Let's go ahead and get started.

OK, so today we have one topic to finish up very briefly from last time.

So if you remember, when we finished off last time, we were talking about the example of a multithreaded Web server.

So the example that we were talking about, and this is an example that I'm going to use throughout the lecture today consisted of a Web server with, this example consists of a Web server with three main modules or main components.

So it consists, the three modules are a networking module, a Web server module -- -- which is in charge of generating, for example, HTML pages, and then a disk module which is in charge of reading data off a disk.

OK, so this thing is going to be communicating with the disk, which I've drawn as a cylinder here.

So what happens is that client requests come in to this Web server.

They come in to the network module.

The network module forwards those requests on to the Web server.

The Web server is in charge of generating, say, the HTML page that corresponds to the request.

And in order to do that, it may need to read some data off of the disk.

So it forwards this request onto the disk module, which goes and actually gets the page from the disk and at some point later, the disk returns the page to the Web server.

The Web server returns the page to the network module, and then the network module sends the answer back over the network to the user.

So this is a very simple example of a Web server.

It should be sort of familiar to you since you have just spent a while studying the Flash Web server.

So you can see that this is sort of a simplified description of what a Web server does.

So if you think about how you actually go about designing a Web server like this, of course it's not the case that there is only one request that is moving between these modules at any one point in time.

So, in fact, there may be multiple client requests that come in to the network module.

And the network module may want to have multiple outstanding pages that it's asking the Web server to generate.

And the Web server itself might be requesting multiple items from the disk.

And so, in turn, that means that at any point in time there could be sort of multiple results.

There could be results streaming back in from the disk which are going into the Web server, which is sort of chewing on results and producing pages to the network module.

And so it's possible for there to be sort of queues that are building up between these modules both on the send and receive.

So, I'm going to draw a queue, I'll draw a queue just sort as a box with these vertical arrows through it.

So there is some buffering that's happening between the sort of incoming requests and the outgoing requests on these modules.

OK, and this buffering is a good thing.

And we're going to talk more about this throughout the lecture today because what it allows us to do is it allows us to decouple the operations of these different modules.

So, for example, the disk module can be reading a page from disk while the HTML page, while the HTML server is, for example simultaneously generating an HTML page that wants to return to the client.

But in this architecture, you can see that, for example, when the Web server wants to produce a result, it can only produce a result when the disk pages that it needs are actually available.

So the Web server is dependent on some result from the disk module being available.

So if we were to look at just this Web server, I'm going to call this the HTML thread here and the disk thread, so these two threads that are on the right side of this diagram that I've drawn here, If you were to look at the code that was running in these things, and we saw this last time, the code might look something like this.

So the HTML thread is just going to sit in a loop continually trying to de-queue information from this queue that is shared between it and the disk thread.

And then, the disk thread is going to be in a loop where it continually reads blocks off the disk, and then enqueues them onto this queue.

So this design at first seems like it might be fine.

But then if you start thinking about what's really going on here, there could be a problem.

So, suppose for example that the queue is of a finite length.

It only has a certain number of elements in it.

Now when we keep calling enqueue over and over and over again, it's possible that if the HTML thread isn't consuming these pages off the queue fast enough, that the queue could fill up, and it could overflow, right?

So that might be a problem that we want to sort of make a condition that we would explicitly check for in the code.

And so we could do that by adding a set of conditions like this.

So what you see here is that I have just augmented the code with these two additional variables, used and free, where used indicates the number blocks that are in the queue that are currently in use.

And free indicates the number of blocks that are in the code, the number of blocks that are in the queue that are currently free.

So what this loop does is that the disk thread says it only wants to enqueue something onto the queue when there are some free blocks.

So, it has a while loop that just loops forever and ever and ever while they're waiting when there are no free blocks, OK?

And similarly, the HTML thread is just going to wait forever when there are no used blocks, OK?

And then, when the disk thread enqueues a block onto the queue, it's going to decrement the free count because it's reduced the number of things that are in the queue.

And it's going to increment the used count because now there is one additional thing that's available in the queue, OK?

So this is a simple way in which now we've made it so these things are waiting for each other.

They are coordinating with each other by use of these two shared variables used in free.

OK, so these two threads share these variables.

So that's fine.

But if you think about this from a scheduling point of view, there still is a little bit of a problem with this approach.

So in particular, what's going on here is that, oops, when one of these threads enters into one of these while loops, it's just going to sit there checking this condition over and over and over and over again, right?

So then the thread scheduler schedules that thread.

It's going to repeatedly check this condition.

And that's maybe not so desirable.

So suppose, for example, that the HTML thread enters into this loop and starts looping because there's no data available.

Now, what really we would like to have happen is for the disk thread to be allowed to get a chance to run, so maybe it can produce some data so that the HTML thread can then go ahead and operate.

But, with this while loop there, we can't quite do that.

We just sort of waste the CPU during the time we are in this while loop.

So, instead what we are going to do is introduce the set of what we call sequence coordination operators.

So in order to introduce this, we're going to add something, a new kind of data type that we call an eventcount.

An eventcount, you can just think of it as an integer that indicates the number of times that something has occurred.

It's just some sort of running counter variable.

And we're going to introduce two new routines.

So these two routines are called wait and notify.

OK, so wait takes two arguments.

It takes one of these eventcount variables.

And it takes a value.

OK, so what wait says is check the value of this eventcount thing, and see whether when we check it, the value of this eventcount is less than or equal to value.

If eventcount is less than or equal to value, then it waits.

And what it means for it to wait is that it tells the thread scheduler that it no longer wants to be scheduled until somebody later calls this notify routine on this same eventcount variable.

OK, so wait says wait, if this condition is true, and then notify says, wake up everybody who's waiting on this variable.

So we can use these routines in the following way in this code.

And it's really very straightforward.

We simply change our iteration through our while loops into wait statements.

So what we're going to do is we're going to have the HTML thread wait until the value of used becomes greater than zero.

And we're going to have our disk thread wait until the value of free becomes greater than zero.

And then the only other thing that we have to add to this is simply a call to notify.

So what notify does is it indicates to any other thread that is waiting on a particular variable that that thread can run.

So the HTML thread will notify free, which will tell the disk thread that it can now begin running if it had been waiting on the variable free.

OK, so this emulates the behavior of the while loop that we had before except that the thread scheduler, rather than sitting in an infinite while loop simply doesn't schedule the HTML thread or the disk thread while it's waiting in one of these wait statements.

OK, so what we're going to talk about for the rest of the lecture today is related to this, and I think you will see why as we get through the talk.

The topic for today is performance.

So performance, what we've looked at so far in this class are these various ways of structuring complex programs, how to break them up into several modules, the client/server paradigm, how threads work, how a thread

scheduler works, all of these sort of big topics about how you design a system.

But we haven't said anything about how you take a system design and in an ordered, regular, systematic way, think about making that system run efficiently.

So that's what we're going to try and get at today.

We're going to look at a set of techniques that we can use to make a computer system more efficient.

And so, these techniques, there are really three techniques that we're going to look at today.

The first one is a technique called concurrency.

And concurrency is really about allowing the system to perform multiple operations simultaneously.

So, for example, in our sample Web server, it may be the case that we have this disc that we can sort of read pages from at the same time that, for example, the CPU generates some Web pages that it's going to output to the client.

OK, so that's what concurrency is about.

We are also going to look at a technique called caching, which you guys should have all seen before.

Caching is really just about saving off some previous work, some previous computation that we've already done, or our previous disk page that we've already read in.

We want to save it off so that we can reuse it again at a later time.

And then finally, we are going to look at something called scheduling.

So scheduling is about when we have multiple requests to process, we might be able to order those requests in a certain way or group the requests together in a certain way so that we can make the system more efficient.

So it's really about sort of choosing the order in which we do things in order to make the system run more efficiently.

And throughout the course of this, I'm going to use this example of this Web server that we've been talking about to sort of motivate each of the applications, or each of these performance techniques that we're going to talk about.

So in order to get to the point where we can understand how these performance techniques work, we need to talk

a little bit about what we mean by performance.

How do we measure the performance of the system, and how do we understand where the bottlenecks in performance in a system might be?

So one thing we might want to, the first thing we need to do is to define a set of performance metrics.

These are just a set of terms and definitions that we can use so that we can talk about what the performance of the system is.

So the first metric we might be interested in is the capacity of the system.

And capacity is simply some measure of the amount of resource in a system.

So this sounds kind of abstract, but what we mean by a resource is some sort of thing that we can compete with.

It's a disk, or a CPU, or a network, so we might, for example, talk about the capacity of a disk might be the size in gigabytes or the capacity of a processor might be the number of instructions it can execute per second.

OK, so once we have capacity, now we can start talking about how much of the system we are actually using.

So we talk about the utilization.

So utilization is simply the percentage of capacity we're using.

So we might have used up 80% of the disk blocks on our computer.

So now there are two sort of properties, or two metrics that are very commonly used in computer systems in order to classify or sort of talk about what the performance of the system is.

So, the first metric is latency.

So, latency is simply the time for a request to complete.

The REQ is request, OK, and we can also talk about sort of the inverse of this, what at first will seem like the inverse of this, which is throughput.

That's simply the number of requests per second that we can process.

So when you think about latency and throughput, when you first see this definition, it's tempting to think that simply throughput is the inverse of latency, right?

If it takes 10 ms for a request to complete, well, then I must be able to complete 100 requests per second, right?

And, that's true in the simple case where in the very simple example where I have a single module, for example, that can process one request at a time, so a single computational resource, for example, that can only do one thing at a time, if this thing has some infinite set of inputs in it, it takes 10 ms to process each input, we'll see, say, 100 results per second coming out, OK?

So if something takes 10 ms to do, you can be 100 of them per second.

So we could say the throughput of this system is 100 per second, and the latency is 10 ms.

What we're going to see throughout this talk is that in fact a strict relationship between latency and throughput doesn't hold I mean, you guys have probably already seen the notion of pipelining before in 6.004, and you understand that pipelining is a way in which we can improve the throughput of the system without necessarily changing the latency.

And we'll talk about that more carefully as this talk goes on.

OK, so given these metrics, now what we need to do is think a little bit about, OK, so suppose I have some system, and suppose I have some sort of set of goals for that system like I want the system to be able to process a certain number of requests per second, or I want the latency of this system to be under some amount.

So then the question is, so you are given this computer system and you sit down and you want to measure it.

And so you're going to measure the system.

And what do you expect to find?

So, in the design of computer systems, it turns out that there is some sort of well-known performance pitfalls, or so-called performance bottlenecks.

And the goal of sort of doing performance analysis of a system is to look at the system and figure out where the bottlenecks are.

So, this typically in the design of the big computer system, what we're worried about is which of the little individual modules within the system is most responsible for slowing down my computer.

And what should I do in order to, sort of, and then once you've identified that module, picking about how to make a particular module that slow run faster.

So that's really what finding performance bottlenecks is about.

And there's a classic bottleneck that occurs in computer systems that you guys all need to know about.

It's this so-called I/O bottleneck.

OK, so what the I/O bottleneck says it's really fairly straightforward.

If you think about a computer system, it has a hierarchy of memory devices in it, OK?

And these memory devices start, or storage devices.

So these storage devices first start with the CPU.

So the CPU has some set of registers on it, a small number of them, say for example, 32. And you can access those registers very, very fast, say once per instruction, once per cycle on the computer.

So, for example, if your CPU is one gigahertz, you may be able to access one of these registers in 1 nanosecond.

OK, and so typically at the tallest level, of this pyramid, we have a small storage that is fast, OK?

As we go down this pyramid adding new layers, and looking at this storage hierarchy, we're going to see that things get bigger and slower.

So, just below the CPU, we may have some processor cache, OK, and this might be, for example, 512 kB.

And it might take 20 ns to access a single, say, block of this memory.

And then we're going to have the RAM, the main memory of the device, which on a modern machine might be 1 GB.

And it might take 100 ns to access.

And then below that, you take a big step down or big step up in size and big step down and performance.

You typically have a disk.

So a disk might be as big as 100 GB, right?

But, performance is very slow.

So it's a mechanical thing that has to spin, and it only spins so fast.

So a typical access time for a block of the disk might be as high as 10 ms or even higher.

And then sometimes people will talk in this hierarchy the network is actually a level below that.

So if something isn't available on the local disk, for example, on our Web server, we might actually have to go out into the network and fetch it.

And if this network is the Internet, right, the Internet has a huge amount of data.

I mean, who knows how much it is.

It's certainly orders of terabytes.

And it could take a long time to get a page of the Internet.

So it might take 100 ms to reach some remote site on the Internet.

All right, so the point about this I/O bottleneck is that this is going to be a very common, sort of the disparity in the performance of these different levels of the system is going to be a very common source of performance problems in our computers.

So in particular, if you look at the access time, here's 1 ns.

The access time down here is 100 ms.

This is a ten to the eighth difference, right, which is equal to 100 million times difference in the performance of the fastest to the slowest thing here.

So, if the CPU has to wait for something to come over the network, you're waiting for a very long time in terms of the amount of time the CPU takes to, say, read a single word of memory.

So when we look at the performance of a computer system, we're going to see that often this sort of I/O bottleneck is the problem with that system.

So if we look, for example, at our Web server, with its three stages, where this stage is the one that goes to disk, this is the HTML stage, which maybe can just be computed in memory.

And this is the network stage.

We might be talking about 10 ms latency for the disk stage.

We might be talking about just 1 ms for the HTML page, because all it has to do is do some computation in memory.

And we might be talking about 100 ms for the network stage to run because it has to send some data out to some remote site.

So if you, in order to process a single request, have to go through each of these steps in sequence, then the total performance of the system, the time to process a single request is going to be, say for example, 111 ms, the sum of these three things, OK?

And so if you look at the system and you say, OK, what's the performance bottleneck in this system?

So the performance bottleneck, right, is clearly this network stage because it takes the longest to run.

And so if we want to answer a question about where we should be optimizing the system, one place we might think to optimize is within this network stage.

And we'll see later an example of a simple kind of optimization that we can apply based on this notion of concurrency to improve the performance of the networking stage.

So as I just said, the notion of concurrency is going to be the way that we are really going to get at sort of eliminating these I/O bottlenecks.

So -- And the idea is going to be that we want to overlap the use of some other resource during the time that we are waiting for one of these slow I/O devices to complete.

And, we are going to look at two types of concurrency.

We're going to look at concurrency between modules -- -- and within a module, OK?

So we may have modules that are composed, for example, our networking module may be composed of multiple threads, each of which can be accessing the network.

So that's an example of concurrency within a module.

And, we're going to look at the case of between module concurrency where, for example, the HTML module can be processing and be generating an HTML page, while the disk module is reading a request for another client at the same time.

OK, and so the idea behind concurrency is really going to be that by using concurrency, we can hide the latency of

one of these slow I/O stages.

OK, so the first kind of concurrency we're going to talk about is concurrency between modules.

And the primary technique we use for doing this is pipelining.

So the idea with pipelining is as follows.

Suppose we have our Web server again.

And this time let's draw it as I drew it at first with queues between each of the modules, OK?

So, we have our Web server which has our three stages.

And suppose that what we are doing is we have some set of requests, sort of an infinite queue of requests that is sort of queued up at the disk thread, and the disk thread is producing these things.

And we're sending them through.

Well, we want to look at how many pages come out here per second, and what the latency of each page is.

So, if we have some list of requests, suppose these requests are numbered R1 through Rn, OK?

So what's going to happen is that the first request is going to start being processed by the disk server, right?

So, it's going to start processing R1. Now, in a pipelining system, what we're going to want to do is to have each one of these threads sort of working on a different request, each one of these modules working on a different request at each point in time.

And because the disk is an independent resource from the CPU, is an independent resource from the network, this is going to be OK.

These three modules aren't actually going to contend with each other too much.

So what's going to happen is this guy's going to start processing R1, right?

And then after 10 ms, he's going to pass R1 up to here, and start working on R2, OK?

And now, 1 ms after that, this guy is going to finish R1 and send it to here.

And then, 9 ms after that, R2 is going to come up here.

And this guy can start processing R3. OK, so does everybody sort of see where those numbers are coming from?

OK.

[LAUGHTER] Good.

So now, what we're going to do is if we look at time starting with this equal to time zero, in terms of the requests that come in and out of this last network thread, we can sort of get a sense of how fast this thing is processing.

So the first R1 enters into this system after 11 ms, right?

It takes 10 ms to get through here and 1 ms to get through here.

And, it starts processing R1 at this time.

So, I'm going to write plus R1 to suggest that we start processing it here.

The next time that this module can do anything is 100 ms after it first started processing, the next time this module does anything is 100 ms after it started processing R1. So, at time 111 ms, it can output R1, or it's done processing R1. And then, of course, by that time, R2 and R3, some set of requests have already queued up in this queue waiting for it.

So it can immediately begin processing R2 at this time, OK?

So then, clearly what's going to happen is after 211 ms, it's going to output R2, and it's going to begin processing R3, OK?

So, there should be a plus there and a plus there.

So, and similarly, at 311 we're going to move on to the next one.

So, if you look now at the system, we've done something pretty interesting, which is that it still took us, sort of the time for this request to travel through this whole thing was 110 ms.

But if you look at the enter - arrival time between each of these successive outputs of R1, they are only 100 ms, right?

So we are only waiting as long as it takes R1 to process a result in order to produce these results, in order to produce answers.

So by pipelining the system in this way and having the Web server thread and the disk thread do their processing

on later requests while R1 is processing its request, we can increase the throughput of the system.

So in this case, we get an arrival every 100 ms.

So the throughput is now equal to one result every 100 ms, or ten results per second, OK?

So, even though the latency is still 111 ms, the throughput is no longer one over the latency because we have separated them in this way by pipelining them.

OK, so that was good.

That was nice.

We improve the performance of the system a little bit.

But we didn't really improve it very much, right?

We increased the throughput of this thing a little bit.

But we haven't really addressed what we identified earlier as being a bottleneck, which the fact that this R1 stage is taking 100 ms to process.

And in general, when we have a pipeline system like this, we can say that the throughput of the system is bottlenecked by the slowest stage of the system.

So anytime you have a pipeline, the throughput of the system is going to be the throughput of the slowest stage.

So in this case, the throughput is 10 results per second.

And that's the throughput of the whole system.

So if we want to improve the throughput anymore than this, what we're going to have to do is to somehow improve the performance of this module here.

And the way that we're going to do that is also by exploiting concurrency.

This is going to be this within a module concurrency.

So if you think about how a Web server works, or how a network works, typically when we are sending these requests to a client, it's not that we are using up all of the available bandwidth of the network when we are sending these requests to a client, right?

You may be able to send 100 MB per second out over your network.

Or if you're connected to a machine here, you may be able to send 10 MB a second across the country to some other university.

The issue is that it takes a relatively long time for that request to propagate, especially when that request is propagating out over the Internet.

The latency can be quite high.

But you may not be using all the bandwidth when you are, say, for example, sending an HTML page.

So in particular it is the case that multiple applications, multiple threads, can be simultaneously sending data out over the network.

And if that doesn't make sense to you right now, we're going to spend the whole next four lectures talking about network performance.

And it should make sense for you.

So just take my word for it that one of the properties of the network is so that the latency of the network may be relatively high.

But in this case we are not actually going to be using all the bandwidth that's available to us.

So that suggests that there is an idle resource.

It means that we sort of have some network bandwidth that we could be using that we are not using.

So we'd like to take advantage of that in the design of our system.

So we can do this in a relatively simple way, which is simply to say, let's, within our networking module, rather than only having one thread sending out requests at one time, let's have multiple threads.

Let's, for example have, say we have 10 threads.

So we have thread one, thread two, thread ten, OK?

And we're going to allow these all to be using the network at once.

And they are all going to be talking to the same queue that's connected to the same HTML module that's

connected to the same disk module.

And there's a queue between these as well.

OK, so now when we think about the performance of this, now let's see what happens when we start running requests through this pipeline.

And let's see how frequently we get requests coming out of the other end.

We draw our timeline again.

You can see that R1 is going to come in here, and then after 10 ms it's going to move to here.

And then after 11 ms it'll arrive here.

We'll start processing request one.

Now the second request, R2, is going to be here.

And, we're going to have 9 ms of processing left to do on it.

After R1, it gets sent on to the next thread.

So, R2 is going to be in here for 9 ms.

It will be in here for 1 ms.

So, 10 ms after R1 arrives here, R2 is going to arrive here.

So, what we have is we have 11 ms.

We have R1. Now, 10 ms later, we have R2. OK, so now you can see that suddenly this module, this system is able to process multiple requests, so it has multiple requests that processing at the same time.

And so 10 ms after that, R3 is going to start being processed, and then, so what that means is that after some passage of time, we're going to have R10 in here.

And, that's going to go in after 101 ms, right?

So, we're going to get R10. OK, and now we are ready to start processing.

Now we've sort of pushed all these through.

And now, suppose we start processing R11. OK, so R11 is going to flow through this pipeline.

And then, it's at time 111, R11 is going to be ready to be processed.

But notice that at time 111, we are finished processing R1, right?

So, at this time, we can add R11 to the system, and we can output R1. OK, so now every 10 ms after this, another result is going to arrive, and we're going to be able to output the next one.

OK, and this is just going to continue.

So now, you see what we've managed to do is we've made this system so that every 10 ms after this sort of startup time of 111 ms, after every 10 ms, we are producing a result, right?

So we are going to get, actually, 100 per second.

This is going to be the throughput of this system now.

OK, so that was kind of neat.

How did we do that?

What have we done here?

Well, effectively what we've done is we've made it so that this module here can process sort of 10 times as many requests as it could before.

So this module itself now has 10 times the throughput that it had before.

And we said before that the bottleneck in the system is, the throughput of the system is the throughput of the slowest stage.

So what we've managed to do is decrease the throughput of the slowest stage.

And so now the system is running 10 times as fast.

Notice now that the disk thread and the network threads both take 10 ms, sort of the throughput of each of them is 100 per second.

And so, now we have sort of two stages that have been equalized in their throughput.

And so if we wanted to further increase the performance of the system, we would have to increase the

performance of both of these stages, not just one of them.

OK, so that was a nice result, right?

It seems like we've done something sort of, we've shown that we can use this notion of concurrency to increase the performance of a system.

But, we've introduced a little bit of a problem.

In particular, the problem we've introduced is as follows.

So, remember, we said we had this set of threads, one through, say for example, ten, that our processing, they're all sharing this queue data structure that is connected up to our HTML thread.

So, the problem with this is that what we've done is to introduce what's called a race condition on this queue.

And I'll show you what I mean by that.

So if we look at our code snippet up here, for example for what's happening in our HTML thread, we see that what it does is it calls dequeue, right?

So the problem that we can have is that we may have multiple of these modules that are simultaneously executing at the same time.

And they may simultaneously both call dequeue, right?

So depending on how dequeue is implemented, we can get some weird results.

So, let me give you a sort of very simple possible implementation of dequeue.

Suppose that what dequeue does is it reads, so, given this queue here, let's say the queue is managed by, there's two variables that keep track of the current state of this queue.

There is a variable called first, which points to the head of the queue, and there's a variable called last, which -- first points to the first used element in this queue, and last points to the last used element.

So, the elements that are in use in the queue at any one time are between first and last, OK?

And, what's going to happen is when we dequeue, we're going to sort of move first over one, right?

So, when we dequeue something, we'll free up this cell.

And when we enqueue, we'll move last down one.

And then, when last reaches the end, we are going to wrap it around.

So this is a fairly standard implementation of a queue.

It's called the circular buffer.

And if first is equal to last, then we know that the queue is full.

So that's the condition that we can check.

So we are not going to go into too many details about how this thing is actually implemented.

But let's look at a very simple example of how dequeue might work.

So remember we have these two shared variables first and last that are shared between these, say, all these threads that are accessing this thing.

And what dequeue might do is to say it's going to read a block, read a page from this queue, so read the next HTML page to output, and it's going to read that into a local variable called page.

Let's call this queue buf, B-U-F, I mean we'll use a array notation for accessing it.

So it's going to read buf sub first, OK, and then it's going to increment first.

First gets first plus one, and then it's going to return page.

OK, that seems like a straightforward implementation of dequeue.

And so we have one thread that's doing this.

Now, suppose we have another thread that's doing exactly the same thing at the same time.

So it runs exactly the same code.

And remember that these two threads are sharing the variables buf and first.

OK, so if you think about this if you think about these two things running two threads at the same time, there is sort of an interesting problem that can arise.

So one thing that might happen when we are running these two things at the same time is that the thread

scheduler might first start running thread one.

And it might run the first instruction of thread one.

And then it might run the second instruction.

And then it might run this return thing.

And then it might come over here, and it might start running T2. So, it might, then, stop running T1 and start running T2, and execute its three instructions.

So if the thread scheduler does this, there's nothing wrong.

It's not a problem, right?

The thread scheduler, each of these things read its value from the queue and incremented it. T1 read one thing from the queue, and then T2 read the next thing from the queue.

So clearly some of the time this is going to work fine.

So let's make a list of possible outcomes.

Sometimes we'll be OK.

The first possible outcome was OK.

But let's look at a different situation.

Suppose what happens is that the first thing the thread scheduler does is schedule T1. And T1 executes this first instruction, and then just after that the thread scheduler decides to pre-empt T1, and allow T2 to start running.

So it in particular allows T2 to execute this dequeue instruction to its end, and then it comes over here and it runs T1. OK, so what's the problem now?

Yeah?

Right, OK, so they've both read in the same page variable.

So now both of these threads have dequeued the same page.

So the value first, for T1, it was pointing here.

And then we switched.

And it was still pointing here, right?

And now, so both of these guys have read the same page.

And now they are both at some point going to increment first.

So you're going to increment it once.

Then you're going to increment it again.

So this second element here in the queue has been skipped.

OK, so this is a problem.

We don't want this to happen.

Because the system is not outputting all the pages that it was supposed to output.

So what can we do to fix this?

So the way that we fixed this is by introducing something we call isolation primitives.

And the basic idea is that we want to introduce an operation that will make it so that any time that the page variable gets read out of the queue, that we also at the same time increment first without any other sort of threads' accesses to this queue being interleaved with our accesses to this queue, or our dequeues from the queue.

So in sort of technical terms, what we say is we want these two things, the reading of page and the incrementing of first to be so-called atomic.

OK, and the way that we're going to make these things atomic is by isolating them from each other, that by isolating these two threads from each other when they are executing the enqueue and dequeue things.

So, these two terms we're going to come back to in a few months in the class towards the end of the class.

But all you need to understand here is that there is this race condition, and we want some way to prevent it.

And the way that we're going to prevent it is by using these isolation routines also sometimes called locks.

So in this case, the isolation schemes are going to be called locks.

So the idea is that a lock is simply a variable, which can be in one of two states.

It can either be set or unset.

And we have two operations that we can apply on a lock.

We can acquire it, and we can release it.

OK, and acquire and release have the following behavior.

What acquire says is check the state of the lock, and if the lock is unset, then change the state to set.

But if the lock is set, then wait until the lock becomes unset.

What a release says is it simply says change the state of the lock from unset to set, or from set to unset, excuse me.

So let's see how we can use these two routines in our code.

So let's go back to our example of enqueue and dequeue.

Let's introduce a lock variable.

We'll call it TL for thread lock.

And, what we're going to do is simply around these two operations to access the queue, to modify this page and first, to read the page and modify first, we're going to put in an acquire and a release.

OK so we have ACQ on this thread lock, and we have release on this thread lock.

OK, so let's look, so this seems fine.

It looks like we've done this.

But it's sort of positing the existence of this acquire procedure that just sort of does the right thing.

If you think about this for a minute, it seems like we can have the same race condition problem in the acquire module as well, right, or the acquire function as well.

With two guys both try and acquire the lock at the same time?

How are we going to avoid this problem?

And there's a couple of ways that are sort of well understood for avoiding this problem in practice, and they're talked about in the book.

I'm just going to introduce the simplest of them now, which is that we're going to add a special instruction to the microprocessor that allows us to do this, acquire efficiently.

It turns out that most modern microprocessors have an equivalent instruction.

So we're going to call this instruction RSL for read-set- lock.

OK, so the idea with RSL is as follows.

We can basically, the implementation of the acquire module is going to be like this.

What it's going to do, remember we want to wait until we want to loop.

We don't have the lock.

If we don't have the lock, we want to loop until we've had the lock.

So the implementation require may look as follows.

We'll have a local variable called held.

We'll initially set it to false in a while loop while we don't hold the lock.

We're going to use this RSL instruction.

So, what this says is held equals RSL of TL, OK?

So, what the RSL instruction does is it looks at the state of the lock, and if the lock is unset, then it sets it.

And if the lock is set, then it sets it and it returns true.

And if the lock is set, then it returns false.

So it has the property that it can both read and set the lock within a single instruction, right?

And we're going to use this read and set lock sort of primitive, this basic thing, as a way to build up this sort of more complicated acquire function, which we can then use to build up these locks.

OK, so anytime you're designing a multithreaded system in this way, or a system with lots of concurrency, you

should be worrying about whether you have race conditions.

And if you have race conditions, you need to think about how to use locks in order to prevent those race conditions.

Alright, so there are a couple of other topics related to performance that appear in the text.

And one of those topics is caching.

And I just want to spend one very brief minute on caching.

So you guys have already seen catching presumably in the context of 6.004 with processor caches.

And what we would want to do, so you might want to sit down and think through as an example of how you would use a cache to improve the performance of our Web server.

So one thing that you might do in order to improve the performnence of the Web server is to put a cace in the disk thread that you use instead of going to disk in order to sort of reduce the latency of a disk access.

And I'll at the beginning of class next time take you through a very simple example of how we can actually use the disk thread in order to do that.

But you guys should think about this a little bit on your own.

So barring that little digression that we'll have next time, this takes us to the end of our discussion of sort of modularity, abstraction, and performance.

And what we're going to start talking about next time is networking, and how networks But I want you guys to make sure you keep in mind all these topics that we've talked about because these are going to be the sort of fundamental tools that we are going to use throughout the class in the design of computer systems.

So because we've finished this module, it doesn't mean that it's sort of OK to stop thinking about this stuff.

You need to keep all of this in mind at the same time.

So we'll see you all on Wednesday.