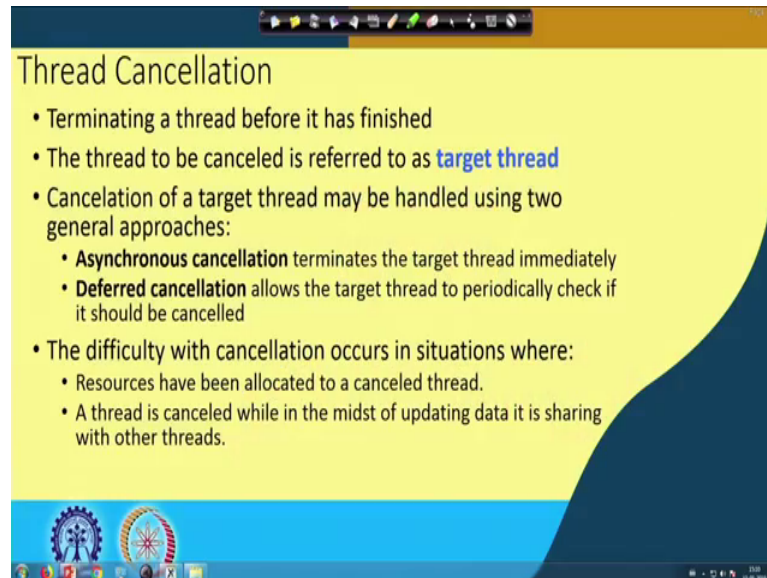


Operating System Fundamentals
Prof. Santanu Chattopadhyay
Department of Electronics and Electrical Communication Engineering
Indian Institute of Technology, Kharagpur

Lecture – 23
Threads, Scheduling

(Refer Slide Time: 00:29)



Thread Cancellation

- Terminating a thread before it has finished
- The thread to be canceled is referred to as **target thread**
- Cancellation of a target thread may be handled using two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- The difficulty with cancellation occurs in situations where:
 - Resources have been allocated to a canceled thread.
 - A thread is canceled while in the midst of updating data it is sharing with other threads.

In our last class we were discussing about, Thread cancellation that is when we think that a particular thread should not run any more then how it can be terminated. So, cancellation is the process in which we terminate a thread before it is finished, now the thread can be cancelled that is there so that thread that you want to cancel will refer to as the target thread.

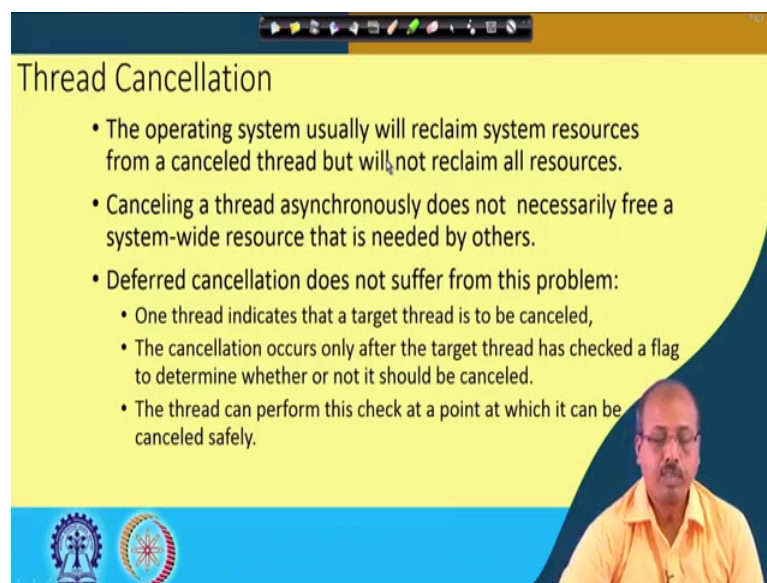
Now a cancellation maybe handled in two approaches as we have seen previously one is asynchronous cancellation. So, it terminates the target thread immediately and deferred cancellation in which case the target thread is inform that you should be terminated and the target thread itself checks periodically that whether there is any message to terminate me and then accordingly it will terminate. So, the problem with this termination or cancellation is that the resources that are allocated to a thread so they also what to do with those resources so, we have to take a decision on that.

So resources have been allocated to cancel thread and a thread is cancelled where in the middle of updating the data, it is sharing with other threads. So, this type of situation so

if we try to cancel threads so it becomes a problem. So, particularly for the second situation where the thread was updating some data and where on and that data is being shared by other threads as well.

So, they are actually the difficulty is more and that is why it is said that we should be the thread should be checking itself that, whether there is any cancellation request is there and if there is any cancellation request then it will terminate it in a proper fashion so that this data synchronization is not lost across threads.

(Refer Slide Time: 02:13)



The slide is titled "Thread Cancellation" and features a yellow background with a blue border. It contains a list of bullet points explaining thread cancellation. In the bottom right corner, there is a video inset showing a man in a yellow shirt speaking. The slide also includes logos of institutions at the bottom left.

Thread Cancellation

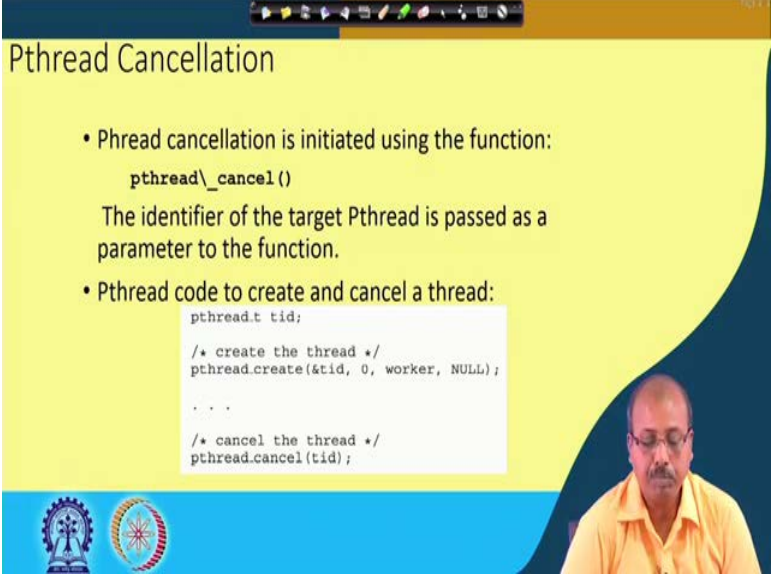
- The operating system usually will reclaim system resources from a canceled thread but will not reclaim all resources.
- Canceling a thread asynchronously does not necessarily free a system-wide resource that is needed by others.
- Deferred cancellation does not suffer from this problem:
 - One thread indicates that a target thread is to be canceled,
 - The cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled.
 - The thread can perform this check at a point at which it can be canceled safely.

Now, operating system usually reclaims the system resources from a cancelled thread, but will not reclaim all resources that is of course, desirable because some resources may be shared across a number of threads. So, if the resource is taken back because a thread is terminating so that will create problem. So, cancelling a thread asynchronously does not necessarily free a system wide resource that is needed by other threads so that you that policy has to be followed.

So, this is the point in differed cancellation we have the advantage. So, because the one thread indicates that the target thread has to be cancelled the cancellation occurs only after the thread has checked a flag at to determine whether or not it should be terminate it should be cancelled.

So, there we do not have the problem that these thread resources will be half updated and all that so that situation will not occur, because thread will perform the check and it can come to a situation where everything is fine. So, this deferred cancellation is a much better proposition.

(Refer Slide Time: 03:21)



Pthread Cancellation

- Pthread cancellation is initiated using the function:
`pthread_cancel()`
The identifier of the target Pthread is passed as a parameter to the function.
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

...

/* cancel the thread */
pthread_cancel(tid);
```

In case of pthread; so, pthread cancellation is initiated by using the function pthread cancel. So, this pthread cancel, so this is this will asks the thread to be cancelled identifier of the three of the target pthread is passed as a parameter like in this call you see that pthread canceled tid.

So, here this thread id is passed, so from so this tid is has been obtained when if the corresponding thread was created. So, this thread id is passed here and this tells that this thread has to be cancelled.

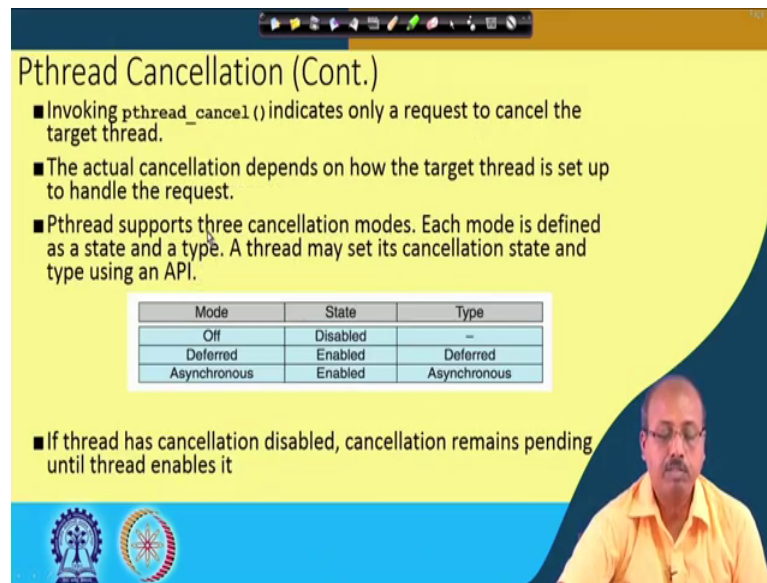
(Refer Slide Time: 04:01)

Pthread Cancellation (Cont.)

- Invoking `pthread_cancel()` indicates only a request to cancel the target thread.
- The actual cancellation depends on how the target thread is set up to handle the request.
- Pthread supports three cancellation modes. Each mode is defined as a state and a type. A thread may set its cancellation state and type using an API.

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it



Now pthread so this is the code for pthread cancellation and as we have noted in the last class also like, this invoking this pthread cancel it indicates that only a request to cancel the target threads. So, the thread is not cancelled immediately, just a request is sent to the target thread that if there is a cancellation message. So, about the actual cancellation it will depend on how the target thread is setup to handle the request. Now pthread supports three cancellation modes each mode is defined as a state and a type. So, one of them is the off mode, so this off mode means the state will be disabled and the type it has does not have any meaning their.

So, the mode is off then it will now it cannot be cancelled. On the other hand so mode if the mode is deferred or mode is asynchronous, now there is a question of disabling and enabling the thread cancellation. So, in a deferred mode if the thread cancellation is enabled, in that case the type of cancellation will be deferred. So, if a thread is created in deferred mode and the current state is disabling then it will not check for this periodic thread cancellation requests.

On the other hand if it is deferred and the state is enabled, in that case of course, it will be looking for looking periodically for the deferred for the cancellation message and in the if the cancellation request comes then it will cancel at some point. If the mode is asynchronous again the state can be disabled or enable. So, the state is disabled then the

thread cannot be cancelled, if the thread is enabled then the thread will be terminated immediately, so that is asynchronous mode of cancellation.

So, pthread supports three cancellation modes and each mode is defined as a state and a type. A thread may say its cancellation state you and type using an API if thread has cancellation disabled cancellation remains pending until thread enables it so that is what I was telling. So, the cancellation is disabled then no cancellation will take place.

(Refer Slide Time: 06:13)

The slide is titled "Thread Cancellation (Cont.)" and contains the following text:

- The default cancellation type is the deferred cancellation
 - Cancellation only occurs when thread reaches **cancellation point**
 - One way for establishing a cancellation point is to invoke the `pthread_testcancel()` function.
 - If a cancellation request is found to be pending, a function known as a **cleanup handler** is invoked. This function allows any resources a thread may have acquired to be released before the thread is terminated.
- On Linux systems, thread cancellation is handled through signals

Hand-drawn diagram: A vertical rectangle with two arrows pointing to it from the left, both labeled "pthread_testcancel()".

If default cancellation type is the is deferred cancellation. So, by default the cancellation is deferred that is good as we discussed previously. So, default has got advantage because this resource sharing and all so they becomes easy. So, cancel one way above establishing cancellation point is to invoke the pthread test cancel function. So, in a code of p of a particular thread, so if you put some where this pthread test cancel, so it will be checking for this cancellation point thread so it is like this.

Suppose this is the quote for the thread and at some point say, so suppose it sees that this is a reasonable point at which the thread can release the control or thread can terminate itself. So, at this point maybe it is the situation where it does not hold any resources with it. So, at this point can it can call this test cancel function so it can call this test cancel function here.

So, it will test whether a cancellation request is there or not. Similarly after sometime again maybe at this point it puts another such test cancel function call. So, this way the in between if there is a when the thread was in this region, so if a cancellation request had has come then the thread will not be terminated immediately, but it will when the next time this test cancel function is executed, so it will be testing whether the thread is cancelled or not and the accordingly the thread will terminate.

So, this way the developer of the thread or the writer of the thread can decide like at which points it can go for this cancellation and that is up to the programmer. So, this thread programming is expected that the user is quite knowledgeable person, then that you that and that fellow can put this cancelation points appropriately in the code. If a cancellation request is found to be pending a function known as cleanup handler is invoked and then this function allows any resources a thread may have acquired to be released before the thread is terminated.

So, in the thread cancel in the test cancel function while this test cancel is called. So, if some resources are held those resources are to be released and for that purpose it calls a cleanup handler so that will automatically do this resource release and all. On Linux system thread cancellation is handled through signals whereas, other operating systems they may do it in a different fashion. So, these are thread cancel function.

(Refer Slide Time: 08:55)

The slide is titled "Thread-Local Storage" and contains the following bullet points:

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to static data
 - TLS is unique to each thread

Handwritten notes on the slide include:

- A circle containing "Code", "Data", and "Stack".
- A bracket on the right side grouping "cancel(--)" and "static int(0)", with a vertical ellipsis below it.

A video inset in the bottom right corner shows a man in a yellow shirt speaking. The slide also features logos of institutions at the bottom left.

There is another important issue which is known as Thread Local Storage or TLS. So, thread local storage it allows each thread to have its own copy of data. So, this is very important because as we have discussed previously, so if there is any at the process level you have got code data and stack segments and for a setup threads of any process the code and data so these two are same. So, these are shared across all the threads only the stacks are different stack segments are different that you have seen.

But sometimes it is necessary that thread has got its own data, which is not accessible to other threads and which are the so that way we can create some thread local storage that will allow the thread to have its own copy of data. So, this is useful when you do not have control over thread creation process for example, that is when you are using a thread pool so maybe from thread pool you are taking some thread and you do not have control over the thread creation process. So, in this case I should have some variable which is used as used in the thread local storage.

So, this is not as local variables because local variables visible only during single function invocation, but TLS is visible across function invocations. So, it is like this it is some sort of static data. So, in C language program, so you have you must have seen this static data type so if a variable is defined statics so, even within a function 1 So, if you have defined a static int static integer x.

Then across different invocations of this function 1 this the variable x remains live and this x retains the previous value. So, see in TLS also if you define some variable so, they remain active they are they remain valid across different invocations of the functions. So, as a result, so this is not a local variable, but rather it is a static variable of, but of course, it is not visible outside the function, but it is for multiple invocation of the function the variable is available. So, this is the issue of thread local storage. So, this is another advanced feature that this threads will provide.

(Refer Slide Time: 11:27)

Scheduler Activations

- Both many-to-many and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- Typically uses an intermediate data structure between user and kernel threads
- This data structure is known as a – **lightweight process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP is attached to kernel thread.
 - The kernel threads are the ones that the operating system schedules to run on physical processors.

So, sometimes we need to use them. Another important issue with the threads is the scheduler activation. So, we have got many to many and two level models that require communication to maintain the appropriate number of kernel threads allocated to the application. So, scheduler has to schedule those threads and so that this proper execution takes place.

Typically uses an intermediate data structure, between user and kernel threads because what happens is that there is a mapping. So, then that mapping may not be fixed ok. So, they particularly the many to many and type of mapping so this mapping may not be fixed and the so we have to try once at some point of time, so, many to many type of mapping. So, we have got a number of user level threads ok, so they are map to in again a number of kernel level thread.

So, this is the theses are the user level threads and these are the kernel level threads. So, they are mapped that way so, at some point of time maybe this thread was mapped to this kernel thread this user thread was mapped to this kernel thread. And sometimes some other time maybe this particular kernel thread may be associated with this user thread. So, as a result the data the association between this user level thread and kernel level thread so that will vary over different invoke or different execution of the system calls.

So, we must have some intermediate data structure between this user and kernel level kernel threads. And this data structure is known as lightweight process or LWP. So, this

is called the lightweight process this data structure which has got this data that can be transfer between user and kernel threads.

So, it appears to be a virtual processor on which process can schedule user thread to run. So, this data structure we have got a set of data structures and that way that defines the interface. So, if you are putting one user level thread onto a kernel threads, so essentially you are mapping it onto the data structure. So, this is basically a some sort of virtual processor on which process can schedule user thread to run. Each lightweight process is attached to kernel thread.

So, each such data structure mapping so that is mapped to that is attached to a kernel thread. The kernel threads are the ones that the operating system schedules to run on physical processors ultimately, this kernel threads will be scheduled on to the; on to the physical processors; so that is how this scheduler will operations will take place.

(Refer Slide Time: 14:05)

The slide is titled "Scheduler Activations (Cont.)" and features a yellow background. At the top, there is a navigation bar with various icons. Below the title, a bullet point reads "Lightweight process schema". To the right of this text is a diagram showing a vertical stack of three elements: a blue curly bracket labeled "user thread", a grey rectangle labeled "LWP" (lightweight process), and a blue circle labeled "k" (kernel thread). Arrows point from the text labels to their respective elements in the diagram. Below the diagram, there are two more bullet points: "Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library" and "This communication allows an application to maintain the correct number kernel threads". At the bottom of the slide, there are two logos on the left and a video inset of a man in a yellow shirt on the right.

So, basically this is the scenario, so we have got a user level thread, so that is mapped onto a lightweight process and from this lightweight process. So, there is a kernel thread so that will be running. So, this is the lightweight process schema the way processes it. So, scheduler activations provide up calls a communication mechanism from the kernel to the up call handler in the thread library. So, this one when this is done then it will be for transferring the data back to the user thread, so it will be using this up call this communication allows an application to maintain the correct number of kernel threads.

Because the number of kernel threads is not increased, so they remain fixed so that way we can maintain this order.

(Refer Slide Time: 14:51)

Conclusion

- Thread is a flow of control within a process
- Benefits include increased responsiveness to the user, resource sharing, economy and ability to exploit multiprocessor architecture
- Threads may be at user-level or at kernel-level
- Models may be one-to-one, many-to-many

So, we have discussed about threads and to summarize the discussion. So, thread is a flow of control within a process benefits include increase responsiveness to the user, resource sharing economy and ability to exploit multiprocessor architecture. And threads maybe at user level or kernel level and we can have different types of models one to one many to many etcetera.

So, threads are very useful as we can understand that threads have to required to have this multi core architecture for exploitation of that thread is very useful and thread level programming it also helps because one particular in a client server environment. So, if one client request gets stuck at some point other client requests can proceed or in a operating system implementation also if one process has made a system call and that system call is stuck in the kernel mode. So, it does not happen that other system calls will also get stuck.

Because of this multithreaded kernel structure, so they are they can proceed so that this these are the various advantages and that is why is thread level programming is very popular and thread level design of operating system so that is also very popular. So, with this we conclude our discussion on threads and all. So, in the next portions will be

looking into the CPU scheduling policies and that will be very important because it will so it will be talking about how this CPU is allocated to different processes and threads.

So, this is a very important issue because the CPU is one of the very important resource that we have and the everybody wants that my job be run on the CPU. So, my job should get attention of the CPU immediately. So, but that is not possible also because even if you have got multiple CPU's, so at one point of time you can cater to a number of users only. So, it cannot be all the users that are there in the system so you do not have so many processors so that they can be run simultaneously.

And there are other issues like there are priorities among the users some of the processes are very important, so they should be run immediately some processors are not that important so they have got lower priority. But at the same time a process has got lower priority does not mean that it will always suffer delay because of this other processors that are there in the system. So, we have to be careful on that line.

Then also we have to see like this CPU; CPU scheduling a policy should be such that every process gets some time for execution. And the system resources that we have so they are also getting utilize properly in the way.

(Refer Slide Time: 17:49)

CONCEPTS COVERED

Concepts Covered:

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling

Ready → Running
CPU
Scheduler/Dispatcher

So, will be discussing about the CPU scheduling in detail. The concepts that we are going to cover first will start with the basics of this CPU scheduling, then the scheduling

criteria scheduling algorithms and thread scheduling. So, to just recapitulate a bit so, you remember that whenever a process is available in the memory so, this is in the ready state.

So, once the CPU the process gets a chance for execution so it goes to the running state. Now in a multi user or multiprocessing system what happens is that there will be many jobs which are there in the ready state. So, if you represent it in the form of a queue, so you can say that all these jobs they are waiting in the queue. So, these are different jobs that are waiting in the queue.

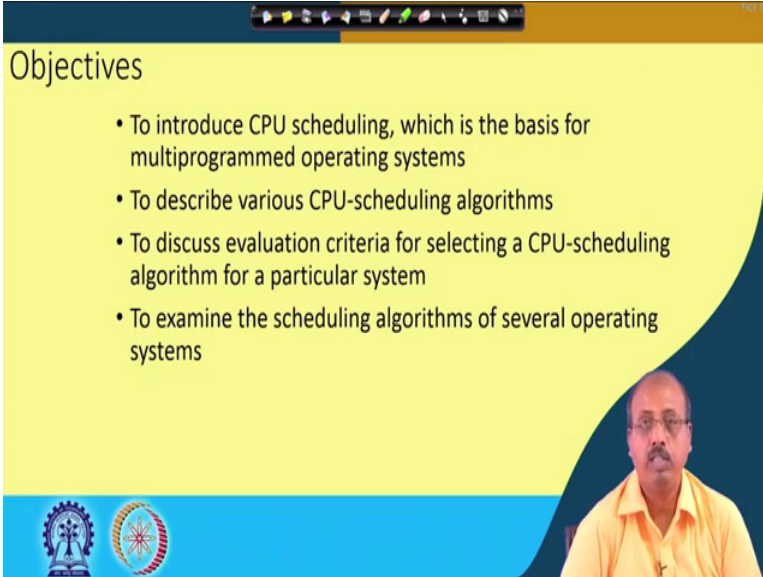
Now, I have to run a routine which is called scheduler or sometimes it is called a dispatcher. So, which will be you should taking up one job from this queue and it will be putting it into the running state so that this job that is selected gets chance for execution. Now one thing you try to understand that this scheduler or dispatcher that we are designing this scheduler has to be very very fast.

Because, even if it takes a very fraction of time, but that is an overhead because that does not contribute to the system usage like we do not say that if the scheduler has for the entire day if the scheduler has run for say 10 second then also it is not going to add to the system utilization because, this is run of the scheduler is not taken as part of the user jobs that are getting completed.

So, if they are important the scheduler is very important because I have to schedule this user level jobs at the same time they do not come into credit for the system utilization so that is how that is why. So, if you want to maximize if you utilization somehow I have to reduce the time that this scheduler takes for execution. So, we have got this ready processes that are there in the memory and they are to be put on to the running process running state for execution.

So, how do you do this so that is the overall thing that are that we are going to learnt in this particular portion of the discussion.

(Refer Slide Time: 20:21)



Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

So, objective that we have used to introduce CPU scheduling which is the basis for multiprogrammed operating systems. So, if you have got a single user system and the user is just firing one job and there is nothing else to do then of course, the scheduler is not necessary because the scheduler has got only one thing to do and that can be done very easily, so you do not have to depend on we do not have to make a choice like which job to run next.

So, only one job will be running and that will be running always and when that is over then the CPU will remain idle until and unless the user gives another job for execution. So, that is a first point is that for multiprogrammed operating system. So, we have got this CPU scheduling problem you have to describe various CPU scheduling algorithms. So, the different algorithms have been proposed, one thing that we want that I would like to mention is that the CPU scheduling algorithms that we have here they must be very very simple ok.

So, because if it is a complex logic then there is execution will take more time so that will add to the overhead of the system. But you so this CPU scheduling algorithms they must be very very simple; compared to the other scheduler that you may possibly remember the long term schedulers which were selecting the jobs to be submitted to the system. So, that way it was choosy because that long term scheduler was trying to optimize utilization of all the system resources. So, it was trying to make a job mix such

that all the system resources are utilized properly, but that is run to a not that frequently so, we can be choosy there.

But this CPU scheduling algorithms, so they must they are not that they cannot be that much choosy they have to do it very fast. Now to discuss will discuss about the evaluation criteria for selecting a CPU scheduling algorithm for a particular system like, different algorithms they will have different properties and the some criteria the evaluation criteria that you want to optimize so based on that we may like to choose one CPU scheduling policy or the other on the particular system. We will be looking into the scheduling algorithms of some operating systems.

(Refer Slide Time: 22:35)

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- Most processes exhibit the following behavior:
 - **CPU burst** followed by **I/O burst**
- CPU-I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- CPU burst distribution is of main concern

The diagram illustrates the process execution cycle with the following components:

- CPU burst:** load store - add store - read from file
- I/O burst:** wait for I/O
- CPU burst:** store increment - index - write to file
- I/O burst:** wait for I/O
- CPU burst:** load store - add store - read from file
- I/O burst:** wait for I/O

Handwritten notes on the right side of the slide:

$ax^2 + bx + c = 0$
 Input a, b, c, 10
 $b^2 - 4ac$
 $r_1 =$
 $r_2 =$
 Print r1, r2

Now, to start with we look into the structure of any process. So, any process so, it will be doing something like this. So, to start with it will be doing some it will be. So, any program that it starts so, it gets the chance for execution in the CPU, so it is. So, if you look into this any program look into any program after it starts executing then most probably it will after doing some basic initialization of variables and all.

So, it will be asking for some user input like say maybe if I am finding the roots of a quadratic equation $x^2 + bx + c = 0$. So, this equation then their maybe some initialization of some of the temporary variables after that it will take the inputs abc from the user. So, when it is doing this so this is an IO operation where as the initialization that it was doing before this so that is some CPU operation because the

memory location initialization. So, for that you will have instructions and those instructions are executed.

So, they are load store add store read from file etcetera so they are the so this load store add store is fine. So, this is these are these instructions are accessing memory and all, but as soon as it comes to a instruction like read from files so in this particular case input abc the abc may be coming from a file or abc may be coming from keyboard, but now the CPU the program cannot proceed further. So, it will be stored here until and unless this or this IO operation is over.

So, it is waiting for the IO to be over and once the IO is over then it will again go into competition. So, again it does say two steps of competition then again it will be doing some IO like here after getting abc. So, it may be calculating the discriminant $b^2 - 4ac$ and all that it will be calculating the roots r_1 and r_2 . So, up to that much is fine so after that it goes into printing the routes so print $r_1 r_2$. So, as soon as it goes to the print statement again it has to wait for the IO operations. So, this wait for wait to write to files.

So, it is equivalent to write to IO device, so it is trying to it is trying to write something on to the output device. So, again it has to wait for IO. So this is a small program so after printing this. So, this terminates, but in what in general what can happen is that this alternation of CPU operation and IO operation can go on for quite some time till the program terminates, so that is how this is going to happen.

So, any program execution you can divide it into a number of sections. So, one part is for doing the CPU operation and the next part is for the IO operation again the next part is the CPU operation followed by an IO operation like that. So, these are called CPU burst and IO burst. So, CPU burst is the range of program execution when it is doing some CPU operation the CPU is actually needed for doing those operations, so all the competition that we have so, they are part of this CPU burst and then for the IO operations so, it has to go to an IO burst.

Now depending upon the program that you have in your hand the sizes of this CPU burst and IO burst may vary like if you have got a database type of application, then you will see that most of the time it will be try it will trying to read the records from the database. So, as a result it is mode of IO operation then the CPU operation. On the other hand if it

is doing some signal processing algorithm is being executed, so at the beginning it reads the values of the signal samples and then it goes into computation and as a result the CPU burst is large and possibly it will do very little amount of IO operation.

So, that way the size of CPU burst and IO burst so they will vary for different processes. And we have got this CPU and IO burst cycle so process execution consists of a cycle of CPU execution and then IO wait operation and CPU burst distribution is a main CPU is that is the main concern, because IO anyway if the processor if the program is waiting for some IO operation to be done then it does not matter because in that time CPU can pick up some other job and it can do that.

So, this time IO burst size does not affect the CPU scheduling say anyway. But what is going to happen is a size of the CPU burst, so if the CPU burst is very large then I can have different types of policies one policy maybe that I give the CPU for this entire time to the process. So, the so a process is given a CPU and only when it finishes the current CPU burst and goes into the next IO burst then only the CPU is taken back from the process.

So, this is one type of situation, but that way some other process that has arrived may be waiting and particularly the CPU burst is large then we have got this problem. So, another policy maybe ok, I give some it give the CPU to the process, but after sometime I take it back since it is taking too long time so that is another type. So, this way you can think about different policies.

So, in this portion of discussions so we will be talking about what can be the different policies so that based on the CPU burst size we can take a decision possibly and we can come up with a policy so that this CPU utilization is maximized. So, the this is the basic idea so how you what you want to do is to maximize this CPU utilization. So, this is the point that we want to highlight, so we always try to utilize CPU and what is the ideal value for the CPU utilization? So that is definitely 100 percent we want to make CPU utilization 100 percent, but that is not possible as we will see because there will be many other things to do, so utilization will not touch 100 percent, but we will try to do that.