Proposal for 1987 Smalltalk Standard: Processes

prepared by:
L. Peter Deutsch, Xerox, ParcPlace Systems

other contributors (alphabetical order, partial listing):
Mark Miller, Xerox PARC/ISL
Allen Wirfs-Brock, Tektronix

This is a working document specifying the standard behavior of Processes and Semaphores for the 1987 version of the Smalltalk-80 language. Comments are solicited.

Version history:

[1] 25 February 1987: first version, split off from exec87.text. This is the version distributed at Feb. 26-27 implementors workshop.

Process

Processes are lightweight threads of control, as in Smalltalk-80 and many Lisp systems: they are not protection domains or separate address spaces, as in Unix, or separate naming environments or explicit linear stack areas, as in ZetaLisp. (This is not meant to imply that such facilities are not useful, only that the term "process" as used in this document, and the Smalltalk standard class Process, do not cover them.)

From the client's point of view, a process has just a few pieces of visible state:

- A scheduling state: Dead, Suspended, Waiting <Semaphore>, WaitingSuspended <Semaphore>, or Active.
- A numeric priority, a positive integer with an implementation-dependent maximum value. Processes with higher priority run in preference to those with lower priority: see below for more details. We require that implementations provide at least 8 priority levels.
- An execution state, consisting of a Context and possibly some implementation-dependent information such as saved registers. Each of these state components is the subject of sections below.

In the current Smalltalk-80 system, there is a ProcessorScheduler that holds the priority lists of runnable Processes, and also handles the fine-grained timer. Since Smalltalk appears to be a promising language for multiprocessors, we wish to specify its process facilities in a way that does not limit it to execution on a uniprocessor. For this reason, we deliberately choose not to define a Processor or ProcessorScheduler object; instead, we reassign the functions of the current

ProcessorScheduler to class Process, except for the timer function which we recommend be given to a new Timer object.

Scheduling states

From the client's point of view, a process is in one of four scheduling states:
- Dead: it cannot be run. The only time a process is dead is after it has terminated.
- Suspended: it is potentially runnable, but it must be explicitly resumed.
- Waiting: it can be run as soon as a particular Semaphore is signalled.
- WaitingSuspended: it can be run as soon as a particular Semaphore is signalled and it is explicitly resumed.
- Active: it can run at any time, according to the priority decisions made by the scheduler. This includes any processes that are running now (in a multi-processor system, there may be more than one.)

The transitions between these scheduling states occur as follows:
Active -> Waiting: the process sends a wait message to a Semaphore with no signals.
Active -> Suspended: the process receives a suspend message.
Waiting -> WaitingSuspended: the process receives a suspend message.
Active -> Dead: the process returns from its root context.
(Active, Waiting, Suspended, WaitingSuspended) -> Dead: the process receives a kill message.
Waiting -> Active: the process is waiting on a Semaphore, and the Semaphore receives a signal message.
Suspended -> Active: the process receives a resume message.
WaitingSuspended -> Waiting: the process receives a resume message.
WaitingSuspended -> Suspended: the process is waiting on a Semaphore, and the Semaphore receives a signal message.
Dead -> Suspended: the process' execution state is reset.

We have deliberately omitted any distinction between processes that CAN run and processes that ARE running (aside from the ability for a running process to find out its own identity), and any facilities for querying the scheduler as to the current set of Active processes. This implies, in particular, that the Smalltalk environment must explicitly keep track of which process will be interrupted if the user types a control-C. We consider this an improvement over the current haphazard selection of a running process.

Priorities

It has been observed that the current Smalltalk-80 notion of process priorities doesn't work very well for any of its intended uses: the desire to provide fast response to I/O events (including user interaction) would better be served by a deadline mechanism, and the desire to allocate CPU shares would better be served by a bidding mechanism or some other "soft" priority arrangement. While we agree

with these observations, limited time has prevented us from attempting to specify a new scheduling mechanism.

The current Smalltalk-80 scheduler makes some strong guarantees about process priorities and non-preemption, on which a good bit of current software unfortunately depends. These guarantees are not compatible with multiprocessors, or even with most operating system schedulers. Therefore, we propose only to guarantee a weaker set of properties:

- If all running processes are at priority M or lower, and a process at priority N > M becomes ready, the latter process will run, and one of the running processes may be preempted.
- If a process stops running, and the highest priority process still running is at priority M, and a process at priority N > M is ready, the latter process will run, and one of the running processes may be preempted. This is not implied by the first property, since the first property only deals with what happens when a new process becomes ready.

Note that we do NOT guarantee the following, which is commonly assumed by current Smalltalk systems:

- As long as a process at priority M is running, no (other) process at priority N <= M will run.

(By convention, if there is no process running, we pretend there is a process running at non-existent priority 0.) In a multi-processor implementation, we cannot guarantee that the highest-priority processes will always be the ones running: it may simply be too expensive to exchange enough priority information between processors to always maintain this property.

Process protocols

Creation

<Process> := <Process class> forBlock: <BlockClosure> priority: <SmallInteger>
Create a Suspended process that will start execution in an activation of the closure when Activated. The process is killed when it returns from the context created for the closure. (In Mark Miller's meta-objects proposal, the first argument should be a MetaClosure, not a BlockClosure.)

Scheduling

<Process class> current
Return the currently running Process, i.e. the one whose execution sent this message.

<Process> resume
Make a Suspended process Active, and a WaitingSuspended one Waiting.

<Process> suspend
Suspend a process in any state except Dead. A Waiting process becomes WaitingSuspended.

<Process> kill
Make a process in any scheduling state Dead. A Dead process cannot be put into any other scheduling state until it has had its execution state reset.

<Symbol> := <Process> schedulingState
Return the scheduling state of a process: #active, #waiting, #suspended, #waitingSuspended, or #dead.

<Semaphore> := <Process> currentSemaphore
If the receiver is Waiting or WaitingSuspended, return the Semaphore on which the receiver is Waiting. If the receiver is not Waiting, return nil. Note that the Semaphore may be signalled at any time -- there is no guarantee that the process will still be Waiting the next time it is queried.

Priority

<Integer> := <Process> priority
Return the priority of a process.
<Process> priority: <Integer>
Set the priority of a process. Note that even if a process has a priority higher than all other processes, there is no guarantee that it will be the only running process: setting the priority to the highest value is not an adequate means to guarantee mutual exclusion.
<Integer> := <Process class> highestPriority
Return the highest acceptable priority value.

Semaphores

The semantics of Semaphores similar to those the current Smalltalk-80 system, except that we do not guarantee that processes waiting on a Semaphore will be activated in FIFO order: instead, we only guarantee that if a process P is waiting on a Semaphore, only a finite number of other processes waiting on the same Semaphore will be activated before P is activated. (This is called a "non-starvation" condition.)

<Semaphore> wait
If the receiver is holding any signals, decrement its signal count. If the receiver is not holding any signals, make Waiting the (Running) process sending the wait message.
<Semaphore> signal
If any processes are Waiting on the receiver, make one of them Active. If no processes are Waiting, increment the

excess signal count of the receiver. If a process is WaitingSuspended, make it Suspended.

<Semaphore> waitingProcesses

Return a collection of the processes Waiting on the receiver, if any. The order of processes in the collection, or even whether the collection is ordered, is not guaranteed. If there are no processes Waiting on the receiver, return an empty collection.

It has been observed that Semaphores are a very low-level synchronization mechanism, and that their use for I/O devices in particular leads to awkward arrangements where the synchronizing signal and the data transfer must be handled separately. Two alternatives have been proposed: request/reply, as in the V Kernel (an experimental lightweight operating system kernel being developed at Stanford), or monitors or serialized objects. We have tentatively rejected request/reply because its semantics are more complex and more difficult to specify fully and precisely, and monitors or serialized objects because they seem to interact with the language in more complex ways. However, we recognize that Semaphores are not a fully satisfactory mechanism, and would welcome detailed proposals for better ones.

Execution state

<Context> := <Process> currentContext

Return the current context of a process, if Suspended, or nil, if Dead. Attempting to read the current context of a Waiting or Active process is an error: a performance monitor like the current Smalltalk-80 MessageTally should suspend the process being monitored, read or copy as much of its state as desired, and then resume the process.

<Process> currentContext: <Context>

Set the current context of a Suspended or Dead process. The context must be on the sender chain of the process. If the process was Dead, it becomes Suspended.

<Symbol> := <Process> nextStepAction

Return a symbol indicating what the (Suspended) process will do in response to the next 'step' message: #send, #return, or #advance (execute straight-line code).

<Process> step

Single-step the (Suspended) process to the next send or return (if executing straight-line code) or through the send or return. Note that a send may cause the process to change state from Suspended to Waiting, and a return may cause the process to die (if it returns from its root context).

<Process> restartContext: <Context>

Adjust the state of a Suspended process so that when it next becomes active, it will start executing the method of the indicated context from the beginning again. The context must be on the sender chain of the process.