EE443 - Embedded Systems

Lecture 9
Real-Time Operating Systems

Contents:

The common characteristic feature of a real-time operating system (RTOS) is its ability to serve real-time processing demands. An RTOS working in an embedded system must be consistent in terms of the amount of time it takes to respond to processing demands and to complete the related task. An embedded system must behave deterministically in meeting deadlines set by the external physical environment.

9.1 When Do We Need an RTOS?

Previously we stated that balancing the processor load between the main operation loop and the ISRs can be the best solution in most embedded systems applications. The data transfers between the main loop and the ISRs can be synchronized using queue buffers or semaphores. The following example involves two ISRs acquiring data and the two related processes running in the main operation loop.

```
// Main program:
void Main(void)
// Processor load is shared between main and ISRs
{
  <Initialize peripherals>
  <Initialize tasks>
  while (TRUE)  // Main Loop
  { Check queue buffer-1:
      <Process event-1 data, if available>
    Check status of semaphore-2:
      <Process event-2 data, if available>
    • • •
  }  // end of Main Loop
  • • •
}
```

```
void ISR1(void)
{
  Give immediate response to event-1
  Store data in queue buffer-1
  • • •
}
void ISR2(void)
{
  Give immediate response to event-2
  Store data in memory and set semaphore-2
  • • •
}
```

We can calculate the total process time required in unit time (i.e. in one second) using the time spend for each task and the maximum frequency of the task activity. Consider the following timing specifications for the tasks mentioned in the example given above:

Task Description	**Time per single run**	**Maximum frequency**	**Process time for the task in 1s**
ISR1	1 ms	10 /s	10 ms
ISR2	1 ms	1 /s	1 ms
Process event-1 data	10 ms	10 /s	100 ms
Process event-2 data	300 ms	1 /s	300 ms
Total process time required for all tasks in 1 s:			**411 ms**

It seems like the processor chosen for this application can easily handle the required processing load. But we may have a timing problem even if the processor is capable of handling twice as much the required load. Once the processing of event-2 data starts in the main loop it will take 300 ms to complete the required task. The processor will still be able to respond to the interrupt requests during that time, but it will not be possible to process the event-1 data at regular time intervals.

The processor cannot dedicate 300 ms of its time to the handling of event-2 data continuously. The solution is to interrupt the processing of event-2 data at least once every 100 ms and check the status of queue buffer-1 to see if there is any data waiting to be processed. In fact, that is the basic multitasking functionality that we expect from a real-time operating system.

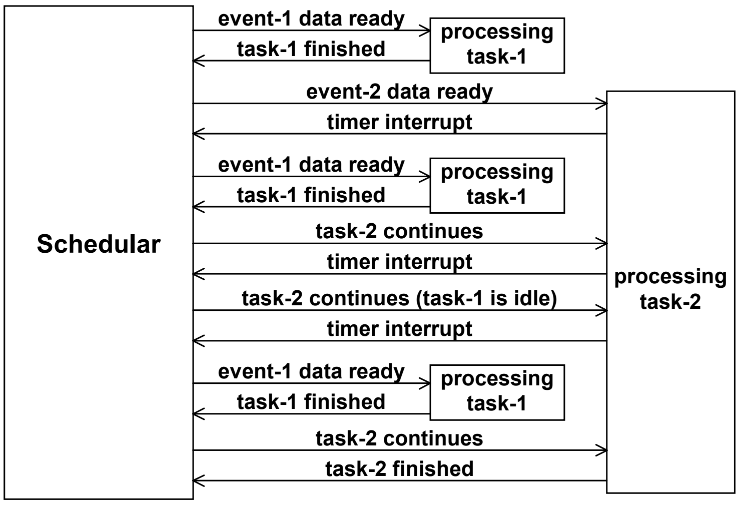
9.2 Scheduling

Consider the following ISR that is invoked by a dedicated timer. The timer is programmed during system initialization to generate interrupts at regular time intervals. The status information for the two processing tasks are stored in the variables, `T1status` and `T2status`. These variables can take four possible values:

`IDLE:`	There is nothing to do.
`DATAREADY:`	There are data waiting to be processed as indicated by the status of queue buffer or semaphores.
`RUNNING:`	A process has been started already, and it was running or it was waiting to be reactivated when the timer interrupt is received.
`FINISHED:`	Processing task is completed.

```
void ISR0(void)
// Activated by a timer interrupt to schedule execution of two tasks.
  if (T1status != IDLE)  // Task-1 has higher priority
  { if (T1status == DATAREADY)
    { T1status = RUNNING;
      <start processing task-1>
    }
    else if (T1status == RUNNING)
      <continue processing task-1>
    else if (T1status == FINISHED)
    { <terminate processing task-1>
      T1status = IDLE;
    }
  }
  else if (T2status != IDLE)
  { if (T2status == DATAREADY)
    { T2status = RUNNING;
      <start processing task-2>
    }
    else if (T2status == RUNNING)
      <continue processing task-2>
    else if (T2status == FINISHED)
    { <terminate processing task-2>
      T2status = IDLE;
    }
  }
  • • •
```

ISR0 checks the status of all processes registered in the system every time it is activated by the timer interrupt. The first process to be checked is the one that has the highest priority. The following figure shows the sequence of events that are likely to occur according to the timing requirements specified in the previous section, assuming that the timer interrupt is received at every ~100ms.



The simple algorithm given in ISR0 is the very basic form of a *scheduler* that constitutes the backbone of an RTOS. The **scheduler** is a program that decides on how the processor time is shared between the tasks according to their priorities and other timing constraints. Once we become familiar with the scheduler functionality in an RTOS we start thinking about several questions:

- What should be the **period** of the timer interrupt that invokes the scheduler?
- What should be the **priority** of the timer interrupt that invokes the scheduler?
- How many processes can a scheduler handle?

The period of the timer interrupt that invokes the scheduler must be short enough to match the timing requirements. In this example, processing of event-1 data is expected to be done at every **100 ms**. If we can afford a timing variation of **10 ms**, then activating the scheduler with **10 ms** intervals is sufficient. The maximum possible activation rate depends on the additional processing load required for execution of the scheduler. The available processor time is **1000-411=589 ms** in the example. If it takes **1 ms** to run the scheduler, then the maximum possible activation rate is **589** per second. Switching between the tasks does not take a significant time on a basic microcontroller where all program and data memory of all tasks are always resident in a directly accessible storage. More complex systems with cache memory and virtual memory arrangements require more time to manage memory according to the task requirements.

The priority of the timer interrupt that invokes the scheduler can be lower than the other interrupts provided that the execution time of the ISRs are shorter than the timer period. Otherwise, other interrupts can cause unwanted delays in activation of processing tasks. As an alternative, all interrupt calls can be routed through the scheduler. In that case the ISRs become ordinary procedures that are activated by the scheduler just like the other processing tasks.

The scheduling algorithm given in ISR0 can handle two concurrent tasks only as required by the specific example given above. An RTOS keeps a dynamically changing list of tasks. Real-time processes register their tasks with the scheduler.

9.2.1 Scheduling Strategies

The following are some of the common scheduling strategies that are used for distributing processor time among concurrent tasks running on a computer.

First in first out (FIFO)
FIFO, also known as First Come, First Served (FCFS), queues processes in the order that they arrive. FIFO strategy is not any different than the simple main loop operation that ignores all other tasks until the current task is finished.

Shortest remaining time
The scheduler gives highest priority to the task that requires the least estimated processing time to be completed. This requires beforehand knowledge of the time required for a process to be finished. The programmer should write faster procedures for the tasks that have shorter deadlines.

Fixed priority scheduling
A fixed priority is assigned to every process. Scheduler executes the tasks according to their priority, so that higher priority tasks have shorter waiting and response times. If the tasks that have shorter deadlines are given higher priorities, then they can be completed on time while other processes get interrupted.

Round-robin scheduling
In round-robin scheduling a fixed processing time is assigned to every process. The scheduler cycles through the active processes allowing each task to execute for this fixed time period. Selection of the cycling time unit is critical in round-robin scheduling since switching between the tasks may require extensive overhead.

Multilevel queue scheduling
This scheduling strategy is applicable where the system tasks can be divided into groups that have different response-time requirements and scheduling needs. For example, all urgent tasks can be placed in a high-priority queue. The schedular treats these urgent tasks equally while other low-priority tasks are not served unless the high-priority queue is empty.

Note that these scheduling strategies alone may not be able to answer response time requirements in real-time applications. An RTOS should have additional decision mechanisms to meet the real-time constraints in embedded systems applications.

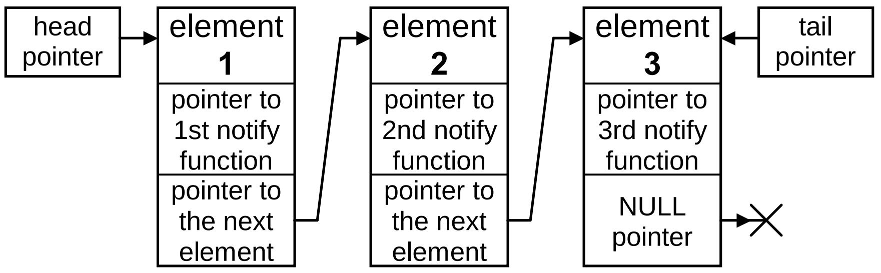
9.3 Threads

Concurrency is an essential feature of embedded systems. A computer program can be considered as **concurrent** if different parts of the program conceptually execute simultaneously. **Concurrency** should not be confused with the concept of **parallelism** where parts of a program execute simultaneously on **physically distinct** hardware (such as multicore processors). On a single processor, multithreading is realized having a processor switch between different threads according to the scheduling strategies mentioned above. This sharing of processor time between multiple tasks is also known as time-division multiplexing.

Threads are the branches of the programs that execute simultaneously on a processor. The following set of functions provide a basic mechanism for building threads starting from scratch without using any specialized thread library.

```
#include <stdlib.h>
#include <stdio.h>
int  x; // Value that gets updated.
typedef void  notifyProcedure(int); // Type of notify procedure
struct element
{ notifyProcedure* listener; // Pointer to notify procedure
  struct element*  next;     // Pointer to the next item
};
typedef struct element   element_t; // Type of list elements
element_t*   head = NULL; // Pointer to start of list
element_t*   tail = NULL; // Pointer to end of list

// Procedure to add a listener.
void addListener(notifyProcedure* listener)
{
  if (head == NULL)
  { head = malloc(sizeof(element_t));
    head->listener = listener;
    head->next = NULL;
    tail = head;
  }
  else
  { tail->next = malloc(sizeof(element_t));
    tail = tail->next;
    tail->listener = listener;
    tail->next = NULL;
  }
}
```


head pointer
element 1
pointer to 1st notify function
pointer to the next element
element 2
pointer to 2nd notify function
pointer to the next element
element 3
pointer to 3rd notify function
NULL pointer
tail pointer

```
// Procedure to update x.
void update(int newx)
element_t*   element;
{
  x = newx;
// Notify listeners.
  element = head;
  while (element != NULL)
  { (*(element->listener))(newx);  // call the listener function
    element = element->next;
  }
}

// Example of notify procedure.
void PrintNote(int arg)
{
  printf("%d ", arg);
}
```

The following sample program will produce the result "**1 1 2 2 2**". First the **`PrintNote`** function is added to the link list twice and the **`update`** procedure is called producing the output "**1 1**". Then the **`PrintNote`** function is added to the link list a third time. Next time the **`update`** procedure is called it produces the result "**2 2 2**".

```
int main(void)
{
  addListener(&PrintNote);
  addListener(&PrintNote);
  update(1);
  addListener(&PrintNote);
  update(2);
}
```

In a typical RTOS, user-defined procedures are registered with the scheduler as call-back functions. A separate thread is created for every process. Whenever an event is triggered, the related process thread is invoked with the available event data. As an example, an ISR gets the event data from the interrupt source and calls the **`update`** procedure corresponding to the related process. The rest is left to the scheduler. Scheduler calls the registered call-back functions according to the priority settings of the processes for each thread.

9.4 Common RTOS

Considering the real-time constraints of an embedded system, the main features of in a real-time operating system are minimal latency and minimal processing overhead in switching between tasks. More important than these efficiency factors is the predictable response of an RTOS that provides bounded latency for completion of system tasks and interrupt services.

The following is a list of common real-time operating systems specifically developed for embedded applications.

- Embedded Linux (an open source community effort)
- FreeRTOS (another open source community effort)
- Windows CE (WinCE) (from Microsoft)
- VxWorks (from Wind River Systems, acquired by Intel in 2009)

Another type of RTOS is the mobile operating systems that are developed specifically for handheld devices such as cellular phones and PDAs. These operating systems provide support for wireless communications and media formats. Examples are:

- Symbian OS (an opensource effort maintained by the Symbian Foundation)
- Android (from Google),
- BlackBerry OS (from RIM)
- iPhone OS (from Apple)
- Palm OS (from Palm, Inc., acquired by Hewlett-Packard in 2010)
- Windows Mobile (from Microsoft)

The RTOSs listed so far are developed for relatively complex applications mostly involving multimedia formats and they require several megabytes of storage for the system code. On the other hand, RTOSs for simple MCUs are also available in the form of microkernels that occupy only a few kilobytes of memory. These RTOSs can handle the basic scheduling, memory and resource management functions that are sufficient for most embedded systems for industrial applications.