Real-Time Systems

Prof. Dr. Rajib Mall

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

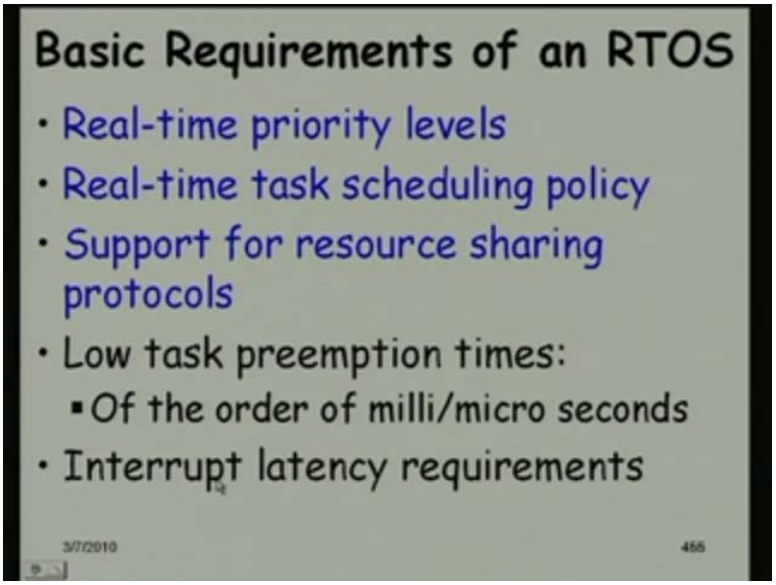
Module No. # 01

Lecture No. # 21

A Few Basic Issues in Real-Time Operating Systems

Good morning. So, let us get started today. If you remember last time, we were discussing some issues on, issues on scheduling and clock synchronization. Today, we will look at some basic issues in real-time operating systems. With this basic knowledge, that we will have today, we will slowly start discussing about commercial real-time operating systems that are actually used to developed real-time applications.

(Refer Slide Time: 00:53)



So, first, let us try to understand what are the requirements of a real-time operating systems and how is it different from a traditional operating system that we are used to like windows, unix, etcetera. The first, possibly most important requirement is to have real-time priority levels. Actually windows, unix and other non real-time operating

systems, they use a dynamic priority level for tasks. Priority level assigned by the programmer keeps on changing as the program executes.

As we proceed, we will see the reason why they do that, but if that happens for real-time tasks, that would be unacceptable, because high priority task can miss their deadline. So, we need real-time priority levels, we will also call this as static priority levels. Whatever priority level is assigned by the programmer should remain throughout the execution unless the programmer changes it.

The next requirement is a proper real-time tasks scheduling policy. The scheduling policy used by traditional operating systems is to maximize throughput, but here, maximizing throughput is not a objective. Here, the objective is to let each task meet their deadlines and we had discussed about uniprocessor algorithms like rate monotonic earliest deadline. First, should be able to somehow support those and possibly should, it should support the programmer to incorporate his own scheduling policy. For example, for some task, some application might find that deadline. Monotonic is more suited than rate monotonic algorithm. We should be able to use that.

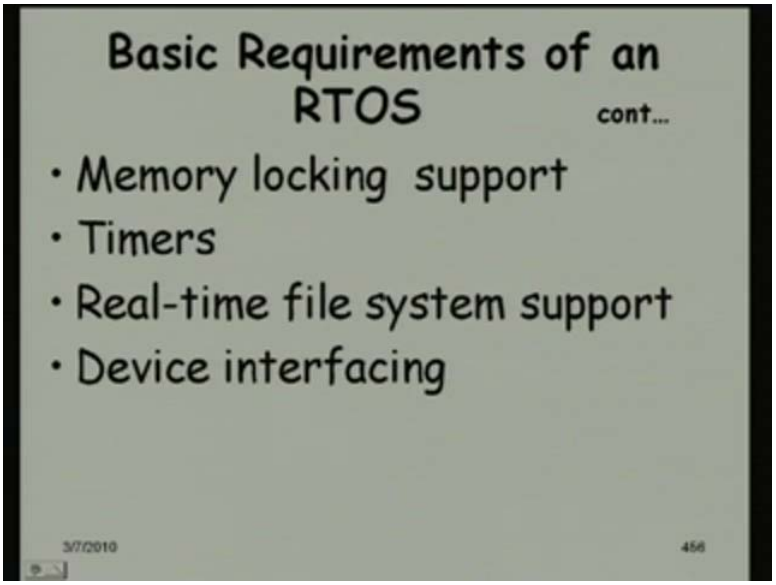
The third most important thing possibly is to let this tasks share resources without priority inversions. We had seen in our previous discussions that whenever tasks share resources, that is critical sections or non-preemptible resources. Priority inversions occur, that is the simplest case and we can have unbounded priority inversions.

So, we had discussed some protocols where priority inversions can be minimized. Priority inversion cannot be eliminated when resource sharing occurs, but it can be minimized and we had seen some protocols. Do you remember anybody Rremembers the protocols? Yeah, inheritance is the basic scheme and then we looked at the ceiling protocol.

So, the traditional resource sharing in operating systems like mutex semaphores, etcetera, cannot directly be used here for real-time tasks. Then, there are several other requirements for example, low task preemption times. Once we have a higher priority task arriving or under certain situation, a task need to be preempted, and the preemption time should be of the order of milli or microseconds, because that is the application the real-time tasks typically run in few milliseconds or several microseconds.

So, if the preemption time itself is of the order of second, you cannot use it, but unfortunately, the traditional operating systems, the preemption time is of the order of second. We will see the reason why tasks cannot be preempted in milli and microseconds, and then, the interrupt latency requirements, low interrupt latency.

(Refer Slide Time: 05:23)



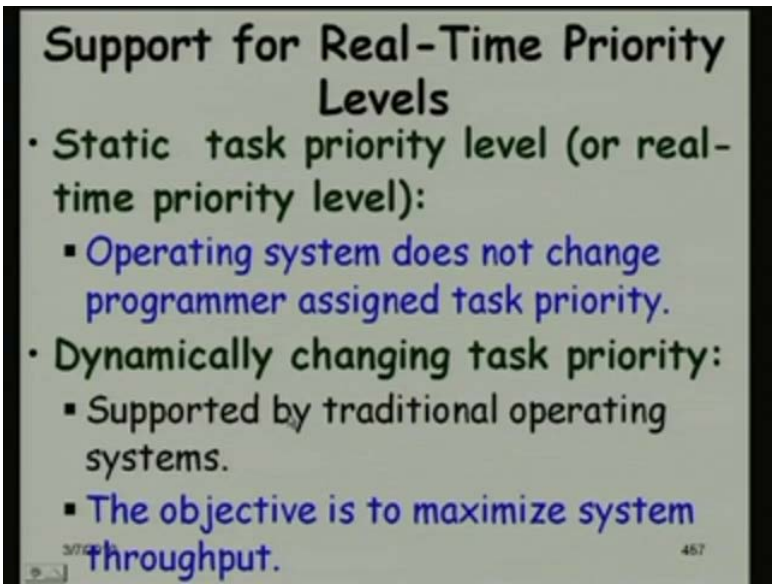
We need to have Memory locking support. A task should be able to lock its pages in the memory; they should not undergo page swapping in the hard disk and need timers. We had discussed about two categories of timers. What are the two categories? Yeah, the one set timers and the periodic timers they need to be supported, real-time file system support.

So, the file system block should be written contiguously cannot really have a traditional file system where the blocks are written as is available in the desk to have predictable file access, need to have a real-time file systems support, device interfacing support, because it should be able to support additional devices as, and when they become available without really having to compile the kernel, should be able to easily have device drivers incorporated into the operating system.

(( )).

Exactly should be able to, even if the system is operating like he says that plug and play, yes, even if the system is operating, we should be able to attach additional devices, because some applications cannot stopped at all.

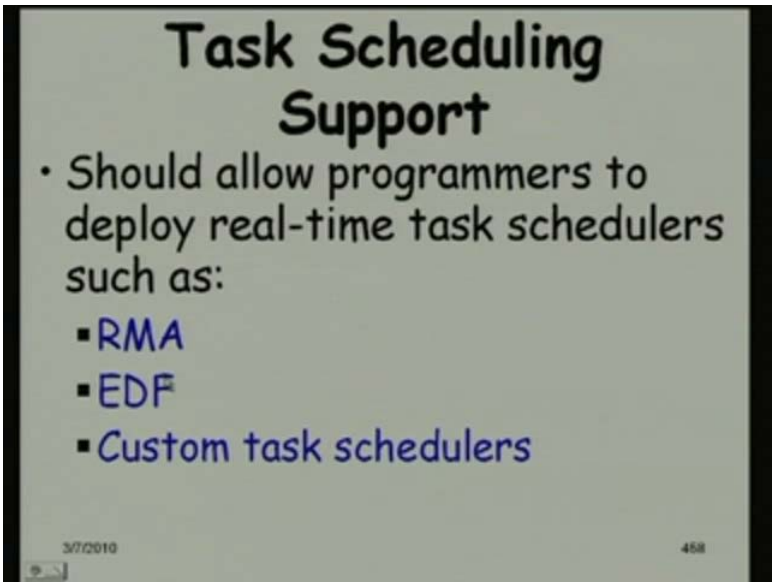
(Refer Slide Time: 06:59)



Now, let us look at these issues. Some of the important issues more carefully as we were saying that support for static priority levels or real-time priority levels is possibly a very important requirement for any real-time operating system unless this is there, you cannot really run any meaningful real-time tasks.

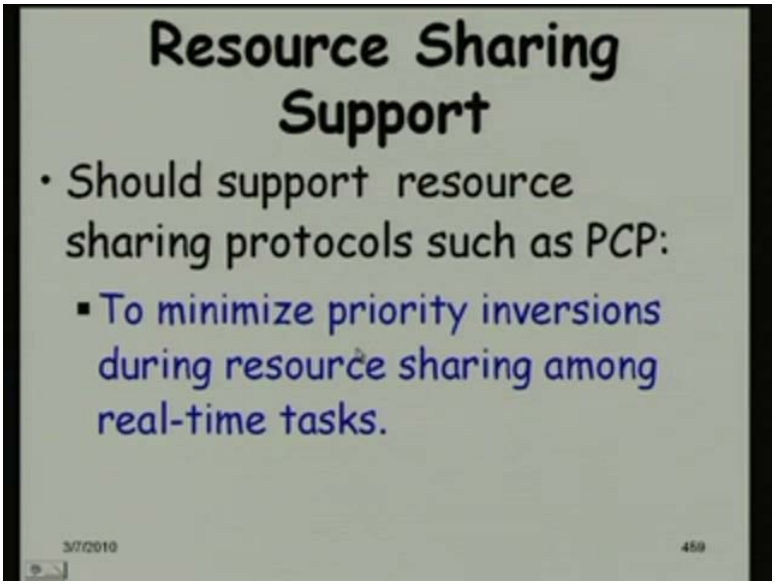
In contrast, the traditional operating systems change the task priority dynamically, and in the, when we have a support for static task priority level, once the programmer assigns a task priority, it remains throughout until the programmer changes it or the task completes, cannot change, the system does not change it. In a dynamically changing task priority which is supported by traditional operating system, you might wonder why does it keep on changing the task priority. The idea is to maximize system throughput. We will see later how dynamically changing task priority level can help improve the task throughput.

(Refer Slide Time: 08:23)



The task scheduling support that is needed is that, should be able to deploy real-time tasks schedulers. For example, the rate monotonic algorithm, earliest deadline first algorithm or may be some custom task schedulers as may be required for specific applications.

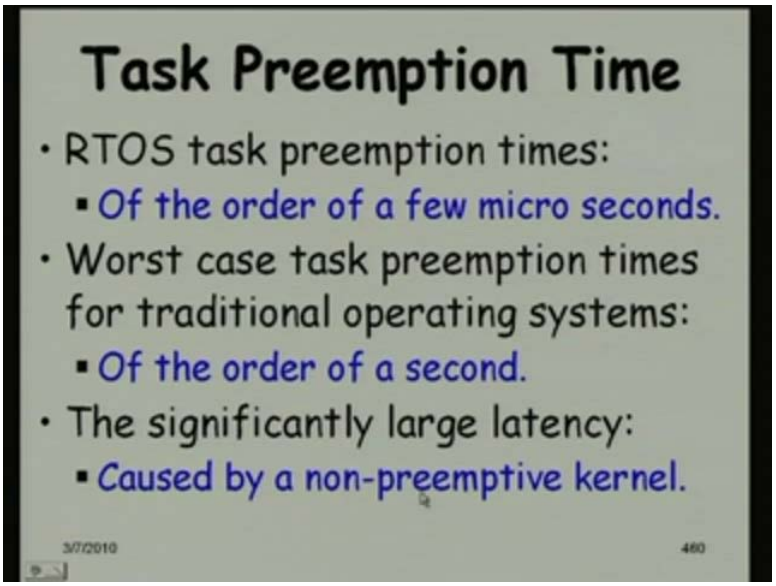
(Refer Slide Time: 08:47)



Need to have resource sharing support among the real-time tasks should support resource sharing protocols like priority ceiling protocol, we, by know what is a priority ceiling

protocol. The idea is to minimize priority inversions during resource sharing among real-time tasks.

(Refer Slide Time: 09:10)



The task preemption time should be of the order of a few microseconds or milliseconds depending on the application, but we will see that in the traditional operating system, the task preemption time is of the order of a second. So, why is such a big difference here? The main reason why the traditional operating systems have a very large task preemption time is due to a non preemptive kernel. I think as we, next couple of lectures, we will be talking about why the traditional operating systems have a non preemptive kernel, that is, in kernel mode, all interrupts are must; interrupts are disabled as long as the, it operates in the kernel mode.

Why it needs to do that? We will discuss about that and a corollary of this to improve the task preemption time. We need to have a preemptive kernel. So, the real-time operating systems need to have a preemptive kernel.

(( ))

Yes.

So, later see the question is that why do a non-preemptive kernel increase the preemption latency? Later, we will discuss it in more detail, but just brief answer to that is that, see,

if a system service takes, let us say one second. Let me just ask you this question first that, let us say a application program, real-time application program is running in the user mode. Later, we will see that many of the real-time operating, there actually run in the kernel mode, but suppose, they run in the user mode. So, how does it invoke some operating system services? Because every application needs services from the operating system, system calls. So, give some example of a system call - fork or let us say access some i o devices, because processors, user processor not allowed to access devices basic operating system principle.

So, once let us say, it gives a read call or a write call or let us say a message transmission or create a process or something. So, it can take up to several seconds. Let us say of the order of a second, and until that service completes, it cannot be preempted. So, the task has entered, the real-time task or the user task whatever you say has entered the kernel mode through real-time, sorry, a kernel service invocation, the operating systems service invocation, and until the service completes, it cannot be preempted and that can take up to a second does that appear.

If it gets preempted in the middle,

What is the problem is it?

So, the question that he is now asking is that why I mean we need to make it non preemptible that once it starts a operating system service, why it cannot be preempted? Simple answer again. See, we will discuss it more detail later on, but the simple answer is that the kernel data structures, they will become inconsistent. So, the traditional operating systems the way they operate is that until one service completes, other services cannot start and the way to do that is to mask interrupts.

So, let me just, since we just discussed this, let me just ask you one question that this user mode and kernel mode application is executing in user mode and it makes a service invocation enters kernel mode, but how does this occur? I mean how can just by calling a procedure function; it is basically a service is a function or a procedure.

So, how can it enter kernel mode? Anybody would like to answer this question? Is the question clear? The application is executing in the user mode. Now, just by calling, say

calling means it is basically a code is getting linked, isn't it? Some code is getting linked to the, and it starts executing. So, how is the more getting changed from user

flag eset

Flag eset? Is the answer is flag eset? Then it has no meaning.

So, every user process will set the flag and start using in the kernel mode. Kernel mode will have no meaning.

If we try to module the (( )) until it is completely executed.

I mean how do you do that? See, you know that the user mode it does not have certain privileges, it can do some operations like accessing the hardware devices doing certain kind of operations, where as in kernel mode, it can do all operations; it is a privilege mode. So, how can a user application enter kernel mode just by invoking a service, it is after all the user process?

By controlling transfer to the operating system (( )) then call it.

No. What do you mean control is transferred? It is the program is executing. It just made some code; the service code got linked to it. That is all. It started executing that service routines.

There is a switch sort of thing status of the…

I mean how that switch will occur?

Sir (( )) then for the (( )) decoded for the in the (( )) with the.

No. I mean, you mean that in the code, in the, in the service code or the system library, there is some code which will be able to save the p s w and it will change the mode, etcetera. Then, every user program will run that code. If there is a code which can do it, then every user program will do that. They will change their mode. All malicious programs then will take over the system. They will execute those codes and then they will become the user the kernel or super user.

It is implemented for (( )).

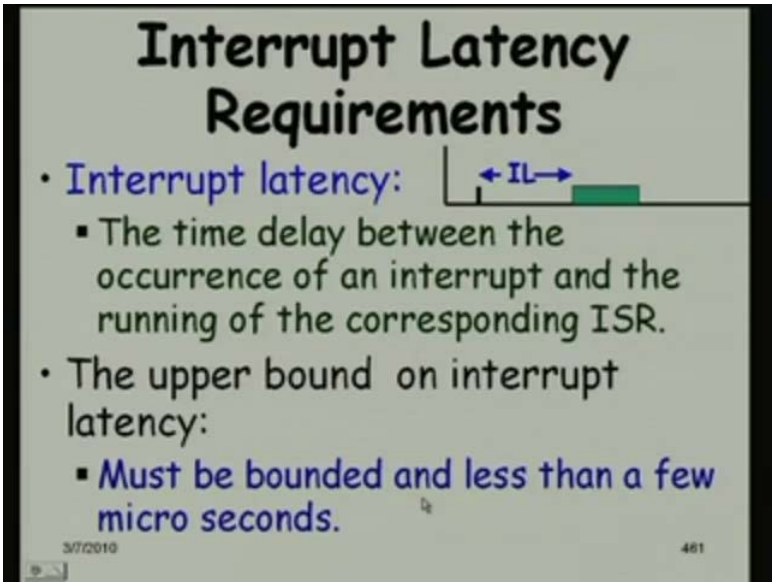
How can? See, it is, the user is executing; the program is executing; the code is executing. How can hardware where, where does the hardware come? It is basically instruction after instruction getting executed.

(( )).

How is that? I mean very fundamental issue actually, I mean it is done in the traditional operating system course called a software interrupts Ccertain instruction which when executed; trap is different. So, may be in next class, I will just come prepared with this. What is the difference between an exception trap and a software interrupt? We will discuss that in the next class.

So, let us proceed that, assuming that the traditional operating systems, they disable interrupts when it is in the kernel mode, and therefore, the task preemption time is of the, the worst case the order of a second which is unacceptable for any real-time application.

(Refer Slide Time: 17:29)



Now, let us look at the latency requirements, the interrupt latency requirements. This is basically the time delay between the occurrence of an interrupt in the running of the corresponding interrupt service routine. So, if this is the interrupt that occurred by a

device or by a sensor, then by the time, the interrupt service routine runs some time elapses. Time, this time is composed of many factors for example.

Instruction is (( )).

instruction

(( ))

So, one, he says that see that time it takes for the interrupt signal to travel up to the c p u small time, but that is there, and then, he says that the current instruction must complete, and any other thing that you can think of?

(( ))

Yeah.See, he says that if it is a non reentrant ISR, one ISR running, you cannot really start another ISR. It has to wait for it, yes,

but if the higher priority interrupts comes in that ISR will not be (( )).

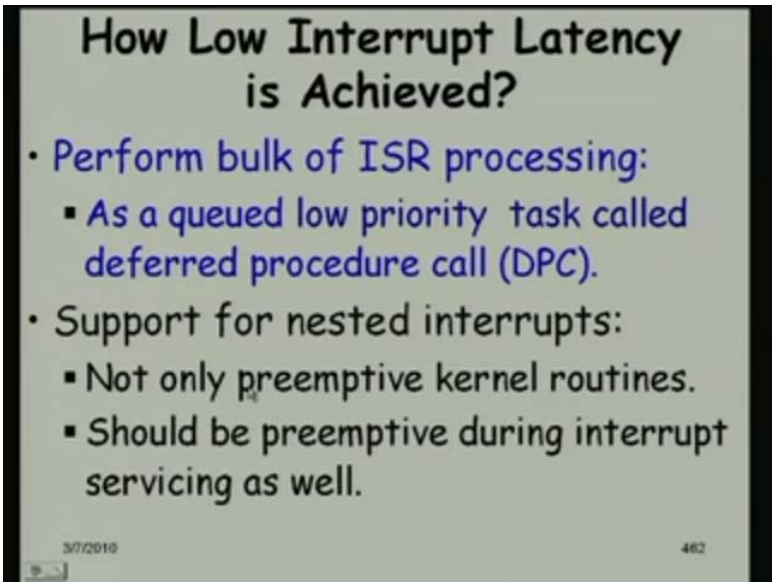
It depends on what kind of system that you are talking of. If it is non reentrant until the ISR completes, which is typically in every operating system traditional, they mask all the interrupts until the ISR completes.

Masking is to be done by the programmer

No, no, no, the interrupt service routine. As the interrupt occurs, there is a standard protocol, what happens? How the interrupt is handled, which is their in every first level operating system course. How, once a interrupt occurs, how does the operating system respond to it? And one of that is that until the ISR is complete, the interrupts are masked, and we will need a reentrant ISR handling. Let us, let us proceed.

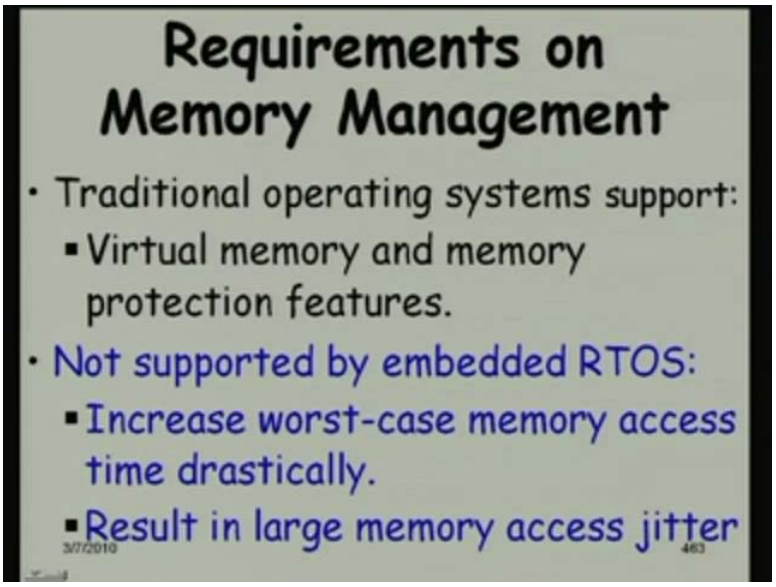
So, the upper bound on the interrupt latency needs to be bounded within few microseconds for a real-time application. For a traditional operating system where the interrupt is raised by a printer or something, we do not really bother if it gets delayed by second, the users will hardly notice it. But here, you are talking of automating systems and robo’s and so on, even milliseconds matter.

(Refer Slide Time: 20:24)



But how do you really lower the interrupt latency? One very basic thing that is done is to perform the bulk of the processing in a low priority task called as a deferred procedure call or DPC. So, an interrupt service routine should execute few crucial instructions and the rest of the ISR instruction should be queued in a low priority task called as a deferred procedure call. As we discuss further and further. We will see more about this how the deferred procedure calls are queued and also need to support nested interrupt. We are just talking about that, but when the ISR is running, it should not mask the interrupt until the ISR completes. So, not only need preemptive kernel routines, but also system should be preemptive during interrupt servicing as well.

(Refer Slide Time: 21:35)



Now, let us look at the requirements on memory management for a real-time operating system. We know that the traditional operating system support virtual memory and memory protection, and of course, we will see that this virtual memory and memory protection for very small applications where we do not even have a hard disk, this do not to make any sense.

So, virtual memory, memory protection, etcetera, not supported by many embedded real-time operating system. Not only that, they do not have a hard disk but also this increase the worst case memory access time drastically. Why is that? Can anybody say that why virtual memory support will increase the worst-case memory access time?

It is the (( )).

Yeah. If there is a page fault occurs, then it takes significant time to really get the page and to get the data that is required. So, if the data is available in the memory, the access time is very fast. If it is available in the cash fastest, if it is in the main memory, it is, but if it is there in the hard disk, takes significant amount of time.

Sir, but memory protection features (( )) there (( )).

So, the question there is that what about memory protection system? Actually if you analyze the traditional operating system, we will see that the virtual memory and the

memory protection features are somewhat linked. One goes with the other, virtual memory helps memory protection, how is that?

(( )).

Ok.

(( )) there is a flag protection. So, (( ))

Yeah. So, you just read up the first level operating system book. See, how the virtual memory system works and you will see that, you will come across this protection also is tied to the virtual memory. So, if you do not have virtual memory, then how do you protect one task from accessing the data or instruction of another task? How do you do that?

(( ))

See, paging know, you have access this protection information naturally with pages. If you do not have paging etcetera, then how do you do it? very…

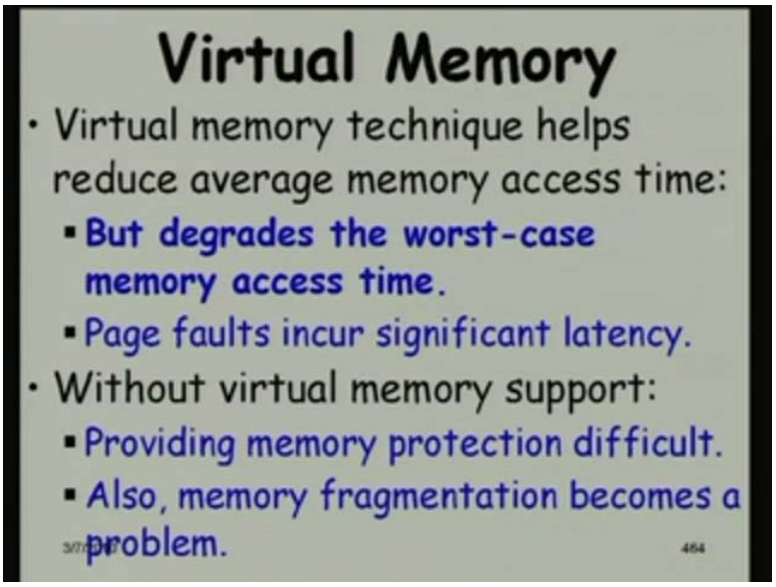
(( )) multiple task (( ))

You will have multiple task, yes, sure. So, how do you protect one task from the other task in the absence of paging, without paging? Just think of it.

And moreover the operating system itself needs to (( )).

Yeah, that is the next thing that how in the operating system be protected without virtual memory. So, this thing just revise your first level operating system. We will see that the virtual memory implicitly provides protection among task. And if some of the memory access is completed in micro or milliseconds and some take up to second, there will be a large jitter. This is the term we will use. The jitter is the difference between the longest access minus the smallest access is the jitter. So, if all the data that is required is resident in memory, the jitter will be low. If some of the data needs to be fetched from the hard disk, then the, we will have a high jitter.

(Refer Slide Time: 25:29)

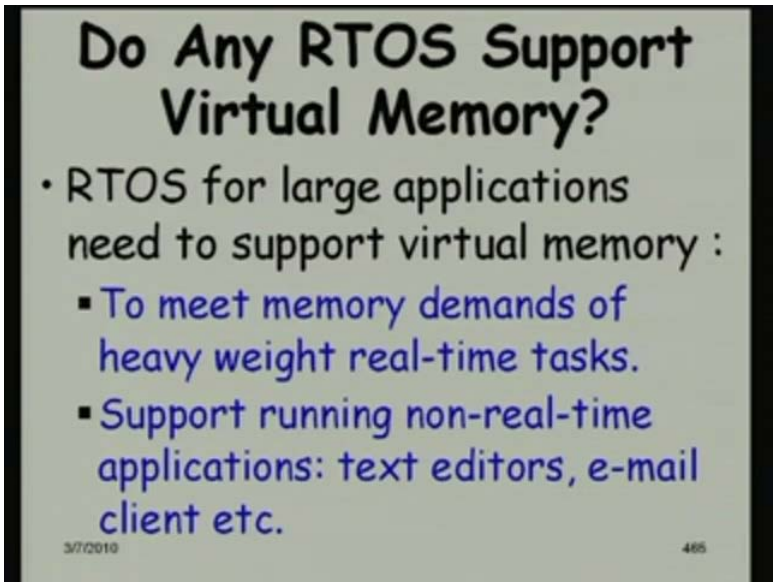


The virtual memory technique as we know is used in all operating systems, because it reduces the average memory access time, but degrades the worst-case memory access time.

If we did not have a virtual memory, then we will have the average memory access time higher, but the worst-case memory access time will be lower. But with virtual memory, we have to deal with this that we have improve the average memory access time, but we have degraded the worst-case memory access time, because page faults incur significant latency, and not only that, if we do not have virtual memory support, providing memory protection becomes difficult and also other issues we have like how do you handle with fragmentation of the main memory.

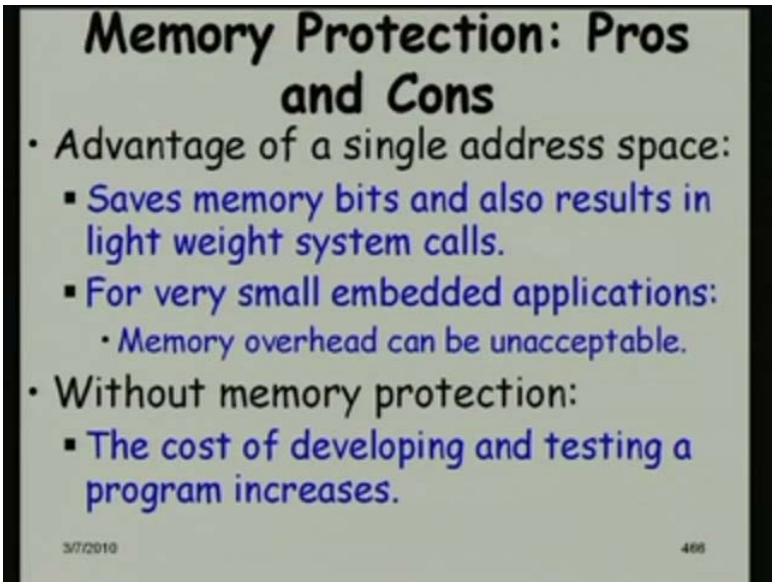
Here we have a nice paging scheme and the pages nicely come and fit in the memory. One page goes next page takes the space, but if you do not have a paging, then main memory fragmentation is a problem.

(Refer Slide Time: 26:51)



But we said that embedded real-time operating systems do not support virtual memory because they might not have a hard disk or there may be other issues, but do any real-time operating systems support virtual memory. Larger applications, larger real-time applications need to support virtual memory because of the memory demand of the heavy weight real-time tasks and also for supporting non real-time applications like text editors, email clients, etcetera. We will see that even in the embedded area, many of the operating systems - real-time operating systems - they do support virtual memory. As we will proceed from here towards specific real-time operating systems, commercial and open source, we will see that.

(Refer Slide Time: 27:48)

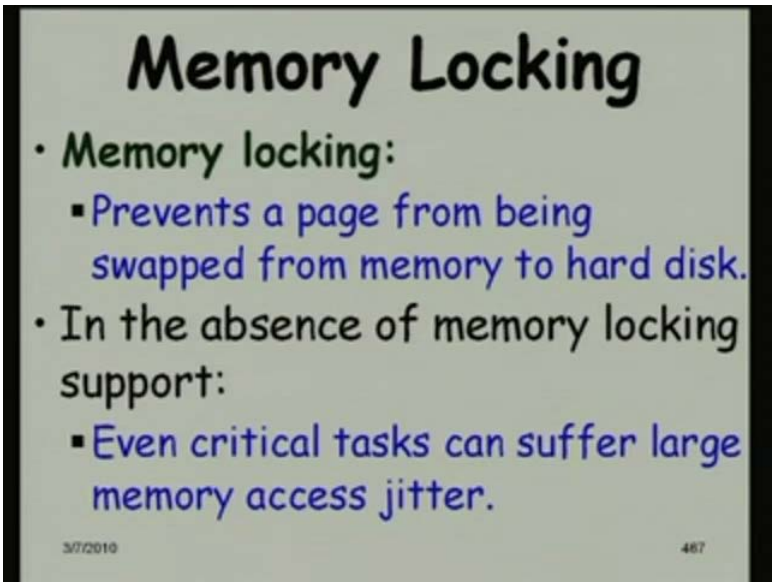


Now, let us see the advantages and disadvantages of memory protection. One is that if we do not have memory protection, we will have a single address space and memory bits can be saved and system calls can be lightweight, because do not have to check the protection aspect, and for very small applications because just one task is running and the task has been thoroughly debugged and run. So, you need efficiency.

The task is small, sorry, the application is small and the tasks that are there. It is basically belong to one application after all and we can, the programmer can ensure that they do not interfere with each other's, they do not interfere with the hard disk, sorry, with the operating system. Unlike in a general operating system where various categories of users run their application, here it just one application that is running and can ensure that it does not interfere with operating system and do not interfere with each other.

And in that case, we save the memory bits and also lightweight system calls makes it faster, but without memory protection, the cost of developing and testing a program increases. Even for a small application, it becomes very difficult to debug it. Once you find the error, do not know, where, which process which task has corrupted which data and also the maintenance cost increases. If you later try to add new features or modify some of the features debug etcetera becomes very difficult.

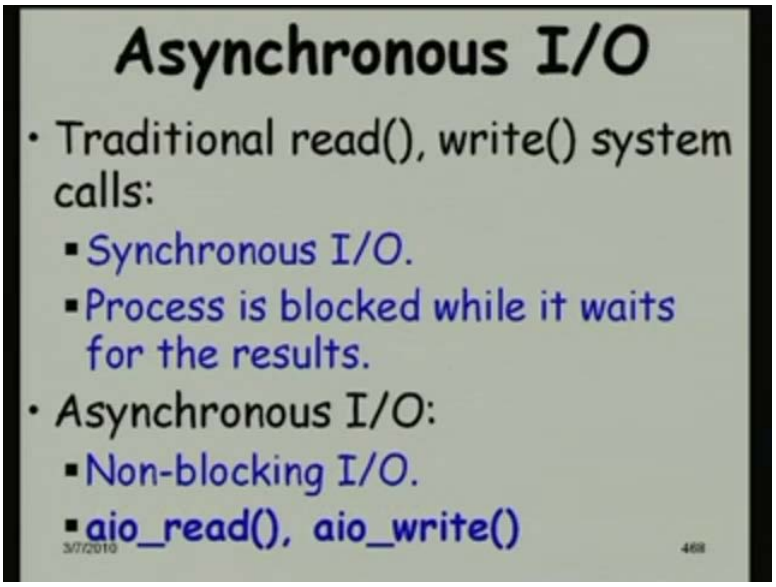
(Refer Slide Time: 29:50)



So, most of the real-time operating systems as we will see they do provide virtual memory, but to help with the jitter, access time jitter, they support memory locking. What is memory locking? It prevents a page that is specified by the programmer. The programmer can specify which pages should be prevented from being swapped to the disk - hard disk - even when they are not being used.

The traditional operating systems keep track of which page is used how frequently and based on that, they swap a page, but here once you lock a page, even if it is not used for some time, it will never be swapped to the hard disk. So, we will have some system calls, using which, the programmer can lock specific pages in the memory or can lock the entire application pages in the memory, and of course, we know that without the locking support, even the critical tasks can suffer large memory access jitters.

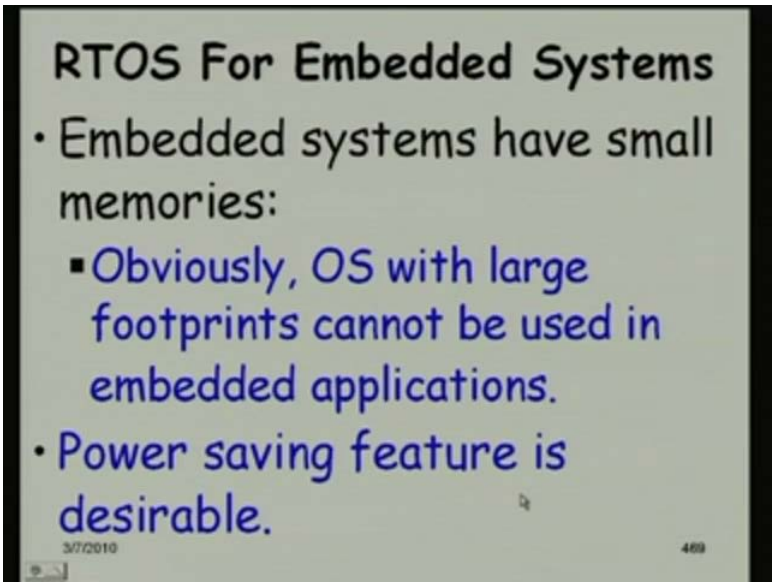
(Refer Slide Time: 31:04)



Another support that is needed is asynchronous I O. The traditional operating system calls like read, write etcetera are synchronous, that is, the process is blocked while it waits for the result. Just try writing a large block of data to hard disk, try outputting it to the terminal. You will see that the process does not terminate or the process just waits for the data to be written on the hard disk or on the terminal.

An asynchronous I O is a non blocking I O, for example, calls like a I O read, aio write, etcetera, the task might be performing some logging activities and so on, but it should not block while the activity goes on. So, you need asynchronous I O support.

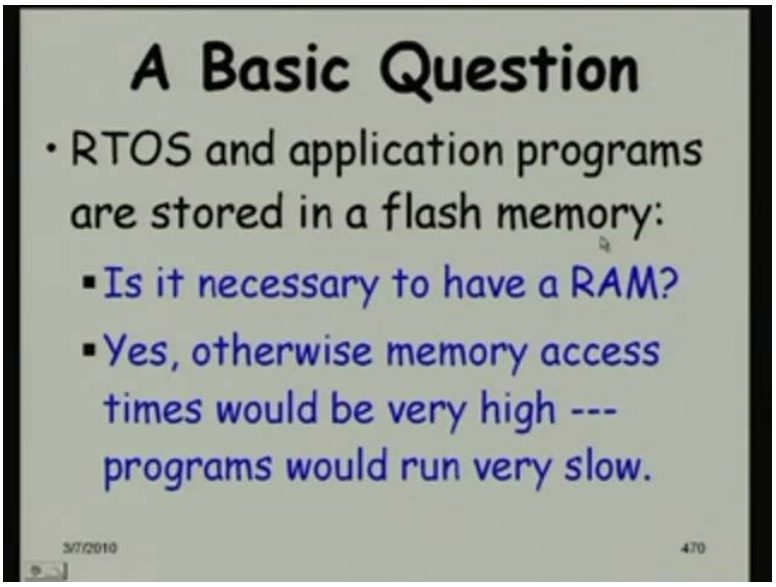
(Refer Slide Time: 32:01)



We will categorize the operating systems, real-time operating systems. We will see that for the smaller applications, those are the embedded systems. We have very small memories. And obviously, if the operating system is large, we cannot really accommodate that in the embedded applications.

So, we need operating systems with small footprints for those who occupy less space, and not only that, the operating system should help with power saving because many of this embedded applications run on battery and need to save power by running in a lower power mode, switching off the power whenever it is not required, etcetera.

(Refer Slide Time: 33:00)



But let us answer a basic question that in many of these embedded applications, the programs are actually stored in a flash memory, because hard disk is not there. Hard disk is large power consumption, large weight and costly also. So, they are not there in many of the embedded applications space, weight, etcetera consideration and use flash memory.

And flash memory we were saying that it is kind of a EEPROM, isn't it? Now, when we have a flash memory which is basically EEPROM, do we need to have a ram in that application, I mean in that embedded system? See, we have this flash memory. Now, do we need a ram in that or it can directly run on the flash memory? What do you think?

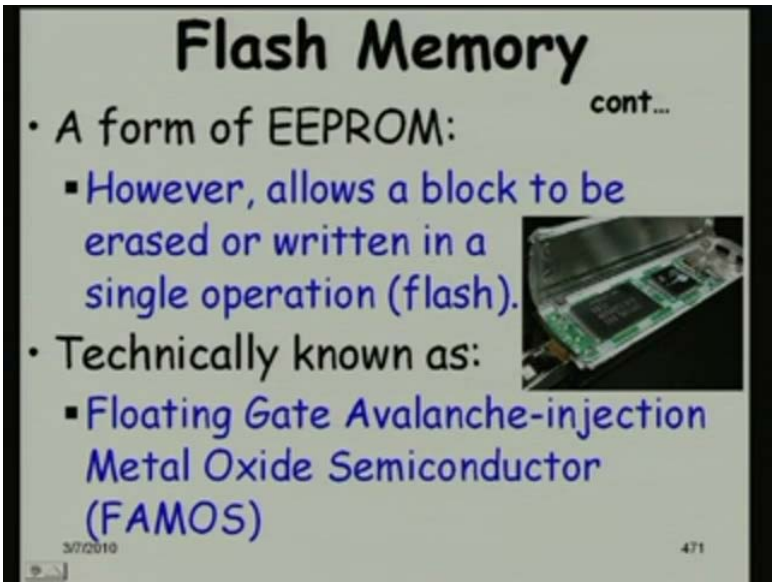
(( )) perfect on that (( )).

Yeah. So, one thing is that a flash memory is not like a ram, the write time is very high here. It is a permanent storage, no doubt, it is a semi conductor. So, those are the similarities, but the write time specially to, if you try to write to a flash it takes significant amount of time. The read time even is slower than ram, but still it is comparable to a ram access time. But the write time is extremely high and also the number of rights to a flash memory is restricted, you cannot just use it is a like a ram where you keep on writing millions or billions of time. Here, the number of writes being performed is restricted.

So, you cannot really have a flash memory replacing the ram not at least in the foreseeable future. Now of course, you might have heard about laptops coming up with without any hard disk just flash memory. Have you seen those? Very light weight laptops less than one k g right eight hundred grams or so, laptop, and they do not have hard disk, because hard disk if you have, it will be an excess of weight one and half kilos or twelve hundred grams and also the flash memory consume very little power. The hard disk is a power guzzler; the battery power it will exhaust. So, the flash memory has the advantage of making it lightweight and also low power, reliable, etcetera.

But still, see, earlier few years back, the number of rights to a flash memory was very low. Now, it is significantly improved, but still it is not infinite. There are limits on how many times you can perform writes on a flash memory. We will see possibly the reason why that is. So, number of writes is restricted. So, let us continue with our discussion. So, yes, we need a ram. Otherwise, the write times will be unexpectedly high. Read times may be comparable may be slightly slower, but write times will be unacceptable. The program would run very slowly.

(Refer Slide Time: 36:46)



I mean at least in the foreseeable future, see the access times on flash memory has been increasing improving, but in the foreseeable future, the access time, especially the write times are too high to be used in a embedded application replacing the ram. As we are saying that it is basically a EEPROM, because the flash memory is very common in

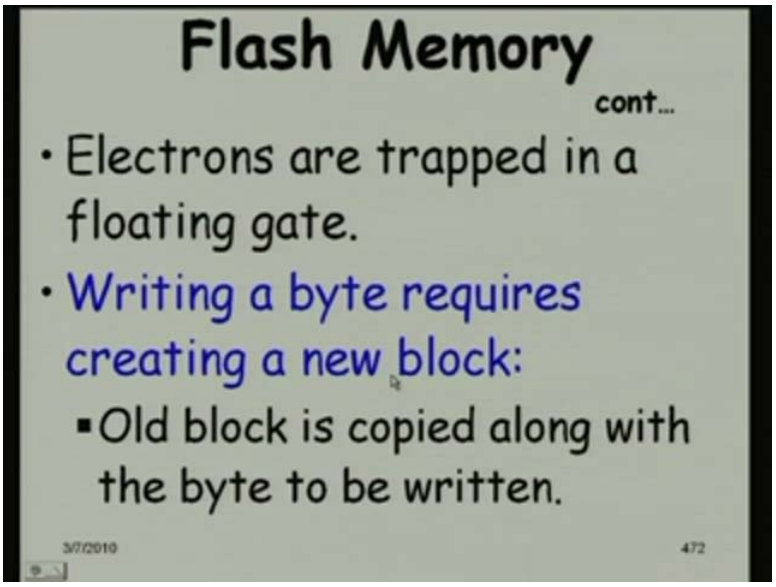
many of these real-time applications. We will just spend few minutes couple of minutes discussing about flash memory so that our later discussions fit into the ideas that are presented here.

So, as we are mentioning that, it is a semiconductor memory in the form of EEPROM, but in contrast to a EEPROM, it allows a block to be written or erased in a single operation called as a flash. Think the inventor Japanese, in Japan, it was invented, The inventor he thought that it operated like a camera flash just one operation, everything is deleted, large block is deleted it. Appeared to him like a flash everything is deleted.

See, here, if you open your pen drive here, you will see that there is a microcontroller sitting here and this is actually your memory, the flash memory, and the microcontroller actually writes in a flash and technically the kind of memory that is used here even though we called it as a EEPROM, it is known as the floating gate avalanche injection, avalanche injection metal oxide semiconductor; it is a mos device.

See, we are not going to concentrate on the hardware that we, that was the basic premise, but this is just for a general knowledge. It is not that you know even you will find these terms in the newspaper or somebody who does not know much about computer, he will come and tell you see I have this flash memory and its write time is much higher than the read time, etcetera, that even a non computer person will tell you. So, very basic ideas we will need here.

(Refer Slide Time: 39:15)



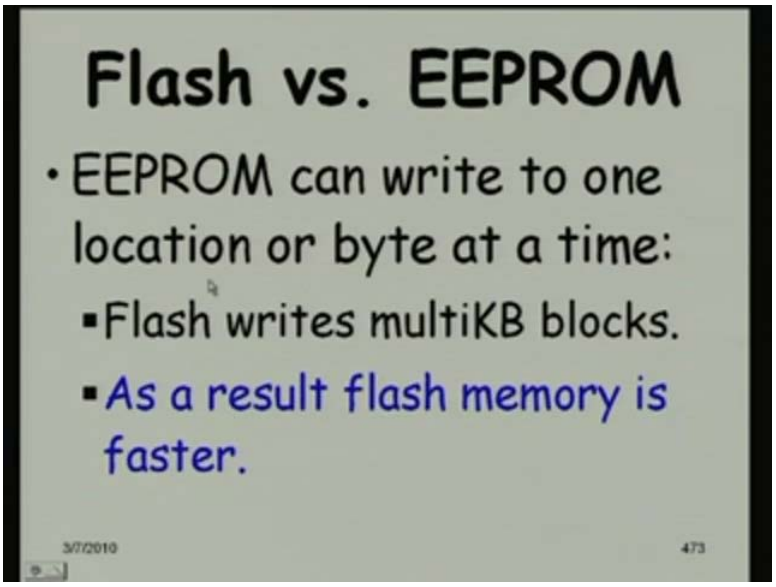
So, the idea here is that the way the data is stored is by trapping the electrons in a floating gate and each block here is several kilobytes, may be hundreds or thousands of kilobytes, not thousand, at least several hundreds of kilobyte is one block, may be five hundred twelve kilobyte is just a block.

Now, let us say, you want write one byte onto that. You cannot really write one byte, it writes the entire block, and to optimize it, you might have seen that the computers when you attach a flash, a pen drive, they do not really write as you keep on writing to the flash memory.

They keep it on a buffer, and finally, you just, when you, yeah, when you say that I want to disconnect the flash drive, they write the data that time or unless the block size is large enough, they write it on the flash memory, or unless you try to un-mount the flash memory, they do not write it; they keep it pending in the buffer, just that you do not, your, know write byte by byte onto it.

So, because the, the, otherwise, you will be ending up writing multi kilobyte blocks there and then number of writes is also limited, it takes power, etcetera. So, they do not do that. So, if you ever try to write a byte, let us say you have written a byte and you are un-mounting the pen drive. What will happen is that the old block will be copied along with the byte to be written.

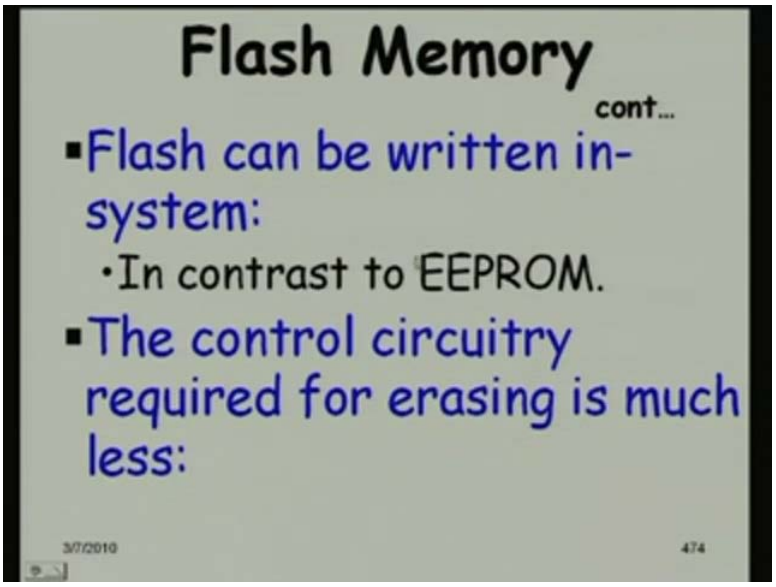
(Refer Slide Time: 41:09)



So, just one byte and rest old data is again rewritten. The EEPROMs they existed for longtime, thirty, forty, fifty years EEPROMs existed, but how come the flash memory is in last ten years, may be twelve years, they have become so popular, because the flash memory is invented in Japan may be about twenty five years back, not too long back, just twenty five years, and then, by the time it got commercialized, just ten fifteen years that you are seeing this memory. So, it writes multi kilobyte blocks. Therefore, the control required is much simpler, do not really write a single byte. So, the control required is much simpler here and it is faster.

(( ))

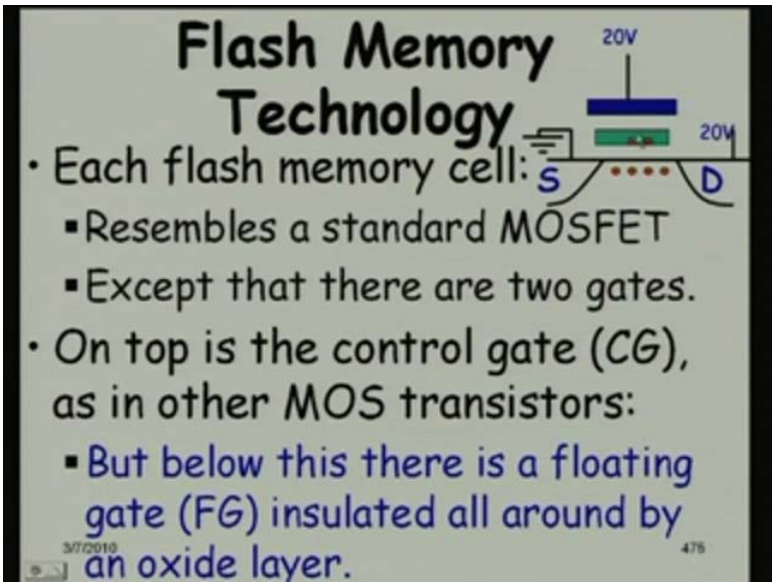
(Refer Slide Time: 42:10)



Yeah, also the memory override is much less; the control circuit overhead is much less. And the other very big advantage with a flash memory compared to EEPROM that it is written in system.

See, earlier in EEPROM, you need a special EEPROM programmer. It could be operated with large voltage, twenty four volts or at least 12 volts EEPROMs and you need a EEPROM programmer, but the flash memory operates in a much less voltage. Now, the current flash memory operates in 2.7 volts. So, that is a big advantage of flash memory. You can program it, write it in the computer itself. In contrast to EEPROM where you need a special EEPROM programmer, the control circuit for writing and reading is much less leads to higher capacity.

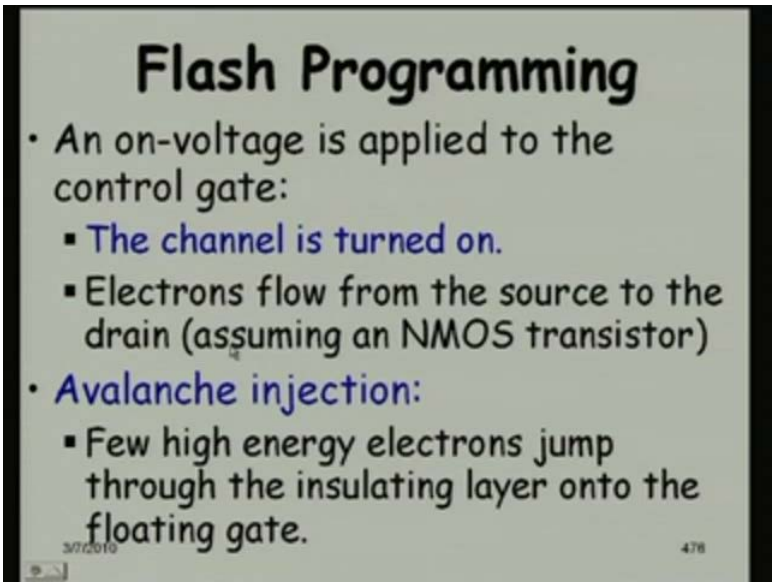
(Refer Slide Time: 43:15)



Each flash memory cell actually resembles a standard MOSFET, a standard MOSFET resembles excepting that there are two gates. See, here, there are two gates here. A standard MOS has just one gate. The top gate is called as the control gate and the other is called as the floating gate. See, here, the floating gate.

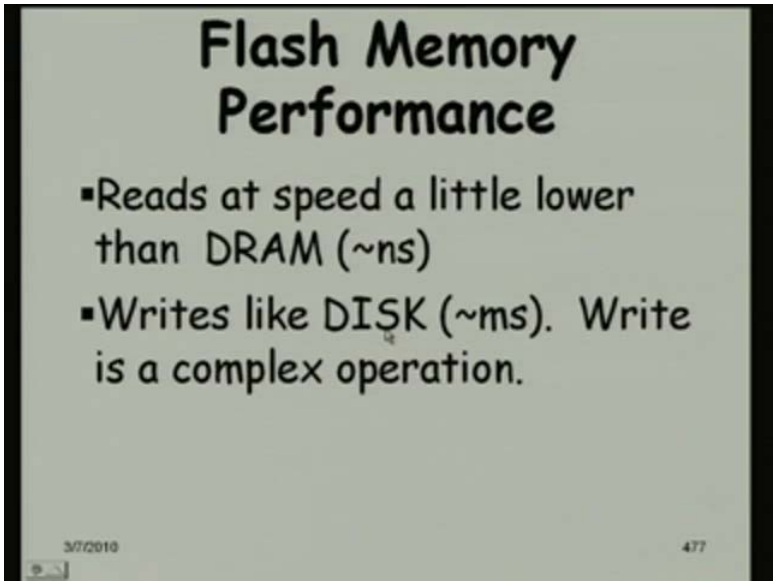
Now, as you enable the source to drain with a voltage some energetic electrons, they jump over to the floating gate because of the voltage here and they get trapped here. The floating gate here that is insulated actually. Once the electrons jump onto the floating gate, they get trapped here can stay here for years together; the charge stays on here for years together. So, the floating gate is insulated by all around by an oxide layer, and originally, as I was saying that it used to operate on very large voltage 20 volt and so on, 12 volt, but now, we have 2.7 volt operation.

(Refer Slide Time: 44:41)



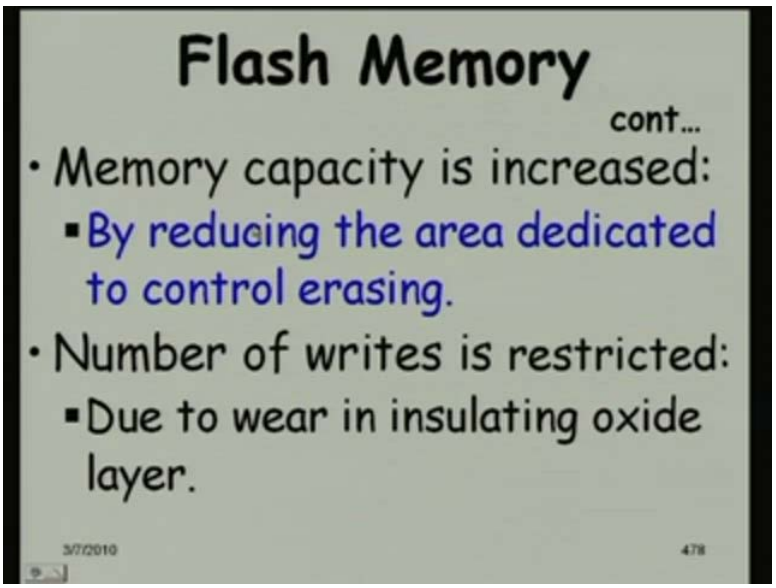
So, the way the data is written here is that when a on voltage is applied to the control gate, the channel is turned on and then electrons can flow from the source to the drain for NMOS transistor. Otherwise, it will be from drain to source PMOS transistor. And the technique is known as the avalanche injection where high energy electrons jump through the insulating layer onto the floating gate and they get trapped there. So, this is the primary mechanism, the avalanche injection and the floating gate.

(Refer Slide Time: 45:25)



These read at the speed of a little less than the d ram, the order of several nanoseconds; it is the performance of a flash memory, but it writes like a hard disk. If you write to a flash memory, take several milliseconds for byte; I mean it write s multi kilobyte, but still if you look at the access time per byte, its comparable to a disk, because write is a complex operation because of the avalanche injection.

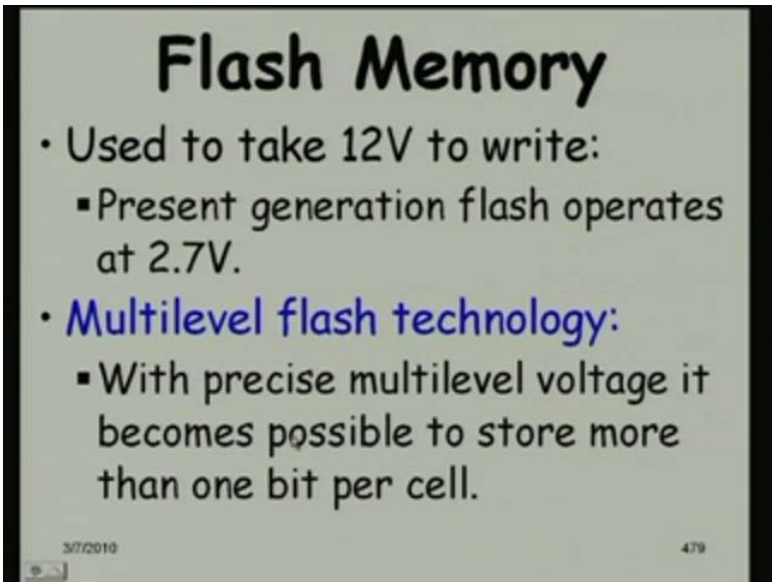
(Refer Slide Time: 46:03)



Originally when the flash appeared, the memory capacity was very small - few megabytes, even kilobyte flashes existed. But now, we have gigabytes, 16 gigabytes, 64 gigabyte pen drives, etcetera, are extremely common now a days. So, how could the memory capacity keep on increasing? One is by reducing the area dedicated to control erasing of course, but also by storing multiple bits on a single cell by controlling the charge that is stored on the floating gate.

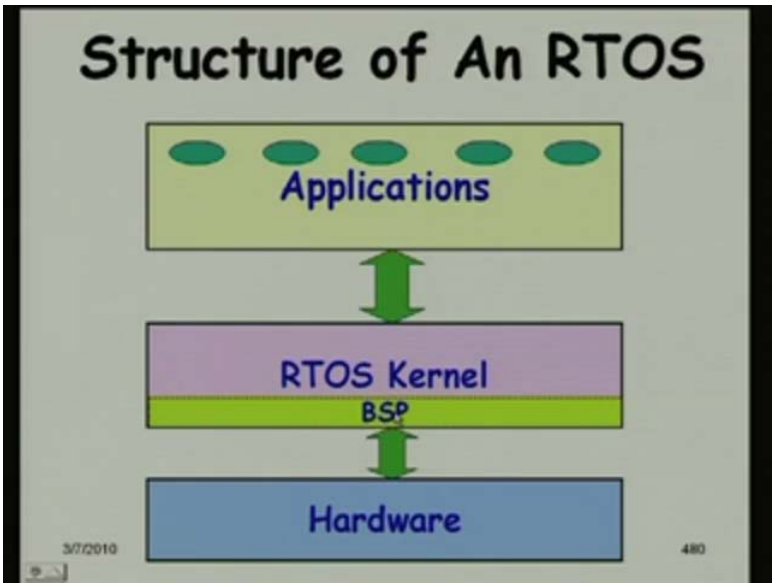
Now, all this flash memory even though the number of writes has been increasing, but still it is restricted, you cannot write infinite number of times due to the wear in the insulating oxide layer, because these operate under some voltage.

(Refer Slide Time: 47:05)



It used to take 12 volt, not very long back in ten twelve years back, but presently operates in 2.7 volt, and even few years back, it used to take 5 volt. Since you are using a u s b drive u s b flash, you do not really notice it, because the u s b delivers power to the device. It senses the power requirement and delivers needs 2.7 or 5 volt. And now, we have the multilevel flash technology. As I was saying that the storage area has really increased here with the precise multilevel voltage becomes to possible to store more than one bit per cell.

(Refer Slide Time: 47:55)



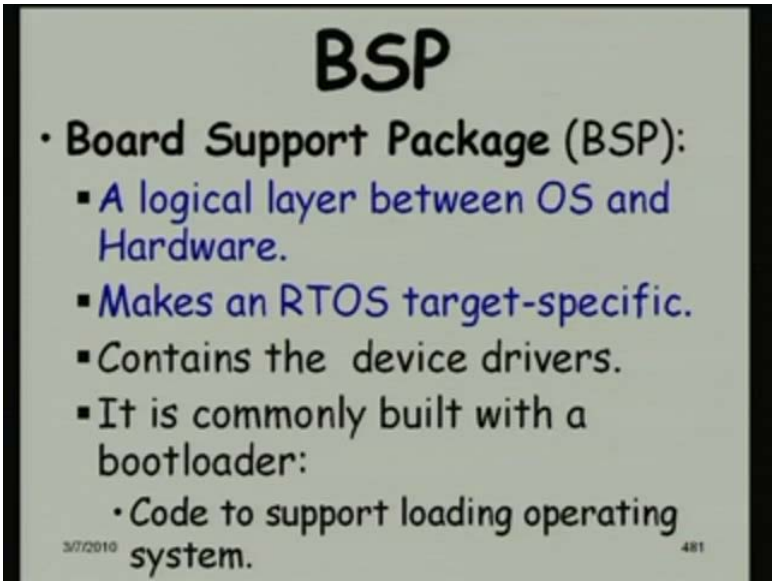
So, we will fall back on our discussion on a flash memory as we proceed, because all these embedded applications and most of these real-time applications they will have some flash component coming in. Now, let us look at very basic concept - the structure of a real-time operating system. So, we have this hardware, on this, we have this BSP or the board support package, and then, the real-time operating system kernel and then the applications run by invoking service station, the real-time operating system kernel which in turn invokes services of the BSP and that drives the hardware.

So, what about this BSP? Has anybody heard of about it board support package? Very common term actually.

(( )).

So, let us see. So, he says that this has the drivers. Driver is one part of it definitely device drivers, because that makes the operating system, logically operate on various types of hardware. So, it will obstruct the operating system and it you can easily port with different hardware configurations and so on.

(Refer Slide Time: 46:26)

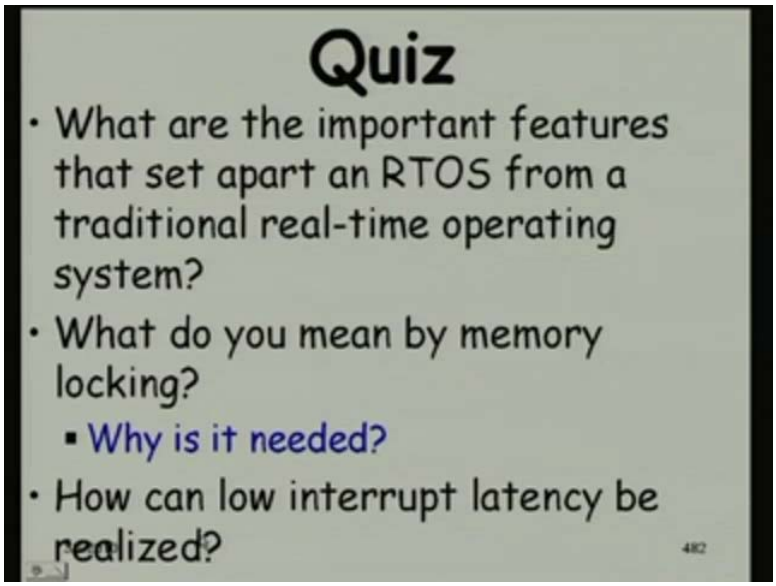


So, let us see before we proceed further, what is a BSP? It is the board support package. It is a logical layer between the operating system and the hardware and makes the real-time operating system. See the real-time operating system is a generic operating system and it can operate in a target specific manner by using specific BSPs. The BSP's

naturally contain the device drivers as he was mentioning, but also it is commonly built with a boot loader, that is the code to support loading the operating system onto the board.

So, each time, it boots the operating system code needs to be loaded. So, we will discuss the operating systems in this context that we have a more generic operating system code which runs on specific target boards by using BSP's which become available with the target boards.

(Refer Slide Time: 50:29)



Now, let us just for few a minutes evaluate our understanding of what we did today, is very basic question which anybody will ask you after having done this course that how is a real-time operating system different from a traditional non real-time? I am sorry, from a, should have been a non real-time, from a traditional non real-time operating system.

(( )).

So, very important thing he said that should support, a real-time operating system should support static priority level.

(( )).

Yeah. So, it should support, see, the traditional operating systems try to maximize task throughput. They use a different type of scheduling algorithm. But here, real-time scheduling algorithms which help task meet their deadline have to be supported in any real-time operating system that we can think of, yes. So, could say two important points, any other thing?

Resource sharing

Resource sharing, yes, we should have some way, operating system should provide some way for tasks, real-time task to share resources without incurring priority inversions or without incurring too much of priority inversion, and then, we have other things like memory locking asynchronous I O so on, but at least the three points I hope everybody remembers. Now, let us look at one or two more (( )) before complete today. What is memory locking and why is it needed?

See this memory locking we will not really discuss later in any greater detail. So, whatever we discuss, discuss only that much. So, what do you understand by memory locking and why do you need a memory locking?

When the task is brought in to the (( )).

Not task actually it is.

Figure means (( )).

Yeah, some pages, the page maybe code or maybe instruction; we do not know.

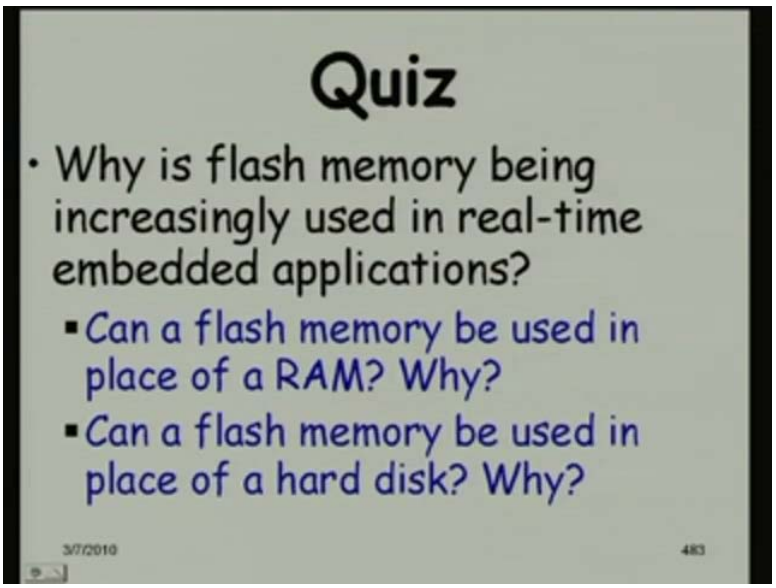
So, the programmer can lock certain pages in the memory so that the worst case memory access time for that data is bounded and the operating systems as we will see, the routines or the services that they provide. Even the posix real-time system, it mandates providing locking page level and also the entire task its data code, etcetera.

How can low interrupt latency be realized?

(( )).

Yeah. So, there are some of you are remembering it well. One is that need to use differed procedure calls to reduce memory latency and also reentrant ISR's even before one ISR completes. If a higher priority interrupt occurs should be able to handle the higher priority ISR. We will see that the real-time operating systems actually support priority. Based on priority, they can even interrupt ISR and from the higher priority ISR.

(Refer Slide Time: 54:12)



Now, what is the advantage of flash memory in a real-time embedded application?

(( ))

Yeah, exactly. So, it is lightweight, less value and more important is the power consumption, but can a flash memory replace RAM? No, what is the reason? Write takes significant time and also number of writes is restricted at present. So, we will stop here; we meet in the next class. Thank you.