

Admin

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

Today: Thanks for the memory!

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

Let's use gdb to watch in action

```
Breakpoint 2, sqr (v=v@entry=3) at program.c:7
7      int sqr(int v) {
(gdb) bt
#0  sqr (v=v@entry=3) at program.c:7
#1  0x000000040000094 in delta (a=a@entry=3, b=b@entry=7) at program.c:12
#2  0x0000000400000d8 in main () at program.c:17
#3  0x000000040000048 in _cstart () at cstart.c:7
#4  0x000000040000010 in _start ()
(gdb) step
8          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

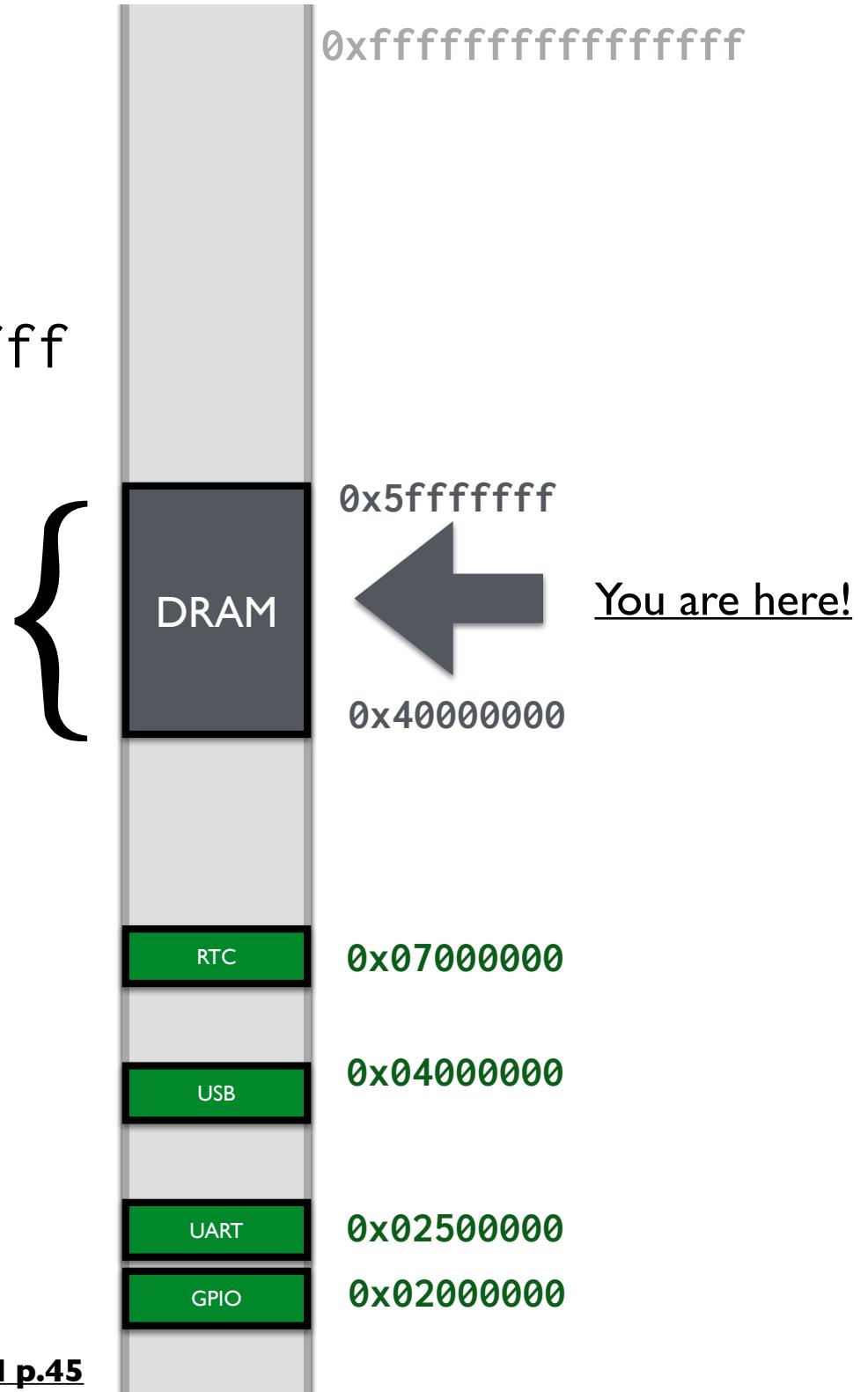
Also repeat, tab completion, history

Memory map

64-bit address space

0x0 - 0xffffffffffff

512MB physical RAM at
0x40000000-0x5fffffff



SECTIONS

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

(zeroed data) .bss

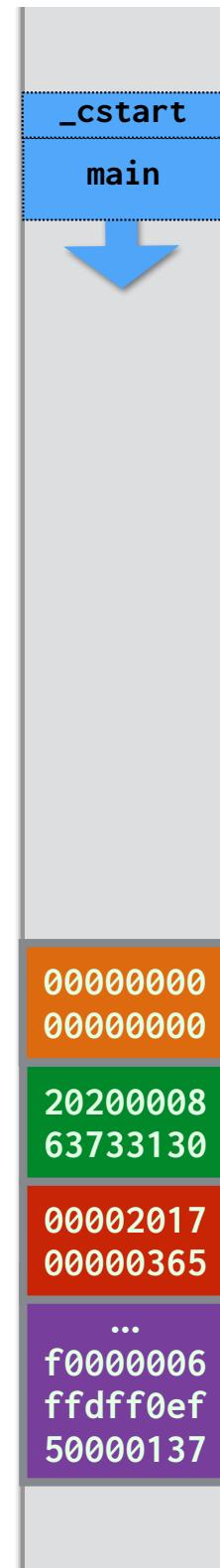
(initialized data) .data

(read-only data) .rodata

.text

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

```
$ xfel exec 0x40000000
```



0x50000000

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

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

__bss_end

__bss_start

} clock.bin

00000000
00000000

20200008
63733130

00002017
00000365

...
f0000006
ffdff0ef
50000137

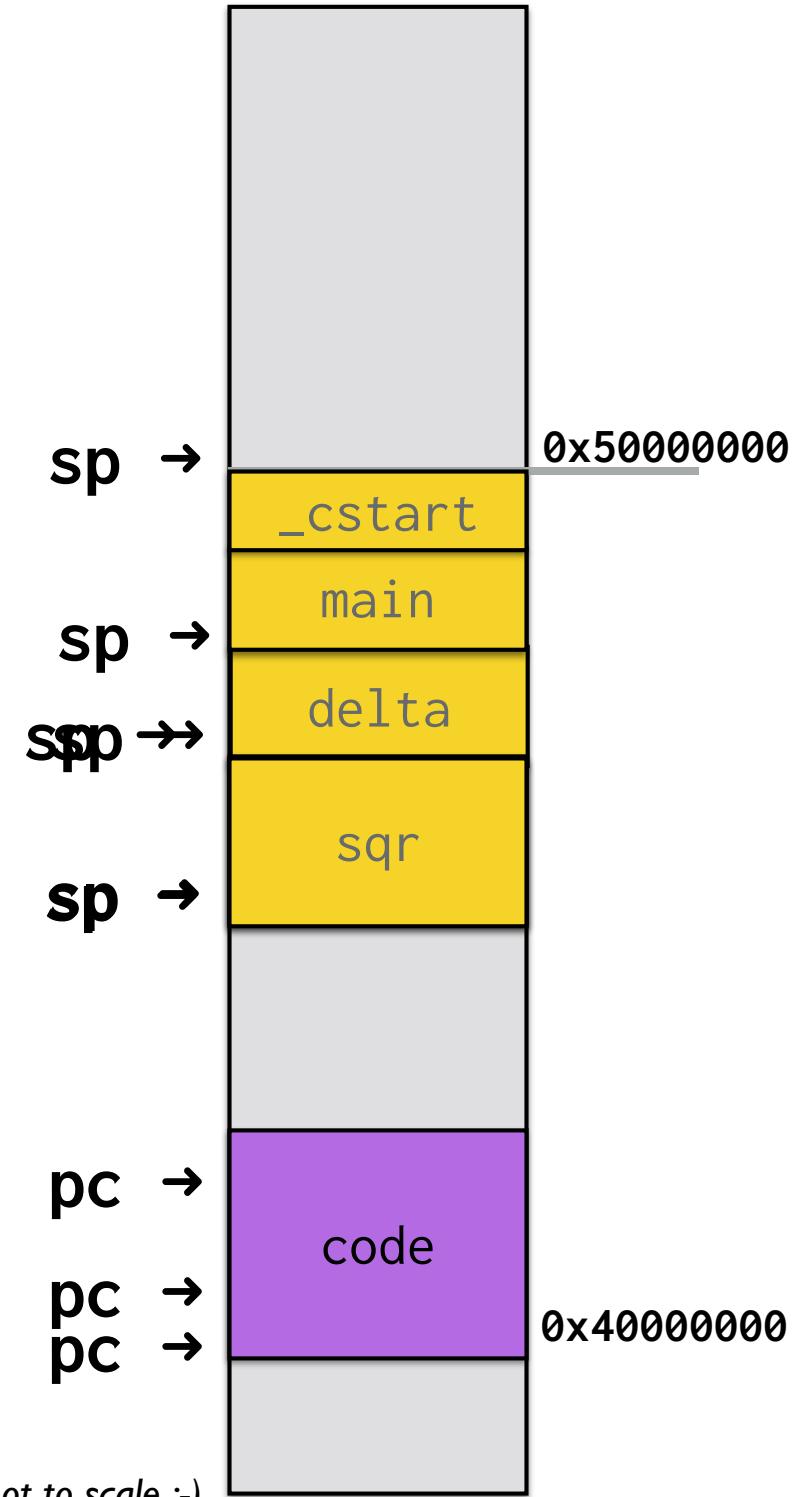
0x40000000

```
// start.s
lui    sp,0x5000
jal    _cstart
```

```
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;
}
```



Frame pointer

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

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

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 bytes + rest)

Save registers **ra** and **fp** on stack, first two slots

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

Body

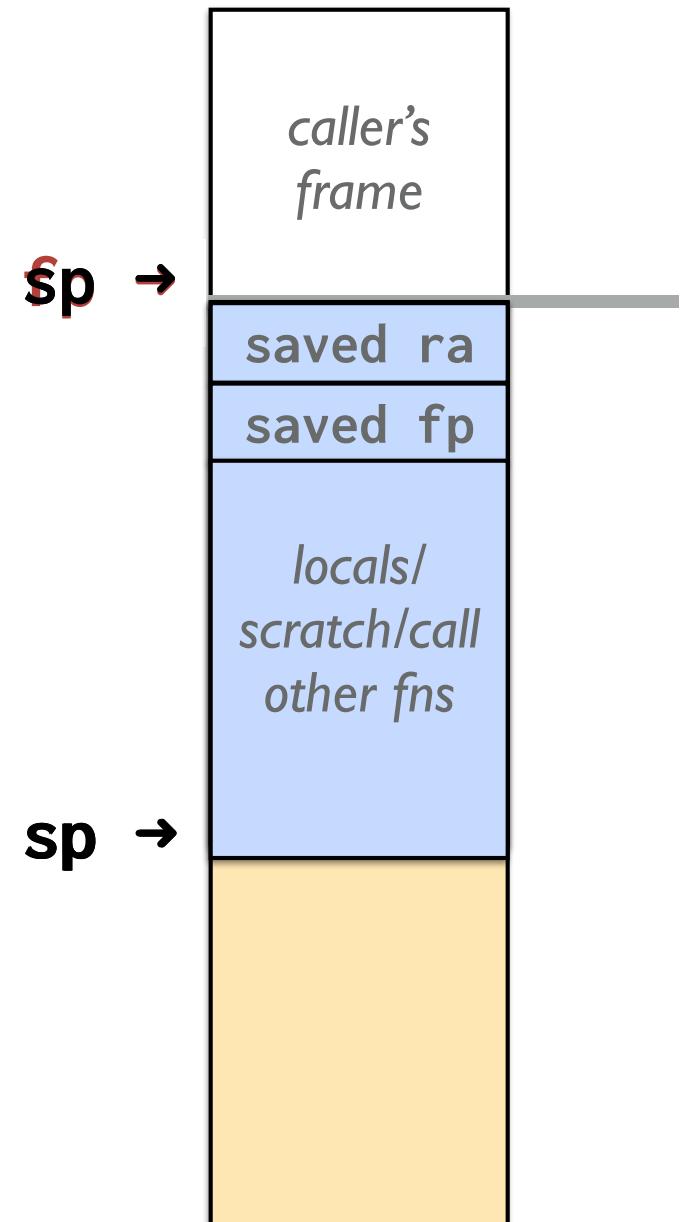
sp and **fp** anchors mark start/end of current frame

access data on stack **sp**-relative

Epilog

Restore **ra** and **fp** to saved values on stack

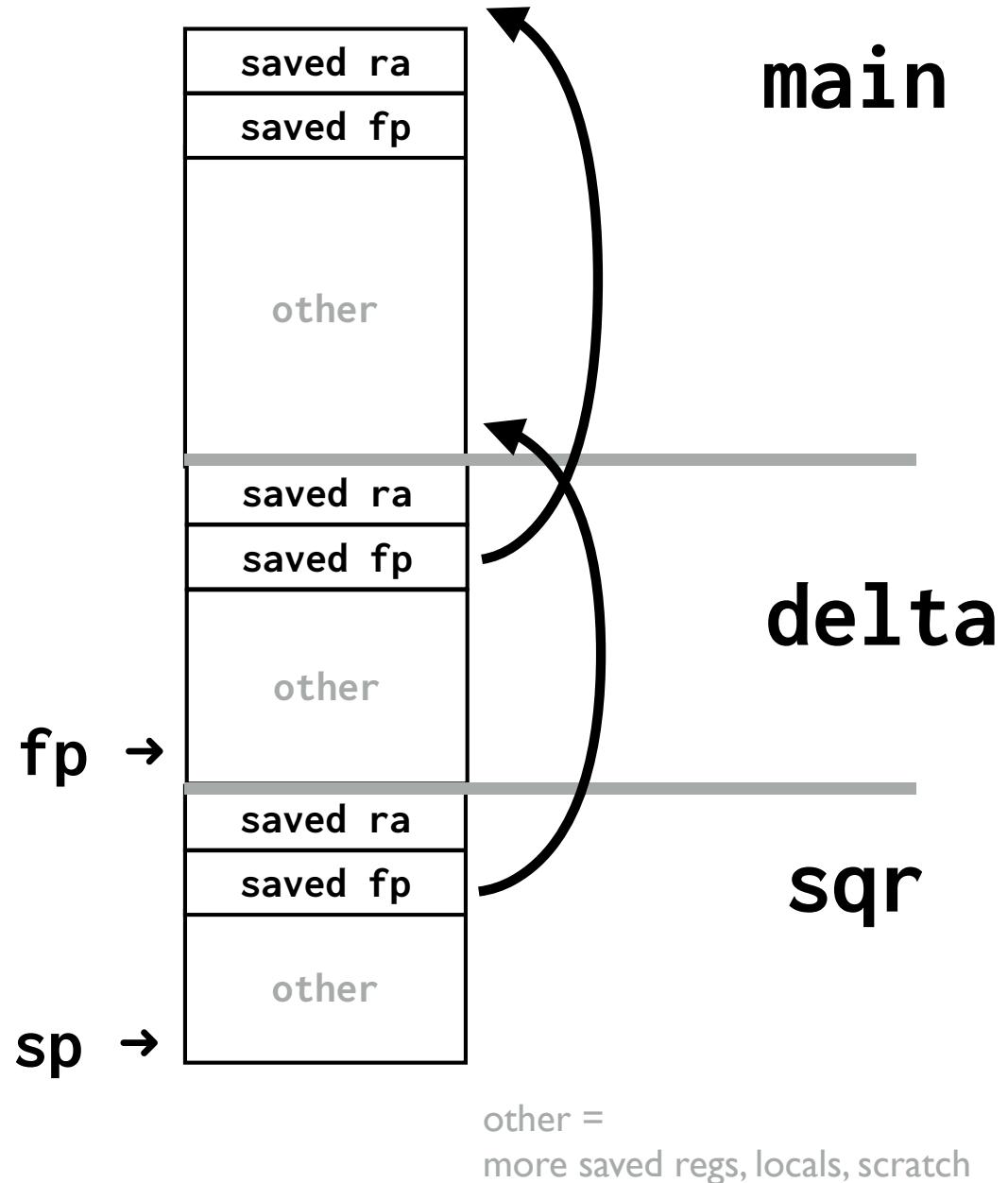
Adjust **sp** to remove frame



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
- Added 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

- Heavily 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

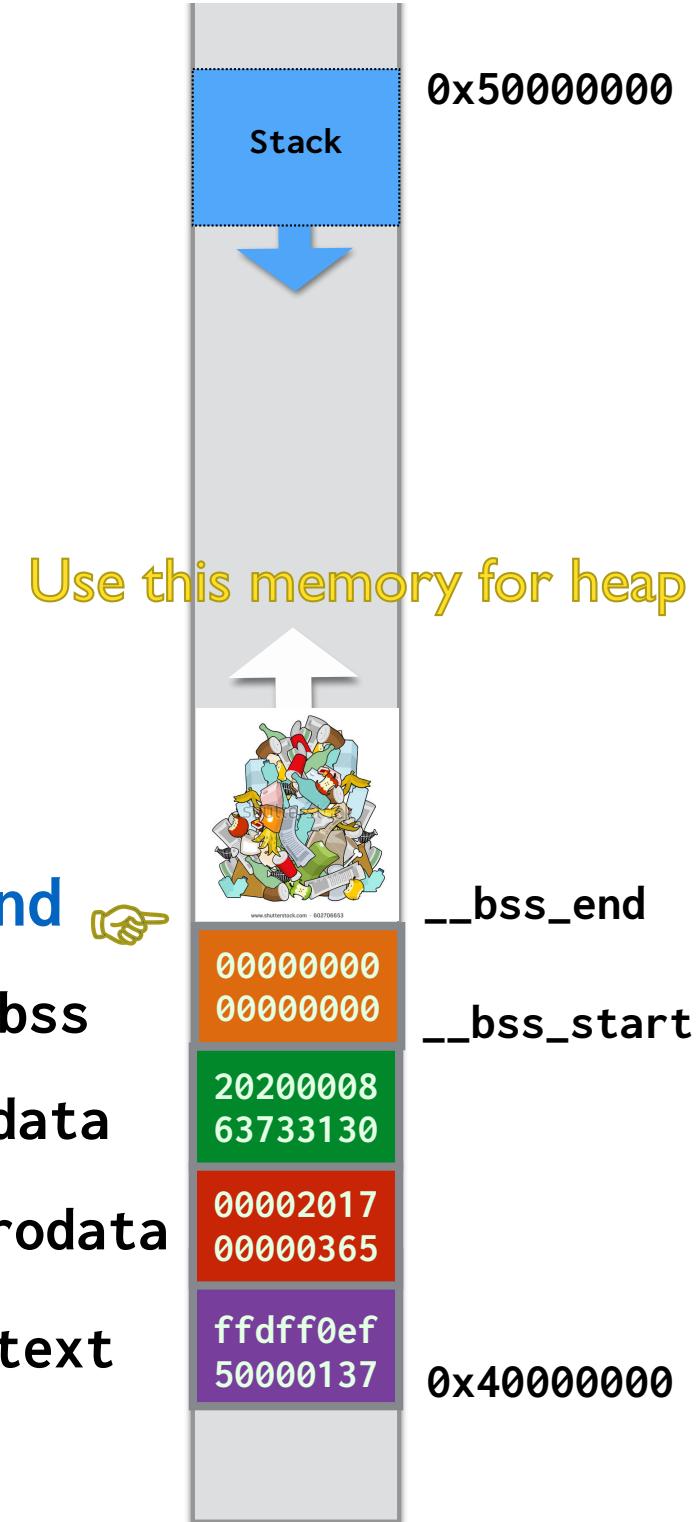
Bump allocator

```
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;
}
```

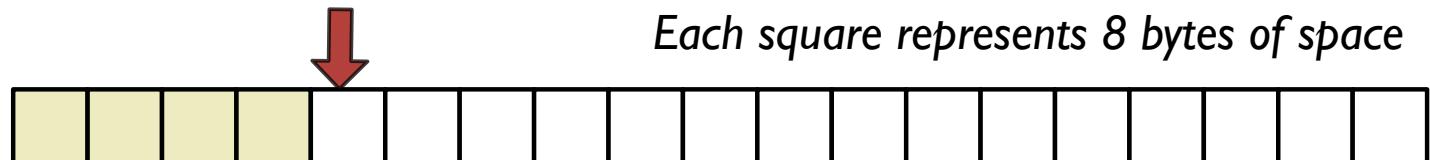
void *cur_heap_end 

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

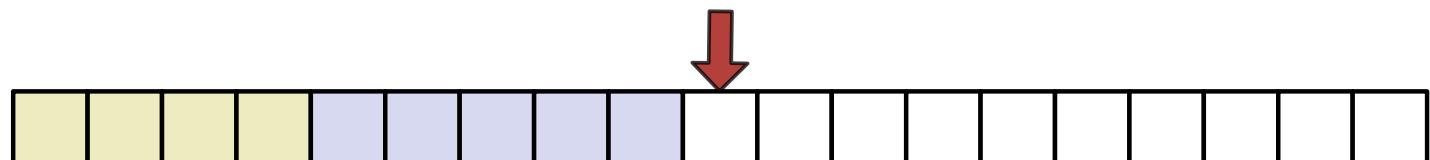


Tracing the bump allocator

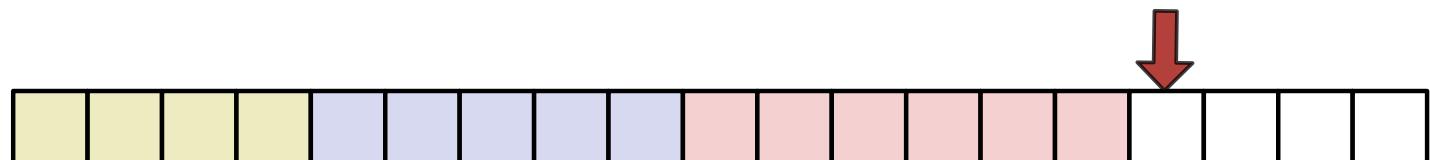
`p1 = malloc(32)`



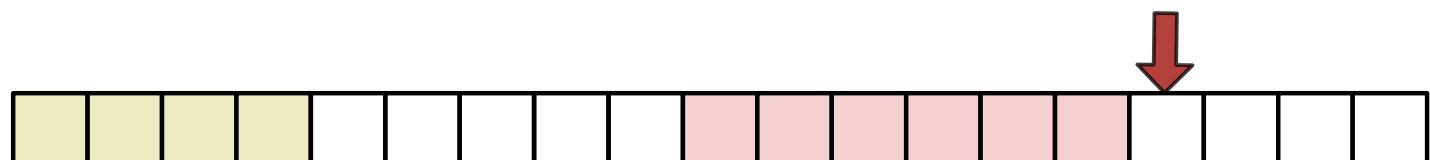
`p2 = malloc(40)`



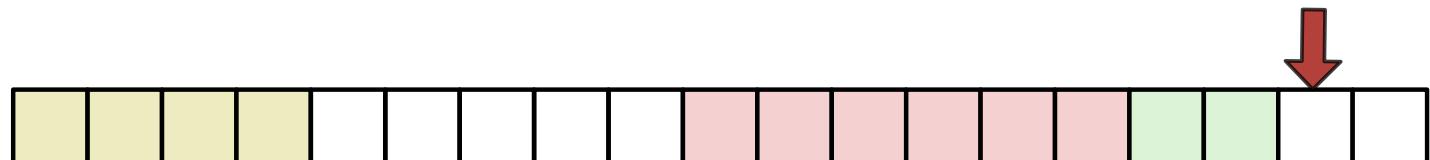
`p3 = malloc(48)`



`free(p2)`



`p4 = malloc(16)`



Bump Memory Allocator

`code/heap/malloc.c`

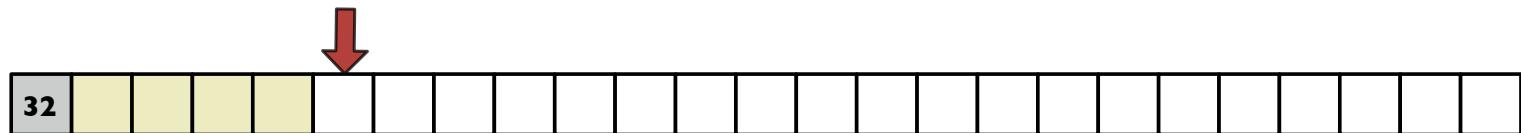
Evaluate bump allocator

- + 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?)

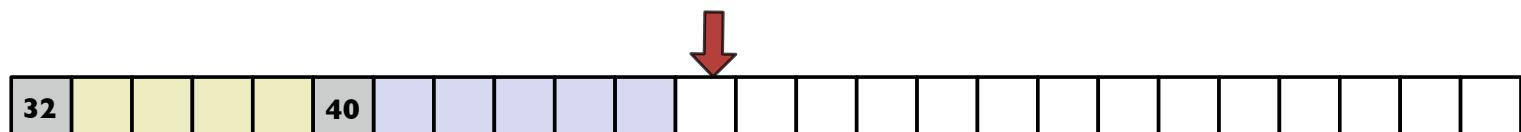
Pre-block header, implicit list

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

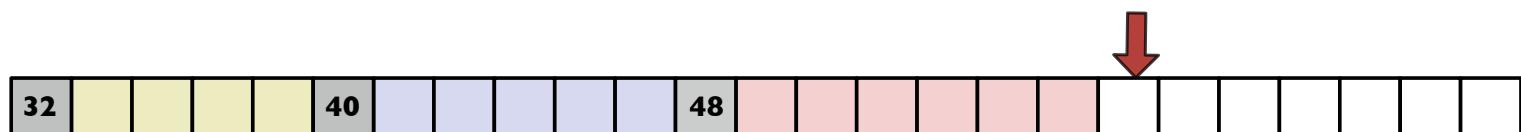
p1 = malloc(32)



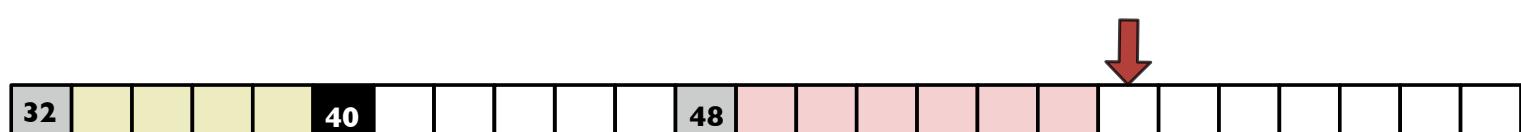
p2 = malloc(40)



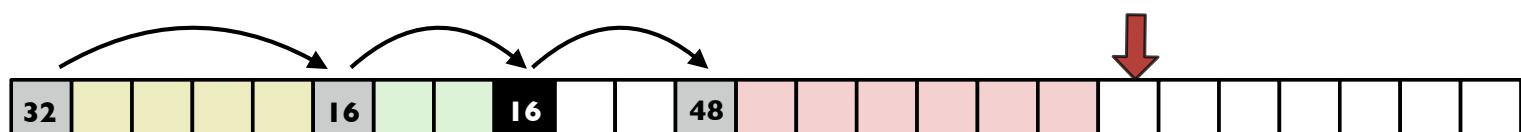
p3 = malloc(48)



free(p2)



p4 = malloc(16)



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