

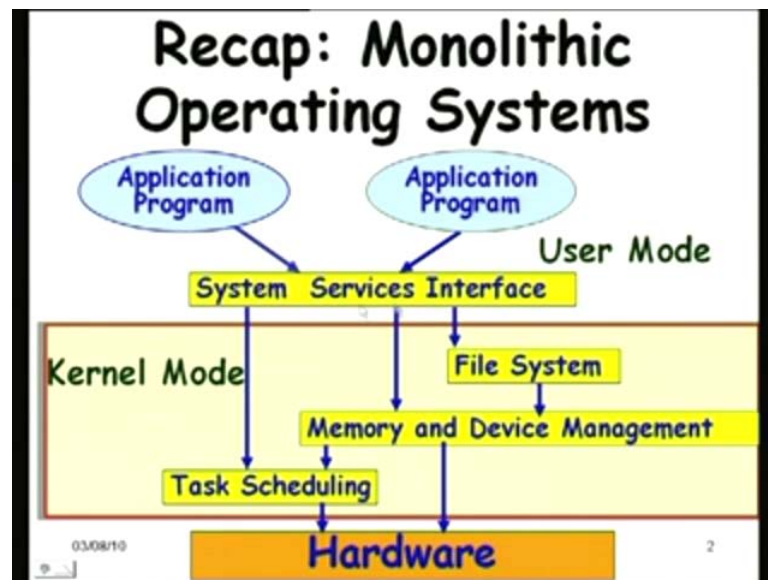
**Real – Time Systems**  
**Prof. Dr. Rajib Mall**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture No. # 24**  
**Unix and Windows as RTOS**

Good morning. So, we will get started today, last time we had discussed about some very basic issues in real time operating systems, what are the basic requirements and so on. So, today let us build on that; today we will try to examine the traditional operating systems, the popular ones being the Unix and the windows.

We will examine what are the issues if we try to use those traditional operating systems in real time applications. Actually, the idea is that, this will give us in insight into **what is** how the real time systems will be actually designed, what is really necessary required of them. So let us proceed.

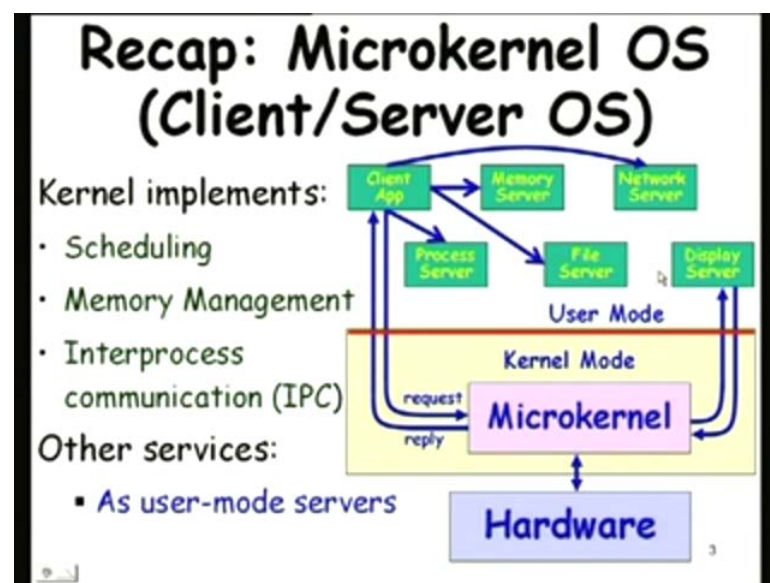
(Refer Slide Time: 01:09)



Let us just recollect what we are doing last time. We are saying that the Unix system, which is possibly has the most profound influence on the entire operating system is a monolithic operating system.

In the Unix, we had said that most of the operating system services here, they operate in the kernel mode. The application programs, they run in the user mode, and they can invoke the system service interfaces, which traps into the kernel mode and provide the required services. For example, there are four main categories of services that are available to the application program: dealing with tasks, creating tasks and so on, memory and device access, file access. So, all of these require trapping into the kernel mode, and then providing the required service, and then returning the result to the application program.

(Refer Slide Time: 02:35)



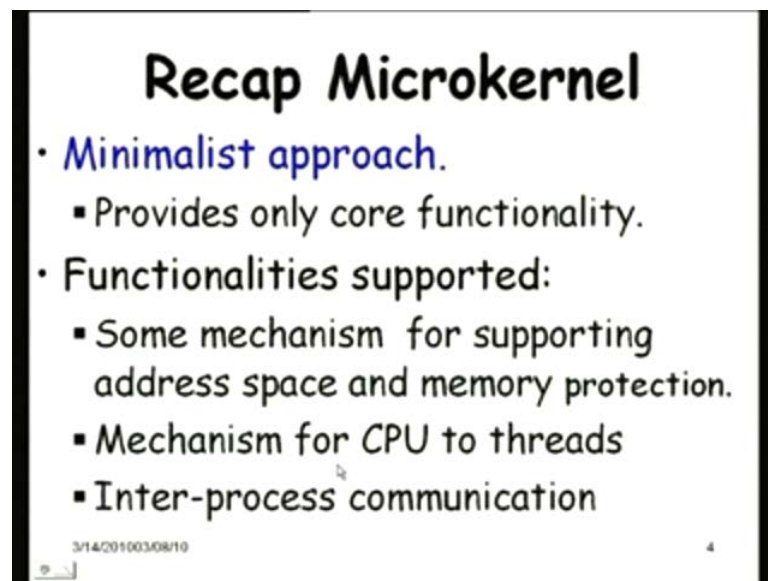
In contrast, we had seen the Microkernel operating system, which is also popularly used as a client server operating system, where the kernel implements the minimal functionalities; for example, task scheduling, memory management, some basic memory management and inter-process communication. These are supported in the kernel and rest all other services are supported as user mode services through servers.

So, if we try to look at a schematic of a Microkernel operating system, we see that the Microkernel only operates in the kernel mode and interacts with the hardware, and rest all other services, which used to be in the kernel are operating in the user mode. For example, memory; here some very basic memory functionalities, like the address space and memory protection are provided, rest everything here in the memory server, the process server, the file server, the display server.

Then the file server has to directly interact with the hardware.

No, the file server through the microkernel, see all this, we did not show the inter connection, see the client application is actually invoking the servers here which are shown in the user mode right. And these have to finally, call the microkernel to interact with the hardware, please do not directly interact with the hardware.

(Refer Slide Time: 04:25)

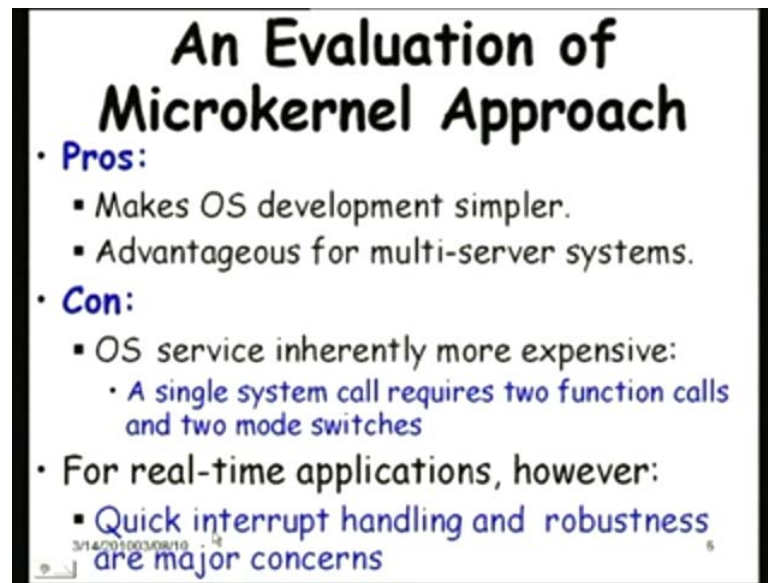


So, this we had said that, this is the microkernel is the minimalist approach. The kernel provides only the very basic or core functionality based on which operating system can be built, and the elaborate functionality based on the core functionality provided by the kernel run in the user space.

Typically, the functionality supported in the kernel mode is it varies from one implementation to other, but some mechanism for supporting address space and memory protection, because this is a basic requirement without this processes, the user processes and the system level processes, they will have, it will become difficult to ensure memory protection.

Some mechanism CPU for allocating CPU to threads, that is scheduling basically some mechanism, I am sorry I missed out here scheduling, mechanism for allocating or scheduling the CPU, allocating the CPU to threads, and then the basic Inter-process communication mechanism.

(Refer Slide Time: 05:44)



**An Evaluation of Microkernel Approach**

- **Pros:**
  - Makes OS development simpler.
  - Advantageous for multi-server systems.
- **Con:**
  - OS service inherently more expensive:
    - A single system call requires two function calls and two mode switches
- **For real-time applications, however:**
  - Quick interrupt handling and robustness are major concerns

3/14/2015 00:34:08:10

But we had said that the Microkernel Approach has become popular, but how does it evaluate, I mean how does it compare with respect to a monolithic kernel approach. Let us look at the advantages first. One important advantage is that the microkernel approach makes operating system development simpler, because those who would have little bit tried out in kernel level debugging and programming, they would know that it is extremely hard compared to a writing a user program. The main difficulties are that they do not have debug or which traps and stops at certain place and reports you and so on.

Because this itself is running in the kernel mode, and also if there is some data getting corrupted, it becomes very difficult to identify, because it can access any data is not it, there is no protection here in the kernel mode.

So, the micro kernel approach definitely will make the design and development much simpler, and it is especially advantageous in a multi-server system as you are saying that inclined server applications, where the different servers run on different systems it becomes advantageous, but there are disadvantages as well in the microkernel approach. For example, every operating system service takes more time to execute its expensive, why is that, because just making a single system call would require two function calls and two mode switches, why two function calls.

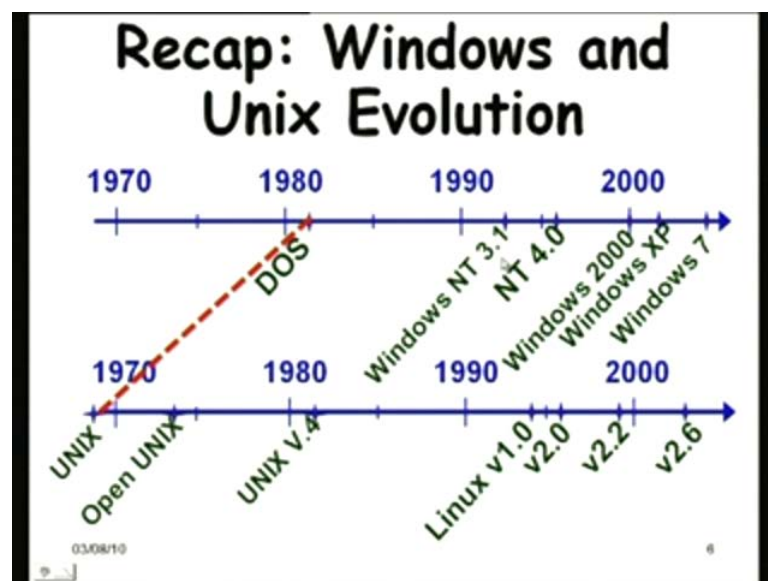
(( ))

Yeah, exactly, there are two function calls necessary, because one to the server which operates in the user space, and that needs to again invoke the kernel and similarly 2 mode switches are required.

So, it appears like a problem here, actually the initial microkernel operating systems like MAC and I think the spring kernel, they gave very negative results, as far as its performance is concerned, performance was much worse than a monolithic kernel, but the later generation microkernel's, the performance is not all that bad as this source and especially for real time applications.

The issue of importance is a quick interrupt handling and robustness, if you have bugs in the kernel can debug then it is a big problem, because most of these are used in embedded and safety critical applications, and one of the primary requirement is robustness and microkernel scores there and also, because the kernel is very small, the interrupt handling becomes quicker.

(Refer Slide Time: 09:14)



Now, we had in some context I think discussed about the influence of Unix, and how Unix developed and also alongside the windows developed.

So, if you look at it, I think 1968 or 69 the Unix system was developed at A T and T bell labs told about that. And then in 1975 A T and T bell labs made Unix open, distributed

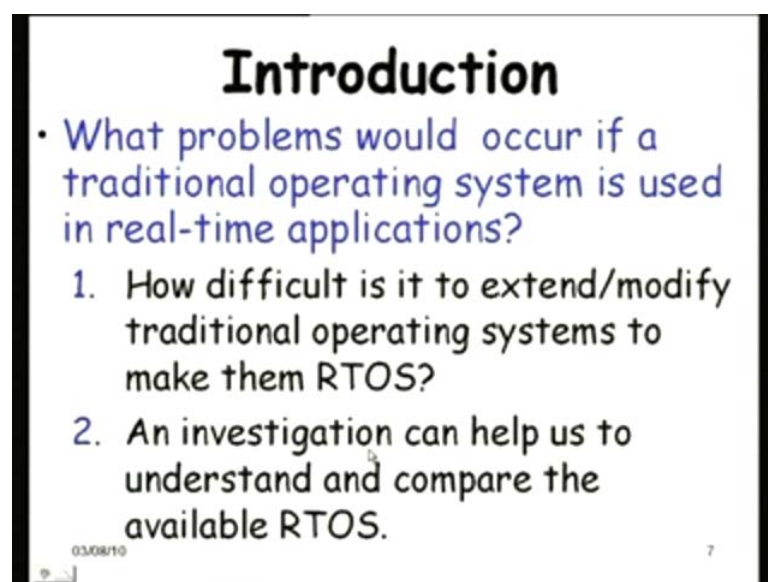
the source code to whoever needed it, and then we had seen that Unix, started developing from there onwards in different versions.

The University of California Berkeley developed the BSD and then the IBM with AIXHP with HPUNIX and so on, but AT and T also internally developed Unix further, and Unix system 5.4 is released in 82 in between there are many other Unix versions, and this has become one of the popular in standard operating systems, Unix with the system 5.4, and then in the mid 90s. So, here the PC evolution started PC's become extremely popular Unix did not run on that.

And, the linux was the attempt to make Unix run on the PC's. Unix is more of a server operating system, and then the linux started evolving in the currently in version 2.6, but see here that the DOS started somewhere around 1981, 1982 I think it was released.

And after that there are several versions of DOS, initially you work on floppies and then the disk and then the windows NT 3.1, which was I think another landmark here, the windows appeared, the windows 1.0 appeared something like 1986 I think, and then the windows NT was landmark here we will see why, and then the NT 4.0, and then windows 2000, windows XP, vista and windows 7.

(Refer Slide Time: 11:54)



**Introduction**

- What problems would occur if a traditional operating system is used in real-time applications?
  1. How difficult is it to extend/modify traditional operating systems to make them RTOS?
  2. An investigation can help us to understand and compare the available RTOS.

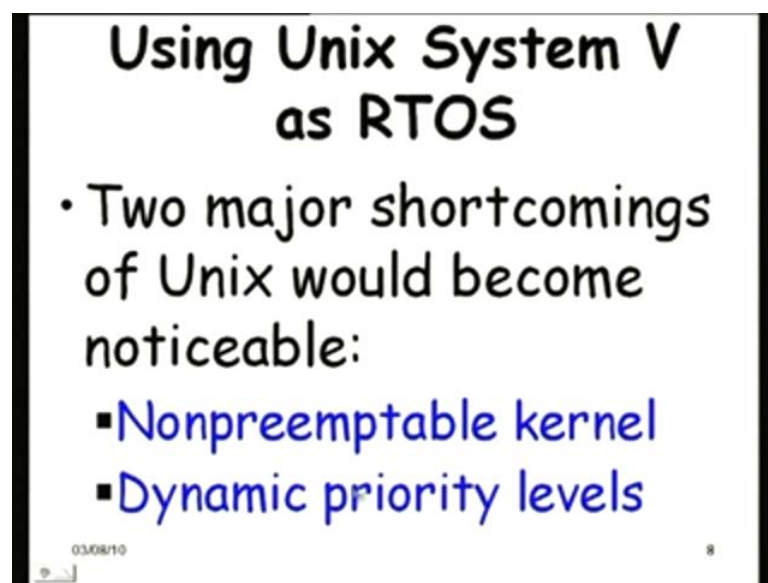
03/08/10 7

So, let us with that knowledge let us try to see what problems would occur, if the traditional operating systems are used in real time applications. If we take just a Unix or

a Linux or may be the windows, and then try to develop some real time applications what would happen and why. We would examine also, how difficult it is to extend or modify these traditional operating systems to make them real time operating systems.

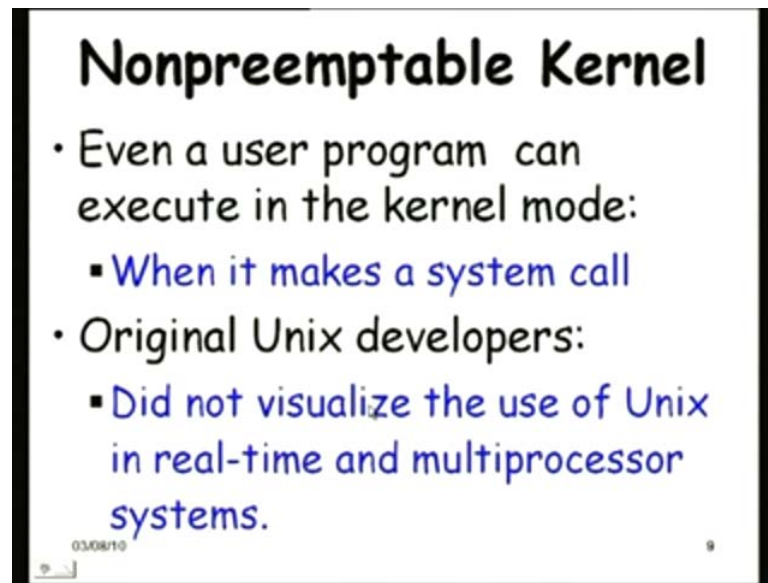
This investigation that how to make the traditional operating systems into a real time operating system will give us the clues to understand the working of the current real time operating systems, and also we can compare the features of the available real time operating systems that will give us a better insight.

(Refer Slide Time: 12:50)



Let us first look at the Unix system V, if we want to use as a real time operating systems what would be the problem. There are two main problems which we will notice. Anybody who wants to use Unix as a real-time operating system, one is the nonpreemptable kernel, and the other big problem we will find the Dynamic priority levels, besides this there are many other problems, but these 2 are the major problems that would be noticed.

(Refer Slide Time: 13:24)



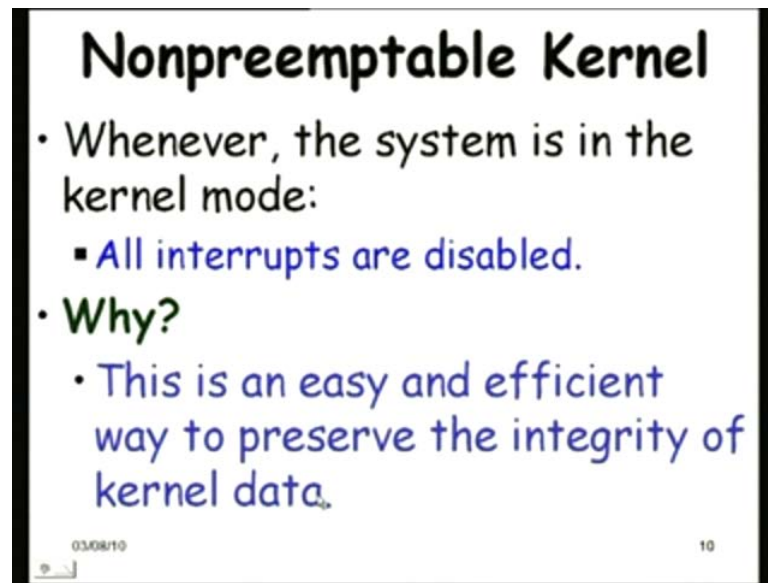
Now, let us look at the Nonpreemptible kernel. Let us investigate why this was necessary in the Unix, and can it be overcome in what way.

So, we had said that even a user program can execute in the kernel mode, how is that, how can user program run in the kernel mode.

Making a system call.

By making a system call, the user program runs in the kernel mode by using as software interrupt. The original Unix developers did not visualize that Unix would someday be used in real time multiprocessor systems and so on. They were addressing the problem at that time, where they are more concerned about developing a time shared operating system that was the in thing those days, a time shared operating system and later virtual memory, those were the focus those days, and they did not think that someday somebody will try to use Unix in real time or in multiprocessor systems.

(Refer Slide Time: 14:41)



The origin of the problem of a Nonpreemptable kernel is that whenever it operates in the kernel mode all interrupts are disabled, why is that, why should as soon as it starts executing in the kernel mode all interrupts are disabled, why is that anybody would like to answer this question. Why should they disable the interrupts as soon as they it enters the kernel mode.

(( ))

Another interrupt occurs that needs to be handled. So, what problem would occur?

There are some inconsistencies, when instruction is being executing and then interrupt has occurred (( )).

No, see as you were saying, see when the interrupt occurs in the first level course, the currently executing instruction must be completed that has to be, because you know after all instructions are atomic, you cannot just stop it halfway and then start.

But it is in a higher level program, a single statement is programmed into several instructions.

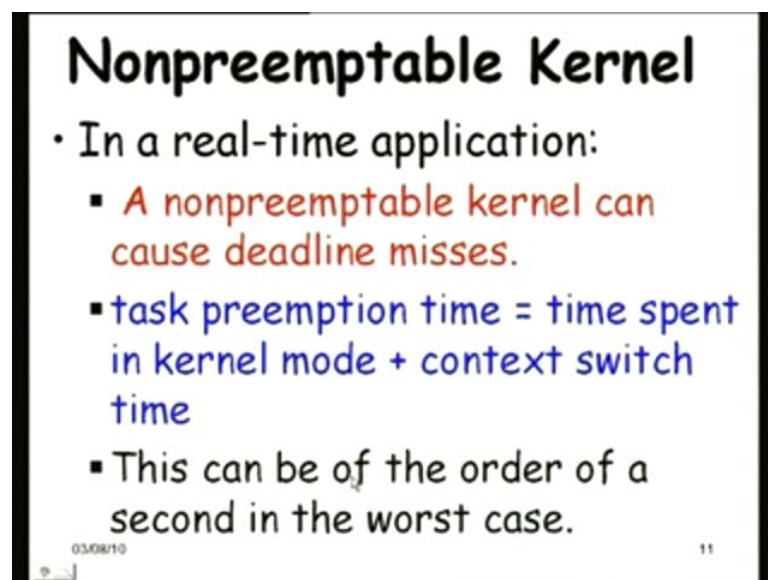
Yes.

And a single statement like  $sum\ a\ is\ equal\ to\ b$ , it is split into 3 or 4 machine of instruction. Now, executing that instructions before updating the  $(( ))$  if it gets interrupted, then mean parallel program state effected  $(( ))$ .

Not really, it is not that complicated. The thing is that, in kernel mode every data is available to every process and in every kernel code, and if 1 process, 1 kernel process has halfway modified a data. And then there is an interrupt, and there is a new process starts coming kernel process starts executing and modifying this data. The kernel data becomes inconsistent that is the main problem the kernel data becomes in-consistent, when 1 kernel code is interrupted and another kernel code starts operating on that, and of course somebody can use locks.

If a process has not completed using some data can use lock, but disabling the interrupt was an easy and efficient way to preserve the integrity of the kernel data before the kernel mode completes, no other process can change it. And of course that time the requirement of... Before the kernel completes another interrupt being handled did not arise kind of applications, they had emphasized those days.

(Refer Slide Time: 17:41)



**Nonpreemptable Kernel**

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

03/08/10 11

So, the main problem why they made kernel Nonpreemptable that has masked all the interrupts, when it is in kernel mode is to preserve the integrity of the kernel data that was the simplest thing to do.

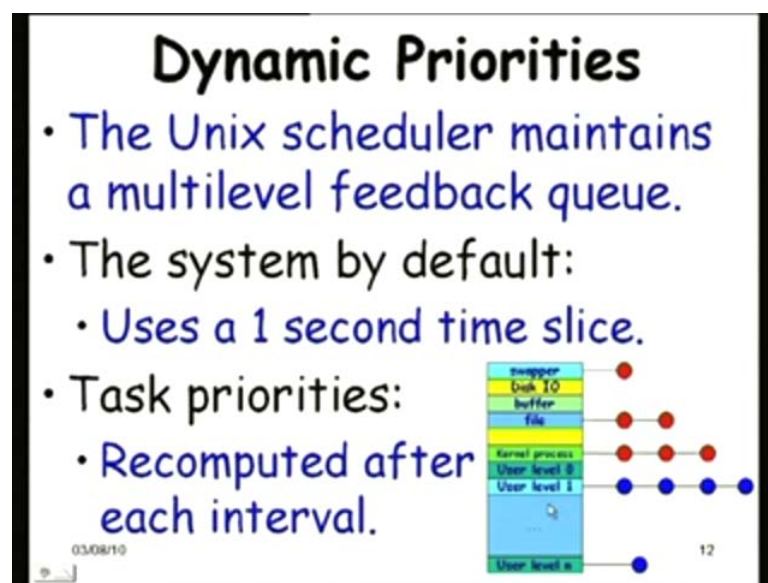
Let the current process complete and then only another process can start, but in a real time application a nonpreemptable kernel can cause deadline misses, why is that.

When a higher level task comes, it does not interrupt their lower level.

It needs to complete the current processing. So, if we look at the preemption time; the task preemption time that would become the time spent in the kernel mode, because this cannot be interrupted. So, the time spent in the kernel mode, plus the context switch time and the time spent in kernel mode can be a second or even more.

So, it can be even more than a second in the worst case, and second granularity in real time applications clearly not acceptable, we need microseconds that is the preemption time which is expected and if it is second, no way it can run hard real time applications

(Refer Slide Time: 19:06)



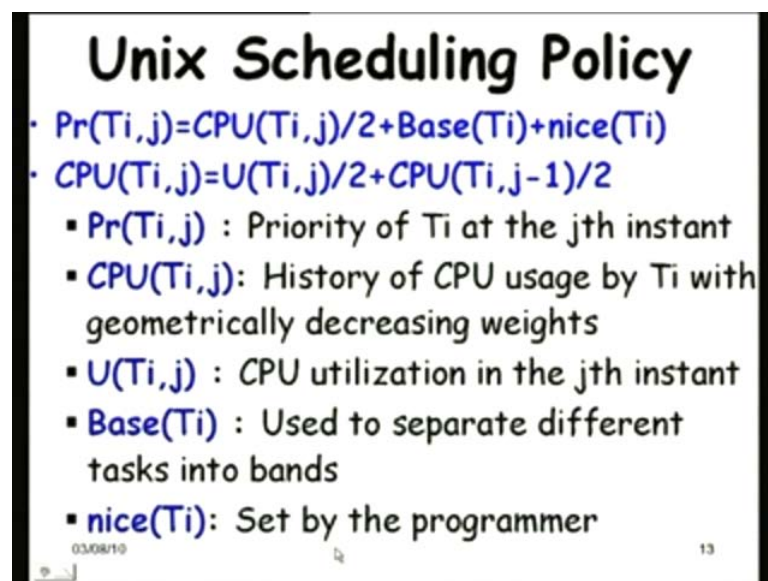
Now let us look at the dynamic priorities. What is origin of the dynamic priorities? How the priorities change dynamically and why it is needed. The Unix scheduler maintains a multilevel feedback queue, see there are different priority processes here starting from the user level processes to the kernel level processes, and then the file handling buffer disk I o swapper and so on, and at each level there are tasks or processes that are queued up, one of them executes until its time slices over, and then it gets feeded back to the same level or may be at a different level.

So, this is the multilevel feedback queue, there are multiple queues here, multiple levels of queues, and each of them there is a feedback mechanism after the time slice just gets reinserted into the queue.

The system by default uses a 1 second time slice. So, after 1 second it gets reinserted back into the queue. Of course, this is a configuration operating system, configuration parameter you can change from 1 second time slice, but if you just install Unix and start using it, it uses a 1 second time slice, because there are advantages of using a smaller time slice larger time slice and so on, which are covered in a first level course.

And after a task completes its time slice, its priorities are recomputed. So, a task at the lowest level here in user, it might just come to a higher level or a higher level process might go into a lower level, of course a user level process will not become a kernel process, It would not go to that priority, the kernel processes operate at a higher priority. Actually, we will see that it is not even that simple there are priority bands.

(Refer Slide Time: 21:23)



### Unix Scheduling Policy

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

03/08/10 13

We will just discuss that little later. So, let us look at the scheduling policy that is used in Unix, this is the policy, but a small variation of this or in slightly different flavor is used in all other operating systems including windows.

So, let us look at this the priority of a task  $T_i$  at its  $j$  time slice is the CPU utilization, it is actually history CPU utilization of the task  $T_i$  at this instant that is the history of CPU

utilization divided by 2 plus some base priority and some nice value. And, see here the history of CPU usages, this actually defined recursively, the history of CPU utilization by task  $T_i$  at  $j$ th instance is utilization in the last instance the CPU utilization,  $u$  is the CPU utilization. In the last time slice that just got completed that has half priority that is 50 percent weightage, and its history before that is from its first time slice to the  $j$  minus 1 time slice is given half weightage.

So, we will see the implication of this. So,  $P_{r T_i j}$  is the priority of the task  $T_i$  at the  $j$ th instance.  $U_{T_i j}$  is the history of CPU just by  $T_i$ , and we will see that this translates this formula of priority computation translates into a geometrically decreasing weight for the history of CPU utilization.

Sir, what is that nice  $(( ))$ .

We will come to that, we will see what are the components here the base priority and the nice priority why they are there, we will see that.

$U_{T_i j}$  is the CPU utilization in the  $j$ th instant, then the  $j$ th time slice it has got. So, how much  $j$ th time slice is the last time slice it utilized.

So, last time it was allocated CPU, how much CPU it could use. Could it use the entire time slice or could it just use a small fraction of it, why cannot it utilize the full time slice.

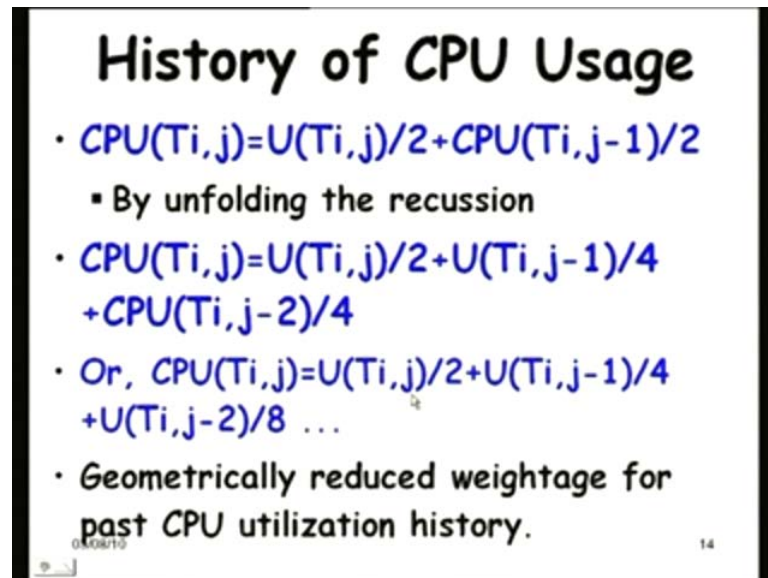
$(( ))$

No, see it can block itself, when it performs I/O or waits for some condition, it changes its state from executing to waiting, I think those are the first level operating system course that when a task waits for something for a page fault or for some I/O to occur, file I/O or may be waiting for some other condition to occur, it relinquishes the CPU and the base component here, the base  $T_i$  is used to separate the different tasks into bands, that is what we are saying that the different tasks are segregated into bands.

The user tasks into several bands and the kernel tasks also into several bands, we just see that what are the bands. And, the nice  $T_i$  is a value set by programmer when creating a process. If you look at the Unix system call a fork or something, you will see that you can specify a nice value.

A nice as the name says, you can be nice to the other processors, and decrease your priority, your task the programmer can decrease his tasks priority by setting the nice value to a plus some value, and negative nice value is not allowed in the user mode. So, a user mode program can only be nice to other programs, let them execute at higher priority and this can execute at a lower priority.

(Refer Slide Time: 25:59)



**History of CPU Usage**

- $CPU(T_i, j) = U(T_i, j)/2 + CPU(T_i, j-1)/2$ 
  - By unfolding the recursion
- $CPU(T_i, j) = U(T_i, j)/2 + U(T_i, j-1)/4 + CPU(T_i, j-2)/4$
- Or,  $CPU(T_i, j) = U(T_i, j)/2 + U(T_i, j-1)/4 + U(T_i, j-2)/8 \dots$
- Geometrically reduced weightage for past CPU utilization history.

08/08/10 14

Now, let see this history of CPU usage, because that was one important term there, besides the base and the nice values which are constant. The history of CPU usage was the main term there. So, this was the term actually the history of CPU utilization by task  $T_i$  at the  $j$ th instance is defined as half the weightage to the utilization in the last time slice, whether it was 100 percent or 20 percent or whatever, that is given half the weightage and all of the time slice has before that are given the other half weightage 50 percent weightage to this and 50 percent weightage to this, as we are saying that this is a recurrence relation.

Should that not the  $U T_i$  a minus 1, because you are just telling us that previous.

No,  $T_i j$  is the  $j$ th time slice that just got completed.

And  $CPU T_i j$  that is the  $(( ))$ .

This is  $CPU T_{i,j}$  is the history of CPU utilization at the completion of the  $j$ th time slice. So, this is the utilization that occurred in the last time slice plus the history of CPU utilization that existed or that exists at the completion of the  $j - 1$  time slice.

Now, this is a recurrence relation, see here  $CPU T_{i,j}$  is defined in terms of  $CPU T_{i,j}$ . Now, whenever we have recurrence relations we know from a first level knowledge in mathematics we need to unfold the recurrence.

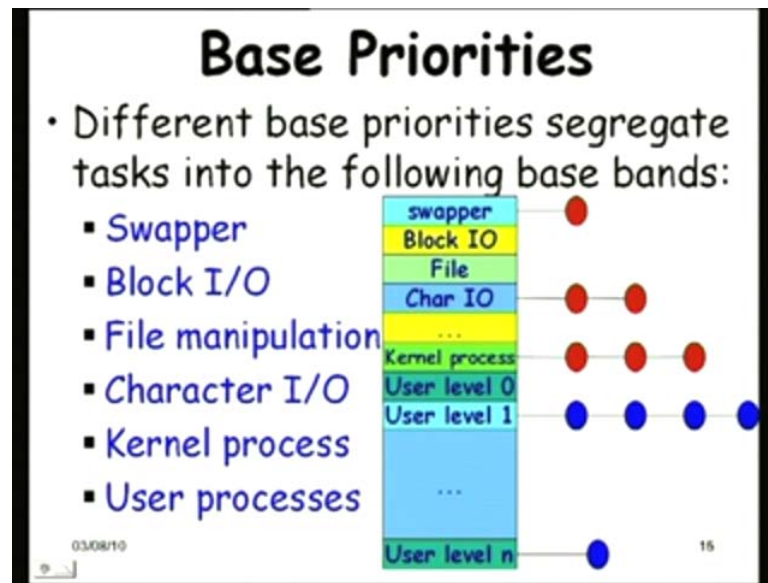
So, this should have been unfold recurrence. So, see here if we unfold it, so  $CPU T_{i,j}$  minus 1 by 2, we can write in terms of  $U T_{i,j}$  minus 1 by 2 plus  $CPU T_{i,j}$  minus 2 by 2.

We can substitute this here, and then we will get by substituting this one here we will get  $U T_{i,j}$  by 2 plus  $U T_{i,j}$  minus 1 by 4, plus  $CPU T_{i,j}$  minus 2 by 4. Is that ok or should we need to do that, it is clear to everybody.

So, that is by just one level of expansion, we can keep on expanding that. So, we can get this relation  $CPU T_{i,j}$  is  $U T_{i,j}$  by 2. So, that is the CPU utilization in the last time slice has 50 percent weightage. The utilization at the time slice just before that it is the  $j - 1$ th time slice is given 25 percent weightage,  $j - 2$  time slice is given 12.5 percent weightage and so on, this is geometrically decreasing weightage to the history of CPU utilization.

But what is the motive, why do they do this. Let us investigate that, because as you are saying that it is not that the Unix does this, but every other traditional operating system does this, they recomputed the priority at the end of a time slice.

(Refer Slide Time: 29:42)



Before that before we look at why they do that lets understand the base priority and the nice values. The base priority segregate tasks into bands for example, the kernel tasks are segregated into swapper, block I O, File manipulation processes, character I O for processes, and the kernel process and then the user process these are the bands.

So, all the tasks that are there in the system are segregated into these bands, see here this is the user band, this is one band, User level n to user level 0 then we have the kernel processes and then this ones to handle different types of I O; the character I O, file I O block I O, and the swapper, what is the difference between a character I O, file manipulation, block I O and swapper, can anybody given an example of a character I O.

Keyboard.

Keyboard is character I O exactly, and what about block I O.

I g b transport module (( )).

In block I O you perform blocks of data transfer for example, in a DMA. So, When DMA becomes useful, I mean just give one example of operation where DMA is used.

When one system transfers the data (( )).

You mean to read out onto the media.

Yes, when you are outputting this data to printer (()).

Writing data to printer, but printer is I mean you mean a printer has a buffer there and the buffer is written is it, because if it takes a character at time DMA cannot occur.

One of the most popular examples is a page fault. A page of data is transferred that is a popular example of a block I O is a page fault, but what about swapper.

Page fault.

No, that is page fault you mean.

When page fault occurs (()) one you have to put one out one page out and.

No, it does not occur like that. See, when a new page comes if there is a block available there just write on that, and if block is not there you replace it and it does not get written immediately.

It need not be written immediately, it does not occur like we have to swap bring in 1 page and at the same write no.

What's a swapper, what is the swapper functionality in an operating system you have component there called as a swapper if you heard.

See, if a user task remains inactive for long time it is swapped, all its context and all its pages are swapped out from the memory. So, here multiple pages are swapped out.

As a process needs memory (())

Not really you see dormant processes which are inactive it determines when a process becomes inactive.

Periodically checks these are the (()).

The operating system checks, whether a processes is active if it is a Dormant process, it is swapped out.

If you check the status of processes, if you give a listing of processes and then see you will say that it will. So, some of the process are swapped out, because they are inactive they are swapped out and here multiple pages needs to be written at the same time.

(( ))

It swaps them out unless the process itself terminates, it does not, it swaps them an inactive process, exactly just frees the memory all the pages of an inactive process are swapped, but then after having understood that, Why is this priority bands ordered in this way kernel process which do not do I O, then see here character I O, file I O, block I O and then swapper.

Why swapper at the highest priority than block I O next highest file I O then character I O at a lower priority and then the kernel process among all the kernel process.

So, it is inactive and memory is full.

No, the question is that why is this priority allocation into bands here, the swapper is the highest priority process, block I O next, file I O next, character I O next, and the kernel process is not requiring I O, they operate at this level why is that.

Of course one thing you can notice is that higher amount of data are handled by this character I O just handles some characters some data this larger amount of data, this is blocks of data this entire address space of a process. So, let us investigate that issue.

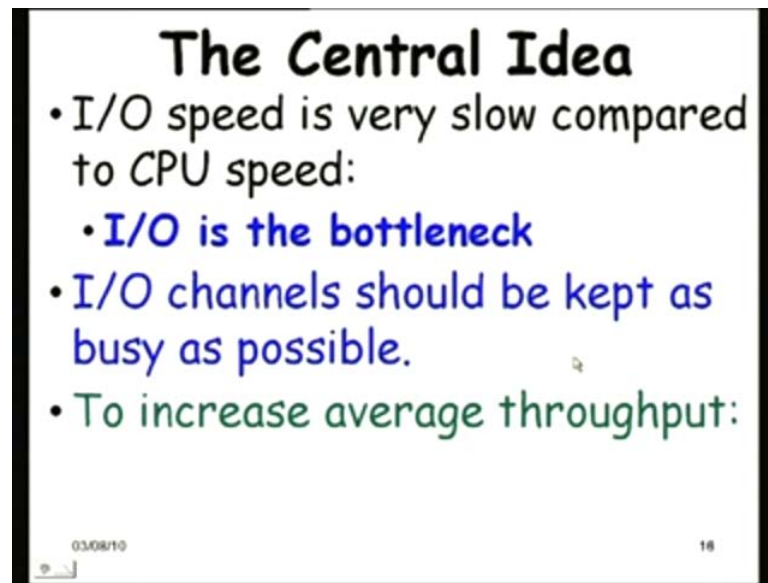
Because they has to release more amounts of data than of this swapper has the individually more amount of space and inactive portion

So, why should it get higher priority, it definitely deals with larger data as we precede from here this?

(( ))

Not really, lets investigate, because that is the coax of the problem we need to understand, and that will become useful throughout our other discussions in this operating system, real time operating systems.

(Refer Slide Time: 36:42)



### The Central Idea

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

03/08/10 18

The central idea in all traditional operating systems including Unix and windows is that the I O operates at a very slow speed compared to the CPU speed, we know that the I O is typically thousand times or may be tens of thousand times slower than CPU, CPU operates at gigahertz.

That's the CPU clock cycle, but I O milliseconds even the fastest I O. SO, in every operation that you analyze the time taken by a process to complete, I O is the bottleneck and we are in a traditional operating system.

We are concerned with the average throughput that is the most important parameter in traditional operating system is average throughput. How many processes could be completed, how many tasks could be completed per unit time.

So, since I O is the bottleneck, the I O channels should be kept as busy as possible do you agree with this.

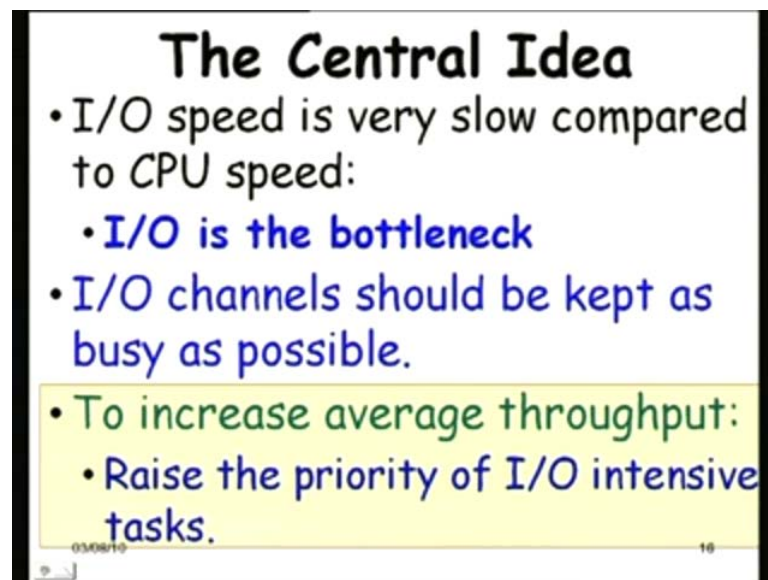
See I O is slow, so the operating system should try to maximize utilization of the I O channel if a process is trying to do I O, it should be given the highest priority, do you agree with this, some of you do not seem to be very convinced, but let me just tell you a small story or incidence.

See, suppose you want to write down something from a notebook, now you ask one of your friends to read out the book or something, and the other person just keeps on writing the two persons who are helping you out.

One is reading out the book and the other is writing, now let us say once in a while they just doze off and you know they waste some time. So, if you want it to be done at the shortest time would you watch the person who is reading or the person who is writing?

See the person is reading very fast now if he is reading the person writing he cannot even complete before he reads one sentence, and he pauses anyway and the writer is the one who is crucial if he pauses then your work will get delayed by that time.

(Refer Slide Time: 39:26)



### The Central Idea

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

So, you have to watch the person who is writing, he is writing always is not dozing off and so on, then he will complete it the fastest that is the basic idea here.

So, the I O channels should be kept as busy as possible and how do you ensure that, increase the priority I O processors. So, to increase the average throughput, raise the priority of I O intensive tasks.

A kernel process which is not doing I O should operate at the lowest priority and then we have the character I O etcetera.

But why should it translate into dynamic priority level. So, this same idea that task performing I O should be given higher priority, that is the idea with which the user process priorities are recomputed after every time slice dynamic re-computation of the priorities after every time slice, this is also the idea anybody would like to guess.

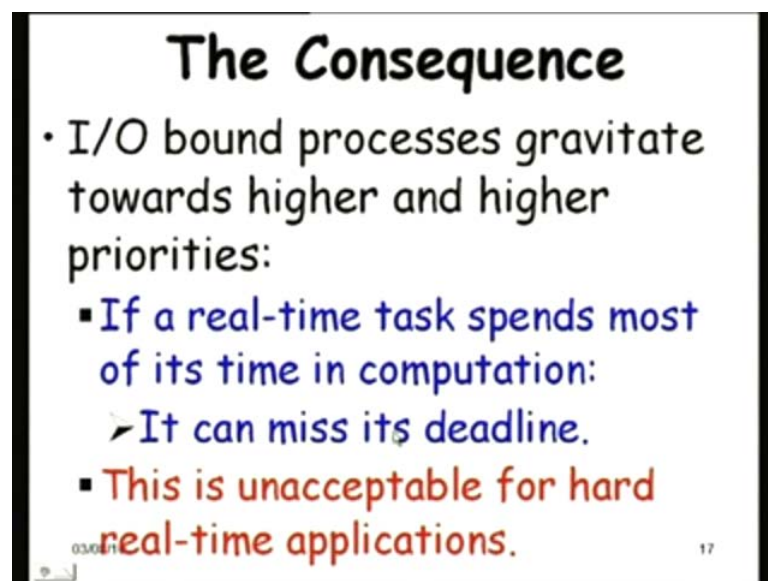
CPU intensive task (()).

The CPU intensive tasks who utilize their time slot completely, they are the CPU intensive tasks. So, they should be given lower priority.

The ones that are doing I O, they have could use only part of their time slice, their priority should be increased.

So, that they will do more and more I O and they do not have to wait, the processes doing I O should operate at a higher priority, and those recently being doing processing their priority should be reduced.

(Refer Slide Time: 41:06)



**The Consequence**

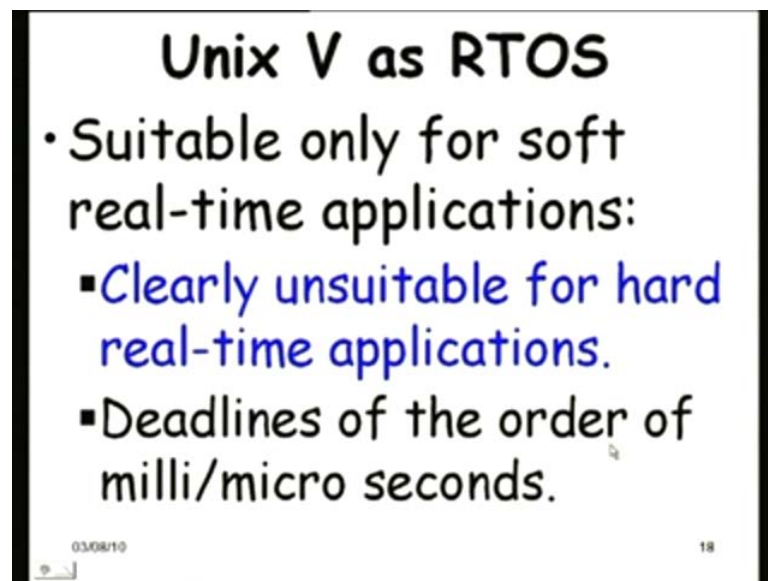
- I/O bound processes gravitate towards higher and higher priorities:
  - If a real-time task spends most of its time in computation:
    - It can miss its deadline.
  - This is unacceptable for hard real-time applications.

03/07/11 17

So, the I O bound processes they gravitate towards higher and higher priorities, the more and more a process does I O, the Unix or windows they will ensure that its priority successively becomes larger and larger. Of course, there is a limit they cannot really cross over from one band to another, a user band process cannot become a kernel band process or kernel band process cannot become a character band process, they operate within their own bands, priority bands, but their priority increases or decreases.

But in a real time situation, if a real time tasks spends much of its time in computation then its priority will decrease and it will miss its deadline, and this is not acceptable, because the programmer is the one who should decide at what priority a task should operate that was the basic thing that we studied in the RMA or EDF, it is based on the deadline requirement, that the priorities are computed rather than whether they are doing I O or using CPU. So, this is unacceptable for hard real time applications definitely lead to many deadline misses.

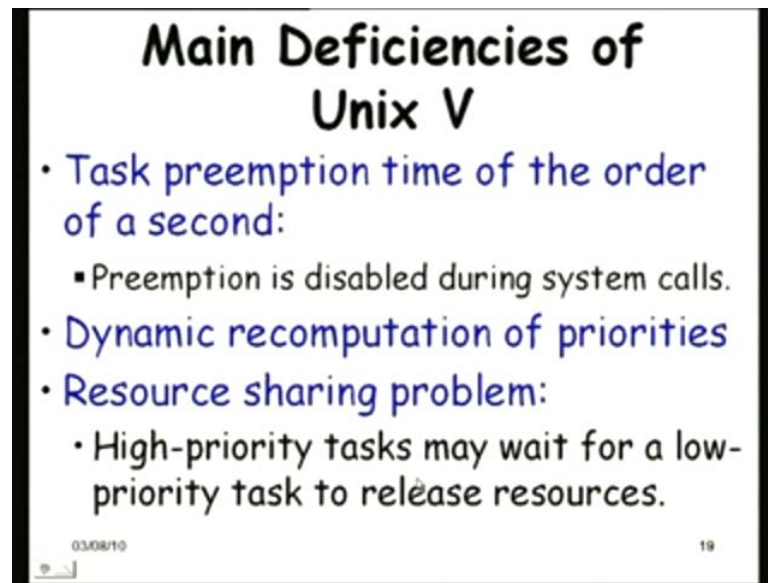
(Refer Slide Time: 42:32)



So, Unix five, system five if we try to use this as a real time operating system we will see that it is suitable only for soft real time applications where the deadlines are of the order of several seconds, because the Nonpreemptable kernel can be delayed by second and then this priority manipulations. So, it can be off by several seconds.

So, if we are talking of a soft real time task where the deadline itself is tens of seconds then Unix can be used, but when the deadlines are of the order of milli or a microseconds; obviously, you cannot use Unix five, it will only lead to failure.

(Refer Slide Time: 43:31)



**Main Deficiencies of Unix V**

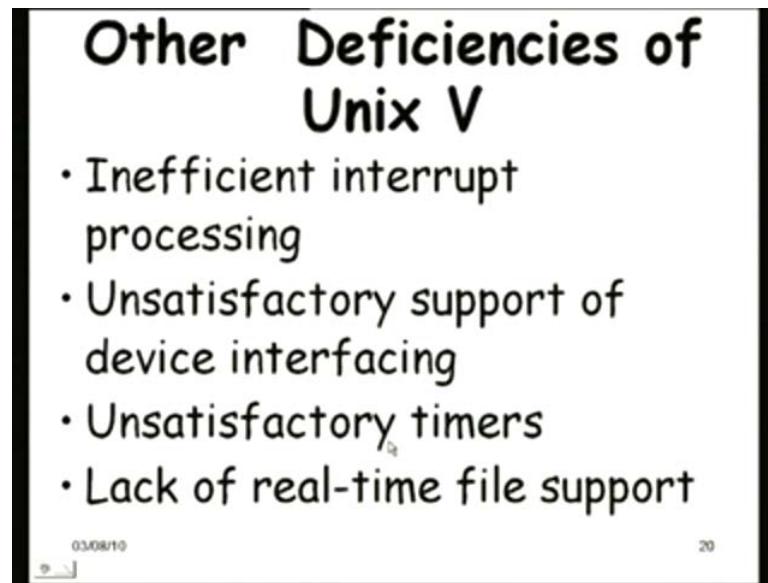
- Task preemption time of the order of a second:
  - Preemption is disabled during system calls.
- Dynamic recomputation of priorities
- Resource sharing problem:
  - High-priority tasks may wait for a low-priority task to release resources.

03/08/19 19

So, the main deficiencies of Unix if we summarize; one is that the task preemption time is of the order of a second, dynamic re-computation of priorities, and then we have the resource sharing problem we did not tell about that, but that is also another problem, because If a low priority process is holding the resource then the higher priority process will keep on waiting for it.

So, the task preemption time is order of a second, because preemption is disable during system calls. There is dynamic re-computation of priorities, because the goal of traditional operating system is to increase the average response time for tasks, rather than letting tasks meet their deadline, and the resource sharing problem is that a high priority task may wait for a low priority task to release its resources and priority inversion can occur.

(Refer Slide Time: 44:37)



There are other deficiencies of Unix V, see those three are the main, there many other deficiencies. For example, interrupt processing is inefficient; it takes several milliseconds for interrupt to be processed, unsatisfactory support for device interfacing, unsatisfactory **timers, lack** of real time file support.

This to some extent can be overcome by doing some extension to the Unix kernel, simple extension. As we were saying that a real time file support, basically the file needs to be written contiguous area in the disk, and also the space for the file needs to be pre-allocated, because if it is written halfway and finds the disk cannot support then there will be a failure.

So, this can be supported even this to some extend by simple extensions to the operating system can be supported, but those three are the main problems that needs to be handled before this can be used for real time applications.

(Refer Slide Time: 45:50)



**Microsoft Windows as RTOS**

- Sometimes it may be required to use Windows as RTOS.
- Windows NT series:
  - More stable than the Win98 series
- POSIX support
- Multithreading

03/08/10 21

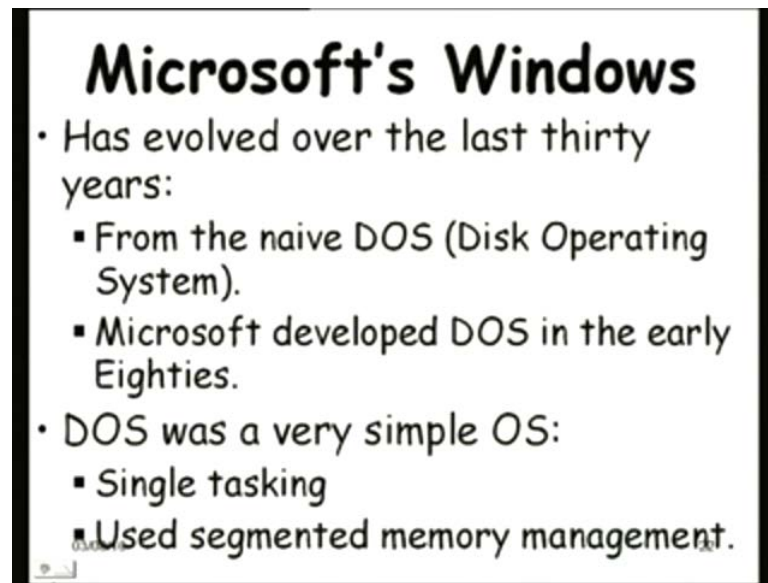
So, we examined the Unix as a real time operating system. Now, let us do the same thing for windows, because of availability of windows, easy availability on desktops and so on laptops we might attempt to use windows as a real time operating system, one thing that we should know is that after Windows NT, the system has become stable it does not crash unnecessarily whereas, before the NT series was the 98 series where it would just crash for no reason.

For example you are doing a Microsoft word application and suddenly see that the screen has frozen, it is not that you by mistake clicked, normal users think that did I do something wrong, did I click the mouse two times or three times; what caused the problem, no those are not the person who is using his problem, it is the bug in the operating system which made it crash. So, before 98, win 98 it was unstable it will crash for no reason, basically bugs in the operating system.

But then the NT series came up after that and we are all using the derivatives of the NT series which is much more stable, and you can think of this as being used in real time operating system.

An unstable operating system cannot use for real time application or safety critical application. And we have posit support for windows, we will shortly see what this means the posit support and then it supports multithreading some good features are there for windows.

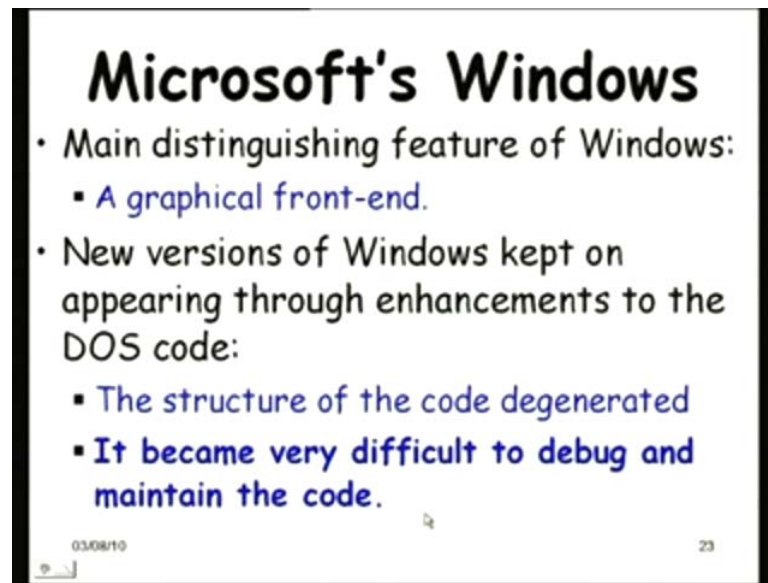
(Refer Slide Time: 47:38)



Now, let us see what are the difficulties that might occur if we use windows for real time applications, but before that lets look at little bit inside the windows just like we did for Unix.

The windows if you look at it has evolved over the last 30 years from a very simple disk operating system, which was kind of a toy operating system where every task can do anything, you can even delete the operating system itself. It has evolved from there and has become a dominant operating system nowadays. The DOS as you were saying was developed in 1981 released in 1982 very simple operating system, the DOS in those days it was single tasking, used segmented memory management.

(Refer Slide Time: 48:37)

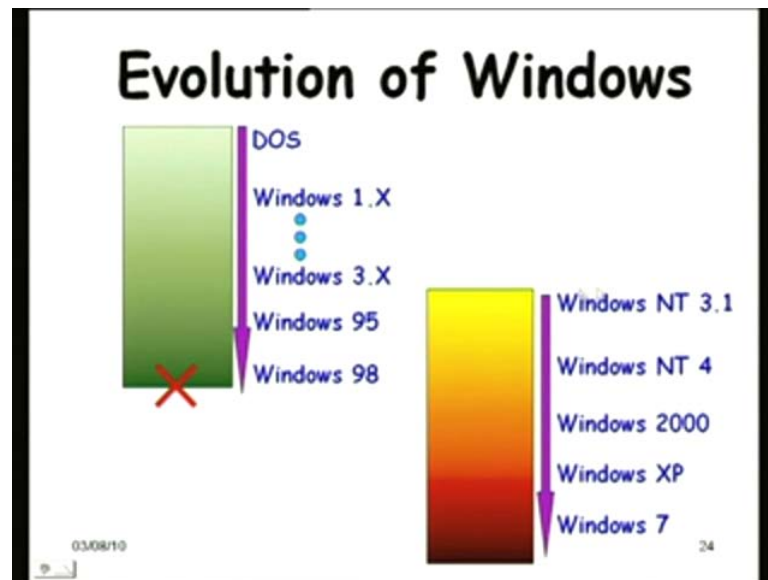


And, then the windows appeared in 1986. The main distinguishing feature of the windows from the DOS was a graphical front end, initially the windows was also single tasking, but it was a graphical front end which was one of the major innovations of the windows operating system, Unix lacked that and they even did not support it until the Linux supported a graphical front end.

New versions of the windows kept on appearing very frequently every year 2, 3 releases through enhancement to the DOS code, and you know if you have a code a DOS code or something 82 and you keep on changing it each time trying to add new functionality, picks the bugs. And so on it will become a bad code any maintained code it becomes a bad code and the structure of the code degenerated after every time the code was changed to bring a new version, and then it became very difficult to debug and maintain the code, and it is said that in Windows 95, 50000 known bugs existed.

They have been reported, but they could not be corrected, because the code is so hard to correct it, it is become the structure is really bad 50,000 known bugs, and definitely it was unstable use Microsoft word and just clicked it crosses and that kind of thing. Of course, these could not be used for real time applications.

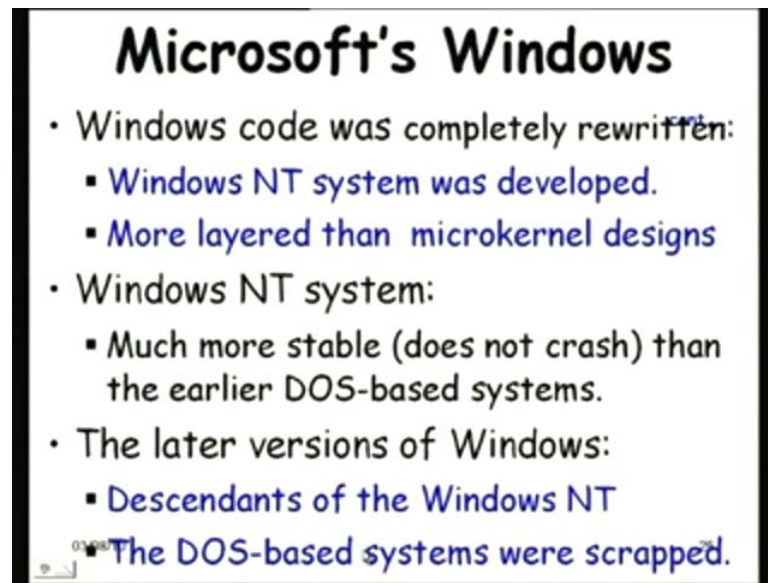
(Refer Slide Time: 50:18)



So, if you look at the evolution, the DOS in 1982 and kept on evolving finally appeared windows in windows 1.0 and then Windows 3.X, Windows 95 and Windows 98 and here this code was junked.

And alongside this a new code was written form scratch that is the NT code. The NT code was written here alongside here just before Windows 95, something around 1993 the Windows NT 3.1 was released parallels with the Windows 95, and Windows NT 4 is the Windows NT that we know popularly, and from there the derivatives windows 2000 and Windows XP, Vista, windows 7 etc kept on appearing.

(Refer Slide Time: 51:20)

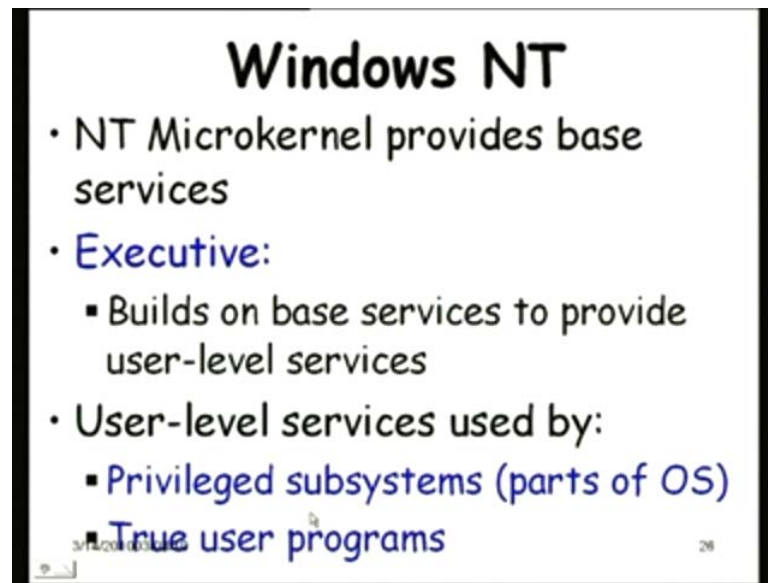


So, the windows code was completely rewritten in the NT system, Windows NT and it used microkernel design, but if we really analyze it, it has layers, more layered than actually microkernel, it has a small microkernel and that is executed as we will see in the diagram.

The Windows NT system is much more stable than the DOS based systems, needless to say, because we will see that the microkernel designs their it is much more easier to develop, much more easier to handle bugs, because their user level processes most of them remember only small kernel and debugging kernel code is difficult, but debugging user level process is not a problem and definitely the NT system becomes stable with the partly microkernel design.

And, the later versions of windows that become available are descendants of the Windows NT, and the DOS based systems were scrapped.

(Refer Slide Time: 52:40)



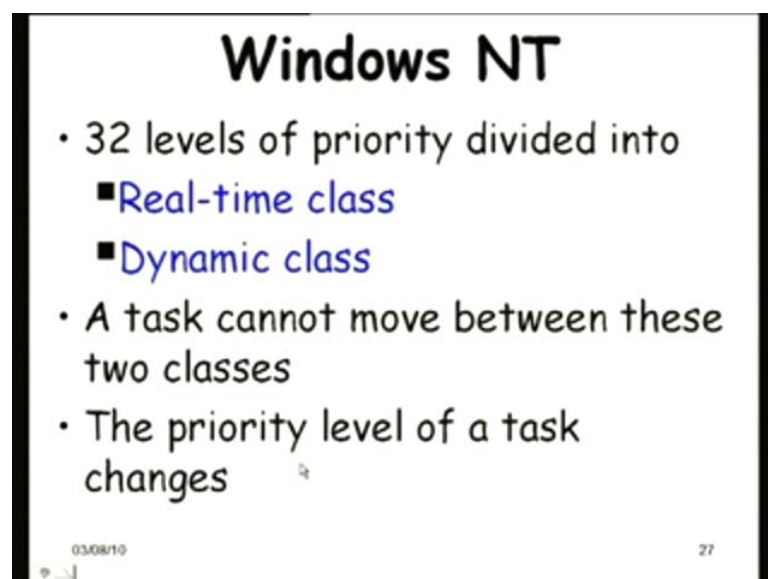
**Windows NT**

- NT Microkernel provides base services
- **Executive:**
  - Builds on base services to provide user-level services
- User-level services used by:
  - **Privileged subsystems (parts of OS)**
  - **True user programs**

3/13/2010 10:30 AM 26

As we will see that the NT microkernel provides very basic services, just like any microkernel and then there is an executive. The Windows NT executive which builds on the base services provided by the Microkernel to provide user level services, and these user level services are used both by the operating system that is the privileged subsystem and also the user programs they invoke the user level services provided by the Windows NT.

(Refer Slide Time: 53:19)



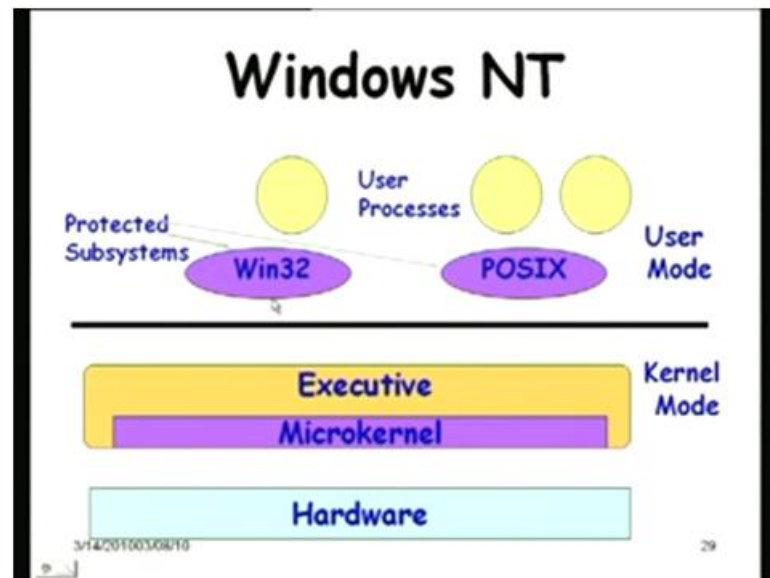
**Windows NT**

- 32 levels of priority divided into
  - **Real-time class**
  - **Dynamic class**
- A task cannot move between these two classes
- The priority level of a task changes

03/08/10 27

So, I think we will just examine little bit about the way the NT operates I think I have a diagram here.

(Refer Slide Time: 53:37)



We will see the executive microkernel and the Win 32 and the POSIX interface, and today we are running out of time, so in next class we will look at Windows NT the basic features just like we looked at Unix, and then see that what are the difficulties if we use it as for real time applications, and how it can be extended or how it has been extended by Microsoft to provide Win C which is for real-time, which is the real time operating system for from Microsoft the Win C, whereas in Unix there are many variants of Unix which are based on the basic Unix and trying to make it suitable for real-time application. So, we will stop here, then meet in the next class.