Evolving the UNIX System Interface to Support Multithreaded Programs

Paul R. McJones and Garret F. Swart
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

Abstract

Allowing multiple threads to execute within the same address space makes 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. We present several 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.

1. Introduction

Most existing general-purpose operating systems place in one-to-one correspondence virtual address spaces and threads, where by a thread we refer to the program counter and other state recording the progress of a sequential computation. This one-to-one correspondence between address spaces and threads makes it more difficult to construct applications dealing with asynchrony and to exploit the speed of multiprocessors. To address these problems, several newer operating systems allow multiple threads within a single virtual address space. The existence of multiple threads within an address space places 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 (SRC). We show how we used these guidelines to evolve the Topaz interface from the 4.2BSD UNIX [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 [19] 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 [14].

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, which are 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.

---

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

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 [12] and Xerox's Cedar [17]. Topaz borrows the 4.2BSD file system semantics and large-grain process structure, populates these processes (address spaces) with Cedar-like threads, and interconnects them with Cedar-like remote procedure call [5]. 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 all of the state components that a UNIX process has, such as virtual memory, a set of open files, a user id, and signal-handling information. While a UNIX process has only one stack and set of registers, a Topaz address space has a separate stack and set of registers for each thread of control living in that address space.

A Topaz programmer can use threads for fine-grained cooperation, as is done in the Cedar system. Unlike a Cedar programmer, a Topaz programmer can also use multiple address spaces to separate programs of different degrees of trustworthiness. Many Topaz address spaces contain long-running servers handling remote procedure calls from other address spaces on the same or different machines.

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. One example is that the Topaz remote procedure call mechanism executes concurrent incoming calls in separate threads. Another example is that the Topaz window system uses several threads in a pipeline arrangement to spread the work of transporting and processing painting requests over several CPUs.

Modeling threads as separate UNIX processes would mean that threads could not freely share open file descriptors, since UNIX only allows these descriptors to be inherited by a child process from its parent process. It would also be difficult to share pointer-containing data structures among threads modeled as separate UNIX processes, since a pointer into the stack segment would have a different meaning in each process. Many Topaz applications create dozens or hundreds of threads. This would be slow and extravagant of kernel resources if each was a full UNIX process, even if most of the virtual memory could be shared.

A Topaz application is written as if there is a processor for every thread; the implementation of Topaz 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 [9]; 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 [7], 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 [8]). In this section we present three guidelines abstracted from our experience designing the Topaz operating system interface.

Our first guideline addresses an aspect of interface design that is complicated by multiple threads: avoiding unnecessary serialization to mutable state defined by the interface. Our second guideline addresses an aspect of interface design that is simplified by multiple threads: dealing with asynchrony without resorting to ad hoc techniques. Our third guideline addresses the problem of cancelling undesired computations in a multithreaded program.

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 store the generator state in a record and to pass a pointer to this record on each call of the generator.

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 by 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. The use of a multiway waiting procedure hinders program modularity, since it requires centralized knowledge of all asynchronous operations initiated anywhere in the program.

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. For example, 4.2BSD UNIX provides polling, waiting, and interrupt mechanisms. 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`** operation 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 to 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 Topaz system provides the *alert* mechanism [4] for this purpose; it is similar to Mesa's *abort* mechanism [9]. 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, and therefore 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 in an effort to define 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 Topaz 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+ [16], which extends Wirth's Modula-2 [20] with concurrency, exception handling, and garbage collection. The facilities of Topaz are provided through a set of interfaces, 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 UNIX facilities. More information about the Topaz OS interface can be found in its reference manual [10].

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 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, for reasons that had little to do with the presence of multiple threads. It is worth noting that exceptions have the advantage over return codes that they can't be accidentally ignored, since an exception which has no handler results in abnormal termination of the program. This problem is serious enough that UNIX uses signals to report certain synchronous events; for example **`SIGPIPE`** is raised when a process writes to a pipe whose reading end is no longer in use.

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 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 indices 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 indices 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, such as 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 a 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 [18]. 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 **`StartProcess`**, 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 UNIX 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.

Process Group.	Constant (fixed at the time the process is created).
User identity.	Constant/parameterized. Most processes run with a fixed user identity. A super-user program, usually a server, acting on behalf of many users may pass an additional parameter on each call giving the user identity on whose behalf it is acting.
Control Terminal.	Constant.
UMask.	Client-synchronized. (The umask is used to set the access control bits when a file is created.)
Priority.	Thread-dependent.

4.6. Summary of control-structure changes

We can summarize the changes to the control structure of 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, for example to run a parallel version of the UNIX make [15].

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.

Acknowledgments

The Topaz system as a whole is the work of many people at SRC. The guidelines for alertability in Section 3.3 resulted from discussions with Andrew Birrell and Roy Levin. The OS interface and its reference manual 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 this paper.

References

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

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

3. Bach, M. J. and Buroff, S. J. "Multiprocessor UNIX operating systems". *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct. 1984), 1733-1749.

4. Birrell, A. D., Guttag, J. V., Horning, J. J., and Levin, R. Synchronization primitives for a multiprocessor: a formal specification. Proceedings of the Eleventh Symposium on Operating System Principles, ACM, New York, Nov., 1987, pp. 94-102.

5. Birrell, Andrew D. and Nelson, Bruce Jay. "Implementing remote procedure calls". *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39-59.

6. Hoare, C. A. R. "Monitors: an operating system structuring concept". *Comm. ACM 17*, 10 (Oct. 1974), 549-557.

7. Jonathan Kepecs. Lightweight processes for UNIX implementation and applications. USENIX Association Conference Proceedings, June, 1985, pp. 299-308.

8. Lampson, Butler W. "Hints for computer system design". *IEEE Software 1*, 1 (Jan. 1984), 11-28.

9. Lampson, Butler W. and Redell, David D. "Experience with processes and monitors in Mesa". *Comm. ACM 23*, 2 (Feb. 1980), 105-117.

10. McJones, Paul R. and Swart, Garret F. The Topaz operating system programmer's manual. In *Evolving the UNIX System Interface to Support Multithreaded Programs, Research Report #21*, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, 1987.

11. Parnas, D. L. "On the criteria to be used in decomposing systems into modules". *Comm. ACM 15*, 12 (Dec. 1972), 1053-1058.

12. Quarterman, John S., Silberschatz, Abraham, and Peterson, James L. "4.2BSD and 4.3BSD as examples of the UNIX system". *Comput. Surv. 17*, 4 (Dec. 1985), 379-418.

13. Rashid, Richard F. "Threads of a new system". *UNIX REVIEW 4*, 8 (Aug. 1986), 37.

14. Ritchie, Dennis M. and Thompson, Ken. "The UNIX time-sharing system". *Comm. ACM 17*, 7 (July 1974), 365-375.

15. Roberts, Eric S. and Ellis, John R. parmake and dp: Experience with a distributed, parallel implementation of make. Proceedings of the Second IEEE-CS Workshop on Large Grained Parallelism, Oct., 1987.

16. Rovner, Paul. "Extending Modula-2 to build large, integrated systems". *IEEE Software 3*, 6 (November 1986), 46-57.

17. Swinehart, Daniel C., Zellweger, Polle T., and Hagmann, Robert B. The structure of Cedar. Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, ACM, New York, June, 1985, pp. 230-244.

18. Tevanian, Avadis, Jr., Rashid, Richard F., Golub, David B., Black, David L., Cooper, Eric, and Young, Michael W. Mach Threads and the Unix Kernel: The battle for control. USENIX Association Conference Proceedings, Phoenix, June, 1987, pp. 185-197.

19. Thacker, Charles P. and Stewart, Lawrence C. Firefly: a multiprocessor workstation. Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, ACM and IEEE Computer Society, Oct., 1987.

20. Wirth, Niklaus. *Programming in Modula-2*. Springer-Verlag, 1985.