EE458 - Embedded Systems
Lecture 5 – Intro to RTOSes

• Outline
  – RTOS Components
  – Multitasking
  – Objects
  – Services
  – Key Characteristics of an RTOS
  – RTEMS Key Concepts

• References
  – RTC: Chapter 4
  – CUG: Chapter 3
Introduction to RTOSes

- There are many similarities between modern RTOSes and GPOSes:
  - support for *multitasking*
  - software and hardware resource management
  - providing OS services to applications
  - abstract the hardware from the software
Introduction to RTOSes

• But there are differences too. RTOSes have:
  – better reliability in embedded applications
  – the ability to scale up or down
  – better performance
  – reduced memory requirements
  – scheduling policies tailored for embedded apps
  – support for diskless systems
  – better portability to different hardware platforms
Introduction to RTOSes

- An RTOS consists of a kernel that provides core services plus modules for networking, device I/O, filesystem access, etc.

- The kernel consists of the following:
  - **scheduler**: determines which tasks execute when and for how long.
  - **objects**: tasks, semaphores, queues, timers, etc.
  - **services**: interrupt, device, time, and memory management
The Scheduler

- The scheduler allocates time to schedulable entities which are also known as independent threads of executions (tasks and processes).

- Tasks are independent threads of execution that share a common memory space. Tasks are supported in most RTOSes.

- Processes run in separate memory spaces. This memory protection comes at the expense of performance. Processes are supported by only a few RTOSes.
Multitasking

- Note: Different terminology is used with GPOSes. A RTOS task is more commonly called a thread in a GPOS.
- The kernel handles multiple tasks by multitasking. Different tasks appear to be running concurrently. The kernel is actually interleaving task segments sequentially.
Context Switches

- Tasks are interleaved according to a preset scheduling algorithm.

- A *context switch* occurs when the scheduler switches from one task to another.

- A *task control block* (TCB) is created every time a new task is created. When a task is stopped all context information (CPU registers, IP, SP) is saved in the TCB.
Context Switches

- When a new task is started the context is restored from the task's TCB.
- A task switch may occur any time a task makes a system call, when an interrupt service routing (ISR) completes or under other conditions as determined by the scheduling algorithm.
- The *dispatcher* is the specific part of the scheduler that performs context switching.
Scheduling Algorithms

• Most kernels support two common scheduling algorithms:
  – preemptive priority-based scheduling
  – round-robin scheduling

• Depending on the RTOS these may be known by different names.

• Many RTOSes allow the scheduling algorithms to be configured to fit the needs of the application.
Priority-Based Scheduling

• In preemptive priority-based scheduling each task is assigned a priority. The task with the highest priority that is ready to run is executed.

• Most RTOSes support 256 levels. Usually priority 0 is the highest and 255 is the lowest.

• Tasks are assigned priorities when they are created. The OS provides system calls which can be used to change the priority.
Priority-Based Scheduling

Task 1  Task 2  Task 3

Premption

Task Completion

Task 2

Task 1

Time

Task Priority

HIGH

LOW
Round Robin Scheduling

- With true round-robin scheduling each task would receive an equal share (a time slice) of CPU time. This cannot satisfy real-time system requirements.
- RTOSes augment round-robin scheduling with preemptive priority-based scheduling. Tasks of the same priority execute for a time slice in an ongoing cycle.
- A higher priority task may preempt a lower priority task at any time.
Round Robin Scheduling

- Task 1
- Task 2
- T3
- Task 4

<table>
<thead>
<tr>
<th>Task</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>HIGH</td>
</tr>
<tr>
<td>Task 2</td>
<td>LOW</td>
</tr>
<tr>
<td>T3</td>
<td>LOW</td>
</tr>
<tr>
<td>Task 4</td>
<td>HIGH</td>
</tr>
</tbody>
</table>

Premption: Task 1
Task Completion: Task 4

Time:
- Task 1
- Task 2
- T3
- Task 1
Objects

- Kernel objects are the building blocks for application development. The most common RTOS objects are:
  - tasks: threads of execution that compete for CPU time.
  - semaphores: tokens that can be used by tasks for synchronization and mutual exclusion.
  - message queues: buffers that can be used for synchronization, mutual exclusion and data exchange between tasks.
• Most kernels provide services that help developers create applications for real-time embedded systems.

• These services are usually provided through sets of API calls.

• Typical services perform operations on kernel objects or facilitate timer management, interrupt handling, device I/O, and memory management.
Key Characteristics of an RTOS

• Some of the more common attributes of an RTOS are:
  - reliability
  - predictability
  - performance
  - compactness
  - scalability
Key Characteristics of an RTOS

- **Reliability**
  - Different applications require different levels of reliability. Reliability depends on the hardware, the BSP, the RTOS, and the application.

- **Predictability**
  - OS calls are always completed within a given timeframe. *(Deterministic)*

- **Performance**
  - A function of both the processor and the RTOS. CPU performance is measured in MIPS. Data transfer rates in bps (bits per second).
Key Characteristics of an RTOS

• Compactness
  – In most embedded systems the amount of system memory is limited by size and/or cost. The RTOS must be small and efficient.

• Scalability
  – Most RTOSes can scale up or down to meet application requirements. This allows the same RTOS to be used in very different applications. Modular components can be added or deleted in the RTOS.
• All RTEMS objects have an RTEMS assigned ID and a user assigned name.

• An object name is a 32 bit entity. The data type `rtems_name` is used to store the name. You could use a 32 bit integer as a name, but the `rtems_build_name()` routine allows you to assign a meaningful name to the object:

```c
rtems_object_name myname;
myname = rtems_build_name('T','S','K','1');
```
An object ID is a unique 32 bit entity composed of four parts: class, api, node, and index. The data type `rtems_id` is used to store object IDs.

- class indicates the object type (task, queue, etc). The api indicates the API to which the object belongs (classic, POSIX, ITRON). The node is the processor on which the object is defined. The index identifies a particular object of a given class on a given node.
All RTEMS directives (system calls) involving objects require the ID as an argument. The ID is returned when an object is created so it is not usually necessary to refer to an object by name in a single processor application.

Identification directives (\texttt{task\_ident()}, \texttt{semaphore\_ident()}) are provided to find an object's ID based on its name and node.
Intro to RTEMS: Key Concepts

• RTEMS provides the following managers for task synchronization and communication:
  – Semaphore: mutual exclusion, priority inheritance
  – Message Queue: synchronization and communications
  – Event: high performance synchronization
  – Signal: asynchronous communication primarily used for exception handling
Intro to RTEMS: Key Concepts

- RTEMS provides support for several time related operations. The basic unit of time is known as a **tick**. The tick default time is BSP dependent. All time intervals are specified as multiples of the tick time.

- Each task automatically has one sleep timer at its disposal. The timer manager can be used to create other timers as required.
RTEMS provides three memory managers:

- **partition**: manages fixed size memory blocks
- **Region**: manages dynamically allocated, variable-size memory blocks
- **Dual Ported**: manages shared-memory for multiprocessor applications