Chapter

16

Processes

Smalltalk allows you to create separate processes so that your application can do several things in parallel. For example, you might have a process for handling communication with sockets, another process for handling communication with an external device, and separate processes for each request that comes over a socket. Note however, that processes are *not* operating system processes — they are internal to the Smalltalk image. There is a process scheduler which schedules the Smalltalk processes, allocating time based on their priority and when they last ran. The scheduler is stored in the global variable *Processor*, and is the single instance of the class *ProcessorScheduler*. The most common way to fork a new process is to send a block of code the `fork` message. For example,

```
[Transcript cr; show: 'Hello World'] fork.
```

When this process runs, it prints 'Hello World' to the Transcript. Once the right hand bracket of the block is reached, the process terminates. Let's took at the interactions of two processes.

```
[10 timesRepeat: [Transcript show: '1']] fork.
[10 timesRepeat: [Transcript show: '2']] fork.

1111111112222222222
```

You will see 10 1's printed on the Transcript then 10 2's. This is because processes are *partially preemptive*. A running process will not be interrupted by another process of equal or lesser priority. A process will keep running until it does a time consuming operation such as waiting on a socket read or on an instance of *Delay*, or the process explicitly gives up control, or a higher priority process is ready to run. If we add a delay to both loops we see that the numbers alternate.

```
[10 timesRepeat: [Transcript show: '1'. (Delay forMilliseconds: 1)
wait]] fork.
[10 timesRepeat: [Transcript show: '2'. (Delay forMilliseconds: 1)
wait]] fork.

12121212121212121212
```

Alternatively, we can give up control by sending a `yield` message to *Processor*. The scheduler will then figure out the next process to run.

```
[10 timesRepeat: [Transcript show: '1'. Processor yield]] fork.
[10 timesRepeat: [Transcript show: '2'. Processor yield]] fork.

12121212121212121212
```

Priority

When the process scheduler is looking for the next process to run, it looks first at the priority of the processes that are waiting to run. When you create a new process using `fork`, it inherites the priority of the process that forked it. However, you can specify the priority of the processes you create by sending the `forkAt:` message to the BlockClosure, with the priority as the parameter.

For example, if your application communicates with other applications via sockets, you might have one process sitting in an input loop, another process sitting in an output loop, and other processes doing the application work. You might run the application processes at one priority, the input process at a higher priority, and the output process at a still higher priority.

Let's look at how a higher priority process can preempt a lower priority process. We'll use absolute numbers rather than names for our priorities to make the examples clearer. VisualWorks supports priorities from 1 to 100, where the default priority is 50.

In the first example, we fork a process with priority 50. Since the user interface is also running at priority 50, the forked process is not interrupted, and therefore completes. The user interface then forks the second process, which also runs to completion. (The delay exists to make sure that the first process gets a chance to run. If we didn't have the delay, the user interface process would keep running and would fork the second process. At that point it would have nothing to do so the scheduler would look for another process to run, and would run the second process since it has higher priority.)

```
[10 timesRepeat: [Transcript show: '1']] forkAt: 50.
(Delay forMilliseconds: 1) wait.
[10 timesRepeat: [Transcript show: '2']] forkAt: 51.

11111111112222222222
```

In the second example, the first process is forked with a priority of 49, which is lower than the priority of the user interface process. The forked process just gets started and is then interrupted by the user interface process, which forks the second process. The second process is running at a higher priority than the first process, so it runs to completion. Then the first process gets the scheduler and completes.

```
[10 timesRepeat: [Transcript show: '1']] forkAt: 49.
(Delay forMilliseconds: 1) wait.
[10 timesRepeat: [Transcript show: '2']] forkAt: 51.

11222222222111111111
```

In the third example, the first process gives up control after printing the first number. The user interface then forks the second process, which gets to run since it has a higher priority. Even though the second process gives

up control after printing its first number, it is immediately run again because it is the highest priority process with something to do.

```
[10 timesRepeat: [Transcript show: '1'. Processor yield]] forkAt: 50.
(Delay forMilliseconds: 1) wait.
[10 timesRepeat: [Transcript show: '2'. Processor yield]] forkAt: 51.

12222222222111111111
```

Smalltalk processes are *partially preemptive*. They can be interrupted by higher priority processes, but will *not* be interrupted by lower or equal priority processes. This is different to a multi-tasking system such as Unix, which gives timeslices to the different processes, interrupting them when they have consumed their timeslice. Since a Smalltalk process will not be interrupted by another process of equal or lower priority, it is easy to get into loops that you can't break out of. We'll look more at this in the section Working with Processes later in this chapter.

We used numbers in these examples, but a much better policy is to use the names defined in the `priority names` protocol of ProcessorScheduler, sending these messages to *Processor*. For example, at the time of writing, VisualWorks defines the following priorities.

```
systemRockBottomPriority          1
systemBackgroundPriority         10
userBackgroundPriority           30
userSchedulingPriority           50
userInterruptPriority            70
lowIOPriority                    90
highIOPriority                   98
timingPriority                  100
```

Using names allows the values to be changed in future releases of the image without affecting your application. In fact, I'd go further and define my own names then fork the processes using my names.

```
MyGlobals class>>appProcessPriority
   ^Processor userSchedulingPriority

[some code] forkAt: MyGlobals appProcessPriority
```

Unless you have good reason, application process priorities should be kept in the range from userBackgroundPriority to userInterruptPriority.

Communicating between Processes

In a production application, you will often find two types of process. One type of process is started, runs to completion, then automatically terminates. The other type of process stays in a loop waiting for an object to appear, then sends a message to the object. This requires an ability to communicate between processes, and Smalltalk provides such a mechanism, the *SharedQueue*.

The SharedQueue

To set up communication between processes, you create an instance of SharedQueue and tell both processes about it. One process will put objects on the shared queue using `nextPut:` and the other process will send

`next` to get an object from the queue. When a process sends the `next` message, it blocks until there is something on the queue. (It blocks by waiting on a semaphore, which is set when an object is put on the SharedQueue. We'll talk more about semaphores in Chapter 17, Coordinating and Sequencing Events.) Figure 16-1 illustrates two processes communicating via a SharedQueue.

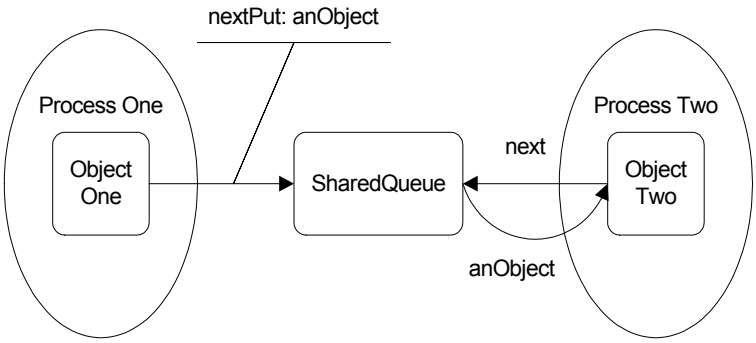

Figure 16-1.
Using a SharedQueue.

In the following example, the second process prints a number then puts it on a shared queue, while the first process reads the queue then prints the number. If you try this example, the Transcript will show: W1 R1 W2 R2 W3 R3 W4 R4 W5 R5.

```
sharedQueue := SharedQueue new.
readProcess := [[Transcript show: ' R', sharedQueue next printString]
repeat] fork.
[1 to: 5 do: [:index |
   Transcript show: ' W', index printString.
   sharedQueue nextPut: index.
   Processor yield]] fork.
(Delay forSeconds: 5) wait.
readProcess terminate.
```

Notice that the read process will stay in a loop forever unless we terminate it. This we do with the `terminate` message (there is more on terminating processes in the next section). If you try this example again after removing the `Processor yield` in the second process, you will see: W1 W2 W3 W4 W5 R1 R2 R3 R4 R5. This is because Smalltalk is only partially preemptive and the first process doesn't get a chance to run until the second process is completed. I generally add `Processor yield` after putting an object on the SharedQueue, which ensures that the process waiting on the queue has a chance to run. For example,

```
sharedQueue nextPut: anObject.
Processor yield.
```

If the reading process doesn’t want to block it can send `isEmpty` to find out if there is anything on the SharedQueue, or `peek` to get the next object from the SharedQueue without removing it.

The PrioritySharedQueue

SharedQueues are FIFO queues — objects are read from the queue in the order they were put on the queue. However, sometimes a simple FIFO queue is not sufficient because you might need a prioritization scheme. SharedQueues don't provide a mechanism for prioritizing objects on the queue, but you can subclass off

SharedQueue to create a PrioritySharedQueue. Here is a simple example of a PrioritySharedQueue, with an instance variable, *defaultPriority*. You can find the code for this example in the file `psq.st`.

```
PrioritySharedQueue>>defaultPriority: aPriority
   defaultPriority := aPriority

PrioritySharedQueue>>defaultPriority
   ^defaultPriority isNil
      ifTrue: [defaultPriority := 0]
      ifFalse: [defaultPriority]
```

A simple implementation puts all objects on the shared queue in an *Association* of the priority and the object. You will need to override `next`, `peek`, and `nextPut:` to handle the Association. Additionally, `nextPut:` will simply invoke `nextPut:priority:` as follows.

```
PrioritySharedQueue>>nextPut: anObject
   ^self nextPut: anObject priority: self defaultPriority
```

The heart of the PrioritySharedQueue is in the `nextPut:priority:` method, which places objects in the queue according to their priority first, and their chronology second. Higher numbers indicate higher priority.

```
PrioritySharedQueue>>nextPut: anObject priority: aPriority
   [accessProtect critical:
      [ |foundElement |
      foundElement := contents
            reverseDetect: [:element | aPriority <= element key]
            ifNone: [nil].
      foundElement isNil
         ifTrue: [contents addFirst: (aPriority -> anObject)]
         ifFalse: [contents add: (aPriority -> anObject) after:
foundElement]]
   ] valueUninterruptably.
   readSynch signal.
   ^anObject
```

A general SharedQueue reading mechanism

A general mechanism for taking objects off a SharedQueue and processing the objects is to sit in a loop like the following. This is very generic code and can easily be inherited. It has two specific virtues. First, it makes no assumptions about where the object is coming from and how it will be processed, so `myProcessObject:` and `myGetObject` can be overridden by subclasses to do things differently. Second, it allows you to put breakpoints in these two methods when you are debugging. Since `myDoLoop` is in a repeat loop, you can't add a breakpoint to this method while it is executing and have the breakpoint take effect.

```
MyClass>>myDoLoop
   [self myProcessObject: self myGetObject.
   Processor yield] repeat

MyClass>>myGetObject
   ^self mySharedQueue next.
```

We can extend this further for situations where we process the object in its own forked process. There are operations where it may be necessary to cancel the processing, such as time consuming operations or operations where a needed resource does not become available. If we need to cancel the processing, we can set an instance

variable in the object (protected by a mutual exclusion semaphore). At appropriate times during its processing, the object can look to see if it has been cancelled and take the appropriate action. In the code below, we store the object, then after the the object has executed the `processYourself` method, we remove the object from our collection. The `valueNowOrOnUnwindDo:` message ensures that the object is removed whether the `processYourself` method completes or is interrupted by a raised exception.

```
MyClass>>myProcessObject: anObject
   self myAdd: anObject.
   [[anObject processYourself]
      valueNowOrOnUnwindDo:
         [self myRemove: anObject]] fork
```

Terminating processes

Processes terminate when they reach the right square bracket of the block that was forked. However, some processes are set up to stay in a loop, never reaching the right square bracket. We therefore need another way to terminate processes, which we get by sending the `terminate` message to the process. For example,

```
process := [[Transcript cr; show: 'Hi there'. (Delay forSeconds: 1)
wait] repeat] fork.
(Delay forSeconds: 5) wait.
process terminate.
Transcript cr; show: 'All Done'.
```

You can send `terminate` to an already terminated process with no ill-effects so `terminate` is a benign message. If you are concerned about terminating a process that is in the middle of doing something, you may want a way to shut down the process gracefully, allowing it to complete what it is working on. One option is to allow the process to terminate itself by sending `Processor terminateActive` or `Processor activeProcess terminate`. If the process reads from a SharedQueue, you can put a special termination object on the SharedQueue. If the SharedQueue reader does something like `sharedQueue next processYourself`, this special object would have the following code.

```
TerminationObject>>processYourself
   Processor terminateActive
```

Making shared data threadsafe

Objects running in different processes often require access to shared data. To make sure that only one process is reading or modifying the data, you need a mechanism to *serialize* access to the data. Smalltalk provides the *mutual exclusion semaphore* for just this situation. To create a mutual exclusion semaphore, you send the `forMutualExclusion` message to *Semaphore*. To serialize access to shared data, you ask the semaphore to run the access code. For example,

```
mutexSemaphore := Semaphore forMutualExclusion.
mutexSemaphore cricital: [code to access shared data]
```

We talk more about mutual exclusion semaphores in Chapter 17, Coordinating and Sequencing Events.

Controlling the running of the process

In most applications you won't suspend and resume processes, but the ability is there should you need it. When you want to suspend a process, send it the `suspend` message. To resume the process, send it the `resume` message. You can start a process in suspended state by sending `newProcess` to a BlockClosure, and in fact, `fork` is implemented by sending `newRequest` followed by `resume`. Here's an example of creating a process that does not immediately run.

```
process := [Transcript cr; show: 'Done'] newProcess.
(Delay forSeconds: 3) wait.
process resume
```

If you decide to control the running of processes using `suspend` and `resume`, be very careful about when you suspend the process. It may be in the middle of doing something that should be completed before the process is suspended. One option is to have the process suspend itself when it is safe to do so. If you try the following code, you'll see that the process prints out the time five times, suspends itself, then is resumed five seconds later.

```
process :=
   [1 to: 10 do: [:index |
      Transcript cr; show: Time now printString.
      index = 5 ifTrue: [Processor activeProcess suspend]]] fork.
(Delay forSeconds: 5) wait.
process resume.
```

Interrupting Processes

Smalltalk allows you to interrupt a process to have it do something else. You may never need to do this, but we'll mention the mechanism briefly. To interrupt another process and have it run some code, send `interruptWith: aBlockOfCode` to the process. The process saves its context, executes the passed-in block, restores its context, then resumes its business.

There is a way to prevent interruptions, in case the process is doing something it really doesn't want interrupted. You can protect the uninterruptable code by putting it in a block and sending the block the message `valueUninterruptably`. SharedQueues make use of this capability. All access to a SharedQueue is protected by a mutual exclusion semaphore. This access is wrapped in another block which is executed by sending `valueUninterruptably`. For example, `SharedQueue>>size` is implemented as `^[accessProtect critical: [contents size]] valueUninterruptably`. Will you ever use the interrupt capability? Probably not, but it's worth knowing about.

Working with Processes

When you first work with processes, it's easy to write processes that stay in a tight loop so that you can't do anything. You can't even interrupt them with `ctrl-C` if they are running at `userSchedulingPriority` or above — ie, at or above the priority of the user interface process. For this reason it's always a good idea to file out (or save in some way) your changes before trying out new process code. It's also a good idea to run processes at a priority lower than `userSchedulingPriority`.

If you do get locked up, you can kill the virtual machine process or you can use the Emergency Evaluator. Unless you have remapped the interrupt key, pressing `Shift-Ctrl-C` will bring up the Emergency Evaluator.

From here you can type `ObjectMemory quit` followed by pressing the Escape key, which will cause Smalltalk to quit.

Another thing that can happen when working with forked processes is that you can lose them. Suppose you have a process running in a loop, possibly even yielding control. If you don't have a way to reference the process you can never terminate it. Here are some mechanisms for tracking processes.

Tracking processes with a Global variable

One way to track your processes is to keep each process in a global variable. Let's say you choose to keep track of global variables in class *MyGlobals*. If you have a class variable called *Processes* that you initialize to an OrderedCollection (using class initialization or lazy initialization), you can have two methods:

```
MyGlobals class>>addProcess: aProcess
   ^Processes add: aProcess

MyGlobals class>>removeProcess: aProcess
   ^Processes remove: aProcess ifAbsent: [nil]
```

At any time you can inspect MyGlobals to see a collection of all the processes you have forked. You can terminate them individually or create a method to terminate all of them. To better know what process you are looking at, you could add an Association of the process and a name.

```
MyGlobals class>>addProcess: aProcess name: aName
   ^Processes add: aProcess -> aName

MyGlobals class>>removeProcess: aProcess
^Processes removeAllSuchThat: [:each | each key = aProcess]
```

This scheme assumes that the forked processes you add can also be removed. Many processes are simply short-lived process that terminate naturally and there will be no easy way to remove them from the collection (unless they remove themselves before expiring). This scheme works best with long-lived processes that the application can both start and terminate.

Tracking processes by subclassing

Another approach is to create your own process class, say *MyProcess*, as a subclass of Process (you can find the code in the file `process.st`). We'll give it an instance variable to store the process name.

```
Process subclass: #MyProcess
   instanceVariableNames: 'processName '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'MyStuff'
```

Add `processName` and `processName:` accessors, and write a `printOn:` method.

```
MyProcess>>printOn: aStream
   super printOn: aStream.
   aStream nextPutAll: ' ('.
   aStream nextPutAll: self processName.
   aStream nextPut: $)
```

Without going into all the details, this involves writing three new methods for *BlockClosure*. The first method is `newProcessWithName:`, which creates an instance of MyProcess rather than Process, and sets the process name. The other two methods are `forkWithName:` and `forkAt:withName:` which send `newProcessWithName:` rather than `newProcess`. You will then need to add an `allInstancesAndSubInstances` method to *Behavior* (we show this method in Chapter 29, Meta-Programming). Once you have this, the final change is to modify `Process>>anyProcessesAbove:` to send `allInstancesAndSubInstances` rather than `allInstances`. You can now fork all your processes by doing:

```
[some code] forkWithName: 'some name'.
```

If you then inspect `MyProcess allInstances`, you can take a look at all your processes that are running (or which have terminated but not been garbage collected). To terminate a lost process, you can open an inspector on the appropriate element of the array then evaluate `self terminate` in the inspector window.

To handle the problem of a process locking up the image, you can write a class side method to terminate all instances of MyProcess. From the Emergency Evaluator (`Shift-Ctrl-C`), you can now type `MyProcess terminateAll` followed by Esc, and you should get back your cursor.

```
MyProcess class>>terminateAll
   self allInstances do:
      [:each |
         Transcript cr; show: each printString.
         each terminate]
```

It may be easier to modify Process rather than subclass off it. If you choose this approach, you would add a *processName* instance variable. You will still need to add fork-type methods to BlockClosure to set the process name before sending resume to the process. Finally, instead of `MyProcess class>>terminateAll`, you might write something like the following.

```
Process class>>terminateNamed
   self allInstances do:
      [:each | each processName notNil ifTrue:
         [Transcript cr; show: each printString.
         each terminate]]
```