

Smalltalk Processes

Rob Vens
Sepher Software

ESUG Smalltalk Summer School, Utrecht 1995

The slide features a blue background with a perspective grid. A 3D pie chart with various colored slices is positioned on the left side of the grid.

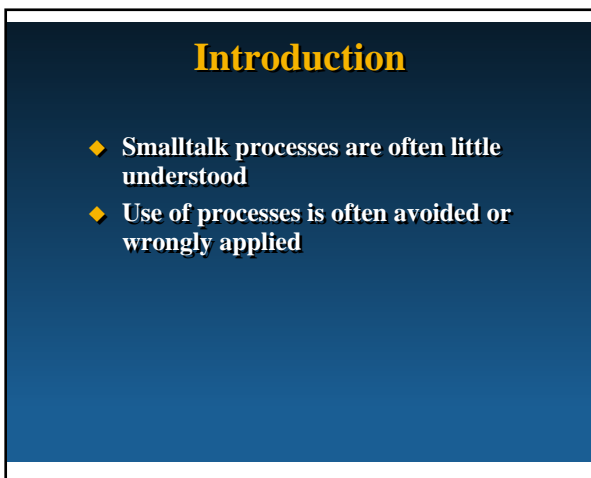


SEPHER SOFTWARE

Torenstraat 12-14
9988 SP Usquert
The Netherlands
Voice: +31 5950 5025 (after 10 oct. '95: +31 59 542 50 25)
Fax: +31 5950 5283 (after 10 oct. '95: +31 59 542 52 83)
E-mail: 73064.3461@compuserve.com

Training, consultancy, mentoring, development

The slide features a blue background with a perspective grid. A 3D pie chart with various colored slices is positioned on the left side of the grid.



Introduction

- ◆ Smalltalk processes are often little understood
- ◆ Use of processes is often avoided or wrongly applied

The slide features a blue background with a perspective grid.

Agenda

- ◆ Introduction to Smalltalk processes
 - Quick tour: SmalltalkAgents implementation
- ◆ 15 min. coffee break
- ◆ Processes and the *External World*
- ◆ A Simulation Example

Vocabulary

- ◆ Smalltalk code or named objects is in this font, message patterns in **this font**, comment in *this font*
- ◆ C code is in `this font`
- ◆ Unless specified the examples are for ParcPlace® VisualWorks® 2.0
- ◆ However, we strive for general applicability in all Smalltalk dialects (ANSI standardization effort)

Smalltalk Processes

- ◆ Priority levels
- ◆ Scheduling
- ◆ Synchronization

Priority Levels

- ◆ ParcPlace® VisualWorks® 2.0:
 - 100 priority levels available
 - 8 common levels (“Blue Book”) accessed by methods, i.e.: Processor lowIOPriority
 - Other levels specified by sending to the process:
... priority: 33

“Blue Book” Priority Levels

VisualWorks® level

- 100 timingPriority
- 98 highIOPriority
- 90 lowIOPriority
- 70 userInterruptPriority
- 50 userSchedulingPriority
- 30 userBackgroundPriority
- 10 systemBackgroundPriority
- 1 systemRockBottomPriority

Process creation

1. Use Blocks to create new processes
 - Implicit process creation using fork
 - Explicit process creation using newProcess
2. Spawn a new process from the active user interface process by blocking it (e.g., on a semaphore, or sending it suspend). This will force the user interface process to create a new

Implicit process creation

- ◆ Examples using fork:

[55 factorial] fork.

[55 factorial] forkAt:

Processor userSchedulingPriority

Explicit process creation

- ◆ Example using newProcess:

| myProc |

myProc := [55 factorial] newProcess.

myProc resume

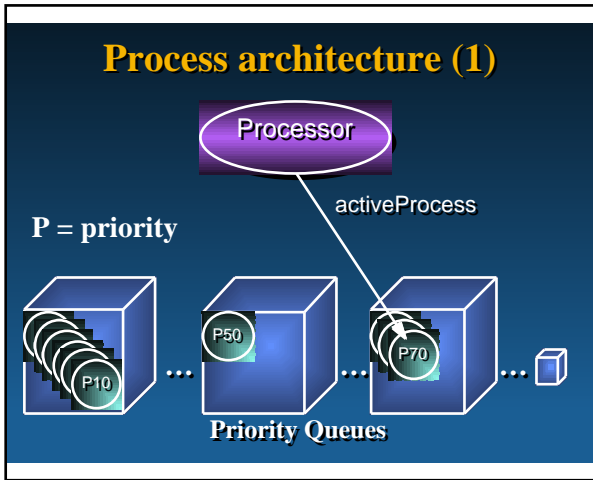
- ◆ Equivalent with fork, only returns reference to process (here myProc)

- ◆ Provide arguments at runtime with newProcessWithArguments:
anArray

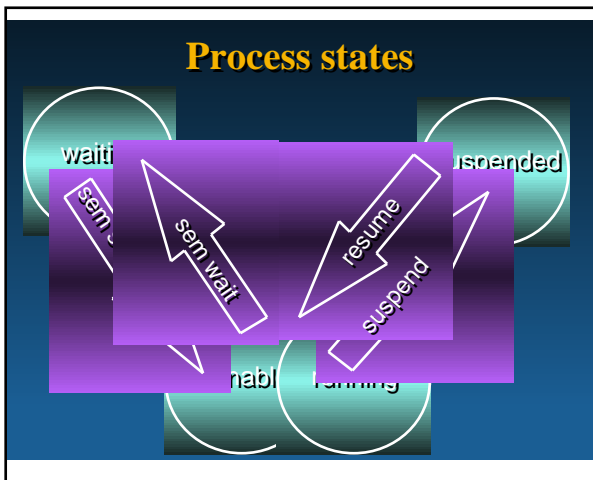
Process termination

- ◆ On return from the block

- ◆ By sending terminate



- ### Process architecture (2)
- ◆ Processor is sole instance of class ProcessScheduler
 - ◆ Processor manages instances of class Process
 - ◆ Processes are non-preemptively scheduled according to priority
 - ◆ Running processes can choose to yield control with Processor yield. Waiting processes with equal priority get a chance to run



Processes in standard VisualWorks image (1):

- ◆ On startup VW 2.0 there are 18, 19, or 20 processes; after garbage collection, there are 10
- ◆ There are four window processes:
 - 2 for the windows on the screen
 - 1 for the transient window which is a class variable in Menu
 - The third window (labeled 'Scratch Window') has no handle, and is almost gone
- ◆ The 10 processes have priorities: 100 100 98 98 90 90 90 50 50 10

Processes in standard VisualWorks image (2):

```
VisualWorks Transcript window process    50 -> a Process in nil
Workspace window process                 50 -> a Process in nil
TimingProcess (in Delay class):          100 -> a Process in
Semaphore>>wait
TerminationProcess (in Process class):   100 -> a Process in
Semaphore>>wait
FinalizationProcess (in WeakArray class): 98 -> a Process in
Semaphore>>wait
CallbackProcess (in CCallback class):    98 -> a Process in
Semaphore>>wait
outerFinalizationLoop (in WeakArray class):90 -> a Process in nil
InputProcess (in InputState class):      90 -> a Process in
Semaphore>>wait
LowSpaceProcess (in ObjectMemory class): 90 -> a Process in
Semaphore>>wait
IdleLoopProcess (in ObjectMemory class): 10 -> a Process in
Semaphore>>wait
```

Example 1: Sorting process (1)

- ◆ Create and schedule a process in the background to sort a large collection:

```
| myProc answer |
myProc := [answer := (Object withAllSubclasses collect:
[each | each name]) asSortedCollection. answer
inspect] newProcess.
myProc priority: Processor userBackgroundPriority.
myProc resume.
^answer
```

- ◆ This example returns immediately with nil, and sometime later a handicapped inspector

Example 1 (2)

What you've learned

- ◆ Standard Process architecture
- ◆ Processes in standard Smalltalk image
- ◆ How to create new processes

Process synchronization techniques

- ◆ Semaphores
- ◆ Monitors
- ◆ Shared Queues
- ◆ Delays
- ◆ RecursiveLock

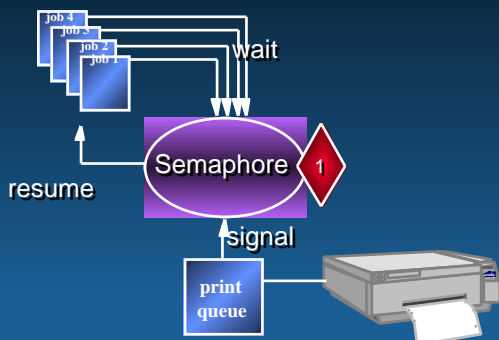
Semaphores

- ◆ What are semaphores?
- ◆ How can I use them?

What are semaphores?

- ◆ Simple mechanisms to handle processes
- ◆ The running process sends wait to the semaphore and gets suspended (if there are no excess signals)
- ◆ The process is resumed when the semaphore is signaled by anyone

Use of a semaphore



Monitors

- ◆ A Monitor is a section of code which is protected against more than one process
 - ◆ Can be done with one or more semaphores that are signaled or waited before and after the critical section
- criticalBlock valueUninterruptably

SharedQueue

- ◆ Is an implementation of a monitor
- ◆ Processes attempting to get elements from the queue are blocked when there are no elements available
- ◆ Processes putting elements on the queue eventually wake-up a requesting process
- ◆ Queue access is concurrent-safe (which is NOT the case for “regular” collections!)

Promise

- ◆ Returns an “object” but forks a process to fetch the actual object
- ◆ Can be used to provide the user-interface with model objects so that it starts-up immediately
- ◆ Actual display of real values occurs later

Delays

- ◆ Pause the current process a specified amount of time

RecursiveLock

- ◆ Nested critical regions surrounded by blocks can be created
- ◆ Deadlock is avoided

What you've learned

- ◆ Several process synchronization techniques
- ◆ Look for several implementations of "classical" concurrence problems in the example directory

C:\TUTOR\VENS

SmalltalkAgents (1)

- ◆ Processes are called *Threads*
- ◆ Architected for multi-platform, distributed environments
- ◆ Avoids use of semaphores
- ◆ Thread architecture permeates application architecture
- ◆ Architecture based on *semantic messages* posted on thread queues

SmalltalkAgents (2)

- ◆ Exactly *one* active thread at any given time
- ◆ UIScheduler thread coordinates user-interface activity
- ◆ All threads have execution state (e.g. running, waiting, sleeping, halted, etc.)
- ◆ Pre-emptive scheduling in a round-robin fashion

SmalltalkAgents (3)

Q := Thread run:
[1 to: 10000 do:
[:anIndex |
Console message: anIndex].

Q sleep. "*suspends execution*"

Q wakeup. "*resumes execution*"

Q terminate "*terminates execution*"

- ◆ Creates a thread pre-emptively time sliced with user interface thread
- ◆ Background threads are usually unable to perform user-interface operations
- ◆ Thread swapping can be disabled to achieve this

SmalltalkAgents (4)

- ◆ Use pseudo-variable thread to access the currently running thread. Example below passes control sooner than the scheduled time-slice (usually 1/60 second):

```
Q := Thread run:  
  [1 to: 10000 do:  
    [:anIndex |  
     Console message: anIndex.  
     thread yield.  
    ].  
  ].
```

External resources

- ◆ Semaphores can be signaled by external processes
- ◆ Polling external resources by Smalltalk is delegated to the operating system facilities

Signaling Semaphores (1)

- Create a semaphore in Smalltalk
- Get a pointer to the semaphore in your C code:
(type oeOop)
- Register the semaphore with the VM:
static oeInt slot;
slot = oeAllocRegistrySlot();
oeRegisteredHandleAtPut(slot,
sem);

Signalling Semaphores (2)

- When you want to signal the semaphore:
 - Get a pointer to the semaphore from the registry (do not use the previous one!):
`sem = oeRegisteredHandleAt(slot);`
 - Signal the semaphore:
`oeSignalSemaphore(sem);`

What you've learned

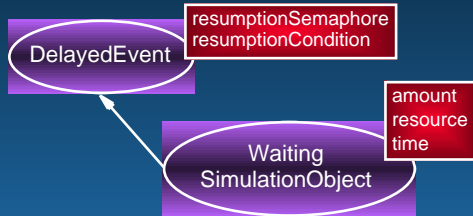
- ◆ Handling processes with the “external world”

Simulation

- ◆ Processes can be effectively utilized in discrete event simulation frameworks
- ◆ A simple framework consists of:
 - An event queue with blocked processes sorted by wake-up time
 - Simulation Objects with tasks implemented using blocks
- ◆ As an example, we will use the “Blue Book” simulation framework

Class DelayedEvent (1)

- ◆ Instances are placed on the event queue



Class DelayedEvent (2)

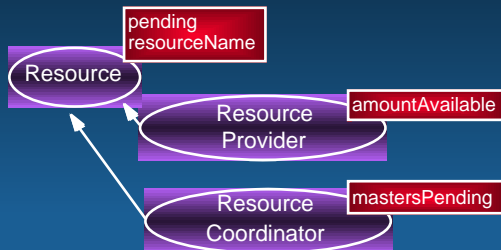
pause

"Suspend the current active process, that is, the current event that is running."

Simulation active stopProcess.
resumptionSemaphore wait

Resources (1)

- ◆ All blocking resources are implemented as Resource subclasses:



Resources (2): Class Resource

◆ Class comment:

This abstract class represents the resource in terms of its name and the queue of requests that must be satisfied.

Instance Variables:

pending <SortedCollection of: WaitingSimulationObject>
resourceName <String> with the name of the resource

Resources (3): Class ResourceProvider

- ◆ Model resources without tasks
- ◆ Count number of items (amount)
- ◆ On successful acquire, provide the resource by:
 - returning the delayed event
 - decrement the amount

Instance Variables:

amountAvailable <Integer>

Resources (4): Acquiring static resources-1

acquire: amountNeeded withPriority:
priorityNumber

| waiting |
waiting := WaitingSimulationObject
for: amountNeeded
of: self
withPriority: priorityNumber.
waiting time: Simulation active time.
self addRequest: waiting.
^waiting

Resources (5): Acquiring static resources -2

addRequest: aDelayedEvent

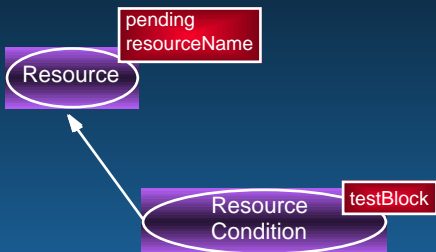
“Add a request to obtain a resource. Try to fulfill the request.”

```
self addToPending: aDelayedEvent.  
self provideResources.  
aDelayedEvent pause
```

Resources (6): Add a new Class

- ◆ Models a resource that represents a *condition*, which must evaluate to true for waiting (pending) objects to resume
- ◆ Each waiting object can have its own condition, as well as one shared
- ◆ Name the new class ResourceCondition
- ◆ Re-use the existing Resource framework

Resources (7)



Resources (7): Acquiring resources-1

```
acquireOnCondition: aBlock withPriority: priorityNumber
| waiting |
(testBlock value and: [aBlock value])
ifTrue: [^self].

"Get here if one or both of the conditions are not met."
waiting := WaitingSimulationObject
    for: 1
    of: aBlock
    withPriority: priorityNumber.
self addRequest: waiting.
^waiting
```

What you've learned

- ◆ **Process utilization in implementing a simple simulation framework**
 - use of semaphores and monitors
 - explicitly yielding control to other processes

Summary

- ◆ **You've learned to apply processes for:**
 - Keeping the user interface responsive
 - Connecting Smalltalk to the external world
 - Simulation
- ◆ **Please fill in the evaluation forms**

Where to get more information

- ◆ Other training sessions
- ◆ Example source code on the exercise machines in directory:
C:\TUTOR\VENS
- ◆ Consulting services available

Questions