0:00:00 Let's go ahead and get started. 0:00:02 0:00:07 OK, so today we have one topic to finish up very briefly from 0:00:12.142 last time. So if you remember, 0:00:14.628 when we finished off last time, we were talking about the 0:00:19.428 example of a multithreaded Web server. 0:00:22.6 So the example that we were talking about, 0:00:26.114 and this is an example that I'm going to use throughout the 0:00:31.085 lecture today consisted of a Web server with, this example 0:00:35.971 consists of a Web server with three main modules or main 0:00:40.685 components. So it consists, 0:00:44.666 the three modules are a networking module, 0:00:49.222 a Web server module -- 0:00:52 0:00:58 -- which is in charge of generating, for example, 0:01:01.115 HTML pages, and then a disk module which is in charge of 0:01:04.685 reading data off a disk. OK, so this thing is going to 0:01:08.125 be communicating with the disk, which I've drawn as a cylinder 0:01:12.085 here. So what happens is the client 0:01:14.292 requests come in to this Web server. 0:01:16.563 They come in to the network module. 0:01:18.77 The network module forwards those requests on to the Web 0:01:22.34 server. The Web server is in charge of 0:01:24.742 generating, say, the HTML page that corresponds 0:01:27.728 to the request. And in order to do that, 0:01:31.281 it may need to read some data off of the disk. 0:01:33.684 So it forwards this request onto the disk module, 0:01:36.247 which goes and actually gets the page from the disk and at 0:01:39.291 some point later, the disk returns the page to 0:01:41.694 the Web server. The Web server returns the page 0:01:44.15 to the network module, and then the network module 0:01:46.766 sends the answer back over the network to the user. 0:01:49.436 So this is a very simple example of a Web server. 0:01:52 It should be sort of familiar to you since you have just spent 0:01:55.257 a while studying the Flash Web server. 0:01:57.233 So you can see that this is sort of a simplified description 0:02:00.383 of what a Web server does. So if you think about how you would 0:02:04.823 actually go about designing a Web server like this, 0:02:07.968 of course it's not the case that there is only one request 0:02:11.553 that is moving between these modules at any one point in 0:02:15.012 time. So, in fact, 0:02:16.081 there may be multiple client requests that come into the 0:02:19.54 network module. And the network module may want 0:02:22.433 to have multiple outstanding pages that it's asking the Web 0:02:26.081 server to generate. And the Web server itself might 0:02:29.226 be requesting multiple items from the disk. 0:02:33 And so, in turn, that means that at any point in 0:02:35.647 time there could be sort of multiple results. 0:02:38.126 There could be results streaming back in from the disk 0:02:41.112 which are going into the Web server, which is sort of chewing 0:02:44.492 on results and producing pages to the network module. 0:02:47.422 And so it's possible for there to be sort of queues that are 0:02:50.746 building up between these modules both on the send and 0:02:53.732 receive. So, I'm going to draw a queue, 0:02:55.873 I'll draw a queue just sort as a box with these vertical arrows 0:02:59.366 through it. So there is some buffering 0:03:02.406 that's happening between the sort of incoming requests and 0:03:05.49 the outgoing requests on these modules. 0:03:07.546 OK, and this buffering is a good thing. 0:03:09.602 And we're going to talk more about this throughout the 0:03:12.47 lecture today because what it allows us to do is it allows us 0:03:15.716 to decouple the operations of these different modules. 0:03:18.583 So, for example, the disk module can be reading 0:03:21.072 a page from disk while the HTML page, while the HTML server is, 0:03:24.427 for example simultaneously generating an HTML page that 0:03:27.348 wants to return to the client. But in this architecture, 0:03:31.544 you can see that, for example, 0:03:33.335 when the Web server wants to produce a result, 0:03:36.115 it can only produce a result when the disk pages that it 0:03:39.512 needs are actually available. So the Web server is dependent 0:03:43.156 on some result from the disk module being available. 0:03:46.307 So if we were to look at just this Web server, 0:03:49.086 I'm going to call this the HTML thread here and the disk thread, 0:03:52.978 so these two threads that are on the right side of this 0:03:56.313 diagram that I've drawn here, here to look at the code that 0:03:59.896 was running in these things, and we saw this last time, 0:04:03.232 the code might look something like this. 0:04:07 So the HTML thread is just going to sit in a loop 0:04:09.395 continually trying to de-queue information from this queue that 0:04:12.489 is shared between it and the disk thread. 0:04:14.485 And then, the disk thread is going to be in a loop where it 0:04:17.379 continually reads blocks off the disk, and then enqueues them 0:04:20.373 onto this queue. So this design at first seems 0:04:22.618 like it might be fine. But then if you start thinking 0:04:25.213 about what's really going on here, there could be a problem. 0:04:28.157 So, suppose for example that the queue is of a finite length. 0:04:32 It only has a certain number of elements in it. 0:04:34.353 Now when we keep calling in queue over and over and over 0:04:37.167 again, it's possible that if the HTML thread isn't consuming 0:04:40.186 these pages off the queue fast enough, that the queue could 0:04:43.153 fill up, and it could overflow, right? 0:04:45.046 So that might be a problem that we would want to sort of make a 0:04:47.911 condition that we would explicitly check for in the 0:04:50.469 code. And so we could do that by 0:04:52.055 adding a set of conditions like this. 0:04:53.897 So what you see here is that I have just augmented the code 0:04:56.865 with these two additional variables, used and free, 0:04:59.423 where used indicates the number blocks that are in the queue 0:05:02.441 that are currently in use. And free indicates the number 0:05:06.798 of blocks that are in the code, the number of blocks that are 0:05:10.519 in the queue that are currently free. 0:05:12.751 So what this loop does is that the disk thread says it only 0:05:16.348 wants to enqueue something onto the queue when there are some 0:05:20.069 free blocks. So, it has a while loop that 0:05:22.55 just loops forever and ever and ever while they're waiting when 0:05:26.395 there are no free blocks, OK? 0:05:29 And similarly, the HTML thread is just going 0:05:31.377 to wait forever when there are no used blocks, 0:05:33.865 OK? And then, when the disk thread 0:05:35.689 enqueues a block onto the queue, it's going to decrement the 0:05:38.951 free count because it's reduced the number of things that are in 0:05:42.434 the queue. And it's going to increment the 0:05:44.7 used count because now there is one additional thing that's 0:05:47.907 available in the queue, OK? 0:05:49.344 So this is a simple way in which now we've made it so these 0:05:52.551 things are waiting for each other. 0:05:54.375 They are coordinating with each other by use of these two shared 0:05:57.858 variables used in free. OK. so these two threads share 0:06:00.788 these variables. So that's fine. 0:06:04.144 But if you think about this from a scheduling point of view, 0:06:08.644 there still is a little bit of a problem with this approach. 0:06:13.144 So in particular, what's going on here is that, 0:06:16.652 oops, when one of these threads enters into one of these while 0:06:21.305 loops, it's just going to sit there checking this condition 0:06:25.728 over and over and over and over again, right? 0:06:30 So then the thread scheduler schedules that thread. 0:06:32.401 It's going to repeatedly check this condition. 0:06:34.563 And that's maybe not so desirable. 0:06:36.148 So suppose, for example, that the HTML thread enters 0:06:38.598 into this loop and starts looping because there's no data 0:06:41.288 available. Now, what really we would like 0:06:43.209 to have happen is for the disk thread to be allowed to get a 0:06:46.043 chance to run, so maybe it can produce some 0:06:48.061 data so that the HTML thread can then go ahead and operate. 0:06:50.847 But, with this while loop there, we can't quite do that. 0:06:53.489 We just sort of waste the CPU during the time we are in this 0:06:56.323 while loop. So, instead what we are going 0:06:58.244 to do is introduce the set of what we call sequence 0:07:00.646 coordination operators. 0:07:03 0:07:09 So in order to introduce this, we're going to add something, 0:07:13.829 a new kind of data type that we call an event count. 0:07:18.003 An event count, you can just think of it as an 0:07:21.686 integer that indicates the number of times that something 0:07:26.27 has occurred. It's just some sort of running 0:07:29.79 counter-variable. And we're going to introduce 0:07:35.162 two new routines. So these two routines are 0:07:39.906 called wait and notify. OK, so wait takes two 0:07:44.876 arguments. It takes one of these event 0:07:49.055 count variables. And it takes a value. 0:07:53.234 OK, so what wait says is check the value of this event count 0:07:59.898 thing, and see whether or when we check it, the value of this 0:08:06.674 event count is less than or equal to value. 0:08:13 If the event count is less than or equal to value, 0:08:16.988 then it waits. And what it means for it to 0:08:20.325 wait is that it tells the thread scheduler that it no longer 0:08:25.127 wants to be scheduled until somebody later calls this notify 0:08:29.93 routine on this same event count variable. 0:08:34 OK, so wait says wait, if this condition is true, 0:08:37.412 and then notify says, wake up everybody who's waiting 0:08:41.109 on this variable. So we can use these routines in 0:08:44.521 the following way in this code. And it's really very 0:08:48.146 straightforward. We simply change our iteration 0:08:51.417 through, our while loops into wait statements. 0:08:54.616 So what we're going to do is we're going to have the HTML 0:08:58.597 thread wait until the value of used becomes greater than zero. 0:09:04 And we're going to have our disk thread wait until the value 0:09:07.793 of free becomes greater than zero. 0:09:09.915 And then the only other thing that we have to add to this is 0:09:13.709 simply a call to notify. So what notify does is it 0:09:16.86 indicates to any other thread that is waiting on a particular 0:09:20.718 variable that that thread can run. 0:09:22.84 So the HTML thread will notify free, which will tell the 0:09:26.569 disk thread that it can now begin running if it had been 0:09:30.106 waiting on the variable free. OK, so this emulates the 0:09:34.576 behavior of the while loop that we had before except that the 0:09:38.518 thread scheduler, rather than sitting in any 0:09:41.343 infinite while loop simply doesn't schedule the HTML thread 0:09:45.153 of the disk thread while it's waiting in one of these wait 0:09:48.897 statements. OK, so what we're going to talk 0:09:51.656 about for the rest of the lecture today is related to 0:09:55.072 this, and I think you will see why as we get through the talk. 0:10:00 The topic for today is performance. 0:10:02.373 So performance, what we've looked at so far in 0:10:05.515 this class are these various ways of structuring complex 0:10:09.355 programs, how to break them up into several modules, 0:10:12.915 the client/server paradigm, how threads work, 0:10:15.987 how a thread scheduler works, all of these sort of big topics 0:10:20.176 about how you design a system. But we haven't said anything 0:10:24.225 about how you take a system design and in an ordered, 0:10:27.856 regular, systematic way, think about making that system 0:10:31.626 run efficiently. So that's what we're going to 0:10:35.714 try and get at today. We're going to look at a set of 0:10:38.787 techniques that we can use to make a computer system more 0:10:42.097 efficient. And so, these techniques, 0:10:44.166 there are really three techniques that we're going to 0:10:47.239 look at today. The first one is a technique 0:10:49.722 called concurrency. And concurrency is really about 0:10:52.677 allowing the system to perform multiple operations 0:10:55.573 simultaneously. So, for example, 0:10:57.405 in our sample Web server, it may be the case that we have 0:11:00.715 this disc that we can sort of read pages from at the same time 0:11:04.321 that, for example, the CPU generates some Web 0:11:06.921 pages that it's going to output to the client. 0:11:11 OK, so that's what concurrency is about. 0:11:12.867 We are also going to look at a technique called caching, 0:11:15.5 which you guys should have all seen before. 0:11:17.51 Caching is really just about saving off some previous work, 0:11:20.287 some previous computation that we've already done, 0:11:22.632 or our previous disk page that we've already read in. 0:11:25.122 We want to save it off so that we can reuse it again at a later 0:11:28.09 time. And then finally, 0:11:29.954 we are going to look at something called scheduling. 0:11:32.819 So scheduling is about when we have multiple requests to 0:11:35.907 process, we might be able to order those requests in a 0:11:38.884 certain way or group the requests together in a certain 0:11:41.917 way so that we can make the system more efficient. 0:11:44.669 So it's really about sort of choosing the order in which we 0:11:47.926 do things in order to make the system run more efficiently. 0:11:51.184 And throughout the course of this, I'm going to use this 0:11:54.273 example of this Web server that we've been talking about to sort 0:11:57.811 of motivate each of the applications, 0:11:59.833 or each of these performance techniques that we're going to 0:12:03.09 talk about. So in order to get to the point 0:12:07.118 where we can understand how these performance techniques 0:12:10.878 work, we need to talk a little bit about what we mean by 0:12:14.637 performance. How do we measure the 0:12:16.893 performance of the system, and how do we understand where 0:12:20.721 the bottlenecks in performance in a system might be? 0:12:24.207 So one thing we might want to, the first thing we need to do 0:12:28.24 is to define a set of performance metrics. 0:12:32 These are just a set of terms and definitions that we can use 0:12:36.901 so that we can talk about what the performance of a system 0:12:41.722 is. So the first metric we might be 0:12:44.5 interested in is the capacity of a system. 0:12:48.013 And capacity is simply some measure of the amount of 0:12:52.179 resource in a system. So this sounds kind of 0:12:55.692 abstract, but what we mean by a resource is some sort of thing 0:13:00.676 that we can compete with. It's a disk, 0:13:03.699 or a CPU, or a network, so we might, 0:13:06.558 for example, talk about the capacity of a 0:13:09.826 disk might be the size in gigabytes or the capacity of a 0:13:14.32 processor might be the number of instructions it can execute per 0:13:19.467 second. OK, so once we have capacity, 0:13:24.546 now we can start talking about how much of the system we are 0:13:29.726 actually using. So we talk about the 0:13:32.799 utilization. So utilization is simply the 0:13:36.312 percentage of capacity we're using. 0:13:40 So we might have used up 80% of the disk blocks on our computer. 0:13:45.542 So now there are two sort of properties, or two metrics that 0:13:50.733 are very commonly used in computer systems in order to 0:13:55.395 classify or sort of talk about what the performance of the 0:14:00.41 system is. So, the first metric is 0:14:03.313 latency. So, latency is simply the time 0:14:06.656 for a request to complete. The REQ is request, 0:14:12.082 OK, and we can also talk about sort of the inverse of this, 0:14:18.438 what at first will seem like the inverse of this, 0:14:23.698 which is throughput. That's simply the number of 0:14:28.849 requests per second that we can process. 0:14:34 So when you think about latency and throughput, 0:14:36.316 when you first see this definition, it's tempting to 0:14:38.884 think that simply throughput is the inverse of latency, 0:14:41.604 right? If it takes 10 ms for a request 0:14:43.467 to complete, well, then I must be able to complete 0:14:45.935 100 requests per second, right? 0:14:48 And, that's true in the simple case where in the very simple 0:14:53.198 example where I have a single module, for example, 0:14:57.515 that can process one request at a time, so a single 0:15:01.92 computational resource, for example, 0:15:05.004 that can only do one thing at a time, if this thing has some 0:15:10.202 infinite set of inputs in it, it takes 10 ms to process each 0:15:15.4 input, we'll see, say, 100 results per second 0:15:19.277 coming out, OK? So if something takes 10 ms to 0:15:23.242 do, you can be 100 of them per second. 0:15:28 So we could say the throughput of this system is 100 per 0:15:31.051 second, and the latency is 10 ms. 0:15:32.827 What we're going to see throughout this talk is that in 0:15:35.823 fact a strict relationship between latency and throughput 0:15:38.931 doesn't hold I mean, you guys probably have already 0:15:41.705 seen the notion of pipelining before in 6.004, 0:15:44.202 and you understand that pipelining is a way in which we 0:15:47.199 can improve the throughput of the system without necessarily 0:15:50.472 changing the latency. And we'll talk about that more 0:15:53.302 carefully as this talk goes on. OK, so given these metrics, 0:15:56.521 now what we need to do is think a little bit about, 0:15:59.295 OK, so suppose I have some system, and suppose I have some 0:16:02.458 sort of set of goals for that system like I want the system to 0:16:05.843 be able to process a certain number of requests per second, 0:16:09.061 or I want the latency of this system to be under some amount. 0:16:14 So then the question is, so you are given this computer 0:16:18.879 system and you sit down and you want to measure it. 0:16:23.397 And so you're going to measure the system. 0:16:27.102 And what do you expect to find? So, in the design of computer 0:16:32.524 systems, it turns out that there is some sort of well-known 0:16:37.765 performance pitfalls, or so-called performance 0:16:41.831 bottlenecks. And the goal of sort of doing 0:16:45.446 performance analysis of a system is to look at the system and 0:16:48.44 figure out where the bottlenecks are. 0:16:50.237 So, this typically in the design of the big computer 0:16:52.781 system, what we're worried about is which of the little 0:16:55.476 individual modules within the system is most responsible for 0:16:58.419 slowing down my computer. And what should I do in order 0:17:01.114 to, sort of, and then once you've identified that module, 0:17:03.908 picking about how to make a particular module that slow run 0:17:06.802 faster. So that's really what finding 0:17:10.263 performance bottlenecks is about. 0:17:12.76 And there's a classic bottleneck that occurs in 0:17:16.351 computer systems that you guys all need to know about. 0:17:20.487 It's this so-called IO bottleneck. 0:17:23.063 OK, so what the IO bottleneck says is really fairly 0:17:27.121 straightforward. If you think about a computer 0:17:30.634 system, it has a hierarchy of memory devices in it, 0:17:34.536 OK? And these memory devices start, 0:17:37.19 or storage devices. So these storage devices first 0:17:41.932 for start with the CPU. So the CPU has some set of 0:17:45.089 registers on it, a small number of them, 0:17:47.602 say for example, 32. 0:17:48.826 And you can access those registers very, 0:17:51.338 very fast, say once per instruction, once per cycle on 0:17:54.753 the computer. So, for example, 0:17:56.621 if your CPU is one gigahertz, you may be able to access one 0:18:00.294 of these registers in 1 ns. OK, and so typically at the 0:18:05.296 tallest level, it is pure med, 0:18:07.762 we have a small storage that is fast, OK? 0:18:11.164 As we go down this pyramid adding new layers, 0:18:14.906 and looking at this storage hierarchy, we're going to see 0:18:19.668 that things get bigger and slower. 0:18:22.475 So, just below the CPU, we may have some processor 0:18:26.642 cache, OK, and this might be, for example, 0:18:30.129 512 kB. And it might take 20 ns to 0:18:33.625 access a single, say, block of this memory. 0:18:36.25 And then we're going to have the ram, the main memory of the 0:18:39.937 device, which on a modern machine might be 1 GB. 0:18:42.875 And it might take 100 ns to access. 0:18:45 And then below that, you take a big step down or big 0:18:48.187 step up in size and big step down and performance. 0:18:51.25 You typically have a disk. So a disk might be as big as 0:18:54.625 100 GB, right? But, performance is very slow. 0:18:57.375 So it's a mechanical thing that has to spin, 0:19:00.125 and it only spins so fast. So a typical access time for a 0:19:04.656 block of the disk might be as high as 10 ms or even higher. 0:19:07.858 And then sometimes people will talk in this hierarchy the 0:19:10.95 network is actually a level below that. 0:19:13.049 So if something isn't available on the local disk, 0:19:15.754 for example, on our Web server, 0:19:17.411 we might actually have to go out into the network and fetch 0:19:20.613 it. And if this network is the 0:19:22.214 Internet, right, the Internet has a huge amount 0:19:24.754 of data. I mean, who knows how much it 0:19:26.797 is. It's certainly orders of 0:19:28.288 terabytes. And it could take a long time 0:19:31.743 to get a page of the Internet. So it might take 100 ms to 0:19:35.11 reach some remote site on the Internet. 0:19:37.395 All right, so the point about this IO bottleneck is that this 0:19:41.003 is going to be a very common, sort of the disparity in the 0:19:44.43 performance of these different levels of the system is going to 0:19:48.158 be a very common source of performance problems in our 0:19:51.344 computers. So in particular, 0:19:52.968 if you look at the access time, here's 1 ns. 0:19:55.553 The access time down here is 100 ms. 0:19:57.598 This is a ten to the eighth difference, right, 0:20:00.303 which is equal to 100 million times difference in the 0:20:03.43 performance of the fastest to the slowest thing here. 0:20:08 So, if the CPU has to wait for something to come over the 0:20:11.869 network, you're waiting for a very long time in terms of the 0:20:15.946 amount of time the CPU takes to, say, read a single word of 0:20:19.953 memory. So when we look at the 0:20:21.957 performance of a computer system, we're going to see that 0:20:25.827 often this sort of IO bottleneck is the problem with that system. 0:20:30.249 So if we look, for example, 0:20:32.046 at our Web server, with its three stages, 0:20:34.809 where at this stage is the one that goes to disk, 0:20:38.126 this is the HTML stage, which maybe can just be 0:20:41.305 computed in memory. And this is the network stage. 0:20:45.737 We might be talking about 10 ms latency for the disk stage. 0:20:49.097 We might be talking about just 1 ms for the HTML page, 0:20:52.167 because all it has to do is do some computation in memory. 0:20:55.469 And we might be talking about 100 ms for the network stage to 0:20:58.945 run because it has to send some data out to some remote site. 0:21:03 So if you, in order to process a single request, 0:21:06.167 have to go through each of these steps in sequence, 0:21:09.536 then the total performance of the system, the time to process 0:21:13.58 a single request is going to be, say for example, 0:21:16.815 111 ms, the sum of these three things, OK? 0:21:19.578 And so if you look at the system and you say, 0:21:22.543 OK, what's the performance bottleneck in this system? 0:21:26.047 So the performance bottleneck, right, is clearly this network 0:21:30.091 stage because it takes the longest to run. 0:21:34 And so if we want to answer a question about where we should 0:21:37.609 be optimizing the system, one place we might think to 0:21:40.791 optimize is within this network stage. 0:21:43.054 And we'll see later an example of a simple kind of optimization 0:21:46.848 that we can apply based on this notion of concurrency to improve 0:21:50.702 the performance of the networking stage. 0:21:53.088 So as I just said, the notion of concurrency is 0:21:55.902 going to be the way that we are really going to get at sort of 0:21:59.635 eliminating these IO bottlenecks. 0:22:01.592 So -- 0:22:03 0:22:09 And the idea is going to be that we want to overlap the use 0:22:12.824 of some other resource during the time that we are waiting for 0:22:16.846 one of these slow IO devices to complete. 0:22:19.483 And, we are going to look at two types of concurrency. 0:22:22.978 We're going to look at concurrency between modules -- 0:22:27 0:22:31 -- and within a module, OK? 0:22:32.714 So we may have modules that are composed, for example, 0:22:36.21 our networking module may be composed of multiple threads, 0:22:39.97 each of which can be accessing the network. 0:22:42.74 So that's an example of concurrency within a module. 0:22:46.104 And, we're going to look at the case of between module 0:22:49.599 concurrency where, for example, 0:22:51.578 the HTML module can be processing, can be generating an 0:22:55.008 HTML page, while the disk module is reading a request for another 0:22:59.229 client at the same time. OK, and so the idea behind 0:23:04.449 concurrency is really going to be by using concurrency, 0:23:09.536 we can hide the latency of one of these slow IO stages. 0:23:15 0:23:22 OK, so the first kind of concurrency we're going to talk 0:23:25.955 about is concurrency between modules. 0:23:28.544 And the primary technique we use for doing this is 0:23:32.068 pipelining. So the idea with pipelining is 0:23:35.017 as follows. Suppose we have our Web server 0:23:37.965 again. And this time let's draw it as 0:23:40.554 I drew it at first with Q's between each of the modules, 0:23:44.51 OK? So, we have our Web server 0:23:46.595 which has our three stages. And suppose that what we are 0:23:50.551 doing is we have some set of requests, sort of an infinite 0:23:54.65 queue of requests that is sort of queued up at the disk thread, 0:23:59.109 and the disk thread is producing these things. 0:24:04 And we're sending them through. Well, we want to look at how 0:24:08.498 many pages come out here per second, and what the latency of 0:24:12.997 each page is. So, if we have some list of 0:24:16.046 requests, suppose these requests are numbered R1 through RN, 0:24:20.545 OK? So what's going to happen is 0:24:22.909 that the first request is going to start being processed by the 0:24:27.636 disk server, right? So, it's going to start 0:24:31.279 processing R1. Now, in a pipelining system, 0:24:33.616 what we're going to want to do is to have each one of these 0:24:36.844 threads sort of working on a different request, 0:24:39.403 each one of these modules working on a different request 0:24:42.464 at each point in time. And because the disk is this 0:24:45.246 independent resource from the CPU, is an independent resource 0:24:48.585 from the network, this is going to be OK. 0:24:50.81 These three modules aren't actually going to contend with 0:24:53.926 each other too much. So what's going to happen is 0:24:56.597 this guy's going to start processing R1, 0:24:58.767 right? And then after 10 ms, 0:25:00.27 he's going to pass R1 up to here, and start working on R2, 0:25:03.441 OK? And now, 1 ms after that, 0:25:07.016 this guy is going to finish R1 and send it to here. 0:25:11.048 And then, 9 ms after that, R2 is going to come up here. 0:25:15.403 And this guy can start processing R3. 0:25:18.306 OK, so does everybody sort of see where those numbers are 0:25:22.822 coming from? OK. 0:25:24.032 [LAUGHTER] Good. So now , what we're going to do 0:25:27.822 is if we look at time starting with this equal to time zero, 0:25:32.58 in terms of the requests that come in and out of this last 0:25:37.177 network thread, we can sort of get a sense of 0:25:40.725 how fast this thing is processing. 0:25:45 So the first R1 enters into this system after 11 ms, 0:25:48.762 right? It takes 10 ms to get through 0:25:51.344 here and 1 ms to get through here. 0:25:53.778 And, it starts processing R1 at this time. 0:25:56.803 So, I'm going to write plus R1 to suggest that we start 0:26:00.786 processing and here. The next time that this module 0:26:05.674 can do anything is 100 ms after it first started processing, 0:26:10.933 the next time this module does anything is 100 ms after it 0:26:16.013 started processing R1. So, at time 111 ms, 0:26:19.668 it can output R1, or it's done processing R1. 0:26:23.59 And then, of course, by that time, 0:26:26.532 R2 and R3, some set of requests have already queued up in this 0:26:31.969 queue waiting for it. So it can immediately begin 0:26:37.059 processing R2 at this time, OK? 0:26:39.348 So then, clearly what's going to happen is after 211 ms, 0:26:43.544 it's going to output R2, and it's going to begin 0:26:47.129 processing R3, OK? 0:26:48.426 So, there should be a plus there and a plus there. 0:26:52.164 So, and similarly, at 311 we're going to move onto 0:26:55.903 the next one. So, he look now at the system, 0:26:59.183 we've done something pretty interesting, which is that it 0:27:03.455 still took us, sort of the time for this 0:27:06.43 request to travel through this whole thing was 110 ms. 0:27:12 But if you look at the inter- arrival time between each of 0:27:15.814 these successive outputs of R1, they are only 100 ms, 0:27:19.176 right? So we are only waiting as long 0:27:21.504 as it takes R1 to process a result in order to produce these 0:27:25.318 results, in order to produce answers. 0:27:27.646 So by pipelining the system in this way and having the Web 0:27:31.331 server thread and the disk thread do their processing on

0:27:34.887 later requests while R1 is processing its request, 0:27:38.056 we can increase the throughput of the system. 0:27:42 So in this case, we get an arrival every 100 ms. 0:27:45.823 So the throughput is now equal to one result every 100 ms, 0:27:50.461 or ten results per second, OK? 0:27:52.82 So, even though the latency is still 111 ms, 0:27:56.318 the throughput is no longer one over the latency because we have 0:28:01.444 separated them in this way by pipelining them. 0:28:06 OK, so that was good. That was nice. 0:28:08.042 We improve the performance of the system a little bit. 0:28:11.134 But we didn't really improve it very much, right? 0:28:13.935 We increased the throughput of this thing a little bit. 0:28:17.086 But we haven't really addressed what we identified earlier as 0:28:20.587 being a bottleneck, which the fact that this R1 0:28:23.271 stage is taking 100 ms to process. 0:28:25.197 And in general, when we have a pipeline system 0:28:27.822 like this, we can say that the throughput of the system is 0:28:31.148 bottlenecked by the slowest stage of the system. 0:28:35 So anytime you have a pipeline, the throughput of the system is 0:28:38.647 going to be the throughput of the slowest stage. 0:28:41.411 So in this case, the throughput is 10 results 0:28:44 per second. And that's the throughput of 0:28:46.294 the whole system. So if we want to increase the 0:28:49 throughput anymore than this, what we're going to have to do 0:28:52.47 is to somehow improve the performance of this module here. 0:28:55.823 And the way that we're going to do that is also by exploiting 0:28:59.352 concurrency. 0:29:01 0:29:05 This is going to be this within a module concurrency. 0:29:09.666 So if you think about how a Web server works, 0:29:13.615 or how a network works, typically when we are sending 0:29:18.282 these requests to a client, it's not that we are using up 0:29:23.307 all of the available bandwidth of the network when we are 0:29:28.333 sending these requests to a client, right? 0:29:33 You may be able to send 100 MB per second out over your 0:29:36.144 network. Or if you're connected to a 0:29:38.183 machine here, you may be able to send 10 MB a 0:29:40.745 second across the country to some other university. 0:29:43.657 The issue is that it takes a relatively long time for that 0:29:46.976 request to propagate, especially when that request is 0:29:50.004 propagating out over the Internet. 0:29:51.926 The latency can be quite high. But you may not be using all 0:29:55.304 the bandwidth when you are, say, for example, 0:29:57.866 sending an HTML page. So in particular it is the case 0:30:00.895 that multiple applications, multiple threads, 0:30:03.457 can be simultaneously sending data out over the network. 0:30:08 And if that doesn't make sense to you right now, 0:30:10.145 we're going to spend the whole next four lectures talking about 0:30:13.238 network performance. And it should make sense for 0:30:15.632 you. So just take my word for it 0:30:17.179 that one of the properties of the network is so that the 0:30:19.922 latency of the network may be relatively high. 0:30:22.167 But in this case we are not actually going to be using all 0:30:25.011 the bandwidth that's available to us. 0:30:26.807 So that suggests that there is an idle resource. 0:30:30 It means that we sort of have some network bandwidth that we 0:30:33.916 could be using that we are not using. 0:30:36.306 So we'd like to take advantage of that in the design of our 0:30:40.156 system. So we can do this in a 0:30:42.081 relatively simple way, which is simply to say, 0:30:45.068 let's, within our networking module, rather than only having 0:30:48.719 one thread sending out requests at one time, let's have multiple 0:30:52.901 threads. Let's, for example have, 0:30:55.025 say we have 10 threads. So we have thread 1, 0:30:58.012 thread 2, thread 10, OK? 0:31:01 And we're going to allow these all to be using the network at 0:31:05.207 once. And they are all going to be 0:31:07.522 talking to the same queue that's connected to the same HTML 0:31:11.589 module that's connected to the same disk module. 0:31:14.885 And there's a queue between these as well. 0:31:17.761 OK, so now when we think about the performance of this, 0:31:21.548 now let's see what happens when we start running requests 0:31:25.475 through this pipeline. And let's see how frequently we 0:31:30.529 get requests coming out of the other end. 0:31:33.792 We draw our timeline again. You can see that R1 is going to 0:31:38.524 come in here, and then after 10 ms it's going 0:31:42.114 to move to here. And then after 11 ms it'll 0:31:45.54 arrive here. We'll start processing request 0:31:48.967 one. Now the second request, 0:31:51.17 R2, is going to be here. And, we're going to have 9 ms 0:31:55.494 of processing left to do on it. After R1, it gets sent on to 0:32:00.307 the next thread. So, R2 is going to be in here 0:32:05.127 for 9 ms. It will be in here for 1 ms. 0:32:07.842 So, 10 ms after R1 arrives here, R2 is going to arrive 0:32:11.73 here. So, what we have is we have 11 0:32:14.298 ms. We have R1. 0:32:15.326 Now, 10 ms later, we have R2. 0:32:17.38 OK, so now you can see that suddenly this module, 0:32:20.902 this system is able to process multiple requests, 0:32:24.423 so it has multiple requests that processing at the same 0:32:28.385 time. And so 10 ms after that, 0:32:31.718 R3 is going to start being processed, and then, 0:32:35.011 so what that means is that after some passage of time, 0:32:38.806 we're going to have R10 in here. 0:32:41.026 And, that's going to go in after 101 ms, 0:32:43.818 right? So, we're going to get R10. 0:32:46.181 OK, and now we are ready to start processing. 0:32:49.331 Now we've sort of pushed all these through. 0:32:52.338 And now, suppose we start processing R11. 0:32:55.202 OK, so R11 is going to flow through this pipeline. 0:33:00 And then, it's at time 111, R11 is going to be ready to be 0:33:04.814 processed. But notice that at time 111, 0:33:08.024 we are finished processing R1, right? 0:33:11.065 So, at this time, we can add R11 to the system, 0:33:14.95 and we can output R1. OK, so now every 10 ms after 0:33:19.089 this, another result is going to arrive, and we're going to be 0:33:24.242 able to output the next one. OK, and this is just going to 0:33:29.057 continue. So now, you see what we've 0:33:32.76 managed to do is we've made this system so that every 10 ms after 0:33:37.095 this sort of startup time of 111 ms, after every 10 ms, 0:33:40.752 we are producing results, right? 0:33:42.852 So we are going to get, actually, 100 per second. 0:33:46.103 This is going to be the throughput of this system now. 0:33:49.693 OK, so that was kind of neat. How did we do that? 0:33:52.944 What have we done here? Well, effectively what we've 0:33:56.398 done is we've made it so that this module here can process 0:34:00.258 sort of 10 times as many requests as it could before. 0:34:05 So this module itself now has 10 times the throughput that it 0:34:09.192 had before. And we said before that the 0:34:11.847 bottleneck in the system is, the throughput of the system is 0:34:15.969 the throughput of the slowest stage. 0:34:18.414 So what we've managed to do is decrease the throughput of the 0:34:22.606 slowest stage. And so now the system is 0:34:25.262 running 10 times as fast. Notice now that the disk thread 0:34:29.174 and the network threads both take 10 ms, sort of the 0:34:32.737 throughput of each of them is 100 per second. 0:34:37 And so, now we have sort of two stages that have been equalized 0:34:41.26 in their throughput. And so if we wanted to further 0:34:44.696 increase the performance of the system, we would have to 0:34:48.476 increase the performance of both of these stages, 0:34:51.774 not just one of them. OK, so that was a nice result, 0:34:55.279 right? It seems like we've done 0:34:57.341 something sort of, we've shown that we can use 0:35:00.433 this notion of concurrency to increase the performance of a 0:35:04.419 system. But, we've introduced a little 0:35:09.099 bit of a problem. In particular, 0:35:12.303 the problem we've introduced is as follows. 0:35:16.643 So, remember, we said we had this set of 0:35:20.673 threads, 1 through, say for example, 0:35:24.496 10, that are processing, they're all sharing this queue 0:35:30.179 data structure that is connected up to our HTML thread. 0:35:37 So, the problem with this is that what we've done is to 0:35:40.767 introduce what's called a race condition on this queue. 0:35:44.534 And I'll show you what I mean by that. 0:35:47.116 So if we look at our code snippet up here, 0:35:49.976 for example for what's happening in our HTML thread, 0:35:53.534 we see that what it does is it calls de-queue, 0:35:56.674 right? So the problem that we can have 0:35:59.255 is that we may have multiple of these modules that are 0:36:02.953 simultaneously executing at the same time. 0:36:07 And they may simultaneously both call de-queue, 0:36:10.184 right? So depending on how de-queue is 0:36:12.746 implemented, we can get some weird results. 0:36:15.653 So, let me give you a sort of very simple possible 0:36:19.046 implementation of de-queue. Suppose that what de-queue does 0:36:23.061 is it reads, so, given this queue here, 0:36:25.692 let's say the queue is managed by, there's two variables that 0:36:29.846 keep track of the current state of this queue. 0:36:34 There is a variable called first, which points to the head 0:36:37.585 of the queue, and there's a variable called 0:36:40.227 last, which first points to the first used element in this 0:36:43.813 queue, and last points to the last used element. 0:36:46.77 So, the elements that are in use in the queue at any one time 0:36:50.544 are between first and last, OK? 0:36:52.431 And, what's going to happen is when we de-queue, 0:36:55.388 we're going to sort of move first over one, 0:36:58.03 right? So, when we de-queue something, 0:37:00.357 we'll free up this cell. And when we enqueue, 0:37:04.21 we'll move last down one. And then, when last reaches the 0:37:07.601 end, we are going to wrap it around. 0:37:09.72 So this is a fairly standard implementation of the queue. 0:37:13.11 It's called the circular buffer. 0:37:14.987 And if first is equal to last, then we know that the queue is 0:37:18.62 full. So that's the condition that we 0:37:20.799 can check. So we are not going to go into 0:37:23.221 too many details about how this thing is actually implemented. 0:37:26.914 But let's look at a very simple example of how de-queue might 0:37:30.546 work. So remember we have these two 0:37:33.897 shared variables first and last that are shared between these, 0:37:37.887 say, all these threads that are accessing this thing. 0:37:41.289 And what de-queue might do is to say it's going to read a 0:37:44.953 block, read a page from this queue, so read the next HTML 0:37:48.616 page to output, and it's going to read that 0:37:51.364 into a local variable called page. 0:37:53.523 Let's call this queue buf, B-U-F, I mean we'll use 0:37:56.859 array notation for accessing it. 0:38:00 So it's going to re-buff sub first, OK, and then it's going 0:38:04.923 to increment first. First gets first plus one, 0:38:08.742 and then it's going to return page. 0:38:11.628 OK, that seems like a straightforward implementation 0:38:15.957 of de-queue. And so we have one thread 0:38:19.098 that's doing this. Now, suppose we have another 0:38:23.002 thread that's doing exactly the same thing at the same time. 0:38:28.01 So it runs exactly the same code. 0:38:32 And remember that these two threads are sharing the 0:38:36.605 variables buf and first. 0:38:39 0:38:46 OK, so if you think about this and you think about how these two 0:38:49.392 things running two threads at the same time, 0:38:51.865 there is sort of an interesting problem that can arise. 0:38:54.971 So one thing that might happen when we are running these two 0:38:58.364 things at the same time is that the thread scheduler might first 0:39:01.987 start running thread 1. And it might run the first 0:39:05.58 instruction of thread 1. And then it might run the 0:39:08.681 second instruction. And then it might run this 0:39:11.417 return thing. And then it might come over here, 0:39:14.031 and it might start running T2. So, it might, 0:39:16.645 then, stop running T1 and start running T2, and execute its 0:39:20.171 three instructions. So if the thread scheduler does 0:39:23.211 this, there's nothing wrong. It's not a problem, 0:39:26.069 right? The thread scheduler, 0:39:27.71 each of these things read its value from the queue and 0:39:30.932 incremented it. T1 read one thing from the 0:39:34.798 queue, and then T2 read the next thing from the queue. 0:39:38.464 So clearly some of the time this is going to work fine. 0:39:42.199 So let's make a list of possible outcomes. 0:39:45.035 Sometimes we'll be OK. The first possible outcome was 0:39:48.632 OK. But let's look at a different 0:39:50.845 situation. Suppose what happens is that 0:39:53.474 the first thing the thread scheduler does is schedule T1. 0:39:57.347 And T1 executes this first instruction, and then just after 0:40:01.359 that the thread scheduler decides to preempt T1, 0:40:04.61 and allow T2 to start running. So it in particular allows T2 0:40:10.653 to execute this de-queue instruction to its end, 0:40:14.954 and then it comes over here and it runs T1. 0:40:18.797 OK, so what's the problem now? 0:40:22 0:40:28 Yeah? 0:40:29 0:40:33 Right, OK, so they both read in the same page variable. 0:40:36.454 So now both of these threads have de-queued the same page. 0:40:40.101 So the value first, for T1, it was pointing here. 0:40:43.172 And then we switched. And it was still pointing here, 0:40:46.5 right? And now, so both of these guys 0:40:48.803 have read the same page. And now they are both at some 0:40:52.194 point going to increment first. So you're going to increment it 0:40:56.161 once. Then you're going to increment 0:40:58.4 it again. So this second element here in 0:41:01.71 the queue has been skipped. OK, so this is a problem. 0:41:04.675 We don't want this to happen. Because the system is not 0:41:07.754 outputting all the pages that it was supposed to output. 0:41:10.89 So what can we do to fix this? 0:41:13 0:41:20 So the way that we fixed this is by introducing something we 0:41:23.39 call isolation primitives. 0:41:25 0:41:31 And the basic idea is that we want to introduce an operation 0:41:35.237 that will make it so that any time that the page variable gets 0:41:39.695 read out of the queue, that we also at the same time 0:41:43.421 increment first without any other sort of threads' accesses 0:41:47.659 to this queue being interleaved with our accesses to this queue, 0:41:52.263 or our de-queues from the queue. 0:41:54.528 So in sort of technical terms, what we say is we want these 0:41:58.766 two things, the reading of page and the incrementing of first to 0:42:03.369 be so-called atomic. OK, and the way that we're going 0:42:07.763 to make these things atomic is by isolating them from each 0:42:11.004 other, that by isolating these two threads from each other when 0:42:14.53 they are executing the enqueue and de-queue things. 0:42:17.374 So, these two terms we're going to come back to in a few months 0:42:20.9 in a class towards the end of the class. 0:42:23.175 But all you need to understand here is that there is this race 0:42:26.644 condition, and we want some way to prevent it. 0:42:30 And the way that we're going to prevent it is by using these 0:42:34.701 isolation routines also sometimes called locks. 0:42:38.367 So in this case, the isolation schemes are going 0:42:42.113 to be called locks. So the idea is that a lot is 0:42:45.858 simply a variable, which can be in one of two 0:42:49.365 states. It can either be set or unset. 0:42:52.313 And we have two operations that we can apply on a lock. 0:42:56.616 We can acquire it, and we can release it. 0:43:01 OK, and acquire and release have the following behavior. 0:43:05.782 What acquire says is check the state of the lock, 0:43:09.956 and if the lock is unset, then change the state to set. 0:43:14.652 But if the lock is set, then wait until the lock 0:43:18.739 becomes unset. What a release says is it 0:43:22.13 simply says change the state of the lock from unset to set, 0:43:27.173 or from set to unset, excuse me. 0:43:31 So let's see how we can use these two routines in our code. 0:43:35.38 So let's go back to our example of enqueue and de-queue. 0:43:39.533 Let's introduce a lock variable. 0:43:41.875 We'll call it TL for thread lock. 0:43:44.291 And, what we're going to do is simply around these two 0:43:48.294 operations to access the queue, to modify this page and first, 0:43:52.901 to read the page and modify first, we're going to put in an 0:43:57.281 acquire and a release. 0:44:00 0:44:21 OK so we have ACQ on this thread lock, and we have release 0:44:24.128 on this thread lock. OK, so let's look, 0:44:26.214 so this seems fine. It looks like we've done this. 0:44:28.904 But it's sort of positing the existence of this acquire 0:44:31.868 procedure that just sort of does the right thing. 0:44:34.503 If you think about this for a minute, it seems like we can 0:44:37.632 have the same race condition problem in the acquire module as 0:44:40.926 well, right, or the acquire function as well. 0:44:43.341 With two guys both try and acquire the lock at the same 0:44:46.305 time? How are we going to avoid this 0:44:48.226 problem? And there's a couple of ways 0:44:50.202 that are sort of well understood for avoiding this problem in 0:44:53.496 practice, and their talked about in the book. 0:44:55.801 I'm just going to introduce the simplest of them now, 0:44:58.656 which is that we're going to add a special instruction to the 0:45:01.949 microprocessor that allows us to do this, acquire efficiently. 0:45:07 It turns out that most modern microprocessors have an 0:45:12.605 equivalent instruction. So we're going to call this 0:45:17.996 instruction RSL for read and set lock. 0:45:21.985 OK, so the idea with RSL is as follows. 0:45:26.081 We can basically, the implementation of the 0:45:30.609 acquire module is going to be like this. 0:45:36 What it's going to do, remember we want to wait until 0:45:39.724 we want to loop. We don't have the lock. 0:45:42.517 If we don't have the lock, we want to loop until we 0:45:46.384 have the lock. So the implementation require 0:45:49.535 may look as follows. We'll have a local variable 0:45:52.901 called held. We'll initially set it to false 0:45:55.981 in a while loop while we don't hold the lock. 0:45:59.132 We're going to use this RSL instruction. 0:46:03 So, what this says is held equals RSL of TL, 0:46:06.262 OK? So, what the RSL instruction 0:46:08.615 does is it looks at the state of the lock, and if the lock is 0:46:13.168 unset, then it sets it. And if the lock is set, 0:46:16.658 then it sets it and it returns true. 0:46:19.314 And if the lock is set, then it returns false. 0:46:22.728 So it has the property that it can both read and set the lock 0:46:27.281 within a single instruction, right? 0:46:31 And we're going to use this read and set lock sort of 0:46:34.438 primitive, this basic thing, as a way to build up this sort 0:46:38.272 of more complicated acquire function, which we can then use 0:46:42.107 to build up these locks. OK, so anytime you're designing 0:46:45.743 a multithreaded system in this way, or a system with lots of 0:46:49.644 concurrency, you should be worrying about whether you have 0:46:53.413 race conditions. And if you have race 0:46:55.793 conditions, you need to think about how to use locks in order 0:46:59.76 to prevent those race conditions. 0:47:03 All right, so there are a couple of other topics related 0:47:06.971 to performance that appear in the text. 0:47:09.715 And one of those topics is caching. 0:47:12.17 And I just want to spend one very brief minute on caching. 0:47:16.286 So you guys have already seen catching presumably in the 0:47:20.258 context of 6.004 with processor caches. 0:47:23.002 And what we would want to do, so you might want to sit down 0:47:27.19 and think through as an example of how you would use a cache, 0:47:31.522 you improve the performance of our Web server. 0:47:36 So one thing that you might do in order to improve the 0:47:39.806 permanence of the Web server is to put a cash in the disk thread 0:47:44.331 that you use instead of going to disk in order to sort of reduce 0:47:48.856 the latency of a disk access. And I'll at the beginning of 0:47:52.95 class next time take you through a very simple example of how we 0:47:57.475 can actually use the disk thread in order to do that. 0:48:02 But you guys should think about this a little bit on your own. 0:48:04.808 So barring that little digression that we'll have next 0:48:07.248 time, this takes us to the end of our discussion of sort of 0:48:09.918 modularity, abstraction, and performance. 0:48:11.76 And what we're going to start talking about next time is 0:48:14.292 networking, and how networks work. 0:48:15.812 But I want you guys to make sure you keep in mind all these 0:48:18.482 topics that we've talked about because these are going to be 0:48:21.198 the sort of fundamental tools that we are going to use 0:48:23.639 throughout the class in the design of computer systems. 0:48:27 So because we've finished this module, it doesn't mean that 0:48:29.676 it's sort of OK to stop thinking about this stuff. 0:48:31.938 You need to keep all of this in mind at the same time. 0:48:34.384 So we'll see you all on Wednesday.