So in case you are all worried that you may not be in the right class, you are still in 6.033. If you're wondering what happened to the Indian guy who normally lectures, he's sitting right over there.

We are trading off lectures throughout the class.

I'm going to do about the next eight lectures up until spring break, and then Hari will be back.

So, my name is Sam, and feel free to address any questions about the lecture to me.

So today we are going to keep talking about this concept of enforcing modularity that we started talking about last time.

So last time we saw how we could use this notion of the client service model in order to separate two modules from each other.

So the idea was by running the client and the service on separate machines, we can isolate these two things from each other.

So we can make it so, for example, when the client wants to invoke some operation on the server, that the server can verify the request that the client makes, and make sure that the client isn't asking to do something malicious.

Similarly, so this is what we saw last time was this client service model.

And so the other benefit of the client service model was it meant that we could decouple the client from the service.

So it meant that when the client issued a request against the service, it didn't necessarily mean that the client, if the service failed when the client issued the request, it wasn't necessarily the case that the client would also fail.

So the server might crash executing the request, and then the client would get a timeout and would be able to retry.

And similarly, if the client failed, the server could continue handling requests on the behalf of other clients.

OK, so that was a nice separation, the ability to split the client and the server apart from each other.

That's a good thing.

But the problem with this approach is that we had to have two separate machines.

The way we presented this was that the client was running on one computer, and the server was running on

another computer.

And if you think about this, this isn't exactly what we want, right?

Because that means, suppose we want to build up a big, complicated computer system that's composed of multiple different modules.

If we have to run each one of those modules on a separate computer, that's not really very ideal, right?

To build up a big service, we're going to need a whole lot of computers.

And clearly, that's not the way that computer systems that we actually interact with work.

So what we've seen so far is this notion: we have a module per computer.

OK, and what we're going to do today is see how we can generalize this so that instead of having one module per computer, we can instead have multiple modules running within a single computer.

But when we do that, we still want to maintain these nice sort of protection benefits that we had between the client and service when these two things were running on separate computers.

OK, so the way we are going to do that is by creating something we call a virtual computer.

So -- What we're going to do is we're going to create this notion of multiple virtual computers.

They're all running on one single computer.

And then we're going to run each one of these modules within a virtual computer.

OK, and I'll talk more about what I mean by a virtual computer throughout the next couple of lectures.

But the idea is that a virtual computer has exactly the same, to a program that's running on it, a virtual computers looks just like one of these client or server computers might have looked.

Now, in particular, a virtual computer has the same sorts of abstractions that we studied in the previous lectures available to it that a real computer has.

So a virtual computer has a virtual memory -- -- and a virtual processor.

So, today what we're going to talk about is this notion of a virtual memory.

And so you guys have already probably seen the term virtual memory in 6.004. So, the word should be familiar to

you.

And we're going to use exactly the same abstraction that we used in 6.004. So we're just going to show how this abstraction can be used to provide this protection between the client and server that we want.

OK, we're also going to introduce today this notion of something called a kernel.

And a kernel is something that's in charge, basically, of managing all of these different virtual computers that are running on our one physical computer.

So the kernel is going to be the sort of system that is going to be the piece of the system that actually knows that there are multiple, virtual computers running on this system.

OK, so I want to start off by first just talking about why we want a virtualized memory, why virtualizing memory is a good thing.

So I'm going to abbreviate virtual memory as VM.

So why would we want to virtualize memory?

Well, let's see what might happen if we build a computer system with multiple modules running on it at the same time that didn't have a virtualized memory.

So suppose we have some microprocessor and we have some memory.

And within this memory, we have the code and data for a couple of different modules is stored.

So we have, say, module A and module B, and this is the things like the code and the data that these modules are using to currently execute.

So, in this environment, suppose the module, A, executes some instruction.

So if you think of this as memory, let's say that module A begins at address A here, and module B begins at address B here.

So, suppose that module A executes some instruction like store some value, R1, in memory address, B.

OK, so module A writes into memory address B here.

So, this can be a problematic thing, right, because if the microprocessor just allows this to happen, if it just allows A to write into module B's memory, then module B has no way of being isolated from module A at all, right?

Module A can, for example, write some sort of garbage into the memory of module B.

And then when module B tries to read from that memory address, it will either, say, try and execute an illegal instruction, or it will read in some data that doesn't make any sense.

So we've lost the separation, the isolation that we had between if module A and module B are a client and a service trying to interact with each other.

We've lost this sort of ability.

We've lost the ability to separate and isolate each other, isolate them from each other that we had when they were running on separate computers.

Furthermore, this sort of arrangement of virtual memory.

So one problem with this is A can overwrite B's memory.

But we have another problem, which is that this sort of arrangement also is very, very difficult to debug.

So, these kinds of bugs can be very hard to track down.

And really, when we're building computer systems, we'd like to get rid of them.

So, for example, suppose that A stores into B's memory, overwrites an instruction that somewhere sort of arbitrarily within B's memory.

And then, 10,000 instructions later, B tries to load in that memory address and execute that instruction.

And it gets an illegal instruction.

Now, imagine that you have to debug this program.

So you sort of are sitting there, and so you run application B, and it works just fine.

And then you run application A, and A overwrites part of B's memory.

And now, when program B runs sometimes long after program A, perhaps, is already terminated, B mysteriously crashes on you.

So how are you going to track down the problem?

It's very tricky, and it's the kind of problem that's extremely hard to debug these kinds of issues.

So we really would like to make it so that A isn't able to simply overwrite arbitrary things into B's memory.

We'd like to protect A from B.

OK, and that's what this virtual memory abstraction that we're going to talk about today is going to do for us.

So, if you think about how you might protect A from B, the solution that you sort of come up with after thinking about this and staring at this for a little bit is that you want to verify that all of the memory accesses that A does are actually valid memory accesses that A should be allowed to make, so for example, that refer to objects that are actually that A has allocated or that A owns.

OK, so in order to do that, what we're going to need to do is to interpose some sort of software in between the memory accesses that each module makes, and the actual memory access.

So the idea is as follows.

So this is the virtual memory abstraction.

OK, so the idea is as follows.

For each one of our modules, say A and B, we're going to create what's known as an address space, OK?

And this address space is going to be, for example, on a 32 bit computer, it's two to the 32 minus one, it's going to be a 32 bit memory space.

So, this is the virtual address space of, say, a process, A.

So, process A can use any memory address that's in this two to the 32 bit range, OK?

But this is going to be a virtual address space.

And what I mean by that is when process A refers to some address within its memory space, call it virtual address VA.

That address is going to be mapped by the virtual memory system into a physical address, OK?

So this virtual address here is going to be mapped into some physical address -- -- somewhere within the actual physical memory of the system, OK?

So this physical memory of the system is also a 32 bit space.

But the sort of mapping from this virtual address in A's memory into this physical address space is going to be handled by this virtual memory system.

So now the idea is that each thing that, say, a process A or B might want to reference is going to be mapped into some different location within this memory.

So, for example, the code from process A or from module A can be mapped into one location, and the code from module B is going to be mapped into a different location in the physical address space.

OK, so we have this notion of address spaces.

Each module that is running on the system in a virtual memory system is going to be allocated to one of these address spaces.

This address space is going to be, and this address space is virtual in the sense that these addresses are only valid in the context of this given module, OK?

And, they translate into, each address in the virtual space translates into some address in the physical space.

Now, if you look at this for a minute, you might think, well, this is kind of confusing because now B and A both have a 32 bit address space.

And the computer itself only has a 32 bit address space.

So, there are sort of more addresses between all the modules than the computer itself physically has.

And, if you remember back from 6.004, the story that we told you about that was that, well, some of these virtual addresses can actually be mapped into the disk on the computer.

So in 6.004, virtual memory was presented as a way to create a computer that appeared to have more physical memory than it, in fact, did.

And the way that that was done was to map some of these addresses onto disk.

So you might have addresses from B mapped onto disk.

And then, when B tries to reference an address that's mapped onto disk, the system would load that data from memory, load that data from disk and into memory.

So, we're not going to talk about that aspect of virtual memory very much today.

The thing to remember is just that these virtual address spaces, there may be parts of this virtual address space in each of the modules that isn't actually mapped into any physical address space.

So if one of these modules tries to use one of these unmapped virtual address spaces, it's not allowed to do that.

This virtual memory system will signal an error when it tries to do that.

And on your computer, sometimes you'll see programs that will mysteriously crashed with things that say things like, tried to access an illegal memory address.

When it does that, that's because the program tried to access some memory location that wasn't mapped by the virtual memory system.

OK, so let's dive a little more into actually how this virtual memory abstraction works so we can try to understand a little bit more about what's going on.

So this is going to be a simplified VM hardware, OK?

It's a little bit simplified even from what you learned about in 6.004. So, the idea in this simplified hardware is that we have our processor.

OK, and then we're going to have this VM system, which is sometimes called a memory management unit, MMU.

And, this is a piece of hardware that's going to help us to do this mapping from these logical addresses in the modules' address spaces into the physical memory.

And then, we're going to have the physical memory, OK?

Now the idea is that when an instruction tries to access some virtual address, so for example suppose we execute instruction load some virtual address, load into R1 some virtual address, OK?

What's going to happen when we do that is that the microprocessor is going to send this virtual address to the VM system.

And then the VM system is going to translate that into some physical address that can be resolved within the memory itself.

OK, and the way that the virtual memory system is going to decide this mapping between a virtual address and a physical address is by using something that we call a page map, OK?

OK, so this table is an example.

So this is a page map.

And what a page map basically has is its just a table of virtual address to physical address mappings.

So this is the virtual address to physical address.

So the idea is that when some virtual address comes in here, the virtual memory manager looks up that virtual address in this page map, finds the corresponding physical address, and then looks that physical address up in the actual memory.

OK, so now there's one more detail that we need, right?

So what this gives us is we have this notion of a page map that does this mapping for us.

But we're missing a detail that is, OK, what we wanted was for each one of these different modules that's running in the system to have a different address space associated with it.

So what we want is we want to have separate page maps for each of these different modules, so, for A and B, OK, we're going to have a different page map.

And we're going to have this sort of same, we might have multiple copies of a particular virtual address in each one of these page maps.

And then what we're going to do is we're going to allocate a special register on the hardware, on the processor.

We're going to add a little register that's going to allow us to keep track of which one of these page maps we are currently looking at, OK?

So this thing is called the PMAR or the Page Map Address Register.

OK, and the page map address register simply points at one of these page maps.

OK, so what happens is that the virtual memory system, when it wants to resolve a virtual address, looks at this page map address register and uses that to find a pointer to the beginning of the page map that's currently in use.

And then he uses the page map that's currently in use to resolve, to get what physical address corresponds to this logical address.

OK so this is really the core concept from virtual memory.

So what we have now is we have this page map address register that can be used to select which one of these address spaces we are currently using.

OK, and so when we have selected, for example, the page map for module A, then module A can only refer to virtual addresses that are in its page map.

And those virtual addresses can only map into certain physical addresses.

So, for example, suppose this block here is the set of virtual addresses, the set of physical addresses that correspond to the virtual addresses in A's page map.

OK, these are the only physical addresses that A can talk about.

So if we, for example, have a different block of memory addresses that correspond to the virtual addresses that B can reference, we can see that there's no way for module A to be able to reference any of the memory that B uses.

So we are able to totally separate the physical memory, pieces of physical memory that A and B can talk about by using this page map mechanism that virtual memory gives us.

So, basically what we've done is we've sort of added this extra layer of indirection, this virtual memory system, that gets to map virtual addresses into physical addresses.

So the rest of this lecture is really going to be details about how we make this work, about how we actually decide, how we assign this PMAR register based on which one of the modules is currently executing about things like what the format of this page map table is going to look like, OK?

So this is really the key concept.

So what I want to do now is sort of turn to this second question I asked which is, how does this page map thing actually work?

How is it actually represented?

So one very simple representation of a page map might be that it simply is a pointer to, the page map just says where A's memory begins on the processor, right?

So it's just one value.

It says A's memory begins at this location in the physical memory, and all virtual addresses should be resolved relative to this beginning location of A's memory.

The problem with that representation is that it's not very flexible.

So, for example, suppose there's a third module, C, which is laid out in memory right next So, its storage is placed right next to A in memory.

And now, suppose that A wants to allocate some additional memory that it can use.

Now, in order to do that, if the page map is simply a single pointer to the beginning of A's address space, we're kind of in trouble.

We can't just add memory onto the bottom because then we would overlap C.

So we're going to have to move all of A's memory around in order to be able to make space for this.

OK, so this seems like a little bit problematic simply have it be a pointer.

The other thing we could do is suppose we could, instead, have a different option where we could say, for example, for every virtual address in A's address space, so for each 32 bit value that A wants to resolve, there might be an entry in this page map table, right?

So for every 32-bit virtual address, there would be a corresponding 32-bit physical address.

And there would just be a one-to-one mapping between all these things.

So, if A could reference a million blocks of memory, there would be a million entries in this page map table.

So, if you think about this for a minute, that sounds like kind of a bad idea, too, because now these tables are totally huge, right, and in fact they are almost as big as the memory itself, right, because if I have a million entries, if A can reference a million blocks, then I'm going to need a million entries in the table.

So the table becomes just as big as the memory itself.

So we need some in between sort of alternative hybrid between these two extremes.

And the idea, again, is very simple.

And you saw it in 6.004. So the idea is to take this 32-bit virtual address.

So suppose this is our 32-bit virtual address.

Now what we're going to do is we're going to split it up into two pieces, a page number and an offset.

OK, and we're going to choose some size for these two things.

For now I'll just arbitrarily pick a 20 bit page number and a 12 bit offset.

OK, so what this is going to do, so now what we're going to do is instead of storing a single word of memory at each entry in this table, we're going to store a page of memory at each entry in this table.

So -- So this table is going to look like a mapping between a page, and a physical address.

OK, so what a page is, so if the page number is 20 bits long, then that means that each page is going to be two to the 12th bits big, which is equal to, say, 4,096 words, OK?

So the idea is that we're going to have two to the 20th pages within each address space, and each page is going to map to one of these 4,096 byte blocks, OK?

So, if we have our memory here this page, say, page one is going to map into some physical address.

And page two is going to map into some other physical block, OK, so each one of these things is now 4,096 bytes, each block here, OK?

And so this, let's just expand this.

So this is now a page.

And this 12 bit offset is going to be used in order to look up the word that we want to actually look up in this page.

So, if the virtual address is, say, for example, page one offset zero, what that's going to do is we're going to look up in the page map.

We're going to find the page number that corresponds to, we're going to find page number one.

We're going to find the physical address that corresponds to it, we're going to go down here, and look at this block of memory.

And then within this 4,096 block memory, we're going to take the address zero, the first word within that thing, OK?

So now the size of these page tables is much smaller, right?

They're no longer two to the 30th bits.

Now they're two to the 20th bits, which is some small number of megabytes big.

But we've avoided this problem that we have before.

We have some flexibility in terms of how each page maps into the physical memory.

So I can allocate a third page to a process.

And I can map that into any 4,096 byte block that's in memory that's currently unused.

OK, so I have flexibility.

I don't have this problem, say, where A and C were colliding with each other.

OK, so this is sort of the outline of how virtual memory works.

But what I haven't yet described to you is how it is that we can actually go about creating these different address spaces that are allocated to the different modules, and how we can switch between different modules using this PMAR register.

So I've sort of described this as, suppose these data structures exist, now here's how we can use them.

But I haven't told you how these data structures all get together, and created, and set up to begin with.

And I hinted at this in the beginning.

But in order to do this, what we're going to need is some sort of a special supervisory module that's able to look at the page maps for all of these different, that's able to create new page maps and look at the page maps for all of the different modules that are within the system.

And the supervisory module is going to be able to do things like add new memory to a page map or be able to destroy, delete a particular module and its associated page map.

So we need some sort of a thing that can manage all this infrastructure.

So, this supervisory module -- -- is called the kernel.

OK, and the kernel is really, it's going to be the thing that's going to be in charge of managing all these data

structures for us.

So -- So here's our microprocessor with its PMAR register on it.

And, what we're going to do is we're going to extend the microprocessor with one additional piece of hardware, and this is the user kernel bit, OK?

So, this is just a bit specifies whether we are currently running a user module that is just a program that's running on your computer, or whether the kernel is currently executing, OK?

And the idea with this kernel bit is that when this kernel bit is set, the code that is running is going to have special privileges.

It's going to be able to manipulate special things about the hardware and the processor.

And in particular, we're going to have a rule that says that only the kernel can change the PMAR, OK?

So the PMAR is the thing that specifies which process is currently running, and selects which address space we want to be currently using.

And what we're going to use is we're going to use, so what we're going to do is have the kernel be the thing that's in charge of manipulating the value of this PMAR register to select which thing is currently being executed.

And we want to make it so that only the kernel can do this because if we, for example, allowed one of these other programs to be able to manipulate the PMAR, right, then that other program might be able to do something unpleasant to the computer, right?

It changes the PMAR to point at some other program's memory.

And now, suddenly all the memory addresses in the system are going to be resolved.

Then suddenly, we are going to be sort of resolving memory addresses relative to some other module's page map.

And that's likely to be a problematic thing.

It's likely to cause that other module to crash, for example, because the processor is set up to be executing instructions from the current program.

So we want to make sure that only something this kernel can change the PMAR.

And this kernel is going to be this sort of supervisory module that all of the other modules are going to have to trust to kind of do the right thing and manage the computer's execution for you.

And this kernel, except for this one difference that a kernel can change the PMAR, the kernel is, in all other respects, essentially just going to be another module that's running in the computer system.

So in particular, the kernel is also going to have one of these 32-bit virtual address spaces associated with it, OK?

But, what we're going to do is we're going to say that the kernel within its address space has all of the page maps of all the other programs that are currently running on the system mapped into its address space.

OK, so this is a little bit tricky because I presented this as though A and B referenced totally different pieces of memory.

So, I sort of have told you so far only about modules that are referencing disjoint sets of memory.

But in fact these page maps, right, they just reside in memory somewhere.

And it's just fine if I, for example, have multiple modules that are able to reference, that have the same physical addresses mapped into their virtual address space.

So it's very likely that I might want to have something down here at the bottom that both A and B can access.

So, this might be stuff that's shared between A and B.

I can map that into both A and B's memory.

In the same way, what I'm going to do is I'm going to map these page maps, which are also stored in memory into all of the page maps into the kernel's address space.

So the kernel is going to be able to reference the address spaces of all the modules.

And the kernel is also going to keep a little table of all the page maps for all of the currently running programs on the system.

So these are, for example, page maps for A and B.

And this is a list of all the maps that are currently running in the system.

So what's going to happen, now what can happen is because the kernel is allowed to change the PMAR, and because it knows about the location of all the other address spaces that are in the system, when it wants to start

running one of these programs that's running in the system, it can change the value of the PMAR to be sort of the PMAR for A or the PMAR to B.

And, it can manipulate all the values of the registers in the system so that you can start executing code for one of these.

You can switch between one of these two modules.

So the actual process of switching between which module is currently running.

We're going to focus on that more next time.

So, don't worry too much if you don't understand the details of how you actually switch from executing one program to another program.

But, you can see that the kernel can switch which address space is currently active by simply manipulating the value of this PMAR register.

Furthermore, the kernel can do things like it can create a new map.

So the kernel can simply allocate one of these new tables, and it can set up a mapping from a set of virtual addresses to a set of physical addresses so that you can create a new address space that a new module can start executing within.

Similarly, the kernel can do things like allocate new memory into one of these addresses.

So it can map some additional virtual addresses into real physical memory so that when one of these modules, say for example, requests additional memory to execute, the kernel can add that memory, add an entry into the table so that that new memory that the program has requested actually maps into a valid, physical address.

OK, so the question, of course, then is, so you can see the value of having this kernel module.

But the question is, how do we communicate between these modules that are running these user level modules that are running, and the kernel module that the user modules need to invoke, for example, request new memory.

So the way that we are going to do this is just like we've done everything else in this lecture so far: by adding a little bit of extra, by changing the processor just a little bit.

So in particular, what we're going to do -- What we're going to do is to is to add this notion of a supervisor call.

Call that SVC.

So a supervisor call is simply a special instruction that is on the processor that invokes the kernel.

So when the supervisor call instruction gets executed, it's going to set up the state of these PMAR and user kernel bit instructions so that the kernel can begin executing.

The supervisor call is also going.

But when the supervisor call instruction is executed, we need to be a little bit careful because we also need to decide somehow which code within the kernel we want to begin executing when the supervisor call instruction gets executed, right?

So what the supervisor call instruction does is it accepts a parameter which is the name of a so-called gate function.

So a gate function is a well defined entry point into another module that can be used to invoke a piece of code in that other module.

So, for example, the kernel is going to have a particular gate function which corresponds to allocating additional memory for a module.

So when a user program executes an instruction, supervisor call, to gate say for example, this might be some memory allocation code, this special instruction is going to do the following things.

First it's going to set the user kernel bit to kernel.

Then it's going to set the value of the PMAR register to the kernel page map.

It's going to save the program counter for the currently executing instruction, and then it's going to set the program counter to be the address of the gate function, OK?

So we've introduced a special new processor instruction that takes these steps.

So when this instruction is executed, we essentially switch into executing within the kernel's address space.

OK, and this kernel, and we begin executing within the kernel address space at the address of this gate function within the kernel's address space.

OK, so what this has done is this is a well defined entry point.

If the program tries to execute this instruction with the name of a gate function that doesn't exist, then that

program is going to get an error when it tries to do that.

So the program can only name gate functions that actually correspond to real things that the operating system can do, that the kernel can do.

And so, the kernel is then going to be invoked and take that action that was requested of it.

On return, essentially, so when the kernel finishes executing this process, when it finishes executing this, say, memory allocation instruction, we are just going to reverse this step.

So we are just going to set the PMAR to be, we're going to set the PMAR back to the user's program.

We are going to set the user kernel bit back into user mode.

And then we're going to jump back to the saved return address that we saved here, the saved program counter address from the user's program.

So when the program finishes, when the kernel finishes executing this service instruction, the control will just return back to the program in the place where the program needs to begin executing.

OK, so now using this gate, so using this notion of, so what we've seen so far now is the ability for the, we can use the virtual memory abstraction in order to protect the memory references of two modules from each other.

And we see when we have this notion of the kernel that can be used to, for example, that can be used to manage these address spaces to allocate new memory to these address spaces, and in general, to sort of manage these address spaces.

So this kernel is also going to allow us to do exactly what we set out trying to do, which is to act as the so-called trusted intermediary between, say for example, a client and a server running on the same machine.

OK, so I trust that intermediary is just a piece of code.

So suppose we have a client and a server, right, and these are two pieces of code written by two different developers.

Maybe the two developers don't necessarily trust that the other developer has written a piece of code that's 100% foolproof because it doesn't have any bugs.

But both of those developers may be willing to say that they will trust that the kernel is properly written and doesn't have any bugs in it.

And so they are willing to allow the kernel to sit in between those two programs and make sure that neither of them has any bad interactions with each other.

So let's see how we can use this kernel, the kernel combined with virtual memory in order to be able to do this.

So suppose this is our kernel.

So the idea is that this kernel running has, so suppose we have two processes A and B that want to communicate with each other in some way.

And, we already said that we don't want these two processes to be able to directly reference into each other's memory because that makes them dependent on each other.

It means that if there's a bug in A, that A can overwrite some memory in B's address space and cause B to crash.

So, that's a bad thing.

So instead, what we are going to do is we're going to create, well, one thing we can do is to create a special set of these supervisor calls that these two modules can use to interact with each other.

So in particular, we might within the kernel maintain a queue of messages that these two programs are exchanging with each other, a list of messages that they're exchanging.

And then, suppose A is, we're going to call this guy, A is a producer.

He's creating messages.

And B is a consumer.

A can call some function like Put which will -- supervisor call like Put -- which will cause a data item to be put on this queue.

And then sometime later, B can call this supervisor call Get, and pull this value out of the queue, OK?

So in this way, the producer and the consumer can interact with each other.

They can exchange data with each other and because we have this gate interface, this well-defined gate interface with these Put and Get calls being invoked by the kernel, the kernel can be very careful, just like in the case of the client and server running on two different machines.

In this case, the kernel can carefully verify that these put and get commands that these two different modules are calling actually are correctly formed, that the parameters are valid, that they're referring to valid locations in memory.

And therefore, the kernel can sort of ensure that these two modules don't do malicious things to each other or cause each other to break.

So this is an example here of something that's like an inter-process communication.

So you saw, for example, you've seen an instance of inter-process communication when he studied the UNIX paper.

We talked about pipes, and a pipe abstraction, and how that works.

Well, pipes are sort of something that's implemented by the kernel in UNIX as a way for two programs to exchange data with each other.

And, there's lots of these kinds of services that our people tend to push into the kernel that the kernel provides to the other applications, the modules that are running, so that these modules can, for example, interact with hardware or interact with each other.

So commonly within a kernel, you'd find things like a file system, an interface to the network, and you might, for example, find things like an interface to the graphics hardware.

OK, there is some sort of -- so if you look at what's actually within a kernel, there is a huge amount of code that's going into these kernels.

So I think we talked earlier about how the Linux operating system is many millions of lines of code.

If you go look at the Linux kernel, the Linux kernel is probably today on the order of about 5 million lines of code, most of which, say two thirds of which, is related to these so-called device drivers that manage this low-level hardware.

So this kernel has gotten quite large.

And one of the side effects of the kernel getting large is that maybe it's harder to trust it, right?

May be you sort of have less confidence that all the code in the kernel is actually correct.

And you can imagine that if you don't trust the kernel then the computer is not going to be as stable as you like to

be.

And this is one argument for why Windows crashes all the time is because it has all these drivers in it and these drivers aren't necessarily all perfectly written.

There are tens of millions of lines of code in Windows, and some of them crash some of the time, and that causes the whole computer to come down.

So there's a tension in the operating systems community about whether you should execute these things, you should keep these things look the file system or graphics system within the kernel or whether you should move them outside as separate services, which can be invoked in the same way, for example, that A and B interact with each other, by having some data structure that's stored within the kernel that buffers the requests between, say, the service and this, say, for example the graphics service and the users' programs that want to interact with it.

OK, so that's basically it for today.

What I've shown you is, well, actually we have a few more minutes.

Sorry.

[LAUGHTER] I know you're all getting up to leave, but so, OK, I just want to quickly touch on one last topic which is that, so, what I've shown you so far is how we can use the notion of virtual memory in order to protect the data, protect two programs from each other so that they can't necessarily interact with each other's data.

But there is some situations in which we might actually want to have two programs able to share some data with each other.

So I don't know if you guys remember, but when Hari was lecturing earlier, he talked about how there are libraries that get linked into programs.

And one of the common ways that libraries are structured these days is a so-called shared library.

So a shared library is something that is only stored in one location in physical memory.

But multiple different modules that are executing on that system can call functions within that shared library.

So in order to make this work, right, we need to have mapped the memory for the shared library that has all the functions that these other modules want to call into the address spaces for both of these modules so that they can actually execute the code that's there.

So the virtual memory system makes it very trivial to do this.

So suppose I have my address space for some function, A, and I have my address space for some function, B.

And suppose that function A references library one and module B references libraries one and two.

OK, so using the virtual memory system, suppose we have, so this is our physical memory.

So, suppose that module A, program A, is loaded.

And when it's loaded, the program that runs other programs, the loader in this case, is going to load this shared library, one, as it loads for program A.

So, it's going to first load the code for A, and then it's going to load the code for one.

And, these things are going to be sitting in memory somewhere.

So, within A's address space we're going to have the code for A is going to be mapped and the code for one is going to be mapped.

Now, when B executes, right, what we want to avoid, so the whole purpose of shared libraries is to make it so that when two programs are running and they both use the same library there aren't two copies of that library that are in memory using twice as much of the memory up, right, because there's all these libraries that all computer programs share.

For example, on Linux there's the libc library that implements all of the standard functions that people use in C programs.

If there's 50 programs written in C on your Linux machine, you don't want to have 50 copies of this libc library in memory.

You want to have just one.

So what we want is that when module B gets loaded, we want it to map this code that's already been mapped into the address space of A into its memory as well, OK?

And then, of course, we're going to have to load additional memory for B itself, and for the library two which A hasn't already loaded.

So now, those are going to be loaded into some additional locations in B's memory, are going to be loaded into some additional locations in the physical memory and mapped into A's memory.

So, this is just showing that we can actually use this notion of address spaces, in addition to using it to isolate two modules from each other so they can't refer to each other's memory, we can also use it as a way to allow two modules to share things with each other.

And in particular, this is a good idea in the case of things like shared libraries where we have two things, both programs need to be able to read the same data.

So they could use this to do that.

OK, so what we saw today was this notion of virtual memory and address spaces.

We saw how we have the kernel that's a trusted intermediary between two applications.

What we're going to see next time is how we can take this notion of virtualizing a computer.

We saw how to virtualize the memory today.

Next time we're going to see how we virtualize the processor in order to create the abstraction of multiple processors running on just one single, physical piece of hardware.

So I'll see you tomorrow.