

**Introduction to Operating Systems**  
**Prof. Chester Rebeiro**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Madras**

**Week - 07**  
**Lecture - 34**  
**Threads (Light Weight Processes) Part 2**

So now that we have seen how threads are created and how they can be used to solve large jobs by paralyzation. Now, we will look at how threads are managed in systems essentially, as we have seen threads are executing context and therefore, we need some entities which manages the thread resources, decides which thread should execute in the CPU what CPU should be used and so on.

(Refer Slide Time: 00:45)



**Who manages threads?**

- Two strategies
  - User threads
    - Thread management done by user level thread library. Kernel knows nothing about the threads.
  - Kernel threads
    - Threads directly supported by the kernel.
    - Known as light weight processes.

12

In order to do this the two strategies that are available are known as the User threads and Kernel threads. So, user threads are threads where the thread management is done by user level thread library. So, typically in this the kernel does not know anything about the threads running. In kernel threads, the threads are directly supported by the kernels and these are sometimes known as light weight processes.

So, we will look user threads and kernel threads in more detail taking one at a time.

(Refer Slide Time: 01:20)

## User level threads

- Advantages:
  - Fast (really lightweight)  
(no system call to manage threads. The thread library does everything).
  - Can be implemented in an OS that does not support threading.
  - Switching is fast. No switch from user to protected mode.
- Disadvantages:
  - Scheduling can be an issue. (Consider, one thread that is blocked on an IO and another runnable.)
  - Lack of coordination between kernel and threads. (A process with 100 threads competes for a timeslice with a process having just 1 thread.)
  - Requires non-blocking system calls. (If one thread invokes a system call, all threads need to wait)

13

Let us start with user threads. In a user thread, as shown over here we have the kernel space where the operating system or the kernel executes and we have the user space which has different processes executing. Now, each of these processes would have multiple threads or running. In addition to this we have a run time system over here this rectangle over here, which manages the several threads in this particular process, also what is required is a thread table which is stored as part of this run time system. So, note that this thread table contains information which is local to only the threads in this particular process.

In other process it would have its own run time system and its own thread table. So, it notices that the number of entries in the thread table is equal to the number of threads that are executing. Also note that the thread table which is sometimes known as the TCB or the thread control block is different from the process control block which is stored in the kernel space, essentially besides the fact that the thread table will have fewer entries compared to the process control block in the kernel space, it is also in user space.

So, the advantage of user level threads is that it is extremely fast, and it is really lightweight essentially there are no system calls required to manage

threads, the run time system which is present over here, does all the thread management all the context switch between the threads and so on.

As such the kernel would not have any knowledge that a particular process has multiple threads. So, the second advantage which we will see that this particular mechanism of supporting threads in a system is useful on operating systems that does not support threading. And other important advantage of this user level thread is that switching between threads is the extremely fast, and the reason why it is fast is that there is no switch from user mode to protected mode and back to user mode again.

On the other hand switching between threads would just require switch within the user mode itself from one thread to another. So, the drawbacks on are also several. So, one important drawback is that there is a lack of coordination between the operating system kernel and the threads. So, this is because the OS is not aware that a process is multi threaded, and also it does has no indication about how many threads are present in particular process.

To take an example of what problems it could cause. So, consider a system which has two processes, and one of these processes has 100 threads now the kernel does not know about this because the kernel is unaware about the several threads that are present in a process and therefore, when it does context switching it is unaware that one process requires far more number of threads compare to the other one. So, it would allocate the same time slice interval for the process with 100 threads as well as the process with the single thread.

So, what happens is that, because the scheduler in the kernel is unaware about the number of threads that are executing therefore, scheduling decisions cannot be made to fewer processes with the larger number of threads. So, another drawback with respect to the scheduling comes when the threads are in different states. For example, let us this particular process has three threads, and one of these threads is in run able state, while the other two threads are in blocked state. Now the operating system has is unaware of this. So, what should the decision be? So, since one of the threads should be run able it can execute in the processor or so, should it be considered as a run able process, or since

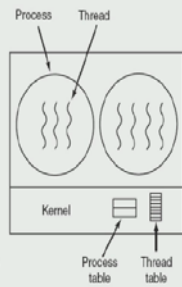
there are two threads which are blocked, should this process be considered as a blocked process.

I am not going to execute in the processor. Third issue occurs with respective system calls, now suppose system calls are blocking in the sense that when one of these threads invoke a system call, then all other threads will need to wait until that first thread completes its system call invocation. Thus in order to support this user level thread model, what is required is that the OS should preferably be supporting non blocking system calls.

(Refer Slide Time: 06:35)

### Kernel level threads

- **Advantages:**
  - Scheduler can decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel-level threads are especially good for applications that frequently block.
- **Disadvantages:**
  - The kernel-level threads are slow (they involve kernel invocations.)
  - Overheads in the kernel. (Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads.)



The diagram shows two circles representing processes, each containing several wavy lines representing threads. Below these is a box labeled 'Kernel' containing two tables: 'Process table' and 'Thread table'. Arrows point from the processes to the process table and from the threads to the thread table.

14

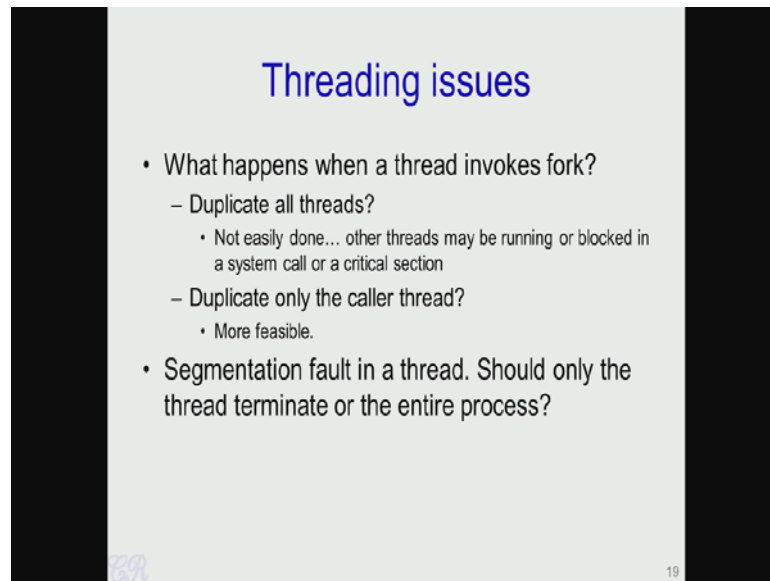
Now let us look at the kernel level threads. So, as before the process which runs in user space could have multiple threads which each thread being a separate or independent execution unit and the management of these thread resources is done in the kernel space. So, along with the process control block, that is shown in here as a process table the kernel also maintains thread control block in the kernel space. So, this is quite unlike the user level threads, where the thread table is maintained in user space, here the thread table or the TCB thread control block is maintained in the kernel space. And therefore, the kernel is aware of the number of threads that a process executes, and therefore, could make a lot of decisions based on this fact.

For example; the scheduler could make more smart in decisions about how much time slices should be allocated to a process, depending on the number of threads the process is running. So, for example, the scheduler could decide to give threads with larger number of processes more amount of time to execute, and other advantage is that since threads are managed by the kernel. So, the blocking on system calls is not required essentially when thread executes a system call the other threads in the particular process can continue its execution.

It does not have to block until this thread which invoke the system call has completed its invocation the drawback of this particular kernel level thread model is that it is low essentially managing the thread would in kernel invocations, and therefore; since this kernel invocations involve system calls therefore, it is considerably slow also there are overheads with respect to the operating system and this occurs because the OS the kernel needs to manage the scheduling threads as well as the processes.

So, this means that in addition to the metadata present in the kernel about the process, more metadata needs to be present for each thread that is executing in the process and therefore, the overheads in the kernel could be significant. So, when we actually design an operating system of a that matter and entire system which supports threads there are several aspects which need to be taken care of, which may lead to lot of complexities, this particular slide highlights some of those issues.

(Refer Slide Time: 09:15)



The slide is titled "Threading issues" in blue text. It contains a bulleted list of questions and options. The first question is "What happens when a thread invokes fork?". It has two options: "Duplicate all threads?" and "Duplicate only the caller thread?". The second option has a sub-bullet: "More feasible.". The second question is "Segmentation fault in a thread. Should only the thread terminate or the entire process?".

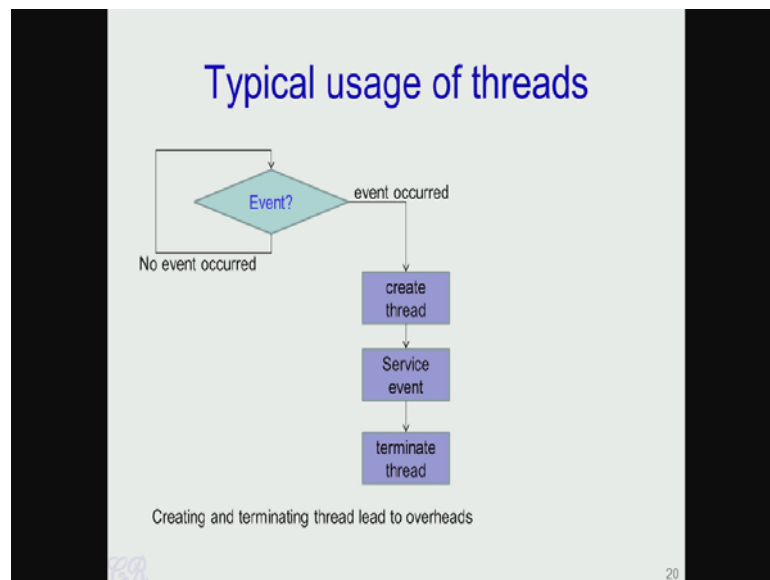
- What happens when a thread invokes fork?
  - Duplicate all threads?
    - Not easily done... other threads may be running or blocked in a system call or a critical section
  - Duplicate only the caller thread?
    - More feasible.
- Segmentation fault in a thread. Should only the thread terminate or the entire process?

For example what should the system do when thread invokes fork. So, as we know fork is a system call which causes the operating system to execute and it would result in duplication of that process. So, a new process would be created which is an exact copy of the invoking process, Now what would happen if a thread invokes the system call fork. So, there are several options that one could think off. So, what should the OS do should call the threads that are executing in the process it should be duplicated in other words should a new process be created, which also has the same number of threads executing in the same stages.

So, this is easy as had been done, essentially it could also create lot of synchronization issues, For example, consider the case that we have a process with two threads, one thread is executing critical operation say in a critical section, which is accessing critical resource while the other thread invokes the fork system curve. So, what would happen if the OS duplicates the entire process along with all it is threads? So, as we have mentioned the second thread in the new process would also be in the critical section and as we have seen in earlier videos this could be catastrophic essentially it could change the output of the program. And other approach that one could follow while designing or managing this particular aspect is where we duplicate the caller thread.

So, even though the process may have 10 different threads, a one of these threads invoke the fork system call the new process created will only duplicate that thread. So, the new process created will not have the remaining 9 threads, but will only have one of these threads. So, another thing to think about is what should happen when there is a segmentation fault in a thread. So, should the operating system terminate the just the single thread, or should the entire process be terminated. Now making choices for these is not easy and operating system designers would need to make critical choices about how to manage these aspects while designing the operating system.

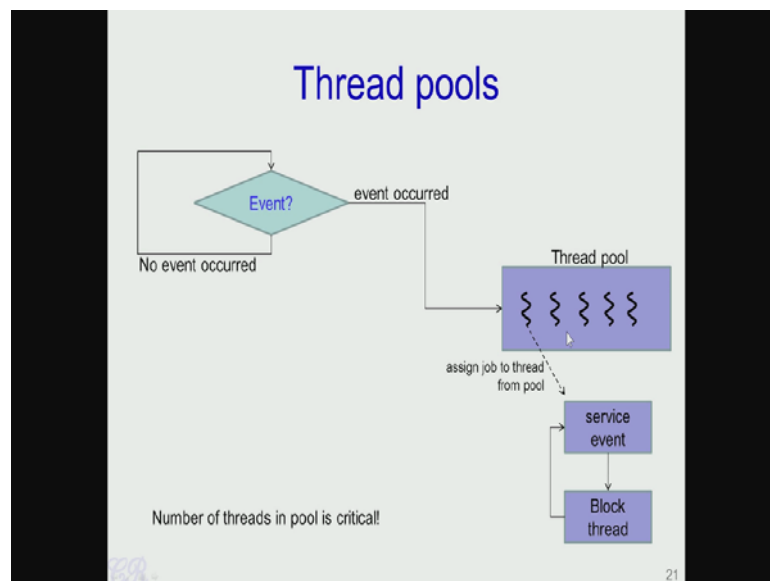
(Refer Slide Time: 11:46)



Let us see one typical application of the use of threads. So, you could take for example, network card where packets keep coming in through the network and these packets have to be serviced let us say through threads. Let us say we have a loop over here, which keeps waiting for an event to occur for example, the packet on the network and when that when an event occurs, it spawns or it creates a thread which services that event and then terminates. So, during the process of servicing of that event, if another event occurs, then new thread is created. In this way we could have several events which are serviced simultaneously.

This approach is scalable because it could service multiple events simultaneously. So, the drawback of this particular model is the overheads, essentially creating and terminating threads though have less overheads, is still an overhead which could reduce performance and these could affect the entire system. So, what applications typically do is use the technique known as thread pools.

(Refer Slide Time: 13:10)



In this particular technique, what the application does on creation is that it is going to create pool of threads. For example, it could create a 50 or 100 different threads and these threads are would typically be in a blocked state.

Now, there would be a main loop which keeps waiting for an event to occur and when an event occurs one of these threads which in the block state is woken up that thread would then service the event and go back to the block state. So, in this way if there are 50 threads in this pool which are already created, there are 50 events which could be serviced simultaneously without any overheads if the 50 first event occurs while all the 50 threads in the thread pool are busy servicing the event, then the 50 first event would need to wait. So, this way we see that we have eliminated the overheads of creating and destroying threads whenever an event occurs.



Now, the only requirement is to pick out thread from the thread pool which would then service the event now and other important aspect with thread pools is the number of threads in the pool. So, this is critical for every application and is going to be very application dependent. For example, if you have a thread pool with very few numbers of threads may be say 4 or 5. Then it could service only 4 or 5 events simultaneously.

If more events occur then the events would have to be queued until the threads are have complete their servicing and therefore, your performance is affected. On the other hand if you have a large number of threads in the thread pool for example, 100 threads while the events do not occur. So, often therefore, large number of threads are simply wasting resources sitting idle in the thread pool therefore, the number of threads in the thread pool is critical choice that an application designable have to make.

So, with this we have given a brief introduction to threads, we have seen the difference between threads and processes and how threads could be actually used to reduce the execution time and improve performance of applications, and we have also seen brief introduction of how threads are managed in operating systems and the various usage of threads.

Thank you.