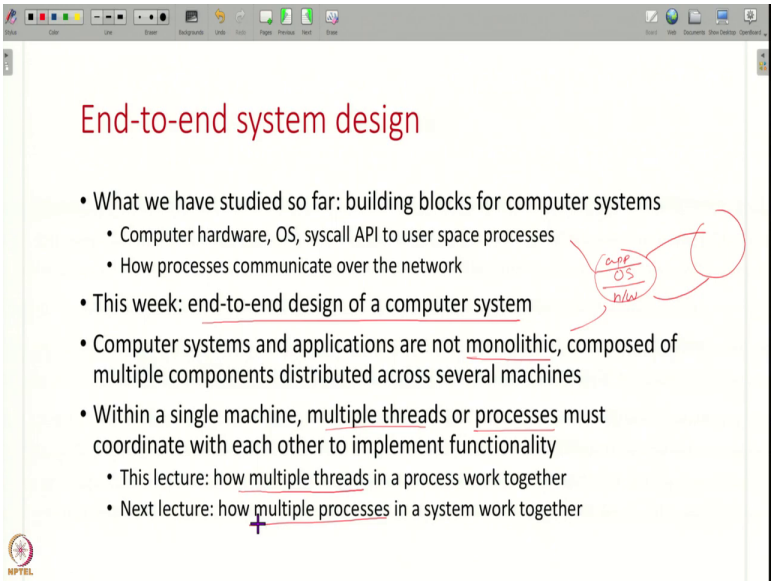
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 38
Multithreaded Application Design

Hello, everyone, welcome to the 26th lecture in the course Design and Engineering of Computer Systems. So, far in the previous 5 weeks, we have seen the various building blocks of computer systems, we have seen how hardware works, how the operating system manages the hardware, how computers communicate over the network.

So, we have seen all the building blocks. So, in this week, we are now ready to start putting all the pieces together and designing a system end to end. So, we have a couple more building blocks to cover in the first couple of lectures of this week, where we understand how the different threads in a process and different processes in a system communicate with each other. And after that, we are going to put together an end to end design of a somewhat realistic computer system. So, let us get started.

(Refer Slide Time: 01:14)



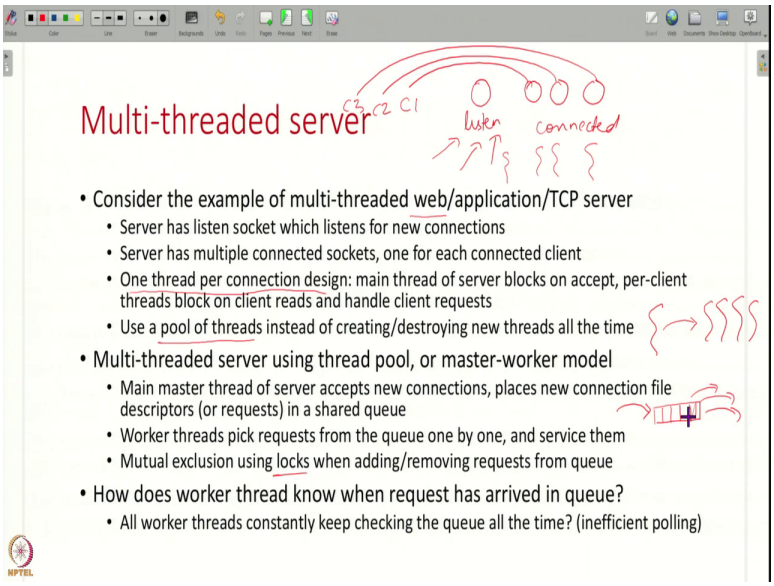
So, this week, as I said, we will cover end to end design of computer systems. And the one thing to note is that computer systems in real life, they are not monolithic. So, they are composed of multiple components that are distributed across several machines. So, each of those components will have some hardware, some OS running and managing the hardware and some applications

running on top of it and all of these are communicating over the network that is how real-life computer systems are composed of multiple such components.

And all of these building blocks of hardware OS the system call API for user applications, communication over the network, all of that we have seen. So, the next thing we going to see is how to put all of these pieces together. So, in a single system, also you do not have just one process or something. You typically have, your application will have multiple threads or multiple processes, which are working together with each other.

So, before we put together an end to end system, these are the two things we need to understand, we need to understand how multiple threads in a process work together and how multiple processes in a system work together. So, we will do that in the next 2 lectures and the lecture after that, we will start putting together an end to end system design.

(Refer Slide Time: 03:10)



So, first, let us understand how you design applications that have multiple threads in them. So, suppose you have a web server or any other application server that is talking to various clients and providing some service typically, a lot of these servers have multiple threads, they are multi threaded. So, why is that? We have seen the reason for this in one of the previous lectures. For example, if you consider a web server, it needs to have multiple threads.

Why? Because the server is managing multiple sockets you know the web server has one listen socket that is listening for new connections and once a connection is established with the client,

you have multiple connected sockets, each talking to a different client. So, the server needs to manage all of these sockets, the listen socket, as well as the corrected sockets.

And one way to do it is you to use a one thread per connection design. So, the server will have multiple threads, the main thread of the server can be listening to this listen socket and handling new connection requests. Then once a new connection is established, you can create multiple threads and give each thread the socket and this thread will handle this client, read the request of the client process the request send a reply back whatever is needed for this client C1, C2 for each client, you have a thread at the server, that is the one thread per connection design.

This thread blocks on the accept system call and these threads block on reading from the socket. So, this is something that we have seen before. That is why most application servers in a client server model need to have multiple threads. So, the one subtlety is that instead of creating a thread every time a client connects and destroying the thread later on instead of having this churn of threads, what we can do is we can have a pool of threads.

The server has a pool of threads, there is a master thread and there is a pool of worker threads. So, the master thread is listening for new connections and whenever a new connection comes it will give it to one of the worker threads and say okay, you handle this connection and when the worker threads handle handles the connection, it will come back to the master and say, okay I am done. Then the master will give more work.

In this way, this is how real systems use thread pools, these are called thread pools. And primarily to avoid the overhead of constantly creating and destroying threats. So, when you have a thread pool, you need some way to coordinate between this main master thread that is giving out work to all of these worker threads.

And typically, the way it is done, this is usually some kind of a queue or a shared, buffer or something where the master thread will keep putting requests, hey, a new client has come, a new request has come. And the worker threads will keep taking the requests from the queue and servicing them one by one.

And of course, all of these threads if there any shared data structures, we have seen we use locks for mutual exclusion to avoid race conditions, all of that we have seen. So, now, you can

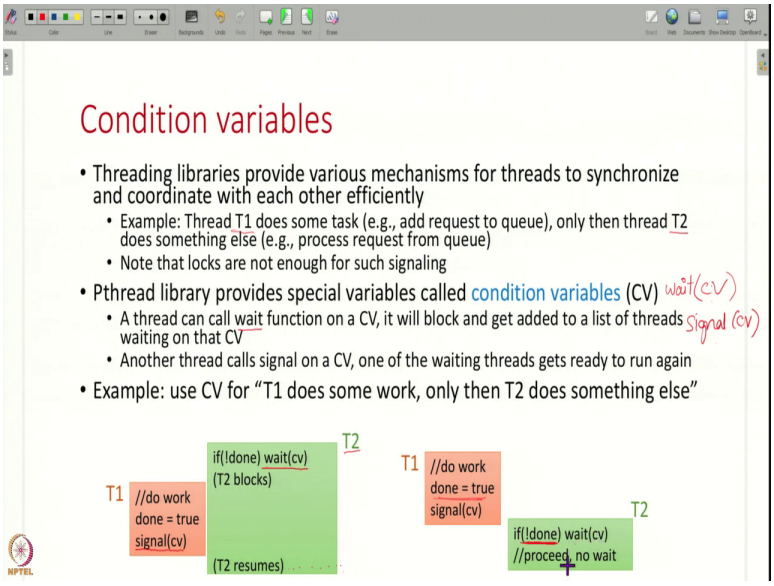
visualize this master worker model, work coming in being distributed to multiple people to do the work. But there is still a complication.

If you think about it closely, how do these worker threads know that work has arrived in the queue? There is a master that is creating work and there are worker threads waiting for work. So, now, how will the worker know, a request has come I should go check the queue. One ways, all the workers are constantly reading the queue is there work, is there work is there work. This is like the polling-based model.

But that is clearly inefficient, I mean, if work comes only once in a while, what is the reason for everybody to constantly keep waiting. A better way can be, the master can somehow wake up the threads, the threads can be sleeping in block state not running, the master can somehow wake up the threads and tell them, hey, look, some work has arrived for you.

We need this co-ordination mechanism between threads where a thread can tell another thread, hey, do this and this thread will say, okay, I am done with this. So, that is what we will study in this lecture. What is the mechanism available for threads to co-ordinate in this way with each other.

(Refer Slide Time: 07:19)



And one way to do it is using what are called condition variables. So, threading libraries provide many mechanisms for this thread synchronization. In this lecture, we will not have time to cover

all of them. But the simplest and the most useful of them is a mechanism called condition variables.

And this is useful for threads to do simple co-ordination between each other. For example, let us consider a concrete example. There are two threads in your application. Thread T1 does some work does some task, say it accepts a request and puts it in a queue. And only after that you want thread T2 to take the request from the queue and process it.

So T1 does something only then T2 does something. We do not want T2 to constantly keep waiting, running, wasting CPU time before T1 has done its share of the work. And so, we need some mechanism for this for patterns like this condition variables are useful. And note that locks alone, we have seen locks threading libraries that provide mechanisms for creating thread also provide locks like P thread locks.

But that is not enough, we need a different mechanism here. For this kind of signalling this kind of coordination, you need a different mechanism, which is condition variables. So, how do condition variables work? Conditioned variables is any variable, it is like any other variable like a lock or something you create in a program.

And on a condition variable, you can call two functions, you can call a wait function, and a signal function on a condition variable. So, whenever a thread calls wait on a condition variable, it will be blocked. And every condition variable internally, it maintains some sort of a list or queue of threads that are waiting.

And a thread can say, I also want to join the queue, I do not have work to do now, my task or my turn has not come, my task has not arrived, a thread can call wait. And another thread that wants this thread, the sleeping thread to run can call signal. So, thread that calls wait will be blocked and added to a list. Another thread that called signal will wake up one of these threads.

We will make one of these sleeping threads ready to run again, that is how wait and signal work. So, let us revisit this example of, we want this semantics of T1 does some work only then T2 does something else. How do you implement it using condition variables? Without condition variables, T2 is always checking is it done, is it done, is it my turn, is it my turn, that is inefficient. We do not want that.

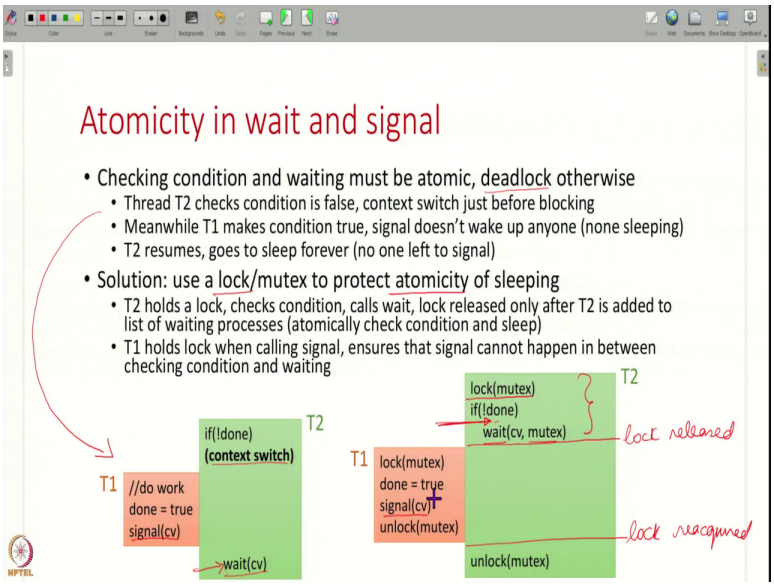
A better way will be something like this where T2 will check if the work is not done it will call wait on a condition variable and T1, when it does the work, which will unblock T2, it will basically call signal on a condition variable. So, for example, if T2 runs before T1, we do not want that, we want T1 to do its work only then T2 to run. So, if T2 runs first, it will check that this variable does not something it will check in the data structures indicating the work is not done it will call wait at this point T2 blocks it is context switched out, it will not be run.

Now, when T1 calls signal on a condition variable after doing the work, then this thread is marked as ready to run and the CPU scheduler will resume it at a later point of time. So, now whatever work T2 has to do after T1 that will be done here. So, this is an easy way for threads to synchronize.

And this wait and signal are of course implemented with the operating system support of moving some threads to a block state, ready state all of that is done. So, the wait function will put this thread into a block state, the signal function will wake up one of the threads and make them ready to run.

Now, what if T2 did not run first and only T1 ran first, then T1 does the work. At this point T2 is ready to run it does not have to wait. So, it will check. That is why there is usually some variable, you will check some condition you will check. Now, there is no need to wait this condition is already true. If this condition was false, you call wait, but if this condition is already true you know the work is already done, then there is no need to wait you will proceed. So, in this way, check the condition then only sleep only if the condition you are waiting for is not yet true, then you sleep. If the condition is already true, there is no need to call wait T2 will proceed.

(Refer Slide Time: 11:51)



So, now there is a slight complication with this wait and signal, there is another scope for race conditions that let us understand carefully. If you do not do this wait and signal carefully, you might end up in a very bad state, you might end up in what are called deadlocks. What is a deadlock? Where a thread is somehow sleeping not doing any work.

So, let us check what these conditions are. So, I am describing that condition here in this figure where some bad things can happen. So, for example, we had this code T2 is checking, if the condition is true, if it is true, it will not wait but if the condition is not yet satisfied T1 is not yet done, then it has to call wait.

Now, suppose T2 has checked this condition and figured out that the condition is not yet satisfied T1 is not yet done. In this case T2 has to wait. So, now T2 has decided to wait has moved its program counter to this line. At that point, just before it calls wait context switch has happened unfortunately. This is always the problem with the threads and processes at a unfortunate time a context which can happen.

Now, T2 has decided to sleep, but it has not yet called wait it has not yet added itself to the list of waiting processes on this condition variable. At this point a context which occurs T1 runs it does the work it calls signal, will this signal wake anybody up? No, because T2 has not yet joined the queue. So, the signal will not wake anybody up, T1 is done.

Now, T2 resumes again, the program counter is here. So, T2 will execute this wait statement it will go to sleep. Now, will anybody wake T2 up? No, because T1 already did it signalling. So, T2 will sleep forever, this is a deadlock. Why did this happen? Because all of this should be done atomically, checking a condition and sleeping should be done in one go we do not want people to interrupt in between.

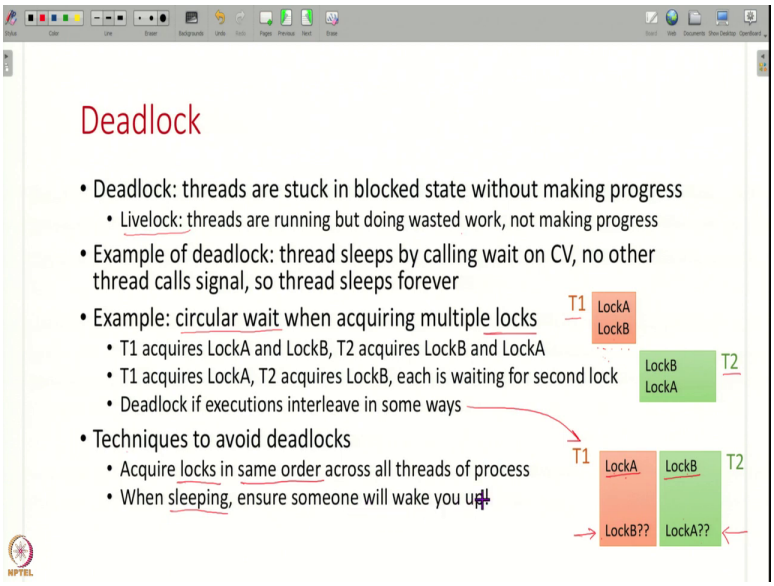
So, that is why if to protect this atomicity of sleeping we use a lock along with condition variables. So, the way condition variables are used is that the thread that wants to sleep will acquire a lock, check the condition and go to sleep and this lock will be passed to the wait function and after this thread goes to sleep fully, then this lock is released by this P thread or whatever library that is implementing this wait function.

After T2 sleeps then this lock is released. So, that the lock is now available for T1 before calling signal also you take the lock then call signal then you unlock. Now, T2 has slept. Now, T1 has taken the lock signal has woken T2 up now when T2 returns once again, the lock is reacquired by this library.

And T2 returns with the lock in it can unlock it, do whatever it wants with the lock. So, we are using a lock to protect this atomicity of checking a condition and going to sleep. Because of this lock, now this situation cannot occur, you cannot have a signal run here. Why? Because the signal also needs a lock, and you do not have the lock here, all of this is a critical section, nobody can interrupt you in between or no other critical section can be squeezed in here.

Therefore, even if T2 is context switched out, it is okay. Because T1 does not have the lock it cannot run the signal. So, therefore, all of this will happen only then the signal will happen. So, therefore, you have to use locks along with condition variables, of course, the implementation is taken care of by the libraries, but you have to understand that every time you call a sleep on a condition variable, you also have to acquire a lock and pass that lock. And whenever you call signal, you also have to hold this lock and call signal. This will ensure that the wait and the signal will happen either before or after, but not interleaved in this problematic manner.

(Refer Slide Time: 16:18)



So, what we have seen is an example of a situation that is called a deadlock. When you have multiple threads in a process, sometimes these threads can get stuck in a very bad state where they are all sleeping, they not doing any work and your system is stuck, that is called a deadlock. And we in general do not want these deadlocks to happen, because you want life to go on.

Another problem that can also occur with threads is what is called a livelock. Where the threads are running, but still not doing any useful work. The difference between deadlock and livelock is, in deadlock, the threads are blocked and not being woken up and livelock the threads are running, but not making progress.

These are related but slightly different problems. So, we have seen one example of a deadlock which is a thread sleeps by calling wait on a condition variable but no other thread calls signal. So, this thread sleeps forever, you have called wait and nobody else calls signal. Another example is even without condition variables with just locks you can have deadlocks, when you have multiple locks in your system and you acquire them in some weird order.

So, consider this example where there are two threads T1, T2 both of them need 2 locks they acquire 2 locks do some work release 2 locks. Now, T1 acquires them in this order lock A, lock B, T2 acquires lock B, lock A. Then what will happen, if sometimes all is well life is good, T1 acquires both locks, does it work releases the locks, then T2 acquires both locks does it work one after the other they work, that is okay.

But sometimes, you will have a weird situation like this, with some weird interleaving's you will have a situation where T1 has acquired lock A, then there is a context switch, T2 has acquired lock B then there is again a context switch now you are back to T1, T1 is waiting for lock B, will it get it? No, because T2 has it. T2 is waiting for lock A, it also will not get it because T1 has it. Will be one finish its work and release these locks? No, it will not. Similarly, with T2.

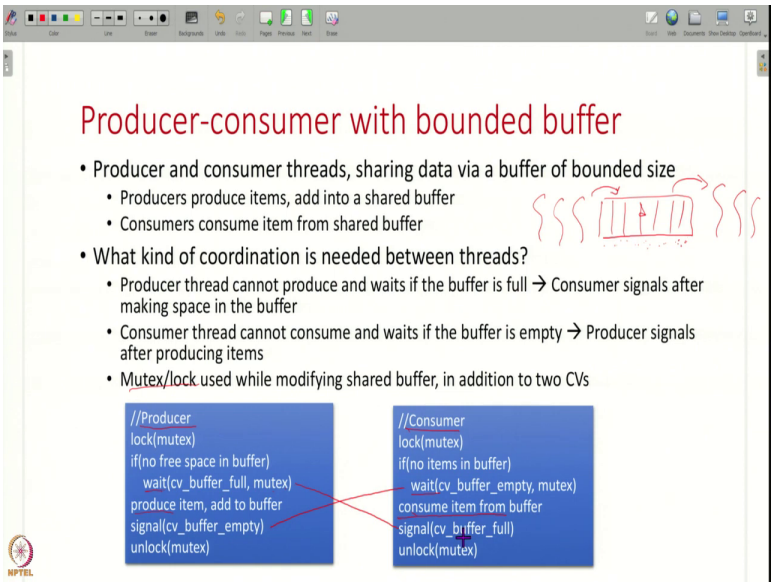
So, both of these have one thing they are waiting for the second thing neither will finish now if T1 gets both lock it will finish its critical section release the locks but we are not there yet. Similarly, for T2 so this is a deadlock. Even without condition variables also you can have deadlocks when you have multiple locks and you acquire them in different orders across different threads.

So, what are some of the best practices in your user programs to avoid deadlocks is, there are many techniques. Two simple ones is whenever you acquire locks, make sure that you acquire them in the same order across all threads. If you do lock A, lock B everywhere in your program do lock A, lock B do not do this jumbled up order that will cause deadlocks.

The other thing is, this acquiring locks in a jumbled-up order will cause what is called circular wait, where this guy is waiting for this, this lock this guy is waiting for this lock. And the other thing is when you have conditioned variables and whenever you are, doing any wait or sleep or something ensure that the code that has to wake you up can run.

Whenever you call wait on a condition variable always check that the code path that call signal can also run. For example, if that code path needs some logic needs some lock need something to run make sure that that path is open so that whenever you are sleeping, somebody will eventually wake you up.

(Refer Slide Time: 19:53)



So, now we have seen what our condition variables and a common pattern design pattern that is found in computer systems is what is called a producer consumer situation. You have some threads that are producing some work there are threads that are consuming somewhere and there is a shared buffer of some fixed size of some bounded size and these producer threads are adding items into this buffer and the consumer threads are reading items from this buffer and consuming them.

This is a common pattern in which threads in an application interact with each other. So, in such a scenario, let us see how we can write the code for such programs which have this producer consumer pattern using condition variables. So, here is some simple code for producers and consumer thread.

So, what the producer thread will do is, it will first check is there space in this buffer to produce more items, of course, everywhere you have a lock, any access to any of the shared buffer has to happen with a lock and the producer will check. If there is space in the buffer it will produce an item, if there is no space in the buffer then it will wait until the space is freed up in this buffer it will wait on a condition variable.

And who is waking up this producer? Whenever the consumer consumes an item it will call signal on this condition variable that will wake up the producer. So, the producers keep on producing at some point this buffer is full, then the producer threads will wait, then later on the

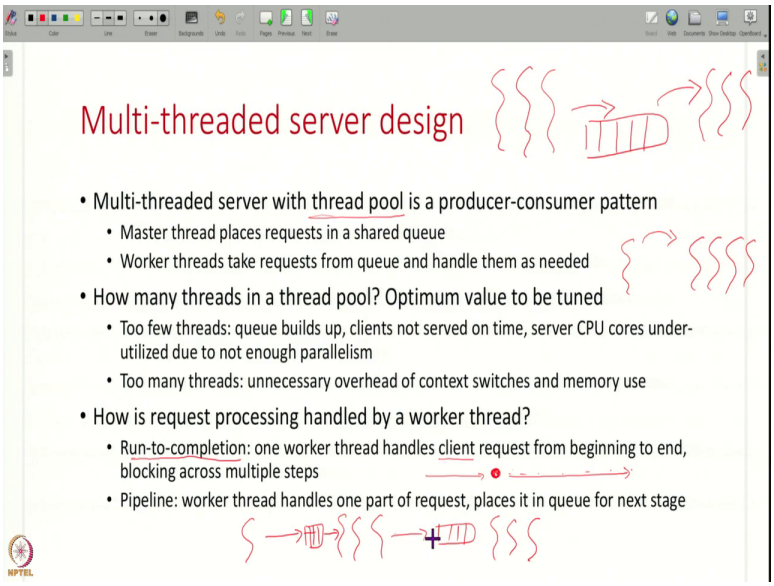
consumer threads will run they will consume items free up space in the buffer and they will signal the producer threads which will wake up produce again. So, you have a nice sort of tango going on here producers are producing, they are waiting the consumers are consuming telling the producers hey, produce more. Than putting the producers put them in the buffer consumers consume.

So, you have like a nice coordination happening between the threads in a system where they are both doing their work. Similarly, the consumer also will wait if there are no items in the buffer if suppose the producers have not yet produced anything, then there is no point for the consumer to run. So, the consumer will check if there are no items in the buffer the consumer will wait. Note that this is another queue for the consumers another condition variable and when the producer produces an item, the producer will signal the consumer and wake up this consumer thread. So, in this way, we are using two condition variables for roughly there are two types of waiting two types of queues.

One is for producer threads that are waiting for space in the buffer. One is for consumer threads that are waiting for items to consume. Therefore, you use two condition variables and of course, there is a Mutex lock that has to be held while accessing the shared data structures or while waiting, you can use the same lock for all of these multiple purposes.

So, this is a simple example of a producer consumer code. Producer will lock, check, if there is space to produce, otherwise, it will wait giving this lock as the argument to wait, so this producer thread will go to sleep this lock is released. At a later point of time when the producer is woken up, it will produce and signal the consumer. Similarly, the consumer code is also symmetric.

(Refer Slide Time: 23:26)



So, this is a very common pattern that occurs in multi threaded applications where some threads do some work, then signal the other threads, which will do the rest of the work. So, in a multithreaded server a TCP server, web server, any application server also, this producer consumer pattern is seen. For example, when you have a thread pool, you have a master thread that is or one or more master threads that are producing work.

They will put all these requests in a shared buffer and you have the worker threads that will take requests from the shared buffer and handle them. TCP server as whenever a new client comes the client socket or something will be put in this queue and the worker threads will take that socket from the queue and service that client read the client's request, send a reply back, and so on.

So, this is a very common pattern with multi threaded servers that have a thread pool to do some work for them. And now the question comes up, you have one master thread that is producing work and you have multiple worker threads that are doing the work in a thread pool. And whenever the worker thread finishes its work it will come back again to the queue, take the next item, it do that handle that request and come back again.

So, in this scenario, the question comes up, how many such threads should I have in the thread pool? What is this optimum value? Of course, this has to be tuned carefully in your particular system. There are pros, cons if you have too few threads, what will happen? The producer is putting a lot of requests there are not enough consumers enough worker threads to handle these

requests then your, your queue will build up, your clients may not be served on time, or you do not have enough threads to utilize all your CPU cores. So, you cannot have a thread pool with one or two threads usually that does not make sense. You need some respectable number of threads to quickly process all the request to use all the CPU cores for parallelism and so on.

At the same time, you cannot have like millions of threads, each thread occupies some memory, there is a PCB, there scheduler has to take care of it every thread adds some overhead. So, you cannot have like infinite or a very large number of threads in your system that is also counterproductive. So, this optimum value is usually has to be tubed.

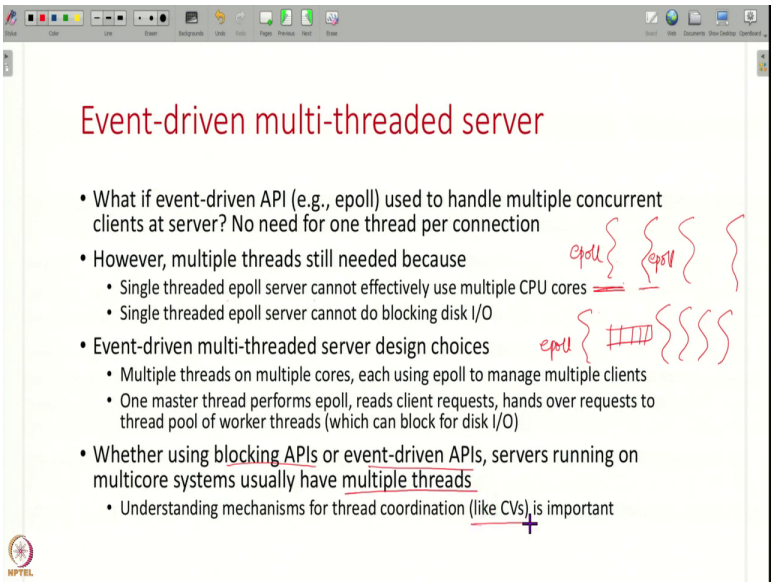
So, when we study performance, we will revisit how to tune all of these things for good performance, we will come back to this question later. Now, the other design aspect you have to consider when building multithreaded servers is, now you have this worker thread, it takes a request, a client has connected to a server, that client is assigned a worker thread, which will handle the clients request.

Now, there are again, different design choices here. Either you can do a run to completion model, that is each worker thread can basically fully handle a client, each worker thread is given a connected client socket, it can read from the socket, then handle the request, send a reply back again the client will say something, whatever the entire work that is needed for a request can be handled by just one thread, that is called run to completion model.

The other model is a pipeline model where you have some worker threads, the master has given some work, put something in a queue, and some worker threads do some part of the work, then they will put it in another queue, then other worker threads will do the other part of the work, you can have a pipeline of stages, one worker thread does something then says okay, my part is done, it will give it up to another worker thread, another worker thread between different stages, you can keep going back to the queue.

Of these, of course can be part of the same thread pool also. Each worker thread is specialized for a particular task, it will do that task, then hand it over to somebody. So, this is called the run to completion or the pipeline model. Both of these are valid design choices.

(Refer Slide Time: 27:27)



So, then you might be thinking of after all this discussion, you might be thinking, what if I use an event driven API, we had studied this, if I use something like epoll, I do not need to block on any socket read or anything, I can have a single threaded server process handle multiple clients. So, then the question might come up, why do I need all this multi threading worker threads, thread pool, and all of that if I use event driven programming.

But even with event driven programming in real life, you will need multiple threads because if you have a single threaded epoll server and you have multiple cores. In real life systems, you have to handle a large amount of traffic, and you are running on powerful high-end servers that have multiple cores.

So, you cannot just do a single threaded epoll server it cannot handle. It cannot use multiple cores. And it also cannot handle any blocking things like disk IO, if you have a single threaded server that is in an event loop, epoll wait request comes handle the request, then epoll wait. If you are doing that, if you have a single threaded server, this thread cannot block for disk read for example.

Then all the events will be kind of left without anybody to handle them. Therefore, in practice, even with event driven servers, you will have multiple threads. For example, you can have multiple threads on multiple cores, each of them calling epoll, each thread calls epoll handles

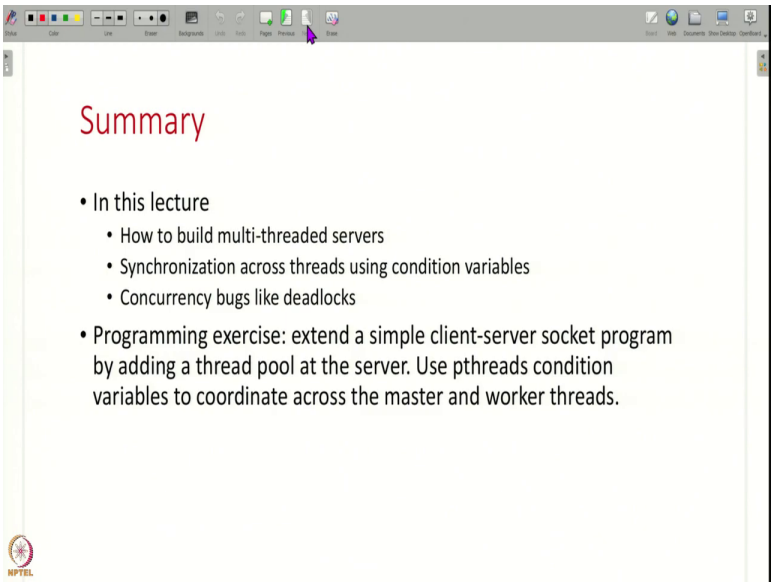
some requests on this core, handle some events handles a set of clients on this thread. Another thread on another core handles another set of events and so on.

So, you can have multiple threads each doing epoll. So, in order to use the multiple cores in a system, or you can also have a master worker model with epoll where there is one thread that is doing epoll collecting all the request new requests that are coming up, and then you have a pool of worker threads that are handling the work that is generated by this epoll master thread.

And these worker threads can handle disk IO, they can block, this thread, this epoll thread should not block the master thread. Why? Because the events are continuously coming up from the kernel and you have to handle them. But these other threads can block. So, the summary is that whatever API's you using, blocking API's or event driven API's. Typically, real life servers on multi core systems will end up having multiple threads and all of these threads need some mechanism for co-ordination like condition variables.

In this lecture, I have only spoken about the condition variables. But different programming languages, different libraries will give you many other such mechanisms where you can build these sorts of pipelines, thread pools, where one thread does some work, signals some other thread, then some other thread does something else. There are many mechanisms to do all of these. So, when you are building computer systems, when you are building a single server, a multi threaded server, it is important for you to understand what these mechanisms are and how to use them in a real system.

(Refer Slide Time: 30:47)



So, that all I have for today lecture we have seen how to build multi-threaded servers whether using blocking API is or event driven APIs and we have seen how mechanisms like condition variables are useful in such situations of course, there are many more such mechanisms and other programming language libraries. We have also studied briefly about your bugs like deadlocks that can occur in multi-threaded programs, where we have to take care that all your threads are running all the time and are not blocked in some unstable state.

So, as a programming exercise, I will request you to try to write a client server program, where the server has a thread pool to handle all the requests coming in from multiple clients. You can use condition variables from the pthread library and this will help you understand the concepts of this lecture better with a practical example. So, thank you all, that is all I have for this lecture and let us continue this discussion in the next lecture.