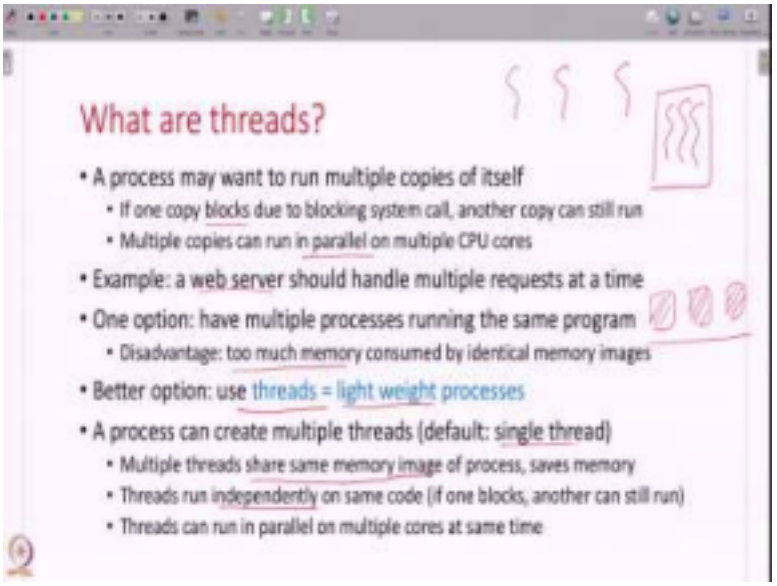
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture 08
Threads

Hello everyone, welcome to the eighth lecture in the course Design and Engineering of Computer Systems. In this lecture, we will continue our discussion of operating systems concepts. Specifically, we will be discussing the concepts of threads in this lecture. So, let us get started.

(Refer Slide Time: 00:32)



So, what are threads? So, we have seen what are processes. And sometimes it may so happen that a process may want to run multiple copies of itself on the same machine. Why? For example, consider something like a web server that is handling requests from multiple clients at the same time, then it may so happen that one of the web server processes has blocked because the user requested a file and it had to read the file from the disk and it made a blocking system call.

In that case, we want another copy of the web server process around to service other requests from other users. Similarly, you may want to have multiple copies of a process to run in parallel on multiple CPU cores. One process can only run on one core, if you had multiple copies, you can use the multiple CPU cores efficiently. So, there are many reasons why a process may want to have multiple copies of itself.

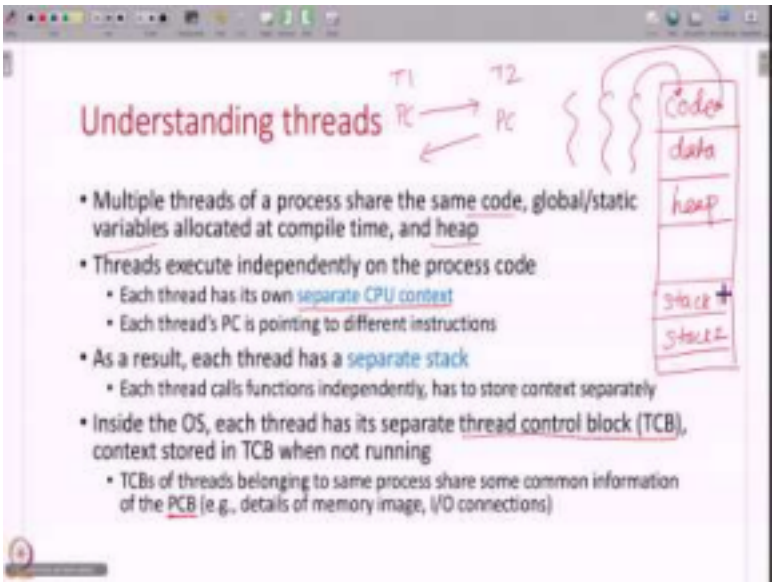
But if you simply just create multiple processes and run them on the same hardware, then

we are wasting memory. Every process the code, data, the memory image of the process, unnecessarily multiple copies of those are existing in RAM and they are occupying memory unnecessarily. We want to avoid this.

So, if you have such a situation where you want to run multiple versions of an application, but want to keep only one copy of the code in memory, then a better option to do that is to use what are called threads. So, you can think of threads as nothing but lightweight processes. A process can create multiple threads of execution and these multiple threads will run on the same memory image of the process, but they will run independently.

These multiple threads will share the same memory image but they will run independently like separate processes. So, if one of the threads blocks the other can run, and these threads can run in parallel on different CPU cores. So, therefore, it is useful for a process to have multiple threads. If you do not do anything, if you do not create threads, then the default is just a single-threaded process but most real-life programs and applications are typically multi-threaded for these reasons.

(Refer Slide Time: 02:56)



So, let us understand clearly what is a thread? Multiple threads of a process share the code, all the global static variables and heap, in your memory image, if you look at the memory image of a process, there is the code and there is the compiled time data and heap. All of these are common. If you have multiple threads, they all share all of these elements of the memory image of a process.

But these threads will execute independently on the process code. So, one thread could be

running some part of the code, another thread could be running at another location in the code. Each thread will run independently on the process code. Therefore, each thread will have its own separate CPU context.

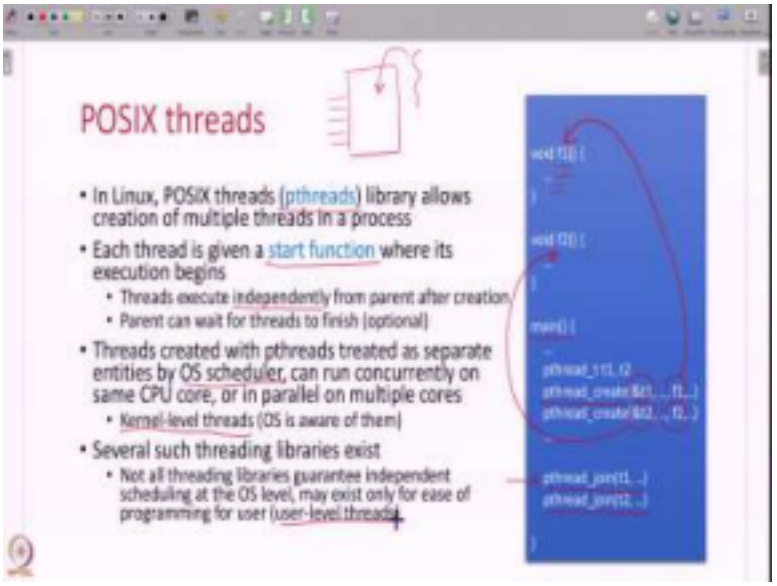
So, for thread one, the program counter could be pointing to one location. And thread two, the program counter could be pointing to some other location executing a different piece of code. So, each threads program counter registers, they will all be different because they are all running independently on the same process code.

So, as a result, because they execute independently, they will make function calls independently, they will push elements on the stack independently. As a result, each thread will have its own separate stack. So, the user space stacks are different for different threads, because each thread runs independently, calls functions independently and so on.

And inside the operating system, a thread also has its own separate data structure, which is usually referred to as the thread control block or TCB. This is like the PCB, but if a thread is blocked, its context will be stored in the TCB and so on. Information about a thread, the context of a thread and other things will be maintained in the TCB.

And these TCBs of a process has multiple threads all the TCBs will somehow be connected up with the PCB also. Because details like the memory image and everything are common. So, each thread does not have its own memory image. So, that information you will get from the common PCB. So, the summary that you have to understand is threads share a lot of code and data. But they have their own separate stacks, because they run independently and they have their own separate CPU contexts because they run independently.

(Refer Slide Time: 05:32)



So, how do you create threads in a program. So, there are many libraries available for different programming languages that let you create threads. In Linux, a very popular library is what is called pthreads, the stands for POSIX threads. So, this pthreads library allows you to create multiple threads and also provide various functions to deal with these threads.

So, we will just give you a very simple example of how pthreads works. But you are encouraged to look up the exact syntax of this library online. So, if you look at a simple program, normally a program will just have default one thread of execution. There is just one way in which you can go through a program code. But if you do something like this pthread

create, this creates a separate thread of execution in your program.

So, here is your program code, there is one thread that is running through the program code in a certain order. If you create a separate thread, that separate thread will execute a different part of the program's code. So, if you create a thread t1 over here, this thread is given a start function. And this thread t1 starts running the program from this point, it executes this function, it can make other function calls.

It can do other things on the memory image. Similarly, a thread t2 is given another start function f2, and this thread t2 starts executing the memory image from here. So, each thread starts at a different location and runs through the program's memory image, runs through the programs code independently and after they are created, the threads run independently from the parent.

And the parent can also wait for the threads to complete So, you have pthreads also provides

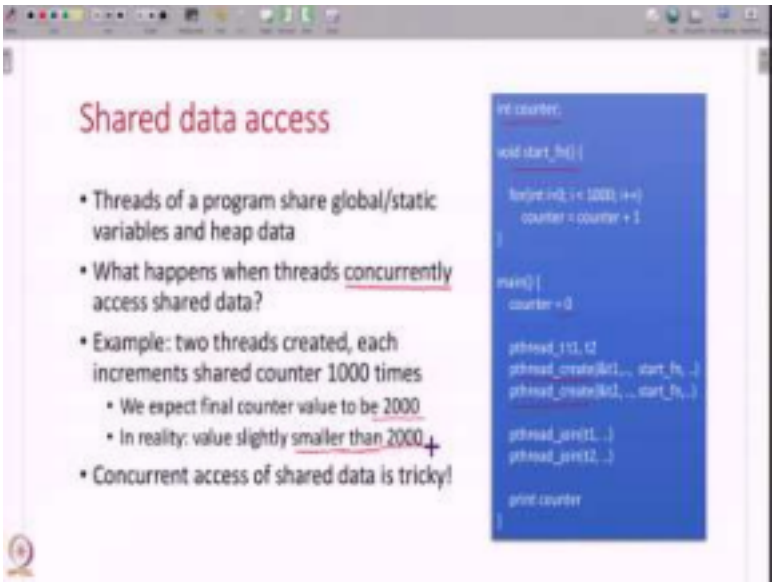
an API called join. So, if the parent execution. So, the threads are executing in their corresponding functions, the parent can wait for the threads to complete using this function pthread join. This is similar to the wait system call, though with threads, this is optional, it is not necessary for parent thread to wait for the threads it has spawned to finish.

So, the other thing to understand is that with pthreads, the separate threads that are created are treated as separate entities by the OS scheduler, that is if a process creates two threads, then this main process and the two different threads are scheduled independently by the CPU scheduler, they are treated as almost like separate processes that can run independently on the CPU scheduler.

So, these are called Kernel-level threads. Why, because the operating system is aware that these threads exist. But not all threading libraries give you this functionality. Some threading libraries will let you create a thread but the OS is not aware of it. And all these threads are scheduled as part of the same process, they are not scheduled independently. So, such threads are called user level threads.

Then why you might ask, why do we even have such threads? There are many reasons for ease of programming and so on. That threading libraries may provide you the abstraction of threads, but actually these threads do not run independently on the underlying CPU hardware. But with pthreads, these threads that you create are scheduled independently by the OS scheduler and therefore they are called Kernel-level threads.

(Refer Slide Time: 09:10)



So now, let us understand a little bit more about how threads share the memory of a program. So, threads of a program share all the code in the memory image as well as all the global variables, static variables, heap data, except for the data in the stack, everything else is shared between the threads of a program.

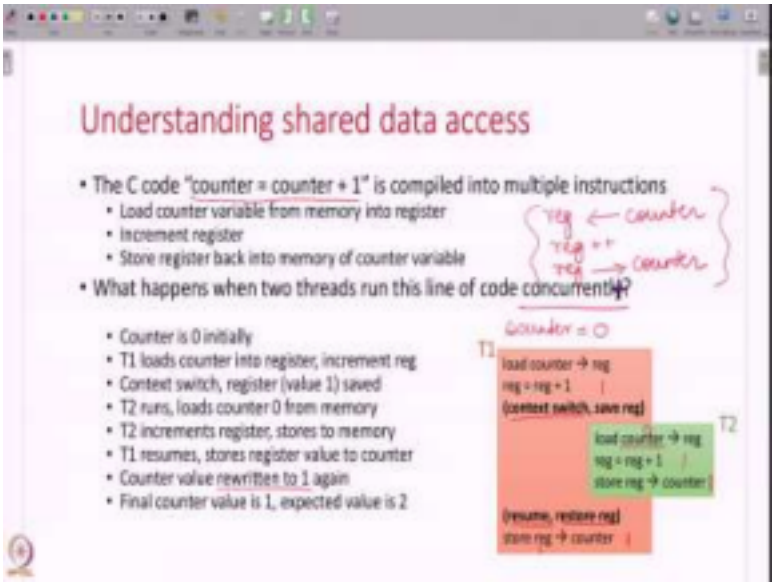
So, now let us understand when two threads simultaneously concurrently access the same data what happens, you might even wonder what will happen they will just access the data. But that is not the case. There are some challenges here. Let us understand that with an example. So, again, it is the same program, I am just giving you a small example using the pthread API.

Here is a program that has created two threads t1 and t2. And both these threads execute the same start function here. And what these threads are doing, there is a global counter, which is initialized to zero. And each of these threads executes the same function and terminates. And what is this function doing? It is incrementing this counter 1000 times.

So, the parent just executes, thread 1 runs this function increments counter 1000 times, thread 2 also increments counter 1000 times and both these threads as well as the parent process run independently, they can run simultaneously on different core, same core whatever, but by the time all of this finishes, what do you expect the final counter value to be? You expect it to be 2000. Two threads have incremented 1000 times each.

But if you write a simple program like this and actually run it, in reality, the value will not be 2000 it will be smaller than 2000. So, what is happening here suddenly with threads, what is the computer forgotten how to add two numbers? No, that is not the case. In fact, something very tricky is happening here. So, let us understand what are the challenges when two threads simultaneously access a shared data variable.

(Refer Slide Time: 11:13)



So, now, let us take a deep dive into this example. Both the threads incremented a counter, they did something like this counter equals counter plus 1, but this is just a C language code but when this is compiled, and actually in the memory image, it is in the form of CPU instructions. And this line of C code will actually be three different CPU instructions, for example, which is you will load this value of the counter from memory.

You will read it from memory, load it into a register, then you will increment the register. And then you will store back from the register into the counter. This is how this line of code is implemented in your compiled executable. Now, when two threads try to run these instructions concurrently, what happens?

Here is one possible scenario that can happen, of course, one thing that can happen is thread 1 fully runs increments the counter, then thread 2 runs fully increments the counter, then again, thread 1 runs again, thread 2 runs. If they run exactly one after the other, what will happen the counter will be incremented once twice, and so on and you will get your value of 2000 that we have seen in the previous slide.

But sometimes that may not happen like that, sometimes if the execution of the thread. So, this is the one thread execution, this is the other thread execution if they overlap like this, something weird will happen. What is happening here? Initially, your counter value is 0. And thread 1 first ran it loaded this counter into a register, then this register value is incremented.

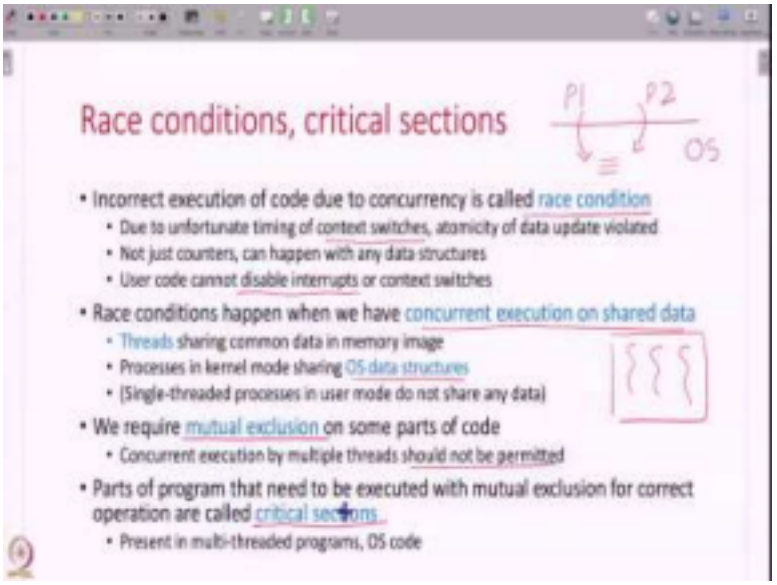
So, now your register has a value of 1 from 0 to one. At this point, a context which occurred, thread one has stopped, all its register values have been saved. And now thread 2 starts, you

have a concurrent execution. At this point when thread 2 starts what is the counter value? The counter value is still 0 because thread one has not started back yet, it has not completed.

So, now thread 2 once again begins with a counter value of 0, stores it in the register, increments the register to value 1 and writes back a counter value of 1 into memory, thread two is complete. Now, thread 1 runs again it resumes where it left off remember. So, now once again this register value is 1, it is written into this counter variable 1 and the counter value is once again we return to a value of 1.

So, what has happened here, two threads have run each has run the code to increment a counter at the end after incrementing a counter two times the value is still one. Note that this may not happen all the time, sometimes they will run one after the other and the increment will happen correctly. But sometimes the increment may not happen correctly, because of this concurrent execution.

(Refer Slide Time: 14:09)



So, such incorrect execution of code that happens due to concurrent execution that is called a race condition. The two threads are somehow racing each other and interleaving in their executions, if they run one after the other in an orderly fashion, there is no problem. But when they interleave like this, when they race each other like this, that is when sometimes bad things can happen. That is why it is called a race condition.

And this is due to this unfortunate timing of context switches, and you do not know where a context switch can happen. It can happen at a very inconvenient location. And note that the user code cannot disable interrupts or context switches and say that I do not want to have a

context switch anymore. This is not in the control of the user program. So, therefore this can happen at any point and it can happen with any data structures, not just counters.

Data structures, shared data structures can be updated incorrectly due to race conditions. So, when can race conditions happen? They can happen any time we have concurrent execution on shared data. For example, threads in a process that share the memory of the process, that is one place where you can have race conditions.

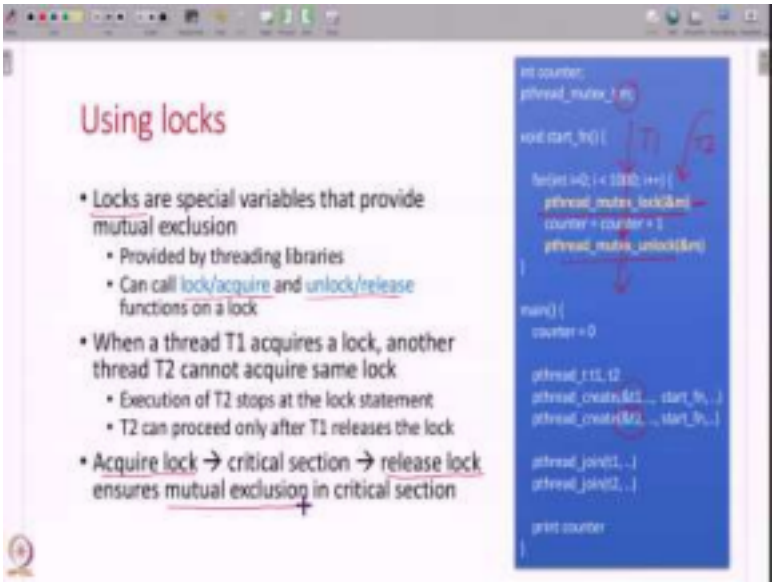
The other common example is when processes go to kernel mode, even single threaded process when they are in kernel mode and they share OS data structures. So, there is the common OS data structures and two different processes both go to kernel mode, say on different CPUs, and they try to access the OS data structures.

So, the operating system is shared the kernel mode execution is shared across all the user processes. So, in such cases also to processes in kernel mode accessing the kernel data structures can result in race conditions. Note that however, if you just have single threaded processes in user mode, they do not share anything. So, therefore, between two different processes, you may not have any race conditions.

And how do we fix these race conditions, we require a property called mutual exclusion. That is whenever any shared data has been accessed in some parts of the code; concurrent execution should not be permitted. We do not want concurrent execution at all and parts of the program that need this mutual exclusion are called critical sections.

So, some parts of multi-threaded program some parts of the OS code, they are called critical sections. Note that of course, not the entire multi-threaded program need not be a critical section. Sometimes if the two threads are working on different pieces of data, they can run concurrently, they can run in parallel whatever it does not matter.

But only when they are accessing shared data, common data that is when we need mutual exclusion, and such parts of the code that have access to shared data structures, they are called critical sections. And inside a critical section, we need mutual exclusion; we need only one thread at a time or one process at a time accessing these critical sections.
(Refer Slide Time: 17:26)



So how do we ensure this, we ensure this using a mechanism called locking. So, this is intuitive. So, if you do not want people to enter a room at the same time, what do you do you put a lock and the person who enters the room closes the door locks the door and when that person comes out, the next person can enter, that is a simple concept that we all know. Similarly for threads also we have a mechanism called locks.

So, various threading libraries will provide these locks to you. For example, pthread also provides locks. So, locks are nothing but they are just special variables that one thread can acquire the lock or lock the lock and the thread when it is done can unlock the lock. And as long as a thread has the lock no other thread can execute the critical section.

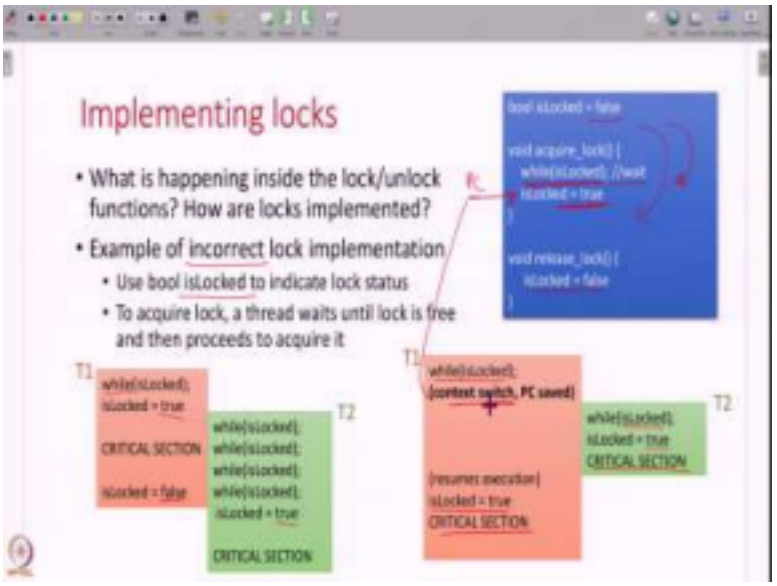
For example, let us see this example of two threads being created t1 and t2 and both these threads run this function. So, now suppose thread t1 starts running this function, it acquires a lock you have a pthread mutex lock variable that is there and you can acquire or lock this lock. So, once thread t1 locks this lock and you know proceeds to the critical section.

At this point if another thread t2 also tries to run the same piece of code t2 will block here, t2 cannot proceed here because it does not have the lock and once t1 finishes and releases the lock at this point then t2 can enter, after t1 has released the lock then t2 will be able to acquire the lock and enter the critical section.

So, any thread that when entering a critical section before entering a critical section, it will acquire the lock after the critical section it will release the lock, this kind of programming will ensure mutual exclusion in the critical section. And this is a very important thing to

keep in mind when running multi-threaded programs or when accessing operating system code in kernel mode and so on.

(Refer Slide Time: 19:36)



So, how are these locks implemented? We have seen how locks are used. Before entering a critical section, you acquire a lock, after critical section you release a lock. But how is this lock implemented, say by the pthread library how do you implement the lock? So, this is not easy. This is quite tricky. So, let us see why. So, here I have tried to show you a very simple implementation of a lock.

What I am doing is I am using a simple Boolean variable to indicate the lock status. If nobody has taken the lock, this Boolean variable is false. And a thread that tries to acquire this lock, what it will do is it will check is this lock acquired by somebody else is locked? Is this variable true? While it is true what do you do?

This is a busy while loop, it does nothing, you keep going back while, while, is it locked, is it locked, is it locked, you keep on checking, you just keep spinning in this while statement doing nothing, you just wait here. And if nobody has this lock, if this is locked is false, nobody has this lock, then there is no need to wait.

You immediately enter the lock function, you set this lock to true, so that nobody else can take this lock after you and then you go away. Now you have acquired the lock, you have checked that nobody else has the lock, you have acquired it, you have set it to true. And now whichever thread completes, this code has the lock with itself.

Now until this thread releases the lock again sets it to false again, any other thread that starts will basically get stuck here. Any other thread that tries to acquire the lock will keep on waiting here. That is the idea of this simple implementation of a lock using a simple Boolean variable or you can use any other integer or whatever you want.

So, this seems to work, for example, this thread t1 is the first thread trying to acquire a lock. It does not spin here because there is while returns false the lock is free, then it sets the lock to true, it enters the critical section all good. Now while it is in the critical section, another thread that comes is just simply spinning here, it is checking, locked is true.

So, while true, while true, while true, it is just in this while loop all the time. Once the lock is released, then this thread this while stops executing. While returns false, the condition returns false. Now, after thread 1 has released the lock thread 2 has acquired the lock and it enters the critical section.

In this way they are entering the critical section one after the other. So, we are happy, then why are we calling this an incorrect lock implementation. Because once again, bad things can happen if context switches happen at unfortunate locations, which we have no control over. For example, this thread, look at this sequence of execution. Thread t1 first checks that is the lock free, the lock is free, it is the first thread coming in, lock is free.

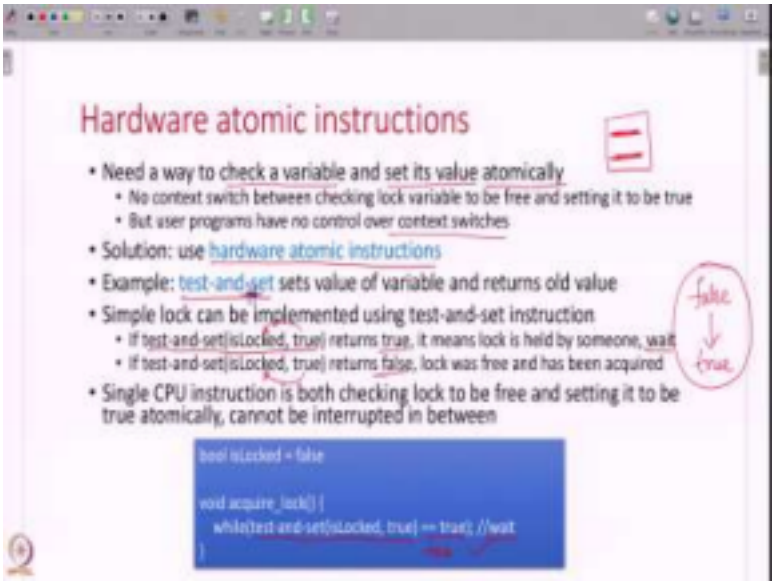
At this point, it has decided that the lock is free and the program counter has moved on to the next slide. It has decided to acquire the lock, it has moved there. At this point before it can set this to true a context switch has unfortunately happened, the program counter value saved, all the context is saved and t1 stops execution at this point t2 runs. Now, t1 has not yet set the lock variable.

So, now t2 checks lock is free, it will set it to true, it will enter the critical section. Now at a later point of time t1 resumes execution, the program counter is here, it is going to run this instruction it will not go back. Why should it go back because it executed that while loop already. So, it will resume from here. It will once again set the lock to true. Lock the lock, enter the critical section.

Now what has happened we have two threads both running inside the critical section. We try to implement a lock but we did not achieve mutual exclusion. So, therefore, it is the same problem, if you understand it is the same problem as the previous example, we needed locks

to avoid these unfortunate contexts switches but what if these contexts switches happened in the implementation of the lock? It is the same problem. We are not solving the problem at all here. So, how do we solve this problem?

(Refer Slide Time: 23:56)



We have some special hardware support in modern systems to solve this problem. We need a way to check a variable as well as set its value. And do both these steps atomically that is in one block, I want to check is the lock free. If it is free, I want to lock it. I want to set it to true. And in between these two steps. I do not want to break. I do not want to check a lock. I think it is free, not set it and somebody else then comes and takes the lock.

I do not want that. So, I want these two instructions to happen atomically. But how do I do that? As I said before, user programs have no control over context switches. A user program cannot say Oh, please do not context switch me out for some time. That is a huge security violation. What if process just says that suppose there is a way a system called to tell a process that no context switches and then a process can just do that and completely take over the CPU.

It can just disable all context switches at any point of time. That is a very bad thing. It starves the other processes in the system. It violates isolation. So, we do not want to ever give user programs that control saying all context switches are disabled for you for some time. So, then how do we solve this problem in a context switch can happen and break this atomicity.

The solution is to use what are called hardware atomic instructions. So, these are instructions that do two things at the same time in one CPU instruction itself. So, we have what is called a test and set instruction. In one instruction itself, you can set the value of a variable, and you can check it is old value.

And because it is one hardware instruction, one instruction obviously happens atomically, you cannot have a context switch in between a CPU instruction, you can only have context switch between two CPU instructions, but not in the middle of one CPU instruction. So, if you do both these steps in one CPU instruction itself in what are called atomic instructions, then your problem is solved.

So, for example, here is an example of a lock implementation using test and set. Instead of directly just writing using C language code to set this lock variable. I am using this test and set instruction to set the lock variable to true. So, what does test and set return? If you say test and set isLocked true, it will write the value of true into this variable, and it will return the old value.

And if the old value returned is true, what does this mean, this lock was already held by somebody else, locked was true, it is held by somebody else, and you simply try to write true value again into it, it means you have not acquired the lock, the lock is with somebody else, and you have to wait. So, if test and set of this lock variable, if you try to set it to true and the old value returned is also true, it means you do not have the lock, you will wait.

On the other hand, if you set a value of true, but the old value returned is false. That means this variable was false, nobody else had the lock and you have managed to set it to true that means you have acquired the lock, you own this lock now, the lock was free, and you have acquired it and this whole transition checking if it was free or not and acquiring the lock happened in one step.

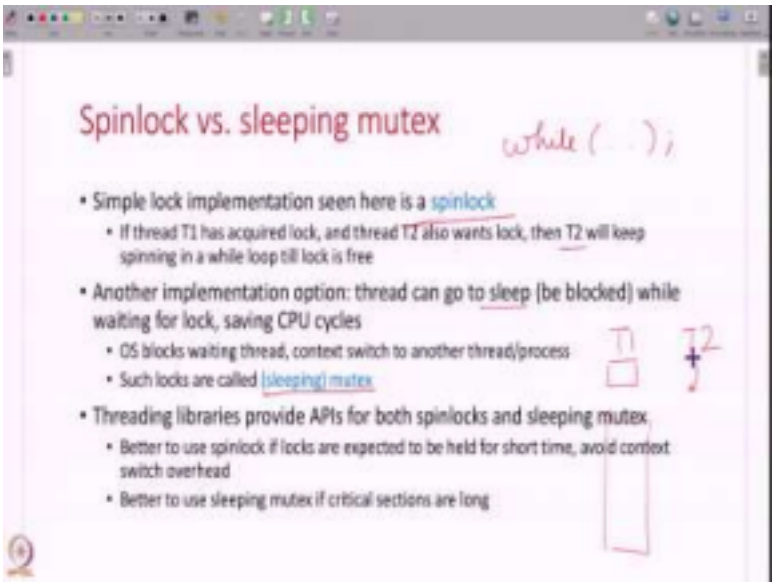
And if test and set, this returns false, it means that the lock is free, you have acquired it, and then you can break from this while loop. So, this is the code to acquire a lock using a hardware atomic instruction. And as you can see, this is a single CPU instruction, therefore there is no way in which you break atomicity here, there is no way where you check the lock to be free.

And then you forget to set it you come back later to set it incorrectly all of that is not going

to happen here, you cannot be interrupted. So, all modern threading libraries provide locking primitives. And all of these locking primitives are implemented using hardware atomic instructions. So, before CPUs supported such instruction so this is a complicated instruction. This is not your regular load, store add kind of instruction, it is a special complicated CPU instruction.

Before CPU supported these kinds of instructions, a lot of work was done on how to write software algorithms that can do locking. But all of those are not very easy to do, as we have seen in the previous slide. And therefore, we use today all lock implementations use these hardware atomic instructions.

(Refer Slide Time: 28:44)



So now in the previous examples of locks that I have seen, if you do not get a lock, if a thread does not acquire a lock, what does it do, it simply keeps spinning in a while loop. So, while if the lock is not free, you just do nothing you keep spinning. So, such lock implementations are called spin locks. That is if t1 has acquired the lock, t2 also calls the acquire function, it will keep on busy spinning while, while, while, while.

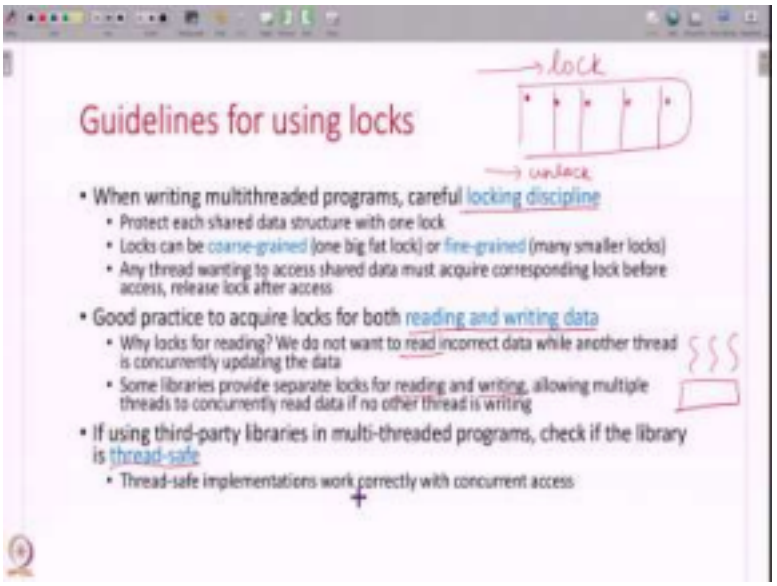
It will keep on checking this condition in a while loop. And this is a spin lock. But you can see that this is not very efficient. Why is this thread even running when the lock is not free? Why is it wasting CPU cycles for a long time? So, a better implementation can be a thread that realizes that the lock is not free, can just go to sleep, can be blocked. And when the log becomes free, it can run again. It is context can be saved and it can restart.

So, such implementations are also possible for locks. Such locks are called mutexes or sleeping mutexes. You can either have a spin lock or you can have a sleeping mutex. So, whenever you use any threading library, and there is a lock API provided, you can check in the description of the library, does it implement it as a spin lock or does it implement it as a sleeping mutex.

And both these are valid implementations. If a thread t1 has a lock, the thread t2 can either busily spin or go to sleep, whatever it is, both of these are correct implementations of mutual exclusion, but they just differ in terms of efficiency. And there are cases where both can be used. For example, if t1 will hold the lock only for a very short period of time, then t2 can just wait, busily spin and then get the lock. A spin lock is useful.

But on the other hand, if t1 will hold the lock for a very long time, then it is better for t2 to go to sleep be blocked, let some other process run on the CPU and come back again later. So, it depends on your application, how long your critical sections are based on that you can choose either a spin lock or a sleeping mutex for your locking variables.

(Refer Slide Time: 30:54)



So, there are some guidelines for using locks, which is, as we have seen the reasons before, whenever you write multi-threaded programs with shared data structures and critical sections, you must follow locking discipline, every shared data structure should be protected by a lock. And it is up to you, the programmer to ensure that you acquire the lock before access and release the lock after access.

And these locks can be coarse grained or fine grain. What does this mean? Suppose you have a big array. You can protect this whole array by one big lock, saying acquire this lock, access any element of the array, and then release the lock, you can do that. Or you can have individual locks for each of these array elements.
I want to access this array element, so I will acquire this lock access this element release this lock, you can do fine-grained locking or coarse-grained locking, it depends on your underlying software architecture. Coarse-grained locking is easy to do, you do not have to think which lock what, you just have one big fat lock. But on the other hand, it is inefficient; it does not allow parallel concurrent execution.

So, there are trade-offs here. And this is an aspect of application design that the programmer must think about. And also are locks only for reading or writing? So, it is a good habit to acquire locks both for reading as well as writing data. So, you might say what is the harm in multiple threads reading the same variable; all of this issues are happening only if you are trying to modify a variable.

When we are incrementing a counter, a problem is happening. But if I am reading a counter, there is no issue. So, two threads can read the counter. Why do I need locks? You need locks because while you are reading somebody else might be writing. And therefore, you do not know. Unless you take a lock, you cannot guarantee that nobody else is also updating the data at the same time.

Therefore, it is good practice to take a lock whether for reading or writing shared data. But if you think taking a lock for reading is inefficient, a lot of libraries provide separate locks for reading and writing. So, if multiple threads just want to read a common variable, they will be allowed if they say we want a lock, only for reading. But if another thread wants to write, then this reading will not be allowed.

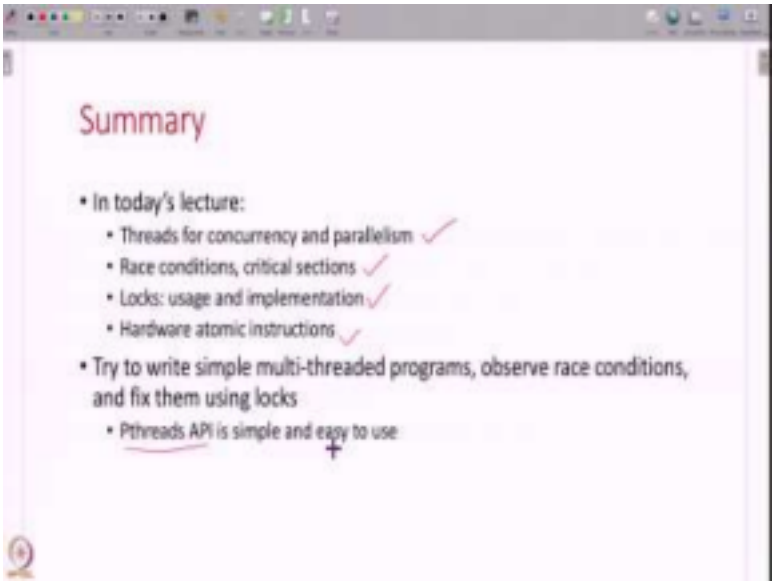
So, you have threading libraries that provide separate locks for reading as well as writing. And if you are using any other third-party libraries, implementations of other data structures in your multi-threaded programs, you should also check that all those other libraries are thread safe.

That is, if you are in your program, you have locks but if you use another implementation of say, a hash table or a queue or something, and inside that library, they are not using locks, then you are still in trouble. Your multiple threads will access that shared data and get into

trouble. So, therefore, whenever you are using a library, you should check if the library's thread safe.

That is if the libraries implementation will work correctly when multiple threads access the library concurrently. So, these are all the things to keep in mind as a programmer, when you are writing multi-threaded programs.

(Refer Slide Time: 34:17)



So, that is all I have for today's lecture. To summarize, we have studied what are threads and why they are needed for concurrency and parallelism. But using threads is not easy. You have race conditions; you have critical sections that have to be protected by mutual exclusion. And locks is a mechanism to do that. We have seen how to use locks as well as how to implement locks.

And we have studied the concept of hardware atomic instructions, which are useful for implementing locks. So, as a simple programming exercise for you, please try to write a simple multi-threaded program. You have many tutorials available online. The pthreads API is particularly simple and easy to use.

Try to write a program that has multiple threads and all of these threads are updating a shared counter like the example we have seen, try to observe race conditions in practice and try to fix them using locks. This is a small exercise that will give you a hand on feel for whatever we have studied in today's lecture. Thank you all and we will continue our discussion in the next lecture.