Systems I – CS 6013

Computer Architecture and Operating Systems

Lecture 22: Malloc Assignment

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

Lecture 22 – Topics

- Malloc Assignment
  - Structure Sizes
  - Malloc / Free
  - mmap
  - Hash Table

Announcements / Questions

- Week 9 Readings – Concurrency: OSTEP Chapters:
  - 25: Dialogue
  - 26: Concurrency and Threads
  - 27: Thread API
  - 28: Locks

What is Malloc?

- `malloc()` allocates memory for us…
  - But it is only a function call…
    - Although a fairly advanced piece of code…
- But wait, if we are grabbing memory from the system, don't we need a system call?
  - Yes, to get raw blocks of memory, we use the `mmap()` system call.
  - However, `malloc()` does this behind the scenes for us so that we don't have to worry about it.

Storing an Address / Size in HT


```
int    * i = malloc( sizeof( int ) )
double * d = malloc( 8 )
ht.insert( i, 4 );
ht.insert( d, 8 );
```

- Declaration of `insert()`?
- What is the *type* of a generic pointer?
  - void *
- What is the size of (all) pointers?
  - 64 (or 32) bits.
- How can we tell?
  - sizeof(void *)
  - g++ compilation flag: -m32

- So, `insert()` is?

```
void insert( void *, size )
```

- What is the type of size?
  - int? long?
  - How can we handle it regardless of 32 vs 64 bit system / compilation?
  - Built-in type:
    - `size_t`

```
void insert( void * address,
             size_t size );
```

man malloc

- **malloc**
- Defined in header `<stdlib.h>`
- `void* malloc( size_t size );`
- Allocates size bytes of uninitialized storage.
- If allocation succeeds, returns a pointer that is suitably aligned for any object type with fundamental alignment.
- If size is zero, the behavior of malloc is implementation-defined. For example, a null pointer may be returned. Alternatively, a non-null pointer may be returned; but such a pointer should not be dereferenced, and should be passed to free to avoid memory leaks.
- malloc is thread-safe: it behaves as though only accessing the memory locations visible through its argument, and not any static storage. (synchronized)

How big is a structure?

- How many bytes does `struct Data` use in memory?
- How can we ask the computer?
  - `sizeof( struct Data )`
- Let's figure it out ourself…

```
struct Data {
    bool   b1;
    int    i1;
    bool   b2;
    bool   b3;
    void * p;
    bool   b4;
    bool   b5;
    bool   b6;
    bool   b7;
    bool   b8;
    int    i2;
};
```

How big is a structure?

- Size of each piece?
  - So the size is?
    - 24!
  - Well, no, it's 40…
  - What?
- Let's continue by printing out the addresses…

```
struct Data {        Bytes      Address
    bool   b1;       // 1       209808
                     // 3 bytes of padding
    int    i1;       // 4       209812 <- Aligned!
    bool   b2;       // 1       209816
    bool   b3;       // 1       209817
                     // 6 bytes of padding
    void * p;        // 8       209824 <- Aligned!
    bool   b4;       // 1       209832
    bool   b5;       // 1       209833
    bool   b6;       // 1       209834
    bool   b7;       // 1       209835
    bool   b8;       // 1       209836
                     // 3 bytes of padding
    int    i2;       // 4       209840 <- Aligned!
};
```

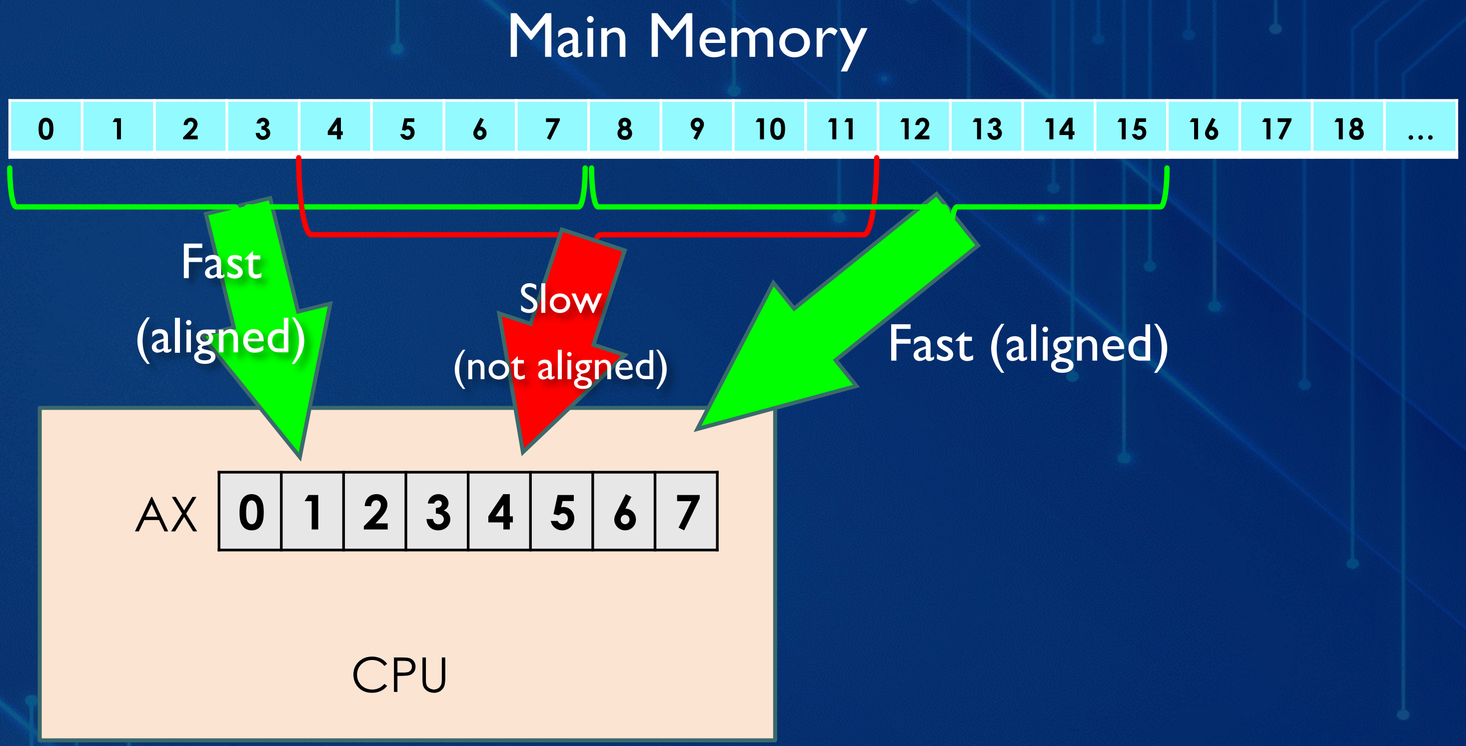
How big is a structure?

- Ah ha! The size is:
  - 36!
- Well, no…
  - What!?
  - Structs must be aligned based on the size of pointers…
    - 8, 16, 24, 32, 40, etc
  - So…
    - Need 4 more bytes of padding… giving the final answer as:
  - 40!

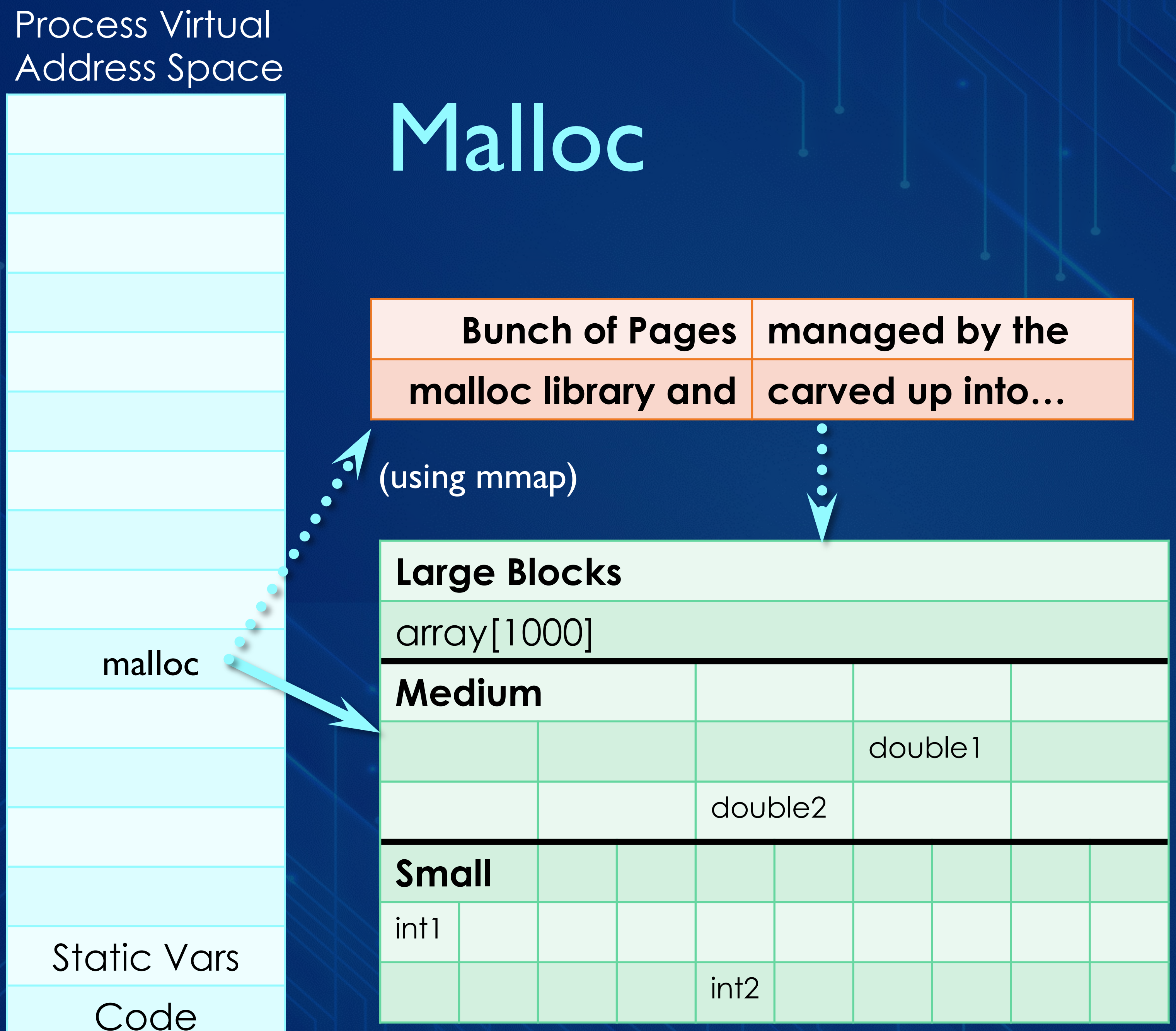
```
struct Data {                  Bytes    Address
      bool     b1;             // 1     209808
                               // 3 bytes of padding
      int      i1;             // 4     209812
      bool     b2;             // 1     209816
      bool     b3;             // 1     209817
                               // 6 bytes of padding
      void * p;                // 8     209824
      bool     b4;             // 1     209832
      bool     b5;             // 1     209833
      bool     b6;             // 1     209834
      bool     b7;             // 1     209835
      bool     b8;             // 1     209836
                               // 3 bytes of padding
      int      i2;             // 4     209840
};                             // 4 bytes of padding
```

Copying Data From Memory to CPU

- `mov rax, [0]`
  - Copy 64 bits from address 0 to the rax register
- `mov rax, [4]`
  - Copy 64 bits of data from address 4 to the rax register
- Hardware dependent alignment requirements and effect on speed.



Malloc

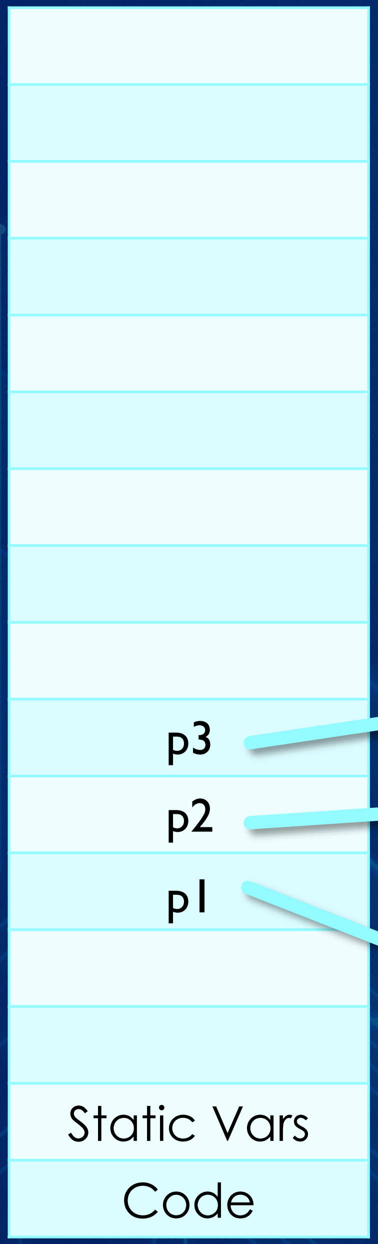


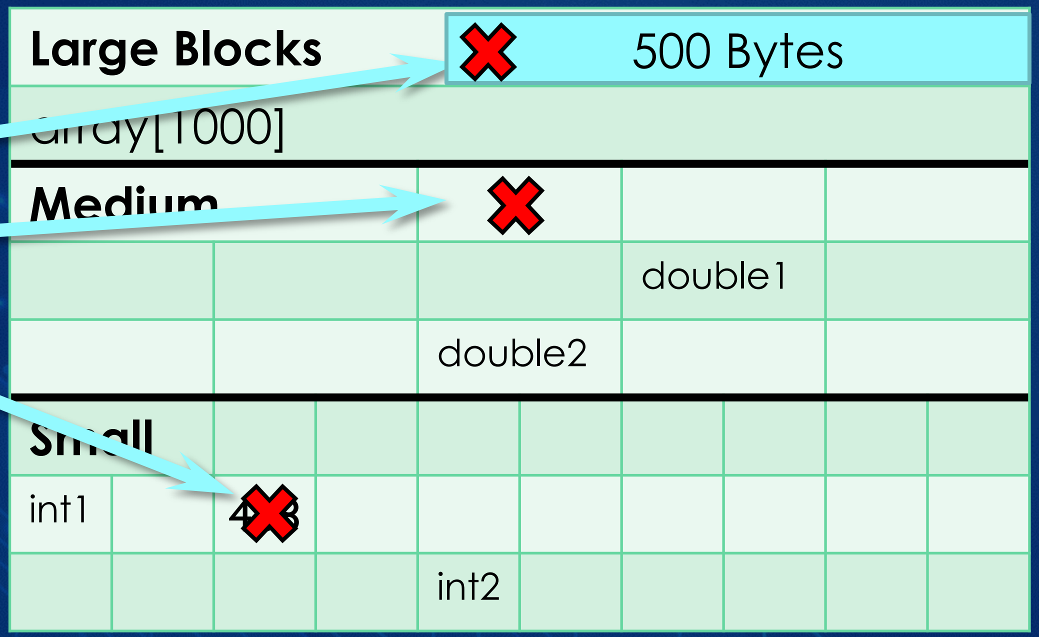
Malloc Library Memory Pool

- How does it work?
  - Malloc (library) grabs a bunch of memory and then handles it for you
    - ie, a bunch of pages that it gets via mmap
  - [Mmap gives you exactly what you asked for* and you manage it]
- So on first call to malloc (or more likely as the program initially begins to run)…
- Which the malloc library breaks up into…

Malloc

Process Virtual Address Space





Malloc Library Memory Pool

```
int     * p1 = malloc( 4 )
*p1 = 468;
double * p2 = malloc( 8 )
char * p3 = malloc( 500 )
```

- The malloc library remembers each of these addresses (and the size associated with them) so that when you call `free()`, it knows what to do.
- And as good memory citizens…

```
free( p1 )
free( p2 )
free( p3 )
```

- Malloc library has reclaimed that memory to provide to the program in the future.

mmap()

- System call to manipulate address space
- Map anonymous pages to add heap space
  - Not really “adding” space to the heap because the virtual space already exists… mmap is more allocating / mapping the (currently empty) virtual address space to actual (physical) memory pages.

mmap()


```
void* mmap( void *addr, size_t len,
            int prot, int flags,
            int fd,   off_t offset);
```

- addr is the pointer you'd like to get back. If this is 0 (null), then OS will just give you a pointer to some chunk of memory. Alternatively, can use as a hint to OS about where the page boundary starts.
- len is the number of bytes you want to allocate **(beware: it will rounded up to a multiple of the page size!)**
- prot describes protections on the chunk of memory, a combination of PROT_READ, PROT_WRITE, and PROT_EXECUTE
- flags describe properties of the memory (backed by a file, shared with your child on fork(), etc) private, anonymous
- fd is the file descriptor for the file that we're using to back this memory (-1 for none)
- If we want to map the middle of the file, we can specify a nonzero offset

mmap() flags…

- Need to request read / write pages…
  - Why?
    - Because anything that calls malloc() (and thus your allocator) will want to write data to that memory and read the value from that memory.
  - Use `PROT_READ` and `PROT_WRITE`…
    - How do you provide both of these flags in a single parameter (`prot`) of mmap?

`PROT_READ` `PROT_WRITE` are actually (something like):

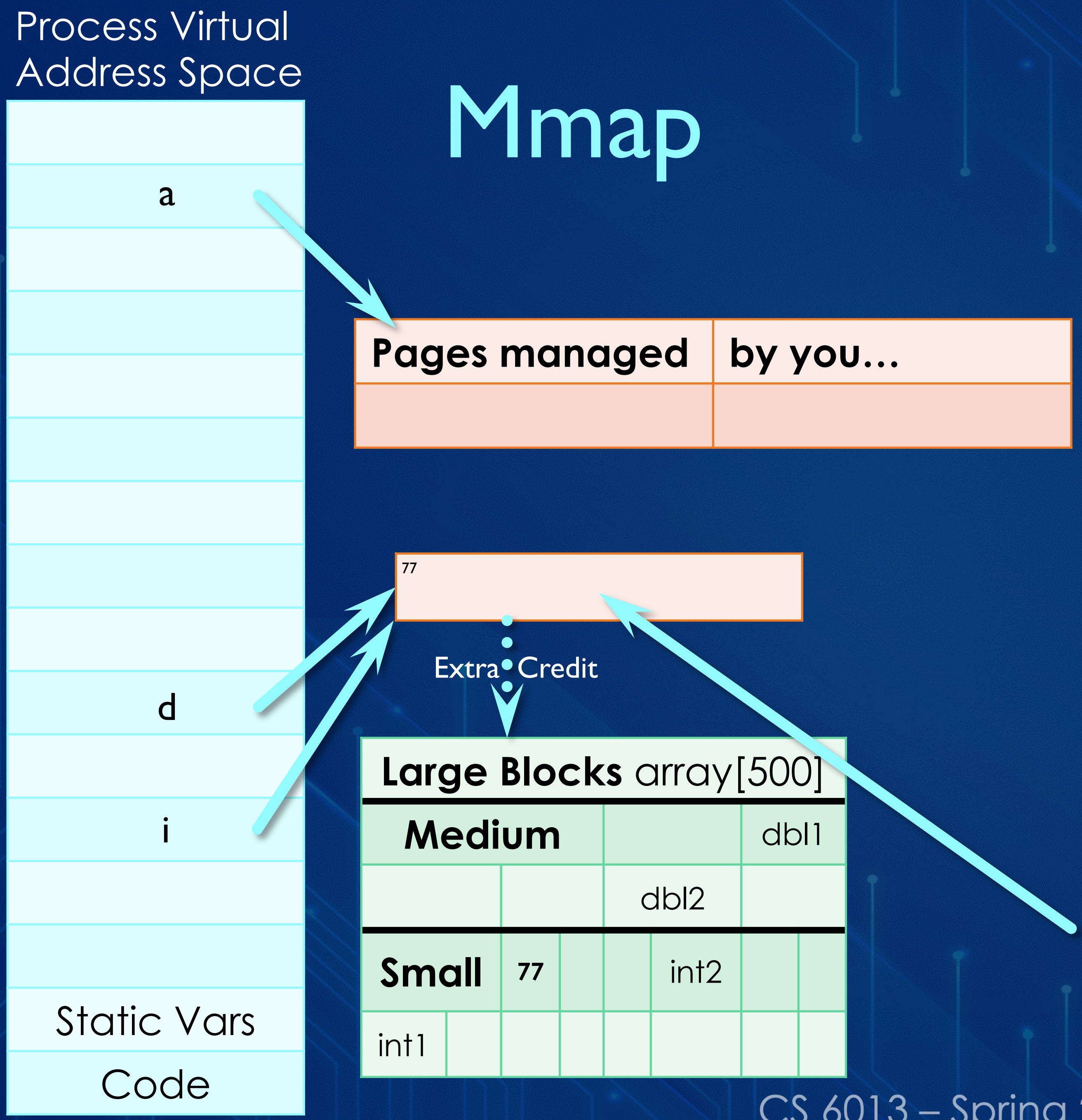
`0010` `0001`

- Combine them using the bitwise-or operation:

`PROT_READ | PROT_WRITE` // Which actually means "*and*" in this case…

`0011`

Mmap



- mmap() will returns the address of one (or more) pages in memory (depending on how much memory you asked for).

```
void * a = mmap( nullptr, size,…)
```

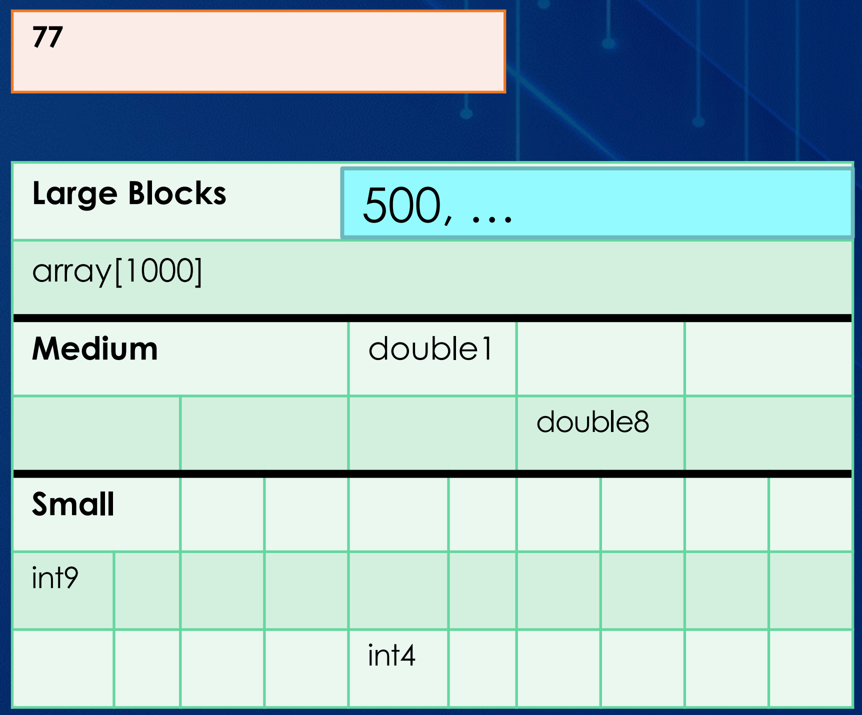
- Note, size should be equal to X * Page_Size
  - In other words, you should only allocate in multiples of a page. Why?
  - Because this is what mmap() is going to return to you anyway.
- `int * i = malloc( sizeof( int ) )`
  - Becomes (in our assignment)?

```
void* d = mmap( nullptr, 4096… )
return d
```

- `*i = 77`
- Lots of unused space – but we aren't going to worry about that. (See Extra Credit)

Opposite of mmap?

- What is the opposite of `ptr = malloc( size )?`
  - `free( ptr )`
- What is the opposite of `ptr = mmap( … size … )?`
  - `munmap( … )`
  - What parameters does it take?
  - `munmap( ptr ) // ?`
    - Almost but not quite…
  - `munmap( ptr, size )` // Yes, both ptr and size!!!
- But `free()` doesn't take size as a parameter?! How does this work?
  - The malloc library remembers it for us…
- Which brings us to your hash table…



Hash Table and Memory Deallocation

- Our allocator declares its deallocate method as:

  `myAllocator.deallocate( void * ptr )`
- To give back the memory, why do we need to know both the size of the data and the pointer (address) to the memory?
  - Because `munmap()` needs both of those pieces of information!
- How do we know the size of data that `myPtr` represents (when we go to deallocate it)?
  - `delete( myPtr );`
  - We have to save it when `myPtr` is originally allocated.
- So we store it in the hash table
  - This is the entire point of having the hash table.
- We are keeping track of all the allocations we do (via the hash table) so that when we deallocate memory, we can look up the size and use the size as a parameter to `munmap`!

The Hash Table's Table

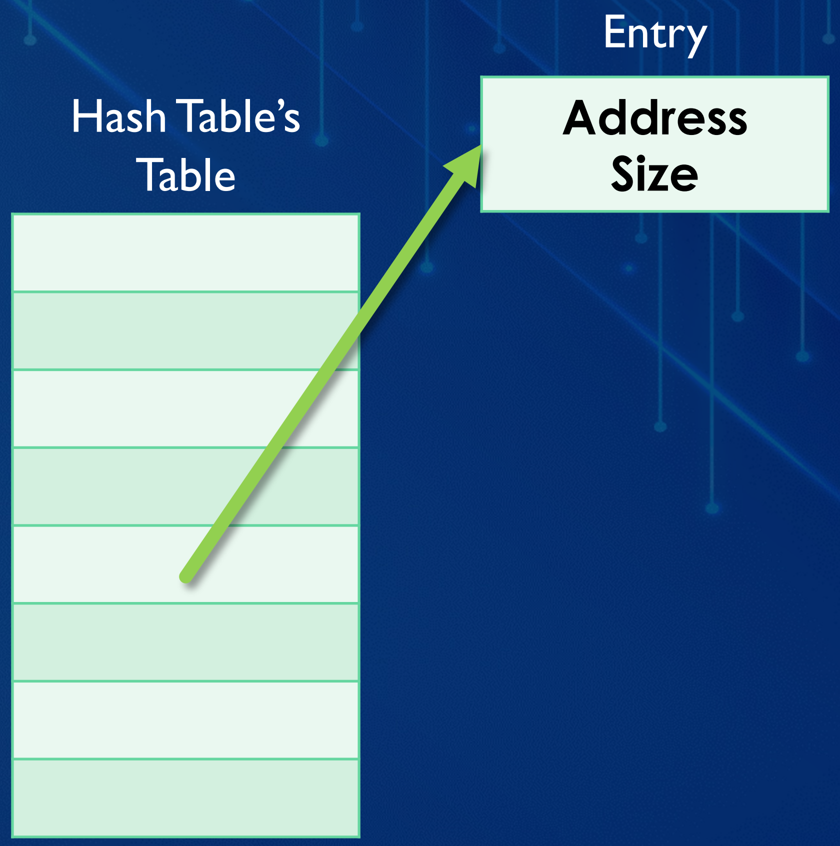
- What data are you storing in your hash table?
  - The size of memory that was allocated (and the key is the address of that memory).
- What does the hash table's table actually look like?
  - Well, we know that it is just an array.
- 1) What is the importance of a hash function? 2) What hash function are you going to use?
  - 1's answer) To minimize the number of collisions
  - 2's answer) What about:
    - `int hash( void * ptr ) { return ptr % table_size; }`
    - Good or bad?
      - Probably bad…
  - Try different ways of generating the hash… do some timings and let us know what you find.

The Hash Table's Table

- What happens when the table becomes (too) full?
  - Make a bigger table!
  - Specifically, when does this happen?
    - When a new entry (a new malloc) is to be added to the table.
  - How do we grow the table?
    - malloc(size * 2)?
      - What happens when we call malloc?
      - Infinite loop! Crash and burn…
  - So how do we grow the table?
    - `mmap` a new table
      - 2x the old size
    - copy the old table into the new table (rehash)
    - `munmap` the old table.

The Hash Table's Table?

- The hash table's table is just…
  - An array
- What does it look like?
  - Entry *[100]?
  - How did you create the new Entry?
    - new Entry()?
    - What does new do under the covers?
      - malloc…
    - What happens when malloc is called?
    - Infinite loop – crash and burn!!!
- So, what must it look like?



The Hash Table's Table

- ~~Entry*[100]~~
- Entry[100]
- What values are in the table to start with?
  - Address?
    - Something that means this entry is not currently in use…
    - nullptr seems reasonable
  - Size?
    - Hash table is keying off the address, so if the address is invalid, it really doesn't matter what size is… however, perhaps 0.

Hash Table's Table

Address Size
nullptr 0
nullptr 0
nullptr 0
nullptr 0
nullptr 0
nullptr 0
nullptr 0

Inserting…

- insert(ptr1, 8)
  - ptr1 hashes to location 5.
  - What does the table look like now?

Hash Table's Table

0	nullptr 0
1	nullptr 0
2	nullptr 0
3	nullptr 0
4	nullptr 0
5	nullptr 0
6	nullptr 0
7	nullptr 0

Inserting…

- insert(ptr1, 8)
    - ptr1 hashes to location 5.
    - What does the table look like now?
- insert(ptr2, 4)
    - ptr2 hashes to location 5
    - What does the table look like now?

Hash Table's Table

0	nullptr 0
1	nullptr 0
2	nullptr 0
3	nullptr 0
4	nullptr 0
5	ptr1 8
6	nullptr 0
7	nullptr 0

Inserting…

- insert(ptr1, 8)
  - ptr1 hashes to location 5.
  - What does the table look like now?
- insert(ptr2, 4)
  - ptr2 hashes to location 5
  - What does the table look like now?
- delete(ptr1)
  - What does the table look like now?

Hash Table's Table

0	nullptr 0
1	nullptr 0
2	nullptr 0
3	nullptr 0
4	nullptr 0
5	ptr1 8
6	ptr2 4
7	nullptr 0

Inserting…

- insert(ptr1, 8)
    - ptr1 hashes to location 5.
    - What does the table look like now?
- insert(ptr2, 4)
    - ptr2 hashes to location 5
    - What does the table look like now?
- delete(ptr1)
    - What does the table look like now?
- Now look up ptr2…
    - What happens?
        - Hash to 5… What is at 5?
        - This is a problem!!! How to fix?
        - Need to know that Entry #5 was previously used…

Hash Table's Table

0	nullptr 0
1	nullptr 0
2	nullptr 0
3	nullptr 0
4	nullptr 0
5	nullptr 8
6	ptr2 4
7	nullptr 0

nullptr
-1? 0?

Inserting…

- Now look up ptr2…
  - What happens?
    - Hash to 5… What is at 5?
    - This is a problem!!! How to fix?
    - Need to know that Entry #5 was previously used…
      - Mark it some how…
        - In this example, I used a `-1` "address"
      - Alternatively could add a "deleted" flag to the Entry
        - Though this "wastes" a lot of space.

Hash Table's Table

0	nullptr 0
1	nullptr 0
2	nullptr 0
3	nullptr 0
4	nullptr 0
5	-1 0
6	ptr2 4
7	nullptr 0

Finding…

- On this slide I've written "`ptr101`". What is actually in the able there?
  - The address that `ptr101` is pointing to
    - `0x29ef47a0`
- Is `ptr7` in the table? It hashes to 5.
  - How do you know (determine) if it is in the table?
    - Must "wrap" around to continue looking
    - If you hit a `nullptr`, it is not
  - What is the other case in which ptr7 is not in the table, but you don't find a `nullptr`?
    - Hint: There are no `nullptr`s in the table…
      - Answer: If the table were to be completely full.

Hash Table's Table

0	ptr101 8
1	ptr999 1000
2	nullptr 0
3	nullptr 0
4	ptr19 50
5	ptr2 4
6	ptr222 4
7	ptr123 120

Malloc Assignment

- Do not use:
  - System `malloc()`, `delete()` :-p
    - Except for testing / comparison to your version
  - c++ vectors, etc
  - `new()` / `free()`

Replacing malloc / free

- `malloc()` is just a function.
- We can overload (well, override) it just by creating out own:

```
void * malloc( size_t size ) {
    // What do we do in here?
    void * mem = myAllocator.allocate( size );
    return mem;
}
```

- And what about `free()`?

```
void free( void * ptr ) {
    // What do we do in here?
    myAllocator.deallocate( ptr );
}
```

- What is myAllocator?
  - An object of type of the class you are writing.
- Where does it come from / how to I create it?
- Use a static variable:

```
// Somewhere near the top of main progam…
static MyAllocator myAllocator;
```

- What is a static variable?
  - One that always exists and is created (constructed) before main() begins.
- Note: when our `malloc()` and `free()` are linked in, they will replace the default ones provided by the malloc library / compiler.
  - You may want to put a debug print statement in your malloc() to verify it is being used. Once you've done this, you can remove the print statement.
- Note: Some of (debug) functions may not work correctly if they use malloc() internally.

Timing


```
#include <ctime>
// clock() returns the ~amount of CPU time a process
// has used.
clock_t start_time = clock();
// Do some work here…
clock_t end_time = clock();
float time_used = end_time - start_time;
float cpu_time_in_seconds = time_used / CLOCKS_PER_SEC
```

~ Fin ~