critical, but annoying nonetheless)

Heap interface

```
void *malloc (size_t nbytes);
void  free (void *ptr);
```

void* pointer

"Generic" pointer, a memory address

Type of pointee is not specified, could be any data

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)

Cannot use array indexing (size of pointee not known!)

Why do we need a heap?

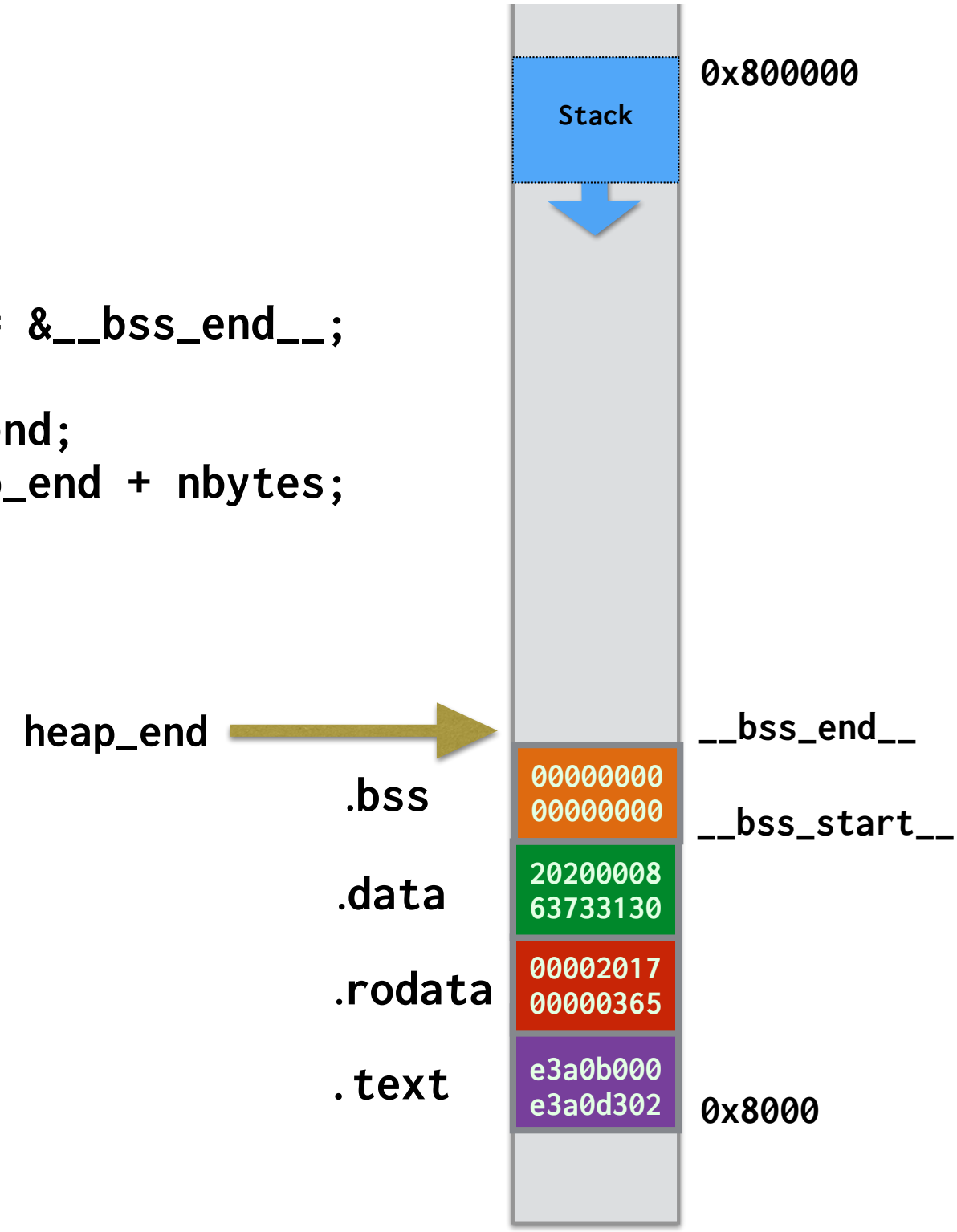
Let's see an example!

`code/heap/names.c`

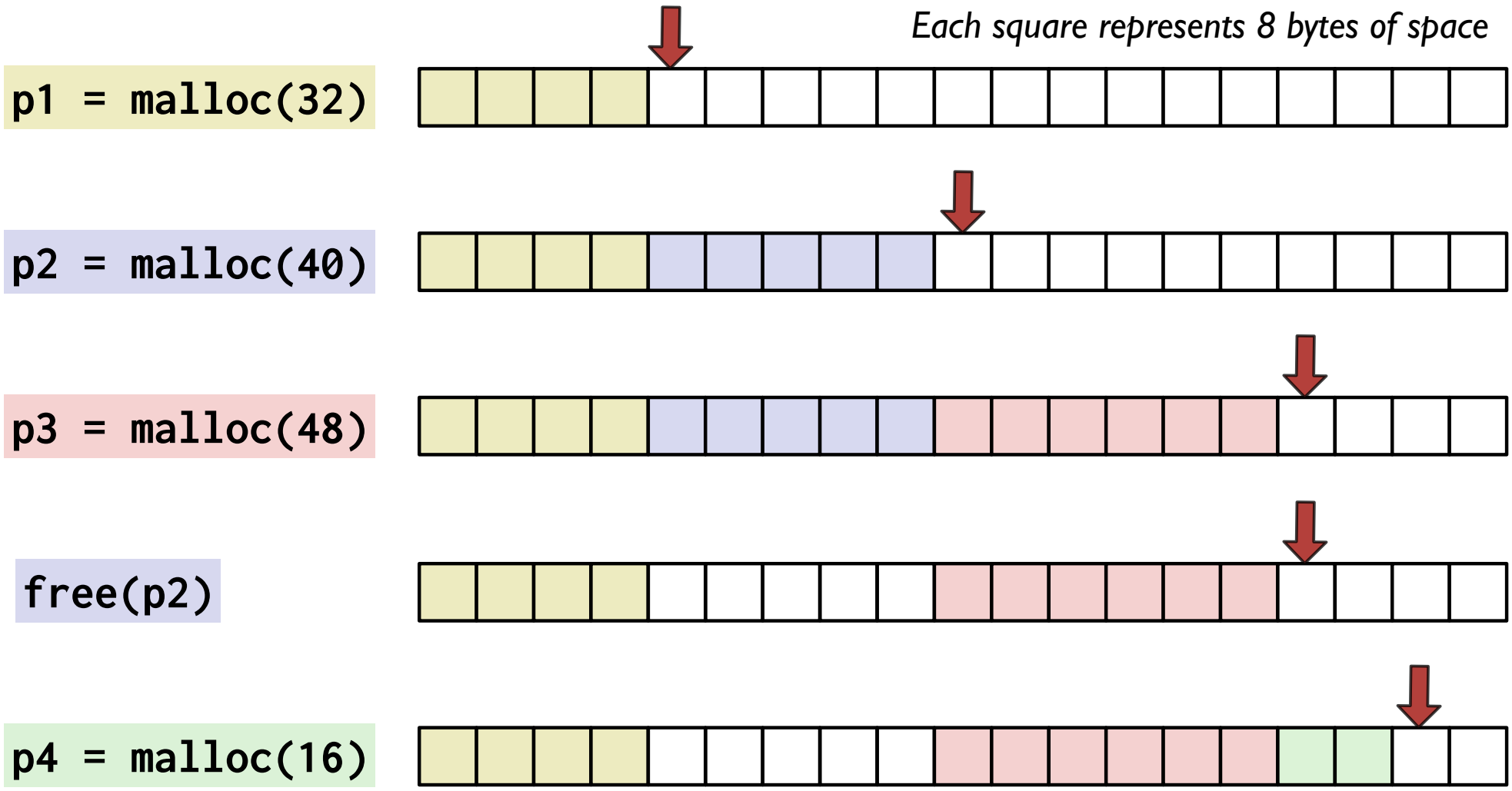
How to implement a heap

```
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?)

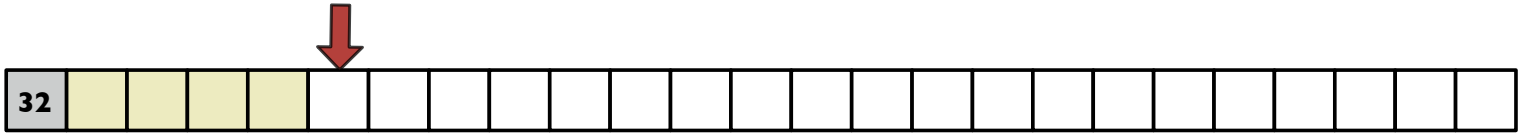
- Sad consequences when `sbrk()` advances into stack

  (what can we do about that?)

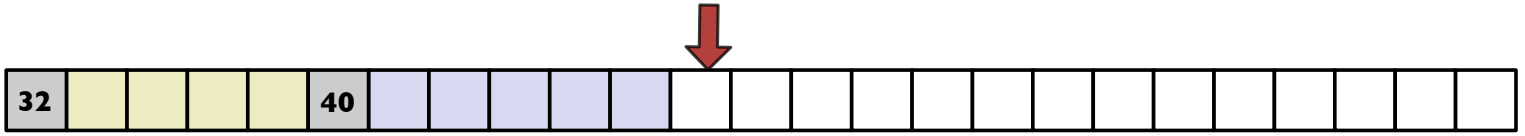
Pre-block header, implicit list

Each square represents 8 bytes, header records size of payload in bytes

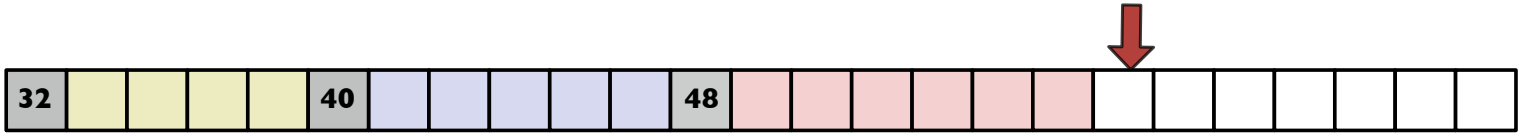
`p1 = malloc(32)`



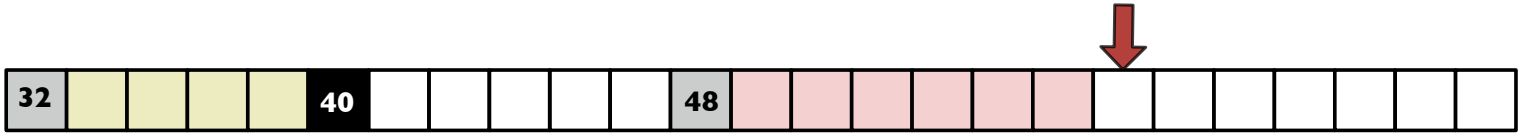
`p2 = malloc(40)`



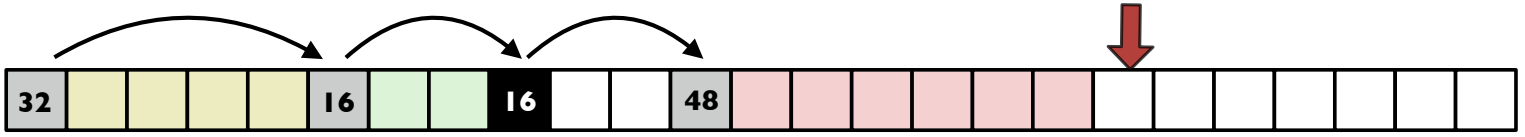
`p3 = malloc(48)`



`free(p2)`



`p4 = malloc(16)`



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); // extend end of heap
    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