

Goals for today

Leftover from previous

Stack, APCS full frame

Thanks for the memory!

Linker memory map

Address space layout

Loading

How an executable file becomes a running program

Heap allocation

Malloc and free

This week:

printf perseverance and pride!!



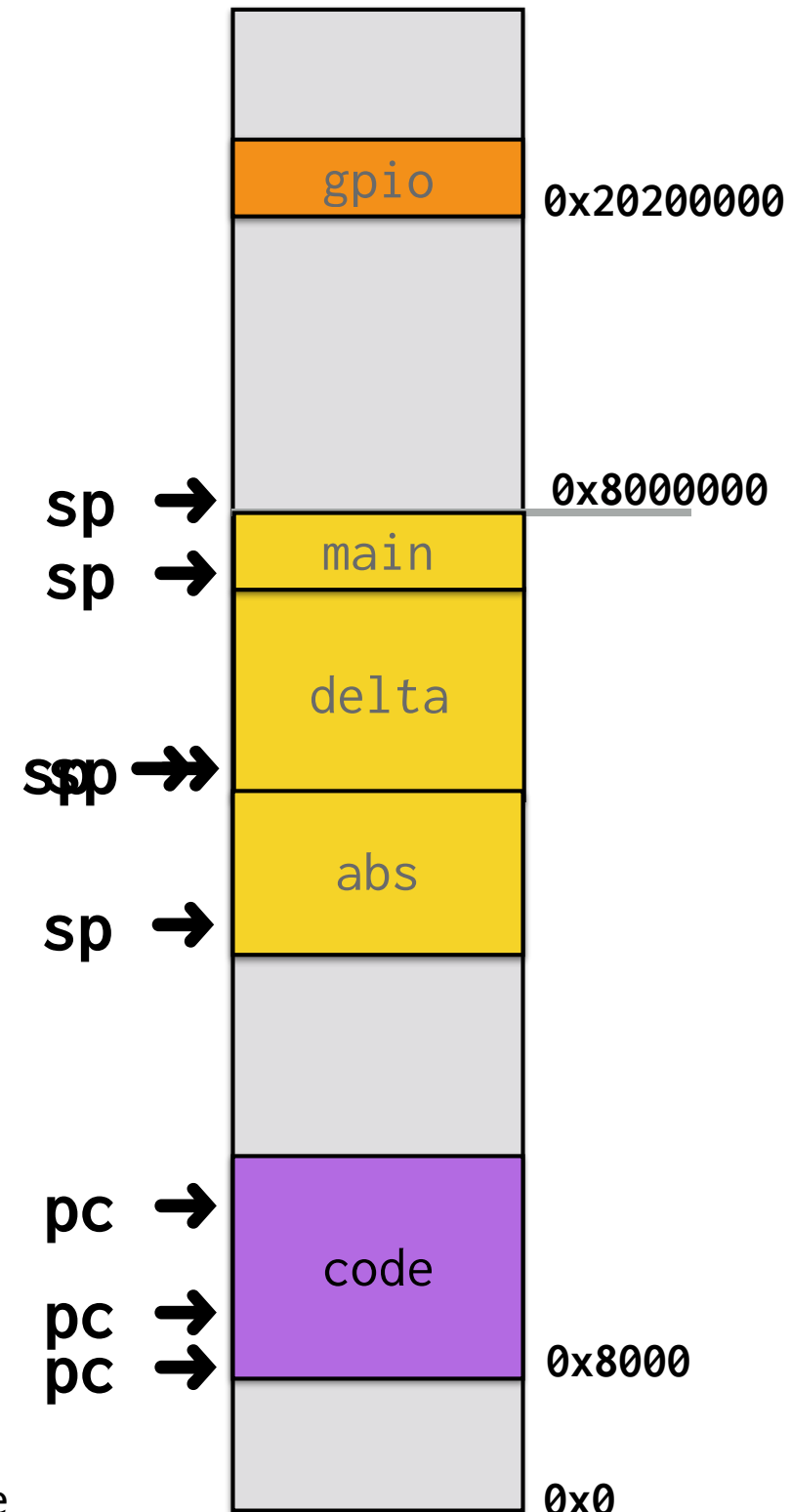
```
// start.s
mov sp, #0x8000000
bl main
```

```
void main(void)
{
    delta(10,7);
}

int delta(int x, y)
{
    return abs(x - y);
}

int abs(int v)
{
    return v < 0 : -v : v;
}
```

Diagram not to scale

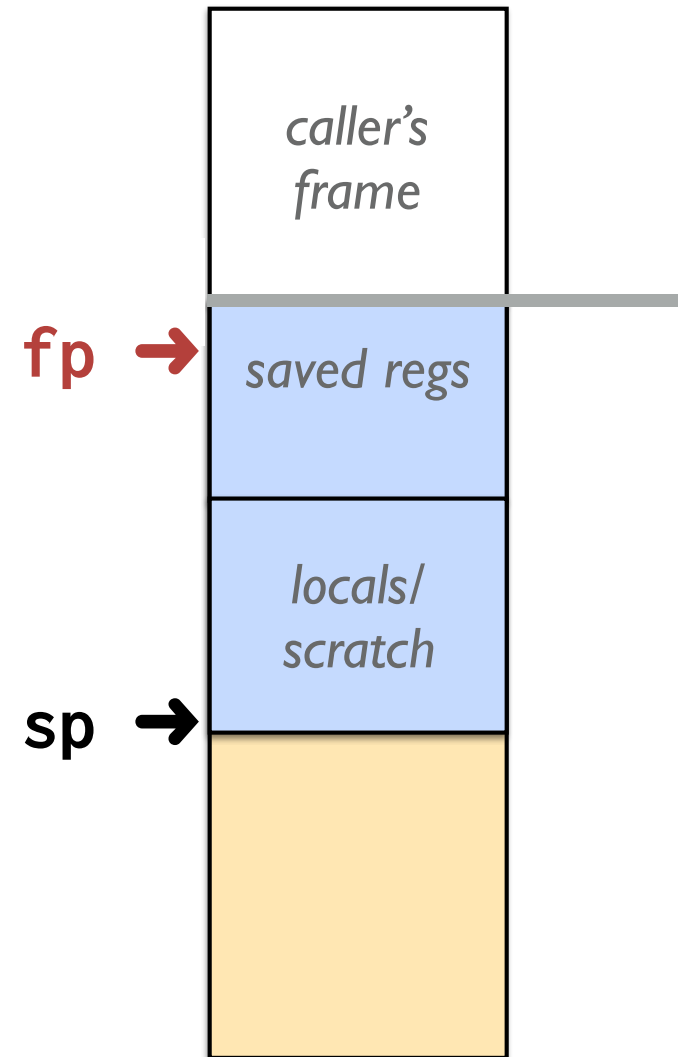


Add frame pointer

Dedicate fp register to be used as fixed anchor

Assign on entry to function to point to new stack frame

fp doesn't change, can access data at fixed offset relative to fp



APCS “full frame”

APCS = ARM Procedure Call Standard

Conventions for frame pointer and frame layout

Enable reliable stack introspection

CFLAGS to enable: `-mapcs-frame`

r11 used as fp

Adds a prolog/epilog to each function that sets up/tears down the standard frame and manages fp

Trace APCS full frame

Prolog

push fp, sp*, lr, pc
set fp to location of first pushed word

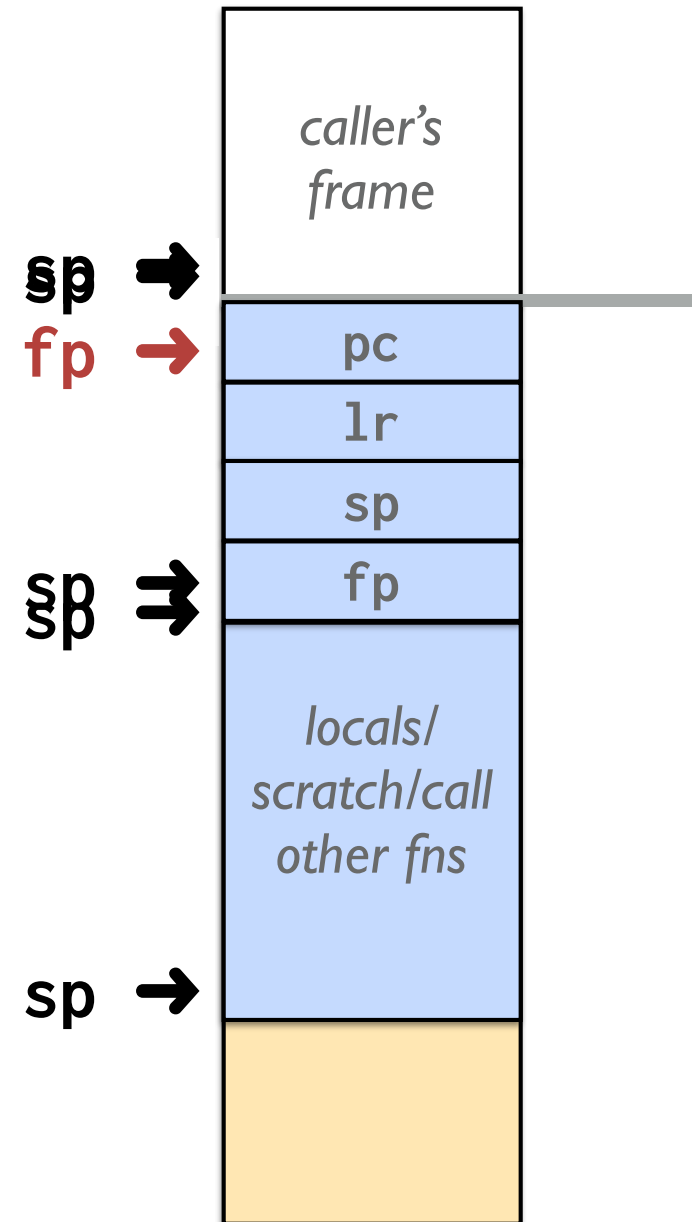
Body

fp stays anchored during body
access data on stack fp-relative
offsets don't vary even if sp changing

Epilog

pop fp, sp*, lr, pc*

* I am fudging a bit about direct use of push and pop
sp cannot be directly pushed/popped, instead moved through r12
pc not popped, instead removed from stack, no restore orig value



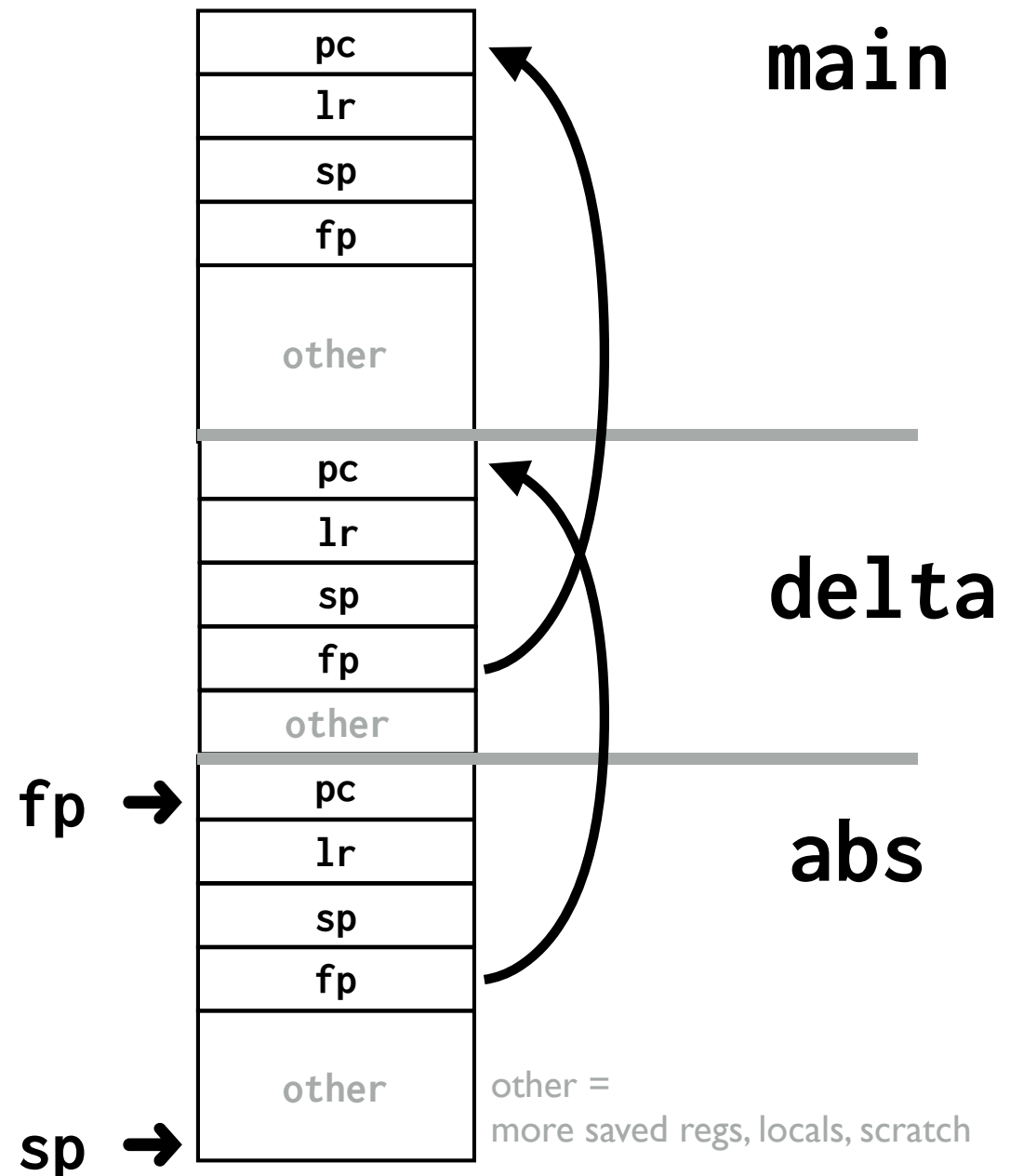
Frame pointers form linked chain

Can start at currently executing call (**abs**) and back up to caller (**delta**), from there to its caller (**main**).

Where does chain end?

```
// start.s
```

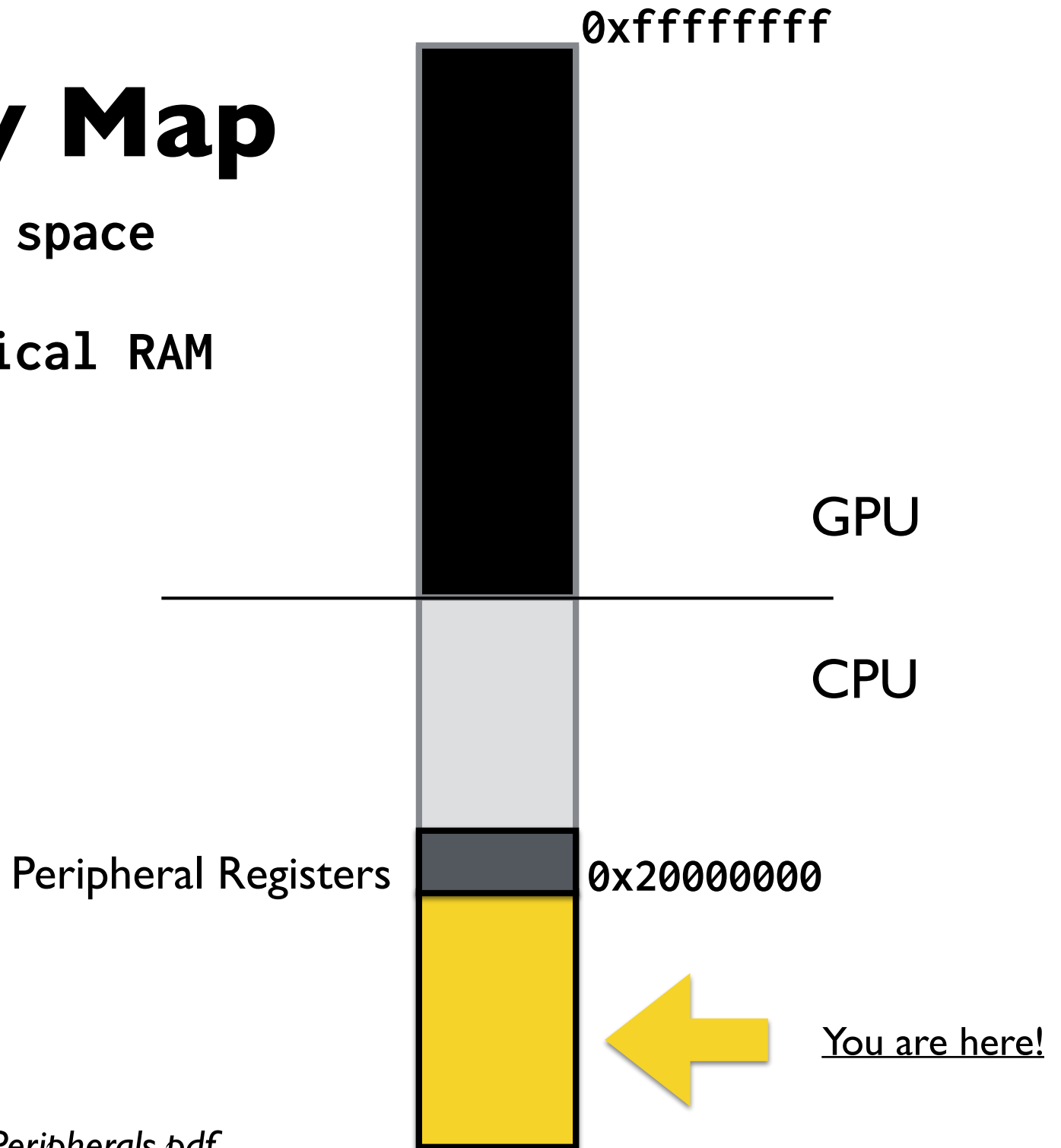
```
// init fp = NULL  
mov sp, #0x80000000  
mov fp, #0  
bl main
```



Memory Map

32-bit address space

512 MB of physical RAM



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

- + **Fairly efficient, plentiful**

 - No explicit allocation/deallocation

 - Oversize pays bootloader cost

- + **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, plentiful**

 - Automatic alloc/dealloc on function entry/exit

- + **Efficient, fairly plentiful**

 - Fast to allocate/deallocate, ok to oversize

- + **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 call 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

- **Lots of 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 *realloc(void *ptr, size_t nbytes);
```

void* pointer

Variable of type address with unspecified/unknown pointee type

What you can do with a void *

Pass to/from function, pointer assignment

What you cannot

Cannot dereference (must cast first)

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

Cannot use array indexing

Why do we need a heap?

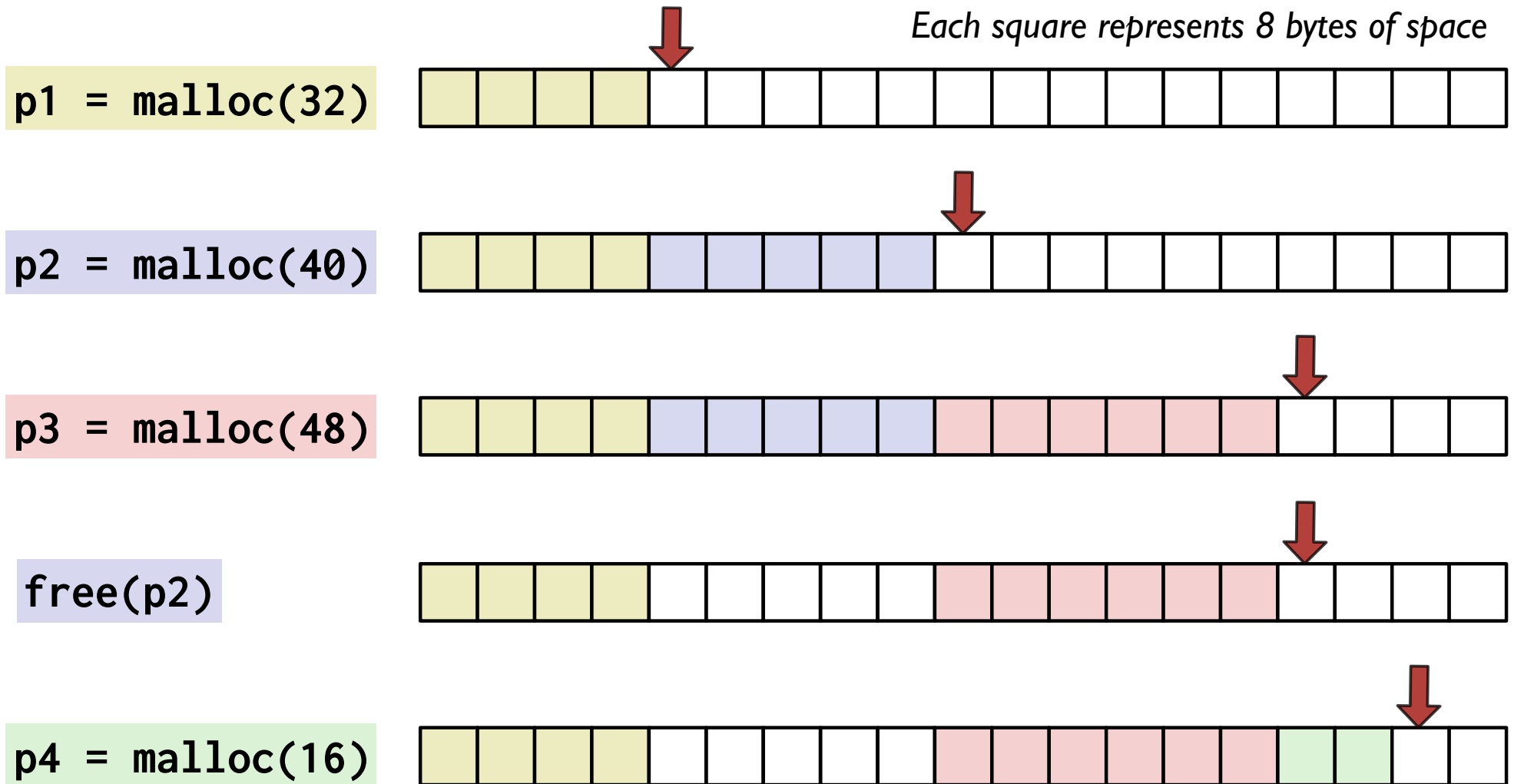
Let's see an example!

code/heap.c

How is a heap implemented?



Tracing the bump allocator



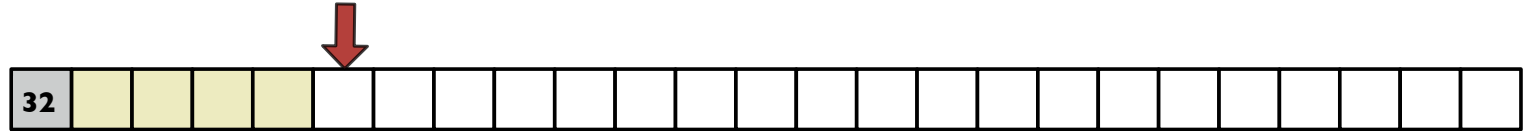
Bump Memory Allocator

malloc.c

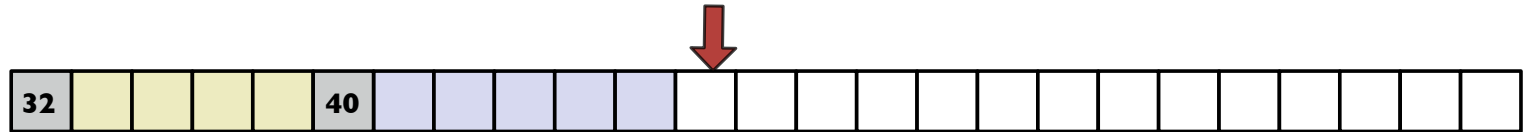
Pre-block header, implicit list

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

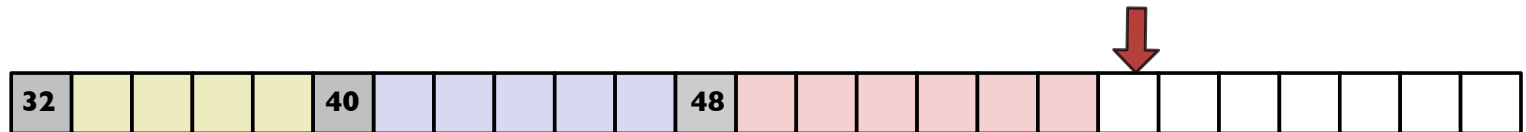
`p1 = malloc(32)`



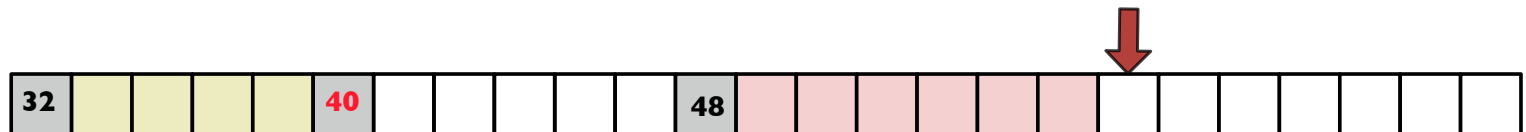
`p2 = malloc(40)`



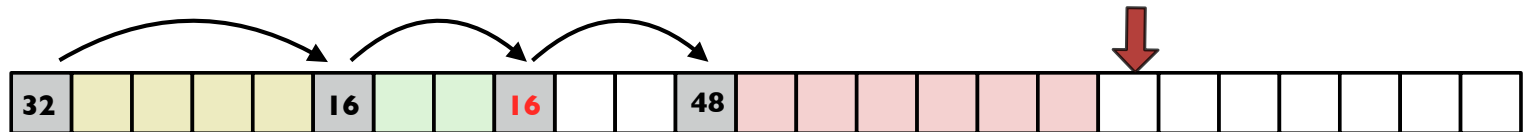
`p3 = malloc(48)`



`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};

static void *heap_end = &__bss_end__;

void *malloc(size_t nbytes)
{
    nbytes = roundup(nbytes, 8);

    struct header *hdr = (struct header *)heap_end;
    heap_end = (char *)heap_end + nbytes + sizeof(struct header);
    hdr->size = nbytes;
    hdr->status = IN_USE;
    return hdr + 1;    // payload
}
```

Challenges for malloc client

- **Correct allocation (size)**
- **Correct access to block (within bounds, not freed)**
- **Correct free at correct time**

What happens if you...

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

Challenges for malloc implementor

just malloc is easy 😎

malloc with free is hard 🤔

Efficient malloc with freeYikes! 😓

Tricky code (pointer math, typecasts)

Testing is difficult (even more than usual)

Critical system component

correctness is non-negotiable, ideally also fast and compact

Survival strategies:

draw pictures

printf (you've earned it!!)

Early tests on examples small enough to trace by hand if need be