Mutex Locks	• Previous solutions are complicated and generally inaccessible to application programmers • OS designers build software tools to solve critical section problem • Simplest is **mutex lock** • Protect a critical section by first **acquire()** a lock then **release()** the lock ○ Boolean variable indicating if lock is available or not • Calls to **acquire()** and **release()** must be atomic • But this solution requires **busy waiting**

Semaphore	• Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities
	• Semaphore ***S*** – integer variable
	• Can only be accessed via two indivisible (atomic) operations
	    ○ **Wait()** and **signal()** *originally called P() and V()
	• Definition of the **wait() operation**
	    ○ Wait(S)
	    ○ { while (s <=0) ;
	    ○ // busy wait
	    ○ S--;
	    ○ }
	• Definition of the **signal() operation**
	    ○ Signal(S) {
	    ○ S++;
	    ○ }

Deadlock and Starvation	• Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes • Starvation – indefinite blocking ○ A process may never be removed from the semaphore queue in which it is suspended • Priority inversion – scheduling problem when lower-priority process holds a lock needed by higher-priority process

s