数理解析研究所講究録
第 586 巻 1986 年 134-152

On the concurrency and a yet another standard form of concurrent programs of Smalltalk-80 †

Norihisa Doi
土居範久
Institute of Information Science, Keio University
Kiyoshi Segawa
瀬川 清
Jobu College of Commerce

Abstract

In this paper, the concurrency of Smalltalk-80 is clarified by examining the execution results of the several programs, and a standard form of concurrent programs is presented by using the classical concurrent problems in Smalltalk-80. A scheme to enrich the concurrency in Smalltalk-80 is also proposed.

0. Introduction

In this paper, the concurrency of Smalltalk-80 is clarified by examining the execution results of several concurrent programs, not by looking into the Smalltalk-80 system itself (as physicians do, not as surgeons do). Then, using the classical concurrent problems, such as *Producer Consumer Problem* and *Readers and Writer's Problem*, a standard form of concurrent programs of Smalltalk-80 is presented. Finally, the scheme to enrich the concurrency in Smalltalk-80 is proposed.

1. class `Process`

In Smalltalk-80, a *process* is a sequence of actions described by expressions and performed by the Smalltalk-80 virtual machine. Each process is represented by an *instance* of class `Process`.

A process is created by sending the message `fork`, `forkAt:`, `newProcess` and `newProcessWith:` to a *block*. The process created by `fork` or `forkAt:` is scheduled to be executed by the virtual machine (but it is not always executed immediately), so such a process is called *scheduled*. On the other hand, the process created by `newProcess` or `newProcessWith:` is not scheduled to be executed and is called *suspended*. A suspended process is scheduled by sending the message `resume` to it.

```
Example 1.1
     [actionA] fork.
      actionB.
```

In this example, `actionA` and `actionB` are executed concurrently. ■

```
Example 1.2
     process ← [actionA] newProcess.
     actionB.
     process resume.
     actionC.
```

In this example, `actionB` is executed alone, then `actionA` and `actionC` are executed concurrently. ■

The return value of `[actionA]fork` is the block `[actionA]` itself, not the created process. So if the created process is referred to later, the process must be created by `newProcess` or `newProcessWith:` as in `example 1.2`.

There is only one processor capable of carrying out the sequence of actions a process represented. That is, in spite of saying *"be executed concurrently,"* there is only one process actually being executed. The process which is currently being executed is called *active*.

There exists only one instance of class `ProcessorScheduler` globally named `Processor`. `Processor` selects the active process among the scheduled processes. Moreover, there are some messages for `Processor` to inquire about or change the state of the system. For example, the priority of the active process is returned

† Smalltalk-80 is a trademark of Xerox Corporation

by sending activePriority to Processor and the active process is terminated by sending terminateActive to Processor.

To select the active process, Processor uses a *FCFS(First-Come, First-Serve)* ready queue with priorities. More strictly speaking, it adopts the *multi-queue scheduling algorithm* which uses a *FCFS* ready queue for each priority. The process waiting for the processor longest becomes active first among the processes with the same priority, and the processes with the highest priority become active before the processes with lower priorities. Once a process becomes active, it will be active continuously unless it relinquishes the processor by itself or a process with the higher priority is scheduled (The processor will never be assigned to processes in a *time-slicing* manner). When a process whose priority is higher than that of the active process is scheduled, the new scheduled process preempts the processor. The situation, however, is slightly different if the priority of the scheduled but not active process is changed by the message priority:. In example 1.1, actionB is executed before actionA and in example 1.2 the execution order is "actionB,actionC and actionA," unless each process relinquishes the processor by itself.

A process created by fork, newProcess or newProcessWith: receives the same priority as the process that creates it. On the other hand, a process created by forkAt:level receives the priority level. By sending priority:level to the process, its priority becomes level.

Example 1.3

```
[actionA] forkAt: level1.
 actionB.
```

Assume that the above program is executed in the process with priority level2. If
level1 > level2,
actionA is executed before actionB, and if
level1 ≤ level2,
actionB is executed before actionA. ▪

There are a fixed number of priority levels numbered by ascending integers. Ordinarily, user processes are executed at priority 4. Even if a process whose priority is less than 4 is created in a user process, it is uncertain when the process will be executed (The process created in the user process might be executed after the user process terminates, but it is not correct).

If Processor receives the message yield, Processor suspends the active process and places it at the end of the ready list pertaining to its priority (There are as many lists as priority levels). Then the first process on that list becomes active. If there are no other processes on that list, the process just suspended becomes active again, so yield has no effect in such a case. Sending yield to Processor is one of the methods for relinquishing the processor.

If the block representing the process has arguments, the message newProcessWith: must be used to create the process. The argument of this message must be an instance of Array and the elements of it are given to the block arguments.

2. class Semaphore

A semaphore is used as a synchronization primitive in Smalltalk-80. An instance of a semaphore is created by sending the message new to class Semaphore, and the value of the created semaphore is 0. A semaphore is also created by the message forMutualExclusion instead of new, and its initial value is 1.

Sending the message wait to an instance of Semaphore corresponds to a *P–operation*, and sending signal corresponds to a *V–operation*. A semaphore in Smalltalk-80 is a *counting semaphore* whose value is a non-negative integer. So, if several signal exceed wait, the semaphore remembers the number of the excess signal.

A *semaphore queue* in Smalltalk-80 is a *FIFO queue*, so the processes suspended in the queue will be resumed in the same order in which they were suspended independently of the priority. If the priority of the process resumed by signal is higher than that of the process which sends signal, the resumed process will become active.

In Smalltalk-80, there are two special messages for the *mutual exclusion*. For an instance creation, the message forMutualExclusion is prepared. By this message, as stated before, a semaphore whose value is 1 is created. To execute a block, aBlock, mutual exclusively, the message critical:aBlock is sent to the semaphore. This message is implemented as follows:

```
critical: aBlock
  | value |
  self wait.
  value ← aBlock value.
  self signal.
  ↑value
```

3. class Delay

In Smalltalk-80, semaphores are used to handle *hardware interrupts*. Each process handling a hardware interrupt sends wait to the appropriate semaphore and suspends itself. When an interrupt occurs, the Smalltalk-80 virtual machine detects it and sends signal to the appropriate semaphore to resume the interrupt handling process. The Smalltalk-80 virtual machine detects the following three conditions[Gol]:

1. user event: a key has been pressed on the keyboard, a button has been pressed on the pointing device, or the pointing device has moved,
2. timeout: a specific value of the millisecond clock has been reached, and
3. low space: available object memory has fallen below certain limits.

A class Delay is prepared to handle the second condition. Using this class, the active process might be suspended for a specified amount of time. An instance of Delay is created by the following messages.

forMilliseconds:time — to suspend the active process for time milliseconds.
forSeconds:time — to suspend the active process for time seconds.
untilMilliseconds:time — to suspend the active process until the millisecond clock reaches time.

The active process is actually suspended when the message wait is sent to an instance of Delay. The suspended process will be scheduled when the specified amount of time elapses.

The following two examples show how to suspend the active process for 1 second.

Example 3.1

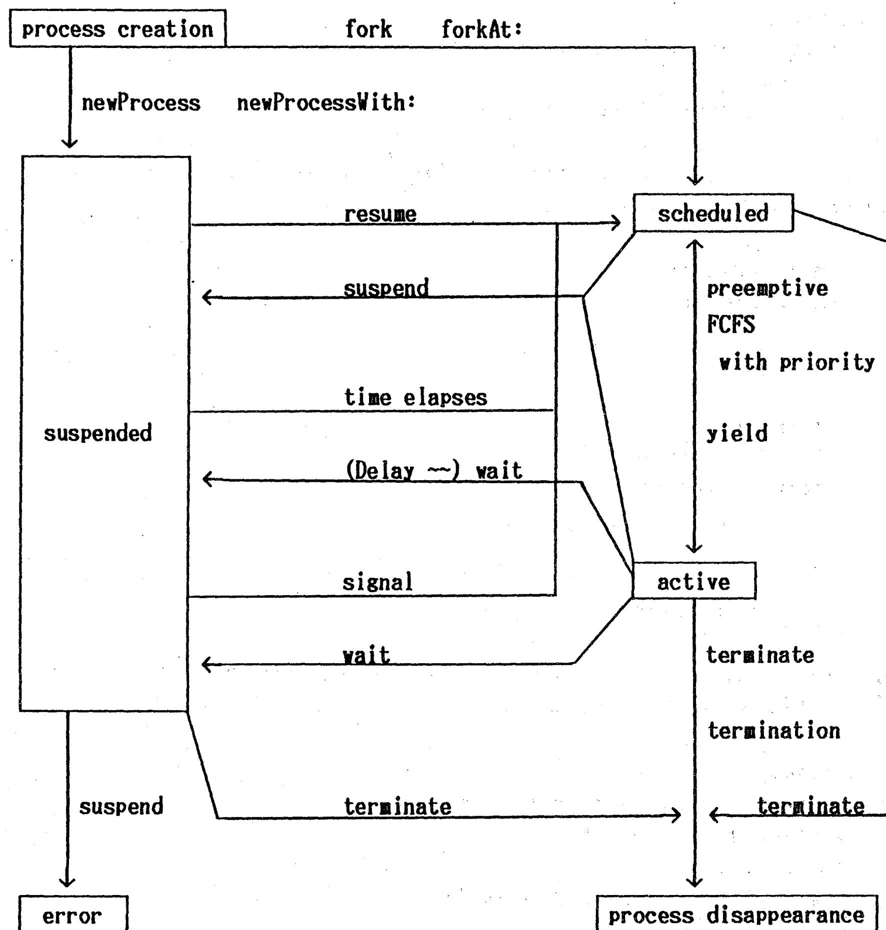
```
| sleepingTime |
sleepingTime ← Delay forSeconds: 1.
sleepingTime wait.
```

Example 3.2

```
(Delay forSeconds: 1) wait.
```

4. State transition of a process

The following is a state transition diagram of a process.



5. Carrying out concurrent processes

In this chapter, we clarify how concurrent processes are executed by the Smalltalk-80 virtual machine through illustrative examples. Techniques for carrying out processes concurrently are also discussed.

5.1 Selection of the active process

As stated before, a preemptive, priority based *FCFS* ready queue is used to select the active process among the scheduled processes in Smalltalk-80. In principle, the active process can be continuously executed to its end. If we run the program:

```
1:  [10 timesRepeat:
2:     [Transcript show: 'forked process'; cr] fork.
3:  10 timesRepeat:
4:     [Transcript show: 'main process'; cr]
```

the child process created by `fork` at line 2 will become active when the parent process terminates. So the result of this program is as follows:

```
main process
main process
      .
      .
      .
main process
forked process
forked process
      .
      .
      .
forked process
```

To execute two processes — *parent* and *child* — concurrently, that is, to activate them alternately, `Processor yield` must be used as follows:

```
1:  [10 timesRepeat:
2:     [Transcript show: 'forked process'; cr.
3:      Processor yield]] fork.
4:  10 timesRepeat:
5:     [Transcript show: 'main process'; cr.
6:      Processor yield].
```

Each process relinquishes the processor at lines 3 and 6 in its own loop, so each process alternately executes its own loop. The result of this program is as follows:

```
main process
forked process
main process
forked process
      .
      .
      .
main process
forked process
```

Instead of `Processor yield` at lines 3 and 6, `(Delay forSeconds: 0)wait` can also be used. If the active process wants to be suspended even for 0 seconds, the context switching occurs and this process becomes inactive. Thus, to exchange the active process, `(Delay ...)wait` may be used(Note that, in some cases, it is better to use `(Delay ...)wait` instead of `Processor yield`. See the section 7.1).

5.2 Changing priority

The priority of a process can be changed by sending the message `priority:` to it. This message, however, does not take effect immediately. The priority of the process will be changed to the specified level after the process becomes active.

```
1 :  | proc1 proc2 |
2 :  proc1 ←
3 :   [Transcript show: 'proc1 - start'; cr.
4 :    Processor yield.
5 :    Transcript show: 'proc1 - end'; cr] newProcess.
6 :  proc2 ←
7 :   [Transcript show: 'proc2 - start'; cr.
8 :    Processor yield.
9 :    Transcript show: 'proc2 - end'; cr] newProcess.
10:  proc1 resume.   proc2 resume.
11:  Transcript show: 'main - 1'; cr.
12:  proc2 priority: 5.
13:  Transcript show: 'main - 2'; cr.
14:  Processor yield.
15:  Transcript show: 'main - 3'; cr.
16:  Processor yield.
17:  Transcript show: 'main - end'; cr.
```

The result is as follows:

```
main - 1
main - 2
proc1 - start
proc2 - start
proc2 - end
main - 3
proc1 - end
main - end
```

The process **proc2** is put into the ready queue for priority 4 by `resume` at line 10. `Priority:5` at line 12 changes the priority of **proc2**, but **proc2** still remains in the ready queue for priority 4. So, the process with priority 4 (lines 13 and 14, and **proc1** activated by line 14) is still being executed. **Proc2** is activated by line 4 in **proc1**. After that the priority of **proc2** is 5, so `Processor yield` at line 8 in **proc2** has no effect because there are no other processes with priority 5.

5.3 Implementation of mutual exclusion by semaphores

The following example shows how to implement a *mutual exclusion* by Semaphore.

```
 1 :  | sem proc1 proc2 |
 2 :  sem ← Semaphore new.
 3 :  proc1 ← [1 to:20 do:
 4 :            [:c1 | sem wait.
 5 :                      c1 printString displayAt:100@100.
 6 :                   sem signal]] newProcess.
 7 :  proc2 ← [1 to:20 do:
 8 :            [:c2 | sem wait.
 9 :                      c2 printString displayAt:100@150.
10:                    sem signal]] newProcess.
11:   proc1 resume.   proc2 resume.
12:   Processor yield.
13:   sem signal.
```

This program alternately displays the numbers from 1 to 20 at the two points (coordinates (100,100) and (100,150)) on *bit mapped display*. The heart of this program is line 12. Processor yield at line 12 activates proc1, but the value of the semaphore sem is 0, so proc1 is suspended at line 4. Then proc2 becomes active and for the same reason becomes suspended at line 8. Now the main process which executes the whole program becomes active again. Sem signal at line 13 is executed and proc1 is scheduled. At this point, the main process terminates and proc1 becomes active. By line 5, 1 is displayed at the points (100,100) and by line 6 proc2 is scheduled (Note that the value of sem is 0). Then, proc1 is suspended at line 4. In turn, proc2 becomes active. By line 9, 1 is displayed at the point (100,150) and by line 10, proc1 is scheduled. Then, proc2 is suspended at line 8. Proc1 becomes active again and so on.

Now, line 12 is assumed to be omitted. After the execution of line 13, proc1 becomes active. The value of sem is 1, so proc1 sets it to 0 and displays 1 on the display. By sem signal at line 6, the value of sem becomes 1 because there are no processes in the semaphore queue(proc2 is waiting in the ready queue). Proc1 can pass through sem wait at line 4 again and display 2. After proc1 has displayed the numbers from 1 to 20, proc2 becomes active. So if Processor yield is lacking, the numbers from 1 to 20 are displayed at point (100,100) and then at point (100,150).

Two special messages for *mutual exclusion* are provided in Smalltalk-80. Using these messages, the above program can be rewritten as follows:

```
 1 :  | sem proc1 proc2 |
 2 :  sem ← Semaphore forMutualExclusion.
 3 :  proc1 ← [1 to: 20 do:
 4 :            [:c1 |
 5 :              sem critical: [c1 printString displayAt:100@100]]
 6 :                                                          ] newProcess.
 7 :  proc2 ← [1 to:20 do:
 8 :            [:c2 |
 9 :              sem critical: [c2 printString displayAt:100@150]]
10:                                                           ] newProcess.
11:   proc1 resume.   proc2 resume.
```

By these messages, the structure of the program is made clear. Proc1, however, precedes proc2, because there are no essential differences between this program and the former one without line 12. To activate proc1 and proc2 alternately, Processor yield must be inserted after displaying the number as stated in 5.1 or the following three lines must be added after line 11 to make this program functionally same as the first one.

```
12:   sem wait.
13:   Processor yield.
14:   sem signal
```

6. Sharing resources among objects

In *sequential programs* or *concurrent programs*, there are two common standard ways to share *resources* among *objects*:

1. using *shared variables*, and
2. passing resources (which are also objects) to objects by *messages*.

6.1 Using shared variables

There are five kinds of *variables* in Smalltalk-80. They are distinguished by their *scopes* and *lifetimes*. These variables may be grouped in two categories, *private (variables)* and *shared (variables)*.

A *private variable* can be accessed by only *one* object, and *instance variables* and *temporary variables* are *private variables*. *Instance variables* are existed during the lifetime of the object which accesses them and *temporary variables* are existed during the activation of one action(e.g. a message). Private variable names must be begun by the lower alphabetical characters.

On the other hand, a *shared variable* can be accessed by *more than one* object. Shared variables may be grouped, and each group is called as a *pool*. A pool can be named and the name of the pool is called as *name of pool*.

The following three variables are shared variables:

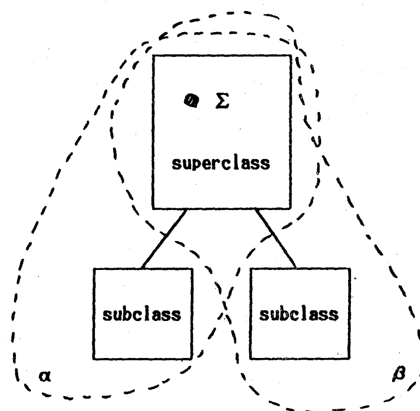
(1) *class variables* — A class variable is shared among all instances of one class. Class variables are declared in the class definition. Each class has a special pool in which all class variables of it are kept.

(2) *global variables* — A global variable is shared among all instances of all classes. To use global variables, they must be added into the special pool named `Smalltalk`.

(3) *pool variables* — A pool variable is shared among the instances of more than one class. To use pool variables, the name of the pool must be declared in the class definition, the pool variables must be added into the pool and the name of the pool must be added into the pool `Smalltalk`.

Shared variable names must be begun by the uppercase alphabetical characters.

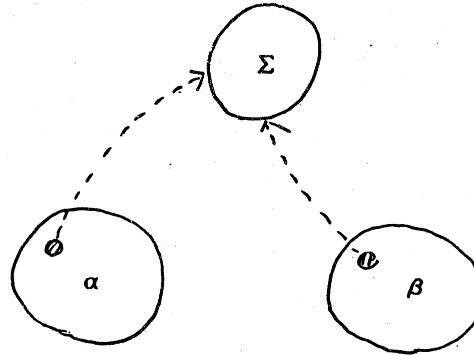
Moreover, in Smalltalk-80, the hierarchy among classes may be defined. The child process (*subclass*) *inherits* the variables and the methods of the parent class (*superclass*) in principle(If the superclass has the method whose pattern is same as the one of the superclass, the subclass uses the own method according to the *rewriting rule* in Smalltalk-80. On the other hand, the subclass can not have the variable whose name is same as the one of the superclass). So, the instances α and β whose classes have the same superclass can share the variable Σ, which is the class variable of the superclass(Note that the contents of the instance of the subclass are constructed of the subclass and all of its ancestor classes).



The instance variables of the superclass can not be shared between α and β because they are created each time when the instance of the subclasses are created.

6.2 Passing the shared object by message

To accomplish the situation in which the object Σ is shared between the objects α and β, the following way can be used. At first, the object Σ is created, then when the objects α and β are created, it is passed as an argument in the instance creation message. In the classes, which are the templates of α and β, the shared object is the argument in the instance creation methods and in the bodies of the instances the shared object is accessed by sending messages to this argument.



By this way, the information about the shared object Σ is hidden automatically(Of course, the classes whose instances are α and β can be defined only when it is known that the object Σ is passed by the argument).

7. A yet another standard form of concurrent programs

In this chapter, we consider the method in which the relation between the *shared resources* and its *users* is defined by the *class-subclass relationship*. This method basically uses the shared variables, but, different from the usual methods, it may be expected that the *readability* of programs is increased because the *shared resource-users relationship* is clarified. This method can be used for both *sequential programs* and *concurrent programs*, and here this method is used for the *classical concurrent processing problems*. For each problem, the Smalltalk-80 program is presented at first to clarify the problem according to the *procedural oriented programming*. Then, the program is examined and the solution according to the *object oriented programming* is presented.

7.1 Producer Consumer Problem

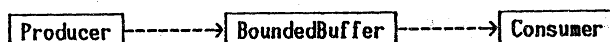
A process producer and a process consumer communicate through a shared bounded buffer buffer. Producer sends data to buffer and consumer gets data from buffer. The following is the solution based on the *procedural oriented programming concept*. Suppose that producer sends a random number between 1 and 10 as its data and that the size of buffer is 5.

```
1 :  | buffer bufferSize valueAvailable spaceAvailable mutex
2 :    readPosition writePosition producer consumer count value rand |
3 :  producer ←
4 :   [[true] whileTrue:
5 :      [(Delay forSeconds: rand next * 10) wait.
6 :       spaceAvailable wait.
7 :       mutex critical:
8 :         [count ← (rand next * 10) truncated + 1.
9 :          buffer at: writePosition put: count.
10:          writePosition ← writePosition \\ bufferSize + 1].
11:       valueAvailable signal]] newProcess.
12:  consumer ←
13:   [[true] whileTrue:
14:      [valueAvailable wait.
15:       mutex critical:
16:         [value ← buffer at: readPosition.
17:          Transcript show: 'get ', value printString; cr.
18:          readPosition ← readPosition \\ bufferSize + 1].
19:       spaceAvailable signal.
20:       (Delay forSeconds: rand next * 10) wait]] newProcess.
21:  bufferSize ← 5.
22:  buffer ← Array new: bufferSize.
23:  valueAvailable ← Semaphore new.
24:  spaceAvailable ← Semaphore new.
25:    bufferSize timesRepeat: [spaceAvailable signal].
26:  mutex ← Semaphore forMutualExclusion.
27:  readPosition ← writePosition ← 1.
28:  rand ← Random new.
29:  producer resume.  consumer resume.
```

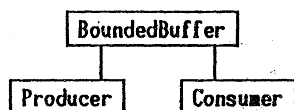
If there are no spaces in buffer, producer will be suspended in the semaphore queue of spaceAvailable, and if buffer is empty, consumer will be suspended in the semaphore queue of valueAvailable. Semaphore mutex is used to accomplish mutually exclusive execution of the critical sections, but in these sections two processes *never* relinquish the processor, so there is no need to construct these parts as critical sections by mutex!

(Delay ...)wait at line 5 determines the time needed to prepare data by a random number and line 20 determines the time needed to process data. They only simulate the situation of taking some time to prepare or process data, and in *real* program there are no triggers such as (Delay ...)wait or Processor yield to exchange the active process. So, if producer becomes active prior to consumer, it is being active until buffer is full. Then consumer becomes active and remains active until buffer is empty. On the other hand, if consumer becomes active prior to producer, it will be suspended at line 14, and after that the situation will be same as before. To activate producer and consumer alternately, it is necessary to insert either Processor yield or (Delay ...)wait at *the points where data is prepared and where data is processed* in the program. If Processor yield is simply inserted at each point, however, the number of data in buffer is at most one.

We are ready to consider the *object oriented version* of this program. BoundedBuffer, Producer and Consumer might be seen as main *objects* in this problem. The relation among these three objects is as follows:



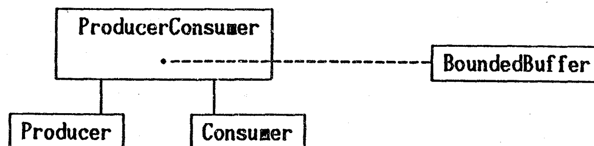
Then, the relationship among these three objects might be seen as follows:



The shared resources — bounded buffer — which is an instance of BoundedBuffer might be assigned to a class variable of BoundedBuffer. If the bounded buffer is assigned to an instance variable of BoundedBuffer instead of a class variable, then the instance of BoundedBuffer might be created each time the instance of Producer or Consumer is created and these bounded buffers can not be shared among producers and consumers.

In Smalltalk-80, *subclass* inherits all the variables and the methods of its *superclass*. In this case, the variables of BoundedBuffer for implementing the buffer, such as the *buffer area* and *pointers*, are in the scope of Producer or Consumer. That is, the information about the bounded buffer can not be hidden from its users — Producer and Consumer.

Then, let define BoundedBuffer as an independent class and ProducerConsumer as a superclass of Producer and Consumer. The instance of BoundedBuffer is assigned to a class variable of ProducerConsumer.



By this, the information about the bounded buffer can be hidden from its users. Moreover, as a side effect, the multiple instances of Producer and Consumer can be created if necessary.

The following Smalltalk-80 program is constructed according to this idea. In this case, each of the producers and consumers is given its identification number. Each producer repeats sending its identification number to the bounded buffer, and each consumer repeats getting it from the bounded buffer and displays it with its own identification number.

```
 1 :class       BoundedBuffer
 2 :superclass  Object
 3 :inst vars   buffer bufferSize readPosition writeposition
 4 :            valueAvailable spaceAvailable mutex

 5 :class method
 6 :  instance creation
 7 :    new: size
 8 :      ↑super new init: size

 9 :instance methods
10:  accessing
11:    remove
12:      | value |
13:      valueAvailable wait.
14:      mutex critical:
15:        [value ← buffer at: readPosition.
16:         readPosition ← readPosition \\ bufferSize + 1].
17:      spaceAvailable signal.
18:      ↑value

19:    deposit: value
20:      spaceAvailable wait.
21:      mutex critical:
22:        [buffer at: writePosition put: value.
23:         writePosition ← writePosition \\ bufferSize + 1].
24:      valueAvailable signal.
25:      ↑value  "no need to return value"

26:  private
27:    init: size
28:      buffer ← Array new: size.
29:      bufferSize ← size.
30:      readposition ← writePosition ← 1.
31:      valueAvailable ← Semaphore new.
32:      spaceAvailable ← Semaphore new.
33:        size timesRepeat: [spaceAvailable signal].
```

```
34:      mutex ← Semaphore forMutualExclusion

1 :class       ProducerConsumer
2 :superclass  Object
3 :class vars  Buffer Rand

4 :class method
5 :  initialization
6 :    initialize
7 :      Buffer ← BoundedBuffer new: 5.
8 :      Rand ← Random new.
9 :      Producer initialize.
10:      Consumer initialize

11:  example
12:    example
13:      | p1 p2 c1 c2 |
14:      ProducerConsumer initialize.
15:      p1 ← Producer new.
16:      p2 ← Producer new.
17:      c1 ← Consumer new.
18:      c2 ← Consumer new.
19:      p1 resume.    p2 resume.
20:      c1 resume.    c2 resume

1 :class       Producer
2 :superclass  ProducerConsumer
3 :class var   NoOfProducer
4 :inst var  myName

5 :class methods
6 :  initialization
7 :    initialize
8 :      NoOfProducer ← 0

9 :  instance creation
10:    new
11:      | newProducer |
12:      NoOfProducer ← NoOfProducer + 1.
13:      newProducer ← super new.
14:      newProducer setName.
15:      ↑newProducer define

16:instance methods
17:  private
18:    setName
19:      myName ←  NoOfProducer

20:    define
21:      | newProducer |
22:      newProducer ←
23:       [[true] whileTrue:
24:          [Buffer deposit: myName.
25:           Transcript show: 'Producer - ', myName printString; cr.
26:           (Delay forSeconds: Rand next * 10) wait]] newProcess.
27:      ↑newProducer

1 :class       Consumer
2 :superclass  ProducerConsumer
3 :class var   NoOfConsumer
4 :inst var    myName

5 :class methods
6 :  initialization
7 :    initialize
8 :      NoOfConsumer ←  0
```

```
9 :  instance creation
10:    new
11:      | newConsumer |
12:      NoOfConsumer ← NoOfConsumer + 1.
13:      newConsumer ← super new.
14:      newConsumer setName.
15:      ↑newConsumer define

16:instance methods
17:  private
18:    setName
19:      myName ← NoOfConsumer

20:    define
21:      | newConsumer data |
22:      newConsumer ←
23:       [[true] whileTrue:
24:          [data ← Buffer remove.
25:           Transcript show: 'Consumer - ', myName printString,
26:             ' : ', data printString; cr.
27:           (Delay forSeconds: Rand next * 10) wait]] newProcess.
28:      ↑newConsumer
```

An example of using these classes is shown as a class method example in the class ProducerConsumer, and two instances of Producer and Consumer are created. The following is one of the results of this example:

```
Producer - 1
Producer - 2
Consumer - 1 : 1
Consumer - 2 : 2
Producer - 1
Producer - 1
Producer - 2
Consumer - 1 : 1
Consumer - 2 : 1
Producer - 2
Consumer - 1 : 2
   .
   .
   .
```

There is no need to use a semaphore mutex in the instance methods remove and deposit: if these classes are used as in example. If the priorities of producers and consumers are different, mutex becomes necessary. For example, suppose that the priority of p1 is higher than that of p2. Now p1 is suspended during the time determined by a random number, and p2 is active and begins to execute deposit:. At this point, if the waiting time of p1 expires, p1 becomes active and then executes deposit:. So, to preserve the consistency, we can not omit mutex. (Incidentally, note that p1, p2, c1 and c2 are the instances of class Process, not class Producer or Consumer.)

7.2 Readers and Writer's Problem

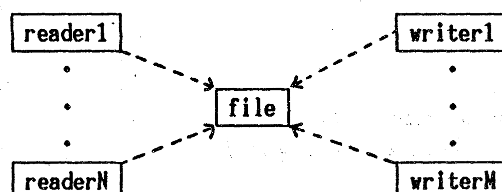
The following is the solution for this problem based on the *procedural oriented programming concept*[Cou]. In this solution, it is assumed that *two* readers and *one* writer access the shared resource.

```
1 :  | readCount mutex w reader1 reader2 writer rand |

2 :  reader1 ←
3 :   [[true] whileTrue:
4 :      [mutex wait.
5 :         readCount ← readCount + 1.
6 :         (readCount = 1) ifTrue: [w wait].
7 :       mutex signal.

8 :         Transcript show: 'reader1 - start'; cr.
9 :         (Delay forSeconds: rand next * 10) wait.
10:         Transcript show: 'reader1 - end'; cr.

11:       mutex wait.
12:         readCount ← readCount - 1.
13:         (readCount = 0) ifTrue: [w signal].
14:       mutex signal.

15:       (Delay forSeconds: rand next * 10) wait]] newProcess.

          reader2  same as reader1

16:  writer ←
17:   [[true] whileTrue:
18:      [w wait.
19:         Transcript show: 'writer - start'; cr.
20:         (Delay forSeconds: rand next * 10) wait.
21:         Transcript show: 'writer - end'; cr.
22:       w signal.

23:       (Delay forSeconds: rand next * 10) wait]] newProcess.

24:  readCount ← 0.
25:  mutex ← Semaphore forMutualExclusion.
26:  w ← Semaphore forMutualExclusion.
27:  rand ← Random new.
28:  reader1 resume.     reader2 resume.     writer resume.
```

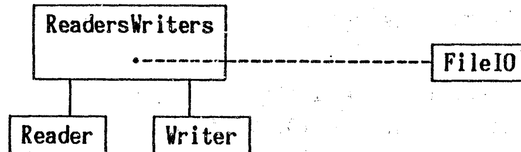
Lines 8 to 10 mean that *reader1 is reading from the shared resource* and lines 19 to 21 mean that *writer is writing into the shared resource.* There are some (Delay ...)wait in this program. As in the *Producer Consumer Problem*, they simulate the time to perform *reading* and *writing*, and they never appear in the *real* program, but are needed to execute the processes concurrently.

The semaphore mutex is used to construct the critical section. This kind of guard can be omitted in the *Producer Consumer Problem*, but it can not be omitted in the following case. Suppose that lines 4 and 7 are omitted so there is no guard around the critical section. When writer is using the shared resource, if reader1 attempts to use the resource because the value of readCount is 1 (after incremented by reader1), it is suspended at line 6. At this point, if reader2 becomes active and attempts to use the resource, it is able to use it because the value of readCount is 2 so reader2 skips w wait at line 6. If line 4 exists, reader2 is suspended at line 4 in this case, so there is no problem. On the other hand, in the critical section guarded by lines 11 and 14, there is no chance to exchange the active process so there is no need to construct this part as critical section.

Now we will solve this problem based on the *object oriented programming concept*. In this case, *File*(shared resource), *Reader* and *Writer* are considered as *objects*. Multiple instances of *Reader* and *Writer* can be created. The relation among these is sketched as follows, where, reader1, ..., readerN are the instances of Reader, writer1, ..., writerM are the instances of Writer and file is the instance of File:



This relation has the difference in *nuance* which the relation in the *Producer Consumer Problem* has, but in both cases there is one *shared resource* and there are two kinds of *users*. So, the standard form derived from the *Producer Consumer Problem* can also be applied to this problem. To perform the *information hiding* of File and emphasize the combination of the file itself and its operations, the class for the shared resource is named as FileIO. The instance of FileIO is assigned to the class variable of ReadersWriters, which is the superclass of Reader and Writer. The relation among these classes is as follows:



A program based on this idea is shown below.

```
 1 :class        FileIO
 2 :superclass   Object
 3 :inst vars    mutex w readCount rand

 4 :class method
 5 :  instance creation
 6 :    new
 7 :      ↑super new initialize

 8 :instance methods
 9 :  accessing
10:     concurrentRead: reader
11:       self startRead.
12:       Transcript show: 'reader', reader printString,
13:         ' - start'; cr.
14:       (Delay forSeconds: rand next * 10) wait.
15:       Transcript show: 'reader', reader printString,
16:         ' - end'; cr.
17:       self endRead

18:     exclusiveWrite: writer
19:       self startWrite.
20:       Transcript show: 'writer', writer printString,
21:         '- start'; cr.
22:       (Delay forSeconds: rand next * 10) wait.
23:       Transcript show: 'writer', writer printString,
24:         ' - end'; cr.
25:       self endWrite

26:   private
27:     initialize
28:       readCount ← 0.
29:       mutex ← Semaphore forMutualExclusion.
30:       w ← Semaphore forMutualExclusion.
31:       rand ← Random new

32:     startRead
33:       mutex wait.
34:         readCount ← readCount + 1.
35:         (readCount = 1) ifTrue: [w wait].
36:       mutex signal

37:     endRead
38:       mutex wait.
39:         readCount ← readCount - 1.
40:         (readCount = 0) ifTrue: [w signal].
41:       mutex signal

42:     startwrite
```

```
43:       w wait

44:     endWrite
45:       w signal

1 :class         ReadersWriters
2 :superclass    Object
3 :class vars    File Rand

4 :class methods
5 :   initialization
6 :     initialize
7 :       File ← FileIO new.
8 :       Reader initialize.
9 :       Writer initialize.
10:       Rand ← Random new

11:   example
12:     example
13:       | r1 r2 r3 w1 w2 |
14:       ReadersWriters initialize.
15:       r1 ← Reader new.
16:       r2 ← Reader new.
17:       r3 ← Reader new.
18:       w1 ← Writer new.
19:       w2 ← Writer new.
20:       r1 resume.    r2 resume.    r3 resume.
21:       w1 resume.    w2 resume

1 :class         Reader
2 :superclass    ReadersWriters
3 :class var     NoOfReaders
4 :inst var      myName

5 :class methods
6 :   initialization
7 :     initialize
8 :       NoOfReaders ← 0

9 :   instance creation
10:     new
11:       | newReader |
12:       NoOfReaders ← NoOfReaders + 1.
13:       newReader ← super new.
14:       newReader setName.
15:       ↑newReader define

16:instance methods
17:   private
18:     setName
19:       myName ← NoOfReaders

20:     define
21:       ↑[[true] whileTrue:
22:           [(Delay forSeconds: Rand next * 10) wait.
23:            File concurrentRead: myName]] newProcess

1 :class         Writer
2 :superclass    ReadersWriters
3 :class var     NoOfWriters
4 :inst vars     myName

5 :class methods
6 :   initialization
7 :     initialize
8 :       NoOfWriters ← 0

9 :   instance creation
```

```
10:     new
11:       | newWriter |
12:       NoOfWriters ← NoOfWriters + 1.
13:       newWriter ← super new.
14:       newWriter setName.
15:       ↑newWriter define
16:instance methods
17:  private
18:     setName
19:       myName ← NoOfWriters
20:     define
21:       ↑[[true] whileTrue:
22:          [(Delay forSeconds: Rand next * 10) wait.
23:           File exclusiveWrite: myName]] newProcess
```

An example of using these classes is shown as a class method example in the class ReadersWriters, and three instances of Reader and two instances of Writer are created. One of the execution results of this problem is as follows.

```
reader2 - start
reader3 - start
reader1 - start
reader3 - end
reader1 - end
reader3 - start
reader1 - start
reader2 - end
reader1 - end
reader3 - end
writer1 - start
writer1 - end
writer2 - start
writer2 - end
reader1 - start
     .
     .
     .
```

Unlike the *monitor procedures* of *Hoare's monitor*[Hoa], more than one process is able to use the methods in one class simultaneously (a method even can be used by more than one process at the same time). So the users of the shared resource have only to use the two methods **concurrentRead:** and **exclusiveWrite:**, unlike *Hoare's monitor* in which we have to use **startRead**, **endRead**, **startWrite** and **endWrite**. Accordingly, protection of the shared resource is easily accomplished in Smalltalk-80, and cannot in *Hoare's monitor*. If *monitor* is used for this problem, the users must execute **startread** or **startWrite** before using the shared resource and **endRead** or **endWrite** after completion of using it. Moreover, accessing the shared resource is done directly by users, the users can use it without executing **startRead** or **startWrite**. So it is natural that the idea has come up which restrict the execution order of *monitor procedure* by *path expressions*[Cam], but such constructs may not be necessary if Smalltalk-80 is used.

There are, however, some problems. If the class FileIO is defined as before, the processes which use the instance of it must know how to access it completely. By this, it is not suitable that the resource is capsuled at which it is scheduled. Moreover, in Smalltalk-80, if several processes execute the methods in the same class at the same time, the copy of the text is prepared for each process. So it must be noted about that and in some cases such methods must be constructed as *critical section*.

The semaphore mutex in the instance methods **startRead** and **endRead** can not be omitted. The reason for the case in **startRead** is the same as in first version, and the reason for the case in **endRead** is as follows. Supposed that there are two readers, say r1 and r2, and the priority of r1 is higher than r2. Now r1 is being suspended at line 22 and r2 has used the shared resource. If the suspending time of r1 elapses just before r2 executes **w signal**, r1 becomes active, executes **startRead**, uses the shared resource, and comes to suspended at line 14. At this point, if r2 becomes active, it executes **w signal**. Consequently, the writer can enter into the critical section even if r1 is in the critical section.

8. Toward realistic concurrent processes

As stated before, in principle, once a process of Smalltalk-80 gets the processor, the process keeps it continuously unless the process relinquishes it by itself. If the existing algorithms for concurrent processing are used to compose Smalltalk-80 programs, not only may the programs contain useless parts but also the processes may not be executed concurrently.

In the following sections, we will discuss how to execute the processes concurrently. Three methods are suggested below. The first two methods impose some burden on the programmers and the last one is to reconstruct the Smalltalk-80 system so that there is no such imposition on the programmers.

8.1 The method using `Processor yield` or `(Delay ...)wait`

This method is the easiest one. If `Processor yield` is inserted at every space among the lines in the program, the program turns out to be executed as if the original program is executed on the system which adopts *time slicing*. Clearly, this method not only imposes an enormous overhead on the system, but it is also troublesome.

To avoid such annoyances, `Processor yield` or `(Delay ...)wait` has only to be inserted at the *appropriate* points. Here, the appropriate points mean the points which must be executed concurrently or the points at which the active process must be exchanged.

Concrete points are different according to the programs, but the typical patterns can be put in order as the standard form of the Smalltalk-80 programs argued in chapter 6. Moreover, it is not desirable that the things which are not related to the algorithms like `Processor yield` or `(Delay ...)wait` must be used in the programs and the programmers must do such work. Process scheduling is one of the important things with which the operating system must deal.

8.2 The method making monitors

Let consider the method in which the *monitors* managing the processes are prepared and nothing is inserted into the user programs. This monitor is one of the user's processes managed by `Processor` and a private scheduler for the particular program.

The outline of this monitor is as follows. Let $p_1, p_2, \ldots, p_n$ be the processes created in the program and the priorities of these processes be lower than that of the monitor.

```
create processes
  p1 ← ... newProcess.
     .
     .
     .
  pn ← ... newProcess.
  i ← 1.
repeat the following block
  [pi resume.
   (Delay forMilliseconds:slice)wait.
  pi suspend.
   i ← i \\ n + 1]
```

This method has the following problems:

1. How to manage processes which are created dynamically,
2. How to detect the process termination, and
3. This monitor is private to the program, so the new monitor must be prepared for each program.

8.3 Reconstructing ProcessorScheduler

Because *objects*, originally, exist independently, the Smalltalk-80 system may be reconstructed to carry out all objects concurrently. One of the examples based on this idea is *Concurrent Smalltalk*[Yok]. Concurrent activities, however, depend on the scheduling algorithms. In Smalltalk-80, this problem may be solved by using simple *time slicing*.

To implement the time slicing, it seems to be enough to prepare the process TimeSlicer as follows. Of course, Processorscheduler must be reconstructed, and the extent of the effort to reconstruct the system will depend on future research.

```
1:  TimeSlicer ←
2:     [[true] whileTrue:
3:         [i ← Processor activeProcess. "set the active process"
4:          Processor yield: i. "put process i on end of the ready queue"
5:          (Delay forMilliseconds: slice) wait]].
```

Note that activeProcess and Processor yield: are not existing methods in Smalltalk-80.

Acknowledgment

Authors would like to thank Y. Yamamoto for discussing with them and his helpful comments, and M. Tokoro for his many comments about the usage of shared resources.

References

[Cam] Campbell,R. H. and Habermann,A. N., "The Specification of Process Synchronization by Path Expressions," in *Lecture Notes in Computer Science*, Vol.16, 89-102.

[Cou] Courtois,P. J., Heymans,F. and Parnas,D. J., "Concurrent Control with Readers and Writers," *CACM*, Vol.14, No.10, 667-668(1971).

[Gol] Goldberg,A. and Robson,D., "Smalltalk-80 : the language and its implementation," Addison-Wesley, 1983.

[Hoa] Hoare,C.A.R., "Monitors : An Operating System Structuring Concept," *CACM*, Vol.17, No.10, 549-557(1974).

[Yok] Yokote,Y. and Tokoro,M., "Concurrent Smalltalk," *Computer Software*, Vol.2, No.4, 2-18(1985)(in Japanese).