Real Time Operating Systems
Shirvaikar Chapter 4

Real Time Systems Design

- Various design approaches implemented by system designers to meet real-time requirements
- Three general approaches to task scheduling:
  - *ad-hoc scheduling*
  - *deterministic scheduling* using a cyclic executive
  - non-deterministic, *priority-driven scheduling* using a multitasking executive
- Evolved over time with successively more sophisticated approaches culminating in a full-fledged real-time operating system

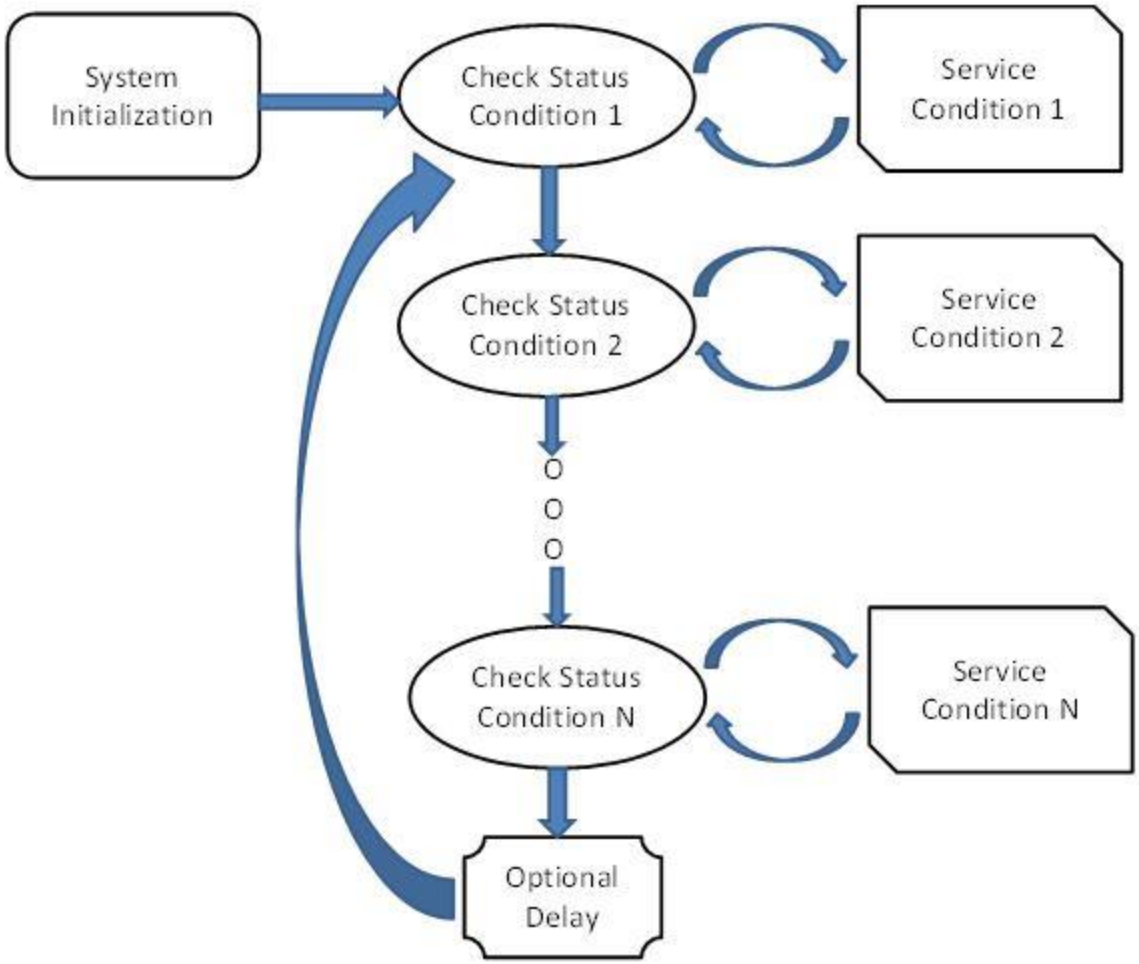
Basic Solutions

- *Ad-hoc scheduling* is the simplest form of task scheduling, if it can be called task scheduling at all
- For straightforward processing with no specific periodic requirements, it provides satisfactory results
  - Tasks are functional program units
  - Dependencies are limited to precedence relationships
  - Not periodic since repetition is not at precise rate
- Program may be essentially written as an endless loop with each task executing to completion in some predetermined sequence

Polled-Loop Systems

- Poll devices attached to the system continuously
- Keeps checking for a service request event, which can be a flag or signal in the software
- Sometimes a small delay is introduced between successive attempts at checking the status to achieve "*debouncing*" (if event is slow in nature)
- Event flag is typically cleared upon servicing so that the system is ready for the next event or "*burst*"

Polled-Loop Systems



Polled-Loop Systems

- Simplest form of a real-time kernel
- Simple to design and debug, as it does not require interrupts
- Fast reaction to single events that require a guaranteed response time

```
loop      {                                   /* do forever     */
if (packet_here)                      /* check flag     */
          {
                    packet_here=0;    /* reset flag     */
                    process_data();   /* process data */
          }
}
```

Polled-Loop Systems

- Dedicated to the status loop, thereby making it impossible to do other tasks or even enter a "*power-saving*" or sleep mode

- Not possible to guarantee that the peripherals will be serviced in the correct order or priority level

The Cyclic Executive

- One of the most common approaches used when the system specifications require strict periodicity is a priority-driven scheduling solution known as the *cyclic executive*
  - deterministic scheduling
  - low implementation overhead
  - control software is typically small and fast
- Tasks are scheduled for execution by an interrupt driven control program called a cyclic executive.

The Cyclic Executive

- Works extremely well for systems that are
  - Periodic - with a high periodic content
  - Predictable - the tasks have predictable execution times
  - Synchronous - have relatively few asynchronous requirements
  - Deterministic – fixed execution sequence requirements

The Cyclic Executive

- Inertial navigator
  - instruments provide data at precise periodic rates
  - data must be processed prior to the arrival of the following data
  - asynchronous events, such as control panel entries by the operator, are relatively rare
  - execution times of the various tasks are fairly predictable, since they involve about the same amount of calculation each time they are executed

The Cyclic Executive

- Tasks are scheduled for execution by an interrupt driven control program called a cyclic executive.
- It has a scheduler that bases its timing upon a hardware timer subsystem that generates periodic interrupts.
- Such timer subsystems are an integral part of the embedded computer or CPU, and the interrupts correspond to *ticks of the real-time clock*.

The Cyclic Executive

- One of the major differences from ad-hoc scheduling is that task scheduling in a cyclic executive system is periodic (time-based) rather than repetitive.
- One or more tasks are scheduled at a precise periodic rate, rather than being based on completion of the prior task in the sequence of tasks as seen in some of the previous "*state machine*" based approaches.

The Cyclic Executive

- Asynchronous events are handled by:
  - interrupt processing
  - background processing, or
  - special periodic server tasks
- The scheduling of tasks in a cyclic executive system is completely deterministic except for the asynchronous event processing requirements.

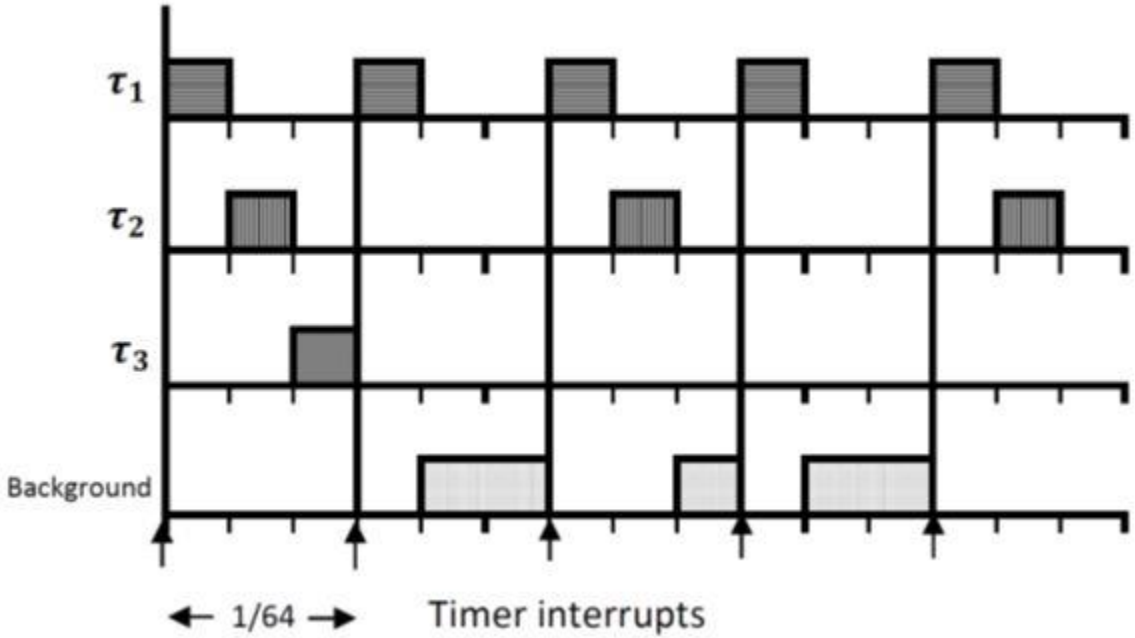
The Cyclic Executive

- Two types of timing requirements :
  - Periodic - tasks that must be performed on a periodic basis
  - Deadline - tasks that, when required, must be completed within a certain time interval
- A periodic timing requirement will have a deadline that is typically the end of the period, or in some cases earlier if the task is required to be completed within an interval shorter than the period

The Cyclic Executive

- The underlying idea is that the cyclic executive design is a loop, or *cycle*, synchronized by timer-driven interrupts.
- The order in which tasks are scheduled by such a cyclic executive is *predetermined*, with each task having control of the processor (and other resources) for a fixed time period.

The Cyclic Executive



Example 4.2.1: A simple cyclic executive schedule.

The highest rate task, τ_1, executes at 64 Hz. It is scheduled as a result of periodic timer-driven interrupts received by the system.

The Cyclic Executive

Task τ_2 executes at 32 Hz. and is scheduled as a result of every other completion of task τ_1 .

Task τ_3 is scheduled as a result of every other completion of task τ_2 . In this simple example, the highest frequency task τ_1 is therefore scheduled by the timer driven interrupt, and the lower rate tasks are based upon the completion of τ_1.

It is possible that periodic scheduling of tasks will result in some unused processor time, known as *idle time*.

This excess time is typically used for low priority *background processing*. Examples of low priority functions are diagnostics, self-checks, and other non-periodic processing.

Processor Utilization

- *Processor utilization* $\boldsymbol{U}$ for the single processor case for a system with N periodic tasks,

$$U = \sum_{n=1}^{N} \frac{e_n}{p_n} \leq 1$$

where e_n is the execution time for each task and p_n is the period of each task.

the inequality is a *necessary but not sufficient* condition for a task system to be feasible on a single processor (does not model asynchronous tasks)

Processor Utilization

Assuming each unit of time on the x-axis is equal to $1/3 * 64$ seconds without loss of generality):

$$N = number\ of\ tasks = 3$$

Period	Deadline	Execution Time
$p_1 = 3$	$d_1 = 3$	$e_1 = 1$
$p_2 = 6$	$d_2 = 6$	$e_2 = 1$
$p_3 = 24$	$d_3 = 24$	$e_3 = 1$

The processor utilization is determined to be

$$U = \frac{1}{3} + \frac{1}{6} + \frac{1}{24} = 0.542 < 1.$$

Processor Utilization

The value can provide an excellent insight into how "stressed" the system is and whether it will meet deadlines consistently. Based on empirical observations it has been observed that systems:

$0.00 - 0.25$	Underutilized
$0.25 - 0.69$	Safe
$0.69 - 0.83$	Questionable (depends on number of tasks)
$0.83 - 0.99$	Dangerously poised
$0.99 - 1.00$	Marginally stressed or critical
$1.00\ or\ more$	Definitely overloaded or stressed

The Concurrently Executing Task

- RTOS must provide for the creation, deletion, preemption and monitoring of tasks or threads

- The term "*thread*" is increasingly used instead of the term "*task*" to describe a distinct sequence of operations, since the use of multicore processors has become commonplace over the last few years

The Concurrently Executing Task

- Storage space requirements for the context of a task must be specified and provision must be made for multiple instances of the same task
- Other properties such as the task identifier and the task priority must also be maintained
- It is convenient to maintain such information about a task or thread in a data structure or object - *task control block*, or TCB or Thread Definition Structure (TDS).

The Concurrently Executing Task

- Specific implementation varies with RTOS but typically TCB or TDS will be in a section of memory with restricted access
- Possible implementation of the data structure
  - a *register save area*, in which the context of the task can be saved
  - a number of link fields to implement the various functions by forming various linked lists used in task scheduling and in the implementation of operators such as semaphores

The Concurrently Executing Task

TDS – Thread Definition Structure

```
Struct osThreadDef {
Function pointer *pthread;
Priority_level    priority;
Uint32_t          instances;
Uint32_t          stacksize;
};
```

TCB – Task Control Block

```
Struct osTaskControlBlock {
String            TASK_NAME;
Uint32_t          priority;
Uint32_t          STATE;
Time_t            ACTIVATION_TIME;
Char *Register_Save;
};
```

The Concurrently Executing Task

- The various fields of a simple TDS are :
  - THREAD_ADDRESS – the pointer to the function which implements the thread
  - THREAD_PRIORITY – the initial thread priority which may remain fixed or change during the system operation based on static or dynamic priority assignment
  - INSTANCES – the maximum number of concurrent instances of the thread function allowed
  - STACKSIZE – the stack size requirements in bytes for the thread (0 is default stack size)

The Concurrently Executing Task

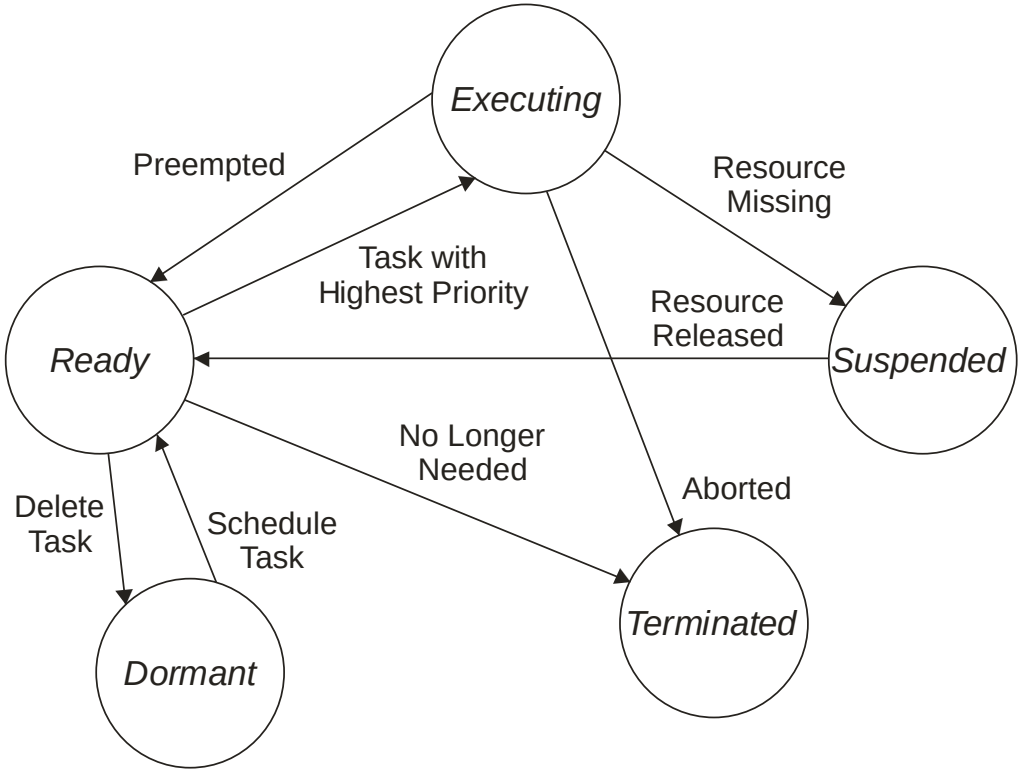
- The fields of a typical TCB are listed below:
  - TASK_NAME – the name of the task that acts as an identifier for starting the task, suspending the task, or performing some other operation affecting the task
  - PRIORITY – the priority of the task which may remain fixed or change during the system operation based on static or dynamic priority assignment
  - Register save area – the area where the processor register contents are stored upon task suspension

The Concurrently Executing Task

- The fields of a typical TCB are listed below:
  - STATE – the state of the task typically
    - *running* (currently has the necessary resources and is executing)
    - *suspended* (currently blocked from execution awaiting action)
    - *ready*, (not blocked from execution, but waiting for resources necessary for execution)
  - ACTIVATION_TIME – the time instance in the future when the task will be activated

The Concurrently Executing Task

- A process state diagram as a partially defined finite state machine



The System Timer

- RTOS - one of the most important tasks is to maintain accurate time (not world time for which there are special chips)
- Rather it is a measure of elapsed time between events
- Special timer sub-system that can be set to interrupt the operating system at regular intervals (timer interrupt based on clock)

The System Timer

- Typically this value can range from 1ms to 100ms (commonly 10ms)
- The clock tick interrupt can be viewed as the system's heartbeat
- Granularity – system need vs. overhead
- The "*real-time clock*" – supported in hardware by a peripheral - on or off chip

The System Timer

- Most of the modern ARM processor cores
  - have an additional component on them known as the "*SysTick*" timer
  - Implemented as simple increment or decrement counters that produce a hardware interrupt or system exception that must be handled by the CPU
  - RTOS treats this exception as high-priority and the "*scheduler*" part of the RTOS is invoked at this time
  - It makes decisions about system level issues such as which task should run in the next time slice

The System Timer

- System timer module as a part of CPU core
  - allows software portability across ARM's array of product offerings
  - convenience of operation for real-time applications
  - important tasks
    - Calibration
    - Task time profiling during debugging, and
    - Measurement of time between events

The System Timer

- Some important issues must be kept in mind while using the system timer. Only the operating system should be allowed to change the system timer
- Application programs should be allowed to read the timer value via a protected mode
- This functionality for a privileged mode exists on certain processors

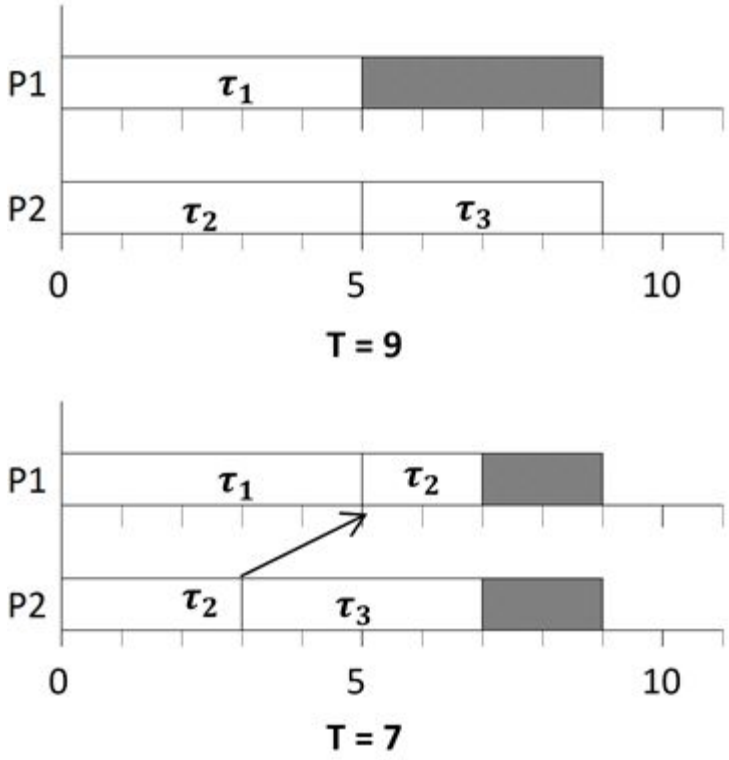
Preemptive Scheduling

- Task "*preemption*" and is one of the powerful tools that allows the RTOS to meet deadlines imposed by scheduling constraints
- Preemption results in the suspension of the currently executing task in order to permit another task to execute (after context switching of course)

Preemptive Scheduling

- The concept is powerful - applied in dynamic run-time environment, and extended to the design of deterministic schedules by arbitrarily dividing a fixed task into subtasks
- The goal, of course, is to facilitate the implementation of an effective task schedule that meets all deadlines
- The advantage is even more apparent with multi-core and multi-threaded systems

Preemptive Scheduling



Preemptive Scheduling

- Static and dynamic task scheduling algorithms do not have guaranteed closed form solutions under most normal operating conditions and can sometimes lead to unpredictable behavior
- As systems get larger with many interdependent parts, it is increasingly difficult to do so, leading to the increasingly prevalent use of dynamic scheduling with techniques based on preemption