

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

Lecture - 19
Some Basic Issues in Real - Time Operating Systems

Welcome to this lecture. So far, we had looked at some basic concepts in real time operating systems and then we had said that the scheduler tasks scheduler is actually the most important part of any real time operating system because it is the one which takes the primary responsibility in meeting a tasks deadline. We had also looked at resource sharing problem in real time operating systems and we had looked at protocols to help solve the problems that arise during resource sharing.

This lecture will continue with some basic issues that arise in real time operating systems and after that will look at whether Unix can be used as real time operating systems what problems may arise if we use Unix as it is and how it can be extended. So, let us get started with the basic issues in real time operating systems.

(Refer Slide Time: 01:41)

Basic Requirements of an RTOS

- Real-time priority levels
- Real-time task scheduling policy
- Support for resource sharing protocols
- Low task preemption times:
 - Of the order of milli/micro seconds
- Interrupt latency requirements

267

One of the most crucial requirement is of course, support for real time priority levels what we mean by real time priority levels is that once the programmer assigns priority to the tasks.

The operating system does not change it. So, these are static priority levels, but as you will see that the traditional operating systems change the priority of tasks dynamically real time task scheduling policy this is one of the central requirements of any real time operating systems support for resource sharing protocols because without use of proper resource sharing protocols.

There can be priority inversions and tasks may miss their deadline low task preemption times the order of milli or micro seconds when a higher priority task becomes ready the low priority tasks that is executing must be preempted and this preemption time is the order of milli micro seconds; the interrupt latency requirements once the interrupt occurs it must be recognized and the interrupt servicing must occur within some bounded small time.

(Refer Slide Time: 03:23)

Additional Requirements of an RTOS

- Memory locking support
- Timers
- Real-time file system support
- Device interfacing



Additional requirements and real time operating systems are memory locking support because in a virtual memory system the page is replaced if it is not used and therefore when a page fault occurs it can take a long time and introduce jitters will just look at this issues and that is a reason why one of the basic requirement for an operating system to serve as a real time operating system is a support to locking memory and this locked pages will not be subject to page swapping support for timers both periodic timers and one set timers high precision timers.

These are frequently required in writing real time applications and need to be supported by the operating system real time file system support the traditional file system the blocks are written based on where spaces available under desk, but then the access time of the blocks where is depending on where it is located on the desk in a real time file system the blocks are written conjicatively. So, that the access time to the blocks becomes predictable and finally.

The real time operating systems should facilitate device interfacing because many of the real time systems they incorporate many sense sensors actuators and. So, on using a new type of sensor or actuator should not require to rewrite part of the operating system the device drivers for these devices should be able to be incorporated on the fly do not have to really shut down the system recompile the operating system and so on should not take place.

(Refer Slide Time: 05:46)

Support for Real-Time Priority Levels

- **Static task priority level (or real-time priority level):**
 - Operating system does not change programmer assigned task priority.
- **Dynamically changing task priority:**
 - Supported by traditional operating systems.
 - The objective is to maximize system throughput.

269


First let us look at the real time priority levels the real time priority level implies static priority level that is once a priority is assigned to a task by the programmer should not change. The operating system should not change a programmer assigned priority, on the other hand, the dynamic priority; the operating system keeps on changing the priority level of a tasks you might wonder that why does the operating system needs to do this; will see this in some detail in this lecture or possible in the next lecture that all traditional

operating system, they change the tasks priority after every times lies the main reason is to enhance the throughput of the operating system.

(Refer Slide Time: 06:59)

Task Scheduling Support

- Should allow programmers choice of real-time task schedulers such as:
 - RMS
 - EDF
 - Custom task schedulers




Tasks scheduling support; we had seen that task scheduler is central aspect of any real time operating system and therefore, the real time operating systems should support task schedulers like rate monotonic scheduler earliest deadline first scheduler and other custom task schedulers.

(Refer Slide Time: 07:24)

Resource Sharing Support

- Should support resource sharing protocols such as PCP:
 - To minimize priority inversions during resource sharing among real-time tasks.




Resource sharing support; so, tasks in a non trivial application need to share resources for example, the results of one tasks needs to be pass done to another task they may use a shared memory for this they may use shared device and so on and we have seen that the traditional operating systems solution to resource sharing which is the sema force leads to priority inversions and unbounded priority inversions and we had seen protocols like priority ceiling protocol the real time operating system need to support the priority ceiling protocol. So, that that high priority tasks do not miss their deadlines due to the priority inversion problem.

(Refer Slide Time: 08:22)

Task Preemption Time

- RTOS task preemption times:
 - *Of the order of a few micro seconds.*
- Worst case task preemption times for traditional operating systems:
 - *Of the order of a second.*
- The significantly large latency:
 - *Caused by a non-preemptive kernel.*



The task preemption time need to be low and bounded when a high priority task becomes ready the low priority task must be contex switched and the high priority task must run and that should be of the order of few micro seconds because the task deadline is hundreds of micro seconds. So, if the task preemption time is hundreds of micro seconds or a second or several milli seconds then slightly that the hard real time tasks will miss their deadline.


Now, next lecture will look at Unix as a real time operating system and we will see that one of the major problem is that task preemption time in Unix is the order of the second and therefore, if a hard real time task has deadline of a 100 micro second or something, then the tasks preemption time itself is going to cause a deadline in this, but then you might be wondering at this point that why is that the traditional operating systems like

Unix have such a high task preemption time of a second or something will see this issue in slightly more detail, but right now, let us just understand that the kernel of a traditional operating system is non preemptive what we mean by non preemptive is that when a interrupt occurs the kernel does not service it.

Actually, the kernel disables when in the kernel mode again that will raise many questions in your mind that why does the kernel have to do it that it disables all interrupts when executing kernel routines and why does it have to be a second and so on will look at these point in more detail, but right now will just mention that the large preemption time is due to the non preemptive kernel and the kernel disables interrupts when executing kernel routines.

(Refer Slide Time: 10:59)

Interrupt Latency Requirements

- **Interrupt latency:** 
- The time delay between the occurrence of an interrupt and the running of the corresponding ISR.
- The upper bound on interrupt latency:
 - **Must be bounded and less than a few micro seconds.**

273

There are also interrupt latency requirements on a real time operating system what we mean by the interrupt latency is that.


When an interrupt occurs and by the time the interrupt service routine runs for that interrupt the time must be small and bounded; let us look at this diagram at this point an interrupt occurs the interrupt is recognized by the operating system and then it does some processing for example, saving the occurrent set of registers and so on and then it runs the interrupt service routine. So, the point where the interrupt occurs the time point at which interrupt occurs to the time point where the interrupt service routine in starts running, we call it as a interrupt latency time and for traditional operating system this

time is large of the order of a second, but then in for a real time operating system, this time should be of the order of few micro seconds and also the time bound on the interrupt latency must be deterministic that is once we say that the bound is that is a 10 milli seconds in no case, it should exceed 10 milli second, we should be able to give a deterministic bound on the latency time.

(Refer Slide Time: 12:44)

How Low Interrupt Latency is Achieved?

- Perform bulk of ISR processing:
 - As a queued low priority task called deferred procedure call (DPC).
- Support for nested interrupts requires:
 - Not only preemptive kernel routines.
 - Should be preemptive during interrupt servicing as well.



But then you might wonder that how does such low interrupt latency is achieved in real time operating system because when a interrupt service routine is running, it may take several micro seconds and in the mean while if another interrupt occurs then there will be problem, it has to keep on waiting the way in which the many of the real time operating systems achieve low interrupt latency and is by performing much of the interrupt service routine as a deferred procedure call which runs like any other task is acute as low priority task called as deferred procedure call.

Therefore, as soon as a interrupt occurs the code that runs is a very small code that creates the deferred procedure call and queues it up as you will see that this is the main technique that is used to achieve low interrupt latency for the traditional operating system not only that the kernel should become preemptive that is even if it is running in the kernel mode the kernel code is executing is still is should be able to preempt kernel code and also should be able to handle nested interrupts that is when the interrupts

service routines itself is running further interrupts should be recognized and should be able to process it.

(Refer Slide Time: 14:47)

Requirements on Memory Management

- Traditional operating systems support:
 - Virtual memory and memory protection features.
- Not supported by embedded RTOS:
 - Increase worst-case memory access time drastically.
 - Result in large memory access jitter

275

There are requirements on memory management; for example, in the virtual memory system and in the memory protection features we need improvement of course, in a very small embedded real time operating systems where the task size is a small only very few tasks and the overhead of the operating system needs to be very small the foot print or the size of the operating system need to be small then the virtual memory is not supported and even the memory protection feature is whether one task can unintentionally over write the results of the another tasks or the code of another task is not guaranteed by the operating system.

Because that would require the additional overhead and we need only a very small size operating system and therefore, many of the embedded real time operating systems do not really support virtual memory and memory protection systems, but unless we do something for let us say a non embedded applications or where in embedded application where you can afford to have large sophisticated operating system.

We need to do something. So, that the worst case may memory access time is not increased drastically will just look at that is because unless we do that there will be memory access jitter what we mean by memory access jitter is that if the data is

sometimes obtained very fast and sometimes obtained after considerable delay variation in access time is called as the jitter in a virtual memory operating system.

If the page is resident in memory or in the cache then the access time is very first very quickly the data is obtained by the processor, but if the page is not available on the memory and page fault occurs, then the hard disk needs accessed and the access time becomes hundreds of time larger than when it is available in the main memory we need to do something to prevent such access memory access jitter times otherwise this will lead to tasks missing their deadline.

(Refer Slide Time: 17:39)

Virtual Memory

- Virtual memory technique helps reduce average memory access time:
 - But degrades the worst-case memory access time.
 - Page faults incur significant latency.
- Without virtual memory support:
 - Providing memory protection is difficult.
 - Also, memory fragmentation becomes a problem.

Virtual memory which you might have studied in first level operating system course it helps to reduce the average memory access time, but then the worst case memory access time degrades because if the page is not available in memory a page fault occurs and then it has to be obtained from the hard disk and the page faults incur significant latency, but the virtual memory also has memory protection as a byproduct all virtual memory systems they support memory protection feature that is each task can access only those part of the memory for which it is entitled.

It cannot really access the entire memory space that is all the data that is resident in the computer and this is called as the memory protection feature and if for some reason we do not support virtual memory may be because the operating system needs to be small embedded system and we cannot really afford to support a virtual memory system then

memory protection becomes a issue because without memory protection if there is a error in the operating system code, it becomes very very difficult to debug because the code the operating system code is over written in the when there is no memory protection and it just crashes you cannot find out where exactly it is over writing and so on and also without virtual memory support memory fragmentation becomes a problem.

(Refer Slide Time: 19:52)

Do Any RTOS Support Virtual Memory?

- RTOS for large applications need to support virtual memory :
 - To meet memory demands of heavy weight real-time tasks.
 - Support running non-real-time applications: text editors, e-mail client, Web browsers, etc.


277

Naturally a question arises which are the types of the real time operating systems which support virtual memory, we need virtual memory when the tasks require large data or the code is significant that is to meet the memory demands of heavy weight real time tasks which are large code or require large data; we need virtual memory system and also, if we need to support running non real time applications in our system like text editor email client web browsers etcetera it becomes necessary to support virtual memory part of the real time operating system.

(Refer Slide Time: 20:39)

Memory Protection: Pros and Cons

- Advantage of a single address space:
 - Saves memory bits and also results in light weight system calls.
 - For very small embedded applications:
 - Memory overhead can be unacceptable.
- Without memory protection:
 - The cost of developing and testing a program increases.
 - Also, maintenance cost increases.



But let us look at memory protection we said that each task has its own address space a task cannot access the address space of another task and that is ensured by the memory protection mechanism when we do not have memory protection all tasks operate in the same address space the advantage of having a single address space that it is efficient no checking of protection bits etcetera is necessary and also it saves on the memory its required to ensure protection.

Therefore the calls become light weight that is efficient and require also less memory for very small embedded applications which require only one or two small tasks which the programmer can easily manage then memory protection may be done away with it, but for applications having many tasks it becomes very difficult to work without memory protection.

So, for very small embedded applications with just a small memory and a small number of task memory overhead in the form of protection becomes unavoidable sorry unacceptable and the programmer can easily manage to program the application even without memory protection, but then for larger application without memory protection the cost of development of the program increases because debugging becomes very hard one task can over write it the data or the code of the another task and not only that maintenance cost increases later you want to change something becomes very difficult the change application.

(Refer Slide Time: 22:46)

Memory Locking

- **Memory locking:**
 - Prevents a page from being swapped from memory to hard disk.
- In the absence of memory locking support:
 - Even critical tasks can suffer large memory access jitter.

279

If virtual memory supported by a real time operating system then we need the memory locking feature in the memory locking feature?

The programmer can lock a memory page it prevents the page from being soft from the memory to the hard disk and therefore, the jitter in memory access reduces in the absence of memory locking support even the critical task can suffer large memory access jitter and therefore, the tasks scheduling has to be extremely conservative and still the tasks may miss their deadline because of this memory access jitter.

(Refer Slide Time: 23:32)

Asynchronous I/O

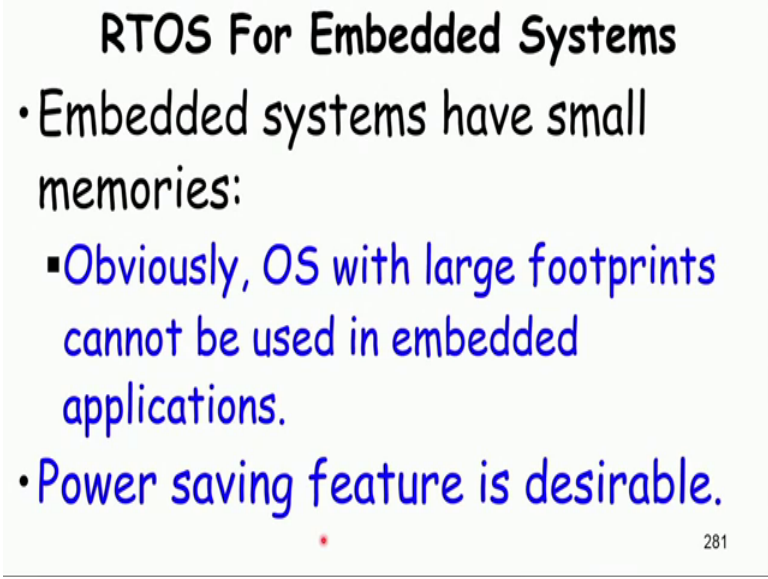
- Traditional read(), write() system calls:
 - Synchronous I/O.
 - Process is blocked while it waits for the results.
- Asynchronous I/O:
 - Non-blocking I/O.
 - aio_read(), aio_write()

280

The real time operating systems also need to support asynchronous I O the traditional I O in operating system is synchronous that is a process blocks when it waits for the result; for example, you are doing a what processing application and you want to save the file while the file is being saved you can do anything because it is a synchronous call, but if you are given a same command and at the same time you are able to do also do editing and other operations.

Then it is asynchronous I O also called as a non blocking I O you can initiate the I O and then continue operating and call such as asynchronous I O read asynchronous I O write these need to be supported by a real time operating system because I O is typically slow and if a process has to a task process has to block on account of a I O then may be sits deadline becomes very difficult to program a real time task without the support of asynchronous I O.

(Refer Slide Time: 24:51)



RTOS For Embedded Systems

- Embedded systems have small memories:
 - Obviously, OS with large footprints cannot be used in embedded applications.
- Power saving feature is desirable.


281

Also if the real time operating system is to be used for embedded systems then the size of the operating systems sometimes called as a footprint of the operating systems needs to be very small the other feature that is desirable for embedded applications is the power saving feature the operating system should take responsibility to save power that is run the processor in low frequency mode when the load is not high switch off the processor part and so on when the load is not high and that is how conserve the power.

(Refer Slide Time: 25:43)

Using Traditional Unix as RTOS

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



So far, in this lecture, we looked at what are the features that set apart a real time operating system from a non real time operating system we identified about a dozen features which an operating system needs to support in order to be called as a real time operating system to make the issues clear and to get a deeper inside in to these aspects will consider what problems may occur if the traditional Unix is used as a real time operating system and what is the origin of this problems because unless we know this issues well we cannot be able to really develop a real time operating system.

If we want to use Unix as a real time operating system we will find many problems actually all the problems that we all the issues that needs to be supported by a real time operating system that we identified in this lecture are not satisfactorily supported by Unix, but then there are two problems which really standout one is the non preemptable kernel when the Unix operating system is running the operating system code interrupts are disabled and the second point is dynamic priority levels.

Even if the programmer assigns a priority to a task the operating system keeps on shifting it the end of every times lies this lecture we are almost at the end of the time will just look at these issues because these are crucial issues to understand why a traditional operating system cannot be used as a real time operating system and also why this problems are there with a traditional operating system in the first place with this will just conclude this lecture and we will continue from this point in the next lecture.