SEMAPHORES

ü A more robust alternative to simple mutexes is to use ***semaphores***, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.

ü Note that not only must the variable-changing steps (S-- and S++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruptions before S gets decremented. It IS okay, however,forthebusylooptobe interruptedwhenthetesistrue,which prevents the system from hanging forever.

Wait:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

Signal:

```
signal(S) {
    S++;
}
```

1. Semaphore Usage

ü In practice, semaphores can take on one of two forms:

- **Binary semaphores** can take on one of two values, 0 or 1. They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that do not provide a separate mutex mechanism.. The use of mutexes for this purpose is shown in Figure below.

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

ü Semaphores can also be used to synchronize certain operations between processes. For example, suppose it is important that process P1 execute statement S1 before process P2 executes statement S2.

ü First we create a semaphore named synch that is shared by the two processes, and initialize it to zero. Then in process P1 we insert the code:

S1;

signal(synch);

and in process P2 we insert the code:

wait(synch); S2;

Because synch was initialized to 0, process P2 will block on the wait until after P1 executes the call to signal.

2. Semaphore Implementation

ü The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as a ***spinlock***, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches, and so it is sometimes used in multi-processing systems when the wait time is expected to be short - One thread spins on one processor while another completes their critical section on another processor.

ü An alternative approach is to block a process when it is forced to wait for an available semaphore, and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem.)

ü The new definition of a semaphore and the corresponding wait and signal operations are shown as follows:

Semaphore Structure:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Wait Operation:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

Signal Operation:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

3. Deadlocks and Starvation

One important problem that can arise when using semaphores to block processes waiting for a limited resource is the problem of deadlocks, which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other (blocked) processes, as illustrated in the following example.

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

ü Another problem to consider is that of starvation, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait( ) call, or selecting one to be removed from the queue in the signal( ) call. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.