Processes and Concurrency

This module explores the features available in Smalltalk for the expression of concurrency. It introduces classes Process and ProcessorScheduler. Classes to support various synchronization operations, including Semaphore, SharedQueue and critical sections are explored though the use of examples and exercises. Class Delay is also considered. This module concludes by describing how instances of class Promise may be used to provide background tasks.

1. Introduction

VisualWorks supports *multiple independent processes*. These are *lightweight* processes, as they share a common address space (object memory). Each instance of class Process represents a sequence of actions which can be performed by the virtual machine, concurrently with other processes. However, the current implementations do not support genuine parallelism, but timeslice (at the bytecode level — see the description of bytecodes later).

We already know that blocks are used to implement a wide range of control constructs. Blocks are also used as the basis for creating Processes in Smalltalk. The simplest way to create a Process is to send a block the message fork. For example, selecting and executing the expression:

```
[Transcript cr; show: 100 factorial printString] fork
```

creates an instance of class Process containing a block which will display the factorial of 100 in the Transcript. The new Process is added to the list of scheduled Processes. This Process is runnable (i.e. scheduled for execution) immediately, and will start executing as soon as the current Process (i.e., the one controlling the window manager) releases control of the processor.

We can create a new instance of class Process which is not scheduled by sending the newProcess message to a block:

```
| aProcess |
aProcess := [Transcript cr;
                        show: 100 factorial printString] newProcess
```

The Process is not actually *runnable* until it receives the resume message:

```
aProcess resume
```

A Process can be temporarily stopped using the suspend message. A suspended Process can be restarted later using the resume message. The terminate message is used when a Process is no longer required; once a Process has received the terminate message, it cannot be restarted.

It is also possible to create and evaluate a Process containing a block with any number of arguments.

```
| aProcess |
aProcess :=
        [ :first :second |
        Transcript cr; show: (first raisedTo: second) printString]
                                        newProcessWith: #(2 20).
aProcess resume.
```

The example above creates a Process which, when it runs, will display 1048576 (2^{20}) in the Transcript.

Thus, a Process may be in one of five states:

1. suspended,
2. waiting,
3. runnable,
4. running, or
5. terminated.

States 1 & 2 are similar: the difference between them is that a suspended Process may be restarted with the resume message, whereas a waiting Process cannot be restarted until it receives permission from a semaphore (see later).

The five messages that are of interest in forcing a Process to make a transition from one state to another are suspend, terminate, and resume, sent to a Process, and wait and signal, sent to a Semaphore (the former acts on the active Process). The state transition diagram (Fig.1) shows how these affect a Process.

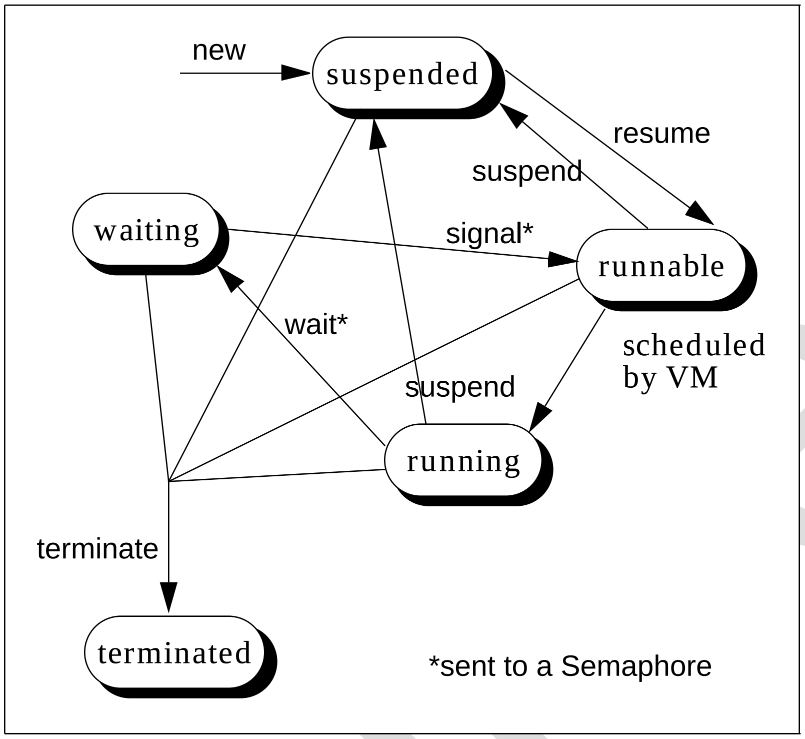


Figure 1: Process States

2. Priorities

Smalltalk supports *prioritized* Processes, so that we can create Processes of high priority which run before other Processes of lower priority. Eight priorities have names assigned to them, described in Table 1 below.

Priority	Name	Purpose
100	timingPriority	Used by Processes that are dependent on real time. For example, Delays (see later).
98	highIOPriority	Used by time–critical I/O Processes, such as handling input from a network.
90	lowIOPriority	Used by most I/O Processes, such as handling input from the user (keyboard, pointing device, etc.).

Table 1: Smalltalk Priorities

70	userInterruptPriority	Used by user Processes desiring immediate service. Processes run at this level will pre–empt the window scheduler and should, therefore, not consume the Processor forever.
50	userSchedulingPriority	Used by Processes governing normal user interaction. Also the priority at which the window scheduler runs.
30	userBackgroundPriority	Used by user background Processes.
10	systemBackgroundPriority	Used by system background Processes. Examples are an optimizing compiler or status checker.
1	systemRockBottomPriority	The lowest possible priority.

Table 1: Smalltalk Priorities (Continued)

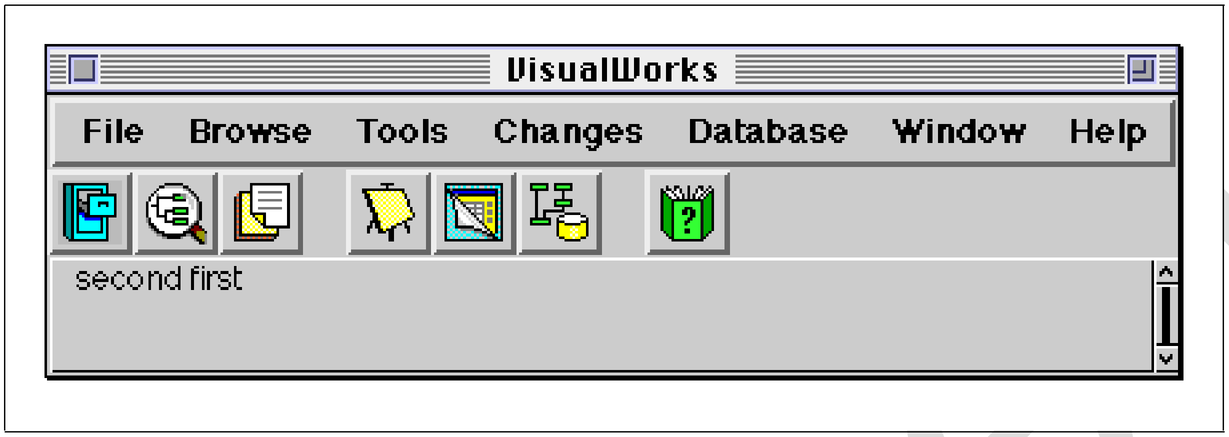
We can create a *runnable* Process with specified priority using the forkAt: message. The argument is an integer, but should be obtained by sending a message to Processor (described later), for example:

```
[Transcript cr; show: 100 factorial printString]
                    forkAt: Processor userBackgroundPriority.
```

Alternatively, we can use the priority: message to change the priority of an existing Process. For example, in the code below we create two Processes: *process1* and *process2*, which are given priorities 10 and 98 respectively. Note that *process1* is resumed before *process2*. The result on the Transcript is shown in Fig.2.

```
| process1 process2 |
Transcript clear.
process1 := [Transcript show: ' first'] newProcess.
process1 priority: Processor systemBackgroundPriority.
process2 := [Transcript show: ' second'] newProcess.
process2 priority: Processor highIOPriority.
process1 resume.
process2 resume.
```

The *default Process priority* (and the priority at which expressions are evaluated using the user interface) is userSchedulingPriority (50). The scheduling algorithm used is described in detail later.



second first

Figure 2: The result of evaluating the message expressions above.

Ex 1. Try some examples of creating simple Processes using the fork and newProcess messages sent to blocks. You may like to display trace messages in the System Transcript.

Ex 2. Browse class Process. Try using the resume, suspend and terminate messages on a Process created using the newProcess (or newProcessWith:) message sent to a block.

Ex 3. Browse class BlockClosure to see how the fork and newProcess methods are implemented.

Ex 4. Modify the priorities in the code above so that the priority of each process is

a. less than 50

b. greater than 50

3. Scheduling Processes

Class ProcessorScheduler manages the runnable Processes. As the virtual machine has only one processor, its single instance is represented by a single global variable Processor. We have already come across the use of the variable as the receiver of messages to return an appropriate priority.

The active Process (the one actually running) can be identified by the expression

```
Processor activeProcess
```

This can be controlled by suspend or terminate messages.

The processor is given to the Process having the highest priority. When the highest priority is held by multiple Processes, the active Process can give up the processor, moving itself to the back of the queue of quiescent Processes at that priority, with the expression Processor yield. Otherwise it will run until it is suspended or terminated before giving up the processor, or preempted by a higher priority Process. However, a Process that is “pushed to the back of the queue” will regain control before a Process of a lower priority (see Fig.3).

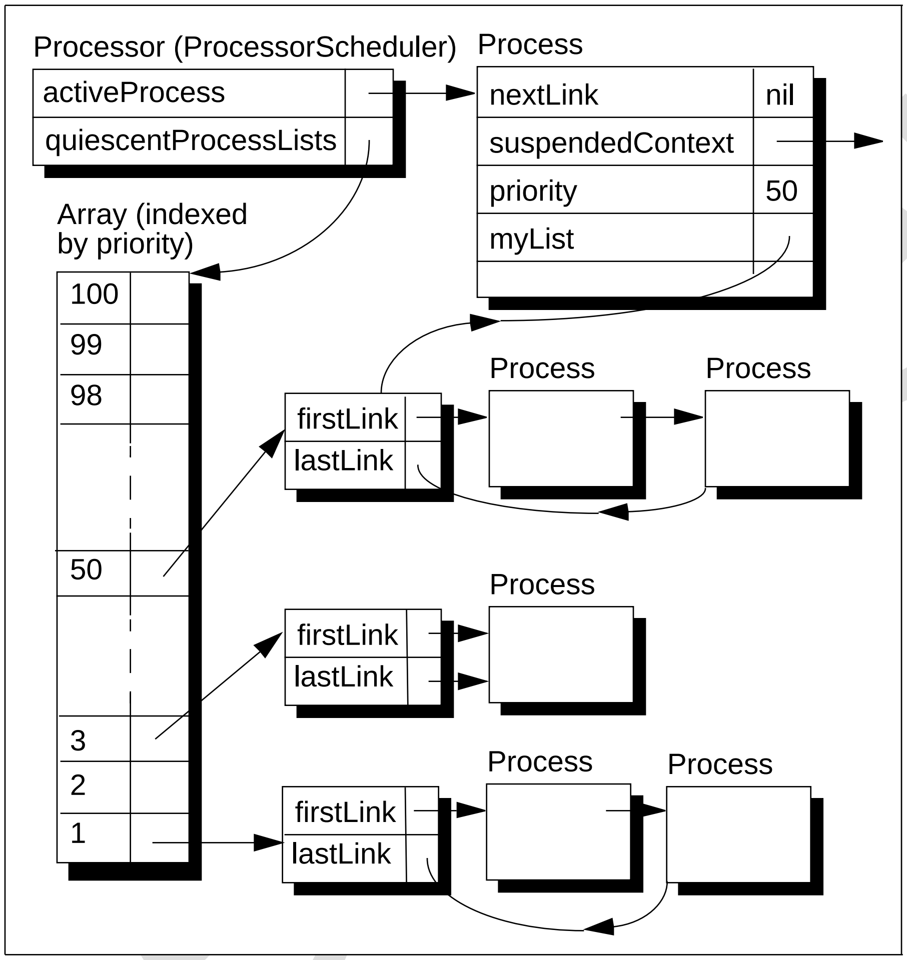


Figure 3: Process Queues

For example, in the code below, the first forked Process contains a Processor yield expression which causes it to be pushed to the back of the queue, thus allowing the second forked Process to run.

```
| process1 process2 |
Transcript clear.
process1 := [Transcript cr; show: ' first:1'.
                              Processor yield.
                              Transcript cr; show: 'first:2'] newProcess.
process2 := [Transcript cr; show: ' second'] newProcess.
process1 resume.
process2 resume
```

Apart from these two messages (activeProcess and yield), most application programmers will never use ProcessorScheduler directly.

The scheduling is actually performed by the virtual machine. Note that the scheduling algorithm interrupts Processes with a low priority to run Processes with a higher priority, but will not preempt a Process to run one at the same priority.

Ex 5. Browse the class ProcessorScheduler. Note the instance protocol which answers various scheduling priorities.

Ex 6. Try altering the priority of your Processes created earlier. You may need to include Processor yield expressions to prevent one Process from blocking others at the same priority level.

Try not to create Processes which run for ever at high priority levels — such Processes are very difficult to stop!

Ex 7. The standard scheduler does not provide for preemption within a priority level, but does allow higher priority Processes to preempt lower priority Processes. Can you think of a way to mimic truly preemptive Processes by adding a single Process to the system? What are the disadvantages?

Sketch how to implement a scheduler entirely in Smalltalk based on the facilities provided by the virtual machine.

4. Synchronization using Semaphores

So far, we have shown how we create independent Processes using Smalltalk. However, for realistic applications, we expect that there will be some interaction between Processes: these Processes will have references to some objects in common, and such objects may receive messages from several Processes in an arbitrary order. This may lead to unpredictable results.

To illustrate this, consider a class Counter, with one instance variable count:

```
Object subclass: #Counter
    instanceVariableNames: 'count'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Processes–Experiments'
```

On creation, the count instance variable is set to 0 by an initialization method in the instance protocol:

```
reset
    count := 0
```

The ‘instance creation’ class protocol method used is:

```
new
    ^super new reset
```

Instance protocol methods are also provided to read the count instance variable, and to increment this variable.

```
count
    ^count

increment
    count := count + 1
```

Now, suppose we create two Processes, which can both send messages to the same instance of Counter. The first Process repeatedly sends the message increment; the other Process is of a lower priority and sends the message count, checks whether the result is 9 or greater, and if it is then prints it on the Transcript, then resets the count to zero (using the reset message). Note that the two Processes are not synchronized.

```
| counter |
counter := Counter new.
Process1 :=[| delay |
                delay := Delay forMilliseconds: 40.
                [counter increment.
                delay wait] repeat] newProcess.
Process2 := [| delay |
                delay := Delay forMilliseconds: 40.
                [|currentCount|
                currentCount := counter count.
                currentCount >= 9
                    ifTrue:[Transcript cr;
                                show: currentCount printString.
                        counter reset].
                delay wait] repeat] newProcess.
Process2 priority: Processor userBackgroundPriority.
```

Process1 and *Process2* are global variables. The use of Delay is described later in the module.

The two Processes may be started using:

```
Process1 resume.
Process2 resume.
```

The result may appear as in Fig.4 (depending on the amount of user interface activity).

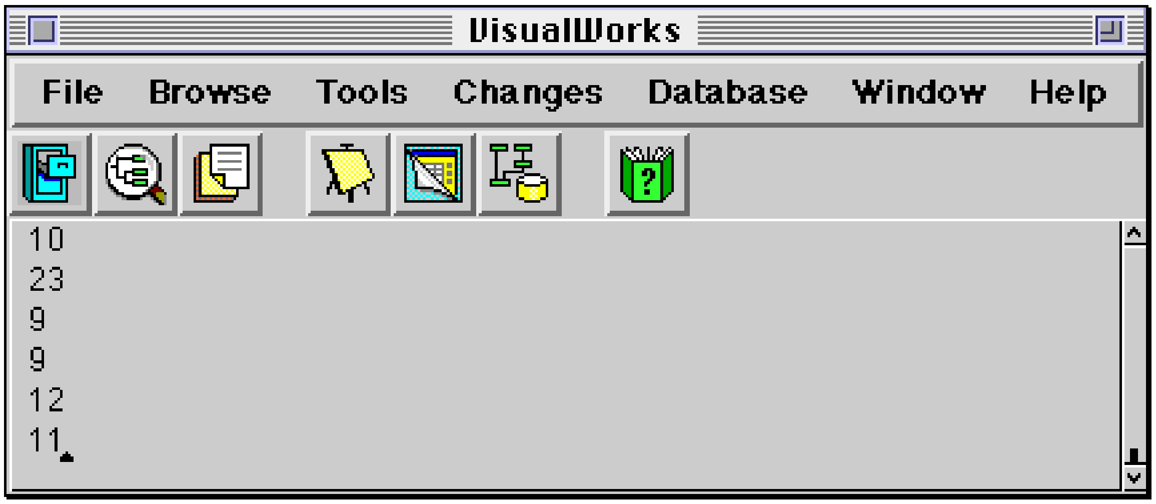
The Processes may be terminated by evaluating the following expressions.

```
Process1 terminate

Process2 terminate
```

Ex 8. Implement the class Counter and type in and evaluate the example code. Start both Processes and examine the Transcript. Explain its behavior.

Ex 9. Now move the cursor around the screen and manipulate the windows. What do you notice abut the sequence of numbers in the Transcript? Can you explain it?



```
10
23
9
9
12
11
```

Figure 4: Two Processes sharing one object

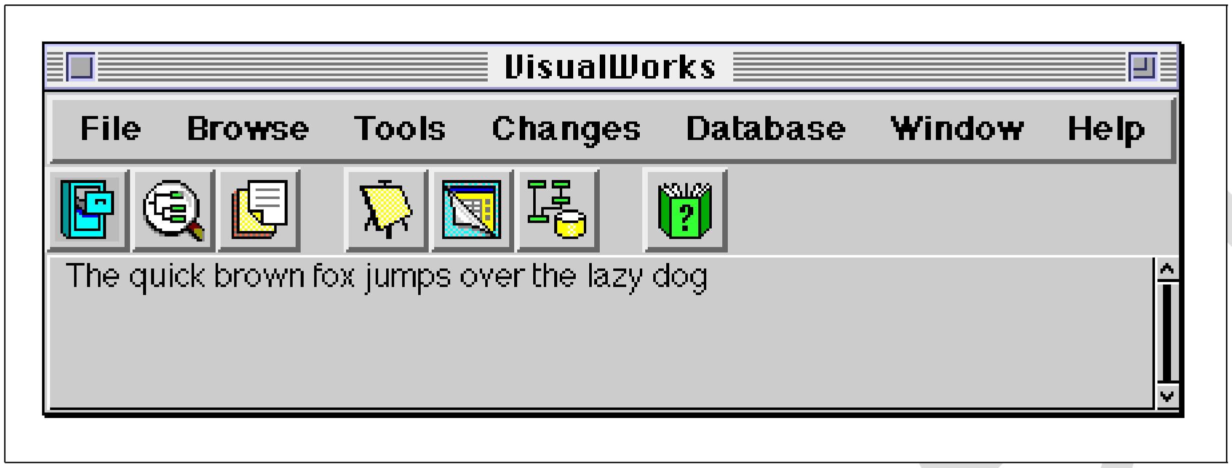
In the example above, the sequence of numbers displayed in the Transcript appears non–deterministic. Fortunately in Smalltalk there is a class, called Semaphore, which is used to synchronize multiple Processes.

5. Semaphores

A Semaphore is an object used to synchronize multiple Processes. A Process waits for an event to occur by sending the message wait to a Semaphore. Another Process then signals that the event has occurred by sending the message signal to the Semaphore. The Process waiting for the signal will not proceed until one is sent. For example,

```
| sem |
Transcript clear.
sem := Semaphore new.
[Transcript show: 'The '] fork.
[Transcript show: 'quick '.
    sem wait.
    Transcript show: 'fox '.
    sem signal] fork.
[Transcript show: 'brown '.
    sem signal.
    sem wait.
    Transcript show: 'jumps over the lazy dog'; cr] fork
```

gives the following result (Fig.5).



The quick brown fox jumps over the lazy dog

Figure 5: Synchronization using a Semaphore

A Semaphore will only release as many Processes from wait messages as it has received signal messages. When a Semaphore receives a wait message for which no corresponding signal has been sent, the Process sending the wait is suspended. Each Semaphore maintains a linked list of suspended Processes, and releases them on a first–in first–out basis (see Fig.6).

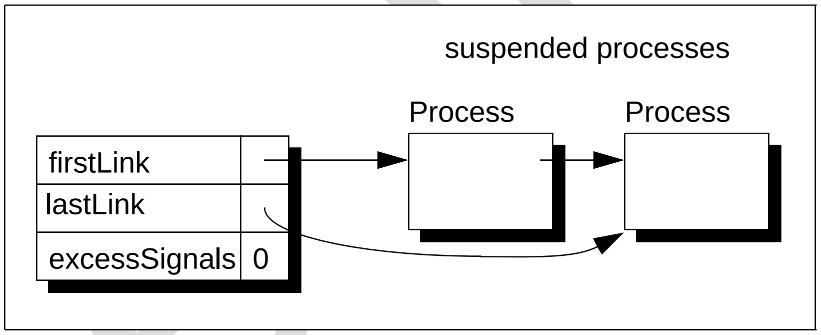


Figure 6: A Semaphore

If a Semaphore receives a wait from two or more Processes, it resumes only one Process for each signal it receives from the Process it is monitoring.

Unlike a ProcessorScheduler, a Semaphore pays no attention to the *priority* of a Process, queuing Processes in the order in which they waited on the Semaphore.

Ex 10. Repeat Ex.5 using a Semaphore to provide simple synchronization between two Processes.

In the exercise above, a Semaphore is used to “hand–off” from one process to another, effectively forcing the processes to be serialized. However, a Semaphore can also be used to guard a critical region of code. For example, consider class OrderItem below:

```
Object subclass: #OrderItem
    instanceVariableNames: 'price quantity '
    classVariableNames: ' '
    poolDictionaries: ''
    category: 'Semaphore Examples'
```

Whenever an instance of this class receives a price: or quantity: message, it prints out its current amount on the Transcript, using the method printAmount, below:

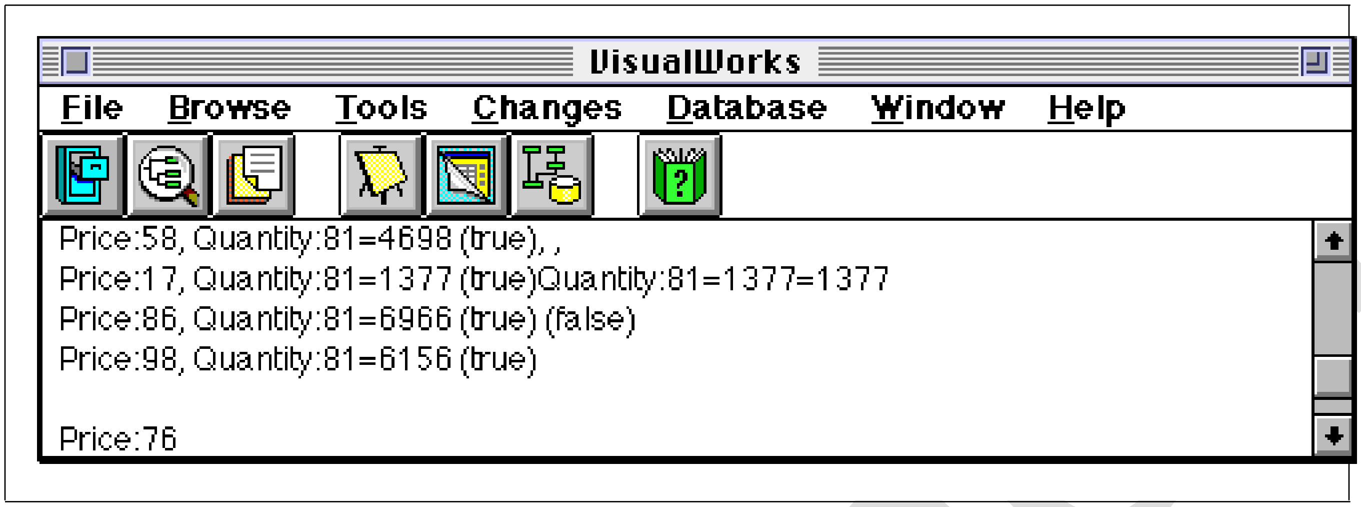
```
printAmount
    | amount |
    Transcript cr; show: 'Price:' , price printString.
    Transcript show: ', '.
    Transcript show: 'Quantity:' , quantity printString.
    Transcript show: '=' , (amount := self amount) printString.
    Transcript space; show: '(' , (amount = (price * quantity)) printString , ')'
```

Note that the printAmount method sends the receiver the message amount. Now, let’s create two Processes, similar to the Counter example earlier:

```
| orderItem delay1 delay2 random |
orderItem := OrderItem new.
random := Random new.
orderItem price: (random next * 100) truncated + 1; quantity: (random next * 100)
truncated + 1.
delay1 := Delay forMilliseconds: 350.
delay2 := Delay forMilliseconds: 660.
Process1 := [
                [| price |
                price := (random next * 100) truncated + 1.
                orderItem price: price.
                delay1 wait] repeat] newProcess.
Process2 := [
                [| quantity |
                quantity := (random next * 100) truncated + 1.
                orderItem quantity: quantity.
                delay2 wait] repeat] newProcess.
Process1 priority: Processor userBackgroundPriority.
Process2 priority: Processor userBackgroundPriority - 2.
```

When the Processes are resumed, the Transcript displays the results of the computation (Fig.7) — note the “false” entry.

(The amount method has been designed to support the example. However, it is not unusual to find methods that are as compute–intensive.)



```
Price:58, Quantity:81=4698 (true), ,
Price:17, Quantity:81=1377 (true)Quantity:81=1377=1377
Price:86, Quantity:81=6966 (true) (false)
Price:98, Quantity:81=6156 (true)

Price:76
```

Figure 7: The result of two Processes accessing one instance of OrderItem

A Semaphore is frequently used to provide mutual exclusion from a “critical region” of code. This is supported by the instance method critical:. The block argument is only executed when no other critical blocks sharing the same Semaphore are evaluating. A Semaphore for mutual exclusion must start out with one extra signal, otherwise the critical section will never be entered. A special instance creation method is provided:

```
Semaphore forMutualExclusion
```

Ex 11. Modify class OrderItem to ensure that the details it displays on the Transcript are always correct. (**Hint:** consider the use of a Semaphore and a critical region.)

6. Delays

Instances of class Delay are used to cause a real time delay in the execution of a Process. An instance of Delay will respond to the message wait by suspending the active Process for a certain amount of time. The time for resumption of the active Process is specified when the Delay is created.

The resumption time can be specified relative to the current time with the messages Delay forMilliseconds: anInteger and Delay forSeconds: anInteger. Delays created in this way can be sent the message wait again after they have finished a previous delay. Examples:

```
Delay forSeconds: 10
```

```
Delay forMilliseconds: 350
```

Once created, an instance of Delay causes the active Process to be suspended when it receives the message wait. Thus, the expressions:

```
| minuteWait |
minuteWait := Delay forSeconds: 60.
minuteWait wait.
```

suspend the active Process for a minute. This could also be expressed as:

```
(Delay forSeconds: 60) wait
```

The resumption time can also be specified at an absolute time with respect to the system's millisecond clock with the message Delay untilMilliseconds: anInteger. Delays created in this way cannot be sent the message wait repeatedly.

Ex 12. Use a Delay to implement a simple clock which prints the current time in the Transcript every few seconds. You may want to use the expression:

```
Transcript cr; show: Time now printString
```

Ex 13. Browse class Delay. Explain how Semaphores and critical regions are used to implement this class.

Ex 14. Add a class method to class Delay which creates an instance of Delay that will wait *until* a specified time. Call the method untilTime:, and ensure that it produces the correct behavior when the following expression is evaluated:

```
(Delay untilTime: (Time readFrom: '14:15' readStream)) wait.
Screen default ringBell
```

7. Shared Queues

When it's necessary to match the output of one Process with the input of another, it's important to ensure that the Processes are synchronized. This synchronization may be achieved using an instance of class SharedQueue, which provides synchronized communication of arbitrary objects between Processes. (SharedQueue uses Semaphores to achieve its synchronization.)

An object is added to a SharedQueue from a Process by sending the message nextPut: (with the object as argument) and retrieved by another Process sending the message next. If no object has been added to the queue when a next message is sent, the Process requesting the object will be *suspended* until an object is available. For example, in the code below, a forked Process puts integers into a SharedQueue which are then retrieved in a repeat loop:

```
| queue |
queue := SharedQueue new.
 [| delay |
delay := Delay forSeconds: 1.
1 to: 100 do:[:i | queue nextPut: i.
                                        delay wait]] fork.
[Transcript cr; show: queue next printString] repeat
```

Ex 15. (Hard) Attempt a Smalltalk representation of "Dijkstra's Dining Philosophers problem":

Five philosophers spend their lives eating and thinking. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the centre of the table, there is a bowl of rice and the table is laid with five chopsticks. When a philosopher thinks, he or she does not interact with any colleagues.

From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest (the chopsticks that are between the philosopher and his or her left and right neighbors). A philosopher may only pick up one chopstick at a time. Obviously, a philosopher cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both chopsticks at the same time, he or she eats without releasing the chopsticks. After eating enough, the philosopher puts down both chopsticks and starts thinking again.

The suggested solution represents each chopstick as a Semaphore, using the wait and signal messages. This guarantees that no two philosophers use the same chopstick simultaneously. The suggested solution is also asymmetric; an odd philosopher picks up the left chopstick first, then the right chopstick, while an even philosopher picks up the right chopstick first.

It is suggested that each philosopher is represented by an instance of class Philosopher. The problem could be a class DiningPhilosophers, with the philosophers and the chopsticks are maintained in instance variables.

You may want to display tracing messages in the System Transcript. How could you introduce some indeterminacy into the solution, given the way the Smalltalk scheduler handles Processes of the same priority?

(The problem presented here is loosely based on the one originally proposed for Little Smalltalk, presented in Tim Budd's book “A Little Smalltalk”, pp. 116—121.)

8. Promises

Class Promise provides a means of evaluating a block in a concurrent Process. An instance of Promise can be created by sending the message promise to a block. For example:

```
[10000 factorial] promise
```

This message creates an instance of Promise, causing the block to be evaluated in a new Process. Alternatively, the message promiseAt: may be used to control the priority of the Process created.

The result of the block can be accessed by sending the message value to the Promise. For example, the following message expressions will print the factorial of 1000 in the Transcript:

```
| promise |
promise := [1000 factorial] promise.
Transcript cr; show: promise value printString
```

However, it’s important to note that if the block has not completed evaluation, then the Process that attempts to read the value of a Promise (by sending it the message value) will wait until the Process evaluating the block has completed. Not surprisingly, Promise achieves this concurrency control by using a Semaphore. A Promise may be interrogated to discover if the process has completed by sending it the message hasValue, for example:

```
| promise delay |
promise := [1000 factorial] promiseAt: Processor userBackgroundPriority.
delay := Delay forMilliseconds: 100.
[promise hasValue]
        whileFalse:[Transcript show: '.'.
                                delay wait].
Transcript cr; show: promise value printString
```

You should also note that it’s not possible to terminate a Promise!

Ex 16. Browse class Promise and implementors of the promise message. Draw a diagram describing the way in which the class uses a block and a Semaphore to provide concurrency control.

Ex 17. Implement a subclass of Promise (called Pledge, say), which extends the functionality of Promise so that it is possible to terminate its Process.