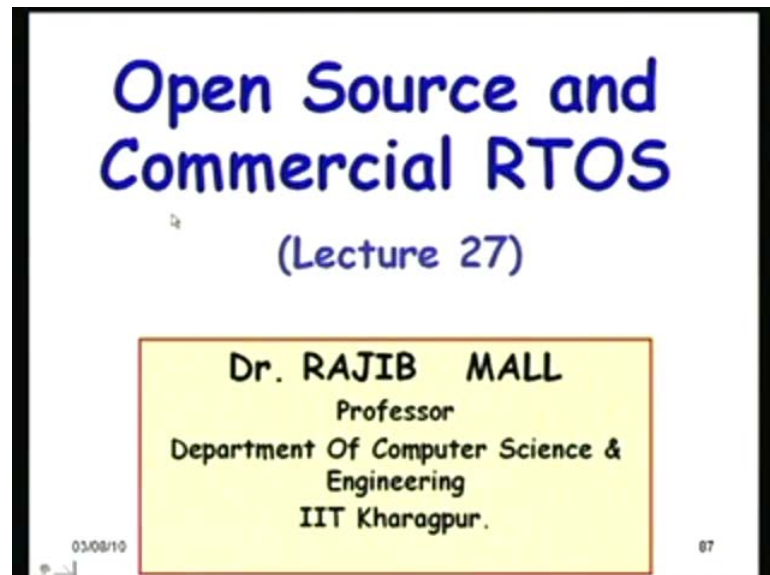


Real – Time Systems
Prof. Dr. Rajib Mall
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

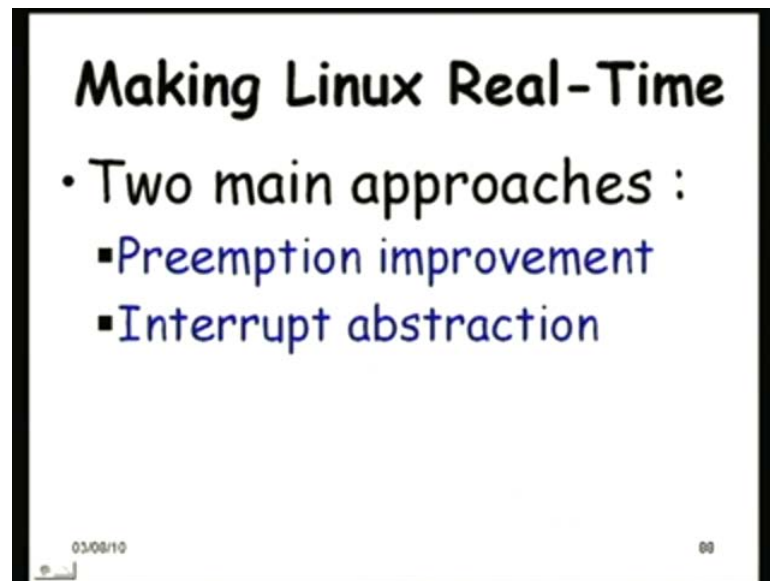
Module No. # 01
Lecture No. # 27
Open Source and Commercial RTOS

Good morning, now let us get started with where we had left last time.

(Refer Slide Time: 00:29)



(Refer Slide Time: 00:33)



So, we were discussing about the requirement for a real time operating system and then we were trying to find out what are the difficulties with the current traditional operating systems, and then we were trying to see how to overcome that. And we had lived at the posix real time requirement, and now let us see how we make Linux real time. So, **what was** do you remember - I mean - if we use Unix and windows for real time applications, what difficulties we will face.

Non-preemptible kernel.

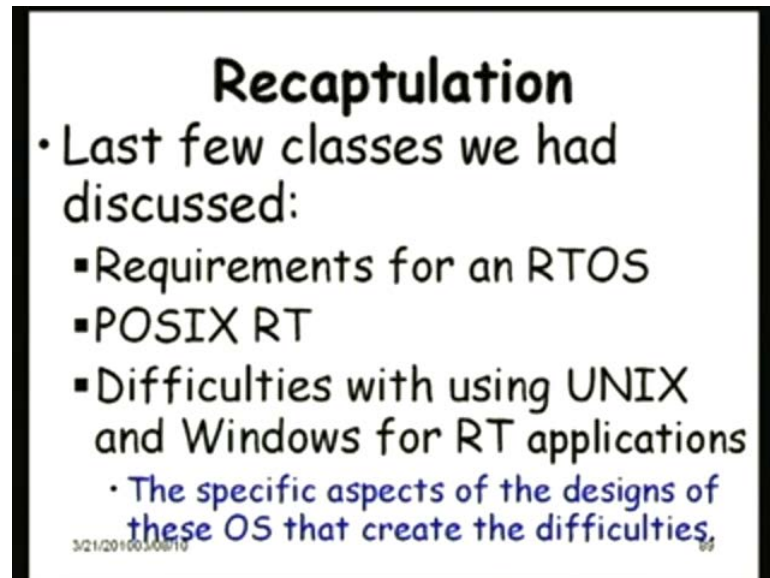
Yeah, so one of the main problem was non-preemptible kernel, and other difficulties - few other difficulties.

(())

Yeah, dynamically changing priority level for a traditional task, through put is the important criterion rather than meeting deadlines. And here, for real time operating system the situation is entirely different; we need all the tasks to meet their deadlines and through put is of not a consequence. As long as the tasks meet their deadline no reward - - special reward - in completing tasks early. And let us see how to make Linux real time. There have been two main approaches, one is to improve the preemption time improvement, that is the non blocking part of the kernel, whether interrupts or massed to reduce that and the other has been to interrupt abstraction.

So, here in the interrupt abstraction, the Linux kernel does not get the interrupts directly, the interrupt is taken care by another layer in between - which sits between - the Linux kernel and hardware.

(Refer Slide Time: 02:49)

A slide titled "Recaptulation" with a black border. It contains a bulleted list of topics discussed in previous classes. The text is in black and blue. A date stamp "3/21/2010 03:08:10" is visible in the bottom left corner.

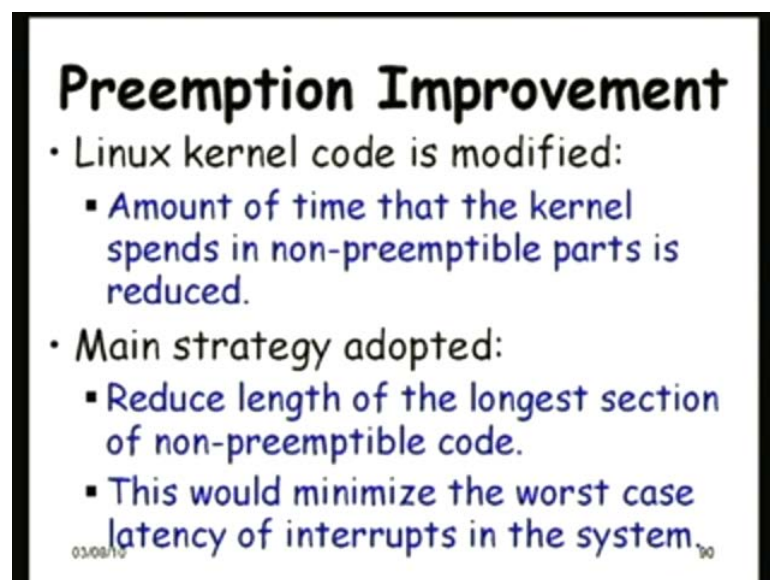
Recaptulation

- Last few classes we had discussed:
 - Requirements for an RTOS
 - POSIX RT
 - Difficulties with using UNIX and Windows for RT applications
 - The specific aspects of the designs of these OS that create the difficulties,

3/21/2010 03:08:10

I think, this we had seen last time, that is, requirement of a real time operating system, posix, difficulties with using Unix and windows for real time applications, and the specific aspects of designs of the operating system that create the difficulties.

(Refer Slide Time: 03:12)

A slide titled "Preemption Improvement" with a black border. It contains a bulleted list of modifications to the Linux kernel code. The text is in black and blue. A date stamp "03/08/10" is visible in the bottom left corner.

Preemption Improvement

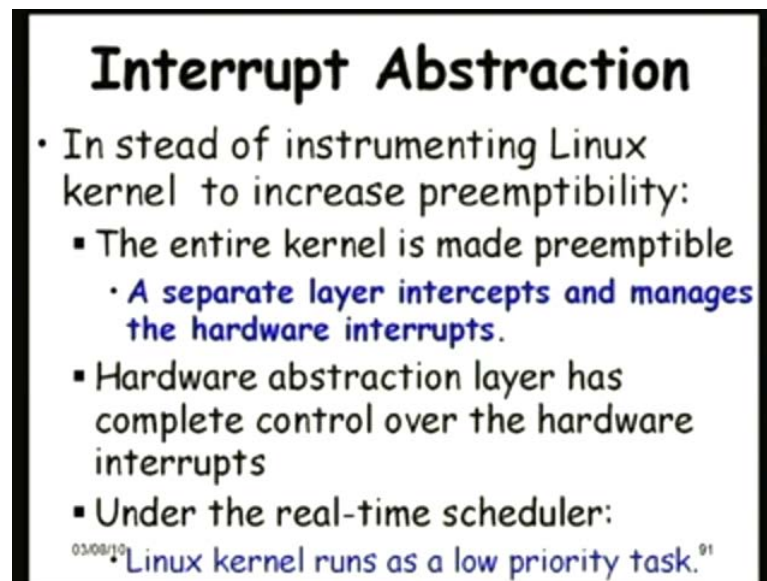
- Linux kernel code is modified:
 - Amount of time that the kernel spends in non-preemptible parts is reduced.
- Main strategy adopted:
 - Reduce length of the longest section of non-preemptible code.
 - This would minimize the worst case latency of interrupts in the system.

03/08/10

Now, let us look at the first approach for making Linux real time that is preemption improvement. So, need to modify the kernel code such that the amount of time the kernel spends in the non-preemptible part is reduced that is what we are saying that, as soon as it enters in a traditional Linux interrupts are massed. So, you need to reduce the time that is spent in the kernel part with interrupts massed has to be reduced. So, the main strategy adopted is to reduce the length of the longest section of the non-preemptible code.

So, in between when **where** the kernel data structure is consistent, it checks for interrupts and it handles the interrupt service routine at only those points where the kernel data structure is consistent. And of course, if we add such points - preemption points - where the interrupts are examined and they are handled; the worst case latency of the system would be improved isn't it? Because, instead of the entire system call being non-preemptible, here there are points called as an interrupt points, where the interrupts are examined and they are handled.

(Refer Slide Time: 04:49)



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:
 - Linux kernel runs as a low priority task.⁹¹

The other approach is the interrupt abstraction, actually instrumenting the Linux kernel to increase preemptibility has several difficulties. One is that still you cannot guarantee the maximum latency, because even if we instrument the code the longest path, there can be a various paths in the kernel, so making a guarantee about the longest preemption time is very difficult.

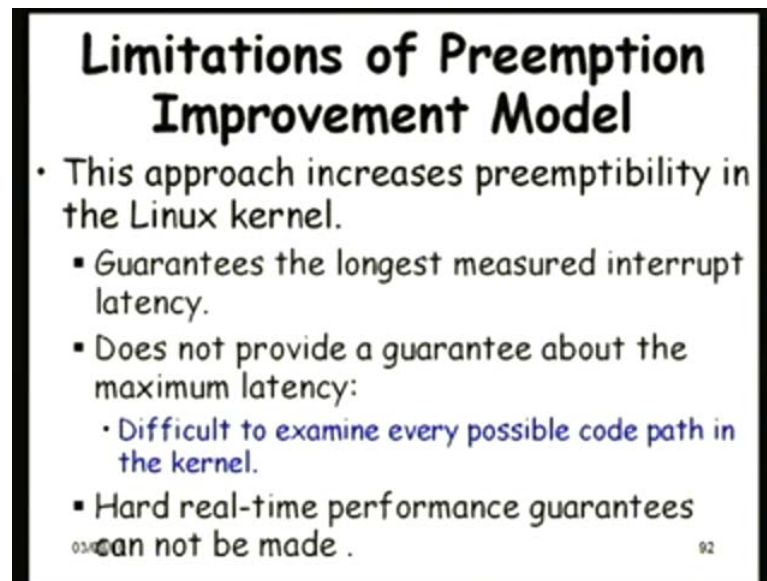
So, typically these are cleverly worded where they use the interrupt point approach is that, the paths that have been absorbed are guaranteed to be within certain latency. So, the word absorbed is there; that means that it is possible that, there can be a paths where the latency could be much more and here in real time situation. We are involved, we are concerned with the worst case and not the average case or usual case and so on, so that is the major disadvantage of the interrupt point approach.

So, let us look at the interrupt abstraction approach. So, here do not instrument the kernel code, the entire kernel is made preemptible, how is that possible? Because the kernel does not get the interrupts directly, there is a separate layer which intercepts and manages the hardware interrupt. We will just see this approach in slightly more detail, because it is possibly the more promising of the two approaches.

So, if the Linux kernel executes in with massed interrupt, it becomes difficult to preempt it, but here the interrupts are not directly given to the kernel. There is a separate layer which traps the kernels, hijacks the kernels from the hijacks the interrupts from the Linux kernel, and whatever is meant for the Linux kernel it gives only those interrupts and the others that are for real time tasks and so on, they are handled differently.

So, we will just see some more of this approach, so here the Linux kernel traditional Linux kernel loses its control over the hardware interrupts, it is selectively given interrupt and not only that, the traditional Linux kernel runs as the lowest priority task. So, that is a real time scheduler, we will just see as schematic, and under the real time scheduler the Linux kernel runs as a low priority task.

(Refer Slide Time: 07:46)



Limitations of Preemption Improvement Model

- This approach increases preemptibility in the Linux kernel.
 - Guarantees the longest measured interrupt latency.
 - Does not provide a guarantee about the maximum latency:
 - Difficult to examine every possible code path in the kernel.
 - Hard real-time performance guarantees can not be made .

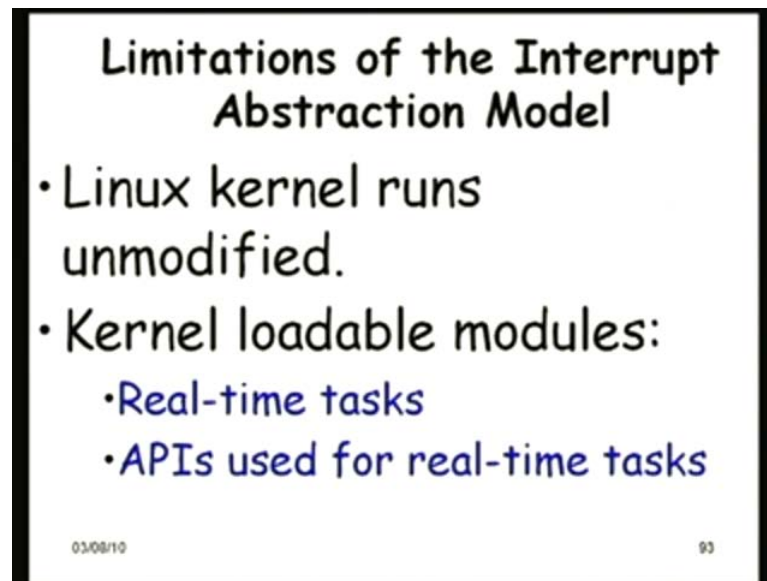
03/40 92

So, we were just discussing about the preemption improvement model or the points where the interrupts are examined, the interrupt point. This approach increases preemptibility of the Linux kernel definitely because we are adding points where the interrupts are examined, but still the limitations, let us be very careful about the limitations of this approach, should know where to use or not to use where which situation it cannot handle the preemption point approach.

So, it guarantees the longest measured interrupt latency, just see the word here, guarantees the longest measured interrupt latency; that means, all those interrupt latencies which were measured, it was found to be within certain bound. But what about those which were not measured or they were not identified, there can be several paths, it is not possible to identify all the paths through the kernel. So, therefore, it does not provide a guarantee about maximum latency, it does not see here, cleverly it has not said, guarantee is the longest interrupt latency it has not said that, it has said guarantee is the longest measured interrupt latency.

So, the reason is that it is difficult to examine every possible code path in the kernel, and as a result, it becomes very difficult to provide guarantee about the worst case latency that can be encountered in this approach. And naturally it cannot be used for hard real time tasks with dead lines or few milliseconds.

(Refer Slide Time: 09:38)



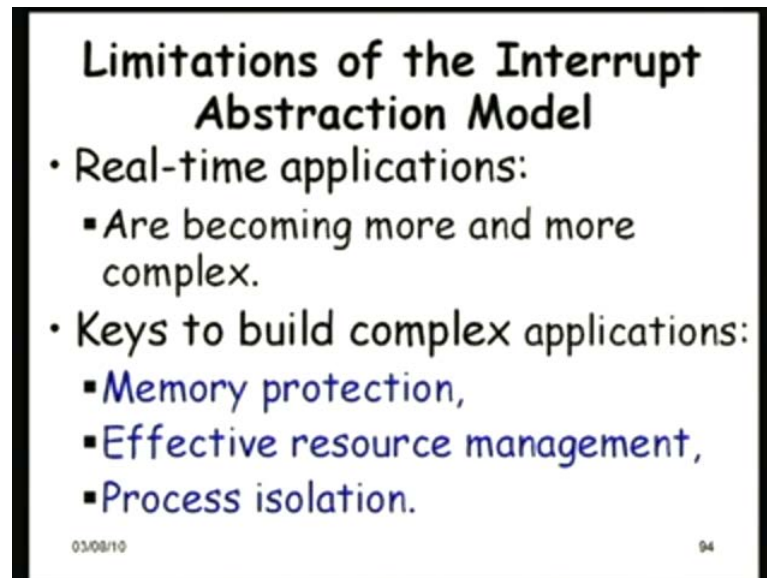
Now, let us look at the limitations of the interrupt abstraction model before, before we look at the technique in more detail. So, here the Linux kernel runs unmodified, that is what we are saying, that the only thing that happens is that there is a separate abstraction layer which hijacks the interrupts from the kernel and selectively provides the interrupts which are specifically required for the kernel and rest it uses, provides it to the real time task scheduler.

And the Linux kernel becomes a low priority task of the real time scheduler, and here the real time tasks become kernel loadable modules, so they operate, the real time programmer has to add those tasks in the kernel. They will have full access to the hardware and other resources of the system as kernel task, they run as privilege tasks and also same is the case with the interfaces or the libraries that are used for real time tasks the APIs.

Is that ok, - I mean - is it making sense, the two main approaches to making Linux real time, because Linux is a popular open source operating system and PCs are extremely expensive and also extremely popular. So, there is naturally a demand to have p c based real time applications running Linux. And the traditional Linux you know the shortcomings in real time applications, and here we are just examining the modifications to the Linux, traditional Linux there are available to make them real time.

So, the kernel is unmodified and the real time programs run in the kernel space and even the library calls etcetera they have to be run on the kernel space.

(Refer Slide Time: 12:02)



But now, let us see the difficulties with the interrupt abstraction model, because if we have to use this we not only should know the advantages of this, but also the limitations of this. One is that the complexities of the real time applications are increasing day by day, it was to be more sophisticated work controlling and that too many events the same time.

But whenever you want to build a complex application, you need memory protection, because when you write a large code, even if a small part of the code few statements erroneously access memories of the other part, it becomes extremely difficult to debug in the kernel mode. In the user mode you can easily debug, you can have debug or set or break points and so on, you can examine the data structures and so on.

Here, in the kernel mode debugging is entirely different game, if you have a difficulty and it crosses, **you know you** the symptom you will get when a real time application in the kernel mode misbehaves the system crosses. So, no trace left of about where the problem has occurred how to correct the debug, because if totally the control is gone how the debugger will run, whereas, **in a** if it is a user space there is a problem.

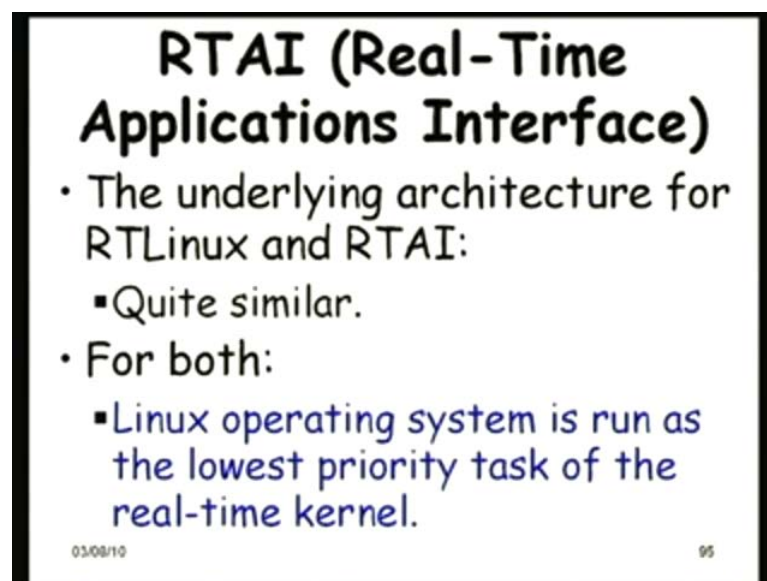
The operating system is still in control and the debug or etcetera, can function and give information about which code was running which was crossed, and what is the situation of the memory and so on - does that appear ok, fine. So, if we want to build a complex application having say 10's or 1000's of lines of code, one of the major requirements is memory protection and also effective resource management.

And also process isolation in a kernel space, all these three are not there, because it is a privileged process, it can access its own resources, directly control the devices do I O and also process isolation is not there because, there is no memory protection.

(())

See, in the traditional operating system, so the question is about what is process isolation? In a traditional operating system each process operates on in its own addresses, one process cannot cause or access data from another process directly unless there is a shared memory explicitly defined and also it can modify anything of another process **right**. But, if it is privilege process, process isolation is not there, because memory protection itself is not there, so how do you isolate the processes. So, they **say** are the same address process basically, so one process can also cross everything.

(Refer Slide Time: 15:28)



RTAI (Real-Time Applications Interface)

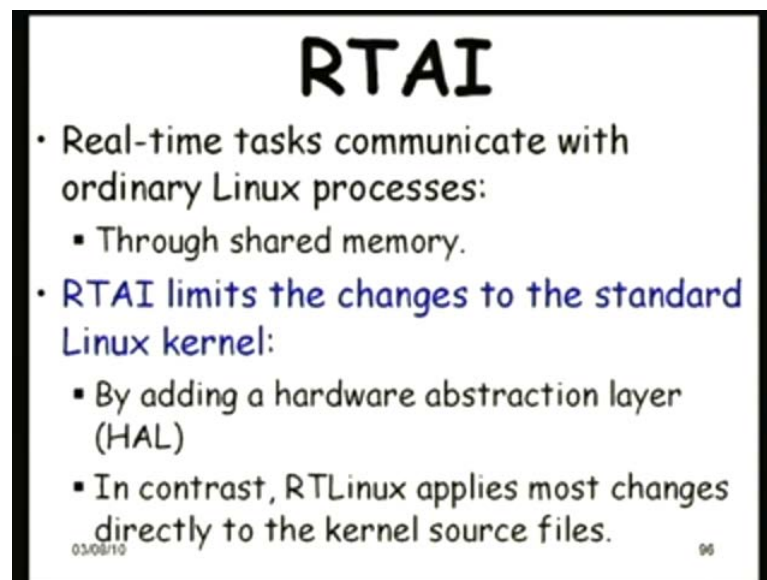
- The underlying architecture for RTLinux and RTAI:
 - Quite similar.
- For both:
 - Linux operating system is run as the lowest priority task of the real-time kernel.

03/08/10 95

Now, we will see two main approaches in this interrupt abstraction, one is the RTAI - The Real Time Applications Interface a freeware, just like Linux is a freeware, this is

also freeware, and this is the university initiative has become extremely popular, is large project undertaken at a university in Italy, the real time applications interface. And as we will see the RT Linux also in some detail, the underlying architecture for both RT Linux and RTAI are both the same, they are very similar. But there are some differences, we will also examine that, and RT Linux is more from commercial initiative, different programmers commercial and so on, they have collaborated, whereas RTAI is a university project. For both these approaches the Linux, the traditional Linux, operating system runs as the lowest priority task, the real time kernel.

(Refer Slide Time: 16:47)



And some tasks specifically the non real time and the soft real time tasks, they run as ordinary Linux processes, for example, handling e mails or may be running a compiler, debugger, etcetera, they still run as traditional Linux processes, the ordinary Linux kernel handle those tasks.

The soft real time tasks like logging and so on, and also e mails a compilers, debuggers etcetera, they all run on with a traditional Linux process, because if you put all of them in the real time kernel your application will become large and it would become extremely difficult to develop the application. So, all these run as usual user level tasks, the soft real time tasks, logging etcetera, and also the other supporting programs as tasks of the traditional operating system.

But we need to access data, for example, a logging activity needs to have the data required for logging, and they cannot use the traditional pipe mechanism of the Linux what is the difficulty, what would be the difficulty pipe mechanism is used?

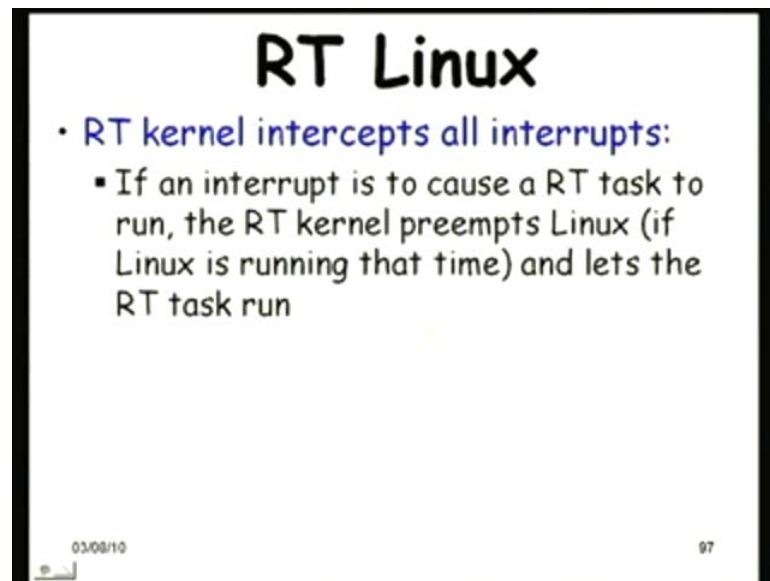
Anybody would like to answer?

(())

Not really, see, here the **traditional**, the ordinary Linux tasks they will run as the lowest priority tasks. So, if we have a synchronous communication, a message or a pipe with a non real time task - I mean - those which are at the lowest priority task, even the real time tasks will suffer delays. So, the way it is handled is through a shared memory, the communication of the real time tasks, if those soft real time and non real time tasks running on the traditional Linux kernel they exchange data such as login, etcetera through a shared memory.

And of course, as with the interrupt abstraction approaches, the changes to the standard Linux kernel are a very small is not there actually, almost no changes. The only thing that is required is a hardware abstraction layer, which takes care of the interrupts and distributes it to the traditional Linux if at all required. But there is a small difference between the RTAI approach and the RT Linux, the RT Linux applies changes to the kernel source files instead of having a separate hardware abstraction layer, this is also part of the kernel a small difference between RTAI and RT Linux.

(Refer Slide Time: 20:03)



Now, let us look at RT Linux, see both of this if you are interested you can download from the net, RTAI just give a search on the Google or look at the Italy university site if you know that, and you will easily find the RTAI site and download the source code and you can install it on Linux, same is with RT Linux you can actually, we had the several student projects where downloaded the RT Linux code and implemented on real time tasks on that.

Sir, RTAI's freeware

Yes, of course, both RTAI and RT Linux are freeware open source.

(())

Not really, see Linux is a freeware same is with RT Linux - Real Time Linux is also a freeware, see that I think we had told earlier for an open source if you develop something that also will become open source that is the agreement, so RT Linux is a freeware.

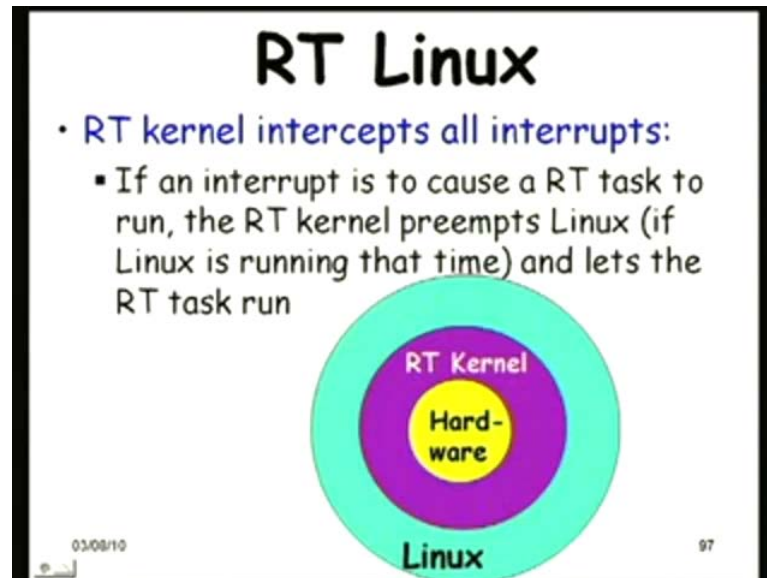
(())

Linux.

(())

No, our students have downloaded it - some students - and they have, it is freely download you check the website, you can check it, its freely downloadable, you can use it.

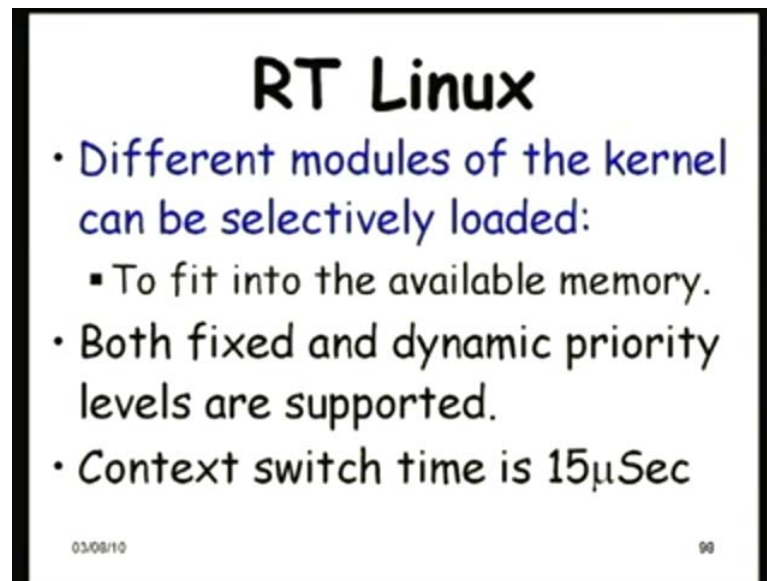
(Refer Slide Time: 21:42)



So, let us see this, here the real time kernel intercepts all the interrupts; see here, the Linux was there on the hardware and as soon as you download the RT kernel, the RT Linux and install it, it will sit in between the Linux and the hardware. And it will get all the interrupts here, and selectively if at all required it will pass it on to the Linux and the Linux runs as the lowest priority task under the RT kernel. And here if the traditional Linux - the ordinary Linux is running - now let us say an interrupt comes, so the interrupt is for a real time task let us assume, then what happens is the RT kernel preempts the Linux, so even **if that it** is that the ordinary Linux operating system, whether it is preemptible etcetera, it does not matter.

Because it is a task now, the Linux is a task, so even if it is non preempted we will just made from running state to ready state, just another task. So, it is in the ready state and it is preempted and it allows the real time task to run. So, the beauty is that just see without changing anything of the kernel, we have made it fully preemptible, so very good approach.

(Refer Slide Time: 23:20)



(())

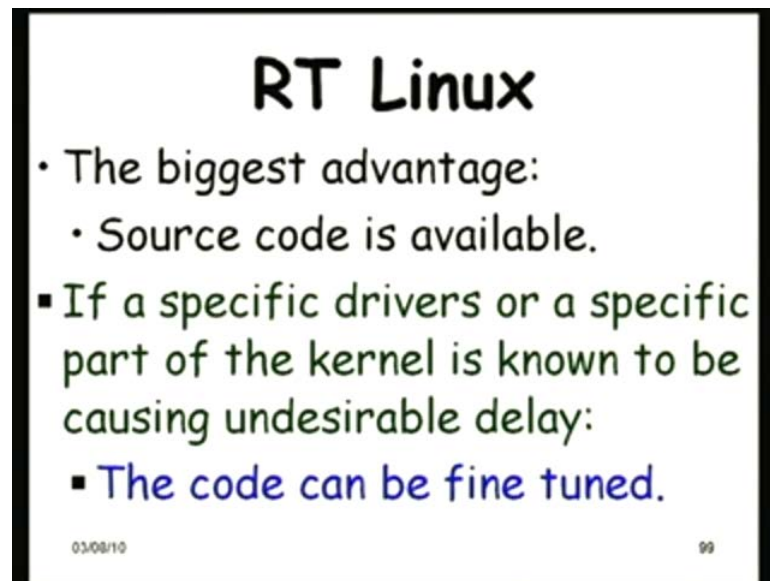
Ok, yes.

(())

See, the question is basically about, if there are some supporting tasks that run in the ordinary Linux kernel and that is preempted, will the real time tasks get affected? The answer is No, because see, for a successful running of a real time application all those tasks which need to run, they will be implemented in the real time kernel. Only those tasks where meeting the dead line etcetera is not critical, and they will be implemented in the ordinary Linux operating system, as I was saying logging activity if it is delayed little bit or does not matter or let us say while developing the program, compilers, debuggers, e mail, user level interactions etcetera, those run in the ordinary Linux operating system.

So, here the different modules of the kernel can be selectively loaded to fit available memory, because you might also need some times to run only the real time kernel, do not need the ordinary Linux, it can also you can also configure it for small embedded applications. And even in that you can configure the specific modules of the kernel that need to be loaded, and here both fixed and dynamic priority levels are supported and the context switch time is about 15 micro second.

(Refer Slide Time: 25:15)

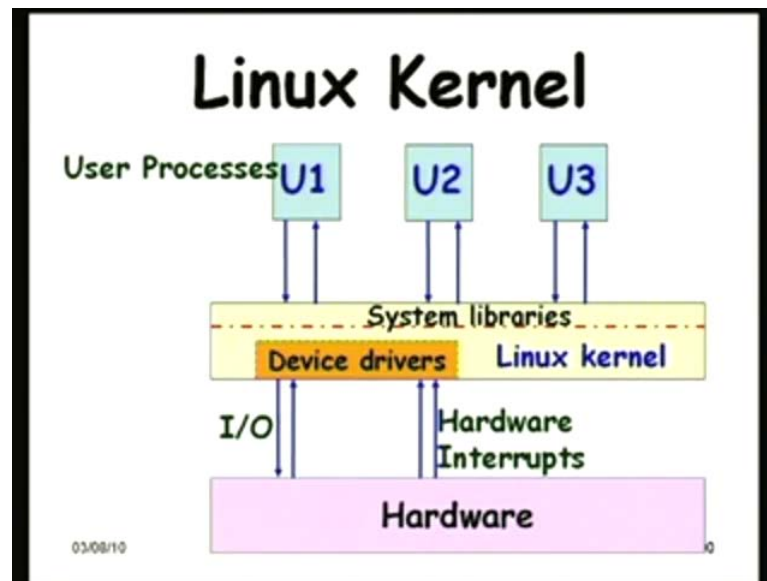


Now, let us see the advantages and disadvantages of the RT Linux, the biggest advantage possibly is that the source code is available, and as a result if you find that when you run the application find that specific drivers or specific part of the kernel is causing a delay that is unacceptable to the application.

Then you can change those part of the source code, whereas in a commercial real time operating system you find that some part of your task is not meeting their deadlines, then you can do much, you will have to rework your application or possibly if you are **you know** really a large scale manufacturer, possibly you can make a request to the operating system that we are getting millions of copies of your operating system can you just do this small thing for us.

But here you are in full control and if you find that it should be difficult to develop the application because, if the delay you can fine tune the code at some places.

(Refer Slide Time: 26:25)



Sir,

Yes.

(()).

Yes, yes,

(())

Yes

(())

Yes, this is the context stage time not the clock time, yeah.

(())

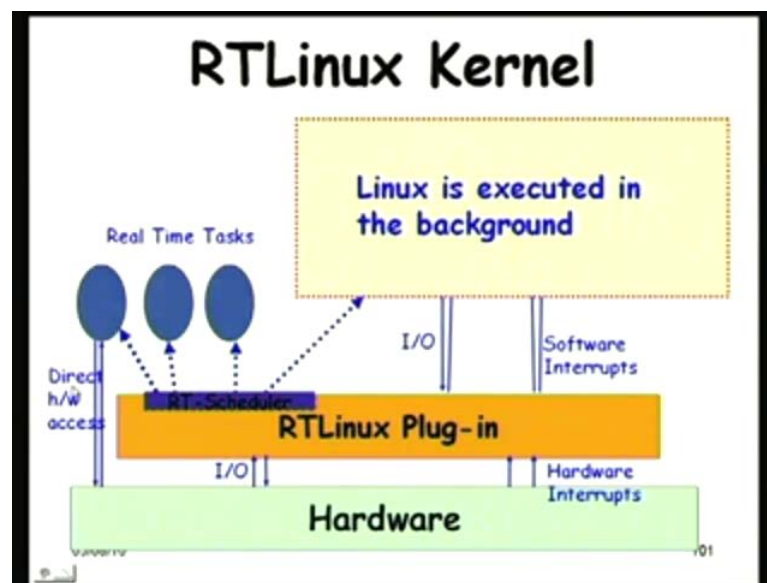
So, here basically, the question pertains to how the timer and the clock precision increased, we are just going to look at that we have a slide on that, but what we discussed is the context switch time.

So, the context switch time is faster because of the small kernel here, and also to make it faster, see when a context switch occurs you have to save the context and load the

context. **So and** also the non-preemptible part that also plays a role in the context switch time, because even if there is a need for context switch determined, it may not be a possible to do a context switch. So, here, it is both a light weight kernel and also the contexts that are required, only the context is said.

So, this is the schematic diagram or the traditional Linux kernel, traditional Linux kernel we have the hardware which directly reports the interrupts to the device drivers and the device drivers also drive the I O. And here the Linux kernel gets all the interrupts from the device drivers, and then, the user inter user processes - user level processes - they call the system utilities the system calls, and through that they can access the I O and also the kernel services, this is the traditional situation and all these are kernel level and this is the user level.

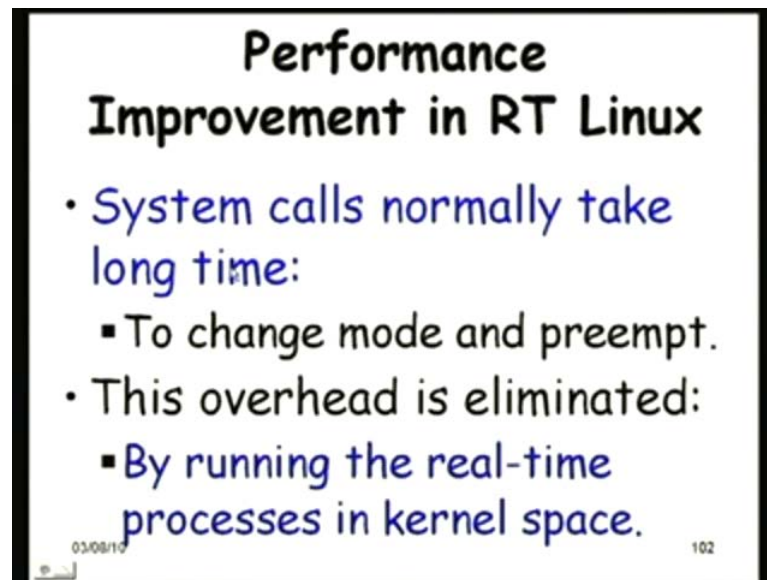
(Refer Slide Time: 28:33)



Now, this is the real time Linux kernel see here, the traditional Linux has come here is a background task, the real time Linux has come in here sit in between hijacking all the hardware interrupts and notifying if at all required anything to the traditional Linux through software interrupts. And if the traditional Linux once to do any I O etcetera, it just can give a request to the RT Linux, and see here that part is a real time scheduler - real time task scheduler, and the real time tasks absorb here they are running as the highest priority tasks.

And since they are running at the kernel as privileged processes they can directly access the hardware and do I O and save time, possibly the overhead rather than doing I O through calls to certain routines, they do I O themselves.

(Refer Slide Time: 29:44)



Performance Improvement in RT Linux

- System calls normally take long time:
 - To change mode and preempt.
- This overhead is eliminated:
 - By running the real-time processes in kernel space.

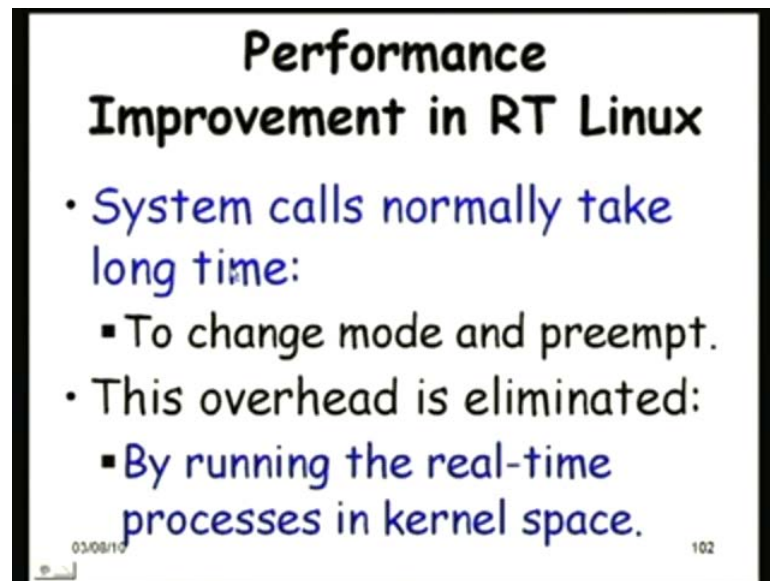
03/08/10 102

Now, let us see the performance improvements that have become possible in the RT Linux, one thing is that the system calls take long time, the traditional Linux operating system, one of the reason is the mode needs to be changed, and preemption, it becomes non-preemptible.

Now, this overhead is eliminated by running the real time processes in the kernel space. So, a system call essentially becomes a function call, so earlier the mode change was, how was the mode change occurring - I mean - the system call was made, it changes from user mode to kernel mode isn't it.

So, how was the mode change occurring when a system call is made how does the mode change from user mode to the kernel mode, anybody would like to answer? That happens in any operating system, you make a system call, and it changes from the user mode to the kernel mode.

(Refer Slide Time: 29:44)



Performance Improvement in RT Linux

- System calls normally take long time:
 - To change mode and preempt.
- This overhead is eliminated:
 - By running the real-time processes in kernel space.

03/08/10 102

Yes, anybody would like to guess anything, how would the mode change occur? Because after all it is just a function that is getting executed is it not, a system call is basically some code in functions, so how can the mode change occur?

(())

How the permission, see it is the user level process which is running, and it has made a system call, system call means it is just sub code getting executed, so we will change the permission, how will the mode change, who will change it.

(())

Who will change the mode bit, see user pressures cannot change the mode bit, otherwise then every user pressure will become a super user, no operating system will work in that case everything will be crashed maliciously.

(())

How will it change that is the question.

(())

No, no, see, if there is a way to change a program into kernel mode, then you people can write very malicious program and run in kernel mode and crash the system.

Task manager, sorry, Task manager.

No, but how will that task manager get invoked first of all, because it is just making a system call, and it is the same task running you just remember, it is not that a kernel task is created, it is part of the same task just some code will execute.

(())

How? What kind of interrupt? See, interrupts are given by external hardware isn't it, see this is you are making a function call.

(())

Yes

(()) itself invokes a system work for right system as soon as it sees a right system.

Who sees right system?

The interpreter which (())

Interpreter has no control, it cannot change any mode nothing, see it is the user task that is running and any code running there is becoming a user task and it has very little privilege. Since no one is forthcoming, see this is a topic in a traditional operating system that a code that runs in a user space has very limited privilege and the kernel has a privileged process and that is how the integrity of the system is maintained.

Now, when it makes a system call, so software interrupt need to be executed a software interrupt is an instruction basically.

I S R

Not I S R, I S R is different, it is a service routine, here it is one instruction called as a software interrupt instruction, most of the processors they have specific instructions for

raising software interrupts, and once the software interrupt is raised then the control is transferred from the program.

The user space level to the kernel level and then, under the kernel control it knows that what is the service routine being invoked, and only that is executed, it is not that the user space can execute any program it is only that specific service routine will be executed.

(())

If it was possible to change the mode bit, then it would everybody will just, every process can run as a kernel mode, there will be no discipline every malicious person will come and crash the system.

(())

No, no see you can execute

(())

See, this is a basic question actually in traditional operating system, see, if you run a software interrupt, software interrupt instruction, it is the kernel which is reported about the interrupt not the user **pressures**. And the kernel gets the control and then, it can change the mode or whatever and also it knows which code will run, it can be that arbitrary code of the user will run. It was possible to run that way people would have long time back crossed all operating systems.

So, whether this is the (())

See, viruses, you know that Unix is less affected by viruses.

(())

In windows is more affected by virus, you might have seen that if you are running a windows operating system, how many antivirus these, that, signature, detection, updating, etcetera, you will have to do, you do not do that in Unix isn't it, you have been using Unix, all of you isn't it.

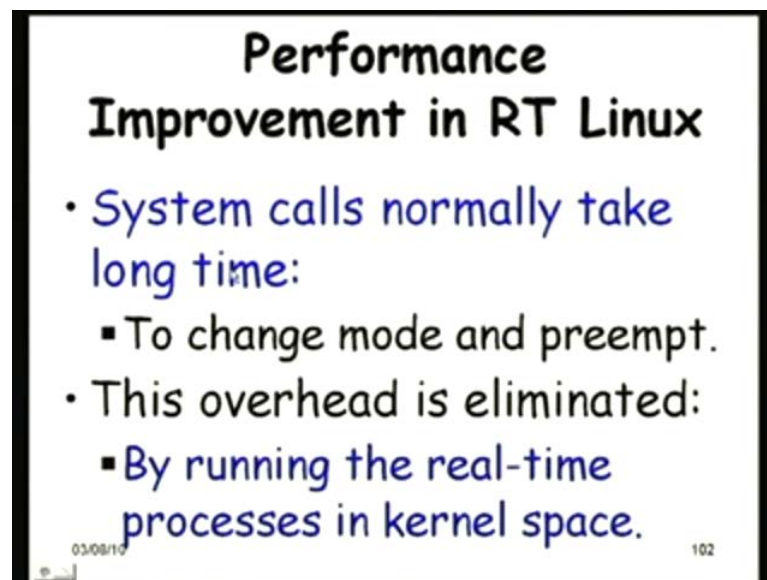
And you have seen that it has never crashed at least I have not seen - I mean - the way in the I have been using for 10's of years

And

(())

No, **no no** see they exploit some loop holes in the windows code, see as I was telling sometime that even though it is not a part of this course - I mean - we are slightly digressing from our discussion, but since you have raised it, see the thing is that the way we were discussing windows some time back.

(Refer Slide Time: 29:44)



And there we said that see, it started with a toy operating system, a DOS a Disk Operating System, it was a single tasking operating system and there was no kernel and user processors. Everything you know had full control, everything was like a kernel processor - even a small **program** c program you can write it and crash the system used to happen.

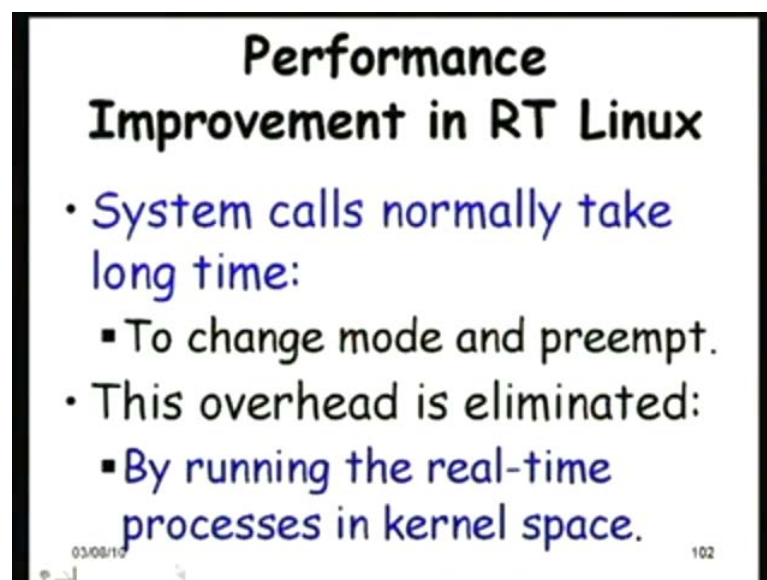
And the advantage was that it can access I O, your program can directly get devices, you can pole ports and so on, you could able to do that. But the thing is that as the dos evolved and became more and more powerful and there is kernel mode and dos, the user mode were separate out and so on.

But one difficulty that still remind is that the older programs have to run, see if you develop a operating system tomorrow - I mean - you develop a new version of the operating system and say that see, till now whatever programs you are running will not be able to run anymore, because we had done a very drastic change with the operating system. And we have **you know** created these modes and these programs that we are running which were exploiting this kernel mode etcetera they will see is to run, then you will never buy that operating system.

So, let see what is this, my application will not run old application then I better not take the new operating system. So, they have to live with that constant or backward compatibility with applications and that is the reason why windows are facing so many problems, more virus problem than Linux. Linux was from the beginning designed to be protected, know there is a clear user processes kernel processes their privileges defined, and from the beginning it was a neat operating system, but DOS have to evolve slowly, so that is the problem is that **ok**.

So, how the mode changes is there any comments on this, is that clear, that when a system call is made, the mode changes from user mode to the kernel mode.

(Refer Slide Time: 38:50)



Performance Improvement in RT Linux

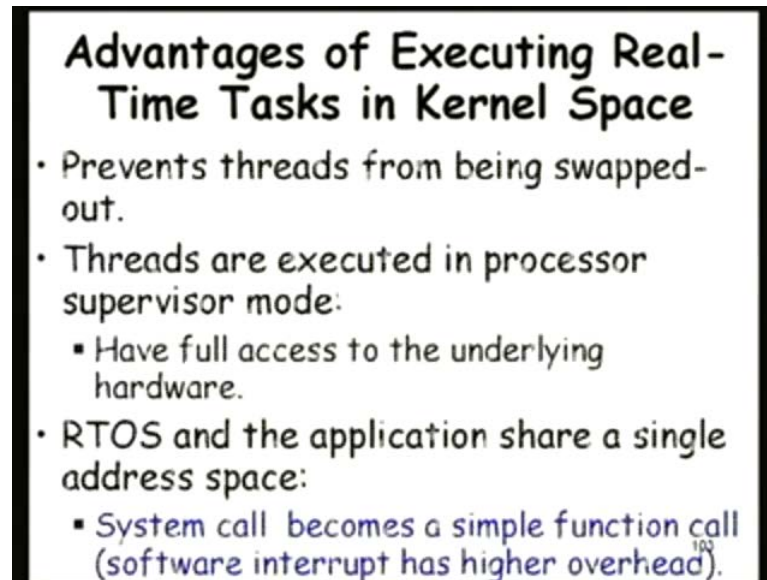
- System calls normally take long time:
 - To change mode and preempt.
- This overhead is eliminated:
 - By running the real-time processes in kernel space.

03/08/10 102

So, there is an overhead involved in changing the mode, the software interrupt has overhead, the kernel is reported, and then it examines and runs. This overhead is

eliminated by running the real time processes in the kernel space **right**, here no system call is necessary, it is only a function call by, for getting kernel services you just make the real time task makes function calls rather than system calls.

(Refer Slide Time: 39:30)



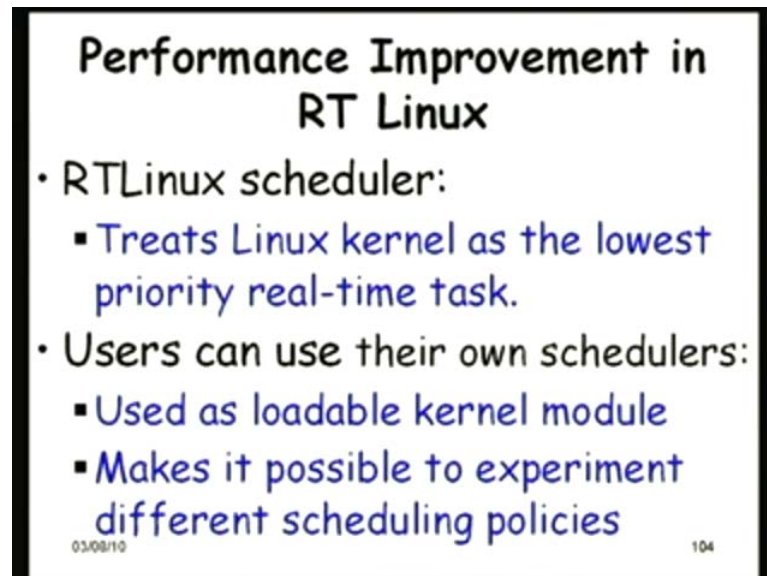
There are other advantages as well, for example, we know that the kernel is memory resident, we were discussing about that isn't it, that the user space all those virtual memory faced out etcetera, all those apply virtual memory phasing and phase faults all those are applied to user processors. The kernel is memory resident and the virtual memory concept does not apply to that.

So, it prevents threads from being swapped out, they are memory resident, the real time threads, and the threads are also executed in the supervisor mode, as a result they have full access to the underlying hardware, if they want to pull something most efficiently they can do that. Of course there is a violation of the discipline here, there are many processes and they do contradictory things the system can crash.

But if you are a real time programmer and concerned about efficiency, doing it fast, sometimes you would just directly access the hardware, and the real time operating system and the applications they share the same address space, and the system calls become a function call, that is what I think we were discussing just now, and the

software interrupts have overhead and that is no more required just a simple function call to obtain any kernel services.

(Refer Slide Time: 41:05)



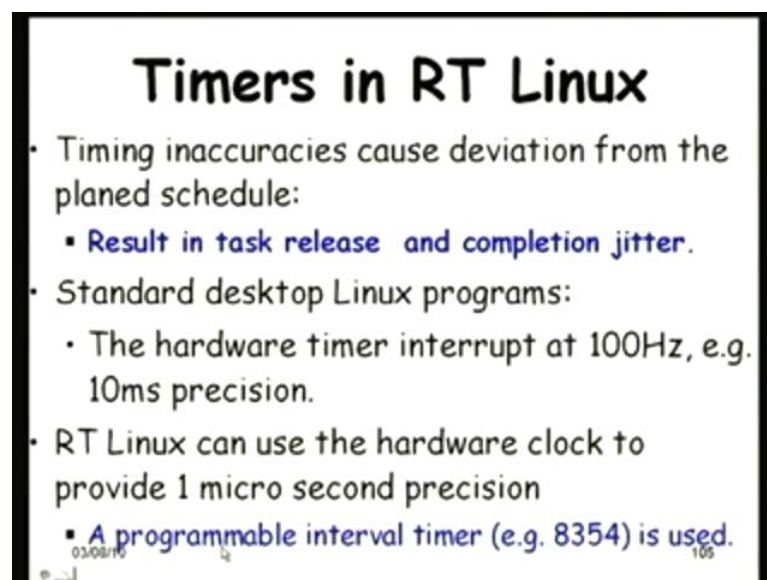
Performance Improvement in RT Linux

- RTLinux scheduler:
 - Treats Linux kernel as the lowest priority real-time task.
- Users can use their own schedulers:
 - Used as loadable kernel module
 - Makes it possible to experiment different scheduling policies

03/08/10 104

The RT Linux scheduler treats the Linux kernel as the lowest priority real time task and not only that the users can use their own schedulers as loadable kernel modules or the kernel plug-ins. And you can try out different scheduling policy, RT Linux supports that and we had actually done some experiments on this.

(Refer Slide Time: 41:37)



Timers in RT Linux

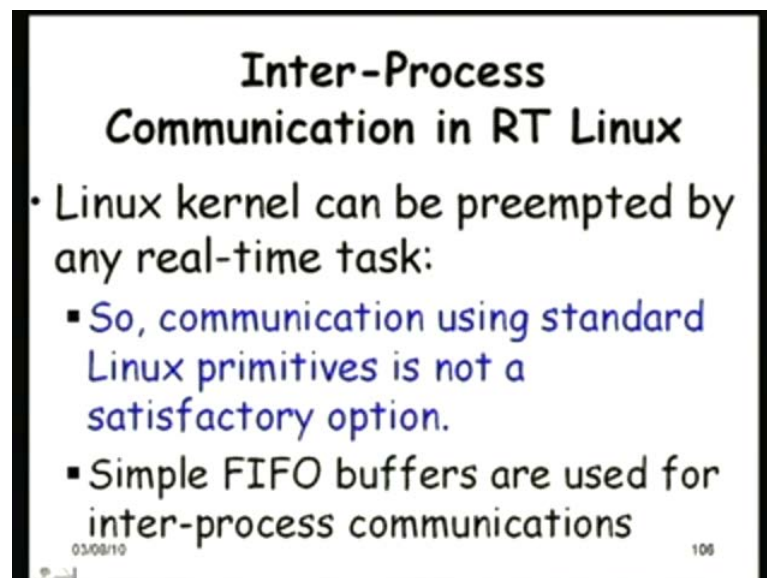
- Timing inaccuracies cause deviation from the planed schedule:
 - Result in task release and completion jitter.
- Standard desktop Linux programs:
 - The hardware timer interrupt at 100Hz, e.g. 10ms precision.
- RT Linux can use the hardware clock to provide 1 micro second precision
 - A programmable interval timer (e.g. 8354) is used.

03/08/10 105

Now, I think some of the question that you had asked about a timers, how these are supported in RT, Linux timing accuracies are very important for real time applications, any deviation or inaccuracy from the a precise time can cause deviation from the planned schedule. And this can result in both task release jitters and completion jitters, I hope you understand what is a jitter **right**, we were discussing I think in some context. Jitter is basically the earliest time the task can arrive here, it is a release time, the earliest time the task can arrive and the latest time the task can arrive, and naturally, if there is a release time jitter it would typically soak in completion time jitter as well.

And even otherwise for completion time jitter, if there is timing in accuracy even if the task release times are similar still they consider jitter. The standard Linux programs use a hardware timer interrupt which has 10 millisecond precision, but here 1 microsecond precision in RT Linux is provided by using a programmable interval timer 8354.

(Refer Slide Time: 43:12)



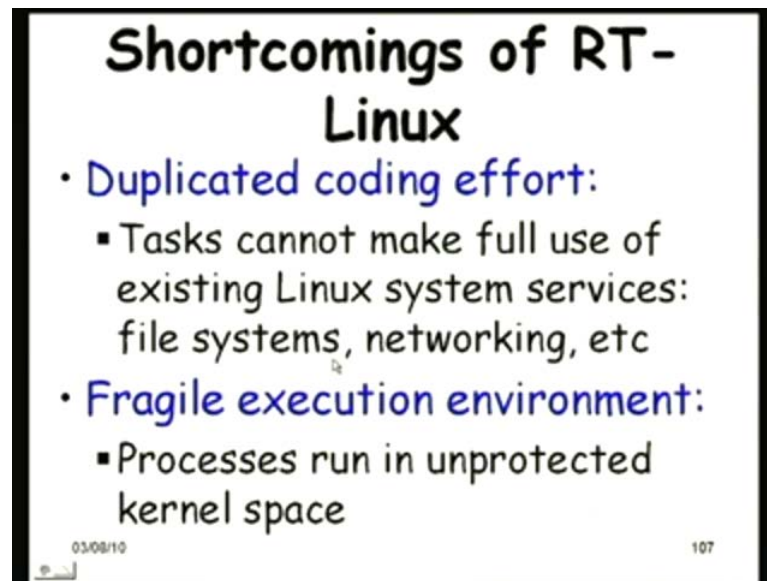
Inter-Process Communication in RT Linux

- Linux kernel can be preempted by any real-time task:
 - So, communication using standard Linux primitives is not a satisfactory option.
 - Simple FIFO buffers are used for inter-process communications

03/08/10 106

Now, let us see the inter process communication in RT Linux, see one of the major difficulty here, I think we are just mentioned it briefly that the Linux kernel is preempted by any real time task, and therefore, communication using standard Linux primitives is not a satisfactory option, we cannot use the message passing or pipes and so on, typically the communication occurs through FIFO buffer or shared memory.

(Refer Slide Time: 43:50)

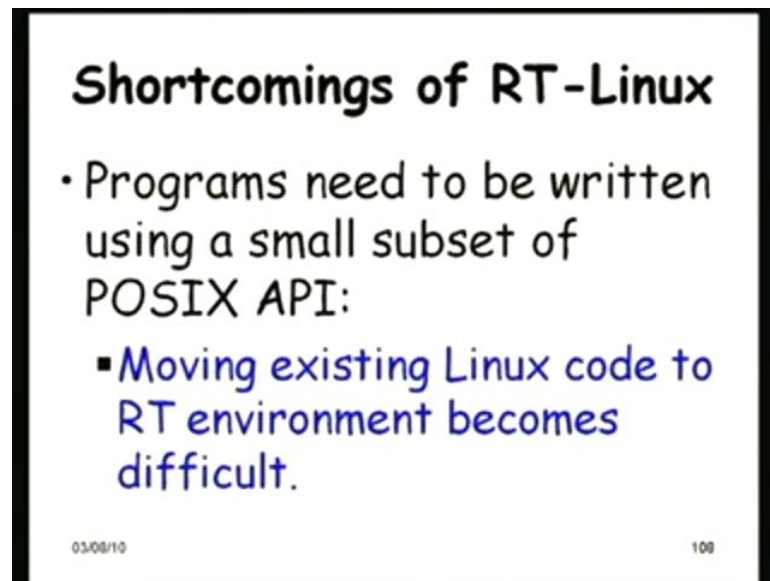


But, let us be aware of the problem that has to be encountered when using the RT Linux, one is duplicated coding effort, you know that the traditional kernel has many services, large amount of code is there and see here you are writing a real time task, but you cannot have all those services available to you, for example, file systems, networking etcetera, these are there in traditional real time operating system.

But to have them in the real time kernel, you will have to again write the same code that is there here or possibly port here, but you know porting to kernel is very difficult, just try writing some kernel programs and see how difficult it is. And one of the main reason of these porting not only the coding effort is duplicated you have to implement networking etcetera here, but also the difficulty is that fragile execution environment, because they run in unprotected space, when you try to put they will crash the system, and once the system is crashed how you debug it **right**, again you examine the code and try to find out where the error was and then - I mean- debugger support becomes absent.

So, it is just basically changing the code and again running and seeing that the system had again crashed and then making again changes and trying out, it is very and each time you have to boot and run it you know, it is time consuming writing a kernel program.

(Refer Slide Time: 45:45)



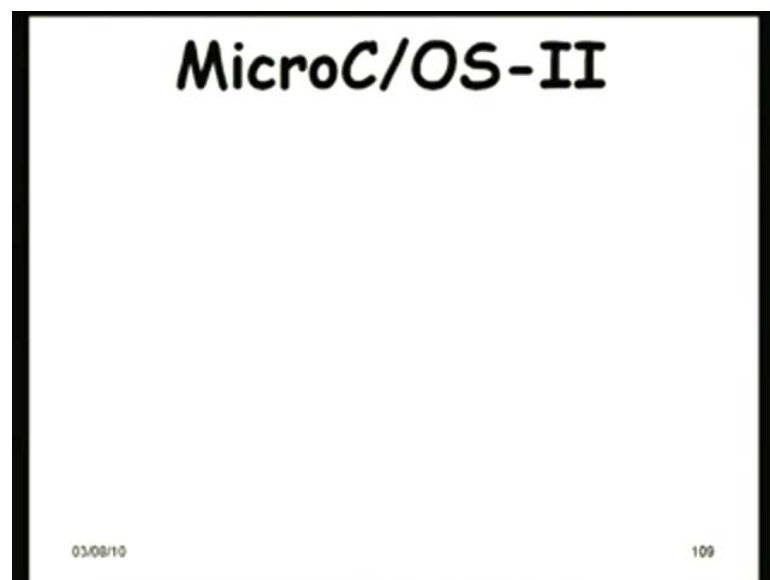
Shortcomings of RT-Linux

- Programs need to be written using a small subset of POSIX API:
 - Moving existing Linux code to RT environment becomes difficult.

03/08/10 109

The other thing is that in RT Linux if you look at the documentation and also the code, it is the POSIX API, only a small subset of that is supported, and as we are saying that moving the services available in traditional Linux to real time environment is difficult.

(Refer Slide Time: 46:11)

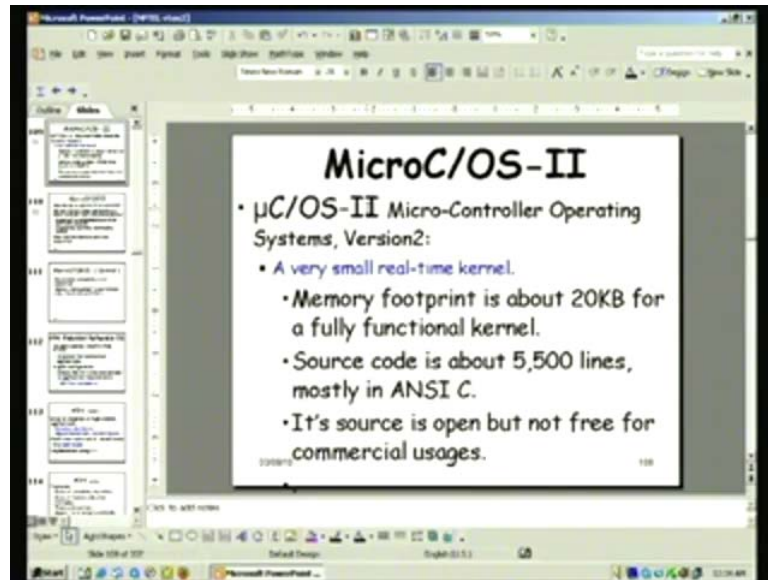


MicroC/OS-II

03/08/10 109

Now, let us see some other free operating systems that are also popular and being used, specifically the micro COS and the QNX and the VS works.

(Refer Slide Time: 46:35)



So, let us look at the micro cos, there is it is called as the micro controller operating system very small kernel, because it is to be used with a micro controller and **we are know the** other they are saying that micro controllers are being used everywhere, the smallest application, even I think we are just discussing about the pen drive, and we are seeing that controlling inside that small device micro controller controlling where the bits are been - I mean - the blocks are getting return and so on.

So, micro controllers are extremely used in extremely cheap and small devices and you need an operating system in micro controllers and the micro C is the one of the example operating system. So, mu C O S 2 is, the term you might find, the micro controller operating system we are discussing about that. So, it has a very small real time kernel, takes typically 20 kilo bytes for the full kernel extremely small operating system.

(Refer Slide Time: 47:54)

MicroC/OSII (Contd)

- Round robin scheduling is not supported
- Memory management is performed using fixed size partitions.

03/08/10 111

It is a small operating system basically targeted for micro controller applications and here it is only the FIFO scheduling, Round robin scheduling is not supported, and it does not use virtual memory, very simple operating system. Memory management using fixed size partitions.

(Refer Slide Time: 48:17)

eCos (Embedded Configurable OS)

- An open source, royalty-free, RTOS:
- Intended for embedded applications.
- Highly configurable:
 - Allows the OS to be customised to application requirements.

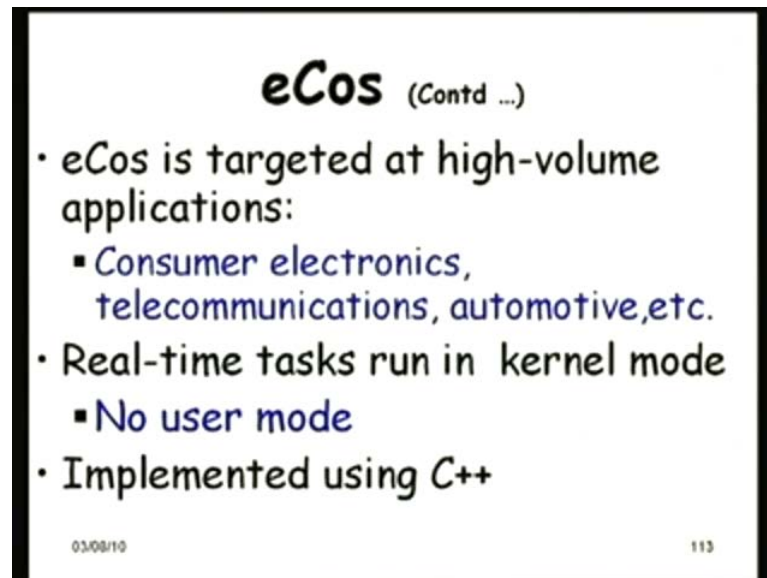
<http://ecos.sourceware.org>

03/08/10 112

You can also find eCOS easily downloadable, this the website, eCOS dot source ware dot org, open source a royalty free real time operating system, again intended for

embedded applications, you can configure it for specific features that you need from the operating system and leave out the rest, you can customize to application requirements.

(Refer Slide Time: 48:48)



eCos (Contd ...)

- eCos is targeted at high-volume applications:
 - Consumer electronics, telecommunications, automotive, etc.
- Real-time tasks run in kernel mode
 - No user mode
- Implemented using C++

03/08/10 113

Typically targeted for high volume applications like consumer electronics, telecommunications, automotive, I think we are just mentioning that in automotive applications, there are hundreds of processors that run operating systems and every vehicle have, and it is hard to find a vehicle on the road which does not use at least 10 or so processes. And here in eCOS the real time tasks run in kernel mode and in fact, there is no user mode concept here, everything runs in kernel mode and implemented in C plus plus.

(Refer Slide Time: 49:32)

eCos (Contd ...)

- **Features:**
 - Choice of scheduling algorithms
 - Choice of memory-allocation strategies
 - Timers and counters
 - Support for interrupts and DSRs
 - Exception handling
 - Host debug and communications support

03/08/10 114

The features include the choice of scheduling algorithms; you can use your own schedulers also. You have choice of memory allocation strategies, the timers and counters, even you can plug in support for interrupts and DSRs are available, exception handling, host debug and communication support, so these are some of the features **ok**.

(Refer Slide Time: 49:59)

VxWorks

- **Monolithic Kernel**
- Supports RT-POSIX in addition to its own APIs
- Not a multiprocessor OS
- **Uses MMU:**
 - Provides virtual-to-physical memory mappings.

03/08/10 115

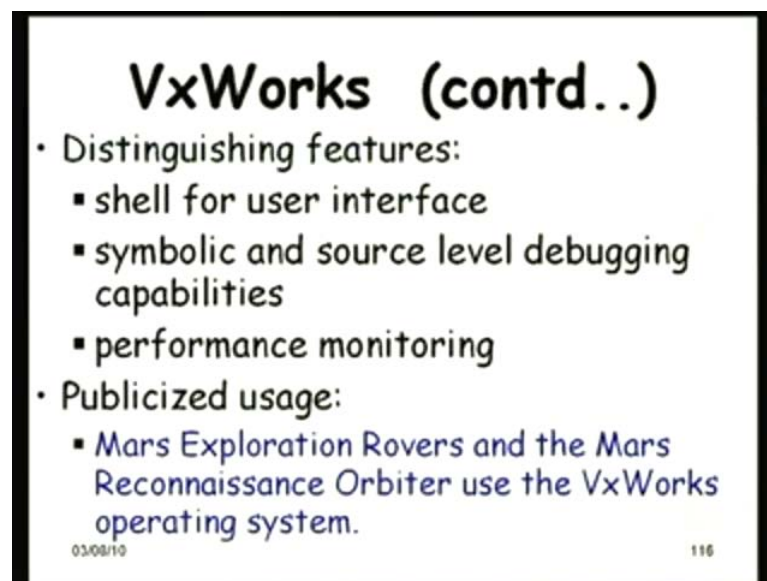
Now, let us look at V X works, it is extremely popular I think in some context we had mentioned about V X works.

(())

Exactly, I think some of you remember about the mars path finder, we are just discussing that in the context, in what context did we discussed that was regarding the resource sharing, real time resource sharing and priority inversions - unbounded priority inversion - that was causing the mars path founder problem. So, there we had said that V X works had a feature for supporting the protocol - inheritance protocol - and which was not turned and so on, yes ok.

So, if say monolithic kernel operating system supports the real time POSIX standards, and in addition, it also supports its own application programming interfaces. It was not a multiprocessor operating system, because it is quite old, for long time it is existing and it uses the memory management unit to provide virtual to physical memory mapping, so not really meant for applications like a real time micro controller based applications, not extremely small applications, but relatively larger applications.

(Refer Slide Time: 51:27)



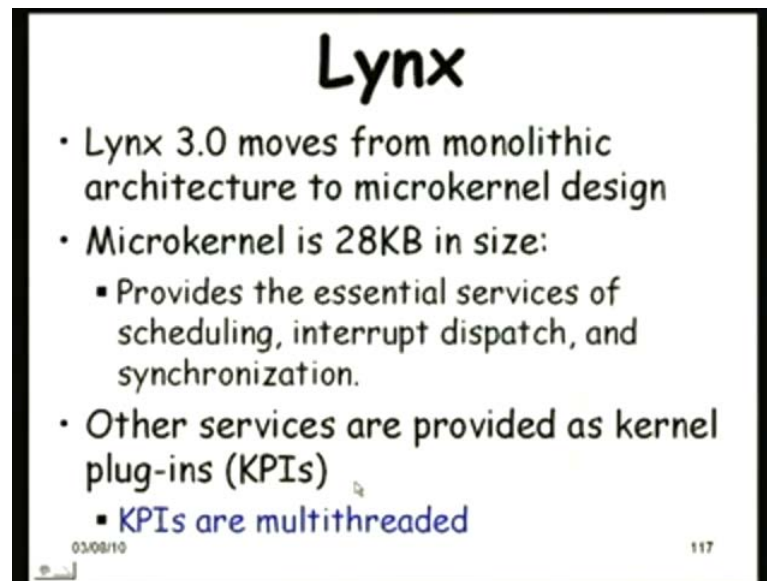
VxWorks (contd..)

- Distinguishing features:
 - shell for user interface
 - symbolic and source level debugging capabilities
 - performance monitoring
- Publicized usage:
 - Mars Exploration Rovers and the Mars Reconnaissance Orbiter use the VxWorks operating system.

03/08/19 116

So, it has user interface, sales and, symbolic and source level debugging capacities, performance monitoring and tuning etcetera, are available. So, the most publicized uses of V X works as we are discussing was the mars rover and mars reconnaissance mission that was the orbiter which both use the V X works operating system.

(Refer Slide Time: 51:56)



Lynx

- Lynx 3.0 moves from monolithic architecture to microkernel design
- Microkernel is 28KB in size:
 - Provides the essential services of scheduling, interrupt dispatch, and synchronization.
- Other services are provided as kernel plug-ins (KPIs)
 - KPIs are multithreaded

03/08/10 117

And we will look at a few other operating system, using the lynx operating system which is also a very popular real time application development operating system, which at present is a monolithic - it has moved out from the monolithic architecture and at present it is a microkernel based design.

So, we had discussed about the advantages of a microkernel based design isn't it, especially for the real time application. We had said that for a traditional operating system microkernel may not be that advantages, we have discussed the details of a microkernel versus a monolithic operating system. So, indeed some time for discussing the Linux and the QNX, and few other operating systems, before we look at windows e, so today we are running out of time we will just stop here, thank you.