Diffeernece between General perpose OS and Real time operating system

The basic difference of using a GPOS or an RTOS lies in the nature of the system – i.e whether the system is ***"time critical"*** or not! A system can be of a single purpose or multiple purpose. Example of a "time critical system" is – Automated Teller Machines (ATM). Here an ATM card user is supposed to get his money from the teller machine within 4 or 5 seconds from the moment he press the confirmation button. The card user will not wait 5 minutes at the ATM after he pressed the confirm button. So an ATM is a time critical system. Where as a personal computer (PC) is not a time critical system. The purpose of a PC is multiple. A user can run many applications at the same time. After pressing the SAVE button of a finished document, there is no particular time limit that the doc should be saved within 5 seconds. It may take several minutes (in some cases) depending upon the number of tasks and processes running in parallel. I hope you got the basic idea now!

A GPOS is used for systems/applications that are not time critical

Example:- Windows, Linux, Unix etc.

An RTOS is used for time critical systems.

Example:- VxWorks, uCos etc. We can also say an RTOS is supposed to give quick and predictable response.

Now lets get into differences in the working of GPOS and RTOS.Before going deep down, it will be good if you read this article fro

Task Scheduling

In the case of a GPOS – task scheduling is not based on "priority" always! GPOS is programmed to handle scheduling in such a way that it manages to achieve high throughput. Here throughput means – the total number of processes that complete their execution per unit time. In such a case, some times execution of a high priority process will get delayed inorder to serve 5 or 6 low priority tasks. High throughput is achieved by serving 5 low priority tasks than by serving a single high priority one.

Where as in an RTOS – scheduling is always priority based. Most RTOS uses pre-emptive task scheduling method which is based on priority levels. Here a high priority process gets executed over the low priority ones. All **"low priority process execution"** will get paused. A high priority process execution will get override only if a request comes from an even high priority process.

Hardware and Economical factors

An RTOS is usually designed for a low end, stand alone device like an ATM, Vending machines, Kiosks etc. RTOS is light weight and small in size compared to a GPOS. A GPOS is made for high end, general purpose systems like a personal computer, a work station, a server system etc. The basic difference between a low end system and high end system is in it's hardware configuration. Now a days a personal computer or even a smart phone comes with high speed processors (in the range of many Gigahertz), large RAM's (in the range 2 or 3 GB's and even higher) etc. But an embedded system works on low hardware configurations usually – speed in the range of Megahertz and RAM in the range of Megabytes. A GPOS being too heavy demands very high end hardware configurations. It is economical to port an RTOS to an embedded system of limited expectations and functionalities (Example: An ATM is supposed to do only certain functions like Money transfer, Withdrawal, Balance check etc). So it is more logical to use an RTOS inside the ATM with its limited hardware. It is not economical to improve the hardware of an ATM just to port a GPOS as it's user interface.

Latency issues

Another major issue with a GPOS is unbounded dispatch latency, which most GPOS falls into. The more number of threads to schedule, latencies will get added up! An RTOS has no such issues because all the process and threads in it has got bounded latencies – which means – a process/thread will get executed within a specified time limit.

Preemptible Kernel

The kernel of an RTOS is preemptible where as a GPOS kernel is not preemptible.

This is a major issue when it comes to serving high priority process/threads first. If kernel is not preemptible, then a request/call from kernel will override all other process and threads. For example:- a request from a driver or some other system service comes in, it is treated as a kernel call which will be served immediately overriding all other process and threads. In an RTOS the kernel is kept very simple and only very important service requests are kept within the kernel call. All other service requests are treated as external processes and threads. All such service requests from kernel are associated with a bounded latency in an RTOS. This ensures highly predictable and quick response from an RTOS.

These all are the basic differences between an RTOS and GPOS.

Real-Time Operating System : Features and Requirements of embedded system

Clocks and time services are among some of the basic facilities provided to programmers by every real-time operating system. The time services provided by an operating system are based on a software clock called the system clock maintained by the operating system. The system clock is maintained by the kernel based on the interrupts received from the hardware clock. Since hard real-time systems usually have timing constraints in the micro seconds range, the system clock should have sufficiently fine resolution1 to support the necessary time services. However, designers of real-time operating systems find it very difficult to support very fine resolution system clocks. In current technology, the resolution of hardware clocks is usually finer than a nanosecond (contemporary processor speeds exceed 3GHz). But, the clock resolution being made available by modern real-time operating systems to the programmers is of the order of several milliseconds or worse. Let us first investigate why real-time operating system designers find it difficult to maintain system clocks with sufficiently fine resolution. We then examine various time services that are built based on the system clock, and made available to the real-time programmers.

The hardware clock periodically generates interrupts (often called time service interrupts). After each clock interrupt, the kernel updates the software clock and also performs certain other work .A thread can get the current time reading of the system clock by invoking a system call supported by the operating system (such as the POSIX clock-gettime()). The finer the resolution of the clock, the more frequent need to be the time service interrupts and larger is the amount of processor time the kernel spends in responding to these interrupts. This overhead places a limitation on how fine is the system clock resolution a computer can support. Another issue that caps the resolution of the system clock is the response time of the clock-gettime() system call is not deterministic. In fact, every system call (or for that matter, a function call) has some associated jitter. The problem gets aggravated in the following situation. The jitter is caused on account of interrupts having higher priority than system calls. When an interrupt occurs, the processing of a system call is stalled. Also, the preemption time of system calls can vary because many operating systems disable interrupts while processing a system call. The variation in the

response time (jitter) introduces an error in the accuracy of the time value that the calling thread gets from the kernel. Jitter is defined as the difference between the worst-case response time and the best case response time. In commercially available operating systems, jitters associated with system calls can be several milliseconds. A software clock resolution finer than this error, is therefore not meaningful.

Each time a clock interrupt occurs, besides incrementing the software clock, the handler routine carries out the following activities:

Process timer events: Real-time operating systems maintain either per-process timer queues or a single system-wide timer queue. A timer queue contains all timers arranged in order of their expiration times. Each timer is associated with a handler routine. The handler routine is the function that should be invoked when the timer expires. At each clock interrupt, the kernel checks the timer data structures in the timer queue to see if any timer event has occurred. If it finds that a timer event has occurred, then it queues the corresponding handler routine in the ready queue.

Update ready list: Since the occurrence of the last clock event, some tasks might have arrived or become ready due to the fulfillment of certain conditions they were waiting for. The tasks in the wait queue are checked, the tasks which are found to have become ready, are queued in the ready queue. If a task having higher priority than the currently running task is found to have become ready, then the currently running task is preempted and the scheduler is invoked.

Update execution budget: At each clock interrupt, the scheduler decrements the time slice (budget) remaining for the executing task. If the remaining budget becomes zero and the task is not complete, then the task is preempted, the scheduler is invoked to select another task to run.

Overhead associated with processing the clock interrupt becomes excessive. Secondly, the jitter associated with the time lookup system call (clock-gettime()) is often of the order of several milliseconds. Therefore, it is not useful to provide a clock with a resolution any finer than this. However, some real-time applications need to deal with timing constraints of the order of a few nanoseconds. Is it at all possible to support time measurement with nanosecond resolution? A way to provide sufficiently fine clock resolution is by mapping a hardware clock into the address space of applications. An application can then read the hardware clock directly (through a normal memory read operation) without having to make a system call. On a Pentium processor, a user thread can be made to read the Pentium time stamp counter. This counter starts at 0 when

the system is powered up and increments after each processor cycle. At today's processor speed, this means that during every nanosecond interval, the counter increments several times.

However, making the hardware clock readable by an application significantly reduces the portability of the application. Processors other than Pentium may not have a high resolution counter, and certainly the memory address map and resolution would differ.

Timers

We had pointed out that timer service is a vital service that is provided to applications by all real-time operating systems. Real-time operating systems normally support two main types of timers: periodic timers and aperiodic (or one shot) timers. We now discuss some basic concepts about these two types of timers.

Periodic Timers: Periodic timers are used mainly for sampling events at regular intervals or performing some activities periodically. Once a periodic timer is set, each time after it expires the corresponding handler routine is invoked, it gets reinserted into the timer queue. For example, a periodic timer may be set to 100 msec and its handler set to poll the temperature sensor after every 100 msec interval.

Aperiodic (or One Shot) Timers: These timers are set to expire only once. **Watchdog timers** are popular examples of one shot timers.

Watchdog timers are used extensively in real-time programs to detect when a task misses its deadline, and then to initiate exception handling procedures upon a deadline miss. A watchdog timer is set at the start of a certain critical function f*()* through a *wd_start(t1)* call. The *wd_start(t1)* call sets the watch dog timer to expire by the specified deadline (t_1) of the starting of the task. If the function *f()* does not complete even after t_1 time units have elapsed, then the watchdog timer fires, indicating that the task deadline must have been missed and the exception handling procedure is initiated. In case the task completes before the watchdog timer expires (i.e. the task completes within its deadline), then the watchdog timer is reset using a *wd_ tickle()* call.

Features of a Real-Time Operating System

Before discussing about commercial real-time operating systems, we must clearly understand the features normally expected of a real-time operating system and also let us compare different real-time operating systems. This would also let us understand the differences between a traditional operating system and a real-time operating system. In the following, we identify some important features required of a real-time operating system, and especially those that are normally absent in traditional operating systems.

Clock and Timer Support: Clock and timer services with adequate resolution are one of the most important issues in real-time programming. Hard real-time application development often requires support of timer services with resolution of the order of a few microseconds. And even finer resolution may be required in case of certain special applications. Clocks and timers are a vital part of every real-time operating system. On the other hand, traditional operating systems often do not provide time services with sufficiently high resolution.

Real-Time Priority Levels: A real-time operating system must support static priority levels. A priority level supported by an operating system is called static, when once the programmer assigns a priority value to a task, the operating system does not change it by itself. Static priority levels are also called *real-time priority levels*. All traditional operating systems dynamically change the priority levels of tasks from programmer assigned values to maximize system throughput. Such priority levels that are changed by the operating system dynamically are obviously not static priorities.

Fast Task Preemption: For successful operation of a real-time application, whenever a high priority critical task arrives, an executing low priority task should be made to instantly yield the CPU to it. The time duration for which a higher priority task waits before it is allowed to execute is quantitatively expressed as the corresponding *task preemption time.* Contemporary real-time operating systems have task preemption times of the order of a few micro seconds. However, in traditional operating systems, the worst case task preemption time is usually of the order of a second. We discuss in the next section that this significantly large latency is caused by a non-preemptive kernel. It goes without saying that a real-time operating system needs to have a preemptive kernel and should have task preemption times of the order of a few micro seconds.

Predictable and Fast Interrupt Latency: Interrupt latency is defined as the time delay between the occurrence of an interrupt and the running of the corresponding ISR (Interrupt Service Routine). In real-time operating systems, the upper bound on interrupt latency must be bounded and is expected to be less than a few micro seconds. The way low interrupt latency is achieved, is by performing bulk of the activities of ISR in a deferred procedure call (DPC). A DPC is essentially a task that performs most of the ISR activity. A DPC is executed later at a certain priority value. Further, support for nested interrupts are usually desired. That is, a real-time operating system should not only be preemptive while executing kernel routines, but should be preemptive during interrupt servicing as well. This is especially important for hard real-time applications with sub-microsecond timing requirements.

Support for Resource Sharing Among Real-Time Tasks: If real- time tasks are allowed to share critical resources among themselves using the traditional resource sharing techniques, then the response times of tasks can become unbounded leading to deadline misses. This is one compelling reason as to why every commercial real-time operating system should at the minimum provide the basic priority inheritance mechanism. Support of priority ceiling protocol (PCP) is also desirable, if large and moderate sized applications are to be supported.

Requirements on Memory Management: As far as general-purpose operating systems are concerned, it support virtual memory and memory protection features. However, embedded real-time operating systems almost never support these features. Only those that are meant for large and complex applications do. Real-time operating systems for large and medium sized applications are expected to provide virtual memory support, not only to meet the memory demands of the heavy weight tasks of the application, but to let the memory demanding non-real-time applications such as text editors, e-mail software, etc. to also run on the same platform. Virtual memory reduces the average memory access time, but degrades the worst-case memory access time. The penalty of using virtual memory is the overhead associated with storing the address translation table and performing the virtual to physical address translations. Moreover, fetching pages from the secondary memory on demand incurs significant latency. Therefore, operating systems supporting virtual memory must provide the real-time applications with some means of controlling paging, such as memory locking. Memory locking prevents a page from being swapped from memory to hard disk. In the absence of memory locking feature, memory

access times of even critical real-time tasks can show large jitter, as the access time would greatly depend on whether the required page is in the physical memory or has been swapped out.

Memory protection is another important issue that needs to be carefully considered. Lack of support for memory protection among tasks leads to a single address space for the tasks. Arguments for having only a single address space include simplicity, saving memory bits, and light weight system calls. For small embedded applications, the overhead of a few Kilo Bytes of memory per process can be unacceptable. However, when no memory protection is provided by the operating system, the cost of developing and testing a program without memory protection becomes very high when the complexity of the application increases. Also, maintenance cost increases as any change in one module would require retesting the entire system.

Embedded real-time operating systems usually do not support virtual memory. Embedded real-time operating systems create physically contiguous blocks of memory for an application upon request. However, memory fragmentation is a potential problem for a system that does not support virtual memory. Also, memory protection becomes difficult to support a non-virtual memory management system. For this reason, in many embedded systems, the kernel and the user processes execute in the same space, i.e. there is no memory protection. Hence, a system call and a function call within an application are indistinguishable. This makes debugging applications difficult, since a runaway pointer can corrupt the operating system code, making the system “freeze”.

Additional Requirements for Embedded Real-Time Operating Systems:

Embedded applications usually have constraints on cost, size, and power consumption. Embedded real-time operating systems should be capable of diskless operation, since many times disks are either too bulky to use, or increase the cost of deployment. Further, embedded operating systems should minimize total power consumption of the system. Embedded operating systems usually reside on ROM. For certain applications which require faster response, it may be necessary to run the real- time operating system on a RAM. Since the access time of a RAM is lower than that of a ROM, this would result in faster execution. Irrespective of whether ROM or RAM is used, all ICs are expensive. Therefore, for real-time operating systems for embedded applications it is desirable to have as small a foot print (memory usage) as possible. Since

embedded products are typically manufactured large scale, every rupee saved on memory and other hardware requirements impacts millions in profit.

Scheduling in RTOS:

Introduction

In the physical world, the purpose of a real-time system is to have a physical effect within a chosen time-frame. Typically, a real-time system consists of a controlling system (computer) and a controlled system (environment). The controlling system interacts with its environment based on information available about the environment. On a real-time computer, which controls a device or process, sensors will provide readings at periodic intervals and the computer must respond by sending signals to actuators. There may be unexpected or irregular events and these must also receive a response. In all cases, there will be a time bound within which the response should be delivered. The ability of the computer to meet these demands depends on its capacity to perform the necessary computations in the given time. If a number of events occur close together, the computer will need to schedule the computations so that each response is provided within the required time bounds. It may be that, even so, the system is unable to meet all the possible unexpected demands. In this case we say that the system lacks sufficient resources; a system with unlimited resources and capable of processing at infinite speed could satisfy any such timing constraint. Failure to meet the timing constraint for a response can have different consequences; there may be no effect at all, or the effects may be minor or correctable, or the results may be catastrophic. Each task occurring in a real-time system has some timing properties. These timing properties should be considered when scheduling tasks on a real-time System. The timing properties of a given task refer to the following items

***Release time* (or *ready time*):** Time at which the task is ready for processing.

***Deadline*:** Time by which execution of the task should be completed, after the task is released.

Minimum delay: Minimum amount of time that must elapse before the execution of the task is started, after the task is released.

***Maximum delay*:** Maximum permitted amount of time that elapses before the execution of the task is started, after the task is released.

***Worst case execution time*:** Maximum time taken to complete the task, after the task is released. The worst case execution time is also referred to as the *worst case response time*.

***Run time*:** Time taken without interruption to complete the task, after the task is released.

***Weight* (or *priority*):** Relative urgency of the task.

A real-time system will usually have to meet many demands within a limited time. The importance of the demands may vary with their nature (e.g. a safety-related demand may be more important than a simple data-logging demand) or with the time available for a response. So the allocation of the system resources needs to be planned so that all demands are met by the time of their respective deadlines. This is usually done using a scheduler which implements a scheduling policy that determines how the resources of the system are allocated to the program.

A process is a program in execution, whereas a thread is a path of execution within a process. Processes are generally used to execute large, ‘heavyweight’ jobs such as running different applications, while threads are used to carry out much smaller or ‘lightweight’ jobs such as auto saving a document in a program, downloading files, etc. Whenever we double-click an executable file such as Paint, for instance, the CPU starts a process and the process starts its primary thread.

Each process runs in a separate address space in the CPU. But threads, on the other hand, may share address space with other threads within the same process. This sharing of space facilitates communication between them. Therefore, Switching between threads is much simpler and faster than switching between processes.

1. Threads are easier to create than processes since they don't require a separate address space.

2. Multithreading requires careful programming since threads share data strucures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.

3. Threads are considered lightweight because they use far less resources than processes.

4. Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other.
Process can consists of multiple threads.

MULTITASKING IN REAL-TIME OPERATING SYSTEMS

A Real-Time OS (RTOS) is an OS with special features that make it suitable for building real-time computing applications also referred to as Real-Time Systems (RTS). An RTS is a (computing) system where correctness of computing depends not only on the correctness of the logical result of the computation, but also on the result delivery time. An RTS is expected to respond in a timely, predictable way to unpredictable external stimuli.

Real-time systems can be categorized as Hard or Soft. For a Hard RTS, the system is taken to have failed if a computing deadline is not met. In a Soft RTS, a limited extent of failure in meeting deadlines results in degraded performance of the system and not catastrophic failure.

A *Good* RTOS is one that enables bounded (predictable) behavior under all system load scenarios. Note however, that the RTOS, by itself cannot guarantee system correctness, but only is an enabling technology. That is, it provides the application programmer with facilities using which a correct application can be built. Speed, although important for meeting the overall requirements, does not by itself meet the requirements for an RTOS.

Programs, Processes, Tasks and Threads

The above four terms are often found in literature on OS in similar contexts. All of them refer to a unit of computation. A ***program*** is a general term for a unit of computation and is typically used in the context of programming. A ***process*** refers to a program in execution. A process is an independently executable unit handled by an operating system. Sometimes, to ensure better utilization of computational resources, a process is further broken up into ***threads***. Threads are sometimes referred to as lightweight processes because many threads can be run in parallel, that is, one at a time, for each process, without incurring significant additional overheads. A ***task*** is a generic term, which, refers to an independently schedulable unit of computation, and is used typically in the context of scheduling of computation on the processor. It may refer either to a process or a thread

Multitasking

A multitasking environment allows applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The inter-task communication facilities allow these tasks to synchronize and coordinate their activity. *Multitasking* provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events and is therefore essential for many real-time applications. Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. This also leads to efficient utilization of the CPU time and is essential for many embedded applications where processors are limited in computing speed due to cost, power, silicon area and other constraints. In a multi-tasking operating system it is assumed that the various tasks are to cooperate to serve the requirements of the overall system. Co-operation will require that the tasks communicate with each other and share common data in an orderly and disciplined manner, without creating undue contention and deadlocks. The way in which tasks communicate and share data is to be regulated such that communication or shared data access error is prevented and data, which is private to a task, is protected. Further, tasks may be dynamically created and terminated by other tasks, as and when needed. To realize such a system, the following major functions are to be carried out.

A. Process Management
- interrupt handling
- task scheduling and dispatch
  - create/delete, suspend/resume task
  - manage scheduling information
    - priority, scheduling policy, etc

B. Interprocess Communication and Synchronization
- Code, data and device sharing
- Synchronization, coordination and data exchange mechanisms
- Deadlock and Livelock detection

C. Memory Management
- dynamic memory allocation
- memory locking

- Services for file creation, deletion, reposition and protection

D. Input/Output Management

• Handles request and release functions and read, write functions for a variety of peripherals

The following are important requirements that an OS must meet to be considered an RTOS in the contemporary sense.

A. The operating system must be multithreaded and preemptive. e.g. handle multiple threads and be able to preempt tasks if necessary.

B. The OS must support priority of tasks and threads.

C. A system of priority inheritance must exist. Priority inheritance is a mechanism to ensure that lower priority tasks cannot obstruct the execution of higher priority tasks.

D. The OS must support various types of thread/task synchronization mechanisms.

E. For predictable response :

a. The time for every system function call to execute should be predictable and independent of the number of objects in the system.

b. Non preemptable portions of kernel functions necessary for interprocess synchronization and communication are highly optimized, short and deterministic

c. Non-preemptable portions of the interrupt handler routines are kept small and deterministic

d. Interrupt handlers are scheduled and executed at appropriate priority

e. The maximum time during which interrupts are masked by the OS and by device drivers must be known.

f. The maximum time that device drivers use to process an interrupt, and specific IRQ information relating to those device drivers, must be known.

g. The interrupt latency (the time from interrupt to task run) must be predictable and compatible with application requirements

F. For fast response:

a. Run-time overhead is decreased by reducing the unnecessary context switch.

b. Important timings such as context switch time, interrupt latency, semaphore get/release latency must be minimum

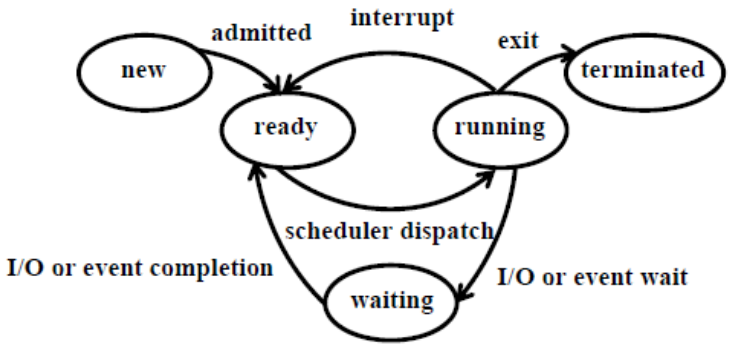
Process Management

On a computer system with only one processor, only one task can run at any given time, hence the other tasks must be in some state other than running. The number of other states, the names given to those states and the transition paths between the different states vary with the operating system. A typical state diagram is given in following figure and the various states are described below.

Task States

- ♦ Running: This is the task which has control of the CPU. It will normally be the task which has the highest current priority of the tasks which are ready to run.

- ♦ Ready: There may be several tasks in this state. The attributes of the task and the resources required to run it must be available for it to be placed in the 'ready' state.
- ♦ Waiting: The execution of tasks placed in this state has been suspended because the task requires some resources which is not available or because the task is waiting for some signal from the plant, e.g., input from the analog-to-digital converter, or the task is waiting for the elapse of time.
- ♦ New: The operating system is aware of the existence of this task, but the task has not been allocated a priority and a context and has not been included into the list of schedulable tasks.
- ♦ Terminated: The operating system has not as yet been made aware of the existence of this task, although it may be resident in the memory of the computer.



Task Control Functions

RTOSs provide functions to spawn, initialize and activate new tasks. They provide functions to gather information on existing tasks in the system, for task naming, checking of the state of a given task, setting options for task execution such as use of co-processor, specific memory models, as well as task deletion. Deletion often requires special precautions, especially with respect to semaphores, for shared memory tasks.

Task Context

Whenever a task is switched its execution context, represented by the contents of the program counter, stack and registers, is saved by the operating system into a special data structure called a task control block so that the task can be resumed the next time it is scheduled. Similarly the context has to be restored from the task control block when the task state is set to running. The information related to a task stored in the TCB is shown below.

Task Control Block consists

Task ID: the unique identifier for a task
Address Space : the address ranges of the data and codeblocks of the task loaded in memory including statically and dynamically allocated blocks
Task Context: includes the task's program counter(PC) , the CPU registers and (optionally) floating-point registers, a stack for dynamic variables and function calls, the stack pointer (SP), I/O device assignments, a delay timer, a time-slice timer and kernel control structures
Task Parameters : includes task type, event list
Scheduling Information : priority level, relative deadline, period, state
Synchronization Information : semaphores, pipes, mailboxes, message queues, file handles etc.
Parent and Child Tasks

Task Scheduling and Dispatch

The basic purpose of task scheduling and dispatch in a real-time multi-tasking OS is to ensure that each task gets access to the CPU and other system resources in a manner that is necessary for successful and timely completion of all computation in the system. Secondly, it is desired that this is done efficiently from the point of view of resource utilization as well as with correct synchronization and protection of data and code for individual tasks against incorrect interference. Various task scheduling and dispatch models are in use to achieve the above. The appropriateness of a particular model depends on the application features.

Multitasking in RTOS

This is the most complex model for real-time multi-tasking. The major features that distinguish it from the other ones described above are the following.

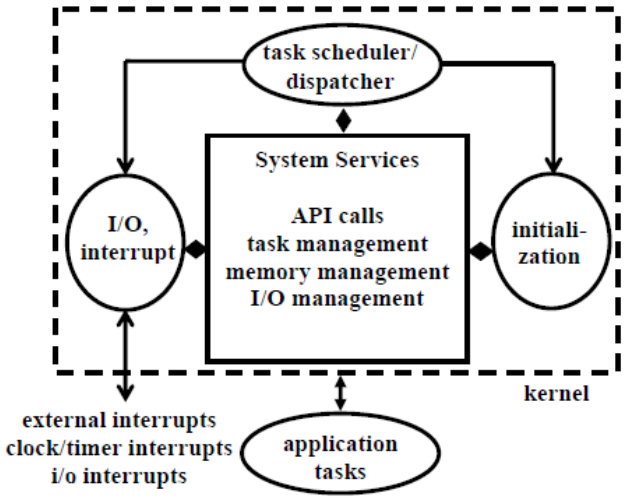
1. The explicit implementation of a scheduling policy in the form of a scheduler module. The scheduler is itself a task which executes every time an internal or external interrupt occurs and computes the decision on making state transitions for every application task in the system that has been spawned and has not yet been terminated. It computes this decision based on the current priority level of the tasks, the availability of the various resources of the system etc. The scheduler also computes the current priority levels of the tasks based on various factors such as deadlines, computational dependencies, waiting times etc.
2. Based on the decisions of the scheduler, the dispatcher actually effects the state transition of the tasks by
   a. saving the computational state or context of the currently executing task from the hardware environment.
   b. enabling the next task to run by loading the process context into the hardware environment.

   It is also the responsibility of the dispatcher to make the short-term decisions in response to, e.g., interrupts from an input/output device or from the real-time clock.

The dispatcher/scheduler has two entry conditions:

1. The real-time clock interrupt and any interrupt which signals the completion of an input/output request
2. A task suspension due to a task delaying, completing or requesting an input/output transfer.

In response to the first condition the scheduler searches for work starting with the highest priority task and checking each task in priority order. Thus if tasks with a high repetition rate are given a high priority they will be treated as if they were clock-level tasks, i.e., they will be run first during each system clock period. In response to the second condition a search for work is started at the task with the next lowest priority to the task which has just been running. There cannot be another higher priority task ready to run since a higher priority task becoming ready always preempts a lower priority running task.



The typical structure of an RTOS kernel showing the interaction between the System and the Application tasks