# Real Time Operating Systems

LY ETRX
Sem VIII

Arati Phadke
AY 2021-22

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

# Computer Organization and Architecture

**Course Outcomes**

At the end of successful completion of the course the student will be able to

**CO1:** Understand real time operating system concepts.

**CO2:** Create various instances of a task and send and receive data to and from a queue

**CO3:** Implement the APIs within an Interrupt Service Routine

**CO4:** Develop algorithms for real time processes using FreeRTOS

**CO5:** Understand memory allocation schemes

# Module 2

**Task Management**

**2.1** Multi Tasking, Top level task states, Creating tasks, Task Priorities, Time measurement and Tick interrupt, Deleting Tasks, Scheduling algorithms: Preemptive and Non Preemptive

**2.2** Queue Management, Characteristics of a queue, Using a queue using APIs to create Mailbox

# Multitasking

- Classification of tasks : hard , soft , firm real time etc
- Generation of task /arrival of task
  - Internal / external


- Appropriate scheduling of tasks is the basic mechanism adopted by a real-time operating system to meet the time constraints of a task.
- Basic Terminology : a recap
- Types of scheduling algorithms

# Multitasking Basic Terminology

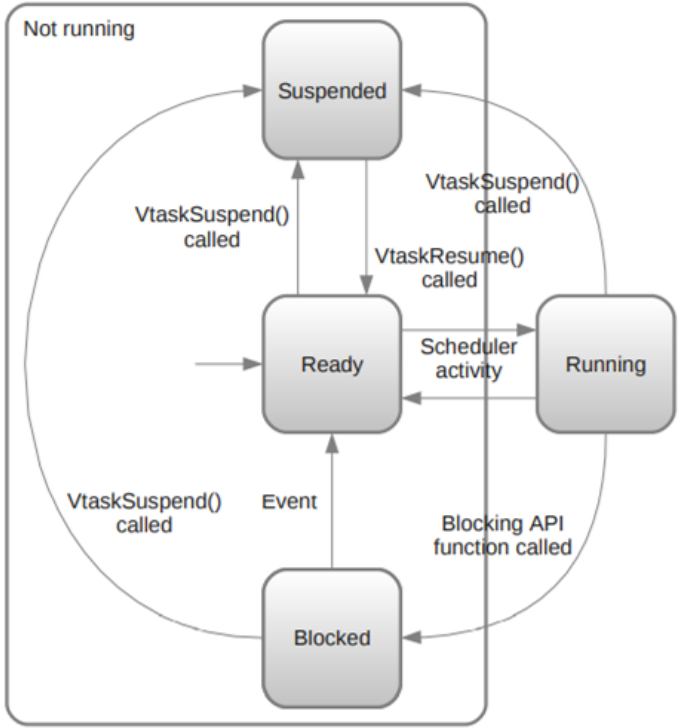



*Figure 2: Life cycle of a task*

# Multitasking Basic Terminology

NPTEL Notes Chapter 29 and 30

- Task instance
- Arrival of task
- Deadline : absolute and relative
- Response Time : from Arrival till it produces desired results
- Task precedence
- Data sharing
  - leading to task precedence
  - In concurrent / overlapping tasks
- Types of tasks : Periodic , sporadic, aperiodic

# Scheduling Terminology

NPTEL Notes Chapter 29 and 30

- **Valid Schedule** For a set of tasks
  - at most one task is assigned to a processor at a time,
  - no task is scheduled before its arrival time,
  - and the precedence and resource constraints of all tasks are satisfied.
- **Feasible Schedule:**
  - When a Valid Schedule meets time constraints of all tasks
- Proficient schedule / optimal schedule
- **Scheduling Points:**
  - Instance at which scheduler makes decisions regarding which task is to be run next.
  - Task scheduler is activated by OS at scheduling points.
- Preemptive / Non Preemptive
- **Utilization**

# Scheduling Algorihms

NPTEL Notes Chapter 29 and 30

- **Clock Driven**
  - Table-driven
  - Cyclic
- **Event Driven :** can handle sporadic , aperiodic tasks
  - Simple priority-based
  - Rate Monotonic Analysis (RMA)
  - Earliest Deadline First (EDF)
- **Hybrid**
  - Round-robin
- **Acceptance test**
- **Planning Based Scheduler**
- **Best effort Scheduler**

# Types of Scheduling Algorithms

- **Clock driven Scheduling**
  - The scheduling points are determined by timer interrupts.
  - They fix the schedule before the system starts to run or pre-determines which task will run when.
  - Incur very little run time overhead.
  - They can not satisfactorily handle aperiodic and sporadic tasks since the exact time of occurrence of these tasks cannot be predicted.
  - Also called static scheduler.

# Types of Scheduling Algorithms

- **Table driven Scheduling**
  - Precomputed table of scheduling points of tasks
  - How long the table should be ?
  - Concept of Major Cycle
  - A major cycle of a set of tasks is an interval of time on the timeline such that in each major cycle, the different tasks recur identically.
  - M = LCM ( P1,P2,P3,...Pi)

# Types of Scheduling Algorithms

- **Cyclic Scheduling**
  - Cyclic schedulers are simple, efficient, and are easy to program, hence widely used
  - Major Cycle is divided in Minor cycles : frame
  - Scheduling points at frame boundaries
  - Frame boundaries are defined through the interrupts generated by a periodic timer.
  - Task is assigned to run in one or more frames.
  - Choice of size of the frame
    - Minimum context switching :max({ei}) < F
    - Minimum size of table: M mod F = 0.
    - Meeting task deadlines :for every Ti, 2F – gcd(F, pi) ≤ di

# Cyclic Scheduling Algorithms

- **Example**
- T1 = (e1=1, p1=4), T2 = (e2=1.5, p2=5), T3 = (e3=1, p3=20), T4 = (e4=2, p4=20).
- Select an appropriate frame size.
- **Constraint 1:** max({ei}) < F so F ≥ 2.
- **Constraint 2:** The major cycle M for the given task set is given by M = LCM(4,5,20) = 20. And M mod F = 0. So F can take on the values 2, 4, 5, 10, 20.
- **Constraint 3:** check whether a selected frame size F satisfies the inequality: 2F – gcd(F, pi) < di for each pi.
  - F = 2, F = 4, F = 5, F = 10

# Cyclic Scheduling Algorithms

- **Inclusion of aperiodic and sporadic tasks**
- Arrival of aperiodic and sporadic tasks
- Lowered achievable utilization of the system.
- A schedule (assignment of tasks to frames) for only periodic tasks is prepared.
- The sporadic and aperiodic tasks are scheduled in the slack times that may be available in the frames.
- Slack time in a frame is the time left in the frame after a periodic task allocated to the frame completes its execution.
- Non-zero slack time in a frame can exist only when the execution time of the task allocated to it is smaller than the frame size.

# Cyclic Scheduling Algorithms

- A sporadic task is taken up for scheduling only if enough slack time is available for the arriving sporadic task to complete before its deadline.
- Acceptance test.
- Aperiodic tasks do not have strict deadlines, they can be taken up for scheduling without any acceptance test and best effort can be made to schedule them in the slack times available.
- Slack times are stored in a table and during acceptance test this table is used to check the schedulability of the arriving tasks.
- Best effort is made to meet respective deadlines.

# Event Driven Scheduling

- **Scheduling points are defined by event occurrences**
- This class of schedulers is normally preemptive, i.e., when a higher priority task becomes ready, it preempts any lower priority task that may be running.
- **Simple priority-based** : foreground-background scheduler is possibly the simplest priority-driven preemptive scheduler.
- Foreground tasks: higher priority, periodic
- Background tasks: lower priority, non real time , aperiodic
- Completion time for background task
- $ctB = eB / (1 - \Sigma\ ei / pi)$

# Event Driven Scheduling

- Example
- In a simple priority-driven preemptive scheduler, two periodic tasks T1 and T2 and a background task are scheduled. The periodic task T1 has the highest priority and executes once every 20 milliseconds and requires 10 milliseconds of execution time each time. T2 requires 20 milliseconds of processing every 50 milliseconds. T3 is a background task and requires 100 milliseconds to complete. Assuming that all the tasks start at time 0, determine the time at which T3 will complete.

# Event Driven Scheduling

- **Earliest Deadline First (EDF)**
- Schedulability: sum of utilization <1
- Deadline may be smaller than period
- Priority : static or dynamic?
- Implementation of EDF
  - Queue of ready tasks with its absolute deadlines
  - Sorted queue of ready tasks : tasks sorted according to the proximity of their deadline.
- Time complexity for inserting a task , searching a task and deleting a task is concern in different implementations
- Efficient resource sharing and implementation are concerns in EDF

# Event Driven Scheduling

- **Rate Monotonic Algorithm(RMA)**
- The priority of a task is directly proportional to its rate (or, inversely proportional to its period).
- Schedulability
- Necessary: $\Sigma$ ui <1
- Sufficient: $\Sigma ui \leq n(2^{1/n} - 1)$
- A set of periodic real-time tasks is RMA schedulable under any task phasing, iff all the tasks meet their respective first deadlines under zero phasing
- The worst case response time for a task occurs when it is in phase with its higher
- Context Switching Overhead

# Event Driven Scheduling

- **A preemptive static priority real-time task scheduler is used to schedule two periodic tasks T1 and T2 with the following characteristics:**

| Task | Phase mSec | Execution Time mSec | Relative Deadline mSec | Period mSec |
|---|---|---|---|---|
| **T1** | **0** | **10** | **20** | **20** |
| **T2** | **0** | **20** | **50** | **50** |

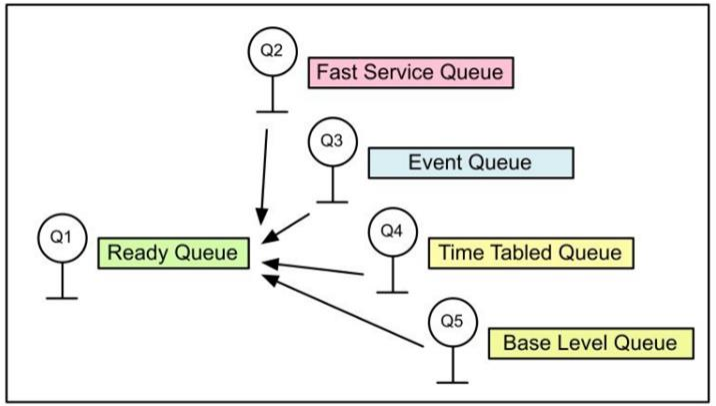
Assume that T1 has higher priority than T2. A background task arrives at time 0 and would require 1000 mSec to complete. Compute the completion time of the background task assuming that context switching takes no more than 0.5 mSec.

# Types of Scheduling Algorithms

Chapter 2 Scheduling - Concepts and implementation. : Jim Cooling

## Implementation of Queues

- **Queue of ready tasks**
  - With priority
  - With deadline ( absolute)
  - Sorted
- **Multiple queues**
  - Ready tasks
  - Suspended / blocked tasks
    - Fast service
    - Event queue
    - Time tabled queue
    - Base level queue



# Creating and Deleting a Task in FreeRTOS

- **A typical Task signature :** void ATaskFunction( void *pvParameters );

```
void ATaskFunction( void *pvParameters )
{ /* Variables : each invocation of task creates a copy
 int32_t lVariableExample = 0;
 for( ;; )
 { /* infinite loop The code to implement the task functionality will go here. */
 }
 vTaskDelete( NULL ); /* delete calling task , useful in dynamic memory management*/
}
```

# Creating and Deleting a Task in FreeRTOS

- **A typical Task signature :** void ATaskFunction( void *pvParameters );
- **Creating a task: vTaskCreate()**
  - **pvTaskCode:** a pointer to the function where the task is implemented.
  - **pcName:** given name to the task. This is useless to FreeRTOS but is intented to debugging purpose only.
  - **usStackDepth:** length of the stack for this task in words. The actual size of the stack depends on the micro controller. If stack with is 32 bits (4 bytes) and usStackDepth is 100, then 400 bytes (4 times 100) will be allocated for the task.
  - **pvParameters:** a pointer to arguments given to the task. A good practice consists in creating a dedicated
  - **uxPriority:** priority given to the task, a number between 0 and MAX_PRIORITIES – 1.
  - **pxCreatedTask:** a pointer to an identifier that allows to handle the task. If the task does not have to be handled in the future, this can be leaved NULL.

# Creating and Deleting a Task in FreeRTOS

portBASE_TYPE xTaskCreate ( pdTASK_CODE pvTaskCode, const signed portCHAR * const pcName, unsigned portSHORT usStackDepth, void *pvParameters,unsigned portBASE_TYPE uxPriority, xTaskHandle *pxCreatedTask );

xTaskCreate(mytask2, "task2", 128, (void*)count, 2, &mytaskHandle2);

**Returned value**
There are two possible return values:
1. **pdPASS** This indicates that the task has been created successfully.
2. **pdFAIL** This indicates that the task has not been created because there is insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack.

- **Deleting a task: vTaskDelete( xTaskHandle pxTask)**

vTaskDelete(mytaskHandle2);

# Task functions in FreeRTOS

- vTaskStartScheduler();
- for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ ) { /* crude delay implementation. There is nothing to do in here. */ }
- **vTaskDelay( TickType_t xTicksToDelay );**The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state
- vTaskSuspend()
- vTaskResume() or xTaskResumeFromISR()
- vTaskPriorityGet()
- vTaskPrioritySet()
- The FreeRTOS scheduler will always ensure that the highest priority task that is able to run is the task selected to enter the Running state. Where more than one task of the same priority is able to run, the scheduler will transition each task into and out of the Running state, in turn.

# Task functions in FreeRTOS

- xTaskGetCurentTaskHandle();
- xTaskGetIdleTaskhandle()
- pcTaskGetName(TaskHandle)
- xTaskGetHandle(“task2”)

# Tasks in Blocked State

- **The Blocked State** A task that is waiting for an event is said to be in the 'Blocked' state, which is a sub-state of the Not Running state.
- Tasks can enter the Blocked state to wait for two different types of event:
  1. **Temporal (time-related) events**—the event being either a delay period expiring, or an absolute time being reached. For example, a task may enter the Blocked state to wait for 10 milliseconds to pass.
  2. **Synchronization events**—where the events originate from another task or interrupt. For example, a task may enter the Blocked state to wait for data to arrive on a queue. Synchronization events cover a broad range of event types.

  FreeRTOS queues, binary semaphores, counting semaphores, mutexes, recursive mutexes, event groups and direct to task notifications can all be used to create synchronization events.