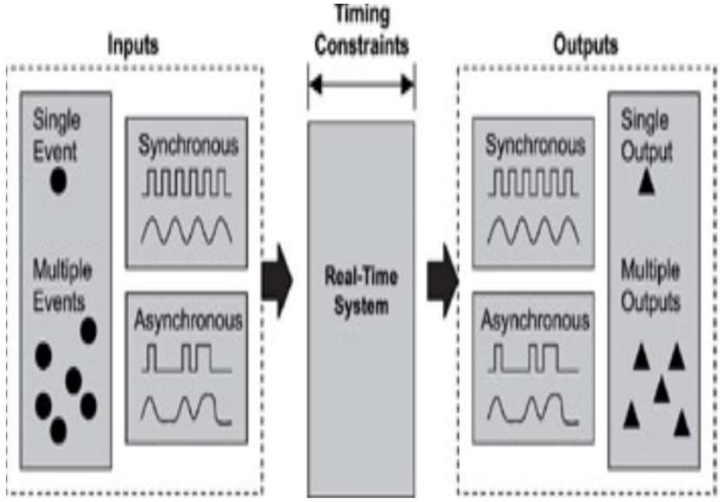
REAL TIME OPERATING SYSTEM (RTOS)

Topics Covered

- Overview
- Introduction RTOS
- Example of RTOS

Real-Time Embedded Systems

- In the simplest form, real-time systems can be defined as those systems that respond to external events in a timely fashion, where the response time is guaranteed.
- External events can have synchronous or asynchronous characteristics.
- Responding to external events includes recognizing when an event occurs, performing the required processing as a result of the event, and outputting the necessary results within a given time constraint.
- Timing constraints include finish time, or both start time and finish time.



A simple view of real-time systems.

Characteristics of Real-Time Systems

- The characteristics of real-time systems
  - must produce correct computational results, called *logical or functional correctness*, and
  - that these computations must conclude within a predefined period, called *timing correctness.*
  - Should have substantial knowledge of the environment of the controlled system and the applications running on it i.e. Real time system should be deterministic

Types of Real-Time Systems

- **Real-Time Systems may of two types :**
  - **Hard Real-Time Systems and**
  - **Soft Real-Time Systems**
- A ***hard real-time system*** is a real-time system that must meet its deadlines with a near-zero degree of flexibility. The deadlines must be met, or catastrophes occur. The cost of such catastrophe is extremely high and can involve human lives. The computation results obtained after the deadline have either a zero-level of usefulness or have a high rate of depreciation as time moves further from the missed deadline before the system produces a response.
- Timing correctness is critical to most hard real-time systems. Therefore, hard real-time systems make every effort possible in predicting if a pending deadline might be missed. For example Missie system
- A ***soft real-time system*** is a real-time system that must meet its deadlines but with a degree of flexibility. The deadlines can contain varying levels of tolerance, average timing deadlines, and even statistical distribution of response times with different degrees of acceptability. In a soft real-time system, a missed deadline does not result in system failure, but costs can rise in proportion to the delay, depending on the application.

A Brief History of Operating Systems

- In the early days of computing, **developers created software applications that included low-level machine code to initialize and interact with the system's hardware directly**. This tight integration between the software and hardware resulted in **non-portable applications**. A **small change in the hardware might result in rewriting** much of the application itself. Obviously, these **systems were difficult and costly to maintain**.
- As the software industry progressed, **operating systems that provided the basic software foundation for computing systems evolved and facilitated the abstraction of the underlying hardware from the application code**. In addition, the evolution of operating systems helped shift the design of software applications from large, monolithic applications to more modular, interconnected applications that could run on top of the operating system environment.

A Brief History of Operating Systems

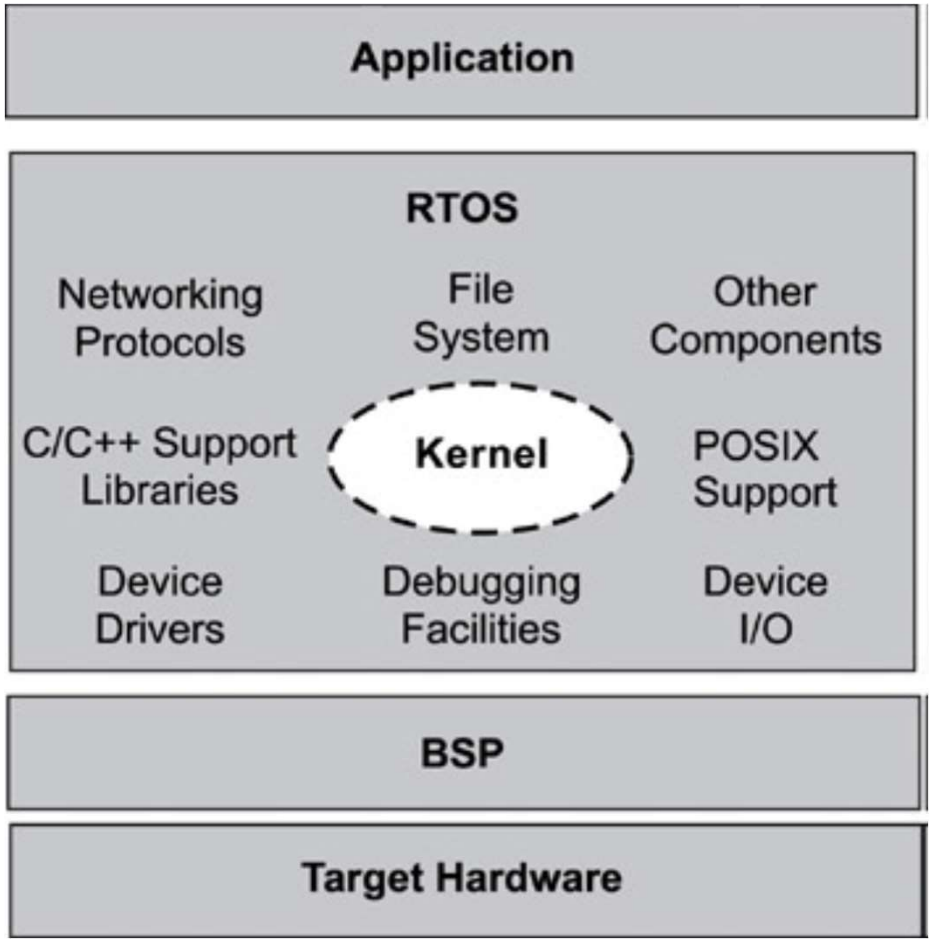
- In the **60s and 70s**, when **mid-sized and mainframe computing was in its prime, UNIX** was developed to facilitate multi-user access to expensive, limited-availability computing systems. UNIX allowed many users performing a variety of tasks to share these large and costly co
- In the **80s**, **Microsoft introduced the Windows operating system**, which emphasized the personal computing environment. Targeted for residential and business users interacting with PCs through a graphical user interface, the Microsoft Windows operating system helped drive the personal-computing era.
- **Later in the decade**, momentum started building for the next generation of computing: the post-PC, **embedded-computing** era. To meet the needs of embedded computing, **commercial RTOSes, such as VxWorks, were developed**.

Core functionality of RTOS and GPOS

- **Some core similarity of RTOS and GPOS are**
  - some level of multitasking,
  - software and hardware resource management,
  - provision of underlying OS services to applications, and
  - abstracting the hardware from the software application.
- **Some other advantages of RTOS over GPOS are**
  - gives better reliability in embedded application contexts,
  - the ability to scale up or down to meet application needs,
  - faster performance,
  - reduced memory requirements,
  - scheduling policies tailored for real-time embedded systems,
  - support for diskless embedded systems by allowing executables to boot and run from ROM or RAM,
  - better portability to different hardware platforms.

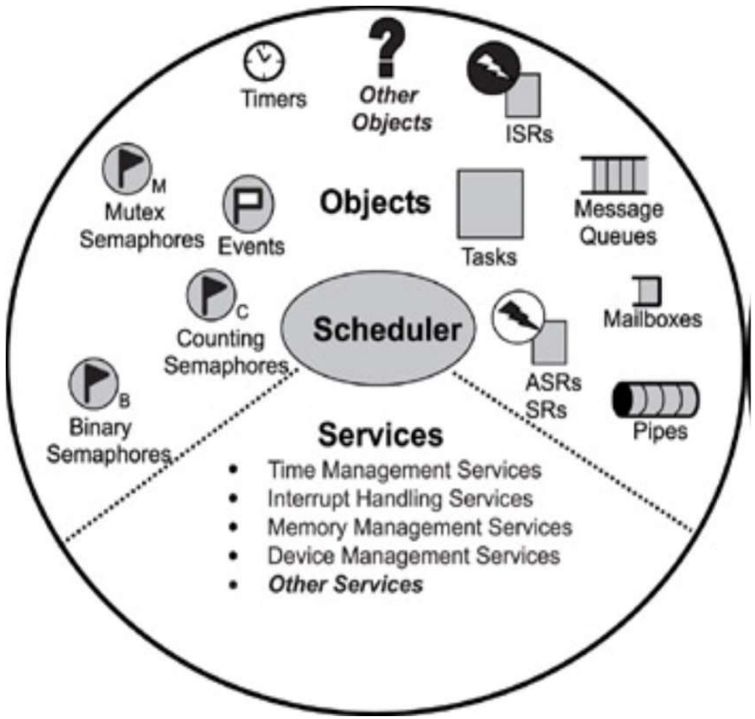
What is RTOS?

- A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code. Application code designed on an RTOS can be quite diverse, ranging from a simple application for a digital stopwatch to a much more complex application for aircraft navigation. Good RTOSes, therefore, are scalable in order to meet different sets of requirements for different applications.
- The heart of RTOS is the **KERNEL**



Kernel Components

- Although many RTOSes can scale up or down to meet application requirements, most RTOS kernels contain the following components:
- **Scheduler**-is contained within each kernel and follows a set of algorithms that determines which task executes when. Some common examples of scheduling algorithms include round-robin and preemptive scheduling.
- **Objects**-are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues.
- **Services**-are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management.



The Scheduler

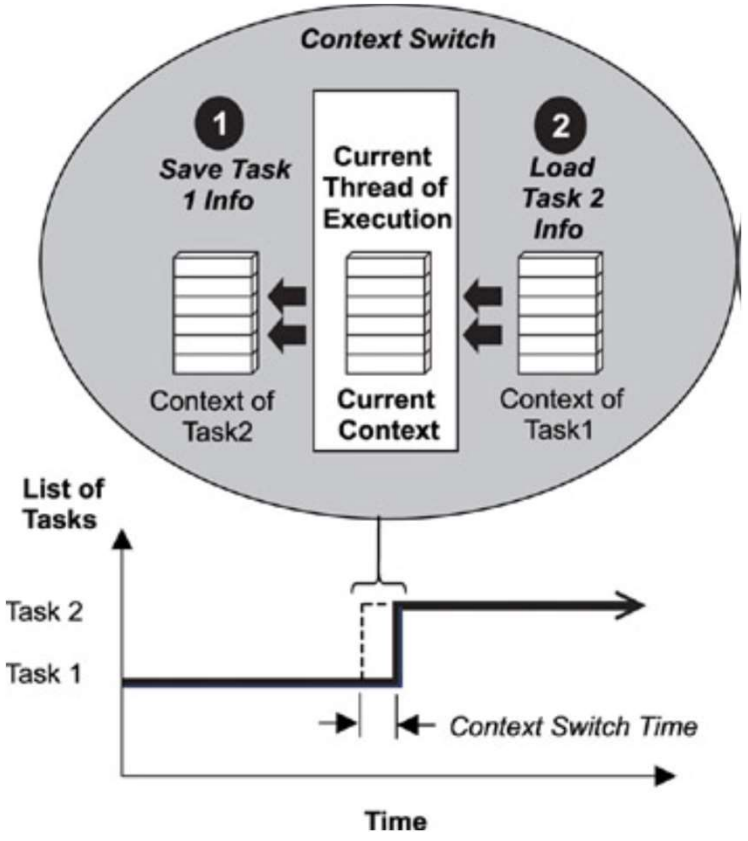
- The scheduler is at the heart of every kernel. A scheduler provides the algorithms needed to determine which task executes when. Let us discuss following point of the scheduler:
- schedulable entities,
- multitasking,
- context switching,
- dispatcher, and
- scheduling algorithms.

Schedulable Entities

- A schedulable entity is a kernel object that can compete for execution time on a system, based on a predefined scheduling algorithm. Tasks and processes are all examples of schedulable entities found in most kernels.
- A task is an independent thread of execution that contains a sequence of independently schedulable instructions.
- Some kernels provide another type of a schedulable object called a process. Processes are similar to tasks in that they can independently compete for CPU execution time. Processes differ from tasks in that they provide better memory protection features, at the expense of performance and memory overhead. Despite these differences, for the sake of simplicity, this book uses task to mean either a task or a process.
- So, how exactly does a scheduler handle multiple schedulable entities that need to run simultaneously? The answer is by multitasking. The multitasking discussions are carried out in the context of uniprocessor environments.

Multitasking

- Multitasking is the ability of the operating system to handle multiple activities within set deadlines. A real-time kernel might have multiple tasks that it has to schedule to run as shown in figure.
- In this scenario, the kernel multitasks in such a way that many threads of execution appear to be running concurrently; however, the kernel is actually interleaving executions sequentially, based on a preset scheduling algorithm. The scheduler must ensure that the appropriate task runs at the right time.
- the tasks follow the kernel s scheduling algorithm, while interrupt service routines (ISR) are triggered to run because of hardware interrupts and their established priorities.
- As the number of tasks to schedule increases, so do CPU performance requirements. This fact is due to increased switching between the contexts of the different threads of execution.



Multitasking using a context switch.

The Context Switch

- Each task has its own context, which is the state of the CPU registers required each time it is scheduled to run. A context switch occurs when the scheduler switches from one task to another.
- Every time a new task is created, the kernel also creates and maintains an associated task control block (TCB). TCBs are system data structures that the kernel uses to maintain task-specific information. TCBs contain everything a kernel needs to know about a particular task. When a task is running, its context is highly dynamic. A typical context switch scenario is illustrated in previous slide.
- When the kernel s scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps:
  - The kernel saves task 1 s context information in its TCB.
  - It loads task 2 s context information from its TCB, which becomes the current thread of execution.
  - The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch.

The Dispatcher

- The dispatcher is the part of the scheduler that performs context switching and changes the flow of execution. At any time an RTOS is running, the flow of execution, also known as flow of control, is passing through one of three areas:
  - **through an application task,**
  - **through an ISR, or**
  - **through the kernel.**
- When a task or ISR makes a system call, the flow of control passes to the kernel to execute one of the system routines provided by the kernel. When it is time to leave the kernel, the dispatcher is responsible for passing control to one of the tasks in the user s application.
- Depending on how the kernel is first entered, dispatching can happen differently.
- When a task makes system calls, the dispatcher is used to exit the kernel after every system call completes which is done on a call-by-call basis so that it can coordinate task-state transitions that any of the system calls might have caused.
- On the other hand, if an ISR makes system calls, the dispatcher is bypassed until the ISR fully completes its execution. This process is true even if some resources have been freed that would normally trigger a context switch between tasks.

Scheduling Algorithms

- As mentioned earlier, the scheduler determines which task runs by following a scheduling algorithm (also known as scheduling policy). Most kernels today support two common scheduling algorithms:
  - preemptive priority-based scheduling, and
  - round-robin scheduling.
- **Real-time kernels generally support 256 priority levels**, in which 0 is the highest and 255 the lowest. **Some kernels appoint the priorities in reverse order**. Regardless, the concepts are basically the same. With a preemptive priority-based scheduler, each task has a priority, and the highest-priority task runs first. If a task with a priority higher than the current task becomes ready to run, the kernel immediately saves the current task s context in its TCB and switches to the higher-priority task.

Kernel objects

- Kernel objects are special constructs that are the building blocks for application development for real-time embedded systems. The most common RTOS kernel objects are
  - Tasks are concurrent and independent threads of execution that can compete for CPU execution time.
  - Semaphores are token-like objects that can be incremented or decremented by tasks for synchronization or mutual exclusion.
  - Message Queues are buffer-like data structures that can be used for synchronization, mutual exclusion, and data exchange by passing messages between tasks. Developers creating real-time embedded applications can combine these basic kernel objects (as well as others not mentioned here) to solve common real-time design problems, such as concurrency, activity synchronization, and data communication. These design

Services

- Along with objects, most kernels provide services that help developers create applications for real-time embedded systems. These services comprise sets of API calls that can be used to perform operations on kernel objects or can be used in general to facilitate timer management, interrupt handling, device I/O, and memory management. Again, other services might be provided; these services are those most commonly found in RTOS kernels.

Key Characteristics of an RTOS

- An application's requirements define the requirements of its underlying RTOS. Some of the more common attributes are
  - reliability,
  - predictability,
  - performance,
  - compactness, and
  - scalability.

Reliability

- Embedded systems must be reliable. Depending on the application, the system might need to operate for long periods without human intervention.
- Different degrees of reliability may be required. For example, a digital solar-powered calculator might reset itself if it does not get enough light, yet the calculator might still be considered acceptable. On the other hand, a telecom switch cannot reset during operation without incurring high associated costs for down time. The RTOSes in these applications require different degrees of reliability.
- Although different degrees of reliability might be acceptable, in general, a reliable system is one that is available (continues to provide service) and does not fail.
- While RTOSes must be reliable, note that the RTOS by itself is not what is measured to determine system reliability. It is the combination of all system elements-including the hardware, BSP, RTOS, and application-that determines the reliability of a system.

Predictability Performance

- The RTOS used should be **predictable to a certain degree.** The term deterministic describes RTOSes with predictable behavior, in which the completion of operating system calls occurs within known timeframes. Developers can write simple benchmark programs to validate the determinism of an RTOS i.e. the variance of the response times for each type of system call is very small.

- **Performance of** an embedded system must perform fast enough to fulfill its timing requirements. Typically, the more deadlines to be met-and the shorter the time between them-the faster the system's CPU must be. Typically, the processor's performance is expressed in million instructions per second (MIPS).

- Sometimes developers measure RTOS **performance on a call-by-call basis**. Benchmarks are written by producing timestamps when a system call starts and when it completes. Although this step can be helpful in the analysis stages of design, true performance testing is achieved only when the system performance is measured as a whole.

Compactness

- Design constraints and cost constraints help determine how compact an embedded system can be.

- For example, a cell phone clearly must be small, portable, and low cost.
- These design requirements limit system memory, which in turn limits the size of the application and operating system also such embedded systems, the hardware real estate is limited due to size and costs.

- In these cases, the RTOS memory footprint can be an important factor.

- To meet total system requirements, designers must understand both the static and dynamic memory consumption of the RTOS and the application that will run on it.

Scalability

- Because RTOSes can be used in a wide variety of embedded systems, they must be able **to scale up or down to meet application-specific** requirements i.e. an RTOS should be capable of adding or deleting modular components, including file systems and protocol stacks.

- **If an RTOS does not scale up well**, development teams might have to buy or build the missing pieces. Suppose that a development team wants to use an RTOS for the design of a cellular phone project and a base station project.

- **If an RTOS scales well**, the same RTOS can be used in both projects, instead of two different RTOSes, which saves considerable time and money.

Tasks

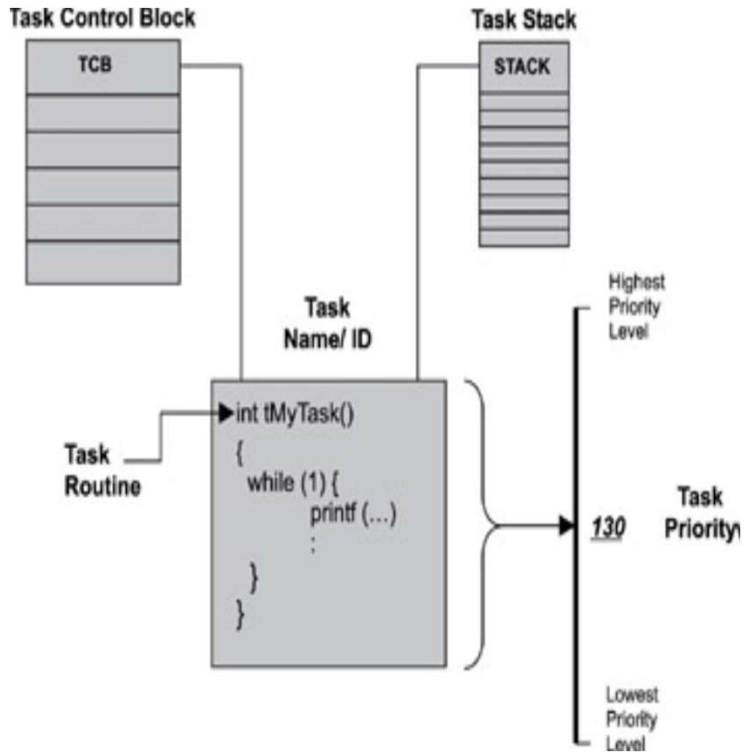
- Simple software applications are typically designed to run sequentially , one instruction at a time, in a pre-determined chain of instructions. However, real-time embedded applications generally **handle multiple inputs and outputs within tight time constrai**nts. Real-time embedded software **applications must be designed for concurrency**.
- ***Concurrent*** *design* requires developers to decompose an application into small, schedulable, and sequential program units. Most RTOS kernels provide task objects and task management services to facilitate designing concurrency within an application.

- ***A task is an independent thread of execution that can compete with other concurrent tasks for processor execution time. Developers decompose applications into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints.***

Task

- A task is schedulable i.e. the task is able to compete for execution time on a system, based on a predefined scheduling algorithm.
- A task is defined by its distinct set of parameters and supporting data structures.
- Specifically, upon creation, each task has an associated
  - name,
  - a unique ID,
  - a task control block (TCB),
  - a stack, and
  - a task routine

These components together called as **task objects**.



Task

- When the kernel first starts, it creates its own set of system tasks and allocates the appropriate priority for each from a set of reserved priority levels.
- The reserved priority levels refer to the priorities used internally by the RTOS for its system tasks. For most RTOSes, these reserved priorities are not enforced. The kernel needs its own system tasks and their reserved priority levels to operate. These priorities should not be modified.
- Examples of system tasks include:
  - **initialization or startup task** initializes the system and creates and starts system tasks,
  - **idle task** uses up processor idle cycles when no other activity is present,
  - **logging task** logs system messages,
  - **exception-handling task** handles exceptions, and
  - **debug agent task** allows debugging with a host debugger.

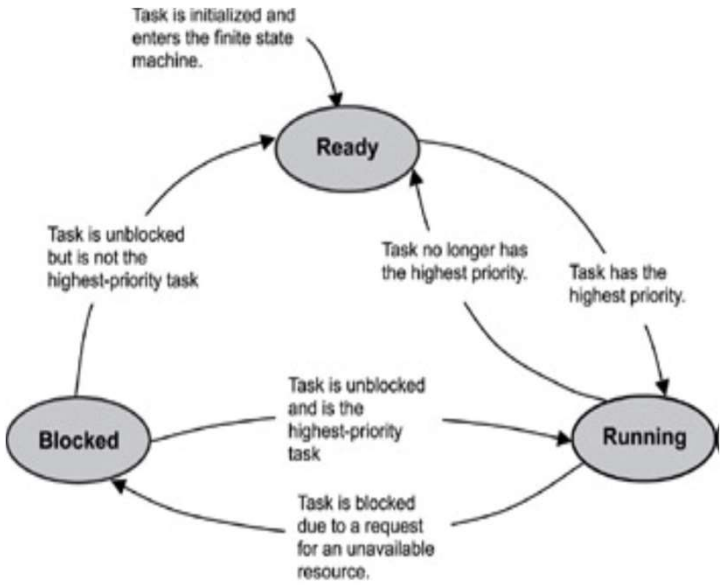
Task

- **The idle task is created at kernel startup with lowest priority**, runs when either no other task can run or when no other tasks exist, for the sole purpose of using idle processor cycles.
- **The idle task executes the instruction to which the program counter register points while it is running**.
- The processor can be **suspended**, the **program counter must still point to valid instructions even when no tasks exist** in the system or when no tasks can run.
- Therefore, **the idle task ensures the processor program counter is always valid** when no other tasks are running.
- **the kernel can be allowed a user-configured routine to run instead of the idle task** in order to implement special requirements for a particular application.
- **After the kernel has initialized and created all of the required tasks**, the **kernel jumps to a predefined entry point that serves**, in effect, as the beginning of the application.

Task States and Scheduling

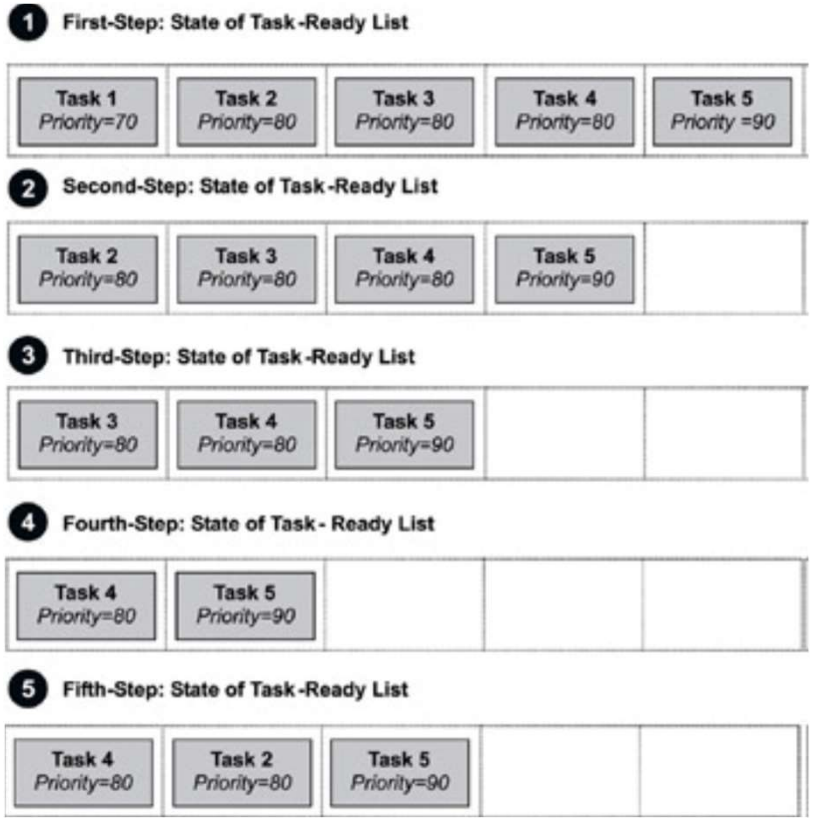
- At any time, each task exists in one of a small number of states, including ready, running, or blocked. As the real-time embedded system runs, each task moves from one state to another, according to the logic of a simple finite state machine (FSM) as shown in figure.

- Although kernels can define task-state groupings differently, few basic states are including:
  - **ready state**-the task is ready to run but cannot because a higher priority task is executing.
  - **blocked state**-the task has requested a resource that is not available, has requested to wait until some event occurs, or has delayed itself for some duration.
  - **running state**-the task is the highest priority task and is running.



Ready State

- When a task is first created and made ready to run, the kernel puts it into the ready state.
- In this state, the task actively competes with all other ready tasks for the processor's execution time.
- For a kernel that supports only one task per priority level, the scheduling algorithm is straightforward-the highest priority task that is ready runs next.
- However, most kernels support more than one task per priority level, allowing many more tasks in an application. In this case, the scheduling algorithm is more complicated and involves maintaining a *task-ready list*.



Running State

- On a single-processor system, one task run at a time and when the task is moved to the running state, the processor loads its registers with this task's context.
- A task can move back to the ready state while it is running. When a task moves from the running state to the ready state, it is preempted by a higher priority task. In this case, the preempted task is put in the appropriate, priority-based location in the task-ready list, and the higher priority task is moved from the ready state to the running state.
- A running task can move to the blocked state in any of the following ways:
  - by making a call that requests an unavailable resource,
  - by making a call that requests to wait for an event to occur, and
  - by making a call to delay the task for some duration.

Blocked State

- In a real time system, both lowest priority and highest priority are used and higher priority task are blocked to allow lowest priority task by blocking higher priority tasks. Otherwise CPU starvation can result due to which higher priority tasks use all of the CPU resources
- . A task can only move to the blocked state by making a blocking call upon getting blocking request. When a blocking conditions are met include the following:
  - a semaphore token (described later) for which a task is waiting is released,
  - a message, on which the task is waiting, arrives in a message queue, or
  - a time delay imposed on the task expires.
- When a task becomes unblocked, the task might move from the blocked state to the ready state if it is not the highest priority task otherwise it moves to running state.

Task Management

- In addition to providing a task object, kernels also provide *task-management services which* can perform
  - **creating** : task creation makes context switching and clean-up task on deleting.
  - **deleting tasks: task deletion happens on** a corrupt data structure, due to an incomplete write operation, an unreleased semaphore, an inaccessible data structure,
  - **controlling task scheduling**: From the time a task is created to the time it is deleted, the task can move through various states resulting from program execution and kernel scheduling
  - **obtaining task information**: gets the information on curremt task ID and TCB..

Synchronization, Communication, and Concurrency

- Tasks synchronize and communicate amongst themselves by using inter task primitives , which are kernel objects that facilitate synchronization and communication between two or more threads of execution.

- The concept of concurrency and how an application is optimally decomposed into concurrent tasks. For now, remember that the task object is the fundamental construct of most kernels.

- Tasks, along with task-management services, allow developers to design applications for concurrency to meet multiple time constraints and to address various design problems inherent to real-time embedded applications.

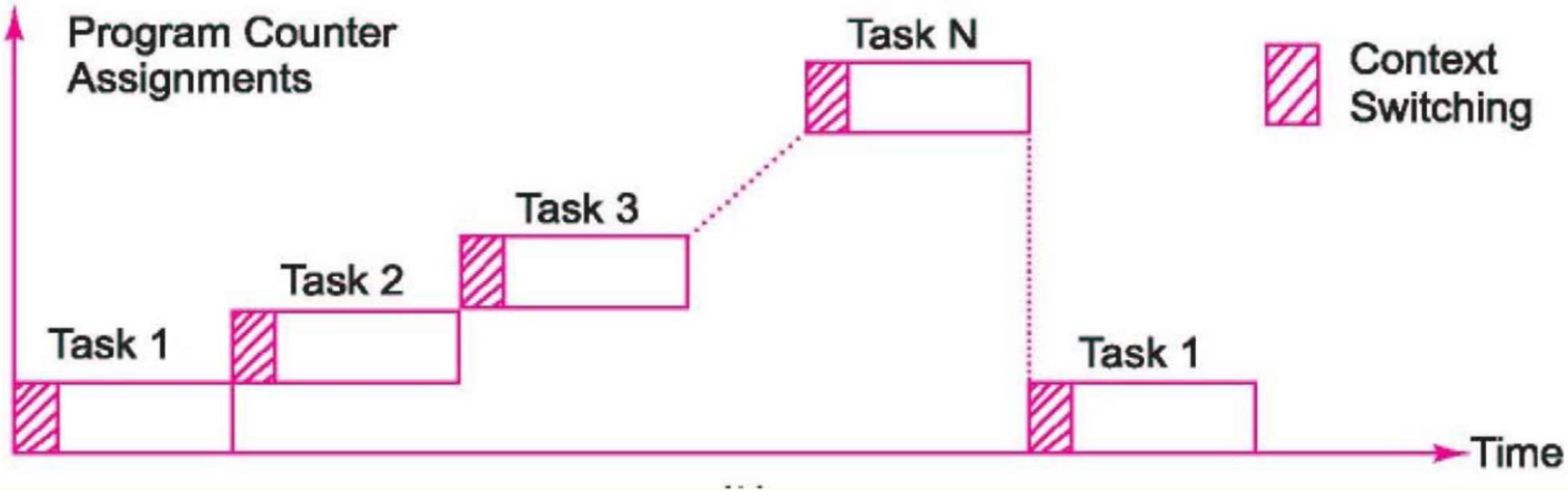
Task Scheduling Cooperative Models

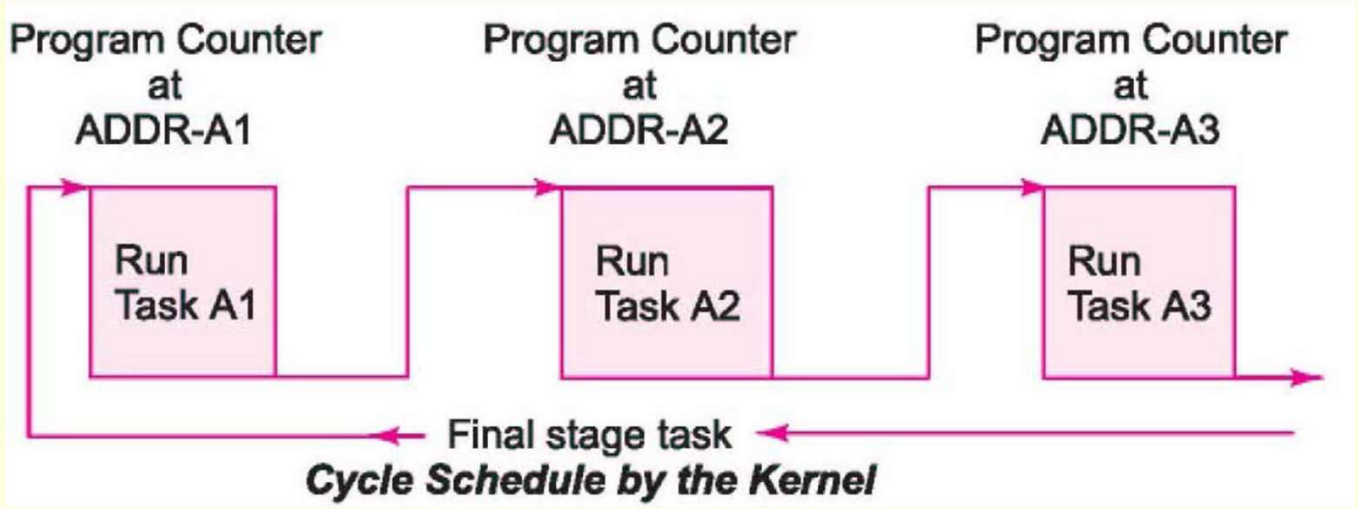
- Common scheduling models
  - Cooperative Scheduling of ready tasks in a circular queue. It closely relates to function queue scheduling.
  - Cooperative Scheduling with Precedence Constraints
  - Cyclic Scheduling of periodic tasks and Round Robin Time Slicing Scheduling of equal priority tasks
  - 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 aperiodic Tasks
  - Advanced scheduling algorithms using the probabilistic Timed Petri nets (Stochastic) or Multi Thread Graph for the multiprocessors and complex distributed systems.

Cooperative Scheduling in the cyclic order

- Each task cooperate to let the running task finish
- Cooperative means that each task cooperates to let the a running one finish.
- None of the tasks does block in-between anywhere during the ready to finish states.
- The service is in the cyclic order
- Same for every task
- Tworst = {(sti + eti)1 + (sti + eti)2 +...+ (sti + eti)N-1 + (sti + eti)N} + tISR.
- tISR 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
- i = 1, 2, …, N 1 , N, when number of tasks = N

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.

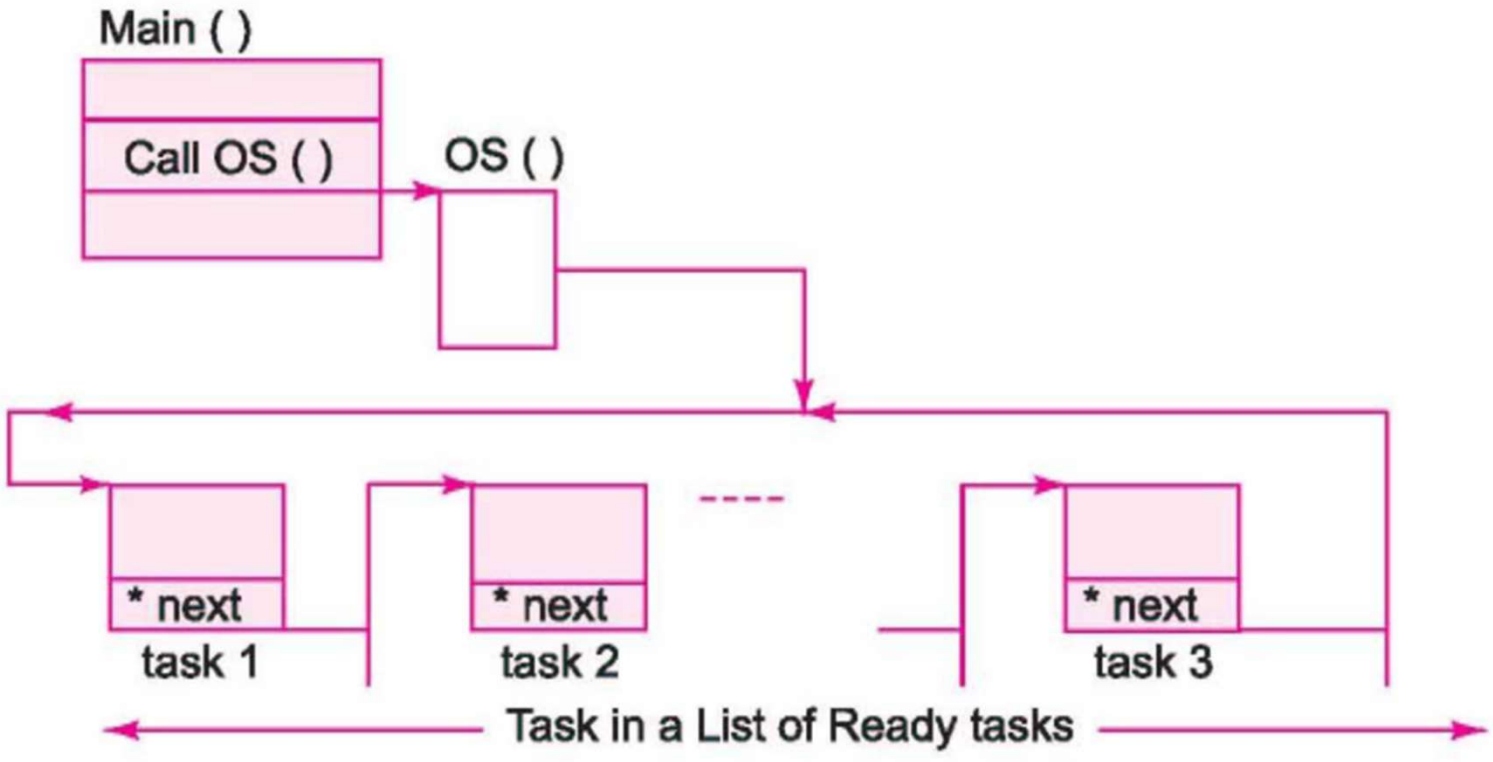




Cycle Schedule by the Kernel

Cooperative Scheduling of Ready Tasks in List

- 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.
- Same for every task in the ready list
- Tworst = {(dti + sti + eti)1 + (dti + sti + eti)2
- +...+ (dti + sti + eti)n-1 + (dti + sti + eti)n} + tISR.
- tISR 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
- i = 1, 2, …, n 1 , n


Main ( )
Call OS ( )
OS ( )
* next
task 1
* next
task 2
* next
task 3
Task in a List of Ready tasks

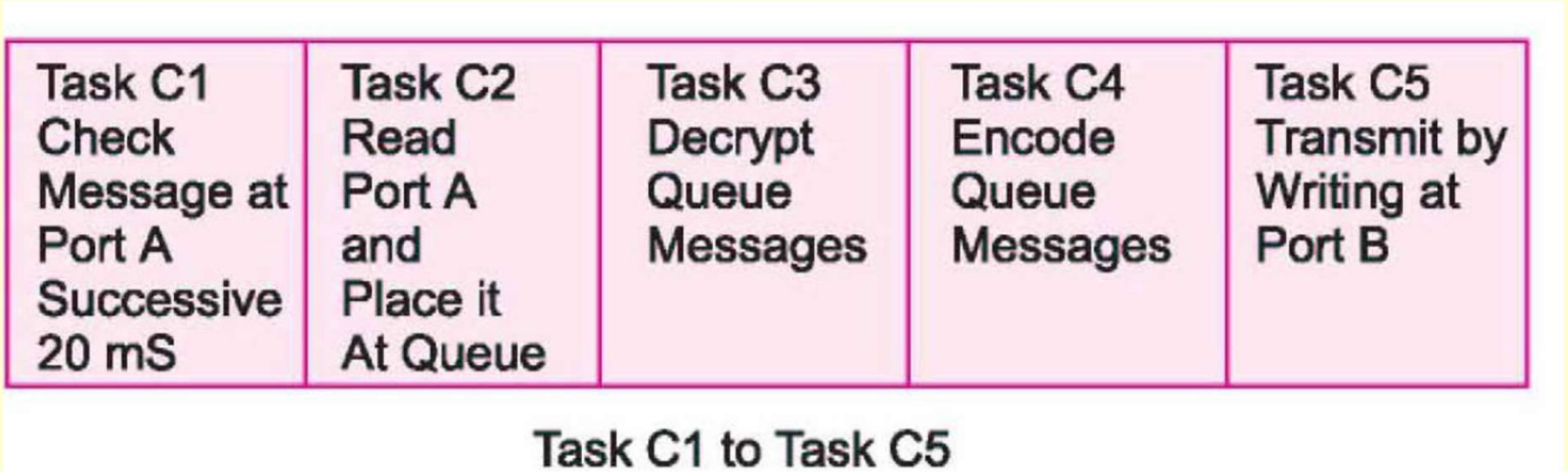
Cyclic Scheduling Periodic tasks

- Assume periodically occurring three tasks
- Let in time-frames allotted to the first task, the task executes at t1, t1 + T_{cycle}, t1+ 2× T_{cycle}, .., second task frames at t2, t2 + T_{cycle}, t2+ 2 × Tcycle and third task at t3, t3 + T_{cycle}, t3+ 2 × T_{cycle}, … .
- Start of a time frame is the scheduling point for the next task in the cycle.
- T_{cycle} 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.
- T_{cycle} is period after which each task time frame allotted to that repeats

Case : t_{cycle} = N × Sum of the maximum times for each task

- Then each task is executed once and finishes in one cycle itself.
- When a task finishes the execution before the maximum time it can takes, there is a waiting period in-between period between two cycles.
- The worst-case latency for any task is then N × Sum of the maximum times for each task. A task may periodically need execution. A task period for the its need of required repeat execution of a task is an integral multiple of t_{cycle}.

Tasks C1 to C5 Cyclic Scheduling



Example of Video and audio signals

- Signals reaching at the ports in a multimedia system and processed
- The video frames reach at the rate of 25 in one second.
- The cyclic scheduler is used in this case to process video and audio with T_{cycle} = 40 ms or in multiples of 40 ms.

Orchestra Playing Robots

- First the director robot sends the musical notes. Then, the playing robots receive and acknowledge to the director.
- In next cycle, the master robot again sends the musical notes.
- The cyclic scheduler is used in this case to send and receive signals by each robot after the cycle period

Round Robin Time Slice Scheduling of Equal Priority Tasks

- **Equal Priority Tasks**
  - Round robin means that each ready task runs turn by in turn only in a cyclic queue for a limited time slice.
  - 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

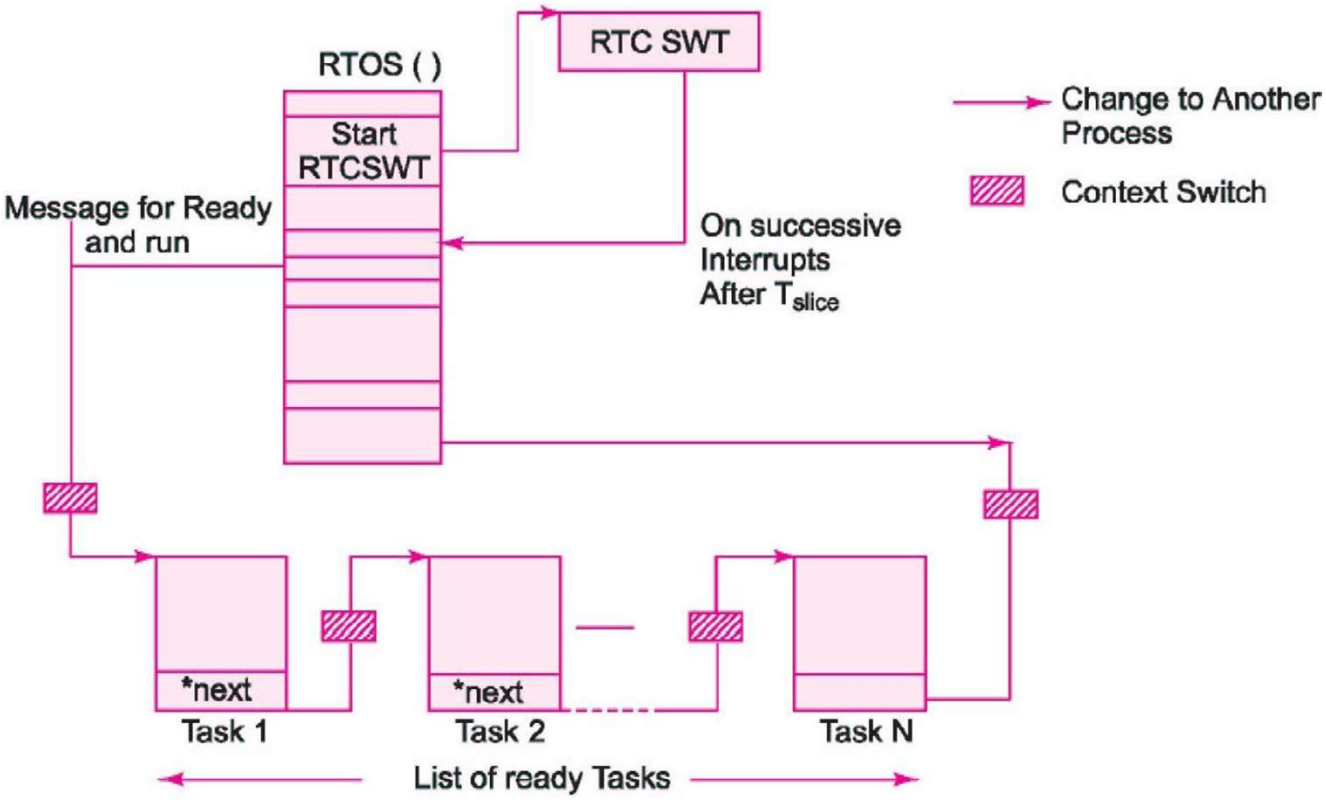
Blocked after Saving Context ?

Running —

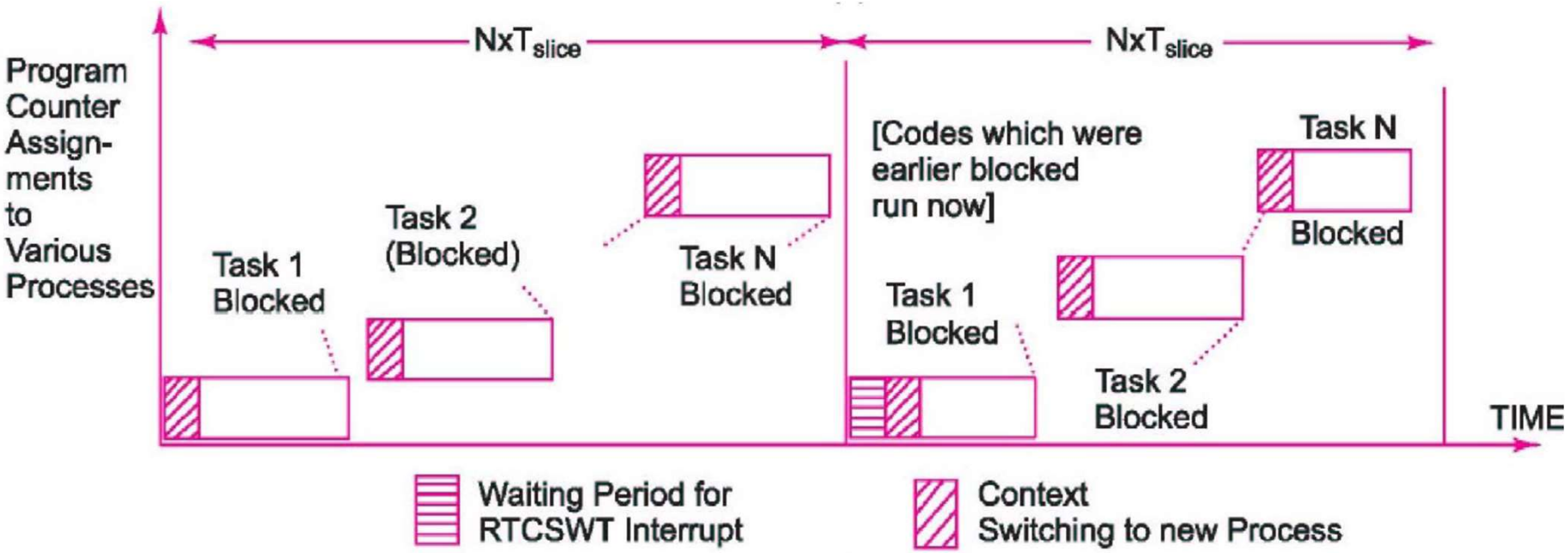
Finished ✓

Time Slicing Scheduling by the RTOS Kernel

Programming model for the Cooperative Time sliced scheduling of the tasks



Program counter assignments on the scheduler call to tasks at two consecutive time slices. Each cycle takes time = $N \times t_{slice}$



Case : $T_{cycle} = N \times T_{slice}$

- Same for every task = T_{cycle}
- $T_{cycle} = \{T_{slice}\ )\} \times N + t_{ISR}$.
- 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

Worst-case latency

- Same for every task in the ready list
- $T_{worst} = \{N \times (T_{slice})\} + t_{ISR}$.
- t_{ISR} is the sum of all execution times for the ISRs
- Where i = 1, 2, …, N-1 , N

VoIP Tasks Example

- Assume a VoIP [Voice Over IP.] router.
- It routes the packets to N destinations from N sources.
- It has N calls to route.
- Each of N tasks is allotted from a time slice and is cyclically executed for routing packet from a source to its destination

Round Robin

- Case 1: when each task is executed once and finishes in one cycle itself.
- When a task finishes the execution before the maximum time it can takes, there is a waiting period in-between period between two cycles.
- The worst-case latency for any task is then N ×tslice. A task may periodically need execution. A task, the period for the its need of required repeat execution of a task is an integral multiple of tslice

Case 2: 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 the 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

- 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.