Embedded Systems

Real Time Systems (Part II)

Round-Robin Scheduling

• When two or more tasks have the same priority, the kernel allows one task to run for a predetermined amount of time, called a quantum, and then selects another task
• This process is called round-robin scheduling or time slicing
• The kernel gives control to the next task in line if:
  – the current task has no work to do during its time slice or
  – the current task completes before the end of its time slice or
  – the time slice ends
Task Priorities

- In many real-time systems, a priority is assigned to each task.
- The more important the task, the higher the priority given to it.
  - Essentially translates into how much CPU time a given task gets.
- With most kernels, you are generally responsible for deciding what priority each task gets.
- Task priorities may be static or dynamic.

Static Task Priorities

- Task priorities are static when the priority of each task does not change during the application's execution.
- Each task is thus given a fixed priority at compile time.
- All the tasks and their timing constraints are known at compile time in a system where priorities are static.
- Requires essentially complete a priori information about the system and all tasks to run on the system.
Dynamic Task Priorities

- Task priorities are dynamic if the priority of tasks can be changed during the application's execution
  - Each task can change its priority at run time
- This is a highly desirable feature to have in a real-time kernel to avoid priority inversions
  - Allows the system to adapt to external factors that should affect the execution behavior of the system
- uC/OS-II supports dynamic task priorities

Priority Inversion

- Priority inversion is a problem in real-time systems and can occur in systems using a real-time kernel
- Terms to understand:
  - Semaphore: A protected variable or abstract data type that provides for controlling access by multiple processes to a common resource
    - Semaphore variants:
      - Counting semaphore: allow an arbitrary resource count
        - More than one copy of a resource
        - More than one instance of a resource may be available at a given time
      - Binary semaphore: semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available)
Priority Inversion (continued)

- Mutex: essentially the same thing as a binary semaphore, and sometimes uses the same basic implementation
- The term "mutex" is used to describe a construct which prevents two processes from executing the same piece of code, or accessing the same data, at the same time

Priority Inversion Example

- Assume three tasks in a system
  - Task 1 has the highest priority
  - Task 2 has medium priority
  - Task 3 has the lowest priority

- Assume Task 3 is executing and has been granted access to a resource and has been granted the semaphore associated with the resource
Priority Inversion Example

• The priority of Task 1 has been virtually reduced to that of Task 3 because Task 1 was waiting for the resource that Task 3 owned
• The situation was aggravated when Task 2 preempted Task 3, which further delayed the execution of Task 1
• Remedies:
  – Raise the priority of Task 3 just for the time to access the resource
    • A dynamic priority multitasking kernel would support this
    • Does require CPU time that might be wasted
  – Need a kernel that changes the priority of a task automatically (*priority inheritance*)
    • uC/OS-II supports this feature
Assigning Task Priorities

- Assigning task priorities is not a trivial undertaking because of the complex nature of real-time systems
  - In most systems, not all tasks are considered critical
  - Noncritical tasks should obviously be given low priorities
- Most real-time systems have a combination of soft and hard requirements
  - In a soft real-time system, tasks are performed as quickly as possible, but they don't have to finish by specific times
  - In hard real-time systems, tasks have to be performed not only correctly but on time
Rate Monotonic Scheduling

- *Rate monotonic scheduling* (RMS) assigns task priorities based on how often tasks execute.
- Simply put, tasks with the highest rate of execution are given the highest priority.

![Diagram showing task priorities on a grid with execution rates on the x-axis and priorities on the y-axis.

Rate Monotonic Scheduling

- RMS makes a number of assumptions:
  - All tasks are periodic (they occur at regular intervals)
  - Tasks do not synchronize with one another, share resources, or exchange data
  - The CPU must always execute the highest priority task that is ready to run
    - Preemptive scheduling must be used
- Given a set of $n$ tasks that are assigned RMS priorities, the basic RMS theorem states that all task hard real-time deadlines are always met if the following inequality is verified:

$$\sum_i \frac{E_i}{T_i} \leq n \left(2^{1/n} - 1\right)$$

- $E_i$ corresponds to the maximum execution time of task $i$ and $T_i$ corresponds to the execution period of task $i$
- In other words, $E_i$ corresponds to the fraction of CPU time required to execute task $i$
Rate Monotonic Scheduling

- The upper bound for an infinite number of tasks is given by \( \ln(2) \), or 0.693
  - To meet all hard real-time deadlines based on RMS, CPU use of all
time-critical tasks should be less than 70 percent
  - You can still have non-time-critical tasks in a system and thus use
100% of the CPU's time
  - As a rule of thumb, you should always design a system to use less
than 60-70% of your CPU

<table>
<thead>
<tr>
<th>Number of Tasks</th>
<th>( n(2^{\frac{1}{n-1}}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.828</td>
</tr>
<tr>
<td>3</td>
<td>0.779</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.693</td>
</tr>
</tbody>
</table>

Mutual Exclusion

- The easiest way for tasks to communicate with each other is
through shared data structures
- This process is especially easy when all tasks exist in a
single address space and can reference elements, such as
global variables, pointers, buffers, linked lists, etc.
- Although sharing data simplifies the exchange of
information, you must ensure that each task has exclusive
access to the data to avoid contention and data corruption
- Common methods of obtaining exclusive access to a share
resource
  - Disabling interrupts,
  - Performing test-and-set operations,
  - Disabling scheduling, and
  - Using semaphores
Mutual Exclusion

- Disabling and Enabling Interrupts
- uC/OS-II provides two macros that disable and then enable interrupts from your C code:
  - OS_ENTER_CRITICAL()
  - OS_EXIT_CRITICAL()

```c
void function(void) {
    OS_ENTER_CRITICAL();
    // Access shared data here
    OS_EXIT_CRITICAL();
}
```

Mutual Exclusion and Latency

- Do not disable interrupts for too long
- Doing so affects the response of your system to interrupts (interrupt latency)
- You should consider this method when you are changing or copying a few variables
- Also, this method is the only way that a task can share variables or data structures with an ISR
- In all cases, you should keep interrupts disabled for as little time as possible
- If you use a kernel, you are basically allowed to disable interrupts for as much time as the kernel does without affecting interrupt latency
Test-and-Set Operations

- If you are not using a kernel, two functions could agree that to access a resource, they must check a global variable and if the variable is 0, the function has access to the resource.
- To prevent the other function from accessing the resource, however, the first function that gets the resource sets the variable to 1, which is called a test-and-set (or TAS) operation.
- Either the TAS operation must be performed indivisibly (by the processor), or you must disable interrupts when doing the TAS on the variable.

Test-and-Set Pseudocode

```plaintext
Disable interrupts;
if ("access variable" is 0) {
    Set variable to 1;
    Re-enable interrupts;
    Access the resource;
    Disable interrupts;
    Set the "access variable" back to 0;
    Re-enable interrupts;
} else {
    // No access to the resource
    Re-enable interrupts;
}
```
Disabling and Enabling the Scheduler

• If your task is not sharing variables or data structures with an ISR, you can disable and enable scheduling

• In this case, two or more tasks can share data without the possibility of contention

• You should note that while the scheduler is locked, interrupts are enabled, and, if an interrupt occurs while in the critical section, the ISR is executed immediately

• At the end of the ISR, the kernel always returns to the interrupted task, even if the ISR has made a higher priority task ready to run
  – Similar to a non-preemptive kernel

Disabling and Enabling the Scheduler

• uC/OS-II provides two functions that disable and then enable the scheduler from your C code:
  – OSSchedLock()
  – OSSchedUnlock()

    void function(void) {
      OSSchedLock();
      // Access shared data here
      // Interrupts are still enabled
      OSSchedUnlock();
    }

• Not the best method
  – Defeats the purpose of having the kernel in the first place
Semaphores

• A protocol mechanism offered by most multitasking kernels

• Semaphores are used to:
  – Control access to a shared resource (mutual exclusion),
  – Signal the occurrence of an event, and
  – Allow two tasks to synchronize their activities

• A semaphore is a key that your code acquires in order to continue execution
  – If the semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner
  – In other words, the requesting task says: "Give me the key. If someone else is using it, I am willing to wait for it!"

Semaphore Types

• Two types of semaphores exist:
  – Binary
  – Counting

• A binary semaphore can only take two values: 0 or 1

• A counting semaphore allows values depending on whether the semaphore mechanism is implemented using 8, 16, or 32 bits, respectively
  – The actual size depends on the kernel used
  – Along with the semaphore's value, the kernel also needs to keep track of tasks waiting for the semaphore's availability
Semaphore Operations

- Generally, only three operations can be performed on a semaphore:
  - INITIALIZE (also called CREATE),
  - WAIT (also called PEND), and
  - SIGNAL (also called POST)
- The initial value of the semaphore must be provided when the semaphore is initialized
- The waiting list of tasks is always initially empty

Obtaining a Semaphore

- A task desiring the semaphore performs a WAIT operation
  - If the semaphore is available (the semaphore value is greater than 0), the semaphore value is decremented, and the task continues execution
  - If the semaphore’s value is 0, the task performing a WAIT on the semaphore is placed in a waiting list
- Most kernels allow you to specify a timeout
  - If the semaphore is not available within a certain amount of time, the requesting task is made ready to run, and an error code (indicating that a timeout has occurred) is returned to the caller
Releasing a Semaphore

• A task releases a semaphore by performing a SIGNAL operation
• If no task is waiting for the semaphore, the semaphore value is simply incremented
• If any task is waiting for the semaphore, however, one of the tasks is made ready to run, and the semaphore value is not incremented
  – The "key" is given (by the kernel) to one of the tasks waiting for it
• Depending on the kernel, the task that receives the semaphore is either:
  – The highest priority task waiting for the semaphore (uC/OS-II)
  – First task requesting the semaphore (FIFO)

Sharing Data in uC/OS-II Using Semaphores

• A semaphore is an object that needs to be initialized before it's used; for mutual exclusion, a semaphore is initialized to a value of 1
• Using a semaphore to access shared data doesn't affect interrupt latency
  – If an ISR or the current task makes a higher priority task ready to run while accessing shared data, the higher priority task executes immediately

```c
OS_EVENT *SharedDataSem;

void function(void) {
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    // Access shared data here
    // Interrupts are still enabled
    OSSemPost(SharedDataSem);
}
```
Semaphore Use

- Semaphores are especially useful when tasks share I/O devices

Encapsulating Semaphores

- The previous example implies that each task must know about the existence of the semaphore in order to access the resource
- In some situations, it is better to encapsulate the semaphore
- Each task would thus not know that it is actually acquiring a semaphore when accessing the resource
Encapsulating Semaphores

```
INT8U CommSendCmd(char *cmd, char *response, INT16U timeout)
{
    Acquire port’s semaphore;
    Send command to device;
    Wait for response (with timeout);
    if(timed out) {
        Release semaphore;
        return(error code);
    } else {
        Release semaphore;
        return(no error);
    }
}
```

- Each task that needs to send a command to the device has to call this function
- Semaphore is assumed to be initialized (i.e. available) by the communication driver initialization routine

Notes on Semaphore Use

- Semaphores are often overused
- The use of a semaphore to access a simple shared variable is overkill in most situations
  - The overhead involved in acquiring and releasing the semaphore can consume valuable time
  - You can do the job just as efficiently by disabling and enabling interrupts

- Rule of thumb for variable access: Use semaphores only when the time for the operation to be performed exceeds the interrupt latency time of the kernel