CIS3110 - Lecture 2 - Complete Improved Transcript (2025-01-16)

Assignment 1 Clarifications

Alright, welcome everybody. I want to start by answering some questions that have been percolating in the email accounts regarding the assignment.

First, let's clarify some misunderstandings about the ID numbers. These numbers are not sequential indexes or location indexes - they're just IDs. Think of them like jersey numbers on a team of players. The numbers are all different for different players, but it's not like if Player 11 enters the field, that means players 1 through 10 had to go out beforehand. It just means we've called this particular allocation "11" so that later when we talk about 11 again, it's the same allocation.

Don't assume the IDs will come in any particular order, but you can assume that every time you see the same number again, we're talking about the same allocation. It doesn't make sense for us to request the same allocation to be allocated twice or freed twice. In the testing files, if we have allocated 11 and asked for 11 to be freed, we'll never talk about 11 again. We might have new allocations, but we'll give them new numbers, not necessarily in order, but they will be distinct.

This helps because one thing that can happen in your code is we could ask for, let's say, allocation 11 to be set aside, and your answer might be "no." In that case, you don't have an allocation 11 in your management structure because the answer was no. So if a free request comes for something that doesn't seem to have been allocated, make sure your code doesn't crash. If we ask for 11 in a given testing file and then ask for 11 to be freed, but you said the answer was no, you just don't have that allocation around. So if a free request comes for an ID that doesn't exist, don't crash. I'm not particularly concerned if you have an error message, just make sure it isn't a fatal issue for your program.

Another question I keep getting is whether I really mean there should only be one call to malloc in the program. Yes, because we're modeling the idea that you have one memory board. It's a physical property of the computer you're modeling, and this happens in reality.

If you have a small, simple computer - not your phone, not your Apple Watch, certainly not your laptop - but perhaps the ABS system in your car, or the throttle controller, or some simple embedded computational device like on a security camera for your house... there are millions of these things around. This is Internet of Things. There are lots of little simple processors around. They have a small amount of RAM installed by the factory and shipped out.

If there's code running on it that needs to do allocations and frees, which is quite common, they're going to have code exactly like what you're writing. They're going to say, "Please, memory subsystem, I need some memory. I need 123 bytes of memory." So it's going to call into your code saying, "I'm going to make

an allocation," give it some ID, and let's say it asks for 123 bytes of memory. Then your system looks around and says, "Do I have 123 bytes of memory that I can give out?" Your system is going to want to remember it gave that out so that later, when it's freed, you know how much memory was allocated (because the free doesn't tell you, just like the regular free that you've been using in C - you just say "free this allocation" and you expect the memory subsystem to remember all the details about that allocation).

That means for your system to work well, if the request is, let's say, 1231 bytes of memory for the user to use, your system is likely going to need some additional bytes somewhere for it to use to keep track of this. So more than 1231 bytes is going to be required. How many more and how they're set up is 100% up to you.

You could make a little linked list, you could make a table, you could make a hash, you could make whatever you want, but you're going to need a structure so that later, when I say "free allocation 11," you can look it up in your structure. You can know what got given out so that you can then say, "Oh yes, I can handle the freeing of allocation 11."

So there's going to be an overhead. If you think about it, this means if we run the user program with a 4096-byte memory allocation, and then the first thing we request is an allocation of 4096 bytes, what's going to happen? Are you going to serve one allocation of the entire memory back? If you need an overhead to keep track of that allocation, you can't give back 4096 bytes of a 4096-byte total pool because your overhead isn't going to fit in there.

Maybe you need 8 bytes of overhead, in which case the most you could ever give back from a 4096 pool is 4088 bytes, and then your allocation and your overhead would totally fill all the memory, and you could never make another allocation. But you could at least service the first one.

I'm not asking you to do the management in any particular way, but you are going to need to do the management. This is needed for the later part where I talk about merging memory. If there are two free chunks adjacent to each other, they should be merged.

When I say a "free chunk," once we've done an allocation, you can think of memory as being divided into chunks - the one you gave away and the one you haven't given away yet. One way to look at this assignment is to say, as an initialization of the system, you have one chunk that's free. Later you're going to chop chunks off it. That's one way to proceed. I've seen different approaches from students in the past.

If you have two chunks that have both been freed - let's say we ask for four pieces of memory (1, 2, 3, 4), and now I free up Chunk 2, so I've got an allocated portion, then a hole because I freed up that chunk, then another allocated portion, and the final allocated portion. If I free up Chunk 3, I now have two freed chunks beside each other.

Once you've got the basic system working, what I'm asking for here is that if you're creating a second free chunk that is adjacent to another free chunk, merge them together. This is because if you've asked for a 10-byte chunk, a second 10-byte chunk, a third 10-byte chunk, and a fourth 10-byte chunk, and you free up the second and third ones, and then ask for a 15-byte chunk, you should have a big enough hole there to satisfy that slightly larger allocation.

There are only a few cases to consider since memory is one-dimensional, so there's only a few different adjacencies that can happen with these chunks.

That's the final wrinkle once you've got everything else working. We're all experienced programmers now, so before you do the final wrinkle, I assume you're going to use the copy command to save your perfectly wonderful working code before you possibly turn it into not-perfectly-wonderful working code. If you need to back up, you want to be able to go back to your copy.

Another property of these systems is that if we allocate two dozen different little pieces and then free all those pieces, we would like a well-behaved system to be in effectively the same state it was at the beginning. Like putting all your slices of bread back into one big loaf. This magical loaf, where if we can merge the slices back together, we've got one unsliced loaf ready to go, just as it was at the beginning. Same deal with real malloc and free - if we free everything back, we say "I have no memory leak" because I've given everything back, and the system is in the nice simple state it was at the beginning.

Processes and Threads (3.1, 3.4, 3.5, 4.1, 4.2, 4.3, 4.6)

Let's talk about critical sections. One thing that has confused a few people is what we mean by "thread." The thread is a bit of a metaphor. If you were to take your source code, print it out, and then get a pen and draw on your source code where the instructions are getting executed as you ran it - you'd start at main, come down through main, come to an if statement, maybe in the if statement you don't do the if part but jump to the else, skip over the if part, go through the else, come to a loop, go through the loop, come back up to the top of the loop, and maybe do that again - you're going to get this ink on the paper looping around. That ink on the paper is visually reminiscent of a thread that might be in a garment.

This is what we mean by a thread. It is the instructions executed in sequence by a particular path through a running program. All of the programs you've written so far have one thread per program. When a program is running, there has to be a thread defining which instructions are currently being executed.

What we'll find is that there are multithreaded programs. If there's no thread, that just means the program isn't running, which is uninteresting for an operating system because it's not operating. If there's a thread, however, that means the program is doing stuff.

If we have a computer whose operating system supports multiple processes at once (like every computer you're doing any work on), you can be running more than one program simultaneously. Maybe you've got

a clock running, a text editor running, and a terminal running in another window. All of these things are running at the same time.

Let's say each one has one thread. If you've got one CPU, it's doing this dance of running a bit of one, then running a bit of the next one, making it look like everything is running at the same time, even though you may literally only have one processing unit in the system. We will talk later about how the system decides which one is being run at what time.

What's important for our discussion right now is that the code at the level of trying to manage a critical section has no control over which thread is going to start running when. We also know that if there's an interrupt, what essentially has happened is we've created a new thread to serve that interrupt - one that will disappear as soon as the interrupt handler ends, but that's going to potentially throw everything else off.

So multiple threads of control in potentially many programs, all at some point in their execution, and you want to make sure that if any of them share data, you don't get data corruption because one thread started executing, got interrupted, and a second thread came in and messed with the same data structures while the first one was paused.

That's really what we're talking about here - how to make sure that one thread can't step on the toes of another thread.

Synchronization (6.1, 6.2, 6.4, 6.5, 6.6)

What's the difference between a multithreaded process and multiple processes? Does the operating system make a distinction between them? Yes, and we'll have a whole section on that. So far it sounds pretty similar - different processes (different programs), each of which probably only have one thread in them. But we'll find out it is possible for a process to have multiple threads in it. That's going to be another juggling act we need to control.

What we're going to talk about right now are the tools that will let us control all of these kinds of juggling acts. What we've seen in the device driver is the management for the cooperation between the read and write routines, which are functions called out of the kernel. They're called in kernel mode. They talk to shared data structures that are also managed by the interrupt service routines (the bottom half of a given device driver).

There's a different interrupt service routine for each type of device. The example I've given you includes the read function and the interrupt service routine to support the read function for a single character device, similar to what you'd have for a keyboard or a mouse. Those are both very simple devices because they operate a character at a time, and you can't write to your keyboard, so you don't generally have a write routine. This means the interrupt service routine is pretty simple.

If you had a device where you can write to it, the interrupt service routine would need a way of passing data into the device as well as pulling data out of the device. But this is a nice simple example where we've got one shared data structure, the circular queue, that is shared between the read routine (which is going to consume data from the queue and pass it back as a result of having called read) and the interrupt service routine (which is going to put data in the queue).

We're going to call this a "producer-consumer pattern." A producer-consumer pattern means we'll have one entity running its thread producing data, putting it in a shared data structure, and we'll have a different entity with a different thread consuming data, pulling it out of the shared data structure.

The key here is that SPL_TTY and SPL_X handle the masking. SPL_TTY marks the beginning of the kernel-level critical section (this is actual code from FreeBSD). SPL_TTY says "I'm starting the critical section. No one else can come in." If anyone else calls SPL_TTY after the first thread has called SPL_TTY, it's going to block.

I didn't give you the code for the internals of SPL_TTY and SPL_X, but SPL_TTY will itself use tsleep to go to sleep because of a critical section. So SPL_TTY means "I'm the only one that is going to be running playing with the queue."

You enter the read here with SPL_TTY, you exit at the end of the function with SPL_X. If you need to go to sleep in the loop, we leave the critical section by calling SPL_X, put ourselves to sleep, and then when we wake up, we reenter the critical section by calling SPL_TTY again.

So anytime between an SPL_TTY and an SPL_X, you can assume that no other code, whether it's a second call to read in a different thread or an interrupt, is going to execute because it's going to hit SPL_TTY and go to sleep.

These are functions inside the kernel. While they exist and they're defined, you do not have permission to call them. They're only available when in supervisor mode. All the programs that you're going to write and run on the computer run in user mode.

So if you want to read data, you do that by setting up a trap into the kernel. You call read, or let's say you call scanf, which internally calls read. We have a trap. We go from user mode to kernel mode. Our thread is now running in the kernel in a privileged state. Because it is in a privileged state and running code that is currently part of the operating system (and therefore isn't any old code that a random person could be putting up), it's allowed to do special things.

One of these special things is to call the device driver read routine, so it would call mouse_read or keyboard_read or disk_read. I just said dev_read here as a placeholder for some kind of device read. As a privileged routine, you're then allowed to call SPL_TTY, tsleep, etc. But these are functions inside of the

source base for the kernel and the kernel only because you don't have permission to do the things they do.

If you tried to call tsleep, you would immediately get a permission error because you're not allowed access to the data structures that tsleep is trying to use. They're not even in your program - they're in the kernel. So you have to jump into kernel space. It's like we've hit ourselves with the magic wand, and now we're in a different part of the code on the computer, running as a privileged process, and therefore able to do different things.

When the privileged process has gotten the data that the user code requested, it's going to return back across the trap. Your call to read will then return. In the code you've written, you've all written code like this - you want to read some data, so you read from a file or from the keyboard. You didn't know that this magic trick was going on. You called read - "I want to read some bytes from standard in, what does the user have to say?" You called read, which trapped into the kernel, did all this work, waited for the bytes to show up, populated them in the buffer that you told read to fill up, and then returned back into the user space program. All you knew was that you called a function that seemed to take a while because the user had to type some data on the command line before it came back.

So these kernel functions exist, but they're not available to user space code.

Process Management (3.1, 3.4, 3.5)

Going back to the question about execution models: when you have a running program, we call it a process. The execution of a program is a process. We call it a "process image" when we're talking about the memory associated with the given running process, as well as all of its other data.

The process image consists of the map of all of the memory - the data that is in memory defining your running program. It has three big sections. You've probably seen these in other courses:

1. The **text segment**: All of the actual machine instructions making up your program. If you wrote assembly, ran your assembler and got machine code, that's the text. If you wrote C code and you ran a compiler, then it wrote the assembly for you and assembled it, it's still the text. So the program text is all of the instructions.
2. The **data segment**: Historically, this is the older pool. It's called the "data segment" because when we first started doing this before we had the ability to do recursion, you just had the program instructions and the data they were going to play with. This is why the data segment sits right on top of the text segment. The text segment isn't going to change size once you've compiled your program. When you run your program, you're going to load all of the machine instructions into memory, but you're not going to get more machine instructions while your program is running. It's the version of the program that you ran, with the same instructions the whole time. Therefore, they just slot the data right on top of the text. In the data segment are things like:

- Constants (all your strings are there)
- Any constants that you've initialized integer variables to
- The static segment (any variables declared as static)

Static variables have the lifespan of your program. When your program starts running, memory for static variables gets set aside, and that memory stays allocated to your program until your program exits. So even though you might have a variable that is a static variable inside of a particular function, the reason it has the same value when you come back into that function is because it was allocated on the data segment when the program was put in memory in the first place, and only gets updated when you write to it yourself. Global variables are just static variables that have no scope. So they're also in the static data segment. The difference is that you can talk about them from any source file in your program, whereas one that is marked static can only be talked about either within the function it was mentioned in or the source file it was mentioned in. This means you can have two different static variables with the same name, as long as they're in different source files. That's perfectly fine. You can't have that with globals, because the whole point of a global is that it is global, so it has the same name no matter who's talking about it. If we look at the device driver code, you'll see that I've declared our queue as static. The way these are written is as static variables in the file that is the device driver code for that driver. So you'll have a queue for the disk, a different queue for the keyboard, a different queue for whatever other devices you might have on your system. Another thing that's in the data segment is of course the so-called **heap**. I say "so-called" because there's a data structure called the heap that has absolutely nothing to do with the heap we talk about in the data segment. The heap in the data segment is what we access via malloc. It manages dynamically allocated memory. We use malloc (and free) to access the data segment. Malloc is going to allocate more memory on the heap. In our operating system-based multi-processing system, unlike the throttle controller we're building for Assignment 1, we have the opportunity to make the heap bigger. We can actually make more allocations happen. This is why I have an arrow coming out of the data segment. The top thing in the data segment is our malloc-managed memory, so we can grow it upwards. If we decide we want more allocation, we can essentially do what everybody seems to want to do in Assignment 1 - we can say, "Yeah, allocated RAM of 4096 bytes, but now I'm just going to make it bigger. I'm just going to push the end of RAM out." But notice we're not calling malloc again. We're assuming we know that there's empty memory there. We're just going to move the top of memory up. We actually have a pointer that tells us how far up in memory the heap goes, and if we make it bigger, we move that pointer up.

3. The **stack**: So why do we need a stack if our heap would solve all our problems? The stack is for recursion. They added the stack to the process image so that we could recursively define what's going on in a function. If you have a function with local variables, you need storage for those local variables. If that function calls itself, you need different storage for a new copy of variables with the same name. So the whole point of the stack is that as you call from one function into another function, whether

you're doing recursion or not, it allocates additional memory growing downward in memory for storage for those variables. If you call the same function again, it's just going to get wound onto the bottom of the stack. You're going to get more memory set aside for those variables. You can play with them independent from their old definition further up the stack. And when you return from a function, we wind the stack back up. So when starting in main and when you end in main, the stack has potentially grown very deep and shrunk, and now it's essentially back in the state it was before. Older programs, notably Fortran, COBOL, etc., didn't support recursion. Fortran programs only got recursion in the late 90s. C was already quite popular by that time. Before that, they simply had what they called the "post box plan." You just had a chunk of the data segment for the variables for a given function. You put the data in that place, you ran your function, and when you returned, they were no longer used. You entered the function again, and you just reuse the same memory, which is fine as long as you're not recursively calling into the same function a second time.

The pattern was that the top of memory (hexadecimal all F's or -1 in two's complement) is where the stack starts, and the bottom of memory is 0. So you can think of the stack as growing down from the top, or you can think of the stack as bolted onto the bottom of the text segment.

The text segment is the one thing that isn't going to change - it's essentially the anchor point for everything else. We've got our fixed-size text segment. We bolted the data onto the top of it. We've effectively bolted our stack onto the bottom of it. And now because we've got a fixed-size address space (because we're on a 32-bit, 64-bit, or whatever architecture), we've essentially got a big circular space.

It is possible that if your stack grows really far, it could slam into the data segment. And the data segment, if it grew really big, could slam into the stack. There's no way we can say, "Well, maybe I just want to grow the stack the other way," because it has to go down. The top of the stack is 1 byte below the bottom of the text. There's literally no other space in our number line, so we have one gap.

We've got a fixed-size text block, two growing data blocks (one called data, one called stack), and we're hoping that they don't run into each other. In the days of 16-bit architecture, they ran into each other pretty often. With 32-bit architecture, you've got 4 GB of space, so they didn't run into each other very often anymore, but it was still possible. Now we have 64-bit architectures. Now the gap is so big it's terabytes in size, so the likelihood of the stack recursing out of control and hitting the data segment is pretty much zero. The likelihood of you putting data on the data segment and running into the stack is also pretty much zero, so we can do lots of fancy tricks with loadable libraries.

That's all of our memory needs solved. But there's no I/O. We don't have zero page here. This is in what we're going to call a "virtual address space." So these numbers, while they allow us to talk about where we are in our running process image, are all virtual. It's a map on top of the real RAM.

Any program can grope around as long as it likes inside of that virtual space, and it's never going to be able to accidentally hit any of the I/O ports. And what we're going to find out is that it's never going to find any other programs because they're all in their own private space. So we have our virtual simplified address space, and then we also have all the information we need for our process context.

This is "context" as defined in the phrase "context switch." So everything that we needed to do for the context switch is part of our process image - room for all the registers, stack pointer, everything. Part of the process context is going to be a few other things we'll talk about in a second. But all of that is managed in what we call a "process table entry."

So in the kernel, there's a table of all of the running processes that the kernel is managing. Each process table entry talks about one process. If there's more than one thread in the process, then the process table entry may talk about all the threads, or as Linux has decided to do, they've just allocated some extra entries and said, "Yeah, I'm attached to my buddy over there."

A couple of different ways they get around that, but because every process has at least one thread, if it has only one thread, that's handled in that one main process table entry. If there are additional threads, we may have some additional entries pointing at the main entry, or the entry (as Sun does) may just have a table for all the threads.

Every entry has to have all of the information for a given running process, so it has to have:

- Space for the program counter
- All the registers
- Stack pointer
- Pointer to the virtual address space map
- All the information about scheduling (whether it's blocked, whether it's runnable, whether it is running and is one of the processes actually being executed right this instant on the CPU)
- If there's a priority
- How much time it's gotten (which is important for determining the priority)

So everything needed to decide how to run a given program is either directly in the process table entry or is in this map of virtual address space that's linked from the process table entry.

Process Status and Monitoring

Here is output from PS (Process Status). PS is a Unix command that shows the status of all processes on your computer. In the PS output, you can see it gives us this nice little table:

- First column is entitled "user" - this is whoever owns the process. The user "root" or System Administrator is the owner of anything that has to do with the kernel, so you can see there's a bunch of key pieces of the program that make up part of the operating system. The last four entries here are owned by "andrew" (me).
- Process ID (PID) is the index in the table. Some of them are suspiciously low. One of them is 1 - that is literally the program that got run as part of the boot process of the computer, and it needs to manage all of the other things. These very low-numbered processes must have started when we booted the computer. Then we jump to 1609/1761 - these came along later. Various things that happened by the time root started those. But they're indexes into a table, so you might see the same index number come around again as slots get reused.
- The table historically has just been a fixed size of how many simultaneous programs can you run at one time using this kernel. Typically it's around 65,536. Recently they've upped that a little bit. But if you have too many processes, just like in your Assignment 1, the answer might be "no, you can't start a new process" because you already have too many processes running on this computer. A normal computer runs at the low hundreds of processes, so having tens of thousands would be unusual.
- There's also information about how much CPU we've used, how much total memory we've used. VSZ is the virtual size of the program. We'll come back to talk about the resident set size, which is how much of it we're actually putting in real RAM.
- TTY has to do with which keyboard/mouse pair device we have attached to a virtual terminal.
- Status is interesting in that this tells you whether the program is runnable or not. The one at the bottom that is in status "R" is currently running. You can see that the name of the program is in fact "PS", so this is the process status command itself reporting on the program status of itself, saying that it's running. The rest of them are in a status either "S" or "SW." "S" is sleeping. Every single one of these other processes is in I/O wait. They're sleeping, waiting for a device to do something.
- Extended fields will give us the channel. We also have when it was started and how many units of time in seconds we consumed of the CPU. So you can see that some of them, even though they started weeks ago, had only used a few seconds of CPU time because they spent almost all their time asleep, not wasting anyone's time because they didn't have any work to do - exactly the way we saw it in the interrupt example.

Synchronization in User Space (6.1, 6.2, 6.4, 6.5, 6.6)

In user space, if we're going to have threads or shared memory blocks between multiple processes, we need to be able to have critical sections in user code. As we mentioned earlier, we're not allowed to call tsleep or wakeup - we would need to be the kernel to do that.

So are there user space tools to do this same kind of thing? Yes, there are. The two most popular synchronization primitives available in user space to set up this dance (where multiple processes are

allowed to go or not go so that we can make sure that a shared variable doesn't get corrupted) are:

1. **Mutex**: That's essentially what we've seen so far with the mask. A mutex is a "one at a time" begin/end tool. "Critical begin" means we're entering a critical section. "Critical end" is when we come to the end of the critical section. If you've got two different threads running, maybe in different processes, and there's a shared critical section, what we're saying is it's protected - only one of the two can come in and run that code.
2. **Semaphore**: Maybe we have a more complicated sharing relationship where we want to allow more than one at a time, but we still want to limit what can happen. The example we frequently use for this is the barber shop. Maybe you go to a barber shop or a salon, and there are three chairs. It wouldn't make sense to say "I've got 3 chairs, but only one person allowed in the shop at a time." You would say up to three people can come in and get their hair done, but the 4th person has to wait until one of the chairs is free. If we have that kind of scenario, which we'll find out has lots of patterns where this is valuable, what we want is a glorified counter. We want to be able to say there's a maximum number we can allow at a time. This is what we call a semaphore. We're going to allow a certain number of entities at a time, and we're going to configure the semaphore with how many are allowed. So maybe we've got a three-chair barber shop and just down the street we've got a seven-chair salon. We're going to manage the salon with one semaphore set to maximum 7, and manage the barber shop with one set to maximum 3, but they fundamentally have the same idea - there's a fixed number of customers allowed in a given entity at a time.

We can think about the different processes coming in as though mentally we thought of them each as being on their own separate CPU. Let's say each one has its own CPU, so they're just running. To make it even more realistic, imagine the different CPUs may run at different speeds - one might go faster and then slower, but each one of our running processes has its own CPU.

Here is a mutex critical section example for producer-consumer. This is managing a queue, just like our circular queue. We have an add_queue implementation:

c

Copy

```c
void add_queue(int x) {
    int old_size;
    do {
        critical_begin();
        old_size = num_queue;
        critical_end();
    } while (old_size >= MAX_QUEUE);

    critical_begin();
    queue[num_queue++] = x;
    critical_end();
}
```

We need a loop, and we need to keep spinning in the loop until there is space in the queue. If I come in and the queue is full, I cannot add more to the queue. This is the producer, so the producer is out of luck, but we don't want the producer to crash. And we don't want the producer to get an error. We would like the producer to wait until there's space.

If all we have is a critical_begin/critical_end, we're going to have to busy-loop to do that. Notice I have a variable old_size. I go into the loop. In the loop, I go into a critical section. I copy the actual size into old_size so that I can look at old_size outside of the critical section. I don't want my whole loop in the critical section because if it is, our friend who's going to give us data could never run. So we need to go in and out of the critical section each time through the loop.

But the whole point of the critical section is that the size of the queue is the thing we can't look at outside of the critical section. That's why we make a copy, and the copy is what is evaluated outside the critical section.

So it's busy looping. This is going to burn CPU cycles like nobody's business, but it is going to be safe in that it's going to look at the size of the queue using the copy of old_size and not leave the loop until we're sure there's space. Once we're sure there's space, then we're safe to say, "OK, I start a second critical section." In that critical section, I update the queue.

As long as there's one process running add and a different process running remove, then this can happily add a bunch of things to an empty queue until the queue fills up, and then it's just going to start buzzing around in this loop

until somebody takes something out of the queue.

So then its companion del_queue looks really similar:

c

Copy

```c
int del_queue(void) {
    int old_size;
    do {
        critical_begin();
        old_size = num_queue;
        critical_end();
    } while (old_size <= 0);

    critical_begin();
    return queue[--num_queue];
    critical_end();
}
```

It also has a loop. It also copies out the old size of the queue into a temporary local variable, so num_queue and all of the other queue variables are shared. They're going to be static. Let's say they're shared between both of these functions.

If the queue is empty, del_queue is going to come in and it's just going to start buzzing away - cycling, cycling, cycling until there's something in the queue. Once there's something in the queue, it can come down to this critical section, enter the critical section to manipulate the queue, take the data out, and now it returns the data to whoever called it.

As long as we're lucky and the queue stays roughly half full - let's say we add some things to the queue and then after that we're running both add and remove at roughly the same rate - then neither one of them is going to block. Neither one of them is going to start spinning, because as long as it's always kind of half full, add still has some space to throw a few more in the queue, and del has something to work with. It can pull some out.

So as long as we're lucky, we can get away with this plan without burning CPU cycles because the queue is neither saturated at full nor empty. But who here thinks that sounds like a fantastic management plan? Nobody. Exactly.

So we want a better strategy than that. What we've got so far is safe - we're not going to over-exhaust the queue, we're not going to overfill the queue, and we're not going to screw up our understanding of

what is in the queue because we've got a critical section protecting our two players from each other. But we want to do a better job of what happens if the queue is either completely full or completely empty.

I'll leave it there for today. But we will come back and talk about the better management on Tuesday. Have a good weekend everybody.