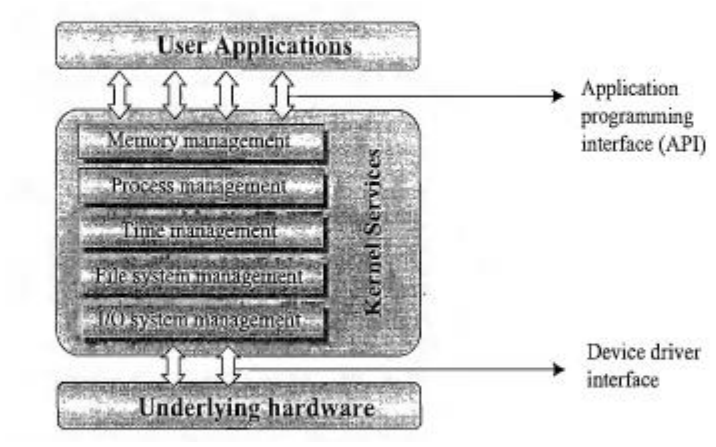
Operating System Services



The kernel is the core of the operating system and is responsible for managing the system resources and the communication among the hardware and other system services. Kernel acts as the abstraction layer between system resources and user applications. Kernel contains a set of system libraries and services. For a general purpose OS, the kernel contains different services for handling the following.

1) Process Management deals with managing the processes/tasks. Process management includes

- Setting up the memory space for the process
- Loading the process's code into the memory space
- Allocating system resources
- Scheduling and managing the execution of the process,
- Setting up and managing the Process Control Block (PCB)
- Inter Process Communication and synchronization
- Process termination/deletion

2) Memory Management includes

- Keeping track of which part of the memory area is currently used by which process
- Allocating and De-allocating memory space on a need basis (Dynamic memory allocation).

3) File System Management service of Kernel is responsible for

- The creation, deletion and alteration of files
- Creation, deletion and alteration of directories

- Saving of files in the secondary storage memory (e.g. Hard disk storage)
- Providing automatic allocation of file space based on the amount of free space available
- Providing a flexible naming convention for the files

4) I/O System (Device) Management Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. In a well-structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel. The kernel maintains a list of all the I/O devices of the system. This list may be available in advance, at the time of building the kernel. Some kernels, dynamically updates the list of available devices as and when a new device is installed. The service Device Manager of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service, called device drivers. The device drivers are specific to a device or a class of devices.

5) Secondary Storage Management deals with managing the secondary storage memory devices, if any, connected to the system. The secondary storage management service of kernel deals with

- Disk storage allocation
- Disk scheduling (Time interval at which the disk is activated to backup data)
- Free Disk space management

6) Protection Systems: Most of the modem operating systems are designed in such a way to support multiple users with different levels of access permissions. Protection deals with implementing the security policies to restrict the access to both user and system resources by different applications or processes or users.

User and Supervisory mode

At every clock tick of the system-clock, there is system interrupt. On each system interrupt, the system time updates, the system context switches to the supervisory mode from user mode. After completing the supervisory functions in the OS, the system context switches back to user mode.

User Mode:

User process is permitted to run and use only a subset of functions and instructions in OS. Use of OS functions in user mode is either by sending a message to the waiting process associated in the OS kernel space or by system call (calling an OS function). The use of hardware resources including memory is not permitted without OS making the call to the OS functions, called system call. User function call, which is not a system call, is not permitted to read and write into the protected memory allotted to the OS functions, data, stack and heap. That protected memory space is also called kernel space. Hence execution of user functions calls is slower than the execution of the OS functions (on system call) due to need to spend time in first checking the access permission to the protected space.

Supervisory mode/ Kernel Mode

It is also called kernel mode OS runs in protected mode the privilegedλ functions and instructions in protected mode that are the privileged ones and the OS (more specifically, the kernel) is only one permitted to access the hardware resources and protected area memory Kernel space functions and processes execute faster than the user space functions and processes. Only a system call is permitted to read and write into the protected memory allotted to the OS functions, data, stack and heap.

Normally all the threads run in the supervisory mode (kernel mode). Therefore, the threads executes fast. It improves the system performance. If the threads are to execute in user mode, as in Unix or in non-real time OS, then the execution slows down due to time spent in checks on the code access to the protected kernel space.

Operating System Structure

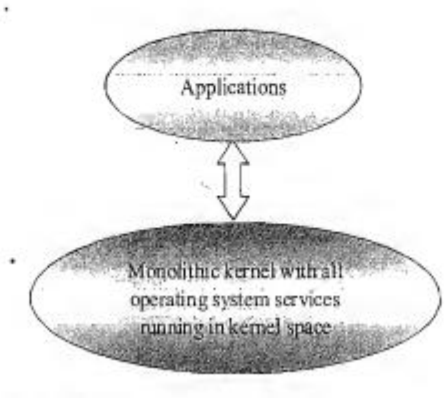
Top to down Structural Layers are as follows

- Software Application Programming Interface (API)
- System software other than the one provided at the OS
- Operating System Interface
- Operating System
- Hardware–OS Interface
- Hardware

OS is the middle in-between layer between the application software and system hardware.

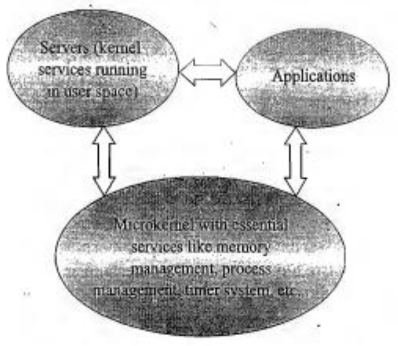
Monolithic Kernel

In monolithic kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread. The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application. LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel. The architecture representation of a monolithic kernel is given in Figure.


Applications
Monolithic kernel with all operating system services running in kernel space

Microkernel

The microkernel design incorporates only the essential set of Operating System services into the kernel. The rest of the Operating System services are implemented in programs known as 'servers' which runs in user space. This provides a highly modular design and OS-neutral abstraction to the kernel. Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. Mach, QNX, Minix 3 kernels are examples for microkernel. The architecture representation of a microkernel is shown in Figure


Servers (kernel services running in user space)
Applications
Microkernel with essential services like memory management, process management, timer system, etc.

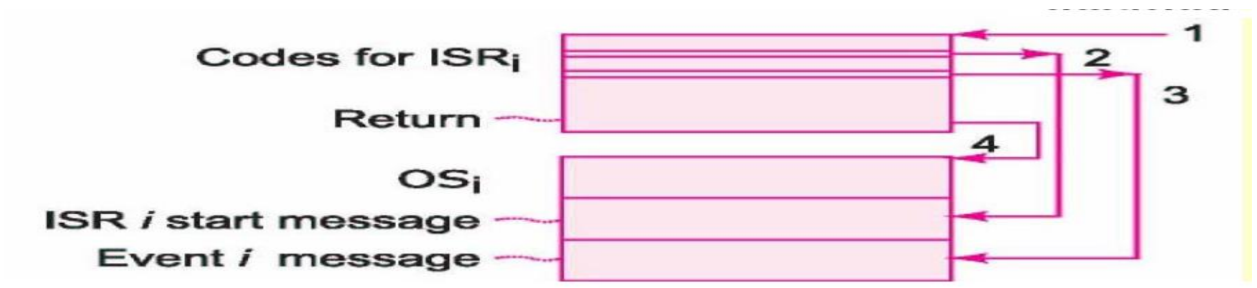
Microkernel based design approach offers the following benefits

• Robustness: If a problem is encountered in any of the services, which runs as 'server' application, the same can be reconfigured and restarted without the need for restarting the entire OS. Thus, this approach is highly useful for systems, which demands high 'availability'.
• Configurability: Any services, which run as 'Server' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

Interrupt Handling in RTOS

Interrupt Service Routines (ISRs) have the higher priorities over the RTOS functions and the tasks. An ISR should not wait for a semaphore, mailbox message or queue message An ISR should not also wait for mutex else it has to wait for other critical section code to finish before the critical codes in the ISR can run. Only the IPC accept function for these events (semaphore, mailbox, queue) can be used, not the post function.
Three alternative ways systems to respond to hardware source calls from the interrupts

1. **Direct Call to an ISR by an Interrupting Source and ISR sending an ISR enter message to OS**



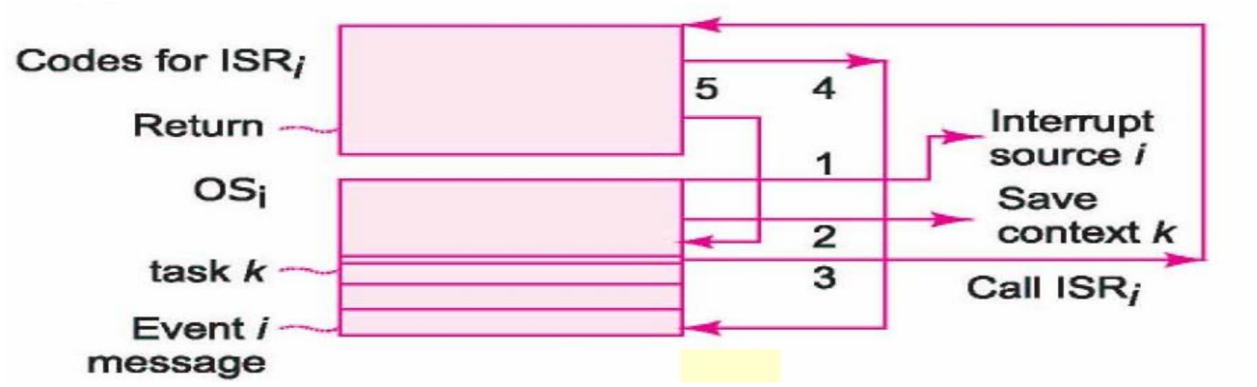
On an interrupt, the process running at the CPU is interrupted. ISR corresponding to that source starts executing. A hardware source calls an ISR directly. The ISR just sends an ISR enter message to the RTOS. ISR enter message is to inform the RTOS that an ISR has taken control of the CPU.

ISR code can send into a mailbox or message queue but the task waiting for a mailbox or message queue does not start before the return from the ISR. When ISR finishes, it sends exit message to OS. On return from ISR by retrieving saved context, the RTOS later on returns to the interrupted process (task) or reschedules the processes (tasks). RTOS action depends on the event-messages, whether the task waiting for the event message from the ISR is a task of higher priority than the interrupted task on the interrupt. On certain RTOSes, there may be a function **OSISRSemPost ( ).**

Multiple ISR Nesting

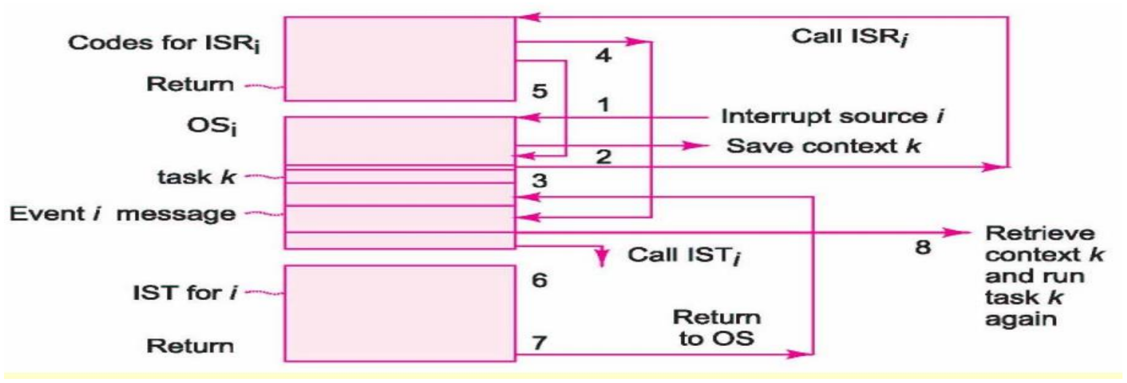
Each ISR low priority sends on high priority interrupt the ISR interrupt message (ISM) to the OS to facilitate return to it on the finishing and return from the higher priority interrupt. Nesting means when an interrupt source call of higher priority, for example, **SysClkIntr** occurs, then the control is passed to higher priority and on return from the higher priority the lower priority ISR starts executing. Each ISR on letting a higher priority interrupt call sends the ISM to the RTOS. Common stack for the ISR nested calls, similar to the nested function calls.

2. **RTOS first interrupting on an interrupt, then RTOS calling the corresponding ISR**



On interrupt of a task, say, k-th task, the RTOS first gets itself the hardware source call and initiates the corresponding ISR after saving the present processor status (or context). Then the ISR during execution can post one or more outputs for the events and messages into the mailboxes or queues. The ISR must be short and it must simply puts post the messages for another task. This task runs the remaining codes whenever it is scheduled. RTOS schedules only the tasks (processes) and switches the contexts between the tasks only. ISR executes only during a temporary suspension of a task. Each device event has the codes for an ISR, which executes only on scheduling it by the RTOS and provided an interrupt is pending for its service.

3. **RTOS first interrupting on interrupt, then RTOS calling the corresponding ISR, the ISR sending messages to priority queue of Interrupt Service threads by Temporary Suspension of a scheduled Task**



An RTOS can provide for two levels of interrupt service routines, a fast level ISR, FLISR and a slow level ISR (SLISR). The FLISR can also be called hardware interrupt ISR and the SLISR as software interrupt ISR. FLISR is called just the ISR in RTOS Windows CE. The SLISR is called interrupt service thread (IST) in Windows CE.

The use of FLISR reduces the interrupt latency (waiting period) for an interrupt service and jitter (worst case and best case latencies difference) for an interrupt service. An IST functions as deferred procedure call (DPC) of the ISR. An i-th interrupt service thread (IST) is a thread to service an i-th interrupt source call.

Step 1: RTOS get hardware source call

Step 2: After getting the call, initiate corresponding ISR and save the processor status after finishing the critical section of current process.

Step 3: ISR execute the code

Step 4: ISR during execution sends one or more outputs for events and messages into mailbox or queues for IST.

Step 5: ISR just before end, unmasks further preemption from the same or other hardware sources.

Step 6: Call the IST and executes the corresponding code.

Step 7: When no ISR or IST pending in queue, interrupt task run on return.

Step 8: Retrieve the saved processor status and run the task that is being suspended.

Real Time Operating System

A real time is the time which continuously increments at regular intervals after the start of the system and time for all the activities at difference instances take that time as a reference in the system. A real time operating system (RTOS) is multitasking operation system for the applications with hard or soft real time constraints. Real-time constraint means constraint on occurrence of an event and system expected response and latency to the event.

Basic Design

- An embedded system with a single CPU can run only one process at an instance. The process at any instance may either be an ISR, or kernel function or task
- Provides running the user threads in kernel space so that they execute fast.
- Provides effective handling of the ISRs, device drivers, ISTs, tasks or threads.
- Disabling and enabling of interrupts in user mode critical section code.
- Provides memory allocation and de-allocation functions in fixed time and blocks of memory.
- Provides for effectively scheduling and running and blocking of the tasks in cases of number of many tasks I/O Management with devices, files, mailboxes, pipes and sockets becomes simple using an RTOS.
- Provides for the uses of Semaphore (s) by the tasks or for the shared resources (critical sections) in a task or OS functions.
- Effective management of the multiple states of the CPU and, internal and external physical or virtual devices

Design Principle

1. **Design with the ISRs and Tasks**

The embedded system hardware source calls generates interrupts ISR ─ only post (send) the messages. It provides of nesting of ISRs, while tasks run concurrently. A task waits and takes the messages (IPCs) and post (send) the messages using the system calls. A task or ISR should not call another task or ISR.

2. **Each ISR design consisting of shorter code**

Since ISRs have higher priorities over the tasks, the ISR code should be made short so that the tasks don't wait longer to execute. A design principle is that the ISR code should be optimally short and the detailed computations be given to an IST or task by posting a message or parameters for that. The frequent posting of the messages by the IPC functions from the ISRs be avoided.

3. **Design with using Interrupt Service Threads or Interrupt Service tasks**

In certain RTOSes, for servicing the interrupts, there are two levels, fast level ISRs and slow level ISTs, the priorities are first for the ISRs, then for the ISTs and then the task. In certain RTOSes, three levels. ISRs post the messages for the ISTs and do the detailed computations. If RTOS is providing for only one level, then use the tasks as interrupt service threads.

4. Design Each Task with an infinite loop

Each task has a while loop which never terminates. A task waits for an IPC or signal to start. The task, which gets the signal or takes the IPC for which it is waiting, runs from the point where it was blocked or preempted. In preemptive scheduler, the high priority task can be delayed for some period to let the low priority task execute.

5. Design in the form of tasks for the Better and Predictable Response Time Control

It should provide the control over the response time of the different tasks. The different tasks are assigned different priorities and those tasks which system needs to execute with faster response are separated out. For example, in mobile phone device there is need for faster response to the phone call receiving task than the user key input. In digital camera, the task for recording the image needs faster response than the task for down loading the image on computer through USB port.

6. Design in the form of tasks Modular Design

System of multiple tasks makes the designλ modular. The tasks provide modular design.

7. Design in the form of tasks for Data Encapsulation

System of multiple tasks encapsulates the code and data of one task from the other by use of global variables in critical sections getting exclusive access by mutex.

8. Design with taking care of the time spent in the system calls

Expected time in general depends on the specific target processor of the embedded system and the memory access times. Some IPCs takes longer than the other.

9. Design with Limited number of tasks

Limit the number of tasks and select theλ appropriate number of tasks to increase the response time to the tasks, better control over shared resource and reduced memory requirement for stacks. The tasks, which share the data withλ number of tasks, can be designed as one single task.

10. Use appropriate precedence assignment strategy and Use Preemption

Use appropriate precedence assignment strategy and Use Preemption in place of Time Slicing.

11. Avoid Task Deletion

Create tasks at start-up only and avoid creating and then deleting tasks later at the later times. Only advantage in deleting is the availability of additional memory space.

12. Use CPU idle CPU time for internal functions

During CPU idle time, Read the internal queue, manage the memory and search for a free block of memory.

13. Design with Memory Allocation and De-Allocation By the Task

If memory allocation and de-allocation are done by the task then the number of functions required as the RTOS functions is reduced. This reduces the interrupt-latency periods. As execution of these functions takes significant time whenever the RTOS preempts a task. Further, if fixed sized memory blocks are allocated, then the predictability of time taken in memory allocation is there.

14. Design with limited RTOS functions

Use an RTOS, which can be configured. RTOS provides during execution of the codes enabling the limited RTOS functions. For example, if queue and pipe functions are not used during execution, then disable these during run.

15. Scalable RTOS and Hierarchical RTOS

Use an RTOS, which provides for creation of scalable code. Use an RTOS, which is hierarchical. The needed functions extended and interfaced with the functionalities. RTOS is configured with specific processor and devices.

Task Scheduling Models

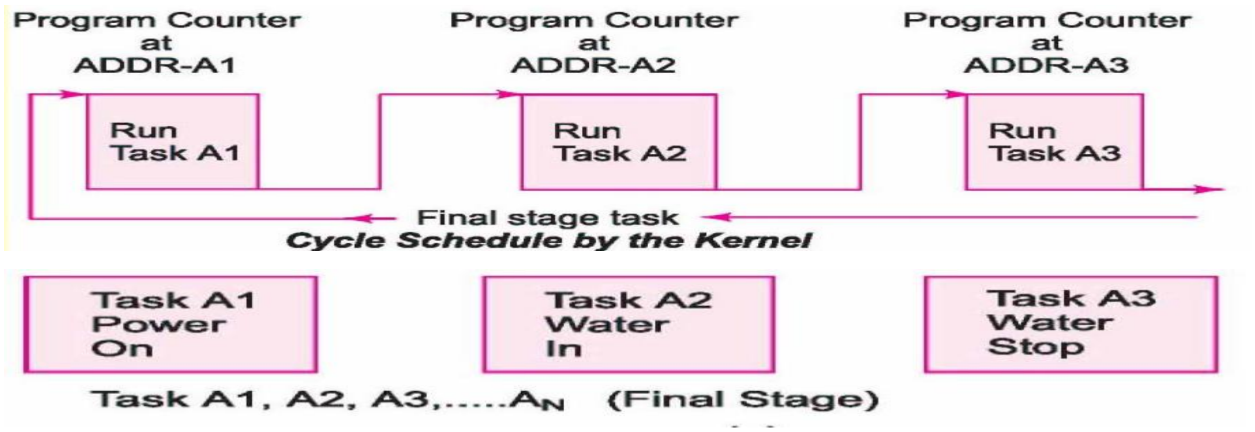
Common Scheduling Models are as follows:

- Cooperative Scheduling of ready tasks in a circular queue. It closely relates to function queue scheduling.
- Cooperative scheduling with precedence constraints.
- Cyclic scheduling of periodic tasks and Round Robin Time Slicing Scheduling of equal priority tasks.
- Preemptive Scheduling.
- Scheduling using 'Earliest Deadline First' (EDF) precedence.
- Rate Monotonic Scheduling using ‘higher rate of events occurrence First’ precedence.
- Fixed Times Scheduling

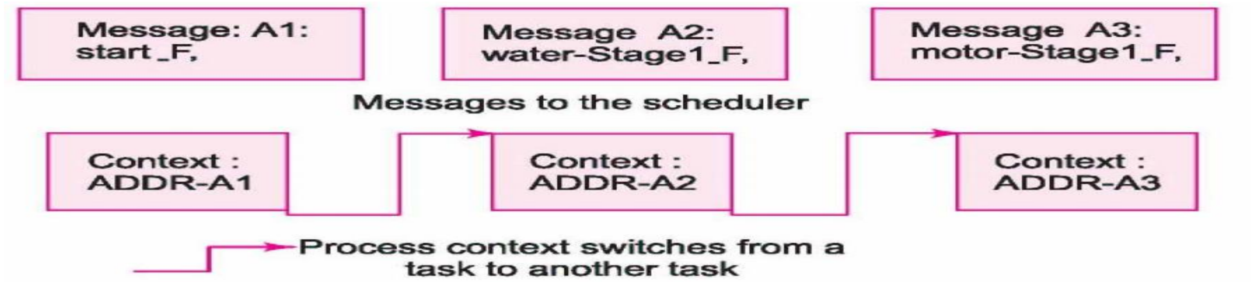
- Scheduling of Periodic, sporadic and a periodic Tasks
- Advanced scheduling algorithms using the probabilistic Timed Petri nets (Stochastic) or Multi Thread Graph for the multiprocessors and complex distributed systems.

1. **Cooperative Scheduling in the cyclic order**

   Each task cooperates to let the running task finish. Cooperative means that each task cooperates to let a running task to finish. None of the tasks does block in-between anywhere during the ready to finish states. The service is in the cyclic order.


Program Counter at ADDR-A1
Program Counter at ADDR-A2
Program Counter at ADDR-A3
Run Task A1
Run Task A2
Run Task A3
Final stage task
Cycle Schedule by the Kernel
Task A1 Power On
Task A2 Water In
Task A3 Water Stop
Task A1, A2, A3,.....A$_N$ (Final Stage)

   Messages from the scheduler and task programs contexts at various instances in washing machine tasks scheduling.


Message: A1: start_F,
Message A2: water-Stage1_F,
Message A3: motor-Stage1_F,
Messages to the scheduler
Context : ADDR-A1
Context : ADDR-A2
Context : ADDR-A3
Process context switches from a task to another task

   Worst-case latency is same for every task. It is given by

$$T_{worst} = \{(sti + eti)1 + (sti + eti)2 + ... + (sti + eti)N-1 + (sti + eti)N\} + t_{ISR}.$$

   where t_{ISR} is the sum of all execution times for the ISRs , for an i-th task, switching time from one task to another be is sti and task execution time be is eti, 1 .. N, when number of tasks = N– i = 1, 2, …, N

2. **Cooperative Scheduling of Ready Tasks in List**

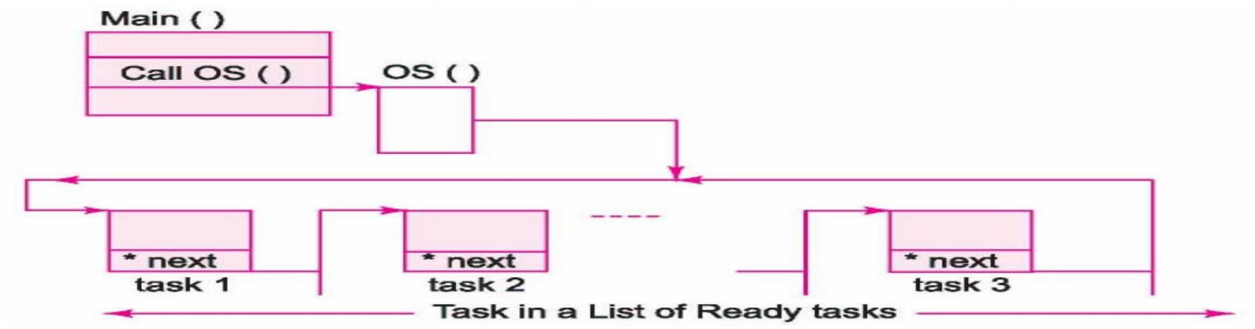
Cooperative Scheduling in the order in which a task is initiated on interrupt. None of the tasks does block in-between anywhere during the ready to finish states. The service is in the order in which a task is initiated on interrupt.

Worst-case latency is same for every task in the ready list. It is given by

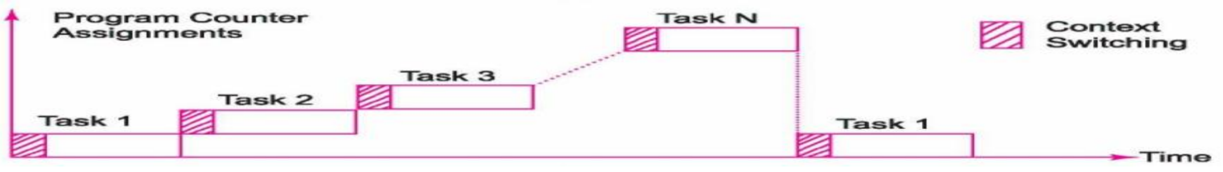
T_{worst} = {(dti + sti + eti)1 + (dti + sti + eti)2 +...+ (dti + sti + eti)n-1 + (dti + sti + eti)n} + t_{ISR}.

t_{ISR} is the sum of all execution times for the ISRs For an i-th task, let the event detection time with when an event is brought into a list be is dti , switching time from one task to another be is sti and task execution time be is eti.

Scheduler in which the scheduler inserts into a list the ready tasks for a sequential execution in a cooperative mode.



Program counter assignments (switch) at different times, when the on the scheduler calls the o tasks from the list one by one in the circular queue from the list.

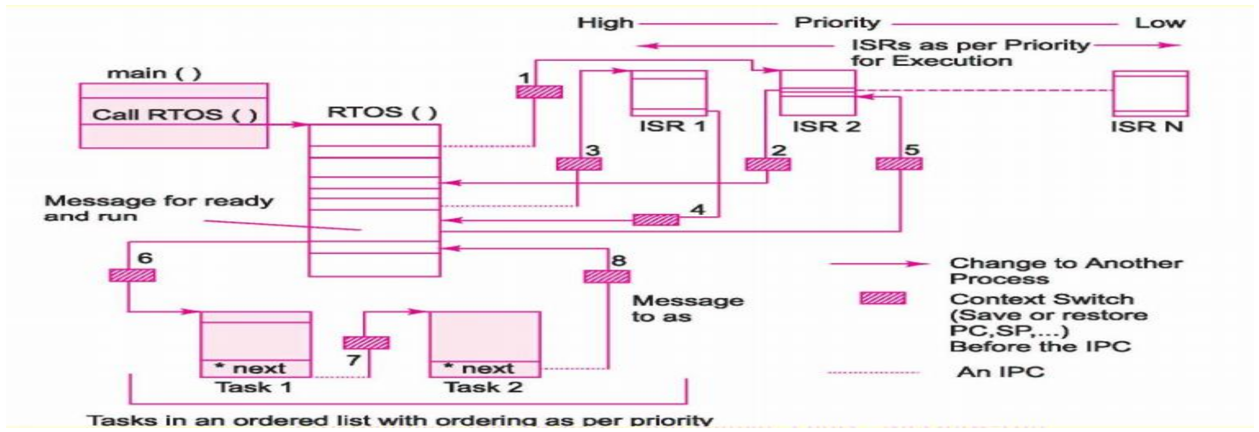


3. **Cooperative Scheduling of Ready Tasks Using an Ordered List as per precedence Constraints**

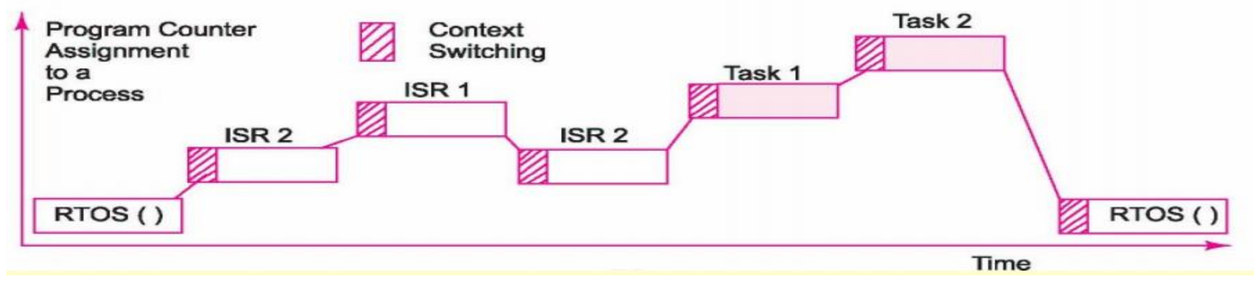
Cooperative Scheduling in the order of Ready Tasks using an Ordered List as per priority precedence. Scheduler using a priority parameter, task Priority does the ordering of list of the tasks─ ordering according to the precedence of the interrupt sources and tasks. The scheduler first executes only the firstλ task at the ordered list, and the total, equals to the period taken by the first task on at the list. It is deleted from the list after the first task is executed and the next task becomes the first. The insertions and deletions for forming the ordered list are made only at the beginning of the cycle for each list.

Worst-case latency is not same for every task. It varies from **(dti +λ sti + eti) p(m)} + t_{ISR}** to **{(dti + sti + eti)p1 + (dti + sti + eti) p2 +...+ (dti + sti + eti) p(m-1) + (dti + sti + eti) p(m)} + t_{ISR}.** p1,p2..p_m are the priority of each task t_{ISR} is the sum of all execution times for the ISRs. For an i-th task, let the event detection time withλ when an event is brought into a list be is dti , switching time from one task to another be is sti and task execution time be is eti. 1 , m; m is number of ISRs and– i = 1, 2, …, m λ tasks in the list.

Cooperative Priority based scheduling of the ISRs executing in the first layer and Priority based ready tasks at an ordered list executing in the second layer.


High
Priority
Low
ISRs as per Priority for Execution
main ( )
Call RTOS ( )
RTOS ( )
ISR 1
ISR 2
ISR N
1
2
3
4
5
6
7
8
Message for ready and run
Message to as
* next
Task 1
* next
Task 2
Change to Another Process
Context Switch (Save or restore PC,SP,...) Before the IPC
An IPC
Tasks in an ordered list with ordering as per priority

Program counter assignments at different times on the scheduler calls to the ISRs and the corresponding tasks



4. Cyclic Scheduling

Assume periodically occurring three tasks. Let in time-frames allotted to the first task, the task executes at t1, t1 + Tcycle, t1+ 2 *Tcycle, second task frames at t2, t2 + Tcycle,t2 + 2* Tcycle and third task at t3, t3 + Tcycle, t3+ 2* Tcycle.

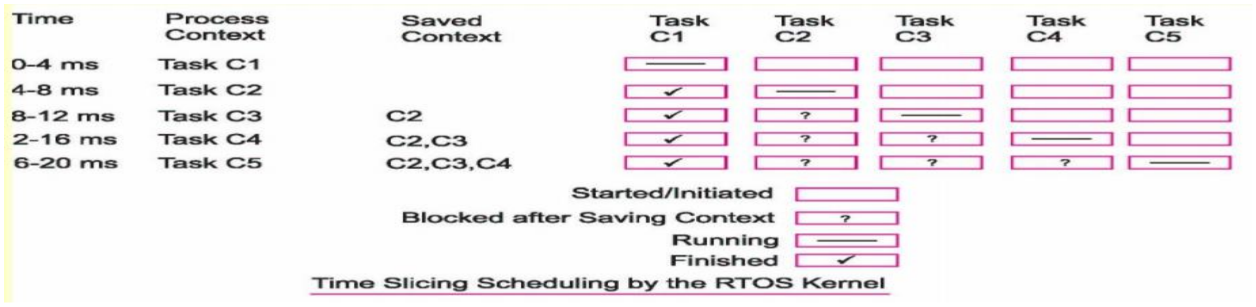
Start of a time frame is the scheduling point for the next task in the cycle. Tcycle is the cycle for repeating cycle of execution of tasks in order 1, 2 and 3 and equals start of task 1 time frame to end of task 3 frame. Tcycle is period after which each task timeλ frame allotted to that repeats.

Each of N tasks in a cyclic scheduler completes in its allocated time frame when the frame size is based on deadline. A cyclic scheduler is clock driven and is used for periodic tasks. Each tasks has same priority for the execution in cyclic mode.

5. Round Robin Time Slicing Scheduling

Round robin means that each ready task runs turn by in turn only in a cyclic queue for a limited time slice. It is widely used model in traditional OS. Round robin is a hybrid model of clock-driven model (for example cyclic model) as well as event driven (for example, preemptive). A real time system responds to the event within a bound time limit and within an explicit time.

Tasks programs contexts at the five instances in the Time Scheduling Scheduler for C1 to C5

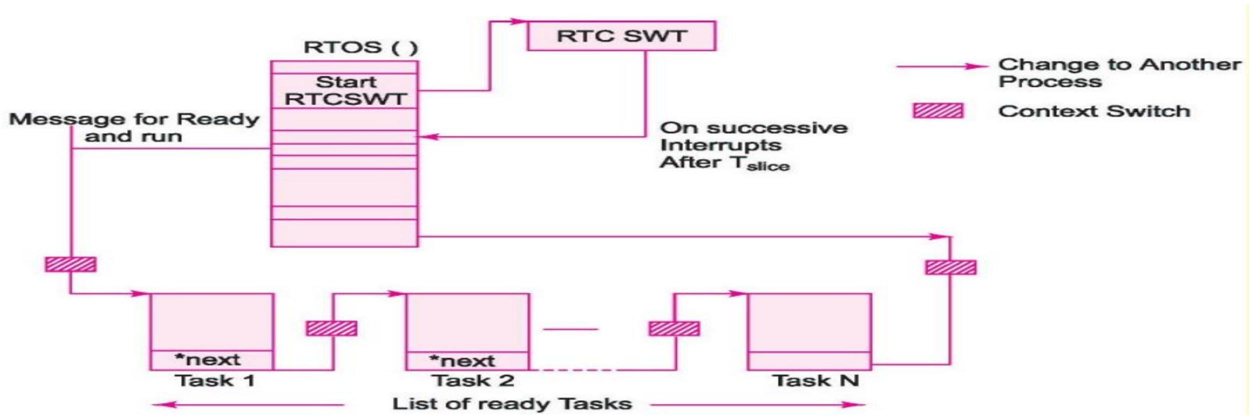


Time	Process Context	Saved Context	Task C1	Task C2	Task C3	Task C4	Task C5
0-4 ms	Task C1		—	▭	▭	▭	▭
4-8 ms	Task C2		✓	—	▭	▭	▭
8-12 ms	Task C3	C2	✓	?	—	▭	▭
2-16 ms	Task C4	C2,C3	✓	?	?	—	▭
6-20 ms	Task C5	C2,C3,C4	✓	?	?	?	—

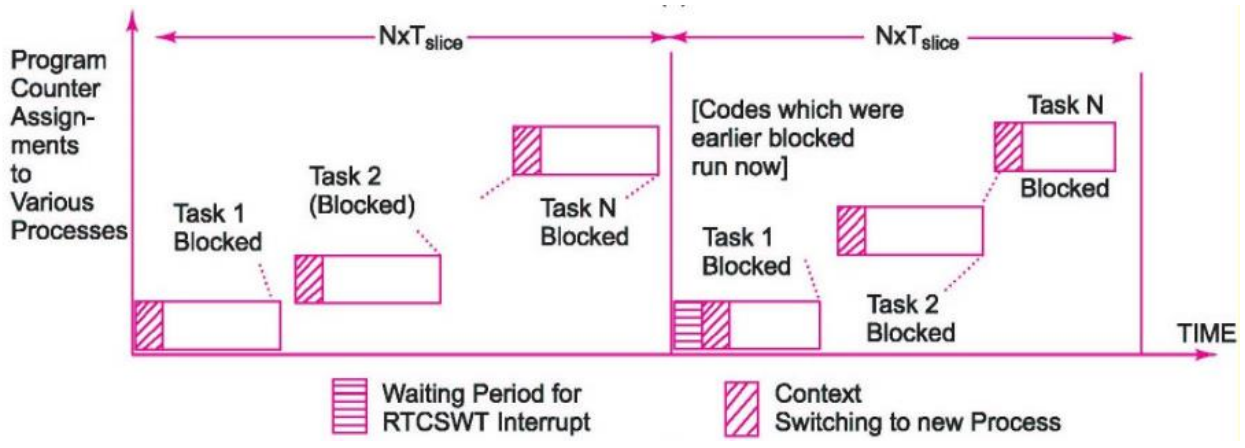
Started/Initiated	▭
Blocked after Saving Context	?
Running	—
Finished	✓

Time Slicing Scheduling by the RTOS Kernel

Programming model for the Cooperative Time sliced scheduling of the tasks


RTOS ( )
Start RTCSWT
RTC SWT
Message for Ready and run
On successive Interrupts After T_{slice}
Change to Another Process
Context Switch
*next
Task 1
*next
Task 2
Task N
List of ready Tasks

Program counter assignments on the scheduler call to tasks at two consecutive time slices. t×Each cycle takes time = N slice


Program Counter Assign-ments to Various Processes
$N x T_{slice}$
$N x T_{slice}$
Task 1 Blocked
Task 2 (Blocked)
Task N Blocked
[Codes which were earlier blocked run now]
Task 1 Blocked
Task 2 Blocked
Task N Blocked
TIME
Waiting Period for RTCSWT Interrupt
Context Switching to new Process

Same for every task = Tcycle

Tcycle =. {Tslice * N} + tISR

t_{ISR} is the sum of all execution times for the ISRs. For an i-th task, switching time from one task to another be is st and task execution time be is et. Number of tasks = N.

Alternative model strategy

Certain tasks are executed more than once and do not finish in one cycle. Decomposition of a task that takes the abnormally long time to be executed. The decomposition is into two or four or more tasks. Then one set of tasks (or the odd numbered tasks) can run in one time slice, t'slice and another set of tasks (or the even numbered tasks) in another time slice, t''slice.

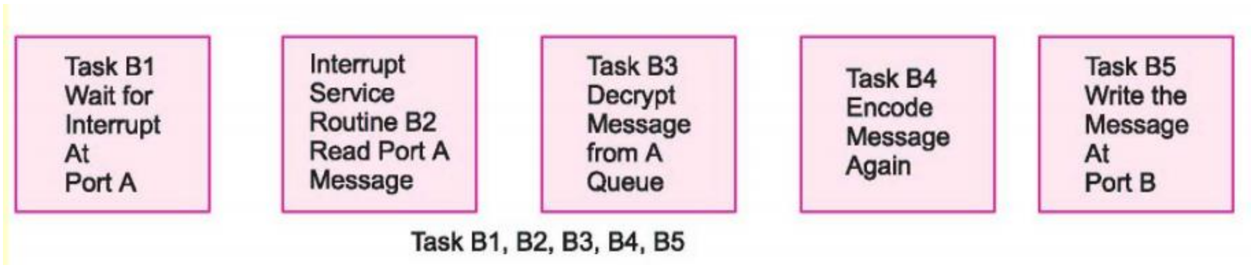
Decomposition of the long time taking task into a number of sequential states or a number of node places and transitions as in finite state machine. (FSM). Then its one of its states or transitions runs in the first cycle, the next state in the second cycle and so on. This task then reduces the response times of the remaining tasks that are executed after a state change.

Difficulties in cooperative and cyclic scheduling of tasks

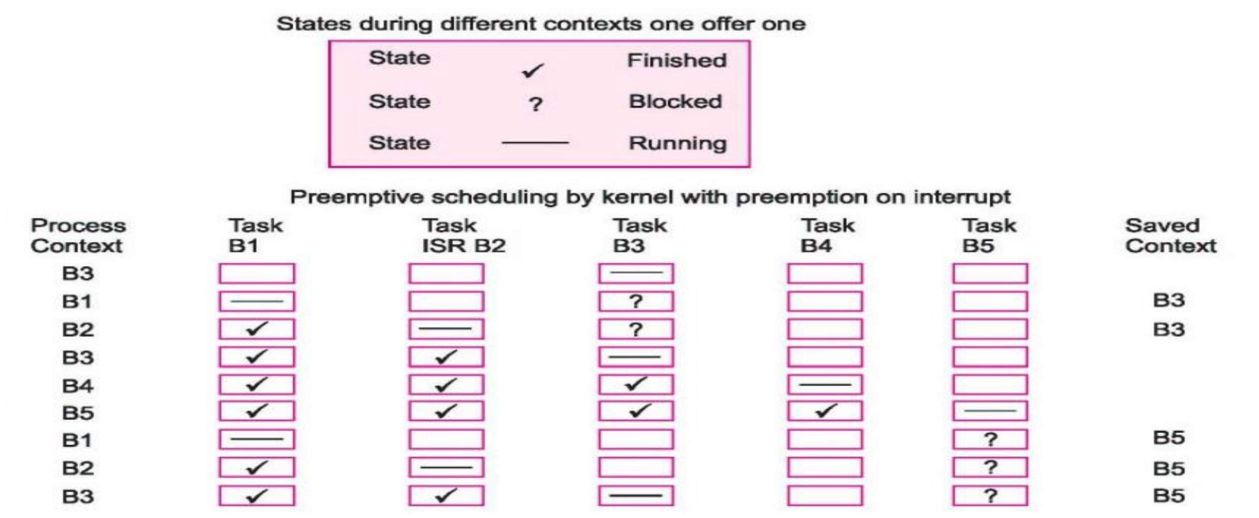
Cooperative schedulers schedule such that each ready task cooperates to let the running one finish. However, a difficulty in case of the cooperative scheduling is that a long execution time of a low-priority task lets a high priority task waits at least until that that finishes. Difficulty when the cooperative scheduler is cyclic but without a predefined tslice. Assume that an interrupt for service from first task occurs just at the beginning of the second task. The first task service waits till all other remaining listed or queued tasks finish. Worst case latency equals the sum of execution times of all tasks

Preemptive Scheduling of tasks

OS schedules such that higher priority task, when ready, preempts a lower priority by blocking Solves the problem of large worst case latency for high priority tasks.


Task B1 Wait for Interrupt At Port A
Interrupt Service Routine B2 Read Port A Message
Task B3 Decrypt Message from A Queue
Task B4 Encode Message Again
Task B5 Write the Message At Port B
Task B1, B2, B3, B4, B5

Task programs contexts at various instances in preemptive scheduling


States during different contexts one offer one

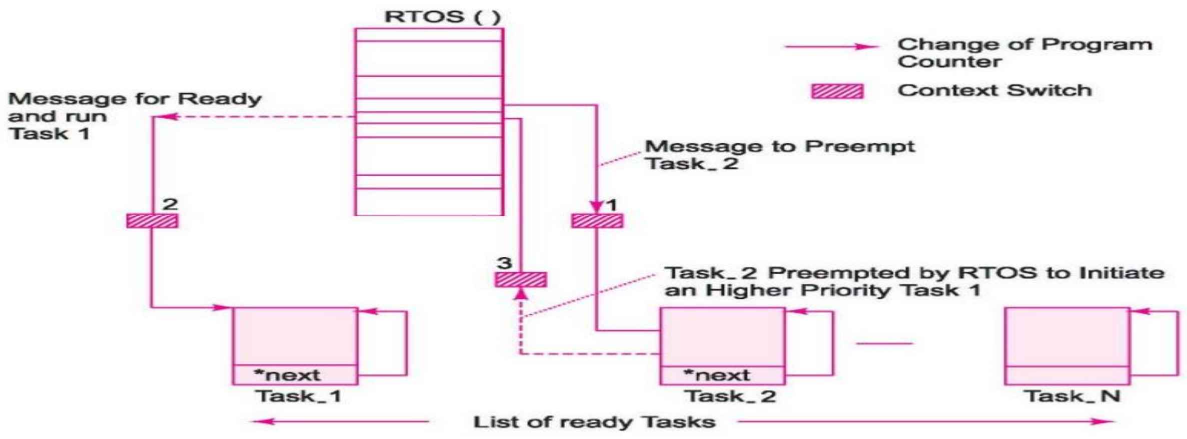
State	✓	Finished
State	?	Blocked
State	—	Running

Preemptive scheduling by kernel with preemption on interrupt

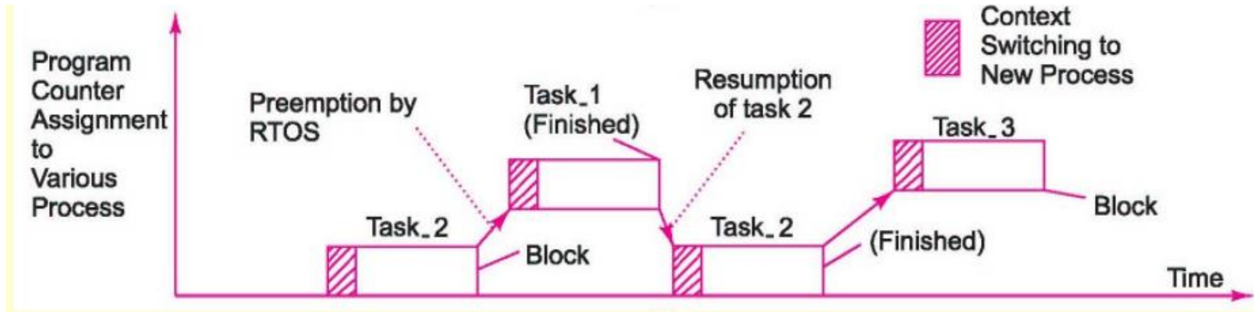
Process Context	Task B1	Task ISR B2	Task B3	Task B4	Task B5	Saved Context
B3			—			
B1	—		?			B3
B2	✓	—	?			B3
B3	✓	✓	—			
B4	✓	✓	✓	—		
B5	✓	✓	✓	✓	—	
B1	—				?	B5
B2	✓	—			?	B5
B3	✓	✓	—		?	B5

RTOS Preemptive Scheduling

Processes execute such that scheduler provides for preemption of lower priority process by higher priority process. Assume priority of task_1 > task_2> task_3 > task_4…. > task N. Each task has an infinite loop from start (Idle state) up to finish. Task 1 last instruction points to the next pointed address, *next. In case of the infinite loop, *next points to the same task 1 start.


RTOS ( )
Change of Program Counter
Context Switch
Message for Ready and run Task 1
Message to Preempt Task_2
2
1
3
Task_2 Preempted by RTOS to Initiate an Higher Priority Task 1
*next
Task_1
*next
Task_2
Task_N
List of ready Tasks

Program counter assignments on a scheduler call to preempt task 2. when priority of task_1 > task_2 > task_3



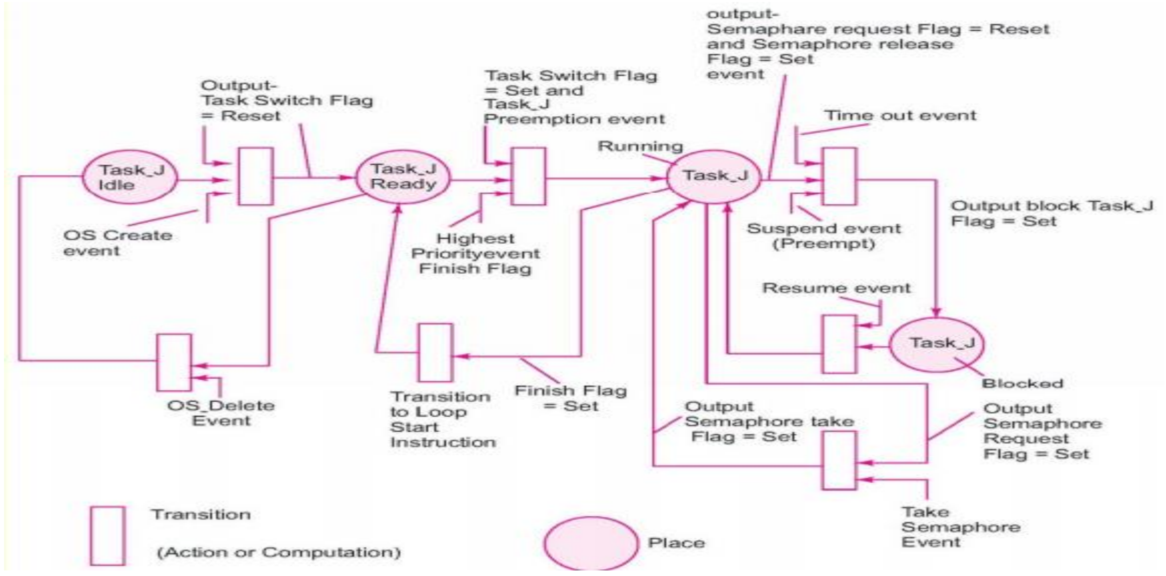
Worst-case latency is not same for every task. Highest priority task latency smallest. Lowest priority task latency highest.

Model for Critical Service by Preemptive Scheduler

Petri net concept based model

Petri net concept based model which models and helps in designing the codes for a task. The model shows places by the circles and transitions by the rectangles.

1. Each task is in idle state (or at idleTaskPlace) to start with, and a token to the RTOS is taskSwitchFlag = reset.
2. Let us consider the task_J_Idle place,which is currently has of highest priority among the ready tasks. When the RTOS creates task, task_J, the place, task_J_Idle undergoes a transition to the ready state (or to readyTaskPlace), task_J_Ready place. The RTOS initiates idle to ready transition by executing a function, task create ( ). For the present case it is done by executing a function, task_J_create ( ). A transition from the idle state of the task is fired as follows. RTOS sends two tokens, RTOS_CREATE Event and taskJSwitchFlag. The output token from the transition is taskSwitchFlag = true.
3. When after task J finishes, the RTOS sends a RTOS_DELETE event (a token) the task, it returns to the task_J_Idle place and its corresponding taskJSwitchFlag resets.



4. At task_J_Ready place, the scheduler takes the priority parameter into account. If the current task current happens to be of highest priority, the scheduler sets two tokens, taskJSwitchFlag

= true (sends a token) and highest Priority Event = true, for the transition to the running task J place, task_J_Running. The scheduler also resets and sends the tokens, task switch flags, for all other tasks that are of lesser priority. This is because the system has only one CPU to process at an instant

5. From the task_J_Running place, the transition to the task_J_Ready place will be fired when the task finish flag sets.
6. At task_J_Running place, the codes of the switched task J are executed.
7. At the runningTaskPlace, the transition for preempting will be fired when RTOS sends a token, suspendEvent. Another enabling token if present, is time_out_event will also fire the transition. An enabling token for both situations is the semaphore release flag, which must be set. Semaphore release flag is sets on finishing the codes of task J critical-sections. On firing, the next place is task_J_Blocked. Blocking is in two situations, one situation is of preemption. It happens is when the suspendEvent occurs on a call at the runningTaskPlace asking the RTOS to suspend the running. Another situation is a time-out of an SWT, which that associates with the running task place
8. On a resumeEvent (a token from RTOS) the transition to task_J_Running place occurs.
9. At the task_J_Running place, there is another transition that fires so that the task J is at back at to the to task_J_Running place when the RTOS sends a token, take_Semaphore_Event for to asking the task J to take the semaphore
10. There can be none or one or several sections taking and releasing semaphore or message. RTOS during the execution of a section, the RTOS resets the semaphore release flag and sets the take semaphore event token.

Earliest Deadline First (EDF) precedence

When a task becomes ready, it will be considered at a scheduling point. The scheduler does not assign any priority. It computes the deadline left at a scheduling point. Scheduling point is an instance at which the scheduler blocks the running task and re-computes the deadlines and runs the EDF algorithm and finds the task which is to be run. An EDF algorithm can also maintain a priority queue based on the computation when the new task inserts.

a) Precedence Assignment in the Scheduling Algorithms

Best strategy is one, which is based on EDF (Earliest Deadline First) precedence. Precedence is made the highest for a task that corresponds to an interrupt source, which occurs at the earliest a succeeding times and which deadline will finish earliest at the earliest. We assign precedence by appropriate strategy in the case of the variable CPU loads for the different tasks and variable EDFs .

b) Dynamic Precedence Assignment

Firstly, there is deterministic or static assignment of the precedence. It means firstly there is rate monotonic scheduling (RMS). Later on the scheduler dynamically assigns andλ fixes the timeout delays afresh, and assigns the precedence as per EDF. The need for of the dynamic assignment arises dueλ to the sporadic tasks and the distributed or multiprocessor indeterminate environment.

c) Rate Monotonic Scheduler

Rate monotonic scheduler computes the priorities, p, from the rate of occurrence of the tasks. The i-th task priority, pi is proportional to (1/ti) where ti is the period of the occurrence of the task event. RMA gives an advantage over EDF because most RTOSes have provisions for priority assignment. Higher priority tasks always get executed

Fixed (Static) Real Time Scheduling of the Tasks

A scheduler is said to be using a fixed time scheduling method when the schedule is static and deterministic.

Methods for Fixed (Static) Real Time Scheduling

(i) Simulated annealing method─ different schedules fixed and the performance is simulated. Now, schedules for the tasks are gradually incremented by changing the interrupt timer settings (using a corresponding OS function) till the simulation result shows that none is missing its deadline.

(ii) Heuristic method. Here, the reasoning or past experience lets us helps to define and fixing the fixed schedules

(iii) Dynamic programming model. This is as follows: Certain specific running program first determines the schedules for each task and then the timer interrupt s loads the timer settings from the outputs from that program.

How to choose RTOS

The decision of an RTOS for an embedded design is very critical. A lot of factors need to be analyzed carefully before making a decision on the selection of an RTOS.
These factors can be either

1. Functional
2. Non-functional requirements.

Functional Requirements:

1. Processor support

It is not necessary that all RTOS's support all kinds of processor architectures. It is essential to ensure the processor support by the RTOS

2. Memory Requirements

The RTOS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the OS service. Since embedded systems are memory constrained, it is essential to evaluate the minimal RAM and ROM requirements for the OS under consideration.

3. Real-Time Capabilities

It is not mandatory that the OS for all embedded systems need to be Real Time and all embedded OS's are 'Real-Time' in behavior. The Task/process scheduling policies plays an important role in the Real Time behavior of an OS.

4. Kernel and Interrupt Latency

The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency. For an embedded system whose response requirements are high, this latency should be minimal.

5. Inter process Communication (IPC) and Task Synchronization

The implementation of IPC and Synchronization is OS kernel dependent.

6. Modularization Support

Most of the OS's provide a bunch of features. It is very useful if the OS supports modularization where in which the developer can choose the essential modules and re-compile the OS image for functioning.

7. Support for Networking and Communication

The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking. Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

8. Development Language Support

Certain OS's include the run time libraries required for running applications written in languages like JAVA and C++. The OS may include these components as built-in component, if not, check the availability of the same from a third party.

Non-Functional Requirements

1. Custom Developed or Off the Shelf

It is possible to go for the complete development of an OS suiting the embedded system needs or use an off the shelf, readily available OS. It may be possible to build the required features by customizing an open source OS. The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

2. Cost

The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.

3. Development and Debugging tools Availability

The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain OS's may be superior in performance, but the availability of tools for supporting the development may be limited.

4. Ease of Use

How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

5. After Sales

For a commercial embedded RTOS, after sales in the form of e-mail, on-call services etc. for bug fixes, critical patch updates and support for production issues etc. should be analyzed thoroughly.