Concurrent Programming in Pharo

Stéphane Ducasse, Guillermo Polito

▶ To cite this version:

Stéphane Ducasse, Guillermo Polito. Concurrent Programming in Pharo. pp.52, 2021. hal-03358770

HAL Id: hal-03358770

https://inria.hal.science/hal-03358770

Submitted on 29 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

Concurrent Programming in Pharo

Stéphane Ducasse and Guillermo Polito

September 29, 2021

Copyright 2017 by Stéphane Ducasse and Guillermo Polito.

The contents of this book are protected under the Creative Commons Attribution-NonCommercial-NoDerivs CC BY-NC-ND You are free to:

Share — copy and redistribute the material in any medium or format

The licensor cannot revoke these freedoms as long as you follow the license terms. Under the following conditions:

Attribution. — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial. — You may not use the material for commercial purposes.

NoDerivatives. — If you remix, transform, or build upon the material, you may not distribute the modified material.

No additional restrictions. — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.

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 4
1.4 Process lifetime 5
1.5 Creating processes 6
1.6 First look at ProcessorScheduler 7
1.7 Process priorities 8
1.8 ProcessScheduler rules 9
1.9 Let us trace better what is happening 10
1.10 Yielding the computation 10
1.11 Important API 12
1.12 Conclusion 13

2 Semaphores **15**
2.1 Understanding semaphores 15
2.2 An example 17
2.3 wait and signal interplay 18
2.4 A key question about signal 19
2.5 Prearmed semaphore 20
2.6 Semaphore forMutualExclusion 22
2.7 Deadlocking semaphores 22
2.8 Mutex 23
2.9 Implementation: the language perspective 24
2.10 Implementation: the VM perspective 25
2.11 Conclusion 27

3 Scheduler's principles **29**
3.1 Revisiting the class `Process` 29
3.2 Looking at some core process primitives 30
3.3 Priorities 33
3.4 signal and preemption 33
3.5 Understanding `yield` 35
3.6 `yield` illustrated 36

3.7 Considering UI processes 36
3.8 About the primitive in yield method 39
3.9 About processPreemption settings 40
3.10 Comparing the two semantics 41
3.11 Second example: preempting P1 42
3.12 Conclusion 43

4 Some examples of semaphores at work **45**
4.1 Promise 45
4.2 Illustration 45
4.3 Promise implementation 46
4.4 Implementation 47
4.5 SharedQueue: a nice semaphore example 48
4.6 About Rendez-vous 50
4.7 Conclusion 52

Illustrations

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

2-1 The semaphore protects a resource: P0 is using the resource, P1...2 are waiting for the resource. 16
2-2 The process P4 wants to access the resource: it sends the message `wait` to the semaphore. 16
2-3 P4 is added to the waiting list. 17
2-4 P0 has finished to use the resource: it send the message `signal` it to the semaphore. The semaphore resumes the first pending process. 17
2-5 The resumed process, P1, is added to the scheduled list of process of the ProcessScheduler: it becomes runnable. 17

3-1 Revisiting process (green thread) lifecycle and states. 31
3-2 Sequences of actions caused by two processes yielding the control to the process scheduler. 37

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.

We did several iterations and integrated many feedback and we want to thank all the revieiwes. Still we are interesting in typos, english corrections, potential mistakes or any kind of feedback.

You can simply contact us at stephane.ducasse@inria.fr

18 February 2020.

Thanks DiagProf, Eliot Miranda, Sven Van Caekenberghe, and Ben Coman for their feedback, ideas, suggestions. Than you again. Special thanks to DiagProf for his patience fixing typos. Special thanks to Ben Coman for the great examples.

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 priority-based *preemptive* and *collaborative*. It is *preemptive* because a process with higher priority interrupts (preempts) processes of lower priority. 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 will show how the process scheduler manages the system.

In a subsequent chapter we will present the semaphores in details and revisit scheduler principles then later we will present other abstractions such as 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 own memory and communicate via pipes supporting a strong isolation. In

Pharo, processes are what is usually called a (green) thread or fiber in other languages. They have their own execution flow but share the same memory space and use concurrent abstractions such as 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: the first process from 1 to 10 and the second one form 101 to 110. During each loop step, using the expression `Processor yield`, the current process mentions the scheduler that it can relinquish the CPU to give a chance to other processes with the same priority to get executed. We say that the active process relinquishes its execution.

```
[ 1 to: 10 do: [ :i |
  i trace; trace: ' '.
  Processor yield ] ] fork.

[ 101 to: 110 do: [ :i |
  i trace; trace: ' '.
  Processor yield ] ] fork
```

The output is the following one:

```
1 101 2 102 3 103 4 104 5 105 6 106 7 107 8 108 9 109 10 110
```

We see that the two processes run concurrently, each outputting a number at a time and not producing two numbers in a row. We also see that a process has to explicitely give back the execution control to the scheduler using the expression `Processor yield`. We will explain this with more details in the following. Let us look at what a process is.

1.3 Process

In Pharo, a process (green thread) is an object as anything else. A process is an instance of the class `Process`. Pharo follows the Smalltalk naming and from a terminology point of view, this class should be called a `Thread` as in other languages. It may change in the future.

A process is characterized by three information:

- A process has a priority (between 10 lowest and 80 highest). Using this priority, a process will preempt other processes having lower priority and it will be managed by the process scheduler in the group of process with the same priority as shown in Figure 1-2.

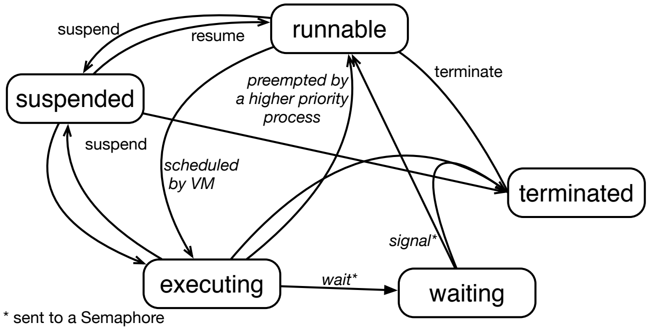
runnable
suspended
terminated
executing
waiting
suspend
resume
terminate
preempted by a higher priority process
suspend
scheduled by VM
signal*
wait*
* sent to a Semaphore

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

- when suspended, a process has a suspendedContext which is a stack reification of the moment of the suspension.
- when runnable, a process refers to one of the scheduler priority lists corresponding to the process' priority. Such a list is also called the run queue to which the process belongs.

1.4 Process lifetime

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

We define the states in which a process can be:

- **executing** - the process is currently executing.
- **runnable** - the process is scheduled. This process is in one of the priority lists of the scheduler.
- **terminated** - the process has run and finished its execution. It is not managed anymore by the scheduler. It cannot be executed anymore.
- **suspended** - the process is not managed by the scheduler: This process is not in one of the scheduler lists or in a semaphore list. The process can become runnable sending it the `resume` message.
- **waiting** - the process is waiting on a semaphore waiting list. It is not managed by the scheduler. The process can become runnable when the semaphore releases it.

We will use systematically this vocabulary in the rest of the book.

1.5 Creating processes

Let us now write some code snippets.

Creating a process without scheduling it

We create a process by sending the message `newProcess` to a block defining the computation that should be carried in such process. This process is not immediately scheduled, it is **suspended**. Then later on, we can schedule the process by sending it the message `resume`: the process will become **runnable**.

The following creates a process in a `suspended` state, it is not added to the list of the scheduled processes of the process scheduler.

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

To be executed, this process should be scheduled and added to the list of suspended processes managed by the process scheduler. This is simply done by sending it the message `resume`.

In the inspector opened by the previous expression, you can execute `self resume` and then the process will be scheduled. It means that it will be added to the priority list corresponding to the process priority of the process scheduler and that the process scheduler will eventually schedule it.

```
self resume
```

Note that by default the priority of a process created using the message `newProcess` is the active priority: the priority of the active process.

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 crTrace ] ] newProcessWith: #(20).
pr resume
```

The arguments are passed to the corresponding block parameters. It means that in the snippet above, `max` will be bound to `20`.

Suspending and terminating a process

A process can also be temporarily suspended (i.e., stopped from executing) using the message `suspend`. A suspended processed can be rescheduled using the message `resume` that we saw previously. We can also terminate a process using the message `terminate`. A terminated process cannot be scheduled anymore. The process scheduler terminates the process once its execution is done.

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

Creating and scheduling in one go

We can also create and schedule a process using an helper method named: `fork`. It is basically sending the `newProcess` message and a `resume` message to the created process.

```
[ 1 to: 10 do: [ :i | i trace ] ] fork
```

This expression creates an instance of the class `Process` whose priority is the one of the calling process (the active process). The created process is `runnable`. It will be executed when the process scheduler schedules it as the current running process and gives it the flow of control. At this moment the block of this process will be executed.

Here is the definition of the method `fork`.

```
BlockClosure >> fork
  "Create and schedule a Process running the code in the receiver."

  ^ self newProcess resume
```

Creating a waiting process

As you see on Figure 1-1 a process can be in a waiting state. It means that the process is blocked waiting for a change to happen (usually waiting for a semaphore to be signalled). This happens when you need to synchronize concurrent processes. The basic synchronization mechanism is a semaphore and we will cover this deeply in subsequent chapters.

1.6 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.

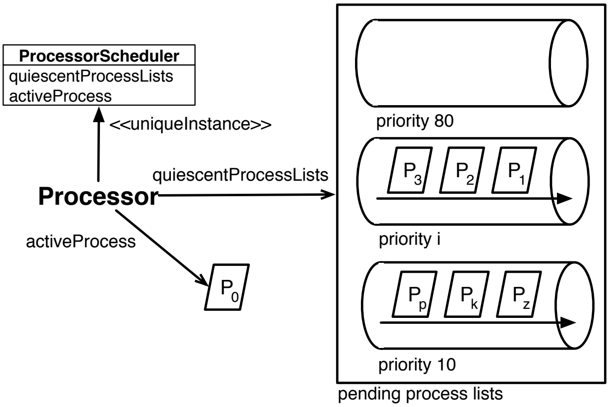
ProcessorScheduler
quiescentProcessLists
activeProcess
<<uniqueInstance>>
Processor
quiescentProcessLists
activeProcess
P_0
priority 80
P_3
P_2
P_1
priority i
P_p
P_k
P_z
priority 10
pending process lists

Figure 1-2 The scheduler knows the currently active process as well as the lists of runnable processes based on their priority.

The scheduler maintains *priority lists*, also called *run queues*, of pending processes as well as the currently active process (See Figure 1-2). To get the running process, you can execute: `Processor activeProcess`. Each time a process is created and scheduled it is added at the end of the run queue corresponding to its priority. The scheduler will take the first process and executes it until a process of higher priority interrupts it or the process gives back control to the processor using the message `yield`.

1.7 Process priorities

At any time only one process is executing. First of all, the processes are being run according to their priority. This priority can be given to a process with the `priority:` message, or `forkAt:` message sent to a block. By default, the priority of a newly created process is the one of the active process. 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 trace ] ]
  forkAt: Processor userBackgroundPriority
```

The scheduler has process priorities from 10 to 80. Only some of these are named. The programmer is free to use any priority within that range that they see fit. The following table lists all the predefined priorities together with their numerical value and purpose.

Priority	Name or selector
80	timingPriority For processes that are dependent on real time. For example, Delays (see later).
70	highIOPriority The priority at which the most time critical input/output processes should run. An example is the process handling input from a network.
60	lowIOPriority The priority at which most input/output processes should run. Examples are the process handling input from the user (keyboard, pointing device, etc.) and the process distributing input from a network.
50	userInterruptPriority For user processes desiring immediate service. Processes run at this level will preempt the ui process and should, therefore, not consume the Processor forever.
40	userSchedulingPriority For processes governing normal user interaction. The priority at which the ui process runs.
30	userBackgroundPriority For user background processes.
20	systemBackgroundPriority For system background processes. Examples are an optimizing compiler or status checker.
10	lowestPriority The lowest possible priority.

Here is an example showing that how to use such named priorities.

```
[3 timesRepeat: [3 trace. ' ' trace ]] forkAt: Processor
    userBackgroundPriority.
[3 timesRepeat: [2 trace. ' ' trace ]] forkAt: Processor
    userBackgroundPriority + 1.
```

1.8 ProcessScheduler rules

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 1-2. It uses the priority lists to manage processes that are runnable in the first-in first-out way.

There are simple rules that manage process scheduling. We will refine the rules a bit later:

- Processes with higher priority preempt (interrupt) lower priority processes, if they have to be executed.

- Assuming an ideal world where processes could execute in one shot, processes with the same priority are executed in the same order they were added to scheduled process list. (See below for a better explanation).

Here is an example showing that process execution is ordered based on priority.

```
[3 timesRepeat: [3 trace. ' ' trace ]] forkAt: 12.
[3 timesRepeat: [2 trace. ' ' trace ]] forkAt: 13.
[3 timesRepeat: [1 trace. ' ' trace ]] forkAt: 14.
```

The execution outputs:

```
1 1 1 2 2 2 3 3 3
```

It shows that the the process of priority 14 is executed prior to the one of priority 13.

1.9 Let us trace better what is happening

Let us define a little trace that will show the current process executing code. It will help understanding what is happening. Now when we execute again the snippet above slightly modified:

```
  | trace |
  trace := [ :message | ('@{1} {2}' format: { Processor
    activePriority. message }) crTrace ].
  [3 timesRepeat: [ trace value: 3 ]] forkAt: 12.
  [3 timesRepeat: [ trace value: 2 ]] forkAt: 13.
  [3 timesRepeat: [ trace value: 1 ]] forkAt: 14.
```

We get the following output, which display the priority of the executing process.

```
@14 1
@14 1
@14 1
@13 2
@13 2
@13 2
@12 3
@12 3
@12 3
```

1.10 Yielding the computation

Now we should see how a process relinquishes its execution and lets other processes of the same priority perform their tasks. Let us start with a small example based on the previous example.

```
| trace |
trace := [ :message | ('@{1} {2}' format: { Processor
  activePriority. message }) crTrace ].
[3 timesRepeat: [ trace value: 3. Processor yield ]] forkAt: 12.
[3 timesRepeat: [ trace value: 2. Processor yield ]] forkAt: 13.
[3 timesRepeat: [ trace value: 1. Processor yield ]] forkAt: 14.
```

Here the result is the same.

```
@14 1
@14 1
@14 1
@13 2
@13 2
@13 2
@12 3
@12 3
@12 3
```

What you should see is that the message `yield` was sent, but the scheduler rescheduled the process of the highest priority that did not finish its execution. This example shows that yielding a process will never allow a process of lower priority to run.

Between processes of the same priority

Now we can study what is happening between processes of the same priority. We create two processes of the same priority that perform a loop displaying numbers.

```
| p1 p2 |
p1 := [ 1 to: 10 do: [:i| i trace. ' ' trace ] ] fork.
p2 := [ 11 to: 20 do: [:i| i trace. ' ' trace ] ] fork.
```

We obtain the following output:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

This is normal since the processes have the same priority. They are scheduled and executed one after the other. `p1` executes and displays its output. Then it terminates and `p2` gets the control and executes. It displays its output and get terminated.

During the execution of one of the processes nothing forces it to relinquish computation. Therefore it executes until it finishes. It means that if a process has an endless loop it will not release the execution except if it is preempted by a process of higher priority (see Chapter scheduler's principle).

Using yield

We modify the example to introduce an explicit return of control to the process scheduler.

```
| p1 p2 |
p1 := [ 1 to: 10 do: [:i| i trace. ' ' trace. Processor yield ] ]
    fork.
p2 := [ 11 to: 20 do: [:i| i trace. ' ' trace. Processor yield ] ]
    fork.
```

We obtain the following trace showing that each process gave back the control to the scheduler after each loop step.

```
1 11 2 12 3 13 4 14 5 15 6 16 7 17 8 18 9 19 10 20
```

We will come back to yield in future chapters.

Summary

Let us revisit what we learned in this chapter.

- Processes with the same priority are executed in the same order they were added to scheduled process list. In fact processes within the same priority should collaborate to share the execution amongst themselves. In addition, we should pay attention since a process can be preempted by a process of higher prioriy, the semantics of the preemption (i.e., how the preempted process is rescheduled) has an impact on the process execution order. We will discuss this is in depth in following chapters.
- Processes should explicitly give back the computation to give a chance to other pending processes of the same priority to execute. The same remark as above works here too.
- A process 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).

1.11 Important API

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

- `[ ] newProcess` creates a suspended (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.
- `priority:` defines the priority of a process.

- `[ ] fork` creates a new scheduled process having the same priority than the process that spawns it. 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.
- `ProcessorScheduler yield` releases the execution from the current process and give a chance to processes of same priority to execute.

1.12 Conclusion

We presented the notion of process (green thread) and process scheduler. 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.

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. A writer will wait for the pen and once the pen is free. He will be able to access and use it concurrently. 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 needing to use the pen just register to the pen: they are added at the end of the waiting list. In fact, this 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 solved with Semaphore: *The Little Book of Semaphores*. It is clearly a nice further reading.

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 accessed by a single process at the time. It is also said that the semaphore protects the resource.

A process that wants to access to a resource will declare it to the semaphore protecting the resource 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 it protects. When the process currently using the resource 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.

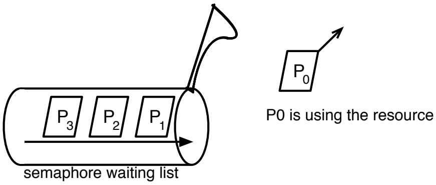

Figure 2-1 The semaphore protects a resource: P0 is using the resource, P1...2 are waiting for the resource.

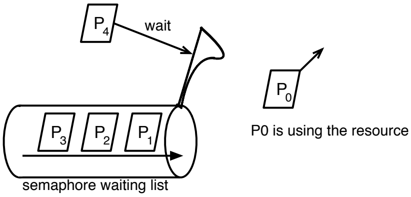

Figure 2-2 The process P4 wants to access the resource: it sends the message `wait` to the semaphore.

Here are the steps illustrating a typical scenario:

1. The semaphore protects a resource: P0 is using the resource. Processes P1, P2, P3 are waiting for the resource (Fig. 2-1). They are queued in the semaphore waiting list.
2. The process P4 wants to access the resource: it sends `wait` to the semaphore (Fig. 2-2).
3. P4 is added to the waiting list (Fig. 2-3) - it passes from the executing to the waiting state.
4. P0 has finished to use the resource: it sends the message `signal` to the semaphore (Fig. 2-4).
5. The semaphore resumes the first waiting process of its pending queue, here P1 (Fig. 2-5).
6. The resumed process, P1, become runnable and 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.

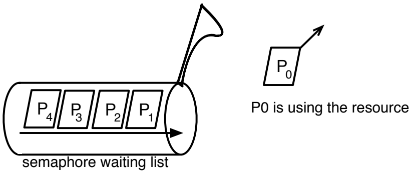
P4
P3
P2
P1
semaphore waiting list
P0
P0 is using the resource

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

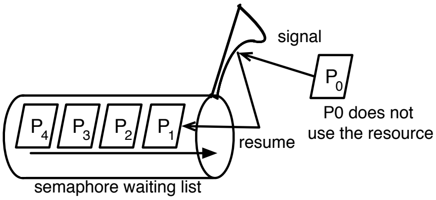
signal
P0
P4
P3
P2
P1
resume
P0 does not use the resource
semaphore waiting list

Figure 2-4 P0 has finished to use the resource: it send the message `signal` it to the semaphore. The semaphore resumes the first pending process.

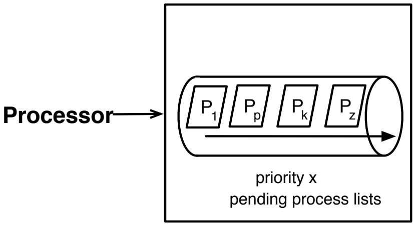
Processor
P1
Pp
Pk
Pz
priority x
pending process lists

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

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.

When a process sends a `wait` message to the semaphore, if the waiting list is empty then the process is directly scheduled.

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 wait on 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 suspends the process sending the message and adds 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

To understand the interplay between wait and signal we propose the following example. Can you guess what is the displayed information?

```
| semaphore p1 p2 p3 |
semaphore := Semaphore new.
p1 := [ ' Pharo ' trace ] forkAt: 30.

p2 := [' is ' trace.
  semaphore wait.
  ' super ' trace.
  semaphore signal.
  ' p2 finished ' trace ] forkAt: 35.

p3 := [' really ' trace.
  semaphore signal.
  ' cool ' trace.
  semaphore wait.
  ' and powerful! ' trace ] forkAt: 33
```

You should obtain `is really super p2 finished cool and powerful! Pharo`

Let us describe what's happened:

- The three processes are created and scheduled.
- The process with the highest priority, `p2`, is executed. It prints `is` and waits on the semaphore.
- `p2` is not runnable anymore.
- The next highest priority process, `p3`, is scheduled, it prints `really` and signals the semaphore.
- The semaphore is signaled so it schedules its waiting process `p2`. Since `p2` has higher priority than `p3`, it is executed.
- `p2` prints `super`, and it signals the semaphore.
- There is no process waiting on the semaphore and in addition, `p2` is the highest priority process so it continues to execute, prints `p2 finished` and terminates.
- `p3` is then resumed and it prints `cool`, then it sends the message `wait` to the semaphore but since there was no waiting process, `p3` continues, prints `and powerful!` and terminates.
- Finally `p1` is executed, prints `Pharo` and terminates.

We really suggest to play with different priorities and predict the behavior. Note that this scenario pays attention that the processes are lower priority that the UI process that is refreshing the display. In addition as we will see later, we do not have processes with the same priority since the preemption may impact the order of execution.

2.4 A key question about signal

Let us imagine that we have the following two processes of different priorities and one semaphore. We would like to show the influence of `signal` on the scheduling of such processes.

```
| trace semaphore p1 p2 |
semaphore := Semaphore new.
trace := [ :message | ('@{1} {2}' format: { Processor
    activePriority. message }) crTrace ].
p1 := [
  trace value: 'Process 1a waits for signal on semaphore'.
  semaphore wait.
  trace value: 'Process 1b received signal and terminates' ] forkAt:
    20.
p2 := [
  trace value: 'Process 2a up to signalling semaphore'.
  semaphore signal.
  trace value: 'Process 2b continues and terminates' ] forkAt: 30.
```

Here the higher priority process (`p2`) produces a trace, signals the semaphore and finishes. Then the lower priority process produces a trace, waits and since the semaphore has been signalled, it executes and terminates.

```
  @30 Process 2a up to signalling semaphore
  @30 Process 2b continues and terminates
  @20 Process 1a waits for signal on semaphore
  @20 Process 1b received signal and terminates
```

Now let us swap the prioriy.

```
| trace semaphore p1 p2 |
semaphore := Semaphore new.
trace := [ :message | ('@{1} {2}' format: { Processor
    activePriority. message }) crTrace ].
p1 := [
  trace value: 'Process 1a waits for signal on semaphore'.
  semaphore wait.
  trace value: 'Process 1b received signal and terminates' ] forkAt:
    30.
p2 := [
  trace value: 'Process 2a up to signalling semaphore'.
  semaphore signal.
  trace value: 'Process 2b continues and terminates' ] forkAt: 20.
```

Here the higher priority process (`p1`) produces trace and waits on the semaphore. `p2` is then executed: it produces a trace, then signals the semaphore. This `signal` message reschedules `p1` and since it is of higher priority, it is executed first preempting (`p2`) and it terminates. Then `p2` terminates.

```
  @30 Process 1a waits for signal on semaphore
  @20 Process 2a up to signalling semaphore
  @30 Process 1b received signal and terminates
  @20 Process 2b continues and terminates
```

There is subtle point that the second example does not illustrate but that is worth that we discuss: while the lowest priority process signaled the semaphore it gets preempted by the higher priority ones. This raises the question of what it the process to be rescheduled after preemption. The example does not show it because we got only one process of priority 20. We will go over this point in the next Chapter.

2.5 Prearmed semaphore

A process wanting a resource protected by a semaphore does not have to be systematically put on the waiting list. There are situations where the system would be blocked forever because no process can signal the semaphore: no pending process would be resumed.

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

As an implementation note, a semaphore holds a counter of the signals that it received but did not lead to a process execution. It will not block a 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. The semaphore is then prearmed.

```
| trace semaphore p1 p2 |
semaphore := Semaphore new.
semaphore signal.
trace := [ :message | ('@{1} {2}' format: { Processor
    activePriority. message }) crTrace ].
p1 := [
   trace value: 'Process 1a waits for signal on semaphore'.
   semaphore wait.
   trace value: 'Process 1b received signal and terminates' ]
    forkAt: 30.
p2 := [
   trace value: 'Process 2a up to signalling semaphore'.
   semaphore signal.
   trace value: 'Process 2b continues and terminates' ] forkAt: 20.
```

The previous example produces the following trace:

```
@30 Process 1a waits for signal on semaphore
@30 Process 1b received signal and terminates
@20 Process 2a up to signalling semaphore
@20 Process 2b continues and terminates
```

This example illustrates that a process does not have to systematically wait on a semaphore.

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 others.

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

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

2.6 Semaphore forMutualExclusion

Sometimes we need to ensure that a section of code is executed by only a single process at a time i.e., no other process will enter it. We want to make sure that only one process at a time executes a section of code. This code section is called a *critical section*.

The class `Semaphore` offers the message `critical: aBlock` to define a critical code section for the block passed as argument. It evaluates aBlock only if the receiver is not currently in the process of running the `critical:` message. If the receiver is currently executed, aBlock will be executed after the other `critical:` message is finished.

To use a critical section, first the semaphore should be prearmed using the class creation message `forMutualExclusion`. It makes sure that the first execution of the critical section will pass without getting blocked (i.e., put on the semaphore waiting list and waiting for a `signal` message).

Here is an example of critical section use: The memory logger makes sure that when several processes work concurrently, the execution of the different processes does not mess the recording of the item addition. Similarly, the logger makes sure that when only one process can reset the recordings.

```
MemoryLogger >> nextPut: aSignal
  mutex critical: [
    recordings add: aSignal ].
  self announcer announce: aSignal
```

```
MemoryLogger >> reset
  mutex critical: [
    recordings := OrderedCollection new ]
```

Now the previous code use a mutex and not a semaphore.

2.7 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] ]
```

Mutexes (also named RecursionLock) solve this problem. This is why a Mutex and a Semaphore are not interchangeable. So let's see what is a Mutex.

2.8 Mutex

A Mutex (MUTual EXclusion) is a semaphore with more information: the current process running held in the `owner` instance variable. As such a Mutex is an object that protects a shared resource. 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:`.

Nested critical sections

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 sometimes 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. A Mutex and a semaphore are not interchangeable from this perspective.

Mutex implementation

```
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, and return. Else it means that the process waits on the critical section and when the semaphore resumes it it sets the process as owner of the section and makes sure that the owner is reset once the critical section is passed through.

```
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 ]]
```

2.9 Implementation: the language perspective

We propose to have a look at the implementation of semaphores. In a first reading you can skip the following sections. We take two perspectives: how the `Semaphore` class is defined within Pharo and later how the virtual machine defines the primitives mandatory for the semaphore implementation. Let us start with the language level definition.

Pharo's implementation.

A semaphore keeps a number of excess signals: the amount of signals that did not lead to schedule a waiting process. The message `wait` and `signal` maintain this information: as the implementations below show it, a `signal` will increase the excess number and a `wait` will decrease it.

If the number of waiting processes on a semaphore is smaller than the number allowed to wait, sending a `wait` message is not blocking and the process continues its execution. On the contrary, the process is stored at the end of the pending list and we will be scheduled when the semaphore will have received enough signals.

The fact that the semaphore waiting list is a linked list has an impact on the semaphore semantics. It makes sure that waiting processes are managed in a first in first out manner.

While conceptually a semaphore has a list and a counter. At the Pharo implementation level, the class `Semaphore` inherits from the class `LinkedList`, so the waiting process list is 'directly' the semaphore itself. Since `Process` inherits from `Link` (elements that can be added to linked list), they can be directly added to the semaphore without being wrapped by an element object. This is a simplification for the virtual machine.

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

Signal implementation.

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 (i.e., the process scheduler resumes the process).

```
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]"
```

Wait implementation.

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 and added to the semaphore waiting list.

```
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]"
```

2.10 Implementation: the VM perspective

Here we look at the virtual machine definition of the primitives. We show for quick reference the StackInterpreter code since it is a little simpler than the JIT version.

As we saw previously two primitives are defined: one for `wait` and one for `signal`.

```
StackInterpreter class >> initializePrimitiveTable
  ...
  "Control Primitives (80-89)"
  (85 primitiveSignal)
  (86 primitiveWait)
  ...
```

We see that the `wait` primitive checks the number of signal of the semaphore. When such number is positive, it is decreased and the process is not suspended. On the contrary, it grabs the active process, adds it to the semaphore list and give back the control to the highest process.

```
InterpreterPrimitives >> primitiveWait
  | sema excessSignals activeProc |
  sema := self stackTop.  "rcvr"
  excessSignals := self fetchInteger: ExcessSignalsIndex ofObject:
    sema.
  excessSignals > 0
    ifTrue:
      [self storeInteger: ExcessSignalsIndex ofObject: sema
    withValue: excessSignals - 1]
    ifFalse:
      [activeProc := self activeProcess.
      self addLastLink: activeProc toList: sema.
      self transferTo: self wakeHighestPriority]
```

```
InterpreterPrimitives >> primitiveSignal [
  "Synchronously signal the semaphore.
  This may change the active process as a result."

  self synchronousSignal: self stackTop  "rcvr"
```

Here if the semaphore list is empty, the signal primitive is incrementing the signal count of the semaphore. Else, the first pending process is resumed.

```
StackInterpreter >> synchronousSignal: aSemaphore
  "Signal the given semaphore from within the interpreter.
  Answer if the current process was preempted."

  | excessSignals |
  (self isEmptyList: aSemaphore) ifTrue:
    ["no process is waiting on this semaphore"
    excessSignals := self fetchInteger: ExcessSignalsIndex ofObject:
    aSemaphore.
    self storeInteger: ExcessSignalsIndex
      ofObject: aSemaphore
      withValue: excessSignals + 1.
    ^false].
```

```
objectMemory ensureSemaphoreUnforwardedThroughContext: aSemaphore.
^ self
  resume: (self removeFirstLinkOfList: aSemaphore)
  preemptedYieldingIf: preemptionYields
```

We will explain the `preemptionYields` used in the last line in a future chapter.

2.11 Conclusion

Semaphore is the lowest level synchronisation mechanism. Pharo offers other abstractions to synchronize such as Mutexes (also named recursion lock), Monitors, shared queues, and atomic queues.

CHAPTER 3

Scheduler’s principles

In this chapter we revisit the way to scheduler works and present some implementation aspects. In particular we show how yield is implemented. The Pharo scheduler is cooperative, preemptive across priorities, non-preemptive within priorities, scheduler. But let us start with the class `Process`.

3.1 Revisiting the class **Process**

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 process scheduler list of processes to which the suspended process belongs to. This list is also called it run queue and it is only for suspended processes.

You can do the following tests to see the state of a process.

The first example opens an inspector in which you can see the state of the executed process.

```
| pr |
pr := [ 1 to: 1000000 do: [ :i | i traceCr ] ] forkAt: 10.
pr inspect
```

It shows that while the process is executing the expression `self suspendingList` is not nil, while that when the process terminates, its suspending list is nil.

The second example shows that the process suspendedContext is nil when a process is executing.

```
Processor activeProcess suspendedContext isNil.
>>> true
```

Now a suspended process suspended context should not be nil, since it should have a stack of the suspended program.

```
([ 1 + 2 ] fork suspend ; suspendedContext) isNotNil
```

Implementation details.

The class `Process` 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 *process* linked list is tailored for the process scheduler logic. This *process* linked list is for internal usage. If you need a linked link, better uses another one if you need one.

States

We saw previously the different states a process can be in. We also saw semaphores which suspend and resume suspended processes. We revisit the different states of a process by looking its interaction with the process scheduler and semaphores as shown in 3-1 :

- **executing** - the process is currently executing.
- **runnable** - the process is scheduled. This process is in one of the priority lists of the scheduler. It may be turned into the executing state by the scheduler.
- **terminated** - the process ran and finished its execution. It is not managed anymore by the scheduler. It cannot be executed anymore.
- **suspended** - the process is not managed by the scheduler: This process is not in one of the scheduler lists or in a semaphore list. The process can become runnable sending it the `resume` message. This state is reached when the process received the message `suspend`.
- **waiting** - the process is waiting on a semaphore waiting list. It is not managed by the scheduler. The process can become runnable when the semaphore releases it.

3.2 Looking at some core process primitives

It is worth to look at the way `Process` key methods are implemented.

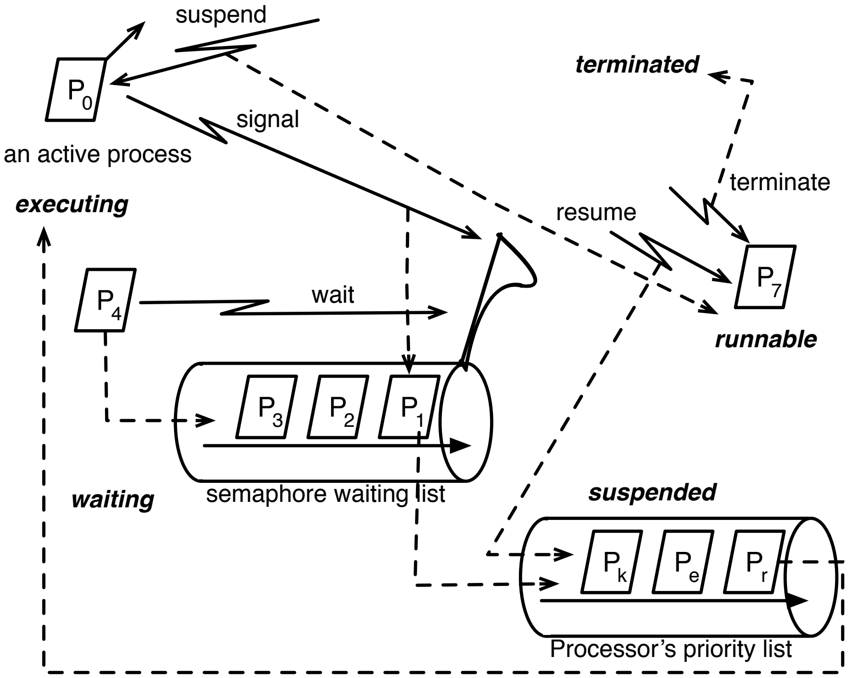
suspend
P_0
an active process
signal
terminated
terminate
executing
resume
P_7
P_4
wait
runnable
P_3
P_2
P_1
waiting
semaphore waiting list
suspended
P_k
P_e
P_r
Processor's priority list

Figure 3-1 Revisiting process (green thread) lifecycle and states.

The method `suspend` is a primitive and implemented at the VM level. Since the process list (`myList`) refers to one of the scheduler priority list in which it is, we see that the message `suspend` effectively remove the process from the scheduler list.

```
Process >> suspend
  "Stop the process that the receiver represents in such a way
  that it can be restarted at a later time (by sending the receiver
    the
  message resume). If the receiver represents the activeProcess,
    suspend it.
  Otherwise remove the receiver from the list of waiting processes.
  The return value of this method is the list the receiver was
    previously on (if any)."

  <primitive: 88>
  | oldList |
  myList ifNil: [ ^ nil ].
  oldList := myList.
  myList := nil.
  oldList remove: self ifAbsent: [ ].
  ^ oldList
```

The `resume` method is defined as follows:

```
Process >> resume
  "Allow the process that the receiver represents to continue. Put
  the receiver in line to become the activeProcess. Check for a nil
  suspendedContext, which indicates a previously terminated Process
    that
  would cause a vm crash if the resume attempt were permitted"

  suspendedContext ifNil: [ ^ self primitiveFailed ].
  ^ self primitiveResume
```

```
Process >> primitiveResume
  "Allow the process that the receiver represents to continue. Put
  the receiver in line to become the activeProcess. Fail if the
    receiver is
  already waiting in a queue (in a Semaphore or ProcessScheduler)."

  <primitive: 87>
  self primitiveFailed
```

Looking at the virtual machine definition shows that the resumed process does not preempt processes having the same priority and that would be executing.

```
InterpreterPrimitives >> primitiveResume
  "Put this process on the scheduler's lists thus allowing it to
    proceed next time there is
   a chance for processes of its priority level. It must go to the
    back of its run queue so
   as not to preempt any already running processes at this level. If
    the process's priority
   is higher than the current process, preempt the current process."
  | proc |
  proc := self stackTop.  "rcvr"
  (objectMemory isContext: (objectMemory fetchPointer:
    SuspendedContextIndex ofObject: proc)) ifFalse:
    [^self primitiveFail].
  self resume: proc preemptedYieldingIf: preemptionYields
```

Now we can have a look at the implementation of `newProcess`. The method `newProcess` creates a process by reifying a new block as a stack representation. The responsibility of this new block is to execute the receiver and terminates the process.

```
BlockClosure >> newProcess
  "Answer a Process running the code in the receiver. The process is
    not
  scheduled."
```

```
<primitive: 19>
^ Process
  forContext:
    [ self value.
    Processor terminateActive ] asContext
  priority: Processor activePriority
```

3.3 Priorities

A runnable process has a priority. It is always executed before a process of an inferior priority. Remember the examples of previous chapters:

```
| trace |
trace := [ :message | ('@{1} {2}' format: { Processor
  activePriority. message }) crTrace ].
[3 timesRepeat: [ trace value: 3. Processor yield ]] forkAt: 12.
[3 timesRepeat: [ trace value: 2. Processor yield ]] forkAt: 13.
[3 timesRepeat: [ trace value: 1. Processor yield ]] forkAt: 14.
```

```
@14 1
@14 1
@14 1
@13 2
@13 2
@13 2
@12 3
@12 3
@12 3
```

This code snippet shows that even if processes relinquish execution (via a message `yield`), the processes of lower priority are not scheduled before the process of higher priority got terminated.

> **Note** In the case of a higher priority level process preempting a process of lower priority, when the preempting process releases the control, the question is then what is the next process to resume: the interrupted one or another one. Currently in Pharo, the interrupted process is put at the end of the waiting queue, while an alternative is to resume the interrupted process to give it a chance to continue its task.

3.4 signal and preemption

In the previous chapter we presented this example:

```
| trace semaphore p1 p2 |
semaphore := Semaphore new.
trace := [ :message | ('@{1} {2}' format: { Processor
  activePriority. message }) crTrace ].
```

```
p1 := [
   trace value: 'Process 1a waits for signal on semaphore'.
   semaphore wait.
   trace value: 'Process 1b received signal and terminates' ]
  forkAt: 30.
p2 := [
   trace value: 'Process 2a up to signalling semaphore'.
   semaphore signal.
   trace value: 'Process 2b continues and terminates' ] forkAt: 20.
```

Here the higher priority process (`p1`) produces trace and waits on the semaphore. `p2` is then executed: it produces a trace, then signals the semaphore. This signal reschedules `p1` and since it is of higher priority, it preempts (`p2`) and it terminates. Then `p2` terminates.

```
@30 Process 1a waits for signal on semaphore
@20 Process 2a up to signalling semaphore
@30 Process 1b received signal and terminates
@20 Process 2b continues and terminates
```

Now we add a second process of lower priority to understand what may happen on preemption.

```
| trace semaphore p1 p2 p3 |
semaphore := Semaphore new.
trace := [ :message | ('@{1} {2}' format: { Processor
  activePriority. message }) crTrace ].
p1 := [
   trace value: 'Process 1a waits for signal on semaphore'.
   semaphore wait.
   trace value: 'Process 1b received signal and terminates' ]
  forkAt: 30.
p2 := [
   trace value: 'Process 2a up to signalling semaphore'.
   semaphore signal.
   trace value: 'Process 2b continues and terminates' ] forkAt: 20.
 p3 := [
   trace value: 'Process 3a works and terminates'. ] forkAt: 20.
```

Here is the produced trace. What is interesting to see is that `p2` is preempted by `p1` as soon as it is signalling the semaphore. Then `p1` terminates and the scheduler does not schedule `p2` but `p3`.

```
@30 Process 1a waits for signal on semaphore
@20 Process 2a up to signalling semaphore
@30 Process 1b received signal and terminates
@20 Process 3a works and terminates
@20 Process 2b continues and terminates
```

This behavior can be surprising. In fact the Pharo virtual machine offers two possibilities as we will show later. In one, when a preempting process terminates, the preempted process is managed as if an implicit yield happened, moving the preempted process to the end of its run queue on preemption return and scheduling the following pending process. In another one, when a preempting process terminates, the preempted process is the one that get scheduled (it does not move at the end of the pending list). By default, Pharo uses the first semantics.

3.5 Understanding `yield`

We mentioned in the first chapter, Pharo's concurrency model is preemptive between processes of different priorities and collaborative among processes of the same priority. We detail how the collaboration occurs: a process has to explicitly give back its execution. As we show in the previous chapter, it does it by sending the message `yield` to the scheduler.

Now let us study the implementation of the method `yield` itself. It is really elegant. It creates a process whose execution will signal a semaphore and the current process will wait on such a semaphore until the created process is scheduled by the processor. Since

```
ProcessScheduler >> yield
  "Give other Processes at the current priority a chance to run."

  | semaphore |
  semaphore := Semaphore new.
  [ semaphore signal ] fork.
  semaphore wait
```

Note that this implementation implies that

The `yield` method does the following:

1. The fork creates a new process. It adds it to the end of the active process's run queue (because fork create a process whose priority is the same as the active process).
2. The message `wait` in `semaphore wait` removes the active process from its run queue and adds it to the semaphore list of waiting processes, so the active process is now not runnable anymore but waiting on the semaphore.
3. This allows the next process in the run queue to run, and eventually
4. allows the newly forked process to run, and
5. the signal in `semaphore signal` removes the process from the semaphore and adds it to the back of the run queue, so
6. all processes at the same priority level as the process that called `yield` have run.

3.6 `yield` illustrated

`yield` only facilitates other processes having the same priority getting a chance to run. It doesn't put the current process to sleep, it just moves the process to the back of its priority run queue. It gets to run again before any lower priority process gets a chance to run. Yielding will never allow a lower priority process to run.

Figure 3-2 illustrates the execution of the two following processes yielding their computation.

```
P1 := [1 to: 10 do: [:i| i trace. ' ' trace. Processor yield ]] fork.
P2 := [11 to: 20 do: [:i| i trace. ' ' trace. Processor yield ]]
    fork.
```

Here is the output

```
1 11 2 12 3 13 4 14 5 15 6 16 7 17 8 18 9 19 10 20
```

Here are the steps:

1. Processes P1 and P2 are scheduled and in the list (run queue) of the processor.
2. P1 becomes first active, it writes 1 and sends the message `yield`.
3. The execution of `yield` in P1 creates a Semaphore S1, a new process Py1 is added to the processor list after P2. P1 is added to S1's waiting list.
4. P2 is active, it writes 11 and sends the message `yield`.
5. The execution of `yield` in P2 creates a Semaphore S2, a new process Py2 is added to the processor list after Py1. P2 is added to S2's waiting list.
6. Py1 is active. S1 is signalled. Py1 finishes and is terminated.
7. P1 is scheduled. It moves from semaphore pending list to processor list after Py2.
8. Py2 is active. S2 is signalled. Py2 finishes and is terminated.

3.7 Considering UI processes

We saw that the message `signal` does not transfer execution unless the waiting process that received the signal has a higher priority. It just makes the waiting process runnable, and the highest priority runnable process is the one that is run. This respects the preemption semantics between processes of different priorities.

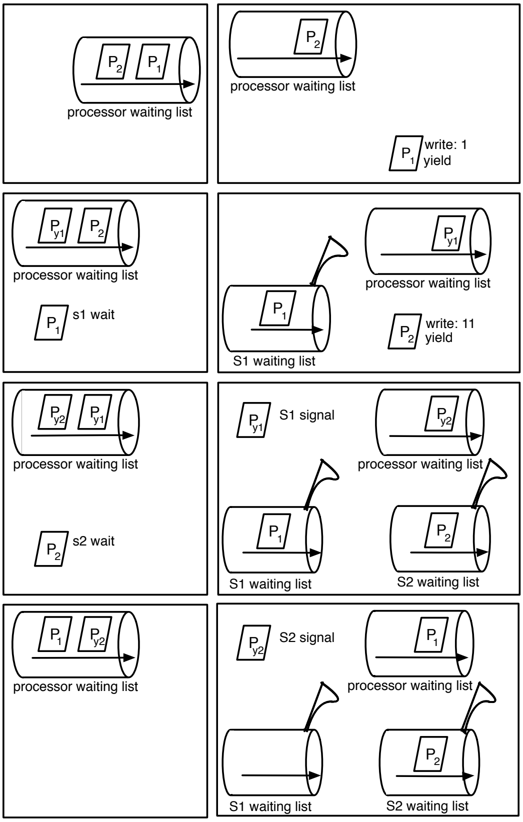
P2
P1
processor waiting list
P2
processor waiting list
P1
write: 1
yield
Py1
P2
processor waiting list
P1
s1 wait
Py1
processor waiting list
P1
S1 waiting list
P2
write: 11
yield
Py2
Py1
processor waiting list
P2
s2 wait
Py1
S1 signal
Py2
processor waiting list
P1
S1 waiting list
P2
S2 waiting list
P1
Py2
processor waiting list
Py2
S2 signal
P1
processor waiting list
S1 waiting list
P2
S2 waiting list

Figure 3-2 Sequences of actions caused by two processes yielding the control to the process scheduler.

The following code snippet returns false since the forked process got the priority than the current process and the current process continued its execution until the end. Therefore the `yielded` did not get a chance to be modified.

```
| yielded |
yielded := false.
[ yielded := true ] fork.
yielded
>>> false
```

Now Let us image that

```
| yielded |
yielded := false.
[ yielded := true ] fork.
Processor yield.
```

```
yielded
>>> true
```

This expression returns true because `fork` creates a process with the same priority and the `Processor yield` expression allows the forked process to execute.

Now let us change the priority of the forked process to be lower than the active one (here the active one is the UI process). The current process yields the computation but since the forked process is of lower priority, the current process will be executed before the forked one.

```
| yielded |
yielded := false.
p := [ yielded := true ] forkAt: Processor activeProcess priority -
    1.
Processor yield.
yielded
>>> false
```

The following illustrates this point using the UI process. Indeed when you execute interactively a code snippet, the execution happens in the UI process (also called UI thread) with a priority of 40.

```
  | trace semaphore p1 p2 |
  semaphore := Semaphore new.
  trace := [ :message | ('@{1} {2}' format: { Processor
    activePriority. message }) traceCr ].
  p1 := [
    trace value: 'Process 1a waits for signal on semaphore'.
    semaphore wait.
    trace value: 'Process 1b received signal and terminates' ]
    forkAt: 30.
  p2 := [
    trace value: 'Process 2a signals semaphore'.
    semaphore signal.
    trace value: 'Process 2b continues and terminates' ] forkAt: 20.
  trace value: 'Original process pre-yield'.
  Processor yield.
  trace value: 'Original process post-yield'.
```

The following traces shows that `Processor yield` does not change the execution of higher priority processes. Here the UI thread is executed prior to the other and yielding does not execute processes of lower priorities.

```
  @40 Original process pre-yield
  @40 Original process post-yield
  @30 Process 1a waits for signal on semaphore
  @20 Process 2a signals semaphore
  @30 Process 1b received signal and terminates
  @20 Process 2b continues and terminates
```

Now if we make the UI thread waiting for small enough time (but long enough that the other processes get executed), then the other processes are run since the UI process is not runnable but waiting.

```
| trace semaphore p1 p2 |
semaphore := Semaphore new.
trace := [ :message | ('@{1} {2}' format: { Processor
  activePriority. message }) traceCr ].
p1 := [
  trace value: 'Process 1a waits for signal on semaphore'.
  semaphore wait.
  trace value: 'Process 1b received signal' ] forkAt: 30.
p2 := [
  trace value: 'Process 2a signals semaphore'.
  semaphore signal.
  trace value: 'Process 2b continues' ] forkAt: 20.

trace value: 'Original process pre-delay'.
1 milliSecond wait.
trace value: 'Original process post-delay'.
```

```
@40 Original process pre-delay
@30 Process 1a waits for signal on semaphore
@20 Process 2a signals semaphore
@30 Process 1b received signal and terminates
@20 Process 2b continues and terminates
@40 Original process post-delay
```

Yielding will never allow a lower-priority-process to run. For a lower-priority process to run, the current process needs to suspend itself (with the way to get woken up later) rather than yield.

3.8 About the primitive in yield method

If you look at the exact definition of the `yield` message in Pharo, you can see that it contains an annotation mentionning that this is primitive. The primitive is an optimisation.

```
ProcessScheduler >> yield
  | semaphore |
  <primitive: 167>
  semaphore := Semaphore new.
  [semaphore signal] fork.
  semaphore wait.
```

When this method is executed, either the primitive puts the calling process to the back of its run queue, or (if the primitive is not implemented), it performs what we explained earlier and that is illustrated by the Figure 3-2.

Note that all the primitive does is to circumvent having to create a semaphore, to create, to schedule a process, and to signal and to wait to move a process to the back of its run queue. This is worthwhile because most of the time a process's run queue is empty, it being the only runnable process at that priority.

```
| yielded |
yielded := false.
[ yielded := true ] fork.
Processor yield.
yielded
>>> true
```

In the provious snippet, the expression `Processor yield` gives a chance to the created process to run. Note that the example does not show precisely if the UI thread was executed first after the `yield` and that in its logic it yields periodically to let lower processes running or if it was just put at the end of its run queue.

Here is the code of the primitive: if the run queue of the active process priority is empty nothing happens, else the active process is added as last item in the run queue corresponding to its priority and the highest priority process is run.

```
InterpreterPrimitives >> primitiveYield
  "Primitively do the equivalent of Process>yield, avoiding the
    overhead of a fork and a wait in the standard implementation."

  | scheduler activeProc priority processLists processList |
  scheduler := self schedulerPointer.
  activeProc := objectMemory fetchPointer: ActiveProcessIndex
    ofObject: scheduler.
  priority := self quickFetchInteger: PriorityIndex ofObject:
    activeProc.
  processLists := objectMemory fetchPointer: ProcessListsIndex
    ofObject: scheduler.
  processList := objectMemory fetchPointer: priority - 1 ofObject:
    processLists.

  (self isEmptyList: processList) ifFalse:
    [self addLastLink: activeProc toList: processList.
     self transferTo: self wakeHighestPriority]
```

3.9 About processPreemption settings

Now we will discuss a bit the settings of the VM regarding process preemption: What exactly happens when a process is preempted by a process of a higher priority, and which process is scheduled after the execution of a `yield` message. The following is based on an answer of E. Miranda on the VM mailing-list.

The virtual machine has a setting to change the behavior of process preemption and especially which process gets resumed once the preempting process terminates.

In Pharo the setting is true. It means that the interrupted process will be added to the end of the queue and giving other processes a chance to execute themselves without having to have an explicit `yield`.

```
Smalltalk vm processPreemptionYields
>>> true
```

If `Smalltalk vm processPreemptionYields` returns false then when preempted by a higher-priority process, the current process stays at the head of its run queue. It means that it will the first one of this priority to be resumed.

Note that when a process waits on a semaphore, it is removed from its run queue. When a process resumes, it always gets added to the back of its run queue. The `processPreemptionYields` setting does not change anything.

3.10 Comparing the two semantics

The two following examples show the difference between the two semantics that can be controlled by the `processPreemptionYields` setting.

First example: two equal processes

- Step 1. First we create two processes at a lower priority than the active process and at a priority where there are no other processes. The first expression will find an empty priority level at a priority lower than the active process.
- Step 2. Then create two processes at that priority and check that their order in the list is the same as the order in which they were created.
- Step 3. Set the boolean to indicate that this point was reached and block on a delay, allowing the processes to run to termination. Check that the processes have indeed terminated.

```
| run priority process1 process2 |
run := true.
"step1"
priority := Processor activePriority - 1.
[(Processor waitingProcessesAt: priority) isEmpty] whileFalse:
  [priority := priority - 1].
"step2"
```

```
process1 := [[run] whileTrue] forkAt: priority.
process2 := [[run] whileTrue] forkAt: priority.
self assert: (Processor waitingProcessesAt: priority) first ==
    process1.
self assert: (Processor waitingProcessesAt: priority) last ==
    process2.
"step3"
run := false.
(Delay forMilliseconds: 50) wait.
self assert: (Processor waitingProcessesAt: priority) isEmpty
```

3.11 Second example: preempting P1

The steps 1 and 2 are identical. Now let's preempt `process1` while it is running, by waiting on a delay without setting run to false:

```
| run priority process1 process2 |
run := true.
"step1"
priority := Processor activePriority - 1.
[(Processor waitingProcessesAt: priority) isEmpty] whileFalse:
  [priority := priority - 1].
"step2"
process1 := [[run] whileTrue] forkAt: priority.
process2 := [[run] whileTrue] forkAt: priority.
self assert: (Processor waitingProcessesAt: priority) first ==
    process1.
self assert: (Processor waitingProcessesAt: priority) last ==
    process2.

"Now block on a delay, allowing the first one to run, spinning in
    its loop.
When the delay ends the current process (the one executing the code
    snippet)
will preempt process1, because process1 is at a lower priority."

(Delay forMilliseconds: 50) wait.

Smalltalk vm processPreemptionYields
  ifTrue:
    "If process preemption yields, process1 will get sent to the
    back of the run
    queue (give a chance to other processes to execute without
    explicitly yielding a process)"
    [ self assert: (Processor waitingProcessesAt: priority) first ==
    process2.
    self assert: (Processor waitingProcessesAt: priority) last ==
    process1 ]
  ifFalse: "If process preemption doesn't yield, the processes
    retain their order
    (process must explicit collaborate using yield to pass control
    among them."
    [ self assert: (Processor waitingProcessesAt: priority) first ==
    process1.
     self assert: (Processor waitingProcessesAt: priority) last ==
    process2 ].

"step3"
run := false.
(Delay forMilliseconds: 50) wait.
"Check that they have indeed terminated"
self assert: (Processor waitingProcessesAt: priority) isEmpty
```

Run the above after trying both `Smalltalk vm processPreemptionYields: false` and `Smalltalk processPreemptionYields: true`.

What the setting controls is what happens when a process is preempted by a higher-priority process. The `processPreemptionYields = true` does an implicit yield of the preempted process. It changes the order of the run queue by putting the preempted process at the end of the run queue letting a chance to other processes to execute.

3.12 Conclusion

This chapter presents some advanced parts of the scheduler and we hope that it gives a better picture of the scheduling behavior and in particular the preemption of the current running process by a process of higher priority as well as the way yielding the control is implemented.

CHAPTER 4

Some examples of semaphores at work

Semaphores are low-level concurrency abstractions. In this chapter, we present some abstractions built on top of semaphores: `Promise`, `SharedQueue`, and discuss Rendez-vous.

4.1 Promise

Sometimes we have a computation that can take times. We would like to have the possibility not be blocked waiting for it especially if we do not need immediately. Of course there is no magic and we accept to only wait when we need the result of the computation. We would like a promise that we will get the result in the future. In the literature, such abstraction is called a promise or a future. Let us implement a simple promise mechanism: our implementation will not manage errors that could happen during the promise execution. The idea behind the implementation is to design a block that

1. returns a promise and will get access to the block execution value
2. executes the block in a separated thread.

4.2 Illustration

For example, `[ 1 + 2 ] promise` returns a promise, and executes `1 + 2` in a different thread. When the user wants to know the value of the promise it sends the message `value` to the promise: if the value has been computed, it is handed in, else it is blocked waiting for the result to be computed.

The implementation uses a semaphore to protect the computed value, it means that the requesting process will wait for the semaphore until the value is available, but the user of the promise will only be blocked when it requests the value of the promise (not on promise creation).

The following snippet shows that even if the promise contains an endless loop, it is only looping forever when the promise value is requested - the variable `executed` is true and the program loops forever.

```
  | executed promise |
  executed := false.
  promise := [ endless loops ] promise.
  executed := true.
  promise value
```

4.3 Promise implementation

Let us write some tests: First we checks that a promise does not have value when it is only created.

```
testPromiseCreation
  | promise |
  promise := [ 1 + 2 ] promise.
  self deny: promise hasValue.
  self deny: promise equals: 3
```

The second test, create a promise and shows that when its value is requested its value is returned.

```
testPromise
  | promise |
  promise := [ 1 + 2 ] promise.
  self assert: promise value equals: 3
```

It is difficult to test that a program will be blocked until the value is present, since it will block the test runner thread itself. What we can do is to make the promise execution waits on a semaphore before computing a value and to create a second thread that waits for a couple of seconds and signals semaphore. This way we can check that the execution is happening or not.

```
testPromiseBlockingAndUnblocking

  | controllingPromiseSemaphore promise |
  controllingPromiseSemaphore := Semaphore new.

  [ (Delay forSeconds: 2) wait.
  controllingPromiseSemaphore signal ] fork.

  promise := [ controllingPromiseSemaphore wait.
        1 + 3 ] promise.
  self deny: promise hasValue.

  (Delay forSeconds: 5) wait.
  self assert: promise hasValue.
  self assert: promise value equals: 4
```

We have in total three threads: One thread created by the promise that is waiting on the controlling semaphore. One thread executing the controlling semaphore and one thread executing the test itself. When the test is executed, two threads are spawned and the test will first check that the promise has not been executed and wait more time than the thread controlling semaphore: this thread is waiting some seconds to make sure that the test can execute the first assertion, then it signals the controlling semaphore. When this semaphore is signalled, the promise execution thread is scheduled and will be executed.

4.4 Implementation

We define two methods on the `BlockClosure` class: `promise` and `promiseAt:`.

```
BlockClosure >> promise
  ^ self promiseAt: Processor activePriority
```

`promiseAt:` creates and return a promise object. In addition, in a separate process, it stores the value of the block itself in the promise.

```
BlockClosure >> promiseAt: aPriority
  "Answer a promise that represents the result of the receiver
    execution
  at the given priority."

  | promise |
  promise := Promise new.
  [ promise value: self value ] forkAt: aPriority.
  ^ promise
```

We create a class with a semaphore protecting the computed value, a value and a boolean that lets us know the state of the promise.

```
Object subclass: #Promise
  instanceVariableNames: 'valueProtectingSemaphore value hasValue'
  classVariableNames: ''
  package: 'Promise'
```

We initialize by simply creating a semaphore and setting that the value has not be computed.

```
Promise >> initialize
  super initialize.
  valueProtectingSemaphore := Semaphore new.
  hasValue := false
```

We provide on simple testing method to know the state of the promise.

```
Promise >> hasValue
  ^ hasValue
```

Nwo the method `value` wait on the protecting semaphore. Once it is executing, it means that the promise has computed its value, so it should not block anymore. This is why it signals the protecting semaphore before returning the value.

```
Promise >> value
  "Wait for a value and once it is available returns it"

  valueProtectingSemaphore wait.
  valueProtectingSemaphore signal. "To allow multiple requests for
    the value."
  ^ value
```

Finally the method `value:` stores the value, set that the value has been computed and signal the protecting semaphore that the value is available. Note that such method should not be directly use but should only be invoked by a block closure.

```
Promise >> value: resultValue

  value := resultValue.
  hasValue := true.
  valueProtectingSemaphore signal
```

4.5 SharedQueue: a nice semaphore example

A `SharedQueue` is a FIFO (first in first out) structure. It is often used when a structure can be used by multiple processes that may access the same structure. The implementation of a `SharedQueue` uses semaphores to protect its internal queue from concurrent accesses: in particular, a read should not happen when a write is under execution. Similarly two reads would not read elements in the correct order.

The definition in Pharo core is different because based on `Monitor`. A monitor is a more advanced abstraction to manage concurrency situations.

Let us look at a possible definition. We define a class with the following instance variables: a contents holding the elements of the queue, a read and write position and two semaphores for reading and writing control.

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

`accessProtect` is a mutual exclusion semaphore used to synchronise write operations while `readSync` is a semahore used for synchronizing read operations. These variables are instantiated in the `initialize` method as follows:

```
SharedQueue >> initialize
  super initialize.
  accessProtect := Semaphore forMutualExclusion.
  readSynch := Semaphore new
```

These two semaphores are used in the methods to access (`next`) and add elements (`nextPut:`). The idea is that a read should be blocked when there is no element and adding an element will enable reading. In addition any modification of the internal elements should happen within one single process at the same time.

```
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
```

In the method `next` used to access elements, the semaphore `readSynch` guards the beginning of the method (line 3). If a process sends the `next` message when the queue is empty, the process will be suspended and placed in the waiting list of the semaphore `readSync`. Only the addition of a new element will make this process executable (as shown in the `nextPut:` method below). The critiical section managed by the `accessProtect` semaphore (lines 4 to 10) ensures that queue elements cannot be interrupted by another process that could be make the queue inconsistent.

In the method `nextPut:`, the critical section (lines 3 to 6) protects the contents of the queue. After such a critical section, the `readSyn` semaphore is signalled. This is makes sure that the waiting read processes can now work. Again the modification of the internal queue is ensured to not be transversed by different processes.

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

4.6 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 ]
```

4.7 Conclusion

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