MicroC-OS: The Real Time Kernel

Real-Time Kernel Concepts

Real-time systems are characterized by the fact that severe consequences will result if logical as well as timing correctness properties of the system are not met. There are two types of real-time systems: *soft* and *hard*. In a *soft* real-time systems, tasks are performed by the system as fast as possible, but don't have to finish by specific times. In *hard* real-time systems, tasks have to be performed not only correctly, but also in a timely fashion. This book assumes a *soft* real-time system.

Real-time software applications are typically more difficult to design than non real-time applications. The use of a real-time kernel will simplify the design process by allowing the application to be divided into multiple tasks managed by the kernel.

Multitasking

Multitasking is the process of scheduling and switching the CPU between several tasks; a single CPU switches it attention between several sequential tasks. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherit in real-time applications. Application programs are typically easier to design and maintain if multitasking is used.

Task

A task, also called a thread, is a simple program that thinks it has the CPU all to itself. Each task is assigned a priority, its own set of CPU registers, and its own stack area. Each task is typically an infinite loop that can be any one of six states: dormant, ready, running, delayed, waiting for an event, or interrupted.

Context Switch or Task Switch

When the multitasking kernel decides to run a different task, it simply saves the current task's context (CPU registers) in the current task's context storage area.

Non-Preemptive Kernel

Non-Preemptive kernels require that each task does something to explicitly give up control of the CPU. Non-preemptive schedule is also called *cooperative multitasking*; tasks cooperate with each other to relinquish control of the CPU. Non-preemptive kernels are much simpler to design than preemptive kernels. One advantage of a non-preemptive kernel is that interrupt latency is typically low. Non-preemptive kernel can also make use of non-reentrant functions (at the task level). Non-reentrant function can be used by each task without fear of corruption by another task. This is because each task can run to completion before it relinquishes the CPU. Non-reentrant functions, however, should not be allowed to give up control of the CPU.

Another advantage of non-preemptive kernels is the smaller need to guard shared data throughout the use of semaphores protecting shared variables, because each task owns the CPU without the fear of being preempted. This is not an absolute rule and in some instances, semaphores should still be used. Shared I/O devices may require the use of mutual exclusion semaphores; for example, a task might still need exclusive access to a printer.

The only way a task can be preempted is from an interrupt; an *Interrupt Service Routine* (ISR) always has priority over a task. The ISR always returns to the interrupted task. The most important drawback of a non-preemptive kernel is responsiveness. A higher priority task that has been made ready to run may have to wait a long time to run, because the current task must give up the CPU when it is ready to do so. Most real-time kernels are preemptive because of this possible delay. Non-preemptive kernels are non-deterministic; you never really know when the highest priority task will get control of the CPU.

Preemptive Kernels

µC/OS is a preemptive kernel. A preemptive kernel is used when system responsiveness is important. The highest priority task ready to run is always given control of the CPU.

With a preemptive kernel, execution of the highest priority task is deterministic; you can determine *when* the highest priority task will get control of the CPU.

Preemptive kernels should not make use of non-reentrant functions unless exclusive access is ensured though the use of mutual exclusion semaphores, because both a low priority task and high priority task can make use of a common function. Corruption may occur if the higher priority task preempts a lower priority task which is making use of the function.

Reentrancey

A reentrant function is a function that can be used by more than one task without the fear of corruption. A reentrant function can be interrupted at any time and resume at a later time without loss of data. Reentrant functions either use local variables (i.e. CPU registers or variables on the stack) or protect data when global variables are used.

Static/Dynamic Priority

Task priorities are said to be *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.

It is said to be dynamic if the priority of the task can be changed during runtime. µC/OS allows task priorities to be changed dynamically.

Priority Inversions

Priority inversion is any situation in which a low priority task holds a resource while a higher priority task is ready to use it. In this situation the low priority task prevents the high priority task from executing until it releases the resource. To correct this situation, the priority of the low priority task can be raised while accessing the resource and restore it to its original value when done. A multitasking kernel should thus allow priorities to be changed dynamically.

Semaphores

Semaphores are used to: 1. Control access to a shared resource (mutual exclusion) 2. Signal the occurrence of an event 3. 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.

There are two types of semaphores: binary and counting semaphores. There are generally only three operations that 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 initialled. The waiting list of tasks is always initially empty.

A task desiring the semaphore will perform a *WAIT* operation. If the semaphore is available (greater than 0), the semaphore will decrement and the task continues execution. 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 an error code (indicating a timeout occurred) is returned to it.

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 no incremented; the key is given to a waiting task. Depending on the kernel used, the task which will receive the semaphore is either:

1. The highest priority task waiting
2. The first task the requested the semaphore

Mutual Exclusion

To prevent multiple items accessing the same thing at the same time (think two tasks running on a printer at the same time), you can create exclusive access to the item (say the printer) by utilizing a binary semaphore. As an example, an RS-232C port is used by multiple tasks to send commands and receive responses from a device. The function `CommSendCmd()` is called with three arguments: the ASCII string containing the command, a pointer to the response string from the device, and finally a timeout in case the devices doesn't respond.

```
UBYTE CommSendCmd(char* cmd, char* response, UWORD timeout)
{
    Acquire port's semaphore;
    Send command to the device;
    Wait for response (with timeout);
    if (timed out)
    {
        Release semaphore;
        return(error code);
    }
```

```
    else
    {
        Release semaphore;
        return (no error);
    }
}
```

Semaphores are often overused. The use of a semaphore to access a simple shared variable is overkill in some situations. The overhead of acquiring and releasing th semaphore can be time consuming. Disabling and enabling interrupts could do the job more efficiently. All real-time kernels will disable interrupts during critical sections of code. You are thus basically allowed to disable interrupts for as much as the kernel does without affecting interrupt latency.

For example, supposed that two tasks are sharing a 16 bit integer. If you consider how long a processor takes to perform either task's respective operation, you do not need a semaphore to gain exclusive access to the variable. Each task simply needs to disable interrupts before performing its operation on the variable and enable interrupts when done. A semaphore should be used, however, if the variable to manipulate is a floating point and the microprocessor doesn't support floating point in hardware. In this case, the processing time could affect interrupt latency.

Deadlock (or Deadly Embrace)

A deadlock is a situation in which two tasks are unknowingly waiting for resources that hare held by each other. For example, if task `T1` has exclusive access to resource `R1` and task `T2` has exclusive access to resource `R2`. Now if `T1` needs exclusive access to `R2` and `T2` also needs exclusive access to `R1`, neither task can continue. The simplest way to avoid this is:

1. Acquire all resources before proceeding
2. Acquire the resource in the same order

Synchronization

A task can by synchronized with an interrupt service routine (ISR) or another task when no data is being exchanged by using a semaphore. When used as a synchronization mechanism, the semaphore is initialized to 0. Using a semaphore for th is type of synchronization is called a *unilateral rendezvous*.

Note that more than one task can be waiting for the event to occur. In this case, the kernel could signal the occurrence of the event either to:

1. The highest priority task waiting for the event to occur
2. The first task waiting for the event

Intertask Communication

It is sometimes necessary for a task or an ISR to communicate information to another task. Information may be communicated in two ways: through global data and by sending messages. Note that a task can only communicate with an ISR by using global variables.

Message Mailbox

Messages can be sent to a task through kernel services. The two most common kernel services for sending messages are the Message Mailbox and Message Queue. A Message Mailbox is typically a pointer size variable. Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox. Similarly, one or more task can receive messages through a service provided by the kernel. Both sending and receiving task will agree as to what the pointer is pointing to.

A message queue is simply an array of mailboxes.

ISR Processing

You should consider whether the overhead involved in signaling a task is more than the processing of the interrupt. Signaling a task from an ISR (i.e. through a semaphore, mailbox, or queue) requires some processing time. If processing of your interrupt requires less than the time required to signal a task, you should consider processing the interrupt in the ISR itself.

Non-Maskable Interrupts (NMIs)

Sometimes, an interrupt must be serviced as quickly as possible and cannot afford to have the latency imposed by the kernel. In these situation, you may be able to use the Non-Maskable Interrupt (MSI) provided on most microprocessors. The NMI is generally reserved for drastic measures such as saving important information during a power down.

When you are servicing an NMI, you cannot use kernel services to signal a task. NMIs cannot be disable to access critical section of code. You can, however, still pass parameters to and from the NMI. Parameters passed must be global variables and the size of these variables must be read or written indivisibly, that is, not as separate byte read or write instructions.