Thread-Safe Code

A multitasking operating system permits a single process (program) to have multiple threads of execution. Each thread is also known as a *task*. A program might be using one thread to request data from an FTP download site, for example, while another thread responds to the user's mouse clicks and keystrokes. While this is certainly a good idea, it often becomes necessary to protect critical sections of code from being entered by two execution threads at the same time. A single bit can be used as a *semaphore*, controlling access to the critical section.

In the next example, bit 7 of a variable named **semaphore** is clear when it is safe to enter a critical code section. If bit 7 is set, the current thread jumps to the label named **Block** and waits for the bit to clear:

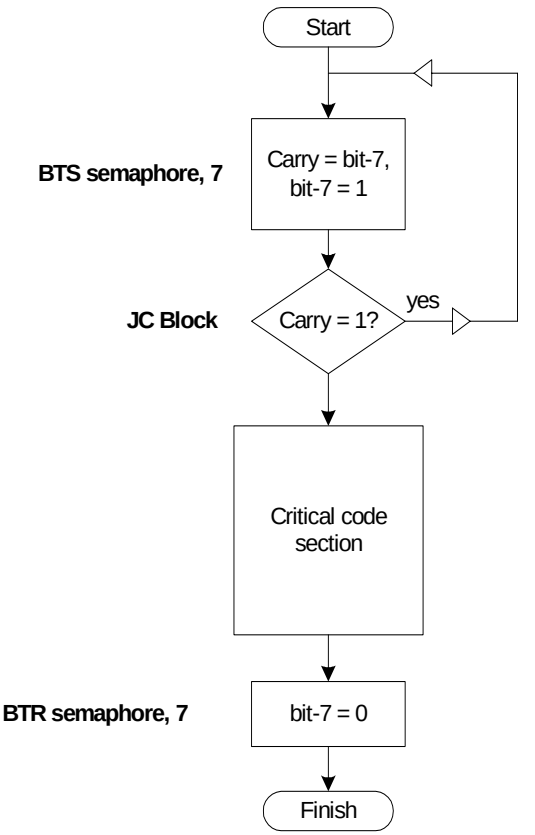
```
Block:
   bts semaphore,7              ; copy bit 7 to Carry flag
   jc Block                     ; carry flag set, so wait

; (bit 7 is now set, preventing other threads from entering)
;--------- Begin critical code section -------------------
   .
   .
   .
;---------- End critical code section --------------------
   btr semaphore,7              ; clear bit 7
```

It is important to note that each thread has its own copy of the CPU flags. The **semaphore** variable, on the other hand, is shared between multiple threads.

The nice thing about the BT, BTC, BTR, and BTS instructions is that they are atomic. It either executes completely or not at all, regardless of when the current task might be interrupted. A task is never interrupted during the execution of a single instruction. The following flowchart

describes the actions taken by BTS and JC in our example. To simplify the names, the flowchart uses the name **bit-7** to identify bit 7 of the semaphore variable:



Here is a possible scenario, using two execution threads named **Thread-1** and **Thread-2.** They both need to execute the instructions in the critical code section, but they must not be permitted to enter the section at the same time. We assume here that a preemptive multitasking operating system is used, causing each thread to automatically be interrupted by the OS kernel on a regular basis:

- Initially, bit-7 (in semaphore) equals 0.
- Thread-1 starts executing and executes the BTS instruction, which clears the Carry flag (because bit-7 was equal to 0) and at the same time, sets bit-7 to 1.
- The JC instruction does not jump, because the Carry flag is clear.
- Thread-1 enters the critical code section.
- Before Thread-1 has left the critical code section, it is interrupted by the task scheduler.

- Thread-2 starts, and executes the BTS instruction. Bit-7 equals 1, so BTS copies this value into the Carry flag. BTS also sets bit-7 to 1, the same value it already had.
- Thread-2 executes the JC instruction. Because the Carry flag is set, the JC transfers control to the label named **Block**. This is called a *wait loop* because Thread-2 repeatedly loops back to Block, each time checking bit-7.
- While Thread-2 is executing its wait loop, the task scheduler interrupts and returns control back to Thread-1.
- Eventually, Thread-1 leaves the critical code section, executes BTR and clears bit-7. When Thread-2 next executes BTS, the latter clears the Carry flag and sets bit-7. When the JC instruction executes, the jump is not taken and Thread-2 enters the critical code section. Later, when Thread-2 leaves the critical code section, it executes the BTR instruction and clears bit-7.

Your turn . . .

Suppose there were no BTS instruction and we were forced to use the following three instructions instead:

```
mov ax,semaphore          ; copy the semaphore
shr ax,8                  ; shift bit 7 into Carry flag
or semaphore,10000000b    ; set bit 7
```

What would happen if the task scheduler interrupted Thread-1 between the second and third instructions, and Thread-2 began to execute this same group of instructions?