Concurrent Programming in Pharo

Stéphane Ducasse and Guillermo Polito

December 22, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

	Illustrations	**iii**
1	**Concurrent Programming in Pharo**	**3**
1.1	Studying an example	3
1.2	A simple example	4
1.3	Process Lifetime	4
1.4	Creation API Summary	6
1.5	First look at ProcessorScheduler	7
1.6	Process	9
1.7	Conclusion	9
2	**Semaphores**	**11**
2.1	Understanding semaphores	11
2.2	An example	14
2.3	wait and signal interplay	14
2.4	Prearmed semaphore	15
2.5	forMutualExclusion	16
2.6	Implementation	17
2.7	Conclusion	18
3	**Scheduler's principles**	**19**
3.1	Delay	19
3.2	Example	21
4	**Synchronisations**	**23**
4.1	Motivation	23
4.2	Using a semaphore	24
4.3	Using a Mutex	24
4.4	About Rendez-vous	25
4.5	Shared Queue	27
4.6	Monitor	28
4.7	Basic usage:	29
4.8	We need one example here	29
5	**Mutex implementation**	**31**

6 ShareQueue: a Semaphore Example **33**
6.1 Conclusion 34

Illustrations

1-1 Two interleaving processes. . . . 4
1-2 Process states: A process (green thread) can be in one of the following states: **runnable**, **suspended**, **executing**, **waiting**, **terminated**. . . . 5
1-3 The scheduler knows the currently active process as well as the lists of pending processes based on their priority. . . . 8

2-1 The semaphore protects resources: P0 is using the ressources, P1...2 are waiting for the resources. . . . 12
2-2 The process P4 wants to access the resources: it sends wait to the semaphore. . . . 12
2-3 P4 is added to the waiting list. . . . 13
2-4 P0 has finished to use the resources: it signals it to the semaphore. The semaphore resumes the first pending process. . . . 13
2-5 The resumed process, P1, is added to the scheduled list of process of the ProcessScheduler. . . . 13

3-1 BBB . . . 20
3-2 Revisiting the process states . . . 20

This book describes the low-level abstractions available in Pharo for concurrent programming. It explains pedagogically different aspects. Now, if you happen to create many green threads (called Process in Pharo) we suggest that you have a look at TaskIt. TaskIt is an extensible library to manage concurrent processing at a higher-level of abstractions. You should definitively have a look at it.

CHAPTER 1

Concurrent Programming in Pharo

Pharo is a sequential language since at one point in time there is only one computation carried on. However, it has the ability to run programs concurrently by interleaving their executions. The idea behind Pharo is to propose a complete OS and as such a Pharo run-time offers the possibility to execute different processes in Pharo lingua (or green threads in other languages) that are scheduled by a process scheduler defined within the language.

Pharo's concurrency is *collaborative* and *preemptive*. It is *preemptive* because a process with higher priority can interrupt the current running one. It is *collaborative* because the current process should explicitly release the control to give a chance to the other processes of the same priority to get executed by the scheduler.

In this chapter we present how processes are created and their lifetime. We present semaphores since they are the most basic building blocks to support concurrent programming and the infrastructure to execute concurrent programs. We will show how the process scheduler manages the system.

In a subsequent chapter we will present the other abstractions: Mutex, Monitor and Delay.

1.1 Studying an example

Pharo supports the concurrent execution of multiple programs using independent processes (green threads). These processes are lightweight processes as they share a common memory space. Such processes are instances of the class `Process`. Note that in operating systems, processes have their

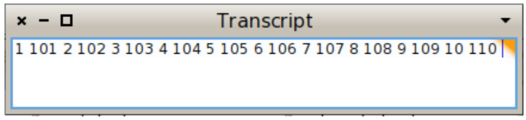

Figure 1-1 Two interleaving processes.

own memory and communicate via pipes supporting a strong isolation. In Pharo, processes are what is usually called a (green) thread in other languages. They have their own execution flow but share the same memory space and use concurrent abstractions such semaphores to synchronize with each other.

1.2 A simple example

Let us start with a simple example. We will explain all the details in subsequent sections. The following code creates two processes using the message `fork` sent to a block. In each process we enumerate numbers. During each loop step, using the expression `Processor yield`, the current process stops its execution to give a chance to other processes with the same priority to get executed. At the end of the loop we refresh the `Transcript` output.

```
[ 1 to: 10 do: [ :i |
  Transcript nextPutAll: i printString, ' '.
  Processor yield ].
Transcript endEntry ] fork.

[ 101 to: 110 do: [ :i |
  Transcript nextPutAll: i  printString, ' '.
  Processor yield ].
Transcript endEntry ] fork
```

Figure 1-1 shows the output produced by the execution of the snippet.

We see that the two programs run concurrently, each outputting a number at a time and not producing two numbers in a row.

Let us look at what is a process.

1.3 Process Lifetime

A process can be in different states depending on its life-time (**runnable**, **suspended**, **executing**, **waiting**, **terminated**) as shown in Figure 1-2. We look at such states now.

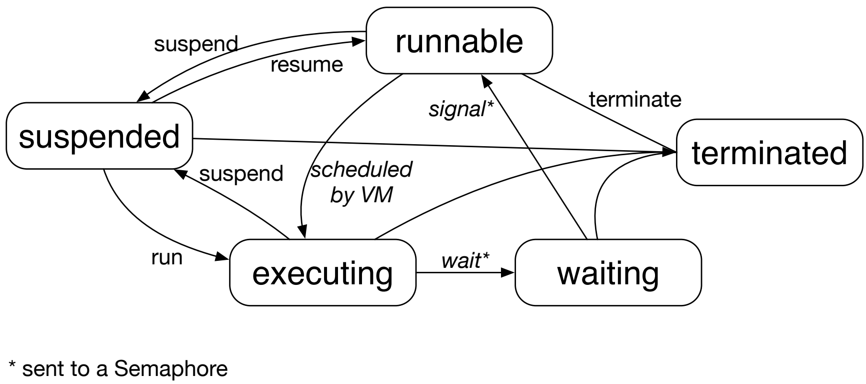

Figure 1-2 Process states: A process (green thread) can be in one of the following states: **runnable**, **suspended**, **executing**, **waiting**, **terminated**.

Creating and launching a new process

To execute concurrently a program, we write such a program in a block and send to the block the message `fork`.

```
[ 1 to: 10 do: [ :i | i  printString traceCr ]
  ] fork
```

This expression creates an instance of the class `Process`. It is added to the list of scheduled processes of the process scheduler (as we will explained later). We say that this process is `runnable`: it can be potentially executed. It will be executed when the process scheduler will schedule it as the current running process and give it the flow of control. At this moment the block of this process will be executed.

Creating a process without scheduling it

We can also create a process which is not scheduled (hence **suspended**) using the message `newProcess`.

You can i

```
| pr |
pr := [ 1 to: 10 do: [ :i |
   i  printString traceCr ] ] newProcess.
pr inspect
```

This creates a process in `suspended` state, it is not added to the list of the scheduled processes of the process scheduler. It is not that is not `runnable`. It can be scheduled sending it the message `resume`.

In the inspector open by the previous expression, you can execute `self resume` and then the process will be scheduled.

```
self resume
```

Also **suspended** process can be executed immediately by sending it the `run` message. The message `run` suspends the current process and execute the receiver process at the highest priority.

Passing arguments to a process

You can also pass arguments to a process with the message `newProcessWith: anArray` as follows:

```
| pr |
pr := [ :max |
  1 to: max do: [ :i |
    i  printString crTrace ] ] newProcessWith: #(20).
pr resume
```

Note that the elements of the argument array are passed to the corresponding block parameters.

Suspending and terminating a process

A process can also be temporarily suspended (i.e., stopped) using the message `suspend`. A suspended processed can be rescheduled using the message `resume`. We can also terminate a process using the message `terminate`. A terminated process cannot be scheduled any more.

Creating a waiting process

As you see on Figure 1-2 a process can be in a waiting state. It means that the process is blocked waiting to be rescheduled. This happens when you need to synchronize concurrent processes. The basic synchronization mechanism is a semaphore and we will cover this deeply in subsequent sections.

1.4 Creation API Summary

The process creation API is composed of messages sent to blocks.

- `[ ] newProcess` creates a unscheduled process whose code is the receiver bloc. The priority is the one of the active process.
- `[ ] newProcessWith: anArray` same as above but pass arguments (defined by an array) to the block.
- `[ ] fork` creates a new scheduled process. It receives a `resume` message so it is added to the queue corresponding to its priority.

- `[ ] forkAt:` same as above but with the specification of the priority.

1.5 First look at ProcessorScheduler

Pharo implements time sharing where each process (green thread) has access to the physical processor during a given amount of time. This is the responsibility of the `ProcessorScheduler` and its unique instance `Processor` to schedule processes.

The scheduler maintains lists of pending processes as well as the currently active one (See Figure 1-3). To get the running process, you can execute: `Processor activeProcess`.

Process priority

At any time only one process can be executed. First of all, the processes are being run according to their priority. This priority can be given to a process with `priority:` message, or `forkAt:` message sent to a block. There are couple of priorities predefined and can be accessed by sending specific messages to `Processor`. For example, the following snippet is run at the same priority that background user tasks.

```
[ 1 to: 10 do: [ :i | i  printString crTrace ]
  ] forkAt: Processor userBackgroundPriority
```

Next table lists all the predefined priorities together with their numerical value and purpose.

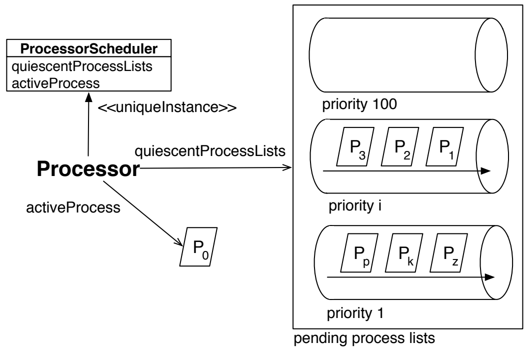

Figure 1-3 The scheduler knows the currently active process as well as the lists of pending processes based on their priority.

Priority	Name
100	timingPriority For processes that are dependent on real time. For example, Delays (see later).
98	highIOPriority For time–critical I/O processes, such as handling input from a network.
90	lowIOPriority For most I/O processes, such as handling input from the user.
70	userInterruptPriority For user processes desiring immediate service. Processes run at this level will preempt the window scheduler and should, therefore, not consume the Processor forever.
50	userSchedulingPriority For processes governing normal user interaction. The priority at which the window scheduler runs.
30	userBackgroundPriority For user background processes.
10	systemBackgroundPriority For system background processes. Examples are an optimizing compiler or status checker.
1	systemRockBottomPriority The lowest possible priority.

The scheduler knows the currently active process as well as the lists of pending processes based on their priority. It maintains an array of linked-lists per priority as shown in Figure 1-3. It uses the priority lists to manage processes that are suspended (and waiting to be scheduled) in the first in first out way.

There are simple rules to interrupt and change the process to be run:

- Processes with higher priority can interrupt lower priority processes if they have to be executed.

- Processes with the same priority are executed in the same order they were added to scheduled process list.
- As mentioned before, a process (green thread) should use `Processor yield` to give an opportunity to run to the other processes with the same priority. In this case, the yielding process is moved to the end of the list to give a chance to execute all the pending processes (see below Scheduler's principles).

> **Note** In the case of a higher priority level process interrupting a process of lower priority, when the interrupting process releases the control, the question is then what is the next process to resume: the interrupted one or another one. In Pharo legacy, the interrupted process is put at the end of the waiting queue, while a better design is to resume the interrupted process to give it a chance to continue its tasks.

1.6 Process

A process is an instance of the class `Process`. This is class is a subclass of the class `Link`. A link is an element of a linked list (class `LinkedList`). This design is to make sure that processes can be elements in a linked list without wrapping them in a `Link` instance. Note that this linked list is tailored for the Process scheduler logic. Better use another one if you need one.

A process has the following instance variables:

- priority: holds an integer to represent the priority level of the process.
- suspendedContext: holds the execution context (stack reification) at the moment of the suspension of the process.
- myList: the list of processes to which the suspended process belongs to.

1.7 Conclusion

We presented briefly the concurrency model of Pharo: preemptive and collaborative. A process of higher priority can stop the execution of processes of lower ones. Processes at the same priority should explicit return control using the `yield` message. We presented the notion of process (green thread) and process scheduler. In the next chapter we explain semaphores since we will explain how the scheduler uses delays to performing its scheduling.

CHAPTER 2

Semaphores

Often we encounter situations where we need to synchronize processes. For example, imagine that you only have one pen and that there are several writers wanting to use it. You will wait for the pen and once the pen is released, you will be able to access and use it. Now since multiple people can wait for the pen, the waiters are ordered in a waiting list associated with the pen. When the current writer does not need the pen anymore, he will say it and the next writer in the queue will be able to use it. Writers needed to use the pen just register to the pen: they are added at the end of the waiting list.

In fact, our pen is a semaphore. Semaphores are the basic bricks for concurrent programming and even the scheduler itself uses them. A great book proposes different synchronization challenges that are solves with Semapharo: The Little Book of Semaphores. It is clearly a nice further readings.

2.1 Understanding semaphores

A Semaphore is an object used to synchronize multiple processes. A semaphore is often used to make sure that a resource is only be accessed by a single process at the time.

A process that wants to access to a resources will declare to the semaphore protecting the resources by sending to the semaphore the message `wait`. The semaphore will add this process to its waiting list. A semaphore keeps a list of waiting processes that want to access to the resource protected by the semaphore. When the process currently using the resources does not use it anymore, it signals it to the semaphore sending the message `signal`. The semaphore resumes the first waiting process which is added to the suspended list of the scheduler.

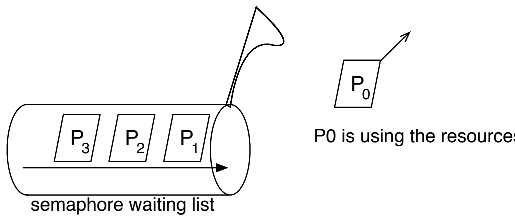

Figure 2-1 The semaphore protects resources: P0 is using the ressources, P1...2 are waiting for the resources.

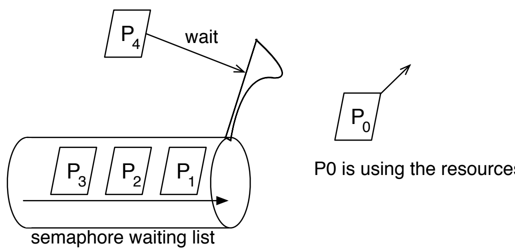

Figure 2-2 The process P4 wants to access the resources: it sends wait to the semaphore.

Here are the steps illustrating a typical scenario illustrated by the following diagram.

1. The semaphore protects a resources: P0 is using the resources. Processes P1, P2, P3 are waiting for the resources (Fig. 2-1). They are queued in the semaphore waiting list.
2. The process P4 wants to access the resources: it sends wait to the semaphore (Fig. 2-2).
3. P4 is added to the waiting list (Fig. 2-3).
4. P0 has finished to use the resources: it signals it to the semaphore (Fig. 2-4).
5. The semaphore resumes the first waiting process, here P1 (Fig. 2-5).
6. The resumed process, P1, will be scheduled by the scheduler.

Details

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.

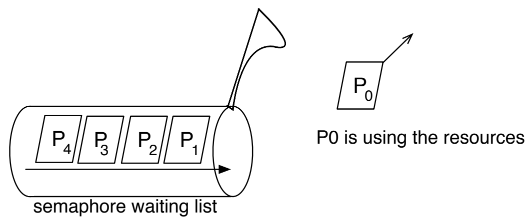

Figure 2-3 P4 is added to the waiting list.

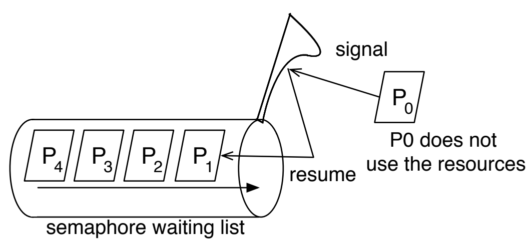

Figure 2-4 P0 has finished to use the resources: it signals it to the semaphore. The semaphore resumes the first pending process.

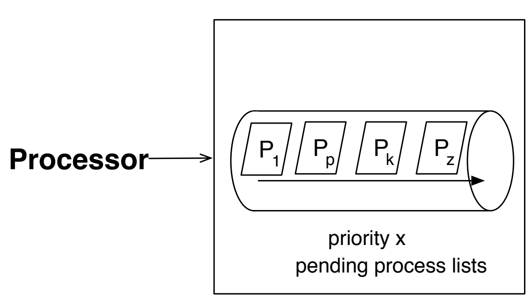

Figure 2-5 The resumed process, P1, is added to the scheduled list of process of the ProcessScheduler.

Unlike the `ProcessorScheduler`, a semaphore does not pay attention to the priority of a process, it dequeues processes in the order in which they waited on the semaphore. The dequeued process is resumed and as such it is added in the waiting list of the scheduler.

2.2 An example

Before continuing let us play with semaphores. Open a transcript and inspect the following piece of code: It schedules two processes and make them both waiting for a semaphore.

```
| semaphore |
semaphore := Semaphore new.

[ "Do a first job ..."
  'Job1 started' crTrace.
  semaphore wait.
  'Job1 finished' crTrace
  ] fork.

[ "Do a second job ..."
  'Job2 started' crTrace.
  semaphore wait.
  'Job2 finished' crTrace
  ] fork.
semaphore inspect
```

You should see in the transcript the following:

```
'Job1 started'
'Job2 started'
```

What you see is that the two processes stopped. They did not finish their job. When a semaphore receives a `wait` message, it will stop the process sending the message and add the process to its pending list.

Now in the inspector on the semaphore execute `self signal`. This schedules one of the waiting process and one of the job will finish its task. If we do not send a new signal message to the semaphore, the second waiting process will never be scheduled.

2.3 wait and signal interplay

The following example schedule three processes. It shows that thread can wait, do some action, signal that they are done that other threads in reaction can get scheduled.

```
| semaphore |
semaphore := Semaphore new.
[ 'Pharo ' crTrace ] fork.

['is ' crTrace .
semaphore wait.
'super ' crTrace.
semaphore signal] fork.

['really ' crTrace.
semaphore signal.
semaphore wait.
'cool!' crTrace ] fork
```

You should obtain `Pharo is really super cool!`

Let us describe what is happening.

- The first one prints `'Pharo'`.
- The second one prints `'is '` and waits.
- The third one prints `'really '` and signal the semaphore and waits. It is added to the waiting list after the second process.
- Since the third process signaled the semaphore, the first waiting process (the second one) is scheduled and prints `'super '` and signals the semaphore.
- The third process is scheduled and prints: `'cool!'`

2.4 Prearmed semaphore

A process wanted a resource protected by a semaphore does not have to be systematically put on the waiting list. There are situations where if it would be the case, the system would be blocked forever because no process could signal the semaphore: an no pending process would be resumed.

A semaphore can be rearmed: it can be signalled (receives `signal` messages) before receiving `wait` messages. In such a case, a process requesting to access the resources will just proceed and be scheduled without first being queued to the waiting list.

A semaphore holds a counter of signals that it receives but did not lead to a process execution. It will not block the process sending a `wait` message if it has got `signal` messages that did not led to scheduling a waiting process.

Example

Let us modify slightly the previous example. We send a `signal` message to the semaphore prior to creating the processes.

```
| semaphore |
semaphore := Semaphore new.
semaphore signal.
[ "Do a first job ..."
  'Job1 started' crTrace.
  semaphore wait.
  'Job1 finished' crTrace
  ] fork.

[ "Do a second job ..."
  'Job2 started' crTrace.
  semaphore wait.
  'Job2 finished' crTrace
  ] fork.
semaphore inspect
```

What you see here is that one of the waiting process is proceed.

```
'Job1 started'
'Job1 finished'
'Job2 started'
```

This example illustrates that signalling a semaphore does not have to be done after a `wait`.

This is important to make sure that on certain concurrency synchronisation, all the processes are waiting, while the first one could do its task and send a signal to schedule another ones.

We can ask a semaphore whether if it is prearmed using the message `isSignaled`.

```
sema := Semapharo new.
sema signal.
sema isSignaled
>>> true
```

2.5 forMutualExclusion

Sometimes we need to ensure that a section of code is only executed when no other process is executing it. We want to make sure that only one process at a time executes the code. This is call a critical section.

For this the class `Semaphore` offers the message `critical: aBlock`. It evaluates aBlock only if the receiver is not currently in the process of running the `critical:` message. If the receiver is, evaluate aBlock after the other `critical:` message is finished. To use a critical, first the semaphore should be prearmed using the class creation message `forMutualExclusion`

Example.

```
Here I need a simple race condition
```

```
Here I need a simple mutual exclusion
```

Deadlocking semaphores.

Pay attention that a semaphore critical section cannot be nested. A semaphore gets blocked (waiting) when being called from a critical section its protects.

```
| deadlockSem |
deadlockSem := Semaphore new.
deadlockSem critical: [ deadlockSem critical: [ 'Nested passes!'
    crTrace] ]
```

2.6 Implementation

A semaphore keeps a number of excess signals: the amount of signals that did not led to schedule a waiting process. If the number of waiting process on a semaphore is smaller than the number allowed to wait, sending a `wait` message is not blocking and the process can continue its operations. On the contrary, the process is stored at the end of the pending list and we will scheduled when the previously pending process will be executed.

Here is the implementation of `signal` and `wait` in Pharo.

The `signal` method shows that If there is no waiting process, the excess signal is increased, else when there are waiting processes, the first one is scheduled.

```
Semaphore >> signal
  "Primitive. Send a signal through the receiver. If one or more
    processes
  have been suspended trying to receive a signal, allow the first
    one to
  proceed. If no process is waiting, remember the excess signal."

  <primitive: 85>
  self primitiveFailed

  "self isEmpty
    ifTrue: [excessSignals := excessSignals+1]
    ifFalse: [Processor resume: self removeFirstLink]"
```

The `wait` method shows that when a semaphore has some signals on excess, waiting is not blocking, it just decreases the number of signals on excess. On the contrary, when there is no signals on excess, then the process is suspended.

```
Semaphore >> wait
  "Primitive. The active Process must receive a signal through the
    receiver
  before proceeding. If no signal has been sent, the active Process
    will be
  suspended until one is sent."

  <primitive: 86>
  self primitiveFailed

  "excessSignals>0
    ifTrue: [excessSignals := excessSignals - 1]
    ifFalse: [self addLastLink: Processor activeProcess suspend]"
```

2.7 Conclusion

Semaphores are the lower synchronisation mechanisms. Pharo offers other abstractions to synchronize such as Mutexes (also named recursion lock), Monitors, shared queue, ...

CHAPTER 3

Scheduler's principles

A process (thread) notifies a semaphore that it frees a reoor Un processus signifie à un sémaphore qu'il libère la ressource en lui envoyant le message signal. Le sémaphore envoie alors un message resume au premier processus en attente de sa file d'attente, ce qui rend ce dernier activable, c'est-à-dire qu'il est ajouté dans la file d'attente du ProcessorScheduler correspondant à sa priorité (voir figure 9-2 ci-après).

Now we can revisit the different states of a process by looking its interaction with the process scheduler and semaphores as shown in 3-2.

- active: it is currently executed.
- activable: it is one of the waiting queue of the scheduler.
- waiting: it is suspendedn on a semaphore. It is the waiting list of a semaphore and it is not yet activable.
- suspended: if this is the active process it is interrupted and can be reactivated later, else it is removed from the queue of the activable process that it belongs to.

3.1 Delay

In case you need to pause execution for some time, you can use **Delay**.

Delays can be instantiated and set up by sending `forSeconds:` or `forMilliseconds:` to the class `Delay` and executed by sending it `wait` message.

For example:

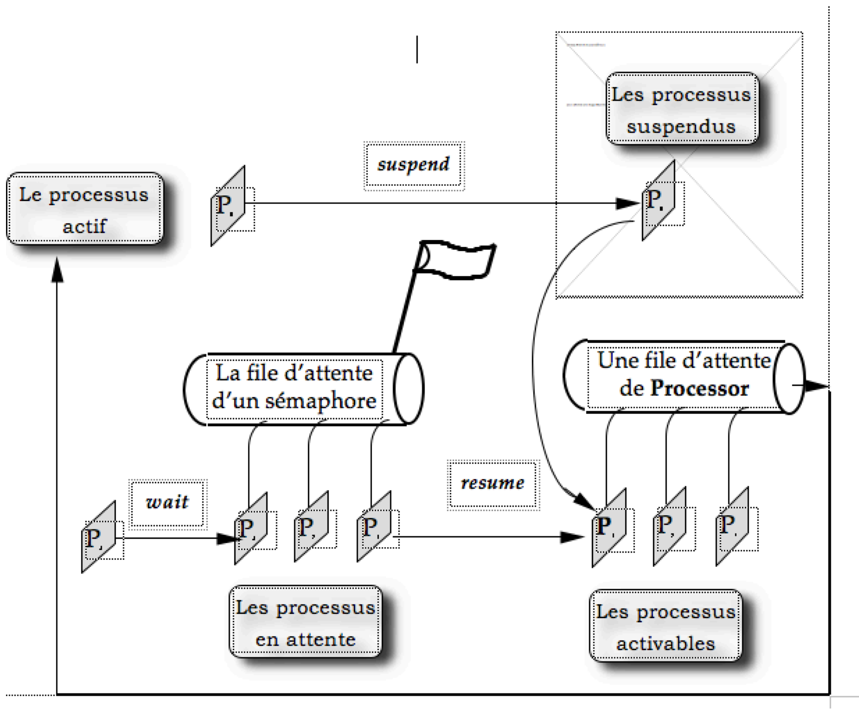

Figure 3-1 BBB

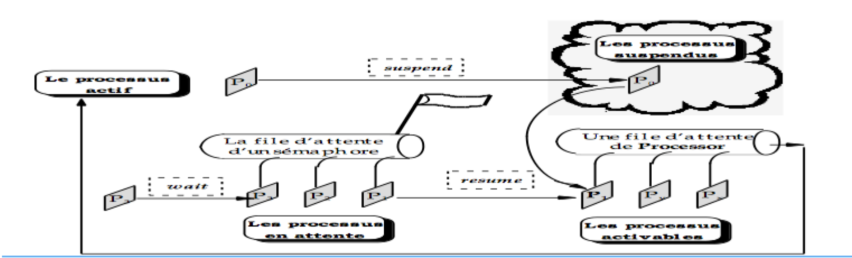

Figure 9-3. Différents états des processus et les messages qui font passer de l'un à l'autre

Figure 3-2 Revisiting the process states

```
| delay |
delay := Delay forSeconds: 3.
[ 1 to: 10 do: [:i |
  Transcript show: i printString ; cr.
  delay wait ] ] fork
```

will print a number each 3 seconds.

Delays suspend the execution of a thread during a precise duration. The thread is then in suspended state.

Typical needs for delas are

- repeat an action every x milliseconds.
- wait a given amount of time before executing an action.

3.2 Example

```
[
| betweenPing |
betweenPing := Delay forMilliseconds: 300.
10 timesRepeat: [
   'ping' crTrace.
   betweenPing wait]
   ] forkAt: Processor userBackgroundPriority.

[
| betweenPong |
betweenPong := Delay forMilliseconds: 100.
10 timesRepeat: [
   'PONG' crTrace.
   betweenPong wait]
   ] forkAt: Processor userBackgroundPriority.
```

CHAPTER 4

Synchronisations

When multiple threads share and modify the same resources we can easily end up in broken state.

4.1 Motivation

Let us imagine that two threads are accessing an account to redraw money. When the threads are not synchronised you may end up to the following situation that one thread access information while the other thread is actually modifying.

Here we see that we redraw 1000 and 200 but since the thread B reads before the other thread finished to commit its changes, we got desynchronised.

Thread A: account debit: 1000	Thread B: account debit: 200
Reading: account value -> 3000	
account debit: 1000	Reading: account value = 3000
account value -> 2000	account debit: 200
	account value -> 2800

The solution is to make sure that a thread cannot access a resources while another one is modifying it. Basically we want that all the threads sharing a resources are mutually exclusive.

When several access a shared resources, only one gets the resources, the other threads got suspended, waiting for the first thread to have finished and release the resources.

4.2 Using a semaphore

As we already saw, we can use a semaphore to control the execution of several threads.

Here we want to make sure that we can do 10 debit and 10 deposit of the same amount and that we get the same amount at the end.

```
| lock counter |
lock := Semaphore new.
counter := 3000.
[ 10 timesRepeat: [
  lock wait.
  counter := counter + 100.
  counter crTrace.
  lock signal ]
  ] fork.

[ 10 timesRepeat: [
  counter := counter - 100.
  counter crTrace.
  lock signal.
  lock wait ]
  ] fork
```

```
2900
3000
2900
3000
```

Notice the pattern, the thread are not symmetrical. The first one will first wait that the resources is accessible and perform his work and signals that he finished. The second one will work and signal and wait to perform the next iteration.

The same problem can be solved in a more robust wait using Mutex and critical sections as we see present in the following section.

4.3 Using a Mutex

A Mutex (MUTual EXclusion) is an object to protect a share resources. A mutex can be used when two or more processes need to access a shared resource concurrently. A Mutex grants ownership to a single process and will suspend any other process trying to aquire the mutex while in use. Waiting processes are granted access to the mutex in the order the access was requested. An instance of the class `Mutex` will make sure that only one thread of control can be executed simultaneously on a given portion of code using the message `critical:`.

In the following example the expressions `Processor yield` ensures that thread of the same priority can get a chance to be executed.

```
| lock counter |
lock := Mutex new.
counter := 3000.
[10 timesRepeat: [
  Processor yield.
  lock critical: [ counter := counter + 100.
            counter crTrace ] ]
  ] fork.

[10 timesRepeat: [
  Processor yield.
  lock critical: [ counter := counter - 100.
          counter crTrace ] ]
  ] fork
```

```
3100
3000
3100
3000
```

Nested critical sections.

In addition a Mutex is also more robust to nested critical calls than a semaphore. For example the following snippet will not deadlock, while a semaphore will. This is why a mutex is also called a recursionLock.

```
|mutex|
mutex := Mutex new.
mutex critical: [ mutex critical: [ 'Nested passes!' crTrace] ]
```

The same code gets blocked on a deadlock with a semaphore.

4.4 About Rendez-vous

As we saw, using `wait` and `signal` we can make sure that two programs running in separate threads can be executed one after the other in order.

The following example is freely inspired from ”The little book of semaphores book. Imagine that we want to have one process reading from file and another process displaying the read contents. Obviously we would like to ensure that the reading happens before the display. We can enforce such order by using `signal` and `wait` as following

```
| readingIsDone read file |
file := FileSystem workingDirectory / 'oneLineBuffer'.
file writeStreamDo: [ :s| s << 'Pharo is cool' ; cr ].
readingIsDone := Semaphore new.
[
'Reading line' crTrace.
read := file readStream upTo: Character cr.
readingIsDone signal.
] fork.
[
readingIsDone wait.
'Displaying line' crTrace.
read crTrace.
] fork.
```

Here is the output

```
'Reading line'
'Displaying line'
'Pharo is cool'
```

Rendez-vous

Now a question is how can be generalize such a behavior so that we can have two programs that work freely to a point where a part of the other has been performed.

For example imagine that we have two prisoners that to escape have to pass a barrier together (their order is irrelevant but they should do it consecutively) and that before that they have to run to the barrier.

The following output is not permitted.

```
'a running to the barrier'
'a jumping over the barrier'
'b running to the barrier'
'b jumping over the barrier'
```

```
'b running to the barrier'
'b jumping over the barrier'
'a running to the barrier'
'a jumping over the barrier'
```

The following cases are permitted.

```
'a running to the barrier'
'b running to the barrier'
'b jumping over the barrier'
'a jumping over the barrier'
```

```
'a running to the barrier'
'b running to the barrier'
'a jumping over the barrier'
'b jumping over the barrier'
```

```
'b running to the barrier'
'a running to the barrier'
'b jumping over the barrier'
'a jumping over the barrier'
```

```
'b running to the barrier'
'a running to the barrier'
'a jumping over the barrier'
'b jumping over the barrier'
```

Here is a code without any synchronisation. We randomly shuffle an array with two blocks and execute them. It produces the non permitted output.

```
{
['a running to the barrier' crTrace.
'a jumping over the barrier' crTrace ]
.
[ 'b running to the barrier' crTrace.
'b jumping over the barrier' crTrace ]
} shuffled do: [ :each | each fork ]
```

Here is a possible solution using two semaphores.

```
| aAtBarrier bAtBarrier |
aAtBarrier := Semaphore new.
bAtBarrier := Semaphore new.
{[ 'a running to the barrier' crTrace.
aAtBarrier signal.
bAtBarrier wait.
'a jumping over the barrier' crTrace ]
.
[ 'b running to the barrier' crTrace.
bAtBarrier signal.
aAtBarrier wait.
'b jumping over the barrier' crTrace ]
} shuffled do: [ :each | each fork ]
```

4.5 Shared Queue

Instances de la classe SharedQueue – Canaux d’Çchange d’objets • Analogues aux sockets/streams ou pipes unix – Mais, synchronisÇs • 1 Çcrivain ou 1 lecteur à la fois • Les lecteurs sont bloquÇs quand la file est vide

```
|file|
file := SharedQueue new.
[
  [ | number | "reader 1"
  number := file next.
  number ifNil: [ProcessorScheduler terminateActive].
  number crTrace ] repeat
  ] fork.
```

```
  [ | counter | "reader 2"
  counter := 0.
  [file next ifNil: [ ('total = ', counter asString) crTrace.
    ProcessorScheduler terminateActive].
    counter := counter + 1 ] repeat
  ] fork.
```

[[[[| delayBetweenWrite | delayBetweenWrite := Delay forMilliseconds: 100. 1 to: 10 do: [:numb| delayBetweenWrite wait. file nextPut: numb]. 2 timesRepeat: [file nextPut: nil]] fork.]]]

4.6 Monitor

A monitor provides process synchronization that is more high-level than the one provided by a semaphore. A monitor has the following properties:

1. At any time, only one process can execute code inside a critical section of a monitor.
2. A monitor is reentrant, which means that the active process in a monitor never gets blocked when it enters a (nested) critical section of the same monitor.
3. Inside a critical section, a process can wait for an event that may be coupled to a certain condition. If the condition is not fulfilled, the process leaves the monitor temporarily (to let other processes enter) and waits until another process signals the event. Then, the original process checks the condition again (this is often necessary because the state of the monitor could have changed in the meantime) and continues if it is fulfilled.
4. The monitor is fair, which means that the process that is waiting on a signaled condition the longest gets activated first.
5. The monitor allows you to define timeouts after which a process gets activated automatically.

4.7 Basic usage:

Monitor»critical: aBlock Critical section. Executes aBlock as a critical section. At any time, only one process can execute code in a critical section. NOTE: All the following synchronization operations are only valid inside the critical section of the monitor!

Monitor»wait Unconditional waiting for the default event. The current process gets blocked and leaves the monitor, which means that the monitor allows another process to execute critical code. When the default event is signaled, the original process is resumed.

Monitor»waitWhile: aBlock Conditional waiting for the default event. The current process gets blocked and leaves the monitor only if the argument block evaluates to true. This means that another process can enter the monitor. When the default event is signaled, the original process is resumed, which means that the condition (argument block) is checked again. Only if it evaluates to false, does execution proceed. Otherwise, the process gets blocked and leaves the monitor again...

Monitor»waitUntil: aBlock Conditional waiting for the default event. See Monitor»waitWhile: aBlock.

Monitor»signal One process waiting for the default event is woken up.

Monitor»signalAll All processes waiting for the default event are woken up.

4.8 We need one example here

CHAPTER 5

Mutex implementation

A Mutex is a semaphore with a little more information: the current process running held in the `owner` instance variable.

```
Object subclass: #Mutex
  instanceVariableNames: 'semaphore owner'
  classVariableNames: ''
  package: 'Kernel-Processes'
```

The `initialize` method makes sure that the semaphore is prearmed for mutual exclusion. Remember it means that the first waiting process will directly proceed and not get added to the waiting list.

```
Mutex >> initialize
  super initialize.
  semaphore := Semaphore forMutualExclusion
```

The key method is the method `critical:`. It checks if the owner of the mutex is the current thread. In such case it executes the protected block. Else it means that

```
Mutex >> critical: aBlock
  "Evaluate aBlock protected by the receiver."

  | activeProcess |
  activeProcess := Processor activeProcess.
  activeProcess == owner ifTrue: [ ^aBlock value ].
  ^ semaphore critical: [
    owner := activeProcess.
    aBlock ensure: [ owner := nil ]].
```

CHAPTER 6

ShareQueue: a Semaphore Example

To do translate stef!

Une SharedQueue, ou file partagée, est une structure FIFO (First In First Out, le premier élément entré est le premier sorti), dotée de sémaphores de protection contre les accès concurrents. Cette structure est utilisée dans les situations où plusieurs processus fonctionnent simultanément et sont susceptibles d'accéder à cette même structure. Sa définition est la suivante :

```
Object subclass: #SharedQueue
  instanceVariableNames: 'contentsArray readPosition writePosition
    accessProtect readSynch '
  package: 'Collections-Sequenceable'
```

`accessProtect` est un sémaphore d'exclusion mutuelle pour l'écriture, tandis que readSync est utilisé pour la synchronisation en lecture. Ces variables sont instanciées par la méthode d'initialisation de la façon suivante :

```
accessProtect := Semaphore forMutualExclusion.
readSynch := Semaphore new
```

Ces deux sémaphores sont utilisés dans les méthodes d'accès et d'ajouts d'éléments (voir figure 6- 5).

```
SharedQueue >> next
  | value |
  readSynch wait.
  accessProtect
    critical: [readPosition = writePosition
        ifTrue: [self error: 'Error in SharedQueue synchronization'.
```

```
            Value := nil]
          ifFalse: [value := contentsArray at: readPosition.
            contentsArray at: readPosition put: nil.
            readPosition := readPosition + 1]].
  ^ value
```

Dans la méthode d'accès, next, le sémaphore de synchronisation en lecture « garde » l'entrée de la méthode (ligne 3). Si un processus envoie le message next alors que la file est vide, il sera suspendu et placé dans la file d'attente du sémaphore readSync par la méthode wait. Seul l'ajout d'un nouvel élément pourra le rendre à nouveau actif. La section critique gérée par le sémaphore accessProtect (lignes 4 à 10) garantit que la portion de code contenue dans le bloc est exécutée sans qu'elle puisse être interrompue par un autre sémaphore, ce qui rendrait l'état de la file inconsistant.

Dans la méthode d'ajout d'un élément, `nextPut:`, la section critique (lignes 3 à 6) protège l'écriture, après laquelle le sémaphore `readSync` est *signalée*, ce qui rendra actif les processus en attente de données.

```
SharedQueue >> nextPut: value
  accessProtect
    critical: [ writePosition > contentsArray size
        ifTrue: [self makeRoomAtEnd].
      contentsArray at: writePosition put: value.
      WritePosition := writePosition + 1].
      readSynch signal.
      ^ value
```

6.1 Conclusion

We presented the key elements of basic concurrent programming in Pharo and some implementation details.