Admin

- Assign 2
  Functionality test results out soon
  Revise and resubmit on open issues will be avail
- Assign 3
  printf perseverance and pride!
- Lab4
  stack, heap, build process

Today: Thanks for the memory!

Linker script `memmap.ld`

Loading, how an executable file becomes a running program

Address space layout, organization of memory

Runtime stack, stack frames, stack allocation

Heap allocation, malloc and free

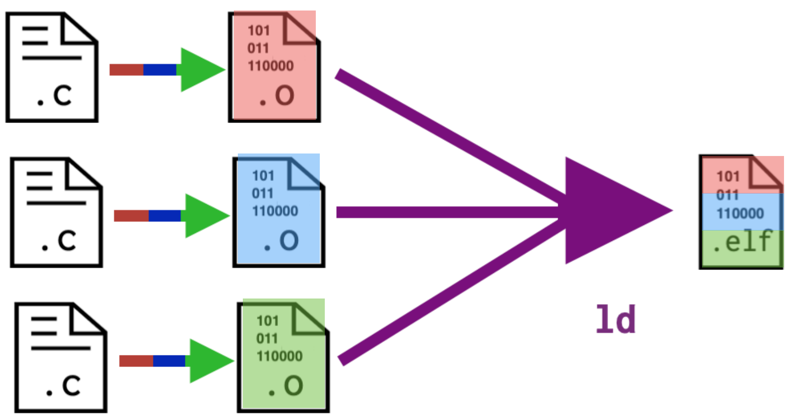
Linking

Combines one or more object files into executable

Resolve inter-module references

Consolidate code, data sections across all modules

Arrange sections in output file in proper order




```
0000000040000000 T _start
0000000040000008 t hang                            start.o
000000004000000c t blink
0000000040000050 T main                            clock.o
0000000040000150 T _cstart                        cstart.o
00000000400001a8 T gpio_init
00000000400001ac T gpio_set_input
00000000400001b0 T gpio_set_output
00000000400001b4 T gpio_set_function
00000000400001b8 T gpio_get_function
00000000400001c0 T gpio_write
00000000400001c4 T gpio_read                        gpio.o
00000000400001cc T timer_get_ticks
00000000400001d4 T timer_init
00000000400001d8 T timer_delay_us
0000000040000210 T timer_delay
0000000040000234 T timer_delay_ms                  timer.o
0000000040000288 R __bss_end
0000000040000288 R __bss_start
```

`clock.o timer.o gpio.o start.o cstart.o -> clock.elf`

`memmap.ld` (linker script)

```
SECTIONS
{
    .text 0x40000000  :  { *(.text.start) *(.text*) }
    .rodata           :  { *(.rodata*) }
    .data             :  { *(.sdata*) }
    __bss_start       = .;
    .bss              :  { *(.bss*) }
    __bss_end         = .;
}
```

What is special about `.text.start` and why must it go first?

What is the significance of `0x40000000`?

Program start sequence

How is a program loaded into memory?

What must happens to start executing a program?

What is known about state of registers or contents of memory?

Does execution actually start at **`main()`**

👀 **Read our files!** 👀

[`$CS107E/src/start.s`]

[`$CS107E/src/cstart.c`]

[`$CS107E/lib/memmap.ld`]

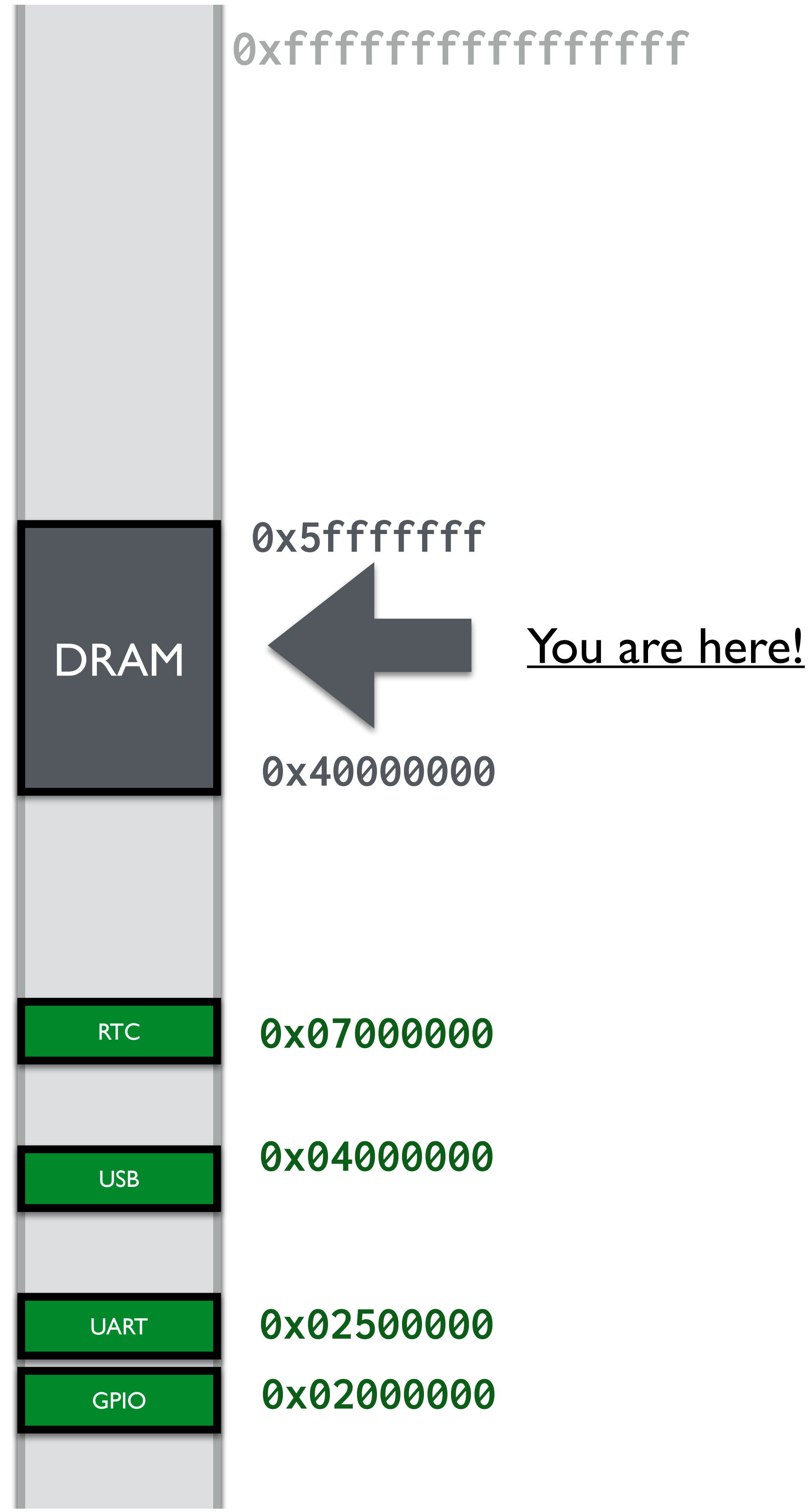
Memory map

64-bit address space

`0x0 - 0xffffffffffffffff`

512MB physical RAM at

`0x40000000-0x5fffffff`

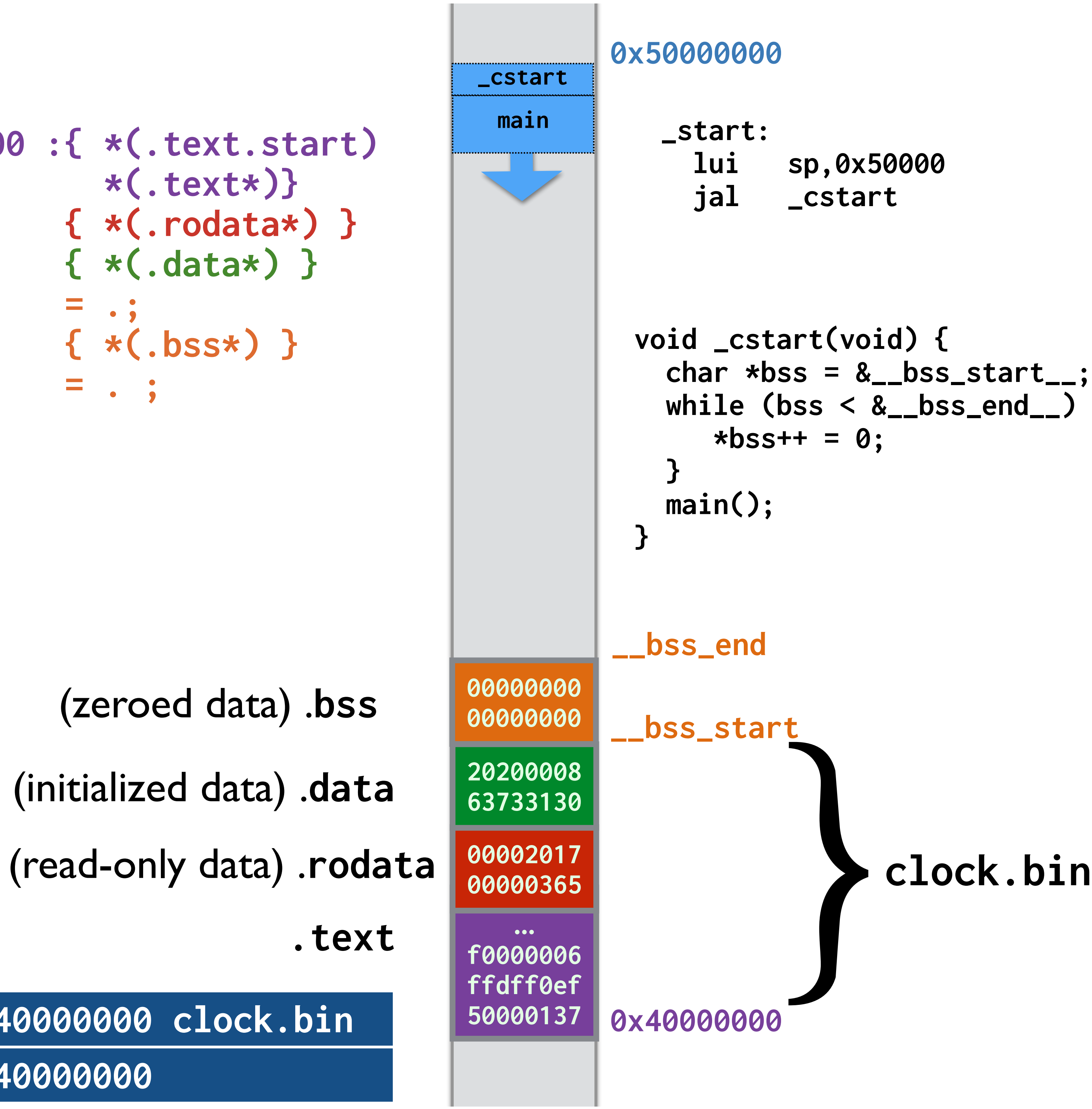


Ref: **D1-H User Manual p.45**

```
SECTIONS
{
   .text 0x40000000 :{ *(.text.start)
                       *(.text*)}
   .rodata :          { *(.rodata*) }
   .data :            { *(.data*) }
   __bss_start        = .;
   .bss :             { *(.bss*) }
   __bss_end          = . ;
}
```

```
_start:
   lui    sp,0x50000
   jal    _cstart
```

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




```
$ xfel write 0x40000000 clock.bin
$ xfel exec  0x40000000
```

Let’s use gdb to watch in action

```
Breakpoint 2, sqr (v=v@entry=3) at program.c:5
5       int sqr(int v) {
(gdb) bt
#0  sqr (v=v@entry=3) at program.c:5
#1  0x0000000040000094 in delta (a=a@entry=3, b=b@entry=7) at program.c:10
#2  0x00000000400000d8 in main () at program.c:15
#3  0x0000000040000048 in _cstart () at cstart.c:7
#4  0x0000000040000010 in _start_gdb ()
(gdb) step
6           return v * v;
(gdb) p v
$1 = 3
(gdb)
```

Some useful gdb commands to learn

```
run, step, next, continue
print, x, display
list, disassemble, info registers
breakpoint
backtrace, info frame, up/down
```

Also repeat, tab completion, history

👀 **Read our course guide** 👀

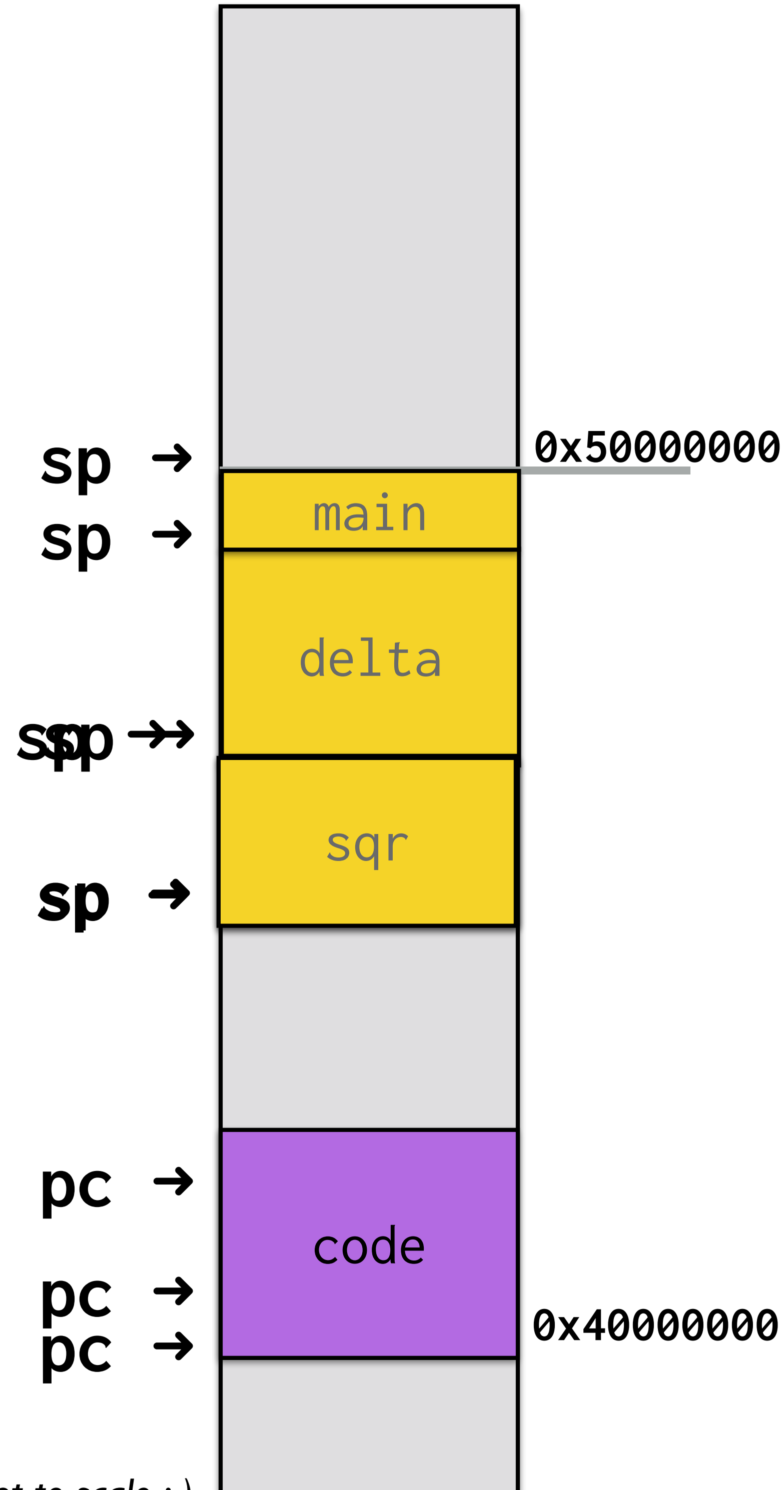
http://cs107e.github.io/guides/gdb/

```
// start.s
lui    sp,0x5000
jal    ra,main
```

```
void main(void)
{
    delta(3, 7);
}

int delta(int a, int b)
{
   int diff = sqr(a) - sqr(b);
   return diff;
}

int sqr(int v)
{
   return v * v;
}
```



Memory diagram not to scale :-)

Frame pointer

Designate register `s0` for use as **frame pointer** `fp`

Stack pointer `sp` is top of stack, additional register `fp` marks boundary of current stack frame

Each frame saves previous `fp` — this gives reliable way to backtrace entire stack

CFLAGS to enable: `-f-no-omit-frame-pointer`

Add instructions to prolog/epilog that set up `fp` on entry and restore saved on exit

Tracing stack frames

Prolog

Adjust stack pointer to make space (16 + `rest)`

Store ra and fp to stack

Set `fp` to where sp was (end of prev stack frame)

Body

`fp` stays anchored

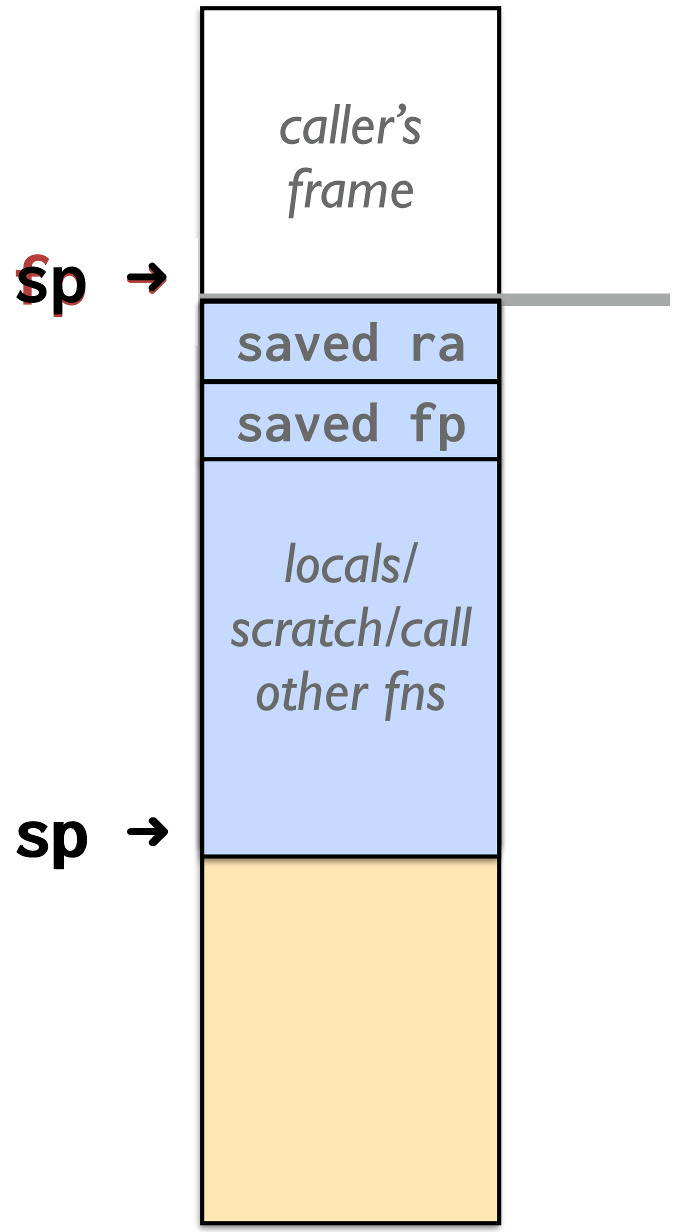
access data on stack `fp`-relative

fp offsets remain stable (even if **sp** changes)

Epilog

Restore ra, fp to saved values

Adjust sp to remove frame



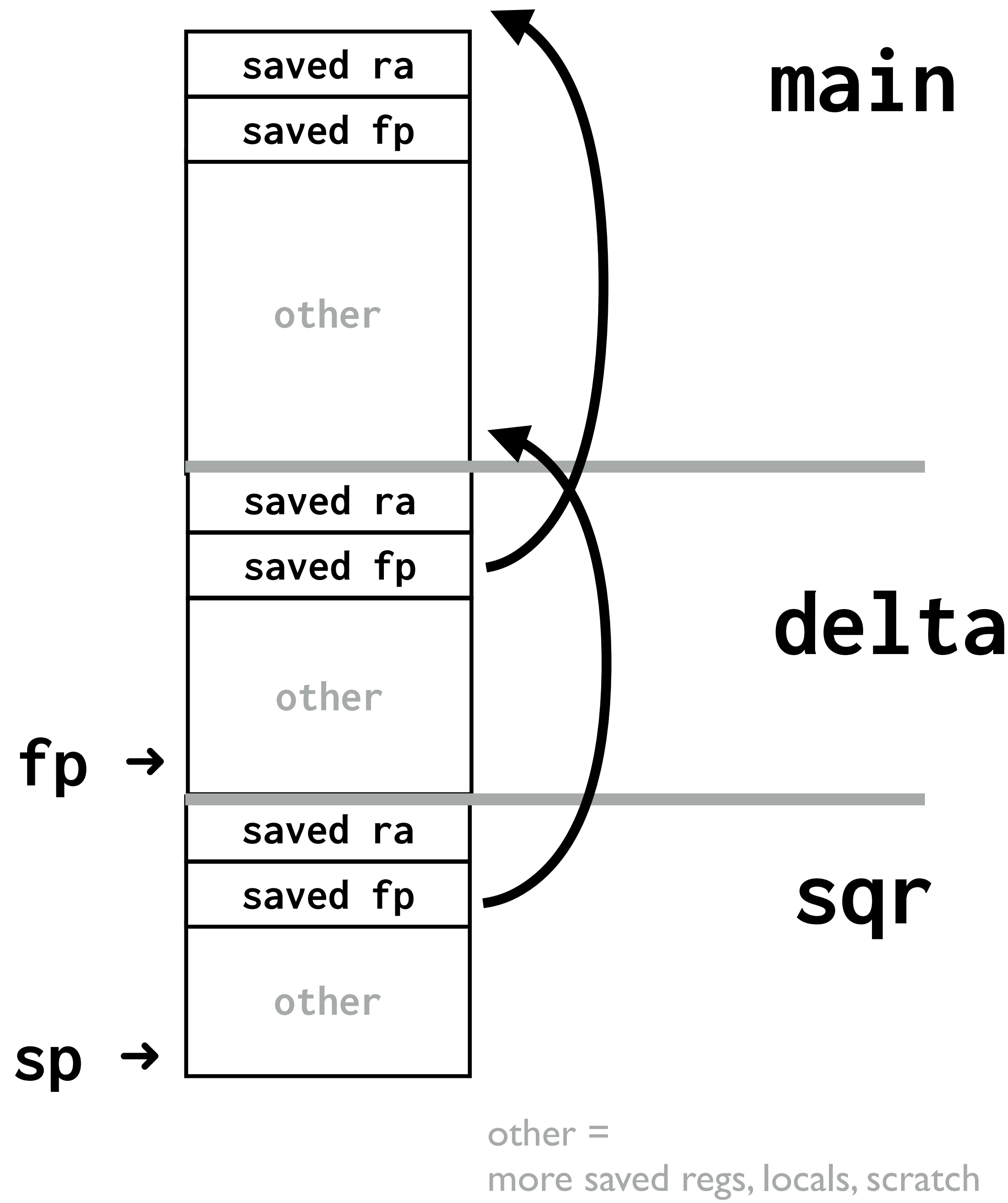
Linked chain of frame pointers

Can traverse from innermost frame (**sqr**) to frame of its caller (**delta**), from there to its caller (**main**) ...

```
// start.s

// init fp = 0 as termination
li fp,0
lui sp,0x50000
jal _cstart
```

Deep dive into full frame coming up in this week's lab!



Frame pointer tradeoffs

+ Anchored `fp`, offsets are constant
+ Standard frame layout enables runtime introspection
+ Backtrace for debugging, instrumentation

- `fp` register removed from general pool
- Add'l 3 instructions in prolog,epilog to manage `fp`
- Adds 8 bytes to frame size, another saved register

Aside: "Fedora's tempest in a stack frame"

https://developers.redhat.com/articles/2023/07/31/frame-pointers-untangling-unwinding

https://fedoraproject.org/wiki/Changes/fno-omit-frame-pointer https://lwn.net/Articles/919940/

We have global storage ...

+ **Convenient**
  Fixed location, shared across entire program
  No explicit allocate/deallocate

+ **Fairly efficient, plentiful**
  (But cost to send over serial line to bootloader)

+/- **Scope and lifetime is global**
  No encapsulation, hard to track use/dependencies
  One shared namespace, possibility of conflicts
  Frowned upon stylistically

... and we have stack storage ...

+ **Convenient**

Automatic alloc/dealloc on function entry/exit

+ **Efficient, fairly plentiful**

(But finite size limit on total stack usage)

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

Private to stack frame

Does not persist after function exits

Why do we also need a heap?

An example:

`code/heap/names.c`

Dynamic storage

+ Programmer controls scope/lifetime

Versatile, precise

Works for situations where global/stack do not

- Needs software runtime support

Library module to manage heap memory

Functions to allocate/deallocate memory (explicit calls by client)

- C version is low on safety

No type safety (raw `void*`, size in number of bytes)

Much opportunity for error

(allocate wrong size, use after free, double free)

Heap module interface

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

What is a `void*` pointer?

"Generic" pointer, holds a memory address

Type of pointee is not specified, could be any type of data

What you can do with a `void*`

Pass to/from function, assignment

What you cannot do with a `void*`

No dereference without cast

No pointer arithmetic without cast (scaling unknown!)

No array indexing (size of pointee unknown!)

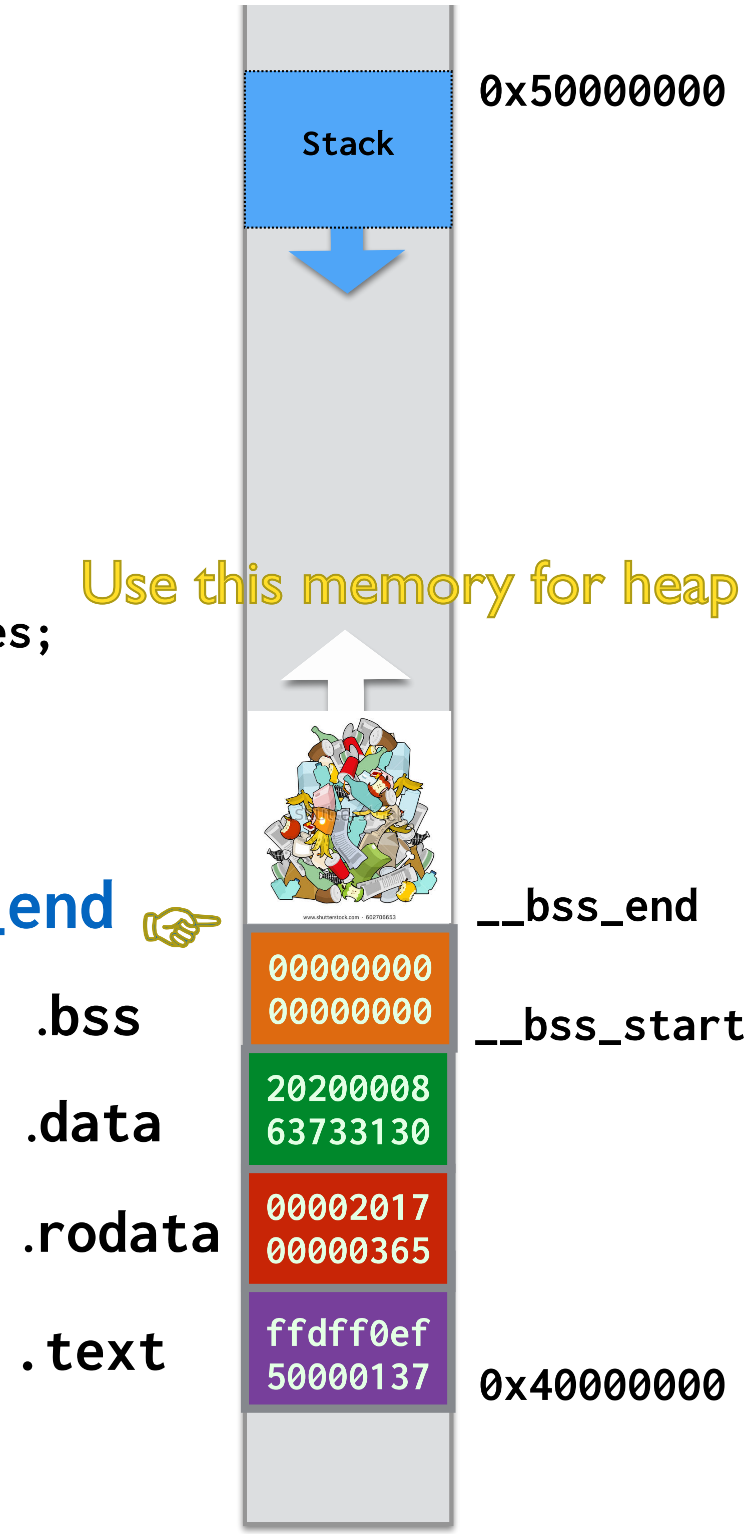
How to implement a heap

Drawing by Jane Lange

Bump allocator

```
void *sbrk(int nbytes)
{
    static void *_cur_heap_end = &__bss_end;

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



- Keep pointer to end of heap segment
- Service malloc request adjusts pointer upward
- Every request extends/grows heap segment
- No reuse/recycle

Tracing the bump allocator

Each square represents 8 bytes of space

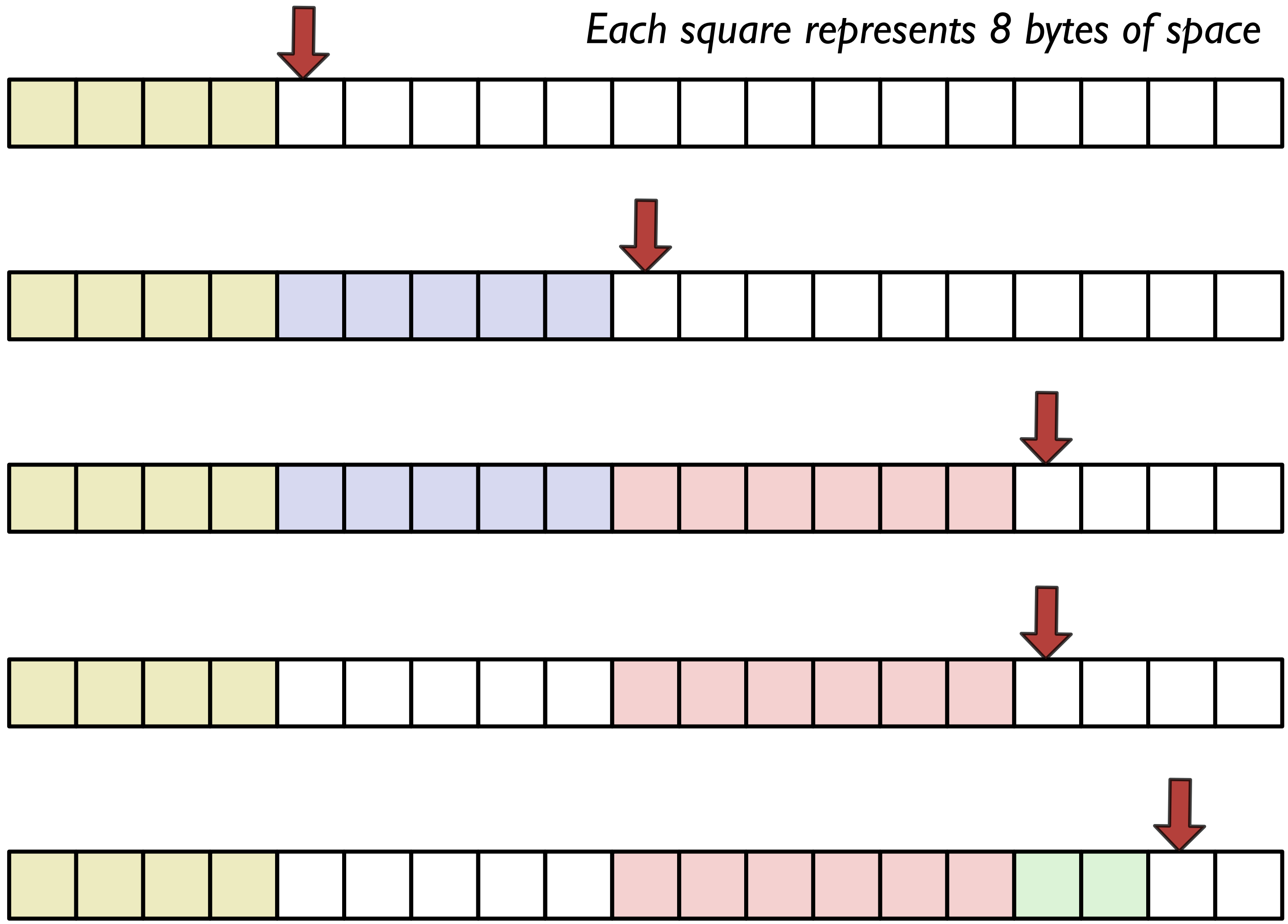
`p1 = malloc(32)`

`p2 = malloc(40)`

`p3 = malloc(48)`

`free(p2)`

`p4 = malloc(16)`



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

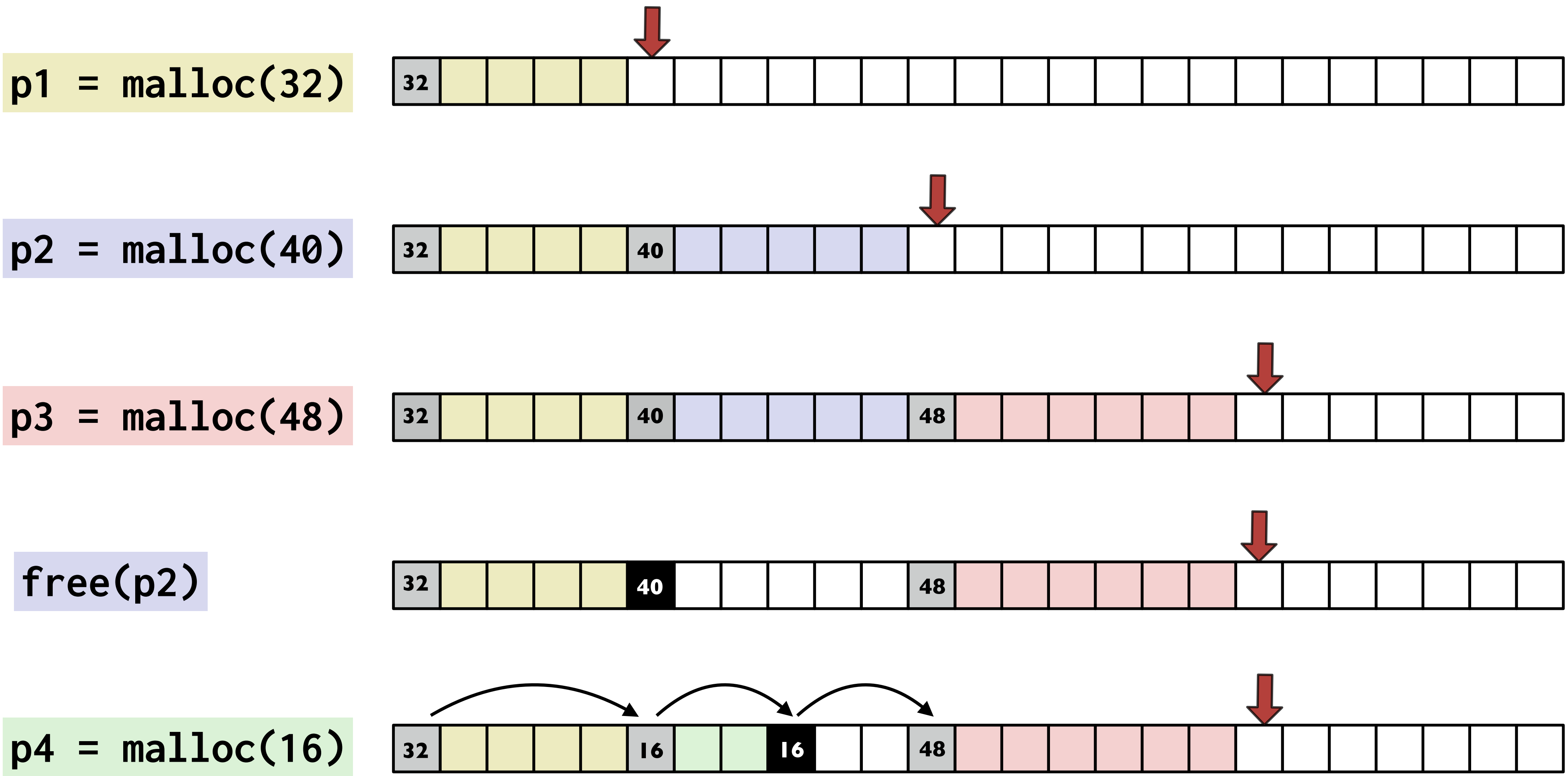
    (in what situations will this be problematic?)

- Sad consequences if **`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



Header struct on each block

```
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
}
```

Challenge 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?

Challenge for malloc implementor

just `malloc` is easy 😎
`malloc` with `free` is hard 🤔
Efficient `malloc` with `free` ….Yikes! 🥵

Complex code (pointer math, typecasts)
Critical system component
correctness is non-negotiable!
Thorough testing is essential

Survival strategies:
draw lots of pictures
printf (you've earned it!!)
early tests on inputs small enough to trace by hand if need be
build up to more complex tests