CSE 29
Lecture 14 Summary

February 19, 2026

📣 Logistical Things

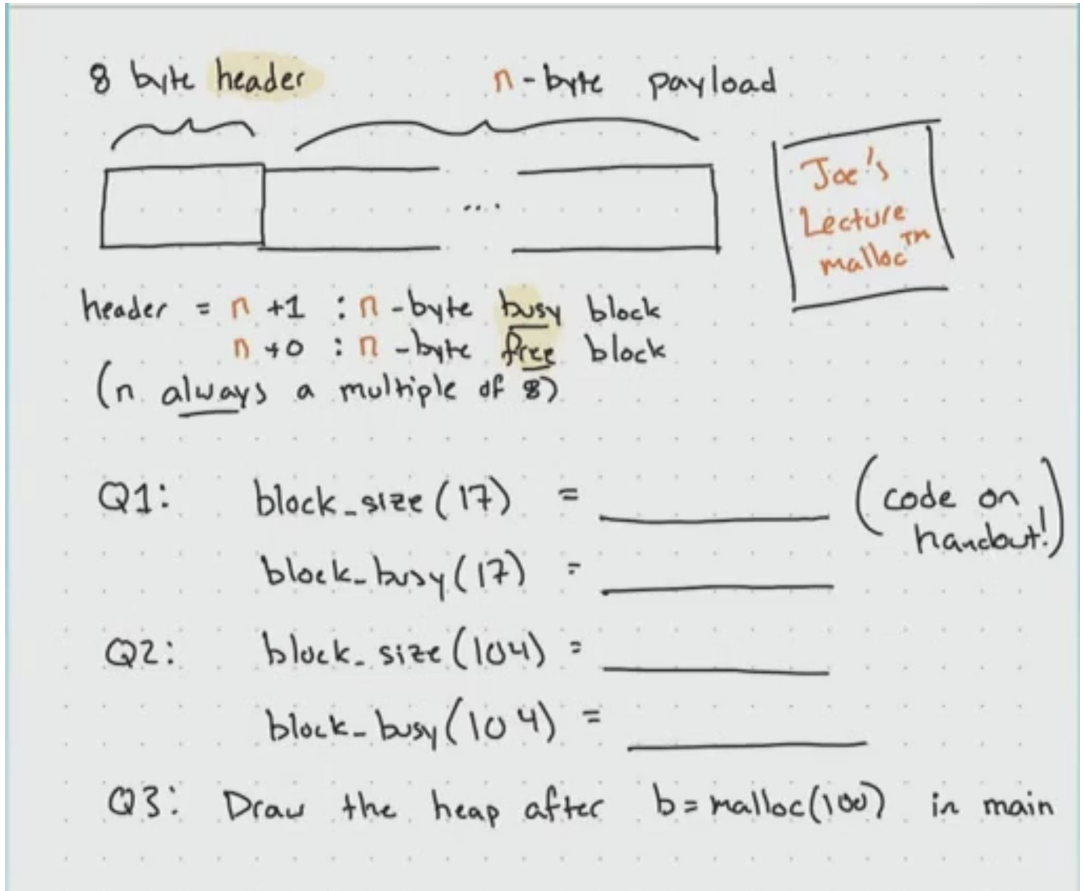
- Assignment 3 grades will be out soon, along with the resubmission specs
  - This will be due the same night as Assignment 4, so plan accordingly!
- Exam 2 is next week!

Today’s Lecture

- We’ll be working with Joe’s lecture malloc™, which is different from C’s implementation of malloc, as well as your PA vmalloc
  - Joe does not have time to go over PA4 vmalloc in lecture, so this is a simplified version
  - There are no footers or endmarker in Joe’s lecture malloc™

Review Questions

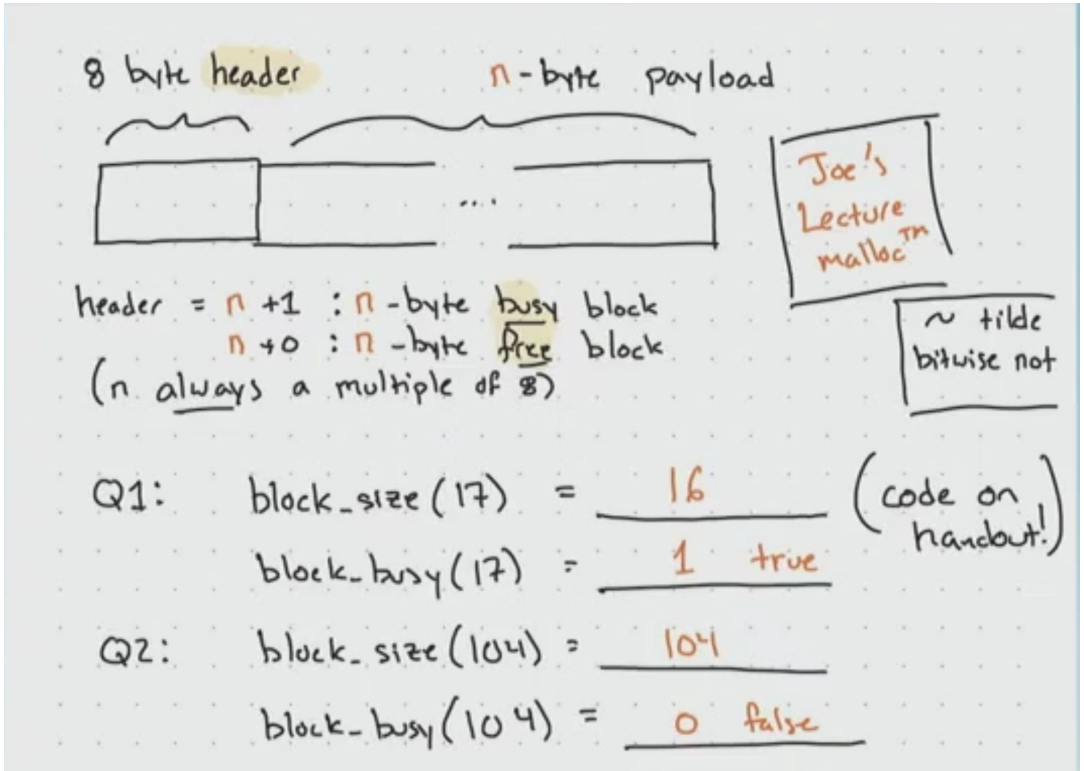
Answers in next slide!




```
#define HEAP_SIZE 4096
#define SLOT_SIZE 8
#define HEAP_SLOTS HEAP_SIZE/SLOT_SIZE

uint64_t* HEAP_START = NULL;

void init_heap() { ... } // set up HEAP_START

size_t block_size(uint64_t s) { return s & (~1); }
int block_busy(uint64_t s) { return s & 1; }
size_t round_size(size_t size) { return (size + 7) & ~7; }



void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}



void* malloc(size_t requested_size) {
  init_heap();
  int val_index = 0;
  size_t rounded = round_size(requested_size);
  while(val_index < HEAP_SLOTS) {
    uint64_t curr_slot = HEAP_START[val_index];
    int current_size = block_size(curr_slot);
    int current_busy = block_busy(curr_slot);
    if(!current_busy && (current_size >= rounded)) {
      return allocate_at(&HEAP_START[val_index], rounded);
    }
    else {
      val_index += (current_size / SLOT_SIZE) + 1;
      continue;
    }
  }
  return NULL;
}



void free(void* ptr) {



}


int main() {
  int* a = malloc(20);
  int* b = malloc(100);
  int* c = malloc(20);

  free(b);

  int* d = malloc(15);
}
```

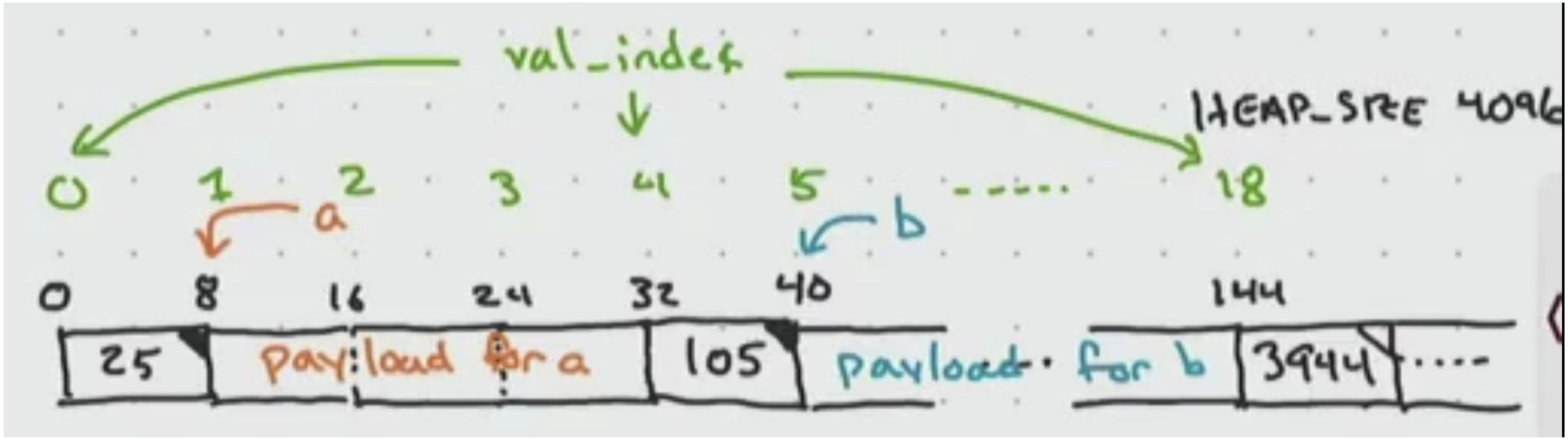
main

Review Question Answers



Review Question Answers Cont.

Q3: Draw the heap after b = malloc(100) in main



Joe’s lecture malloc™

Let’s look at the handout!

How would the heap look like after the first three `malloc` calls?

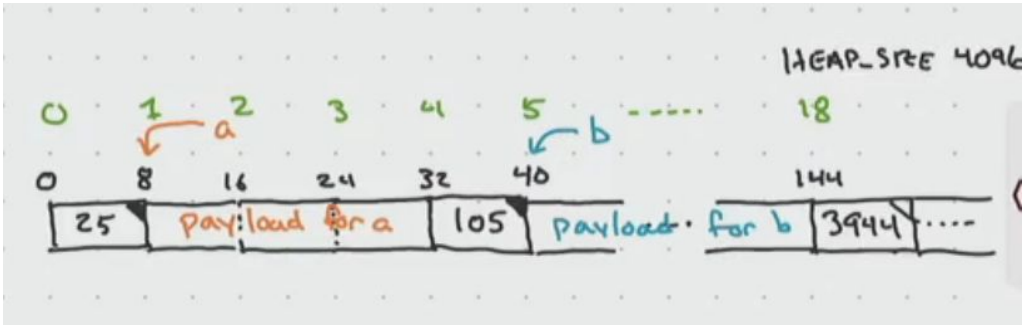
```
int main() {
  int* a = malloc(20);
  int* b = malloc(100);
  int* c = malloc(20);

  free(b);

  int* d = malloc(15);
}
```

```
#define HEAP_SIZE 4096
#define SLOT_SIZE 8
#define HEAP_SLOTS HEAP_SIZE/SLOT_SIZE

uint64_t* HEAP_START = NULL;

void init_heap() { ... } // set up HEAP_START

size_t block_size(uint64_t s) { return s & (~1); }
int block_busy(uint64_t s) { return s & 1; }
size_t round_size(size_t size) { return (size + 7) & ~7; }
```

```
void* malloc(size_t requested_size) {
  init_heap();
  int val_index = 0;
  size_t rounded = round_size(requested_size);
  while(val_index < HEAP_SLOTS) {
    uint64_t curr_slot = HEAP_START[val_index];
    int current_size = block_size(curr_slot);
    int current_busy = block_busy(curr_slot);
    if(!current_busy && (current_size >= rounded)) {
      return allocate_at(&HEAP_START[val_index], rounded);
    }
    else {
      val_index += (current_size / SLOT_SIZE) + 1;
      continue;
    }
  }
  return NULL;
}
```

The heap after the first two `malloc` calls

```
int main() {
  int* a = malloc(20);
  int* b = malloc(100);
  int* c = malloc(20);

  free(b);

  int* d = malloc(15);
}
```

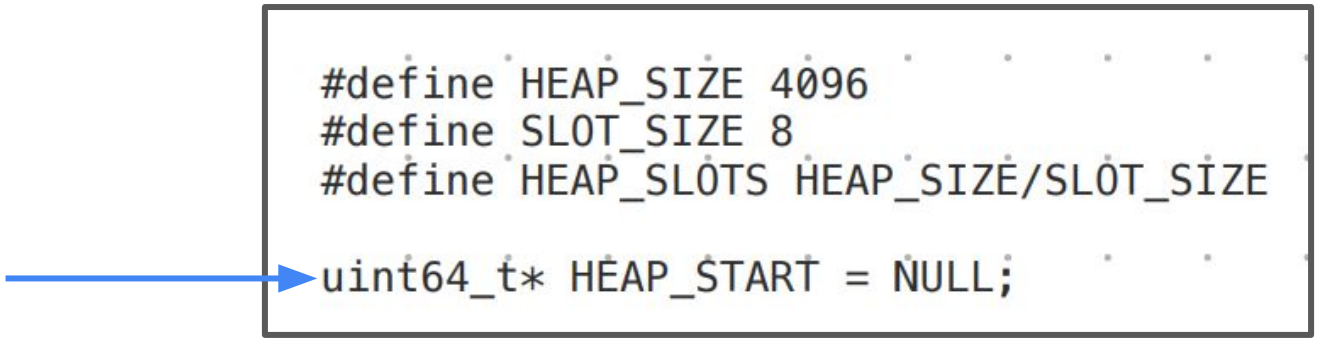


Remaining free block size: 4096 - (24 + 8) - (104 + 8) = 3952 - 8 = 3944

Remember: headers are 8 bytes!

Side Note: We can treat the heap like an array

- `HEAP_START` is defined as a `uint64_t*`, so we can do `HEAP_START[#]`
  - `HEAP_START[0]` and `HEAP_START[1]` are 8 bytes apart (because the size of `uint64_t` is 8 bytes)
- We can use indexing to refer to blocks in the heap
  - Alternate method: use pointer arithmetic (we'll see this in future lectures)



Questions

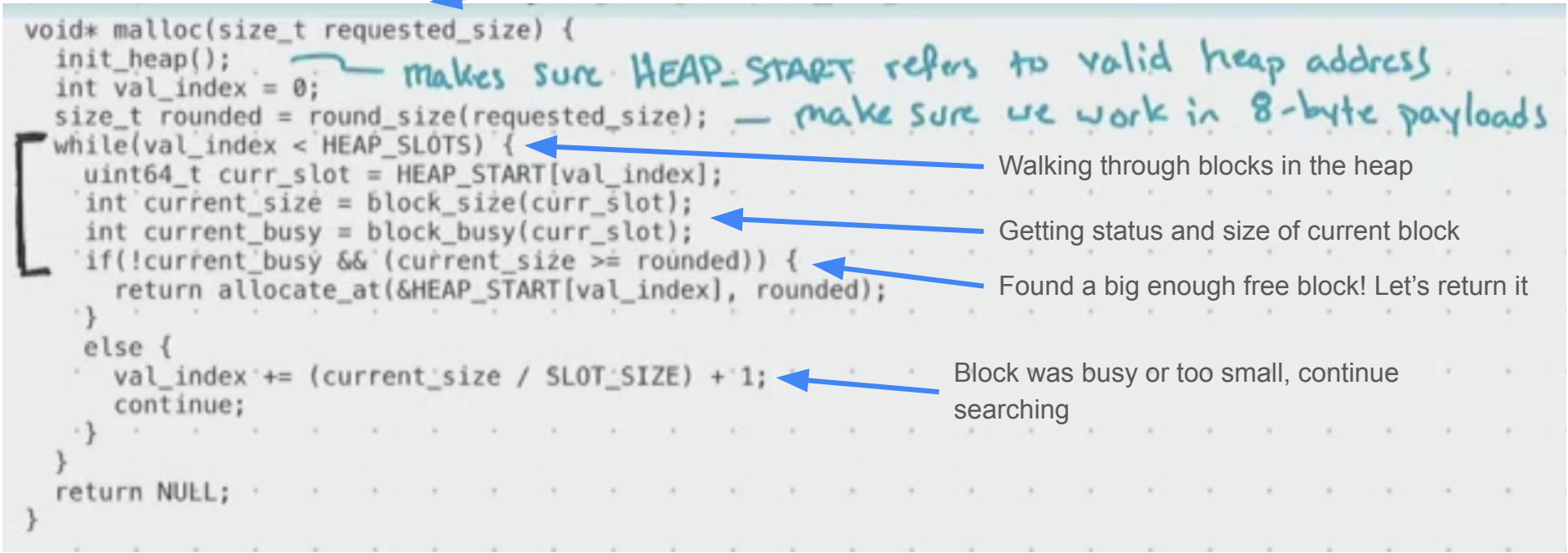
- In real `malloc`, if you `malloc(1)`, what would the size be?
  - Inside of the metadata of the block, it just stores what size it rounded up to not what you gave it
  - size would not be 1, it would be of a multiple of what byte alignment the malloc implementation has
- If you have a block with padding, could you write into it even though what you requested for was smaller?
  - If you write over the size into the padding, it would be fine (there's nothing important there you would be overwriting)

Let's look into our `malloc` function implementation

Try to figure out the purpose of each line here

```
void* malloc(size_t requested_size) {
  init_heap();
  int val_index = 0;
  size_t rounded = round_size(requested_size);
  while(val_index < HEAP_SLOTS) {
    uint64_t curr_slot = HEAP_START[val_index];
    int current_size = block_size(curr_slot);
    int current_busy = block_busy(curr_slot);
    if(!current_busy && (current_size >= rounded)) {
      return allocate_at(&HEAP_START[val_index], rounded);
    }
    else {
      val_index += (current_size / SLOT_SIZE) + 1;
      continue;
    }
  }
  return NULL;
}
```

Our `malloc` function



Working through an example!

```
void* malloc(size_t requested_size) {
  init_heap();
  int val_index = 0;
  size_t rounded = round_size(requested_size);
  while(val_index < HEAP_SLOTS) {
    uint64_t curr_slot = HEAP_START[val_index];
    int current_size = block_size(curr_slot);
    int current_busy = block_busy(curr_slot);
    if(!current_busy && (current_size >= rounded)) {
      return allocate_at(&HEAP_START[val_index], rounded);
    }
    else {
      val_index += (current_size / SLOT_SIZE) + 1;
      continue;
    }
  }
  return NULL;
}
```

makes sure HEAP_START refers to valid heap address

make sure we work in 8-byte payloads

curr_slot:	25	105	3944
current_size:	24	104	3944
current_busy:	true	true	false

↳ called on 3rd iteration (val_index = 18)

+= 4 += 14

```
int main() {
  int* a = malloc(20);
  int* b = malloc(100);
  int* c = malloc(20);

  free(b);

  int* d = malloc(15);
}
```

allocate_at

What is `allocate_at`?

- We use `allocate_at` in our `malloc` implementation
- The purpose of the function is to change a free block into an allocated block
- To do this, `allocate_at` will change the header of the free block to make it allocated
    - Putting in the size of the allocated block and marking it as busy
- We could have just put this into `malloc`, but it's best practice to separate this functionality to make our code more readable and use good abstraction

Let's look at how we called `allocate_at`

```
void* malloc(size_t requested_size) {
  init_heap();
  int val_index = 0;
  size_t rounded = round_size(requested_size);
  while(val_index < HEAP_SLOTS) {
    uint64_t curr_slot = HEAP_START[val_index];
    int current_size = block_size(curr_slot);
    int current_busy = block_busy(curr_slot);
    if(!current_busy && (current_size >= rounded)) {
      return allocate_at(&HEAP_START[val_index], rounded);
    }
    else {
      val_index += (current_size / SLOT_SIZE) + 1;
      continue;
    }
  }
  return NULL;
}
```

What is the first argument we passed into `allocate_at`?

- `&HEAP_START[val_index]`
- This is a pointer to the header of a free block that is big enough to hold the `rounded` size

Let's look at `allocate_at`

Try to figure out the purpose of each line here

```
void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}
```

Our `allocate_at` function

Assumes: start is address of the header of a free block of at least size bytes

```
void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}
```

Finds size of free block passed in

If our free block is too big for what we need

Put a new block header for the new free block

Put the new size + 1 (busy bit)

Return a pointer to the start of the payload

- Note that it is `&start[1]`
- `&start[0]` would be the start of the header

Questions 🤔

- Are you always guaranteed to know the size of the heap?
  - In Joe's lecture malloc™, yes!
- Why can't you move payloads around so that there will always be one free block in order to optimize space?
  - The "user" that called malloc was returned a specific address, and therefore will write data to that address. If you move blocks around, the "user" will still write to the same address, even though you moved their payload to somewhere else.
- What happens if you malloc(20) and then write in bytes 21, 22, 23 (the padding)?
  - Nothing will happen, our implementation does not know when memory is written to
- What if a header is overwritten?
  - Our implementation would no longer work, and things would go horribly wrong :(
- Why do you prefer to work with indices?
  - Pointer arithmetic hasn't been introduced yet!

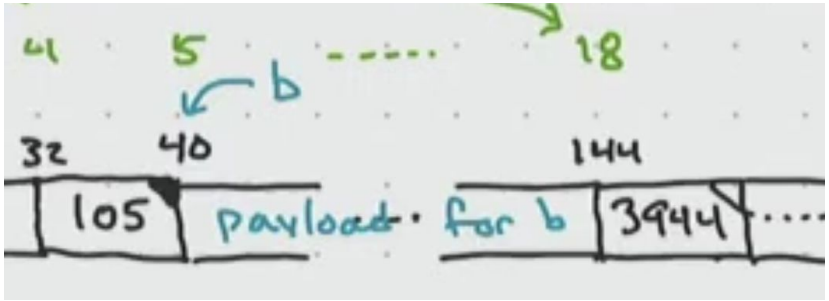
Implementing free

Try implementing our `free` function!

This will be about 2 lines long

```
void free(void* ptr) {


}
```

Implementation!



We need to cast

We get a ptr but we want the header, so we need to go backwards (index by -1)

Ex: freeing b, the address passed is 40, but the header is at 32

```
void free(void* ptr) {
    uint64_t* p = (uint64_t*) ptr;
    p[-1] -= 1;
}
```

We subtract by 1, because the least significant bit shows whether the block is free (0) or busy (1)

We don't handle **coalescing** in this `free` method (it's very important), so think about how you'll handle coalescing in your PA.

Questions

- Why aren’t payloads zeroed out when you free?
  - When you free, payloads aren’t zeroed out
  - This is true for Joe’s lecture malloc™, the PA, and production code.
  - This is great for performance, but could have security ramifications
    - What if a password is stored in the payload?

Joe's Notes (11am)

8 byte header n-byte payload

Joe's Lecture malloc™

header = n+1 : n-byte busy block
n+0 : n-byte free block
(n always a multiple of 8)

~ (tilde) is bitwise not

Q1: block_size(17) = 16

block_busy(17) = true / 1

Q2: block_size(104) = 104

block_busy(104) = false / 0

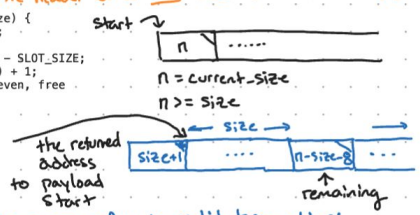
Q3: Draw the heap after b = malloc(100) in main

```
#define HEAP_SIZE 4096
#define SLOT_SIZE 8
#define HEAP_SLOTS HEAP_SIZE/SLOT_SIZE

uint64_t* HEAP_START = NULL;

void init_heap() { ... } // set up HEAP_START

size_t block_size(uint64_t s) { return s & (~1); }
int block_busy(uint64_t s) { return s & 1; }
size_t round_size(size_t size) { return (size + 7) & ~7; }
```

// Assumes: start is address of the header of a free block of at least size bytes

```
void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}
```

describe this return value!




```
void* malloc(size_t requested_size) {
  init_heap();
  int val_index = 0;
  size_t rounded = round_size(requested_size);
  while(val_index < HEAP_SLOTS) {
    uint64_t curr_slot = HEAP_START[val_index];
    int current_size = block_size(curr_slot);
    int current_busy = block_busy(curr_slot);
    if(!current_busy && (current_size >= rounded)) {
      return allocate_at(&HEAP_START[val_index], rounded);
    }
    else {
      val_index += (current_size / SLOT_SIZE) + 1;
      continue;
    }
  }
  return NULL;
}
```

makes sure HEAP_START refers to valid heap address

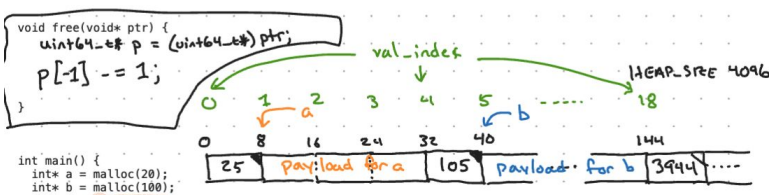
make sure we work in 8-byte payloads

	1st	2nd	3rd
curr_slot:	25	105	3944
current_size:	24	104	3944
current_busy:	true	true	false

↳ called on 3rd iteration (val_index = 18)

+=4 +=14

```
void free(void* ptr) {
  uint64_t* p = (uint64_t*) ptr;
  p[-1] -= 1;
}
```




```
int main() {
  int* a = malloc(20);
  int* b = malloc(100);
  int* c = malloc(20);

  free(b);

  int* d = malloc(15);
}
```

Joe's Notes (12:30pm)

8 byte header n-byte payload

Joe's Lecture malloc™

~ tilde bitwise not

header = n+1 : n-byte busy block
n+0 : n-byte free block
(n always a multiple of 8)

Q1: block_size(17) = 16 (code on handout!)

block_busy(17) = 1 true

Q2: block_size(104) = 104

block_busy(104) = 0 false

Q3: Draw the heap after b = malloc(100) in main

```
#define HEAP_SIZE 4096
#define SLOT_SIZE 8
#define HEAP_SLOTS HEAP_SIZE/SLOT_SIZE

uint64_t* HEAP_START = NULL;

void init_heap() { ... } // set up HEAP_START

size_t block_size(uint64_t s) { return s & (~1); }
int block_busy(uint64_t s) { return s & 1; }
size_t round_size(size_t size) { return (size + 7) & ~7; }
```

// ASSUMES start points to free block at least size bytes long

```
void* allocate_at(uint64_t* start, size_t size) {
  size_t current_size = block_size(start[0]);
  if(current_size > size) {
    uint64_t remaining = current_size - size - SLOT_SIZE;
    int next_block_index = (size / SLOT_SIZE) + 1;
    start[next_block_index] = remaining; // even, free
  }
  start[0] = size | 1; // busy
  return &start[1];
}
```

start → header n

returned (payload address) → size+1 (busy) | n-size-8 (free) ....

describe this return value!

size <= n

n = current_size

```
void* malloc(size_t requested_size) {
  init_heap();
  int val_index = 0;
  size_t rounded = round_size(requested_size);
  while(val_index < HEAP_SLOTS) {
    uint64_t curr_slot = HEAP_START[val_index];
    int current_size = block_size(curr_slot);
    int current_busy = block_busy(curr_slot);
    if(!current_busy && (current_size >= rounded)) {
      return allocate_at(&HEAP_START[val_index], rounded);
    }
    else {
      val_index += (current_size / SLOT_SIZE) + 1;
      continue;
    }
  }
  return NULL;
}
```

ensures HEAP_START is set up

rounded%8 == 0

loop over all blocks to find one big enough

expectation: curr_slot is a header

Is the current block free?

val_index = 0 ↑ 18 - big enough?

current_size = ~~24~~ ~~104~~ 3944

the free space before we allocate for c

```
void free(void* ptr) {
    uint64_t* p = (uint64_t*) ptr;
    p[-1] = p[-1] - 1;
}
```

allocate_at called when val_index = 18 current_size = 3944 what args?

```
int main() {
  int* a = malloc(20);
  int* b = malloc(100);
  int* c = malloc(20);
  free(b);
  int* d = malloc(15);
}
```