Chapter

17

Coordinating and Sequencing events

When you program an application where many things happen at the same time in different threads or processes, you have to concern yourself with access to shared resources. You have to program some amount of event coordination.

Regular semaphores

Semaphores allow you to wait for a resource to become available or to change state before you continue processing. One example might be an object that needs a certain event to happen, such as a door to be opened or a button to be pressed. One solution would be to wait on a semaphore that gets signaled by a process monitoring the door or button. Another example might be a tape drive that is shared among several processes that want to mount and dismount tapes. We could use a semaphore to coordinate and serialize the events.

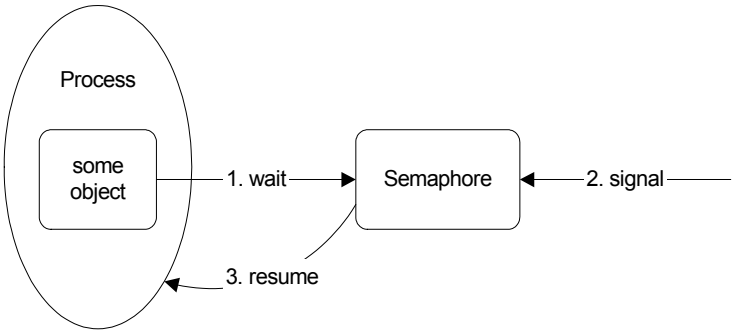

Figure 17-1.
The Semaphore mechanism.

To use a semaphore, you first create an instance of *Semaphore* by sending it the `new` message. A process waits on the semaphore by sending it the `wait` message. This suspends the process until another process signals the

semaphore by sending it the `signal` message. The semaphore will then resume the suspended process that is waiting on it. Figure 17-1 illustrates this. Here's a simple example of waiting on a Semaphore.

```
semaphore := Semaphore new.

[Transcript cr; show: 'About to wait'.
semaphore wait.
Transcript cr; show: 'Semaphore signaled'] fork.

[(Delay forSeconds: 5) wait.
semaphore signal] fork.
```

In our tape example, the tape drive object has a semaphore. When a tape wants to be mounted in a drive, the tape waits on the drive's semaphore. When the drive becomes available, it signals its semaphore, which allows the tape to continue mounting itself. Semaphores have the additional virtue of queuing `wait` requests in the order they were received so that we could have many tapes waiting for the drive. As the drive becomes available and signals its semaphore, the first tape in the queue will mount itself, while the other tape processes remain suspended until the drive becomes available and it is their turn.

```
Tape>>mountYourselfOn: aDrive
   self doSomeMountStuff.
   aDrive reserve.
   self doMoreMountStuff

Tape>>dismountYourselfFrom: aDrive
   self doSomeDismountStuff.
   aDrive release.
   self doSomeMoreDismountStuff

Drive>>reserve
   semaphore wait

Drive>>release
   semaphore signal
```

Besides queueing up wait requests by sending multiple `wait` messages to a semaphore, you can also prime a semaphore by sending multiple `signal` messages. This could be useful if you are using a semaphore to control access to a resource that allows several accesses at once. By sending several `signal` messages to the semaphore when it is created, you allow the first few objects access to the resource. As they release the resource, they signal the semaphore, allowing the next object to get access.

Mutual exclusion semaphores

Sometimes you want to serialize access to shared data. A good example is tracking the state of a resource that is being modified by different processes. The state data needs to be read and written, but we need to make sure that we don't have different processes trying to access it at the same time. (In fact, since Smalltalk is only partially preemptive, this will probably not be a problem, but it's better to be defensive in our programming.) Another

example is reserving a resource, where we want to first check the state of the resource and then reserve it if it is available. We want the checking and reserving to be an atomic operation, without the possibility of interruption.

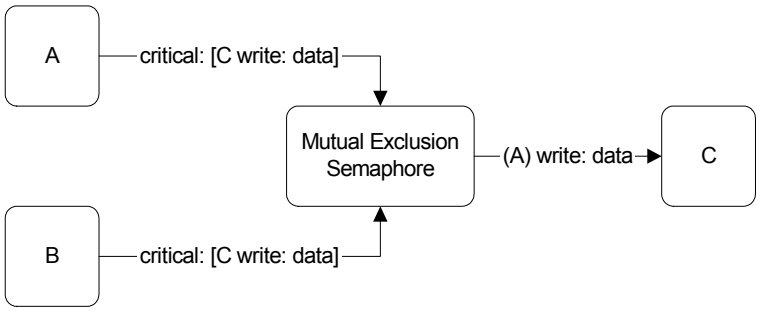

Figure 17-2.
Mutual Exclusion Semaphore.

We use a special type of semaphore called a *mutual exclusion semaphore*, which runs a block of code when sent the `critical:` message. A mutual exclusion semaphore will only run one block of code at a time. This is illustrated in Figure 17-2. As long as all our data access routines are run by the mutual exclusion semaphore, we are guaranteed that the data will be only be accessed by one routine at a time. Here's an example of how we might use a mutual exclusion semaphore to protect a shared resource.

```
MyClass class>>initialize
   AccessProtect := Semaphore forMutualExclusion.

MyClass>>stateData
   ^AccessProtect critical: [code to get the value]

MyClass>>stateData: aValue
   AccessProtect critical: [code to set the value]
```

We ask the mutual exclusion semaphore to run the code critically. The semaphore will run only one block of code at a time, queuing up the other blocks in the order it was asked to run them.

A mutual exclusion semaphore works by immediately signaling itself when it is created. This allows the first block of code to run without waiting (it tries to wait, but since the semaphore has already been signalled, no waiting is needed). Once the block has been executed in the `critical:` method, the method signals the semaphore again. If there is another block of code waiting, it will now run. If there is no code waiting to be run, the semaphore is primed so that the next block to come along will run without waiting. (The code block is run by sending `valueNowOrOnUnwindDo: [self signal]` to the BlockClosure. This causes the code block to be run, and then `self signal` to be executed no matter whether the block runs to completion, returns, or is interrupted by an exception.)

Here's an example of a mutual exclusion semaphore that you can run in a Workspace. It protects access to the Transcript, which in fact is a reasonable thing to do since the Transcript is not threadsafe.

```
accessProtect := Semaphore forMutualExclusion.

[accessProtect critical:
   [Transcript cr; show: 'Process1 ', Time now printString.
```

```
    (Delay forSeconds: 5) wait]] fork.

[accessProtect critical:
    [Transcript cr; show: 'Process2 ', Time now printString]] fork.
```

An important point to note is that to protect access to shared data, the processes must cooperate. If one process asks a mutual exclusion semaphore to control its access to the data, but another process goes directly to the data, that data is no longer threadsafe. Ideally, your shared data will only be accessible through access routines which all cooperate and use a mutual exclusion semaphore.

RecursionLock

Sometimes we may find ourselves in a situation where a mutual exclusion semaphore is executing code in a critical block, and the code needs access to another resource that is protected by the same semaphore.

If we use a mutual exclusion semaphore our code will hang because the semaphore is already running code. To solve the problem we use a *RecursionLock* rather than a mutual exclusion semaphore. A RecursionLock knows which process is running, and allows additional access by that process to resources it is protecting.

Summary

To summarize, use a regular semaphore (`Semaphore new`) to coordinate events such as waiting for a button to be pressed. Use a mutual exclusion semaphore (`Semaphore forMutualExclusion`) to serialize access to resources such as data stores. Use a RecursionLock (`RecursionLock new`) instead of a mutual exclusion semaphore when the code that accesses the resource may need to access another resource protected by the same semaphore.