An Object-Oriented Operating System Interface

Juanita J. Ewing

Artificial Intelligence Machines
Tektronix, Inc
P.O. Box 1000, MS 60-405, Wilsonville, Oregon 97070

Abstract

This paper discusses an object-oriented interface from the Smalltalk-80™ programming environment to a Unix-like operating system. This interface imposes an object-oriented paradigm on operating system facilities. We discuss some of the higher order abstractions that were created to make use of these facilities, and discuss difficulties we encountered implementing this interface. Several examples of cooperating Smalltalk and operating system processes are presented.

1. Introduction

Programming environments, to be complete, need to provide access to the full capabilities of the underlying operating system. Access should include file system functionality as well as other capabilities of the operating system such as multi-tasking. Applications may use these capabilities to reach outside communication channels, to reach specialized computing power or simply to execute normal functions of the operating system from within the programming environment. This operating system interface provides a uniform method of accessing many features of the operating system from the programming environment.

Uniform access to operating system capabilities is furnished by the execution of operating system calls. It is extended by abstractions that encompass important operating system facilities.

In our discussions, we assume the reader is familiar with Smalltalk and multi-tasking [Thompson]. Since Smalltalk has its own threads of control called processes, we shall continue to refer to a Smalltalk process as a **process**. However, an operating system process shall be referred to as a **task**.

1.1 Objectives

Our goal for this implementation was to provide a workable yet flexible operating system interface. Our list of objectives included these points:

- Provide access to low level operating system facilities.
- Implement higher order abstractions for operating system functions to promote easy access to these facilities.
- Support cooperation between Smalltalk programs and programs implemented in conventional compiled languages.
- Create a window oriented operating system command interface.

1.2 Overview

The next three sections contain a description of this interface and its abstractions. The first descriptive section covers the system call abstraction, the basis of our interface. The next section describes communication channels, such as files and pipes. The last descriptive section describes an abstraction of spawned subtasks, and a complex management system that governs the behavior of these tasks. A final section describes some example applications which use this operating system interface.

™ Smalltalk-80 is a Trademark of Xerox Corporation

2. System Calls

Assembly language programs normally have operating system interfaces at the lowest possible level — at the system call level. Our interface takes the same approach, because the lowest level is also the most versatile. Higher level abstractions are created in the interface implementation language, Smalltalk-80.

Operating system calls are represented in Smalltalk by instances of TekSystemCall. Modeled after bitblt [Goldberg], each instance of TekSystemCall denotes a specific operating system call just as each instance of BitBlt denotes a specific bit block transfer. Just as instantiated instances of BlockContext are evaluated by sending the value message, a system call is executed by sending the value message to an instantiated instance of TekSystemCall. Unlike some interfaces which require a separate primitive for each system call, in our interface a single primitive is used to execute all system calls.

Following the object-oriented paradigm of Smalltalk, each instance of TekSystemCall can easily encapsulate the registers and arguments set or read by operating system calls. Because each system call has different requirements for registers and arguments, there are many class methods returning instantiated instances — one for each system call. This encapsulation allows later examination of returned values from an operating system call. Unlike the system call, which is a transitory event, the system call object persists with the recorded state of the system call. There is also encapsulation of error conditions. Error encapsulation is used to determine if an error occured and what type of error it was. The error state of the abstraction can, in an object-oriented language, conditionally create an error notification at any time instead of being limited to the time span in which the system call is executed by the operating system. See Figure 1 for a typical example of execution of a system call and partial examination of its encapsulated state.

```
sysCall ← TekSystemCall
        read: fileDescriptor
        buffer: aStringOrByteArray
        nbytes: aStringOrByteArray size.
sysCall value.
bytesRead ← sysCall D0Out.
```

Figure 1. Execution of a System Call from Smalltalk

Many of the system call functions supported by Smalltalk also are included in another set of higher level protocol. This protocol hides operating system dependent structures and formats, especially those that are unusual for the interfacing language. The protocol is intended to present an operating system independent interface. See Figure 2 for an example implementation dealing with an unusual format required by a system call.

3. Communication Channels

Instances of class FileStream are the standard Smalltalk-80 interface to external files. An instance of FileStream is the abstraction of the state an open file.

Although we preserved the protocol of the Smalltalk-80 Version 2 release, the class FileStream has been rewritten and optimized to speed up accessing protocol using our system call interface. We have also extended the other protocols in many places. In class FileStream, encapsulated state includes the mode of

```
setMouseBounds: boundingRectangle
    "Set the allowable mouse bounds to a boundingRectangle."

    | sysCall xyUpper xyLower visibleArea |
    visibleArea ← Display boundingBox.
    (visibleArea contains: boundingRectangle)
        ifFalse: [Transcript cr; show: 'Mouse bounds set outside display bounds'].
    xyUpper ← boundingRectangle left * 65536 + boundingRectangle top.
    xyLower ← boundingRectangle right * 65536 + boundingRectangle bottom.
    sysCall ← TekSystemCall setMouseBounds: xyUpper lowerRight: xyLower.
    sysCall value
```

Figure 2. Higher Level Protocol For An Operating System Call

opening (e.g. read, write), the position of the file pointer, and the operating system file descriptor. We take advantage of the encapsulation provided by the object-oriented paradigm in this implementation. For example, file descriptors are a scarce resource since the number of open files is limited. Generally not all files need to be open at the same time. Since instances of FileStream include their file position in the encapsulation, the corresponding files can be carefully closed and no data lost in a time of scarce file descriptors. When an instance of FileStream needs to access its file again, the file is simply reopened and the recorded file state is used to position the file to the proper location. In the operating system, a file's position is a transitory state. Encapsulation of the file's state allows the automate closing and reopening of files depending on need. The implementation of this protocol makes system calls from a variety of locations including methods to create, open, read and write files.

Most operating systems have facilities designed for communication between tasks such as pipes and pseudo terminals (commonly called pseudo ttys). The class Pipe is an abstraction of an operating system facility of the same name. Instances of Pipe are created by executing the **pipe** system call. Pipes are uni-directional facilities: information flow is one way. For two way communication two pipes are required. Each pipe has two file descriptors, one for reading and one for writing.

Pipes are similar to files except that they can be opened only once. Once a pipe is closed it is gone. Files can be reset and repositioned, while pipes cannot. Thus, the automatic closing and reopening strategy does not apply to pipes even though they use the same scarce resource, file descriptors.

4. Subtasks

An instance of Subtask class is a higher level abstraction representing a spawned operating system task that loads and executes a binary file. Its encapsulation includes specification of the executable file and, optionally, environment variables and arguments to the executable file. The **vfork** and **exec** system calls are used to accomplish spawning and execution. Instances of Subtask also encapsulate a block, called the initialization block, which is executed after the vfork call and before the exec call. It is possible to create mulitple instances of Subtask which represent mutiple operating system child tasks.

Vfork creates a child task which shares memory with the parent task. Control of the Smalltalk virtual machine transfers to the child task at vfork invocation. The child task is limited, although it has access to memory and control of the virtual machine. For example, the child task at this point has no access to the keyboard but does have access to the display. The initialization block is evaluated by the child task after the vfork call. This block contains Smalltalk code for initialization of the operating system environment and for communication channels. Exec transforms the child task so it is no longer running Smalltalk, but is executing the specified binary program. After the exec system call, control of the Smalltalk virtual machine reverts to the parent task in the state left by the child task before the vfork invocation.

After a Subtask instance is created, it is sent the message start to actually spawn and execute the binary file. Normally the parent task suspends execution while waiting for its child task to terminate. This is done by executing the system call **wait**. In Smalltalk, waiting for the termination of the child task is initiated by sending the message waitOn or absoluteWait to an active instance of Subtask. Our implementation allows the user to suspend either the currently executing Smalltalk process (by sending waitOn) or the entire Smalltalk task (by sending absoluteWait), including the virtual machine. A typical Unix-style C program suspends the entire parent task. See Figure 3 for a simple example of binary file execution and suspension of only the current Smalltalk process.

4.1 Subtask Management

A complex management system, contained in Subtask class, allows a subtask to suspend either the current Smalltalk process or the entire Smalltalk task. It is possible for the user to create and control mutiple child tasks, each of which may suspend the current process or the entire task.

Suspending of only the currently active Smalltalk process is achieved by sending the **wait** message to a semaphore. Each instance of Subtask has two semaphores associated with it. One semaphore is for suspending the current process. The other semaphore is for protecting critical sections.

In the subtask management system, the dead child signal, which indicates a child task has terminated, is connected to a Smalltalk semaphore. A Smalltalk process, running at a relatively high priority, continually monitors this signal. The management system also keeps a list of all the currently active subtasks. Upon receipt of a dead child signal, the management system performs a **wait** system call. Because a dead

```
execProgram: aCommand withArgs: anOrderedCollection
    "Execute a binary program. Create an error if the
    program cannot be exec'ed or if the program
    terminates abnormally."

    | task |
    task ← Subtask
            fork: aProgram
            withArgs: anOrderedCollection
            then: [ ].
    task start isNil
        ifTrue: [task release.
            ↑self error: 'Cannot execute ' , aProgram].
    task waitOn.
    task abnormalTermination
        ifTrue: [task release.
            ↑self error: 'Error from program utility'].
    task release.
```

Figure 3. Executing a Subtask

child signal has been received, it is known that the **wait** call will return immediately with the identification of the child task and so the Smalltalk parent task will not be suspended unnecessarily. With the data from the **wait** call, the management system examines its list of subtasks and determines which has terminated. With this information the system updates its subtask list, signals the proper subtask's suspending semaphore, and the suspended Smalltalk process resumes.

Blocking of the entire parent task is done by issuing a **wait** system call directly and bypassing the dead child monitoring process. Wait calls are issued successively, each suspending the entire parent task, until the desired child task has terminated. Care is taken to update the management system's subtask list after each **wait** call so both types of suspending may occur concurrently. With this type of management, a user can have many child tasks running simultaneously.

Our implementation also has protocol for interrupting and terminating a subtask. The termination status of a child task is part of the encapsulated state and can be examined to determine if it is abnormal — usually signaling an error condition.

4.2 Critical Sections

One of the more difficult parts of implementing the management system was determining which sections of code needed to be made into critical sections. We discoved that updating the status of Subtask instances was a place where inconsistencies could arise. Given multiple processes in Smalltalk, several could each be trying to control a single instance of Subtask at the same time. Each instance of Subtask, besides containing a suspending semaphore, also has a semaphore that is an instance of Semaphore forMutualExclusion. It is used to protect critical sections of code that must not be simultaneously accessed by more than one process.

The list of active subtasks also needed to be protected since the dead child monitoring process, the absoluteWait method and the release method could all potentially access the list simultaneously. The subtask management system has another semaphore for protecting access to the subtask list.

4.3 Extensions

Operating signals can be intercepted, ignored or set to a default action by executing system calls. Expressions to set the action of signals in the child task may be included in the initialization block passed as an argument to Subtask instance creation methods.

A child task can have its priorities altered by the operating system with the execution of a system call. To take advantage of this capability we extended the encapsulation of subtasks to include priorities. Setting an instance variable in the subtask instance records the desired priority. The priority of the operating system child task is adjusted just after the initialization block is evaluated. Normally, the operating system allows user tasks to lower their priority, but not to raise their priority. This is because

user tasks run at the highest user priority possible. In our initial implementation, we found that the Smalltalk virtual machine is "busy waiting" in the user interface code and so does not usually allow the operating system scheduler to switch tasks except on a time limit basis. That situation was inadequate for interactive child tasks; we needed to raise the priority of our child tasks but the operating system would not allow this. We solved the problem by making the Smalltalk parent task run at a lower priority. This way we could raise the priority of a child task up to the original level and get adequate scheduling for reasonable interactive performance. In general, lower priority had no detrimental effect on Smalltalk performance.

Our operating system supports environment variables. Familiar environment variables include PATH — a search path specification used by shell programs and HOME — a home directory specification. Environment variables provide a way to pass information to a child task by name. Our interface supports access to and modification of the environment variables passed to the Smalltalk task at invocation. A subtask can be invoked with a modified set of environment variables by using protocol to change the set of environment variables and passing the set as an argument to a variation of normal instance creation method.

Arguments to an executable file are also passed as arguments to instance creation methods. We have provided several instance creation methods, each of which has some widely used combination of extensions such as arguments, initialization block and environment variables. These instance creation methods also cause encapsulation of the extensions. Thus, by extending the encapsulation it was easy to extend the object-oriented interface to accommodate features of the operating system.

4.4 Subtask Communication

For communication between parent and child tasks, one facility provided by our operating system is pipes. The classes PipeReadStream and PipeWriteStream implement streaming protocol for use with pipes. These classes are used in conjunctions with the pipe abstraction described earlier.

Since pipes are a one way communication facility, these streaming abstractions are clearly named to support only uni-directional information flow. The direction of flow is determined by the application and is part of the pipe's encapsulation.

The operating system supports the system call **dups** which allows a file descriptor to be redirected to another specified descriptor. Each task in the operating system has access to a minimal standard set of descriptors: *standard in*, *standard out*, *standard error*. Generic programs usually read from the keyboard, denoted by *standard in* and write to the display, denoted by *standard out* and *standard error*. Our pipe abstraction has protocol to redirect these standard descriptors to a pipe descriptor(s). This redirection allows existing external applications which communicate with the user using standard conventions, to communicate with a Smalltalk process. See Figure 4 for an example of one way communication with an external application using the standard channels.

Because both the parent and child tasks initially share memory and therefore share the same objects, these shared objects cannot reflect differing encapsulated states in the parent and child task. Usually, pipes are created by the parent task. The process of creating the child task duplicates all open parent file descriptors in the child task, including any pipe descriptors. Pipes, however, are not duplicated. It is desirable for pipe descriptors to be closed or redirected in both the parent and child, each manipulating the same object, an instance of Pipe, in a different way. Therefore, if pipe descriptor redirection were encapsulated, it would be recorded in the same object in both the parent and child tasks. The encapsulated state would reflect the state of task that last manipulated the Pipe object. If the Pipe object were copied so each task had its own copy, pipe descriptors could be encapsulated, but the objects would not reflect the available operating system channels.

We chose to share this Pipe object between two tasks, following the object-oriented paradigm. This was not sufficient to capture the state of a pipe in a straight forward manner. It is clear that additional conventions need to be established for shared objects which represent related but different states. We did not do this.

Most of the initial work in our implementation used pipes for communication. With the addition of pseudo ttys to our operating system, there is more flexibility in dealing with interactive child tasks. Pseudo ttys are more sophisticated in the status reporting and controlling of communications. They also do not operate by suspending the execution of a task on a unfulfilled read operation. Therefore, a Smalltalk process does not have to be so careful in its I/O with an external application.

```
execSystemUtility: aCommand withArgs: anOrderedCollection
    "Execute a binary program and read the entire results through a pipe.
    No mechanism for input to the executable program is provided.
    Create an error if the program cannot be exec'ed or if the program
    terminates abnormally."

    | pipe task inputSide resultOfProgram |
    pipe ← Pipe new.
    task ← Subtask
                fork: aCommand
                withArgs: anOrderedCollection
                then:
                    ["redirect write descriptor"
                    pipe mapWriteTo: Pipe stdOutDescriptor.
                    pipe mapWriteTo: Pipe stdErrorDescriptor.
                    pipe closeWrite.
                    pipe closeRead].
    task start isNil
        ifTrue:
            [pipe closeWrite.
            pipe closeRead.
            task release.
            ↑self error: 'Cannot execute ' , aCommand].
    pipe closeWrite.
    Cursor execute
        showWhile:
            [inputSide ← PipeReadStream openOn: pipe.
            resultOfProgram ← inputSide contentsOfEntireFile].
    task waitOn.
    inputSide close.
    task abnormalTermination
        ifTrue: [task release.
            ↑self error: 'Error from system utility: ' ,
                (resultOfProgram copyUpTo: Character cr)].
    task release.
    ↑resultOfProgram
```

Figure 4. A Subtask with One Way Communication

This powerful subtask management system and its associated abstractions provide easy access to operating system facilities for creating and controlling subtasks. This system allows a user to ignore many of the details neccessary for multi-tasking in a conventional language.

5. Applications

Our interface expands the scope of the Smalltalk programming environment. Operations and programs written in other languages are now within the reach of a Smalltalk programmer. Several applications are discussed in this section, including a window oriented operating system command interface.

As soon as the subtask implementation was started, we saw uses for it in the standard Smalltalk image. For instance, the **mkdir** operating system utility could be accessed from FileDirectory protocol to create directories. The capability to create directories requires special privileges in the operating system. Therefore, the system call to create directories could not be executed directly from Smalltalk, but required a program executing with system privileges. By giving Smalltalk the capability to invoke outside programs, subtasks have given Smalltalk the capability to perform actions requiring special privileges.

More sophisticated applications use the subtask implementation and related features. An application called *Hypertext* [Delisle] consists of a Smalltalk user interface and a distributed version control data system

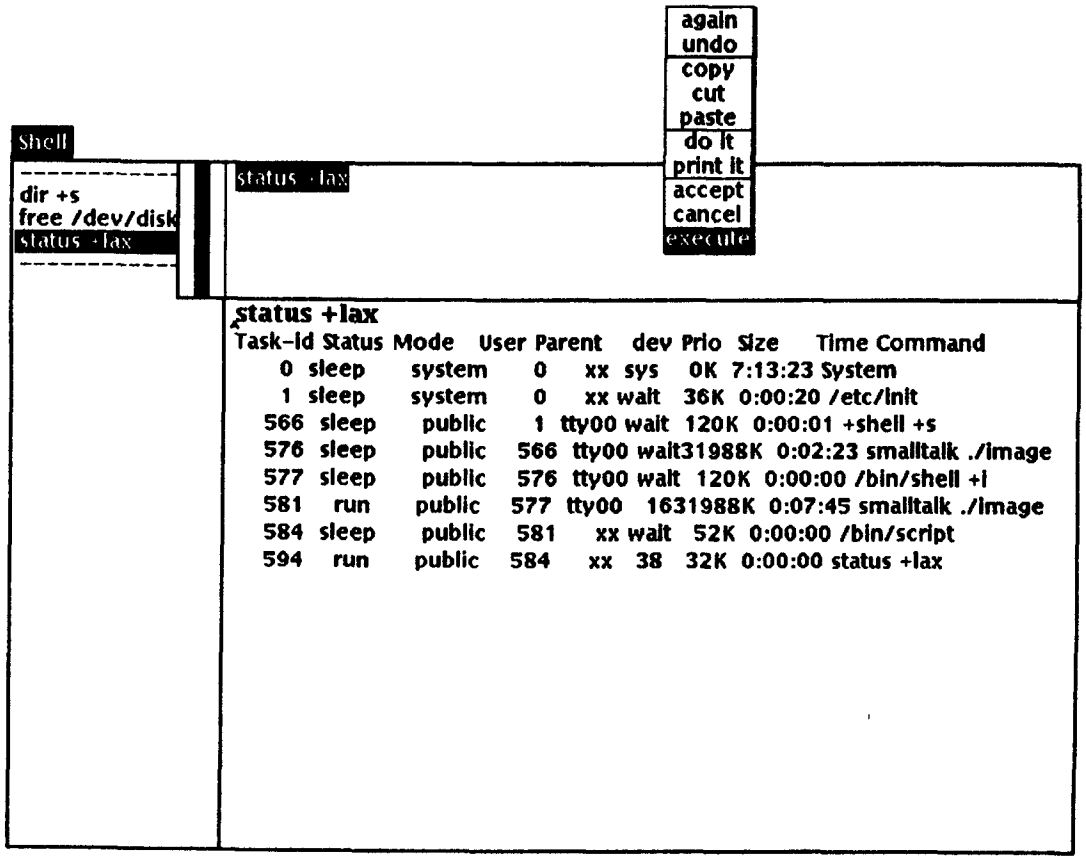

Figure 5. Windowed Interface to Shell

written in the C language. Pipes are used to communication between the user interface and the distributed system. *INKA* [Alexander] is an application using both Smalltalk and Prolog. Again, the user interface was implemented in Smalltalk and a knowledge-based expert system system was built in Prolog.

We also created a windowed interface to the operating system command interpretation program (See Figure 5). The command interpretation program, called shell, is an existing C language program. The shell program is interactive and the interface had to include mechanisms to reply to questions and to signal end-of-file to the shell. We also found that we needed a way to control the operating system tasks that the shell produced.

Making a subclass of FileModel allowed us to quickly create a windowed interface to the C language. This interface only included the edit/compile portion of the typical edit-compile-execute cycle. We determined that the execution of generic C programs was too varied to provide a general execution interface.

6. Conclusion

We have extended the Smalltalk-80 programming environment to provide access to the full capabilities of the underlying operating system via our operating system interface. In our interface, the system call portion serves as a basis for higher level abstractions. In particular, we provide an interface for subtasks, extending the raw operating system capabilities of multi-tasking with a complex subtask management system. We also provide higher level abstractions for communication features of the operating system such as pipes.

7. Acknowledgements

We thank Steve Messick (INKA application) and Norm Delisle (Hypertext application) for valuable

feedback on our interface. Ward Cunningham designed and implemented the interface to the file system. Pat Caudill was responsible for primitive implementation support. Allen Wirfs-Brock was instrumental in the design of the initial approaches.

References

[Goldberg] A. Goldberg & D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA. (1983).

[Thompson] K. Thompson, UNIX Time-Sharing System: UNIX Implementation. B.S.T.J. vol. 57 (1978)

[Delisle] N. Delisle & M. Schwartz, Hypertext system for design information: design and implementation. Applied Research Tech. Rep. CR-85-54 Tektronix, Inc., Beaverton OR (1985).

[Alexander] J. H. Alexander, M. J. Freiling, S. L. Messick, & S. Rehfuss, *Efficient expert system development through domain-specific tools.* Fifth International Workshop on Expert Systems and their Applications (1985).

APPENDIX

TekSystemCall and Subtask protocol

TekSystemCall class methodsFor: 'display operations'

blackOnWhite
Instantiate the receiver with data to perform a system call to set display to normal video.

clearAlarm
Instantiate the receiver with data to perform a system call to clear the alarm used by graphics.

cursorOff
Instantiate the receiver with data to perform a system call to turn off the cursor.

cursorOn
Instantiate the receiver with data to perform a system call to turn on the cursor.

disableCursorPanning
Instantiate the receiver with data to perform a system call to disable panning when the cursor reaches an edge.

TekSystemCall class methodsFor: 'system dependent operations'

alarm: seconds
Instantiate the receiver with data to perform a system call to set alarm to go off in seconds. Default alarm action is task termination. Return previous value of seconds in D0.

chacc: fileName mode: perminteger
Instantiate the receiver with data to perform a system call to check the accessibility of the file fileName.

chdir: directoryName
Instantiate the receiver with data to perform a system call to change working directory.

chown: fileName to: ownerID
Instantiate the receiver with data to perform a system call to change the owner of a fileName to ownerID.

chprm: fileName to: permissions
Instantiate the receiver with data to perform a system call to change permissions for a file.

close: fileDescriptor
Instantiate the receiver with data to perform a system call to Close a file.

TekSystemCall class methodsFor: 'portable operations'

changeDirectory: aString
Execute a system call to change directory to the specified directory.

closeFile: aFileDescriptor
Execute a system call to close the file referred to by aFileDescriptor.

create: aString
Execute a system call to create a new file named aString. Answer a writeOnly fileDescriptor for the file.

createDirectory: aString
Execute a system call to make aString, a full path name, be a new directory.

execSystemUtility: aCommand withArgs: anOrderedCollection
Perform system calls to execute a binary program and read the entire results through a pipe. No mechanism for input to the executable program is provided. Create an error if the program cannot be exec'ed or if the program terminates abnormally.

execUtilityWithErrorMapping: aCommand withArgs: anOrderedCollection
Perform a system call to execute a binary program and read the standard output through a pipe. Read standard error through a pipe. No mechanism for input to the executable program is provided. Create an error if the program cannot be exec'ed or if the program terminates abnormally. Answer an array with a string from standard output and a string from standard error.

existingName: fileName
Execute a system call to determine if a file or directory exists with the name fileName, a String.

nextFileName: directoryStream
Answer the next file name in directoryStream. Answer nil if none.

open: aString
Execute a system call to open the file named aString. Answer a readWrite fileDescriptor for the file.

openForRead: aString
Execute a system call to open the file named aString. Answer a readOnly fileDescriptor for the file.

openForWrite: aString
Execute a system call to open the file named aString. Answer a writeOnly fileDescriptor for the file.

read: fileDescriptor into: aStringOrByteArray
Execute a system call to fill aStringOrByteArray with data from the file referred to by fileDescriptor. Answer the number of bytes read. Answer zero if at end.

remove: aFileName
Execute a system call to remove the file named aFileName.

removeDirectory: aString
Execute system calls to remove the directory named aDirectoryName. The directory must be empty (i.e. contain only . and ..).

rename: aFileName as: newFileName
Execute system calls to rename the file named aFileName to have the name newFileName. Create an error if aFileName does not exist; but not if newFileName exists.

seek: aFileDescriptor to: aFilePosition
Execute a system call to position the file represented by aFileDescriptor to aFilePosition bytes from its beginning.

setInterrupt: anInteger to: anAddressOrSemaphore
Execute a system call to override the default action for an interrupt by connecting it to a semaphore or an address. If the interrupt is connected to a semaphore, the semaphore is posted on interrupt, or if the interrupt is connected to an address, a branch to the specified address is performed on interrupt. After the interrupt is received, the interrupt must be reconnected or it will return to its default action. However, the DeadChildInterrupt, number 26, does not need to be reconnected since its action does not return to the default.

shorten: fileDescriptor
Execute a system call to shorten a file to its current position. Preserved for historical reasons.

status: fileDescriptor
Execute a system call to answer a FileStatus for the file referred to by fileDescriptor, a SmallInteger.

statusName: fileName
Execute a system call to answer a FileStatus for the file referred to by fileName, a String.

write: fileDescriptor from: aStringOrByteArray size: byteCount
Execute a system call to write byteCount bytes of data from aStringOrByteArray to the file referred to by fileDescriptor.

TekSystemCall class methodsFor: 'general inquiries'

maxNameSize
Answer the maximum number of characters permissible for file names.

maxOpenFiles
Answer the maximum number of simultaneously open files.

statusClass
Answer the class whose instances hold the file status returned by system calls that are instances of the receiver.

suffixString
Answer the string appended for backup file names.

TekSystemCall methodsFor: 'accessing'

A0Out
Answer the contents of the A0 register. It may be a boolean indicating if a value should be returned, or it may be the returned value.

D0Out
Answer the contents of the D0 register. It may be a boolean indicating if a value should be returned, or it may be the returned value.

D1Out
Answer the contents of the D1 register. It may be a boolean indicating if a value should be returned, or it may be the returned value.

errno
Answer the error number set by a system call or nil if no error.

operation
Answer the numeric code for this call.

TekSystemCall class methodsFor: 'environments'

argCount
Answer the argument count in the invocation of this image

originalEnvironment
Answer the environment variables passed to this invocation of Smalltalk.

TekSystemCall class methodsFor: 'current working directory'

currentDirectory
Answer the complete pathname of the current working directory.

TekSystemCall methodsFor: 'execution'

invoke
Evaluate the system call represented by the receiver. Answer true if the system call executed without error. Answer false otherwise.

value
Evaluate the system call represented by the receiver. Answer the result of a successful call; notify error otherwise.

valueIfError: aBlock
Evaluate the system call represented by the receiver. Evaluate aBlock if the system call returns an error.

TekSystemCall methodsFor: 'errors'

errorString
Answer the error string associated with the returned error number.

isInterrupted
Answer if the system call was interrupted.

issueError
Pop up a notifier indicating the error from the system call.

Subtask class methodsFor: 'instance creation'

fork: commandName then: aBlock
Set the task object to contain all the necessary information to exec the subtask. There are no arguments.

fork: commandName withArgs: args
Set the task object to contain all the necessary information to exec the subtask. The initialization block is set to an empty block.

fork: commandName withArgs: args standardIn: in standardOut: out standardError: err
Set the task object to contain all the necessary information to exec the subtask. Map standard file descriptors to the specified descriptors. The initialization block is set to an empty block.

fork: commandName withArgs: args then: aBlock
Set the task object to contain all the necessary information to exec the subtask.

fork: commandName withArgs: args withEnv: anEnvironment then: aBlock
Set the task object to contain all the necessary information to exec the subtask including the specified environment..

Subtask class methodsFor: 'environments'

copyEnvironment
Answer the environment with which smalltalk was invoked, in dictionary format.

currentEnvironment
Answer the environment with which smalltalk was invoke, in a format
readable by the OS

initializeEnvironment
Initialize the internal record of the environment with which this image was invoked.

Subtask class methodsFor: 'task management'

initializeWaitManagement
If ChildWaitSemaphore has received a signal indicating a child task death, wait management is performed in the method wait.

kill: aTask
Unconditionally kill the specified task.

markAndSignalAll
Mark the status and signal the wait semaphore of each previously scheduled subtask. This method is used for restoring after a snapshot.

terminate: aTask
Terminate the specified task.

terminateAll
Kill all the scheduled tasks.

waitUpdate
If ChildWaitSemaphore has received a signal indicating a child task death, a wait system call is made. The scheduled subtask is updated accordingly and the status
of the termination is recorded.

Subtask methodsFor: 'initialize-release'

initialize
Set the subtask data to reflect the parent's environment.

release
Remove the receiver from the list of scheduled subtasks.

Subtask methodsFor: 'accessing'

enhancedPriority
Set the priority of the subtask to the lowest possible value, giving the highest possible priority.

environment
Answer the environment for this subtask in an array format.

environment: env
Store and return the OS readable version (Array format) of the environment for this subtask. See environments protocol in the metaClass.

initBlock
Answer the block to be evaluated between vfork and exec system calls.

initBlock: aBlock
Assign aBlock to the instance variable initBlock.

priority
Answer the absolute priority of the subtask.

priority: aPriority
Set the priority of the subtask. Acceptable values range from 0 to 25, zero being the highest and 25 being the lowest.

program
Answer the program of the subtask.

status
Answer the status of the receiver. Usually this instance variable is accessed with the protocol criticalProtect: .

status: stat
Record the status of the receiver. Usually this instance variable is accessed with the protocol criticalProtect: .

taskId
Answer an identifying unique integer assigned by the operating system.

taskId: id
Set my identifying integer.

terminatedWith: terminateStat
Record the status of the terminated task. Use method criticalProtect: to avoid inconsistencies in status.

Subtask methodsFor: 'testing'

abnormalTermination
Answer the status of the termination if abnormal, otherwise nil.

isTerminated
Answer a boolean indicating if the receiver has been terminated.

Subtask methodsFor: 'critical accessing'

criticalSection: aBlock
Use the receiver's accessing semaphore to evaluate a block. Usually this block concerns the instance variable status.

Subtask methodsFor: 'controlling'

absoluteWait
Bypassing the dead child signal management, a wait system call is made. The list of scheduled subtask is updated accordingly and the status of the
termination is recorded for each child process until the receiver's subtask is found. This method replaces the use of the waitOn message. For use with non-interactive tasks that require the shutting down of the Smalltalk process for efficiency.

interrupt: anInterrupt
Send an interrupt to myself. AnInterrupt is an integer indicating which kind of interrupt is to be sent. The usual result of an interrupt is for the task to terminate.

kill
Kill this task unconditionally

signalWaitSemaphore
Send a signal to my suspnding semaphore.

start
Start the receiver by executing a vfork, code to set up the child task (mainly communication and signal processing), and exec'ing the program. If the exec fails terminate the child task. The child task will inherit the priority of the smalltalk process.

terminate
Attempt to terminate the receiver's task using a soft kill

waitOn
Wait for the receiver to terminate.