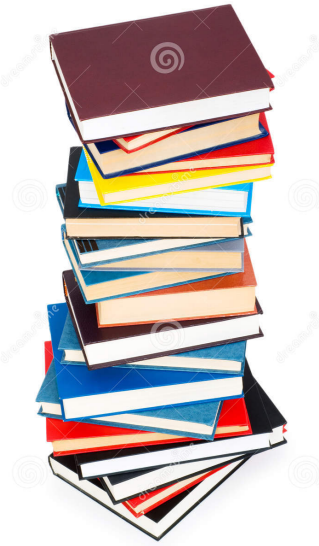
Goals for today



Thanks for the memory!

- Linker memory map
  - Address space layout
- Loading
  - How an executable file becomes a running program
- Heap allocation
  - Malloc, realloc, and free

Admin

- printf perseverance and pride!!

How is global data handled?

Factors to consider:

- extern vs static
- read-only vs read-write
- initialized vs uninitialized

```
                                // uninitialized
int gNum;
static int sgNum;

                                // initialized
int iNum = 1;
static int siNum = 2;

                                // const
const int cNum = 3;
static const int scNum = 4;
```

Note: In C, uninitialized global variables are zeroed!

```
% arm-none-eabi-nm -S data.o
00000000 00000034 T binky
00000000 00000004 R cNum
00000004 00000004 C gNum
00000004 00000004 D iNum
00000034 0000001c T main
00000000 00000004 b sgNum
00000000 00000004 d siNum
```

The global uninitialized **gNum** is common (C).

The static const **scNum** was optimized away!

Guide to symbols (nm)

`T/t` - text (code)
`D/d` - read-write data
`R/r` - read-only data (**`const`**)
`B/b` - bss (*Block Started by Symbol*)
`C` - common (instead of B)

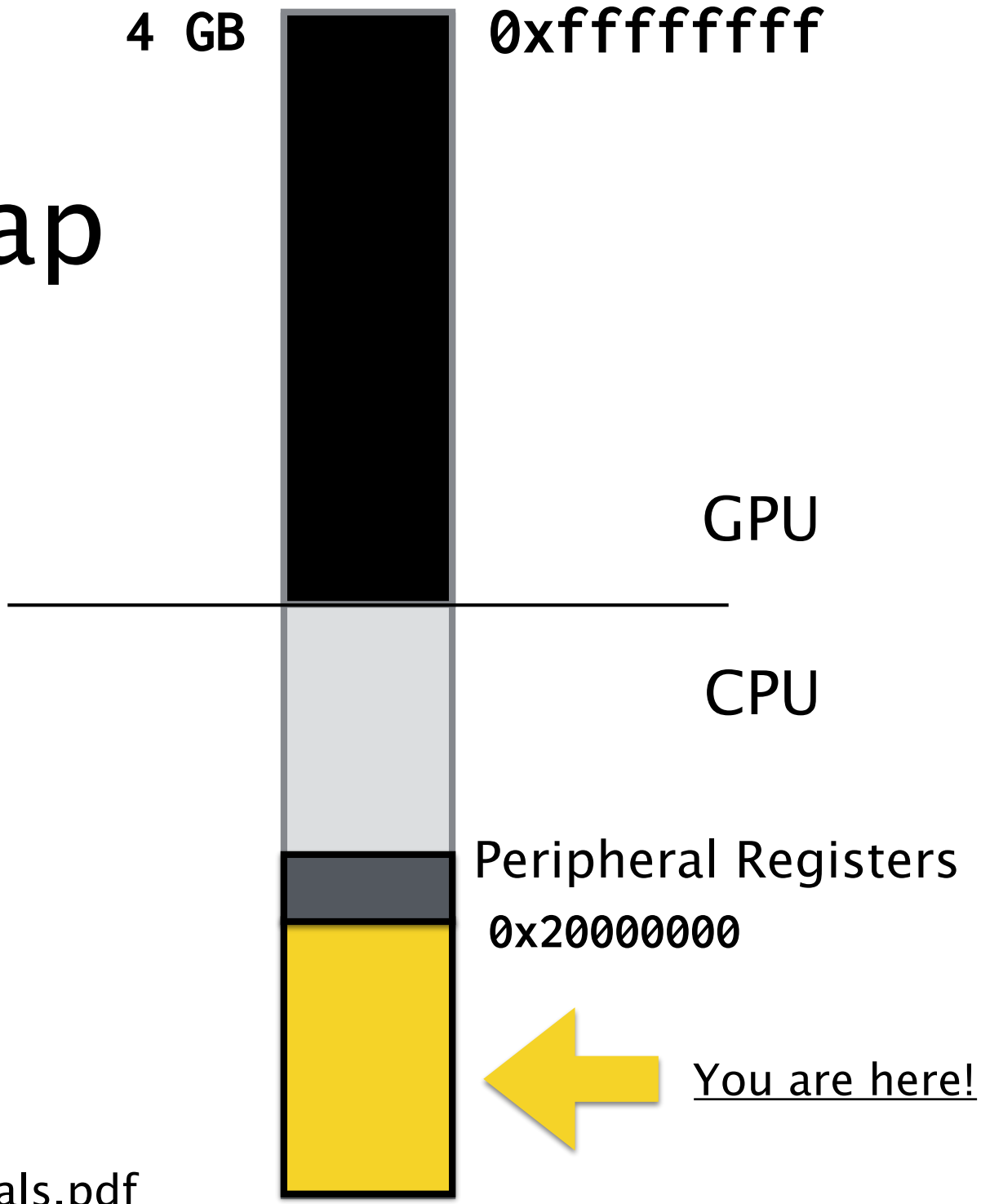
lowercase for static, uppercase for extern

memmap linker script

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

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

```
// memmap wraps bfs section with these symbols
extern int __bss_start__, __bss_end__;

extern void main(void);

void _cstart(void) {
    int *bss = &__bss_start__;
    int *bss_end = &__bss_end__;

    while (bss < bss_end) {
        *bss++ = 0;
    }

    main();
}
```

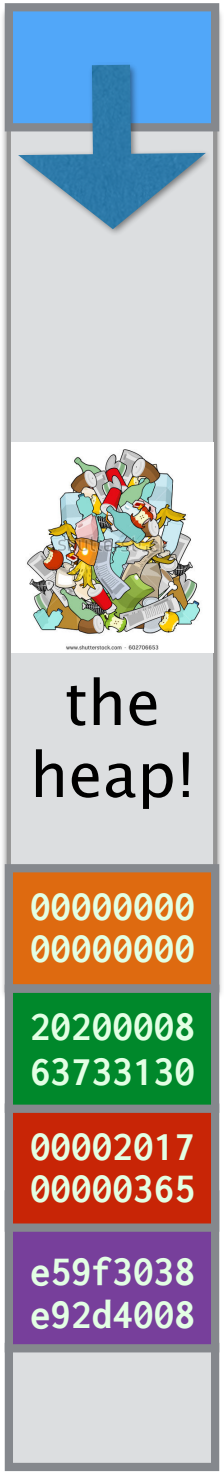
Memory Map



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);
}
```

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



(zeroed data) bss

(read-only data) rodata

data

text

interrupt vectors

Global allocation

+ Convenient, albeit hacky

Global scope, all can access

No encapsulation, hard to track use/dependencies

- Size fixed at declaration, no option to resize

+/- Scope/lifetime is global/whole program

One shared namespace, manually avoid conflicts

Stack allocation

+ **Efficient**
  Fast to allocate/deallocate, ok to oversize
+ **Convenient**
  Automatic alloc/dealloc on function entry/exit
  Can assign with static initializer
+ **Reasonable type safety**

- **Size fixed at declaration, no option to resize**
  Cannot reassign array — there is no pointer!
+/- **Scope/lifetime dictated by control flow**

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

- **Lots of opportunity for error**
  Low type safety
- **Heap memory errors**
  (allocate wrong size, use after free, double free)
- **Leaks** (much less critical)

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

Cannot do pointer arithmetic

Cannot use array indexing (both arithmetic & dereference!)

Why do we need a help?

Let's see an example!

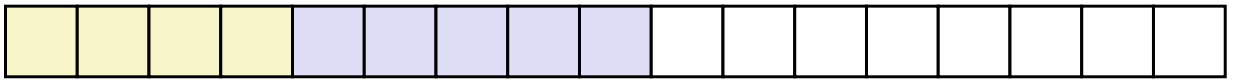
code/heap.c

How is a heap implemented?

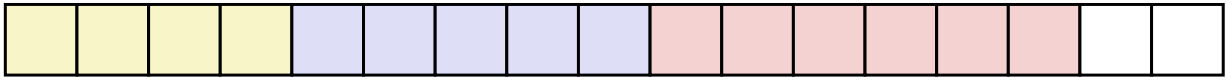
Tracing the heap

```
p1 = malloc(4)
```

```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```

```
p4 = malloc(2)
```

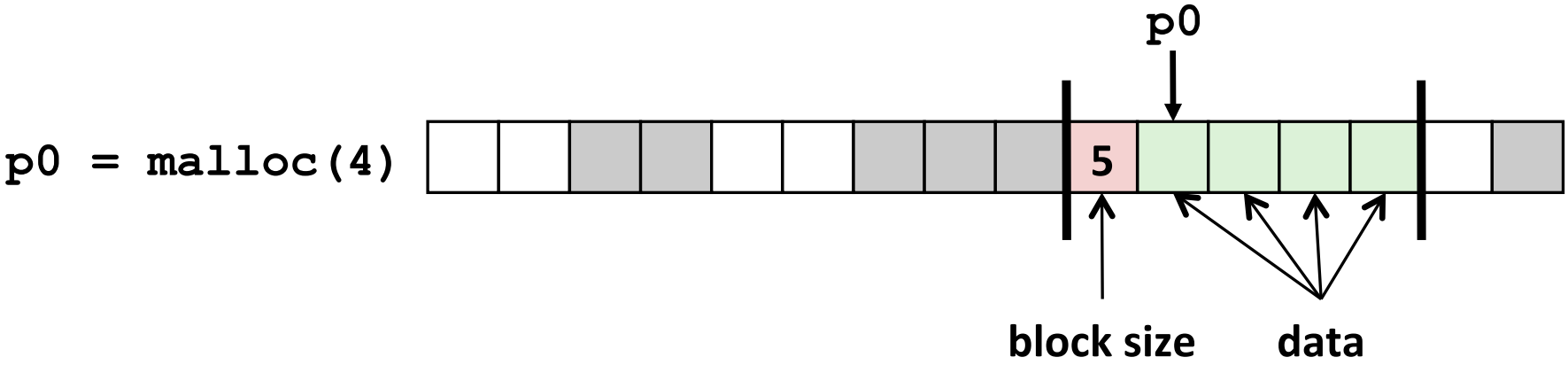
How are we going to do that??

Bump Memory Allocator

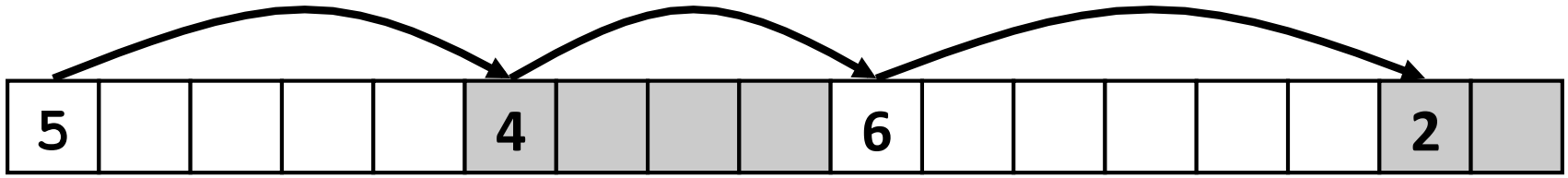
`malloc.c`

Now: track inuse, recycle memory

Standard technique is to track status (length and free/ inuse) about each block in the word preceding the payload data.



The heap now has an implicit list to walk all the blocks



Heap implementation design

just **`malloc`** is easy 🙂

`malloc` with **`free`** is hard 🙁

- **`free`** recycles blocks to service later requests
- **`malloc`** search for existing unused block. Which block to choose (best-fit, first-fit)?
- **`malloc`** may need only some of the block and may split it. Splitting blocks causes fragmentation.
- **`free`** can coalesce with neighboring free blocks to reduce fragmentation

Points to ponder

What happens if you forget to free a pointer after you are done using it?

Can you refer to a pointer after it has been freed?

What is stored in the memory that you malloc?

What if you free a pointer that you didn't malloc?

Can you free the same pointer twice?

Wouldn't it be nice to not have to worry about freeing memory?