HoME: Smalltalk on Mach Environment

Kazuhiro Ogata, Satoshi Kurihara, Mikio Inari and Norihisa Doi

Department of Computer Science
Graduate School of Science and Technology
Keio University
3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223 Japan
ogata@doi.cs.keio.ac.jp

Abstract

We are currently working on a project developing a version of Smalltalk in which Processes can be executed in parallel as units of execution. Mach is used as the underlying operating system. Smalltalk running on the Mach environment is called HoME. By executing a Smalltalk Process with a Mach's thread, Processes can be executed in parallel. The Smalltalk used in this project is Smalltalk-80 ver. 2.5, which is generally called HPS, and does not have a byte-code interpreter. In HPS, the CPU directly executes the machine-code by translating byte-code into machine-code during execution. In this paper, an outline of the project, its present state, and plans for the future are described.

1 Introduction

In the Smalltalk system [Goldberg 83], the class which represents independent threads of control is Process. The class Semaphore is used for synchronizing these threads of control. These classes can be used to program concurrency in Smalltalk [Doi 89]. Smalltalk's Process, however, is not scheduled by time sharing. When a Process starts executing, the other Processes are blocked until it terminates or is suspended, unless the Process yields the CPU or a Process with higher priority is scheduled. Basically, Smalltalk's Process only executes sequentially.

An example of Smalltalk where Processes (units of executing) execute in parallel is Multiprocessor Smalltalk (MS) [Pallas 88] [Pallas 89]. MS is a system which implements Berkeley Smalltalk [Ungar 83] on the Firefly multiprocessor [Thacker 87] with the V distributed operating system [Cheriton 84]. Berkeley Smalltalk has a byte-code interpreter. In MS, concurrent execution is realized by replicating the interpreter, having the same number as the number of physical processors. The queue for Process is not replicated. There is only one queue in MS. Each interpreter gets the next Process to be executed from the queue, and interprets the byte-code corresponding to the Process.

HoME (HPS on Mach Environment) is a version of Smalltalk which executes on the Mach [Tevanian 87] environment and

Processes execute in parallel as units of execution. In MS, if there are more Processes than physical processors, they cannot run simultaneously. In HoME, theoretically an infinite number of Processes can run simultaneously. The Smalltalk used in MS is a type which interprets byte-code. What is used in our research, however, is a type which dynamically translates byte-code into machine-code during execution, and the CPU directly executes the code [Deutch 84]. The ParcPlace Systems Smalltalk-80[1] virtual image ver. 2.5 is used. In HoME, concurrent execution is realized by having Mach's thread execute Smalltalk's Process. The machine used is the OMRON LUNA-88K, which carries four MC88100 RISC microprocessors [Motorola 90].

In the following sections, first the design policy is given and HoME is compared with other versions of Smalltalk. Second, the methods which were used in the implementation, and the present state of the project is described. Lastly, the future work and conclusions are stated.

2 Design Policy

HoME is designed and implemented based on the following four fundamental policies:

1. Theoretically, an infinite number of Processes can become active.
2. The moment a Process is scheduled, it becomes active.
3. Theoretically, an infinite number of processors can be thought to exist, and the role of the processor is performed by Mach's thread.
4. No concept of Process switching exists.

The first policy implies that a great number of Processes can run in parallel,

```
| lock process2 |

 lock := Semaphore forMutualExclusion.
 process2 := [20 timesRepeat: [
                 lock wait.
                 Transcript show: 'DOG'.
                  lock signal.]] newProcess.
 process2 resume.
 20 timesRepeat: [
   lock wait.
   Transcript show: 'CAT'.
   lock signal.].
```

Figure 1. Concurrent Programming in Smalltalk

and there is no limit to the number of Processes. Sending the *resume* message to a Process indicates that it has been scheduled. That is, the second policy means that a new instance of Process is created by sending a *newProcess* message to a block expression (which is enclosed with '[' and ']'), and the new instance starts executing independently and concurrently with other Processes when a *resume* message is sent to it. In HoME, Mach's thread executes Smalltalk Processes. As threads are conceptually thought of as processors, the concept of Process switching disappears in HoME. So, Processes in HoME truly represent independent threads of control.

Processes in HoME have no priority, and all Processes are treated equally. All Processes, such as those for managing I/O, for user interface, and those created by sending a *newProcess* message to a block expression, are treated equally.

3 Comparison

HoME can be compared with ordinary Smalltalk (ST) and MS according to the above four design policies.

In HoME, more than one Process can run in parallel. ST's Process, however, can

[1]Smalltalk-80 is a trademark of ParcPlace Systems.

only execute sequentially. It is possible for a multiple number of Processes to execute in parallel in MS, while there is no limit to the number of Processes which can run simultaneously in HoME, in contrast with the number in MS being dependent on the number of physical processors (replicated interpreters).

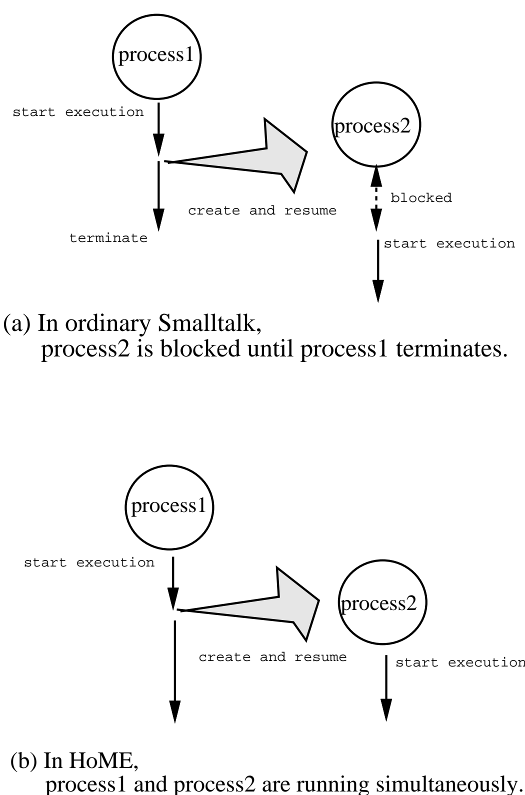
As soon as a Process is newly scheduled in HoME, it can start executing independently and concurrently with other Processes. In ST, if the priority of a newly scheduled Process is not higher than the one running (that is, the active Process), it is blocked until the active Process terminates or suspends. Otherwise, if it is higher, it becomes newly active and starts executing. As the byte-code interpreter is replicated in MS, more than one Process can run in parallel. Except for this point, when a Process is newly scheduled, MS behaves in the same way as ST.

An interpreter can be thought of as a processor in ST and MS. Always only one processor exists in ST. The number of processors in MS is the same as the number of the replicated interpreters. In HoME, a Mach's thread can be thought of as a processor, so an infinite number of processors can be said to exist.

There is no concept of Process switching in HoME. In ST, Process switching is performed when a primitive method for Process or Semaphore is executed, or a Process with a higher priority than the active Process is scheduled.

For example, consider the program in Figure 1. A Process executing this program is called process1. In this program, process1 creates process2, and schedules it by sending it the *resume* message. Then, process1 prints the string "CAT" 20 times on the Transcript. Process2 prints, independently from process1, the string "DOG" 20 times on the Transcript.

Since process1 and process2 cannot run simultaneously in ST, first process1 prints CAT 20 times, and then process2 starts



(a) In ordinary Smalltalk,
process2 is blocked until process1 terminates.

(b) In HoME,
process1 and process2 are running simultaneously.

Figure 2. Behavior of Process

printing DOG (Figure 2 (a)). In HoME, as soon as the *resume* message is sent to process2, it starts executing independently and concurrently with process1. So, the strings DOG and CAT are printed in random order on the Transcript (Figure 2 (b)).

In the program in Figure 1, the semaphore 'lock' is used to protect the object 'Transcript' which is shared by process1 and process2. As with ordinary Smalltalk, the class Semaphore must be used to protect shared objects.

4 Environment

4.1 HPS

The Smalltalk used in this research is generally called HPS, which does not have a byte-code interpreter. By dynamically

translating byte-code into machine-code during execution, the CPU directly executes the machine-code. The design policy of HPS is best expressed by the term "dynamic change of representation." This term means that the same information is represented in more than one structually different way during its lifetime, and is converted transparently between representations as needed for efficient use at any moment.

For example, execution code is usually represented as byte-code (called *V-code* or virtual code). However, it is converted into machine-code (called *N-code* or native code) which can be directly executed by the CPU when actual execution takes place. The activation record (called *Context* in Smalltalk) is created whenever a message is sent. But, most of the Contexts are never referred to as objects during their lifetimes, resulting in most of the Contexts being allocated as frames on activation stacks. Contexts allocated on stacks are called *volatile Contexts*. When Contexts are treated as objects, they are allocated in the object memory in the same way as other ordinary objects. Contexts allocated in the object memory are called *stable Contexts*. Contexts which are allocated in stacks, but have a header as an object, are called *hybrid Contexts*. Volatile Contexts are converted to hybrid Contexts when they are referred to as objects, for example, when a message is sent.

The automatic storage management in HPS uses Generation Scavenging [Ungar 84], which is a stop-and-copy scheme, and not a reference counting scheme.

4.2 MACH

The concept light-weight process denotes multiple threads of control. Mach separates the traditional process abstraction into a task and a thread. A *task* is an activation environment, which includes an address space, file descriptors, etc, and does not carry out any computing. A task is a framework for executing threads, and one with a thread is equal to an ordinary process. A *thread* is the basic unit of execution, containing the state of a processor, an activation stack, etc. A thread exists within exactly one single task. However, one task may contain any number of threads. A thread shares all memories and resources with the other threads executing within the same task. All threads theoretically execute in parallel. So, when running on a multiprocessor, multiple threads may indeed execute in parallel. A thread may be in a *suspended state* (prevented from running), or in a *runnable state* (running or scheduled to run). There is a non-negative *suspend count* associated with each thread. The suspend count is zero for runnable threads and a positive number for suspended threads.

5 Implementation

In this section, a description of the modification of certain methods, the resources to be given to each Process, and modifications in the virtual image (VI) are given. The unsuitability for HoME of the window scheduling scheme in ST is pointed out, and a new window scheduling scheme for HoME is described.

5.1 Modified Methods

The modified methods are as follows: instance methods *resume* and *suspend* in the class Process, instance methods *wait* and *signal* in the class Semaphore, and instance methods *activeProcess* and *yield* in the class ProcessorScheduler.

1. Process>>resume — When a *resume* message is sent to a Process and no threads are allocated to the Process, a thread is newly created and executes the Process. If a thread is already

allocated to the Process, the Process is resumed by resuming the thread.

2. Process>>suspend — When a *suspend* message is sent to a Process, if the Process receiving the message is the same Process that actually carries out the suspend, the thread corresponding to the Process to be suspended is suspended, that is, the suspend count is set to one. If the Process receiving the message is different from the Process actually executing the suspend, the Process executing the suspend does not immediately suspend the thread corresponding to the Process receiving the message. A thread corresponding to a Process cannot be suspended at any given time because the instance variable *suspendedContext* of the Process must contain the Context to be executed when the Process starts executing again. So, the Process executing the suspend notifies the Process receiving the message to suspend. The latter Process checks if it must suspend at a certain point. If it must, it immediately suspends at that point.

3. Semaphore>>wait — When a *wait* message is sent to a Semaphore, if the instance variable *excessSignals* of the Semaphore is positive, *excessSignals* is decremented. If *excessSignals* is zero, the Process executing the wait is appended to the tail of the queue of the Semaphore and the corresponding thread is suspended.

4. Semaphore>>signal — When a *signal* message is sent to a Semaphore, if the queue of the Semaphore is empty, *excessSignals* is incremented. If the queue contains one or more Processes, the head of the queue is dequeued and resumed.

5. ProcessorScheduler>>activeProcess — When an *activeProcess* message is sent to *Processor*, the only instance of the class ProcessorScheduler in ST, the value of the instance variable *activeProcess* is returned. In ST, only one active Process can exist at one time, and the instance variable *activeProcess* points to the active Process. In HoME, the method *activeProcess* has been modified to return the Process which executes it. So, this method is implemented as a primitive method, unlike ST.

6. ProcessorScheduler>>yield — In ST, the *yield* method is used so that the active Process can give the other Processes a chance to execute. In HoME, as all Processes can run simultaneously, this method is of no use. So, the method has been modified so as to do nothing.

5.2 Resources Given to Each Process

Resources and data structures given to each Process are as follows:

1. Stack — A stack for allocating Contexts suitable for execution (i.e. volatile Contexts) is given to each Process. A different stack for executing functions written in C, such as those for primitive methods, is also given to each Process.

2. Region for N-code — In HPS, V-code is dynamically translated into N-code during execution. It is enough to have just one region for N-code for a system to be global, if Processes can only run sequentially. If Processes can run simultaneously, this region may be shared by multiple Processes. In HoME, however, for simplicity a region is given to each Process.

5.3 Shared Resource

Object memory is shared by all Processes. In HPS, automatic storage management uses Generation Scavenging. We plan to give each Process a new space, so that there will be an improvement in performance. Also, some new features will be added to the Generation Scavenging scheme. This will be further discussed in section 6.

5.4 Synchronization

As all Processes shared object memory, object allocations are mutually excluded for synchronization. Processes may translate V-code into N-code in parallel. However, for simplicity, mutual exclusion is carried out between Processes that actually translate V-code into N-code.

There is a special thread for executing Generation Scavenging, and it is different from the threads executing Smalltalk's Processes. This thread is usually suspended. When Generation Scavenging must be carried out, it starts by resuming the thread. The thread does the following:

1. It notifies all currently running Processes to suspend. A running Process is one whose thread's suspend count is zero. All threads executing the notified Processes are registered in the array *suspendThreads*.

2. If there are any threads left which are executing a Smalltalk Process, which is not in suspendThreads and whose suspend count is zero, the Generation Scavenging thread notifies it to suspend. The Generation Scavenging thread waits until all the suspend counts of threads executing Processes become positive, that is, until all threads executing Processes are suspended.

3. After all Processes are suspended, the Generation Scavenging actually starts. The Generation Scavenging scheme is the same as in HPS.

4. When the Generation Scavenging is finished, all Processes corresponding to the threads registered in suspendThreads resume processing. The Generation Scavenging thread is suspended until the next scavenging.

5.5 Modification in VI

A new instance variable *thread* is added to the class Process so that a Process can identify itself. A new instance variable *hoMEProcessList* in the class ProcessorScheduler represents the hash table in which current active Processes are registered. The value of the instance variable *thread* is used as a hash value. In order to point to the next Process in the *hoMEProcessList*, a new instance variable *nextProcess* is added to the class Process. The instance variable *thread* of a Process contains the id number of the thread executing it. For example, when sending a *activeProcess* message to Processor, which is the only instance of the class ProcessorScheduler, the returned value is a thread corresponding to the Process. The following are carried out in the *activeProcess* method:

1. Get the id number of the thread executing the *activeProcess* method by using Mach's system call thread_self().

2. Find the Process corresponding to the id number in the instance variable *hoMEProcessList* of Processor, and return it.

Instance variables *activeProcess* and *quiescentProcessLists* in Processor are not used in HoME.

5.5.1 Window Scheduler

The window scheduling scheme in ST is fairly different from the ones used in con-

ventional window systems. In ST, usually there is only one active window. A Process for the controller controlling the window is created and runs. The other windows do not have any functions as a window, and are just Forms. When a mouse cursor is moved from active window A to nonactive window B and the mouse is clicked, the algorithm of B becoming active is as follows:

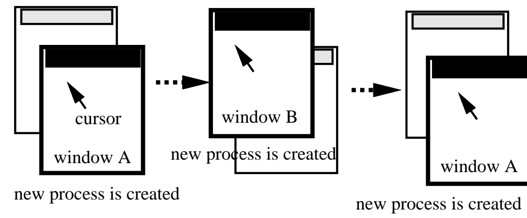
1. The controller of window A looks for a controller requesting control, that is the controller of window B.
2. The controller of window A creates a new Process for the new controller.
3. The controller of window A kills the Process allotted to itself. Then, the process for window B starts execution.

All controllers of the windows are registered in the instance variable *scheduledControllers* of the class ControlManager. The controller of window A can find the controller of window B by scanning *scheduledControllers*.

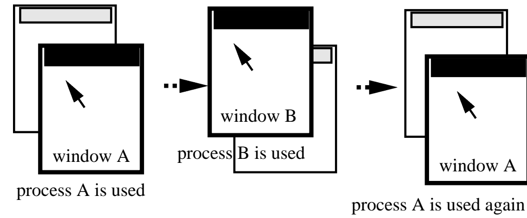
From the algorithm, when control moves from window A to window B, and back to window A, the Process previously used for window A is not used again. Whenever an active window changes, the previously used Process is killed and a new Process is created (Figure 3 (a)).

As creating a Process and executing it is a heavy load in HoME, creating and discarding Processes frequently causes a decline in performance. Thus, we have redesigned the window scheduler for HoME, by attaching a private Process to each window as in conventional window systems (Figure 3 (b)).

In the new window scheduler, a new class variable *WindowTable* is added to the class ControlManager, in which all controllers of windows and Processes for those controllers are registered. Processes for nonactive windows are suspended. When there is a change in active



(a) Window scheduler for ordinary Smalltalk



(b) Window scheduler for HoME

Figure 3. Window Scheduler

windows, the controller for the currently active window looks for the Process for the next one by scanning *WindowTable*, and then sends a *resume* message to it.

By redesigning the window scheduler, Processes are not created each time there is a change in active windows.

6 Future Work

In the present implementation of HoME, since each Process has a region for N-code, there is no need for mutual exclusion in accessing N-code. However, this means that there may be a great amount of the same N-code in the N-code regions. So, we must consider the trade-off between space efficiency and the overhead for mutual exclusion needed when sharing a region for N-code. As most N-codes can be used by all Processes, it seems more appropriate to share one region for N-code with all Processes.

Actual translations from V-code into N-code are mutually excluded. However, taking concurrency and performance into consideration, it seems more desirable to be able to translate in parallel.

When multiple Processes are able to run simultaneously, the creation of objects is in proportion to the number of Processes (physical processors). If one creation space is shared with all Processes, mutual exclusion is needed whenever objects are created. There is no limit as to the number of Processes able to run simultaneously, so the overhead for mutual exclusion increases considerably. It seems, then, more desirable to give a creation space to each Process so that there is no overhead for mutual exclusion when an object is allocated. And by giving a new space in addition to a creation space, all Processes do not have to be suspended for each Generation Scavenging. When scavenging is needed, a Process can do it in its own new space independent of other Processes. Generation Scavenging only accesses the new space. Garbage collection for old space is done only during global GC. Taking space efficiency and access frequency into consideration, it seems desirable to share old space with all Processes.

7 Conclusion

The four fundamental policies for design and implementation of HoME was described, and HoME was compared with ordinary Smalltalk and Multiprocessor Smalltalk. In the implementation of HoME, the methods to be modified and how they are to be modified, the resources to be given to each Process and those shared with all Processes were described. Also, modifications in the virtual image were mentioned.

We explained that the window scheduling scheme of ST is not suitable for HoME. The window scheduler for HoME has been redesigned under the new process scheduler based on time sharing.

In the present implementation of HoME, there are some problematic parts. We described how these are to be improved in the future.

References

[Cheriton 84] David R. Cheriton, The V kernel: a software base for distributed systems, IEEE Software, 1 (2), April 1984.

[Deutch 84] Peter Deutch and Allan Schiffman, Efficient Implementation of the Smalltalk-80 System, Proc. of 11th ACM POPL, 297-302, 1984.

[Doi 89] Norihisa Doi and Kiyoshi Segawa, On the concurrent programming in Smalltalk-80, Advances in Software Science and Technology, Vol. 1, Academic Press, 187-207, 1989.

[Goldberg 83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addision-Wesley, 1983.

[Motorola 90] Motorola, *MC88100 RISC MICROPROCESSOR USER'S MANUAL SECOND EDITION*, Prentice Hall, 1990.

[Pallas 88] Joseph Pallas and David Ungar, Multiprocessor Smalltalk: A Case Study of a Multiprocessor-Based Programming Environment, Proc. of SIGPLAN'88, 268-277, 1988.

[Pallas 89] Joseph Pallas, *Multiprocessor Smalltalk: Implementation, Performance, and Analysis*, PhD thesis, Stanford Univ., 1989.

[Thacker 87] Charles P. Thacker and Lawrence C. Stewart, Firefly: a multiprocessor workstation, Proc. of APSLOS II, Computer Society Press of the IEEE, 164-172, October 1987.

[Tevanian 87] Avadis Tevanian, Jr. and Richard F. Rashid, MACH: A Basis for Future UNIX Development, CMU-CS-87-139, 1987.

[Ungar 83] David M. Ungar and David A. Patterson, Berkeley Smalltalk: Who knows where the time goes?, In *Smalltalk-80: Bits of History, Words of Advice*, Glen Krasner (Ed), Addison-Wesley, 189-206, 1983.

[Ungar 84] David Ungar, Generation Scavenging: a nondisruptive high performance storage reclamation algorithm, Software Engineering Symposium on Practical Software Development Environments, 157-167, 1984.