7

RESOURCE ACCESS PROTOCOLS

7.1 INTRODUCTION

A *resource* is any software structure that can be used by a process to advance its execution. Typically, a resource can be a data structure, a set of variables, a main memory area, a file, or a set of registers of a peripheral device. A resource dedicated to a particular process is said to be *private*, whereas a resource that can be used by more tasks is called a *shared resource*. A shared resource protected against concurrent accesses is called an *exclusive resource*.

To ensure consistency of the data structures in exclusive resources, any concurrent operating system should use appropriate resource access protocols to guarantee a mutual exclusion among competing tasks. A piece of code executed under mutual exclusion constraints is called a *critical section*.

Any task that needs to enter a critical section must wait until no other task is holding the resource. A task waiting for an exclusive resource is said to be *blocked* on that resource, otherwise it proceeds by entering the critical section and holds the resource. When a task leaves a critical section, the resource associated with the critical section becomes *free*, and it can be allocated to another waiting task, if any.

Operating systems typically provide a general synchronization tool, called a *semaphore* [Dij68, BH73, PS85], that can be used by tasks to build critical sections. A semaphore is a kernel data structure that, apart from initialization, can be accessed only through two kernel primitives, usually called *wait* and *signal*. When using this tool, each exclusive resource R_i must be protected by

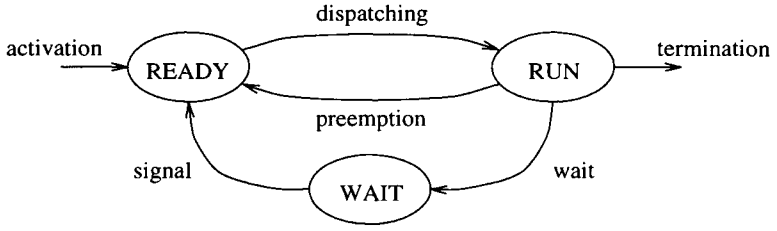

Figure 7.1 Waiting state caused by resource constraints.

a different semaphore S_i and each critical section operating on a resource R_i must begin with a $wait(S_i)$ primitive and end with a $signal(S_i)$ primitive.

All tasks blocked on the same resource are kept in a queue associated with the semaphore that protects the resource. When a running task executes a *wait* primitive on a locked semaphore, it enters a *waiting* state, until another task executes a *signal* primitive that unlocks the semaphore. When a task leaves the waiting state, it does not go in the running state, but in the ready state, so that the CPU can be assigned to the highest-priority task by the scheduling algorithm. The state transition diagram relative to the situation described above is shown in Figure 7.1.

In this chapter, we describe the main problems that may arise in a uniprocessor system when concurrent tasks use shared resources in exclusive mode, and we present some resource access protocols designed to avoid such problems and bound the maximum blocking time of each task. We then show how such blocking times can be used in the schedulability analysis to extend the guarantee formulae found for periodic task sets.

7.2 THE PRIORITY INVERSION PHENOMENON

Consider two tasks J_1 and J_2 that share an exclusive resource R_k (such as a list), on which two operations (such as *insert* and *remove*) are defined. To guarantee the mutual exclusion, both operations must be defined as critical sections. If a binary semaphore S_k is used for this purpose, then each critical section must begin with a *wait(S_k)* primitive and must end with a *signal(S_k)* primitive (see Figure 7.2).