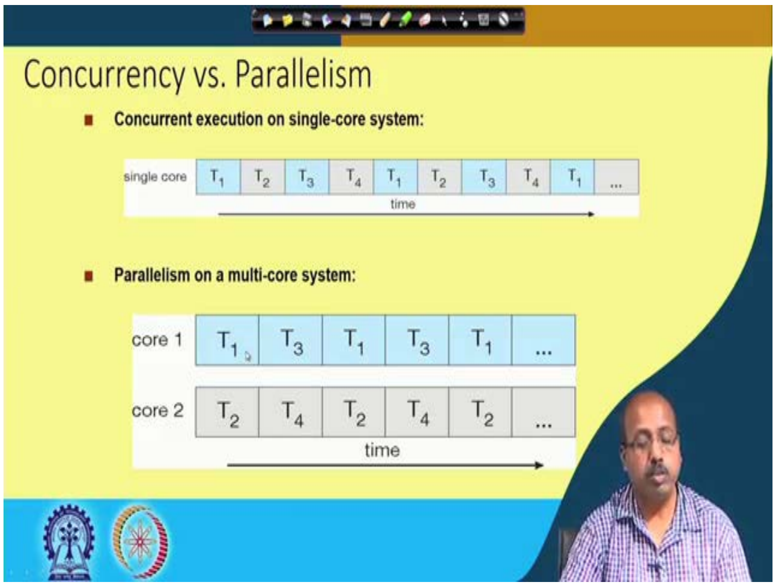
Operating System Fundamentals

Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture - 21
Threads (Contd.)

(Refer Slide Time: 00:29)



Concurrency vs. Parallelism

- Concurrent execution on single-core system:

single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

time →

- Parallelism on a multi-core system:

core 1	T_1	T_3	T_1	T_3	T_1	...
core 2	T_2	T_4	T_2	T_4	T_2	...

time →

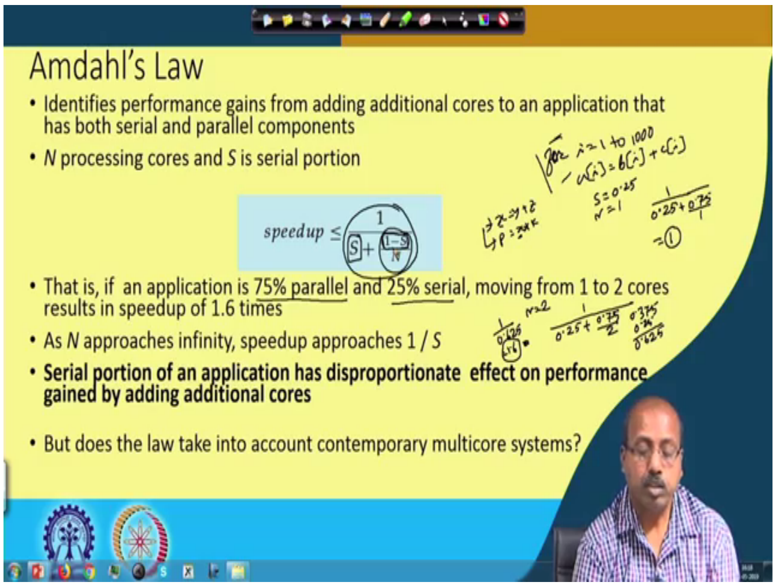
So, here we have got a typical example of this concurrency versus parallelism. So, if you have got a single core system, then this tasks T 1 T 2 T 3 and T 4. So, these are various tasks that we have in the system and over the time the scheduler schedules the tasks in this fashion that first it schedules T 1 for some time quantum, after that it schedules T 2 for some time, then T 3 for some time, then T 4 for sometime then again it picks up T 1 for execution. So, this type of execution we will see later. So, this is known as round robin type of scheduling. So, each task is given CPU for some small amount of time.

So, this is a concurrent situation because all the tasks so, they are progressing. So, if we look into the task at some point of time. So, all the tasks so, you will see that T 1 has progressed to some extent, T 2 has also progressed to some extent, T 3 also progressed to some extent so, like that may be the amount of progress is not uniform across all the tasks, but all of them are progressing. It is not that T 1 is a executing and all of them have not. So, T 1 is just finished and none of the others they could executive in some small fraction. So, that will not be the situation. So this way the schedule or it can be used to have concurrency in a single core system; on a multi core system so, we have got say 2 cores core 1 and core 2, now the job of the scheduler is more complex. So, it has to decide not only with job to execute next, but on to which core. So there are issues in this multi core scheduler design.

So, for a typical case may be like this that in core 1 it first it on core 1 it puts the odd number jobs and on core 2 it puts the even number job. So, this is a totally hypothetical decision that the scheduler has taken. So, this T 1 is run for sometime then it is given to T 3, then T 3 runs for sometime then again given to T 1 so, it goes like this and similarly for core 2, first T 2 runs for sometime then T 4 for some time. Now further it is a good or bad distribution of this task 2 core so, that is that can only be answered when we see the total finish time for all the tasks which constitute the whole application for this.

So, it may so, happened that we need to take a much better decision and this may be turn out to be a very bad decision because T 2 and T 4 those tasks may be quite small compared to T 1 and T 3. So, as a result this second core will remain idle for a good amount of time and that has to be reduced. So, if we are trying to optimize the performance. So, this parallelism we get parallelism in here because this T 1 and T 2 they are progressing parallel T 3, T 4 they are progressing parallel and we also have concurrency because at any point of time all the tasks. So, they have executed up to some point. So, that way you have got both parallelism and concurrency in multi core system.

(Refer Slide Time: 03:33)



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *N* processing cores and *S* is serial portion

$$speedup \leq \frac{1}{\boxed{S} + \frac{\boxed{1-S}}{N}}$$

- That is, if an application is 75% parallel and 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*
- **Serial portion of an application has disproportionate effect on performance gained by adding additional cores**
- But does the law take into account contemporary multicore systems?

Now, if we increase the number of cores then how does this performance improve ok. So, this is so, we ideally we would like to have a situation like if I instead of using 1 core. So, if I use 2 cores then my performance so, throughput should be and double of the previous 1 because per unit time I can do 2 jobs which can be which is possible by a single processor. So, that way that is the expectation, but unfortunately if you look into any program then entire part of it cannot be done parallel, because there will be some part of it which is inherently sequential. So, you cannot do any parallelization of that and there is some portion which are which can be paralyzed.

So, only the portions which can paralyzed so, they can be given to this parallel jobs, parallel cores and that way you can get performance improvement. So, this is the Amdahl's law, it says that identifies performance gains from adding additional cores to an application that has both serial and parallel components. So, if job with a task can have an application can have both serial and parallel components because initially it may be doing some operations and only when it comes to some sort of say array up dation and all for i equal to 1 to 1000. So, ai equal to bi plus ci.

So, something like this. So, then I have got parallelism, but also as long as it is doing operations like say x equal to y plus z then p equal to x into k so, like that. So, I cannot have parallelism because this is the x value. So, these two statements cannot be executed in parallel because this x is just computed here. So, I cannot do them in parallel. So this way some part of the core inherently sequential say you cannot do parallelization and there are some parts of the cores of which can be very easily parallelized. So, this Amdahl's law says that if I have got N processing cores and S is the serial portion then the speed up will be less or equal 1 upon S plus 1 minus S into N. So, for example, suppose we have got an application where 75 percent is parallel and 25 percent is serial. Now if I have got a single core then I have got S so, S equal to 75 percent is parallel 25 percent is parallel. So, S equal to 0.25 and this for a single processor system N is equal to 1.

So, this expression becomes 1 upon 0.25 plus 1 minus S that is 0.75 divided by 1 N equal to 1. So, this gives me equal to 1 the speed up is equal to 1. So if I have got single processor system then with respect to single processor system there is no improvement now suppose I put 2 cores. So, N is equal to 2. Now what will happen? So, this is this expression value becomes 1 by 0.25 plus. So, 0.75 divided by 2 and then so, this value. So, it is 0.75 by 2 so, this is 0.375. So, 0.375 plus 0.25 so, that will make 0.625.

So, 1 upon 0.625 so, this is about 1.6. So, this is equal to 1.6. So, what happens is that so; that means, if you go from 1 processor to 2 processor and the core is such that it is 75 percent parallel and 25 percent serial ok. So, then you can get a speed improvement of 1.6 only. Now you see if you look into this expression carefully. So, it says that so, this is the serial portion. So, this I cannot do anything here and this 1 minus S is the parallel portion. So, if you are having more number of processors. So, this time will reduce by a factor of N theoretically of course. So, this is not the practical thing. So, this is theoretical because practically there will be time for communication, there will be time for data distribution and all that. So, they are not taking care of here.

So, assuming that everything those timings are 0; so, this is S plus 1 minus S by N. So, this part gives me the speed up. So, this so, to look into this possible values of this S of the speed up, when N approaches infinity then this term is almost 0. So, the speed up becomes 1 by S. So, if you have got infinitely much number of processors then the speed up will be given by 1 by S. So, as you can see that the serial portion of an application has disproportionate effect on the performance gain by adding additional cores; so, if you add additional cores.

So, even if I add infinite number of cores also then the speed up will be 1 by S. So, if the sequential portion is more than naturally speed up will be less. So, if a program is 100 percent sequential program then even if you put say infinite number of cores the speed up will remain 1 only. So, that is the idea of these Amdahl’s law. So, how does this law take into account the contemporary multi core systems? So, multi contemporary multi core systems are much more complex. So, naturally this is a theoretical study and this may not be able to give true picture there, but this gives the theoretical upper limit on the degree of speed of that you can achieve.

(Refer Slide Time: 09:49)

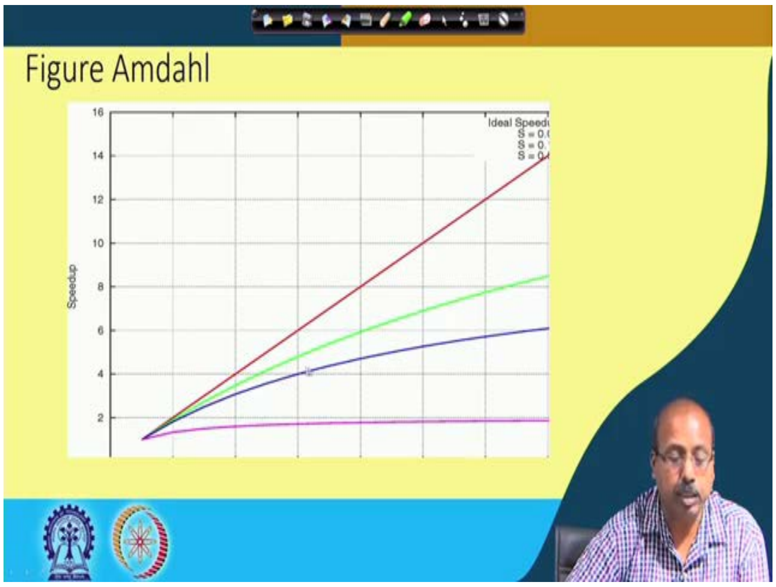
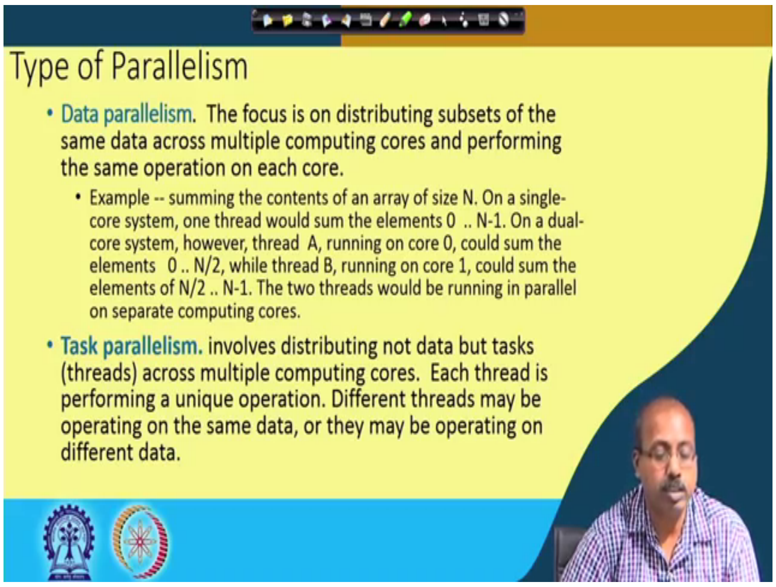


Figure Amdahl

So, next we will have a look at a graph that shows that ok. So, this is the ideal situation. So, as you are increasing number of processors, then the speed up should grow like this. So, this redline so, with 1 the speed up is 1 with 2 the speed up should be 2 with 4 the speed up should be 4.

So, it should be grow like this. On the other hand ideally what happens is as the sequential portion increases. So, this graph is not coming fully. So, as the sequential portion increases this speed up starts decreasing ok. So, these are as you go down the speed up the sequential portion is more as a result this improvement becomes poor. So, that is the graphical interpretation of this Amdahl’s law.

(Refer Slide Time: 10:34)



Type of Parallelism

- Data parallelism. The focus is on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
  - Example -- summing the contents of an array of size N. On a single-core system, one thread would sum the elements 0 .. N-1. On a dual-core system, however, thread A, running on core 0, could sum the elements 0 .. N/2, while thread B, running on core 1, could sum the elements of N/2 .. N-1. The two threads would be running in parallel on separate computing cores.
- Task parallelism. involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

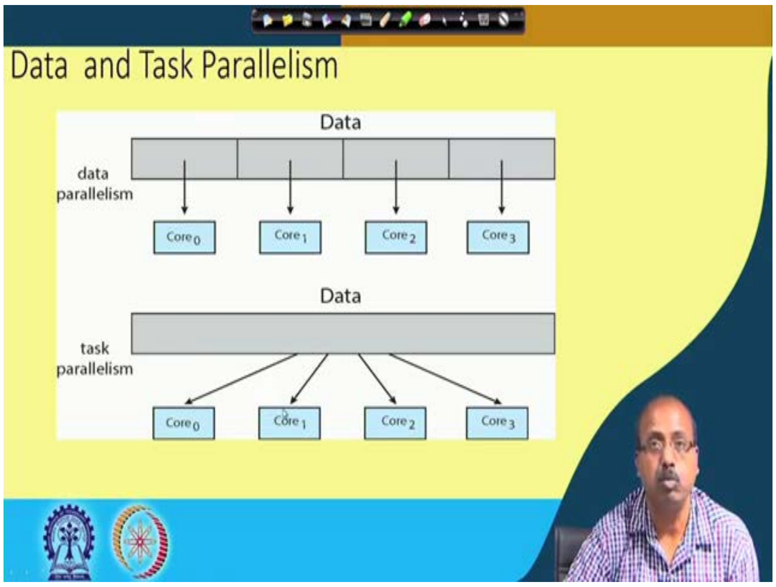
Next looking into the types of parallelism that we have so, we can have data parallelism. So, data parallelism the focus is on distributing subsets of same data across multiple computing cores and performing the same operation on each core. So, the same data is the data has to be distributed. So, the same operation will be done on the different data. For example, if you have to sum the contents of an array of size N, then on a single core system we can have one thread to at to that would sum the element 0 to N minus 1.

So, assuming that array indices run from 0 to N minus 1 so, on a single core system I can have one thread that does the addition for all this N entries in the array. On a dual core system we can have thread A running on core 0 that could sum 0 to n by 2 while thread B running on core 1 could sum N by 2 n minus 1 and then after that I can do the addition of these 2 partial sums by using again maybe core A or core B either of them. The two threads would be running in parallel on separate computing cores. So, that way the once I have distributed the data. So, 0 to N by 2 to core 1 and. So, 0 to sorry this should be N by 2 minus 1 this is not N by 2; so, 0 to N by 2 minus 1 to core A and N by 2 to N minus 1 to core B.

Once we have done the distribution after that, I give the same instruction for both the cores telling that you add all the numbers that you have. So, after that they come up with the partial sums and then using another addition. So, I have to do that. So, that way we have got this thread level data parallelism. And, the other hand we can have task parallelism also that involves distributing not data tasks not data, but tasks across the multiple computing cores. Each thread is performing a unique operation and different threads maybe on operating the maybe operating on the same data or they may be operating on different data. So, different tasks are different to given to different threads so, that is that is the task level parallelism.

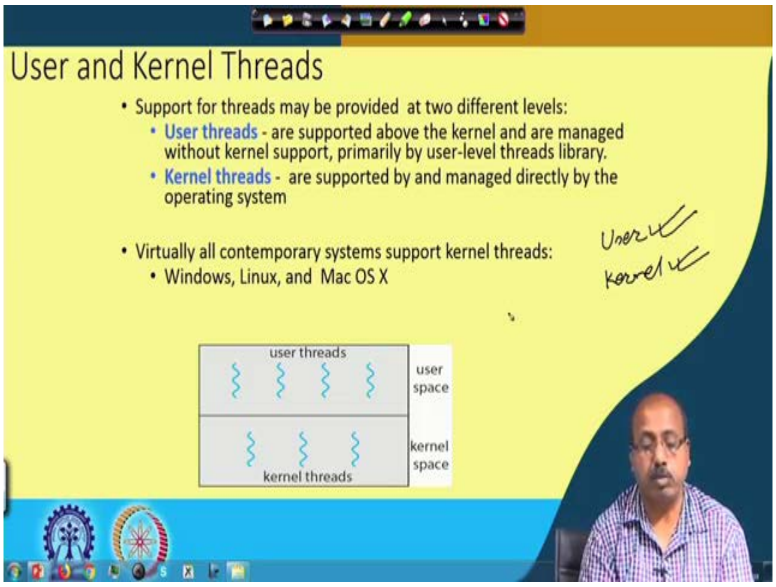
So, this we have like previously I was telling for the same matrix you may be interested to find the transpose of the matrix, inverse of the matrix and determinant of the matrix. So, like that so, they can so, I can defined the same matrix can be given to a number process number of tasks. So, we can and each tasks may be doing one of these operations. So, that way we can have task level parallelism there.

(Refer Slide Time: 13:32)


Data and Task Parallelism

Next so, this is the situation. So, in a data level parallelism so, we have got this the data is divided into portions and each core is given the part of the data. So, core 0 is doing for this part, core 1 is doing for this part. So, the instruction is same across all the cores or the task level parallelism so, data may be the same data is given to individual all the cores or may be different datas I get different data are given to different cores. So, both of them are possible, but the operations that they are doing are all separate. So, that way we have got this task level parallelism. So, this individual cores so, they are doing different tasks, but the data may be same or data maybe different. So, that is the task level parallelism.

(Refer Slide Time: 14:17)



User and Kernel Threads

- Support for threads may be provided at two different levels:
  - User threads - are supported above the kernel and are managed without kernel support, primarily by user-level threads library.
  - Kernel threads - are supported by and managed directly by the operating system
- Virtually all contemporary systems support kernel threads:
  - Windows, Linux, and Mac OS X

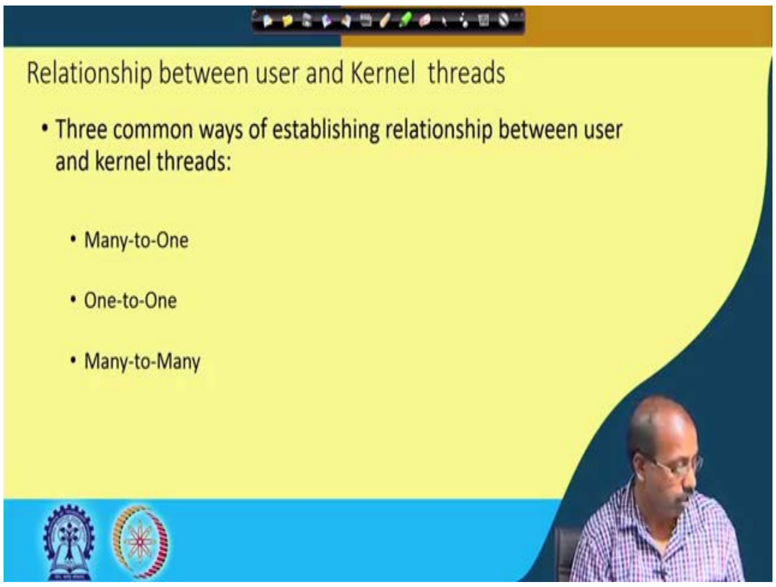
Now, coming back to the discussion on operating system; so, we can we can think about the operating system when it is doing executing the processes. So, we have seen that the process may have multiple threads. Now, we also know that when a process is executing. So, there are 2 modes of execution for some time it executes in the user mode and sometimes it executes in the kernel mode. So, normal computational jobs when it is doing. So, it is executing the user mode whenever it is doing a system calls. So, it is going to the kernel mode. Now so, at both the user level and kernel level we can think about multiple tasks that can go simultaneously. So, maybe at user level I have got say 3 different tasks that that should go parallely.

So, I can have three different user level threads that takes care of that. Similarly at kernel level also I can have multiple such tasks going on parallely. So, I can think about kernel level threads also. So, this support for threads maybe provided at both the user level and kernel level. So, we have got user threads that are supported above the kernel and are manage without kernel support primarily by user level threads library. So, we have at user level threads libraries are there.

So, using those library calls so, we create different threads and those threads are they execute in parallel, but they are not making any system core. So, none of them are going to kernel mode of execution on the other hand this kernel threads they are supported by n manage directly by the operating systems. So, as I operating system designer; so, we have to we have to provide this kernel threads and we have to tell interface for those kernel threads like whenever a system call is made. So, we do not need to tell the user like how this system call is going to be mapped onto threads at the kernel level, but the interface like how this system call would be made.

So, up to that much is sufficient, but at the implementation level the operating system designer may think that I will have multiple threads in the kernel, and when a particular system call is being called so, how this is going to be mapped onto the kernel level threads there. So, at user level; so, it is handled by thread libraries, at kernel level it is handled by the operating system. So, virtually all contemporary systems support kernel thread. So, we have got Windows, Linux, Mac OS X. So, all these operating systems so, they will support kernel level threads.

(Refer Slide Time: 17:00)

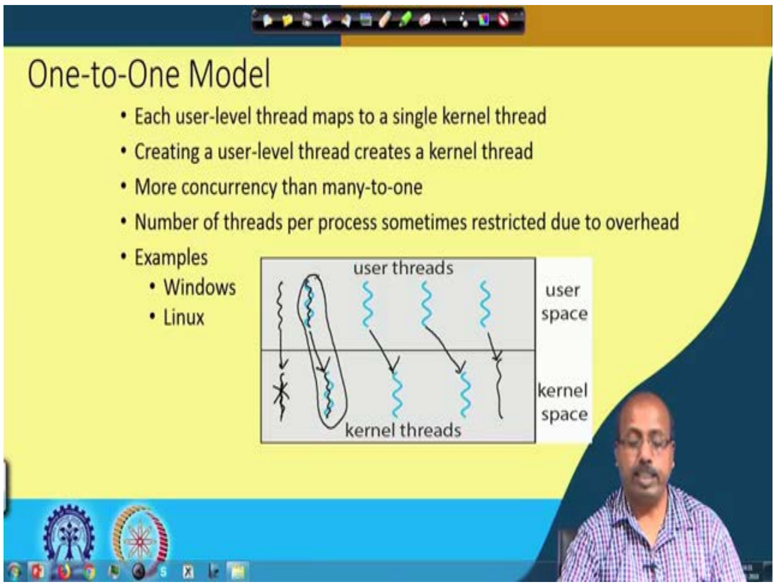


Relationship between user and Kernel threads

- Three common ways of establishing relationship between user and kernel threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

So, we will see how these things are actually taking place. So, there can be different types of relationship between the user and kernel threads there can are there are 3 different 3 very common ways of establishing relationship like many to 1, one-to-one and many to many. So, between the user level threads and kernel level threads how this mapping will be or the relationship will be there. So, just like many other say type of mapping that we have in different domains. So, how is it taking place in case of threads? So, that is so, that will be that will see and the first 1 is the most simple one is the one-to-one model.

(Refer Slide Time: 17:37)



One-to-One Model

- Each user-level thread maps to a single kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux

So, in one-to-one model what happens is that, each user level thread maps into a single kernel thread. So, so creating a user level thread also creates a kernel thread. So, whenever a user thread is created suppose a new thread is created here. So, accordingly a new kernel thread will also be created and this thread will be mapped onto this, similarly this thread is mapped onto this, this thread is mapped onto this, this thread is similarly there should be another thread on to it this will be mapped. So, whenever you create a new thread in the user space, a new thread is created in the kernel mode also.

So, each user level thread maps to a single kernel thread is a creating a user level thread creates a kernel thread as well. So, more concurrency than many to 1 because what happens is that if this thread makes the system call, then this thread will be taking care of that or if this thread makes a system call. So, this thread will take care of that. So, for every user level thread I have got a kernel level thread, which will be taking care of that. So, whether it is good or bad. So, it is good because I do not I do not be bothered about whether some other thread got blocked or not. So, if this thread got blocked here it cannot proceed. So, other threads will not suffer because of that because they have got their own thread. So, it is restricted to that particular operation only or that particular thread only. So, this whole thread may be blocked, but that will not make others to wait.

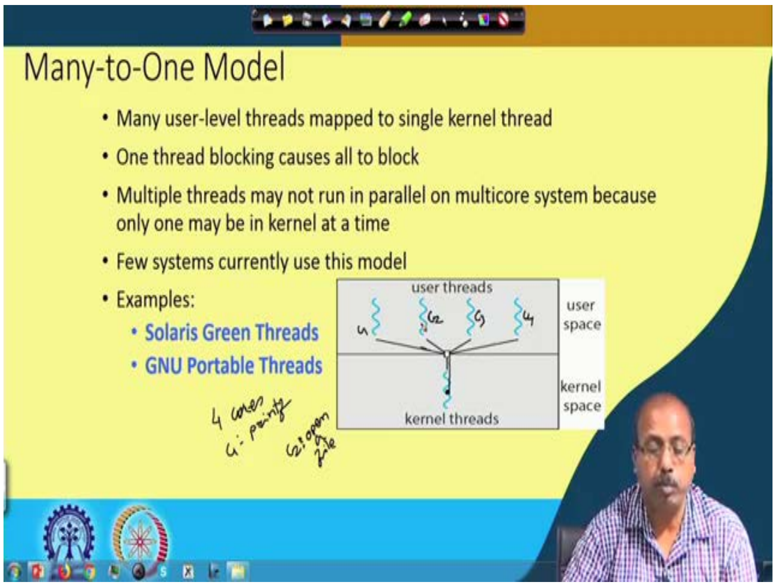
So, this is better in the sense that we have got more concurrency than many to one, but the difficulties that we are creating large number of thread. So, as and when a user thread is created one kernel thread is also getting created. So, that way if the user program does not make a good number of system calls, then this kernel level threads. So, they will be mostly unutilized ok. So, that way it is not a very wise decision to create so, many kernel so, many kernel level threads and that the system resources are blocked because of that.

So, number of. So, we have got more concurrency, but number of threads per process has to be restricted like; one process may go on creating threads user level threads and as a result for each of them it will create some kernel level threads also and since the program that we have. So, program is written by some user who may or may not be very good in this thread level programming may be by mistake the user creates an infinite loop and in that infinite loop it creates some threads.

So, as a result at the kernel level also infinite number of threads will get created. So, that is a very dangerous situation for the system. So, operating system designers were they do? They often put an over restriction and on the number of threads kernel level threads that you can create or number of threads that you can create per process. So, this to today stick this overhead. So, typical examples are windows and Linux. So, they have got this one-to-one module.

So, whenever user level thread is created a kernel level thread is also created and there is an upper bound on the number of threads that you can create in a process.

(Refer Slide Time: 21:06)



Many-to-One Model

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

So, that is one-to-one model then there can be many-to-one model; so, many-to-one model. So, many user level threads mapped to a single kernel thread. So, we have got a single kernel thread here so, many of these user level threads. So, they are mapped here similarly that can be another set of user level thread. So, they are map to another kernel thread.

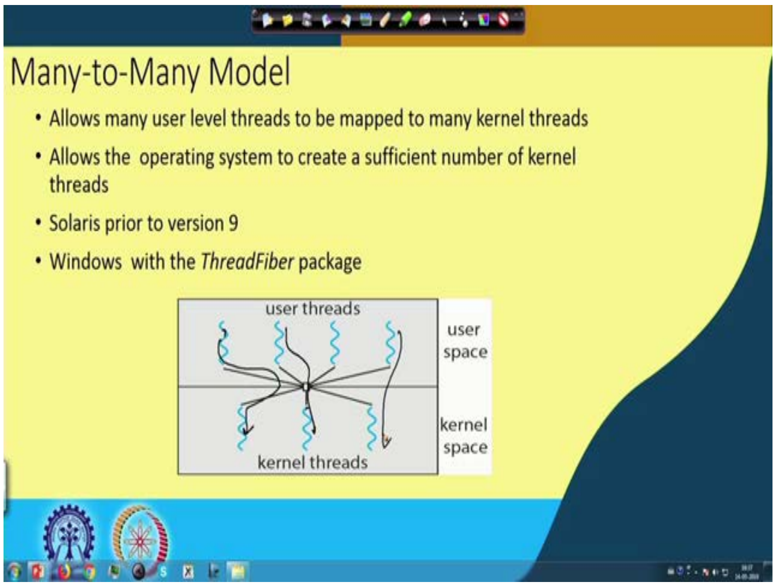
So, I can have mapping, but this mapping is fixed. So, if this thread is blocked somehow may be this fellow has made a system call and it was getting executed by this kernel thread and it got blocked somewhere here, then this threads will also get blocked because they cannot make any system call. Of course, this group does not suffer because of this thread getting blocked. So, this group does not suffer. So, we have got this many user level threads mapped to a single kernel thread and one thread blocking causes all 2 blocks. So, that is the thing that I was telling that if this thread gets blocked, then all this threads they will get blocked. Multiple threads may not run in parallel on multi core system because only 1 may be in the in kernel mode at a time.

So, this is another difficult situation what it says is that. So, maybe I have got multiple cores now this multiple core suppose I do not have this thing I do not have the second one I have got only this 4 user level threads and this is a kernel level thread and now suppose I have got 4 cores in my system. So, I have got 4 cores. So, they are this is given to core c 1 this is to c 2 this is c 3 and this is c 4, but after that c 1 c 1 has made a system call and it is here then if c 2 in its execution also needs to make a system call; may be c 1 1 to print something and. So, it is make it has made a call to print f and as a result it has gone into kernel mode. Now c 2 was.

So, this is for c 1 and c 2 also wants to print something c 2 wants to maybe in may not be printing. So, it may be opening a file it may be open a file, but now this thread cannot proceed because only 1 kernel thread was there for this c 1 c 2 c 3 c 4 group of user threads and that is now busy with c ones system call. So, c 2 has to wait. So, even if you have even if you have got multiple cores available. So, you cannot make them to run parallely. So, multiple threads they may not run in parallel on multi core system because only one maybe in kernel mode at a time. A few systems currently use this model like solar is green threads GNU portable threads.

So, they follow this particular strategy the advantage that we have is of course, thee mapping is unique. So, that way the handling this kernel mode of operation becomes simple. So, that that is why it is done. So, maybe it is it maybe it will work well for may systems and that way we can use it.

(Refer Slide Time: 24:10)



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

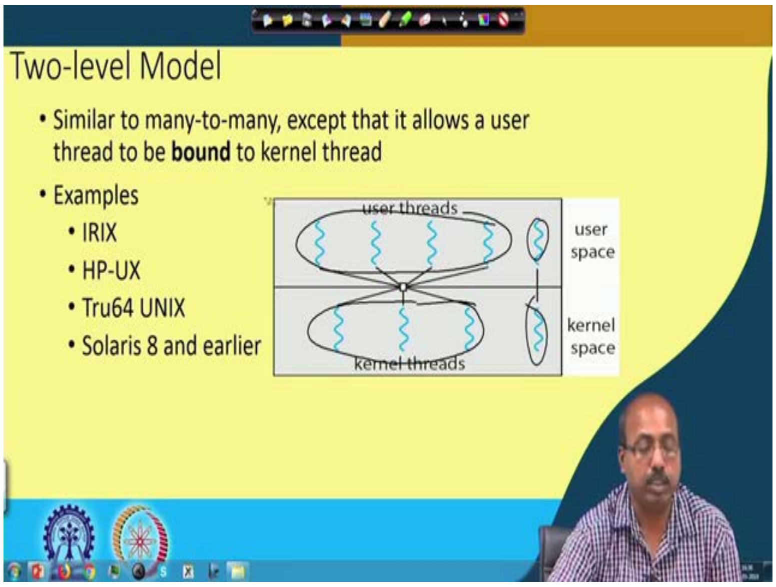
Another module that we have is the many to many mode. So, this is more a more flexible one that way. So, we have got a group of kernel threads and sorry we have got a group of kernel threads like this and we have got a group of user level threads and the mapping is between them. So, whenever this user level thread. So, this user thread supposes it wants to make a system call. So, will the system will find out whether the any kernel level thread is free maybe this one is free.

So, as a result this system call will be executed by this thread. Now unlike the previous one where we had got a single kernel thread for a group of user thread. So, here I have got multiple kernel threads. So, if this fellow now makes a system call then maybe it can find this thread that can take it up. So, there is no distinction between the capabilities of this kernel thread. So, what all of them can execute all of them can realize all the system calls. So, as a result there is no distinction between their capabilities. So, only thing is that they should be free; so, if the user the kernel thread is free and any user thread makes a system call.

So, it will be taken up by that. So, it allows many user level threads to be mapped to many kernel threads and allows the operating system to create a sufficient number of kernel thread. So, now, the west designer is now a bit safe because unlike that one-to-one mapping when that you whenever a user thread is created. So, one kernel thread will be created. So, like that or say that fixed mapping that many-to-one mapping like only one kernel thread is there for a group of user thread. So, that that type of situation so, what happens here is that the west designer can decide I will create a some number of kernel threads which is assumed to be a good enough for the type of load that the system has and it may also be a tunable parameter that way.

So, that way; so, the user the west designer will create a number of kernel threads and then do this mapping between the user threads and kernel threads. So, allows the operating system to create sufficient number of kernel threads and Solaris prior to version 9. So, it was using that. So, windows with thread fiber package so, this can also be the also use this type of facility in which this can be implemented.

(Refer Slide Time: 26:46)

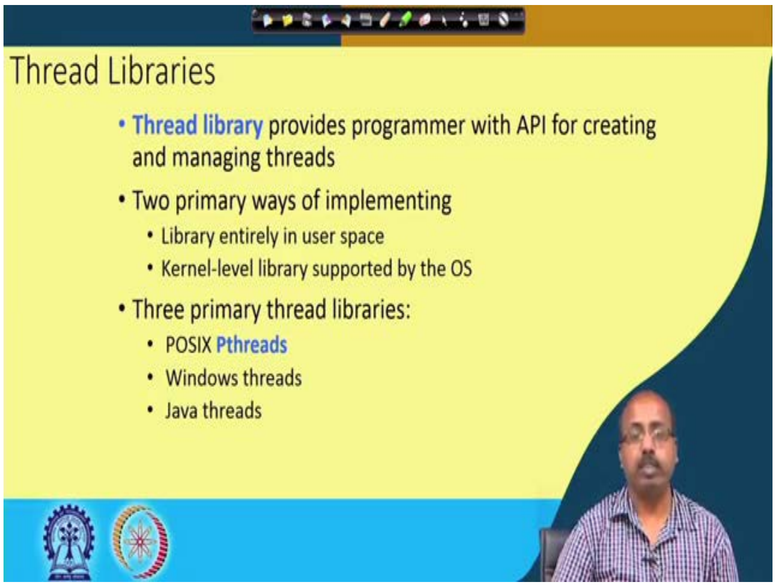


Two-level Model

- Similar to many-to-many, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

There can be another situation which is known as 2 level model. So, it is similar to many to many except that it allows a user thread to be bound to kernel thread. So, this is this is many to many so, we have got here we have got this many of this user thread. So, they are bound to some kernel threads, but the set that is bound to. So, that is fixed. So, this set of user threads it is bound to the kernel thread similarly this set is bound to. So, this thread so, that way it goes on. So, the typical examples of this type of system are IREX, then HP-UX then Tru64 UNIX and Solaris 8 and earlier system. So, they use this 2 level model. So, many to many model is the most generic one. So, that is there. So, that it is there, but this 2 two level model. So, it allows a user sais to be bound to the kernel thread. So, their binding is fixed.

(Refer Slide Time: 27:51)



Thread Libraries

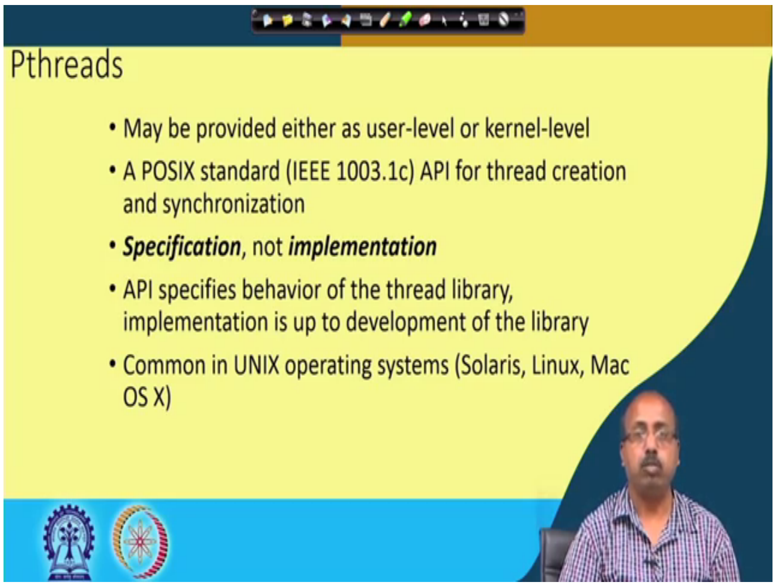
- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads

Now, going to thread libraries; so, they there are different libraries that have been created for helping the user to write programs using the using this threading philosophy. So, otherwise will be it is difficult to develop applications around thread using this thread concept, with thread library it provides the programmer with an application API interface for creating the creating and managing threads.

There are two primary ways of implementation one is a library entirely in the user space and we can have kernel level library supported by the operating system. So, as we know that there are user level threads and kernel level threads. So, it may be that we can have this library entirely created in the user space. So, all that the libraries that that we have. So, the user level thread will be using it and of course, operating system will support this kernel level thread so, that can be utilized.

So, there are three primary thread library is one is the POSIX Pthreads library, then we have got windows threads library and we have got java threads library. So, all these libraries they will provide us some way by which we can create some threads, we can do computation, we can do waiting for some thread to be over this parallelism can be taken care of.

(Refer Slide Time: 29:17)



Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

So, will be looking into these thread libraries. So, the one of them is this Pthreads library and it is provided as a standard. So, it has become a standard now. So, many of the operating systems we will see that we will they will follow this Pthreads libraries like Solaris, Linux Mac OS X and all. So, they support this Pthreads library. So, we will not going to much detail of this Pthreads, but we will try to get a glimpse of like how this thread level programs will look like.

So, that will do in the next lecture.