Real-Time Systems

Prof. Dr. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture No. # 26
Real - Time POSIX. (Contd.)

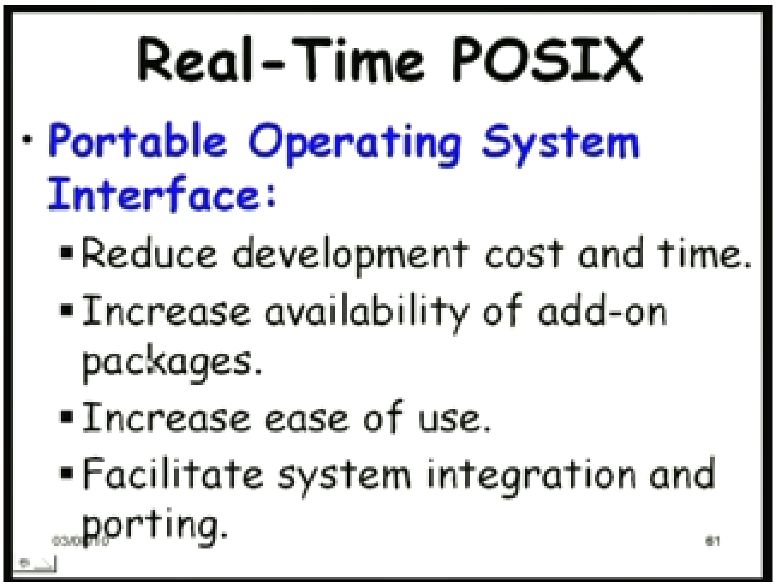
Ok Good morning, so let us get started. So far, we had discussed that the UNIX source code was given free by AT and T and that let to many versions of UNIX. So, one thing that you can notice is that UNIX has become extremely popular possibly because it is an open source. And also possibly it has many good features from which the other operating systems have, which were even not UNIX non UNIX operating system they have derived their features based on UNIX.

So, UNIX is very powerful in that sense and possibly one of the reasons for that it is being open source, besides of course, it having many good features. But, the open source UNIX let to many versions and the applications seems to be portable among the different versions even though all were variants of UNIX.

The applications written on one variant will not work on another variant and that was the origin of the POSIX. And POSIX we had seen that it is an interface; it is not an operating system by itself, it is an interface standard and all UNIX variants have to make themselves compliant to POSIX.

But later other operating systems non UNIX operating systems are also becoming POSIX compliant. So, that they become popular with the users they can run their applications written on POSIX platforms on those platforms as well. So, that was the discussion we had so far, now let us proceed from that point.

(Refer Slide Time: 02:22)


Real-Time POSIX

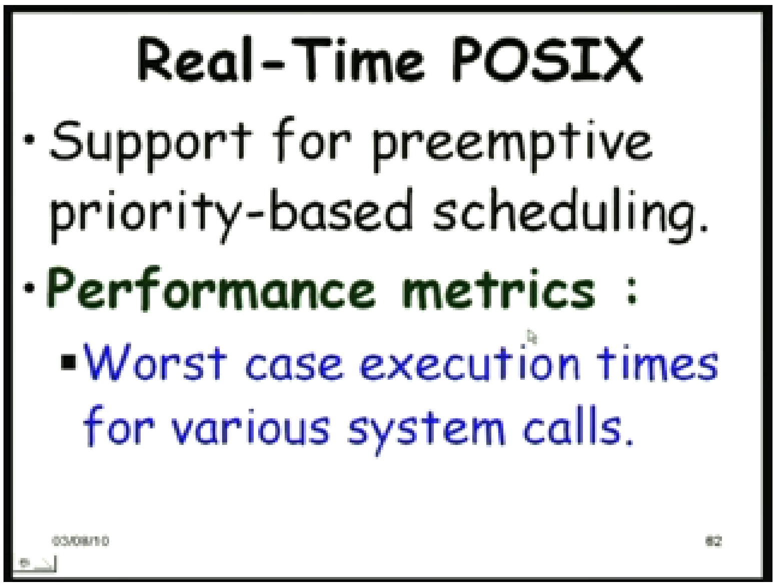
- Portable Operating System Interface:
  - Reduce development cost and time.
  - Increase availability of add-on packages.
  - Increase ease of use.
  - Facilitate system integration and porting.

So, we had said that the real-time POSIX stands for the POSIX stand for portable operating system interface. And we had said that as far as a programmer is concerned it reduces development cost and time, increase availability of add on packages. So, how does that happen reduce development cost and time, increase availability of add on packages, increase ease of use facilitate system integration and porting. So, how will that happen?

Sir, since the technical and all are now, published and well maintained. So, like developers independent developers can develop. And on some. So, basically the packages or the services developed for one operating system; one version of the operating system can run on the other operating system. Like you have developed some application somebody can use it on another platform also.

So, that is the reason for each of these reduced development cost of cost and time. You can use some other pre developed packages increase availability of add on packages, increases ease of use facilitate integration and porting.

(Refer Slide Time: 03:41)



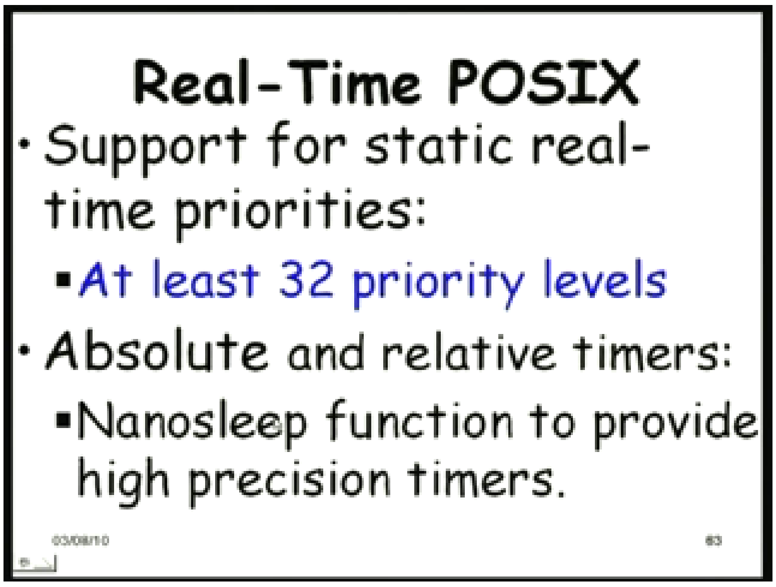
Real-Time POSIX

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

03/08/10 62

Now, let us see the features of real-time POSIX. One is that need to support preemptive priority - based scheduling, and we had seen that this is one of the basic requirements for any real-time operating system. And several performance metrics have been defined, the performance metrics are in terms of worst case execution times of various system calls. So, all the system calls are supported are listed and what is the worst case execution time for them to be real-time POSIX compliant are mentioned.

(Refer Slide Time: 04:29)



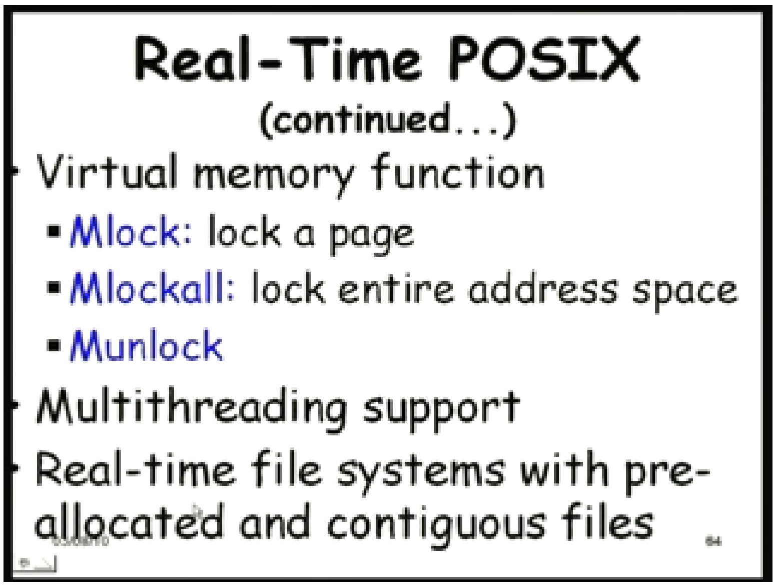
Real-Time POSIX

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

03/08/10 63

Support for static real-time priorities. So, we had discussed real-time priorities and had said that, that is one of the basic necessity for implementing EDF or even RMA, RMA or even EDF. The depending on the dead line and the task characteristics the priorities would be fixed by the programmer. The operating system should not change the priority, once it is assigned by the programmer. And the Real-time POSIX requires at least 32 priority levels 32 real-time priority levels that is a requirement, there has to be absolute and relative timers; both types of timers have to be supported. Nanosleep function to provide high precision; so, high precision timers with nanosleep function needs to be provided.

(Refer Slide Time: 05:34)



Real-Time POSIX
(continued...)

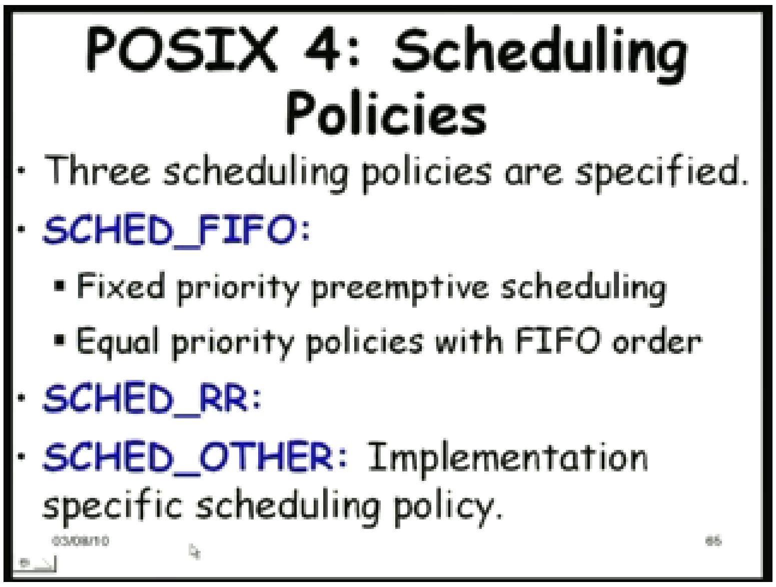
- Virtual memory function
  - Mlock: lock a page
  - Mlockall: lock entire address space
  - Munlock
- Multithreading support
- Real-time file systems with pre-allocated and contiguous files

64

And virtual memory is supported, but need memory locking. Lock a page, it is the name of the surface, M lock is the service to be provided which will lock a specific page or M lock all lock the entire address space and M unlock, M unlock all etcetera. Multi threading support is required.

Real-time POSIX and real-time file system with pre-allocated and contiguous file is required. So, we had said that real-time files provide deterministic data access. So, for that we need to have contiguous file blocks in the desk and pre-allocated files.

(Refer Slide Time: 06:35)



POSIX 4: Scheduling Policies

- Three scheduling policies are specified.
- SCHED_FIFO:
  - Fixed priority preemptive scheduling
  - Equal priority policies with FIFO order
- SCHED_RR:
- SCHED_OTHER: Implementation specific scheduling policy.

03/08/10 65

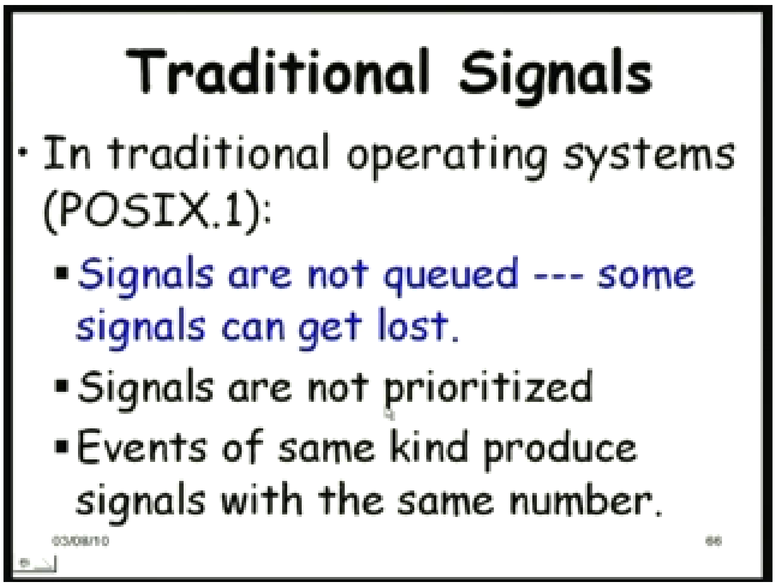
The scheduling policies that are mentioned in the POSIX 4 document, there are three main types: one is the SCHED FIFO; this is the fixed priority preemptive scheduling, we know all these terms. What it means? Fixed priority preemptive scheduling using which we can implement RMA.

Equal priority, if the tasks have equal priority than they will be handled in a FIFO order. So, the SCHED FIFO provides fixed priority preemptive scheduling and if tasks are assigned equal priorities then they will be handled in FIFO order, once a task completes the other task will be taken out at that priority level.

Then we have the policy SCHED Round Robin. So, this is similar to SCHED FIFO; fixed priority preemptive scheduling, except that for equal priority tasks they are handled in a round robin manner. So, SCHED FIFO and SCHED RR are both supported. One is equal priority, see for other tasks these two are same, but only for equal priority task they differ the SCHED FIFO and SCHED RR.

And SCHED other is if the programmer needs specific scheduling policy to be used, it he should be able to use it. SCHED other is implementation of any specific scheduling policy required by the programmer.

(Refer Slide Time: 08:16)



Traditional Signals

- In traditional operating systems (POSIX.1):
  - Signals are not queued --- some signals can get lost.
  - Signals are not prioritized
  - Events of same kind produce signals with the same number.

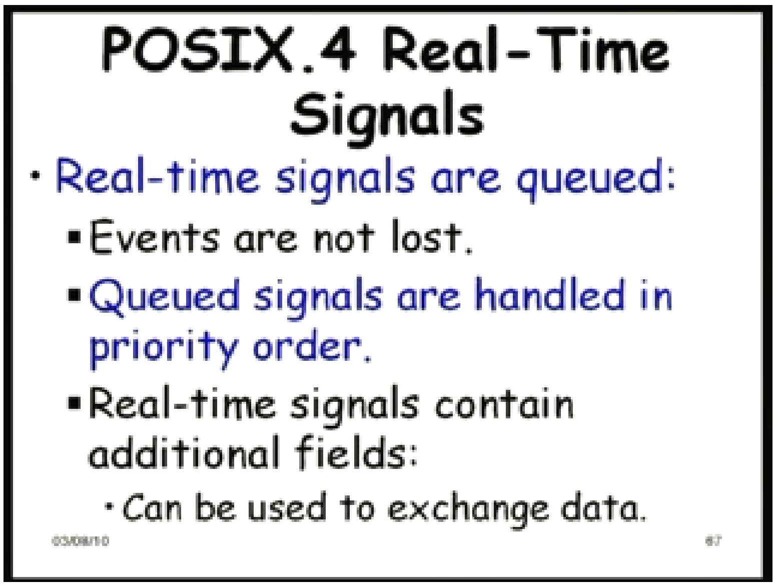
03/08/10 66

Now, the POSIX 4 also requires support for real-time signals, but before we understand real-time signal let us see, what is traditional signals? What is the shortcoming of this? Why isn't it useful for real-time applications?

In the traditional operating system for example, the POSIX-1 which is the interface specification for traditional operating system traditional UNIX based operating systems the signals are not queued, and as a result signals which occur before a signal is handled can get lost, and also signals are not prioritized obviously, they are not queued and therefore, they are not prioritized.

So, once one signal is being handled another signal will be ignored, even if it is a higher priority. Events of same kind produce signal with the same number this is also another characteristic of the traditional system, for example; if you press control D the signal is the same, control Z or something signals are similar or a kill signal or something.

(Refer Slide Time: 09:42)



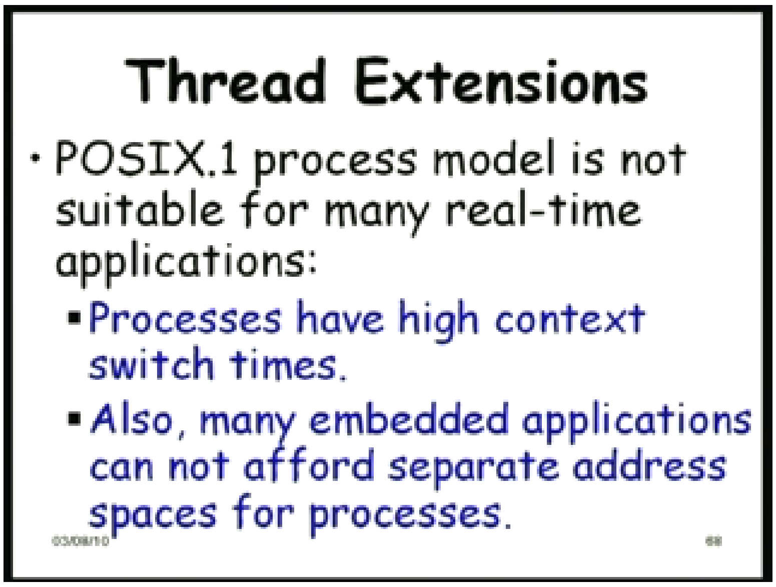
POSIX.4 Real-Time Signals

- Real-time signals are queued:
  - Events are not lost.
  - Queued signals are handled in priority order.
  - Real-time signals contain additional fields:
    - Can be used to exchange data.

03/08/10 67

In a real-time signal the signals have to be queued. So, those events will not get lost and the queued signals are handled in the priority order depending on the application generating the signal, and also these contain additional fields which are used to which can be used to exchange data. It just not signals, but also accompanying data.

(Refer Slide Time: 10:20)



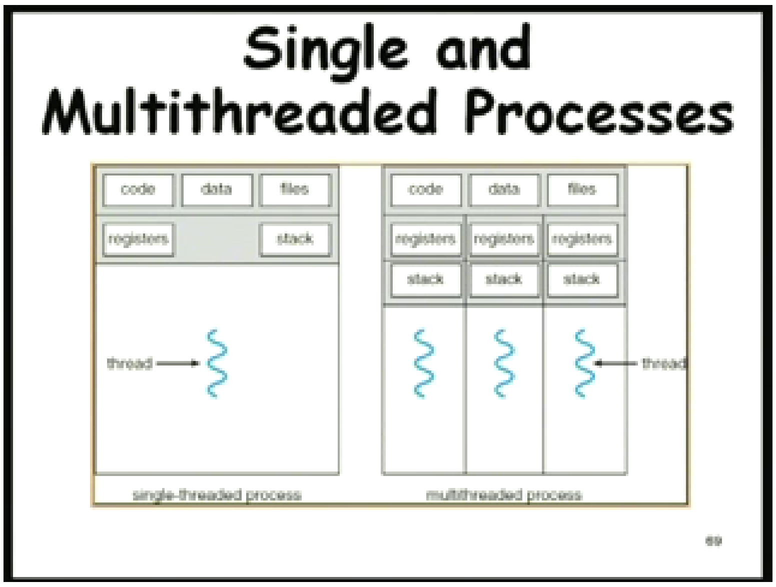
Thread Extensions

- POSIX.1 process model is not suitable for many real-time applications:
  - Processes have high context switch times.
  - Also, many embedded applications can not afford separate address spaces for processes.

03/08/10 68

So, real-time signals are a requirement in POSIX 4 and also the third extensions. The POSIX-1 process model is not suitable for many real-time applications, because processes have high context switch times as some of you are mentioning, not only the high context switch times, but many embedded applications they cannot afford separate address spaces for processes, because even though the data is used by different processes the same, and the code is the same they do not share it. Each one has its own address space; each process has its own address space. The code may be replicated; if the two processes have the same code the code is replicated.

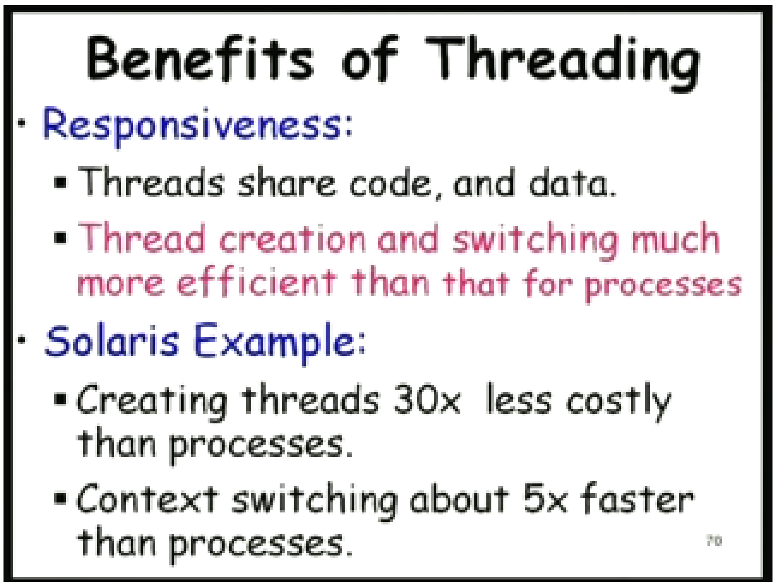
(Refer Slide Time: 11:11)



Single and Multithreaded Processes

So, this is basically the situation in a single thread; the code data, open files, registers, stack, are all private to a process and has one thread. On the other hand in a multithreaded process, we have the code data and open files are shared among all processes. See here, these are all shared, what is not shared among threads is the register's and the stack; they have independent registers and stack.

(Refer Slide Time: 11:58)



Benefits of Threading

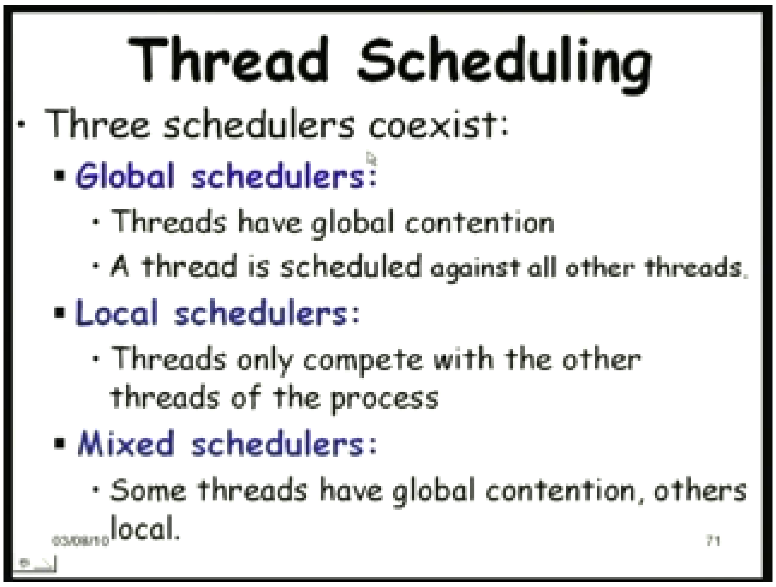
- Responsiveness:
  - Threads share code, and data.
  - Thread creation and switching much more efficient than that for processes
- Solaris Example:
  - Creating threads 30x less costly than processes.
  - Context switching about 5x faster than processes.

70

But the memory is basically shared, the code part, the data part, and the open files, are all shared among multiple threads. know that for I think some some of you already know for a real-time application threads are beneficial one is responsiveness, because threads share code and data. When you create a new thread it takes very less time, because you do not have to copy all that code and data into the memory. The same data code that is available on the memory is being used, what are needed is the registers and the stack.

So, thread thread creation and switching is much more efficient, it is after all the same code and data is being used. So, creation and switching is very efficient, just to given an example: in the solaris operating system creating a thread is 30 times less costly compared to a process creation, and context switch is about 5 times faster than a process.

(Refer Slide Time: 13:06)



Thread Scheduling

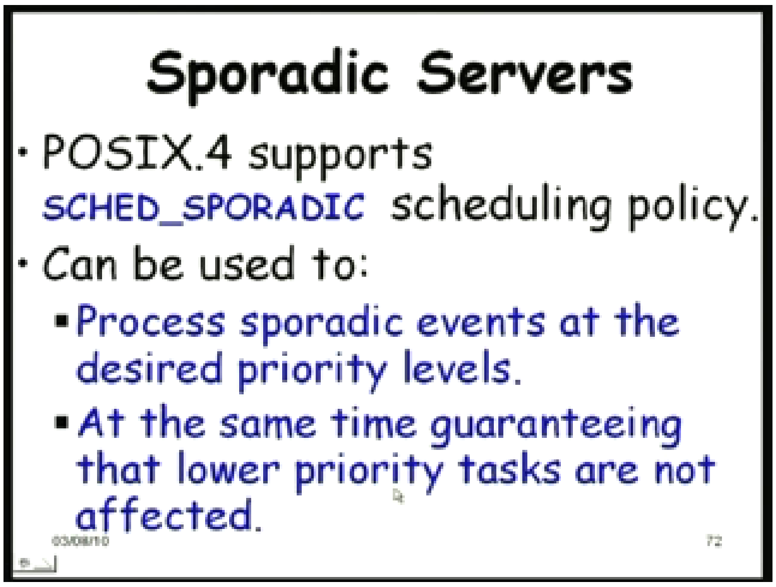
- Three schedulers coexist:
  - Global schedulers:
    - Threads have global contention
    - A thread is scheduled against all other threads.
  - Local schedulers:
    - Threads only compete with the other threads of the process
  - Mixed schedulers:
    - Some threads have global contention, others local.

03/08/10 71

Three types of schedulers are supported in the POSIX 4: one is the global schedulers, and the other is local schedulers, and the other the last one is mixed schedulers. Three different types of schedulers can coexist in the operating system.

In a global scheduler; when it is used, a thread competes with all other threads; thread is scheduled for the processor it competes with all other threads. In a local scheduler the thread competes only with other threads of the same process. Whereas, in a mixed scheduler; some threads have global contention, and others can have local, you can define this; you can define a global scheduling policy or a local scheduling policy or a mixed scheduler. So, that is another requirement for POSIX 4.

(Refer Slide Time: 14:14)



Sporadic Servers

- POSIX.4 supports SCHED_SPORADIC scheduling policy.
- Can be used to:
  - Process sporadic events at the desired priority levels.
  - At the same time guaranteeing that lower priority tasks are not affected.

03/08/10

72

It also POSIX 4 supports sporadic servers. We had seen a sporadic server is not it? We had discussed. What is sporadic server do you remember?

(( )) used handle (( )) regularly checks the event of task arrival the the time interval between two task equal for all like can occur (( ))

No, that is the sporadic task, that what you are saying is the definition of a sporadic task where it is not periodic, right? It can occur in some statistical pattern, but what is a sporadic server.

(( ))

Anybody remember that is

Any sporadic tasks are arriving and excess of decades it will allow it to be scheduled.

You remember partially, but what is the full answer?

(( )) check the (( )) indication and if it is used then it will regenerate.

But for what purpose is it used?

To settle the sporadic task, say sporadic can non periodic.

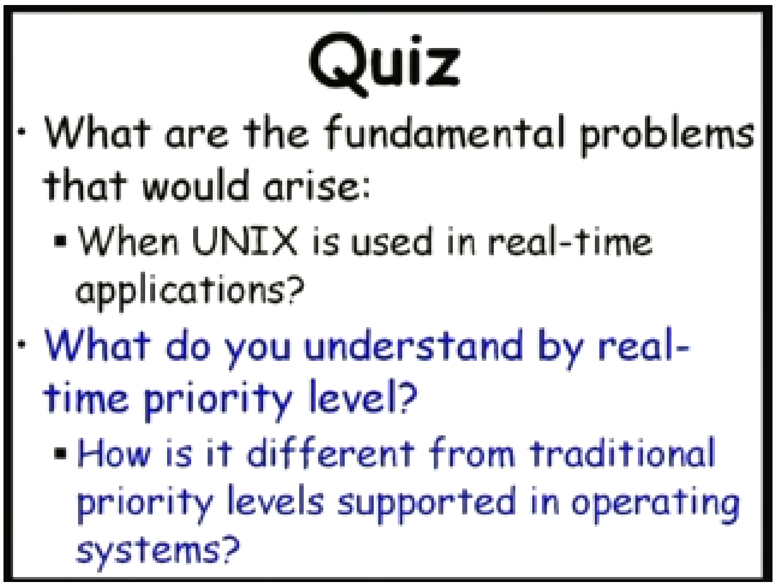
See, it is used the sporadic servers are used for scheduling sporadic tasks. See, one way the sporadic tasks can be handled is that if the CPU is unused for periodic tasks like some of the periodic tasks did not occur, they did not use their slot, completed early, and no tasks are getting executed. So, the lowest priority task can be the sporadic tasks they just lack of the CPU time when no other task is ready to use that is one way to implement, but at this the sporadic tasks executed the lowest priority, but for many applications the sporadic tasks needs to also be operated on some priority because there can be importance sporadic events.

For example let us say a condition needing some attention and that condition does not occur periodically, but it can occur once in a while right. So, some sporadic tasks need to be operated at a priority level, but at the same time we will have to ensure that by giving a priority to a sporadic task it does not really negatively influence the performance of the lower priority task.

See higher priority task it cannot influence, but just because it was given a priority it should not know suddenly some ten sporadic tasks occurred in some statistical pattern and they just lapped up the CPU time and the lower priority tasks they started missing their dead line that should not occur. So, that is the main reason for the sporadic server is that the sporadic tasks can operate in some priority level, but at the same time ensure that lower priority tasks are negatively affected when too many sporadic event occur that is the main idea here.

We had discussed about sporadic servers, if you can recollect the earlier discussions. So, these are used to process sporadic events at any desired priority level and at the same time guaranteeing that lower priority tasks are not negatively affected when too many sporadic events occur. Is that ok? So, the sporadic servers are needs to also be supported under POSIX 4.

(Refer Slide Time: 18:10)


Quiz

- What are the fundamental problems that would arise:
  - When UNIX is used in real-time applications?
- What do you understand by real-time priority level?
  - How is it different from traditional priority levels supported in operating systems?

So, let me just ask few questions. So, what is the main problem when UNIX is used in real-time application? What are the main problems that you will face if you attempt to use a UNIX system in a real-time application?

Doesn’t support the resource sharing.

Sorry.

Resource sharing.

Resource sharing among real-time tasks is a problem, what problem will it cause?

Priority incorrect that higher task priority task might.

Priority inversion; so, it can cause unbounded priority inversions in UNIX, among resources shared, among real-time tasks that is one problem, what are the other problems?

No no Real-time ah.

UNIX does not have a real-time priority support all are dynamic priority levels, what what else? Are there any other fundamental problems which cannot be?

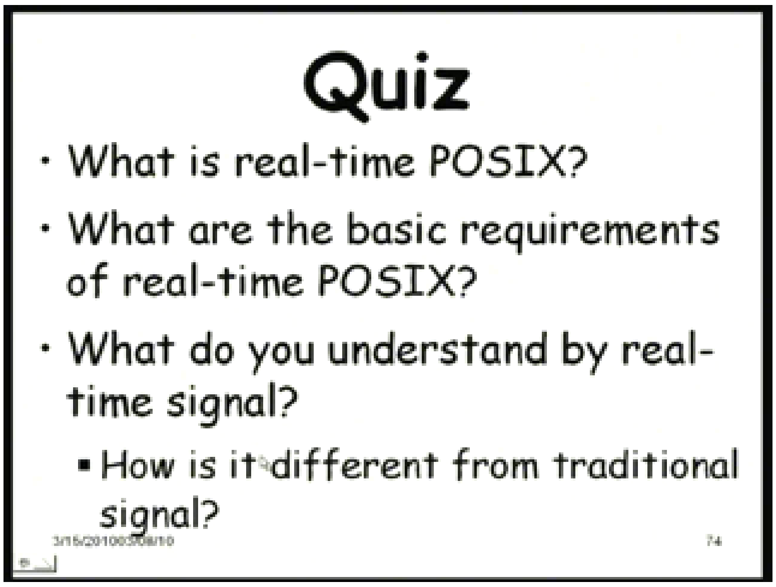
The Real-time files support

Real-time files support you can add, contiguous allocation or pre-allocation of blocks that you can do.

In kernel mode the process are rigid

Yes, that is an important problem of UNIX that it is non-preemptable in the kernel mode. Once a user task makes a system call becomes non-preemptable and therefore the response Time for a critical event can be up to a second, can be more than a second also. So, those are the main problems of UNIX we were discussed about that. This we know, what is a real-time priority level? Static priority level? And how it is different from the traditional priority level? We know that, let me not.

(Refer Slide Time: 20:24)



Quiz

- What is real-time POSIX?
- What are the basic requirements of real-time POSIX?
- What do you understand by real-time signal?
  - How is it different from traditional signal?

3/15/2010 03/08/10 74

So, what is a real-time POSIX?

POSIX .4 (( )).

Anybody likes to answer differently, what is real-time POSIX?

So, it is memory lock

No, no it's just a one feature, but generally somebody asks you that see, you have done this course heard this term real-time POSIX. So, what do you mean? You are calling in your company might as you that see, what is this real-time POSIX? Can you just tell me?

Dead line exchanges to

Anybody else would like to answer this question?

Supports zero time task priority

No, this is specific features. He is saying memory locking, you are saying task priority, real-time priority, etcetera. These are specific features of the; or specific requirement of the real-time POSIX, but what is it? What is real-time POSIX?

(( ))

it is the It defines the basic services that need to be supported by an operating system to be real-time POSIX compliant; or these are the features standard features that are expected of a real-time operating system. Now, do you remember the basic requirements? Some of you are saying about memory locking in a virtual memory system, you are saying real-time priority level. What are the other requirements for a real-time POSIX?

Remember any of the requirements of real-time POSIX? Yes, two you had identified, he had identified memory locking support, and he had identified real-time priority levels and 32 of them, besides that? 32 priority levels you are saying, other than that are there any other requirements of real-time POSIX?

(( )) provided by nano (( ))

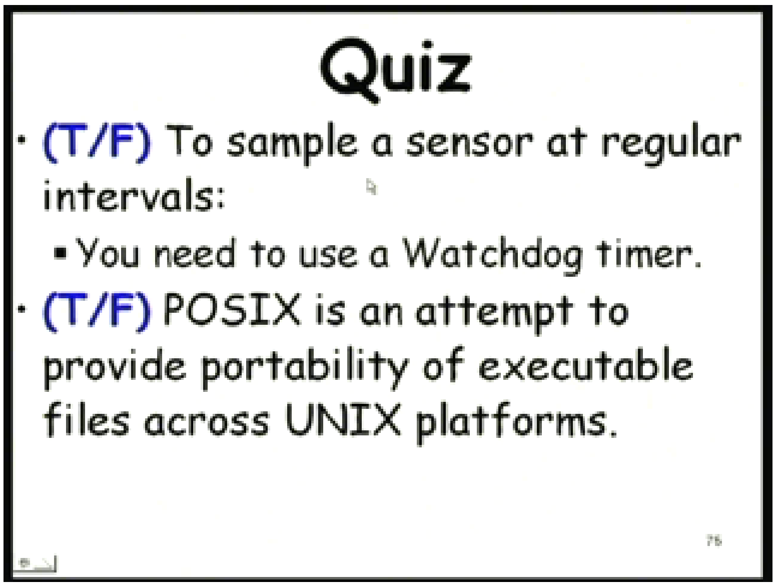
So, the timers periodic, asynchronous timers, periodic aperiodic timers, the nano sleep function, real-time signals, real-time files, and the different scheduling policies.

And then it can support for the static real-time priorities.

That he has said already, preemptive priority based scheduling policy and So on. So, how is the real-time signal different from a traditional signal in operating system?

Events are not queued, events cannot get lost this can have data associated with it and they can have priorities.

(Refer Slide Time: 23:52)



Quiz

- (T/F) To sample a sensor at regular intervals:
  - You need to use a Watchdog timer.
- (T/F) POSIX is an attempt to provide portability of executable files across UNIX platforms.

So, just tell me whether these statements are true or false? Now, as a programmer you want to sample a sensor a temperature sensor; or a pressure sensor something at regular intervals. So, would you use a real watchdog timer for this work?

Event true

Yes, this is false; watchdog timer is a one such timer for a periodic event regular interval you need to use a periodic timer.

POSIX is an attempt to provide portability of executable files across uniform platform UNIX platforms. What do you think is it this true or false?

False

False! So, what is true?

Provides an open source all source code

Yes, the main focus of POSIX is to provide portability at the source code level not at executable code level.

Sir

Yes.

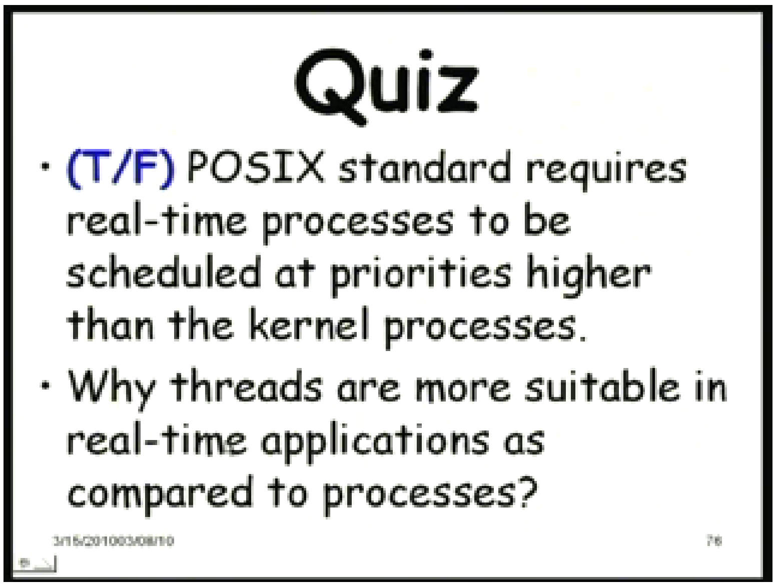
So, in that first question for that (( ))

Yes

We would like during incrementation also we can even that the given that, the watchdog setting and resetting of watchdog timer is like (( )) can be ignored. So, you can set and then reset and then again set the watchdog timer that will be (( )).

No, see for a regular handling of a event you just set a cyclic a periodic timer, and watchdog timer the purpose is different, here you just time out and the process.

(Refer Slide Time: 25:50)



Quiz

- (T/F) POSIX standard requires real-time processes to be scheduled at priorities higher than the kernel processes.
- Why threads are more suitable in real-time applications as compared to processes?

3/15/201003/08/10 76

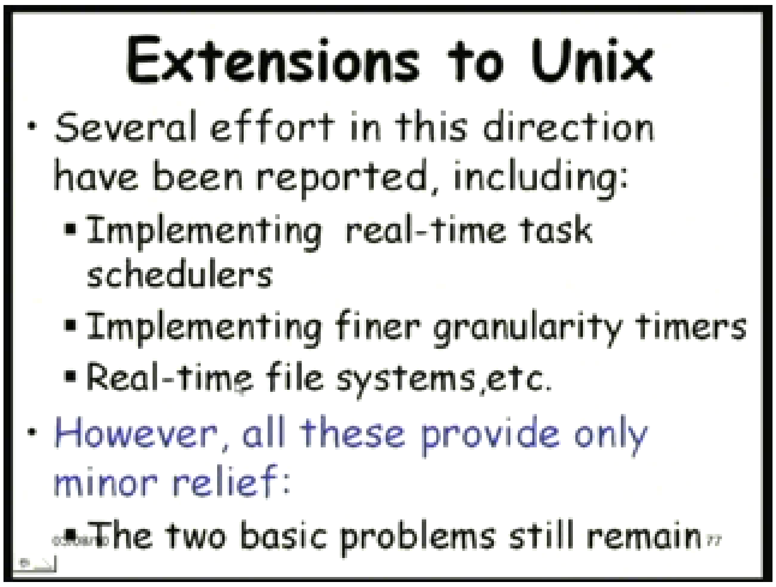
What about this statement POSIX standard requires real-time processes to be scheduled at priorities higher than the kernel processes? Do you agree with this? POSIX standard requires real-time processes to be scheduled at priorities higher than the kernel processes.

False

It provides real-time priority levels where you can make it operate higher than kernel processes, but it is not required to make it operate always higher than the kernel processes. So, real-time processes to be scheduled implicitly it says that always there are scheduled at priorities higher than the kernel processes, which is false.

Why threads are more suitable in real-time applications as compared to using processes? I think that you had answered; that creation, context switches, etcetera. Are faster.

(Refer Slide Time: 27:03)



Extensions to Unix

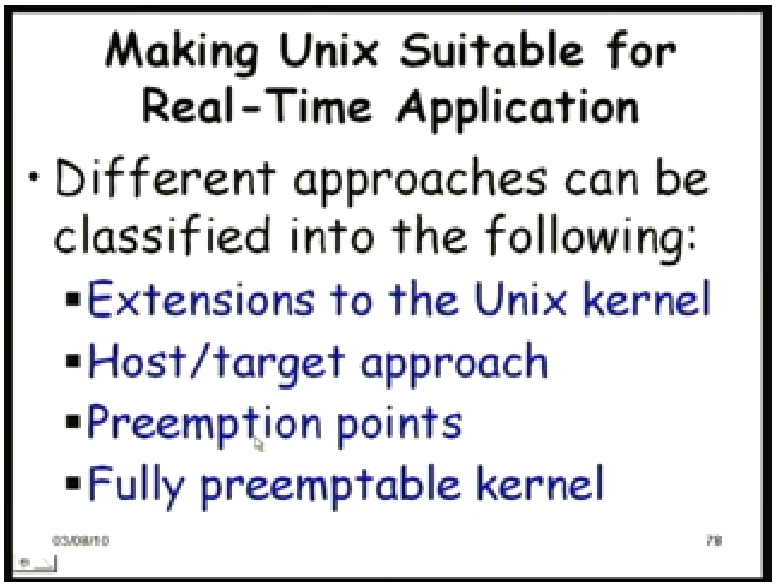
- Several effort in this direction have been reported, including:
  - Implementing real-time task schedulers
  - Implementing finer granularity timers
  - Real-time file systems,etc.
- However, all these provide only minor relief:
  - The two basic problems still remain

Now, let us look at how the UNIX has been extended? We had seen the problems with UNIX as well as Real real-time application development are concerned. So, let us see the different ways in which the vendors have extended UNIX to make it work in a real-time and in an embedded application.

So, several efforts have been reported, some have reported implementing real-time task scheduler, finer granularity timer, real-time file system, this is some basic extensions to the UNIX kernel; simple extensions. Some time back we were even giving this as project work for UG and PG students to provide certain facilities to UNIX, add certain facilities to the UNIX kernel. So that, they support some specific features.

But all these features real-time task scheduler, finer granularity timer, real-time file system, etcetera. They provide only a minor relief, because the fundamental problems are there still. The non-preemptable kernel resource, poor resource sharing support and so on.

(Refer Slide Time: 28:39)



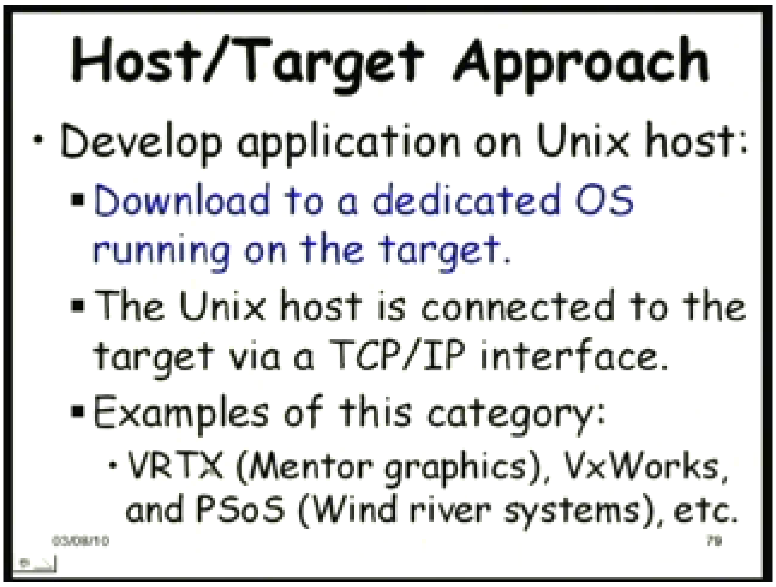
Making Unix Suitable for Real-Time Application

- Different approaches can be classified into the following:
  - Extensions to the Unix kernel
  - Host/target approach
  - Preemption points
  - Fully preemptable kernel

03/08/10 78

Several operating system; open source as well as commercial operating system have appeared. Which attempt to make UNIX suitable for real-time applications. One is extensions to the UNIX kernel, the host target approach, using preemptions points, and then, the fully preemptable kernel. We will just look at these 4 types of approaches. And then we look at specific operating systems.

(Refer Slide Time: 29:22)



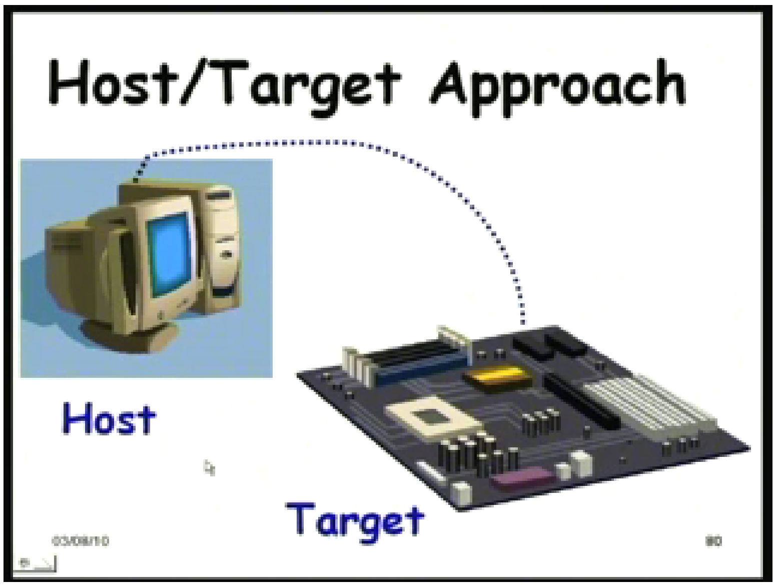
Host/Target Approach

- Develop application on Unix host:
  - Download to a dedicated OS running on the target.
  - The Unix host is connected to the target via a TCP/IP interface.
  - Examples of this category:
    - VRTX (Mentor graphics), VxWorks, and PSoS (Wind river systems), etc.

03/08/10 79

In the host/target approach the application is developed on a UNIX host, of course, may be non UNIX host also, and then, the application is downloaded to a dedicated operating system running on the target; very simple operating system. Basically runs the tasks rather than supporting development of the application. And the UNIX host is connected to the target via a TCPIP interface where you download the application, important examples are PSoS and VRTS, etcetera.

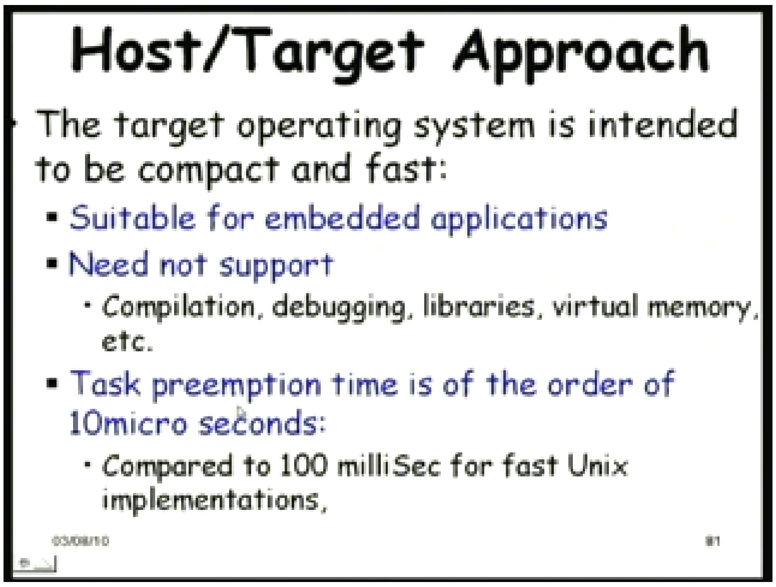
(Refer Slide Time: 30:05)



So, this is the scenario in a host target approach. You have a processor card here to be used in a embedded application, the processor memory, etcetera. Here the extension slots, and then, this is the host running UNIX; or any other operating system. Where you develop the application and then download it on to the ram here, and then you test it. It provides several facilities for running the tasks on the board, and test it you can debug it also.

So, here there is a component which sends the status of various components, and the data values, can be monitored from the host once you are satisfied with the development software, you fuse it onto a flash memory; or a RAM. And then this is used in the embedded application

(Refer Slide Time: 31:39)



Host/Target Approach

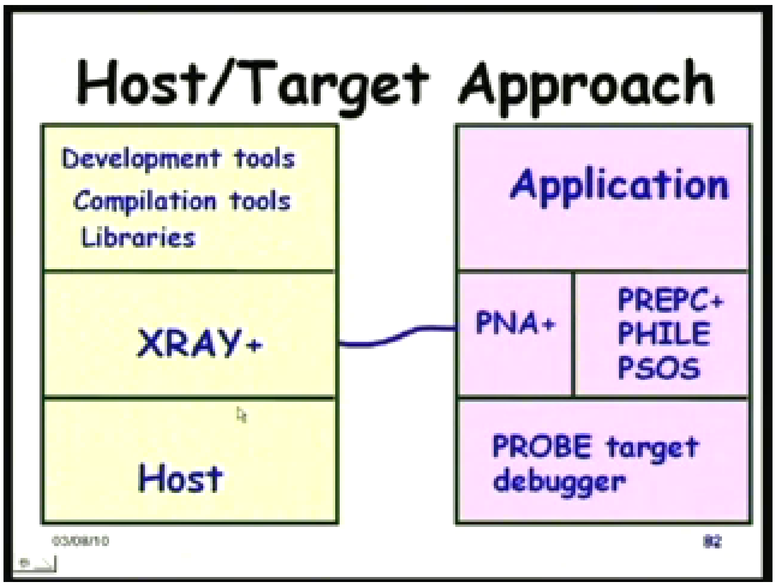
The target operating system is intended to be compact and fast:

- Suitable for embedded applications
- Need not support
  - Compilation, debugging, libraries, virtual memory, etc.
- Task preemption time is of the order of 10micro seconds:
  - Compared to 100 milliSec for fast Unix implementations,

03/08/10 81

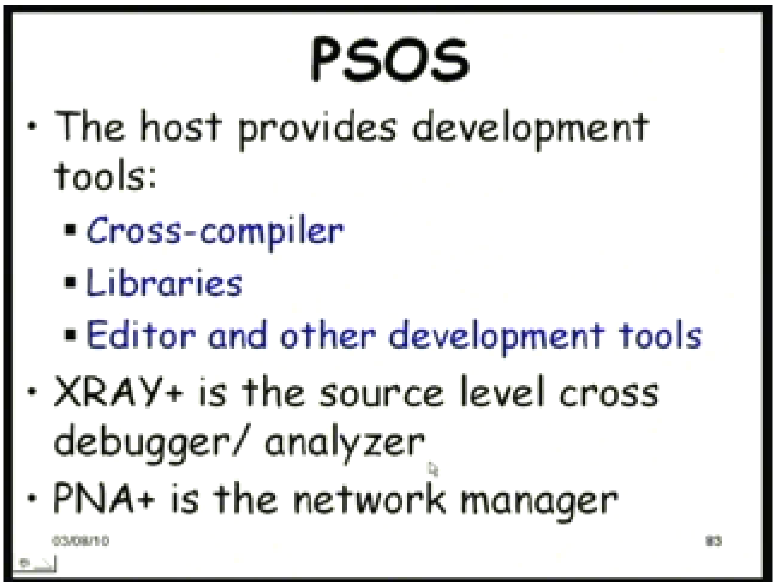
So, the development environment is on the host, the target has only the running the execution environment. The target operating system is very small and fast, it need not support compilation, debugging, libraries, virtual, see all these are development tools, right? You do not need to have compiler, bulky software, debugging support, libraries, virtual memory support, and etcetera. Because unless you run bulky applications like compilation, etcetera. You do not need virtual memory for simple embedded applications. Host/target approach is satisfactory, and the task preemption time is of the order of 10 microseconds compared to milliseconds 100 of milliseconds for UNIX implementations.

(Refer Slide Time: 32:34)



So, this is the scenario for the PSoS operating system, on the host side you have the development tool, compilation tools, etcetera. And there is a component of the PSoS operating system called as x ray, which helps you to download on to the target board, and also help debug it. So, there is a PSOS network component here, and the target debugger where you can set debugging commands from here and get the results, you can set break points, and so on.

(Refer Slide Time: 33:18)



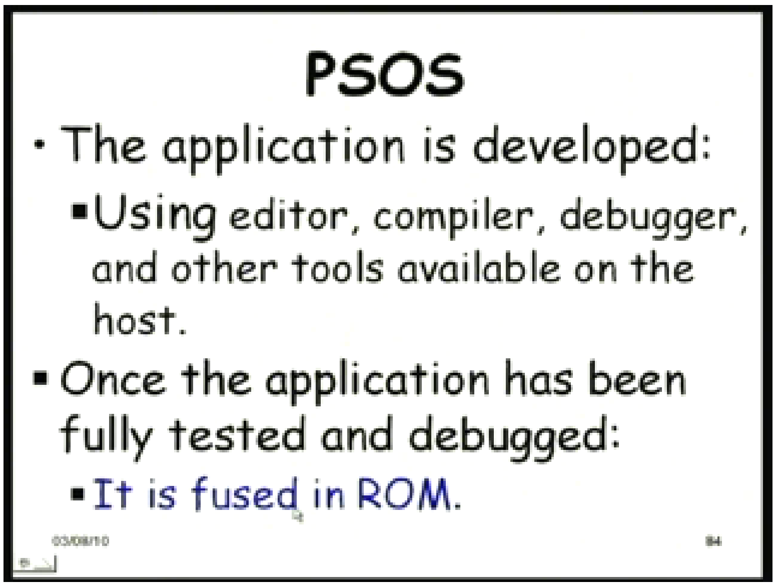
PSOS

- The host provides development tools:
  - Cross-compiler
  - Libraries
  - Editor and other development tools
- XRAY+ is the source level cross debugger/ analyzer
- PNA+ is the network manager

03/08/10 83

The host provides tools like cross compilers, libraries, editors, and other development tools. And you use the debugger on host side to debug the target board; the application running on the target board.

(Refer Slide Time: 33:46)



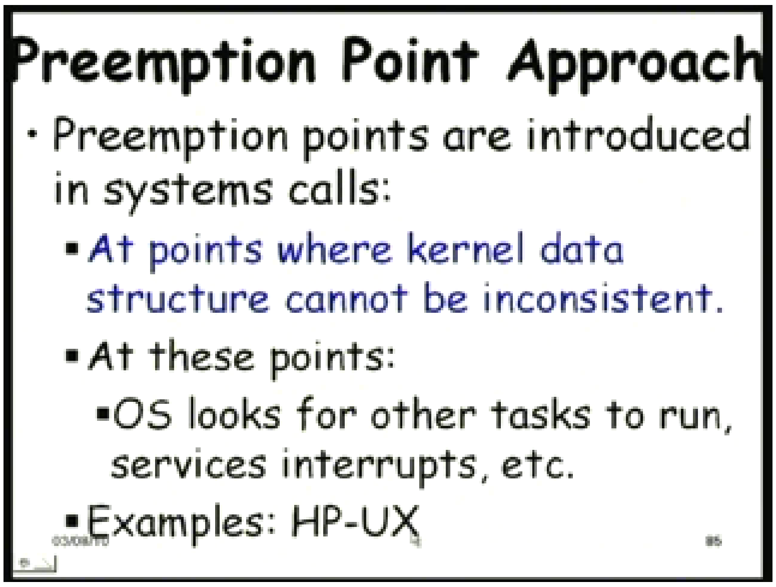
PSOS

- The application is developed:
  - Using editor, compiler, debugger, and other tools available on the host.
- Once the application has been fully tested and debugged:
  - It is fused in ROM.

03/08/10 84

Once the application is developed the tools that are there in the host are not necessary, you just once it is fully tested and debugged you just fuse it on a ROM; or a flash.

(Refer Slide Time: 34:02)



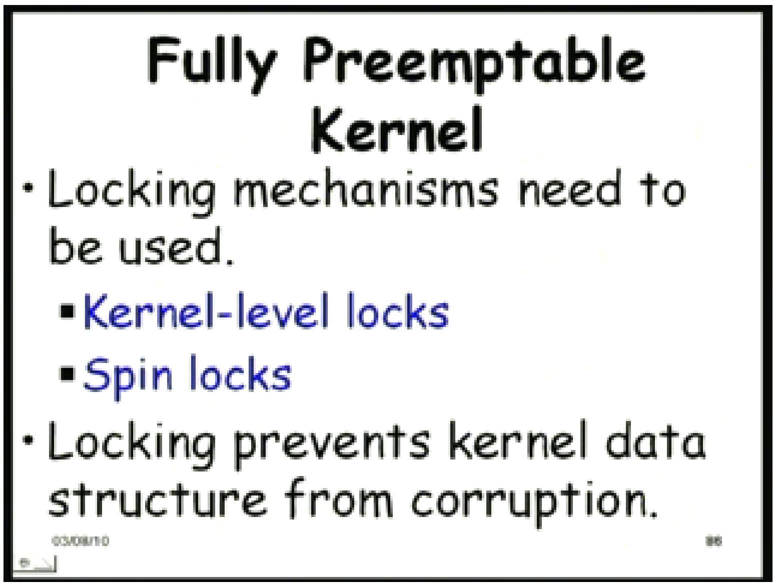
Preemption Point Approach

- Preemption points are introduced in systems calls:
  - At points where kernel data structure cannot be inconsistent.
  - At these points:
    - OS looks for other tasks to run, services interrupts, etc.
  - Examples: HP-UX

03/08/10 85

There is another category of approaches, will also look at this examples of this. The preemption point approach; here in the operating system code the kernel code at specific points, preemption points are introduced in the system calls. The earlier in the traditional UNIX, once the system call occurs until the system call completes it is not preemptable. But here at the points where the kernel data structure is consistent you can look for other tasks to run, and can take up those tasks, and suspend the task which was running; the system call that was running can be suspended, and other tasks can be run. Some example of preemption point, HP-UX is a popular example of the preemption point approach.

(Refer Slide Time: 35:10)



Fully Preemptable Kernel

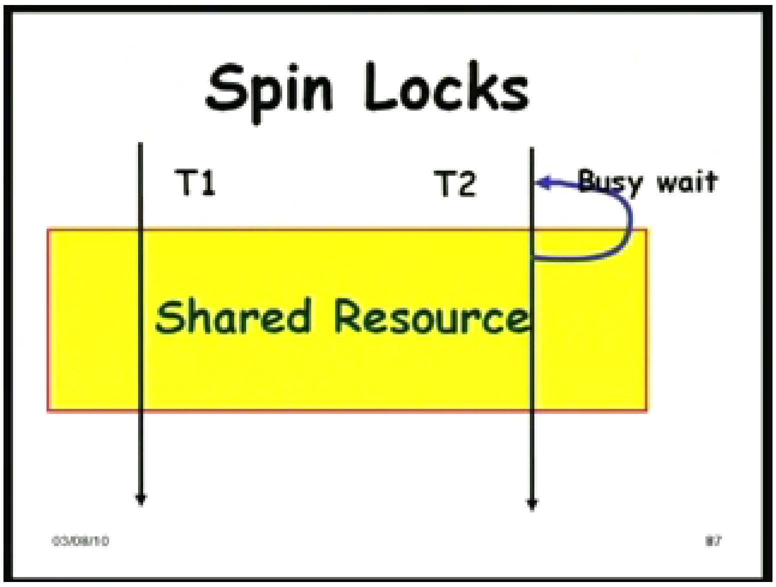
- Locking mechanisms need to be used.
  - Kernel-level locks
  - Spin locks
- Locking prevents kernel data structure from corruption.

03/08/10 86

And then, we have the preemptable fully preemptable kernel. We had said that for making the kernel fully preemptable need locking support. Isn’t it? Because the traditional UNIX there are no locking used for efficiency reasons and the interrupt was disabled. Isn’t it?

So, there are two types of locks that are used: one is the kernel level locks; which are like traditional locks, but what about spin locks? Anybody heard about spin locks? Because these are also being used now days in traditional operating system, especially the multi processor operating systems, like multi core, or multi processor they have to use spin locks, anybody heard about spin locks? So, let us discuss about spin locks, because these are even taught in a traditional operating system course.

(Refer Slide Time: 36:25)



So, the idea behind spin lock is as follows see, if a task both the tasks T 1 and T 2 needs certain shared source now, T 1 is acquired the resource and using T 1 is using the resource, and that time let us say T 2 also requested the resource now, T 2 waits for the resource and by the UNIX operating system semantics as soon as the task waits for something it gets preempted. Isn't it? It gets context switched a waiting task is context switched. But just imagine that suppose the resource is needed here by T 1 for doing some 1 or 2 instructions, let us say just increment a clock value, or read a clock value, or something. So, it has set the lock here for doing some simple operation 2, 3 instructions on the resource, but by the time if the task T 2 wanted the resource then it will be context switched, because it has to wait for it and the UNIX would context switch it. Isn't it?

And context switching involves much more over head, because all its the context needs to be saved, and again it needs to be restated at a later time, but cannot it just busy wait here without context switching, cannot it just keep on wait waiting here and then you know that this is will be used only for 1 or 2 instructions and as soon as this gets done with this it can start using it. So, that is the main idea here.

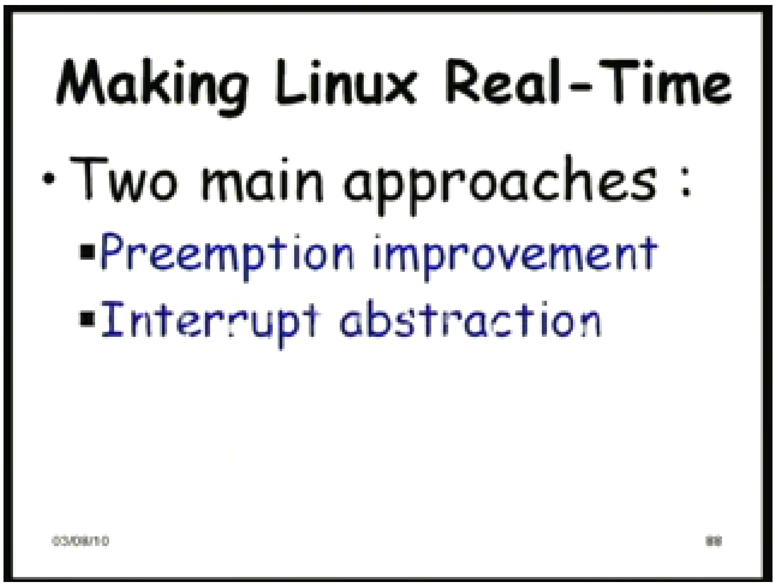
For small processing involving a resource a spin lock is set. So, if you are a programmer and you are using a resource for 1 or 2 instructions to lock it.

Sir is the information about T 1 that it is using resource for only this was for any.

That is the programmer. So, the programmer knows that, what is going to once he locks the resource what is he going to do the code that he knows, right? So, if you are programmer you would know that once you lock the resource, how much processing you are going to do with the resource, right? So, if you know that you are just going to increment some value or do some simple operations, you will not set a kernel lock you will set a spin lock. So, that another task which is trying to access that resource will not context switch, it will just busy wait for that resource. So, a spin lock use of a spin lock can result in faster completion time for tasks and avoid context switches. Is that okay?

What is a spin lock? A spin lock is also a lock, but unlike the traditional locks the spin lock is set for resources or set for processing; for very small processing when a resource is locked. And another process which needs that resource, if the spin lock is set, it will not context switch, it will just do a busy wait and in any case the resource will be available within very short time and therefore, it can continue without incurring a context switch.

(Refer Slide Time: 40:22)



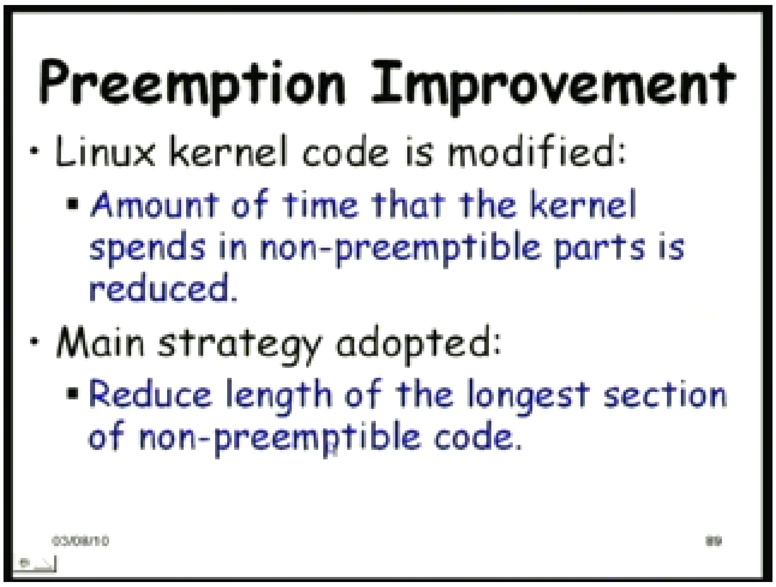
Making Linux Real-Time

- Two main approaches :
  - Preemption improvement
  - Interrupt abstraction

03/08/10 88

let us proceed now, Linux is also become extremely popular as we were discussing, there are many approaches to make Linux real-time mainly two approaches: one is the preemption improvement and the other is a fully preemptable kernel. Preemption improvement and fully preemptable kernel unfortunately, it is not showing there written there interrupt abstraction, fully preemptable kernel.

(Refer Slide Time: 41:06)



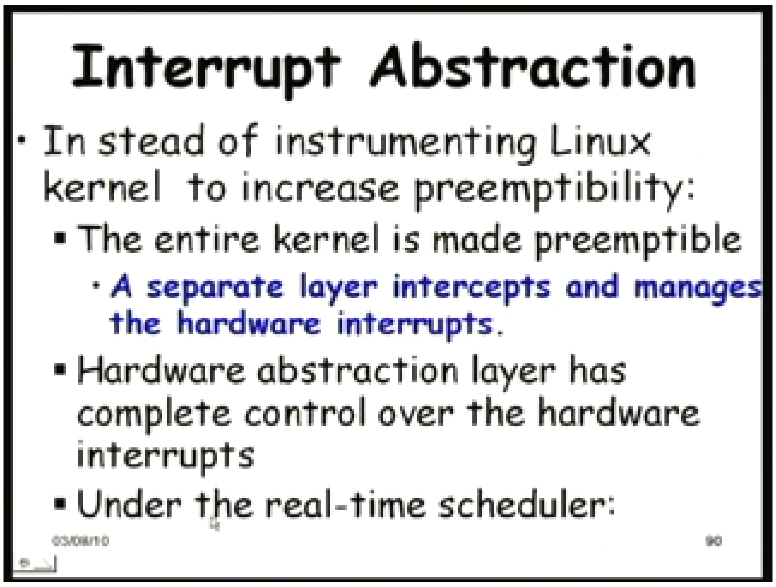
Preemption Improvement

- Linux kernel code is modified:
  - Amount of time that the kernel spends in non-preemptible parts is reduced.
- Main strategy adopted:
  - Reduce length of the longest section of non-preemptible code.

03/08/10 89

So, let us look at these two approaches: first let us look at the preemption improvement; the code is modified. So, that the amount of time the kernel spends in non-preemptable part is reduced. See, here again some part of the code it is non-preemptable, but that is reduced. The main strategy adopted is reduce the length of the longest section of non-preemptable code similar to the preemption point approach.

(Refer Slide Time: 41:40)



Interrupt Abstraction

- In stead of instrumenting Linux kernel to increase preemptibility:
  - The entire kernel is made preemptible
    - A separate layer intercepts and manages the hardware interrupts.
  - Hardware abstraction layer has complete control over the hardware interrupts
  - Under the real-time scheduler:

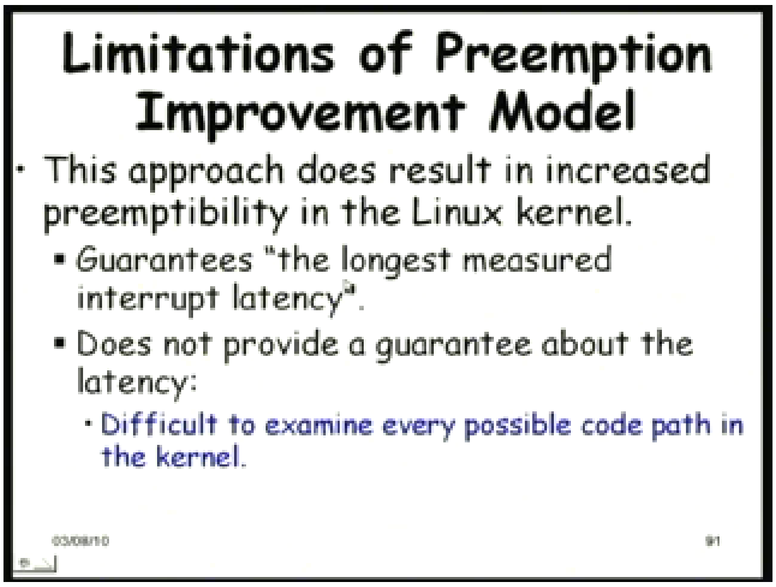
03/08/10 90

Now, let us look at the interrupt abstraction approach; instead of instrumenting the code to increase preempt ability, that is reduce the sections of the code where it is prevented the preemption is prevented the entire kernel is made preemptable.

How is that done? See here, a separate layer intercepts and manages the hardware interrupts, but how will that help? If a separate layer intercepting the interrupts and managing the interrupts, how will that help? How will that make the kernel fully preemptable? let us proceed, possibly these will become clearer that, we have a the kernel itself is becomes preemptable, because there is a separate interrupt abstraction layer which is handling the interrupt, and the kernel is running as a task of this we we will see that.

The hardware abstraction layer has the complete control over the interrupts, and the kernel actually runs as a task of this. The hardware abstraction layer has complete control over the hardware interrupts, and the Linux kernel runs as the task of the real-time scheduler; under the real-time scheduler the kernel runs as a task.

(Refer Slide Time: 43:43)



Limitations of Preemption Improvement Model

- This approach does result in increased preemptibility in the Linux kernel.
  - Guarantees "the longest measured interrupt latency".
  - Does not provide a guarantee about the latency:
    - Difficult to examine every possible code path in the kernel.

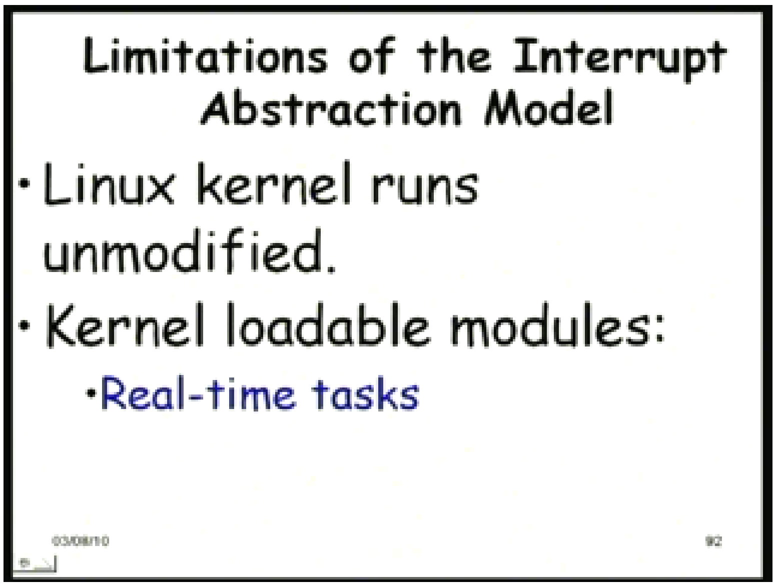
03/08/10 91

One of the major problems with the preemption improvement model is that, this if you look at the literature on the real-time operating system based on the preemption improvement model; you will see that it guarantees the longest measured interrupt latency. See, there is a catch here, the longest measured interrupt latency it will say that see 100 milliseconds or 50 milliseconds is the longest measured interrupt latency.

And there is a catch here, because it does not provide that it will never exceed that 50 milliseconds, it says that, it is the longest measured latency, if you find that something as 100 milliseconds under some of the situation, we will say that that path was not measured, its measured interrupt latency; the path that were measured the meaning of this is that, the paths that were measured on that the longest interrupt latency was 50 milliseconds.

It does not provide guarantee about the latency worst case latency. See, if it was the awarded as the worst case interrupt latency something then, that guarantees, but here it is the longest measured interrupt latency highly misleading, but that is how it appears. So, because all the paths cannot be measured the code is complex.

(Refer Slide Time: 45:23)



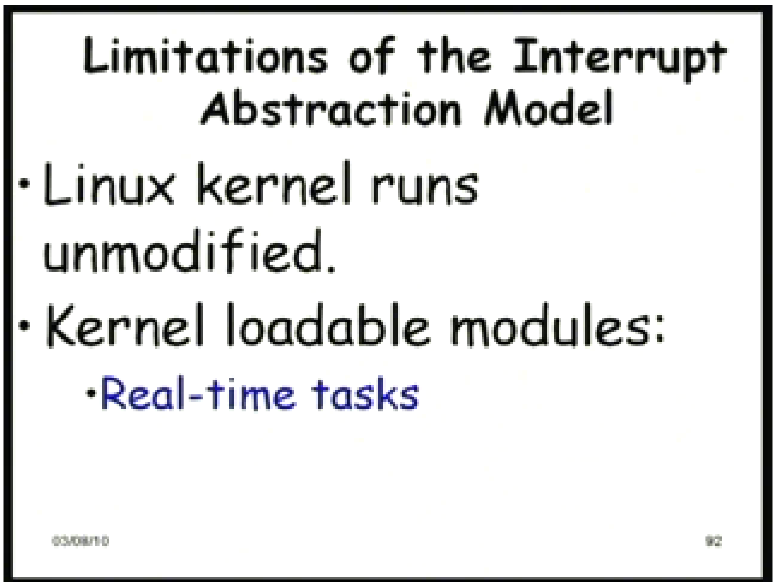
Limitations of the Interrupt Abstraction Model

- Linux kernel runs unmodified.
- Kernel loadable modules:
  - Real-time tasks

03/08/10 92

So, there is no guarantee given on every path, it is just whatever was the measured on that the maximum value was reported.

(Refer Slide Time: 46:20)



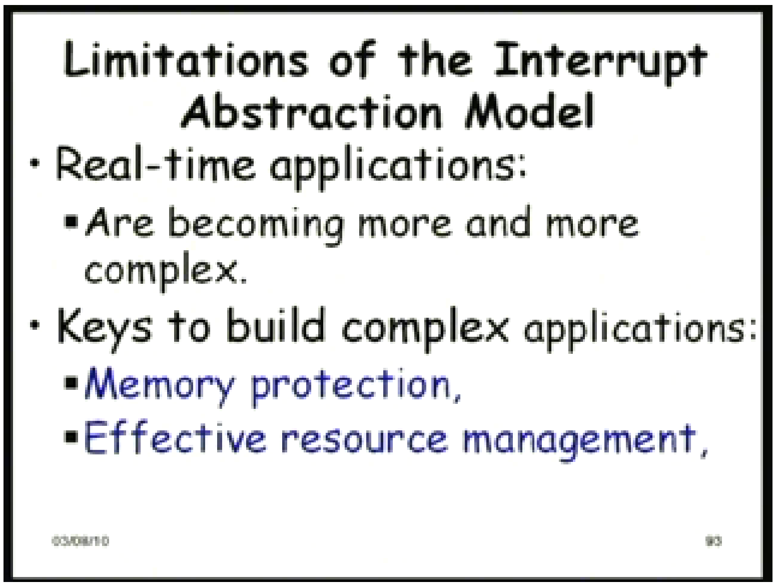
Limitations of the Interrupt Abstraction Model

- Linux kernel runs unmodified.
- Kernel loadable modules:
  - Real-time tasks

03/08/10 92

Now, let us look at the interrupt abstraction model. In the interrupt abstraction model the Linux kernel basically is unmodified, the same Linux kernel runs and the real-time tasks run as kernel loadable modules. So, in the interrupt abstraction model the Linux kernel runs unmodified and the real-time tasks, they run as kernel loadable module. So, they operate as a kernel task at the kernel level priority they operate.

(Refer Slide Time: 46:40)



Limitations of the Interrupt Abstraction Model

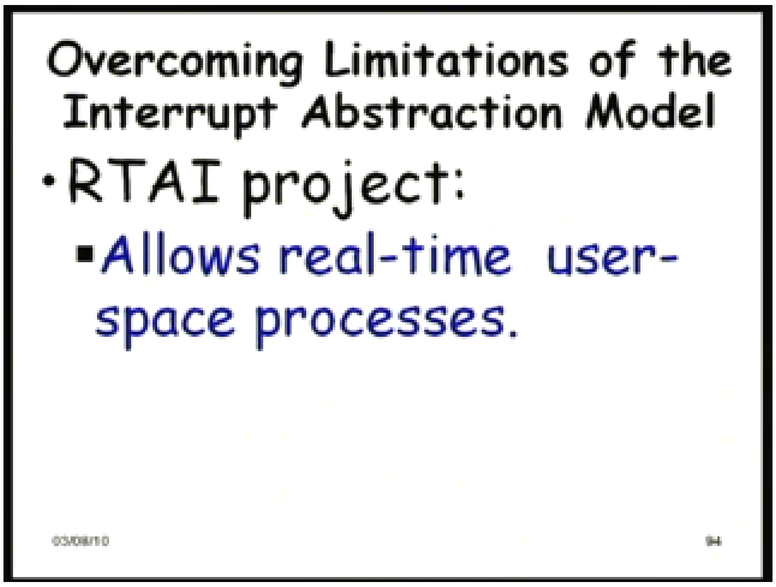
- Real-time applications:
  - Are becoming more and more complex.
- Keys to build complex applications:
  - Memory protection,
  - Effective resource management,

03/08/10 93

But the problem with the interrupt, we will see the interrupt abstraction model in more detail as we proceed, but let us remember the short coming of the interrupt abstraction model.

The main problem is that, the applications are becoming more and more complex and we know that the key to build complex application is memory protection, resource management, process isolation, but here once they run the kernel mode those are not guaranteed. So, the debugging of the applications becomes extremely difficult in the interrupt abstraction model, because after all the applications run in the kernel mode. So, that is the problem with the interrupt abstraction model.

(Refer Slide Time: 47:31)



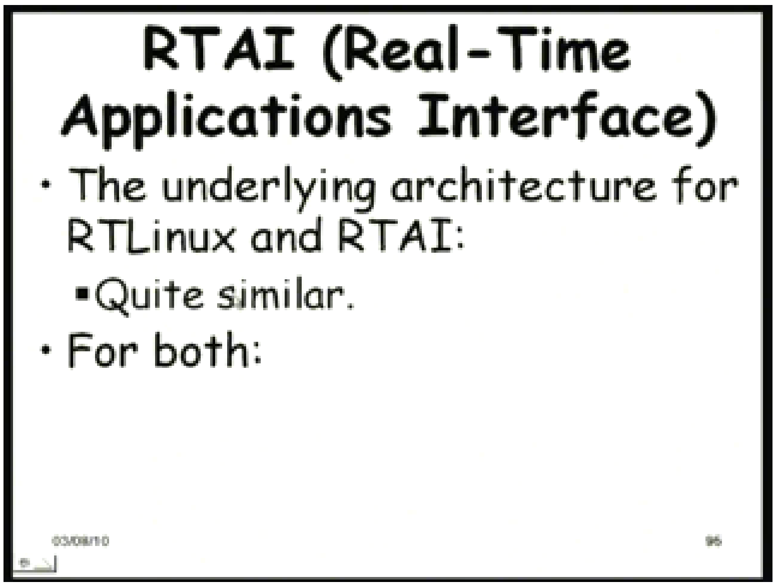
Overcoming Limitations of the Interrupt Abstraction Model

- RTAI project:
  - Allows real-time user-space processes.

03/08/10 94

But there have been attempts to overcome the limitations of the interrupt abstraction model, and the most important of that is the RTAI project. We will see the details of the RTAI project; it allows real-time user space processes. So, the application is no more a kernel process operates as user level processes.

(Refer Slide Time: 48:01)

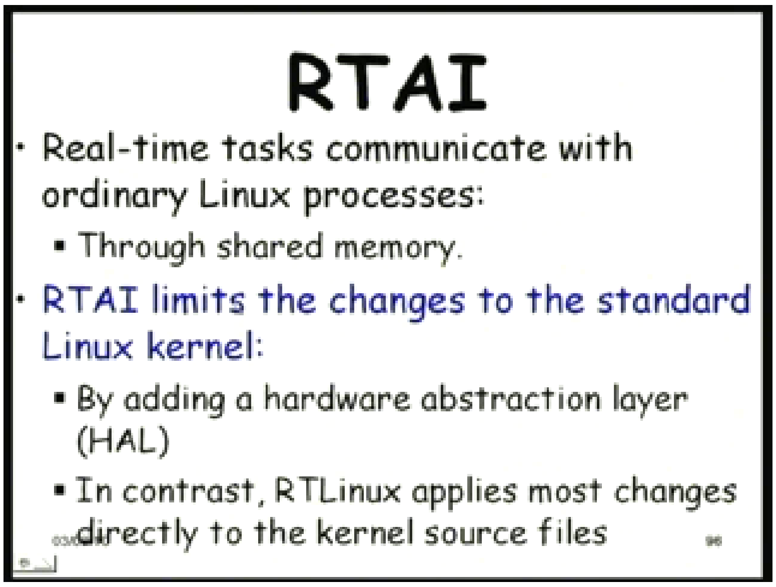

RTAI (Real-Time Applications Interface)

- The underlying architecture for RTLinux and RTAI:
  - Quite similar.
- For both:

03/08/10 95

It is the RTAI stands for real-time application interface. We will see that the R T Linux; we will see the features apparel R T Linux in more detail. This is actually an interrupt abstraction model where the Linux kernel runs as the task of the real-time scheduler at the lowest priority, and the applications runs as a kernel task.

(Refer Slide Time: 48:53)


RTAI

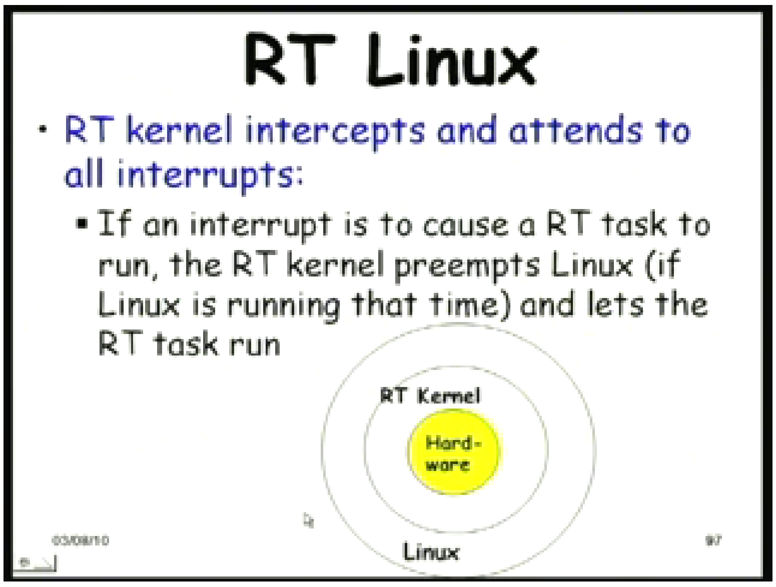
- Real-time tasks communicate with ordinary Linux processes:
  - Through shared memory.
- RTAI limits the changes to the standard Linux kernel:
  - By adding a hardware abstraction layer (HAL)
  - In contrast, RTLinux applies most changes directly to the kernel source files

96

If you think of it, the R T Linux and RTAI are quite similar for both the Linux kernel becomes a task of the real-time scheduler. In RTAI the real-time task communicate with ordinary Linux processes through shared memory. We will see more details of RTAI, because this is an important necessity when the making Linux real-time and since these are free software this are important for many types of embedded applications.

So, here the changes to the linux kernel are by adding a hardware abstraction layer and in contrast the R T Linux applies changes to the kernel source files.

(Refer Slide Time: 49:36)



RT Linux

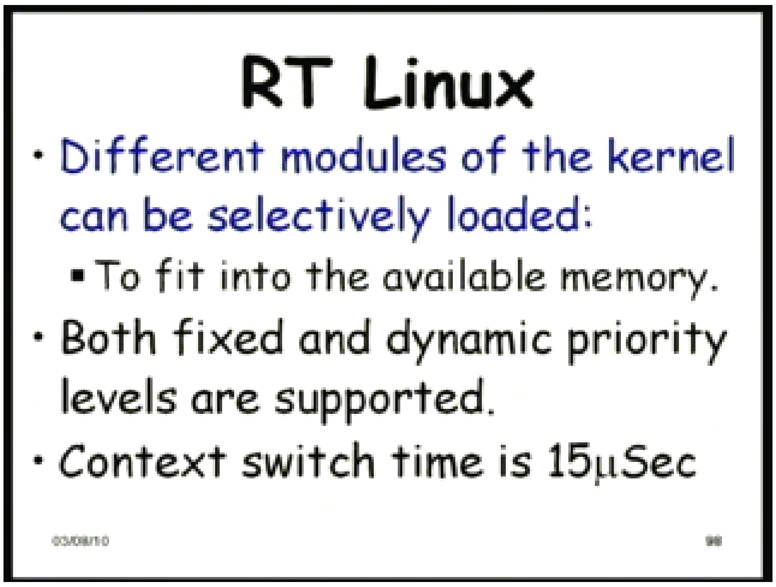
- RT kernel intercepts and attends to all interrupts:
  - If an interrupt is to cause a RT task to run, the RT kernel preempts Linux (if Linux is running that time) and lets the RT task run

RT Kernel
Hard-ware
Linux
03/08/10
97

So, first let us look at the R T Linux, and then we will examine the RTAI. So, that the concepts will become clearer, so far, we just said that; see both are using a interrupt abstraction model where an hardware abstraction layer is there which hijacks the interrupts and then it reports it to the real-time tasks scheduler. And the Linux kernel actually runs as the task of the real-time task scheduler. So, see here, if you have Linux system and you implement the R T Linux on it, then the R T Linux here or the R T kernel that is written here the R T Linux, it will hijack the hardware from the Linux see, it is come and sit between it has come and sit between the Linux and the hardware.

So, the R T kernel or the R T Linux will intercept the hardware and attend to all hardware hardware interrupts. And if an interrupt is to cause a real-time task to run, then it will just interrupt the real-time kernel. So, full preempt ability of the Linux kernel is ensured here, because it is after all the R T kernel which is interrupting it, and then, it allows the real-time task to run. So, this use as a separate approach to make the Linux kernels fully preemptable.

(Refer Slide Time: 51:22)



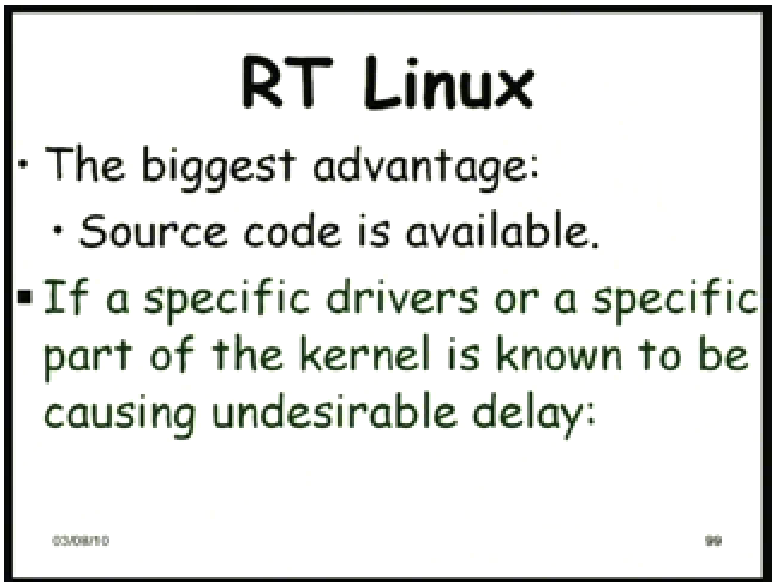
RT Linux

- Different modules of the kernel can be selectively loaded:
  - To fit into the available memory.
- Both fixed and dynamic priority levels are supported.
- Context switch time is 15μSec

03/08/10 98

So, here the features of R T Linux are the different modules of the kernel can be selectively loaded. So, it can be easily tailored to fit into an available memory of an embedded application. Both fixed and dynamic priority levels are supported, context switch is of the order of 15 microseconds.

(Refer Slide Time: 51:45)



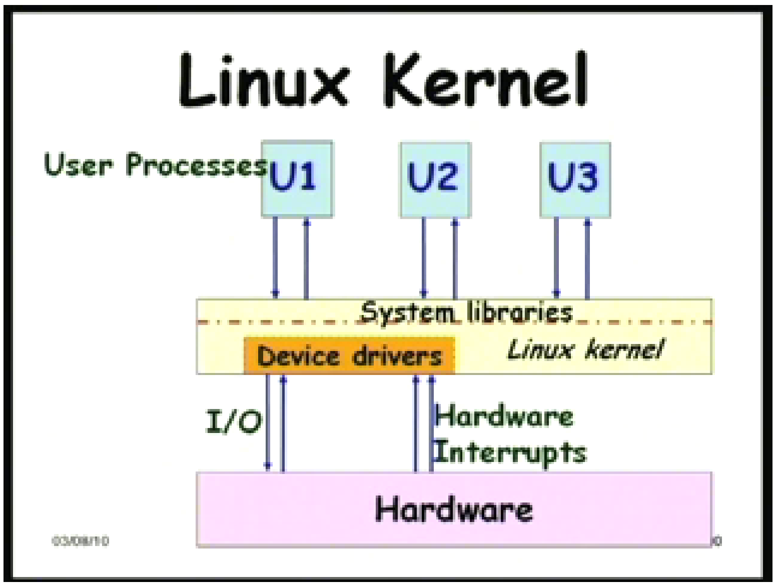
RT Linux

- The biggest advantage:
  - Source code is available.
- If a specific drivers or a specific part of the kernel is known to be causing undesirable delay:

03/08/10 99

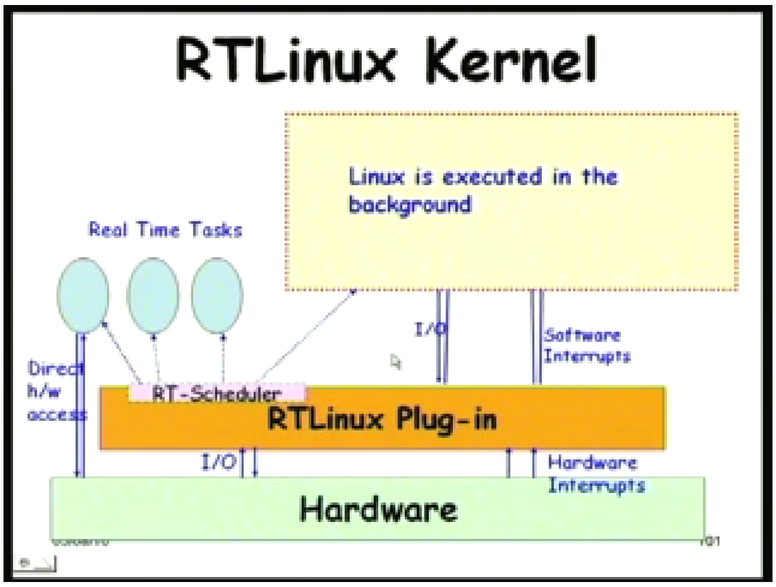
The biggest advantage here is that, the source code is available. And when developing a software, an application, if you find that some specific driver or specific part of the kernel is causing undesirable delay then, you can even attempt to change the source code of the kernel. But if it is the commercial package; commercial operating system cannot do that. If you find that a specific service is taking longer than what you need, you either use that, or do not use that operating system, or you can at best request the vendor whether something can be done, but you have no control. But here you have control on the source code and you can fine tune the source code.

(Refer Slide Time: 52:42)


Linux Kernel

So, this the way it works here. It is the traditional Linux kernel, in a traditional Linux kernel that, we had discussed this model earlier also, the traditional Linux model where the user processes by invoking the system libraries, the avail of the services and the interrupts are reported to the drivers, and the I/O is also performed by the device drivers and the Linux kernel sits on top of the device drivers.

(Refer Slide Time: 53:25)



But in the R T Linux, see here, the R T Linux has hijacked the hardware here from the Linux, the Linux has become a background task a low priority task for the R T Linux kernel here, see here, the real-time tasks are directly running on the real-time scheduler and the Linux is also running as a task of the real-time scheduler. And it can if a real-time task becomes ready it can interrupt the Linux kernel, Linux is the lowest priority real-time task.

So, the interrupts if it is meant for the Linux kernel they are reported here forms of software interrupts. And for I/O the Linux request the R T Linux here whereas, the real-time tasks can have direct access to the hardware if necessary for the faster I/O. See, they can also do I/O through the real-time Linux, but some tasks are also allowed to have direct access to the hardware

So, we will stop here today, in the next class we will look at the R T Linux in more detail and we look at the RTAI which is an improvement or some of the short comings of the R T Linux are attempted to be over come in the RTAI.

See, in the R T Linux we are saying that, the real-time tasks they run in the kernel mode is possibly one of the major problems with the R T with the R T Linux based software development, debugging and development of the application becomes difficult. So, we will see how RTAI tries to overcome this problem. So, we will stop here today.