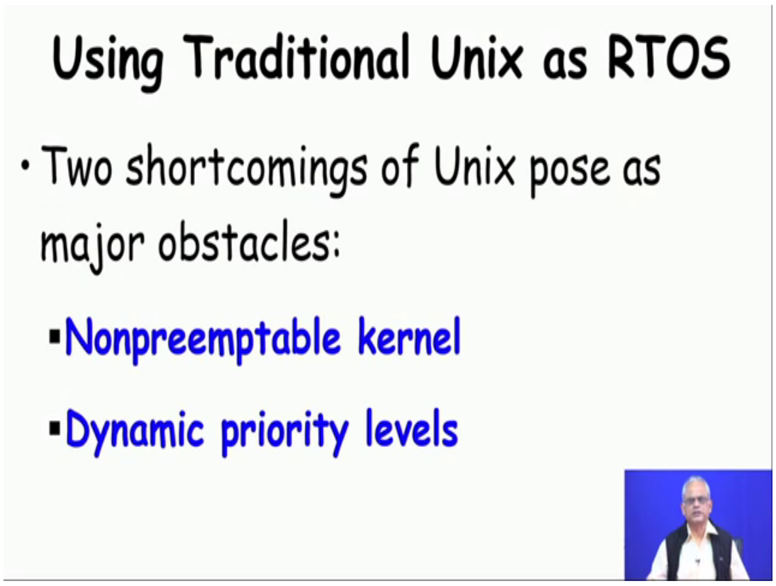
Real Time Operating System
Prof. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 20
Unix as a Real - Time Operating Systems

Welcome to this lecture, in the last lecture we had seen some of the main requirements for a real time operating system, and then we are trying to understand why the traditional operating systems do not support those features, and what needs to be done to support those features. And with this intension we are trying to dissect the Unix operating system and find why it does certain things and how it can be improved.

(Refer Slide Time: 00:57)



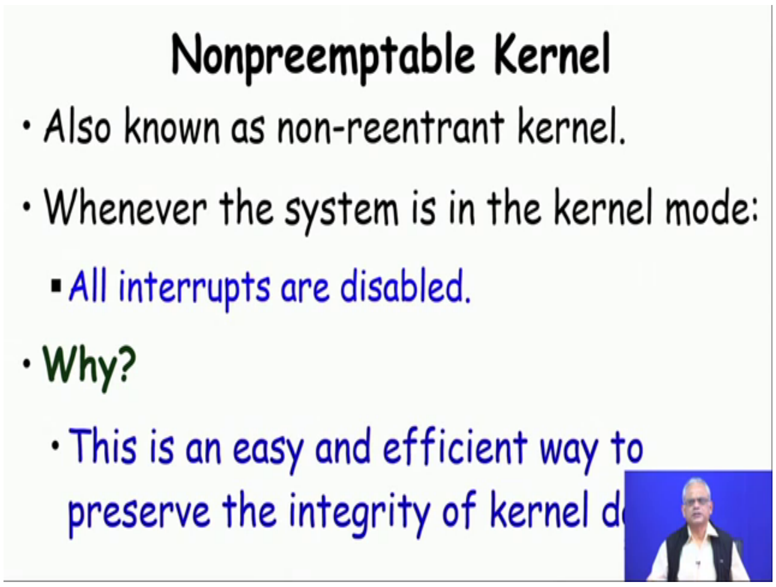
Using Traditional Unix as RTOS

- Two shortcomings of Unix pose as major obstacles:
  - Nonpreemptable kernel
  - Dynamic priority levels

Let us look at using Unix as a real time operating systems. For all the features that we discussed in the last lecture that a real time operating system should needs to support these are not satisfactorily supported by Unix, but then there are two issues that are the major obstacles in using Unix in a hard real time application. The first is a nonpreemptable kernel; when the Unix code is being executed that is the kernel code is being executed then all interrupts are disabled and the second issue is the dynamic priority levels the Unix operating system keeps on shifting the priority level of a task at every time slice.

Now let us try to examine why it does this first of all, why does it disable all interrupts when executing kernel routines and also why does it have to keep on shifting the priority of a task, because in this situation it becomes difficult to run a real time scheduler like rate monotonic scheduler, because the scheduler works on the assumption that the programmer assigned a priorities do not change.

(Refer Slide Time: 02:31)



Nonpreemptable Kernel

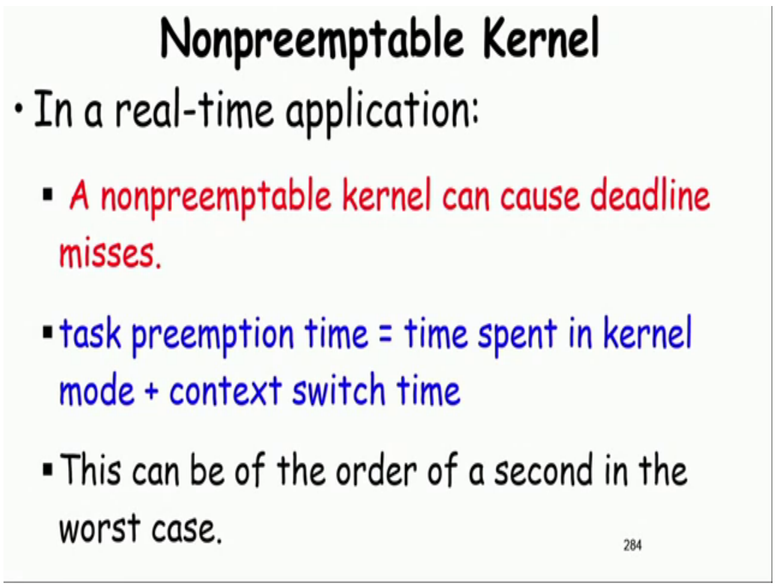
- Also known as non-reentrant kernel.
- Whenever the system is in the kernel mode:
  - All interrupts are disabled.
- Why?
  - This is an easy and efficient way to preserve the integrity of kernel d

First let us look at the Nonpreemptable Kernel. A preemptable kernel is also called as reentrant kernel that is one a once a interrupt occurs then the kernel routine starts it stops after the interrupt and then again after sometime it continues from that same place that is a reentrant kernel that can be interrupted, but then the Unix is a non reentrant or a non preemptable kernel. To achieve this all interrupts are disabled so that the kernel cannot be preempted and it runs all the code that needs to be done, but then you will have question that why is it not non, why it is not reentrant can it be interrupted some other code runs and then starts running the kernel code where it left. The answer is no because that will destroy the integrity of the kernel data structures, the developers of the Unix to make the operating system efficient did not use kernel level locks.

So, if a data structure spot of the operating system that is modified half way, and then there is a preemption and some other part the operating system runs, and then modifies this data inconsistent data it reads modifies then it will lead to an inconsistent system can crash the system. But then to make the problem more severe the user tasks can also run in kernel mode will see this issue. But let us just understand this point that the traditional operating system such as windows, Unix etcetera the kernel is non preemptable.

The main reason is that it is an efficient way to preserve the integrity of the kernel data structure of course, we are talking a Unix 5 and then the later versions of Unix, and the windows, and the Linux they did make the kernel preemptable. Please note that we are discussing here the Unix system 5 as it is, but later developments because this was a big handicap having a non preemptable kernel, and the main reason is that there is no kernel level locks were used and therefore, disabling interrupts was a efficient way to preserve the integrity of the kernel data structure, but this is a big handicap and therefore, later versions of the windows and the Unix they did make the kernel preemptable through use of kernel level locks.

(Refer Slide Time: 06:07)



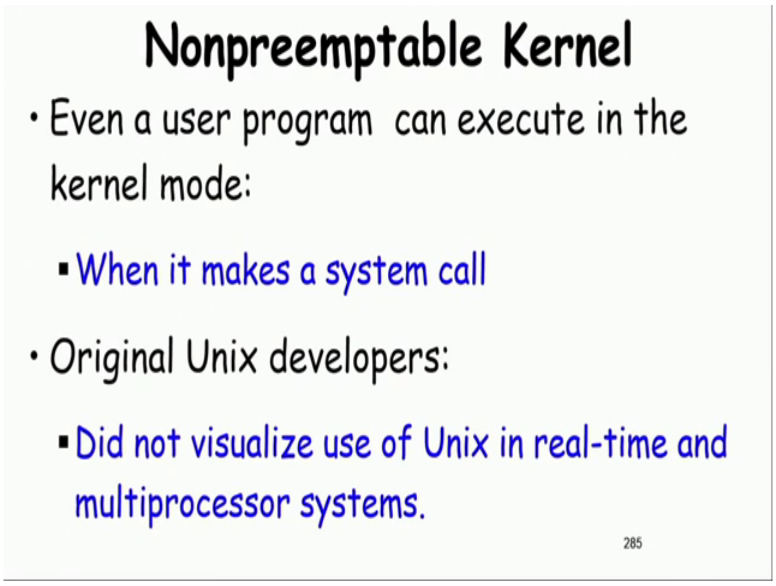
Nonpreemptable Kernel

- In a real-time application:
  - A nonpreemptable kernel can cause deadline misses.
  - task preemption time = time spent in kernel mode + context switch time
  - This can be of the order of a second in the worst case.

284

But if the kernel is non preemptable then it can cause deadline misses because the tasks preemption time becomes the time spent in the kernel mode plus con context switch time. And the time spent in kernel mode can be of the order of a second in the worst case because typically the context switch time is set as one second and the kernel routine can run out to a second and if a task high priority task is prevented from running for one second then definitely it will miss deadline, because typical deadlines are of are of hundreds of micro seconds.

(Refer Slide Time: 06:58)



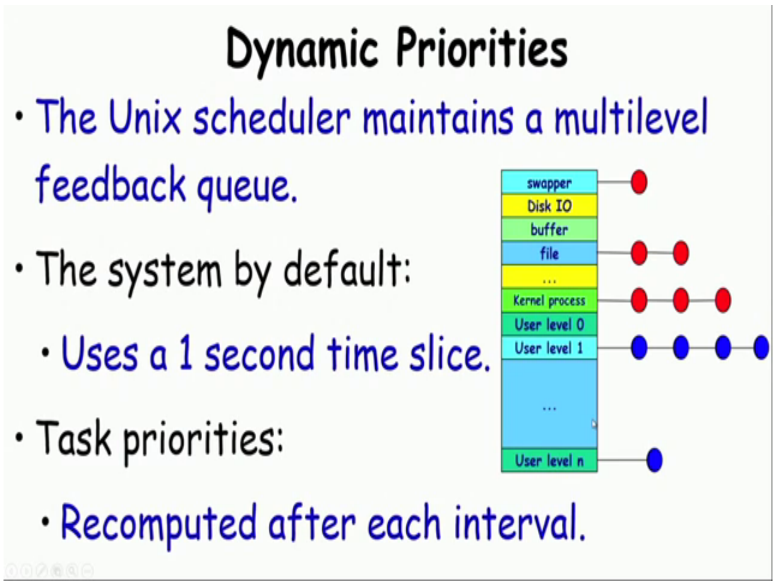
Nonpreemptable Kernel

- Even a user program can execute in the kernel mode:
  - When it makes a system call
- Original Unix developers:
  - Did not visualize use of Unix in real-time and multiprocessor systems.

285

To make the matter worse even a user program can execute in kernel mode, when a user program makes a system call, it runs in kernel mode generates a software interrupt changes the mode to kernel mode and therefore, even when a user program is running and makes a system call, it becomes non preemptable. The original Unix developer wanted a efficient of operating system and they did not visualise use of Unix in real time and multi-processor applications.

(Refer Slide Time: 07:41)



Dynamic Priorities

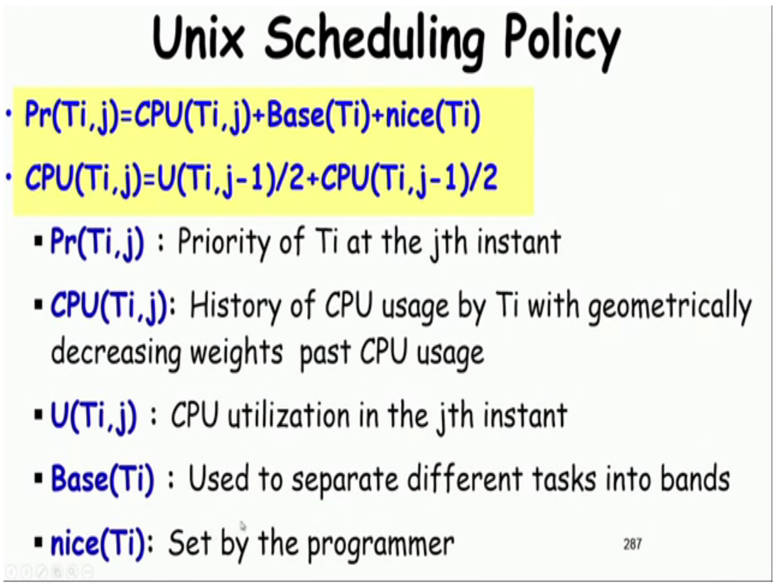
- The Unix scheduler maintains a multilevel feedback queue.
- The system by default:
  - Uses a 1 second time slice.
- Task priorities:
  - Recomputed after each interval.

swapper
Disk IO
buffer
file
...
Kernel process
User level 0
User level 1
...
User level n

Now, let us look at the second issue which is the dynamic priority levels. If we look at the Unix scheduler it uses a multilevel feedback queue, it divides the tasks into several priority ranges and for each range it maintains a se separate queue the different priority ranges are the user levels.

So, there are n user levels may be 256 user levels, then kernel processes are higher priority, then the user processes. The user processes cannot really migrate to kernel level and there are several other bands of priority, the files, buffer, disk I O and the swapper is the highest priority in the Unix operating system. And the system by default uses a one second time slice and the tasks after the complete their time slice is using the task once the complete a time slice the priorities are recomputed.

(Refer Slide Time: 09:00)



Unix Scheduling Policy

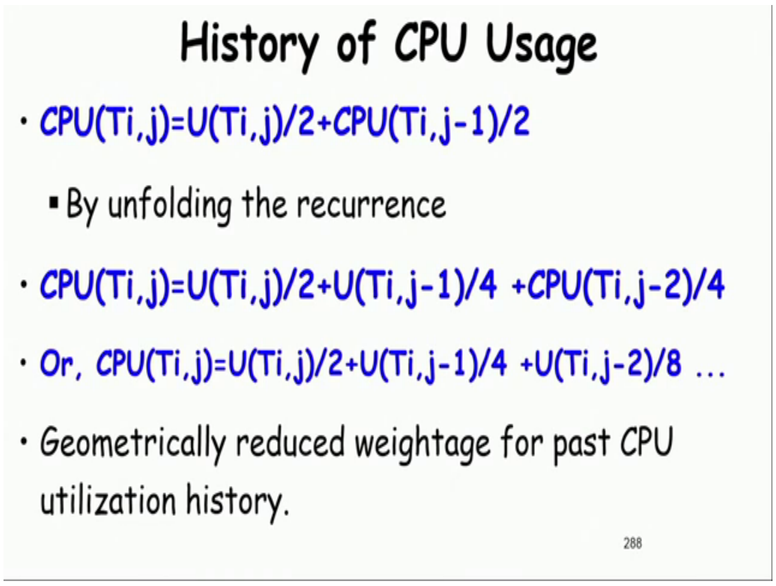
- Pr(Ti,j)=CPU(Ti,j)+Base(Ti)+nice(Ti)
- CPU(Ti,j)=U(Ti,j-1)/2+CPU(Ti,j-1)/2
  - Pr(Ti,j) : Priority of Ti at the jth instant
  - CPU(Ti,j): History of CPU usage by Ti with geometrically decreasing weights past CPU usage
  - U(Ti,j) : CPU utilization in the jth instant
  - Base(Ti) : Used to separate different tasks into bands
  - nice(Ti): Set by the programmer

287

If you look at the traditional operating system book, you will find that the priority of a real of a Unix tasks at its j th times slice is computed at its base priority a nice value and the CPU uses time its a history of CPU uses time. The history of CPU uses time is given in forms of a in the form of a recurrence relation, the CPU just time met j is half of utilization in the j minus 1, 50 percent weight has to how much utilization of the CPU occurred in the previous time slice and the rest have is the history of CPU uses previous to the j minus 1 instant. The priority T i j is the priority of the task Ti at the j th instant of uses time slice. CPU Ti, j is the history of CPU uses a geometrically decreasing weights for the CPU just and U Ti, j is the CPU utilization at the j th instance base Ti,j is a value used by the operating system to separate different tasks in priority bands and nice Ti,j is set by the programmer and as indicated by the name nice.

A programmer can only reduce the priority of these task, cannot really increase the priority.

(Refer Slide Time: 11:00)



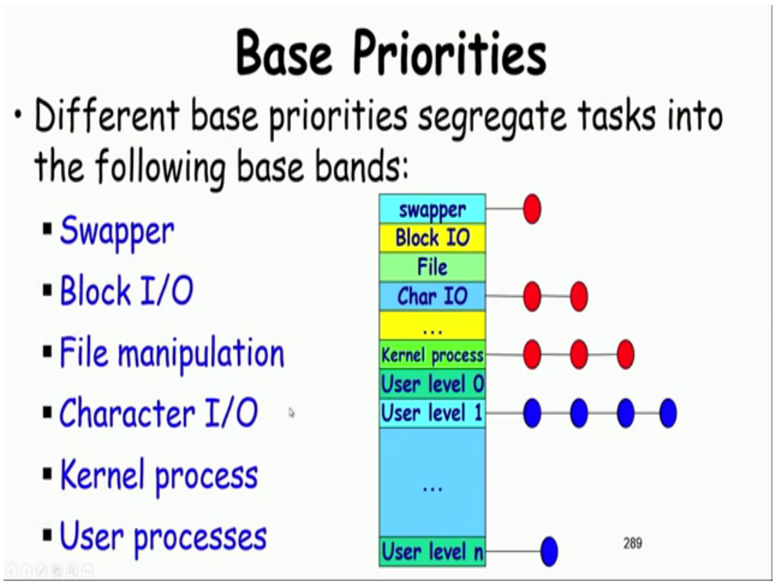
History of CPU Usage

- CPU(Ti,j)=U(Ti,j)/2+CPU(Ti,j-1)/2
  - By unfolding the recurrence
- CPU(Ti,j)=U(Ti,j)/2+U(Ti,j-1)/4 +CPU(Ti,j-2)/4
- Or, CPU(Ti,j)=U(Ti,j)/2+U(Ti,j-1)/4 +U(Ti,j-2)/8 ...
- Geometrically reduced weightage for past CPU utilization history.

288

Now, let us look at this term CPU Ti, j CPU Ti, j is given in the form of a recurrence that is CPU Ti, j minus 1 half of the weight is given to the history of CPU uses over instance j minus 1 and earlier. And half the weight has come from the last time slice that it obtained how much it has used CPU. If we unfold this recurrence we see a geometrically decreasing weightage to the utilization over the different time instances. The last time instance is given 50 percent, the in time slice before that 25 percent, twelve and half percent and so on. So, this is a geometrically reduced weightage for past CPU utilization history, but then you might be wondering that why does it have to do so.

(Refer Slide Time: 12:15)

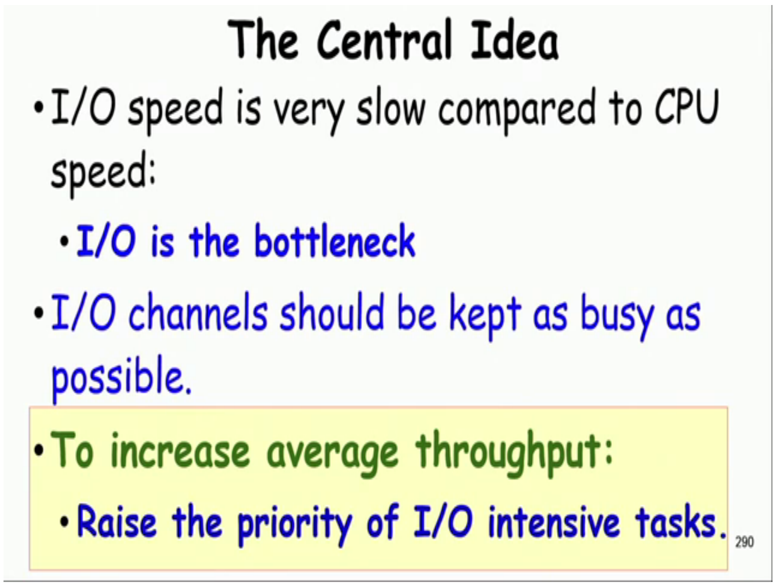


Base Priorities

- Different base priorities segregate tasks into the following base bands:
  - Swapper
  - Block I/O
  - File manipulation
  - Character I/O
  - Kernel process
  - User processes

Before we answer that let us look at another thing is. The base priorities the entire priority band is developed is split into different priority bands swapper block I O file manipulation character I O kernel process, and the lowest are the user processes and the user processes cannot migrate to other priority levels and also a task at a certain priority level cannot cross its band.

(Refer Slide Time: 12:54)



The Central Idea

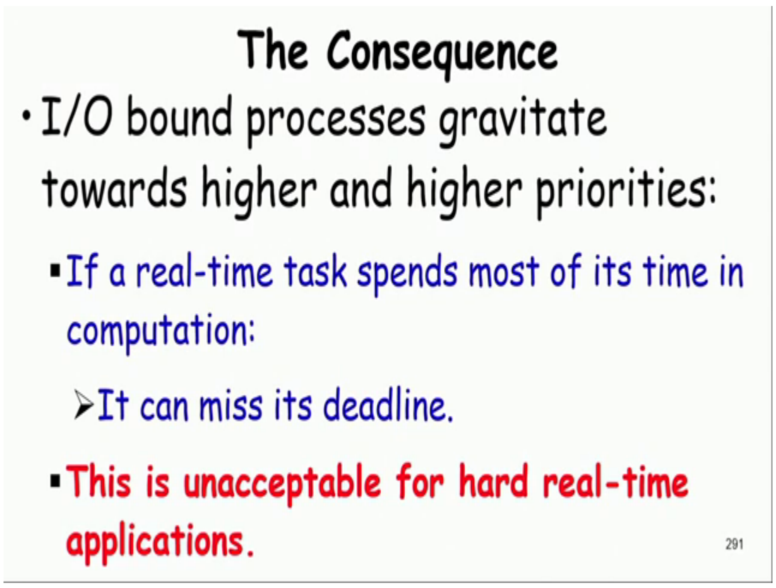
- I/O speed is very slow compared to CPU speed:
  - **I/O is the bottleneck**
- I/O channels should be kept as busy as possible.
- **To increase average throughput:**
  - **Raise the priority of I/O intensive tasks.**

290

The central idea here is that the I O is a bottle neck, CPU executes at high speed, the I O text time and therefore, if a task needs to do its I O, then it priority must be high otherwise the response time will be low. The I O channels need to be kept busy as possible if a task is doing its I O, its in the I O phase it priority should be enhanced.

So, that all the I O requests are raised and the I O channel is kept busy, and that is the reason why I O bound tasks the priority is made higher and higher whereas, CPU bound tasks the priority is made lower and lower by use of the priority scheduling formula sorry the priority computation formula which we just looked at to increase the average throughput, the priority of the I O internship tasks need to be raised and therefore the utilization of the CPU in the previous instance is computed, and if the utilization is low that indicates that it is doing I O, then the priority of the task increases. Remember that in Unix the lower is the priority value, the higher is the priority and therefore, the utilization is slow the priority increases.

(Refer Slide Time: 14:43)



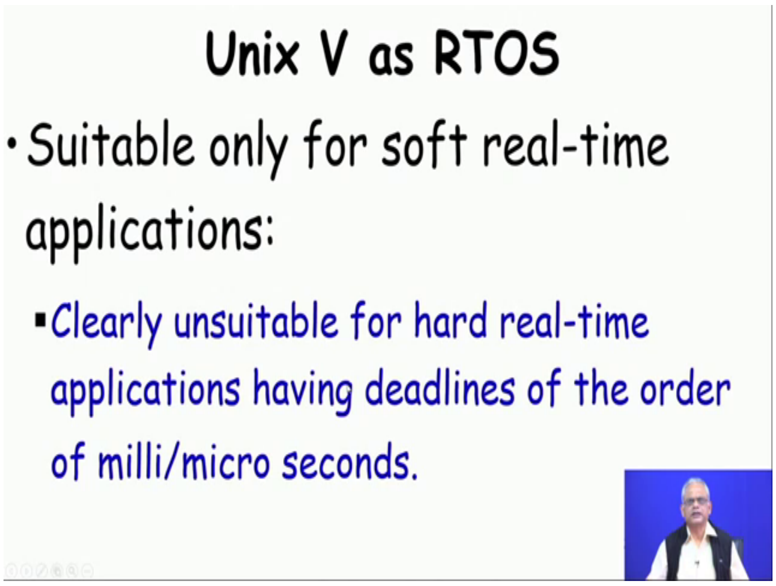
The Consequence

- I/O bound processes gravitate towards higher and higher priorities:
  - If a real-time task spends most of its time in computation:
    - It can miss its deadline.
  - This is unacceptable for hard real-time applications.

291

The I O bound processes gravitate to higher and higher priorities, but if we are trying to use such a operating system in real time situation, and the critical task is doing its processing using CPU heavily, then its priority will keep on decreasing and it can miss deadline and this becomes unacceptable in hard real time applications. So, we just saw two major problems of Unix: non preemptable kernel and dynamic priority levels.

(Refer Slide Time: 15:19)

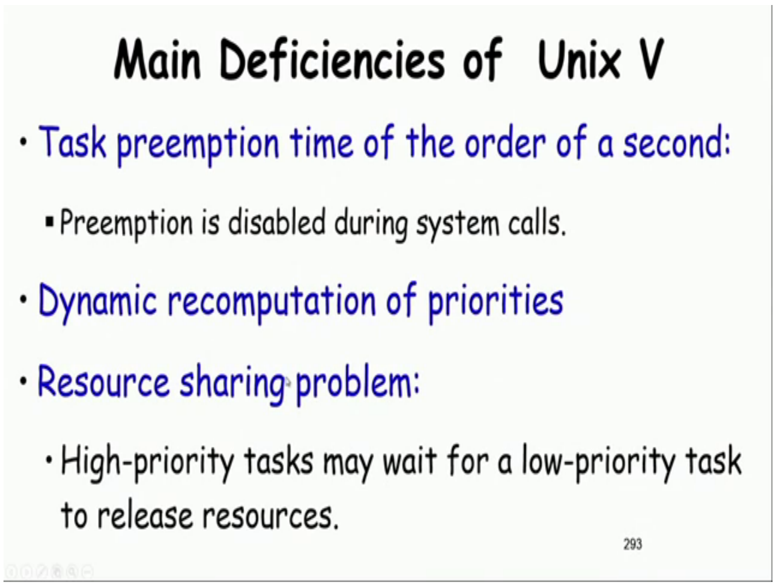


Unix V as RTOS

- Suitable only for soft real-time applications:
  - Clearly unsuitable for hard real-time applications having deadlines of the order of milli/micro seconds.

So, the Unix with its dynamic priority level and its non preemptable kernel can only be used in soft real time applications, its not suitable for hard real time applications where the deadline is the order of milli or micro seconds.

(Refer Slide Time: 15:41)



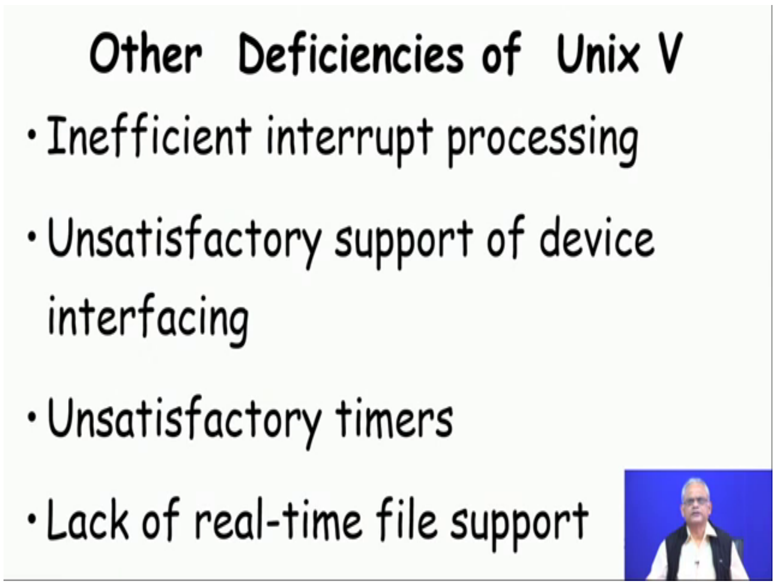
Main Deficiencies of Unix V

- Task preemption time of the order of a second:
  - Preemption is disabled during system calls.
- Dynamic recomputation of priorities
- Resource sharing problem:
  - High-priority tasks may wait for a low-priority task to release resources.

293

If we recapture the main deficiencies of Unix 5 task preemption time of the order of a second, dynamic recomputation priorities, resource sharing protocols are not supported.

(Refer Slide Time: 15:56)

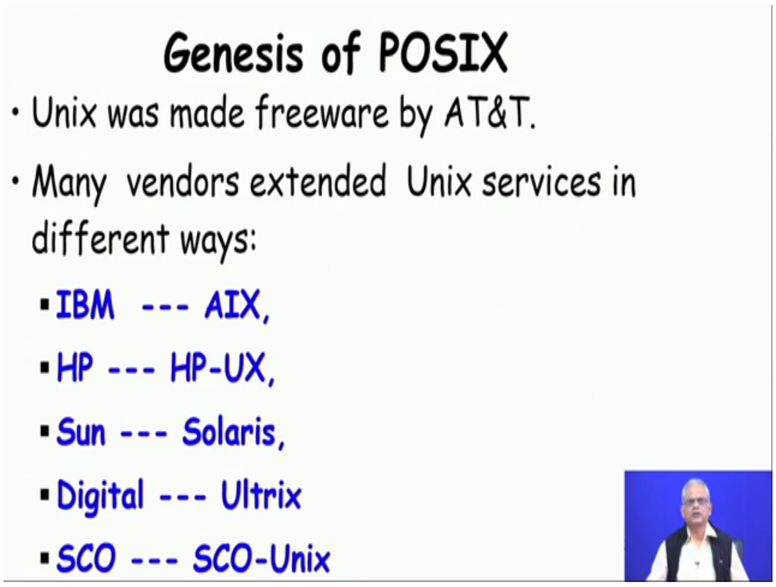


Other Deficiencies of Unix V

- Inefficient interrupt processing
- Unsatisfactory support of device interfacing
- Unsatisfactory timers
- Lack of real-time file support

Inefficient interrupt processing unsatisfactory support of device interfacing like you heard a new sensor, you want to incorporate a new sensor or a actuate a device and then you need to recompile the operating system stop all tasks shutdown the system etcetera that is not acceptable in a safety critical hard real time situation, you should be able to do it when the system is operational. Support for timers is unsatisfactory and lack of real time file support, these are main deficiencies the Unix 5if we want to use it for real time applications.

(Refer Slide Time: 16:49)



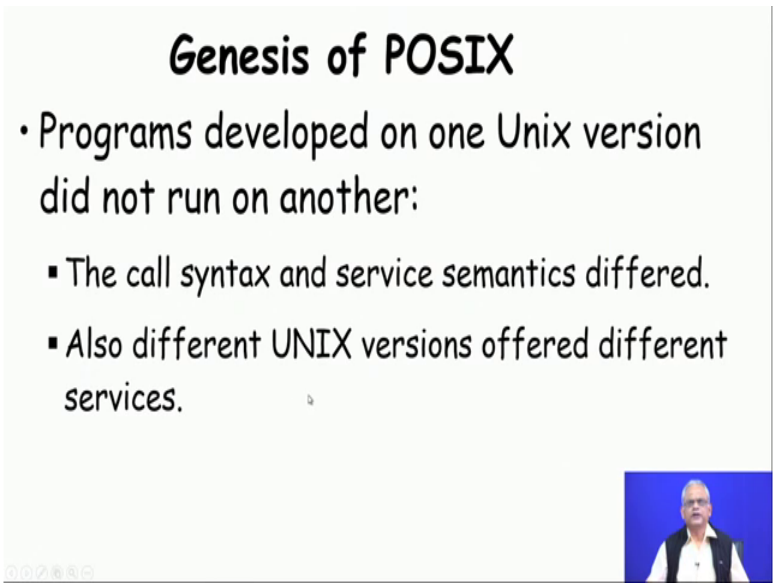
Genesis of POSIX

- Unix was made freeware by AT&T.
- Many vendors extended Unix services in different ways:
  - IBM --- AIX,
  - HP --- HP-UX,
  - Sun --- Solaris,
  - Digital --- Ultrix
  - SCO --- SCO-Unix

Now, let us look at one last issue before we conclude this course is PSIX. If we look at the origin of posix, it had to arise because Unix once it was developed by A T and T it gave it free to many recipients who ever want the Unix code was distributed by AT and T because AT and T was after a telecom company and then that time that point of time, it dint have interest in developing operating system, and those who received it they extended in various ways IBM into AIX HP into HP UX sun into Solaris the digital corporation into ultrix and so on. Even though the Unix code was developed at a t and t it was extended in various ways by different vendors and the result is that Unix code became incomputable.

If you run a application you write a application on one platform, it will not run on another platform.

(Refer Slide Time: 18:06)

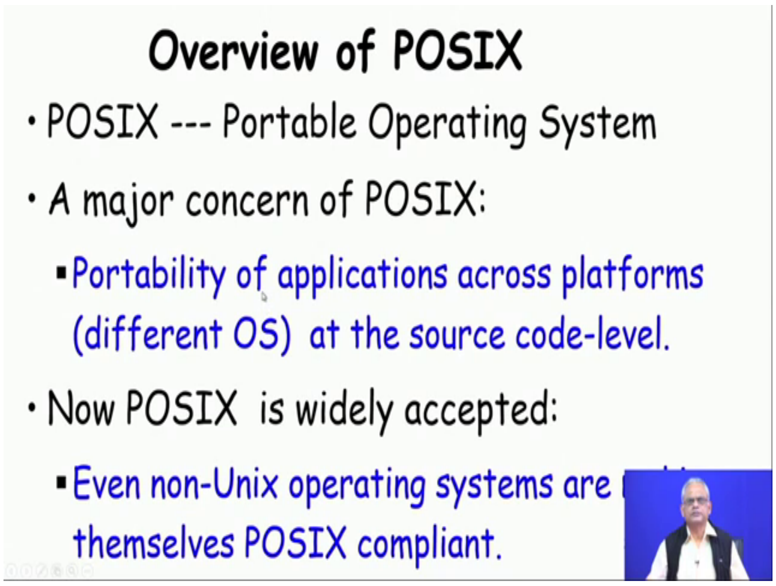


Genesis of POSIX

- Programs developed on one Unix version did not run on another:
  - The call syntax and service semantics differed.
  - Also different UNIX versions offered different services.

Because the syntax and the service semantics differed, different Unix was offered different services.

(Refer Slide Time: 18:17)

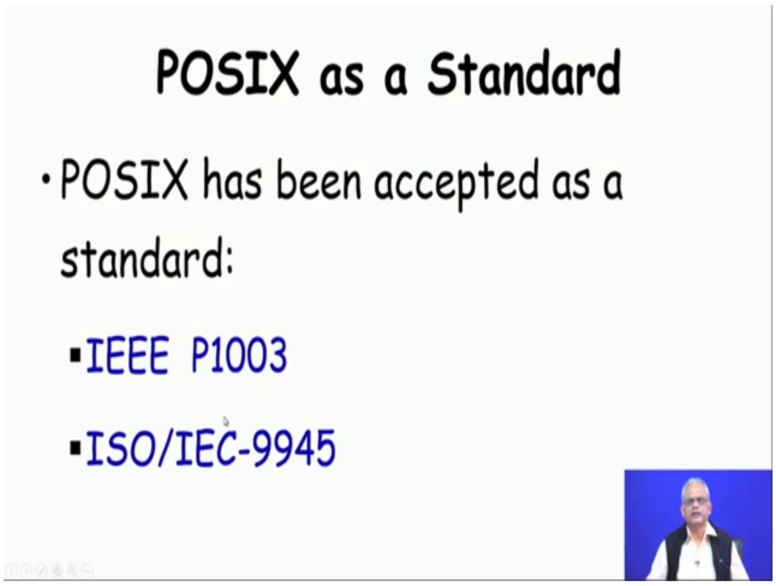


Overview of POSIX

- POSIX --- Portable Operating System
- A major concern of POSIX:
  - Portability of applications across platforms (different OS) at the source code-level.
- Now POSIX is widely accepted:
  - Even non-Unix operating systems are
    themselves POSIX compliant.

So, you develop code on one version do not work on another version. To make the code compatible at the source code level, the posix is attempted stands for portable operating system you might wonder that it comes to po p o s, but what about ix the ix was simply suffix to pos the portable operating system to make it look like Unix standard. But then it becomes so popular, that if the non-Unix operating system also become compatible to the posix and therefore, the posix has been a huge success the major concern for posix is portability of applications written in different operating systems. Now posix is widely accepted even non Unix operating systems use the posix standard.

(Refer Slide Time: 19:18)

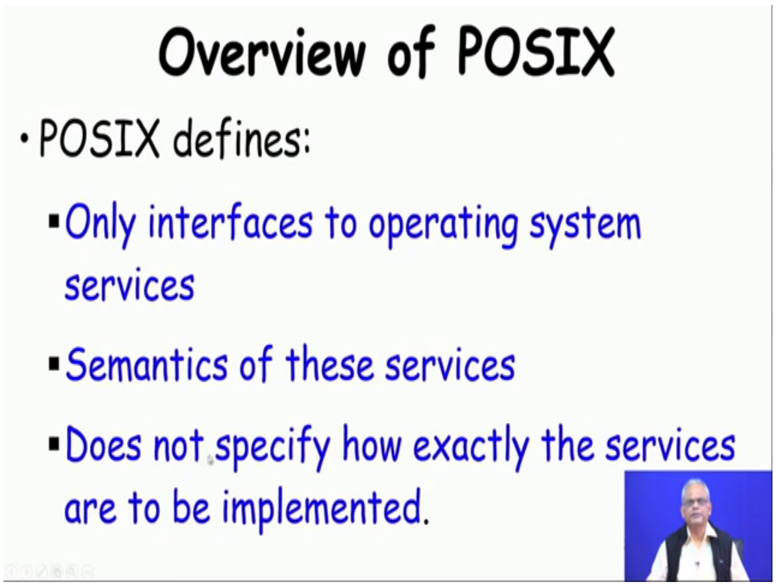


POSIX as a Standard

- POSIX has been accepted as a standard:
  - IEEE P1003
  - ISO/IEC-9945

The posix has been accepted as a international standard the I triple E P1003 the ISO standard 9943 etcetera they recognize posix is a standard.

(Refer Slide Time: 19:36)

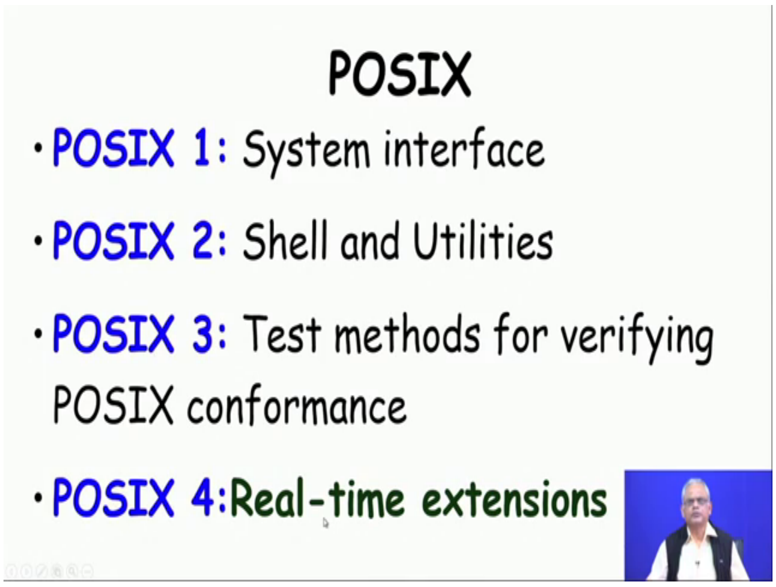


Overview of POSIX

- POSIX defines:
  - Only interfaces to operating system services
  - Semantics of these services
  - Does not specify how exactly the services are to be implemented.

If we look at the posix the standard document, will find that it defines only what should be the operating system calls or the interfaces to the operating system and what should happen once calls are made that is called as the semantics of the call. But it does not tell how the operating system calls are to be implemented, it just says that what parameters it needs to take what should be nom name of the calls and what it should achieve, but how exactly to be written it does not tell about that.

(Refer Slide Time: 20:12)

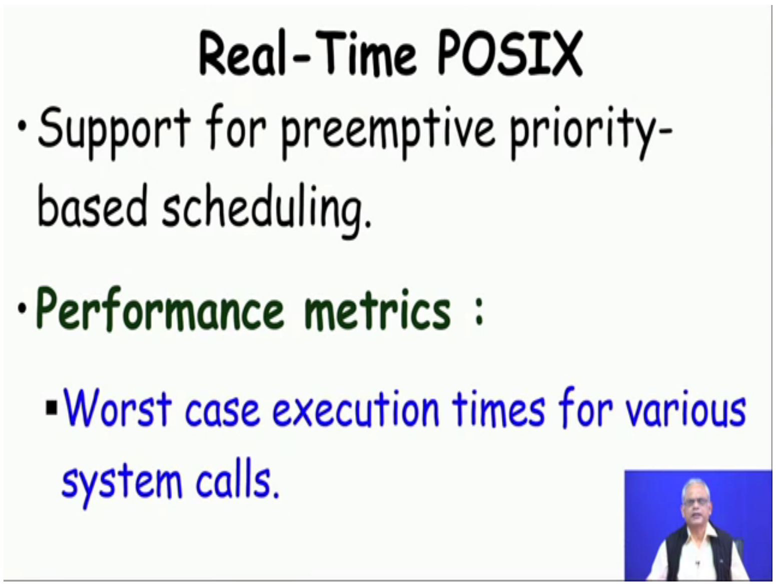


POSIX

- POSIX 1: System interface
- POSIX 2: Shell and Utilities
- POSIX 3: Test methods for verifying POSIX conformance
- POSIX 4:Real-time extensions

If we look at the posix there are several volumes of documents one is posix one which deals with system interface, that is what are the system routines and what are their parameters and the semantics what it does. Posix 2 on the shells and the utilities; posix 3 how you can test whether operating system is posix compliant, and fourth document deals with real time extensions the posix RT.

(Refer Slide Time: 20:41)

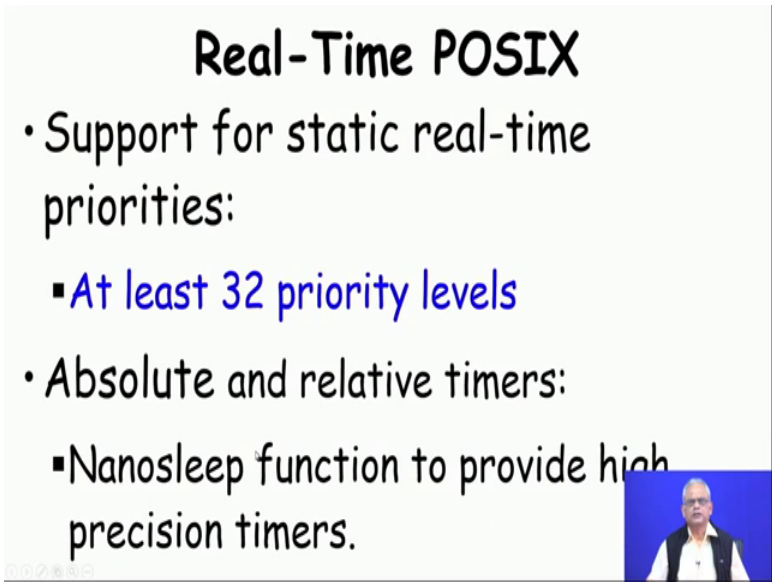


Real-Time POSIX

- Support for preemptive priority-based scheduling.
- Performance metrics :
  - Worst case execution times for various system calls.

Now, let us look at the real time posix, it requires an operating system to support certain features then only it will become RT posix complaint. The first one of course, is preemptive priority based scheduling and it defines some performance metrics who is the operating system must satisfy, that is the worst case bound on the execution time various system calls.

(Refer Slide Time: 21:15)

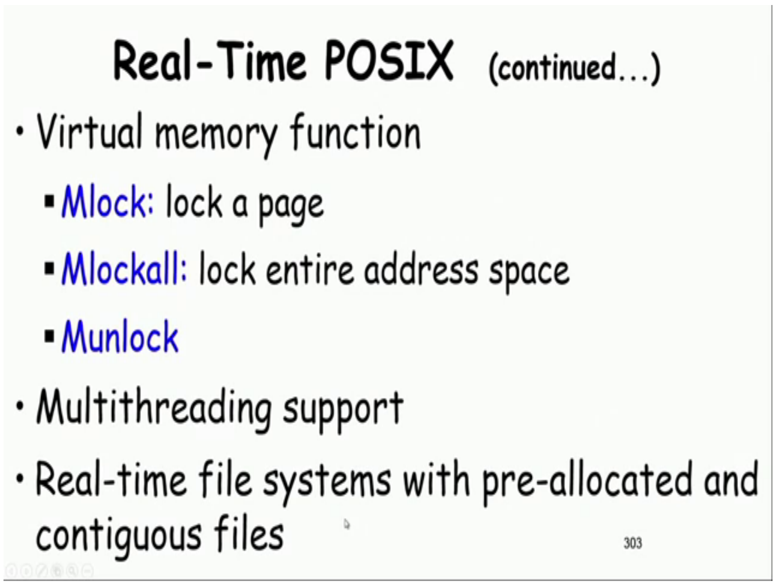


Real-Time POSIX

- Support for static real-time priorities:
  - At least 32 priority levels
- Absolute and relative timers:
  - Nanosleep function to provide high precision timers.

It requires support for static priorities, we had seen that the traditional operating system did not support static priorities, but now many of the traditional operating systems also support static priorities. It requires that at least 32 priority levels must be supported, both absolute and relative timers must be supported nanosleep function to provide high precision timers.

(Refer Slide Time: 21:47)



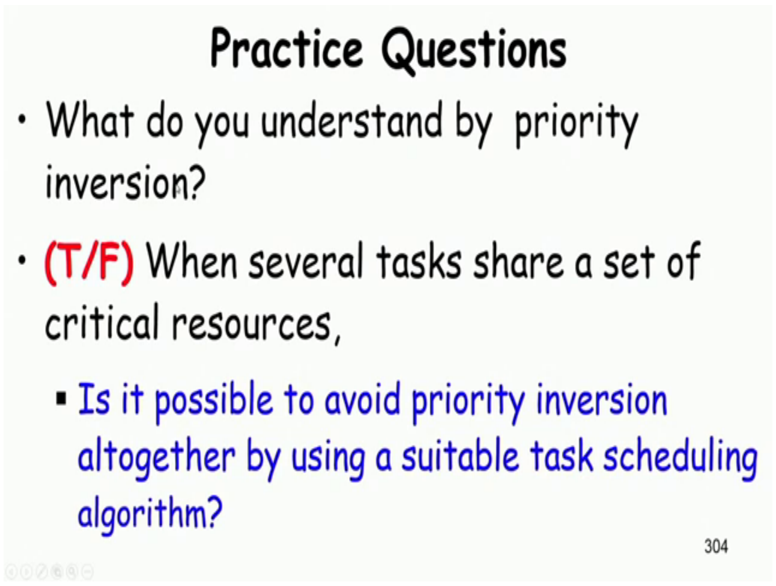
Real-Time POSIX (continued…)

- Virtual memory function
  - Mlock: lock a page
  - Mlockall: lock entire address space
  - Munlock
- Multithreading support
- Real-time file systems with pre-allocated and contiguous files

303

In the virtual memory M lock that is lock a page, M lock all to lock the entire address space, memory unlock that is to unlock a page and so on must be supported.

(Refer Slide Time: 22:16)



Practice Questions

- What do you understand by priority inversion?
- (T/F) When several tasks share a set of critical resources,
  - Is it possible to avoid priority inversion altogether by using a suitable task scheduling algorithm?

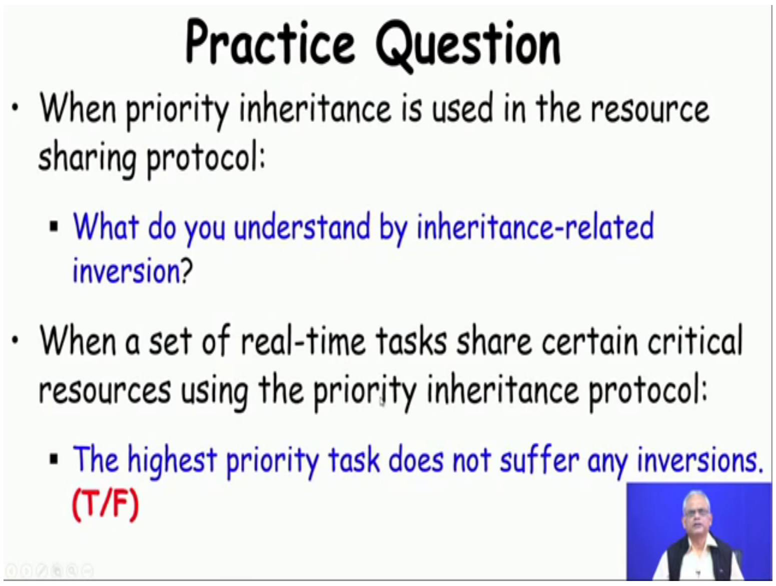
304

Multi-threading support real time file system support for supporting pre allocated and contiguous files must be supported. So, we just right now saw some issues if we want to use the traditional operating systems in real time applications we looked at Unix and then we looked at what problems may occur if Unix is used as a real time operating system and then we looked at a standard the posix real time standard, which all real time operating system must support, we looked at major requirements of the standard one of the stan requirement is static priority levels. 32 priority levels the scheduler should be implementable using preemptable priority based scheduling.

We are almost at the end of the course, now let us do a few practice questions before we complete the course. This is already we have discussed, but then just to check whether you have really followed the course. One of the crucial issue that arises when tasks share resources and we do not have adequate support for resource sharing is a priority inversion problem. What do you understand by the priority inversion problem, how does it arise and what are the solutions of this problem. Please go through the lecture material and try to answer this questions that what is the priority inversion the unbounded priority inversion and solution to this problem.

Now let us move to the next question when several tasks share a set of critical resources is it possible to avoid priority inversion all together can we really make priority inversion zero number of priority inversion zero, when we have tasks that share resources by using any protocol, any resource sharing protocol may be priority ceiling protocol may be highest locker or the simple inheritance based protocol the answer to this question is no because if a low priority task is holding the resource, then the high priority task has to wait and therefore, at least one priority inversion must occur and it is not possible whatever be the priority whatever may be the resource sharing protocol that we use, it is not possible to avoid prior priority inversions all together.

(Refer Slide Time: 25:38)



Practice Question

- When priority inheritance is used in the resource sharing protocol:
  - What do you understand by inheritance-related inversion?
- When a set of real-time tasks share certain critical resources using the priority inheritance protocol:
  - The highest priority task does not suffer any inversions. (T/F)

Let us look at further questions before we conclude this course you can ask specific questions for example, what do you mean by inheritance related inversion when using a priority inheritance scheme may be a priority ceiling protocol, what do you understand by a inheritance related inversion, give an example of a inherited inheritance related inversion. When a set of real time task share certain critical resources using the priority inheritance protocol is it true, that the highest priority task does not suffer any inversions let me repeat the question again. If a set of real time tasks that share critical resources and they are using the priority inheritance protocol, is it true that the highest priority task does not suffer any inversions the answer is false.

If we said the lowest priority task does not suffer any inversions that would be true, but the highest priority task could suffer inversion when a low priority task is already holding a resource and the high priority task becomes enabled. With this will conclude this course, we just looked at some basic aspects of real time operating systems. We saw how real time operating system differs from a non-real time operating system, we considered real time tasks scheduling as the central problem that a real time operating system needs to handle and we discussed the important task scheduling algorithm we looked at analysis of tasks to see whether they can be scheduled on a processer, we looked at resource sharing issues and then we looked at multi processer scheduling, and then finally, we have been looking at some issues that a real time operating systems needs to support the traditional operating systems lacks support for that we investigated why it was so, and then we explained or offered some solutions for that which all real time operating systems they do follow those.