MITOCW | ocw-6.033-lec6

0:00:00 So in case you are all worried that you may not be in the right 0:00:03.568 class, you are still in 6.033. If you're wondering what 0:00:06.676 happened to the Indian guy who normally lectures, 0:00:09.438 he's sitting right over there. We are trading off lectures 0:00:12.719 throughout the class. I'm going to do about the next 0:00:15.654 eight lectures up until spring break, and then Hari will be 0:00:19.165 back. So, my name is Sam, 0:00:20.546 and feel free to address any questions about the lecture to 0:00:23.884 me. So today we are going to keep 0:00:25.726 talking about this concept of enforcing modularity that we 0:00:29.007 started talking about last time. 0:00:32 0:00:40 So last time we saw how we could use this notion of the 0:00:44.302 client service model in order to separate two modules from each 0:00:49.243 other. So the idea was by running the 0:00:52.111 client and the service on separate machines, 0:00:55.537 we can isolate these two things from each other. 0:01:00 So we can make it so, for example, 0:01:02.306 when the client wants to invoke some operation on the server, 0:01:06.501 that the server can verify the request that the client makes, 0:01:10.696 and make sure that the client isn't asking to do something 0:01:14.68 malicious. Similarly, so this is what we 0:01:17.407 saw last time was this client service model. 0:01:20.413 And so the other benefit of the client service model was it 0:01:24.468 meant that we could decouple the client from the service. 0:01:28.382 So it meant that when the client issued a request against 0:01:32.297 the service, it didn't necessarily mean that the 0:01:35.583 client, if the service failed when the client issued the 0:01:39.428 request, it wasn't necessarily the case that the client would 0:01:43.623 also fail. So the server might crash 0:01:47.637 executing the request, and then the client would get a 0:01:51.109 timeout and would be able to retry. 0:01:53.336 And similarly, if the client failed, 0:01:55.628 the server could continue handling requests on the behalf 0:01:59.296 of other clients. OK, so that was a nice 0:02:02.146 separation, the ability to split the client and the server apart 0:02:05.431 from each other. That's a good thing. 0:02:07.308 But the problem with this approach is that we had to have 0:02:10.227 two separate machines. The way we presented this was 0:02:12.886 that the client was running on one computer, 0:02:15.127 and the server was running on another computer. 0:02:17.526 And if you think about this, this isn't exactly what we 0:02:20.341 want, right? Because that means, 0:02:21.957 suppose we want to build up a big, complicated computer system 0:02:25.137 that's composed of multiple different modules. 0:02:27.483 If we have to run each one of those modules on a separate 0:02:30.402 computer, that's not really very ideal, right? 0:02:34 To build up a big service, we're going to need a whole lot 0:02:37.853 of computers. And clearly, 0:02:39.543 that's not the way that computer systems that we 0:02:42.72 actually interact with work. So what we've seen so far is 0:02:46.505 this notion: we have a module per computer. 0:02:49.344 OK, and what we're going to do today is we're going to see how we can 0:02:52.927 generalize this so that instead of having one module per 0:02:56.645 computer, we can instead have multiple modules running within 0:03:00.701 a single computer. But when we do that, 0:03:04.197 we still want to maintain these nice sort of protection benefits 0:03:07.968 that we had between the client and service when these two 0:03:11.32 things were running on separate computers. 0:03:13.774 OK, so the way we are going to do that is by creating something 0:03:17.485 we call a virtual computer. So -- 0:03:20 0:03:29 What we're going to do is we're going to create this notion of 0:03:32.292 multiple virtual computers. They're all running on one 0:03:35.152 single computer. And then we're going to run 0:03:37.473 each one of these modules within a virtual computer. 0:03:40.226 OK, and I'll talk more about what I mean by a virtual 0:03:43.032 computer throughout the next couple of lectures. 0:03:45.569 But the idea is that a virtual computer has exactly the same, 0:03:48.808 to a program that's running on it, a virtual computers looks 0:03:51.992 just like one of these client or server computers might have 0:03:55.176 looked. Now, in particular, 0:03:56.58 a virtual computer has the same sorts of abstractions that we 0:03:59.818 studied in the previous lectures available to it that a real 0:04:03.002 computer has. So a virtual computer has a 0:04:06.8 virtual memory -- 0:04:08 0:04:13 -- and a virtual processor. So, today what we're going to 0:04:17.561 talk about is this notion of a virtual memory. 0:04:21.226 And so you guys have already probably seen the term virtual 0:04:25.95 memory in 6.004. So, the word should be familiar 0:04:29.778 to you. And we're going to use exactly 0:04:32.543 the same abstraction that we used in 6.004. 0:04:34.704 So we're just going to show how this abstraction can be used to 0:04:37.893 provide this protection between the client and server that we 0:04:40.98 want. OK, we're also going to 0:04:42.421 introduce today this notion of something called a kernel. 0:04:45.302 And a kernel is something that's in charge, 0:04:47.463 basically, of managing all of these different virtual 0:04:50.138 computers that are running on our one physical computer. 0:04:52.967 So the kernel is going to be the sort of system that is going 0:04:56.054 to be the piece of the system that actually knows that there 0:04:59.09 are multiple, virtual computers running on 0:05:01.199 this system. OK, so I want to start off by 0:05:05.23 first just talking about why we want a virtualized memory, 0:05:09.615 why virtualizing memory is a good thing. 0:05:12.615 So I'm going to abbreviate virtual memory as VM. 0:05:16.23 So why would we want to virtualize memory? 0:05:19.384 Well, let's see what might happen if we build a computer 0:05:23.615 system with multiple modules running on it at the same time 0:05:28.076 that didn't have a virtualized memory. 0:05:32 So suppose we have some microprocessor and we have some 0:05:37.779 memory. And within this memory, 0:05:40.989 we have the code and data for a couple of different modules is 0:05:47.518 stored. So we have, say, 0:05:49.979 module A and module B, and this is the things like the 0:05:55.652 code and the data that these modules are using to currently 0:06:01.859 execute. So, in this environment, 0:06:06.568 suppose the module, A, executes some instruction. 0:06:11.705 So if you think of this as memory, let's say that module A 0:06:17.806 begins at address A here, and module B begins at address 0:06:23.692 B here. So, suppose that module A 0:06:27.117 executes some instruction like store some value, 0:06:32.147 R1, in memory address, B. 0:06:36 OK, so module A writes in to memory address B here. 0:06:39.324 So, this can be a problematic thing, right, 0:06:42.116 because if the microprocessor just allows this to happen, 0:06:45.839 if it just allows A to write into module B's memory, 0:06:49.229 then module B has no way of being isolated from module A at 0:06:53.085 all, right? Module A can, 0:06:54.681 for example, write some sort of garbage into 0:06:57.54 the memory of module

right? Module A can, check this for example, write some sort of garbage into locations in the memory of module B. And then when module B tries to 0:07:01.876 read from that memory address, it will either, 0:07:04.601 say, try and execute an illegal instruction, or it will read in 0:07:08.355 some data that doesn't make any sense. 0:07:10.595 So we've lost the separation, the isolation that we had 0:07:13.865 between if module A and module B are a client and a service 0:07:17.376 trying to interact with each other. 0:07:19.435 We've lost this sort of ability. 0:07:21.312 We've lost this ability to separate and isolate each other, 0:07:24.763 isolate them from each other that we had when they were 0:07:28.033 running on separate computers. Furthermore, 0:07:31.879 this sort of arrangement of virtual memory. 0:07:34.959 So one problem with this is A can overwrite B's memory. 0:07:38.918 But we have another problem, which is that this sort of 0:07:42.877 arrangement also is very, very difficult to debug. 0:07:46.47 So, these kinds of bugs can be very hard to track down. 0:07:50.429 And really, when we're building computer systems, 0:07:53.949 we'd like to get rid of them. So, for example, 0:07:57.248 suppose that A stores into B's memory, overwrites an 0:08:00.987 instruction that somewhere sort of arbitrarily within B's 0:08:05.093 memory. And then, 10,000 instructions 0:08:08.702 later, B tries to load in that memory address and execute that 0:08:12.283 instruction. And it gets an illegal 0:08:14.278 instruction. Now, imagine that you have to 0:08:16.685 debug this program. So you sort of are sitting 0:08:19.327 there, and so you run application B, and it works just 0:08:22.731 fine. And then you run application A, 0:08:24.844 and A overwrites part of B's memory. 0:08:26.899 And now, when program B runs sometimes long after program A, 0:08:30.362 perhaps, is already terminated, B mysteriously crashes on you. 0:08:35 So how are you going to track down this problem? 0:08:37.603 It's very tricky, and it's the kind of problem 0:08:40.15 that's extremely hard to debug these kinds of issues. 0:08:43.094 So we really would like to make it so that A isn't able to 0:08:46.32 simply overwrite arbitrary things into B's memory. 0:08:49.094 We'd like to protect A from B. OK, and that's what this 0:08:52.15 virtual memory abstraction that we're going to talk about today 0:08:55.66 is going to do for us. So, if you think about how you 0:08:58.603 might protect A from B, the solution that you sort of 0:09:01.547 come up with after thinking about this and staring at this 0:09:04.773 for a little bit is that you want to verify that all of the 0:09:08.056 memory accesses that A does are actually valid memory accesses 0:09:11.509 that A should be allowed to make, so for example, 0:09:14.226 that refer to objects that are actually that A has allocated or 0:09:17.735 that A owns. OK, so in order to do that, 0:09:22.237 what we're going to need to do is to interpose some sort of 0:09:27.044 software in between the memory accesses that each module makes, 0:09:32.182 and the actual memory access. So the idea is as follows. 0:09:38.475 So this is the virtual memory abstraction. 0:09:43.955 OK, so the idea is as follows. For each one of our modules, 0:09:51.707 say A and B, we're going to create what's 0:09:57.054 known as an address space, OK? 0:10:02 And this address space is going to be, for example, 0:10:06.844 on a 32 bit computer, it's two to the 32 minus one, 0:10:11.688 it's going to be a 32 bit memory space. 0:10:15.37 So, this is the virtual address space of, say, 0:10:19.73 a process, A. So, process A can use any 0:10:23.411 memory address that's in this two to the 32 bit range, 0:10:28.546 OK? But this is going to be a 0:10:33.088 virtual address space. And what I mean by that is when 0:10:39.637 process A refers to some address within its memory space, 0:10:46.555 call it virtual address VA. That address is going to be 0:10:53.227 mapped by the virtual memory system into a physical address, 0:11:00.517 OK? So this virtual address here is 0:11:04.325 going to be mapped into some physical address -- 0:11:08 0:11:13 -- somewhere within the actual physical memory of the system, 0:11:18.179 OK? So this physical memory of the 0:11:21.028 system is also a 32 bit space. But the sort of mapping from 0:11:26.035 this virtual address in A's memory into this physical 0:11:30.525 address space is going to be handled by this virtual memory 0:11:35.532 system. So now the idea is that each 0:11:39.135 thing that, say, a process A or B might want to 0:11:42.644 reference is going to be mapped into some different location 0:11:47.144 within this memory. So, for example, 0:11:49.813 the code from process A or from module A is going to be mapped into one 0:11:54.542 location, and the code from module B is going to be mapped 0:11:58.889 into a different location in the physical address space. 0:12:04 OK, so we have this notion of address spaces. 0:12:06.787 Each module that is running on the system in a virtual memory 0:12:10.588 system is going to be allocated to one of these address spaces. 0:12:14.515 This address space is going to be, and this address space is 0:12:18.253 virtual in the sense that these addresses are only valid in the 0:12:22.18 context of this given module, OK? 0:12:24.208 And, they translate into, each address in the virtual 0:12:27.502 space translates into some address in the physical space. 0:12:32 Now, if you look at this for a minute, you might think, 0:12:34.892 well, this is kind of confusing because now B and A both have a 0:12:38.214 32 bit address space. And the computer itself only 0:12:40.839 has a 32 bit address space. So, there are sort of more 0:12:43.678 addresses between all the modules than the computer itself 0:12:46.732 physically has. And, if you remember back from 0:12:49.142 6.004, the story that we told you about that was that, 0:12:51.982 well, some of these virtual addresses can actually be mapped 0:12:55.142 into the disk on the computer. So in 6.004, 0:12:57.392 virtual memory was presented as a way to create a computer that 0:13:00.714 appeared to have more physical memory than it, 0:13:03.125 in fact, did. And the way that that was done 0:13:06.951 was to map some of these addresses onto disk. 0:13:09.813 So you might have addresses from B mapped onto disk. 0:13:13.13 And then, when B tries to reference an address that's 0:13:16.512 mapped onto disk, the system would load that data 0:13:19.634 from memory, load that data from disk and into memory. 0:13:23.081 So, we're not going to talk about that aspect of virtual 0:13:26.658 memory very much today. The thing to remember is just 0:13:30.632 that these virtual address spaces, there may be parts of 0:13:33.728 this virtual address space in each of the modules that isn't 0:13:37.05 actually mapped into any physical address space. 0:13:39.696 So if one of these modules tries to use one of these 0:13:42.567 unmapped virtual address spaces, it's not allowed to do that. 0:13:45.945 This virtual memory system will signal an error when it tries to 0:13:49.491 do that. And on your computer, 0:13:51.124 sometimes you'll see programs that will mysteriously crashed 0:13:54.445 with things that say things like, tried to access an illegal 0:13:57.767 memory address. When it does that, 0:13:59.625 that's because the program try to access some memory location 0:14:03.002 that wasn't mapped by the virtual memory system. 0:14:07 0:14:11 OK, so let's dive a little more into actually how this virtual 0:14:17.129 memory abstraction works so we can try to

let's dive a little more into actually how this virtual VM memory abstraction works so we can try to understand a little 0:14:23.157 bit more about what's going on. So this is going to be a 0:14:28.684 simplified VM hardware, OK? 0:14:32 It's a little bit simplified even from what you learned about 0:14:36.233 in 6.004. So, the idea in this simplified 0:14:39.055 hardware is that we have our processor. 0:14:41.737 OK, and then we're going to have this VM system, 0:14:45.053 which is sometimes called a memory management unit, 0:14:48.581 MMU. And, this is a piece of 0:14:50.486 hardware that's going to help us to do this mapping from these 0:14:54.79 logical addresses in the modules' address spaces into the 0:14:58.742 physical memory. And then, we're going to have 0:15:03.449 the physical memory, OK? 0:15:05.392 Now the idea is that when an instruction tries to access some 0:15:10.46 virtual address, so for example suppose we 0:15:13.923 execute instruction load some virtual address, 0:15:17.724 load into R1 some virtual address, OK? 0:15:20.85 What's going to happen when we do that is that the 0:15:24.989 microprocessor is going to send this virtual address to the VM 0:15:30.141 system. And then the VM system is going 0:15:34.265 to translate that into some physical address that can be 0:15:38.285 resolved within the memory itself. 0:15:40.697 OK, in the way that the virtual memory system is going to decide 0:15:45.302 this mapping between a virtual address and a physical address 0:15:49.687 is by using something that we call a page map, 0:15:52.976 OK? 0:15:54 0:16:01 OK, so this table is an example. 0:16:02.86 So this is a page map. And what a page map basically 0:16:05.919 has is its just a table of virtual address to physical 0:16:09.1 address mappings. So this is the virtual address 0:16:11.919 to physical address. So the idea is that when some 0:16:14.86 virtual address comes in here, the virtual memory manager 0:16:18.22 looks up that virtual address in this page map, 0:16:20.98 finds the corresponding physical address, 0:16:23.379 and then looks that physical address up in the actual memory. 0:16:28 OK, so now there's one more detail that we need, 0:16:30.975 right? So what this gives us is we 0:16:33.064 have this notion of a page map that does this mapping for us. 0:16:36.737 But we're missing a detail which is, OK, what we wanted was for 0:16:40.599 each one of these different modules that's running in the 0:16:44.144 system to have a different address space associated with 0:16:47.626 it. So what we want is we want to 0:16:49.652 have separate page maps for each of these different modules, 0:16:53.387 so, for A and B, OK, we're going to have a 0:16:55.983 different page map. And we're going to have this 0:16:58.959 sort of same, we might have multiple copies 0:17:01.618 of a particular virtual address in each one of these page maps. 0:17:07 And then what we're going to do is we're going to allocate a 0:17:10.503 special register on the hardware, on the processor. 0:17:13.471 We're going to add a little register that's going to allow 0:17:16.856 us to keep track of which one of these page maps we are currently 0:17:20.656 looking at, OK? So this thing is called the 0:17:23.15 PMAR or the Page Map Address Register. 0:17:26 0:17:35 OK, and the page map address register simply points at one of 0:17:38.963 these page maps. OK, so what happens is that the 0:17:42.068 virtual memory system, when it wants to resolve a 0:17:45.239 virtual address, looks at this page map address 0:17:48.277 register and uses that to find a pointer to the beginning of the 0:17:52.439 page map that's currently in use. 0:17:54.553 And then he uses the page map that's currently in use to 0:17:58.186 resolve, to get what physical address corresponds to this 0:18:01.886 logical address. OK so this is really the core 0:18:06.123 concept from virtual memory. So what we have now is we have 0:18:10.371 this page map address register that can be used to select which 0:18:14.912 one of these address spaces we are currently using. 0:18:18.574 OK, and so when we have selected, for example, 0:18:21.87 the page map for module A, then module A can only refer to 0:18:26.045 virtual addresses that are in its page map. 0:18:30 And those virtual addresses can only map into certain physical 0:18:33.812 addresses. So, for example, 0:18:35.437 suppose this block here is the set of virtual addresses, 0:18:38.875 the set of physical addresses that correspond to the virtual 0:18:42.562 addresses in A's page map. OK, these are the only physical 0:18:46.125 addresses that A can talk about. So if we, for example, 0:18:49.5 have a different block of memory addresses that correspond 0:18:53.062 to the virtual addresses that B can reference, 0:18:55.875 we can see that there's no way for module A to be able to 0:18:59.375 reference any of the memory that B uses. 0:19:03 So we are able to totally separate the physical memory, 0:19:07.526 pieces of physical memory that A and B can talk about by using 0:19:12.638 this page map mechanism that virtual memory gives us. 0:19:16.997 So, basically what we've done is we've sort of added this 0:19:21.69 extra layer of indirection, this virtual memory system, 0:19:26.216 that gets to map virtual addresses into physical 0:19:30.156 addresses. So the rest of this lecture is 0:19:34.389 really going to be details about how we make this work, 0:19:38.69 about how we actually decide, how we assign this PMAR 0:19:42.831 register based on which one of the modules is currently 0:19:47.132 executing about things like what the format of this page map 0:19:51.831 table is going to look like, OK? 0:19:54.3 So this is really the key concept. 0:19:56.929 So what I want to do now is sort of turn to this second 0:20:01.23 question I asked which is, how does this page map thing 0:20:05.53 actually work? How is it actually represented? 0:20:10.208 So one very simple representation of a page map 0:20:13.485 might be that it simply is a pointer to, the page map just 0:20:17.546 says where A's memory begins on the processor, 0:20:20.751 right? So it's just one value. 0:20:22.817 It says A's memory begins at this location in the physical 0:20:26.878 memory, and all virtual addresses should be resolved 0:20:30.511 relative to this beginning location of A's memory. 0:20:35 The problem with that representation is that it's not 0:20:39.193 very flexible. So, for example, 0:20:41.612 suppose there's a third module, C, which is laid out in memory 0:20:46.532 right next to A. So, its storage is placed right 0:20:50.322 next to A in memory. And now, suppose that A wants 0:20:54.274 to allocate some additional memory that it can use. 0:20:58.306 Now, in order to do that, if the page map is simply a 0:21:02.5 single pointer to the beginning of A's address space, 0:21:06.693 we're kind of in trouble. We can't just add memory onto 0:21:11.661 the bottom because then we would overlap C. 0:21:14.067 So we're going to have to move all of A's memory around in 0:21:17.332 order to be able to make space for this. 0:21:19.566 OK, so this seems like a little bit problematic simply have it 0:21:23.06 be a pointer. The other thing we could do is 0:21:25.523 suppose we could, instead, have a different 0:21:27.929 option where we could say, for example, 0:21:30.106 for every virtual address in A's address space, 0:21:32.741 so for each 32 bit value that A wants to resolve, 0:21:35.49 there might be an entry in this page map table, 0:21:38.126 right? So for every 32-bit virtual 0:21:40.016 address, there would be a corresponding 32-bit physical 0:21:43.109 address. And there would just

be a 0:21:46.282 one-to-one mapping between all these things. 0:21:48.487 So, if A could reference a million blocks of memory, 0:21:51.102 there would be a million entries in this page map table. 0:21:53.923 So, if you think about this for a minute, that sounds like kind 0:21:57.102 of a bad idea, too, because now these tables 0:21:59.307 are totally huge, right, and in fact they are 0:22:01.564 almost as big as the memory itself, right, 0:22:03.666 because if I have a million entries, if A can reference a 0:22:06.538 million blocks, then I'm going to need a 0:22:08.538 million entries in the table. So the table becomes just as 0:22:13.688 big as the memory itself. So we need some in between sort 0:22:19.064 of alternative hybrid between these two extremes. 0:22:23.672 And the idea, again, is very simple. 0:22:27.031 And you saw it in 6.004. So the idea is to take this 0:22:31.928 32-bit virtual address. So suppose this is our 32-bit 0:22:37.517 virtual address. Now what we're going to do is 0:22:41.423 we're going to split it up into two pieces, a page number and an 0:22:46.892 offset. OK, and we're going to choose 0:22:50.017 some size for these two things. For now I'll just arbitrarily 0:22:55.225 pick a 20 bit page number and a 12 bit offset. 0:23:00 OK, so what this is going to do, so now what we're going to 0:23:05.247 do is instead of storing a single word of memory at each 0:23:10.223 entry in this table, we're going to store a page of 0:23:14.747 memory at each entry in this table. 0:23:17.823 So -- 0:23:19 0:23:29 So this table is going to look like a mapping between a page, 0:23:35.172 and a physical address. OK, so what a page is, 0:23:39.801 so if the page number is 20 bits long, then that means that 0:23:45.767 each page is going to be two to the 12th bits big, 0:23:50.808 which is equal to, say, 4,096 words, 0:23:54.409 OK? So the idea is that we're going 0:23:57.906 to have two to the 20th pages within each address space, 0:24:03.564 and each page is going to map to one of these 4,096 byte 0:24:09.222 blocks, OK? So, if we have our memory here 0:24:15.733 this page, say, page one is going to map into 0:24:21.21 some physical address. And page two is going to map 0:24:27.433 into some other physical block, OK, so each one of these things 0:24:35.15 is now 4,096 bytes, each block here, 0:24:39.506 OK? And so this, 0:24:41.965 let's just expand this. So this is now a page. 0:24:45.587 And this 12 bit offset is going to be used in order to look up 0:24:50.497 the word that we want to actually look up in this page. 0:24:54.843 So, if the virtual address is, say, for example, 0:24:58.626 page one offset zero, what that's going to do is 0:25:02.409 we're going to look up in the page map. 0:25:05.468 We're going to find the page number that corresponds to, 0:25:09.895 we're going to find page number one. 0:25:14 We're going to find the physical address that 0:25:16.885 corresponds to it, we're going to go down here, 0:25:19.901 and look at this block of memory. 0:25:22 And then within this 4,096 block memory, 0:25:24.557 we're going to take the address zero, the first word within that 0:25:28.688 thing, OK? So now the size of these page 0:25:32.215 tables is much smaller, right? 0:25:34.431 They're no longer two to the 30th bits. 0:25:37.334 Now they're two to the 20th bits, which is some small number 0:25:41.842 of megabytes big. But we've avoided this problem 0:25:45.433 that we have before. We have some flexibility in 0:25:49.024 terms of how each page maps into the physical memory. 0:25:52.997 So I can allocate a third page to a process. 0:25:56.283 And I can map that into any 4,096 byte block that's in 0:26:00.332 memory that's currently unused. OK, so I have flexibility. 0:26:05.75 I don't have this problem, say, where A and C were 0:26:09.048 colliding with each other. 0:26:11 0:26:19 OK, so this is sort of the outline of how virtual memory 0:26:23.147 works. But what I haven't yet 0:26:25.259 described to you is how it is that we can actually go about 0:26:29.632 creating these different address spaces that are allocated to the 0:26:34.459 different modules, and how we can switch between 0:26:38.003 different modules using this PMAR register. 0:26:42 So I've sort of described this as, suppose these data 0:26:45.035 structures exist, now here's how we can use them. 0:26:47.838 But I haven't told you how these data structures all get 0:26:51.048 together, and created, and set up to begin with. 0:26:53.792 And I hinted at this in the beginning. 0:26:55.952 But in order to do this, what we're going to need is 0:26:58.93 some sort of a special supervisory module that's able 0:27:01.966 to look at the page maps for all of these different, 0:27:04.943 that's able to create new page maps and look at the page maps 0:27:08.446 for all of the different modules that are within the system. 0:27:13 And the supervisory module is going to be able to do things 0:27:16.992 like add new memory to a page map or be able to destroy, 0:27:20.778 delete a particular module and its associated page map. 0:27:24.496 So we need some sort of a thing that can manage all this 0:27:28.282 infrastructure. So, this supervisory module -- 0:27:32 0:27:37 -- is called the kernel. OK, and the kernel is really, 0:27:41.253 it's going to be the thing that's going to be in charge of 0:27:45.827 managing all these data structures for us. 0:27:49.117 So -- 0:27:50 0:27:58 So here's our microprocessor with its PMAR register on it. 0:28:02.519 And, what we're going to do is we're going to extend the 0:28:06.879 microprocessor with one additional piece of hardware, 0:28:11.002 and this is the user kernel bit, OK? 0:28:13.777 So, this is just a bit specifies whether we are 0:28:17.424 currently running a user module that is just a program that's 0:28:22.181 running on your computer, or whether the kernel is 0:28:26.066 currently executed, OK? 0:28:29 And the idea with this kernel bit is that when this kernel bit 0:28:34.854 is set, the code that is running is going to have special 0:28:40.229 privileges. It's going to be able to 0:28:43.588 manipulate special things about the hardware and the processor. 0:28:49.538 And in particular, we're going to have a rule that 0:28:54.241 says that only the kernel can change the PMAR, 0:28:58.56 OK? So the PMAR is the thing that 0:29:01.592 specifies which process is currently running, 0:29:04.009 and selects which address space we want to be currently using. 0:29:07.36 And what we're going to use is we're going to use, 0:29:10.051 so what we're going to do is have the kernel be the thing 0:29:13.127 that's in charge of manipulating the value of this PMAR register 0:29:16.588 to select which thing is currently being executed. 0:29:19.279 And we want and they get so that only the kernel can do this 0:29:22.52 because if we, for example, 0:29:23.948 allowed one of these other programs to be able to 0:29:26.584 manipulate the PMAR, right, then that other program 0:29:29.331 might be able to do something unpleasant to the computer, 0:29:32.407 right? It changes the PMAR to point at 0:29:35.712 some other program's memory. And now, suddenly all the 0:29:38.64 memory addresses in the system are going to be resolved. 0:29:41.678 Then suddenly, we are going to be sort of 0:29:43.887 resolving memory addresses relative to some other module's 0:29:47.036 page map. And that's likely to be a 0:29:48.914 problematic thing. It's likely to cause that other 0:29:51.62 module to crash, for example, 0:29:53.167 because the processor is set up to be

executing instructions 0:29:56.426 from the current program. So we want to make sure that 0:29:59.354 only something this kernel can change the PMAR. 0:30:03 And this kernel is going to be this sort of supervisory module 0:30:06.664 that all of the other modules are going to have to trust to 0:30:10.149 kind of do the right thing and manage the computer's execution 0:30:13.813 for you. And this kernel, 0:30:15.255 except for this one difference that a kernel can change the 0:30:18.74 PMAR, the kernel is, in all other respects, 0:30:21.263 essentially just going to be another module that's running in 0:30:24.868 the computer system. So in particular, 0:30:27.091 the kernel is also going to have one of these 32-bit virtual 0:30:30.635 address spaces associated with it, OK? 0:30:34 But, what we're going to do is we're going to say that the 0:30:38.034 kernel within its address space has all of the page maps of all 0:30:42.422 the other programs that are currently running on the system 0:30:46.527 mapped into its address space. OK, so this is a little bit 0:30:50.561 tricky because I presented this as though A and B referenced 0:30:54.737 totally different pieces of memory. 0:30:57.143 So, I sort of have told you so far only about modules that are 0:31:01.461 referencing disjoint sets of memory. 0:31:05 But in fact these page maps, right, they just reside in 0:31:08.857 memory somewhere. And it's just fine if I, 0:31:11.785 for example, have multiple modules that are 0:31:14.785 able to reference, that have the same physical 0:31:18 addresses mapped into their virtual address space. 0:31:21.5 So it's very likely that I might want to have something 0:31:25.357 down here at the bottom that both A and B can access. 0:31:30 So, this might be stuff that's shared between A and B. 0:31:33.319 I can map that into both A and B's memory. 0:31:35.887 In the same way, what I'm going to do is I'm 0:31:38.58 going to map these page maps, which are also stored in memory 0:31:42.338 into all of the page maps into the kernel's address space. 0:31:45.908 So the kernel is going to be able to reference the address 0:31:49.478 spaces of all the modules. And the kernel is also going to 0:31:53.048 keep a little table of all the page maps for all of the 0:31:56.43 currently running programs on the system. 0:32:00 So these are, for example, 0:32:01.475 page maps for A and B. And this is a list of all the 0:32:04.484 maps that are currently running in the system. 0:32:07.139 So what's going to happen, now what can happen is because 0:32:10.443 the kernel is allowed to change the PMAR, and because it knows 0:32:14.042 about the location of all the other address spaces that are in 0:32:17.641 the system, when it wants to start running one of these 0:32:20.827 programs that's running in the system, it can change the value 0:32:24.426 of the PMAR to be sort of the PMAR for A or the PMAR to B. 0:32:27.789 And, it can manipulate all the values of the registers in the 0:32:31.329 system so that you can start executing code for one of these. 0:32:34.869 You can switch between one of these two modules. 0:32:39 So the actual process of switching between which module 0:32:41.856 is currently running. We're going to focus on that 0:32:44.448 more next time. So, don't worry too much if you 0:32:46.881 don't understand the details of how you actually switch from 0:32:50.002 executing one program to another program. 0:32:52.118 But, you can see that the kernel can switch which address 0:32:55.08 space is currently active by simply manipulating the value of 0:32:58.254 this PMAR register. Furthermore, 0:33:00.781 the kernel can do things like it can create a new map. 0:33:04.231 So the kernel can simply allocate one of these new 0:33:07.421 tables, and it can set up a mapping from a set of virtual 0:33:11.066 addresses to a set of physical addresses so that you can create 0:33:15.102 a new address space that a new module can start executing 0:33:18.747 within. Similarly, the kernel can do 0:33:21.026 things like allocate new memory into one of these addresses. 0:33:24.867 So it can map some additional virtual addresses into real 0:33:28.512 physical memory so that when one of these modules, 0:33:31.702 say for example, requests additional memory to 0:33:34.631 execute, the kernel can add that memory, add an entry into the 0:33:38.602 table so that that new memory that the program has requested 0:33:42.443 actually maps into a valid, physical address. 0:33:47 OK, so the question, of course, then is, 0:33:49.99 so you can see the value of having this kernel module. 0:33:54.054 But the question is, how do we communicate between 0:33:57.811 these modules that are running these user level modules that 0:34:02.335 are running, and the kernel module that the user modules 0:34:06.552 need to invoke, for example, 0:34:08.623 request new memory. So the way that we are going to 0:34:12.791 do this is just like we've done everything else in this lecture 0:34:16.373 so far: by adding a little bit of extra, by changing the 0:34:19.551 processor just a little bit. So in particular, 0:34:22.151 what we're going to do -- 0:34:24 0:34:37 What we're going to do is to is to add this notion of a 0:34:41.909 supervisor call. Call that SVC. 0:34:44.636 So a supervisor call is simply a special instruction that is on 0:34:50.272 the processor that invokes the kernel. 0:34:53.636 So when the supervisor call instruction gets executed, 0:34:58.454 it's going to set up the state of these PMAR and user kernel 0:35:03.818 bit instructions so that the kernel can begin executing. 0:35:10 The supervisor call is also going. 0:35:12.435 But when the supervisor call instruction is executed, 0:35:16.272 we need to be a low but careful because we also need to decide 0:35:20.773 somehow which code within the kernel we want to begin 0:35:24.61 executing when the supervisor call instruction gets executed, 0:35:29.038 right? So what the supervisor call 0:35:31.473 instruction does is it accepts a parameter which is the name of a 0:35:36.195 so-called gate function. So a gate function is a well 0:35:41.185 defined entry point into another module that can be used to 0:35:45.713 invoke a piece of code in that other module. 0:35:49.07 So, for example, the kernel is going to have a 0:35:52.583 particular gate function which corresponds to allocating 0:35:56.877 additional memory for a module. So when a user program executes 0:36:03.886 an instruction, supervisor call, 0:36:07.773 to gate say for example, this might be some memory 0:36:13.917 allocation code, this special instruction is 0:36:19.308 going to do the following things. 0:36:23.321 First it's going to set the user kernel bit to kernel. 0:36:29.966 Then it's going to set the value of the PMAR register to 0:36:36.862 the kernel page map. It's going to save the program 0:36:43.639 counter for the currently executing instruction, 0:36:47.775 and then it's going to set the program counter to be the 0:36:52.615 address of the gate function, OK? 0:36:55.431 So we've introduced a special new processor instruction that 0:37:00.623 takes these steps. So when this instruction is 0:37:05.439 executed, we essentially switch into executing within the 0:37:10.498 kernel's address space. OK, and this kernel, 0:37:14.383 And, we begin executing within the kernel address space at the 0:37:19.713 address of this gate function within the kernel's address 0:37:24.772 space. OK, so what this has done is 0:37:27.844 this is a well defined entry point,

kernel address 0:37:12 in P2 space. OK, so what this has done is set P2 to 1. This is a well-defined entry point 0:37:32 If the program tries to execute this instruction with the name 0:37:35.302 of a gate function that doesn't exist, then that program is 0:37:38.443 going to get an error when it tries to do that. 0:37:40.934 So the program can only name gate functions that actually 0:37:43.966 correspond to real things that the operating system can do, 0:37:47.106 that the kernel can do. And so, the kernel is then 0:37:49.759 going to be invoked and take that action that was requested 0:37:52.9 of it. On return, essentially, 0:37:54.47 so when the kernel finishes executing this process, 0:37:57.177 when it finishes executing this, say, memory allocation 0:38:00.101 instruction, we are just going to reverse this step. 0:38:04 So we are just going to set the PMAR to be, we're going to set 0:38:07.951 the PMAR back to the user's program. 0:38:10.218 We are going to set the user kernel bit back into user mode. 0:38:14.04 And then we're going to jump back to the saved return address 0:38:17.927 that we saved here, the saved program counter 0:38:20.777 address from the user's program. So when the program finishes, 0:38:24.728 when the kernel finishes executing this service 0:38:27.708 instruction, the control will just return back to the program 0:38:31.595 in place where the program needs to begin executing. 0:38:36 OK, so now using this gate, so using this notion of, 0:38:40.008 so what we've seen so far now is the ability for the, 0:38:44.096 we can use the virtual memory abstraction in order to protect 0:38:48.812 the memory references of two modules from each other. 0:38:52.899 And we see when we have this notion of the kernel that can be 0:38:57.615 used to, for example, that can be used to manage 0:39:01.31 these address spaces to allocate new memory to these address 0:39:05.947 spaces, and in general, to sort of manage these address 0:39:10.192 spaces. So this kernel is also going to 0:39:15.316 allow us to do exactly what we set out trying to do, 0:39:20.772 which is to act as the so-called trusted intermediary 0:39:26.334 between, say for example, a client and a server running 0:39:32.111 on the same machine. 0:39:35 0:39:40 OK, so I trust that intermediary is just a piece of 0:39:42.631 code. So suppose we have a client and 0:39:44.526 a server, right, and these are two pieces of 0:39:46.789 code written by two different developers. 0:39:48.894 Maybe the two developers don't necessarily trust that the other 0:39:52.157 developer has written a piece of code that's 100% foolproof 0:39:55.21 because it doesn't have any bugs. 0:39:56.894 But both of those developers may be willing to say that they 0:40:00 will trust that the kernel is properly written and doesn't 0:40:03 have any bugs in it. And so they are willing to 0:40:07.159 allow the kernel to sit in between those two programs and 0:40:11.81 make sure that neither of them has any bad interactions with 0:40:16.71 each other. So let's see how we can use 0:40:19.867 this kernel, the kernel combined with virtual memory in order to 0:40:25.099 be able to do this. So suppose this is our kernel. 0:40:30 So the idea is that this kernel running has, so suppose we have 0:40:33.386 two processes A and B that want to communicate with each 0:40:36.555 other in some way. And, we already said that we 0:40:39.068 don't want these two processes to be able to directly reference 0:40:42.455 into each other's memory because that makes them dependent on 0:40:45.732 each other. It means that if there's a bug 0:40:47.972 in A, that A can overwrite some memory in B's address space and 0:40:51.359 cause B to crash. So, that's' a bad thing. 0:40:54.2 So instead, what we are going to do is we're going to create, 0:40:57.477 well, one thing we can do is to create a special set of these 0:41:00.755 supervisor calls that these two modules can use to interact with 0:41:04.197 each other. So in particular, 0:41:07.301 we might within the kernel maintain a queue of messages 0:41:11.435 that these two programs are exchanging with each other, 0:41:15.569 a list of messages that they're exchanging. 0:41:18.784 And then, suppose A is, we're going to call this guy, 0:41:22.765 A is a producer. He's creating messages. 0:41:25.751 And B is a consumer. A can call some function like 0:41:29.502 Put which will supervise their call like Put which will cause a 0:41:34.248 data item to be put on this queue. 0:41:38 And then sometime later, B can call this supervisor call 0:41:40.963 Get, and pull this value out of the queue, OK? 0:41:43.387 So in this way, the producer and the consumer 0:41:45.758 can interact with each other. They can exchange data with 0:41:48.775 each other and because we have this gate interface, 0:41:51.469 this well-defined gate interface with these Put and Get 0:41:54.379 calls being invoked by the kernel, the kernel can be very 0:41:57.396 careful, just like in the case of the client and server running 0:42:00.737 on two different machines. In this case, 0:42:03.849 the kernel can carefully verify that these put and get commands 0:42:07.901 that these two different modules are calling actually are 0:42:11.562 correctly formed, that the parameters are valid, 0:42:14.633 that they're referring to valid locations in memory. 0:42:17.967 And therefore, the kernel can sort of ensure 0:42:20.777 that these two modules don't do malicious things to each other 0:42:24.764 or cause each other to break. So this is an example here of 0:42:28.555 something that's like an inter-process communication. 0:42:33 So you saw, for example, you've seen an instance of 0:42:35.859 inter-process communication when you studied the UNIX paper. 0:42:39.176 We talked about pipes, and a pipe abstraction, 0:42:41.75 and how that works. Well, pipes are sort of 0:42:44.152 something that's implemented by the kernel in UNIX as a way for 0:42:47.698 two programs to exchange data with each other. 0:42:50.272 And, there's lots of these kinds of services that our 0:42:53.246 people tend to push into the kernel that the kernel provides 0:42:56.62 to the other applications, the modules that are running, 0:42:59.766 so that these modules can, for example, 0:43:01.939 interact with hardware or interact with each other. 0:43:06 So commonly within a kernel, you'd find things like a file 0:43:09.321 system, an interface to the network, and you might, 0:43:12.235 for example, find things like an interface 0:43:14.624 to the graphics hardware. OK, there is some sort of 0:43:17.538 so if you look at what's actually within a kernel, 0:43:20.86 there is a huge amount of code that's going into these kernels. 0:43:24.473 So I think we talked earlier about how the Linux operating 0:43:27.794 system is many millions of lines of code. 0:43:31 If you go look at the Linux kernel, the Linux kernel is 0:43:34.181 probably today on the order of 5 million lines of code, 0:43:37.717 most of which, say two thirds of which, 0:43:39.956 is related to these so-called device drivers that manage this 0:43:43.491 low-level hardware. So this kernel has gotten quite 0:43:46.437 large. And one of the side effects of 0:43:48.558 the kernel getting large is that maybe it's harder to trust it, 0:43:52.212 right? May be you sort of have less 0:43:54.215 confidence that all the code in the kernel is actually correct. 0:43:57.868 And you can imagine that if you don't trust the kernel then the 0:44:01.521 computer is not going to be as stable as you would like to be. 0:44:06 And this is one argument for why Windows crashes all the time 0:44:09.317 is because it has all these drivers in it and these

argument for why Windows crashes all the time or Mac OS X is because it has all these drivers in it and these drivers 0:44:12.525 aren't necessarily all perfectly written. 0:44:14.737 There are tens of millions of lines of code in Windows, 0:44:17.723 and some of them crash some of the time, and that causes the 0:44:20.986 whole computer to come down. So there's a tension in the 0:44:24.027 operating systems community about whether you should execute 0:44:27.29 these things, you should keep these things 0:44:29.557 like the file system or graphics system within the kernel or 0:44:32.82 whether you should move them outside as separate services, 0:44:35.972 which can be invoked in the same way, for example, 0:44:38.682 that A and B interact with each other, by having some data 0:44:41.834 structure that's stored within the kernel that buffers the 0:44:44.986 requests between, say, the service and this, 0:44:47.364 say, for example the graphics service and the users' programs 0:44:50.682 that want to interact with it. OK, so that's basically it for 0:44:56.596 today. What I've shown you is, 0:44:59.105 well, actually we have a few more minutes. 0:45:03 Sorry. [LAUGHTER] I know you're all 0:45:06.41 getting up to leave, but so, OK, I just want to 0:45:11.023 quickly touch on one last topic which is that, 0:45:15.536 so, what I've shown you so far is how we can use the notion of 0:45:21.654 virtual memory in order to protect the data, 0:45:25.967 protect two programs from each other so that they can't 0:45:31.383 necessarily interact with each other's data. 0:45:37 But there is some situations in which we might actually want to 0:45:40.671 have two programs able to share some data with each other. 0:45:44.046 So I don't know if you guys remember, but when Hari was 0:45:47.243 lecturing earlier, he talked about how there are 0:45:50.026 libraries that get linked into programs. 0:45:52.335 And one of the common ways that libraries are structured these 0:45:55.947 days is a so-called shared library. 0:45:57.96 So a shared library is something that is only stored in 0:46:01.157 one location in physical memory. But multiple different modules 0:46:05.768 that are executing on that system can call functions within 0:46:09.188 that shared library. So in order to make this work, 0:46:12.136 right, we need to have mapped the memory for the shared 0:46:15.319 library that has all the functions that these other 0:46:18.267 modules want to call into the address spaces for both of these 0:46:21.863 modules so that they can actually execute the code that's 0:46:25.165 there. So the virtual memory system 0:46:27.17 makes it very trivial to do this. 0:46:30 So suppose I have my address space for some function, 0:46:35.169 A, and I have my address space for some function, 0:46:39.941 B. And suppose that function A 0:46:42.824 references library one and module B references libraries 0:46:48.292 one and two. OK, so using the virtual memory 0:46:52.567 system, suppose we have, so this is our physical memory. 0:46:58.035 So, suppose that module A, program A, is loaded. 0:47:04 And when it's loaded, the program that runs other 0:47:08.051 programs, the loader in this case, is going to load this 0:47:12.694 shared library, one, as it loads for program A. 0:47:16.577 So, it's going to first load the code for A, 0:47:20.207 and then it's going to load the code for one. 0:47:23.922 And, these things are going to be sitting in memory somewhere. 0:47:30 So, within A's address space we're going to have the code for 0:47:33.726 A is going to be mapped and the code for one is going to be 0:47:37.329 mapped. Now, when B executes, 0:47:39.068 right, what we want to avoid, so the whole purpose of shared 0:47:42.732 libraries is to make it so that when two programs are running 0:47:46.459 and they both use the same library there aren't two copies 0:47:50 of that library that are in memory using twice as much of 0:47:53.478 the memory up, right, because there's all 0:47:55.962 these libraries that all computer programs share. 0:48:00 For example, on Linux there's the LIB C 0:48:02.782 library that implements all of the standard functions that 0:48:06.651 people use in C programs. If there's 50 programs written 0:48:10.384 in C on your Linux machine, you don't want to have 50 0:48:13.914 copies of this LIB C library in memory. 0:48:16.493 You want to have just one. So what we want is that when 0:48:20.158 module B gets loaded, we want it to map this code 0:48:23.416 that's already been mapped into the address space of A into its 0:48:27.624 memory as well, OK? 0:48:30 And then, of course, we're going to have to load 0:48:32.694 additional memory for B itself, and for the library two which A 0:48:36.249 hasn't already loaded. So now, those are going to be 0:48:39.173 loaded into some additional locations in B's memory, 0:48:42.097 are going to be loaded into some additional locations in the 0:48:45.48 physical memory and mapped into A's memory. 0:48:47.888 So, this is just showing that we can actually use this notion 0:48:51.328 of address spaces, in addition to using it to 0:48:53.851 isolate two modules from each other so they can't refer to 0:48:57.119 each other's memory, we can also use it as a way to 0:48:59.986 allow two modules to share things with each other. 0:49:04 And in particular, this is a good idea in the case 0:49:06.563 of things like shared libraries where we have two things, 0:49:09.492 both programs need to be able to read the same data. 0:49:12.16 So they could use this to do that. 0:49:13.887 OK, so what we saw today was this notion of virtual memory 0:49:16.869 and address spaces. We saw how we have the kernel 0:49:19.38 that's a trusted intermediary between two applications. 0:49:22.205 What we're going to see next time is how we can take this 0:49:25.134 notion of virtualizing a computer. 0:49:26.861 We saw how to virtualize the memory today. 0:49:30 Next time we're going to see how we virtualize the processor 0:49:33.064 in order to create the abstraction of multiple 0:49:35.402 processors running on just one single, physical piece of 0:49:38.259 hardware. So I'll see you tomorrow.