University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 220: Computer Systems & Programming

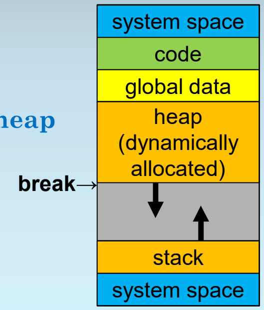
Implementing Dynamic Allocation

Dynamic Allocation Interacts with the OS

Recall our canonical memory map.

The **address**
- **after the end of the heap**
- is called **the "break."**

To change the break, make a system call.



sbrk Adjusts the Address of the Break

In Linux, for example, the call is

```
void *sbrk (intptr_t increment);
```

Calling **sbrk** requests that
- the break be changed by adding **increment**,
- and returns the address of the previous break (or **((void*)-1)** on failure).

One can grow or shrink the heap with **sbrk**.

intptr_t is Needed to Hold the **sbrk** Argument

```
void *sbrk (intptr_t increment);
```

But what's an intptr_t?

An integer large enough to hold a pointer.

These became important with 64-bit address spaces.

An **int** can no longer hold a pointer!

Let's Write a Best-Fit Logarithmic Allocator

Let's implement dynamic allocation!

We'll **start simple: no reclamation**.

Then we'll write
- **a best-fit logarithmic allocator**,
- which was common for a couple of decades.

For Simplicity, We Build on Top of `malloc`

To avoid overriding the **C** library,
- we use `malloc` instead of `sbrk`
- to get a big chunk of memory to manage,
- and store the chunk in file-scope variables.

In particular,

```
static uint8_t* free_bytes;
static size_t n_free_bytes;
```

The **free memory consists of `n_free_bytes` bytes starting at address `free_bytes`**.

First Step: Carving Off a Block

How do we allocate a new block?

If we don't care about reclamation
- (reusing blocks that are freed),
- carving off a block is straightforward.

We'll write a function for doing so:

```
void* mem220_allocate
      (size_t n_bytes);
```

The behavior is identical to that of `malloc`.

An Overly Simple Allocation Routine

```
void* mem220_allocate (size_t n_bytes)
{
    void* new_block = free_bytes;
    if (n_free_bytes < n_bytes) {
        return NULL;
    }
    free_bytes += n_bytes;
    n_free_bytes -= n_bytes;
    return new_block;
}
```

New block starts at start of free memory.

Check Whether Available Memory is Sufficient

```
void* mem220_allocate (size_t n_bytes)
{
    void* new_block = free_bytes;
    if (n_free_bytes < n_bytes) {
        return NULL;
    }
    free_bytes += n_bytes;
    n_free_bytes -= n_bytes;
    return new_block;
}
```

Do we have enough free memory?

Remove the Block from Free Memory and Return It

```
void* mem220_allocate (size_t n_bytes)
{
    void* new_block = free_bytes;
    if (n_free_bytes < n_bytes) {
        return NULL;
    }
    free_bytes += n_bytes;
    n_free_bytes -= n_bytes;
    return new_block;
}
```

Remove the block from free memory.

And return the new block.

Should Add Alignment or Round Up Block Sizes

What about alignment?

In our **next implementation**,
- all blocks will be 2^k bytes for some integer **k**
- and the smallest will be 32 bytes (on the lab machines),
- so all blocks **will maintain `malloc`'s alignment** (typically 16-byte).

To align, round up, then squash the low bits
- X = (X + 15) & -16
- X = (X + 15) ^ ((X + 15) & 15) // safer

Want to Bin Block Sizes and Make Tracking Easy

How should we manage allocated blocks?

Without binning block sizes in some way,
- **fragmentation effects can become bad**,
- especially when coupled with alignment.
- Have you ever played "continuous Tetris?"

Allowing **arbitrary addresses** also **makes tracking blocks more difficult** (and pointers have alignment requirements, too).

Recall Dynamic Resizing's Approach to Array Sizes

Think back to dynamic resizing:
- we double our array
- each time we need more.

When we examined waste space,
- we found that doing so
- gave us a pretty good fit
- (average 38% waste).

We Build a Best-Fit Logarithmic Allocator

Let's use the same idea for allocation:
- **allocate the smallest power of 2 bytes**
- **into which** the desired **block fits**.

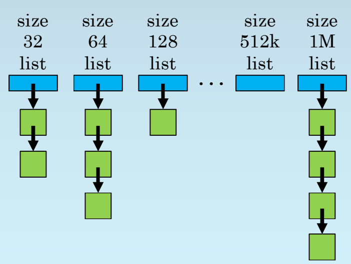
This approach is called a **best-fit logarithmic allocator**.

We **might allow blocks to be split** (into two smaller blocks) **and re-combined**.
- For example, see the page allocation management in the Linux kernel (in ECE391).
- **Our implementation does neither.**

A Linked List Holds Free Blocks of Each Size

Let's talk about data structures.

Free blocks are kept in **linked lists based on the size of the blocks**, as shown to the right.

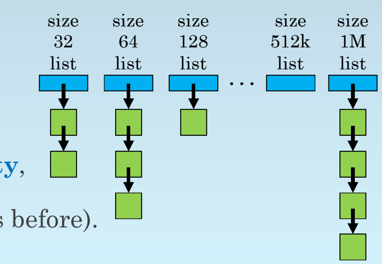


Allocate New Blocks as Necessary (As Done Earlier)

When we need a block, we **look in the list**.

For example, if we want 100 bytes, we look in the size 128 list.

If list is empty, we **allocate a new block** (as before).



Can Build a Data Structure to Find Info about Blocks

When a block is freed, we must know its size.

One **option**:
- build a data structure
- to **translate block address**
- **into other information**
- (look up information based on address).

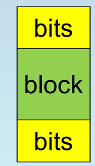
Some memory managers
must take such an approach.

But **we don't need to do so**.

What's in Memory Around a Block?

Let's say that you call **`malloc`**.

Back comes a block.



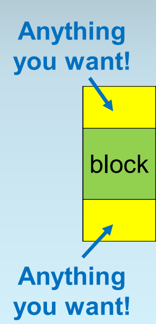
What is stored in the
addresses before the block?

What about the addresses
after the block?

What's in Memory Around a Block?

Now you are writing **`malloc`**.

You need to return a block.



What is stored in the
addresses before the block?

What about the addresses
after the block?

Use a Header Above the Block to Store Information

We **store the block size** in a header
above the block:

```
struct mem_block_t {
    size_t size;
    mem_block_t* next;
};
```

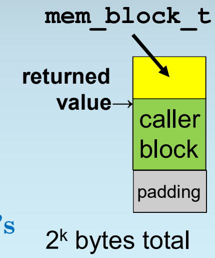
The **`next`** field is for our linked lists.

On 64-bit machines,
`sizeof (mem_block_t)` is 16.

Actual Allocation Contains Three Sections

In other words, the block that we actually allocate includes
- a **`mem_block_t`**,
- **bytes for the caller**, and
- **padding** to a power of 2.

The pointer that we **return** is the **address of the caller's data** (after `mem_block_t`).



Linked List Heads are a File-Scope Array

The linked lists are lists of `mem_block_t`.

What do our bins look like in **C**?

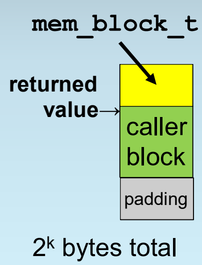
```
#define MEM220_MAX_ALLOC_LOG 20

static mem_block_t*
    mem_bin[MEM220_MAX_ALLOC_LOG+1];
```

The **head of the linked list**
- with **blocks of 2^k bytes**
- **has array index k**.

C Allows Programmers to Hide Details from Compiler

Notice that,
- from the point of view
- of code that manages linked list of free blocks,
- only the `mem_block_t` exists.

The **blocks are generally larger than `sizeof (mem_block_t)`**.



Interface Offers Four Functions Similar to the C Library

Next, let's take a look at the API.

We'll **write four functions**
- corresponding to the four
- that we discussed
- in the **C** library.

```
void* mem220_allocate
          (size_t n_bytes);
```

You've seen this routine already.

It **behaves the same as `malloc`**.

Second Routine Replaces `calloc`

The next routine replaces **`calloc`**.

In new code,
- there's less benefit*
- to matching the original signature,
- so instead we have:

```
void* mem220_allocate_and_zero
      (size_t n_bytes);
```

The routine **tries to allocate and zero a block, returning a pointer to the block or `NULL`**.

*Using distinct parameter lists may help to catch some programmer mistakes.

Third Routine Replaces `realloc`

The third interface replaces **`realloc`**:

```
int32_t mem220_reallocate
    (void** ptr_to_ptr,
     size_t n_bytes);
```

The routine works similarly to **`realloc`**:
- given a pointer to a pointer to an old block*
- and given a new size
- the routine **tries to change the block's size**,
- **copying and freeing the old block as necessary**.

*Sadly, an explicit cast to (void**) is now required.

Third Routine Avoids `realloc` Misuse Case

Also, the new version **avoids the common misuse case for `realloc`**:

```
int32_t mem220_reallocate
    (void** ptr_to_ptr,
     size_t n_bytes);
```

`*ptr_to_ptr` **changes**
- **only on success**, and
- only when the block had to move.

The function **returns 0 on success, or -1 on failure**.

Example of a Value-Result Argument

```
int32_t mem220_reallocate
        (void** ptr_to_ptr,
         size_t n_bytes);
```

Arguments such as **`ptr_to_ptr`**, **that both**
- **convey a value to the function and**
- **convey an output back to the caller**
- **are** sometimes called **value-result arguments**.

Final Routine is Identical to `free`

The last routine behaves identically to `free`:

```
void mem220_free (void* ptr);
```

Note that blocks from C's library API are not interchangeable with blocks from our API.

Blocks allocated with our routines must be freed with `mem220_free`.

When Does Non-Trivial Initialization Occur?

Remember **our file-scope variables**?

```
static uint8_t* free_bytes;
static size_t n_free_bytes;
static mem_block_t*
    mem_bin[MEM220_MAX_ALLOC_LOG+1];
```

You may have noticed that they **are not initialized**.

When does initialization take place?

And how do we cause it to happen?

When Can Non-Trivial Initialization Occur?

What are the options?

1. **static initialization** (`static int x = 42;`) — not a solution for our problem
2. **a new API call** (`int32_t mem220_init (void);`) — requires that other code call it first
3. **compiler/language/Makefile support** (available in C++) — only last available in C, and not always easy to use anyway
4. **on first API call** (check in every call) — requires extra work for every call

Which API Calls Can Be Made First?

Which can the user call first?

`mem220_allocate`

`mem220_allocate_and_zero`

`mem220_reallocate`

~~`mem220_free`~~

But

- **`mem220_allocate_and_zero` and `mem220_reallocate`**
- **call `mem220_allocate`!**

So only one call need be checked…

We Check for Initialization on Each API Call

We choose the last option:
initialize on the first API call.

Why?

Dynamic allocation is already "expensive"
(>200 cycles on my Cygwin desktop in April 2018).

And only one check is needed.
- User cannot call **mem220_free** first.
- Need to add a check to **mem220_allocate**.
- Other API calls call **mem220_allocate** to obtain a block before using file-scope variables.

Initialization Uses File-Scope Variable and Static Function

What's the mechanism?

A **file-scope variable** and a **static function**
(not accessible outside the file).

```
static int32_t init_done = 0;
static void mem220_init ();
// And, at start of mem220_allocate…
if (!init_done) { mem220_init (); }
```

(**mem220_init** sets **init_done** to 1.)

Start with Local Variables and Initialization

Let's look at **mem220_allocate**.

```
void* mem220_allocate
    (size_t n_bytes)
{
    size_t       block_size;
    int32_t      bin;
    mem_block_t* new_block;
    if (!init_done) {
        mem220_init ();
    }
```

Number of bytes needed (including header).

Index into array of free block lists.

Pointer to new block.

Calculate Necessary Values and Check Arguments

```
block_size = n_bytes +
             sizeof (*new_block);
if (n_bytes == 0 ||
    block_size > MEM220_MAX_ALLOC) {
    return NULL;
}
bin = log2_ceil (block_size);
```

We need **n_bytes** plus a **mem_block_t**.

Calculate Necessary Values and Check Arguments

```
block_size = n_bytes +
             sizeof (*new_block);
if (n_bytes == 0 ||
    block_size > MEM220_MAX_ALLOC) {
    return NULL;
}
bin = log2_ceil (block_size);
```

Size too small or too large? Give up.

Calculate Necessary Values and Check Arguments

```
block_size = n_bytes +
             sizeof (*new_block);
if (n_bytes == 0 ||
    block_size > MEM220_MAX_ALLOC) {
    return NULL;
}
bin = log2_ceil (block_size);
```

Find the right bin (function discussed later).

Two Places to Obtain a Block

Does the right list have a free block in it?

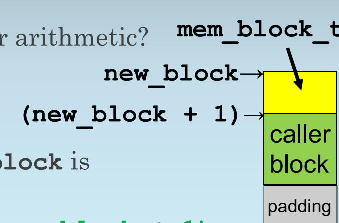
```
if (mem_bin[bin] != NULL) {
    // get block from free list
} else {
    // allocate a new block
}
return (new_block + 1);
```

Both cases set **new_block**.

What's this?

Pointer Arithmetic Gives the Right Answer

Remember pointer arithmetic?



The type of **new_block** is **mem_block_t***.

So where does (new_block + 1) point?

To the block to be returned!

If Free List Not Empty, Remove One Block

Now back to obtaining a block.
First, the easy case: there's one in the free list.

```
// get block from free list
new_block = mem_bin[bin];
mem_bin[bin] = new_block->next;
```

Remove block from linked list.

Check Available Space for a New Block

```
// allocate a new block
n_bytes = (1UL << bin);
if (n_free_bytes < n_bytes) {
    return NULL;
}
new_block =(mem_block_t*)free_bytes;
free_bytes += n_bytes;
n_free_bytes -= n_bytes;
new_block->size = n_bytes;
```

Number of bytes in block (2^{bin}).

No space? Give up.

Allocate a New Block

```
// allocate a new block
n_bytes = (1UL << bin);
if (n_free_bytes < n_by
    return NULL;
}
new_block =(mem_block_t*)free_bytes;
free_bytes += n_bytes;
n_free_bytes -= n_bytes;
new_block->size = n_bytes;
```

Allocate a new block as before (but with an explicit cast).

Write the Block Size into the New Block's Header

```
// allocate a new block
n_bytes = (1UL << bin);
if (n_free_bytes < n_bytes) {
    return NULL;
}
new_block =(mem_block_t*)free_bytes;
free_bytes += n_bytes;
n_free_bytes -= n_bytes;
new_block->size = n_bytes;
```

Mark the size field in the header.

Still Need to Write the Helper Function

That's it for allocation.

But what did this do?

```
bin = log2_ceil (block_size);
```

Calculate k such that $2^k \geq$ `block_size`.

In other words, **return** $\lceil$**log$_2$(block_size)**$\rceil$
(the ceiling of the base 2 logarithm).

How can we calculate that value?

How Can We Calculate Ceiling of Log_2?

```
// Returns ceiling of
// log_2 of its argument.
static int32_t log2_ceil
    (size_t value);
```

One option: library calls (with floating-point).

Instead, **let's use…**

bits!

Find the First 1 Bit and Check for a Power of Two

Let's look at a number as bits:

```
value = 000…000 1 ??????
```

To **calculate ceil (log$_2$ (value))**, we

- **find** the location of **the first 1 bit**, and
- **round up** unless all of the lower bits are 0.

Let's start with the second part.

How can we check: is `value` a power of 2?

Initialize Count to Reflect Whether `value` is a Power of 2

```
static int32_t log2_ceil
    (size_t value)
{
    int32_t ret_val;
    if ((value & (value - 1)) == 0){
        ret_val = -1;
    } else {
        ret_val = 0;
    }
```

Is **`value`** a power of 2?

If so, start counting at -1.

If not, start counting at 0.

Count Number of Non-Zero Bits on Smaller End

```
while (value > 0) {
    ret_val++;
    value >>= 1;
}
return ret_val;
```

Count number of non-zero bits from low end.

Return count adjusted by power of 2 check.

Convert Freed Pointer into a **mem_block_t***

Now, let's look at freeing a block.

Cast pointer into a **mem_block_t***.

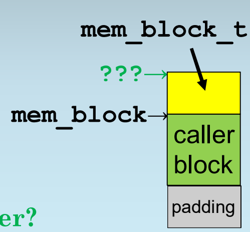
```
void mem220_free (void* ptr)
{
    mem_block_t* mem_block = ptr;
    int32_t      bin;
    if (ptr == NULL) { return; }
```

Ignore requests to free NULL.

Pointer Arithmetic Gives the Right Answer

Here's what we have:

The type of **mem_block** is **mem_block_t***.



How can we get back the pointer to our header?

mem_block - 1

Find Block Size and Insert Block into Free List

Read block size from header and calculate bin number.

```
bin = log2_ceil
    (mem_block[-1].size);
mem_block[-1].next = mem_bin[bin];
mem_bin[bin] = &mem_block[-1];
```

Add block to correct linked list (of free blocks of the same size).

The Code is on the Class Web Page

The other two calls are straightforward.

Reading the code is left as an exercise.

All of it, along with some short tests, can be found on the class web page.