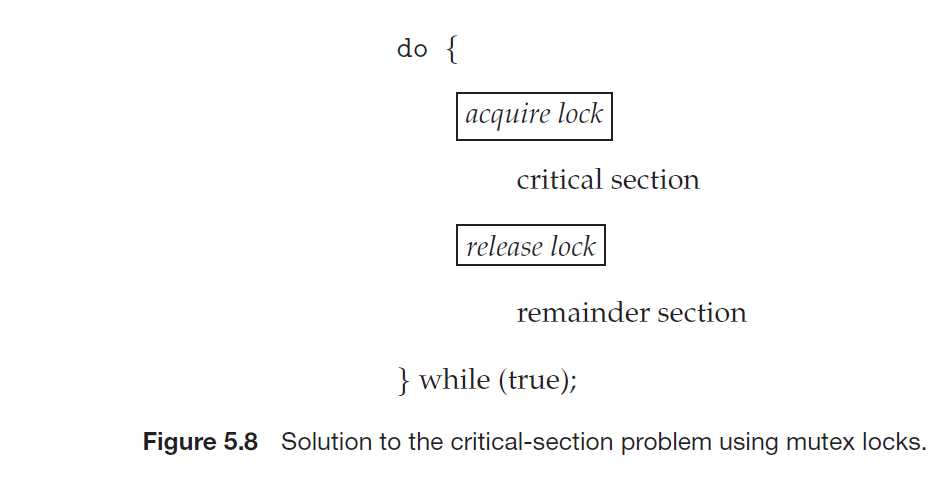
**MUTEX LOCKS**

The hardware-based solutions to the critical-section problem presented in are complicated as well as generally inaccessible to application programmers. Even software-based solution also restricted to only two processes. Instead, operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the mutex lock. In fact, the term mutex is short for mutual exclusion.

The mutex lock is used to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire() function acquires the lock, and the release() function releases the lock, as illustrated in below Figure.



A mutex lock has a boolean variable **available** whose value indicates if the lock is available or not. Initially the value of available is true. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.

The definition of acquire() is as follows:

acquire()

{

while (!available); /\* busy wait \*/

available = false;;

}

The definition of release() is as follows:

release()

{

available = true;

}

Calls to either acquire() or release() must be performed atomically.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a spinlock because the process “spins” while waiting for the lock to become available. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful. They are often employed on multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor.