10 Real-Time Operating Systems

Contents

Introduction 10.2

10.1 Real-Time Kernel Concepts 10.3
- 10.1.1 Threads 10.3
- 10.1.2 Context Switch 10.4
- 10.1.3 Kernel 10.4
- 10.1.4 Scheduler 10.4

10.2 Reentrancy 10.6

10.3 Thread Priority.. 10.7
- 10.3.1 Static Priorities 10.7
- 10.3.2 Dynamic Priorities... 10.7
- 10.3.3 Priority Inversions 10.7

10.4 Mutual Exclusion... 10.8
- 10.4.1 Disabling and Enabling Interrupts 10.8
- 10.4.2 Semaphores... 10.9
- 10.4.3 Deadlock (or Deadly Embrace) 10.16

10.5 Synchronization 10.17

10.6 Interthread Communication 10.19
- 10.6.1 Message Mailboxes 10.20
- 10.6.2 Message Queues 10.21

10.7 Interrupts.. 10.23
- 10.7.1 Interrupt Latency 10.24
- 10.7.2 Interrupt Response 10.24
- 10.7.3 Interrupt Recovery 10.25
- 10.7.4 Interrupt Latency, Response, and Recovery 10.26
- 10.7.5 ISR Processing Time 10.27
- 10.7.6 Clock Tick 10.28

10.8 Memory Requirements 10.30

10.9 Advantages and Disadvantages of Real-Time Operating Systems 10.32

10.2

Introduction

A real-time operating system (RTOS) for an embedded system simplifies the design of real-time software by allowing the application to be divided into multiple threads managed by the RTOS. The *kernel* of an embedded RTOS needs to support multithreading, pre-emption, and thread priority. The RTOS will also provide services to threads for communication, synchronization and coordination. A RTOS is to be used for a “hard” real-time system – i.e. threads have to be performed not only correctly but also in a timely fashion.

Operating systems for larger computers (such as the PC) are non-real-time operating systems and usually provide a much larger range of application services, such as memory management and file management which normally do not apply to embedded systems.

10.1 Real-Time Kernel Concepts

The following sections describe real-time kernel concepts.

10.1.1 Threads

A thread is a simple program that thinks it has the CPU all to itself. The design process for a real-time application involves splitting the work to be done into threads which are responsible for a portion of the problem. Each thread is assigned a priority, its own set of CPU registers and its own stack area.

Each thread is typically an infinite loop that can be in one of four states: `READY`, `RUNNING`, `WAITING` or `INTERRUPTED`.

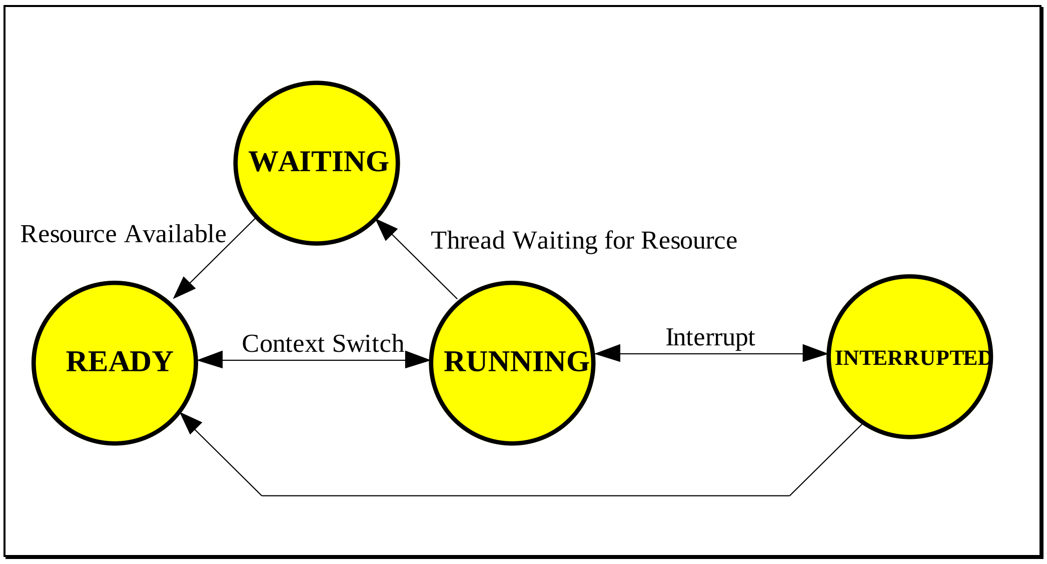

Figure 10.1 – Thread states

A thread is `READY` when it can execute but its priority is less than the current running thread. A thread is `RUNNING` when it has control of the CPU. A thread is `WAITING` when the thread suspends itself until a certain amount of time has elapsed, or when it requires the occurrence of an event: waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur etc. Finally, a thread is `INTERRUPTED` when an interrupt occurred and the CPU is in the process of servicing the interrupt.

10.1.2 Context Switch

When the multithreading kernel decides to run a different thread, it simply saves the current thread's context (CPU registers) in the current thread's context storage area (the *thread control block*, or TCB). Once this operation is performed, the new thread's context is restored from its TCB and the CPU resumes execution of the new thread's code. This process is called a context switch. Context switching adds overhead to the application.

10.1.3 Kernel

The kernel is the part of an OS that is responsible for the management of threads (i.e., managing the CPU's time) and for communication between threads. The fundamental service provided by the kernel is context switching.

10.1.4 Scheduler

The scheduler is the part of the kernel responsible for determining which thread will run next. Most real-time kernels are priority based. Each thread is assigned a priority based on its importance. Establishing the priority for each thread is application specific. In a priority-based kernel, control of the CPU will always be given to the highest priority thread ready to run. In a preemptive kernel, when a thread makes a higher priority thread ready to run, the current thread is pre-empted (suspended) and the higher priority thread is immediately given control of the CPU. If an interrupt service routine (ISR) makes a higher priority thread ready, then when the ISR is completed the interrupted thread is suspended and the new higher priority thread is resumed.

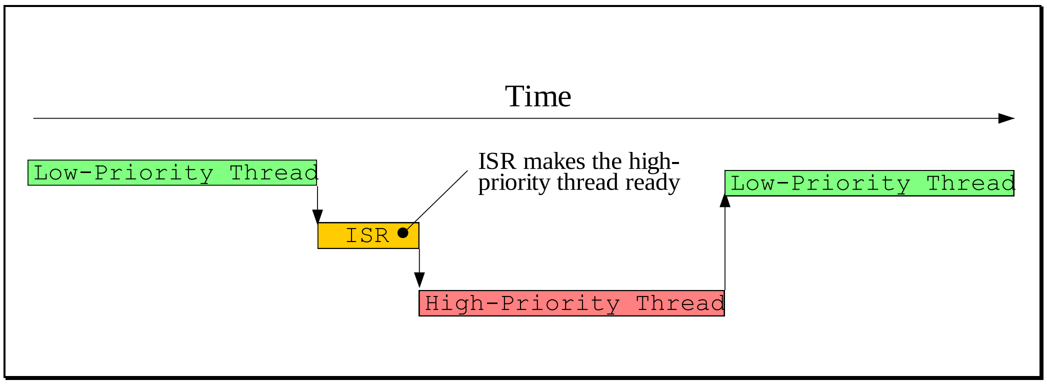

Figure 10.2 – Preemptive kernel

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

Application code using a preemptive kernel should not use non-reentrant functions, unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores, because both a low- and a high-priority thread can use a common function. Corruption of data may occur if the higher priority thread preempts a lower priority thread that is using the function.

To summarize, a preemptive kernel always executes the highest priority thread that is ready to run. An interrupt preempts a thread. Upon completion of an ISR, the kernel resumes execution to the highest priority thread ready to run (not the interrupted thread). Thread-level response is optimum and deterministic.

10.2 Reentrancy

A reentrant function can be used by more than one thread without fear of data corruption. A reentrant function can be interrupted at any time and resumed 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. An example of a reentrant function is shown below:

```
char* strcpy(char* dst, const char* src)
{
  char* ptr = dst;
  while (*dst++ = *src++);
  return ptr;
}
```

Since copies of the arguments to `strcpy()` are placed on the thread's stack, and the local variable is created on the thread’s stack, `strcpy()` can be invoked by multiple threads without fear that the threads will corrupt each other's pointers.

An example of a non-reentrant function is shown below:

```
static int Temp;

void swap(int* x, int* y)
{
  Temp = *x;
  *x   = *y;
  *y   = Temp;
}
```

`swap()` is a simple function that swaps the contents of its two arguments. Since `Temp` is a global variable, if the `swap()` function gets preempted after the first line by a higher priority thread which also uses the `swap()` function, then when the low priority thread resumes it will use the `Temp` value that was used by the high priority thread.

You can make `swap()` reentrant with one of the following techniques:

- Declare `Temp` local to `swap()`.
- Disable interrupts before the operation and enable them afterwards.
- Use a semaphore.

10.3 Thread Priority

A priority is assigned to each thread. The more important the thread, the higher the priority given to it.

10.3.1 Static Priorities

Thread priorities are said to be static when the priority of each thread does not change during the application's execution. Each thread is thus given a fixed priority at compile time. All the threads and their timing constraints are known at compile time in a system where priorities are static.

10.3.2 Dynamic Priorities

Thread priorities are said to be dynamic if the priority of threads can be changed during the application's execution; each thread can change its priority at run time. This is a desirable feature to have in a real-time kernel to avoid priority inversions.

10.3.3 Priority Inversions

Priority inversion is a problem in real-time systems and occurs mostly when you use a real-time kernel. Priority inversion is any situation in which a low priority thread holds a resource while a higher priority thread is ready to use it. In this situation the low priority thread prevents the high priority thread from executing until it releases the resource.

To avoid priority inversion a multithreading kernel should change the priority of a thread automatically to help prevent priority inversions. This is called priority inheritance.

10.4 Mutual Exclusion

The easiest way for threads to communicate with each other is through shared data structures. This is especially easy when all threads exist in a single address space and can reference global variables, pointers, buffers, linked lists, FIFOs, etc. Although sharing data simplifies the exchange of information, you must ensure that each thread has exclusive access to the data to avoid contention and data corruption. The most common methods of obtaining exclusive access to shared resources are:

- disabling interrupts,
- performing test-and-set operations,
- disabling scheduling, and
- using semaphores.

10.4.1 Disabling and Enabling Interrupts

The easiest and fastest way to gain exclusive access to a shared resource is by disabling and enabling interrupts, as shown in the pseudocode:

```
Disable interrupts;
Access the resource (read/write from/to variables);
Reenable interrupts;
```

Kernels use this technique to access internal variables and data structures. In fact, kernels usually provide two functions that allow you to disable and then enable interrupts from your C code: **`OS_EnterCritical`**`()` and **`OS_ExitCritical`**`()`, respectively. You need to use these functions in tandem, as shown below:

```
void Function(void)
{
  OS_EnterCritical();
  .
  .    /* You can access shared data in here */
  .
  OS_ExitCritical();
}
```

You must be careful, however, not to disable interrupts for too long because this affects the response of your system to interrupts. This is known as interrupt latency. You should consider this method when you are changing or copying a few variables. Also, this is the only way that a thread 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. Obviously, you need to know how long the kernel will disable interrupts.

10.4.2 Semaphores

The semaphore was invented by Edgser Dijkstra in the mid-1960s. It is a protocol mechanism offered by most multithreading kernels. Semaphores are used to:

- control access to a shared resource (mutual exclusion),
- signal the occurrence of an event, and
- allow two threads 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 thread is suspended until the semaphore is released by its current owner. In other words, the requesting thread says: "Give me the key. If someone else is using it, I am willing to wait for it!" There are two types of semaphores: binary semaphores and counting semaphores. As its name implies, a binary semaphore can only take two values: 0 or 1. A counting semaphore allows values between 0 and 255, 65535, or 4294967295, 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 threads waiting for the semaphore's availability.

Generally, only three operations can be performed on a semaphore: `Create()`, `Wait()`, and `Signal()`. The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of threads is always initially empty.

A thread desiring the semaphore will perform a **`Wait()`** operation. If the semaphore is available (the semaphore value is greater than 0), the semaphore value is decremented and the thread continues execution. If the semaphore's value is 0, the thread 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 thread is made ready to run and an error code (indicating that a timeout has occurred) is returned to the caller.

A thread releases a semaphore by performing a **`Signal()`** operation. If no thread is waiting for the semaphore, the semaphore value is simply incremented. If any thread is waiting for the semaphore, however, one of the threads is made ready to run and the semaphore value is not incremented; the key is given to one of the threads waiting for it. Depending on the kernel, the thread that receives the semaphore is either:

- the highest priority thread waiting for the semaphore, or
- the first thread that requested the semaphore (First In First Out).

Some kernels have an option that allows you to choose either method when the semaphore is initialized. For the first option, if the readied thread has a higher priority than the current thread (the thread releasing the semaphore), a context switch occurs (with a preemptive kernel) and the higher priority thread resumes execution; the current thread is suspended until it again becomes the highest priority thread ready to run.

Listing 10.1 shows how you can share data using a semaphore. Any thread needing access to the same shared data calls **`OS_SemaphoreWait()`**, and when the thread is done with the data, the thread calls **`OS_SemaphoreSignal()`**. Both of these functions are described later. You should note that a semaphore is an object that needs to be initialized before it is 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 thread makes a higher priority thread ready to run while accessing shared data, the higher priority thread executes immediately.

```
OS_ECB* SharedDataSemaphore;

void Function(void)
{
  OS_ERROR error;

  error = OS_SemaphoreWait(SharedDataSemaphore, 0);
  .
  .    // You can access shared data in here
  .    // (interrupts are recognized)
  .
  error = OS_SemaphoreSignal(SharedDataSemaphore);
}
```

Listing 10.1 – Accessing shared data by obtaining a semaphore

Semaphores are especially useful when threads share I/O devices. Imagine what would happen if two threads were allowed to send characters to a printer at the same time. The printer would contain interleaved data from each thread. For instance, the printout from Thread 1 printing "`I am Thread 1!`" and Thread 2 printing "`I am Thread 2!`" could result in:

“`I Ia amm T Threahread d1 !2!`”

In this case, use a semaphore and initialize it to 1 (i.e., a binary semaphore). The rule is simple: to access the printer each thread first must obtain the resource's semaphore.

Figure 10.3 shows threads competing for a semaphore to gain exclusive access to the printer. Note that the semaphore is represented symbolically by a key, indicating that each thread must obtain this key to use the printer.

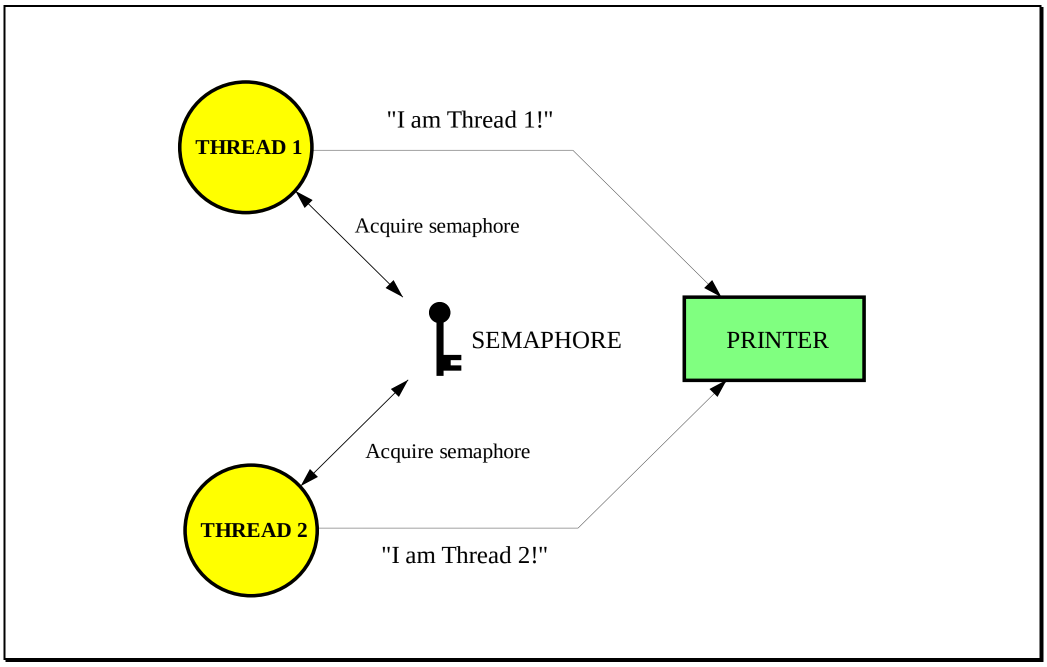

Figure 10.3 – Using a semaphore to get permission to access a printer

The above example implies that each thread must know about the existence of the semaphore in order to access the resource. There are situations when it is better to encapsulate the semaphore. Each thread would thus not know that it is actually acquiring a semaphore when accessing the resource. For example, the UART port may be used by multiple threads to send commands and receive responses from a PC:

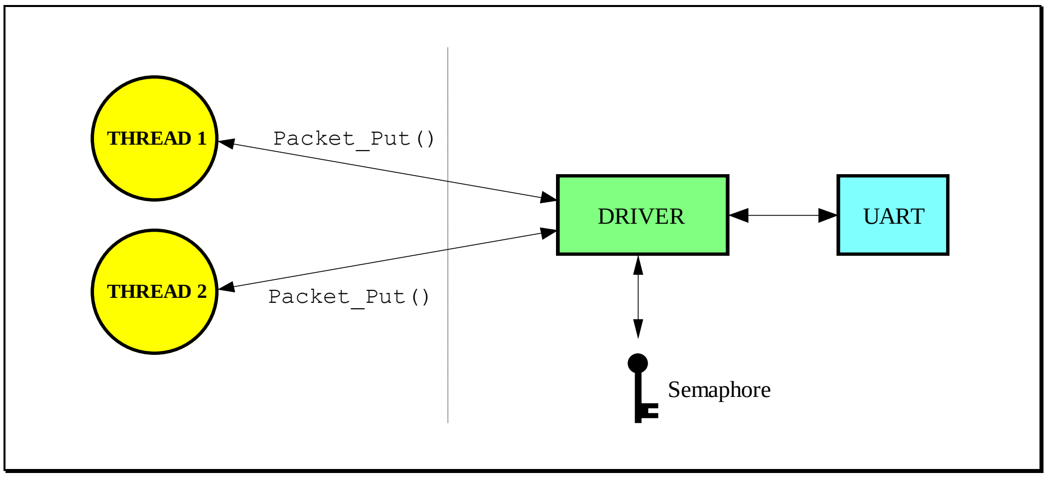

Figure 10.4 – Hiding a semaphore from threads

The function **`Packet_Put`**`()` is called with two arguments: the packet and a timeout in case the device doesn't respond within a certain amount of time. The pseudocode for this function is shown in Listing 10.2.

```
uint8_t Packet_Put(TPacket* packet, const uint16_t timeout)
{
  Acquire serial port's semaphore;
  Send packet to device;
  Wait for response (with timeout);
  Release semaphore;
  if (timed out)
    return (error code);
  else
    return (no error);
}
```

Listing 10.2 – Encapsulating a semaphore

Each thread that needs to send a packet to the serial port has to call this function. The semaphore is assumed to be initialized to 1 (i.e., available) by the communication driver initialization routine. The first thread that calls **`Packet_Put`**`()` acquires the semaphore, proceeds to send the packet, and waits for a response. If another thread attempts to send a command while the port is busy, this second thread is suspended until the semaphore is released. The second thread appears simply to have made a call to a normal function that will not return until the function has performed its duty. When the semaphore is released by the first thread, the second thread acquires the semaphore and is allowed to use the serial port.

A *counting semaphore* is used when a resource can be used by more than one thread at the same time. For example, a counting semaphore is used in the management of a buffer pool as shown in Figure 10.5.

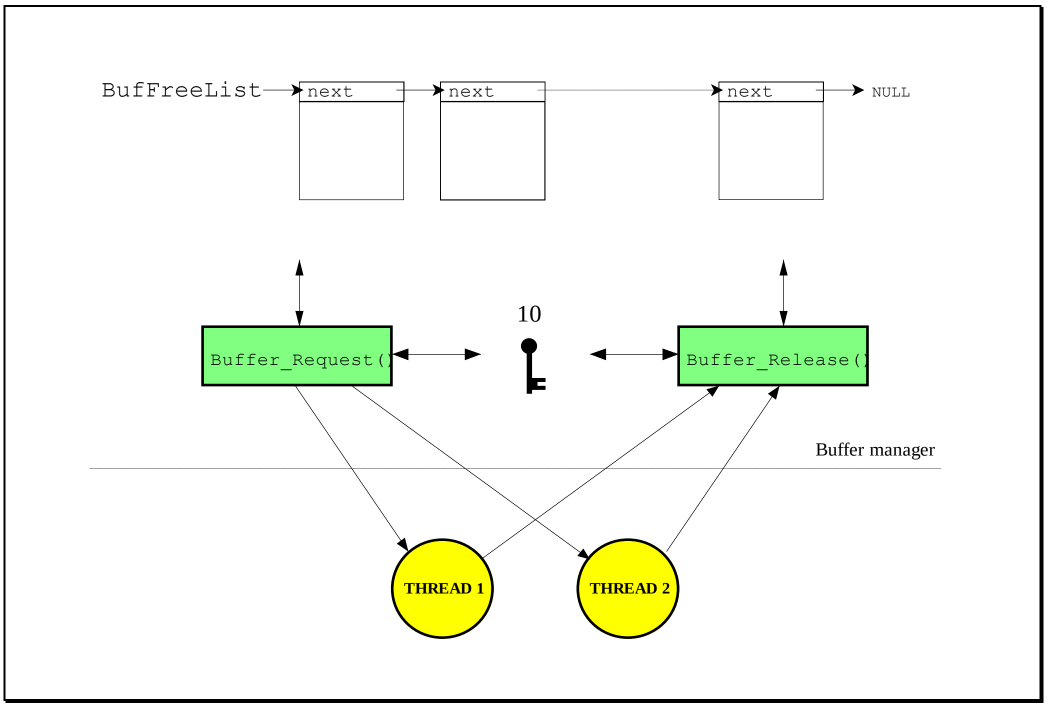

Figure 10.5 – Using a counting semaphore

Assume that the buffer pool initially contains 10 buffers. A thread would obtain a buffer from the buffer manager by calling **`Buffer_Request`**`()`. When the buffer is no longer needed, the thread would return the buffer to the buffer manager by calling **`Buffer_Release`**`()`. The pseudocode for these functions is shown in Listing 10.3.

```
BUF* Buffer_Request(void)
{
  BUF* ptr;

  Acquire a semaphore;
  Disable interrupts;
  ptr          = BufFreeList;
  BufFreeList = ptr->next;
  Enable interrupts;
  return (ptr);
}
```

```
void Buffer_Release(BUF* ptr)
{
  Disable interrupts;
  ptr->next = BufFreeList;
  BufFreeList = ptr;
  Enable interrupts;
  Release semaphore;
}
```

Listing 10.3 – Buffer management using a semaphore

The buffer manager will satisfy the first 10 buffer requests because there are 10 keys. When all semaphores are used, a thread requesting a buffer is suspended until a semaphore becomes available. Interrupts are disabled to gain exclusive access to the linked list (this operation is very quick). When a thread is finished with the buffer it acquired, it calls `Buffer_Release()` to return the buffer to the buffer manager; the buffer is inserted into the linked list before the semaphore is released. By encapsulating the interface to the buffer manager in `Buffer_Request()` and `Buffer_Release()`, the caller doesn't need to be concerned with the actual implementation details.

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. Suppose that two threads are sharing a 32-bit integer variable. The first thread increments the variable while the other thread clears it. If you consider how long a processor takes to perform either operation, you will realize that you do not need a semaphore to gain exclusive access to the variable. Each thread simply needs to disable interrupts before performing its operation on the variable and enable interrupts when the operation is complete. A semaphore should be used, however, if the variable is a floating-point variable and the microprocessor doesn't support floating point in hardware. In this case, the processing time involved in processing the floating-point variable could have affected interrupt latency if you had disabled interrupts.

10.4.3 Deadlock (or Deadly Embrace)

A *deadlock*, also called a *deadly embrace*, is a situation in which two threads are each unknowingly waiting for resources held by the other. Assume thread **`T1`** has exclusive access to resource **`R1`** and thread **`T2`** has exclusive access to resource **`R2`**. If **`T1`** needs exclusive access to **`R2`** and **`T2`** needs exclusive access to **`R1`**, neither thread can continue. They are deadlocked. The simplest way to avoid a deadlock is for threads to:

- acquire all resources before proceeding,
- acquire the resources in the same order, and
- release the resources in the reverse order

Most kernels allow you to specify a timeout when acquiring a semaphore. This feature allows a deadlock to be broken. If the semaphore is not available within a certain amount of time, the thread requesting the resource resumes execution. Some form of error code must be returned to the thread to notify it that a timeout occurred. A return error code prevents the thread from thinking it has obtained the resource. Deadlocks generally occur in large multithreading systems, not in embedded systems.

10.5 Synchronization

A thread can be synchronized with an ISR (or another thread when no data is being exchanged) by using a semaphore as shown in Figure 10.6.

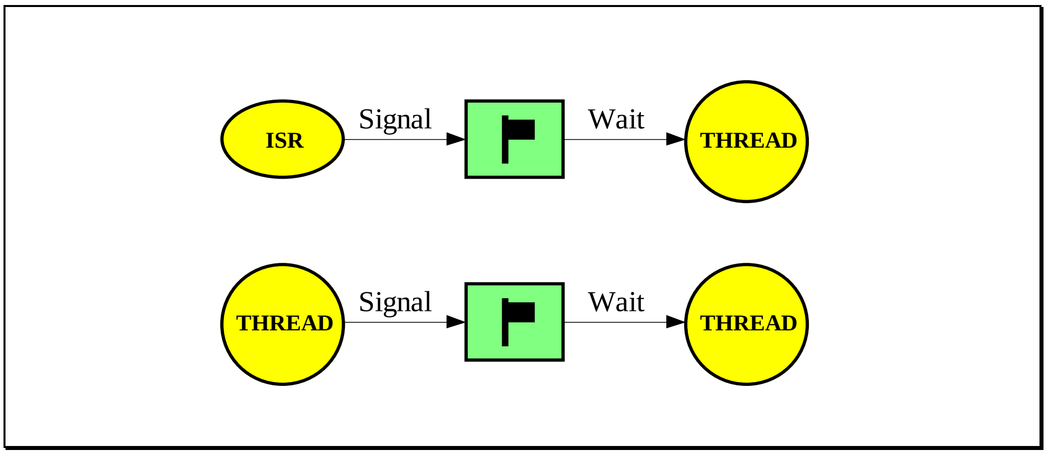

Figure 10.6 – Synchronizing threads and ISRs

Note that, in this case, the semaphore is drawn as a flag to indicate that it is used to signal the occurrence of an event (rather than to ensure mutual exclusion, in which case it would be drawn as a key). When used as a synchronization mechanism, the semaphore is initialized to 0. Using a semaphore for this type of synchronization is called a *unilateral rendezvous*. A thread initiates an I/O operation and waits for the semaphore. When the I/O operation is complete, an ISR (or another thread) signals the semaphore and the thread is resumed.

If the kernel supports counting semaphores, the semaphore would accumulate events that have not yet been processed. Note that more than one thread can be waiting for an event to occur. In this case, the kernel could signal the occurrence of the event either to:

- the highest priority thread waiting for the event to occur or
- the first thread waiting for the event.

Depending on the application, more than one ISR or thread could signal the occurrence of the event.

Two threads can synchronize their activities by using two semaphores, as shown in Figure 10.7. This is called a *bilateral rendezvous*. A bilateral rendezvous is similar to a unilateral rendezvous, except both threads must synchronize with one another before proceeding.

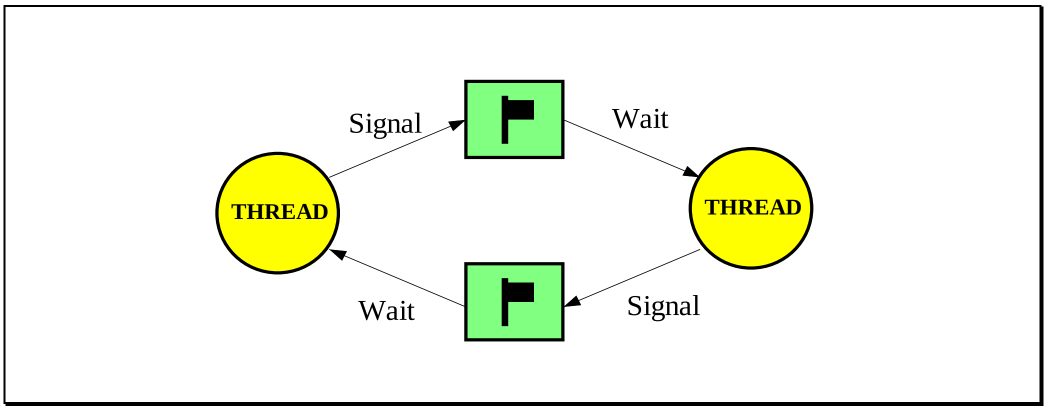

Figure 10.7 – Threads synchronizing their activities

For example, two threads are executing as shown in Listing 10.4. When the first thread reaches a certain point, it signals the second thread (1) then waits for a return signal (2). Similarly, when the second thread reaches a certain point, it signals the first thread (3) and waits for a return signal (4). At this point, both threads are synchronized with each other. A bilateral rendezvous cannot be performed between a thread and an ISR because an ISR cannot wait on a semaphore.

```
void Thread1(void)
{
  for (;;)
  {
    Perform operation 1;
    Signal thread #2;               (1)
    Wait for signal from thread #2; (2)
    Continue operation 1;
  }
}

void Thread2(void)
{
  for (;;)
  {
    Perform operation 2;
    Signal thread #1;               (3)
    Wait for signal from thread #1; (4)
    Continue operation 2;
  }
}
```

Listing 10.4 – Bilateral rendezvous

10.6 Interthread Communication

It is sometimes necessary for a thread or an ISR to communicate information to another thread. This information transfer is called interthread communication. Information may be communicated between threads in two ways: through global data or by sending messages.

When using global variables, each thread or ISR must ensure that it has exclusive access to the variables. If an ISR is involved, the only way to ensure exclusive access to the common variables is to disable interrupts. If two threads are sharing data, each can gain exclusive access to the variables either by disabling and enabling interrupts or with the use of a semaphore (as we have seen). Note that a thread can only communicate information to an ISR by using global variables. A thread is not aware when a global variable is changed by an ISR, unless the ISR signals the thread by using a semaphore or unless the thread polls the contents of the variable periodically. To correct this situation, you should consider using either a message mailbox or a message queue.

10.6.1 Message Mailboxes

Messages can be sent to a thread through kernel services. A Message Mailbox, also called a message exchange, is typically a pointer-size variable. Through a service provided by the kernel, a thread or an ISR can deposit a message (the pointer) into this mailbox. Similarly, one or more threads can receive messages through a service provided by the kernel. Both the sending thread and receiving thread agree on what the pointer is actually pointing to.

A waiting list is associated with each mailbox in case more than one thread wants to receive messages through the mailbox. A thread desiring a message from an empty mailbox is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the thread waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting thread is made ready to run and an error code (indicating that a timeout has occurred) is returned to it. When a message is deposited into the mailbox, either the highest priority thread waiting for the message is given the message (priority based) or the first thread to request a message is given the message (First-In-First-Out, or FIFO). Figure 10.8 shows a thread depositing a message into a mailbox. Note that the mailbox is represented by an I-beam and the timeout is represented by an hourglass. The number next to the hourglass represents the number of clock ticks the thread will wait for a message to arrive.

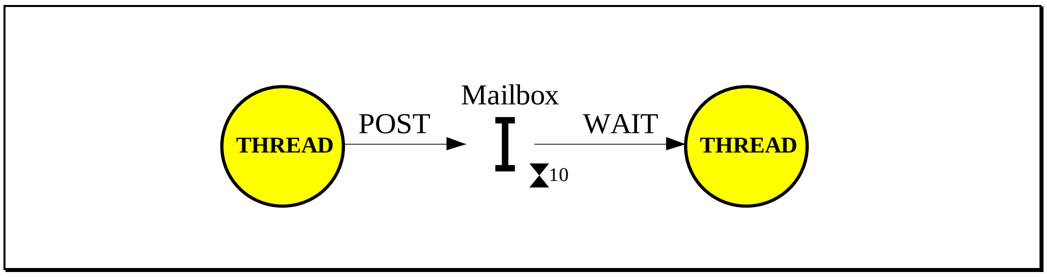

Figure 10.8 – Message mailbox

Kernels typically provide the following mailbox services:

- Initialize the contents of a mailbox. The mailbox initially may or may not contain a message.
- Deposit a message into the mailbox (POST).
- Wait for a message to be deposited into the mailbox (WAIT).
- Get a message from a mailbox if one is present, but do not suspend the caller if the mailbox is empty (ACCEPT). If the mailbox contains a message, the message is extracted from the mailbox. A return code is used to notify the caller about the outcome of the call.

Message mailboxes can also simulate binary semaphores. A message in the mailbox indicates that the resource is available, and an empty mailbox indicates that the resource is already in use by another thread.

10.6.2 Message Queues

A message queue is used to send one or more messages to a thread. A message queue is basically an array of mailboxes. Through a service provided by the kernel, a thread or an ISR can deposit a message (the pointer) into a message queue. Similarly, one or more threads can receive messages through a service provided by the kernel. Both the sending thread and receiving thread agree as to what the pointer is actually pointing to. Generally, the first message inserted in the queue will be the first message extracted from the queue (FIFO).

As with the mailbox, a waiting list is associated with each message queue, in case more than one thread is to receive messages through the queue. A thread desiring a message from an empty queue is suspended and placed on the waiting list until a message is received. Typically, the kernel allows the thread waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting thread is made ready to run and an error code (indicating a timeout has occurred) is returned to it. When a message is deposited into the queue, either the highest priority thread or the first thread to wait for the message is given the message. Figure 10.9 shows an ISR (Interrupt Service Routine) depositing a message into a queue. Note that the queue is represented graphically by a double I-beam. The "10" indicates the number of messages that can accumulate in the queue. A "0" next to the hourglass indicates that the thread will wait forever for a message to arrive.

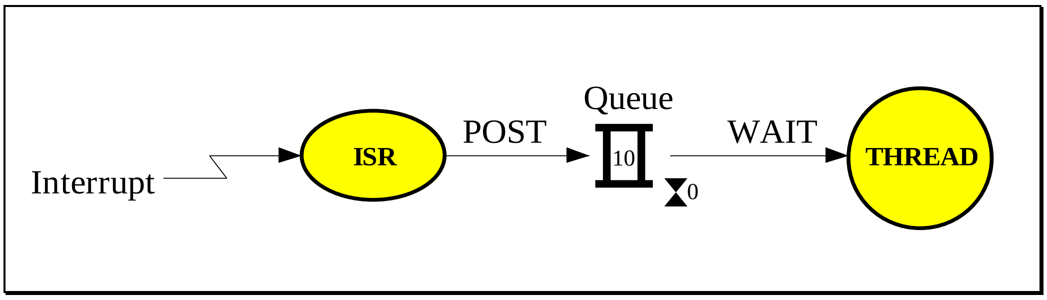

Figure 10.9 – Message queue

Kernels typically provide the message queue services listed below.

- Initialize the queue. The queue is always assumed to be empty after initialization.
- Deposit a message into the queue (POST).
- Wait for a message to be deposited into the queue (WAIT).
- Get a message from a queue if one is present, but do not suspend the caller if the queue is empty (ACCEPT). If the queue contains a message, the message is extracted from the queue. A return code is used to notify the caller about the outcome of the call.

10.7 Interrupts

An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves all of its context (i.e., registers) and jumps to a special subroutine called an Interrupt Service Routine, or ISR. The ISR processes the event, and upon completion of the ISR, the program returns to:

- the background for a foreground / background system,
- the interrupted thread for a non-preemptive kernel, or
- the highest priority thread ready to run for a preemptive kernel.

Interrupts allow a microprocessor to process events when they occur. This prevents the microprocessor from continuously polling an event to see if it has occurred. Microprocessors allow interrupts to be ignored and recognized through the use of two special instructions: disable interrupts and enable interrupts, respectively. In a real-time environment, interrupts should be disabled as little as possible. Disabling interrupts affects interrupt latency and may cause interrupts to be missed. Processors generally allow interrupts to be nested. This means that while servicing an interrupt, the processor will recognize and service other (more important) interrupts, as shown in Figure 10.10.

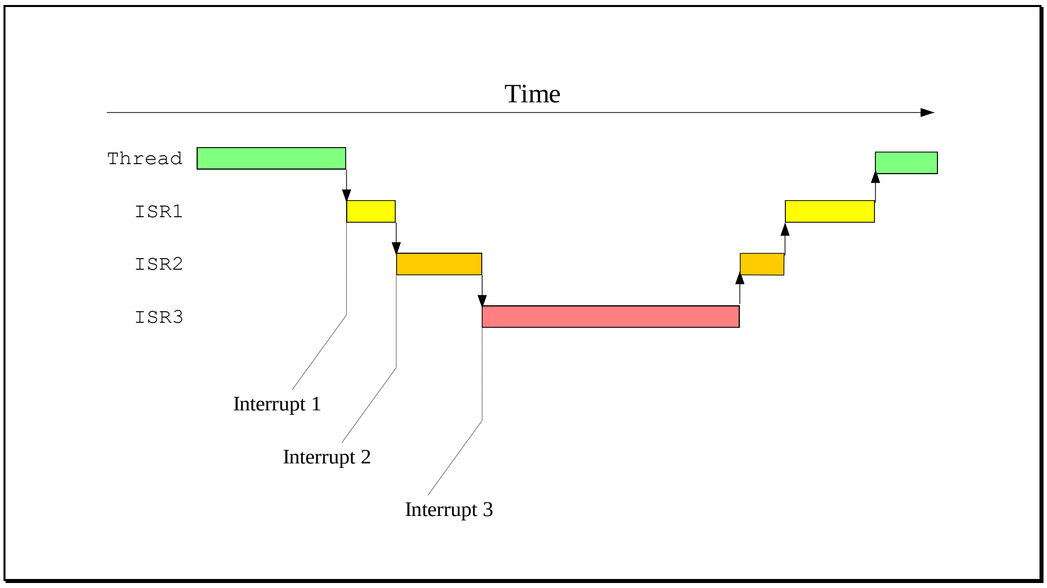

Figure 10.10 – Interrupt nesting

10.7.1 Interrupt Latency

Probably the most important specification of a real-time kernel is the amount of time interrupts are disabled. All real-time systems disable interrupts to manipulate critical sections of code and reenable interrupts when the critical section has executed. The longer interrupts are disabled, the higher the interrupt latency. Interrupt latency is given by Eq. (10.1).

$$\begin{aligned}&\text{Interrupt latency}\\&= \text{Maximum amount of time interrupts are disabled}\\&+ \text{Time to start executing the first instruction in the ISR}\end{aligned} \tag{10.1}$$

10.7.2 Interrupt Response

Interrupt response is defined as the time between the reception of the interrupt and the start of the user code that handles the interrupt. The interrupt response time accounts for all the overhead involved in handling an interrupt.

For a foreground / background system, the user ISR code is executed immediately. The response time is given by Eq. (10.2).

$$\begin{aligned}&\text{Interrupt response time}\\&= \text{Interrupt latency}\end{aligned} \tag{10.2}$$

For a preemptive kernel, a special function provided by the kernel needs to be called. This function notifies the kernel that an ISR is in progress and allows the kernel to keep track of interrupt nesting. This function is called `OS_ISREnter()`. The response time to an interrupt for a preemptive kernel is given by Eq. (10.3).

$$\begin{aligned}&\text{Interrupt response time}\\&= \text{Interrupt latency}\\&+ \text{Execution time of the kernel ISR entry function}\end{aligned} \tag{10.3}$$

A system's worst case interrupt response time is its only response time. Your system may respond to interrupts in 50ms 99 percent of the time, but if it responds to interrupts in 250ms the other 1 percent, you must assume a 250ms interrupt response time.

10.7.3 Interrupt Recovery

Interrupt recovery is defined as the time required for the processor to return to the interrupted code. Interrupt recovery in a foreground / background system simply involves restoring the processor's context and returning to the interrupted thread. Interrupt recovery is given by Eq. (10.4).

Interrupt recovery time

= Time to execute the return from interrupt instruction (10.4)

For a preemptive kernel, interrupt recovery is more complex. Typically, a function provided by the kernel is called at the end of the ISR. This function is called **`OS_ISRExit()`** and allows the kernel to determine if all interrupts have nested. If they have nested (i.e., a return from interrupt would return to thread-level code), the kernel determines if a higher priority thread has been made ready to run as a result of the ISR. If a higher priority thread is ready to run as a result of the ISR, this thread is resumed. Note that, in this case, the interrupted thread will resume only when it again becomes the highest priority thread ready to run. For a preemptive kernel, interrupt recovery is given by Eq. (10.5).

Interrupt recovery time

= Time to determine if a higher priority thread is ready

+ Time to restore the CPU's context of the highest priority thread

+ Time to execute the return from interrupt instruction (10.5)

10.7.4 Interrupt Latency, Response, and Recovery

Figure 10.11 and Figure 10.12 show the interrupt latency, response, and recovery for a foreground / background system and a preemptive kernel, respectively.

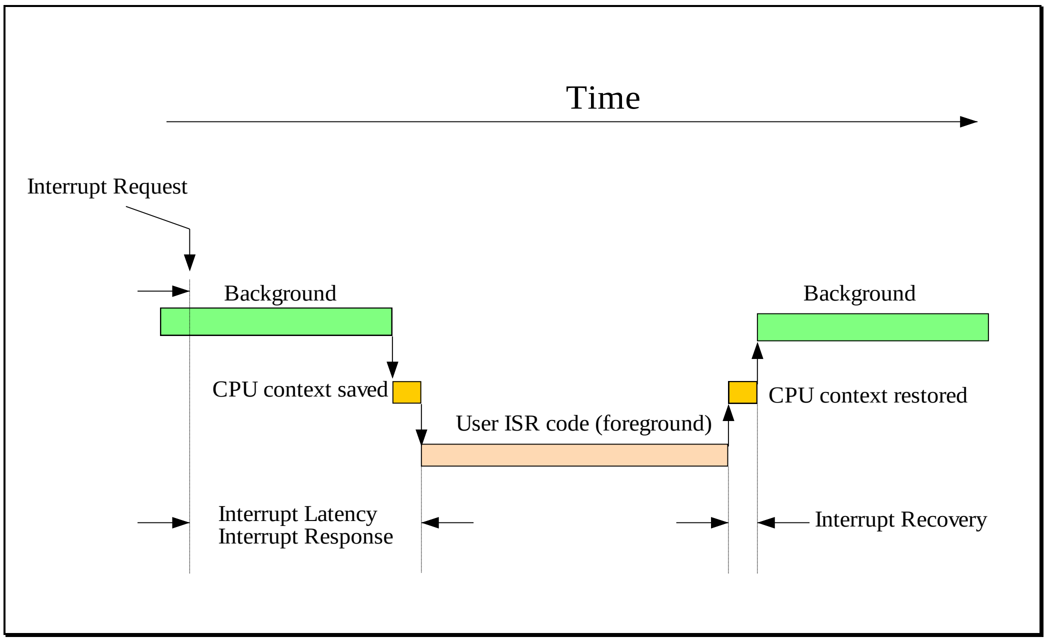

Figure 10.11 – Interrupt latency, response, and recovery (foreground / background)

You should note that for a preemptive kernel, the exit function either decides to return to the interrupted thread (A) or to a higher priority thread that the ISR has made ready to run (B). In the latter case, the execution time is slightly longer because the kernel has to perform a context switch.

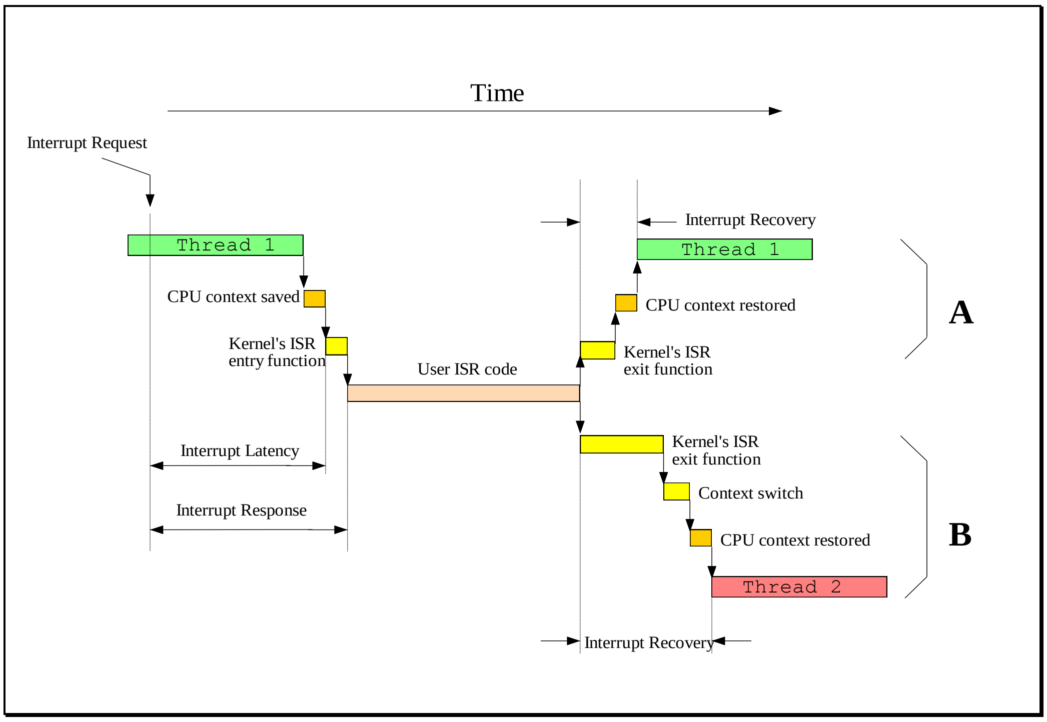

Figure 10.12 – Interrupt latency, response, and recovery (preemptive kernel)

10.7.5 ISR Processing Time

Although ISRs should be as short as possible, there are no absolute limits on the amount of time for an ISR. One cannot say that an ISR must always be less than 100 ms, 500 ms, or l ms. If the ISR code is the most important code that needs to run at any given time, it could be as long as it needs to be. In most cases, however, the ISR should recognize the interrupt, obtain data or a status from the interrupting device, and signal a thread to perform the actual processing. You should also consider whether the overhead involved in signalling a thread is more than the processing of the interrupt. Signalling a thread from an ISR (i.e., through a semaphore, a mailbox, or a queue) requires some processing time. If processing your interrupt requires less than the time required to signal a thread, you should consider processing the interrupt in the ISR itself and allowing higher priority interrupts to be recognized and serviced.

10.7.6 Clock Tick

A clock tick is a special interrupt that occurs periodically. This interrupt can be viewed as the system's heartbeat. The time between interrupts is application specific and is generally between 1 and 200 ms. The clock tick interrupt allows a kernel to delay threads for an integral number of clock ticks and to provide timeouts when threads are waiting for events to occur. The faster the tick rate, the higher the overhead imposed on the system.

All kernels allow threads to be delayed for a certain number of clock ticks. The resolution of delayed threads is one clock tick; however, this does not mean that its accuracy is one clock tick.

Figure 10.13 through Figure 10.15 are timing diagrams showing a thread delaying itself for one clock tick. The shaded areas indicate the execution time for each operation being performed. Note that the time for each operation varies to reflect typical processing, which would include loops and conditional statements (i.e., `if`/`else`, `switch`, and `?:`). The processing time of the Tick ISR has been exaggerated to show that it too is subject to varying execution times.

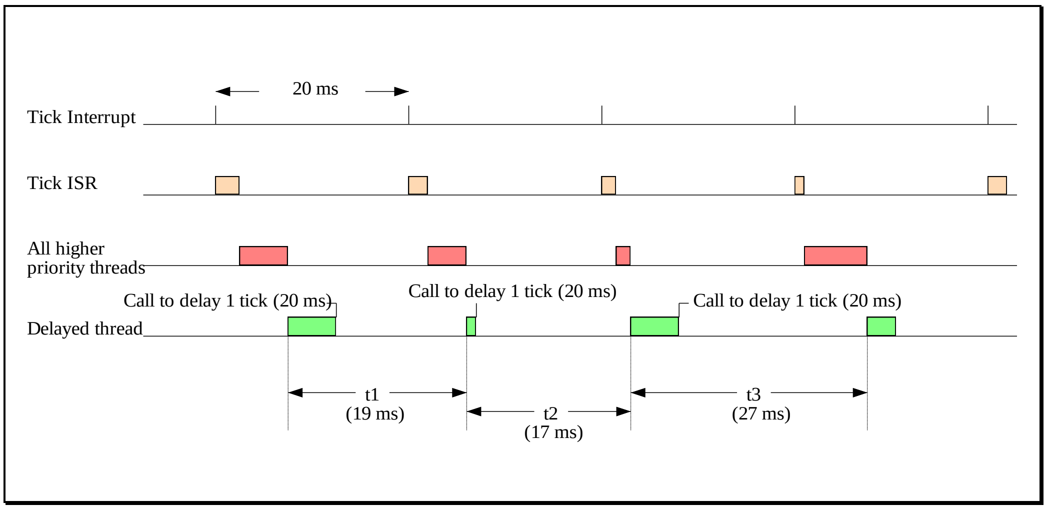
20 ms
Tick Interrupt
Tick ISR
All higher priority threads
Call to delay 1 tick (20 ms)
Call to delay 1 tick (20 ms)
Call to delay 1 tick (20 ms)
Delayed thread
t1 (19 ms)
t2 (17 ms)
t3 (27 ms)

Figure 10.13 – Delaying a thread for one tick (Case 1)

Case 1 (Figure 10.13) shows a situation where higher priority threads and ISRs execute prior to the thread, which needs to delay for one tick. The thread attempts to delay for 20ms but because of its priority, it actually executes at varying intervals. This causes the execution of the thread to jitter.

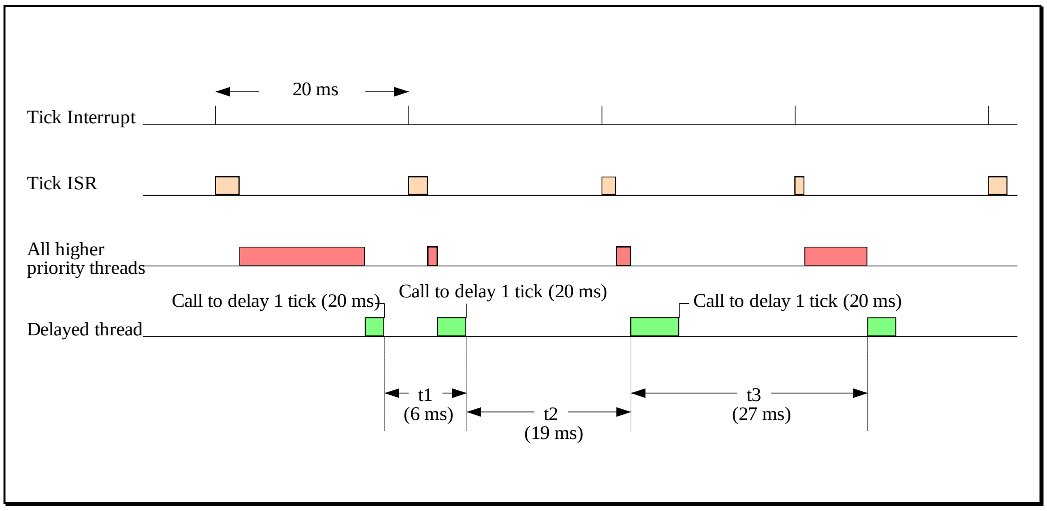
20 ms
Tick Interrupt
Tick ISR
All higher priority threads
Call to delay 1 tick (20 ms)
Call to delay 1 tick (20 ms)
Call to delay 1 tick (20 ms)
Delayed thread
t1 (6 ms)
t2 (19 ms)
t3 (27 ms)

Figure 10.14 – Delaying a thread for one tick (Case 2)

Case 2 (Figure 10.14) shows a situation where the execution times of all higher priority threads and ISRs are slightly less than one tick. If the thread delays itself just before a clock tick, the thread will execute again almost immediately! Because of this, if you need to delay a thread at least one clock tick, you must specify one extra tick. In other words, if you need to delay a thread for *at least* five ticks, you must specify six ticks!

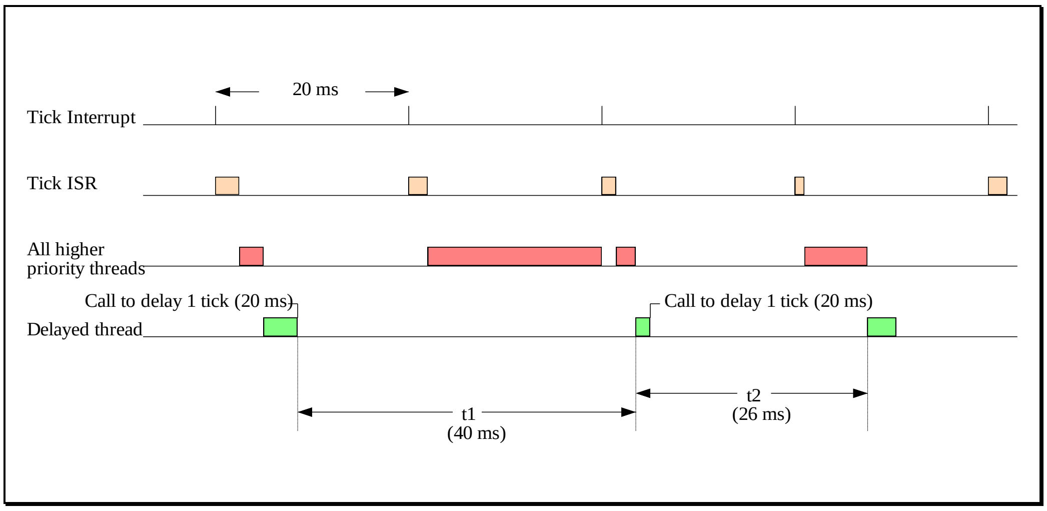
20 ms
Tick Interrupt
Tick ISR
All higher priority threads
Call to delay 1 tick (20 ms)
Call to delay 1 tick (20 ms)
Delayed thread
t1 (40 ms)
t2 (26 ms)

Figure 10.15 – Delaying a thread for one tick (Case 3)

Case 3 (Figure 10.15) shows a situation in which the execution times of all higher priority threads and ISRs extend beyond one clock tick. In this case, the thread that tries to delay for one tick actually executes two ticks later and misses its deadline. This might be acceptable in some applications, but in most cases it isn't.

These situations exist with all real-time kernels. They are related to CPU processing load and possibly incorrect system design. Here are some possible solutions to these problems:

- Increase the clock rate of your microprocessor.
- Increase the time between tick interrupts.
- Rearrange thread priorities.
- Avoid using floating-point maths (if you must, use single precision).
- Get a compiler that performs better code optimization.
- Write time-critical code in assembly language.
- If possible, upgrade to a faster microprocessor in the same family; that is, Cortex®-M0+ to Cortex®-M3, etc.

Regardless of what you do, jitter will always occur.

10.8 Memory Requirements

If you are designing a foreground / background system, the amount of memory required depends solely on your application code. With a multithreading kernel, things are quite different. To begin with, a kernel requires extra code space (Flash). The size of the kernel depends on many factors. Depending on the features provided by the kernel, you can expect anywhere from 1 to 100 KiB. A minimal kernel for a 32-bit CPU that provides only scheduling, context switching, semaphore management, delays, and timeouts should require about 1 to 3 KiB of code space.

Because each thread runs independently of the others, it must be provided with its own stack area (RAM). As a designer, you must determine the stack requirement of each thread as closely as possible (this is sometimes a difficult undertaking). The stack size must not only account for the thread requirements (local variables, function calls, etc.), it must also account for maximum interrupt nesting (saved registers, local storage in ISRs, etc.). Depending on the target processor and the kernel used, a separate stack can be used to handle all interrupt-level code. This is a desirable feature because the stack requirement for each thread can be substantially reduced. Another desirable feature is the ability to specify the stack size of each thread on an individual basis. Conversely, some kernels require that all thread stacks be the same size. All kernels require extra RAM to maintain internal variables, data structures, queues, etc. The total RAM required if the kernel does not support a separate interrupt stack is given by Eq. (10.6).

Total RAM requirements

= Application code requirements

+ Data space (i.e., RAM) needed by the kernel

+ SUM(thread stacks + MAX(ISR nesting)) (10.6)

Unless you have large amounts of RAM to work with, you need to be careful how you use the stack space. To reduce the amount of RAM needed in an application, you must be careful how you use each thread's stack for:

- large arrays and structures declared locally to functions and ISRs,
- function (i.e., subroutine) nesting,
- interrupt nesting,
- library functions stack usage, and
- function calls with many arguments.

To summarize, a multithreading system requires more code space (Flash) and data space (RAM) than a foreground / background system. The amount of extra Flash depends only on the size of the kernel, and the amount of RAM depends on the number of threads in your system.

10.9 Advantages and Disadvantages of Real-Time Operating Systems

An RTOS allows real-time applications to be designed and expanded easily; functions can be added without requiring major changes to the software. The use of an RTOS simplifies the design process by splitting the application code into separate threads. With a preemptive RTOS, all time-critical events are handled as quickly and as efficiently as possible. An RTOS allows you to make better use of your resources by providing you with valuable services, such as semaphores, mailboxes, queues, time delays, timeouts, etc.

You should consider using a real-time kernel if your application can afford the extra requirements: extra cost of the kernel, more ROM/RAM, and 2 to 4 percent additional CPU overhead.