So today we're going to continue our discussion of enforcing modularity.

And in particular if you remember last time we spent a lot of time talking about -- Last time we spent a lot of time talking about how we can create a virtual memory so that we can have multiple programs running on top of one piece of hardware that all appear to have a separate address space or separate memory.

So, two lectures ago, we talked about having one module per computer.

And then last time we talked about this notion of wanting to have a module per virtual computer.

And we saw this notion of virtual memory.

OK, and now this time what we're going to do is we're going to talk about the other piece that our virtual computer needs, which is a virtual processor.

OK, so the idea that we want here when we are designing a virtual processor is to create, have our programs be able to run on a single physical processor, but have different programs be able to run as though they were the only program that was executing on that processor.

So we want the programs to be written in a style where the programmer doesn't have to be aware of the other programs that are running on the machine, but where the machine creates sort of a virtual processor for each one of these programs that's running so that each program has its own sort of set of instructions that it executes.

And those instructions appear to be independent of all the other programs that are running.

So in order to create this illusion of a virtual processor -- -- we are going to introduce a very simple concept.

We're going to have each program run in what's known as a thread.

And thread is really short for a thread of execution.

And a thread is just a collection of all of the state that we need in order to keep track of what one single given program is doing at a particular point in time.

So a thread is a collection of the instructions for the program that the module is running, as well as the current state of the program.

In particular, it's the value of the registers on the processor including of particular import for this lecture the program counter and the stack pointer.

OK, so there's a bunch of registers on the processor, say, 32 registers, and there also are these special registers, the program counter, that keeps track of which instruction is currently being executed, and the stack pointer which keeps track of where on the stack values are currently being pushed or popped from.

So these things combined together with the stack are really the things that define one single thread.

So, and the way to think about this is if you think about a program that's running, you can really encapsulate almost everything about what that program is currently doing by the instructions that it has to execute, the value of the registers that are currently set in the program, and the state that's on the stack.

Now, programs may also have some sort of data that's stored in the global memory in the global variables our things that are stored on the heap that have been allocated via some memory allocation call.

And so those things are a part of the program as well, although I don't want to explicitly list them as being a part of a single thread because if two programs, as we'll see, two programs can share the same address space.

And if two programs are sharing the same address space, then they're going to share those global variables, the stuff that's stored on the heap of the program.

So think of a thread as these things, plus there is some additional, we'll call it a heap, set of memory that may be shared between several different threads that are all running within the same address space.

And I'll explain more about that as we go.

OK, so as I said, now what we want is to create this illusion that each one of these threads has its own virtual processor upon which it's running.

So how are we going to accomplish that?

So if you think about this for a second, the way to accomplish this is simply to allow each one of these, each thread to update these various registers, push things onto the stack, and then to have some function which we can call, which will save all of this current state off for the current thread that's executing, and then load the state in for another thread that might be executing, and stop the current thread from executing, and start the new one executing.

So the idea is if you were to look at a timeline of what the processor was doing, and if the processor is running two threads, thread one and thread two, you would see that during some period of time, thread one would be running.

Then the computer would switch over and thread two would start running, and then again the computer would

switch, and thread one would start running.

So we are going to multiplex the processor resource in this way.

So we have one processor that can be executing instructions.

And at each point in time, that processor is going to be executing instructions from one of these two threads.

And when we switch in these points in between threads, we are going to have to save off this state for the thread that was running, and restore the state for the thread that is currently running.

OK, so that's a very high-level picture of what we're going to be doing with threads.

And there's going to be a number of different ways, and that's what we're going to look at through this lecture is how we can actually accomplish this switching, how we decide when to switch, and what actually happens when this switch takes place.

OK, so we're going to look at several different methods.

So the first thing we're going to look at is an approach called cooperative scheduling, or cooperative switching.

So in cooperative switching, what happens is that each thread periodically decides that it's done processing and is willing to let some other process run.

OK, so the thread calls a routine that says I'm done; please schedule another thread.

So, this cooperative approach is simple, and it's easy to think about.

In the grand scheme of enforcing modularity of creating this illusion that each program really has its own computer that's running, and where each program gets to run no matter what any other program does, cooperative scheduling has a bit of a problem because if we are doing cooperative scheduling, then one process can just never call this function that says I give up time to the other processes running on the system.

One module may just never give up the processor.

And then we are in trouble because no other module ever gets to run.

So instead what we're going to talk about, the alternative to this, is something we call preemptive scheduling.

And in preemptive scheduling, what happens is some part of the computer, say, the kernel, periodically decides that it forces the currently running thread to give up the processor, and starts a new thread running.

And we'll see how that works, OK?

I also want to draw a distinction between whether these threads are running in the same address space or in a different address space, or in a different address space.

OK, so we could, for example, have these two threads, T1 and T2. They might be running as a part of a single, say, application.

So, for example, they might be a part of a, say for example, some computer game that we have running.

So suppose we have a computer game like, but let me just load up my computer here.

Suppose we have a computer game like pick your favorite computer game, say, Halo.

And we are running Halo, and Halo is going to consist of some set of threads that are responsible for running this game.

So we might have one main thread which, what it does when it runs is to simply update; it repeats in a loop over and over again, wait for a little while, check and see if there's any user input, update the state of the game, and then, say for example, redraw the display.

And because Halo is a videogame that somebody is staring at and looking at, it needs to update that display at some rate, say, once every 20 times a second in order to create the illusion of sort of natural animation in the game.

So this wait step is going to wait a 20th of a second between every step.

OK, so this might be one example of a thread.

But of course within Halo, there may be multiple other threads that are also running.

So there may be a thread that's responsible for looking for input over the network, right, to see if there is additional data that's arrived from the user, and from other users that are remote and updating the state of the game as other information comes in.

And there may be a third thread that, say for example, is responsible for doing some cleanup work in the game like reclaiming memory after some monster that's running around the game has died and we no longer need its state, OK?

So if we look at just this, say, main thread within Halo, as I said here we can encapsulate the state of this thread

by a set of registers, say, R1 through Rn, as well as a stack pointer and a program counter.

And then, there also will be a stack that represents the currently executing program.

So, for example, the stack might, if we just have entered into this procedure, it might just have a return address which is sort of the address that we should jump to when we're done executing this Halo program here.

And then finally, we have a program counter that points to this sort of current place where we are executing it.

So we might have started off where we are just executing the first instructions here, the init instruction.

OK, so what we're going to look at now is this case where we have a bunch of threads, say for example, in Halo all running within the same address space.

So they're all part of the Halo program.

They can all access the global variables of Halo.

But we want to have different threads to do different operations that this program may need to take care of.

So we want each of these threads to sort of have the illusion that it has its own processor.

And then later we'll talk about a set of programs, say, that are running in different address And we'll talk about what's different about managing two threads, each of which is running in a different address space.

OK, so the very simple idea that we're going to introduce in order to look at this situation where we are going to start off by looking at cooperative scheduling in the same address space.

We are going to introduce the notion of a routine called yield.

OK, so yield is going to be the thing the currently executing thread calls when it's ready to give up the processor and let another thread run.

So the thread is going to run for a while and then it's going to call yield, and that's going to allow other threads to sort of do their execution.

And this is cooperative because if a program doesn't call yield, then no other program will ever be allowed to run on the system.

So the basic idea is that when a program calls yield, what it's going to do is exactly what I described just a minute ago.

So yield is going to save the state of the current thread, and then it's going to -- -- schedule the next thread, so pick the next thread to run from all of the available threads that are on the system.

And then it's going to dispatch the next thread, so it's going to in particular setup the state for the next thread so that it's ready to run.

And it's going to jump into the return address for that thread.

OK, so I want to take you guys through a very simple example of a program that has multiple threads in it.

And we're going to have a simple thread scheduler.

So, the thread scheduler is the thing that decides which thread to run next.

And this thread scheduler is going to keep an array of fixed size, call it num_threads, that are all the threads that are currently running on the system.

It's going to have an integer that is going to tell us what the next thread to run is.

And then for every one of our threads, we're going to have a variable that we call me which is the ID of the currently executing thread.

So, this is local to the thread.

OK so let's look at an example of how this might work with this Halo thread that we're talking about here.

OK, so suppose we have a Halo thread.

And we'll call this the main thread of execution.

And as I said, we want to have a couple of other threads that are also running on the system at the same time.

So we might have a network thread and a cleanup thread.

Now in this case, what we are going to say is that the yield procedure is going to do these three steps.

And I made these steps a little bit more concrete so that we can actually think about what might be happening on the stack as these things are executing.

So, the yield procedure is going to, when it runs, the first thing it's going to do is save the state.

So to save the state, it simply stores in this table the stack pointer of the currently executing program, the stack pointer of the currently executing thread.

And then what it's going to do is it's going to pick the next thread to run.

So in this case, picking the next thread to run, the thread scheduler is doing something extremely simple.

It's just cycling through all the available threads one after another and scheduling them, right?

So it says next gets next plus one, and then modulo the total number of threads that are in the system.

So suppose num_threads is equal to five.

As soon as next plus one is equal to five, and then five modulo five is zero, and we're going to wrap that back around.

So, once we've executed the fourth thread, we are going to wrap around and start executing thread number zero.

And then finally what it's going to do is it's going to restore the stack pointer.

Here, I'll just change this for you right now.

This should be table sub next.

I didn't want to do that.

So, we have this table.

So, we restore the stack pointer for the next thread that we want to run from the table of stack pointers that are available.

And then what's going to happen is that when we exit out of this yield routine, we are going to load the return address from the stack pointer that we just have set up.

So you'll see how that works again in a second.

But the basic process, then, is just going to be as follows.

So this only works on a single processor.

And I'm not going to have time to go into too much detail about what happens in a multiprocessor, but that book talks carefully about why this doesn't work on a single processor machine.

OK, so the basic process, then, it's going to be as follows.

So we've got our yield procedure.

We have our three threads.

Say we've numbered them one, two, and three, as shown up here.

So, num_threads is equal to three, and say we start off with next equal to one.

So what's going to happen is that the first thread at some point it's going to call a yield routine.

And what that's going to do is it's going to cause this yield to execute.

We're going to increment next.

And we're going to schedule the network thread here.

OK, and the network thread is going to run for a little while.

And then at some point, it's going to call yield, and that's going to sort of cause the next thread to be scheduled.

We're going to call cleanup, and then finally we're going to call yield.

And the third thread is going to call yield and cause the first thread to run again.

And this process is just going to repeat over and over and over again.

OK, so -- OK, so what I want to do now is to look at actually more carefully at what's going on with, how this scheduling process is actually working.

So we can understand a little bit better what's happening to the stack pointer, and how these various yields are actually executing.

So let's look at, so this is going to be an example of the stack on, say, two of these threads, OK?

So, suppose we have thread number one.

I'll call this thread one, and this is our Halo thread.

OK, and this is the stack for Halo.

We also have thread number two, which is our network thread.

OK, and this is also some stack associated with this thing.

So we're just looking at the current state of the stack on these two things, these two threads.

And if you remember, typically we represent stacks as going down in memory.

So, these are the different addresses of the different entries in the stack that represent what's going on currently in the processor.

So we'll say that thread one, perhaps, the stack starts at address 108, and maybe thread two starts at address 208. OK, so suppose we start up this system, and when we start it up, the stack pointer is currently pointing here.

So we take a snapshot of this system at a given point in time.

Suppose the stack pointer is currently pointing at 104. The currently executing thread is this one, thread one.

And we have, remember our thread table that captures the current thread number, and the saved stack pointer for each one of these threads.

So we've got thread number one, thread number two.

OK, so let's suppose we start looking at this point in time where the stack pointer is pointing here.

And if we look at this program, so if we start off with the program that we initially had, in order to understand what happens to the stack, we need to actually look at when the stack gets manipulated.

So if we are just looking at C code, typically the C code isn't going to have, we're not going to see the changes to the stack occurring within the C code.

So what I've done here is in yellow, I've annotated this with the sort of additional operations, the assembly operations, that are happening on the entry and exit to these procedures.

So what happens just before we call the yield procedure is that the Halo thread will push the return address onto the stack.

OK, so if we start executing the yield procedure, Halo is going to push the return address onto the stack.

And then it's going to jump into the yield procedure.

So, suppose we come onto here.

Now if we look at what the return address is after we execute the yield procedure, it's going to be instruction number five.

So we're going to push that address onto the stack because instruction number five is labeled here on the left side is where we're going to return to after the yield.

And then the yield is going to execute.

And what's going to happen as the yield executes is it's going to change the stack pointer to be the stack pointer of the next thread, right?

So, I have this typo here again.

So be careful about that.

We're going to change the stack pointer to be the stack pointer of the next thread.

And so when it executes this pop RA instruction that we see here, it's actually going to be popping the return address from the network thread that it scheduled next as opposed to the return address from this Halo thread.

OK, so suppose that the saved stack pointer for the network thread was pointing here at 204. So, that would have been an entry, 204, here.

And the saved return address that it's going to jump to is going to be here.

So suppose the return address that it jumps to is 1,000, OK?

So if we look at the state of these two threads as they have been running, we saw thread one ran for a little while, and then it called yield, and the instruction following yield was instruction number five.

OK, so that was the address that we pushed on to the stack pointer.

Now, we switched over here to thread two.

And last time we ran thread two, it called yield, and its return address was 1,000. So, there was this 1,000 here, OK?

So now, when we schedule thread two, what's going to happen is that we're going to read the, so this is instruction 1,000. We're going to start executing from 1,000 because we're going to execute this pop return So we're going to

pop the return address off the stack, and we're going to start executing here at instruction 1,000, OK?

So, you guys kind of see how it is that we switch now from one thread to the other, by switching the stack pointer and grabbing our return address from the stack of thread two instead of from thread one.

OK, so now what's going to happen is this thread two is going to start executing from this address.

And it's going to run for a little while.

And at some point later, it's going to call yield again, right, because it's run for a while and it's decided that it's time to yield.

So, suppose that instruction 1,024 it calls yield.

So when it does that, the same thing is going to happen.

It's going to have run for a little while.

So its stack is going to have grown.

But eventually it's going to push the return address onto the stack, say here 1,025, and suppose this value of its stack pointer now has grown down.

So it's gone 204. It's run for a while, pushed some things on the stack, and maybe the stack pointer is now at 148. So, when it calls yield, it's going to write into the stack pointer address in the thread table 148. And we're going to restore the stack pointer addressed from here, which I forgot to show being written in, but what I should have written in there was 104. So, we're going to restore the stack address to 104. We're going to pop the next instruction to execute off of that stack five, and then we're going to keep executing.

So you just switch back and forth in this way.

OK, so -- -- what I want to do now is, so this is the basic process now whereby this yield instruction can be used to switch between the scheduling of these two procedures, right?

And this is sort of the core of how it is.

So we have these two threads of execution, and they just sort of run through.

And they periodically call yield.

And that allows another thread to be written, to execute it.

But otherwise these threads have been written in a style where they don't actually have to know anything about the other threads that are running.

There could have been two other threads or 200 other threads running on the system at the same time.

And this approach that I showed you would have caused all those threads eventually to have been scheduled and to execute properly.

Right, but as we said before, requiring these functions to actually call yield periodically has sort of defeated the purpose of our enforcing modularity, one of our goals of enforcing modularity, which is to make it so that no one thread can interfere with the operation of the other thread, or cause that other thread to crash, right, because if the procedure never calls a yield, then a module never calls yield, excuse me, another thread will never be scheduled.

And so, that module will have the ability, essentially, to take over the whole computer, which is bad.

So what we're going to look at now is how we go from this cooperative scheduling where modules call yield to preemptive scheduling where modules are forced to yield the processor periodically.

OK, so this is the case where you have no explicit yield statements.

All right, so the idea here is that turns out to be very simple.

So programs aren't going to have an explicit yield statement in them.

But what we're going to do is we're going to have a special timer that runs periodically within, say, for example, the kernel.

So suppose the kernel has a timer that runs periodically that causes the kernel to be sort of woken up and allowed to execute some code.

So this timer is going to be connected to what's called an interrupt.

So, we're going to introduce a timer interrupt, and almost all processors essentially have some notion of a timer interrupt.

And an interrupt is something that when it fires, it causes the processor to run a special piece of code, OK?

So, basically this is going to be some processor.

Think of it, if you like, as some line on the processor.

OK, there's a wire that's coming off the processor.

And when this wire gets pulled high, when the timer goes off and fires, this line is going to be pulled high.

And when that happens, the microprocessor is going to notice that and is going to invoke a special function within the kernel.

So this line is going to be checked by the microprocessor before it executes each instruction, OK?

And if the line is high, the microprocessor is going to execute one of these special gate functions.

So, we saw the notion of a gate function before that can be used for a module to obtain entry into the kernel.

Essentially what's going to happen is that when the timer interrupt fires, it's going to go execute one of these special gate functions as well.

And that's going to cause the kernel to then be in control of the processor.

So remember when the gate function runs, it switches the user to kernel mode bit, to kernel mode.

It switches the page map address register to the kernel's page map so that the kernel is in control, and it switches the stack pointer to the kernel's saved stack pointer so that the kernel is in control of the system, and can execute whatever code that it wants.

So this is going to accomplish basically everything that we need, right?

Because if we can get the kernel in control of the system, now, the kernel can do whatever it needs to do to, for example, schedule the next thread.

So the way that's going to work is basically very simple.

So when the kernel gets run, it knows which thread, for example, is currently running.

And basically what it does is it just calls the appropriate yield function, it calls the yield function for the thread that it's currently running, forcing that thread to yield.

So, kernel calls yield on current thread, right?

OK, so in order to do this, of course the kernel needs to know how to capture the state for the currently running

thread.

But for the most part that's pretty simple because the state is all encapsulated in the current values of the registers in the current value of the stack pointer.

So, the kernel is going to call yield on the currently executing thread, and that's going to force that thread to go ahead and schedule another thread.

So the module itself hasn't called yield, but still the module has been forced to sort of give up its control the processor.

So, when it calls yield, it's just going to do it we did before, which is save our state and schedule and run the next thread.

OK, so the only last little detail that we have to think about is what if the kernel wants to schedule a thread that's running in a different address space?

So if a kernel wants to schedule a thread that's running a different address space, it well, it has to do what we saw last time when we saw about how we switch address spaces.

It has to change the value of the PMAR register so that the next address space gets swapped And then it can go ahead and jump into the appropriate location in the sort of newly scheduled address space.

So that brings us to the next topic of discussion.

So, what we've seen so far now, we saw cooperative in the same address space, and I introduced the notion of a preemptive scheduling.

What we want to look at now is sort of what it means to run multiple threads in different address spaces.

Typically when we talk about a program that's running on a computer or in 6.033 we like to call programs running on computers processes.

When we talk about a process, we mean an address space plus some collection of threads.

So this is sort of the real definition of what we mean by process.

And alternately, you will see processes called things like applications or programs.

But any time you see a word like that, what people typically mean is some address space, some virtual address space in which we resolve memory addresses.

And then a collection of threads that are currently executing within that address space, and where each of those threads includes the set of instructions and registers and stack that correspond to it, OK?

So the kernel has explicit support for these processes.

So in particular, what the kernel provides are routines to create a process and destroy a process.

OK, and these create and destroy methods are going to sort of do, the create method is going to do all the initialization that we need to do in order to create a new address space and to create at least one thread that is running within that address space.

So we've sort of seeing all the pieces of this, but basically what it's going to do is it's going to allocate the address space in its table of all the address spaces that it knows about.

So we saw that in the last time, and it's going to allocate a piece of physical memory that corresponds to that address space.

So, it's going to allocate a piece of physical memory that corresponds to that address space.

It's going to allocate one of these page maps that the PMAR register is going to point to when this thing is running.

OK, and now the other thing that it's going to do is to go ahead and load the code for this thing into memory and map it into the address space.

So it's going to add the code for this currently running module into the address space.

And then it's going to create a thread for this new process.

And it's going to add it to the table, the thread table, and then it's going to set up the value of the stack pointer and the program counter for this process so that when the process starts running it, you sort of into the process at some well-defined entry point, say the main routine in that process so that the thread can start executing at whatever starting point it has.

OK, and now destroy is just going to basically do the opposite.

It's going to get rid of all this state that we created.

So it's going to remove the address space, and it's going to remove the thread from the table.

OK, and it may also reclaim any memory that's associated exclusively with this process.

OK so if we look at -- So if you look at a computer system at any one point in time, what you'll see is a collection of processes.

So I've drawn these here as these big boxes.

So you might have a process that corresponds to Halo and, say, we are also editing our code.

So, we have a process that corresponds to emacs.

OK, and each one of these processes is going to have an address space associated with it.

And then there are going to be some set of threads that are running in association with this process.

So in the case of Halo, I'm going to draw these threads as these little squiggly lines.

So in the case of Halo, we saw that maybe it has three threads that are running within it.

So these are three threads that all run within the same address space.

Now, emacs might have just one thread that's running in it, say, although that's probably not true.

emacs is horribly complicated, and probably has many threads that are running within it.

And so, we're going to have these sort of two processes with these two collections of threads.

So if you think about the sort of modularity or the enforcement of modularity that we have between these processes, we could say some interesting things.

So first of all, we've enforced modularity.

We have hard enforced modularity between these two processes, right, because there's no way that the Halo process can muck with any of the memory that, say, the emacs process has executed.

And as long as we're using preemptive scheduling, there's no way that the Halo process could completely maintain control of the processor.

So the emacs process is going to be allowed to run, and its memory is going to be protected from the Halo process.

OK, now within the Halo process, there is sort of a different story.

So within the Halo process, there is sort of a different story.

So within the Halo process, these threads, because they are all within the same address space, can muck with each other's memory.

So they are not isolated from each other in that way.

And typically, because these are all within one process in that if we were to destroy that process in this step, what the operating system would do is in addition to destroying the address space, all of the running threads.

So we say that these threads that are running within Halo share fate.

If one of them dies, or if one of these threads fails or crashes or does something bad, then they are essentially all going to crash or do something bad.

If the operating system kills the process, then all of these threads are going to go away.

So this is the basic picture of sort of what's going on within a computer system.

If you stare at this for a little bit, you see that there is actually sort of a hierarchy of threads that are running on any one computer system at any one point in time.

So we have these larger sorts of processes that are running.

And then within these processes, we have some set of threads that's also running, right?

And so, there is a diagram that sort of useful to help us understand this hierarchy or layering of processes or threads on a computer system.

I want to show that to you.

OK, so on our computer system, if we look at the lowest layer, we have our microprocessor down here.

OK, and this microprocessor, what we've seen is that the microprocessor has two things that it can be doing.

Either the microprocessor can be executing, say, some user program, or the microprocessor can be interrupted and go execute one of these interrupt handler functions.

So these interrupts are going to do things like, is going to be this timer interrupt, or when some IO device, say for example the disk, finishes performing some IO operation like reading a block from memory, then the processor will receive an interrupt, and the processor can do whatever it needs to do to service that piece of hardware so

that the hardware can go and, for example, read the next block.

So in a sense, you can think of the processor itself as having two threads that are associated with it.

One of them is an interrupt thread, and one of them is the main thread now running on top of this.

So, these are threads that are really running within the kernel.

But on top of these things there is this set of applications like Halo and emacs.

And, these are the kind of user programs that are all running on the system.

And there may be a whole bunch of these.

It's not just two, but it's any number of threads that we can multiplex on top of this main thread.

And then each one of these may, in turn, have sub-threads that are running as a part of it.

So if you think about what's going on in this system, you can see that at any one level, these two threads don't really need to know anything about the other threads that are running at that same level.

So, for example, within Halo, these individual sub-threads don't really know anything about what the other sub-threads are doing.

But it is the case that the threads at a lower level need to know about the threads that are running above them because these threads at the lower level implement this thread scheduling policy, right?

So, the Halo program decides which of its sub-threads to run next.

So the Halo program in particular is, so the parent thread implements a scheduling -- -- policy for all of its children threads.

So Halo has some policy for deciding what thread to run next.

The operating system has some.

The kernel has some policy for deciding which of these user level threads to run next.

And the parent thread also provides some switching mechanism.

So we studied two switching mechanisms today.

We looked at this notion of having this sort of cooperative switching where threads call yield in order to allow the next thread to run.

And, we looked at this notion of preemptive scheduling that forces the next thread in the schedule to run.

So, if you look at computer systems, in fact it's a fairly common organization to see that there is preemptive scheduling at the kernel level that causes different user level applications to run.

But within a particular user program, that program may use cooperative scheduling.

So the individual threads of Halo may hand off control to the next thread only when they want to.

And this sort of makes sense because if I'm a developer of a program, I may very well trust that the other threads that I have running in the program at the same time, I know I want them to run.

So, we can talk about sort of different switching mechanisms at different levels of this hierarchy.

OK, so what we've seen up to this point is this sort of basic mechanism that we have for setting up a set of threads that are running on a computer system.

And what we haven't really talked at all about is how we can actually share information between these threads or coordinate access to some shared resource between these threads.

So what I want to do with the rest of the time today is to talk about this notion of coordinating access between threads.

And we're going to talk about this in the context of a slightly different example.

So -- -- let's suppose that I am building a Web server.

And I decide that I want to structure my Web server as follows: I want to have some network thread, and I want to have some thread that services disk requests.

OK, so the network thread is going to do things like accept incoming connections from the clients and process those connections and parse the HTTP requests and generate the HTML results that we send back to users.

And the disk request thread is going to be in charge of doing these expensive operations where it goes out to disk and reads in some data that it may need to assemble these HTML pages that we generate.

So, for next recitation, you're going to read about a system called Flash, which is a multithreaded Web server.

And so this should be a little taste of what you're going to see for tomorrow.

So, these two threads are likely to want to communicate with each other through some data structure, some queue of requests, or a queue of outstanding information.

So suppose that what we're looking at in fact in particular is a queue of disk blocks that are being sent from a disk request thread out to the network thread.

So, whenever the disk finishes reading a block, it enqueues a value into the network thread so that the network thread can then later pull that value off and deliver it out to the user.

So in this case, these are two threads that are both running within the same address space within the same Web server.

So they both have direct access to this queue data structure.

They can both read and write to it at the same time.

This queue data structure, think of it as a global variable.

It's in the memory that's mapped in the address space so both threads can access it.

So let's look at what happens when these two threads, let's look at some pseudocode that shows what might be happening inside of these two threads.

So, suppose within this first thread here, this network thread, we have a loop that says while true, do the following. De-queue a requestn -- call it M -- from the thread, and then process, de-queue a disk block that's in the queue and then go ahead and process that disk queue.

Go ahead and process that.

And now, within the disk request thread, we also have a while loop that just loops forever.

And what this does is it gets the next disk block to send, however it does that, goes off and reads the block from disk, and then enqueues that block onto the queue.

So if you like, you can think of this as simply the disk request thread is going to stick some blocks into here, and then the network thread is going to sort of pull those blocks off the top of the queue.

So if you think about this process running, there's kind of a problem with the way that I've written this, right, which is that suppose that the disk request thread runs much faster than the network thread.

Suppose that for every one network, one block that the network thread is able to pull off and process, the disk thread can enqueue two blocks.

OK, so if you think about this for awhile, if you run this system for a long time, eventually you're going to have enqueued a huge amount of stuff.

And typically the way queues are implemented is they are sort of some fixed size.

You don't want them to grow to fill the whole memory of the processor.

So you limit them to some particular fixed size.

And eventually we are going to have the problem that the queue is going to fill up.

It's going to overflow, and we're going to have a disk block that we don't have anything that we could do with.

We don't know what to do with it.

Right, so OK, you say that's easy.

There is an easy way to fix this.

Why don't we just wait until there is some extra space here.

So we'll introduce a while full statement here that just sort of sits in a spin loop and waits until this full condition is not true.

OK, so as soon as the full condition is not true, we can go ahead and enqueue the next thing on the queue.

And similarly we're going to need to do something over here on the process side because we can't really de-queue a message if the queue is empty.

So if the processing thread is running faster than the enqueueing thread, we're going to be in trouble.

So we're going to also need to introduce a while loop here that says something like while empty.

OK, so this is fine.

It seems like it fixes our problem.

But there is a little bit of a limitation to this approach, which is that now what you see is suppose these two threads

are running.

And they are being sort of scheduled in round robin; they are being scheduled one after the other.

Now this thread runs.

And when it runs, suppose that the queue is empty.

Suppose the producer hasn't put anything on the queue yet.

Now when this guy runs, he's going to sit here, and for the whole time that it's scheduled, it's just going to check this while loop to see if the thing is empty over, and over, and over, and over, and over again, right?

so that's all whole lot of wasted time that the processor could and should have been doing something else useful like perhaps letting the producer run so that it could enqueue some data.

So in order to do this, so in order to fix this problem, we introduce this notion of sequence coordination operators.

And what sequence coordination operators do is they allow a thread to declare that it wants to wait until some condition is true.

And they allow other threads to signal that that condition has now become true, and sort of allow threads that are waiting for that condition to become true to run.

So, we have these two operations: wait on some variable until some condition is true, and signal on that variable.

OK, so we're basically out of time now.

So what I want to do is I'll come back and I'll finish going through this example for the next time.

But what you should do is think about, suppose you had these sequence coordination operators.

How could we sort of modify these two while loops that we have for these two operators in order to be able to take advantage of the fact that in order to be able to make this so we don't sit in this spin loop forever and execute.

So that's it.

Today we saw how to get these multiple processes to run on one computer.

And next time we'll talk about sort making computer programs run more efficiently, getting good performance out of this sort of architecture we've sketched.