MITOCW | ocw-6.033-lec7

0:00:00 So today we're going to continue our discussion of 0:00:05.025 enforcing modularity. And in particular if you 0:00:09.641 remember last time we spent a lot of time talking about -- 0:00:16 0:00:21 Last time we spent a lot of time talking about how we can 0:00:24.438 create a virtual memory so that we can have multiple programs 0:00:28.122 running on top of one piece of hardware that all appear to have 0:00:31.929 a separate address space or separate memory. 0:00:35 So, two lectures ago, we talked about having one 0:00:41.512 module per computer. And then last time we talked 0:00:48.162 about this notion of wanting to have a module per virtual 0:00:55.921 computer. 0:00:58 0:01:02 And we saw this notion of virtual memory. 0:01:04.68 OK, and now this time what we're going to do is we're going 0:01:08.567 to talk about the other piece that are virtual computer needs, 0:01:12.654 which is a virtual processor. 0:01:15 0:01:21 OK, so the idea that we want here when we are designing a 0:01:24.897 virtual processor is to create, have our programs be able to 0:01:29.003 run on a single, physical processor, 0:01:31.439 but have different programs be able to run as though they were 0:01:35.684 the only program that was executing on that processor. 0:01:40 So we want the programs to be written in a style where the 0:01:43.433 programmer doesn't have to be aware of the other programs that 0:01:47.108 are running on the machine, but where the machine creates 0:01:50.481 sort of a virtual processor for each one of these programs 0:01:53.915 that's running so that each program has its own sort of set 0:01:57.409 of instructions that it executes. 0:02:00 And those instructions appear to be independent of all the 0:02:03.352 other programs that are running. So in order to create this 0:02:06.764 illusion of a virtual processor -- 0:02:09 0:02:14 -- we are going to introduce a very simple concept. 0:02:19.847 We're going to have each program run in what's known as a 0:02:26.397 thread. And thread is really short for 0:02:30.725 a thread of execution. And a thread is just a 0:02:35.692 collection of all of the state that we need in order to keep 0:02:40.23 track of what one single given program is doing at a particular 0:02:45 point in time. So a thread is a collection of 0:02:48.384 the instructions for the program that the module is running, 0:02:52.923 as well as the current state of the program. 0:02:57 In particular, it's the value of the registers 0:03:00.152 on the processor including of particular import for this 0:03:04.006 lecture the program counter and the stack pointer. 0:03:07.439 OK, so there's a bunch of registers on the processor, 0:03:11.082 say, 32 registers, and there also are these 0:03:14.025 special registers, the program counter, 0:03:16.687 that keeps track of which instruction is currently being 0:03:20.541 executed, and the stack pointer which keeps track of where on 0:03:24.745 the stack values are currently being pushed or popped from. 0:03:30 So these things combined together with the stack are 0:03:33.416 really the things that define one single thread. 0:03:36.564 So, and the way to think about this is if you think about a 0:03:40.449 program that's running, you can really encapsulate 0:03:43.732 almost everything about what that program is currently doing 0:03:47.684 by the instructions that it has to execute, the value of the 0:03:51.636 registers that are currently set in the program, 0:03:54.784 and the state that's on the stack. 0:03:58 Now, programs may also have some sort of data that's stored 0:04:01.071 in the global memory in the global variables are things that 0:04:04.196 are stored on the heap that have been allocated via some memory 0:04:07.48 allocation call. And so those things are a part 0:04:09.916 of the program as well, although I don't want to 0:04:12.406 explicitly list them as being a part of a single thread because 0:04:15.689 if two programs, as we'll see, 0:04:17.225 two programs can share the same address space. 0:04:19.609 And if two programs are sharing the same address space, 0:04:22.469 then they're going to share those global variables, 0:04:25.117 the stuff that's stored on the heap of the program. 0:04:27.765 So think of a thread as these things, plus there is some 0:04:30.678 additional, we'll call it a heap, set of memory that may be 0:04:33.75 shared between several different threads that are all running 0:04:36.928 within the same address space. And I'll explain more about 0:04:41.629 that as we go. OK, so as I said, 0:04:43.5 now what we want is to create this illusion that each one of 0:04:47.06 these threads has its own virtual processor upon which 0:04:50.258 it's running. So how are we going to 0:04:52.37 accomplish that? So if you think about this for 0:04:55.146 a second, the way to accomplish this is simply to allow each one 0:04:58.948 of these, each thread to update these various registers, 0:05:02.267 push things onto the stack, and then to have some function 0:05:05.706 which we can call, which will save all of this 0:05:08.422 current state off for the current thread that's executing, 0:05:11.862 and then load the state in for another thread that might be 0:05:15.362 executing, and stop the current thread from executing, 0:05:18.56 and start the new one executing. 0:05:22 So the idea is if you were to look at a timeline of what the 0:05:26.637 processor was doing, and if the processor is running 0:05:30.646 two threads, thread one and thread two, you would see that 0:05:35.126 during some period of time, thread one would be running. 0:05:40 Then the computer would switch over and thread two would start 0:05:43.16 running, and then again the computer would switch, 0:05:45.699 and thread one would start running. 0:05:47.461 So we are going to multiplex the processor resource in this 0:05:50.466 way. So we have one processor that 0:05:52.176 can be executing instructions. And at each point in time, 0:05:55.077 that processor is going to be executing instructions from one 0:05:58.186 of these two threads. And when we switch in these 0:06:01.442 points in between threads, we are going to have to save 0:06:04.328 off this state for the thread that was running, 0:06:06.786 and restore the state for the thread that is currently 0:06:09.618 running. OK, so that's a very high-level 0:06:11.702 picture of what we're going to be doing with threads. 0:06:14.48 And there's going to be a number of different ways, 0:06:17.152 and that's what we're going to look at to this lecture is how 0:06:20.358 we can actually accomplish this switching, how we decide when to 0:06:23.725 switch, and what actually happens when this switch takes 0:06:26.664 place. OK, so we're going to look at 0:06:29.925 several different methods. So the first thing we're going 0:06:33.643 to look at is an approach called cooperative scheduling, 0:06:37.294 or cooperative switching. So in cooperative switching, 0:06:40.813 what happens is that each thread periodically decides that 0:06:44.597 it's done processing and is willing to let some other 0:06:48.049 process run. OK, so the thread calls a 0:06:50.506 routine that says I'm done; please schedule another thread. 0:06:54.356 So, this cooperative approach is simple, and it's easy to 0:06:58.074 think about. In the grand scheme of 0:07:01.321

enforcing modularity of creating this illusion that each program 0:07:05.107 really has its own computer that's running, 0:07:07.63 and where each program gets to run no matter what any other 0:07:11.115 program does, cooperative scheduling has a 0:07:13.579 bit of a problem because if we are doing cooperative 0:07:16.643 scheduling, then one process can just never call this function 0:07:20.309 that says I give up time to the other processes running on the 0:07:23.974 system. One module may just never give 0:07:26.197 up processor. And then we are in trouble 0:07:29.657 because no other module ever gets to run. 0:07:32.208 So instead what we're going to talk about, the alternative to 0:07:36.033 this, is something we call preemptive scheduling. 0:07:39.093 And in preemptive scheduling, what happens is some part of 0:07:42.728 the computer, say, the kernel, 0:07:44.577 periodically decides that it forces the currently running 0:07:48.147 thread to give up the processor, and starts a new thread 0:07:51.654 running. And we'll see how that works, 0:07:54.013 OK? I also want to draw a 0:07:55.543 distinction between whether these threads are running in the 0:07:59.305 same address space or in a different address space, 0:08:02.493 or in a different address space. 0:08:06 OK, so we could, for example, 0:08:08.153 have these two threads, T1 and T2. 0:08:10.692 They might be running as a part of a single, say, 0:08:14.384 application. So, for example, 0:08:16.538 they might be a part of a, say for example, 0:08:19.769 some computer game that we have running. 0:08:22.769 So suppose we have a computer game like, but me just load up 0:08:27.307 my computer here. Suppose we have a computer game 0:08:31 like take your favorite computer game, say, Halo. 0:08:36 And we are running Halo, and Halo is going to consist of 0:08:40.77 some set of threads that are responsible for running this 0:08:45.627 game. So we might have one main 0:08:48.229 thread which, what it does when it runs is to 0:08:52.045 simply update; it repeats in a loop over and 0:08:55.775 over again, wait for a little while, check and see if there's 0:09:00.979 any user input, if there is any user input, update the state of the game, 0:09:04.795 and then, say for example, redraw the display. 0:09:10 And because Halo is a videogame that somebody is staring at and 0:09:13.553 looking at, it needs to update that display at some rate, 0:09:16.762 say, once every 20 times a second in order to create the 0:09:19.914 illusion of sort of natural animation in the game. 0:09:22.722 So this wait step is going to wait a 20th of a second between 0:09:26.16 every step. OK, so this might be one 0:09:28.166 example of a thread. But of course within Halo, 0:09:31.5 there may be multiple other threads that are also running. 0:09:34.788 So there may be a thread that's responsible for looking for 0:09:38.134 input over the network, right, to see if there is 0:09:40.903 additional data that's arrived from the user, 0:09:43.442 and from other users that are remote in updating the state of 0:09:46.903 the game as other information comes in. 0:09:49.096 And there may be a third thread that, say for example, 0:09:52.153 is responsible for doing some cleanup work in the game like 0:09:55.5 reclaiming memory after some monster that's running around 0:09:58.788 the game has died and we no longer need its state, 0:10:01.615 OK? So if we look at just this, 0:10:04.845 say, main thread within Halo, as I said here we can 0:10:08.263 encapsulate the state of this thread by a set of registers, 0:10:12.228 say, R1 through RN, as well as a stack pointer and 0:10:15.578 a program counter. And then, there also will be a 0:10:18.859 stack that represents the currently executing program. 0:10:22.482 So, for example, the stack might, 0:10:24.669 if we just have entered into this procedure, 0:10:27.609 it might just have a return address which is sort of the 0:10:31.369 address that we should jump to when we're done executing this 0:10:35.47 Halo program here. And then finally, 0:10:38.976 we have a program counter that points to this sort of current 0:10:42.421 place where we are executing it. So we might have started off 0:10:45.866 where we are just executing the first instructions here, 0:10:49.023 the Init. instruction. OK, so what we're going to look 0:10:52.181 at now is this case where we have a bunch of threads, 0:10:55.167 say for example, in Halo all running within the 0:10:57.808 same address space. So they're all part of the Halo 0:11:00.679 program. They can all access to global 0:11:03.533 variables of Halo. But we want to have different 0:11:06.018 threads to do different operations that this program may 0:11:08.925 need to take care of. So we want each of these 0:11:11.305 threads to sort of have the illusion that it has its own 0:11:14.212 processor. And then later we'll talk about 0:11:16.38 a set of programs, say, that are running in 0:11:18.601 different address spaces. And we'll talk about what's 0:11:21.35 different about managing two threads, each of which is 0:11:24.152 running in a different address space. 0:11:26.055 OK, so the very simple idea that we're going to introduce in 0:11:29.175 order to look at this situation where we are going to start off 0:11:32.453 by looking at cooperative scheduling in the same address 0:11:35.361 space. We are going to introduce the 0:11:39.205 notion of a routine called yield. 0:11:41.639 OK, so yield is going to be the thing the currently executing 0:11:46.202 thread calls when it's ready to give up the processor and let 0:11:50.766 another thread run. So the thread is going to run 0:11:54.416 for a while and then it's going to call yield, 0:11:57.839 and that's going to allow other threads to sort of do their 0:12:02.25 execution. And this is cooperative because 0:12:06.568 if a program doesn't call yield, then no other program will ever 0:12:11.786 be allowed to run on the system. So the basic idea is that when 0:12:16.923 a program called yield, what it's going to do is 0:12:20.816 exactly what I described just a minute ago. 0:12:24.295 So yield is going to save the state of the current thread, 0:12:29.017 and then it's going to -- 0:12:32 0:12:36 -- schedule the next thread, so pick the next thread to run 0:12:40.296 from all of the available threads that are on the system. 0:12:44.444 And then it's going to dispatch the next thread, 0:12:47.925 so it's going to in particular setup the state for the next 0:12:52.296 thread so that it's ready to run. 0:12:54.666 And it's going to jump into the return address for that thread. 0:12:59.259 OK, so I want to take you guys through a very simple example of 0:13:03.851 a program that has multiple threads in it. 0:13:08 And we're going to have a simple thread scheduler. 0:13:11.822 So, the thread scheduler is the thing that decides which thread 0:13:16.659 to run next. And this thread scheduler is 0:13:19.78 going to keep an array of fixed size, call it num threads, 0:13:24.226 that are all the threads that are currently running on the 0:13:28.673 system. It's going to have an integer 0:13:31.925 that is going to tell us what the next thread to run is. 0:13:35.578 And then for every one of our threads, we're going to have a 0:13:39.496 variable that we call me which is the ID of the currently 0:13:43.214 executing thread. So, this is local to the 0:13:45.937 thread. 0:13:47 0:13:52 OK so let's look at an example of how this might work with this 0:13:56.305 Halo thread that we're talking about here. 0:13:59.152 OK, so suppose we have a Halo thread. 0:14:02 And we'll call this the main thread of execution. 0:14:04.482 And as I said, we want to have a couple of 0:14:06.603 other threads that are also running on the system at the 0:14:09.448 same time. So we might have a networked 0:14:11.413 thread and a cleanup thread. Now in this case, 0:14:13.741 what we are going to say is that the yield procedure is 0:14:16.534 going to do these three steps. And I made these steps a little 0:14:19.689 bit more concrete so that we can actually think about what might 0:14:22.948 be happening on the stack as these things are executing. 0:14:25.793 So, the yield procedure is going to, when it runs, 0:14:28.327 the first thing it's going to do is save the state. 0:14:32 So to save the state, it simply stores in this table 0:14:34.697 the stack pointer of the currently executing program, 0:14:37.448 the stack pointer of the currently executing thread. 0:14:40.146 And then what it's going to do is it's going to take the next 0:14:43.32 thread to run. So in this case, 0:14:44.907 picking the next thread to run, this thread scheduler is doing 0:14:48.081 something extremely simple. It's just cycling through all 0:14:51.044 the available threads one after another and scheduling them, 0:14:54.165 right? So it says next gets next plus 0:14:56.069 one, and then modulo the total number of threads that are in 0:14:59.191 the system. So suppose num threads is equal 0:15:01.412 to five. As soon as next plus one is 0:15:05.131 equal to five, and then five modulo five is 0:15:08.116 zero, and we're going to wrap that back around. 0:15:11.385 So, once we've executed the fourth thread, 0:15:14.299 we are going to wrap around and start executing thread number 0:15:18.563 zero. And then finally what it's 0:15:20.766 going to do is it's going to restore the stack pointer. 0:15:24.604 Here, I'll just change this for you right now. 0:15:27.802 This should be table sub next. I didn't want to do that. 0:15:33 0:15:37 So, we have this table. So, we restore the stack 0:15:39.505 pointer for the next thread that we want to run from the table of 0:15:42.916 stack pointers that are available. 0:15:44.675 And then what's going to happen is that when we exit out of this 0:15:48.034 yield routine, we are going to load the return 0:15:50.432 address from the stack pointer that we just have set up. 0:15:53.364 So you'll see how that works again in a second. 0:15:55.816 But the basic process, then, is just going to be as 0:15:58.481 follows. So this only works on a single 0:16:00.507 processor. And I'm not going to have time 0:16:03.796 to go into too much detail about what happens in a 0:16:06.729 multiprocessor, but that book talks carefully 0:16:09.363 about why this doesn't work on a simple processor machine. 0:16:12.776 OK, so the basic process, then, it's going to be as 0:16:15.769 follows. So we've got our yield 0:16:17.565 procedure. We have our three threads. 0:16:19.72 Say we've numbered them one, two, and three, 0:16:22.294 as shown up here. So, num threads is equal to 0:16:24.929 three, and say we start off with next equal to one. 0:16:29 So what's going to happen is that the first thread at some 0:16:31.862 point it's going to call a yield routine. 0:16:33.871 And what that's going to do is it's going to cause this yield 0:16:36.884 to execute. We're going to increment next. 0:16:38.943 And we're going to schedule the network thread here. 0:16:41.505 OK, and the network thread is going to run for a little while. 0:16:44.568 And then at some point, it's going to call yield, 0:16:46.979 and that's going to sort of cause the next thread to be 0:16:49.691 scheduled. We're going to call cleanup, 0:16:51.599 and then finally we're going to call yield. 0:16:53.709 And the third thread is going to call yield and cause the 0:16:56.521 first thread to run again. And this process is just going 0:16:59.333 to repeat over and over and over again. 0:17:01.242 OK, so -- 0:17:03 0:17:08 OK, so what I want to do now is to look at actually more carefully 0:17:12.728 at what's happening with the, how this scheduling process is 0:17:16.983 actually working. So we can understand a little 0:17:20.608 bit better what's happening to the stack pointer, 0:17:24.391 and how these various yields are actually executing. 0:17:28.41 So let's look at the, so this is going to be an 0:17:31.72 example of the stack on, say, two of these threads, 0:17:35.66 OK? So, suppose we have thread 0:17:40.403 number one. I'll call this thread one, 0:17:45.246 and this is our Halo thread. OK, and this is the stack for 0:17:52.706 Halo. We also have thread number two, 0:17:57.418 which is our network thread. OK, and this is also some stack 0:18:04.251 associated with this thing. So we're just looking at the 0:18:08.244 current state of the stack on these two things, 0:18:11.585 these two threads. And if you remember, 0:18:14.344 typically we represent stacks as going down in memory. 0:18:18.192 So, these are the different addresses of the different 0:18:22.041 entries in the stack that represent what's going on 0:18:25.672 currently in the processor. So we'll say that thread one, 0:18:29.738 perhaps, the stack starts at address 108, and maybe thread 0:18:33.877 two starts at address 208. OK, so suppose we start up this 0:18:39.501 system, and when we start it up, the stack pointer is currently 0:18:44.505 pointing here. So we take a snapshot of this 0:18:47.975 system at a given point in time. Suppose the stack pointer is 0:18:52.817 currently pointing at 104. The currently executing thread 0:18:57.336 is this one, thread one. And we have, 0:19:00.902 remember our thread table that captures the current thread 0:19:05.189 number, and the save stack pointer for each one of these 0:19:09.325 threads. So we've got thread number one, 0:19:12.258 thread number two. OK, so let's suppose we start 0:19:15.792 looking at this point in time where the stack pointer is 0:19:19.928 pointing here. And if we look at this program, 0:19:23.313 so if we start off with the program that we initially had, 0:19:27.599 in order to understand what happens to the stack, 0:19:31.209 we need to actually look at when the stack gets manipulated. 0:19:37 So if we are just looking at C code, typically the C code isn't 0:19:40.607 going to have, we're not going to see the 0:19:42.935 changes to the stack occurring within the C code. 0:19:45.728 So what I've done here is in yellow, I've annotated this with 0:19:49.219 the sort of additional operations, the assembly 0:19:51.896 operations, that are happening on the entry and exit to these 0:19:55.387 procedures. So what happens just before we 0:19:57.773 call the yield procedure is that the Halo thread will push the 0:20:01.323 return address onto the stack. OK, so if we start executing 0:20:05.565 the yield procedure, Halo is going to push the 0:20:08.08 return address onto the stack. And then it's going to jump 0:20:11.267 into the yield procedure. So, suppose we come onto here. 0:20:14.341 Now if we look at what the return address is after we 0:20:17.248 execute the yield procedure, it's going to be instruction 0:20:20.378 number five. So we're going to push that 0:20:22.559 address onto the stack because instruction number five is 0:20:25.689 labeled here on the left side is where we're going to return to 0:20:29.155 after the yield. And then the yield is going to 0:20:32.593 execute. And what's going to happen as 0:20:34.558 the yield executes is it's going to change the stack pointer to 0:20:37.851 be the stack pointer of the next thread, right? 0:20:40.294 So, I have this typo here again. 0:20:41.941 So be careful about that. We're going to change the stack 0:20:44.915 pointer to be the stack pointer of the next thread. 0:20:47.571 And so when it executes this pop RA instruction that we see 0:20:50.652 here, it's actually going to be popping the return address from 0:20:53.945 the network thread that it scheduled next as opposed to the 0:20:57.025 return address from this Halo thread. 0:21:00 OK, so suppose that the saved stack pointer for the 0:21:03.692 network thread was pointing here at 204. 0:21:06.409 So, that would have been an entry, 204, here. 0:21:09.475 And the saved return address that it's going to jump to is 0:21:13.446 going to be here. So suppose the return address 0:21:16.651 that it jumps to is 1,000, OK? 0:21:18.672 So if we look at the state of these two threads as they have 0:21:22.782 been running, we saw thread one ran for a 0:21:25.569 little while, and then it called yield, 0:21:28.217 and the instruction following yield with instruction number 0:21:32.258 five. OK, so that was the address 0:21:35.683 that we pushed on to the stack pointer. 0:21:38.053 Now, we switched over here to thread two. 0:21:40.547 And last time we ran thread two, it called yield, 0:21:43.541 and its return address was 1,000. 0:21:45.536 So, there was this 1,000 here, OK? 0:21:47.594 So now, when we schedule thread two, what's going to happen is 0:21:51.398 that we're going to read the, so this is instruction 1,000. 0:21:55.015 We're going to start executing from 1,000 because we're going 0:21:58.757 to execute this pop return address. 0:22:02 So we're going to pop the return address off the stack, 0:22:05.237 and we're going to start executing here at instruction 0:22:08.415 1,000, OK? So, you guys kind of see how it 0:22:10.873 is that we switch now from one thread to the other, 0:22:13.871 by switching the stack pointer and grabbing our return address 0:22:17.528 from the stack of thread two instead of from thread one. 0:22:20.826 OK, so now what's going to happen is this thread two is 0:22:24.064 going to start executing from this address. 0:22:26.582 And it's going to run for a little while. 0:22:30 And at some point later, it's going to call yield again, 0:22:33.619 right, because it's run for a while and it's decided that it's 0:22:37.634 time to yield. So, suppose that instruction 0:22:40.399 1,024 it calls yield. So when it does that, 0:22:43.163 the same thing is going to happen. 0:22:45.335 It's going to have run for a little while. 0:22:48.033 So its stack is going to have grown. 0:22:50.337 But eventually it's going to push the return address onto the 0:22:54.286 stack, say here 1,025, and suppose this value of its 0:22:57.643 stack pointer now has grown down. 0:23:01 So it's gone 204. It's run for a while, 0:23:03.048 pushed some things on the stack, and maybe the stack 0:23:05.797 pointer is now at 148. So, when it calls yield, 0:23:08.276 it's going to write into the stack pointer address in the 0:23:11.295 thread table 148. And we're going to restore the 0:23:13.828 stack pointer dressed from here, which I forgot to show being 0:23:17.063 written in, but what I should have written in there was 104. 0:23:20.243 So, we're going to restore the stack address to 104. 0:23:22.992 We're going to pop the next instruction to execute off of 0:23:26.011 that stack five, and then we're going to keep 0:23:28.382 executing. So you just switch back and 0:23:34.821 forth in this way. OK, so -- 0:23:40 0:23:47 -- what I want to do now is, so this is the basic process 0:23:50.962 now whereby this yield instruction can be used to 0:23:54.358 switch between the scheduling of these two procedures, 0:23:58.108 right? And this is sort of the core of 0:24:00.726 how it is. So we have these two threads of 0:24:03.462 execution, and they just sort of run through. 0:24:05.539 And they periodically call yield. 0:24:07.049 And that allows another thread to be written, 0:24:09.125 to execute it. But otherwise these two threads 0:24:11.059 have been written in a style where they don't actually have 0:24:13.796 to know anything about the other threads that are running. 0:24:16.486 There could have been two other threads or 200 other threads 0:24:19.27 running on the system at the same time. 0:24:21.063 And this approach that I showed you would have caused all those 0:24:23.989 threads eventually to have been scheduled and to execute 0:24:26.584 properly. Right, but as we said before, 0:24:29.995 requiring these functions to actually call yield periodically 0:24:34.124 has sort of defeated the purpose of our enforcing modularity, 0:24:38.253 one of our goals of enforcing modularity, which is to make it 0:24:42.382 so that no one thread can interfere with the operation of 0:24:46.236 the other thread, or cause that other thread to 0:24:49.402 crash, right, because if the procedure never 0:24:52.361 calls a yield, then a module never calls 0:24:55.045 yield, excuse me, another thread will never be 0:24:58.141 scheduled. And so, that module will have 0:25:01.572 the ability, essentially, to take over the whole 0:25:04.122 computer, which is bad. So what we're going to look at 0:25:06.996 now is how we go from this cooperative scheduling where 0:25:09.925 modules calling yield to preemptive scheduling where modules are 0:25:13.179 forced to yield to the processor periodically. 0:25:16 0:25:32 OK, so this is the case where you have no explicit yield 0:25:37.661 statements. 0:25:39 0:25:44 All right, so the idea here is that it turns out to be very 0:25:47.371 simple. So programs aren't going to 0:25:49.455 have an explicit yield statement in them. 0:25:51.908 But what we're going to do is we're going to have a special 0:25:55.463 timer that runs periodically within, say, for example, 0:25:58.712 the kernel. So suppose the kernel has a 0:26:02.081 timer that runs periodically that causes the kernel to be 0:26:06.399 sort of woken up and allowed to execute some code. 0:26:10.176 So this timer is going to be connected to what's called an 0:26:14.571 interrupt. So, we're going to introduce a 0:26:17.655 timer interrupt, and almost all processors 0:26:20.816 essentially have some notion of a timer interrupt. 0:26:24.594 And an interrupt is something that when it fires, 0:26:28.294 it causes the processor to run a special piece of code, 0:26:32.458 OK? So, basically this is going to 0:26:36.202 be some processor. Think of it, 0:26:38.405 if you like, as some line on the processor. 0:26:41.488 OK, there's a wire that's coming off the processor. 0:26:45.159 And when this wire gets pulled high, when the timer goes off 0:26:49.491 and fires, this line is going to be pulled high. 0:26:52.941 And when that happens, the microprocessor is going to 0:26:56.759 notice that and is going to invoke a special function within 0:27:01.091 the kernel. So this line is going to be 0:27:07.341 checked by the microprocessor before it executes each 0:27:15.703 instruction, OK? And if the line is high, 0:27:22.135 the microprocessor is going to execute one of these special 0:27:31.462 gate functions. So, we saw the notion of a gate 0:27:37.152 function before that can be used for a module to obtain entry 0:27:41.319 into the kernel. Essentially what's going to 0:27:44.305 happen is that when the timer interrupt fires, 0:27:47.43 it's going to go execute one of these special gate functions as 0:27:51.736 well. And that's going to cause the 0:27:54.097 kernel to then be in control of the processor. 0:27:57.222 So I remember when the gate function runs, 0:28:00.069 it switches the user to kernel mode bit, to kernel mode. 0:28:05 It switches the page map address register to the kernel's 0:28:07.91 page map so that the kernel is in control, and it switches the 0:28:11.081 stack pointer to the kernel save stack pointer so that 0:28:13.991 the kernel is in control of the system, and can execute whatever 0:28:17.266 code that wants. So this is going to accomplish 0:28:19.604 basically everything that we need, right? 0:28:21.683 Because if we can get the kernel in control of the system, 0:28:24.646 now, the kernel can do whatever it needs to do to, 0:28:27.193 for example, schedule the next thread. 0:28:30 So the way that's going to work is basically very simple. 0:28:35.433 So when the kernel gets run, it knows which thread, 0:28:40.285 for example, is currently running. 0:28:43.487 And basically what it does is it just calls the appropriate 0:28:49.115 yield function, it calls the yield function for 0:28:53.579 the thread that it's currently running, forcing that thread to 0:28:59.498 yield. So, kernel calls yield on 0:29:02.506 current thread, right? 0:29:06 0:29:11 OK, so in order to do this, of course the kernel needs to 0:29:14.467 know how to capture the state for the currently running 0:29:17.811 thread. But for the most part that's 0:29:19.979 pretty simple because the state is all encapsulated in the 0:29:23.509 current values of the registers in the current value of the 0:29:27.1 stack pointer. So, the kernel is going to call 0:29:29.887 yield on the currently executing thread, and that's going to 0:29:33.541 force that thread to go ahead and schedule another thread. 0:29:38 So the module itself hasn't called yield, 0:29:42.462 but still the module has been forced to sort of give up its 0:29:48.932 control the processor. So, when it calls yield, 0:29:54.063 it's just going to do it we did before, which is save our state 0:30:00.98 and schedule and run the next thread. 0:30:06 OK, so the only last little detail that we have to think 0:30:09.084 about is what if the kernel wants to schedule a thread 0:30:12.056 that's running in a different address space? 0:30:14.467 So if a kernel wants to schedule a thread that's running 0:30:17.551 a different address space, well, it has to do what we 0:30:20.635 saw last time when we saw about how we switch address spaces. 0:30:24 It has to change the value of the PMAR register so that the 0:30:27.252 next address space gets swapped in. 0:30:30 And then it can go ahead and jump into the appropriate 0:30:33.75 location in the sort of newly scheduled address space. 0:30:37.5 So that brings us to the next topic of discussion. 0:30:40.968 So, what we've seen so far now, we saw cooperative in the same 0:30:45.284 address space, and I introduced the notion of 0:30:48.398 a preemptive scheduling. What we want to look at now is 0:30:52.219 sort of what it means to run multiple threads in different 0:30:56.253 address spaces. Typically when we talk about a 0:30:59.437 program that's running on a computer or in 6.033 we like to 0:31:03.541 call programs running on computers processes. 0:31:08 When we talk about process, we need an address space plus 0:31:12.09 some collection of threads. So this is sort of the real 0:31:16.035 definition of what we mean by a process. 0:31:18.738 And alternately, you will see process is called 0:31:22.098 things like applications or programs. 0:31:24.727 But any time you see a word like that, what people typically 0:31:29.037 mean is it's some address space, some virtual address space in 0:31:33.493 which we resolve memory addresses. 0:31:37 And then a collection of threads that are currently 0:31:41.23 executing within that address space, and where each of those 0:31:46.223 threads includes the set of instructions and registers and 0:31:51.046 stack that correspond to it, OK? 0:31:53.669 So the kernel has explicit support for these processes. 0:31:59 0:32:03 So in particular, what the kernel provides are 0:32:06.295 routines to create a process and destroy a process. 0:32:09.957 OK, and these create and destroy methods are going to 0:32:13.766 sort of do, the create method is going to do all the 0:32:17.501 initialization that we need to do in order to create a new 0:32:21.676 address space and to create at least one thread that is running 0:32:26.216 within that address space. So we've sort of seeing all the 0:32:31.431 pieces of this, but basically what it's going 0:32:34.882 to do is it's going to allocate the address space in its table 0:32:39.666 of all the address spaces that it knows about. 0:32:43.196 So we saw that in the last time, and it's going to allocate 0:32:47.745 a piece of physical memory that corresponds to that address 0:32:52.294 space in particular. So, it's going to allocate a 0:32:54.96 piece of physical memory that corresponds to that address 0:32:59.352 space. It's going to allocate one of 0:33:04.247 these page maps that the PMAR register is going to point to 0:33:10.741 when this thing is running. OK, and now the other thing 0:33:16.787 that's going to do is to go ahead and load the code for this 0:33:23.393 thing into memory and map it into the address space. 0:33:30 So it's going to add the code for this currently running 0:33:33.898 module into the address space. And then it's going to create a 0:33:38.009 thread for this new process. And it's going to add it to the 0:33:42.191 table, the thread table, and then it's going to set up 0:33:45.948 the value of the stack pointer and the program counter for this 0:33:50.342 process so that when the process starts running it, 0:33:53.886 you sort of into the process at some well-defined entry point, 0:33:58.21 say the main routine in that process so that the thread can 0:34:02.321 start executing at whatever starting point it has. 0:34:07 OK, and now destroy is just going to basically do the 0:34:11.976 opposite. It's going to get rid of all 0:34:15.518 this state that we created. So it's going to remove the 0:34:20.686 address space, and it's going to remove the 0:34:24.706 thread from the table. OK, and it may also reclaim any 0:34:29.778 memory that's associated exclusively with this process. 0:34:36 OK so if we look at -- 0:34:39 0:34:49 So if you look at a computer system at any one point in time, 0:34:53.663 what you'll see is a collection of processes. 0:34:57.082 So I've drawn these here as these big boxes. 0:35:00.424 So you might have a process that corresponds to Halo and, 0:35:04.777 say, we are also editing our code. 0:35:07.341 So, we have a process that corresponds to EMACS. 0:35:11.15 OK, and each one of these processes is going to have an 0:35:15.347 address space associated with it. 0:35:19 And then there are going to be some set of threads that are 0:35:22.107 running in association with this process. 0:35:24.25 So in the case of Halo, I'm going to draw these threads 0:35:27.142 as these little squiggly lines. So in the case of Halo, 0:35:30.035 we saw that maybe it has three threads that are running within 0:35:33.303 it. So these are three threads that 0:35:35.899 all run within the same address space. 0:35:38.166 Now, EMACS might have just one thread that's running in it, 0:35:41.658 say, although that's probably not true. 0:35:43.986 EMACS is horribly complicated, and probably has many threads 0:35:47.539 that are running within it. And so, we're going to have 0:35:50.847 these sort of two processes with these two collections of 0:35:54.278 threads. So if you think about the sort 0:35:56.606 of modularity or the enforcement of modularity that we have 0:36:00.16 between these processes, we could say some interesting

0:36:03.407 things. So first of all, 0:36:06.243 we've enforced modularity. We have hard enforced 0:36:10.131 modularity between these two processes, right, 0:36:13.63 because there's no way that the Halo process can muck with any 0:36:18.372 of the memory that, say, the EMACS process has 0:36:21.793 executed. And as long as we're using 0:36:24.514 preemptive scheduling, there's no way that the Halo 0:36:28.402 process could completely maintain control of the 0:36:32.056 processor. So the EMACS process is going to 0:36:35.561 be allowed to run, and its memory is going to be 0:36:37.929 protected from the Halo process. OK, now within the Halo 0:36:40.7 process, there is sort of a different story. 0:36:42.867 So within the Halo process, there is sort of a different 0:36:45.638 story. So within the Halo process, 0:36:47.3 these threads, because they are all within the 0:36:49.567 same address space, can muck with each other's 0:36:51.835 memory. So they are not isolated from 0:36:53.648 each other in that way. And typically, 0:36:55.512 because these are all within one process and that if we were 0:36:58.485 to destroy that process in this step, what the operating system 0:37:01.609 would do is in addition to destroying the address space, 0:37:04.38 all of the running threads. So we say that these threads 0:37:09.14 that are running within Halo share a fate. 0:37:12.121 If one of them dies, or if one of these threads 0:37:15.636 fails or crashes or does something bad, 0:37:18.541 then they are essentially all going to crash or do something 0:37:23.05 bad. If the operating system kills 0:37:25.573 the process, then all of these threads are going to go away. 0:37:31 0:37:36 So this is the basic picture of sort of what's going on within a 0:37:39.761 computer system. If you stare at this for a 0:37:42.268 little bit, you see that there is actually sort of a hierarchy 0:37:45.91 of threads that are running on any one computer system at any 0:37:49.492 one point in time. So we have these larger sort 0:37:52.298 of processes that are running. And then within these 0:37:55.343 processes, we have some set of threads that's also running, 0:37:58.805 right? And so, there is a diagram that 0:38:02.384 sort of useful to help us understand this hierarchy or 0:38:06.461 layering of processes or threads on a computer system. 0:38:10.538 I want to show that to you. 0:38:13 0:38:32 OK, so on our computer system, if we look at the lowest layer, 0:38:37.53 we have our microprocessor down here. 0:38:40.794 OK, and this microprocessor, what we've seen is that the 0:38:45.781 microprocessor has two things that it can be doing. 0:38:50.314 Either the microprocessor can be executing, 0:38:54.122 say, some user program, or the microprocessor can be 0:38:58.746 interrupted and go execute one of these interrupt handler 0:39:03.824 functions. So these interrupts are going 0:39:07.571 to do things like, is going to be this timer 0:39:09.901 interrupt, or when some IO device, say for example the 0:39:12.773 disk, finishes performing some IO operation like reading a 0:39:15.862 block from memory, then the processor will 0:39:18.083 receive an interrupt, and the processor can do 0:39:20.684 whatever it needs to do to service that piece of hardware 0:39:23.719 so that the hardware can go and, for example, 0:39:26.103 read the next block. So in a sense, 0:39:28.966 you can think of the processor itself as having to threads that 0:39:33.246 are associated with it. One of them is an interrupt 0:39:36.697 thread, and one of them is the main thread now running on top 0:39:40.838 of this. So, these are threads that are 0:39:43.461 really running within the kernel. 0:39:45.67 But on top of these things there is this set of 0:39:48.846 applications like Halo and EMACS. And, these are the kind of user 0:39:53.194 programs that are all running on the system. 0:39:56.162 And there may be a whole bunch of these. 0:39:58.854 It's not just two, but it's any number of threads 0:40:02.168 that we can multiplex on top of this main thread. 0:40:07 And then each one of these may, in turn, have sub-threads that 0:40:10.884 are running as a part of it. So if you think about what's 0:40:14.45 going on in this system, you can see that at any one 0:40:17.698 level, these two threads don't really need to know anything 0:40:21.391 about the other threads that are running at that same level. 0:40:25.148 So, for example, within Halo, 0:40:26.931 these individual sub-threads don't really know anything about 0:40:30.752 what the other sub-threads are doing. 0:40:34 But it is the case that the threads at a lower level need to 0:40:39.014 know about the threads that are running above them because these 0:40:44.368 threads at the lower level implement this thread scheduling 0:40:49.297 policy, right? So, the Halo program decides 0:40:52.866 which of its sub-threads to run next. 0:40:55.926 So the Halo program in particular is, 0:40:58.985 so the parent thread implements a scheduling -- 0:41:04 0:41:08 -- policy for all of its children threads. 0:41:10.887 So Halo has some policy for deciding what thread to run 0:41:14.69 next. The operating system has some. 0:41:17.154 The kernel has some policy for deciding which of these user 0:41:21.239 level threads to run next. And the parent thread also 0:41:24.901 provides some switching mechanism. 0:41:28 So we studied two switching mechanisms today. 0:41:30.788 We looked at this notion of having this sort of cooperative 0:41:34.464 switching where threads call yield in order to allow the next 0:41:38.267 thread to run. And, we looked at this notion 0:41:40.992 of preemptive scheduling that forces the next thread in the 0:41:44.669 schedule to run. So, if you look at computer 0:41:47.394 systems, in fact it's a fairly common organization to see that 0:41:51.26 there is preemptive scheduling at the kernel level that causes 0:41:55.126 different user level applications to run. 0:41:57.661 But within a particular user program, that program may use 0:42:01.274 cooperative scheduling. So the individual threads of 0:42:05.428 Halo may hand off control to the next thread only when they want 0:42:08.642 to. And this sort of makes sense 0:42:10.224 because if I'm a developer of a program, I may very well trust 0:42:13.336 that the other threads that I have running in the program at 0:42:16.346 the same time, I know I want them to run. 0:42:18.387 So, we can talk about sort of different switching mechanisms 0:42:21.397 at different levels of this hierarchy. 0:42:24 0:42:30 OK, so what we've seen up to this point is this sort of basic 0:42:34.124 mechanism that we have for setting up a set of threads that 0:42:38.11 are running on a computer system. 0:42:40.31 And what we haven't really talked at all about is if we can 0:42:44.09 actually share information between these threads or 0:42:47.527 coordinate access to some shared resource between these threads. 0:42:51.858 So what I want to do with the rest of the time today is to 0:42:55.776 talk about this notion of coordinating access between 0:42:59.35 threads. And we're going to talk about 0:43:05.284 this in the context of a slightly different example. 0:43:12.818 So -- 0:43:14 0:43:22 -- let's suppose that I am building a Web server. 0:43:25.698 And I decide that I want to structure my Web server as 0:43:29.781 follows: I want to have some network thread, 0:43:33.094 and I want to have some thread that services disk requests. 0:43:37.717 OK, so the network thread is going to do things like accept 0:43:42.186

incoming connections from the clients and process those 0:43:46.346 connections parse the HTTP requests and generate the HTML 0:43:50.661 results that we send back to users. 0:43:53.28 And the disk request thread is going to be in charge of doing 0:43:57.903 these expensive operations where it goes out to disk and reads in 0:44:02.834 some data that it may need to assemble these HTML pages that 0:44:07.38 we generate. So, for next recitation, 0:44:11.762 you're going to read about a system called Flash, 0:44:15.288 which is a multithreaded Web server. 0:44:17.858 And so this should be a little taste of what you're going to 0:44:22.192 see for tomorrow. So, these two threads are 0:44:25.276 likely to want to communicate with each other through some 0:44:29.463 data structure, some queue of requests, 0:44:32.254 or a queue of outstanding information. 0:44:36 So suppose that what we're looking at in fact in particular 0:44:40.724 is a queue of disk blocks that are being sent from this is a 0:44:45.53 disk request thread out to the network thread. 0:44:49.196 So, whenever the disk finishes reading a block, 0:44:52.943 it enqueues a value into the network thread so that the 0:44:57.587 network thread can then later pull that value off and deliver 0:45:02.474 it out to the user. So in this case, 0:45:05.941 these are two threads that are both running within the same 0:45:09.352 address space within the same Web server. 0:45:11.705 So they both have direct access to this queue data structure. 0:45:15.235 They can both read and write to it at the same time. 0:45:18.235 This queue data structure, think of it as a global 0:45:21.117 variable. It's in the memory that's 0:45:23.117 mapped in the address space so both threads can access it. 0:45:26.47 So let's look at what happens when these two threads, 0:45:29.529 let's look at some pseudocode that shows what might be 0:45:32.647 happening inside of these two threads. 0:45:36 So, suppose within this first thread here, this network 0:45:42.75 thread, we have a loop that says while true, do the following. 0:45:50.375 De-queue a request, call it M from the thread, 0:45:56 and then process, de-queue a disk block that's in 0:46:02 the queue and then go ahead and process that disk queue. 0:46:10 Go ahead and process that. And now, within the disk 0:46:15.105 request thread, we also have a while loop that 0:46:19.7 just loops forever. And what this does is it gets 0:46:24.602 the next disk block to send, however it does that, 0:46:29.605 goes off and reads the block from disk, and then enqueues 0:46:35.323 that block onto the queue. So if you like, 0:46:40.167 you can think of this as simply the disk request thread is going 0:46:45.072 to stick some blocks into here, and then the network thread is 0:46:49.822 going to sort of pull those blocks off the top of the queue. 0:46:54.416 So if you think about this process running, 0:46:57.686 there's kind of a problem with the way that I've written this, 0:47:02.435 right, which is that suppose that the disk request thread 0:47:06.795 runs much faster than the network thread. 0:47:11 Suppose that for every one network, one block that the 0:47:13.871 network thread is able to pull off and process, 0:47:16.363 the disk thread can enqueue two blocks. 0:47:18.422 OK, so if you think about this for awhile, if you run this 0:47:21.51 system for a long time, eventually you're going to have 0:47:24.435 enqueued a huge amount of stuff. And typically the way queues 0:47:27.686 are implemented is they are sort of some fixed size. 0:47:30.449 You don't want them to grow to fill the whole memory of the 0:47:33.591 processor. So you limit them to some 0:47:36.522 particular fixed size. And eventually we are going to 0:47:39.688 have the problem that the queue is going to fill up. 0:47:42.793 It's going to overflow, and we're going to have a disk 0:47:46.02 block that we don't have anything that we could do with. 0:47:49.369 We don't know what to do with it. 0:47:51.317 Right, so OK, you say that's easy. 0:47:53.326 There is an easy way to fix this. 0:47:55.274 Why don't we just wait until there is some extra space here. 0:47:58.867 So we'll introduce a while full statement here that just sort of 0:48:02.702 sits in a spin loop and waits until this full condition is not 0:48:06.416 true. OK, so as soon as the full 0:48:09.472 condition is not true, we can go ahead and enqueue the 0:48:12.474 next thing on the queue. And similarly we're going to 0:48:15.42 need to do something over here on the process side because we 0:48:18.819 can't really de-queue a message if the queue is empty. 0:48:21.821 So if the processing thread is running faster than the 0:48:24.823 enqueueing thread, we're going to be in trouble. 0:48:27.825 So we're going to also need to introduce a while loop here that 0:48:31.337 says something like while empty. OK, so this is fine. 0:48:35.238 It seems like it fixes our problem. 0:48:37.342 But there is a little bit of a limitation to this approach, 0:48:40.933 which is that now what you see is suppose these two threads are 0:48:44.771 running. And they are being sort of 0:48:46.876 scheduled in round robin; they are being scheduled one 0:48:50.157 after the other. Now this thread runs. 0:48:52.447 And when it runs, suppose that the queue is 0:48:55.047 empty. Suppose the producer hasn't put 0:48:57.338 anything on the queue yet. Now when this guy runs, 0:49:01.435 he's going to sit here, and for the whole time that 0:49:04.555 it's scheduled, it's just going to check this 0:49:07.301 while loop to see if the thing is empty over, 0:49:10.047 and over, and over, and over, and over again, 0:49:12.792 right? so that's all whole lot of 0:49:14.789 wasted time that the processor could and should have been doing 0:49:18.659 something else useful like perhaps letting the producer run 0:49:22.278 so that it could enqueue some data. 0:49:24.4 So in order to do this, so in order to fix this 0:49:27.27 problem, we introduce this notion of sequence coordination 0:49:30.828 operators. In what sequence coordination 0:49:33.261 operators do is they'll allow a thread to declare that it wants 0:49:37.13 to wait until some condition is true. 0:49:41 And they allow other threads to signal that that condition has 0:49:45.591 now become true, and sort of allow threads that 0:49:49.053 are waiting for that condition to become true to run. 0:49:52.967 So, we have these two operations: wait on some 0:49:56.354 variable until some condition is true, and signal on that 0:50:00.569 variable. OK, so we're basically out of 0:50:03.473 time now. So what I want to do is I'll 0:50:05.353 come back and I'll finish going through this example for the 0:50:08.352 next time. But what you should do is think 0:50:10.435 about, suppose you had these sequence coordination operators. 0:50:13.484 How could we sort of modify these two while loops that we 0:50:16.33 have for these two operators in order to be able to take 0:50:19.125 advantage of the fact that in order to be able to make this so 0:50:22.225 we don't sit in this spin loop forever and execute. 0:50:24.765 So that's it. Today we saw how to get these 0:50:26.9 multiple processes to run on one computer. 0:50:30 And next time we'll talk about sort of making computer programs 0:50:32.965 run more efficiently, getting good performance out of 0:50:35.579 this sort of architecture we've sketched.