# RTOS Tutorial

Contents

[RTOS Tutorial 1](#_Toc23517069)

[Introduction 2](#_Toc23517070)

[Scheduler criteria: 2](#_Toc23517071)

[CPU utilization: (U) 2](#_Toc23517072)

[Scheduler algorithms: 2](#_Toc23517073)

[First come first serve 2](#_Toc23517074)

[Round robin scheduler 3](#_Toc23517075)

[Features of RTOS-ARM CMSIS 3](#_Toc23517076)

[Thread management 3](#_Toc23517077)

### Introduction

An operating system is a set of instructions (program) that supports the functionality of the hardware and provides meaningful ways to interact and use the hardware. Real time operating system is an operating system that produces results that is time bound. Generally, an OS allows the capability to execute multiple programs, however it must be noted that, the execution is only 1 thread (task) at a time, but the scheduling is done in such a way, that it appears as though multiple tasks are running simultaneously.

### Scheduler criteria:

* Throughput: number of tasks the scheduler can complete per unit time
* Turnaround time: time for completion of each task
* Response time: time between request to first response
* CPU utilization: % of CPU cycle being used (affects turnaround time and response time)
* Wait time: time elapsed since queued in “Ready” state

### CPU utilization: (U)

CPU speed=80Million c/s

CPU U= 45 Million c/s

U= 45/80 \*100=56.25

The goal is to design efficient algorithms which makes max utilization of CPU

U=ƩC/T

### Scheduler algorithms:

* First come first serve
* Round robin
* Weighted round robin
* Rate monotonic
* Shortest job first

### First come first serve

This is a non-preemptive scheduler. As the name suggests the threads are executed based on its time of arrival. The implementation is using FIFO Queue. The average wait time is very high leading to poor performance and efficiency.

### Round robin scheduler

It is a preemptive scheduler. The scheduler assigns time slice (quanta) for each thread, giving a fixed amount of time for its execution, after which, the next priority thread gets executed. The OS takes note of the number of ms each thread has executed and places the preempted thread in the “Ready” queue which will be executed in the next time quanta

#### Factors to consider when choosing quanta:

* For very large quanta size, round robin behaves like FCFS
* For very small-time quanta, round robin displays, true processor sharing
* However, if time quanta is very small, more investment in context switching which may not be very efficient
* Time quanta must be greater than context switch time(depends on the platform)

### Features of RTOS-ARM CMSIS

* Flexible scheduler -both preemptive and non-preemptive
* Unlimited tasks – max of 250 active tasks
* Fast context switching time with low interrupt latency
* Inter thread flags- semaphores, mailboxes, mutex
* API specification

### Thread management

A thread is a task that the OS schedules and runs. There are 3 main parts of thread management:

* Define a thread function

Ex: void thread\_blue(void const\* args)

{

While(1)

{

Blue\_toggle();

}

}

* Define thread parameters
  + Ex: osThreadDef(thread\_blue,osPriorityNormal,1,0);
* Create Thread
  + Ex: id\_blue=osThreadCreate(osThread(thread\_blue),NULL);

In addition to the above, there are various other functions we can use for thread management:

* OS\_TASKCNT: task count is number of thread running at any given moment
* OS\_STKSIZE: default thread stack size
* OS\_PRIVCNT: number of threads with user defined stack
* OS\_TICK: OS time interval
* OS\_SYSTICK: to use systick
* OS\_ROBIN: to enable round robin scheduler
* OS\_ROBINTOUT: to set the Quanta size

Ex:

osThreadDef(name,priority,instances,stacksz);

osThreadId osThreadCreate (const osThreadDef\_t\* thread\_def, void\* arg);

osThreadGetId();

osStatus osThreadTerminate(isThreadId thread\_id)

When a thread is created, a priority level is associated with it, depending on which, the scheduler decides the order of execution. osPriorityIdle, osPriorityLow, BelowNormal, Normal, AboveNormal, High, RealTime, Error

### Periodic threads

* Define a timer: osTimerDef(name,function) and store it in the timerId
  + name: the name of the timer function: the name of the thread to execute
* Timer create: osTimerCreate()
* Start the timer: osTimerStart