Embedded Systems

By

Chandra Sekhar G.

Assistant Professor

Electronics and Communication Engineering Dept.

Bharati Vidyapeeth's College of Engineering

New Delhi

UNIT - IV

- **RTOS:** Embedded Operating Systems, Multi – Tasking, Multi – Threading, Real-time Operating Systems
- RTLinux introduction, RTOS kernel, Real – Time Scheduling.

Note: Prerequisites: Clear Conceptual knowledge in Microprocessors, Microcontrollers and their Functional Units.
This Notes is prepared mostly using NPTEL Videos of Embedded Systems, IIT Delhi. Some concepts are taken from Internet.
This notes is prepared to complete the syllabus in the given number of lectures and in the exam point of view. For more and detailed theory go through the text book Computers as components: Principles of Embedded Computing System Design, Wayne Wolf, Morgan Kaufman Publication, 2000.

Fundamentals of Embedded Operating Systems

In this unit, we will discuss Operating Systems for Embedded Applications and Embedded Appliances. Obviously, there would be spatial requirements, when we are considering embedded systems. We shall start with reviewing the general characteristics of operating systems before going into the specific needs of embedded systems.

Operating System****

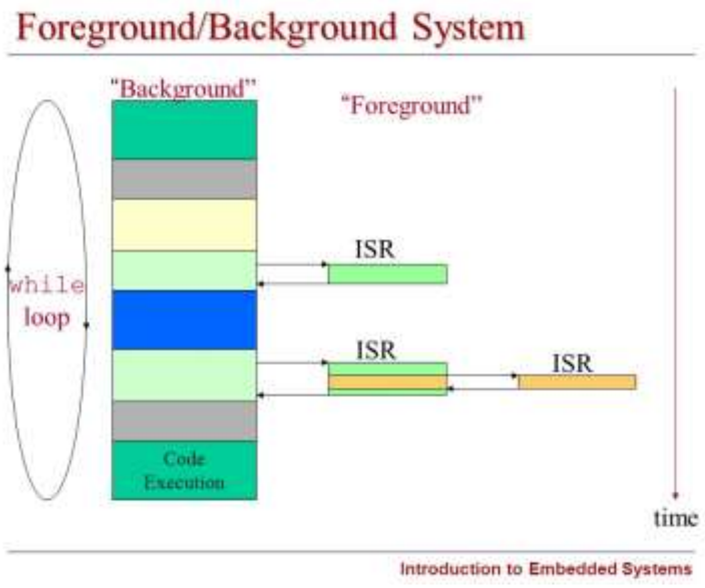
- Exploits the hardware resources of one or more processors
- Provides a set of services to system uses
- Manages memory primary as well as secondary memory and IO devices

What is an Operating System? Operating system is that layer of software which exploits the hardware resources of one or more processors which is their in the Embedded System or for that matter any general purpose computing platform. Therefore, it provides a set of services to system uses because, system is making use of hardware resources through that software layer which is your operating system. OS manages memory primary as well as secondary memory and IO devices.

Foreground/background systems***

- Small simple embedded systems usually do not have an OS
- Instead an application consists of an infinite loop that calls modules or functions to perform various actions in the background, because it is a common flow with different functionalities to be checked to.
- interrupt service routines (ISRs) handle asynchronous events in the "foreground"

This is the basic structure for the more common systems. But, on the other hand, if you really have complex functionalities to support, then in that case we really need to have an OS. The basic model of this kind of simple embedded systems is software management software is basically referred to as foreground background systems.


Foreground/Background System
"Background"
"Foreground"
while
loop
ISR
ISR
ISR
Code
Execution
time
Introduction to Embedded Systems

So, the structure wise this is a general while loop. Because, this software is expected to run for ever, which has got the different modules and this modules gets invoked within the while loop. Asynchronous events which are generated from the external environment or handle by the interrupt service routines. So, this is the basic structure of management software, when, we really do not need to manage multiple concurrent tasks.

Why Embedded Operating System?****

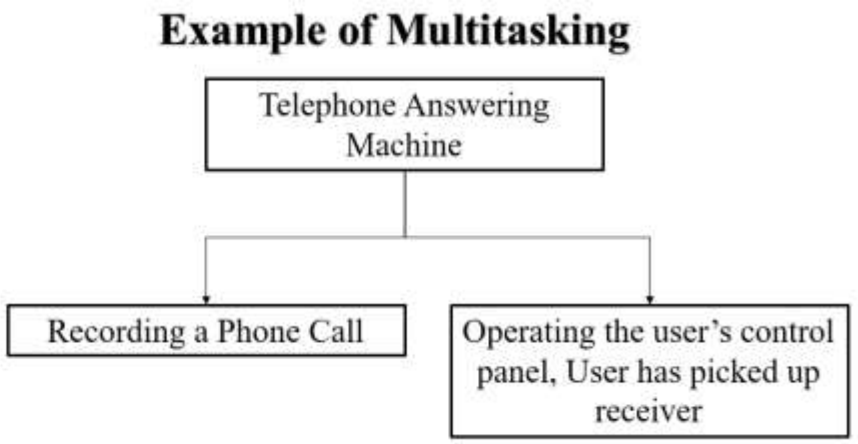
For complex applications which supports a multitude functionality.

- We need mechanisms to implement each functionality independent of others. That means, the functionalities will not be link to each other through say proceed your calls. So, they have to be implemented independently.

- And but still, we need to make them work cooperatively if this is the condition, then we really need an OS to manage that embedded system. And we had already seen that external world has got concurrent event stream or data stream.

Embedded Systems need Multitasking****

- Multitasking – The ability to execute more than one task or program at the same time.
- So, if you need to deal with concurrent event or data stream will require multitasking. Because, with each stream we need to have a special functionality associated.
- And in Multitasking system CPU switches from one program to another quickly. So, that it gives the appearance of executing all of the programs are the same time.
- In fact, we have a same processor and on the same processor these tasks are being time shared. And these tasks are actually independent modules. And these therefore, functionalities are getting implemented or realized through independent modules.


Example of Multitasking
Telephone Answering Machine
Recording a Phone Call
Operating the user's control panel, User has picked up receiver

So simple example of multitasking is here, where you have got a telephone answering Machine, it can record a phone call also the same time. It may meet the user control panel inputs other requests. And for example, user has picked up the receiver. So, that is the external scenario which it needs to create to as well as it record a phone call as well. And these two can be consider independent concurrent task and that is to be managed by the OS resident in this telephone answering machine.

Nature of Multitasking

What are the different kinds of multitasking systems that you encounter?

Cooperative Multitasking

- One is cooperative multitasking each program can control the CPU as long as it needs it. It allows other programs to use it at times when it does not use the CPU. That means, the other programs get the CPU and can do the task only when one of the programs is not using. So, it is a complete cooperation with the complete knowledge about each other, which may not be always true.

Preemptive Multitasking

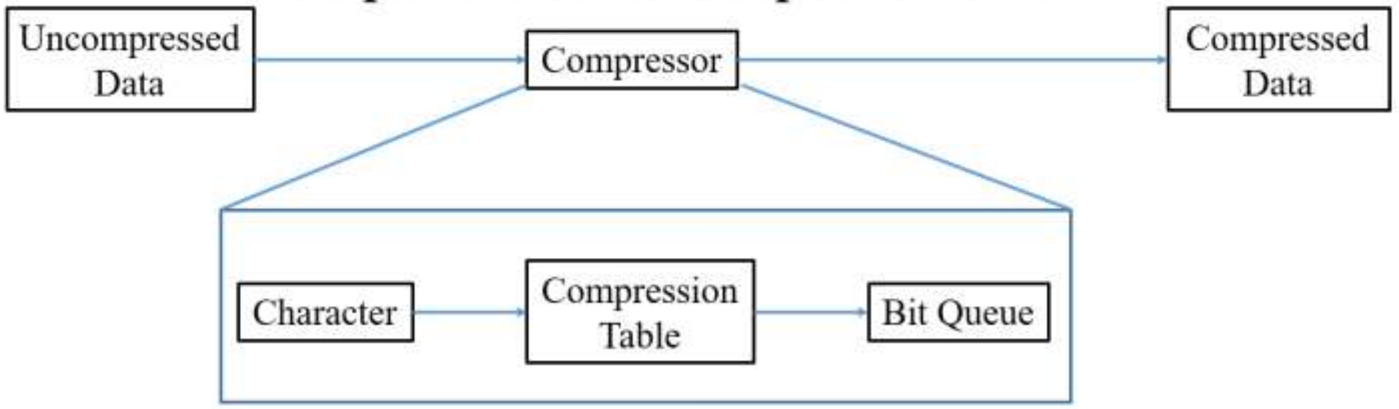
- Side by side you have got preemptive multitasking. In preemptive multitasking, a process, which is in execution or each program, which is in execution can be interrupted that is preempted.
- And another process can be schedule to run in its space. And in this condition OS provides each running program with specific time slices. So, each program runs for its time slice and then it is preempted out. It is not that a process will completes its job or task with in a time slice allotted. It will require possibly multiply time slices to finish its task. Now, when these processes are running concurrently they may have different characteristics as well.

Complexities of Multitasking

- Multi-rate tasks
- Periodic tasks
- Concurrent tasks
- Synchronous and asynchronous tasks

When, we talk about periodic tasks it means the tasks have got as specific period with which their rife and there to be serviced If you consider a simple decompression task for decompressing a video. Then, that task arrives at each frame time interval. If you have 30 frames per second or 25 frames per second then these tasks would arrive at 40 millisecond interval. So, these are an example of a periodic task.
Now, when a process is periodic it may have other processors in the same OS which are also periodic. But, there period can be different from that of a particular periodic process. So, if an OS needs to handle multiple processors which come with multiple periods. We actually encounter, what is call multi rate tasks.
And, all these task are in a way concurrent, because they have to be in state of execution at the same point of time. It is not that one task will be finished then only I can created to another task. There also synchronous and asynchronous tasks, a task which is spooned by an external interrupts. Or an external user action would be typically an asynchronous task and that needs to be created to in this scenario as well. So, all these things together make the problem of multitasking more difficult.

Example: Complex Timing Requirement of a Compression unit



Let us take a simple example of a compressor, a compressor which is compressing data. So, you have got uncompressed data and there is a compressed data's output. And this has to work in time. So that, you should not need any of the input data which is coming in an uncompressed form if, you look into the overall thing the depending on the character.
And if you using a compression table and if you use a variable length coding scheme like half bank coding then a character would give rise to data of a particular length. For that length is not same for each and every character. So, that problem needs to be handled.

Cooperating Tasks

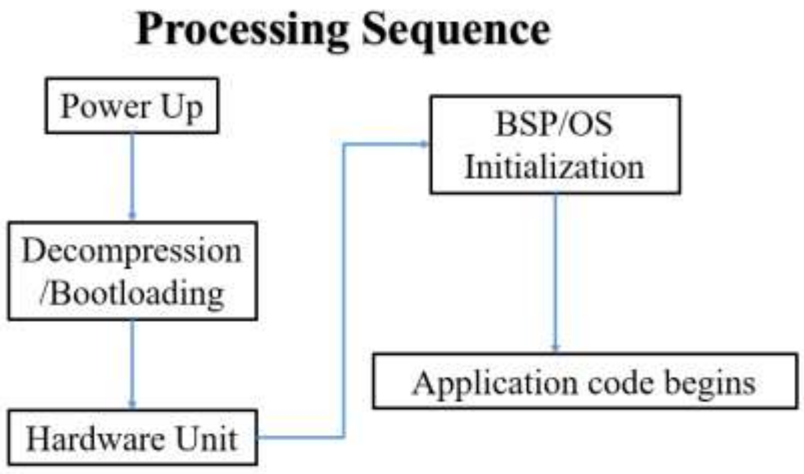
- Data may be received and sent at different rates – example one byte may be compressed to 2bits while another may be compressed to 6bits
- Data should be stored in input and output queues to be read in a specific order
- Time for packaging and emitting output characers should not be so high that input characters are lost.

So what we say, that the input the data may be received and sent at different rates. So, there are two processors which are managing the reception and communication. So, these two processors are working at different rates and then it to cooperate such that no input data is really missed. So, time for packaging and emitting output characters should not be so high that input characters are lost.
So, this puts in a constraint on the compression scheme. And these processors have to cooperate, so that the compression can take place in proper order. Now, let us look at with this background. Let us look at the fundamental issues of the OS. We shall base or discussions primarily around considerations of an embedded system.

How a system works?*****

- Wake Up Call – On Power_ON
- Execute BIOS (Basic Input and Output System) – instructions kept in Flash Memory – type of **read-only memory** (ROM) examines system hardware.
- **Power – on self test** (POST) checks the CPU, memory and basic input-outpur systems (BIOS) for errors and stores the result in a special memory location

So first, what happens, when you power up an embedded system. Typically, there is a power on situation. And if there is an OS in a power on situation, you execute, what is called a basic input output routines, which may be part of ROM. ROM may be flash memory as well we had discussed is other otherwise also. And these memory initializes examines the system hardware.
In most of these embedded systems, there is a power on self desk block. We check the initial when it boosts up it checks the hardware. Typically, memory test is performed, in order to ensure that the embedded system does not give rise to unsaved state or kind of an erroneous situation, because of the hardware fault. So, if you detect the hardware fault it would about.



So basically, what we are looking at, in terms of the processing sequence. The moment the power is up. So typically, you do what is called Decompression or Bootloading. This is basically your basic IO routines and the OS which is part of your ROM or your Flash is loaded onto the RAM for faster execution. Then you going to, what is called this BSP or Board Support Package or OS initialization and then only the control gets transfer to application code.

Board Support Package

What is this BSP or OS initialization?

- BSP is a component that provides board or hardware specific details to OS for the OS. To provide hardware abstractions to the tasks that use it is services
- BSP is specific to both the board and OS for which it is being written.
- In fact, this can be configured also for the board for which it is being written. And if you see, what it takes about is, it provides board and hardware specific details to the operating systems.

BSP is a startup code

- Initializes processor
- Sets various parameters required by processor for
  - Memory initialization
  - Clock setup
  - Setting up various components like cache
- BSP also contains drivers for peripherals

So, the basic component of the BSP is a startup code. And we are already discussing the startup code for a RAW system and what kind of functions it would support. So, it basically initializes the all the components of the system. Does the memory initialization does a clock setup and setup various components like cache.
And BSP also contains typically drivers for the peripherals which now becomes memory resident. So, these BSP are the Board Support Package forms a very important component for all operating systems which are targeted for embedded appliances. And for each appliance is the target card you may need to configure your board support package for a particular OS.
So, for example, if I am talking about ecash, ecash is a OS targeted for embedded appliances. But, for a particular card I need to properly set the board support package. So, that on that card or the target system ecash can operate properly.

Kernel****

- Most frequently used portion of OS
- Resides permanently in main memory
- Runs in a privilege mode
- Response to call from processors and interrupts from devices

Then the control; obviously, gets transferred to the kernel the OS kernel. OS kernel is the most frequently used portion of OS resides permanently in main memory. That means, in the RAM area and even if you are implementing a kind of a virtual memory between flash and a RAM possibly, we would not like the kernel to be part of that kernel would be typically lock in the RAM area of the embedded system.
It runs in a privilege mode, because it supports the system functions. So, if you remember ARM processor. ARM at various modes of which the privilege modes or system modes. So, the kernel on an ARM would run in privilege mode. And it response to call from processors and interrupts from devices. These processors could be the application processors. So, they can make the system calls to make use of OS services.

Kernel's responsibility****

- Manage processors
- Context switching: alternating between the different processes or tasks
- Scheduling: deciding which task/process to run next
    - various scheduling algorithms
- Critical sections = providing adequate memory-protection when multiple tasks/processes run concurrently
    - various solutions dealing with critical sections

So, what are the responsibilities of kernel? Kernels basic job is to manage processors. In fact, all processors are created by the kernel. In fact, they are deleted or killed by the kernel as well. Since there are many processors and we are talking about multi-tasking system. And therefore, what is needed? The kernel needs to manage, what is called context switching, alternating between the different processors or tasks.
This read to this concept of scheduling. When we are context switching which process or task should run next. This is the principle of scheduling and kernels implement a variety of scheduling policy. Then we have critical sections, providing adequate memory protection, when multiple tasks processes run concurrently, because there may be shared resources.
If there are shared resources shared by two or more processors then access to the shared resources have to be disciplined. So that one of them, are is really using the shared resource. That job is also handled by kernel and there are various solutions dealing with critical sections and these solutions differ from OS to OS.

Process****

- Process is a unique sequential execution of a program
    - execution context of the program
    - all information the operating system needs to manage the process
- process characteristics
    - period that is a time between successive executions
    - rate, inverse of period
    - in a multi rate system each process would executes at its own rate

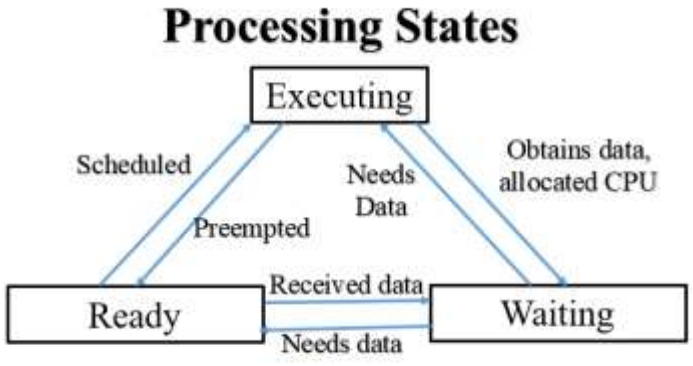
So now, we are talking about multitasking and processors selects formally look at, what a process is? Process is a unique sequential execution of a program. Now, if a program even in a general purpose system is same program if it is being executed by two different users. We actually shall have two different processors associated with the same program.
Even the same program instantiated into two different processors, but the same user will also have distinct existence. So each process therefore, has got an execution context. So, all information this execution context is collection of all information the operating system needs to manage the process. And, what are the different process characteristics? One will be period that is a time between successive executions.
Because, we said that there can be periodic processors associated the period is rate, inverse of period is actually the rate. And in a multi rate system each process would executes at its own rate. So, this information is important for the

kernel as well, because kernel schedules the processes through context switching. So, this is one aspect of the information about the process.

Process control block (PCB)

- Process control block
    - OS level data structure which holds the pieces of information associated with the process
- Process states: new, ready, running, waited, halted etc
- Program counter: contents of the PC
- CPU registers: contents of the CPU registers
- CPU scheduling information: information on priority and scheduling parameters
- Memory management information: Pointers to page or segment tables
- Accounting information: CPU and real time used, time limits, etc.
- I/O status information: which I/O devices (if any) this process has allocated to it, list of open files, etc.

In fact, all these information about the processes as stored in what is called a process control block. Each process has got it is own process control block. This is the OS level data structure which holds the pieces of information associated with the process. And the process has got different states new, ready, running, waited, halted etcetera and what are the states depends on again the definitions each OS really adopts.
In a generic sense, when a process is admitted, it becomes a new process. A new process may not be always ready to run after some transformation meeting some other requirements. And the demands of the new process it becomes ready to run. When the process is actually in execution it is state is running. When the process is waiting for some external input or access from the device it state is waited.
The process can also be halted for some reason and it can be also put to different kinds of other states. In fact, in a general purpose a system a process can be moved out of the memory and put to the hard disk waiting for it is time to come. So, there may be variations of these states that we have talked about. So, this process state information is stored in process control block.
Then, the program counter program counters points to the current instruction of the process which is being executed. So that will be stored in the PCB why, because when there is a context switch. These PC value has to be restored. Then also, the CPU registers which captures the state of the process. And also the CPU scheduling information priority and scheduling parameters for which the period as well as the rate can be important for the OS as well.
It also has memory management information. We had already talked about in the context of memory that your memory can be organized in terms of pages or segments. And there has to be page table for each process. So that pointer to the page table becomes part of your PCB also accounting information.
As well as IO status information; that means, which IO devices if any this processes has been allocated to it. If it is using a file system the list of open files extra, because this records the resource usage information, resources other than that of CPU and memory.

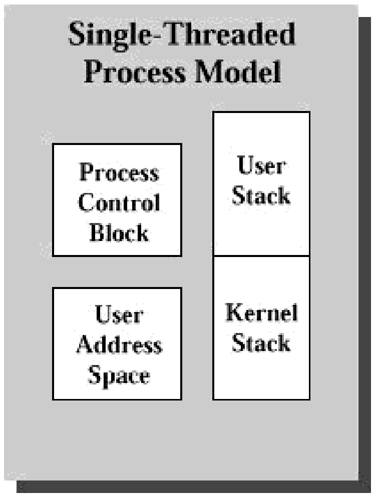


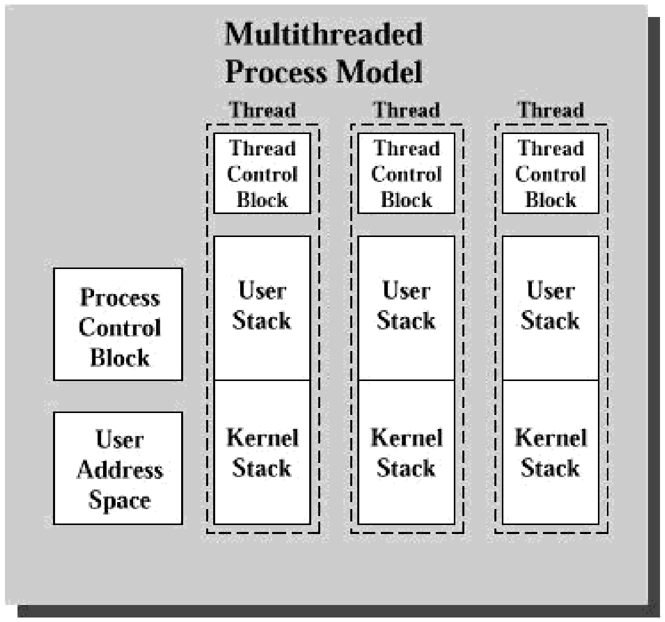
So, if you look at typically a process state transition diagram, this a generic diagram. A process is in ready state from ready it goes to executing state. If it needs data, so it goes to the waiting stage and then again, it can come back to the ready state before it gets scheduled.
In fact, the OS actually schedules processes which are in ready state. So therefore, all such processes join what it commonly called a ready queue. And from the ready queue OS picks up the process which is to move into the executing state and, these process is also known as dispatching.

Multithreading****

- Operating Systems supports multiple threads of execution within a single process
- An executing process is divided into threads that run concurrently
- Thread - a dispatchable unit of work

There are many Operating Systems which also support tasks or concurrency beyond the level of processors. So, what we say supports multiple threads of execution within a single process itself. Because, what is a thread of execution? A thread execution is a sequence of instructions being executed. And many of the OS is support multiple such sequences of instructions to be managed concurrently with in the boundary of a process.
So under those conditions, and what happens, an executing process is divided into threads that run concurrently. And thread becomes dispatchable unit of work, what is dispatchable unit of work? A schedulable unit of work, what I told you is that, if something is a ready queue from the ready queue. When we put that into an executing state it becomes the process of dispatching.





So, if you looking to this model, then what we really find, for a process I have got a PCB or Process Control Block. And along with the process that is, what is associated and user address space that, the address space of the process. And in these address space of the process threads relief. Threads are concurrent sequence of instructions concurrent threads of control. So, you have got with each thread just like PCB a thread control block as well.
And, what is interesting and important to note is that, each thread can have its own local variables, because its following own sequence instruction sequence. So, for each method it invokes along a thread there will be local variables. There will be parameters pass to those methods. There be written addresses corresponding to those methods. So, each thread will have its own stack.
User stack each thread will have its own portion of the user stack. Now, apart from the user stack there will be also the kernel stack. Because, kernel stack what is kernel stack? Kernel Stack is a stack which is not really resident in the User Address space not strictly in the User Address space. It is in the address space which is being managed by the Operating Systems.
So, let us say if this thread is under execution and an interrupt occurs. So, the Interrupt Service Routine would be process outside the boundary of this process. So, when that transfer takes place. The information the current state of the thread has to be saved, where will it be straight? It would be saved in the kernel stack and not in the user stack. User stack will typically maintain the local variables parameters corresponding to precede your calls.

Threads in Embedded Applications

- Light weight processes or threads are ideal for embedded computing systems. Since these platforms typically run only of few concurrent programs
- Concurrency within programs becomes better managed using threads

In fact, light weight processes or threads are ideal for embedded computing systems. Since these platforms typically run only of few concurrent programs. So, the application level concurrency is expected to be handled more through

the threads. And concurrency within programs becomes better managed. Because you can understand also, why should it be better managed, because the moment I switch from one thread to another, what happens, the amount of information that I need to change is much less.
Because my PCB, there is Program Control Block is remaining same. And that is why switching between thread to thread is much faster compare to switching between processors. And that is the reason why, you will find all these OSes which are targeted for Embedded Applications do support multithreaded processors.

Context Switch*****

- The CPU's replacement of the currently running task with a new one is called a "context switch"
- Simply saves the old context and "restore" the new one
- Actions:
    - current task is interrupted
    - Processor registers for that particular task are saved in a task specific table
    - task is placed on the ready list to avoid the next time slice
    - task control blocks stores memory usage priority levels, etc.
    - new task's registers and status are loaded into the processor
    - new task starts to run
        - involve changing the stack pointer the PC as well as program status register

Obviously, related to all these things is context switch. So, what is context switch? Context switch is CPU's replacement of the currently running task with a new one. Now here, you need to save the old context. And restore the new one and so the actions, what are involve in a context switcher current task is interrupted.
Processor registers for that particular task are saved in a task specific table. In fact, this information is typically will be in PCB. In fact, PCB has to be saved in a task specific table. And the task is placed on the ready list to avoid the next time slice. So, slice; that means, next time when the OS schedules the task.
So, task control blocks stores memory usage priority levels all these information. And new task's registers and status are loaded into the processor wire from they are loaded. From the PCB of the process which is currently being dispatched. And then, new task starts to run. And all these things involve changing the stack pointer the PC as well as program status register. So, you can understand that all these context switch; obviously, involves and overhead in terms of the instruction cycles which are involved for this job.

Process Scheduling****

Now at context switch the OS routine schedules the things. So, what is the scheduler?

- Scheduler is that part of the Operating System that decides which process a task to run next.
- It uses a scheduling algorithm that enforces some kind of a policy that is design to meet some criteria.

Scheduling

- Criteria may vary
    - CPU utilization – keep the CPU as busy as possible
    - Throughput – maximize the number of processes completed per unit time
    - Turnaround time – minimize a process' latency (run time), i.e., the time between task submission and termination
    - Response time – minimize the wait time for interactive processes
    - Real time – must meet specific deadlines to prevent "bad things" from happening

Now, this criteria may vary one of the most important criteria would be CPU utilization. Get the CPU as busy as possible. If a process is doing an active it is waiting and currently no process are running really on the CPU. Then, there is a CPU is wasting time.
Throughput is maximize the number of processes completed per unit time. Turnaround time minimize a process's latency that is the time between task submission and termination.
Now, even if you maximize the throughput the turnaround time may not be minimized. Because, it is depends on how the process have been tilt with response time is minimize the wait time for interactive processes. And real time would be that specific dead lines must be met. We are already talked about hard and soft deadlines. And those must be met.

So, when we are looking at the real time. We may suffer from the problem of low throughput or may be less CPU utilization, in order to meet the real time constraints. So, all these criteria it is not that all these criteria's can be met. Simultaneously by following a policy, a policy determines which criteria to be met for the purpose of scheduling.

Scheduling Policies****

- First Come, First Serve (FCFS)
  - The first task that arrives at the request queue is executed first, the second task is executed second and so on.
  - FCFS can make the wait time for a process very long
- Shortest Job First: Schedule processes according to their run – times
  - Generally difficult toknow the run – time of the process

Standard scheduling a policy which you find in most of the computer systems. And the OSes are First Come First Serve. Then, you have got the shortest job first schedule processors according to the run times; obviously, this is extremely difficult to do because you really do not know the runtime of the process occur.

Priority scheduling

- Shortest job first is a special case of priority scheduling
- Priority scheduling assigns a priority to each process and those with higher priorities are run first.

And this whole thing is based upon also, what is called priority scheduling? Your shortest job first is a special case of priority scheduling. It means process can have associated with priorities. And the higher priority process will be schedule, if it is in the ready queue. So, priority scheduling assigns a priority to each process and those with higher priorities are run fast. But, obviously, these kind of a scheme scheduling polices cannot meet our real time requirements.
So, Real Time Scheduling would be different and scheduling policies would be different. In fact, the most fundamental characteristics of a real time OS is the scheduling policies that the OS follows.

Characteristics of Real Time Systems

- Event – driven, reactive
- High cost of failure
- Concurrency and multiprogramming
- Stand – alone and continuous operation
- Reliability and fault tolerance requirement
- Predictable behavior

To meet the deadlines, what are the characteristics of real time systems? Let us have a quick review Event driven reactive. There is a high cost of failure, we must meet the deadlines. There would be concurrency and multiprogramming, because there are multiple concurrent streams in the external world. That is also stand alone continuous operation without any kind of user interventions.
Reliability and fault tolerance requirement, I would like that to run reliable. So, I am also predictable behavior. We should not because it depending on the system load, the scheduling of the processors can vary in a general purpose OS. But, I would to like to have for a real time OS a predictable behavior. The processor should get scheduled predictably, so that the deadlines can be met.

Periodic Sporadic, Aperiodic Tasks

- Periodic tasks:
  - we associate a period Pi with each task Ti
  - Pi is the interval between the job releases
- Sporadic and Aperiodic tasks which are tasks released at arbitrary times
  - Sporadic: has a hard deadline.
  - Aperiodic: has no deadline or a soft deadline.

Then, the tasks can be periodic. So, we associate a period Pi with each task Ti and Pi is the interval between the job releases. Sporadic and Aperiodic tasks which are tasks released at arbitrary times. And we distinguish between them, because a Sporadic has got a hard deadline. And Aperiodic typically has no deadline or a pretty soft deadline. So, this is an example of a periodic task.

Scheduling Real-time Tasks****

- Scheduling is the ability of tasks to meet all the hard deadlines
- Latency is the worst case system response time to events
- Stability in overload means the system meets critical deadlines even if all deadlines cannot be met

Now, when you schedule a real-time task and important issue becomes schedule ability. It is the ability of the task to meet deadlines or not. Because, it is not that all task can be admitted to meet the deadlines. Latency is the worst case system time to events. And stability in overload means the system meets critical deadlines even if all deadlines cannot be met.
So, there is a distinction and you meet those deadlines even when load on the system is high. Now, these are the parameters by which you can actually evaluate the scheduling policies. And schedulability is an issue at on the basis of which, you will decide add on OS decides whether to admit a task or not.

Scheduling policies

- Cyclic executive – where all work (tasks) are fit into a common period (major frame) and executed non – preemptively.
- Rate monotonic – preemptive scheduling where tasks are assigned priority based on rates, or periods
- Deadline monotonic – preemptive execution, where tasks are assigned priorities based on fixed deadlines
- Earliest deadline first – Preemptive scheduling, where tasks are assigned priorities dynamically, based on closeness of current deadlines
- Least laxity first – preemptive scheduling, where tasks are assigned priorities dynamically, based on the amount of slack time that remains between time needed to complete work and time to the deadline depending on that gap you decide which task to be scheduled.

What are different scheduling policies? We shall look at these policies in more detail in subsequent classes; we are just introducing the policies today. Cyclic executive, where all work tasks are fit into a common period and executed in a non preemptive fashion if there are multiple periodic task. If you can compute LCM of this periodic tasks find that period. And then, schedule all these task together in that period. So that execute in a non preemptive fashion we get a cyclic executive policy.
Rate monotonic is a preemptive scheduling by tasks is assigned priority based on rates or periods. In many cases preemptive can actually mean that when a task arrives with a higher priority, I may interrupt currently executing task even when the time slice has not expired. Deadline monotonic is another preemptive execution, where tasks are assigned priorities based on fixed deadlines. In fact, variation of these is Earliest deadline first or EDF.

Criteria for real – time scheduling policies****

- Achieve high utilization and still guarantee hard deadlines
- Provide fast a periodic response and still guarantee hard deadlines
- Low overhead: fixed versus dynamic priorities;
    - Dynamic policies tend to have higher overhead, since they may reorder the run queue at each scheduling decision

And all these scheduling policies, because the issue comes up is which scheduling policy it will select. So, you would like to have high utilization and still guarantee hard deadlines. Now, there can be a conflict on these criteria, because if you are trying to meet the deadline you may not able to admit jobs to increase your CPU utilization. CPU may be waiting for a while, so that you can meet the deadlines of periodic task or even for a periodic task with deadlines.

So, the other objective if comes in is provide fast a periodic response and still guarantee hard deadlines. This is the most difficult thing, because you have might of schedule the task. Now, suddenly and a periodic task arrives and you need to give a first response time. Then, there should be low overhead because you scheduling should not consume lots of time, because it is a fundamental problem. Because, if the scheduling itself consumes time. Then, you will be missing deadline because of you scheduling policy. So, you cannot have a very complex scheduling mechanism. So, what we say the dynamic policies tend to have higher overhead and since they may reorder the run queue at each scheduling decision. So, each time a process arrives and you need to reschedule the processors. The moment you are rescheduling the processors, what happens, there is a time spent.

Because, you have to reorder the queue reordering the queue at least means manipulation of the data structure. Because, in that queue you have kept your PCB's, so that data structure has to be manipulated. So that introduces overhead, you also have an overhead, because of calculation of the policy that you need to do. So, these give us a very basic introduction to OS, OS targeted for Embedded Systems. And, what we should understand is that, this scheduling in terms of the real time constraints is the key problem. And, what we have looked at? We have looked at the basic policies. We need to analyze these policies to understand the implications and the analysis we shall do in the next topics.

Question: What if a process cannot meet the deadline?
So, the question is whether if that a process does not meet a deadline, what shall I do with the process. See if you remember we talked about schedulability. Now, when we are admitting a process a OS can check for it schedulability. If it can be scheduled, then only the process should be admitted. Otherwise I need not admit that process; that means that process does not join the ready queue. So, it will be in a kind of a suspended state.

Embedded – OS

We have so far discussed the basic features of an operating system which are intended to be used in the context of embedded appliances. Now, we shall recap at these features in the context of the complete systems or complete OS packages which are available for deployment on embedded appliances.

Configurability

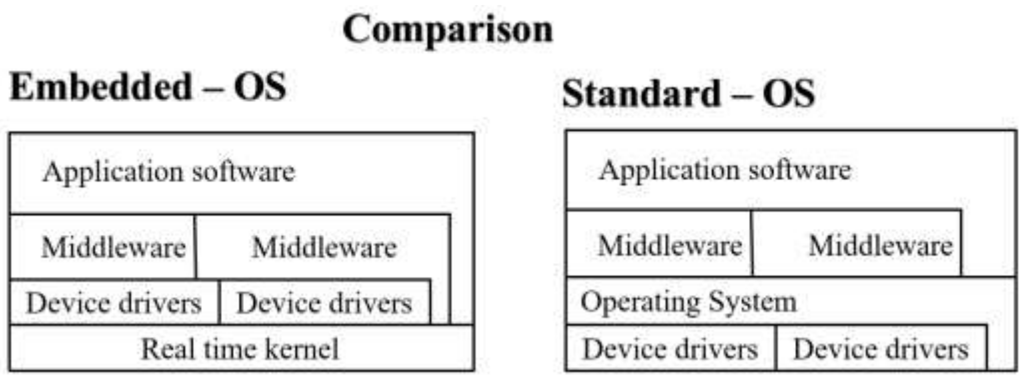
- No single RTOS will fit all needs
    - Configurability needed: no overhead for unused functions tolerated
        - Simplest form remove unused functions
        - Conditional compilation
        - Dynamic data might be replaced by static data
        - Advanced compile – time evaluation useful
        - Object – orientation could lead to a derivation of subclasses
- Testing and Verification of derived OS is a potential problem

The basic issue is that of configurability. Given an appliance, no available package will fit all the needs. And we need configurability, because, no overhead for unused functions be tolerated, because you would not like memory to be used for such purposes. The simplest form remove unused functions and many cases you actually have the feature to compile your OS with applications together. So, we can use conditional compilation to illuminate those modules which are not needed. In a way, such OS are organized in modular approach that means, OS consist of a set of modules, and you make the selection of the modules depending on your requirement. And the advance compile time evaluation is useful for optimizing the whole system and object orientation can lead to derivation of sub classes specialization on the basic functions provided by the core OS. But obviously, testing and verification of the derived OS is a potential problem. So, the whole idea is what that you have an OS a generic version of an OS and you tailor it for your suitable applications depending on the functions that you need. As well as you have to specify if required the features of the hardware on which the OS is expected to run. So, configurability is a very important characteristic of such an OS package. The other issue is management disc and network.

Disc and Network

Not all embedded systems have disc and they are all not connected on the network. But general purpose computer systems variably have OS and network connectivity. So, these 2 aspects are always needed as part of the general purpose operating system. But that is not necessarily true with an embedded system. So, what we say? The effectively no device that needs to be supported by the all versions of the OS except may be the system timer, because critically system timer has to be there to manage various aspects of operation of an embedded system. But other devices are actually depends on what kind of application you are looking at. So, in fact, that is again another reason for configurability and choice of the appropriate device drivers to be made part of the OS. In fact, if you really have discs and networks that can be handled by the tasks, this doesn't form part of the OS kernel and the device driver then comes bundled with the OS. Relatively slow discs & networks can be handled by tasks.

Embedded OS vs Standard OS:****



So, if we compare the basic idea wise embedded OS and standard OS, we have the operating system along with you have got device drivers for a multiple applications in standard OS. In embedded OS, you make choice of the device drivers depending on the devices that you would like to use. And very likely there would be a real time kernel, because it needs to support real time tasks, the kernel has to be a real time kernel which is not a necessity for a standard OS. And in many cases you will be using a standard OS to compile. An OS along with applications bundled in to a package to be downloaded on to an embedded appliance.

Real-time capability

The other point which is fundamental concern for an embedded appliance and a system is a real time capability. So, many embedded systems are real time systems and hence the OS used in these systems must be real time operating systems (RTOSes).

Real-time Operating Systems****

What is the real time operating system?

A real time operating system is operating system that supports the construction of real time systems. And we have already had the definitions of real time systems and we know the difference between soft real time and hard real time systems.

What are the key requirements?

The basic key requirement is timing behavior of the OS must be predictable.

- We have talked about the scheduling strategy but we have to also consider the services that OS provides. Each service has got an overhead, in fact, scheduling services has also got an overhead and we have considered whole scheduling scheme without considering the overhead.
- But in actual real world deployment, this overheads play a crucial role. And we need to have exact timing and predictable timing of the OS services.
- So, for all services of the OS, we should know the upper bound on the execution time. In that sense, RTOSs must be deterministic

Other thing is there should be short times during which interrupts are disabled, because interrupts are disable for a long time then you cannot actually guarantee meeting of the service constrains of the devices for external sensors. Because external sensors, when they are providing an interrupt, it is required that the devices to be serviced with in a deadline. If you actually disable an interrupt for a long time period then interrupt will not be acknowledge and such services cannot be provided. In fact, there has been standards for this kind of real time operating systems.

Real-time Operating System Standards****

- IEEE 1003.1b POSIX Real-Time Extensions (www.ieee.org)
- OSEK (automotive real-time OS standard) (www.osek.org)

POSIX is an IEEE standard and in fact, if I have my own OS, I would like it to be POSIX compatible so that my applications which are POSIX compliant can run on my OS. And in fact, that is the basic motivation for defining this kind of standards. Because actual implementation can vary from system to system, OS to OS. But the basic features can be standardized then your application can run on a variety of operating systems.

The other thing is OSEK this is automotive real time OS standard. This is particularly meant for automotive applications.

So, when we are making a choice of the OS we would like it to be POSIX compliant as OSEK is only for automotive purpose. And when you develop an application you will be using the system calls which should be ideally POSIX compliant so that they can move from one OS to another OS; that means portability across the OS without much of rework required.

Real-time OS: Example****

- Windows platforms: Embedded XP, Windows CE, Pocket Windows
- VxWorks from Wind River system (www.windriver.com)
- Linux variants:
  - (Embedded) Red Hat Linux (www.redhat.com)
  - FSM RT-Linux (www.mvista.com)
  - Monta Vista Linux (www.mvista.com)
- QNX (www.qnx.com)

There are number of such examples which are available and today being used windows platform are primarily sourced from Microsoft, targeted for handle devices. VxWorks from Wind River system is the most well-known commercial operating system used in embedded appliances. There are variety of Linux variants therefore, Linux now is turning out to be the most popular in fact, on an upward swim for its application in embedded domain.

Real-time OS: Requirements

- OS must manage the timing and scheduling
  - OS possibly has to be aware of task deadlines; (unless scheduling is done off-line).
  - OS must provide precise time services with high resolution.

Now, what are the further requirements? The OS must manage the timing and scheduling. If all the OS services had the deadline then all the OSs should provide the services for timing and scheduling. And OS has to aware of task deadlines unless scheduling is done off line; that means, if you are using a table driven scheduling and design time scheduling then the OS need not bother about that deadlines. But otherwise it has to bother about deadlines to decide about the priority of the tasks. And OS must provide precise time services with high resolution.
You can have specificity of the deadlines, because you have a timer and the system may be operating at Giga Hertz, timer has a specific counts and OS has got the internal structure to maintain the timing information. So, for example, I may have a gigahertz clock, but I have implemented an OS which uses a simple four bit to store the timing information internally. In that case I cannot use time resolution of gigahertz level for dealing with

the deadlines. So, time resolution in terms of your internal management becomes a key issue for managing the schedules. And this is related to your underlined hardware as well. So, what we say the time places a central role in real time application.

Time Services

- Time places a central role in "real-time" systems
  - Actual time is described by real numbers
- Two discrete standards are used in real time equipment
  - International automatic time (TAI): Free of any artificial artifacts
  - Universal time coordinator (UTC): UTC is defined by astronomical standards

Time Service is also related to your actual time. So, actual time is normally described by real numbers. And if you see, depending on what is the data structure allowed, that is how many bytes being allowed for storing thee real numbers, would actually determine the timing resolution. And there are two discrete standards used in real time equipment. These are making the references with respect to the physical time. One is international automatic time TAI and universal time coordinator that is UTC. This is for a universalization of the physical time. When you are using in an embedded system which may be network and working with other system located at different time OS, just consider that when you making a mobile phone call in the time zone. If there is a tariff depending on the time period the time zone can actually vary from country to country. So, there has to be timing information associated with the call. So, that way also the physical times becomes an important parameter. So, how the synchronization does is done with respect to the physical time? In fact, many of the embedded systems and the OS do have features to synchronize with physical time.

Synchronization

- Internal synchronization
  - Synchronization with one master clock
    - Typically used in startup phases
  - Distributed synchronization:
    - Collect information from neighbors
- External synchronization
  - External synchronization guarantees consistency with actual physical time.
  - Recent trend is to use GPS for ext. synchronization

One is internal synchronization in fact; this may or may not correspond to synchronization of the physical time. But this would definitely correspond to synchronization with respect to a time stamp which is consistent across systems which are dealing with the time stamped data. So, there can be synchronization with respect to one master clock and this is typically used in startup phases.

And there can be a distributed synchronization by collecting information from other neighbors and trying to offset the errors with respect to the timing information available from the neighbors. So, it something like that I look at your watch and as well as you friends watch and try to find out the best time. And all of us all few of us can synchronize our watch with respect to the joint agreement of a time. The other issue is that of external synchronization. External synchronization guarantees consistency with actual physical time in fact, across multiple time zones as well. In now a days, to use GPS for external synchronization, global positioning system do communicate with the timing information also and it can be of the order of 100 nano seconds the time resolution using the physical time synchronization system. In fact, physical synchronization of time is a very important issue with these kind of embedded communication appliances which needs to manage a variety of tariff structure.

Other RTOS features

- Utilities
  - Bootstrapping support
  - "Headless" operation

        - Display not necessary
- APIs (Application Programming Interfaces)
    - Multiple Threads and/or processes
        - Fixed priority scheduling is most popular
    - Mutex/semaphore support likely with priority inheritance support

Other RTOS features is bootstrapping support, headless operation, they really not have a display to show the status. In many cases you have some kind of, if you want to flag an error. Some kind of external hardware interfaces to be provided, not really a display but you have API's. Now, when you develop an application with reference to an OS you will be using this API's. So, there will be multiple threads or processes in fact, API provides you with the facility to create threads as well as processes. And they will provide support for semaphores with priority inheritance. And we have discussed and we have seen why priority inheritance is so, very important when you were scheduling real time tasks. So, typically they come with priority inheritance and the semaphore support and ability to create multiple threads with fix priority scheduling.

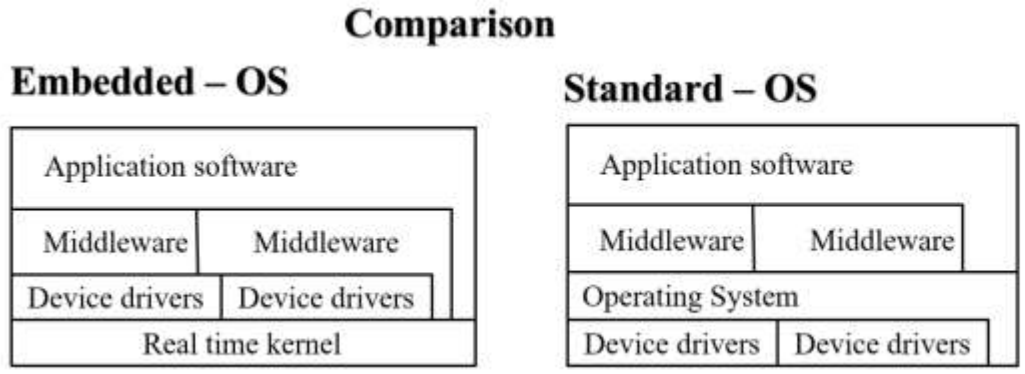
More RTOS Features:

- Inter process communication
    - message queues
- Timers/clock
- Device drivers
- Optional
    - Network protocol stack
    - Graphics support

Also there are features for inter process communication. The most common thing is message queues. If there are two processes next to communicate, how do they do it? So, you can think in terms of a message queue, a standard producer consumer kind of a scenario. So, the producer puts in message from 1 end to the message queue and the consumer removes the massager from the other end of the message queue. It is a kind of a buffer it can consider a simplest implantation of these in terms of a buffer and so, it is a bounded buffer. So, if the buffer is empty; obviously, consumer is stopped from accessing it and when the buffer is full, producer is stopped from writing on to it.
But there could be various other ways also to implement message queues. In fact, it can be a complete asynchronous implementation where the messages are posted, messages are received and the process continuous its own execution. And may come back at an appropriate point in time to check the message queue and take appropriate action. Timers and clock management, in fact, is a very key features, it has to be definitely there. There may be a set of device drivers available and your network protocols stack and graphic support are optional depending on what kind of service you are really trying to provide.

RTOS Kernels

- Distinction between Real time kernels and modified kernels of standard OS'es



Device drivers:

- In Embedded OS device drivers handled by tasks instead of hidden integrated drivers
- Improve predictability; everything goes through scheduler
- Effectively no device that needs to be supported by all versions of the OS, except maybe the system timer.

Interrupts can be employed by any process

- For standard OS: this would be serious source of unreliability. But embedded programs can be considered to be tested.
- It is possible to let interrupts directly start or stop tasks (by storing the tasks start address in the interrupt table). More efficient and predictable than going through OS interfaces and services.
- However, composability suffers: if a specific task is connected to some interrupt, it may be difficult to add another task which also needs to be started by the same event.
- If real-time processing is of concern, time to handle interrupts need to be considered. For example, interrupts may be handled by the scheduler.

Protection mechanisms are not always necessary:

- Embedded systems are typically designed for a single purpose, untested programs rarely loaded, software considered reliable.
- *Privileged* I/O instructions not necessary and tasks can do their own I/O.
- Example: Let **switch** be the address of some switch. Simply use **load, register, and switch** instead of a call to the underlying operating system.
- However, protection mechanisms may be needed for safety and security reasons.

Functionality of RTOS Kernels****

- Resource management
  - Processor management
  - Memory management
- And timer management
- Task management (resume, wait etc.)
- Inter task communication and synchronization

So, what are the functions of the real time kernel? Obviously, the processor management, memory management and timer management. Among the devices, timer is a most important device and it has to be universally managed. Other devices are actually picked up depending on when they are available or not. The other thing is task management that is resume, wait, start etc. and inter task communication and synchronization for sheared resources. So, what we shall see is that with reference to Linux how does this additional kernel features are provided.

Case Study: Linux in embedded systems

There are various strategies which have been followed to enhance Linux kernels to provide the important features. In fact, that gives you what I already said, that two basic approaches for writing a real time kernel are modifying an existing kernel for enabling the services.

Why Linux in Embedded Systems?****

- Reliable, Full-featured OS
  - Rich multi-tasking support
  - Security, Protection
  - Networking Support
    - TCP/IP, RSVP, SIP, MPLS, H.323
  - Multimedia support
    - JPEG, MPEG, GSM

        - Device Drivers
- Standard, known Environment and API's
    - Unix Lineage
        - Familiar environment for many users/developers
    - POSIX Compliance
- The Cost Factor
    - Free runtime royalties
- The Open Source Factor
    - A global team of programmers enhancing the environment literally all the time
    - Availability of libraries, tools, and device drivers
    - Source Code Access allowing "peeking inside the hood" (and customizing as necessary)
- The Popularity Factor
    - Excellent documentation
- Small Embedded Systems
    - Modular Kernel, possible to configure the kernel to suitable size
    - Customizable Root File System
    - Lots of Utilities
- High-End Embedded Systems
    - High Availability
    - Clustering
    - SMP Support

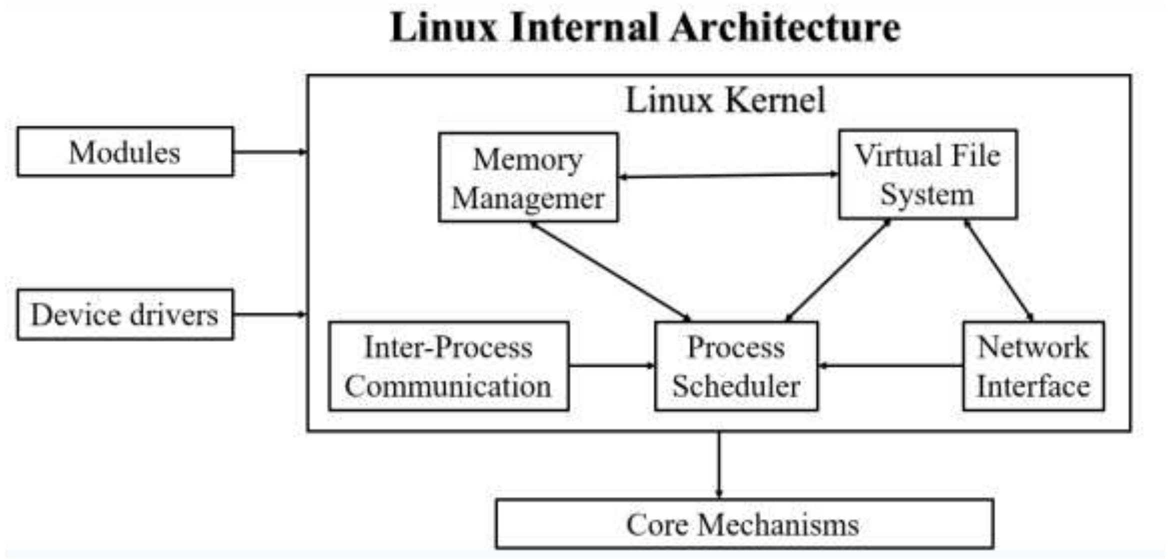
So, why Linux? The basic idea why Linux is for the use in embedded systems. It has got a rich multi-tasking support security and protection, it was also got a networking support although I say networking may not be a key issue. But many ways networking today many of the embedded systems are coming with networking features, multimedia support and variety of device drivers which are available on Linux and standard known environment an APIs. So, the familiar environment for many users and developers as well as POSIX compliances. So, all these features make Linux a popular choice.

Linux API: Tasking

- Process
    - Encapsulates a thread of control and an address space
        - Address space may be share giving threads in effect
    - Schedulable Entity
- Threads
    - Are processes to the Linux kernel
        - Scheduled by the Linux kernel
- Can be created such that they share the address space with the parent process, effectively giving threads

So, if you got the Linux API, the most important aspect of the API is the tasking and here what happens? A processes encapsulates a thread of control and an address space. Address space may be shared giving threads in effect i.e., there are multiple threads within the processes boundary and processes schedulable entity. In fact, threads in a way are processes to the Linux kernel and they are scheduled by the Linux kernel. And can be created such that they share the address space with the parent process effectively giving threads. So, what we are talking about is two ways of leading to threads. So, I can talk about processes the schedulable entity is and process can be consisting of a set of threads which share the address space. So, threads now, become a schedulable entities and a set of threads group together as a process so that the shear a common address space. So, this can be done for your application using standard Linux API.

Linux Internal Architecture:****



So, if you look at typically a Linux internal architecture, a key issue as process scheduler. You can have multimedia support, you can have a inter process communication, networking support and there is a link to the core mechanisms. Now, what you do? You would select the modules which are available in terms of the modular architecture, device drivers which are available together and construct this architecture. Because process scheduler would manage all these processes and, because they are of the different nature and they are linked to process scheduler. Depending on what kind of tasks do you have, you will have the corresponding module configured for your particular version.

Linux Internal: Scheduling

- ➢ Schedulable Entities
  - ❖ Process
    - ▪ Real-Time Class SCHED_FIFO or SCHED_RR
    - ▪ Time-Sharing Class: SCHED_OTHER
  - ❖ Real-Time processes have
    - ▪ Application defined priority
    - ▪ Higher priority than time-sharing processes
- ➢ Non Schedulable Entities
  - ❖ Interrupt Handlers
    - ▪ Have priorities, and can be nested

So, what are schedulable entities? I already talked about processes, threads are also processes. You can have what is called two distinct classes' real time class or time sharing class. When you talk about a time sharing class obviously, what we are implying is that for this task there are no deadlines to be met. Application processor can have application defined priority as well as higher priority than time sharing processes. And what are non schedulable entities? It would have interrupts and interrupt handlers can have priority and can be nested. But they are not schedulable. Why they are not schedulable?

In the sense, when such interrupt arrives the interrupt has to be serviced. Priority between the interrupts means either how they are to be resolve when simultaneously more than one interrupt is pending ard when a particular interrupt is serviced whether you would accommodate another interrupt or not. So, that organization can be done through the OS as well, because OS has to disable interrupts. If there are maskable interrupts, OS will disable those interrupts while a particular interrupt is being serviced. So, that is again a configurable aspect, but there are some problems; like timer granularity.

Linux and Real-Time: Problems

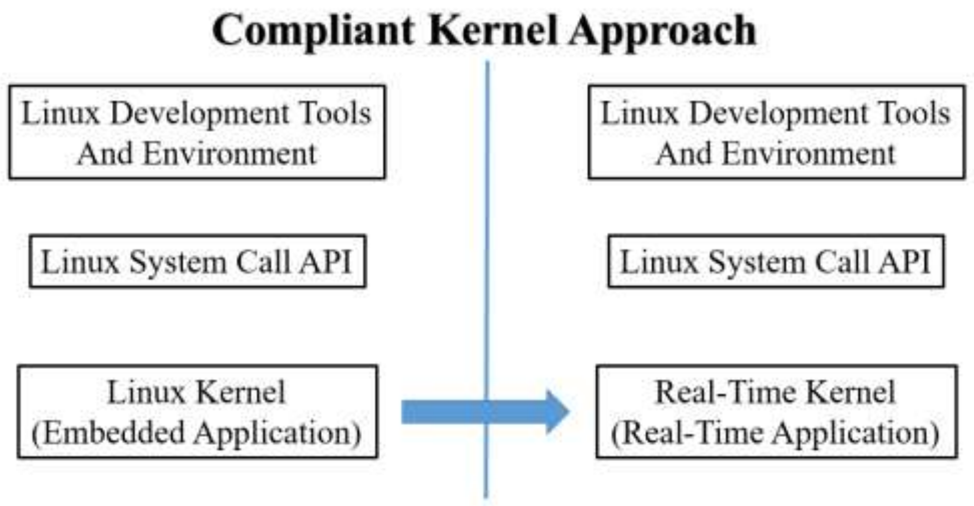
- Timer Granularity
    - Many real-time tasks are driven by timer interrupts
    - In Standard Linux, the timer is set to expire at 10msec intervals
- Scheduler Predictability
    - Linux scheduler keeps tasks in a unsorted list
    - Requires a scan of all the tasks to make a scheduling decision
    - Scales poorly as number of tasks increases
- Various subsystems NOT designed for real-time use
    - Network protocol stack
    - File system
    - Windows manager

In standard Linux the timer is set to expire at 10 millisecond intervals, because this is the smallest interval that you can really have depending on the way it has been configured. So, the granularity can become problematic; that means, you cannot have the deadlines specified at a granularity lower than 10 millisecond level. Scheduler predictability - scheduler keeps tasks in an unsorted list. So, requires a scan of all tasks to make a scheduling decision. So, that has an overhead and it will scale purely as a number of tasks synchronize. If you have a more real time tasks then; obviously, have a real problems on various subsystems on design for a real timings. Obviously, network protocol stack file system windows manager required. You can say that they are needed also in various embedded subsystems. When they are not needed they really do not call as a problem. But when they are needed they do cause a problem.

Approached to Real-Time Linux

- Compliant Kernel Approach
- Dual Kernel Approach
    - Thin Kernel
- Core Kernel Approach
- Resource Kernel Approach

So, how do you modify or the basic issue is that how do you deal with the Linux kernel and bring in this kind of real time features. Currently, if you look at the different flavors is a Linux that is available approaches that have been followed at the following; compliant kernel approach, dual kernel approach, core kernel approach and resource kernel approach. So, these in fact, in a way are examples that how you architect an operating system. We have looked at features of an operating system, we have not really looked at how you architect an operating system. So, what we are looking at with reference to Linux how you architect such an operating system.


Compliant Kernel Approach
Linux Development Tools And Environment
Linux System Call API
Linux Kernel (Embedded Application)
Linux Development Tools And Environment
Linux System Call API
Real-Time Kernel (Real-Time Application)

So, a complaint kernel approach is based on this basic philosophy. You got a Linux kernel which all of us know has got this Linux development tools and environment which uses standard Linux system call API to develop an application, but you do not have a real time feature on this Linux kernel. So, what you do? You write a kernel which is a real time kernel, but it provides a same API as that of the original Linux API. So, that it gives your complaint kernel approach.

Compliant Kernel Approach

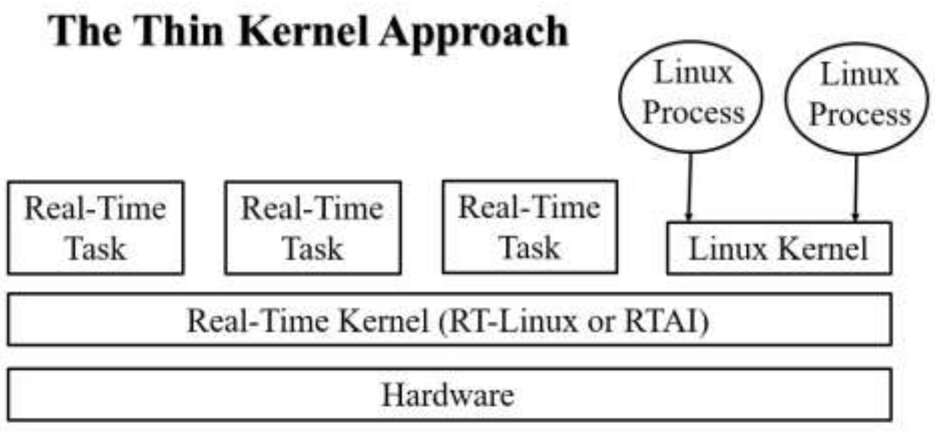
- Basic Feature
  - Linux is defined by its API and not by its internal implementation
  - The real-time kernel is a non Linux kernel
- Implications
  - No benefits from the Linux kernel
  - Not possible to benefit from the Linux kernel evolution
  - Not possible to use Linux hardware support
  - Not possible to use Linux device drivers

The basic philosophy the Linux is defined by its API and not by its internal implementation. For an application developer Linux is defined by an API and not by its internal implementation. So, the real time kernel is strictly unknown Linux kernel. You can provide your own kernel as such with an API same as that of the Linux kernel. So, in this case what are the implications? You really have no benefits from the Linux kernel, not possible to benefit from Linux kernel evaluation, Linux kernel changes over time, because it is evolving in a continuous basis. Not possible to use Linux hardware support, not possible to use Linux device drivers. So, these are the things which have to be developing as part of your complaint kernel approach.

Compliance

- 100% Linux API
  - Support all of Linux kernel API
- Implications
  - Any Linux application can run on real-time kernel
    - Development can be done on Linux host, with rich set of host tools for development
  - All Linux libraries are trivially available to run on real time kernel
    - Third party software
  - Achieving 100% Linux API is not trivial
    - Consider the amount of effort put on Linux kernel development

So, compliance is if you have a 100 percent Linux API they needed to support all of Linux kernel API and all of Linux kernel applications which are built up on the Linux kernel system calls. So, any Linux application therefore, can run on real time kernel, development can be done on Linux host with rich set of host tools for development. This is the basic advantage. All Linux libraries are trivially available to run on real time kernel. So, all third parties comes in, but achieving 100 percent Linux API is not trivial. So, consider the amount of effort put on Linux kernel development which has taken place in a generic fashion. So, this what we are trying to look at as a trade of between complaint kernel approach and what could be the alternative.



Real-time tasks do NOT use the Linux API or Linux facilities
Failure in any real-time task crashes the entire system

Alternative is a dual kernel approach and here what we say one of the dual kernel approach is the thin kernel approach. So, we got a real time kernel at Linux and your Linux kernel or classical Linux kernel becomes a task under real time kernel. So, user processes uses this Linux processors, they can be run on the Linux kernel and your real time tasks gets scheduled by the real time kernel. So, periodic tasks if you go back to a original model which does not have a real time. So, periodic tasks are scheduled by the Linux kernel. In fact, real time kernel schedules Linux kernel and Linux kernel in terms schedules user processes which can not necessarily real time processes. So, real time tasks do not use the Linux API or Linux facilities. Failure in any real time tasks may crashes the system, so, the whole development in the real time task may not be based on Linux API.

Core Kernel Approach

- Basic Ideas
  - Make the kernel more suitable for real-time
  - Ensure that the impact of changes is localized so that
    - Kernel upgrades can be easily incorporated
    - Kernel reliability and scalability is not compromised
- Mechanisms
  - Static Configuration
    - Can be configured at compile time
  - Dynamic Configuration
    - Using loadable kernel modules
- Allows the use of most, if not all existing Linux primitives, applications and tools
  - Need to avoid primitives that can take extended time in the kernel
- Allows the use of most existing device drivers written to support Linux
  - Need to avoid poorly written drivers that unfairly hog system resources
- Robustness and Reliability
  - Core kernel modifications can effect robustness, but source is available

The other approach is core kernel approach where you make the kernel more suitable for real time ensure that the impact of changes is localized so that kernel upgrades can be easily incorporated and kernel reliability and scalability is not compromised. So, there are two ways to do it, one is the static configuration, can be configured at compile time, other is dynamic configuration using loadable kernel modules that is in the run time you make some of the modules enabled. Now, in this case; obviously, what you assume is that you have a secondary storage where you have the stored information from which you can actually add on the modules on demand. So, the core kernel approach allows the use of most, if not all existing Linux primitive, applications and tools and allows use of most existing device drivers written to support Linux. But if you are trying to play with the core kernel then it can effect robustness, but in a way source is available, this still a possible way to architect the Linux system for a real time application.

Resource Kernel

- A kernel that provides to Applications Timely, Guaranteed, and Enforced access to System Resources
- Allows applications to specify only their resource demand, leaving the kernel to satisfy those demands using hidden management schemes

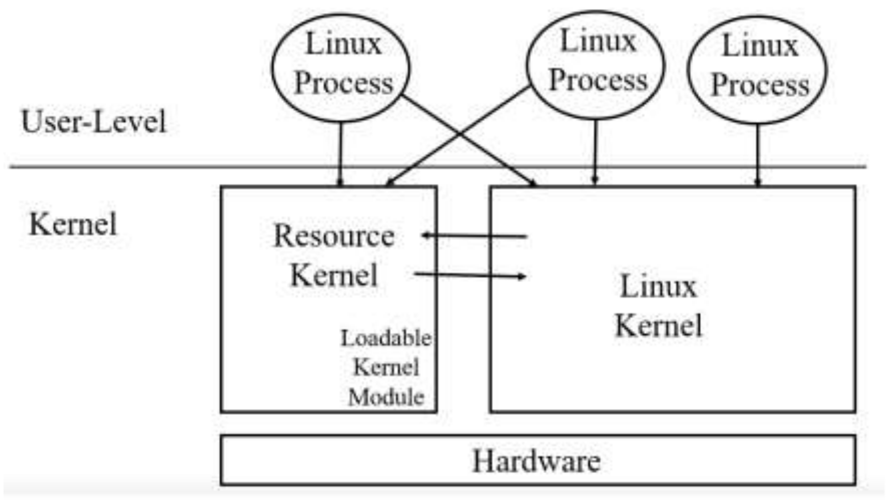
The other approach is that of the resource kernel. Here conceptually it is the distinct approach from that of what you had considered so far. But philosophically it was closer to that of the dual kernel approach. So, what you a say a resource kernel is a kernel that provides to applications timely guaranteed and enforced access to system resource allows applications to specify only their resource demand, leaving the kernel to satisfy those demands using hidden management schemes. In fact, this is architecturally much advanced version in a flexible version compare to others.

Protection in Resource Kernels

- Each application (or a group of collaborating applications) operates in a conceptual virtual machine:
    - A machine which consists of a well-defined and guaranteed portion of system resources
        - CPU capacity, disk bandwidth, network bandwidth and memory resource
- Multiple virtual machines can run simultaneously on a same physical machine
    - Guarantees available to each reserve set is valid despite the presence of other (potentially misbehaving) applications using other reserve sets

So, what is the production in resource kernels? Each application is a group of collaborating application operates in a conceptual virtual machine. A machine which consists of a well-defined and guaranteed portion of system resources that is CPU capacity, disk bandwidth, network bandwidth and memory resource multiple virtual machines can run simultaneously on a same physical machine. So, guarantees available to each reserve set is valid despite the presence of other potentially misbehaving applications using other reserve sets. So, what does that mean? Conceptually I am reserving the resources giving to a set of tasks so, effectively for that tasks with given set of resources allocated at, in a static fashion becomes the definition of the virtual machine.

Linux Resource Kernel Architecture



Linux process running on the Linux kernel can communicate for a resource kernel. So, resource kernel makes reservation of the resources for this Linux process with respect to the Linux kernel and theses resources are made available to the Linux processes. So, the Linux process runs with reference to a virtual machine. Each process has got a resource demand. So, resource kernel handles the resource demand. So, effectively it makes reservations of those resources with reference to Linux kernel. So, Linux kernel knows that this much resource has been already allocated. So, that gets allocated to the Linux processes. So, the Linux process which refers to resource to kernel operates effectively in a virtual machine.

Reserves and Resource Sets

- Reserve
    - A Share of a Single Resource
    - Temporal Reserves
        - Parameters declare Portion and Timeframe of Reserve Usage
            - E.g.. CPU time, link bandwidth, disk bandwidth
    - Spatial Reserves
        - Amount of space
            - E.g., memory pages, network buffers
- Resource Set
    - A set of resource reserves

So, these are related to two basic concepts which we talk about reserve. Reserve is a share of a single resource there can be temporal reserves there can be spatial reserves. Temporal reserves are parameters which declare portion and time frame of resource usage; CPU time, link bandwidth, disk bandwidth is that effectively a temporal resources. Spatial reserves amount of space, it can be memory pages, network buffers. So, everything gets allocated to a process by resource kernel and resource set is a set of resource reserves.

Summary: We have looked at features of embedded OS

We have examined the basic features of the OS which is targeted for this kind of the application, how to architect such a system. The basic question is that time resolution. So, see you have the timers can be set to generate interrupt at fixed intervals. Now, these intervals are real times and the real time has to be internally represented by the OS. Now, OS allocate the fixed word size to represent the time then the resolution is fixed. So, if you have the granularity of the time, depends on how the OS manages that time data structure. It also is related to the underlining hardware, because if your timer cannot support a variety of resolution or a very fine resolution depending on the clock at which it is operating.