"Concurrent Programming in Pharo" by Yuriy Tymchuk on 4 August 2014 at Uko (/hub/Uko)/concurrentProgrammingInPharo (/hub/Uko/concurrentProgrammingInPharo) under
#Concurrent (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&3),
#Programming (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&4),
#Pharo (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&5),
#proces (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&6),
#processor (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&7),
#semaphore (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&8),
#delay (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&9)
🠅 1 thanks (/hub/Uko/concurrentProgrammingInPharo?_s=KvnZ30h1Q1uM21Q8&_k=GOV2oXcpKKeBLkdo&10)

Concurrent Programming in Pharo

Pharo as any Smalltalk 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 Smalltalk was to propose a complete OS and as such a Smalltalk run-time offers the possibility to execute different processes (or threads) 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 can 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. We will present one basic abstraction proposed by: Semaphore and the critical section.

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

- First version: August 15, 2013

1. Processes by Example

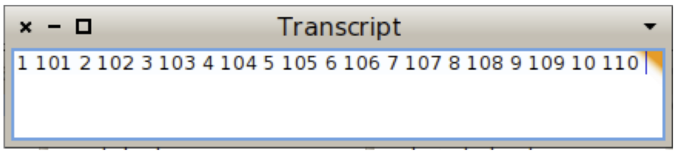
Pharo supports the concurrent execution of multiple programs using independent processes. These processes are lightweight processes as they share a common memory space. Such processes are instances of the class `Process` . Note that in operating systems, processes have their own memory and communicate via pipes supporting a strong isolation. In Pharo, processes are what is usually called a (green) thread. They have their own execution flow but share the same memory space and use concurrent abstractions such semaphores to synchronize with each other.

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 stop 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` .

```
[ 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.

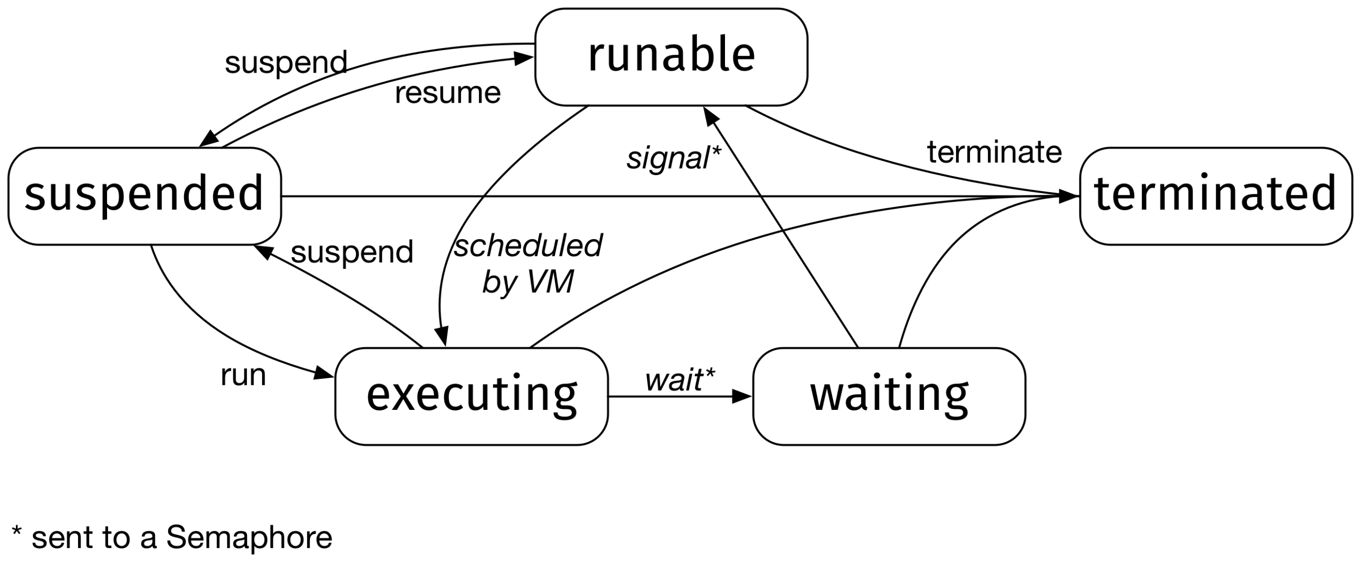


```
1 101 2 102 3 103 4 104 5 105 6 106 7 107 8 108 9 109 10 110
```

1.1. Two interleaving processes.

2. Processes

A process can be in different states depending on its life-time (created, scheduled, executing, waiting, terminated) as shown on Figure 2.1. We will look at such states now.



2.1. Process states

2.1. 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 |
  Transcript show: i  printString ; cr ] ] fork
```

This expression creates an instance of the class `Process` . It is added to the list of scheduled processes of the process scheduler. 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.

2.2. Creating without scheduling a process.

We can also create a process which is not scheduled using the message `newProcess`.

```
| pr |
pr := [ 1 to: 10 do: [ :i |
  Transcript show: i  printString ; cr ] ] newProcess.
pr resume
```

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

Also **suspended** process can be executed immediately by sending it the `run` message.

2.3. 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 |
    Transcript show: i  printString ; cr ] ] newProcessWith: #(20).
pr resume
```

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

2.4. 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`. Now we can also terminate a process using the message `terminate`. A terminated process cannot be scheduled any more.

3. Processor

To schedule processes and so execute them Pharo uses **ProcessorScheduler**. It's unique instance is stored in global variable `Processor`. To get the running process, you can execute: `Processor activeProcess`.

3.1. Process priority

At any time only one process can be executed. Frist 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 accesses by sending specific messages to `Processor`. For example:

```
[ 1 to: 10 do: [ :i |
  Transcript show: i  printString ; cr ]
] forkAt: Processor userBackgroundPriority
```

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

Priority	Name	Purpose
100	timingPriority	Used by Processes that are dependent on real time. For example, Delays (see later).

98	highIOPriority	Used by time–critical I/O Processes, such as handling input from a network.
90	lowIOPriority	Used by most I/O Processes, such as handling input from the user (keyboard, pointing device, etc.).
70	userInterruptPriority	Used by user Processes desiring immediate service. Processes run at this level will pre–empt the window scheduler and should, therefore, not consume the Processor forever.
50	userSchedulingPriority	Used by Processes governing normal user interaction. Also the priority at which the window scheduler runs.
30	userBackgroundPriority	Used by user background Processes.
10	systemBackgroundPriority	Used by system background Processes. Examples are an optimizing compiler or status checker.
1	systemRockBottomPriority	The lowest possible priority.

Precesses 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 processes list.

As mentioned before, a process can use `Processor yield` to give an opportunity to run for the other processes with the same priority. In this case the process is moved to the end of the list.

4. Semaphores

Often we encounter situations where we need to synchronise processes. For example we wanto to do two actions in parallel and then after they both are complete continue the execution. For this job **Semaphore** is your friend. Each time you send it a `wait` message the execution stops until Semaphare receives `signal` message.

Consider the next example:

```
semaphore := Semaphore new.

[ "Do a first job ..."
   semaphore signal. ] fork.

[ "Do a second job ..."
   semaphore signal. ] fork.

semaphore wait; wait.
"notify that two jobs have finished"
```

In the end we send two `wait` messages to the **semaphore**. This means that execution will stop there until **semaphore** will receive `signal` message two times. This is only possible if two jobs have finished their execution as they do `semaphore signal` in the end of a block.

5. Delay

In case you need to pause execution for some time, you can use **Delay**. Delay can be instantiated and set up by sending `forSeconds:` or `forMilliseconds:` to the class and executed by sending it `wait` message.

For example:

```
delay := Delay forSeconds: 5.
[ 1 to: 10 do: [:i |
  Transcript show: i printString ; cr.
  delay wait ] ] fork
```

will print a number each 5 seconds.

6. Conclusion

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

4 Comments **pillarhub** 1 **Login**

Recommend 2 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

PharoLanguage • a year ago

There was an issue tracker entry opened for the pharo issue tracker to fix things:
http://pillarhub.pharocloud...
But sadly only the author can fix the text. I have closed the issue tracker entry.

34 • Reply • Share ›

Mark Smith • 4 years ago

"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 can get executed by the scheduler."

This simply isn't true. Processes of the same priority can interrupt each other - a more accurate description of the scheduling policy would be to say that policy provides round-robin with borrowing. There's nothing cooperative about this model and you will need locks to enforce mutual exclusion. A similar statement was made in a chapter for the now defunct Pharo by Example 2 and it cost me about two weeks of development time trying to figure out the strange behaviour. That was a couple years ago and the scheduler certainly needed some love, but unless it's been rewritten this is a dangerous misconception that needs correcting.

1 • Reply • Share ›

HwaJong Oh • 5 years ago

Typo: wanto