21

Evolving the UNIX System Interface to Support Multithreaded Programs

Paul R. McJones and Garret F. Swart

September 28, 1987
(reformatted for electronic distribution August 28, 1997)

digital

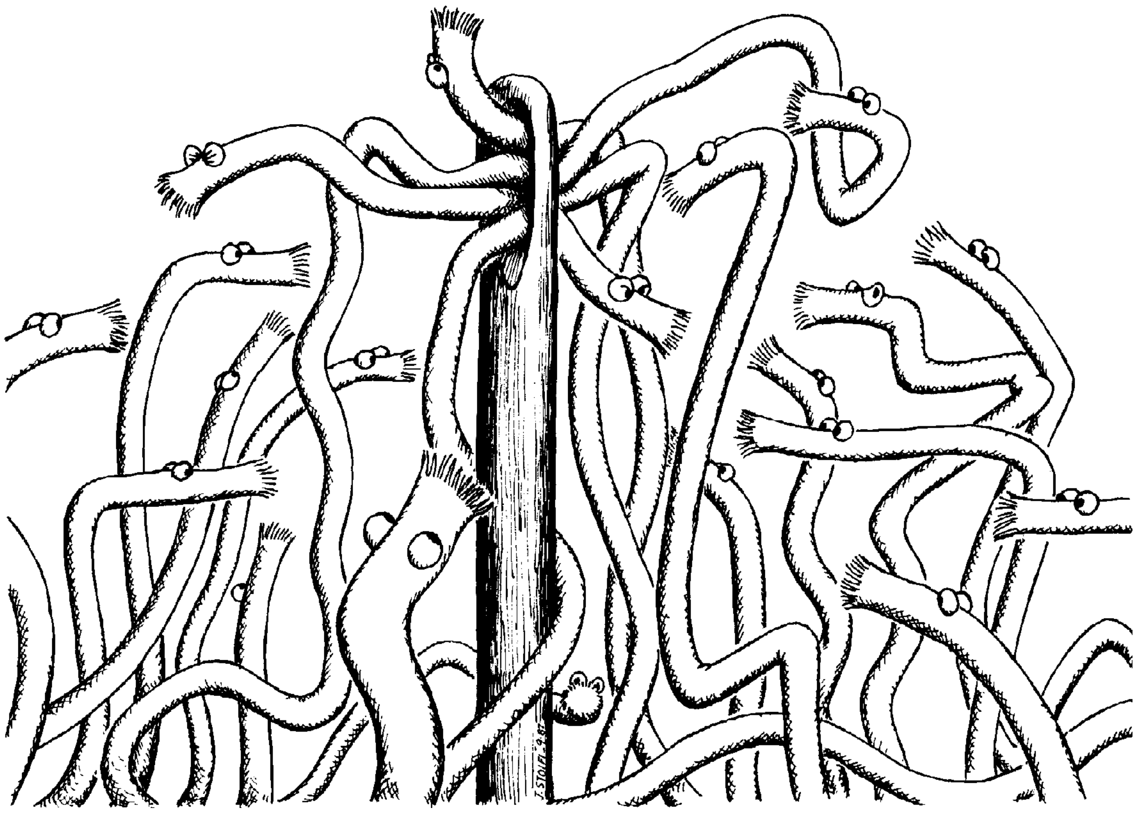
Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Evolving the UNIX System Interface to Support Multithreading Programs

Paul R. McJones and Garret F. Swart

September 28, 1987 (reformatted for electronic distribution August 28, 1997)



A slightly-edited version of Part I of this report appeared as “Evolving the UNIX System Interface to Support Multithreaded Programs” in *Proceedings of the Winter 1989 USENIX Technical Conference*, January 1989, pages 393-404.

This edition of August 28, 1997 is identical to the original except for layout differences resulting from switching from Scribe to LaTeX.

Author’s Abstract

Multiple threads (program counters executing in the same address space) make it easier to write programs that deal with related asynchronous activities and that execute faster on shared-memory multiprocessors. Supporting multiple threads places new constraints on the design of operating system interfaces. Part I of this report presents guidelines for designing (or redesigning) interfaces for multithreaded clients. We show how these guidelines were used to design an interface to UNIX[1]-compatible file and process management facilities in the Topaz operating system. Two implementations of this interface are in everyday use: a native one for the Firefly multiprocessor, and a layered one running within a UNIX process. Part II is the actual programmer’s manual for the interface discussed in Part I.

Paul R. McJones and Garret F. Swart

Capsule review

Unix was designed to support client address spaces containing exactly one program counter. This assumption about a Unix process has had a remarkably pervasive influence on the interface to the Unix kernel. This report documents a successful project to provide an alternative interface, for multithreaded address spaces, to the Unix file and process facilities. Single-threaded processes using the Unix kernel interface and multi-threaded processes using the alternative can cohabit the same system and cooperate.

Part I of the report is a paper that discusses the general issues and solutions. Part II is a programmer’s manual for the new interface. The inclusion of this second part will allow the interested reader to answer quite detailed questions about how the system may be used. The techniques discussed in Part I will apply to efforts to provide a more modern, multi-threaded interface to other operating systems with single-threaded client interfaces.

Michael Schroeder

[1]UNIX is a trademark of AT&T Bell Laboratories

Contents

Part I

Evolving the UNIX System Interface to Support Multithreaded Programs

1 Introduction

Most existing general-purpose operating systems provide rich collections of tools and applications, but lack the support for fine-grain concurrency that would ease the construction of applications dealing with asynchrony and allow the exploitation of multiprocessors. Multiple threads (program counters sharing an address space) provide such support, but at the cost of placing additional constraints on the design of operating system interfaces. In this paper we present several guidelines that we used to design the multithreaded operating system interface of the Topaz system built at DEC's Systems Research Center. We show how we used these guidelines to evolve the Topaz interface from the 4.2BSD UNIX [7, 12] system interface. We believe the guidelines will be useful for adding multithreading to other operating systems.

One implementation of Topaz runs as the native operating system on SRC's Firefly multiprocessor [18] and allows concurrent execution on separate processors of multiple threads within the same address space. A second implementation of Topaz is layered on 4.2BSD UNIX; it uses multiprogramming techniques to create multiple threads within a single UNIX process. Both implementations make it convenient to compose single-threaded UNIX programs and multithreaded programs using the standard UNIX process composition mechanisms.

Topaz is an extension of the architecture of an existing system rather than an entirely new design because of the dual role it plays at SRC. Topaz serves both as the base for research into distributed systems and multiprocessing and also as the support for SRC's current computing needs (mainly document preparation, electronic mail, and software development). When experimental software can be put into everyday use on the same system that runs existing tools and applications, it is easier to get relevant feedback on that software.

There were several reasons for choosing UNIX in particular as an architectural starting point. The machine-independence of UNIX left the way open for future work at SRC on processor design. UNIX also offered a large set of tools and composition mechanisms, and a framework for exchanging ideas about software throughout the research community.

Section 2 gives a brief overview of Topaz, to set the stage for the rest of the paper. Sections 3 and 4 constitute the heart of the paper: our guidelines for multithreaded interfaces and our use of those guidelines in designing the Topaz operating system interface. Section 5 draws some conclusions about the approach taken in Topaz.

2 Topaz Overview

One way of viewing Topaz is as a hybrid of Berkeley's 4.2BSD UNIX [7, 12] and Xerox's Cedar [16]. Topaz borrows the 4.2BSD file system and large-grain process structure, populates these processes (address spaces) with Cedar-like threads, and interconnects them with Cedar-like remote procedure call. Topaz allows single-threaded programs using the standard 4.2BSD system interface and multithreaded programs using a new Topaz operating system interface to run on the same machine, to share files, to send each other signals, and to run each other as processes.

A Topaz address space has most of the same state components as a UNIX process has, such as virtual memory, a set of open files, a user id, and signal-handling information, but is considered as more "heavyweight" than a UNIX process. Like a Cedar programmer, a Topaz programmer can use threads for fine-grained cooperation within a single address space. Unlike a Cedar programmer, a Topaz programmer can also use multiple address spaces to separate programs of different degrees of trustworthiness.

Multiple threads address a problem different from the one addressed by the shared memory segments provided by some versions of UNIX, such as System V [2]. While shared segments are useful in allowing separately developed application programs to have access to a common data structure, as for example a database buffer pool, multiple threads are intended to be a "lightweight" control structure for use within a single program. Modeling threads as separate UNIX processes would place undesirable restrictions on sharing open files and pointer-containing data structures among threads. Creating the dozens or even hundreds of threads used by some Topaz applications would be slow and extravagant of kernel resources even if most of the virtual memory resources could be shared.

The semantics of Topaz threads assumes as many processors as threads; the implementation assigns threads to actual processors. Threads sharing variables must therefore explicitly synchronize. The synchronization primitives provided (mutexes, conditions, and semaphores) are derived from Hoare's monitors [6], following the modifications of Mesa [10]; the details are described by Birrell et al. [4].

Support for multiprocessors in UNIX has evolved over a number of years. Early multiprocessor implementations of UNIX allowed concurrent execution of single-threaded processes but didn't support multiple threads. Many of these implementations serialized execution within the system kernel; Bach and Buroff [3] describe one of the first implementations to allow concurrency within the kernel. Several current systems, such as Apollo's Concurrent Programming Support [1] and Sun's "lightweight process" (lwp) facility [8], support multiple threads within a UNIX process, but can't assign more than one thread within an address space to a processor at any one time. Like the Firefly implementation of Topaz, C-MU's Mach [13] supports concurrent execution of threads within an address space on a multiprocessor. The approach taken by Apollo, Sun, and Mach in adding threads to UNIX is to minimize the impact on the rest of the system interface, to make it easier to add the use of multiple threads to large existing programs. In contrast, the approach taken in Topaz is to integrate the use of threads with all the other programming facilities.

3 Guidelines for Multithreaded Interfaces

By a *multithreaded interface* we mean one usable by multithreaded clients. Good interface design is a challenging art, and has a whole literature of its own (for example, see Parnas [11] and Lampson [9]). We have not found designing multithreaded interfaces to be very much more difficult than designing single-threaded ones. In one respect it is harder: care must be taken that serializing access to shared mutable state defined by the interface does not restrict the usefulness of that interface. In another respect it is easier: a multithreaded interface can avoid the various ad hoc techniques used when a single-threaded interface must deal with concurrency inherent in the application. Threads provide a convenient framework for designing solutions to the problem of cancelling undesired computations.

3.1 Sharing Mutable State

It is not uncommon for a single-threaded interface to reference a state variable that affects one or more procedures of the interface. The purpose is often to shorten calling sequences by allowing the programmer to omit an explicit argument from each of a sequence of procedure calls in exchange for occasionally having to set the state variable. To avoid interference over such a state variable, multiple client threads must often serialize their calls on procedures of the interface even when there is no requirement for causal ordering between the threads.

One example of interference caused by an interface state variable is the stream position pointer within a UNIX open file [14]. The pointer is implicitly read and updated by the stream-like **`read`** and **`write`** procedures and is explicitly set by the **`seek`** procedure. If two threads use this interface to make independent random accesses to the same open file, they have to serialize all their **`seek-read`** and **`seek-write`** sequences. Another example is the UNIX library routine **`ctime`**, which returns a pointer to a statically allocated buffer containing its result and so is not usable by concurrent threads.

While it is important to avoid unnecessary serialization of clients of an interface, serialization within the implementation of a multithreaded interface containing shared data structures is often necessary. This is to be expected and will often consist of fine-grain locking that minimizes interference between threads.

We can think of four basic approaches to designing multithreaded interfaces so as to minimize the possibility of interference between client threads over shared mutable state:

1. Make it an argument. This is the most general solution, and has the advantage that one can maintain more than one object of the same type as the shared mutable state being replaced. In the file system example, passing the stream position pointer as an argument to **`read`** and **`write`** solves the problem. Or consider a pseudo-random number generator with a large amount of hidden state. Instead of making the client synchronize its calls on the generator, or even doing the synchronization within the generator, either of which may slow down the application, a better solution is to allow more than one generator state and to pass a reference to that state to the generator on each call.

2. Make it a constant. It may be that some state component need not change once an application is initialized. An example of this might be the user on whose behalf the application is running.

3. Let the client synchronize. This is appropriate for mutable state components that are considered inherently to affect an entire application, rather than to affect a particular action being done by a single thread.

4. Make it thread-dependent, by having the procedure use the identity of the calling thread as a key to look up the variable in a table. Adding extra state associated with every thread adds to the cost of threads, and so should not be considered lightly. Having separate copies of a state variable can also make it more difficult for threads to cooperate in manipulating a single object.

It is a matter of judgment which of these techniques to use in a particular case. We used each of the four in designing the Topaz operating system interface. Sometimes providing a combination offers worthwhile flexibility. For example, a procedure may take an optional parameter that defaults to a value set at initialization time. Also, it is possible for a client to simulate thread-dependent behavior using a procedure taking an explicit parameter in conjunction with an implementation of a per-thread property list (set of tag-value pairs).

3.2 Avoiding Ad Hoc Multiplexing

Although most operating systems provide only a single thread of control within each address space, application programs must often deal with a variety of asynchronous events. As a consequence, many operating systems have evolved a set of ad hoc techniques for multiplexing the single thread within an address space. These techniques have the disadvantage that they add complexity to applications and confuse programmers. To eliminate the ad hoc techniques, multiple threads can be used, resulting in simpler, more reliable applications.

The aim of all the ad hoc multiplexing techniques is to avoid blocking during a particular call on an operating system procedure when the client thread could be doing other useful work (computing, or calling a different procedure). Most of the techniques involve replacing a single operating system procedure that performs a lengthy operation with separate methods for initiating the operation and for determining its outcome. The typical methods for determining the outcome of such an asynchronous operation include:

Polling. Testing whether or not the operation has completed, as by checking a status field in a control block that is set by the operation. Polling is useful when the client thread wants to overlap computation with one or more asynchronous operations. The client must punctuate its computation with periodic calls to the polling procedure; busy waiting results when the client has no other useful computation. Note that busy waiting is undesirable only when there is a potential for the processor to be used by another process.

Waiting. Calling a procedure that blocks until the completion of a specified operation, or more usefully one of a set of operations. Waiting procedures are useful when the client thread is trying to overlap a bounded amount of computation with one or more asynchronous operations, and must avoid busy waiting.

Interrupts. Registering a procedure that is called by borrowing the program counter of the client thread, like a hardware interrupt. Interrupts are useful in overlapping computation with asynchronous operations. They eliminate busy waiting and the inconsistent response times typical of polling. On the other hand, they make it difficult to maintain the invariants associated with variables that must be shared between the main computation and the interrupt handler.

The techniques are often combined. The VMS **`$QIO`** service is capable of reporting I/O completion via any combination of the following: setting a status field, unblocking a wait on an event flag, and delivering an AST (software interrupt) [5]. Polling, waiting, and interrupt mechanisms are also used in 4.2BSD UNIX. When an open file has been placed in non-blocking mode, the **`read`** and **`write`** operations return an error code if a transfer is not currently possible. Non-blocking mode is augmented with two ways to determine a propitious time to attempt another transfer. The **`select`** call waits until a transfer is possible on one of a set of open files. When an open file has been placed in asynchronous mode, the system sends a signal (software interrupt) when a transfer on that file is possible.

When multiple threads are available, it is best to avoid all these techniques and instead model each operation as a single, synchronous procedure. This is simple for naive clients, and allows more sophisticated clients to use separate threads to overlap lengthy system calls and computation.

3.3 Cancelling Operations

Many application programs allow the cancellation of a command in progress. For example, the user may decide not to wait for the completion of a computation or the availability of a resource. In order to allow prompt cancellation, an application needs a way of notifying all the relevant threads of the change in plans.

If the entire application is designed as one large module, then state variables, monitors, and condition variables may be enough to implement cancellation requests. However if the application is composed of lower-level modules defined by interfaces, it is much more convenient to be able to notify a thread of a cancellation request without regard for what code the thread is currently executing.

The Modula-2+ language [15] in which Topaz programs are written provides the *alert* mechanism [4] for this purpose (Mesa provides a similar *abort* mechanism [10]). Sending an alert to a thread simply puts it in the alerted state. A thread can atomically test-and-clear its alerted status by calling a procedure **`TestAlert`**. Of course this is a form of polling, and isn't always appropriate or efficient. To avoid the need to poll, there exist variants of the procedures for waiting on condition variables and semaphores. These variants, **`AlertWait`** and **`AlertP`**, return prematurely with a special indication if the calling thread is already in, or enters, the alerted state. The variants also clear the alerted status. We refer to the procedures **`TestAlert`**, **`AlertWait`**, and **`AlertP`**, and to procedures that call them, as *alertable*.

What then is the effect of alerts on interface design? Deciding which procedures in an interface should be alertable requires making a trade-off between the ease of writing responsive programs and the ease of writing correct programs. Each call of an alertable procedure provides another point at which a computation can be cancelled, but each such call also requires the caller to design code to handle the two possible outcomes: normal completion and an alert being reported. We have formulated the following guidelines for using alerts, which in some sense lead to the minimum set of alertable procedures necessary to allow top-level programs to cancel operations:

1. Only the owner of a thread, that is the program that forked it, should alert the thread. This is because an alert carries no parameters or information about its sender. A corollary is that a procedure that clears the alerted status of a thread must report that fact to its caller, so that the information can propagate back to the owner.

2. Suppose there is an interface M providing a procedure P that does an unbounded wait, that is a wait whose duration cannot be bounded by appeal to M's specification alone. Then M should provide alertable and nonalertable variants of the procedure, just as Modula-2+ does for waits on condition variables and semaphores. (The interface might provide either separate procedures or one procedure accepting an "alertable" Boolean parameter.) A client procedure Q should use the alertable variant of P when it needs to be alertable itself and cannot determine a bound on P's wait.

3. A procedure that performs a lengthy computation should follow one of two strategies. It can allow partial operations, so that its client can decompose a long operation into a series of shorter ones

separated by an alert test. Or it can accept an “alertable” Boolean parameter that governs whether the procedure periodically tests for alerts.

If all interfaces follow these rules, a main program can always alert its worker threads with the assurance that they will eventually report back. The implementation of an interface might choose to call alertable procedures in more cases than required by the second guideline, gaining quicker response to alerts at the cost of more effort to maintain its invariants.

4 Topaz Operating System Interface

Topaz programs are written in Modula-2+ [15], which extends Wirth’s Modula-2 [19] with concurrency, exception handling, and garbage collection. The facilities of Topaz are provided through a set of interf aces, each represented by a Modula-2+ definition module.

This section describes the Topaz OS interface, which contains the file system and process (address space) facilities. We focus here on how the presence of multiple threads affected the evolution of the OS interface from the comparable 4.2BSD facilities. More information about the Topaz OS interface can be found in its reference manual, included as Part II of this report.

4.1 Reporting Errors

A UNIX system call reports an error by storing an error number in the variable **`errno`** and then returning the value -1. The variable **`errno`** causes a problem for a multithreaded client, since different values could be assigned due to concurrent system calls reporting errors. (Another source of confusion results from system calls that can return -1, e.g., **`nice`** or **`ptrace`**.)

A workable but clumsy solution would be for every system call that could report an error to return an error code via a result parameter. We chose to use Modula-2+ exceptions instead. Exceptions have the advantage over return codes that they can’t be accidentally ignored, because an exception for which no handler exists results in abnormal termination of the program.[2] Exceptions are handled by enclosing a statement sequence that could raise an exception with a **`TRY ... EXCEPT`** statement.

A Modula-2+ procedure declaration may include a **`RAISES`** clause enumerating the exceptions the procedure may raise. The declaration of an exception may include a parameter, allowing a value to be passed to the exception handler. Most procedures in the Topaz operating system interface can raise the exception **`Error`**, which is declared with a parameter serving as an error code, analogous to the UNIX error number. Topaz defines the exception **`Alerted`** for reporting thread alerts (discussed in Section 3.3). Each procedure in the Topaz operating system interface that may do an unbounded wait includes **`Alerted`** in its **`RAISES`** clause. As described in Section 4.3, Topaz also uses exceptions to report synchronous events such as hardware traps.

4.2 File System

A UNIX process contains several components of mutable file system state that would cause problems for multithreaded programs, including the working directory, the table of file descriptor references, and the stream position pointer inside each file descriptor. The Topaz design has made adjustments for each of these.

A UNIX path name is looked up relative to the file system root if it begins with “/”; otherwise it is looked up relative to the working directory. Each process has its own working directory, which is initially equal to the parent’s and may be changed using the **`chdir`** system call. Since looking up a short relative path name can be significantly faster than looking up the corresponding full path name, some UNIX programs use the working directory as a sort of “cursor”, for example when enumerating a subtree of the file system. To facilitate multithreaded versions of such programs (and modular programming in general), Topaz parameterizes the notion of working directory. The **`OpenDir`** procedure accepts the path

[2]One reason why UNIX uses signals to report certain synchronous events such as **`SIGPIPE`** is because its clients are notorious for not checking return codes.

name of a directory, and returns a handle for that directory. Every procedure that accepts a path name argument also accepts a directory handle argument that is used when the path name doesn't begin with "/". The distinguished directory handle **`NIL`** can be used to refer to the initial working directory supplied when the process was created.

Part of the state maintained by UNIX for each process is a table with an entry for each open file held by the process. An application program uses small nonnegative integer indexes in this table to refer to open files. In a multithreaded application it is desirable to avoid the need to serialize sequences of operations affecting the allocation of table entries (e.g., **`open`**, **`dup`**, and **`close`**). To achieve this goal, the table indexes should be treated as opaque quantities: it should not be assumed that there is a deterministic relationship between successive values returned by operations such as **`open`**. (Single-threaded UNIX programs actually depend on being able to control the allocation of table indices when preparing to start another program image. Topaz avoids this dependency, as described in Section 4.4.)

Recall from the example in Section 3.1 that the stream position pointer in a UNIX file descriptor causes interference when threads share the descriptor. Topaz still implements these pointers so that Topaz and UNIX programs can share open files, but to allow multiple threads to share a file descriptor without having to serialize, Topaz provides additional procedures **`FRead`** and **`FWrite`** that accept a file position as an extra argument.

The 4.2BSD UNIX file system interface contains a number of ad hoc multiplexing mechanisms that are described in Section 3.2. These mechanisms allow a single-threaded UNIX process to overlap computation and input/output transfers that involve devices such as terminals and network connections. Topaz simply eliminates these mechanisms (non-blocking mode, the **`select`** procedure, and asynchronous mode) and substitutes **`Read`** and **`Write`** procedures that block until the transfer is complete. **`Read`** and **`Write`** are alertable when a transfer is not yet possible. Note that Topaz violates guideline 2 of Section 3.3 by not providing nonalertable variants of **`Read`** and **`Write`**. (For completeness, Topaz provides a **`Wait`** procedure that waits until a specified open file is ready for a transfer.)

4.3 Signals

A UNIX signal is used to communicate an event to a process or to exercise supervisory control over a process (termination or temporary suspension). A UNIX signal communicates either a synchronous event (a trap, stemming directly from an action of the receiving process) or an asynchronous one (an interrupt, stemming from another process, user, or device).

UNIX models signal delivery on hardware interrupts. A process registers a handler procedure for each signal it wants to handle. When a signal is received, the current computation is interrupted by the creation of an activation record for the handler procedure on the top of the stack of the process. This handler procedure may either return normally, resulting in the interrupted computation continuing, or may do a "long jump", unwinding the stack to a point specified earlier in the computation. If a signal is received for which no handler procedure was registered, a default action takes place. Depending on the signal, the default action is either to do nothing, to terminate the process, to stop the process temporarily, or to continue the stopped process. Following the hardware interrupt model, 4.2BSD UNIX allows each signal to be ignored or temporarily masked.

Topaz signals are patterned after UNIX signals, and in fact Topaz and UNIX programs running on the same machine can send each other signals. However, UNIX signal delivery is another ad hoc way of multiplexing the single program counter of a process. Trying to use interrupt-style signal delivery in a multithreaded environment leads to problems. Which thread should receive the signal? What does a signal handler procedure do if it needs to acquire a lock held by the thread it has interrupted? Rather than answering these questions, we avoided them.

A Topaz process can specify that it wants to handle a particular signal, but it doesn't register a handler procedure. Instead, it arranges for one of its threads to call **`WaitForSignal`**. This procedure blocks until a signal arrives, then returns its signal number. The calling thread then takes whatever action is appropriate, for example initiating graceful shutdown. **`WaitForSignal`** takes a parameter that specifies a subset of the handled signals, so a program may have more than one signal-handling thread. The set of signals that it makes sense to handle is smaller in Topaz than in UNIX, since those used as part of various UNIX ad hoc multiplexing schemes (e.g., **SIGALRM**, **SIGURG**, **SIGIO**, and **SIGCHLD**) are never sent to multithreaded processes. Topaz provides the same default actions as UNIX for signals not handled by the process. The decision about which signals to handle and which to default is necessarily global to the entire process; any dynamic changes must be synchronized by the client.

UNIX system calls that do unbounded waits (e.g., reading from a terminal or waiting for a child process to terminate) are interruptible by signals. But this interruptibility leads to difficulties that are avoidable in the multithreaded case. A client program will normally want to restart a system call interrupted by a signal that indicates completion of some asynchronous operation, but will probably not want to restart a system call interrupted by a signal that indicates a request for cancellation of a computation. Different versions of UNIX have tried different approaches to the restartability of system calls. In Topaz, there is no need for signal delivery itself to interrupt any system call. The signal handling thread may decide to alert one or more other threads, which raises an **Alerted** exception in any thread doing an unbounded wait in a system call.

Instead of using signals to report synchronous events, Topaz uses Modula-2+ exceptions. For example, the **AddressFault** exception is raised when a thread dereferences an invalid address. Since the contexts statically and dynamically surrounding where an exception is raised determine what handler is invoked for that exception, different threads can have different responses.

Modula-2+ exceptions are based on a termination model: scopes between the points where an exception is raised and where it is handled are *finalized* (given a chance to clean up and then removed from the stack) before the handler is given control. While this model has proven its worth in constructing large modular systems, it does lead to a complication involving traps. An exception is an appropriate way to report a trap considered to be an error, but isn't appropriate for a trap such as breakpoint or page fault whose handler wishes to resume. Topaz therefore provides a lower-level trap mechanism that suspends a thread at the point of the trap and then wakes up a trap-handling thread. The trap-handling thread either converts the trap to an exception in the trapping thread, or handles it and restarts the thread, as appropriate.

4.4 Process Creation

UNIX provides simple but powerful facilities for creating processes and executing program images, involving three main system calls: **fork**, **wait**, and **exec**. **fork** creates a child process whose memory, set of open files, and other system-maintained state components are all copied from the calling process. **wait** waits for the next termination of a child of the calling process; there is also a nonblocking form of **wait** and a signal **SIGCHLD** sent by the system whenever a child process terminates. **exec** overlays the address space of the calling process with a new program image.

Typically UNIX programs use these facilities in one of two stylized ways. One is to create an extra thread of control; for example a "terminal emulator" program uses a pair of processes for full-duplex communication with a remote system. The other is to run a new program image; for example, the shell runs each command in a new child process.

In Topaz, the way to create an extra thread of control is simply to fork a new thread within the same process. Topaz must still provide a mechanism for running a new program image, and the UNIX method won't do. In UNIX, the parent calls **fork**; the child (initially executing the same program image as the parent) makes any necessary changes to its process state (e.g., opening and closing files or changing the user id), and finally calls **exec** to overlay itself with the new program image. Since the **fork**-**exec** sequence involves a large amount of shared mutable state (the entire child process), it isn't surprising that it doesn't work for Topaz. Would **fork** copy all threads? In the Apollo and Mach systems, only the thread that calls **fork** is copied [17]. But what happens if locks were held by other threads in the parent process? If **fork** copied all threads, what would it mean to copy a thread blocked in a system call?

To address this problem, Topaz replaces **fork** and **exec** with a single new procedure **Start-Process**, which accepts as parameters all the modifiable components of a process state (namely anything that could be changed between a call to **fork** and the subsequent call to **exec**, such as the set of open files and the user id). Topaz also replaces **wait** with **WaitForChild**, which waits for the termination of a specific child process.

There are several reasons for not merging the functions of **StartProcess** and **WaitForChild** into a single procedure that would block until termination of the child process. As it stands, **`StartProcess`** returns the process identifier of the new process and **`WaitForChild`** returns a status value indicating whether the child terminated or temporarily stopped. These features allow a process to be observed and controlled while it executes, in a way compatible with 4.2BSD job control.

4.5 Other Process State

The preceding subsections describe how Topaz treats many of the mutable state components of a UNIX process. For completeness, here is how the remaining components are treated.

User identity. Constant/parameterized. Each process has a user identity associated with it. Super-user programs, usually servers, acting on behalf of many users may pass an additional parameter on each call giving the user identity on whose behalf they are acting.

UMask. Client-synchronized. (The umask is used to set the access control bits when a file is created.)

Process Group. Constant.

Control Terminal. Constant.

Priority. Thread-dependent.

4.6 Summary

We can summarize the synchronization changes in the Topaz operating system interface as follows:

- A computation is overlapped by performing it in a separate thread.
- An asynchronous event is delivered by unblocking a client thread.
- A synchronous event is delivered by returning a value or raising an exception in the responsible thread.

5 Conclusions

The implementation of Topaz for SRC's Firefly multiprocessor has been in daily use since the spring of 1986, and the version of the multithreaded operating system interface described here has been in use since the spring of 1987. A number of multithreaded application programs and servers have been written for Topaz.

We consider both the idea of a multithreaded extension of UNIX and our new operating system interface to be successes. Users have access to the large collection of UNIX application software, are free to investigate the consequences of multiple threads and remote procedure call for building new applications, and are often able to use UNIX and Topaz applications together. For example most Firefly users use the standard C shell both interactively and through shell scripts to run a mixture of UNIX and Topaz applications. Most Fireflies run a Topaz "distant process" server that allows a user to run arbitrary processes on idle machines throughout the local network.

Supporting multiple threads instead of a single thread adds little to the inherent execution-time cost of the system-call interface. And of course good speedups are possible when clients take advantage of opportunities for concurrent execution. For example, a Topaz command called *updatefs* compares two file systems, bringing one up-to-date with respect to the other. Changing *updatefs* to use concurrent threads to traverse the two trees and compare file modification times resulted in a speed-up of 3 to 4 (on a 5-processor Firefly).

The implementation of Topaz layered on UNIX allows us to write servers (e.g., for remote file access and remote login) that run on Fireflies and on VAX/UNIX systems with few or no source changes. It also provides a way for us to export Topaz application software to UNIX sites.

We believe that the guidelines presented in this paper will be useful in designing other interfaces for use by multithreaded programs. The extra parameterization necessary to avoid shared mutable state might not always be useful in a purely sequential program, although it is likely to ease the construction of modular programs. Eliminating ad hoc multiplexing has the property that the resultant interface can be viewed as appropriate for use by a purely sequential program, so in some sense no complexity is added for use by a multithreaded program. Making operations cancellable is often an externally imposed requirement; the multithreaded approach avoids many of the problems with interrupts and restartability.

References

[1] Apollo Computer Inc. *Concurrent Programming Support (CPS) Reference*. 330 Billerica Road, Chelmsford, MA 01824, June 1987.

[2] AT&T. *System V Interface Definition, Issue 2*. Customer Information Center, P.O. Box 19901, Indianapolis, IN 46219, 1986.

[3] M. J. Bach and S. J. Buroff. Multiprocessor UNIX operating systems. *AT&T Bell Laboratories Technical Journal*, 63(8):1733–1749, October 1984.

[4] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. In *Proceedings of the Eleventh Symposium on Operating System Principles*, New York, November 1987. ACM. To appear.

[5] Digital Equipment Corporation, Maynard, MA. *VAX/VMS System Services Reference Manual*, September 1984.

[6] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[7] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher. 4.2BSD system manual. In *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution*, volume 2C. Computer Systems Research Group, University of California at Berkeley, 1983.

[8] Jonathan Kepecs. Lightweight processes for UNIX implementation and applications. In *USENIX Association Conference Proceedings*, pages 299–308, June 1985.

[9] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, January 1984.

[10] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[11] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[12] John S. Quarterman, Abraham Silberschatz, and James L. Peterson. 4.2BSD and 4.3BSD as examples of the UNIX system. *ACM Computing Surveys*, 17(4):379–418, December 1985.

[13] Richard F. Rashid. Threads of a new system. *UNIX REVIEW*, 4(8):37, August 1986.

[14] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.

[15] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46–57, November 1986.

[16] Daniel C. Swinehart, Polle T. Zellweger, and Robert B. Hagmann. The structure of Cedar. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 230–244, New York, June 1985. ACM.

[17] Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach threads and the Unix kernel: The battle for control. In *USENIX Association Conference Proceedings*, pages 185–197, Phoenix, June 1987.

[18] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM and IEEE Computer Society, October 1987. To appear.

[19] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, third edition, 1985.

Part II
The Topaz Operating System Programmer's Manual

1 Introduction

This manual is the definitive reference for OS, which is the main interface to file and process facilities for Topaz programs written in Modula-2+. Topaz is the software environment built at SRC for research into multiprocessing and distributed personal computing. This environment provides facilities for developing and executing programs on Firefly computers running the Taos operating system and on VAX computers running Ultrix.

OS has been designed so that its clients can coexist and cooperate with standard Ultrix applications running on the same machine, whether it is a Firefly or a VAX. Topaz programs (using OS) and Ultrix programs can read and write the same files, send each other signals, and start processes of the same or opposite kind.

Section 2 gives an overview of the facilities underlying the interface. Sections 3 through 5 describe the individual types, constants, and procedures of the interface. Appendix A describes the related OSFriends interface.

In general this manual applies to both the Firefly/Taos and the VAX/Ultrix implementations of OS. Differences are noted where relevant, and Appendix C describes additional global details of the VAX/Ultrix implementation.

In addition to running Topaz programs, Taos—the Firefly operating system—is capable of running many Ultrix programs. See Appendix D for details.

2 Concepts and Facilities

A Topaz application consists of one or more communicating processes. One way to think of a process is as a virtual computer. Like a real computer, it has components for storage, processing, and input/output. Compared with a real computer, the components of a process are intended to be easier to use; in exchange, some potential performance is given up. At any time there may be one or more processes running on a given computer (Firefly or VAX). These processes are reasonably isolated from each other with respect to programming and operator errors. Just as the Firefly computer contains multiple processors connected to a single store, so a Topaz process can contain multiple threads of control (or threads for short) within its single address space. A thread in one process can communicate with another process on the same or a different computer by making a remote procedure call (RPC). The RPC mechanism lets processes communicate in space; the file system lets them communicate in both time and space. The file system allows access to local and remote disk files, to tty-like devices, and to interprocess pipes. The file system provides a hierarchically organized name space to facilitate sharing of files among programs and users.

2.1 Processes

A Topaz process is a set of threads of control executing within a single virtual address space; a process also has a number of state variables maintained by the operating system such as identification, access control, and open files. A Topaz application is made up of a single process or a set of communicating processes; every program (except parts of the operating system itself) executes within the context of some process.

The Topaz definition of a process is very similar to the Ultrix definition, with the main exception that a Topaz process can contain multiple threads of control.

What can a process do? It can execute object programs within its own address space, it can make remote procedure calls to other processes (on either the same or a different machine), and it can invoke the operating system by calling procedures in Topaz interfaces such as Thread and OS. Through OS, it can manipulate files and create and control other processes. Finally, it can request that its own execution be terminated.

Every process has a process identifier. Its value is an integer between one and about 32000 (for compatibility with Ultrix). The process identifier remains constant over the life of the process and serves as the name used by a client of OS to manipulate that process. At no time do any two processes on the same machine have the same process identifier, but process identifiers are reused over time. The process identifier equal to one always belongs to the Init process.

Every process belongs to one process group, identified by a process group identifier. The process group identifier is generally equal to the process identifier of the process that is the ‘leader’ of the group, and is specified when the process is created. Process groups form a larger level of granularity to which signals can be sent, and are also used as owners of tty devices. Process groups are discussed more thoroughly in Section 2.11, page 18.

The processes running on a machine are ordered into a tree by the parent relation. The transitive closure of the parent relation is the ancestor relation. The inverse of the parent relation is the child relation and the inverse of the ancestor relation is the descendant relation. The root of the tree is the Init process; it is the ancestor of all other processes.

Normally a process becomes the parent of a process it creates. A parent process is expected to look after its children. In particular, it should call WaitForChild for each of its children (see page 50). Also, a parent is allowed to set the process group of, or send a SigCont signal to, any of its descendants. When a process terminates, any children it may still have become orphans and are automatically adopted by the Init process. It is also possible to create a process as an orphan (see the StartProcess procedure, page 47).

To avoid interference between multiple threads running within a process, many components of the process state should not be changed after the process is started. Instead, a process template mechanism is provided for setting these state components at the time the process is created (see section 5.3, page 44). Also, many procedures in OS accept parameters that might be implicit process state components in a single-threaded system such as Ultrix. (Appendix A.1, page 53, describes procedures in the OSFriends interface that modify global process state components; use them at your own risk.)

2.2 Address Spaces

An address space is a partial mapping from addresses to updatable storage locations. Every process has an address space; it holds the program object code, global variables, local variables, and dynamically allocated (heap) variables.

Programmers using languages such as Modula-2+ and Tinylisp are mostly concerned with higher-level abstractions of storage allocation (e.g., NEW). However, knowledge of some of the lower-level details may make it easier to design and debug reliable, efficient programs.

The domain of an address space is a set of 32-bit quantities called addresses and the range is a set of 8-bit storage locations. The mapping from addresses to storage locations is partial: an address is said to be unmapped if it is not in the domain. If address i is unmapped in an address space, then so are the addresses j, j+1, ..., j+P-1, where j = (i DIV P) * P. The constant P is called the page size, and is always a power of two not less than 512. (The exact value for P is a system tuning parameter.) Addresses j, j+1, ..., j+P-1, where j is an integral multiple of P, are said to belong to the same page.

Not all storage locations in an address space may actually be updatable by the program running in that address space. Individual pages within the address space can be specified as unwritable.

If a program makes a reference to an address that is unmapped or unwritable, a hardware trap occurs that normally leads to the termination of the process. The Modula-2+ typechecking rules are intended to make it impossible for a SAFE module to pass such an improper address as a VAR or VAR IN parameter to a procedure in the OS interface. However, if an OS procedure is supplied an unmapped address or an unwritable address for a result parameter, it simulates the appropriate hardware trap in its caller.

When a Modula-2+ program runs on Taos, the first page of its address space (i.e., the one containing addresses 0 through P-1) is unmapped, to help detect attempts to dereference NIL REFs and pointers. Following this page comes the program’s object code, followed by its global variables, followed by dynamically allocated storage. Starting at a point near the middle of the 32-bit address range and growing towards smaller addresses are the stacks of the threads living in the address space. Each stack has the same maximum length, and on Taos is terminated with an unmapped page to help detect stack overflow. Between the last of the dynamically allocated storage and the stacks is a sequence of unmapped pages.

The top half of the 32-bit address range is reserved for operating system code; most of it appears unmapped to a user process. However, on Taos there are a set of RPC buffers in this range that are mapped as writable pages in every address space. If a buggy program stores into these buffers, bad things can happen to all processes running on the machine.

For more information on addressing and virtual memory management, consult the VAX Architecture Handbook and the VM interface.

2.3 Threads

A process can create any reasonable number (e.g., hundreds) of threads of control to deal with asynchronism or to perform independent computations in parallel. The Taos implementation of Topaz can execute as many threads concurrently as it has processors (currently five). The Taos scheduler chooses the highest priority ready threads without regard to the address spaces in which those threads are running. The Ultrix implementation of Topaz simulates concurrent threads within a single-threaded Ultrix process.

The Thread interface contains procedures for forking new threads and rejoining terminated threads, for allowing threads to synchronize via mutexes and condition variables, and for associating private data with a thread. For convenience, Modula-2+ also provides a LOCK statement.

The implementation of OS avoids serializing concurrent calls from different threads (in the same or different processes) except where dictated by the semantics of the individual procedures. For example, two threads concurrently writing the same byte of the same file are serialized. But a thread that blocks within the operating system (e.g. waiting for a character to be typed on a tty device) does not prevent other threads in the same process from calling OS procedures.

Sometimes it is necessary to cancel a computation. Topaz provides the thread alert mechanism for this purpose. Whenever an alert is sent to a thread that is blocked for an indefinite period within a call to a procedure of the OS interface (and most other Topaz interfaces), that procedure will raise the Alerted exception.

2.4 Security

The Topaz system attempts to protect against threats to privacy and denial of service due to human error and, to a lesser extent, human malfeasance. The security model is based on an open, distributed environment of communicating workstations and servers. The implication is that a user can do pretty much anything to his own workstation, but must provide authentication for access to remote resources. There are many gaps in the current implementation (e.g., both data and passwords are transmitted unencrypted on the network), but we believe the basic structure can be elaborated to provide an adequate combination of flexibility and security.

The Topaz operating system implements security by validating that each attempt by a process to perform an operation affecting shared system state is allowed by a set of access control information. The validation rules are based on the notion of a principal, which is supposed to represent a person or agency. The operating system associates with every file and directory an owning principal, and with every process a sponsoring principal. The operating system identifies a principal with his/her/its user name: a string of one or more ASCII characters.

Three credentials are part of the state of every process: a real user name, a password, and an effective user name. The real user name and password specify and authenticate the principal on whose behalf the process is executing; they are used for validating all access to remote files. The effective user name is used for validating all access to local files and for all other local operations. If the effective user name is root, all local file accesses and operations are permitted; such a process is referred to as the super-user. Normally the effective user name is equal to the real user name. However, if a program object file has its SetUIDonExec flag set, then executing it as a process causes that process to have as its effective user name the owner of the file.

A process normally inherits the credentials of the process that starts it, although this may be changed using SetUser for a new process or OSFriends.SetMyUser for an existing process. In any case a valid password is checked for whenever a real user name is supplied. (There is one exception: a process with effective and real user names equal to root and an empty password string is allowed the same remote access as the user named ff_ftp. The user named ff_ftp is given minimal access according to the rules on page 22; its password is never checked.)

A system must provide flexibility in describing access control relationships, or else users tend to disable the security checking. To increase flexibility, the Topaz operating system allows the definition of arbitrary user groups (subsets of the community of users), and allows the owner of each file to single out one user group to receive specific access rights. The owner of a file may also specify access rights to be enjoyed by all others (those not the owner and not in the group).

The truth for password validation and user group membership ultimately resides in the master files /etc/passwd and /etc/group. (At SRC, the truth lives in the copies of those files residing on the machine named palo-alto.) A daemon copies this information to the Interim Name Service every night. The system uses the Interim Name Service if it is available; otherwise it uses a local copy of /etc/passwd and /etc/group. Someday, the truth will move to the [Interim] Name Service. (See ns(8).)

Although a process with effective user name equal to root has super-user privileges for all local operations, such a process does not automatically have super-user privileges when accessing remote files. Only a process with real user name equal to root and having the matching ‘global root’ password can exercise remote super-user privileges.

All procedures that do access checking accept an optional user specification to use for local access checking, which may be specified only by the super-user. This is to allow a server process to access local files on behalf of more than one client. (User specifications are created by OSFriends.LookUpUser; see page 54.)

2.5 Files, Open-File Objects, and File Handles

A file is something that the Read and Write procedures can work on. One way of categorizing files is by whether they provide storage. Regular files provide storage (with a lifetime and addressibility independent of the execution of a single process).

Device files provide an input/output interface to peripheral devices. There are two subclasses of device files: unstructured and structured. Unstructured devices allow stream-like transfers; some of them may be manipulated in idiosyncratic ways via the IOCtl procedure (see page 38). Structured devices provide block-level access to storage devices such as disks and tapes. (No structured devices currently exist in the Taos implementation of OS.)

Pipes and the more general sockets provide an alternative to RPC for stream-style interprocess communication. Pipes only allow communication between processes with a common ancestor, and therefore within the same machine. Sockets allow communication between arbitrary processes on arbitrary machines interconnected by any of several internetwork protocols. (Sockets are not currently available via OS, nor are they present in the kernel-call interface to Taos.)

Taos note: Appendix D.10, page 68, describes the available devices.

Associated with each regular or device file is a set of attributes intended to help manage the file. These attributes include the user name of the owner of the file, the access group of the file, and timestamps recording file usage. The full set of attributes is enumerated in Section 4.6, page 33.

An open-file object represents the ability to perform a specific set of operations on a specific file. It also maintains temporary state associated with access to that file, namely flags governing transfer direction and related matters, a file pointer recording a position (for a regular file), and an advisory lock (for regular files and for device files). Open-file objects are created by the procedures Open, OpenRead, OpenWrite, OpenSearch, and OpenPipe (see Section 4.2, page 23).

A file is marked for deletion when the last name for it is removed from the name space (see the Remove procedure, page 41), but the actual deletion is postponed as long as one or more open-file objects referring to the file continue to exist.

A client of OS uses a file handle (a value of type File) to refer to an open-file object. File handles are returned by the procedures that create new open-file objects. They are also returned by Dup and GetDescriptor, which return file handles referring to existing open-file objects. A process should call Close (see page 30) when it is finished with a file handle. Close breaks a file handle's reference to the open-file object; it is called automatically when there are no more references to a file handle (via the Modula-2+ object cleanup mechanism). Also, when a process terminates, all file handles it holds are closed.

An open-file object continues to exist as long as there are one or more file handles referring to it; when the last such file handle is closed, the open-file object ceases to exist.

Each process has an array of file handles that were supplied to it when it was started (see the SetDescriptor procedure on page 45). Elements of the array are obtained via the GetDescriptor procedure (see page 31). The array itself is read-only, though any of the procedures provided in OS, including Close, can be applied to the file handles returned by GetDescriptor.

2.6 The File Name Space: Directories and Path Names

The file name space seen by a client of OS is a conglomeration of the local file name spaces of a set of machines. Roughly speaking, mapping a name to a file proceeds by using part of the name to determine a machine and then using the rest of the name to determine an individual file on that machine. The actual semantics of names is fairly complicated, but was designed to be able to handle several important cases:

1. Naming a specific file located on a specific machine.
2. Naming a replicated (read-only) file located on any of a set of machines.
3. Naming a generic file relative to the real user name of the process.

The name of a file or related entity in the name space is called a path name, as given by the nonterminal **<**pathname**>** in the following BNF grammar:

```
<pathname> ::= <machine> <abspath> | <abspath> | <relpath>
<machine>  ::= # <element> | <machine> : <element>
<element>  ::= $u | <mname>
<mname>    ::= <one or more characters, excluding # and :>
<abspath>  ::= <slash> <relpath>
<relpath>  ::= <empty> | <relpath1> | <relpath1> <slash>
<relpath1> ::= <pname> | <relpath1> <slash> <pname>
<slash>    ::= / | <slash> /
<pname>    ::= <one or more characters, excluding / and null>
```

First we'll look at how a path name (**<**abspath**>** or **<**relpath**>**) is looked up in the local name space, then we'll look at the extension to remote files (**<**machine**>**).

The file name space local to each machine is defined by a hierarchy of directories. A directory is a set of entries each consisting of a **<**pname**>** and a reference to another directory, a file (regular or device), or a symbolic link. There is a distinguished root directory for each machine; the path name of the root directory itself is a single slash character: /. An **<**abspath**>**, or absolute path name, is translated by looking up each **<**pname**>** in sequence from left to right, starting at the root directory.

Since the name space is hierarchical, it makes sense to consider any directory as the root of a smaller hierarchy and to look up a path name relative to that directory. A **<**relpath**>**, or relative path name, is translated by looking up each **<**pname**>** in turn, starting at a specified base directory. Every procedure in OS that accepts a path name parameter also accepts an optional directory handle parameter; if the path name is a **<**relpath**>**, it is looked up relative to the directory specified by that handle. (If the path name is a **<**relpath**>**, but the directory handle parameter is defaulted, then the working directory of the process is used—see Section 2.8, page 17).

Every directory contains an entry .. (dot-dot) referring to its parent directory and an entry . (dot) referring to itself. For references to local files, the root directory is its own parent, so the two path names

/../ and / mean the same thing. (The entry .. in the root of a remote file system works differently, as described later in this section.)

The directory-manipulation procedures maintain the invariant that there is a unique **<**abspath**>** (not containing . or ..) leading to each directory or symbolic link. This invariant does not apply to files: the HardLink procedure (see page 40) creates an additional path name referring to a given file.

Hard links can't be used for directories, and, as explained in Section 2.7, page 17, they can't span logical volumes. To alleviate these limitations, an additional form of aliasing exists: the symbolic link. A symbolic link is an entity that appears in the name space and whose value is the character string representing another path name. During the translation of a path name, when a symbolic link is encountered as the value of a **<**pname**>**, the value of the symbolic link is normally substituted for the path name up to that point and translation continues. The SymLink procedure (see page 40) creates a symbolic link.

Now it is time to look at how remote path names are translated. A path name beginning with **<**machine**>** (always flagged by the # character) means that the following **<**abspath**>** is to be interpreted relative to the root directory of the specified machine. Such a path name could be passed directly to an OS procedure, or could arise as the value of a symbolic link encountered in the translation of a local path name.

The translation of a **<**machine**>** in a path name to an actual machine (technically, an instance of a remote file service) proceeds as follows:

1. An **<**element**>** of the form '$u' is replaced by the real user name of the process requesting the translation.

2. If the **<**machine**>** consists of more than one **<**mname**>**, it is replaced with the value found by looking up the sequence of **<**mname**>**s as a path in the Interim Name Service tree (see ns(8) and the NS interface); the result should be a single NS.Label conforming to the syntax of **<**mname**>**.

3. The remaining **<**mname**>** is looked up in the Interim Name Service; it is expected to have an attribute 'type' whose value is 'instance', 'host', or 'nameset'. In either of the first two cases, the **<**mname**>** is expected to have an attribute 'address' that determines an instance of a remote file service. If the 'type' is 'nameset', then the **<**mname**>** is expected to have an attribute 'members' whose value is a list of **<**mname**>**s and an attribute 'next' whose value is a single **<**mname**>**. This step is repeated on each element in the 'members' list in random order; if none lead to an available server, this step is repeated using the value of the 'next' attribute.

If the last **<**mname**>** encountered when translating a path name is a 'nameset', then the path name is normally considered to refer to a read-only server set, causing operations that would involve modifying the name space or the data or attributes of a file to be disallowed (and to raise ProtectionViolationEC). This is because the main purpose of server sets is to increase the availability of immutable files. There are several qualifications to this rule:

- Whether a given server set is considered read-only is actually determined by the value of the additional attribute 'readOnly' of an **<**mname**>** whose 'type' is 'nameset'. This attribute should have a value of 'true' or 'false'. Currently all server sets at SRC are read-only.

- Once OS.OpenDir is used (see page 38) to open a directory using a path name involving a read-only server set, then it is permissible to perform file system modifications using that directory handle. This is because the directory handle determines a specific machine.

- Currently, modifications are disallowed if any server set (not just the last) encountered during a path name translation is read-only. This is a bug.

To facilitate the illusion of a single tree spanning several machines, the parent of the root of the remote file system on a machine, say m, appears to be the local directory /remote/m if it exists, or the root itself otherwise. By convention, each subdirectory of /remote, such as m, contains a single entry named r that is a symbolic link containing #m/. The prefix /remote/ is actually a configuration parameter (see Appendix A.6, page 58).

Here are some examples corresponding to the concepts enumerated at the beginning of this section:

1. A specific file on a specific machine: #jumbo/etc/passwd
2. A replicated (read-only) file: #server/etc/passwd
3. A generic file: #$u:homeServer:/user/spool/mail

Only regular files can be accessed remotely; devices can not.

Ultrix note: Access via the OS interface to remote files is not implemented. (This functionality is available, however, through the RFS/RFSClient interface.)

2.7 Logical Volumes

The local file system of each machine is made up of a set of logical volumes. A logical volume is an abstraction of a physical volume such as a disk pack; a logical volume can actually be implemented as a set of subvolumes (pieces) of one or more physical volumes.

Each logical volume is a self-contained unit made up of a set of files and the directory entries that reference them. Before the files on a logical volume can be used, the logical volume has to be added to the file name space of the machine. This is done by 'mounting' the logical volume, which identifies the root directory of that logical volume with some existing directory in the name space. While a logical volume is mounted, its directory hierarchy is a subhierarchy of the local name space.

Even while it is mounted, a logical volume still maintains its autonomy. A hard link (created by the HardLink procedure, page 40) in a directory residing on one logical volume cannot refer to a file residing on a different logical volume. And of course free space on one logical volume cannot be used to create or extend a file residing on a different logical volume. For these reasons, it is typical to create a single large logical volume occupying most of the space on all the physical volumes on a machine.

Taos note: Operations for manipulating logical volumes are described in Appendix A.5, page 55.

Ultrix note: The Ultrix term for a logical volume is a file system. Ultrix allows a disk drive to be partitioned into several file systems, but doesn't allow a file system to span disk drives. The Ultrix implementation of OS does not support the manipulation of logical or physical volumes.

2.8 Working Directories

Every process has a working directory, which is a directory handle that is used by default as the base directory for looking up a relative path name when a NIL directory handle is supplied with the path name. Normally the working directory of a process is specified when the process is created (see the SetWD procedure on page 45) and is never changed. Changing the working directory in a multithreaded process is likely to cause undesired interference between threads. Another reason to avoid changing the working directory is that the operating system writes a file with the path name core in the working directory under some conditions, and it can be hard for the user to find this file if the working directory has been changed (see Section 5.4). If it is really necessary, the working directory of a running process can be changed using OSFriends.SetMyWD, described in Appendix A.1, page 53.

2.9 Signals

Signals combine interprocess communication with process supervision. A signal is a one-bit message that can be sent from one process to another. There are a number of differently named signals; a process has a separate receiving port for each named signal. (Most of the types and procedures related to signals are described in Section 5.2.)

There are default actions that occur when some signals are received, such as stopping the process, restarting the process, or terminating the process; other signals have no default action (the signal is ignored upon receipt). The default action is actually mandatory for some signals, but can be overridden for others. A process overrides the default action by declaring either that the signal is to be ignored or that the signal will be handled. To handle a signal or set of signals, a process calls the procedure WaitForSignal, which blocks the calling thread until one of the signals has been received.

Signals in OS are used to report asynchronous events to processes; one can think of sending a signal to a process as analogous to sending an alert to a thread. The most frequent reason for sending a signal is to report that the user typed Control-C (requesting termination of the current command or program).

Signals in OS are based on the signals in Ultrix; indeed, OS clients and Ultrix clients running on the same machine can send signals to each other. OS differs from Ultrix in its treatment of signals in two ways:

- OS delivers a signal to a client handler by unblocking a call to WaitForSignal, while Ultrix delivers a signal by calling a client-supplied procedure, thereby interrupting the single thread.
- While OS restricts the use of signals to reporting asynchronous events, Ultrix also uses signals to report synchronous events, that is those directly related to the execution of instructions or system calls by the process receiving the signal. (Synchronous events are reported to Topaz programs via Modula-2+ exceptions.)

2.10 Control Terminals

The concept of a control terminal comes from Ultrix; the idea is for a process to be able to direct output to a place that will be seen by its user (a human being) even when its standard input and standard output have been redirected. A process may or may not have a control terminal. If it does, then the control terminal can be opened by opening a particular device file, which usually has the path name /dev/tty. (OS also provides the OpenControlTerminal procedure; see page 26.) Normally a new process inherits the control terminal of the process that created it, but this can be overridden (see SetControlTerminal and UnsetControlTerminal, page 45). A process with no control terminal automatically acquires as a control terminal the first tty device that it opens.

2.11 Job Control

The Topaz operating system provides so-called job control facilities for multiplexing a single terminal device among several jobs (groups of processes). These facilities are provided mainly for compatibility with Ultrix and its C shell (see csh(1)). In an environment with a window manager such as the Firefly, multiplexing a single terminal (i.e., window) is less important, but can still be useful. Since the C shell contains per-instance state in the form of variables and aliases, it can be useful to stop a process running under a given shell and type new commands to that same shell.

Operating-system support for job control facilities comes in several parts: notably a way to stop and restart execution of a process, and a way to reserve a terminal device for use by one process group at a time. A command processor such as the C shell uses these facilities to grant control of a terminal to one ‘foreground’ job at a time, and to move jobs between the ‘foreground’ and ‘background’.

To stop a process, send it a stop signal (e.g., SigTStp; see page 44). To restart a stopped process, send it a continue signal (SigCont). While a process is stopped, none of its threads execute. However, intramachine remote procedure calls, including those to OS procedures, performed before the process was stopped keep executing up until the return. Normally a stop/start cycle need not and does not affect the state of a process. A process can if it desires perform ‘clean-up’ computations before being stopped and after being restarted. To clean up beforehand, it handles SigTStp (and possibly SigTTOu and SigTTIn), performs its clean-up, and then sends itself SigStop. To clean up after being restarted, it handles SigCont.

Stop and continue signals provide the basic mechanism for job control, but there needs to be a quick way for a user at a terminal to request that the current job be stopped as well as a mechanism to make sure that a background job doesn’t interfere with the use of the terminal by the foreground job. For this reason, there is a distinguished process group associated with each terminal device. Only processes in that process group can read and write the terminal; a background process (one not in the group), will normally receive a SigTTIn or SigTTOu signal if it tries to read or write its control terminal (see Read and Write starting on page 26 for more details). Also, the terminal has a mode such that when the user types a particular key (normally Control-Z), the terminal sends a SigTStp signal to the distinguished process group (see the IOCtl interface and tty(4)).

There are a few more details relevant to job control. A command interpreter process implementing job control needs to find out when the current foreground job stops; it uses the procedure WaitForChild (see page 50) to find out when a specific child process has stopped or terminated. The command interpreter needs to reset the distinguished process group of the control terminal to itself or to a new foreground job; it uses SetDevPGRP for this (see page 36).

3 The OS Interface: Preliminaries

This section and the two following sections contain the details of OS.def.

3.1 OS and Modula-2+

Most Topaz applications are written in the Modula-2+ programming language. (It is also possible to use Tinylisp.) Thus there are Modula-2+ definition modules corresponding to the OS and OSFriends interfaces described by this manual. In fact, the source files OS.def and OSFriends.def are mechanically derived from the same Scribe source file from which this manual is compiled.

In the hardcopy version of this manual, the OS interface is typeset in a **`fixed-pitch font`**, interspersed with prose in this font. Other sections of Modula-2+ code (usage examples and semantic equivalents) are typeset boxed and indented in the same **`fixed-pitch font`** used for the interface. (The same conventions are used for the OSFriends interface in Appendix A.)

```
SAFE DEFINITION MODULE OS;

IMPORT
  Base, NubTypes, Rd, System, Text, Time, UID;
```

For a summary of the interfaces imported by OS, see Appendix E, page 69.

3.2 Exceptions

The OS procedures use the Modula-2+ exception mechanism to report abnormal conditions such as incorrect actual parameters and failure of underlying abstractions.

```
TYPE
  EC =
    (OkEC, BadExecutableEC, BadFileEC, BadFileNameEC,
     BadIOCtlOpEC, BadPIDEC, BadStateForSignalEC,
     CannotLinkToDirectoryEC, CannotWriteADirectoryEC,
     CrossDeviceLinkEC, DirectoryNotEmptyEC, DirectoryUnlinkEC,
     FileBusyEC, FileExistsEC, IOErrorEC, InvalidArgumentEC,
     InvalidCredentialsEC, InvalidObjectEC, LookUpEC,
     MinorDeviceDoesNotExistEC, NameTooLongEC,
     NoDriverForDeviceEC, NoMountedVolumesEC, NotADirectoryEC,
     NotATerminalEC, NotEnoughRoomEC, NotEnoughVMEC,
     NotImplementedEC, NotOwnerEC, NotSuperUserEC,
     NotTtyOwnerReadEC, NotTtyOwnerWriteEC, OperationConflictEC,
     PipeHasNoReaderEC, ProtectionViolationEC, PvOfflineEC,
     RanOutOfResourcesEC, RemoteFileEC, ServerNotAvailableEC,
     ShortExecutableEC, TooManyProcessesEC,
     TooManySymbolicLinksInPathEC, UnseekableObjectEC,
     VolumeNeedsCheckingEC, WouldBlockEC);

EXCEPTION
  Error (EC);
```

A value of type EC is an error code used to describe one of many possible error conditions associated with the Error exception.

```
TYPE
  ES = SET OF EC;

CONST
  FileFailureES =
    ES{IOErrorEC, PvOfflineEC, VolumeNeedsCheckingEC, RemoteFileEC};

  FailureES = FileFailureES + ES{PipeHasNoReaderEC};

  PathES =
    ES{BadFileNameEC, LookUpEC, NameTooLongEC, NoMountedVolumesEC,
       NotADirectoryEC, ProtectionViolationEC, ServerNotAvailableEC,
       TooManySymbolicLinksInPathEC} + FileFailureES;
```

Most error codes report an error on the part of the caller, i.e., an incorrect combination of parameters, or parameters that are not consistent with the internal state of the abstraction. In these cases, it is as if the procedure was never called. The error codes in the set FailureES report a problem that occurred at a lower level of abstraction. The error codes in the set FileFailureES errors are logged to the console device and may result in the file system having to be scavenged. Examples include the remote server going down, the disk drive being switched off line, etc.

The error codes raised by an individual procedure are documented by listing them (individually, and FailureES and PathES where appropriate) in a comment following the RAISES clause. If a particular error code has special significance when raised by a particular procedure, it is mentioned in the following text (using the shorthand ‘raises MumbleEC’ to stand for ‘raises Error(MumbleEC)’). Otherwise the standard descriptions in Appendix B apply. The error codes in the set PathES report errors encountered in translating a path name; see Appendix B, page 60, for details.

```
VAR
  errMessage: ARRAY EC OF Text.T;
```

The errMessage variable maps each error code value into a descriptive message suitable for logging or reporting to the user. (These messages are similar to the descriptions in Appendix B, page 60. They are constant and so do not include information relevant to any particular occurrence of an error.)

```
EXCEPTION
  Alerted = Base.Alerted;
```

Alerted means that an alert was sent to the calling thread (usually by another client thread); it is only raised by procedures that can block indefinitely.

3.3 Open Array Parameters and Subarrays

Every procedure in OS that accepts a parameter, say buffer, of an open array type also accepts a pair of parameters, say start and length, designating a subarray. The start and length parameters always have defaults of zero and LAST(CARDINAL), respectively. We use the notation buffer[start FOR length] for the subarray of buffer determined by start and length, namely, the elements with indexes:

start, start+1, ... MIN(start+length-1, HIGH(buffer))

The subarray is empty if length is zero or start is greater than HIGH(buffer).

3.4 Versions

The Taos implementation of OS uses the Topaz RPC mechanism. One implication of this is that the link-edited form of an application program that imports OS will only work with compatible versions of the Taos bootfile. (The symptom of a version mismatch is an unhandled exception during initialization of a program.)

As a matter of policy, the maintainers of OS will generally attempt to avoid non upward-compatible changes to the interface. From time to time, however, there will be ‘D-days’ requiring a coordinated update of link-edited application programs and operating system bootfiles.

```
PROCEDURE Version(
    VAR (* OUT *) string: Text.T;
    VAR (* OUT *) time: Time.T)
  RAISES {};
```

Version returns the version string and time that were set at the time the operating system was built.

Taos note: See NubMisc.Version for the Nub version, which should always be later than the OS version.

4 The OS Interface: Files

This section describes most of the procedures that operate on files and directories. Procedures that are used to set up the initial state of a new process are described in Section 5.3, page 44. Rarely used procedures declared in the OSFriends interface are described in Appendix A, page 53.

4.1 Standard Declarations

There are a number of types and constants related to files and directories that are used throughout the OS interface. This subsection describes most of them, and also describes two procedures ConsFileMode and GetUMask.

```
TYPE
  PID = RECORD i: INTEGER; END;
  PGRP = PID;
```

A value of type PID is a process identifier. (PID is declared as a one-word record so that the compiler’s type checker will help catch some programming errors.)

A value of type PGRP is a process group identifier. Process groups are mainly used to multiplex terminals among several jobs, as described in section 2.11, page 18.

```
TYPE
  File = REF;
```

A value of type File is a file handle (a reference to an open-file object; see Section 2.5, page 14).

```
TYPE
  PathName = Text.T;
```

A value of type PathName is a path name (see Section 2.6, page 15).

```
TYPE
  Dir = REF;
```

A value of type Dir is a directory handle, which serves as a base for looking up a relative path name, as described in Section 2.6, page 15. Supplying NIL for the directory handle means to use the working directory of the process instead, as described in Section 2.8, page 17. Directory handles are created by the OpenDir procedure (see page 38).

```
TYPE
  User = REF;
```

A value of type User is a specification of a user (see page 14).

```
TYPE
  FileAccess = (xOK, wOK, rOK);
  FileAccessMode = SET OF FileAccess;
  AccessClass = (Others, Group, Owner);
  AccessMode = ARRAY AccessClass OF BITS 3 FOR FileAccessMode;
```

An access mode is associated with each directory and file and is used when a path name is looked up by Open or another procedure to decide whether the access is valid. The access mode says who can do what to the file or directory to which it is attached. For a file, 'what' means reading (rOK), writing (wOK), and executing as program object code (xOK). For a directory, 'what' means reading (rOK), making new entries (wOK), and searching as part of translating a path name (xOK).

'Who' means one of three things: a process whose user name is the same as the owner field of the object (Owner); a process whose user name is a member of the user group associated with the object (Group); or any other process (Other). (Recall that the effective user name is used for access to local objects, but the real user name is used for access to remote objects.)

```
TYPE
  FileAttributes =
    (SaveTextAfterUse,
     SetGIDonExec,
     SetUIDonExec);
  FileState = SET OF FileAttributes;
```

A value of type FileState packages up some miscellaneous attributes that are specified when a file is created. SetUIDonExec means to set the effective user name of a process executing the file to the owner of the file. SetGIDonExec means to set the effective group ID of a process executing the file to the group of the owner of the file. SaveTextAfterUse tells the system to save the virtual memory backing file version of the program text portion of the file even when no process is executing the program.

Taos note: SetGIDonExec and SaveTextAfterUse are ignored.

```
TYPE
  FileMode = RECORD
    access: BITS 9 FOR AccessMode;
    fileState: BITS 3 FOR FileState;
  END;
```

A file mode combines an access mode and a set of file attributes into a single quantity.

```
PROCEDURE ConsFileMode(
    owner, group, others: FileAccessMode := FileAccessMode{};
    fileState: FileState := FileState{})
    : FileMode
  RAISES {};
```

ConsFileMode constructs a FileMode, substituting for the lack of record and array constructors in Modula-2+. Note that the order of the first three parameters is the opposite of the order in AccessClass. This makes the default values more useful.

```
CONST
  RW = FileAccessMode{wOK, rOK};
  RWX = FileAccessMode{xOK, wOK, rOK};

VAR
  ReadWriteAll: FileMode;
  ReadWriteExecuteAll: FileMode;
  ReadWriteMe: FileMode;
  ReadWriteExecuteMe: FileMode;
```

These variables are initialized as follows:

```
ReadWriteAll := ConsFileMode(RW, RW, RW);
ReadWriteExecuteAll := ConsFileMode(RWX, RWX, RWX);
ReadWriteMe := ConsFileMode(RW);
ReadWriteExecuteMe := ConsFileMode(RWX);
```

```
PROCEDURE GetUMask(): AccessMode RAISES {};
```

GetUMask returns the value of the file access mode creation mask, traditionally called the umask, associated with the calling process. Whenever a file or directory is created, the mode parameter passed (to Open, MakeDir, or OSFriends.MakeDevice) is adjusted as follows:

```
mode.access[Owner] := mode.access[Owner] - umask[Owner];
mode.access[Group] := mode.access[Group] - umask[Group];
mode.access[Others] := mode.access[Others] - umask[Others];
```

The umask is set when a process is created (see the SetUMask procedure on page 45) and is not normally changed during the execution of a process. (If necessary, it can be changed using OSFriends.SetMyUMask; see Appendix A.1, page 53.)

4.2 Opening Files

Each procedure in this subsection creates a new open-file object and returns a new file handle referring to it.

```
TYPE
  OpenFlags = (fCreate, fExclusive, fTruncate,
               fRead, fWrite, fAppend,
               fNoDelay, fAsync);
  OpenMode = BITS 16 FOR SET OF [fCreate .. fAppend];
```

A value of type OpenMode is maintained within each open-file object. Most, but not all, of the individual flags can be specified when the open-file object is created; the others must be specified using the SetFlags procedure (see page 32).

```
PROCEDURE Open(
    dir: Dir;
    path: PathName;
    flags: OpenMode;
    mode: FileMode;
    getLock: BOOLEAN := FALSE;
    euser: User := NIL)
    : File
  RAISES {Error};
  (* NotSuperUserEC, PathES, FileExistsEC, FileBusyEC,
     NotEnoughRoomEC, RanOutOfResourcesEC, CannotWriteADirectoryEC,
     NoDriverForDeviceEC, MinorDeviceDoesNotExistEC, FailureES,
     NotImplementedEC *)
```

Open returns a file handle referring to a new open-file object for the file (or directory) with the specified path name. The details depend on the flags parameter:

fCreate: If the specified path name doesn't exist (but all the directories it names do exist) and fCreate is specified, create a new file. In this case, the new file is given the specified file mode adjusted using the value of the umask (see the GetUMask procedure on page 23). NotEnoughRoomEC is raised if the volume is too full to create a new file. (The mode parameter is ignored if no new file is created. LookUpEC is raised if a directory specified by the path name doesn't exist, or if fCreate isn't specified and the file itself doesn't exist.)

fExclusive: When specified along with fCreate, raise FileExistsEC if the specified path name already exists. This can be used to implement a mutex that is accessible by several processes on the same or different machines.

fTruncate: Truncate the length of the file to zero.

fRead: Request that the open-file object allow reading.

fWrite: Request that the open-file object allow writing.

fAppend: Open for appending. When this flag is used with fWrite, all writes using this open-file object are performed at the end of the file, ignoring the file pointer.

If the path name specifies a regular file and getLock is TRUE, an advisory lock is simultaneously acquired (see the SetLock procedure on page 31). This lock is an ExclusiveLock if fWrite was specified, or a SharedLock otherwise; it is automatically released when the file is closed.[3]

Opening a directory for reading as if it were a regular file is permitted; attempting to open a directory for writing raises CannotWriteADirectoryEC.

Opening a file for writing has the side-effect of clearing the file's SetUIDonExec flag.

Local access checking is based on the effective user name of the caller unless the caller is the super-user and supplies a non-NIL euser parameter, in which case the checking is based on that user. NotSuperUserEC is raised if a process other than the super-user supplies a non-NIL euser parameter.

Open raises RanOutOfResourcesEC if the path name specifies a file residing on an Ultrix machine and the Ultrix limitation on the maximum number of file descriptors per process is encountered. (This can occur two ways: either a Topaz application is running on Ultrix and trying to open a local file, or a Topaz application is running on Taos and trying to open a remote file residing on an Ultrix machine.)

Ultrix note: When getLock equals TRUE, Open does not atomically open the file and acquire the lock; it is possible that Open will create a new file and then block trying to lock it.

```
PROCEDURE OpenRead(
    dir: Dir;
    path: PathName;
    getLock: BOOLEAN := FALSE)
    : File
  RAISES {Error};
  (* PathES, FileBusyEC, RanOutOfResourcesEC,
     NoDriverForDeviceEC, MinorDeviceDoesNotExistEC, FailureES *)
```

[3]Because the Alerted exception was inadvertently omitted from the RAISES clause of Open, Open was changed to raise NotImplementedEC if getLock equals TRUE. Until the interface is changed, use SetLock.

OpenRead is equivalent to:

```
Open(dir, path, OpenMode{fRead}, anyMode, getLock)
```

Note that mode is irrelevant because no new file is ever created.

```
PROCEDURE OpenWrite(
    dir: Dir;
    path: PathName;
    getLock: BOOLEAN := FALSE)
    : File
  RAISES {Error};
  (* PathES, FileBusyEC, NotEnoughRoomEC, RanOutOfResourcesEC,
     CannotWriteADirectoryEC,
     NoDriverForDeviceEC, MinorDeviceDoesNotExistEC, FailureES *)
```

OpenWrite is equivalent to:

```
Open(dir, path,
     OpenMode{fRead, fWrite, fTruncate, fCreate},
     ReadWriteAll, getLock)
```

It creates a new file if none existed, or truncates an existing file to length zero. Note that the file is also open for reading. The file mode ReadWriteAll is used since the setting of the umask will typically subtract wOK from Group and from Others.

```
TYPE
  SearchPath = ARRAY OF Dir;

PROCEDURE OpenSearch(
    VAR IN searchPath: SearchPath;
    path: PathName;
    VAR (* OUT *) pathIndex: CARDINAL;
    spStart: CARDINAL := 0;
    spCount: CARDINAL := LAST(CARDINAL);
    getLock: BOOLEAN := FALSE;
    euser: User := NIL)
    : File
  RAISES {Error};
  (* NotSuperUserEC, PathES, FileBusyEC, RanOutOfResourcesEC,
     NoDriverForDeviceEC, MinorDeviceDoesNotExistEC, FailureES *)
```

OpenSearch looks up a path name in the sequence of directories specified by path[spStart FOR spCount]. If the file is found, it is opened as in OpenRead and pathIndex is set to the index within searchPath where the file was found. This procedure is equivalent to:

```
IF Text.IsEmpty(path) OR
    NOT (Text.GetChar(path, 0) IN Char.Set{'\#', '/'}) THEN
  ec := OS.LookUpEC;
  FOR pathIndex := spStart TO
    MIN(HIGH(searchPath), spStart + spCount - 1) DO
    TRY
      RETURN
        OS.Open(searchPath[pathIndex], path,
                OpenMode{fRead}, anyMode, getLock, euser);
    EXCEPT Error(ec):
    END;
  END;
  RAISE(OS.Error, ec);
ELSE
  RETURN
    OS.Open(NIL, path, OpenMode{fRead}, anyMode,
            getLock, euser);
END;
```

OpenSearch interprets the euser parameter the same way Open does.

```
PROCEDURE OpenPipe(
    VAR (* OUT *) rdPipe: File;
    VAR (* OUT *) wrPipe: File;
    bufferSize: CARDINAL := 0)
  RAISES {Error}; (* RanOutOfResourcesEC *)
```

OpenPipe creates two new open-file objects. The first is open only for reading and the second is open only for writing. They are linked together so that anything written on wrPipe is read by rdPipe. There are bufferSize bytes of first-in-first-out storage between the two ends; if bufferSize is zero, an implementation-dependent size is chosen. The open-file objects created by this procedure cannot be passed to the Seek procedure.

Once the reading end of a pipe is closed, a pipe is said to be broken. Writing to a broken pipe causes an error; see the Write procedure on page 27 for details on how that error is reported.

Taos note: The current default for bufferSize is 4096. RanOutOfResourcesEC is never raised.

Ultrix note: The bufferSize parameter is ignored; 4096 is always used.

```
PROCEDURE OpenControlTerminal(): File
  RAISES {Error}; (* InvalidObjectEC *)
```

OpenControlTerminal opens the control terminal for the calling process. OpenControlTerminal raises InvalidObjectEC if the calling process has no control terminal.

Taos note: OpenControlTerminal does not have to look up the path name /dev/tty, so it never needs to read from the disk.

4.3 Performing Input/Output

This subsection describes the procedures for reading, writing, and copying files, as well as some miscellaneous procedures for scheduling input/output.

The Taos implementation of these procedures uses enough threads to achieve the maximum throughput; there is no need for the client to use concurrent calls for greater throughput.

```
PROCEDURE Read(
    f: File;
    VAR (* OUT *) buf: ARRAY OF System.Byte;
    start: CARDINAL := 0;
    length: CARDINAL := LAST(CARDINAL))
    : CARDINAL
  RAISES {Error, Alerted};
  (* BadFileEC, OperationConflictEC, NotTtyOwnerReadEC,
     FileFailureES *)
```

Let sub be the subarray buf[start FOR length]. For some nonnegative integer n, Read transfers bytes from the specified file to the subarray sub[0 FOR n] and returns the value n. It raises OperationConflictEC if the open-file object referred to by f was not opened with the fRead flag.

When Read is used on a regular file, the transfer starts at the file position indicated by the file pointer in the open-file object, and continues until the end of sub or the end of the file is reached, whichever comes first; the file pointer is advanced by the amount read. Thus a return value less then the length of sub indicates the end of the file was reached.

When Read is used on a pipe or unstructured device, the transfer starts at the current position. If one or more characters are available, the transfer begins immediately; it continues until the end of sub or the end of the available characters is reached, whichever comes first. If no characters are currently available, but end of file has not been reached, Read blocks until one or more characters become available; this is the only time Read is alertable. Thus a return value of zero indicates the end of the file was reached. (The read-end of a pipe is at end of file when there are no longer outstanding file handles for the write-end and all the buffered data has been exhausted.)

If Read is used on the control terminal and the calling process is not in the distinguished process group of that device, Read sends a SigTTIn signal to the process group of the caller and blocks the calling thread (assuming the calling process is not an orphan and is not ignoring SigTTIn; otherwise Read raises NotTtyOwnerReadEC). See Section 2.11, page 18, for more details.

```
PROCEDURE FRead(
    f: File;
    position: CARDINAL;
    VAR (* OUT *) buf: ARRAY OF System.Byte;
    start: CARDINAL := 0;
    length: CARDINAL := LAST(CARDINAL))
    : CARDINAL
  RAISES {Error};
  (* BadFileEC, UnseekableObjectEC, OperationConflictEC,
     NotTtyOwnerReadEC, FailureES *)
```

Let sub be the subarray buf[start FOR length]. For some nonnegative integer n, FRead transfers bytes from the specified file to the subarray sub[0 FOR n] and returns the value n. It raises UnseekableObjectEC if applied to anything other than a regular file, or OperationConflictEC if the open-file object referred to by f was not opened with the fRead flag.

The transfer starts at the specified position, and continues until the end of sub or the end of the file is reached, whichever comes first. A return value less than the length of sub indicates the end of the file was reached. FRead does not examine or modify the file pointer in the open-file object.

Ultrix note: FRead will fail in non-obvious ways when used on a file whose open-file object is shared with another process. For more details, see Appendix C.9, page 64.

```
PROCEDURE Write(
    f: File;
    VAR IN buf: ARRAY OF System.Byte;
    start: CARDINAL := 0;
    length: CARDINAL := LAST(CARDINAL))
  RAISES {Error, Alerted};
  (* BadFileEC, OperationConflictEC, NotTtyOwnerWriteEC,
     NotEnoughRoomEC, PipeHasNoReaderEC, FailureES *)
```

Let sub be the subarray buf[start FOR length]. Write transfers all of sub to the specified file. It raises OperationConflictEC if the open-file object referred to by f was not opened with the fWrite flag.

When Write is used on a regular file, the transfer normally starts at the position indicated by the file pointer in the open-file object. However, if the fAppend flag is set in the open-file object, the transfer starts at the current end of file. Write always sets the file pointer to the position after the last byte transferred. It automatically extends the length of a regular file with bytes containing zero if the starting position is past the end of the file.

When Write is used on a pipe or unstructured device, the transfer starts at the current position. If the pipe or device is currently incapable of accepting the full amount requested, Write repeatedly writes as much as possible and blocks until more buffer space is available; this is the only time Write is Alertable.

If Write is used on a pipe whose reading end is closed, an error occurs. This error is reported in one of two ways, depending on the setting of the SigPipe signal state (see the SetMySignalState procedure, page 43.) If the setting is SignalIgnore, Write raises PipeHasNoReaderEC; otherwise the call to Write never returns, and the process is sent a SigPipe signal, which by default causes the process to terminate. (Rather than trying to handle SigPipe, a process should be programmed to catch PipeHasNoReaderEC.)

If Write is used on the control terminal and the calling process is not in the distinguished process group of that device, Write sends a SigTTOu signal to the process group of the caller and blocks the calling thread (assuming the calling process is not an orphan and is not ignoring SigTTOu; otherwise Write raises NotTtyOwnerWriteEC). See Section 2.11, page 18, for more details.

```
PROCEDURE FWrite(
    f: File;
    position: CARDINAL;
    VAR IN buf: ARRAY OF System.Byte;
    start: CARDINAL := 0;
    length: CARDINAL := LAST(CARDINAL))
  RAISES {Error};
  (* BadFileEC, UnseekableObjectEC, OperationConflictEC,
     NotTtyOwnerWriteEC, NotEnoughRoomEC, FailureES *)
```

Let sub be the subarray buf[start FOR length]. FWrite transfers all of sub to the file. It raises UnseekableObjectEC if applied to anything other than a regular file. It raises OperationConflictEC if the open-file object referred to by f was not opened with the fWrite flag.

The transfer starts at the specified position in the file, except that if position equals LAST(CARDINAL), FWrite appends to the current end of the file. FWrite automatically extends the length of a regular file with bytes containing zero if the specified position is past the end of the file. FWrite ignores the setting of the fAppend flag in the open-file object, and neither examines nor modifies the file pointer in the open-file object.

Ultrix note: FWrite, like FRead, will fail in non-obvious ways when used on a file whose open-file object is shared with another process. For more details, see Appendix C.9, page 64.

```
PROCEDURE Copy(
    source, destination: File;
    length: CARDINAL := LAST(CARDINAL))
    : CARDINAL
  RAISES {Error, Alerted};
  (* BadFileEC, OperationConflictEC, NotTtyOwnerReadEC,
     NotTtyOwnerWriteEC, NotEnoughRoomEC, PipeHasNoReaderEC,
     FailureES *)
```

```
PROCEDURE FCopy(
    source, destination: File;
    sourcePosition, destinationPosition: CARDINAL;
    length: CARDINAL := LAST(CARDINAL))
    : CARDINAL
  RAISES {Error};
  (* BadFileEC, InvalidObjectEC, OperationConflictEC,
     NotEnoughRoomEC, FailureES *)
```

Copy and FCopy copy a range of bytes from a source file to a destination file and return the number of bytes actually transferred. They raise InvalidArgumentEC if the source and destination refer to the same open-file object; they raise OperationConflictEC if the source was not opened with the fRead flag or the destination was not opened with the fWrite flag.

Copy can be used with any combination of files (regular, device, pipe, etc.). When the source is a regular file, Copy starts the transfer at the position indicated by the file pointer in the open-file object, and advances that file pointer by the amount transferred. When the destination is a regular file, Copy normally starts the transfer at the position indicated by the file pointer in the open-file object. However, if the fAppend flag is set in the open-file object, the transfer starts at the current end of file. Copy always advances the file pointer in a destination regular file by the amount transferred. It automatically extends the length of a destination regular file with bytes containing zero if the starting position is past the end of the file.

When the source or destination is a pipe or unstructured device, the transfer starts at the current position and is alertable if Copy is blocked waiting either for bytes from the source or for space in the destination. In any case, the transfer continues for the specified length or until the end of the source file is reached, whichever comes first.

FCopy raises UnseekableObjectEC unless the source and destination are regular files. The transfer starts at the specified positions in the source and destination, except that if destinationPosition equals LAST(CARDINAL), the transfer begins at the end of the destination file. FCopy ignores the fAppend flag and the the file pointer in the destination open-file object. FCopy automatically extends the length of the destination file with bytes containing zero if the starting position is past the end of the file.

```
PROCEDURE EnsureWritten(f: File) RAISES {Error};
  (* BadFileEC, FileFailureES *)
```

EnsureWritten makes sure that all changes to the data and attributes of the file performed before EnsureWritten was called are flushed to the disk. EnsureWritten doesn’t return until the flushing is complete.

To avoid the need for most programs to call EnsureWritten, the system contains a daemon that regularly flushes all changes to file data and attributes.

Taos note: The file flushing daemon runs every 60 seconds; it does not flush changes to file access times.

Ultrix note: For a file residing on an Ultrix system, it is not possible to guarantee that the changes will be reflected to disk before EnsureWritten returns. The file flushing daemon runs every 30 seconds.

```
PROCEDURE FinishWrites() RAISES {Error}; (* OBSOLETE *)
```

[Because of a change made to the semantics of EnsureWritten and OSFriends.EnsureAllWritten, FinishWrites is obsolete and will be removed from the interface when it is next recompiled.]

```
TYPE
  Direction = (Input, Output);
  Band = (InBand, OutOfBand);

PROCEDURE CharsAvail(
    f: File;
    direction: Direction := Input;
    band: Band := InBand)
    : CARDINAL
  RAISES {Error};
  (* BadFileEC, OperationConflictEC,
     IOErrorEC, PvOfflineEC, RemoteFileEC *)
```

CharsAvail returns the number of characters available for transferring in the specified direction. If the file is a pipe or a socket, this is the number of available characters in the buffer that can be read or written. If the file is a device, the underlying device is queried to determine the most characters it can handle without blocking in the specified direction. If the file is a regular file, the direction is Input, and the file pointer in the open-file object is positioned before the end of file, the result is the difference between the end-of-file position and the file pointer. Otherwise the result is LAST(CARDINAL).

If band is not equal to InBand, zero is returned. (OutOfBand is for future implementation of sockets.)

OperationConflictEC is raised if direction equals Input and the file is not open for reading or if direction equal Output and the file is not open for writing.

```
PROCEDURE Wait(
    f: File;
    direction: Direction := Input;
    band: Band := InBand)
  RAISES {Error, Alerted};
  (* BadFileEC, OperationConflictEC,
     IOErrorEC, PvOfflineEC, RemoteFileEC *)
```

Wait blocks until a call to CharsAvail with the same parameters would return a nonzero value. Wait raises OperationConflictEC under the same conditions as does CharsAvail. If band equals OutOfBand, CharsAvail would return 0, so Wait blocks forever. Note that if f refers to a regular file, Wait always returns immediately. Wait is alertable.

4.4 Manipulating File Handles

This subsection contains procedures for closing a file handle when it is no longer needed, for duplicating a file handle for use by a concurrent activity, and for retrieving a file handle from the constant array provided when the process was started.

```
PROCEDURE Close(f: File) RAISES {Error};
  (* BadFileEC, FailureES *)
```

Close removes the reference from the specified file handle to the underlying open-file object. Once a file handle has been closed, passing it to any procedure other than Close causes BadFileEC to be raised. Passing a file handle to Close a second time does not cause an exception to be raised (or have any other effect).

Once the last file handle referring to a particular open-file object is closed, the open-file object itself is deleted and the advisory lock held by that open-file object, if any, is released.

Once the last open-file object for a particular file is deleted, the file itself is closed. This entails deleting the file if all the directory entries referring to it have been removed, or else clearing its exclusive-use flag and, if it is a regular file, freeing any disk space tentatively allocated past the file's current length (see the SetSize procedure on page 35).

A Close is automatically performed when a file handle is garbage-collected or the process holding it terminates, but it is often desirable to perform the Close explicitly to initiate the associated behavior. (For example, closing the read end of a pipe can be detected at the write end of the pipe.)

Taos note: When a file itself is being closed (because the last open-file object referring to it is being closed) but not deleted (because there are still directory entries referring to it) and its data or attributes (other than access time) have been modified, Close schedules the writes necessary to flush the modifications to disk. Unlike EnsureWritten, Close doesn't wait for the writes to complete.

```
PROCEDURE Dup(f: File): File RAISES {Error}; (* BadFileEC *)
```

Dup returns another file handle that refers to the same open-file object as does f. The point in doing this is to pass the new file handle to a separate thread or package that can then close it when it is finished with it without affecting the original file handle.

Ultrix note: Unlike the Ultrix dup kernel call, Dup does not create another file descriptor (of which there are a finite number), so it should be considered an inexpensive operation.

```
PROCEDURE GetDescriptor(d: CARDINAL): File RAISES {};

PROCEDURE GetMaxDescriptor(): CARDINAL RAISES {};
```

The procedures GetDescriptor and GetMaxDescriptor provide access to the constant array of file handles supplied when the process was created. GetDescriptor returns the d'th file handle, or NIL if the corresponding array element was not set. GetMaxDescriptor returns the largest number i such that GetDescriptor(i) will not return NIL.

```
CONST
  StdIn = 0;
  StdOut = 1;
  StdErr = 2;
```

The constants StdIn, StdOut, and StdErr define the indices of the standard input, standard output, and standard diagnostic output passed by shells (see sh(1) and csh(1)) when they start processes.

4.5 Manipulating Open-File State

Each procedure in this subsection examines or modifies state maintained in an open-file object (see page 14). Recall that separate calls to Open always produce separate open-file objects, but different file handles may share (i.e., refer to) the same open-file object. Sharing results from using the Dup procedure (within a single process) or from using the SetDescriptor and GetDescriptor procedures (across two or more processes); see Section 4.4, page 30.

```
TYPE
  SeekCmd = (SeekSet, SeekIncr, SeekExtend);

PROCEDURE Seek(
    f: File;
    value: INTEGER;
    whence: SeekCmd := SeekSet)
    : CARDINAL
  RAISES {Error};
  (* UnseekableObjectEC, InvalidArgumentEC, RemoteFileEC *)
```

Seek sets the file pointer contained in the open-file object referred to by f, and returns the new value of the file pointer. The new value is determined in one of three ways: by specifying an absolute position (SeekSet, the default), by adding a signed offset to the current value (SeekIncr), or by adding a signed offset to the current end of file (SeekExtend). Seek raises InvalidArgumentEC if the new pointer (as specified by value and whence) would be negative. Seek never changes the length of the file, although a subsequent Write (or Copy) will lengthen the file if the Seek moved the file pointer past the end of file.

Consider using FRead and FWrite (see page 27 and following) instead of Seek, especially if it would be useful to allow multiple threads to share the same file handle (recall that advisory locks are held by open-file objects).

Seek does nothing (and returns 0) if f is a device; it raises UnseekableObjectEC if f refers to a pipe.

```
TYPE
  LockArg = (Unlocked, SharedLock, ExclusiveLock);

PROCEDURE GetLock(f: File): LockArg RAISES {Error};
  (* BadFileEC, InvalidObjectEC, RemoteFileEC *)

PROCEDURE SetLock(
    f: File;
    lock: LockArg;
    noBlock: BOOLEAN := FALSE)
  RAISES {Error, Alerted};
  (* BadFileEC, InvalidObjectEC, WouldBlockEC, RemoteFileEC *)
```

GetLock returns the current setting, and SetLock changes the setting, of the advisory lock held by the open-file object referred to by f. The file should be a regular file. If it is a pipe or socket, GetLock and SetLock raises InvalidObjectEC.

SetLock attempts to obtain the specified advisory lock for the open-file object referred to by f. If there is no conflict for the lock, SetLock succeeds and returns immediately. The action SetLock takes when the requested lock conflicts with a lock already held by a different open-file object for the same file depends on the value of the noBlock parameter. If it is FALSE, SetLock blocks until the conflict goes away; if it is TRUE, SetLock raises WouldBlockEC. SetLock is alertable while blocked.

SetLock can be used to change an existing lock from a SharedLock to an ExclusiveLock, or vice versa. In such a case, SetLock does not first release the existing lock. (This is in contrast to the Ultrix flock kernel call.) Downgrading an ExclusiveLock to a SharedLock always succeeds immediately. Upgrading a SharedLock to an ExclusiveLock blocks as long as any other open-file object for the same file holds a SharedLock.

Advisory locks are intended to be used by cooperating processes. They are called advisory because they do not affect non-cooperating processes. Thus it only makes sense to use an advisory lock on a file when it is known that all programs that read or write the file also use an advisory lock on it.

It is not a good idea to set an advisory lock on a device, as the lock is obtained on the file used to open the device (for example /dev/magtape) rather than on the instance of the device itself (for example the device with a certain major and minor device type). Tty devices are conventionally ‘locked’ by changing the owner of the device.

```
TYPE
  SettableHandleMode = BITS 16 FOR SET OF [fAppend .. fAsync];

PROCEDURE SetFlags(f: File; flags: SettableHandleMode)
  RAISES {Error}; (* BadFileEC *)
```

SetFlags sets several of the flags in the open-file object referred to by the file handle f. The meaning of the flags is as follows:

fAppend: Force all writes to regular files to the end of the file; fAppend is ignored for other types of files.

fNoDelay: Put pipes, devices, and sockets into a state such that reads or writes through the Ultrix kernel-call interface return an error (EWOULDBLOCK) instead of blocking. This does not affect OS procedures (i.e., Read, Write, Copy, Wait). It is only present in the OS interface for use by processes sharing file descriptors with Ultrix processes via the SetDescriptor procedure (see page 45).

fAsync: Cause a signal (SigIO) to be sent to the process group associated with the device when I/O is possible. SigIO cannot be handled by a client of OS; the fAsync flag is provided only for use by processes sharing file descriptors via the SetDescriptor procedure (see page 45).

```
TYPE
  HandleMode = BITS 16 FOR SET OF [fRead .. fAsync];

PROCEDURE GetFlags(f: File): HandleMode
  RAISES {Error}; (* BadFileEC *)
```

GetFlags returns the flags associated with the open-file object referred to by f. These are all the flags settable with SetFlags and with Open.

4.6 Manipulating File Attributes

While the main purpose of a file is either to provide storage (regular files) or to provide a uniform input/output interface (device files and pipes), there are some miscellaneous attributes associated with each file. The procedures in this subsection allow examining and modifying those attributes.

```
TYPE
  FileType =
    (FTDirectory,
     FTLink, (* symbolic link *)
     FTRegular, (* disk file *)
     FTStructured, (* none on Taos *)
     FTUnstructured, (* so-called character device *)
     FTPipe, (* pipe *)
     FTSocket, (* socket--none on Taos *)
     FTFIFO); (* System V named pipe--none on Taos *)
  FileClass = SET OF FileType;

CONST
  FCLeaf = FileClass{FTRegular .. FTFIFO};
  FCDevice = FileClass{FTStructured, FTUnstructured};


TYPE
  Major = CARDINAL;
  Minor = CARDINAL;
```

For each device file (FCDevice) there are two identifying values of type Major and Minor. Major maps onto a device driver, while Minor indicates a particular instance.

Ultrix note: Instances of Major and Minor always have values less than 256.

Taos note: Devices with values larger than 256 are truncated when examined by programs using the Ultrix kernel-call interface.

```
TYPE
  VID = RECORD p: UID.Constant; t: UID.Counter; END;
```

A VID identifies a logical volume.

Taos note: A VID is equivalent to a LocalFile.ID.

```
  FileInfo = RECORD
    size: CARDINAL;
    length: CARDINAL;

    atime: Time.T;
    ctime: Time.T;
    mtime: Time.T;
    mode: FileMode;

    nLinks: CARDINAL;

    CASE fileType: FileType OF
    | FTStructured, FTUnstructured: major: Major; minor: Minor;
    ELSE
    END;

    instance: Text.T;
    vid: VID;
    fileSeq: CARDINAL;
    fileBlock: CARDINAL;

    owner: Text.T;
    group: Text.T;
  END; (* FileInfo *)
```

A FileInfo record packages the attribute information available about a file via the GetInfo and PGetInfo procedures.

The length of a file gives the number of bytes logically contained in the file: it is the position at which Read reports end of file. The size gives the number of bytes currently allocated to the file, in anticipation of further growth.

The atime field records when the file was last accessed (read or written). The mtime field records when the file was last modified (written, or changed in length). The ctime field records when some attribute of the file was last changed (by any of the procedures in this section, or those in Section 4.8, page 39 that affect fields of the FileInfo record).

The mode field gives the access mode and miscellaneous file state (mainly SetUIDonExec).

The nLinks field counts the total number of directory entries (hard links) pointing to this file. It may be an overestimate if the volume needs scavenging.

The fileType field specifies what kind of file this is. For device files, the major and minor fields determine the device driver and instance of that device type.

The instance, vid, fileSeq, and fileBlock fields map the file onto the underlying representation. The instance field is NIL if the file resides on the local machine, otherwise it specifies the instance of the remote file service used to access the file. There may be more than one such instance serving a particular volume, so the instance field is not a substitute for the vid field in testing whether two remote disk files are the same.

The vid field specifies the logical volume on which the file resides; it is unique across all machines.

The fileSeq field is a unique identifier for the file within its volume. On Ultrix, the number may eventually be reused (it is known as an inode number); on Taos, it is unique for all time. Two regular files are identical if and only if they have the same vid and fileSeq values.

On Taos, the fileBlock field specifies the logical block number within the volume on which the file begins. On Ultrix, this field is always zero.

The owner field specifies the user name of the owner of the file; the group field specifies the name of the group of users that is singled out to receive special access to the file (see the discussion of access modes on page 22).

```
PROCEDURE GetInfo(
    f: File;
    VAR (* OUT *) info: FileInfo;
    returnNames: BOOLEAN := TRUE)
  RAISES {Error};
  (* BadFileEC, RemoteFileEC, IOErrorEC, PvOfflineEC *)
```

```
PROCEDURE PGetInfo(
    dir: Dir;
    path: PathName;
    VAR (* OUT *) info: FileInfo;
    follow: BOOLEAN := TRUE;
    returnNames: BOOLEAN := TRUE;
    euser: User := NIL)
  RAISES {Error}; (* NotSuperUserEC, PathES, FailureES? *)
```

GetInfo and PGetInfo return information about a file. GetInfo accepts a file handle for an open file, while PGetInfo accepts the path name of a file. If the file is actually a symbolic link and follow is FALSE, then PGetInfo returns information about the link itself rather than its referent.

If returnNames is TRUE, the owner and group fields are filled in; otherwise they are set to NIL. (This is an optimization to avoid reading /etc/passwd in the Ultrix emulation and to avoid an allocation.)

PGetInfo interprets the euser parameter the same way Open does.

```
PROCEDURE SetLength(f: File; length: CARDINAL)
  RAISES {Error};
  (* BadFileEC, InvalidObjectEC, IOErrorEC, PvOfflineEC,
     RemoteFileEC, NotEnoughRoomEC *)


PROCEDURE PSetLength(
    dir: Dir;
    path: PathName;
    size: CARDINAL;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, InvalidObjectEC, NotEnoughRoomEC,
     FailureES? *)
```

SetLength and PSetLength set the length of the file, that is the point at which Read and FRead report end of file, to the specified number of bytes. If the length increases, the size, that is the number of bytes actually allocated, is increased to that needed to hold the new length and each new byte is set to zero. Decreasing the length has no effect on the size, so usually SetSize or PSetSize should be called afterward. See SetSize and PSetSize below.

SetLength does nothing if f refers to a device; it raises InvalidObjectEC if f refers to a pipe or socket. PSetLength raises InvalidObjectEC if the path name specifies anything other than a regular file.

PSetLength interprets the euser parameter the same way Open does.

```
PROCEDURE SetSize(f: File; size: CARDINAL)
  RAISES {Error};
  (* BadFileEC, InvalidObjectEC, IOErrorEC, PvOfflineEC,
     RemoteFileEC, NotEnoughRoomEC *)


PROCEDURE PSetSize(
    dir: Dir;
    path: PathName;
    size: CARDINAL;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, InvalidObjectEC, InvalidArgumentEC,
     NotEnoughRoomEC, FailureES? *)
```

SetSize and PSetSize reserve enough storage for a file so that it can grow to a specified number of bytes in length. SetSize accepts a file handle for an open file, while PSetSize accepts the path name of a file.

SetSize and PSetSize do nothing if f refers to a device or to a file residing on an Ultrix machine. SetSize raises InvalidObjectEC if f refers to a pipe or socket.

If a program can estimate the eventual length of a file it will be writing, setting the size after opening the file serves to check whether sufficient disk space exists and to reserve it for the specified file. This contributes towards minimizing the fragmentation of disk space.

SetSize and PSetSize raise InvalidArgumentEC if the new size is less than the current length of the file.

PSetSize interprets the euser parameter the same way Open does.

Taos note: Setting the size explicitly is an optimization for use by programmers trying to use disk space carefully. Taos automatically increases the size when a SetLength or a write past the current size occurs. When the last open-file object for a file is deleted, Taos automatically decreases the size to the minimum needed to hold the length if and only if the last size-setting operation was an automatic one.

Ultrix note: SetSize and PSetSize are ignored.

```
PROCEDURE GetDevPGRP(f: File): PGRP RAISES {Error};
  (* InvalidObjectEC, BadPIDEC *)

PROCEDURE SetDevPGRP(f: File; pgrp: PGRP) RAISES {Error};
  (* InvalidObjectEC *)
```

GetDevPGRP and SetDevPGRP get and set the distinguished process group associated with a device. If the file is not a device, GetDevPGRP and SetDevPGRP raise InvalidObjectEC. If the file is a device whose process group hasn't yet been set, GetDevPGRP raises BadPIDEC.

See the discussion of job control in Section 2.11, page 18.

```
PROCEDURE SetMode(
    f: File;
    mode: FileMode;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, BadFileEC, NotOwnerEC,
     IOErrorEC, PvOfflineEC, RemoteFileEC *)


PROCEDURE PSetMode(
    dir: Dir;
    path: PathName;
    mode: FileMode;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, NotOwnerEC, FailureES? *)
```

SetMode and PSetMode set the file mode of a file. SetMode accepts a file handle for an open file, while PSetMode accepts the path name of a file or directory. The file mode contains the access mode, controlling which processes can access the file, and also contains the FileState (e.g., the SetUIDonExec flag).

These procedures raise NotOwnerEC unless the calling process is the owner of the file or is the super-user.

SetMode and SetMode interpret the euser parameter the same way Open does.

Note that changing the owner of a file or opening it for writing clears the SetUIDonExec flag.

```
PROCEDURE SetOwner(
    f: File;
    owner: Text.T;
    group: Text.T;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, BadFileEC, InvalidObjectEC,
     NowOwnerEC, FailureES? *)


PROCEDURE PSetOwner(
    dir: Dir;
    path: PathName;
    owner: Text.T;
    group: Text.T;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, NotOwnerEC, FailureES? *)
```

SetOwner and PSetOwner set the owner and user group of a file, and also clear the SetUIDonExec flag. SetOwner accepts a file handle for an open file, and raises InvalidObjectEC unless it refers to a regular file or device. PSetOwner accepts the path name of a file, directory, or symbolic link.

Specifying NIL for owner or group means to leave it as it stands. A process must be the super-user to change the owner, otherwise NotSuperUserEC is raised. A process must either be the super-user or must be both the owner of the file and a member of the new group to change the user group, otherwise NotOwnerEC is raised.

SetOwner and PSetOwner interpret the euser parameter the same way Open does.

Ultrix note: SetOwner and PSetOwner also clear the SetGIDonExec flag.

```
PROCEDURE PAccess(
    dir: Dir;
    path: PathName;
    euser: User := NIL)
    : FileAccessMode
  RAISES {Error};
  (* NotSuperUserEC, PathES, FailureES? *)
```

PAccess determines the FileAccessMode that the calling process would have for the specified file: a subset of rOK, wOK, and xOK. PAccess raises LookUpEC if the file doesn't exist, but it only raises ProtectionViolationEC if the caller has insufficient privileges to access the prefix of the specified path name leading up to the final directory.

PAccess interprets the euser parameter the same way Open does.

```
PROCEDURE PGetLink(
    dir: Dir;
    path: PathName;
    euser: User := NIL)
    : Text.T
  RAISES {Error};
  (* NotSuperUserEC, PathES, InvalidArgumentEC, FailureES? *)
```

PGetLink returns the value of the symbolic link with the specified path name; it raises InvalidArgumentEC if the path name doesn't specify a symbolic link.

PGetLink interprets the euser parameter the same way Open does.

```
PROCEDURE PSetTimes(
    dir: Dir;
    path: PathName;
    atime: Time.T;
    mtime: Time.T;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, NotOwnerEC, FailureES? *)
```

PSetTimes sets the access and modification times of the file to the specified values, and sets the attribute-change time (ctime) of the file to the current time. It raises NotOwnerEC if called by other than the super-user or the owner of the file.

PSetTimes interprets the euser parameter the same way Open does.

```
PROCEDURE IOCtl(
    f: File;
    operation: UNSIGNED;
    VAR (* IN and/or OUT *) argument: ARRAY OF System.Byte)
  RAISES {Error, Alerted};
  (* BadFileEC, InvalidArgumentEC, OperationConflictEC,
     InvalidObjectEC, BadIOCtlOpEC, IOErrorEC,
     PvOfflineEC, RemoteFileEC *)
```

IOCtl performs a device-specific operation on the device underlying the file handle f. It raises BadIOCtlOpEC if f doesn't refer to a device or if f refers to a device but the operation does not apply to the particular device type. It raises InvalidArgumentEC if the 'argument' parameter does not meet the requirements (size or value) of the specified operation.

See the documentation of the specific device for details about the operations and their arguments, e.g., tty(4). The IOCtl and TtyDevice interfaces contain Modula-2+ types and constants for use with the IOCtl procedure.

No currently implemented IOCtl operations are actually alertable.

Taos note: The only commands that are implemented are the Ultrix device-specific commands for which OS provides no specialized procedure (e.g., GetDevPGRP). This excludes FIOCLEX/FIONCLEX, FIOGETOWN/FIOSETOWN, FIOASYNC, FIONBIO, FIONREAD, TIOCNXCL/TIOCEXCL, TIONOTTY, and TIOCGPGRP/TIOCSPGRP.

4.7 Opening and Examining Directories

This subsection contains procedures for dealing with existing directories: obtaining a directory handle to be used to qualify path name lookups, converting a (directory handle, relative path name) pair to an absolute path name, and listing the contents of a directory.

```
PROCEDURE OpenDir(
    dir: Dir;
    path: PathName;
    euser: User := NIL)
    : Dir
  RAISES {Error};
  (* NotSuperUserEC, PathES, NotADirectoryEC, FailureES? *)
```

OpenDir returns a directory handle for an open directory. This directory handle can then be supplied to Open, OpenDir, and other operations for use as the base for looking up relative path names.

OpenDir performs access checking on the path name up to, but not including, the final component, which must be a directory (or else NotADirectoryEC is raised). Access checking on the actual directory is done each time the directory handle is used by another operation.

OpenDir interprets the euser parameter the same way Open does.

```
PROCEDURE GetPath(
    dir: Dir;
    path: PathName := NIL;
    euser: User := NIL)
    : PathName
  RAISES {Error}; (* NotSuperUserEC, PathES, FailureES? *)
```

GetPath accepts a specification of a file by the usual combination of a path name (possibly relative) and a directory handle; it returns an absolute path name for the file that contains no . or .. or symbolic link components.

The path name returned by GetPath consists of an <abspath>, possibly preceded by a <machine> (i.e., a # followed by a machine name). (See section 2.6, page 15, for the syntax of <abspath> and <machine>.) The <machine> is present if and only if the file or directory to which the path name refers resides on a remote machine. The <abspath> specifies a sequence of directories from the root of the local or specified remote file system (never a symbolic link, or a . or .. entry). Recall that a directory has a unique absolute path name, but a file may have several; it is not predictable which one GetPath returns.

GetPath interprets the euser parameter the same way Open does.

Ultrix note: The current implementation may return a path name containing .. entries. This should be fixed eventually.

```
PROCEDURE ListDir(
    dir: Dir;
    VAR (* IN OUT *) rd: Rd.T;
    euser: User := NIL)
  RAISES {Error}; (* ProtectionViolationEC, FailureES *)

PROCEDURE NextEntry(rd: Rd.T): BOOLEAN RAISES {Error};
  (* ProtectionViolationEC, FailureES *)
```

ListDir and NextEntry enumerate the names of the entries in a directory, including the . and .. entries.

ListDir prepares a directory enumeration reader so that it is ready to read the first entry of the specified directory. (The . entry is always returned first.) If ListDir is passed NIL, it allocates a new reader; if it is passed an old reader (that it had returned on an earlier call), ListDir avoids additional allocations.

NextEntry returns TRUE and modifies rd so it is ready to read the next entry if one exists; otherwise it returns FALSE and does not modify the reader.

ListDir and NextEntry raise ProtectionViolationEC if the calling process lacks read access to the directory being enumerated.

Ultrix note: On Ultrix 2.0 systems using NFS, ListDir and NextEntry do not work on remote directories.

Here is a program to list a directory:

```
OS.ListDir(dir, rd);
REPEAT
  Wr.PrintF(Stdio.stdout,
            "%t\n", Rd.GetText(rd, LAST(Rd.Index)));
UNTIL NOT OS.NextEntry(rd);
```

4.8 Manipulating the Name Space

This subsection contains procedures for creating, deleting, and renaming entries in directories, including creating and deleting directories themselves, creating symbolic links and hard links, removing symbolic links and files (including hard links). Recall that Open (or OpenWrite) creates a file, and therefore a directory entry, under some conditions. For an overview of the file name space, see Section 2.6, page 15.

```
PROCEDURE MakeDir(
    dir: Dir;
    path: PathName;
    mode: FileMode;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, FileExistsEC, FailureES? *)
```

MakeDir creates an empty directory with the specified path name. MakeDir creates at most one directory: it raises LookUpEC if one of the intermediate components in the specified path name does not exist. It raises FileExistsEC if there is an existing directory entry with that path name. MakeDir sets the file mode of the new directory to the specified file mode, adjusted using the value of the umask (see the GetUMask procedure on page 23); normally the caller should specify a mode of ReadWriteExecuteAll.

MakeDir interprets the euser parameter the same way Open does.

```
PROCEDURE RemoveDir(
    dir: Dir;
    path: PathName := NIL;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, DirectoryNotEmptyEC, FileBusyEC,
     FailureES? *)
```

RemoveDir deletes the directory with the specified path name and also deletes the entry for that directory from its parent directory. It raises NotADirectory if the specified path doesn't name a directory, or DirectoryNotEmptyEC if the directory to be deleted contains anything other than the . and .. entries. It raises FileBusyEC if the directory is the root directory of a logical volume, or if there is another logical volume mounted on top of the directory. (Mounting volumes is described in section 2.7, page 17, and Appendix A.5, page 55.)

Once a directory has been deleted, an attempt to use an old directory handle for it raises LookUpEC.

RemoveDir interprets the euser parameter the same way Open does.

```
PROCEDURE SymLink(
    fromDir: Dir;
    from: PathName;
    to: PathName;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, FileExistsEC, FailureES? *)
```

SymLink creates a symbolic link with the path name specified by from and fromDir; it raises FileExistsEC if the path name is already in use. The content of the link is set to the value of the 'to' parameter, without any checking or evaluation performed on it. The use of symbolic links in the file name space is described on page 16.

SymLink interprets the euser parameter the same way Open does.

Ultrix note: SymLink raises NameTooLongEC if the 'to' parameter is longer than 255 characters.

```
PROCEDURE HardLink(
    oldDir: Dir;
    oldPath: PathName;
    newDir: Dir;
    newPath: PathName;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, FileExistsEC,
     CannotLinkToDirectoryEC, CrossDeviceLinkEC,
     FailureES? *)
```

HardLink creates a new directory entry with the path name specified by newPath and newDir that refers to the same file link as does the path name specified by oldPath and oldDir. It raises FileExistsEC if the new path name is already in use, CannotLinkToDirectoryEC if the old path name is that of a directory, or CrossDeviceLinkEC if the new path name lies in a different logical volume than the old one.

Once the link is made, the new and old links are equivalent with respect to all the file system operations.

HardLink interprets the euser parameter the same way Open does.

```
PROCEDURE Remove(
    dir: Dir;
    pathName: PathName;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES, DirectoryUnlinkEC *)
```

Remove deletes the directory entry with the specified path name. This must be an entry created by Open, SymLink, OSFriends.MakeDevice, or HardLink; Remove raises raises DirectoryUnlinkEC if the entry is for a directory.

The entry is removed from the directory immediately. If it refers to a file, the file is not deleted from the disk until no other directory entries (hard links) or open-file objects refer to it. If entry refers to a symbolic link, the link is deleted but the object to which the link referred (if any) is not affected.

Remove interprets the euser parameter the same way Open does.

```
PROCEDURE Rename(
    oldDir: Dir;
    oldPath: PathName;
    newDir: Dir;
    newPath: PathName;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, PathES,
     CrossDeviceLinkEC, DirectoryUnlinkEC, DirectoryNotEmptyEC,
     InvalidArgumentEC, FailureES? *)
```

Rename moves a file or directory ‘atomically’ from one directory to another. Rename guarantees that the operation is atomic with respect to concurrent operations, and with respect to system crashes.

When the old path name specifies a file, Rename requires either that the new path name specifies a file, or that it is nonexistent. When the old path name specifies a directory, Rename requires either that the new path name specifies an empty directory (excluding . and ..), or that it is nonexistent. Rename raises DirectoryUnlinkEC if the old path name specifies a file and the new path specifies an existing directory, or vice versa. Rename raises DirectoryNotEmptyEC if the old path name specifies a directory and the new path name specifies an existing non-empty directory. Rename raises CrossDeviceLinkEC if the new path name is in a different logical volume than the old path name. Rename raises InvalidArgumentEC if the operation would put a cycle into the file system, e.g., renaming /a as /a/b.

Rename interprets the euser parameter the same way Open does.

Taos note: if the system crashes while a file is being renamed, the file will not be lost, but it may be referred to by both the old and new path names, or there may be an extra reference count on the file. The first problem is not detected; the second can be detected and fixed by the scavenger (but will not cause any harm). If a directory is being renamed, the operation will either complete or will be backed out by a forced run of the scavenger when the system reboots.

5 The OS Interface: Processes

This section describes most of the procedures that operate on processes and signals. Rarely used procedures declared in OSFriends are described in Appendix A, page 53.

5.1 More Standard Declarations

PID, the type of a process identifier, and PGRP, the type of a process group identifier, are described in Section 4.1, page 21, since PGRP is referenced by the GetDevPGRP and SetDevPGRP procedures, described on page 36.

```
VAR
  NullPID: PID;
```

NullPID contains the 'default' value for the type PID, namely zero.

5.2 Sending and Handling Signals

This subsection describes the procedures for sending signals to other processes and for handling signals. It also enumerates all the possible signals and their default behavior.

```
TYPE
  SignalOrNone =
    (SigNone, SigHUp, SigInt, SigQuit, SigIll, SigTrap,
     SigIOT, SigEMT, SigFPE, SigKill, SigBus, SigSegV,
     SigSys, SigPipe, SigAlrm, SigTerm, SigUrg, SigStop,
     SigTStp, SigCont, SigChld, SigTTIn, SigTTOu, SigIO,
     SigXCPU, SigXFSz, SigVTAlrm, SigProf, SigWinCh, Sig29,
     SigUsr1, SigUsr2);
  Signal = [SigHUp .. SigUsr2];
```

SignalOrNone enumerates the names of all the signals; there are 31 different named signals (not including SigNone). The definition of Signal is consistent with the numbering of signals in Ultrix 1.1 and 2.0 (see sigvec(2)). However, many of these signals should not be used by OS clients because OS provides different mechanisms for dealing with the underlying conditions. It is highly recommended that only the following signals be used:

Name	*Send ok?*	*Handle ok?*	*Usage*
SigHUp	No	Yes	Modem carrier has dropped
SigInt	Yes	Yes	User changed his mind (Control-C)
SigQuit	Yes	Yes	User wants core image (Control-\)
SigKill	Yes	No	Unconditional termination
SigTerm	Yes	Yes	General software termination signal
SigStop	Yes	No	Unconditional stop
SigTStp	Yes	Yes	User wants to stop (Control-Z)
SigCont	Yes	Yes	Restart stopped process
SigTTIn	No	Yes	Read by background process
SigTTOu	No	Yes	Write by background process
SigUsr1	Yes	Yes	User-defined signal (System V)
SigUsr2	Yes	Yes	User-defined signal (System V)

Sending other signals to a process can confuse the process, because of the Ultrix semantics associated with those signals. Attempting to handle other signals is deprecated because OS and Topaz provide more appropriate mechanisms. (The Ultrix implementation of OS disallows handling most other signals; the Taos implementation may be changed to match.)

Topaz converts hardware traps to exceptions defined in the System and Trap interfaces rather than converting them to the signals SigIll, SigIOT, SigEMT, SigFPE, SigBus, and SigSegV, as Ultrix does. Topaz reserves the hardware breakpoint trap for the debugger rather than converting it to SigTrace. OS raises exceptions for the synchronous events that Ultrix converts to the signals SigSys and SigPipe. OS doesn't provide the Ultrix interval timing facilities that send SigAlrm, SigVTAlrm, and SigProf; instead clients can program their own facilities using the Thread and Time interfaces. OS doesn't send SigChld or SigIO because a client can fork an extra thread to wait for the event (child state change or asynchronous I/O); this will apply to SigUrg if sockets are implemented. OS doesn't provide the Ultrix resource limit facilities that send SigXCPU and SigXFSz. OS doesn't provide the Ultrix 2.0 window size change feature that sends SigWinCh. Sig29 is reserved for future standardization.

```
PROCEDURE SendSignal(
    pid: PID;
    signal: SignalOrNone;
    euser: User := NIL)
  RAISES {Error}; (* NotSuperUserEC, BadPIDEC, NotOwnerEC *)

PROCEDURE SendSignalToGroup(
    pgrp: PGRP;
    signal: SignalOrNone;
    euser: User := NIL)
  RAISES {Error}; (* NotSuperUserEC, BadPIDEC, NotOwnerEC *)
```

SendSignal sends a signal to the process with a specified process identifier; specifying pid equal to NullPID means to use the process identifier of the calling process. (Note that the conventions obeyed by the Ultrix kill kernel call for a pid of 0 or -1 do not apply to SendSignal.) SendSignalToGroup sends a signal to all the processes in a specified process group; specifying a process group identifier equal to NullPID means to send to all the processes in the group of the calling process (including the calling process itself).

Sending SigNone is a way to check the validity of a process identifier: normal error checking is done, but no signal is actually sent.

The sender and receiver must have the same effective user name, unless either the sender is the super-user, or the signal is SigCont and the sender is an ancestor of the receiver. NotOwnerEC is raised if none of these conditions are satisfied.

BadPIDEC is raised if no process has the process identifier passed to SendSignal or the process group identifier passed to SendToGroup.

SendSignal and SendSignalToGroup interpret the euser parameter the same way Open does.

```
TYPE
  SignalState = (SignalDefault, SignalIgnore, SignalHandle);
```

A signal state represents the intention by a process with respect to a particular signal to handle the signal, to ignore the signal, or to accept the default action for that signal.

```
PROCEDURE SetMySignalState(
    signal: Signal;
    state: SignalState)
    : SignalState
  RAISES {Error} (* BadStateForSignalEC *);
```

SetMySignalState allows a process to declare its intention with respect to a specified signal. It returns the previous intention.

SetMySignalState raises BadStateForSignalEC if an attempt is made to handle or ignore a signal other than SigHUp, SigInt, SigQuit, SigPipe, SigTerm, SigTStp, SigCont, SigTTIn, SigTTOu, SigXCPU, SigWinCh, Sig29, SigUsr1, or SigUsr2. It raises BadStateForSignalEC if an attempt is made to ignore SigCont. Requiring SigKill and SigStop to be defaulted provides a reliable way to terminate or stop an errant process. Disallowing SigCont to be ignored ensures that a stopped process can be continued.

As mentioned in Section 2.9, page 17, there are several possible default actions that can occur when a signal arrives for a process and the signal state is SignalDefault: to ignore the signal, to stop execution of the process temporarily, to continue execution of the process after such a temporary stop, or to terminate the process. If a signal is ignored (either because the signal state is SignalIgnore or because the signal state is SignalDefault and the default action for that signal is to ignore it), it is as if the signal had never been sent. Stopping and continuing processes is discussed in Section 2.11, page 18. Terminating processes is discussed in Section 5.4, page 49.

Which of the default actions actually occurs depends on the identity of the signal:

Ignore: SigUrg, SigChld, SigIO, SigWinCh.

Terminate: SigHUp, SigInt, SigKill, SigPipe, SigAlrm, SigTerm, SigXCPU, SigXFSz, SigVTAlrm, SigProf, Sig29, SigUsr1, SigUsr2.

Terminate with core image: SigQuit, SigIll, SigTrap, SigIOT, SigEMT, SigFPE, SigBus, SigSegV, SigSys.

Stop: SigStop, SigTStp, SigTTIn, SigTTOu.

Continue: SigCont.

There is a special case involving the stop signals: if a process is an orphan, then only SigStop will stop it; the default action for SigTStp, SigTTIn, and SigTTOu is to terminate the process. The reason for this is that in the typical usage of stop signals in conjunction with job control, it is up to the parent to continue a stopped child (see Section 2.11, page 18). An orphan would be stopped forever, so it is terminated instead.

It should be noted that signal states are inherently of process-wide significance. It is up to the application to provide its own synchronization if it dynamically alters signal state settings.

```
TYPE
  Signals = BITS 32 FOR SET OF Signal;

CONST
  AllSignals = Signals{SigHUp .. SigUsr2};

PROCEDURE WaitForSignal(allowed: Signals): Signal
  RAISES {Error, Alerted}; (* BadStateForSignalEC *)
```

WaitForSignal waits until one of the signals in the allowed set becomes pending (by being sent to this process), then makes the signal not pending and returns its identity. If several signals in the allowed set are pending, it is unpredictable which of them WaitForSignal will choose. Similarly if a signal becomes pending and that signal is in the allowed sets of several threads that have called WaitForSignal, it is unpredictable which thread will be unblocked. WaitForSignal raises BadStateForSignalEC if any signal in the allowed set is not in the state SignalHandle, since it could never become pending. WaitForSignal is alertable.

Ultrix note: The priority of the thread calling WaitForSignal must be higher than that of any compute-bound thread. See GetPriority and SetPriority in the ThreadFriends interface.

Taos note: Since the Firefly thread scheduler implements time-slicing, the priority of the thread calling WaitForSignal must be at least as high as that of any compute-bound thread.

5.3 Creating Processes

This subsection describes the procedures used to start new processes. In the general case, many pieces of state must be specified for a new process, so in addition to the procedures that actually start a process there are a number of auxiliary procedures for specifying its state.

```
TYPE
  ProcessTemplate = REF;
```

A process template is used as a way of shortening the parameter list to StartProcess and StartProcessSearch. It contains initial values for most components of a process (a few are passed explicitly to StartProcess and StartProcessSearch). One creates a process template with NewProcessTemplate; modifies it using SetDescriptor, SetUMask, SetWD, SetControlTerminal, UnsetControlTerminal, SetPGRP, SetUser, and SetSignalState, as well as OSFriends.SetSTrace and OSFriends.SetPriority described in Appendix A.4, page 55; and finally passes it to StartProcess or StartProcessSearch. One eliminates a surplus process template (one that is not going to be used) by calling CloseTemplate.

```
PROCEDURE NewProcessTemplate(): ProcessTemplate RAISES {};
```

NewProcessTemplate creates a new process template with the following content:

- An empty array of file handles (see SetDescriptor in this section and GetDescriptor on page 31).
- The same process group, real and effective user name and password, umask, working directory, control terminal, and initial thread priority as those of process calling NewProcessTemplate.
- The signal state SignalDefault for all signals, except the state SignalIgnore for those signals whose state in the process calling NewProcessTemplate is SignalIgnore.

Once a process template has been used in a call to StartProcess or StartProcessSearch (unsuccessful or otherwise), it becomes invalid and InvalidArgumentEC is raised if it is used again. The value NIL is also considered to be an invalid process template.

```
PROCEDURE SetDescriptor(
    template: ProcessTemplate;
    d: CARDINAL;
    f: File)
  RAISES {Error}; (* BadFileEC, InvalidArgumentEC *)
```

SetDescriptor sets the d'th entry in the array of file handles passed to the new process to the specified value. Once SetDescriptor returns, calling Close on the same file handle has no effect on the file handle stored in the template. (It is as if SetDescriptor calls Dup.) See the discussion of GetDescriptor on page 31. SetDescriptor raises InvalidArgumentEC if template is invalid.

```
PROCEDURE SetUMask(
    template: ProcessTemplate;
    umask: AccessMode)
  RAISES {Error}; (* InvalidArgumentEC *)
```

SetUMask sets the umask (file access mode creation mask) in the template to the specified value. It raises InvalidArgumentEC if template is invalid.

```
PROCEDURE SetWD(
    template: ProcessTemplate;
    dir: Dir;
    path: PathName := NIL;
    euser: User := NIL)
  RAISES {Error};
  (* NotSuperUserEC, InvalidArgumentEC, NotADirectoryEC,
     FailureES? *)
```

SetWD sets the working directory in the template to be the directory with the path name specified by the path and dir parameters. It raises InvalidArgumentEC if template is invalid, or NotADirectoryEC if this is not the path name of a directory.

SetWD interprets the euser parameter the same way Open does.

```
PROCEDURE SetControlTerminal(
    template: ProcessTemplate;
    f: File;
    setGroup: BOOLEAN := TRUE)
  RAISES {Error};
  (* InvalidArgumentEC, BadFileEC, NotATerminalEC *)
```

SetControlTerminal sets the control terminal in the template to be the specified file. It raises InvalidArgumentEC if template is invalid, or NotATerminalEC if the file is not a tty device. If setGroup is TRUE, then when the new process is started using this template, the distinguished process group of the file is set to be that of the new process.

```
PROCEDURE UnsetControlTerminal(template: ProcessTemplate)
  RAISES {Error}; (* InvalidArgumentEC *)
```

UnsetControlTerminal modifies the process template so that the new process created from it will not have a control terminal. As mentioned in Section 2.10, page 18, a process with no control terminal automatically acquires as a control terminal the first tty device that it opens. UnsetControlTerminal raises InvalidArgumentEC if template is invalid.

```
PROCEDURE SetPGRP(template: ProcessTemplate)
  RAISES {Error}; (* InvalidArgumentEC *)
```

SetPGRP sets a flag in the template so the process started using that template will be its own process group leader: its process group identifier will be the same as its process identifier. (The process identifier of the new process is always returned by StartProcess and StartProcessSearch.) SetPGRP raises InvalidArgumentEC if template is invalid.

```
PROCEDURE SetUser(
    template: ProcessTemplate;
    effective: User;
    real: User := NIL;
    pswd: Text.T := NIL)
  RAISES {Error};
  (* InvalidArgumentEC, InvalidCredentialsEC, NotSuperUserEC *)
```

SetUser sets the effective and real user names and password in the template to the specified values. It raises InvalidArgumentEC if template is invalid or effective is NIL. It raises NotSuperUserEC unless the calling process is the super-user, or the effective parameter is equal to the real parameter, or is equal to the current effective or real user name of the calling process.

Specifying NIL for real means not to change the real user name or password in the template. If real is not NIL, then normally pswd must be the corresponding password (in clear text); after the password has been checked, the real user name and password are stored in the template. There are a few special cases:

- If real=real user name of calling process and pswd=NIL, the real user name and password already in the template are left unchanged.
- If real='root' and pswd=NIL, then NotSuperUserEC is raised if the calling process is not the super-user, otherwise the specified real user name and password are stored in the template without further validation.
- In all other cases, the specified real user name and password are checked. InvalidCredentialsEC is raised if they are not valid, otherwise they are assigned to the template.

Ultrix note: SetUser currently raises NotSuperUser if the calling process is not the super-user.

```
PROCEDURE SetSignalState(
    template: ProcessTemplate;
    signal: Signal;
    state: SignalState)
  RAISES {Error}; (* InvalidArgumentEC, BadStateForSignalEC *)
```

SetSignalState allows changing the initial signal states that will be in effect when a new process is started. The template created by NewProcessTemplate specifies SignalDefault for all signals, except those whose state in the calling process was SignalIgnore are also set to SignalIgnore in the template.

SetSignalState raises InvalidArgumentEC if template is invalid. It raises BadStateForSignalEC if signal equals SigKill or SigStop, or if signal equals SigCont and state equals SignalIgnore.

If an Ultrix application is started with this process template, the SignalState value is mapped to Ultrix signal handler and signal mask settings (see sigvec(2)) as follows:

```
CASE state OF
| SignalDefault: handler := SIG_DFL; mask := FALSE;
| SignalIgnore: handler := SIG_IGN; mask := FALSE;
| SignalHandle: handler := SIG_DFL; mask := TRUE;
END;
```

```
TYPE
  Relationship = (Child, Orphan);
```

```
PROCEDURE StartProcess(
    dir: Dir;
    path: PathName;
    argv: Text.RefArray := NIL;
    template: ProcessTemplate := NIL;
    relationship: Relationship := Child;
    env: Text.RefArray := NIL;
    euser: User := NIL)
    : PID
  RAISES {Error};
  (* NotSuperUserEC, PathES, InvalidArgumentEC,
     BadExecutableEC, ShortExecutableEC, NotEnoughVMEC,
     FailureES? *)
```

```
PROCEDURE StartProcessSearch(
    VAR IN searchPath: SearchPath;
    path: PathName;
    argv: Text.RefArray := NIL;
    template: ProcessTemplate := NIL;
    relationship: Relationship := Child;
    env: Text.RefArray := NIL;
    spStart: CARDINAL := 0;
    spCount: CARDINAL := LAST(CARDINAL);
    euser: User := NIL)
    : PID
  RAISES {Error};
  (* NotSuperUserEC, PathES, InvalidArgumentEC,
     BadExecutableEC, ShortExecutableEC, NotEnoughVMEC,
     FailureES? *)
```

StartProcess starts a new process running the program with the path name specified by dir and path. StartProcessSearch starts a new process running a program with a path name specified by searchPath[spStart FOR spCount] and path. (Like the OpenSearch procedure, described on page 25, it tries successive elements within searchPath to find a file executable by the calling process with the specified path name.) Both StartProcess and StartProcessSearch return the process identifier of the new process.

The file specified by the path name is taken to be an a.out file if its first longword contains one of three 'magic numbers'; for details see a.out(5). StartProcess and StartProcessSearch raise ShortExecutableEC if the a.out file is not as long as indicated by the size fields in its header.

Alternatively, if the specified file starts with an appropriate header, it is deemed to be a script to be executed by an interpreter specified in that header. The format of such a script header is given by the grammar:

```
<scriptheader> ::= # ! <optblanks> <interpreter> <optarg> <newline>
<optblanks>    ::= <empty> | <blanks>
<blanks>       ::= <blank> | <blanks> <blank>
<blank>        ::= <the \040 character>
<interpreter>  ::= <pathname>
<optarg>       ::= <empty> | <arg>
<arg>          ::= <blanks> <zero or more characters, excluding \012>
<newline>      ::= <the \012 character>
```

The **<**interpreter**>** specifies the path name of another file, which must be in a.out format (or else BadExecutableEC is raised). That program is started, with the original argument list prefixed by the name of the interpreter script and **<**arg**>** from the script header.

StartProcess and StartProcessSearch raise BadExecutableEC if the file is neither an a.out file nor an interpreter script.

If the SetUIDonExec flag is set in the file mode of the file passed to StartProcess or StartProcessSearch, then the new process is created with its effective user name equal to the user name of the owner of the file.

If template is NIL, a new template as created by NewProcessTemplate is used. InvalidArgumentEC is raised if template is not NIL but is otherwise invalid.

If relationship is Child, the new process becomes a child of the caller, who assumes the burden of eventually calling WaitForChild with the process identifier of the child. If relationship is Orphan, the new child becomes a child of the Init process (whose process identifier equals one). (WaitForChild raises an exception when called with the process identifier of an orphan.)

If argv is not NIL, it contains character-string arguments to be passed to the child in the standard Ultrix fashion. If the child is a Topaz program, it can retrieve these arguments via Params.GetParameter. By convention, argv should be non-NIL and its first element should be the name of the program, i.e., the last component of the path parameter. (If argv is NIL or is a zero-length array, the child is given one argument equal to the path parameter used to start it.)

If env is not NIL, it contains the environment variables to be passed to the child in the standard Ultrix fashion. (Each element of env is a string of the form name=value; see environ(7) for more details.) If the child is a Topaz program, it can retrieve these environment strings via Params.GetEnvironmentName and Params.GetEnvironment. If env is NIL, the environment of the calling process is passed. (To pass an empty environment, create a zero-length Text.RefArray.)

After StartProcess or StartProcessSearch returns or raises an exception, the supplied template becomes invalid. Each procedure raises TooManyProcessesEC if there are no more address spaces (on Taos) or process table entries (on Ultrix), or NotEnoughVMEC if the new process would require more virtual memory than is allowed by the imposed maximum (see getrlimit(2)).

StartProcess and StartProcessSearch interpret the euser parameter the same way Open does. Note that euser is only used to do the access check on the path parameter; it is not inherited as the user name of the new process unless SetUser is used.

Taos note: The SetUIDonExec flag is ignored if the file passed to StartProcess or StartProcessSearch resides on a remote machine. (In the future, Taos will allow ‘trusted servers to supply SetUIDonExec programs.)

Ultrix note: If relationship is Orphan, an extra transient process is created. If SetUser has been set for the new process, the new user must have execute permission on the given file.

```
PROCEDURE CloseTemplate(template: ProcessTemplate)
  RAISES {Error};
```

CloseTemplate closes a template that is not going to be used to start a new process. This releases all resources held by the template and marks the template as invalid. CloseTemplate has no effect if the template has already been used to start a process or has been closed. CloseTemplate is an optimization for use by careful programmers; its effect occurs automatically when a template is garbage-collected or the creating process terminates.

5.4 Terminating Processes

This subsection describes the different ways a process may terminate, and the procedure (WaitForChild) used by a parent process to find out about the termination of a child process.

A process can terminate normally or abnormally. It terminates normally by calling the Modula-2+ HALT procedure. (Running off the end or returning from the main program is equivalent to calling HALT(0)). It terminates abnormally by making a fatal error, such as raising an exception for which there is no handler, or by receiving a signal for which the default action of termination is in force (see page 44).

When a process terminates abnormally, the operating system may write a file named ‘core’ in the working directory that contains an image of the state of the process suitable for post mortem debugging (see DebugCore(1) and adb(1)). A core image is written if a process terminates because of making a fatal error or because of receiving one of the signals whose default action is defined as termination with a core image (see page 44).

```
TYPE
  WReason = (Stopped, Killed, Exited);
  ExitStatus = [0 .. 255];
  WStatus = RECORD
    CASE reason: WReason OF
    | Stopped: wStopSig: Signal;
    | Killed: wTermSig: Signal; wCoreDump: BOOLEAN;
    | Exited: wRetCode: ExitStatus;
    END;
  END;
```

A WStatus record contains information about why a process terminated or stopped; it is returned by the WaitForChild procedure, described below. The wStopSig field indicates which signal cause the process to stop. The wTermSig field indicates which signal caused the process to terminate; this is set to SigIll if the process terminates because of a fatal error. The wRetCode field contains the value passed to HALT. (HALT performs clean-up of the Modula-2+ runtime environment, and then calls OSSpecial.Exit.)

```
TYPE
  RUsage = RECORD
    userTime, systemTime: Time.T;
  END;
```

An RUsage record contains a summary of execution time of a process. The userTime field is the sum of the CPU times of all threads in the process.

Taos note: CPU time spent executing an intramachine remote procedure call (such as to an OS procedure) is charged to the userTime of the calling thread. The systemTime field is the CPU time spent in the Taos address space by the thread that is dedicated to watching for Ultrix-style kernel calls and other traps from this process. There are also helper threads in the Taos address space that do work for various client address spaces but whose CPU time is not included in the userTime or systemTime of those clients.

Ultrix note: The systemTime field is the CPU time spent by the system executing kernel calls for this process.

```
PROCEDURE WaitForChild(
    pid: PID;
    VAR (* OUT *) rUsage: RUsage;
    waitForStoppedChild: BOOLEAN := FALSE)
    : WStatus
  RAISES {Error, Alerted}; (* BadPIDEC *)
```

WaitForChild waits for the process with the specified process identifier to terminate. If waitForStoppedChild is TRUE, WaitForChild also waits for the process to stop. WaitForChild always returns status; if the process terminated, WaitForChild also returns resource usage of the child and its descendants. A process should eventually call WaitForChild with the process identifier of each process it has started with relationship equal to Child (see the StartProcess procedure, page 47). A process should not call WaitForChild with the process identifier of any other processes (including ones it has started with relationship equal to Orphan); if it does, WaitForChild raises BadPIDEC. WaitForChild is alertable.

5.5 Examining Processes

This subsection describes the procedures for enumerating processes and for obtaining information about processes.

```
TYPE
  ProcessState = (PSRunning, PSStopped, PSTerminating);
```

A process is always in one of three ProcessStates as seen by the operating system. It is running if it has been started, has not yet terminated, and is not currently stopped. It is stopped if it received a stop signal (SigStop, SigTStp, SigTTIn, or SigTTOu) and hasn't yet received the continue signal (SigCont) or a terminating signal. Finally, it is terminating if it has terminated normally or abnormally and its parent has not yet called WaitForChild specifying its process identifier (see Section 5.4, page 49). (A terminating process is sometimes called a zombie.)

```
TYPE
  DebugState = (DSRunning, DSStopped, DSTrapped);
```

A process is always in one of three DebugStates as seen by the debugger. It is running if the debugger hasn't stopped it at the request of the user and if it hasn't spontaneously stopped because of a trap. It is stopped if the debugger has stopped it at the request of the user—for example, typing Control-C to the TeleDebug command. It is trapped if a thread within the process has caused a trap that is handled by the debugger.

```
TYPE
  SignalStates = ARRAY Signal OF BITS 8 FOR SignalState;
```

The type SignalStates represents the set of signal states for a process.

```
TYPE
  RWho = (Self, Children);

TYPE
  ProcessInfo = RECORD
```

```
    pid: PID;
    pgrp: PGRP;
    ppid: PID;

    effUser, realUser: Text.T;

    ctlTty: Text.T;
    windowSystem: Text.T;

    space: NubTypes.Space;

    numThreads: CARDINAL;

    processState: ProcessState;
    debugState: DebugState;
    debuggerConnected: BOOLEAN;

    signalStates: SignalStates;

    command: PathName;

    mappedBytes, residentBytes: CARDINAL;

    rUsage: ARRAY RWho OF RUsage;

  END; (* ProcessInfo *)
```

A ProcessInfo record packages the information available about a file via the GetProcessInfo procedure.

The pid and pgrp fields give the process identifier and process group identifier. The ppid field gives the process identifier of the parent process; if the process is an orphan, ppid is one.

The effUser and realUser fields give the effective and real user names on whose behalf the process is executing.

The ctlTty field gives an identification of the control terminal of the process. The windowSystem field identifies the Trestle instance (see Trestle.RPCInit) used for emulated terminals and typescripts created by this process.

The space field gives the address space number, of interest only for low-level debugging.

The numThreads field tells how many threads of control are executing within the process. For a standard Ultrix application running on Taos, it will equal one; for a Topaz application it will always be greater than one.

The processState field tells whether the process is currently running, stopped, or terminating, as seen by the operating system. The debugState field tells whether the process is running, stopped, or trapped, as seen by the debugger. The debuggerConnected field tells whether there is currently a TeleDebug session in existence for the process.

The signalStates field tells whether the process is currently handling, ignoring, or defaulting each of the 31 kinds of signals.

The command field gives the path name given to StartProcess, StartProcessSearch, or the execve kernel call to load the process.

The mappedBytes and residentBytes fields represent the number of bytes within the address space of the process that are currently mapped and currently resident, respectively. The resident number changes with swapping; the mapped number changes only in response to actions of that process.

The rUsage field summarizes CPU time used by this process (Self) and by all its terminated descendants (Children). (See the discussion of RUsage on page 49.)

```
PROCEDURE GetProcessInfo(
    pid: PID;
    VAR (* OUT *) processInfo: ProcessInfo;
    getUser: BOOLEAN := TRUE)
  RAISES {Error}; (* BadPIDEC *)
```

GetProcessInfo returns information about the specified process. It returns information about the calling process if pid is equal to NullPID; it raises BadPIDEC if pid is not equal to NullPID or the process identifier of any existing process. (Once WaitForChild is called for a terminating process, the process ceases to exist as far as GetProcessInfo is concerned.)

GetProcessInfo fills in the effUser and realUser fields only if getUser is TRUE.

Taos note: The mappedBytes is currently estimated by adding the allocated lengths of the high and low segments; residentBytes is set to zero.

Ultrix note: GetProcessInfo only works when pid is equal to NullPID. When getUser is TRUE, the /etc/passwd file must be read. The space, ctlTty, and command field are not filled in. The residentBytes field is actually a maximum of the number of bytes. The mappedBytes field includes both shared and unshared memory.

```
PROCEDURE NextProcess(pid: PID): PID RAISES {};
```

NextProcess returns the next in-use process identifier larger than the one it is given. To enumerate all the existing processes (using the same definition of existence as does GetProcessInfo): start by passing NullPID to NextProcess, repeatedly call NextProcess with its result from the previous call, and stop when NextProcess returns NullPID.

Ultrix note: NextProcess always returns NullPID.

Taos note: NextProcess returns process identifiers for some ‘system processes’ for which complete information via GetProcessInfo is not available:

PID	*Command*	*Other interesting fields*
1	Init	rUsage[Children]
2	Nub	numThreads, mappedBytes, residentBytes (someday)
3	Taos	space, numThreads, mappedBytes, residentBytes, rUsage

```
PROCEDURE GetPassword(): Text.T RAISES {Error};
```

GetPassword returns the password (in clear text) of the calling process if the caller is the super-user; otherwise GetPassword raises NotSuperUserEC. The password is needed for doing authenticated RPC.

Ultrix note: GetPassword returns the value of the PASSWORD environment variable if it is set, otherwise it reads the file /.ffpw (readable only by user firefly) and returns its contents.

Taos note: Until general authenticated RPC is available, GetPassword does not require the caller to be the super-user.

```
END OS.
```

A The OSFriends Interface

OSFriends contains procedures that are logically a part of the OS interface but are either very dangerous, or are not widely used.

```
SAFE DEFINITION MODULE OSFriends;

FROM OS IMPORT
  AccessMode, Alerted, Dir, Error, ExitStatus, File, FileMode,
  Major, Minor, OpenMode, PathName, PGRP, PID, ProcessTemplate,
  User, VID;

IMPORT NubTypes, Text, ThreadFriends, Time, Rd, Wr;
```

For a summary of the interfaces imported by OSFriends, see Appendix E, page 69. (NubTypes, Time, Rd, and Wr are no longer referenced from OSFriends.)

A.1 Manipulating the State of Existing Processes

```
PROCEDURE SetMyWD(
    dir: Dir;
    path: PathName := NIL;
    euser: User := NIL)
  RAISES {Error}; (* PathES *)
```

SetMyWD changes the working directory of the calling process. Since this affects every thread in the process that calls OS with a NIL directory handle, it should be used with caution; OS.OpenDir is recommended instead. Use SetMyWD only in main programs and try not to write packages that depend on it being set to something special before being called. Using the working directory will make it difficult to make your package multithreaded.

```
PROCEDURE SetMyUMask(umask: AccessMode) RAISES {};
```

SetMyUMask sets the umask of the calling process. The admonition against using SetMyWD applies also to SetMyUMask.

```
PROCEDURE SetMyUser(
    effective: User;
    real: User := NIL;
    pswd: Text.T := NIL)
  RAISES {Error};
  (* InvalidArgumentEC, InvalidCredentialsEC, NotSuperUserEC *)
```

SetMyUser sets the real and effective user name and password of the calling process, as OS.SetUser does for a process template; see page 46 for details. Once again, the admonition against using SetMyWD applies. Instead, use the euser parameter accepted by each procedure that does access checking.

```
PROCEDURE GetPriority(pid: PID): ThreadFriends.Priority
  RAISES {Error};
```

GetPriority returns the priority associated with the specified process. It returns the priority of the calling process if pid is equal to NullPID. For an Ultrix process this is the priority of its only thread. For a Topaz process this is the priority of the first thread in its address space when it was first started.

Ultrix note: Ultrix priorities less than zero maps to ThreadFriends.ForegroundPriority, Ultrix priority zero maps to ThreadFriends.NormalPriority, and Ultrix priorities greater than zero map to ThreadFriends.BackgroundPriority.

```
PROCEDURE SetPriorityPID(
    pid: PID;
    priority: ThreadFriends.Priority)
  RAISES {Error};

PROCEDURE SetPriorityPGRP(
    pgrp: PGRP;
    priority: ThreadFriends.Priority)
  RAISES {Error};

PROCEDURE SetPriorityUser(
    user: Text.T;
    priority: ThreadFriends.Priority)
  RAISES {Error};
```

These procedures set the priority of the process with a specified process identifier (SetPriorityPID), of all the processes in a specified process group (SetPriorityPGRP), or of all the processes with a specified effective user name. They raise BadPIDEC if no process was found.

When the priority of a process is changed this way, the priority of every thread in the process is set to the specified value. Note that doing this to a Topaz process with carefully assigned individual thread priorities is usually inadvisable.

A.2 Manipulating User Specifications

A user specification is an efficient encoding of the text of a user name. As described on page 14, all OS and OSFriends procedures that do local access checking accept a user specification parameter, of type User. The super-user is allowed to supply a non-nil user specification, thereby masquerading as another user. To make the parameter passing across the RPC interface more efficient, the type User is implemented as an opaque ref.

```
PROCEDURE LookUpUser(username: Text.T): User RAISES {};
```

LookUpUser converts the text of a user name into a user specification suitable for passing as the euser parameter to an OS or OSFriends procedure. LookUpUser returns NIL if the user name is not known.

```
PROCEDURE GetUserName(user: User): Text.T RAISES {Error};
```

GetUserName retrieves the text of a user name from a user specification, or raises InvalidArgumentEC if user is NIL.

A.3 Miscellaneous File-System Operations

```
PROCEDURE EqualFile(f1, f2: File): BOOLEAN RAISES {Error};
```

EqualFile returns TRUE if the two file handles refer to the same underlying regular file, device, or pipe. This is not needed if both are regular files, since the vid and fileSeq fields in the record returned by OS.GetInfo uniquely determine regular files.

Ultrix note: EqualFile reports all sockets and pipes as equal.

```
PROCEDURE EnsureAllWritten() RAISES {Error};
```

EnsureAllWritten is equivalent to calling OS.EnsureWritten for every open local file. It is also the only way to ensure that directory modifications have been flushed to disk.

```
PROCEDURE MakeDevice(
    dir: Dir;
    path: PathName;
    devMajor: Major;
    devMinor: Minor;
    mode: FileMode)
  RAISES {Error}; (* FileExistsEC, NotSuperUserEC *)
```

MakeDevice creates a device file with the specified path name and major and minor device number. MakeDevice raises FileExistsEC if the specified path name is already in use, or NotSuperUserEC unless the caller is the super-user. MakeDevice sets the file mode of the new device file to the specified file mode, adjusted using the value of the umask (see the GetUMask procedure on page 23).

```
PROCEDURE SetExclusiveUse(
    f: File;
    flag: BOOLEAN := TRUE)
  RAISES {Error};
  (* BadFileEC, InvalidObjectEC, RemoteFileEC *)
```

SetExclusiveUse[4] sets or resets the exclusive-use flag on the file referred to by f; it raises InvalidObjectEC unless f refers to a regular file or device. When a file has its exclusive-use flag set, further opens on the same file are rejected. Once the flag is set, it stays set until all open-file objects referring to the file are deleted, which occurs when all file handles referring to those open-file objects are closed.

Taos note: There is currently no known good use for the exclusive-use flag.

The exclusive-use flag has no relationship to advisory locks.

A.4 Manipulating Process Templates

```
TYPE
  STrace = (STNever, STOnce, STAlways);

PROCEDURE SetSTrace(template: ProcessTemplate; exec, fork: STrace)
  RAISES {Error};
```

SetSTrace sets the exec and fork STrace flags in the specified process template; it is for use by the Taos debugger (and has no Ultrix implementation). The fork STrace flag is currently ignored.

```
PROCEDURE SetPriority(
    template: ProcessTemplate;
    priority: ThreadFriends.Priority)
  RAISES {Error};
```

SetPriority sets the initial priority for the first thread in the new process to be created from the specified process template. The supplied priority should be one of the three constants defined in the ThreadsPort interface: BackgroundPriority, NormalPriority, or ForegroundPriority; other values are not advised. The default when SetPriority is not called is to use the priority of the calling process's initial thread as of the time the process template was created.

A.5 Manipulating Logical Volumes

The procedures in this section do not have Ultrix implementations.

In addition to the procedures defined here for files and volumes, even lower-level manipulations can be performed using the LocalFile interface.

[4]SetExclusiveUse was inadvertently omitted from OSFriends.def. It will be added next time the interface is recompiled.

Before modifying a volume at the LocalFile level, make sure that it is dismounted. Before manipulating a file at the LocalFile level, make sure that there will be no simultaneous access to it through the OS interface.

All files created by OS have a LocalFile.FileType of 1.

```
TYPE
  VC =
    (AlreadyMountedVC, BusyVC, NotMountedVC,
     RootNotADirectoryVC, VolumeHasFilesVC);
```

```
EXCEPTION
  VolumeError (VC);
```

An error associated with manipulating a volume is reported by raising the exception VolumeError with an accompanying code of type VC describing the particular problem. See Appendix B.2, page 62, for descriptions of the individual VC values.

```
TYPE
  LogicalVolume = REF;
```

A value of type LogicalVolume is a reference to a logical volume. The type is implemented as a LocalFile.LogicalVolume, and the LocalFile interface must be used to perform operations on logical volumes not defined here (such as creation, deletion, and enumeration). The definition of OSFriends.LogicalVolume is opaque to avoid all clients of OS needing to be linked with an implementation of LocalFile, but a program that imports both LocalFile and OSFriends can include the definitions:

```
TYPE
  LogicalVolume = LocalFile.LogicalVolume;
  LogicalVolume = OSFriends.LogicalVolume;
```

```
TYPE
  AVState = (VSReady, VSInUse, VSBadInUse, VSBad, VSScanning);
```

The AVState values enumerate the states that a logical volume can be in:

VSReady: The volume is ready to be mounted; it doesn't need scavenging.

VSInUse: The volume is mounted.

VSBadInUse: The volume needs scavenging, but is still mounted; there may be information about the volume that has not yet been written to it.

VSBad: The volume needs scavenging; it is not mounted.

VSScanning: The volume is currently being scavenged.

```
TYPE
  BadReason = (ZeroLinkFiles, ActiveOps, NoRootFile, BadRootFile);
```

A value of type BadReason indicates why a logical volume needs to be scavenged:

ZeroLinkFiles: The volume contains one or more files with a zero link count (that is, files for which there are no directory entries); scavenging the volume will delete these files and recover the space they occupy.

ActiveOps: The volume went offline, or the system crashed, in the middle of an operation involving a series of writes to the volume.

NoRootFile: The volume has no file system root file.

BadRootFile: The volume has a file system root file, but it is not a proper directory.

```
TYPE
  VolumeInfo = RECORD
    CASE state: AVState OF

    | VSInUse:
        numActiveOps: CARDINAL;
        numZeroLinkFiles: CARDINAL;

        asPath: PathName;

    | VSBadInUse: badAsPath: PathName;

    | VSBad: reason: BadReason;

    ELSE
    END;
  END; (* VolumeInfo *)
```

Information pertaining to the file system's use of a logical volume is packaged as a record of type VolumeInfo.

The numActiveOps and numZeroLinkFiles fields give reference counts of in-progress operations that would require the volume to be scavenged if the system crashed or the volume went offline. The numActiveOps field currently counts directory rename operations to the volume. The numZeroLinkFiles counts files that have no more directory entries but can't be deleted until the outstanding open-file objects are closed. (See section 2.5, page 14.)

The asPath field gives the path name where the volume is mounted in the local file name space; the badAsPath field gives the same information for a mounted volume that needs scavenging.

The reason field tells why the volume needs scavenging.

```
PROCEDURE GetVolumeInfo(
    volume: LogicalVolume)
    : VolumeInfo
  RAISES {VolumeError};
```

GetVolumeInfo returns information about the volume, including directory system information. See also LocalFile.LogicalVolumeInfo.

```
PROCEDURE Mount(
    volume: LogicalVolume;
    dir: Dir;
    path: PathName;
    force: BOOLEAN := FALSE)
  RAISES {Error, VolumeError};
  (* AlreadyMountedVC, RootNotADirectoryVC, BusyVC, ... *)
```

Mount adds the specified logical volume into the local file name space by identifying the root of the volume with the specified path name. The first call to Mount after the system starts must specify a path name of / to establish the root volume of the local file system.

If Mount is successful, it sets the state of the volume to VSInUse. If force is FALSE, the volume must initially be in the state VSReady. If force is TRUE, the volume will be mounted regardless of its state. Rather than do this you should really run a scavenger. (Currently the scavenger can only be run from the tshell.)

```
TYPE
  DismountLevel = [0 .. 2];


PROCEDURE Dismount(
    volume: LogicalVolume;
    force: DismountLevel := 0)
  RAISES {Error, VolumeError}; (* IOErrorEC, PvOfflineEC, BusyVC *)
```

Dismount removes the specified logical volume from the local file name space. All cached files on the logical volume are written out. If force is 0, Dismount raises BusyVC if there are any open files on the volume.

If force is 1, Dismount tries to close open files but raises errors such as IOErrorEC or PvOfflineEC if it has problems flushing the file data.

If force is 2, Dismount closes open files, and ignores errors writing to the volume (however if it gets an error it does make an attempt to mark the volume as needing scavenging).

Before calling NubMisc.Reboot or removing a physical volume, call Dismount on each mounted volume; follow that up with a call to LocalFile.FinishWrites.

```
PROCEDURE OpenFromFID(
    volume: LogicalVolume;
    block, seq: CARDINAL;
    flags: OpenMode;
    getLocks: BOOLEAN := FALSE)
  : File
  RAISES {Error, VolumeError};
```

OpenFromFID opens a file from its unique identifier. It uses the same rules as OS.Open except it may only be done by a super user program.

```
PROCEDURE InsertFID(
    dir: Dir;
    path: PathName;
    block, seq: CARDINAL)
  RAISES {Error}; (* NotSuperUserEC, ... *)
```

InsertFID inserts the specified file identifier in the directory as given. The rules for path are the same as those for OS.HardLink (see page 4.8. The file with that id must exist and must have a valid property page. The property page of a file stores most of the attributes of the file; see the Props interface for details. InsertFID raises NotSuperUserEC unless the caller is the super-user.

A.6 More Taos-only File-System Operations

The procedures in this section do not have Ultrix implementations.

```
PROCEDURE StartRFS(
    dir: Dir;
    path: PathName;
    instance: Text.T := NIL)
  RAISES {Error};(* NotSuperUserEC, ... *)
```

StartRFS exports an instance of the RFS interface with the specified instance name so that the files on the caller's machine can be used from other machines. If instance is NIL, the machine name is used. The path name specifies a log file, which StartRFS opens for appending. StartRFS raises NotSuperUserEC unless the caller is the super-user.

```
PROCEDURE SetRootDir(dir: Dir; path: PathName) RAISES {Error};
```

SetRootDir sets the root of the local file system to the directory with the specified path name. For this to work properly, the directory must be the root of some file system, possibly remote. This changes the value of root for all local file accesses in the system, but not for an RFS server running on this machine. A remote path name with an explicit **<**machine**>** part—such as #srcf34/—can be used to change the root back to the local system. SetRootDir raises NotSuperUserEC unless the caller is the super-user.

```
PROCEDURE GetConfigurationParameter(name: Text.T): Text.T
  RAISES {Error}; (* LookUpEC *)

PROCEDURE SetConfigurationParameter(
    name: Text.T;
    value: Text.T)
  RAISES {Error}; (* LookUpEC, NotSuperUserEC, InvalidArgumentEC *)

PROCEDURE NextConfigurationParameter(name: Text.T): Text.T
  RAISES {}; (* LookUpEC *)
```

These procedures get, set, and enumerate operating-system configuration parameters. If the value of a particular parameter is not printable, it consists of the characters '!!Pickle!!' prefixed to a pickle (see the Pickle interface). All three procedures raise LookUpEC to report that there is no parameter with the specified name. SetConfigurationParameter raises NotSuperUserEC unless it is called by the super-user, and raises InvalidArgumentEC if the the parameter is read-only or the specified value has the wrong type for that parameter. Pass NIL to NextConfigurationParameter to begin an enumeration; a NIL result indicates the end. The names, types, and meaning of the configuration parameters are implementation-dependent and subject to change with little notice; the currently defined parameters are:

Name	*Type*	*Read-only?*	*Description*
OpenFiles	CARDINAL	No	Number of open files to keep in memory even after they have been closed
InfoRecords	CARDINAL	No	Number of info records kept for unopened files
RemoteDirPrefix	Text.T	No	String to use as the prefix to the instance name to find the directory to be used as the parent of a remote root
CacheEntries	CARDINAL	Yes	Number of LocalFile read-cache entries
EntryBlocks	CARDINAL	Yes	Number of 512-byte blocks per cache entry
ReadEntryLimit	CARDINAL	Yes	Maximum number of entries for reads that go through the cache
WriteEntryLimit	CARDINAL	Yes	Maximum number of cache entries to update for a write
WriteBlocks	CARDINAL	Yes	Number of blocks in the write buffer
WriteDescs	CARDINAL	Yes	Number of elements in the write descriptor queue

```
END OSFriends.
```

B Error Code Summary

B.1 The OS Interface: EC

This section explains the meaning of each value ec in the set EC. The format of each paragraph is:

ec: OS.errMessage[ec] Brief description of ec.

Recall that a value of type EC accompanies the exception Error. Error and EC are defined in Section 3.2, page 19.

BadExecutableEC: StartProcess format error The file passed to StartProcess or StartProcessSearch was not recognizable as an a.out file or as an interpreter script.

BadFileEC: Bad file handle The file handle passed to a procedure had already been closed.

BadFileNameEC: Bad remote path name syntax The syntax of a remote path name was incorrect.

BadIOCtlOpEC: Bad IOCtl operation or device The file supplied to IOCtl was not an unstructured device, or else the operation did not apply to the particular device type.

BadPIDEC: No such process A PID or PGRP parameter did not specify any existing process.

BadStateForSignalEC: Invalid signal state An attempt was made to set the signal state for a particular signal to a disallowed value.

CannotLinkToDirectoryEC: HardLink to a directory The new path name supplied to HardLink was a directory.

CannotWriteADirectoryEC: Write to a directory The path name to be opened for writing was that of a directory.

CrossDeviceLinkEC: Cross-device link The parameters to HardLink or Rename would have required creating a link to a file on another logical volume.

DirectoryNotEmptyEC: Directory not empty An attempt was made to delete a directory that was not empty.

DirectoryUnlinkEC: Remove or Rename of a directory The path name of a directory was supplied where that of a file was required (Remove, Rename).

FileBusyEC: File or directory in use An attempt was made to open a file whose exclusive-use flag had been set, or to delete a directory that was the root of a file system or was the mount point for another logical volume.

FileExistsEC: File exists A name proposed for a new directory entry was already in use.

IOErrorEC: I/O error A problem occurred reading or writing the storage medium underlying a file or directory. This may result in the volume being marked as needing scavenging. The system should log the specific problem, but isn't currently.

InvalidArgumentEC: Invalid argument The argument for an operation was invalid, e.g., Seek to a negative position, or use of an invalid process template.

InvalidCredentialsEC: Invalid credentials The supplied real user name and password did not agree with the authentication database.

InvalidObjectEC: Operation/file mismatch An attempt was made to do an operation on a type of file that doesn't support it, for example a GetLock, SetLock, or SetExclusiveUse on a pipe or socket. It is also raised when an operation such as Read or Write (or Close) is applied to a TEm window that has been destroyed.

LookUpEC: No such file or directory A component of the path name was not found in the indicated directory.

MinorDeviceDoesNotExistEC: No such minor device number An attempt was made to open a device, but the minor device number was not known by the driver determined by the major device number.

NameTooLongEC: Name too long A component of a path name was longer than 255 characters, or, on Ultrix, an entire path name exceeded 1023 characters.

NoDriverForDeviceEC: No such major device number An attempt was made to open a device, but no device driver had been registered for the corresponding major device number.

NoMountedVolumesEC: No mounted volumes An attempt was made to look up a path name before any logical volume had been mounted.

NotADirectoryEC: Not a directory A component of a path name was not a directory.

NotATerminalEC: Not a terminal An attempt was made to specify as a control terminal a file other than a terminal device.

NotEnoughRoomEC: Not enough disk space There was insufficient space to create or extend a file.

NotEnoughVMEC: Not enough VM Starting a process would have required more virtual memory than allowed by the imposed maximum.

NotImplementedEC: Unimplemented operation An attempt was made to perform an operation that is not yet implemented but that should be. Consult the wrelease notes for details of a particular version.

NotOwnerEC: Not owner An attempt was made to perform an operation only permissible by the owner of a file or a process.

NotSuperUserEC: Not super-user An attempt was made to perform an operation only permissible by the super-user, e.g., supplying an explicit User specification to an operation such as Open.

NotTtyOwnerReadEC: Background write An attempt was made to read or copy from a device without being in its distinguished process group.

NotTtyOwnerWriteEC: Background read An attempt was made to write or copy to a device without being in its distinguished process group.

OkEC: No error No error (never raised).

OperationConflictEC: Operation/open mode mismatch An attempt was made to do an operation that is not compatible with the way the file was opened, e.g., a Write using a file handle opened for reading.

PipeHasNoReaderEC: Broken pipe An attempt was made to write to a pipe whose reading end is no longer open.

ProtectionViolationEC: Permission denied An attempt was made to modify a read-only server set or perform an operation on a file or directory whose current access mode disallowed it.

PvOfflineEC: Physical volume not online A physical disk volume required to perform the operation was offline. The volume needed should be logged, but isn't currently.

RanOutOfResourcesEC: Ultrix server ran out of file descriptors An open of a file residing on an Ultrix machine failed because of a lack of file descriptors.

RemoteFileEC: Remote file server call failed An RPC call to a remote server failed. The reason was logged in the tshell window. The call may have been partially executed on the remote machine.

ServerNotAvailableEC: Server not available A server needed to look up a remote path name was not available.

ShortExecutableEC: StartProcess header/file mismatch The a.out file supplied to StartProcess or StartProcessSearch was shorter than its header implied.

TooManyProcessesEC: Too many processes There were no more address spaces (on Taos) or process table entries (on Ultrix) when StartProcess or StartProcessSearch was called.

TooManySymbolicLinksInPathEC: Too many levels of symbolic links A possible loop in symbolic links was encountered (more than 5 remote symbolic links or 16 symbolic links per remote file machine).

UnseekableObjectEC: Illegal seek Seek was called with a pipe.

VolumeNeedsCheckingEC: Volume needs scavenging An inconsistency was found on a disk volume requiring that it be scavenged.

WouldBlockEC: Advisory lock conflict SetLock was called with noBlock equal to TRUE, and there was a conflict over the lock. (Read and Write never raise WouldBlockEC.)

B.2 The OSFriends Interface: VC

AlreadyMountedVC An attempt was made to mount a logical volume that is already mounted.

BusyVC An attempt was made to mount a logical volume that was being used by another file system, or to dismount a volume with open files or mounted volumes.

NotMountedVC An attempt was made to dismount a logical volume that was not mounted.

RootNotADirectoryVC An attempt was made to mount a logical volume with a root file that was not a directory.

VolumeHasFilesVC This code is obsolete and should be removed from the interface.

C Running Topaz Applications on Ultrix

It is possible to write many application programs that will work with either the Taos or the Ultrix implementation of Topaz. Specific differences between the two implementations of the OS interface are noted where relevant in Sections 3 through 5. This appendix discusses additional differences in the two implementations of the OS and other Topaz interfaces. (Perhaps someday there will be a separate Topaz Programmer's Manual.) Of necessity, this appendix discusses implementation details.

Although Taos emulates an approximation to the Ultrix 1.1 kernel-call interface (see Appendix D, page 64), there are different libraries for linking Topaz applications to run on Taos or Ultrix. Some applications linked for Ultrix may in fact run on Taos, but many will encounter runtime errors or will suffer bad performance.

Everything not mentioned in the following sections is implemented the same way on both systems.

C.1 C Library

The Ultrix Topaz library depends on a slightly modified version of the C library to call the Ultrix kernel. The Taos Topaz library does not depend on the C library. Because the C library is not reentrant, Topaz programs in general shouldn't use it.

C.2 Scheduling

On Taos, thread scheduling is done at the lowest level of the system (in the Nub) However, on Ultrix scheduling is done at two levels. The Ultrix kernel schedules processes as single-threaded entities, while the Ultrix Topaz library multiplexes a single-threaded Ultrix process to provide a multithreaded Topaz process.

The Ultrix implementation of Topaz implements nearly the full range of functions in the Thread, ThreadFriends, ThreadFriends1, ThreadsPort, TPFriends, and TPSpecial interfaces. The main omissions are the procedures dealing with other address spaces and the ‘Directed’ procedures (intended for use by debugging facilities).

The main observable differences in the Ultrix implementation are:

- Thread priorities are observed, but there is no time-slicing between threads at the same priority within a process. (Of course, the Ultrix kernel continues to perform time-slicing between separate Ultrix processes.)

- In some cases when a thread performs I/O on a device or socket there will be scheduling anomalies. A write to a terminal can block all the threads in a process if the output buffer for the terminal is full (this does not hold for the master side of a pseudo terminal). A read or write of a socket or device that doesn’t support asynchronous mode (SigIO) may wait for up to .5 second of CPU time to be used by lower-priority compute-bound threads in the process.

- Per-thread CPU time (as reported by TPSpecial.GetCPUTime) is not normally recorded. However, linking the object file /proj/ultrix/lib/enable_times.o into an Ultrix Topaz application causes per-thread CPU times to be recorded (at the cost of slowing down context switches by a factor of two). The CPU times so recorded include time spent in the Ultrix kernel on behalf of the thread as well as overhead for thread switching and signal handling.

C.3 Virtual Memory

The Ultrix implementation of the VM interface does not support unmapping pages. This means that stack overflow is not detected and that dereferencing a NIL REF is not detected unless the module is compiled with the -x switch and is either declared SAFE or is compiled with the -C switch. Dereferencing a NIL pointer is never detected. (See mm(1).)

C.4 Profiling

The Ultrix implementation of Topaz uses the standard Ultrix C library profiler, which uses the profil kernel call. (Taos has a different implementation of profiling.)

C.5 RPC

The Ultrix implementation of RPC transports all calls via the network (via Ultrix sockets), while the Taos implementation transports calls between processes on the same machine using a special ‘Xfer’ mechanism. (Taos uses the network for nonlocal calls, but it does not use sockets to access it.) The semantic differences between network and Taos local RPC calls are documented in rpc(3).

C.6 The Time Interface

The Ultrix implementation of the Now procedure is slower by an order of magnitude than the Taos implementation. (It must call the kernel rather than a local procedure.) Also the Ultrix implementation omits the NowTicks, TicksToTime, TimeToTicks, and NowLoc procedures.

C.7 The UID Interface

The Ultrix implementations of the GetCounter and GetFineCounter procedures return values that are only unique within the calling process.

C.8 The StableStorage Interface

The Ultrix implementation uses the file /tmp/StableStorage as storage.

C.9 The OS Interface

The Ultrix implementation of the type OS.Dir is a text string giving a path name (without symbolic links) to the directory. The Ultrix and Taos implementations behave differently when an open directory is renamed. Compared to Taos, the Ultrix implementation also causes OpenDir to be slower and GetPath to be faster. The Ultrix implementation of OpenDir is faster than usual if the path does not contain any / characters.

The Ultrix implementation of OS does not support remote file access via path names beginning with the # character. Thus such literal path names should not be coded within Topaz applications. Programs should either reference files in GF (the Topaz global file name space), or should use environment variables or other parameterization.

There are many obstacles to sharing open files between an Ultrix Topaz process and another process. Sharing a disk file that isn't in append mode will work correctly if the Topaz process uses OS.Read and OS.Write rather than OS.FRead and OS.FWrite. But there are other problems due to the use of several Ultrix features in the Ultrix implementation of the OS file operations:

- All devices other than terminals and sockets are placed into non-blocking mode, in order to avoid having an operation performed by one thread block within the Ultrix kernel and therefore prevent other threads within the same process from executing. Placing a terminal into non-blocking mode would cause unacceptable interference with the parent process, which is sharing the same open-file object, so it is not done.

- All files are placed in asynchronous mode, so that SigIO signals can be used to schedule I/O even in the presence of lower-priority compute-bound threads. Note that not all device types recognize asynchronous mode.

- All files are placed in non-append mode. This is required to implement OS.FWrite.

When OS.StartProcess is called, each file in the process template (set by OS.SetDescriptor) is unconditionally placed in blocking mode and in synchronous mode. Additionally, if the file had originally come from OS.GetDescriptor and had been in append mode, it is returned to append mode.

For documentation of the Ultrix file modes, see fcntl(2).

D Running Ultrix Applications on Taos

In addition to implementing the OS interface, Taos emulates an approximation to a subset of the Ultrix 1.1 kernel-call interface. The emulation is at the level of kernel-call traps, so existing Ultrix a.out files can be run without being recompiled or relinked. This appendix describes the omissions and (known) inaccuracies in the emulation.

The reasons for the omissions can be classified as fundamental or expedient. The fundamental omissions are those deemed infeasible to implement given the underlying structure of Topaz and Taos on which the Ultrix emulation rests. An example is the file /dev/kmem. The expedient omissions are those deemed unimportant compared to other projects competing for the developers' time. An example is sockets.

The reasons for the inaccuracies can be classified as bugs or features. The bugs are the usual problems of incomplete specification and imperfect implementation. There are no known examples. The features stem from the same causes as the fundamental omissions: conflict with underlying structure. An example is the substitution of user names and the user database (maintained using the Interim Name Service) for user identifiers and /etc/passwd.

During the design of the Taos Ultrix emulator, we attempted to justify each omission and variance by appealing either to the existence of alternate facilities in the Topaz environment or to the intended usage (i.e., supporting workstations and servers in an R&D environment). Experience so far has been positive.

Taos uses several techniques to minimize the inconvenience of features it does not fully implement. Some kernel calls raise the signal SIGSYS, some return an error, and some are no-ops, as specified in the following subsections.

D.1 Scheduling

The component of Taos responsible for processor scheduling is called the Nub. The Nub assigns threads to processors via a preemptive priority scheme with time-slicing and optional constraints on which processors can run which threads. There are three priority levels (normal, foreground, and background) available to user programs. There is no automatic adjustment of priorities.

The getpriority kernel call maps from Topaz to Ultrix priorities as follows: background becomes +10, normal becomes 0, and foreground becomes -10. If getpriority is applied to a process containing several threads, it returns the highest priority (lowest numerical value) of any thread.

The setpriority kernel call maps from Ultrix to Topaz priorities as follows: positive values become background, 0 becomes normal, and negative values become foreground. If setpriority is applied to a process containing several threads, it sets the priority of each of them to the same value. This is usually not a good idea.

Neither getpriority nor setpriority ever returns EACCES.

D.2 Virtual Memory

As with processor scheduling, most Taos virtual memory management is performed by the Nub. The Nub virtual memory manager does not currently provide facilities for sharing read-only program text segments or for demand loading.

The virtual memory page size implemented by the Nub is considered to be a tuning parameter, and may end up larger than the 1024 bytes used by Ultrix. This difference is visible to a program that uses the brk/sbrk kernel call, since this kernel call rounds up the break (the end of the data segment) to a multiple of the Nub-defined virtual page size. On the other hand, the getpagesize kernel call always returns 1024. (The Ultrix 1.1 version of the ld command won’t work if getpagesize returns a different value.)

Franz Lisp executes the vadvise kernel call even though no such kernel call is documented in Ultrix 1.1 or 2.0. Therefore Taos implements vadvise as a no-op. (There is a comment ‘72 is old: vadvise’ in /usr/include/syscall.h.)

Taos does not limit the number or total length of arguments and environment strings passed via the execve kernel call.

If virtual memory backing store runs out, Taos may fail in different ways than Ultrix. This is because a process can cause Taos to allocate an unbounded amount of virtual memory (within the Taos address space); when backing store runs out, a Taos thread holding global locks may be blocked.

D.3 Debugging and Profiling

Taos provides a different method than Ultrix for a debugger process to control a debuggee process. The Ultrix method uses the ptrace kernel call; it requires the debuggee to be a child of the debugger, and therefore to be resident on the same machine. The Taos method uses TeleDebug servers built into the Nub; the debuggee need not be a child of, nor even on the same machine as, the debugger. Even the Taos address space and the Nub itself may be debugged (this requires another machine to run the debugger). On Taos, the ptrace kernel call always raises SIGSYS.

Ultrix provides the profil kernel call for program-counter sampling. On Taos, the profil kernel call always raises SIGSYS; the Nub provides an alternate sampling facility. See the gprof package for implementation details. The -pg option works only with Modula-2+ programs, not for other languages such as C.

Taos interprets a range of negative operands to the chmk instruction as Nub calls rather than as errors reported via the SIGSYS signal.

D.4 Security

As discussed in Section 2.4, the Topaz security model differs significantly from that of Ultrix; luckily this has little or no impact on most application programs.

Internally, Taos associates user names rather than user identifiers with files and processes. However, it translates between names and identifiers as necessary at the kernel-call interface (e.g., the stat kernel call).

As described in Section 2.4, page 13, Taos always uses the real user name and password for validating access to remote files; it uses the effective user name for validating access to local files and for all other local operations. If the execve kernel call is given the path name of a remote file, it ignores the setuid bit. (In the future, Taos will allow ‘trusted’ servers to provide setuid programs.)

Taos does not implement the notion of the real and effective group identifiers of a process. The getgid and getegid kernel calls always return 0, and the setregid kernel call is a no-op.

Taos implements the group access list of a process in a different way than Ultrix. The setgroups kernel call is a no-op; instead Taos determines group membership by looking up the current effective and real user names in a database stored in the Interim Name Service. The getgroups kernel call always raises SIGSYS; the Interim Name Service querying facilities can be used instead. (See ns(8).)

Taos implements the setreuid kernel call by translating the user identifiers to user names and then calling OSFriends.SetMyUser (see page 53.) (The Taos implementation of setreuid also accepts an optional third argument: a character pointer specifying a password to pass to SetMyUser.)

Taos implements the access kernel call differently than Ultrix does. The Ultrix implementation of access uses the real user where other kernel calls would use the effective user; apparently the access kernel call was originally intended to allow a program running as effective user root to check whether the user who invoked it is allowed to access a specified path name with a specified mode. However, many programs (e.g., emacs and csh) invoke the access kernel call assuming that real = effective, and don’t work correctly when that assumption is false. For example, the enable command runs a program with effective = root and real = you. Therefore the Taos implementation of access always uses the effective user. This change may break certain setuid programs, but we haven’t encountered any yet.

The Ultrix login and su commands won’t work on Taos; see instead the Taos login and enable commands.

D.5 Files

Taos supports arbitrarily long path names, although it still limits each component of a path name (as defined by the nonterminal **<**pname**>** in the syntax defined in Section 2.6, page 15) to 255 characters, as does Ultrix.

Taos supports access to remote files via kernel calls just as through the OS interface. The kernel interprets path names starting with # as remote. To access a file #foo in the current directory, use ./#foo instead. For more information on remote path names see Section 2.6, page 15. Any operation on a remote file may return EIO, signifying that a remote file server call failed (see OS.RemoteFileEC). This may cause programs that don’t check error returns carefully to fail in strange ways. Note that a remote file operation is not interruptable by an asynchronous signal, so for example a Control-C may take a long time to respond.

Taos allows an unlimited number of descriptors. However, the select kernel call limits its mask parameters to 32 bits, and the dup kernel call only duplicates descriptors less than 64.

Taos does not serialize the write kernel call with respect to reads and other writes of the same bytes of the same file.

Taos doesn’t allow the root directory of the file system to be changed; the chroot kernel call returns EPERM unless called with a path name of / (and an effective user name of root).

Taos makes up the device numbers for file systems on the fly as file systems are accessed. These device numbers may be different on each boot of the system, although they will be unique and invariant during one run.

Taos maintains the atime, ctime, and mtime files attributes to the nearest microsecond. The times returned by the stat and fstat kernel calls include the microsecond information in the 'spare' field following each time field. The atime field for a device file is not kept up to date. (This is for efficiency.)

Ultrix supports 'holes' in files by not allocating disk storage for large unwritten regions, but Taos allocates and zeroes such regions.

Taos may allow more levels of symbolic links than Ultrix before returning the ELOOP error on a a path name lookup, especially when multiple machines are spanned. The length of the value of a symbolic link is unlimited.

Pipes on Taos support non-blocking I/O. A program can (but typically will not) be given a pipe with an internal buffer size different than 4096; a pipe created by the pipe kernel call always has a 4096-byte buffer.

Appendix D.10, page 68, lists the devices supported by Taos.

D.6 Communication

Taos does not currently emulate Ultrix sockets. This is an expedient rather than fundamental omission: adding sockets (the basic machinery, together with one more actual network interfaces) would not present any unusual difficulty and would add useful functionality (e.g., convenient access to VMS machines via DECnet). In the mean time, Taos includes a fairly comprehensive RPC implementation together with facilities that use it (including remote debugging, file access, display access, and login).

On Taos, the following socket-related kernel calls always raise SIGSYS: accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, recvfrom, recvmsg, send, sendmsg, sendto, setsockopt, shutdown, socket, and socketpair.

Taos implements the gethostname kernel call by calling NSUtil.GetMyHost. It implements the sethostname kernel call by always returning EPERM.

Ultrix commands that use sockets won't work on Taos. Alternatives for the most common are available: monarch for rlogin, msh for rsh, and cp or the more efficient copy for rcp. There is no direct analogue for rdist, but several programs provide similar functionality. The copy command maintains time stamps and subdirectory structure. The updatefs command maintains consistency between the 'GF' file system and a local Firefly file system. The 'package tools' (getpackage, shippackage, etc.) facilitate sharing 'project' subdirectories among a set of developers. (See also vget(1), vlink(1), and vship(1).)

D.7 Resource Allocation and Accounting

Taos is intended for use in environments with responsible users and plenty of resources (e.g., disk space and processor cycles). Therefore it was decided that it wasn't important to provide much support for resource allocation or accounting.

Taos does not implement disk quotas; the quota and setquota kernel calls always raise SIGSYS.

Taos, unlike Ultrix, does not reserve extra space on each volume for use only by the super-user (see the minfree parameter in mkfs(8)).

Only some of the resource limits associated with the getrlimit and setrlimit kernel calls are implemented, namely data, stack, and core. The other limits, namely cpu, fsize, and rss, are not implemented: setrlimit no-ops and getrlimit returns 0x7fffffff as both the hard and soft limit.

Most of the resource usage information returned by the getrusage and wait3 kernel calls is always zero; only the user time and system time are maintained.

Taos does not implement the accounting file; the acct kernel call always raises SIGSYS.

The Ultrix ps command doesn't run on Taos because it depends on /dev/kmem; for an alternative, see si(1).

On Taos, the virtual and profiling interval timers are quite expensive to use; the real interval timer is not.

D.8 System Operation

Taos handles many aspects of system operation differently than Ultrix. This has little effect on normal user programs, but does mean that most of the programs documented in Section 8 of the Ultrix Programmer's Manual won't run on Taos. (For Taos analogues, see backup_ff(1), cron(1), df(1), expandpath(1), homeserver(1), lpr(1), mailq(1), pwd(1), restore_ff(1), servername(1), setpgrpme(1), stripcp(1), automount(8), cleardisks(8), crontab(8), dismount(8), givevm(8), installboot(8), mkinit(8), sendmail(8), setconfigurationparameter(8), startrfs(8), tsh(n), and *The Firefly Companion*.)

The Taos file flushing daemon runs every 60 seconds and doesn't flush changes to file access times, whereas the analogous Ultrix update system operation command runs every 30 seconds and flushes all attributes including file access times.

On Taos, there are no block devices or raw disk devices; the mount and umount kernel calls always raise SIGSYS. Analogous facilities are available through the OSFriends interface (see Appendix A.5, page 55).

On Taos, there is no /dev/kmem or /dev/mem file; see instead the OS.GetProcessInfo procedure described in Section 5.5.

On Taos, the swapon kernel call always raises SIGSYS; VMFriends.Donate serves an analogous purpose.

On Taos, the reboot kernel call always raises SIGSYS. (However, its functionality would be convenient for the remote operation of servers).

On Taos, the vhangup kernel call always raises SIGSYS.

On Taos, the settimeofday kernel call always returns EPERM (the tsh settime command can be used instead; see tsh(n)).

D.9 Taos-only Kernel Calls

Taos implements a few new kernel calls, for use where importing OS or a related, RPC-based interface would be undesirable or impossible. Until man pages are written, consult the implementation modules:

Name	*#*	*Implementing package/module*
copy	7	fs/UxFile.mod
nsenumerate	13	fs/UxFile.mod
checkpw	18	ux/UxPro.mod
terminate	63	ux/UxPro.mod
getpw	77	ux/UxPro.mod
dummy	103	ux/UxAS.mod
reservetraps	107	ux/UxAS.mod
strace	147	ux/UxPro.mod
getsignal	115	ux/UxPro.mod (obsolete)
getspaceinfo	119	ux/UxPro.mod (obsolete)
startnewspace	132	ux/UxPro.mod (obsolete)

D.10 Taos Devices

The following table lists the devices supported by Taos:

Major	*Minor*	*Description*	*Normal path name*
2	0	Console device or window	/dev/console
3	0	Null	/dev/null
3	1	Control terminal	/dev/tty
4	0	UART	/dev/tty00
5	0	TEm window 24x80	/dev/ety
6	0-30	Pty slave	/dev/tty[p-r][0-9a-f]
7	0-30	Pty master	/dev/pty[p-r][0-9a-f]
8	0	Ivy typescript window	/dev/ts

The tty driver (used for the console device, TEm windows, and pseudo terminals) always uses the ‘new line discipline’; the TIOCSETD ioctl is a no-op and the TIOCGETD ioctl always returns 2 (NTTYDISC).

The tty driver treats several other ioctl requests as no-ops: TIOCSBRK/TIOCCBRK (set/clear break bit), TIOCSDTR/TIOCCDTR (set/clear data terminal ready), TIOCHPCL (hang up on last close), and TIOCWONLINE (wait for device to come online).

The tty driver does not support packet mode for pseudo terminal devices. Using the TIOCPKT ioctl to clear packet mode is a no-op; using TIOCPKT to set packet mode raises SIGSYS.

Finally, the tty driver treats another group of ioctl requests as ASSERT failures that stop Taos (this should be changed!): TIOCSMLB/TIOCCMLB (set/clear loopback mode), TIOCSFIFO/TIO-CCFIFO (set/clear FIFO transmission mode), TIOCMODG/TIOCMODS (get/set modem control state), TIOCMSET/TIOCMGET (set/get modem bits), TIOCMBIS/TIOCMBIC (bis/bic modem bits), TIO-CNMODEM/TIOCMODEM (ignore/don’t ignore modem status), TIOCNCAR/TIOCCAR (ignore/don’t ignore soft carrier), and TIOCOUTQ (return size of output queue).

The only way to set the actual line speed of the console device is to turn the rotary switch on the back panel of the Firefly; the line speed returned by the TIOCGETP ioctl and set by the TIOCSETP and TIOCSETN ioctls is unrelated to the rotary switch position.

E Topaz Interfaces Imported by OS and OSFriends

To make this manual more self-contained, we briefly describe each of the definitions from other Topaz interfaces that are imported by the OS and OSFriends interfaces.

E.1 Base

The Base interface contains miscellaneous definitions imported by many other Topaz interfaces.

```
EXCEPTION
  Alerted;
```

Alerted is the exception that by convention is raised to indicate that a thread has been alerted.

E.2 NubTypes

The NubTypes interface contains miscellaneous definitions used by many other Nub interfaces.

```
TYPE
  Space = [0..127];
```

A Space is a number used to identify a Taos address space. By convention, 0 refers to the Nub and 127 never refers to any space.

E.3 Rd

The Rd interface provides the abstraction of a readable stream, that is a source of characters. Rd is a class interface; a number of specific implementations exist, for example for reading from a Text, a file, or a terminal. (The Wr interface provides the complementary abstraction of a writable stream.)

```
TYPE
  T = REF;
```

A value of type T represents an instance of a reader. (The type Rd.T is actually equated to type Text.Reader.)

E.4 System

The System interface is defined by the Modula-2+ language, and contains standard types known to the compiler.

```
TYPE
  Byte = BITS 8 FOR Word;
```

The type Byte is parameter-compatible with any 8-bit value; an open array of Byte is parameter-compatible with any value occupying a multiple of 8 bits.

E.5 Text

The Text interface provides the abstraction of immutable strings (finite sequences) of characters.

```
TYPE
  T = REF;
  RefArray = REF ARRAY OF T;
```

A value of type T represents a Text, that is an immutable character string.

E.6 ThreadFriends

The ThreadFriends interface contains specialized facilities associated with the Thread interface.

```
TYPE
  Priority = [0..7];
```

A value of type Priority is used to control the assignment of threads to processors: threads with higher priorities are considered before threads with lower priorities. (The type ThreadFriends.Priority is actually equated to TPFriends.Priority.)

```
CONST
  BackgroundPriority = 1;
  NormalPriority = 2;
  ForegroundPriority = 3;
```

Only the three priorities in the range BackgroundPriority to ForegroundPriority, inclusive, should be used for threads in application programs. ForegroundPriority should be used only for brief computations, such as tracking the keyboard and mouse.

E.7 Time

The Time interface contains types for describing absolute times and time intervals; it also contains procedures for doing arithmetic on time values, for reading and setting the system clock, and for delaying the execution of the calling thread for a specified amount of time.

```
TYPE
  T = RECORD
    seconds: CARDINAL;
    microseconds: [0..999999];
  END;
```

A value of type T represents either an instant in time relative to the epoch (January 1, 1970 at 00:00:00.0 GMT) or the length of an interval in time. (The type Time.T is actually equated to NubTypes.Time.)

E.8 UID

The UID interface contains types and procedures for generating unique identifiers.

```
TYPE
  Counter = CARDINAL;
  Constant = ARRAY [0..5] OF NubTypes.Byte;
```

Each call of the procedure GetCounter returns a value of type Counter that is guaranteed to be different than the values returned by all previous calls to GetCounter on the same machine. The procedure GetConstant returns a value of type Constant that is guaranteed to be different than the value returned by calls to GetConstant on any other machine. Thus the concatenation of a Counter and a Constant is unique across all machines over all time.

Acknowledgments

The Topaz system as a whole is the work of many people at SRC. The guidelines for alertability in Section 3.3 of Part I resulted from discussions with Andrew Birrell and Roy Levin. The OS interface and its reference manual (Part II) profited from Andy Hisgen's careful reading and comments. Andrew Birrell and Michael Schroeder provided valuable advice on the overall design. John DeTreville, Cynthia Hibbard, Michael Schroeder, and Roger Needham provided comments that improved Part I of this report.

Index