Processes and Concurrency in VW

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.listic.univ-savoie.fr/~ducasse/

License: CC-Attribution-ShareAlike 2.0

http://creativecommons.org/licenses/by-sa/2.0/

Processes & Concurrency in VW

- Processes in Smalltalk:
  - Class Process, Process States, Process Scheduling and Priorities
- Synchronization Mechanisms in Smalltalk:
  - Semaphores, Mutual Exclusion Semaphores,
- SharedQueues
- Delays
- Promises

Processes in Smalltalk: Process class

- Smalltalk supports multiple independent processes.
- Each instance of class Process represents a sequence of actions which can be executed by the virtual machine concurrently with other processes.
- Processes share a common address space (object memory)
- Blocks are used as the basis for creating processes
- [Transcript cr; show: 5 factorial printString] fork

- The new process is added to the list of scheduled processes. This process is runnable (i.e., scheduled for execution) and will start executing as soon as the current process releases the control of the processor.

Process class

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

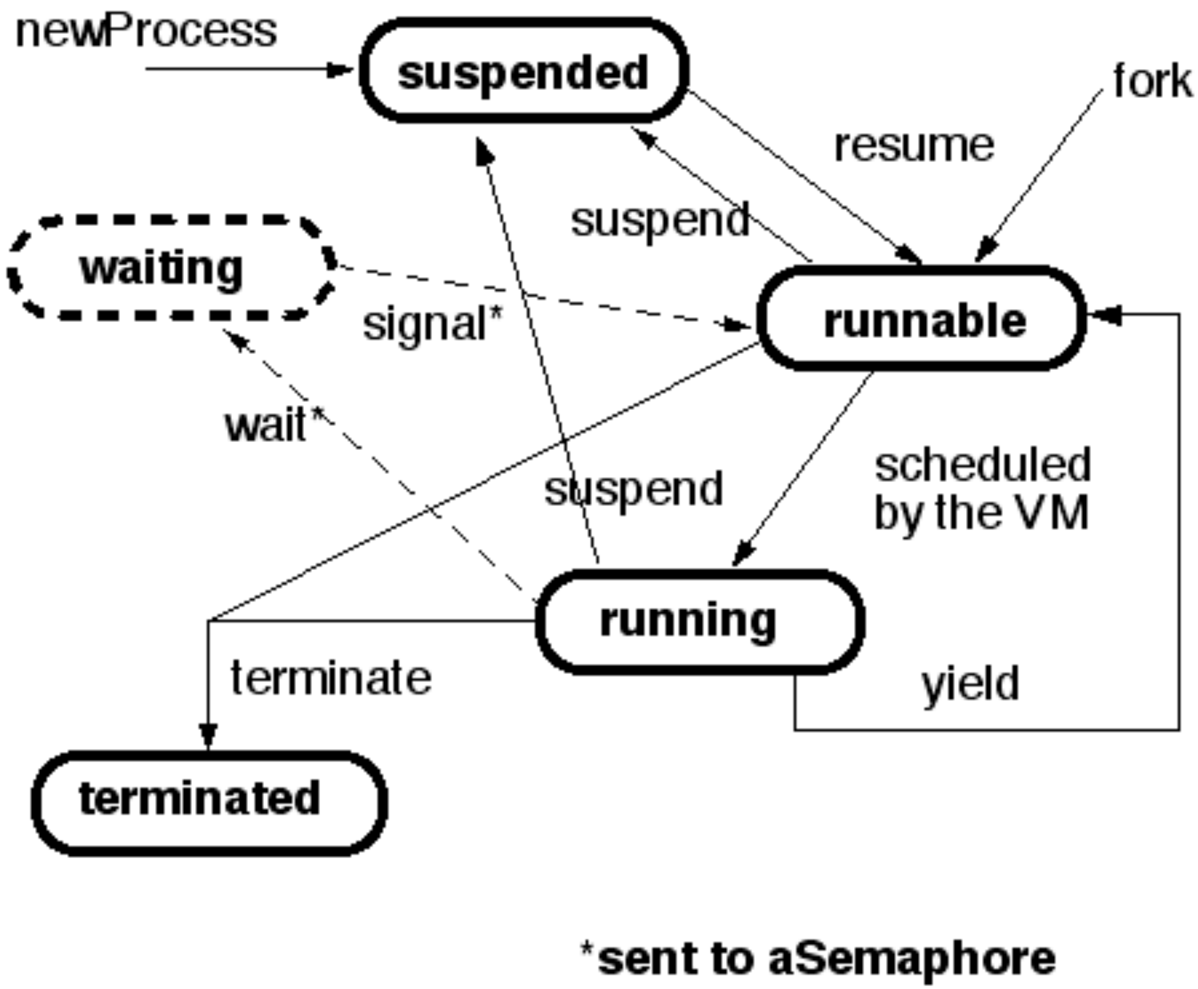
```
| aProcess |
 aProcess := [ Transcript cr; show: 5 factorial printString ]
                                                  newProcess
```

- The actual process is not actually runnable until it receives the #resume message.

Process states

- A process may be in one of the five states:

  suspended
  waiting
  runnable
  running, or
  terminated



Process class

- A process can be created with any number of arguments:

```
aProcess := [ :n | Transcript cr; show: n factorial
printString ] newProcessWithArguments: #(5).
```

- A process can be temporarily stopped using a *suspend* message. A suspended process can be restarted later using the *resume* message.

- A process can be stopped definitely using a message *terminate*. Once a process has received the *terminate*

Process Scheduling and Priorities

- Processes are scheduled by the unique instance of class ProcessorScheduler called Processor.
- A runnable process can be created with an specific priority using the *forkAt:* message:
- ```
  [ Transcript cr; show: 5 factorial printString ]
          forkAt: Processor userBackgroundPriority.
  ```
  ```

Process Scheduling and Priorities (VW)

- Process scheduling is based on priorities associated to processes.
- Processes of high priority run before lower priority.
- Priority values go between 1 and 100.
- Eight priority values have assigned names.

Priority	Name	Purpose
100	timingPriority	Used by Processes that are dependent on real time.
98	highIOPriority	Used by time-critical I/O
90	lowIOPriority	Used by most I/O Processes
70	userInterruptPriority	Used by user Processes desiring immediate service
50	userSchedulingPriority	Used by processes governing normal user interaction
30	userBackgroundPriority	Used by user background processes
10	systemBackgroundPriority	Used by system background processes
1	systemRockBottonPriority	The lowest possible priority

In Squeak

- SystemRockBottomPriority := 10.
- SystemBackgroundPriority := 20.
- UserBackgroundPriority := 30.
- UserSchedulingPriority := 40.
- UserInterruptPriority := 50.
- LowIOPriority := 60.
- HighIOPriority := 70.
- TimingPriority := 80.

In VW

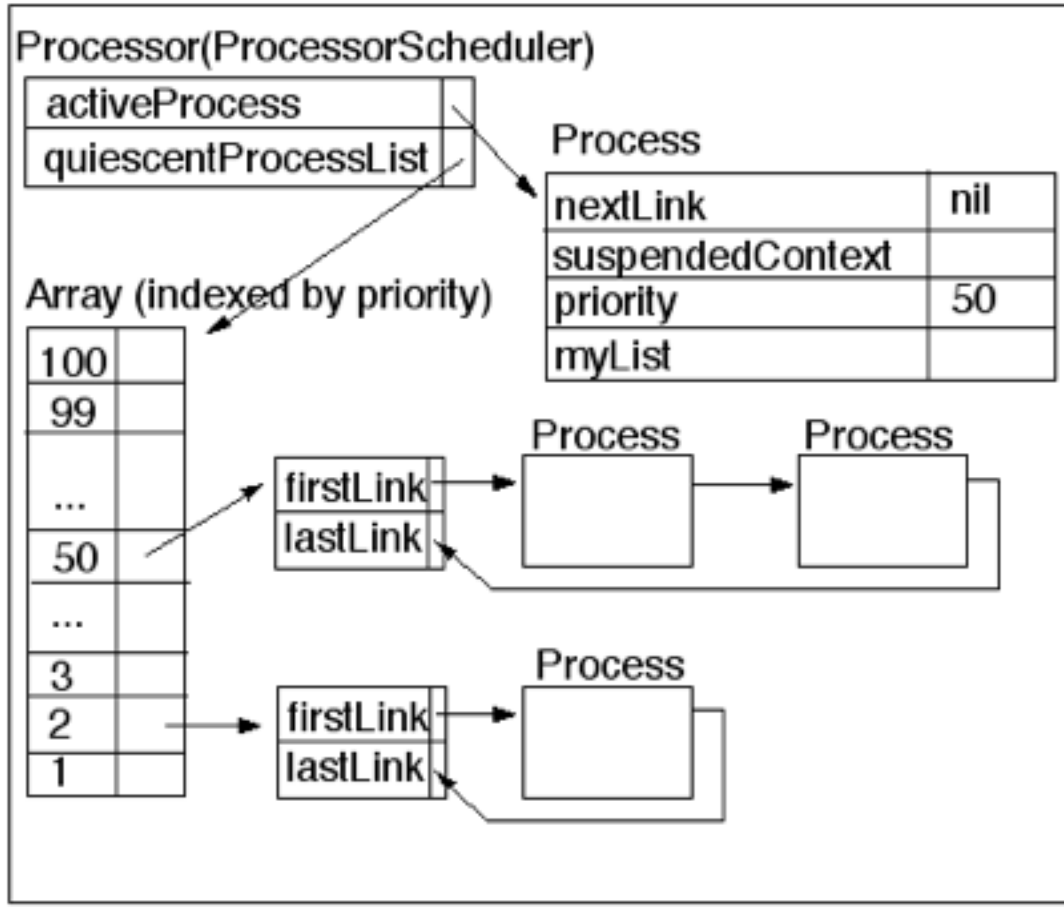
- The priority of a process can be changed by using a #priority: message

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

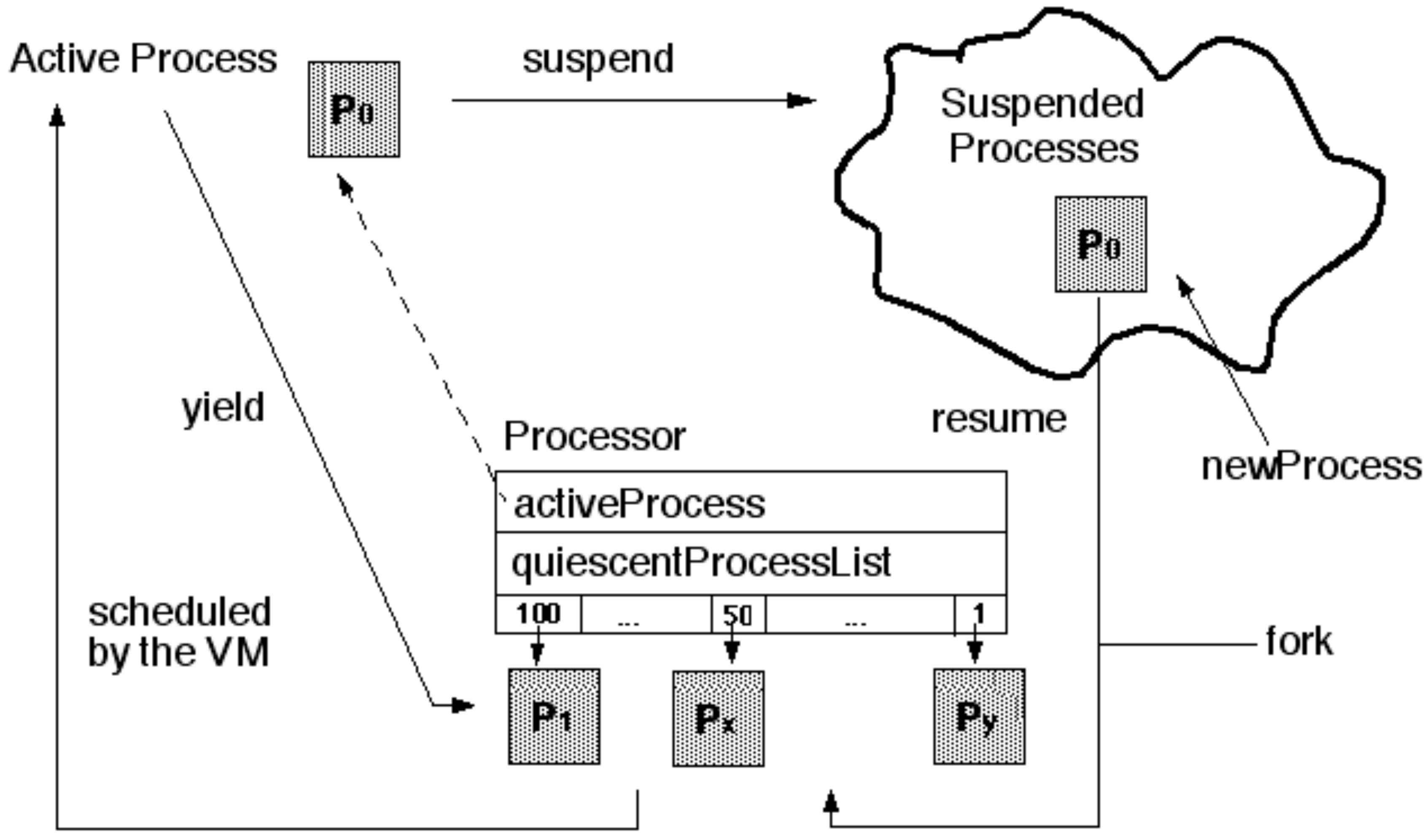
- The default process priority is userSchedulingPriority (50)

Process Scheduling Algorithm

- The active process can be identified by the expression:
  Processor activeProcess
- The processor is given to the process having the highest priority.
- A process will run until it is suspended, terminated or pre-empted by a higher priority process, before giving up the processor.
- When the highest priority is held by multiple processes, the active process can give up the processor by



Process Scheduling



Synchronization Mechanisms

- Concurrent processes typically have references to some shared objects. Such objects may receive messages from these processes in an arbitrary order, which can lead to unpredictable results. Synchronization mechanisms serve mainly to maintain consistency of shared objects.

N naturals

- We can calculate the sum of the first N natural numbers:

```
| n |
n := 100000.
[ | i temp |
Transcript cr; show: 'P1 running'.
  i := 1. temp := 0.
  [ i <= n ] whileTrue: [ temp := temp + i. i := i + 1 ].
Transcript cr; show: 'P1 sum = '; show: temp
printString ]
        forkAt: 60.
```

Accessing a Shared Variable

- What happens if at the same time another process modifies the value of n?

```
| n d |
n := 100000.
d := Delay forMilliseconds: 400.
[ | i temp |
	Transcript cr; show: 'P1 running'.
	i := 1. temp := 0.
	[ i <= n ] whileTrue: [ temp := temp + i.
(i = 5000) ifTrue: [ d wait ].					i := i + 1 ].
	Transcript cr; show: 'P1 sum is = '; show: temp printString ] forkAt: 60.
[ Transcript cr; show: 'P2 running'. n := 10 ] forkAt: 50.
```

P1 running
P2 running

Synchronization using Semaphores

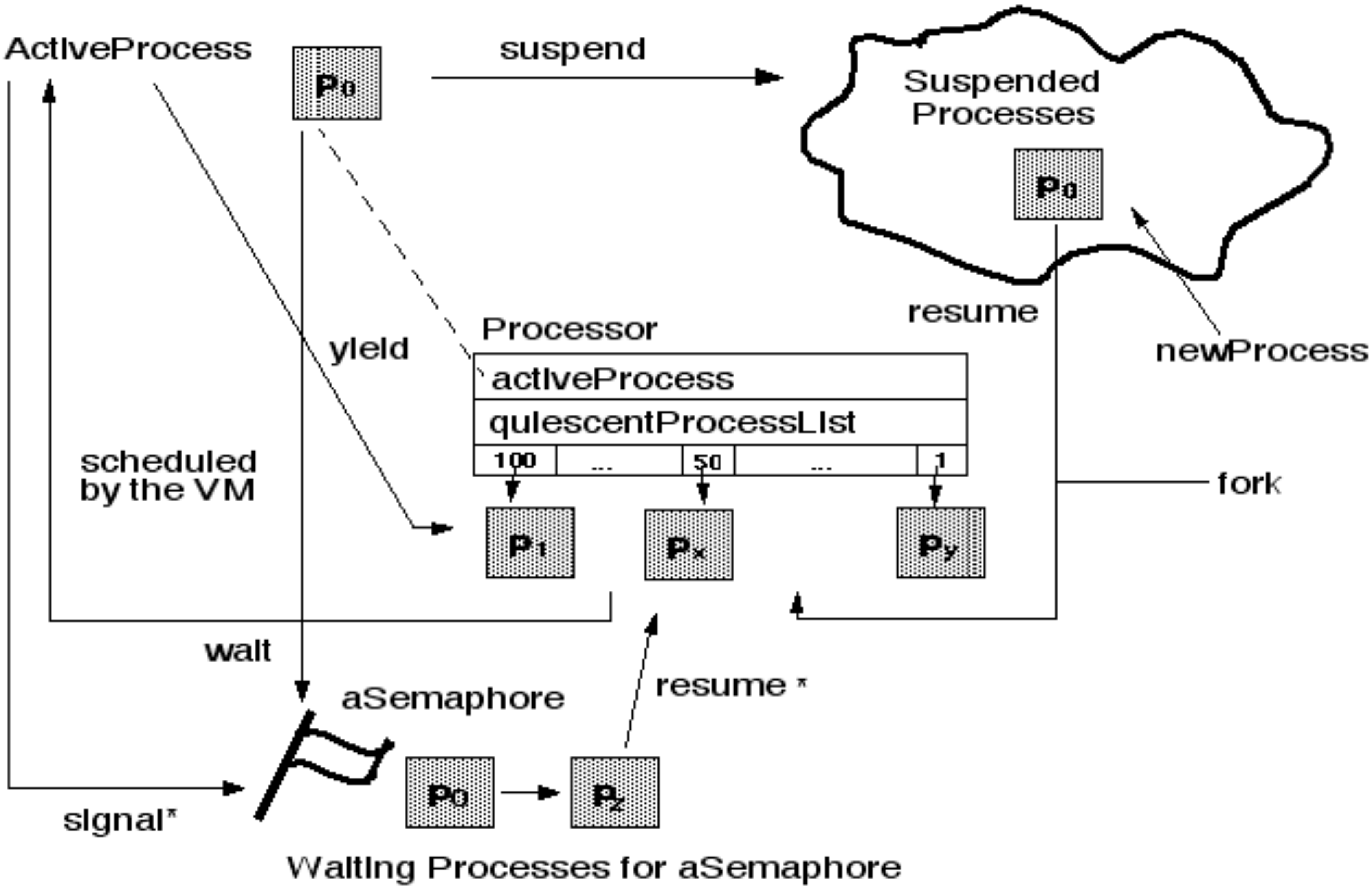
- A semaphore is an object used to synchronize multiple processes. A process waits for an event to occur by sending the message #wait to the semaphore. Another process then signals that the event has occurred by sending the message #signal to the semaphore.

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

Semaphores

- If a semaphore receives a *wait* message for which no corresponding *signal* has been sent, the process sending the *wait* message is suspended.
- Each semaphore maintains a linked list of suspended processes.
- If a semaphore receives a *wait* from two or more processes, it resumes only one process for each signal it receives
- A semaphore pays no attention to the priority of a process. Processes are queued in the same order in which they "waited" on the semaphore.

Semaphores (ii)



S.Ducasse

Semaphores for Mutual Exclusion

- A semaphore for mutual exclusion must start with one extra #signal, otherwise the critical section will never be entered. A special instance creation method is provided:

    Semaphore forMutualExclusion.

- Semaphores are frequently used to provide mutual exclusion for a "critical section". This is supported by the instance method ***critical:***. The block argument is only executed when no other critical blocks sharing the same semaphore are evaluating.

Example

```
| n d sem |
 n := 100000.
 d := Delay forMilliseconds: 400.
 [ | i temp |
 Transcript cr; show: 'P1  running'.
 i := 1. temp := 0.
 sem critical: [ [ i <= n ] whileTrue: [ temp := temp + i.
                                        (i = 5000) ifTrue: [ d wait ].i := i
+ 1 ]. ].
 Transcript cr; show: 'P1  sum is = '; show: temp printString ]
forkAt: 60.
 [Transcript cr; show: 'P2 running'. sem critical: [ n := 10 ]]
```

Synchronization with a SharedQueue

- A SharedQueue enables synchronized communication between processes. It works like a normal queue, with the main difference being that aSharedQueue protects itself against possible concurrent accesses (multiple writes and/or multiple reads).
- Processes add objects to the shared queue by using the message #nextPut: (1) and read objects from the shared queue by sending the message #next (3).
- ```
  | aSharedQueue d |
  d := Delay forMilliseconds: 400.
  aSharedQueue := SharedQueue new.
  [ 1 to: 5 do:[:i | aSharedQueue nextPut: i ] ] fork.
  [ 6 to: 10 do:[:i | aSharedQueue nextPut: i. d wait ] ] forkAt: 60.
  [ 1 to: 5 do:[:i | Transcript cr; show:aSharedQueue next printString] ]
  ```
  ```

Synchronization with a SharedQueue

- If no object is available in the shared queue when the messsage *next* is received, the process is suspended.
- We can query whether the shared queue is empty or not with the message *isEmpty*

Delays

- Instances of class Delay are used to delay the execution of a process.
- An instance of class Delay responds to the message *wait* by suspending the active process for a certain amount of time.
- The time at which to resume is specified when the delay instance is created. Time can be specified relative to the current time with the messages *forMilliseconds:* and *forSeconds:*.

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

untilMilliseconds:...

- The resumption time can also be specified at an absolute time with respect to the system's millisecond clock with the message *untilMilliseconds:*. Delays created in this way can be sent the message wait at most once.

Promises

- Class Promise provides a means to evaluate a block within a concurrent process.
- An instance of Promise can be created by sending the message *promise* to a block:

  [5 factorial] promise

- The message *promiseAt:* can be used to specify the priority of the process created.

Promises (ii)

- The result of the block can be accessed by sending the message value to the promise:

```
| promise |
promise := [ 5 factorial ] promise.
Transcript cr; show: promise value printString.
```

  If the block has not completed evaluation, then the process that attempts to read the value of a promise will wait until the process evaluating the block has completed.

  A promise may be interrogated to discover if the process has completed by sending the message *hasValue*