Agenda for Today

1. Bug of the Day
2. Assignment and Lab Q&A
3. Address Translation on x86 (32-bit)
4. The Clock Algorithm
5. Wrap Up
Bug of the Day
What's wrong here?

```c
int* make_array() {
    int arr[4] = {1, 2, 3, 4};
    return arr;
}
```

- `arr` is a pointer to space on the stack — that’s where functions keep their local variables.
- It’s impossible for the caller of `make_array` to use the pointer it returns safely — it points to deallocated space.

`arr` has type `int[4]`, which is silently converted to `int*`.
If we use a dynamically-resizing array, do we have to shrink it after takes?  
For this assignment: no. You can leave it at full size, or shrink it if you want.  
More generally, that’s tricky and depends on context.

Can we use a global variable to point to our linked list? 
No, you should not use any global variables in your solution.
Queue Assignment Q&A
My group has created the page header with a size and magic number. How do we access those inside of xxmalloc_usable_size(void* ptr)? We’re getting segfaults.

You need to round ptr down to the next lowest multiple of PAGE_SIZE to find the beginning of the page. That location is a pointer that you can cast to your page_header_t struct. If you didn’t use a struct, you’d cast it to an int* you can use as an array, but I don’t recommend this. Using structs is usually cleaner.

If you’re getting a segfault, you **must** be rounding incorrectly. If ptr points into a valid page, the beginning of that page has to be a valid address to access too. Make sure ptr is not NULL, and make sure your mmap call that maps that page is working.

**REMINDER:** You MUST check the return codes from all POSIX functions unless I specifically say you can skip it. Check mmap, mremap, mprotect, etc.

**Hint:** Print pointer values with the %p flag (remember to use snprintf).

0x7FFFFFFF0085000 <- that’s a 4096-byte aligned address
0x7FFFFFFF0085018 <- that’s not.

The last three hexadecimal digits are zero when aligned to 4096.
If we create a page to hold 16-byte objects, then we run out of 16-byte objects, do we create a new page for a 16-byte objects?
   Yes! And make sure you put a header on that new page. Populate the freelist with all the open spaces in that page. You’ll return one of the objects from this new page, but the rest end up on the freelist.

When an object is freed, do we just put it at the front of the freelist without worrying about which page it came from?
   Yes, exactly!

If an object is larger than 2048 bytes, do we just give it whole pages like the starter code did for all xxmalloc calls?
   Yes, exactly! We’re kind of cheating here (realloc for large objects won’t work) but we can get away with it for small programs.
Will we end up with multiple headers?
Yes, every page will have a header. You may have many headers for pages that hold 16-byte objects, for example. The reason you put the header at the start of each page is so you can locate and read it later when an object is freed or passed to xxmalloc_usable_size.

Be Careful: don’t return a pointer to the header when a program calls your xxmalloc function. You don’t want the program writing to this space.

When we create our array of freelists, do we use malloc?
No, you should use a global array. Assume malloc does not exist when you write this code. Plus, you know the size ahead of time and it really is global state, so it should be a global.
Address Translation on x86
Fun Facts

- $2^{10} = 1024$ & $10^0$
- $2^{20} = 2^{10} \times 2^{10} = 1024 \times 1024 = 1 \text{ M}$

How do we map a new page?

1. The kernel chooses virtual address $V$, and physical address $P$. We're going to add to the page dir and tables so the virtual page at $V$ maps to the physical page at $P$.

What's in a page directory/table entry?

<table>
<thead>
<tr>
<th>Top 10 bits of a physical address</th>
<th>...</th>
<th>Dir?</th>
<th>Access</th>
<th>Write-Back</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>12 11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Adding 12 0 bits to the end gives us a page-aligned address.
At physical address $T$, page table

$V_{	ext{tebinder}}$

At this point, any access to virtual address $V$ maps to physical page $P$. 
The Clock Algorithm
This example didn’t work out as I’d expected, so we’ll try a different one in class on Friday. Sorry!