Computer Systems

CS107

Cynthia Lee

Today's Topics

LECTURE:

- Managing the heap
  - Important for your final assignment assign7

Build and runtime components

BUILDING ON PREVIOUS LECTURE…

Build components

PREPROCESSOR:

- Takes #include and #define and other preprocessor directives and replaces them with appropriate text (code.c + .h files → code.i)
  - `gcc -E code.c`

COMPILER:

- Takes processed C code and outputs appropriate IA32 (code.i → code.s)
  - `gcc -S code.c  # uppercase S`

ASSEMBLER:

- Takes assembly output and makes machine output (code.s → code.o)
  - `gcc -c code.c  # lowercase c`

LINKER:

- Takes .o in question, plus other module .o files, joins them together to make the executable (code.o +.o files → a.out or code)
  - `gcc code.c -o code # lowercase o`

Let's think about the output of the **linker**

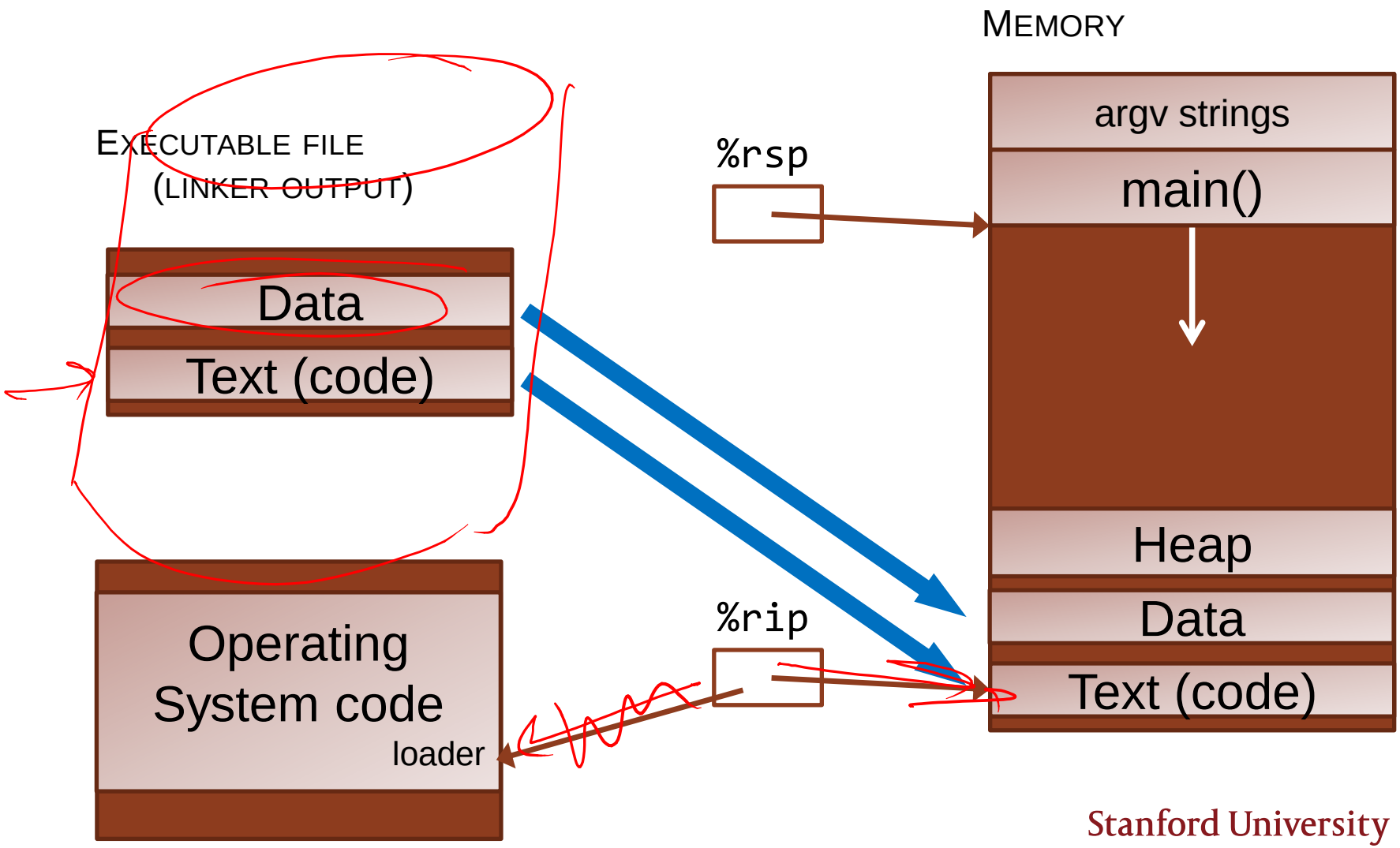
LINKER:

- Takes .o in question, plus other module .o files, joins them together to make the executable (code.o +.o files → a.out or code)
- Output is the executable, what is that?
  - It's raw binary format, so might make sense to look at it in hexadecimal format, but vim takes those raw bytes and interprets them as ASCII, which of course gives nonsense.
  - The contents of the executable are sections of code that are ready to be loaded directly into memory at the specified addresses in order to launch the program.

LOADER: (PART OF THE OS)

- Takes the executable and maps each part (data, code) to memory from the executable file. Also sets up a stack (rbp, rsp), and environment and arguments (argv, argc from the command line). Sets rip to the beginning of main (which of course starts execution).

Linker output loading into memory to begin running



Runtime components

LOADER: (PART OF THE OS)

- Takes the executable and maps each part (data, code) to memory from the executable file. Also sets up a stack (rbp, rsp), and environment and arguments (argv, argc from the command line). Sets rip to the beginning of main (which of course starts execution).

HEAP MANAGER:

- We'll talk about this now!

Heap Manager

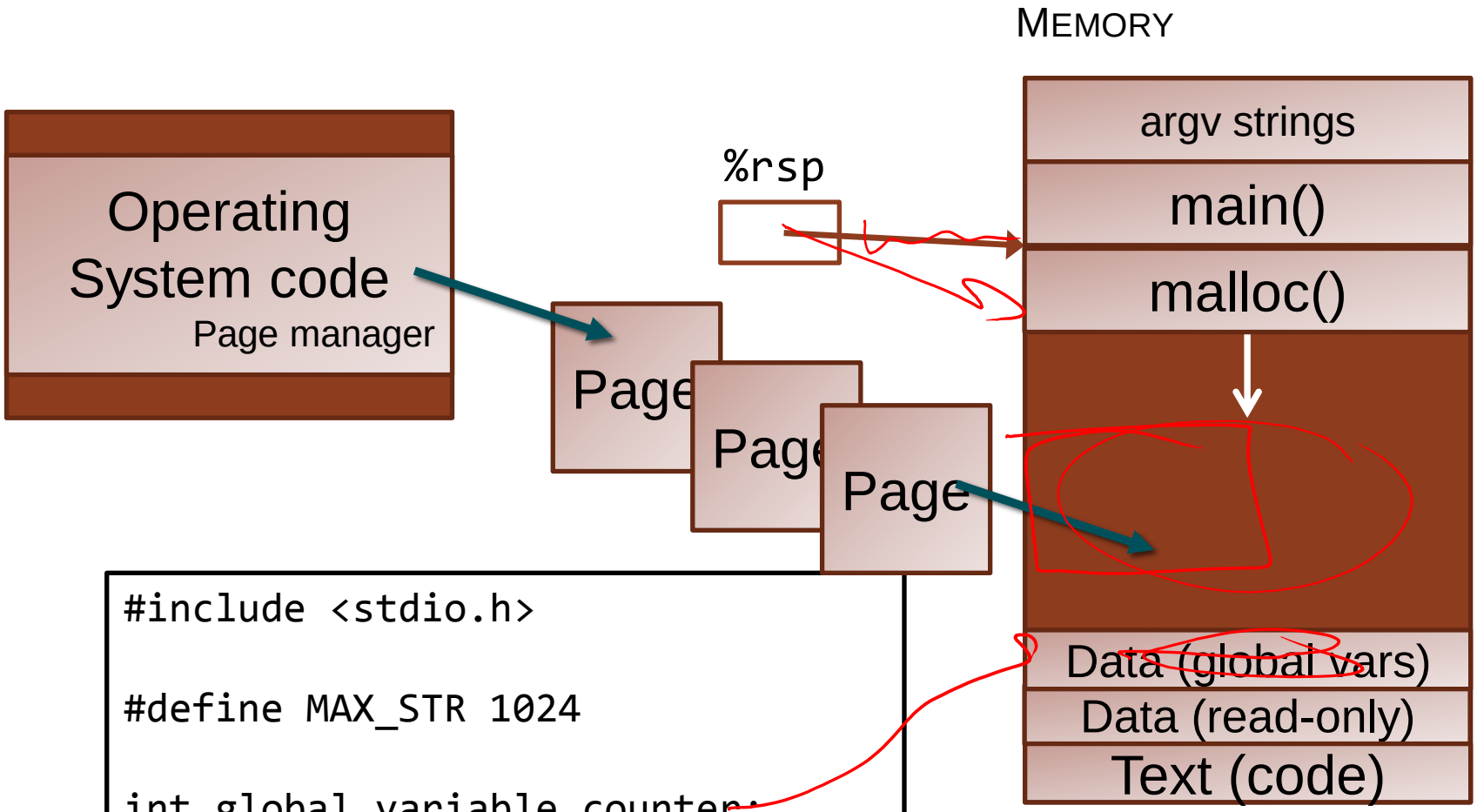
Remember what you know from being a *client* of malloc/free:

The heap manager functions like a hotel front desk—handing out room assignments based on size needs (malloc), and checking people out when they're ready to leave (free).

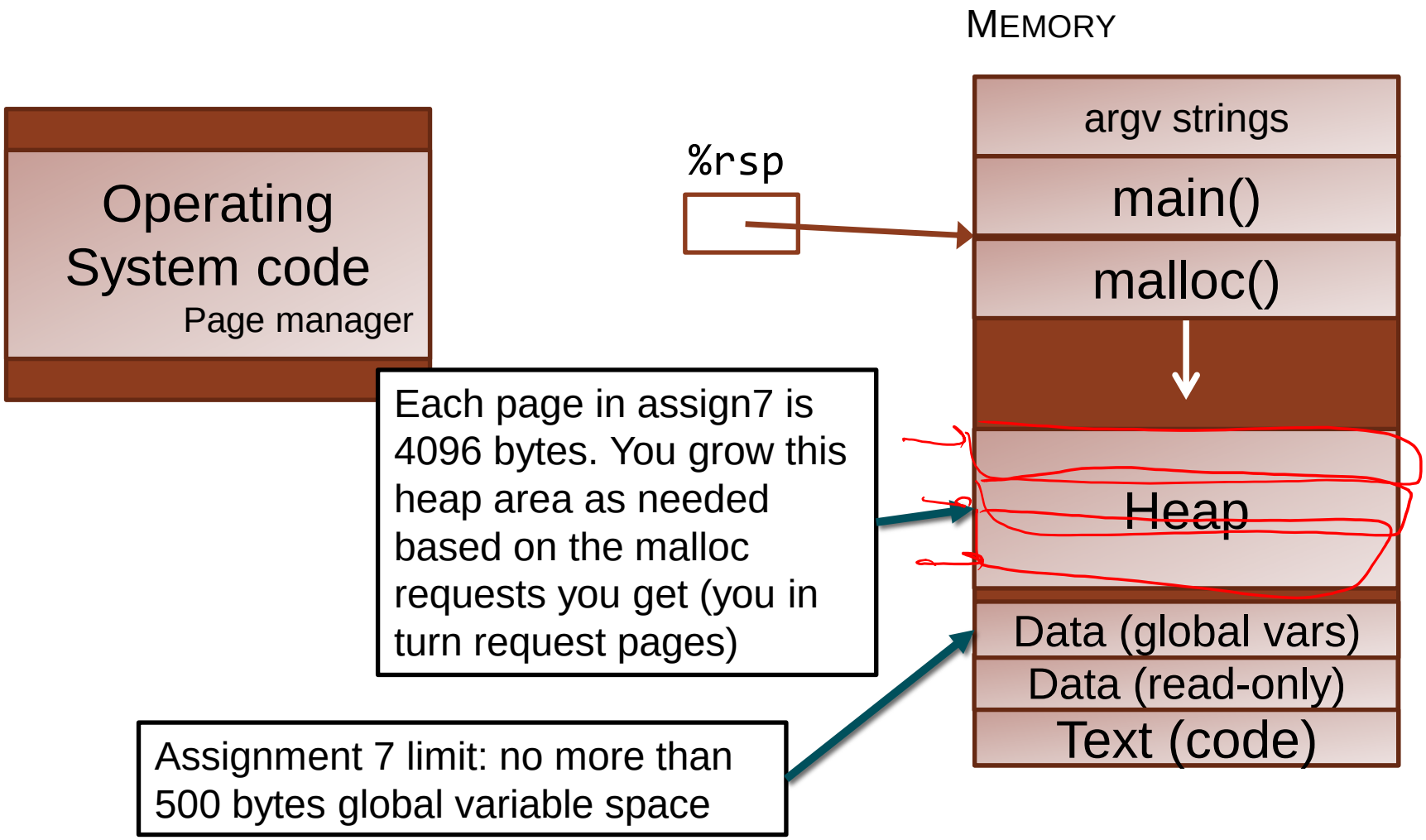
Heap Allocator: you (almost) don't need a spec for this assignment!

- You will implement three functions, whose arguments, return values, and behavior, you already know:
  - `void *malloc(size_t num_bytes_requested)`
  - `void *realloc(void *old_ptr, size_t new_bytes_requested)`
  - `void  free(void *ptr)`
- Other requirements/limits:
  - Maximum request size for a single request is INT_MAX bytes
  - Must be fast (throughput) and make efficient use of space (utilization)
  - No more than 500 bytes of global variables, other storage comes out of the heap memory you would otherwise give to user.

Big picture: what do you have to work with?



Big picture: what do you have to work with?



Heap Manager: Example client code

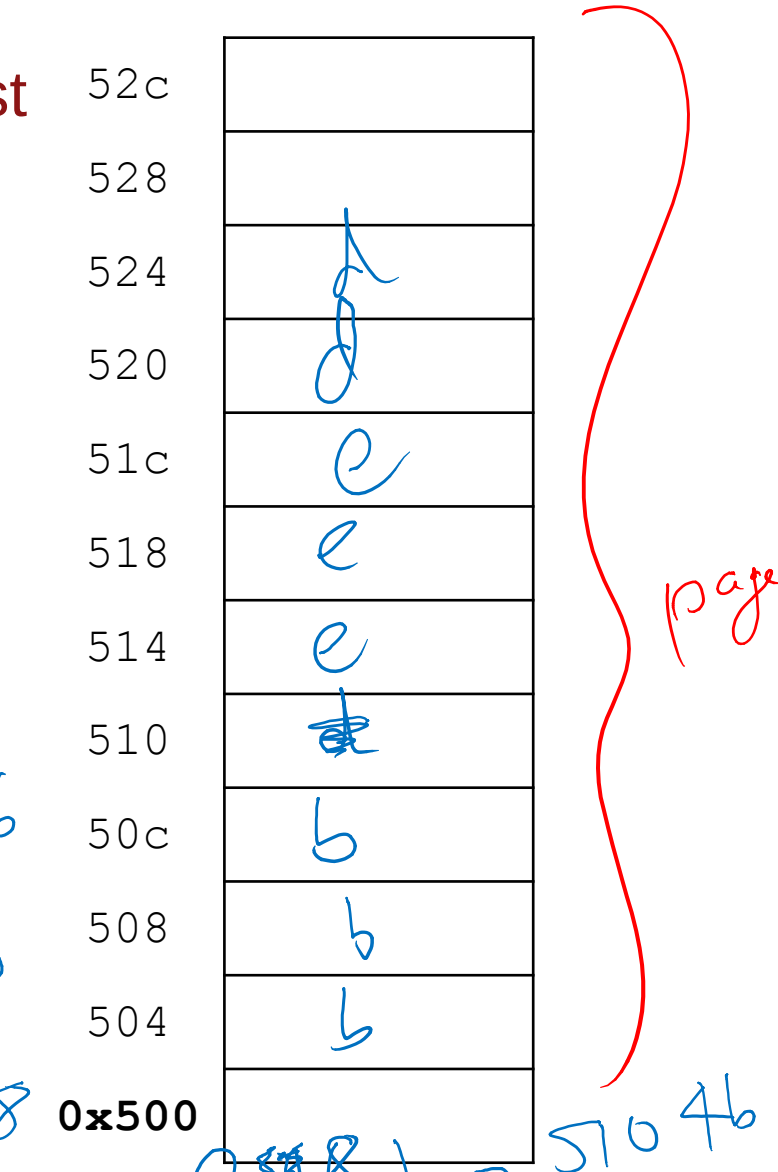
```
void *a, *b, *c, *d, *e, *f;
a = malloc(4);
b = malloc(8);
c = malloc(4);
d = malloc(4);
free(a);
free(c);
e = malloc(12);
b = realloc(b, 12);
d = realloc(d, 8);
f = malloc(12);
```

Implementation #1: In-Use list + Free list

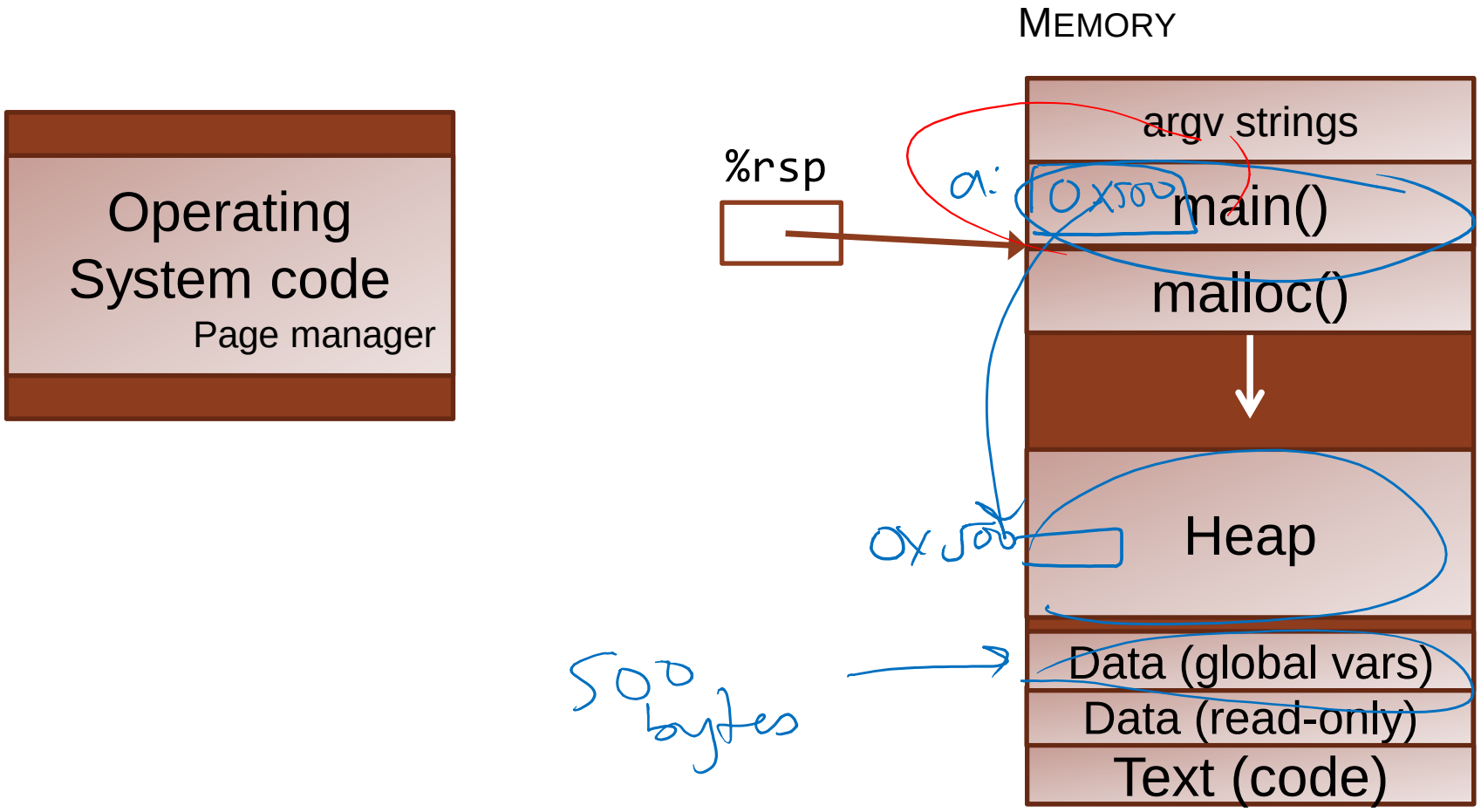
```
void *a, *b, *c, *d, *e, *f;
a = malloc(4);
b = malloc(8);
c = malloc(4);
d = malloc(4);
free(a);
free(c);
e = malloc(12);
b = realloc(b, 12);
d = realloc(d, 8);
f = malloc(12);
```

In-Use:

Free:



Big picture: what do you have to work with?



Discussion question

Could the heap manager pause and do a "defrag" operation on the heap at this point?

- Move data in allocated areas over to coalesce free space into a contiguous block
- Don't change size of any allocated block, and carefully copy data

A. YES, great idea!
B. YES it can be done, but not a good idea for some reason (e.g., not efficient use of time)
C. NO, it can't be done!

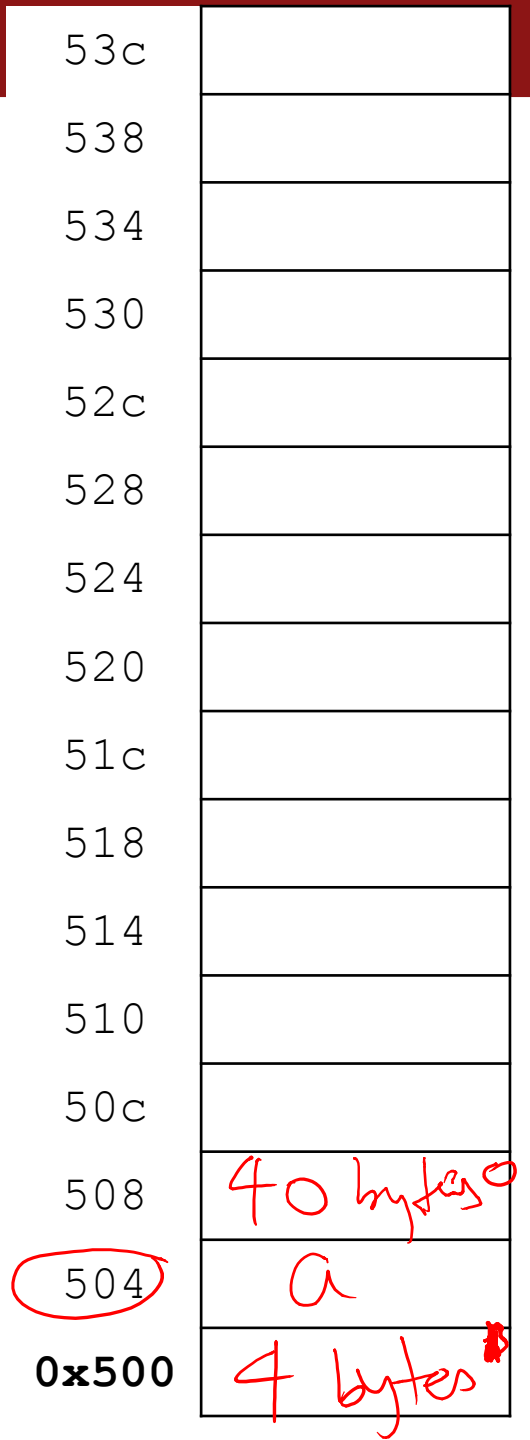
Why or why not?

Implementation #2: Pre-node headers with *implicit* free list

```
void *a, *b, *c, *d, *e, *f;
a = malloc(4);
b = malloc(8);
c = malloc(4);
d = malloc(4);
free(a);
free(c);
e = malloc(12);
b = realloc(b, 12);
d = realloc(d, 8);
f = malloc(12);
```

Discussion question

How do we handle realloc(b, 12)?

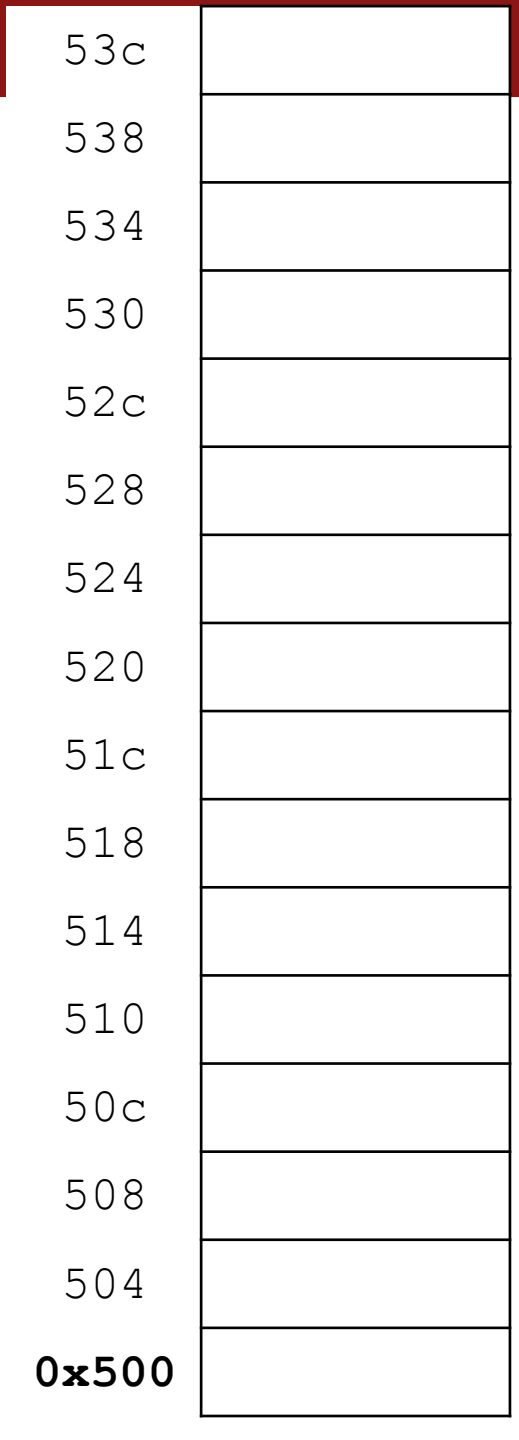


Discussion question

Anything about this approach strike you as a bit…dangerous?

Implementation #3: *Explicit* free list

```
void *a, *b, *c, *d, *e, *f;
a = malloc(4);
b = malloc(8);
c = malloc(4);
d = malloc(4);
free(a);
free(c);
e = malloc(12);
b = realloc(b, 12);
d = realloc(d, 8);
f = malloc(12);
```



Back to the "defrag" idea:

- We can't rearrange pointers at will for currently-allocated memory.
- However, there is nothing to stop us from taking adjacent already-freed blocks and coalescing them into a larger freed block.

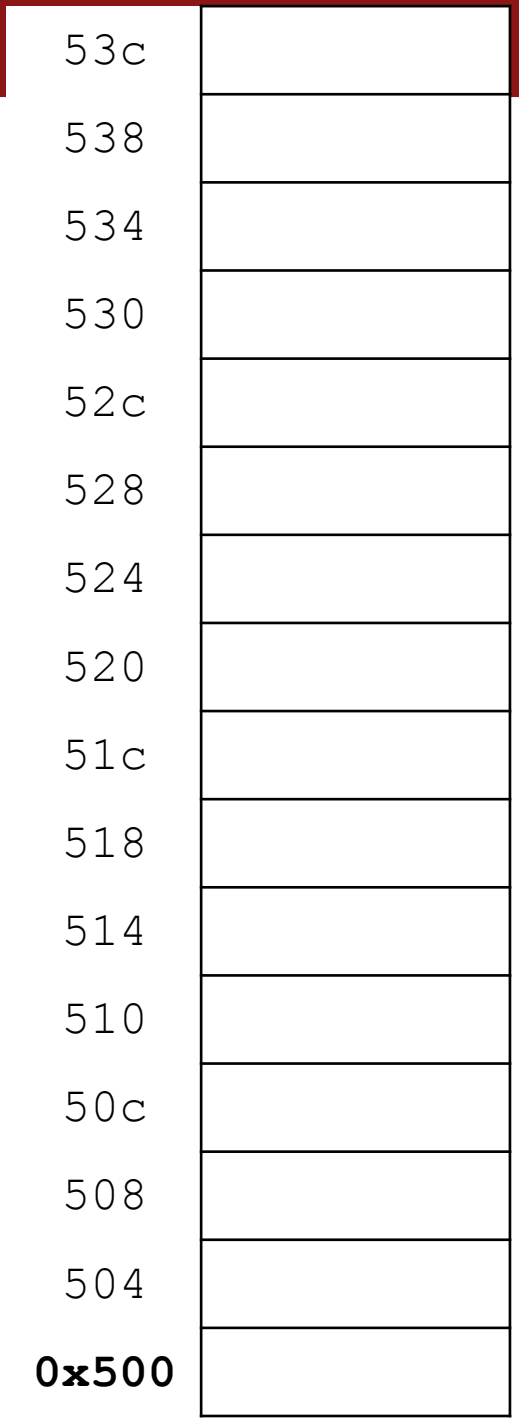
YES, great idea!

The question is, given our free list is a linked list, how do we "notify" incoming pointers to free blocks that the block has now been aggregated into one conglomerate node? Hm….

- See textbook's "footer" solution

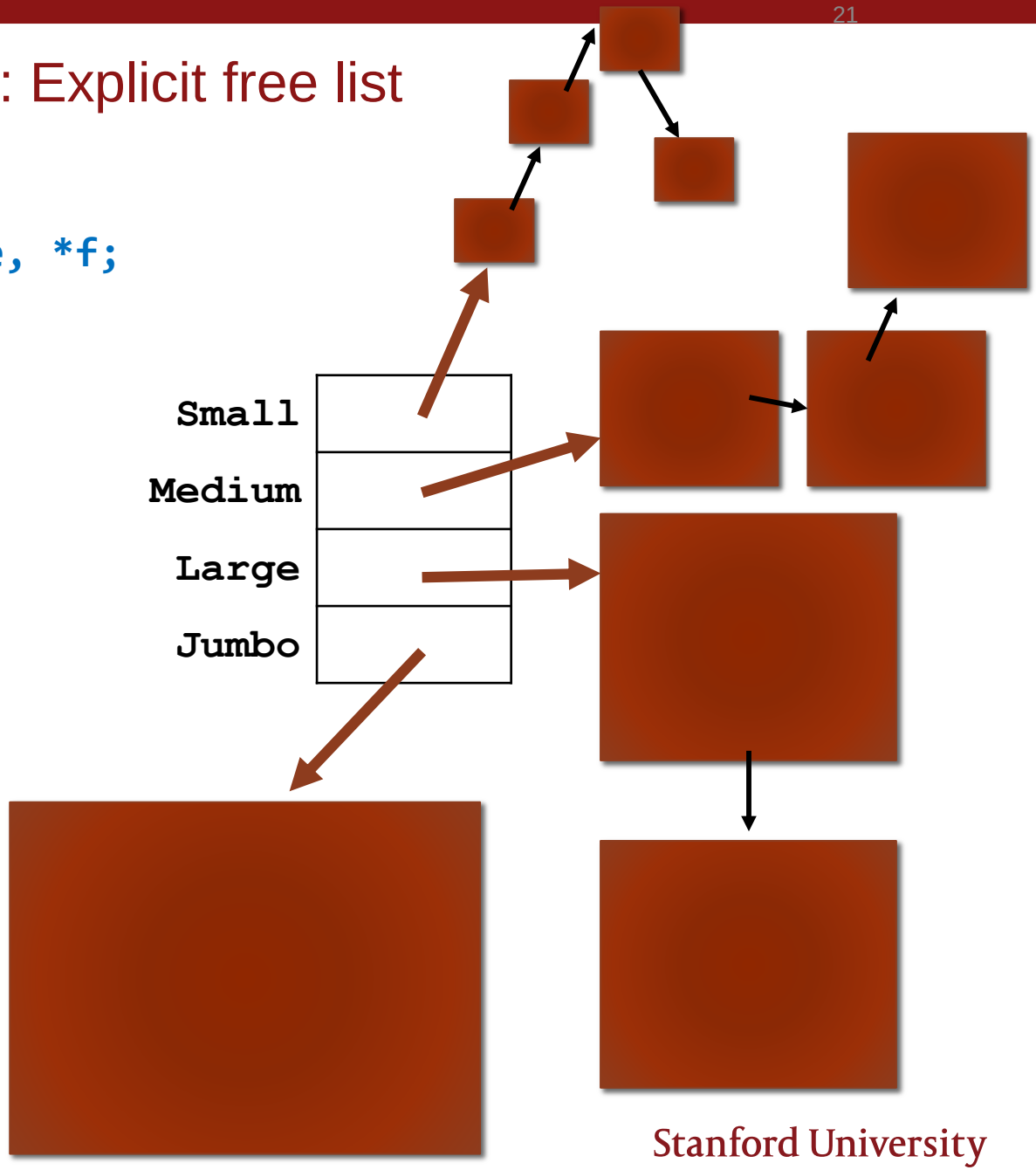
Implementation #4: Explicit free list *bucketed by size*

```
void *a, *b, *c, *d, *e, *f;
a = malloc(4);
b = malloc(8);
c = malloc(4);
d = malloc(4);
free(a);
free(c);
e = malloc(12);
b = realloc(b, 12);
d = realloc(d, 8);
f = malloc(12);
```



Implementation #4: Explicit free list *bucketed by size*

```
void *a, *b, *c, *d, *e, *f;
a = malloc(4);
b = malloc(8);
c = malloc(4);
d = malloc(4);
free(a);
free(c);
e = malloc(12);
b = realloc(b, 12);
d = realloc(d, 8);
f = malloc(12);
```



How can we balance the conflicting goals of the heap manager?

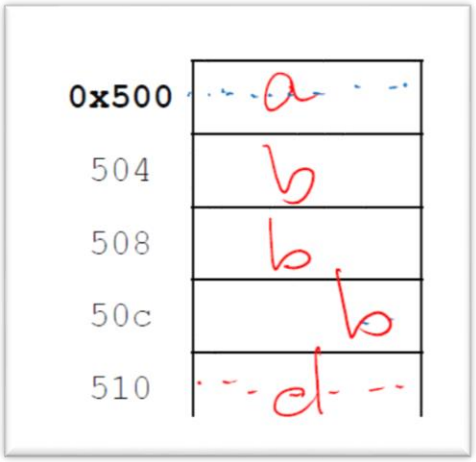
Correctness

(obviously)

Throughput
(speed of handling requests)

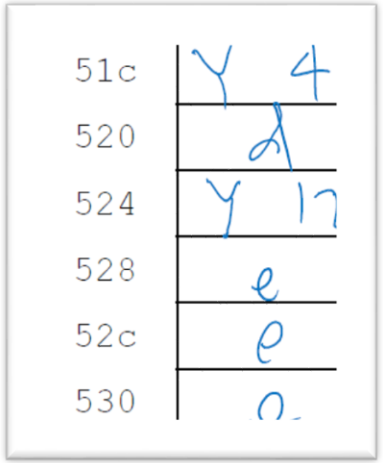
Utilization
(not too many holes in space)

Your Pinterest Board of Design Ideas for Heap Allocator



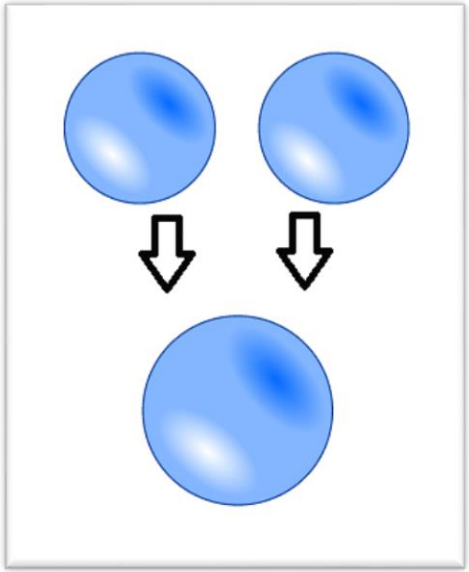
DIY Easy!

In-Use List + Free List



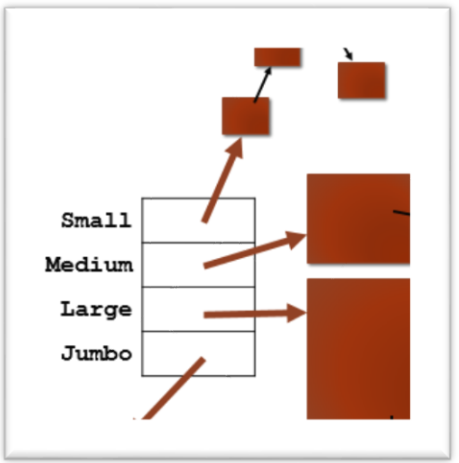
Cute Storage Idea!

Header holds block metadata adjacent to payload



Declutter Tip

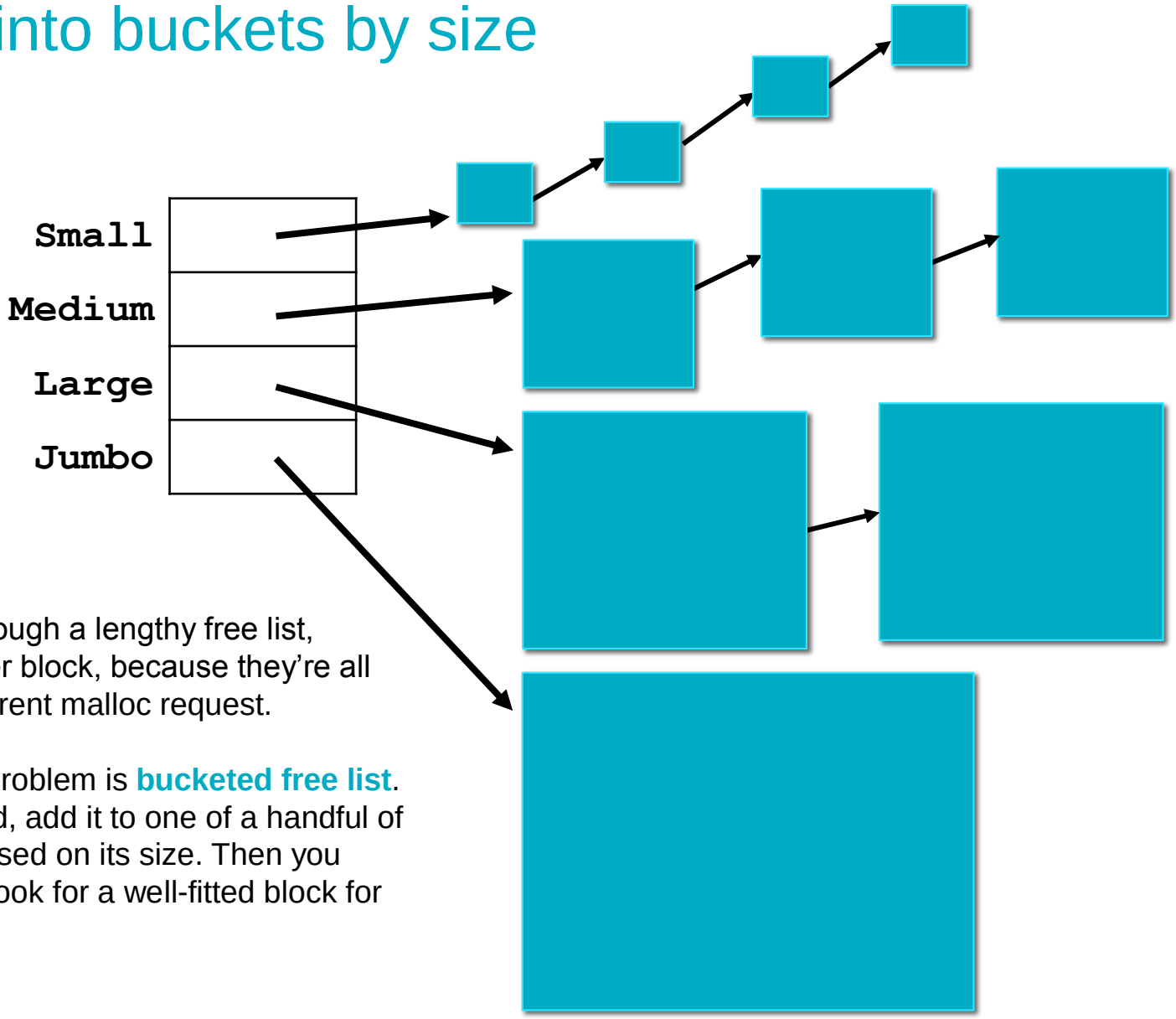
Coalesce adjacent free blocks to reduce fragmentation



Organize!

Multiple free lists, bucketed by block size

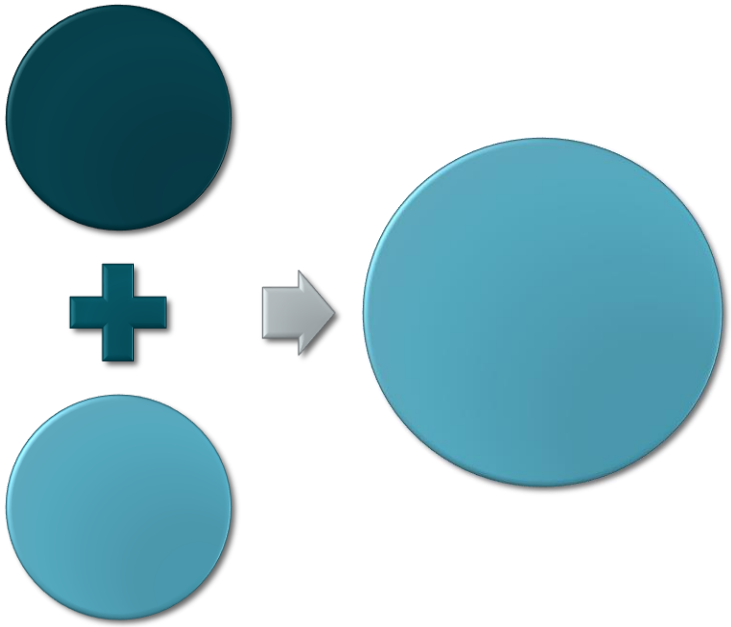
Organize!
Sort free list into buckets by size



It's a drag to step through a lengthy free list, passing up block after block, because they're all too small for your current malloc request.

One solution to this problem is **bucketed free list**. When a block is freed, add it to one of a handful of different free lists, based on its size. Then you know right where to look for a well-fitted block for new requests!

Declutter Tip:
Coalesce adjacent free blocks to reduce fragmentation



As we saw in some of the examples in lecture, after allocating many small blocks, and then freeing some of them, we may end up with enough total free memory to handle a large block request, but nevertheless not be able to fulfill it because the free memory is scattered (fragmented) throughout memory.

One solution to this problem is **coalescing**. When a block is freed, look and see if its "next-door neighbors" on either side are also free. If so, then combine them into one larger free block.

Think carefully about what pointer/offset infrastructure you will need to be able to do this coalescing. Some designs we talked about in class will only allow you to know where your next-door neighbors are to one side (forward in a linked list), not the other side. This might be a tradeoff you're willing to make for storage simplicity? Or you could try to find ways to go bidirectional.