Adding Threads to Standard ML

Eric C. Cooper J. Gregory Morrisett

December 1990

CMU-CS-90-186

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We have added multiple threads of control to the Standard ML programming language. Standard ML's support for first-class functions and automatic storage management influenced the design in a number of ways. We demonstrate how other concurrency and synchronization operations, such as cobegin/coend, futures, and events, can be implemented in terms of the thread interface. Finally, we describe three implementations of the thread interface: a coroutine version, a uniprocessor preemptive version, and a multiprocessor Mach-based version.

This research was sponsored in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035, and in part by a National Science Foundation Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: threads, Standard ML, parallel programming, Mach, continuations, coroutines, processes, mutual exclusion, synchronization, shared memory

1 Introduction

There is no way to express explicit parallelism in the Standard ML programming language [26]. A compiler may, of course, introduce parallelism implicitly, but the state of the art in this area is not sufficiently advanced for our purposes. Instead, we need a means of creating and manipulating multiple threads of control within a single Standard ML program.

A *thread* represents a sequential flow of control or an abstraction of a processor. Threads are also known as *lightweight processes*, but we avoid that term to eliminate confusion with the UNIX[1] notion of (heavyweight) process. An implementation of threads may require support from the compiler, the language runtime system, and the underlying operating system.

Threads are needed for two main reasons: to express the naturally concurrent structure of distributed and interactive systems, and to achieve parallelism on real multiprocessors. Programs constructed with multiple threads can be simpler and more modular than programs using alternative mechanisms.

A system constructed from multiple threads can use simple synchronous interfaces that are easy to understand. Whether or not a module is internally multithreaded is purely an implementation issue, and is not visible in its interface. This improves modularity and allows programs to be developed by composition, with no fear of unpredictable interactions between components.

Implementors of distributed and interactive applications must deal with asynchronous events such as incoming network messages, users' keystrokes, expiration of timers, and so on. The principles of modularity and information hiding dictate that different events be detected and processed in different modules. The use of multiple threads, each waiting for the appropriate class of events, supports this programming methodology.

The alternatives to multiple threads in systems programming languages include software interrupts; non-blocking operations that permit polling; or an operation that allows a program to wait for any of a set of events. In UNIX, for example, all of these mechanisms are provided, adding considerable complexity to the system interface.

None of the alternatives to threads provides modularity. If polling or some form of `select` operation is used, the programmer must write one portion of code to detect and dispatch events appropriately to their handlers; this polling loop violates modularity by "knowing" about all the event-processing modules in the system. Similarly, if software interrupts (such as UNIX signals) are used, logically unrelated modules must share a common signal handler.

We have therefore added multiple threads to Standard ML. We describe the design of a thread module that can be used directly by parallel applications, as well as by higher-level constructs for parallel programming (such as Multilisp *futures* [21] and CSP communication channels [23]). The thread package is a Standard ML module; no modifications to the syntax of the language were made.

1.1 Standard ML

The Standard ML programming language (SML) is a mostly functional language with a complete formal specification of its semantics [26]. It provides first-class functions (closures), static (compile-time) typing, polymorphism, exceptions, automatic storage management (garbage collection), and a powerful module facility.

The Standard ML of New Jersey (SML/NJ) implementation [3, 7] supports type-safe, first-class

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

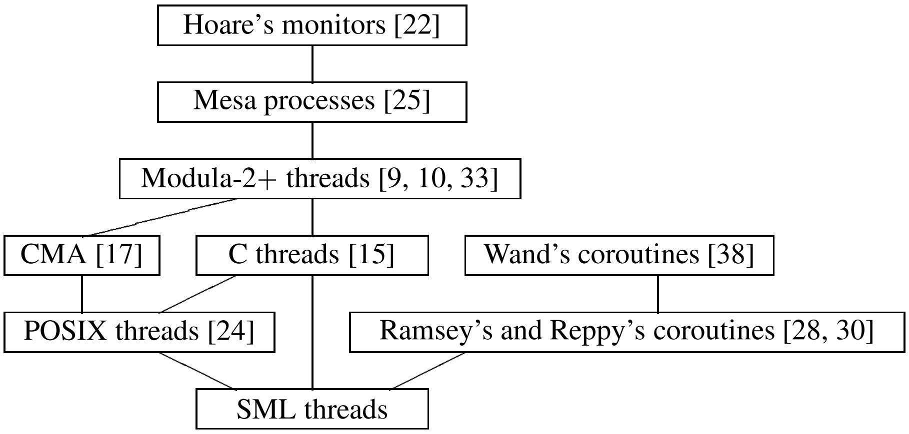

Figure 1: Genealogy of thread interfaces

continuations with *callcc* and *throw,* and provides asynchronous exception handling facilities in the form of signal handlers [32]. We have found that these extensions, along with the clean organization of the compiler and runtime system, make experimentation with threads considerably easier.

1.2 Related Work

The conceptual ancestor of most of the thread interfaces in common use is Hoare's monitor construct [22]. This proposal separated the two most common uses of Dijkstra's semaphores (*P* and *V* operations) into *mutual exclusion* and *synchronization via condition variables.* This separation, along with syntactic support in the programming language, eliminated many sources of errors with semaphores.

The first practical use of the monitor idea occurred at Xerox PARC, with the implementation of processes in Mesa [25]. The precise semantics of Hoare monitors were modified, however, to permit a simpler, more robust implementation. In particular, no guarantee was made that signaling a process would result in it being the next process to enter the critical region. As a result, a "wakeups as hints" style is required, in which processes always recheck the condition for which they are waiting, and go back to sleep if necessary.

At DEC SRC, the researchers who implemented processes and monitors in Mesa went on to add threads to Modula-2 for the Firefly multiprocessor workstation. The DEC SRC thread design [9, 10, 33] is the ancestor of various thread interfaces for UNIX and C [15, 17, 24], as well as the current work. This similarity in ancestry has an important practical implication: there is a large, common subset of functionality in all of these packages. For example, Birrell's excellent tutorial on programming with threads [10] is directly applicable to the SML threads described here.

Other researchers have examined concurrency in conjunction with SML. However, most have adopted message passing as the means of communication and synchronization. Reppy's work on first-class synchronous operations [31] describes an implementation of coroutine-based threads and a number of higher-level constructs (*channels* and *events*). Ramsey presents a similar message-passing, coroutine threads package based on CSP [28]. Both implementations use continuations to simulate concurrency [38].

The genealogy of the SML thread module described here is summarized in Figure 1.

```
signature THREAD =
    sig
        val fork : (unit -> unit) -> unit
        val exit : unit -> ’a
        val yield : unit -> unit

        type mutex
        val mutex : unit -> mutex
        val acquire : mutex -> unit
        val try_acquire : mutex -> bool
        val release : mutex -> unit
        val with_mutex : mutex -> (unit -> ’a) -> ’a

        type condition
        val condition : mutex -> condition
        val mutex_of : condition -> mutex
        val with_condition : condition -> (unit -> ’a) -> ’a
        val signal : condition -> unit
        val broadcast : condition -> unit
        val wait : condition -> unit
        val await : condition -> (unit -> bool) -> unit

        exception Undefined
        type ’a var
        val var : unit -> ’1a var
        val get : ’a var -> ’a
        val set : ’a var -> ’a -> unit
    end
```

Figure 2: The SML Thread Interface

2 Threads in Standard ML

The SML thread interface is given in Figure 2. In the following sections, we describe the interface and give some justification for each of the constructs.

2.1 Creation and Destruction

The basic abstraction in the thread module is, naturally, the notion of a thread: a sequential activity in a computation. Although a *cobegin ... coend* control structure is one way of (implicitly) creating and destroying threads, most current thread packages use a *fork ... join* model. This is more general, since it can create arbitrary thread relationships, not just nested ones. We follow a slightly different approach, providing *fork* but not *join.*

```
val fork : (unit -> unit) -> unit
```

The `fork` function starts an invocation of its argument executing as an independent thread of control. The effect is similar to simply calling the function, except that the caller and callee proceed in separate threads.

Although the function in the forked thread takes no argument, closures can be used easily to simulate passing arguments to the child:

```
fun fork_with_argument f x = fork (fn () => f x)
```

(The `fn` construction in SML denotes a lexically closed anonymous function, like a `lambda` expression in Scheme.)

The child function returns no result; it is executed purely for effect. Results can be communicated between threads via shared mutable objects, and wrapper functions can be used to implement the necessary synchronization and termination protocol. An example, in the form of a futures module, is presented in Section 3.2.

The SML `fork` operation differs from its ancestors in Figure 1 in that no "handle" is returned to identify the newly created thread of control. Indeed, there is no "thread" data type exported by the interface at all. Instead, the *per-thread state* described in Section 2.4 generalizes the notions of thread identifier and thread-local variables.

Thread termination occurs either implicitly when the top-level function of the thread returns, or explicitly when `exit` is called.

```
val exit : unit -> 'a
```

The `exit` function terminates the thread that calls it; since it never returns, its invocation is considered to have arbitrary type.

No synchronization with other threads occurs upon exit, and no *join* operation is provided at this level. Application-specific termination protocols can be implemented by using an appropriate wrapper function as the argument to `fork`; the implementation of futures in Section 3.2 offers one example of this.

2.2 Mutual Exclusion

If two threads perform conflicting operations on the same data, the result is unpredictable—it depends on details of scheduling, relative execution speed, compiler code generation, and hardware architecture. A classic example is an attempt by two threads to increment a counter at the same time. The arbitrary interleaving of each thread's load-increment-store sequence results in the counter being incremented by an unpredictable amount. This problem is solved by introducing *mutual exclusion locks,* of type `mutex`, along with `acquire` and `release` operations.

```
type mutex
val mutex : unit -> mutex

val try_acquire : mutex -> bool
val acquire : mutex -> unit
val release : mutex -> unit
```

The `mutex` function creates a new mutex value.

The `acquire` operation attempts to lock a mutex and does not return until it succeeds, at which point the calling thread is said to hold the mutex. At most one thread may hold a given mutex at any time. The case of a thread attempting to acquire a mutex it already holds is *not* treated specially; the thread will block forever.

The `try_acquire` operation is similar to `acquire` except that it does not wait to acquire the mutex; it tries once and returns an indication of whether or not it succeeded. For example, a busy-waiting version of the `acquire` function could be written in terms of `try_acquire` as follows:

```
fun acquire mutex =
    if try_acquire mutex then () else acquire mutex
```

The `release` operation unlocks a mutex, giving other threads a chance to acquire it.

The following code uses mutex operations to increment a counter safely:

```
val m = mutex ()
val counter = ref 0

acquire m;
counter := (!counter) + 1;
release m
```

The `acquire` and `release` operations on a given mutex must always be correctly paired, even in the presence of exceptions. This common source of errors is remedied in other languages by additional syntax, such as the `LOCK` statement of Modula-3 [13]. We can achieve the same effect in SML simply by delaying the body:

```
val with_mutex : mutex -> (unit -> 'a) -> 'a

fun with_mutex m body =
    let val result = (acquire m;
                      body () handle exn =>
                           (release m; raise exn))
    in
        release m;
        result
    end
```

The `with_mutex` operation acquires the mutex while executing the given function, then releases it before returning the value of the function call. It also catches any exception raised and releases the mutex before re-raising it. The support for closures in SML makes this construct usable directly, without the additional syntactic sugar required in Modula-3.

The example of incrementing a counter can now be expressed as follows:

```
with_mutex m (fn () => counter := (!counter) + 1)
```

2.3 Synchronization

A *condition variable* allows one thread to wait until another thread indicates that some event has occurred. The association between the condition variable and this event is maintained entirely by the application. The event is typically a change to shared data, and requires some application-specific test to detect. A mutex must be used to prevent one thread from testing the shared data while another is updating it; this mutex is specified at the time the condition is created.

```
type condition
val condition : mutex -> condition
val mutex_of : condition -> mutex
val with_condition : condition -> (unit -> 'a) -> 'a

val signal : condition -> unit
val broadcast : condition -> unit
val wait : condition -> unit
```

The `condition` function creates a new condition variable, to be used under the protection of the specified mutex.

The `mutex_of` function returns the mutex associated with a condition. The `with_condition` function is just the composition of `with_mutex` and `mutex_of`.

The `signal` operation is used to indicate that an event has occurred. If any threads are waiting for the specified condition variable, at least one of them is awakened. The `broadcast` operation is similar, except that it guarantees to awaken *all* threads waiting for the condition.

The `wait` operation atomically releases the mutex associated with the specified condition and waits for another thread to signal it. The awakened thread then reacquires the mutex before returning. The application must ensure that the event associated with the condition can occur only while the mutex is held.

Other threads may execute between the time that the condition is signaled and the time that the caller reacquires the mutex. One must therefore view wakeups merely as hints, and always retest the shared data. The `await` operation implements this as follows:

```
val await : condition -> (unit -> bool) -> unit

fun await c test =
    if test () then
        ()
    else
        (wait c; await c test)
```

The `yield` operation advises the runtime system to schedule another thread to run on the current processor.

```
val yield : unit -> unit
```

2.4 Thread State

Thread state is provided by the `var` type constructor and its associated operations. A `var` is similar to a `ref`, but the contents of a `var` are maintained on a per-thread basis, rather than shared among all threads.

```
type 'a var
val var : unit -> '1a var

exception Undefined
val get : 'a var -> 'a
val set : 'a var -> 'a -> unit
```

Unlike a `ref`, a `var` that is defined in one thread may be undefined in another, hence dereferencing it may raise the exception `Undefined`. Weak type variables are required to handle polymorphic `var` types, in the same way they are used for polymorphic `ref` types.

This abstraction of per-thread state allows different "subsystems" to define their own forms of thread identification without conflicting with one another. For example, a lock package might only need to identify threads with unique values of some type that admits equality, so it can tell whether a requesting thread already holds a lock. A master-slave package might need to guarantee that the thread IDs had additional properties, like lying in the range $1 \ldots N$. Another system might require per-thread transaction IDs. Rather than choosing a single form of thread ID, different packages can use different per-thread variables whose values have whatever semantics they need. An example, in the form of *recursive mutex locks,* is given in Section 3.1.

3 Building Abstractions on Top of Threads

The SML thread interface was designed to provide the minimal set of constructs and mechanisms needed to support efficient concurrent programming. In this section, we demonstrate how higher-level constructs and mechanisms can be built on top of the thread interface.

3.1 Recursive Mutex Locks

As was stated before, the case of a thread attempting to acquire a lock a second time before releasing it is not treated specially. In some implementations, the thread could block forever. Recursive mutex locks allow a thread to acquire a lock any number of times before releasing it a corresponding number of times. Often, this can make composition of subsystems easier.

Recursive mutex locks are implemented by a functor parameterized by the `Thread` structure, as shown in Figure 3. This example shows how the basic thread interface can be used to implement higher-level facilities. In particular, it demonstrates how per-thread state can be used by one particular subsystem without having to worry about name clashes with other subsystems.

3.2 Futures

Here is how one can define a variant of Multilisp futures [21].

```
signature REC_MUTEX =
    sig
        type T
        val new : unit -> T
        val lock : T -> unit
        val unlock : T -> unit
    end

functor Rec_Mutex (Thread : THREAD) : REC_MUTEX =
    struct
        datatype thread_id = ID of unit ref

        val self : thread_id Thread.var = Thread.var ()

        fun me () = Thread.get self
            handle Thread.Undefined =>
                let val id = ID (ref ())
                in
                    Thread.set self id; id
                end

        datatype T = RM of { owner : thread_id ref,
                             count : int ref,
                             mutex : Thread.mutex }

        fun new () = RM { owner = ref (me ()),
                          count = ref 0,
                          mutex = Thread.mutex () }

        fun lock (RM { owner, count, mutex }) =
            if !count = 0 orelse !owner <> me () then
                (Thread.acquire mutex; count := 1; owner := me ())
            else
                inc count

        exception NotOwner

        fun unlock (RM { owner, count, mutex }) =
            if !count = 0 orelse !owner <> me () then
                raise NotOwner
            else if !count = 1 then
                (dec count; Thread.release mutex)
            else
                dec count
    end
```

Figure 3: Recursive mutex locks

```
signature FUTURE =
    sig
        type 'a future
        val future : ('a -> '2b) -> 'a -> '2b future
        val touch : 'a future -> 'a
    end

functor Future (Thread : THREAD) : FUTURE =
    struct
        datatype 'a cell = BUSY | DONE of 'a | EXN of exn

        datatype 'a future =
            FUTURE of Thread.condition * 'a cell ref

        fun future function arg =
            let val c = Thread.condition (Thread.mutex ())
                val f = ref BUSY
                fun wrapper () =
                    let val result = DONE (function arg)
                                     handle exn => EXN exn
                    in
                        Thread.with_condition c
                        (fn () => (f := result; Thread.broadcast c))
                    end
            in
                Thread.fork wrapper;
                FUTURE (c, f)
            end

        fun touch (FUTURE (c, f)) =
            let fun touch' () =
                case !f of
                    BUSY => (Thread.wait c; touch' ())
                  | DONE x => x
                  | EXN exn => raise exn
            in
                Thread.with_condition c touch'
            end
    end
```

Figure 4: A variant of Multilisp futures

An α future represents a computation of a value of type α that may not yet have finished. To use the value, the function `touch` must be applied to the future; this will block if necessary until the computation is finished. Unlike Multilisp futures, which can be used interchangeably with "normal" values, our variant requires that `touch` always be used, even after the future has completed.

The implementation of futures in Figure 4 shows how result-producing computations can be started as threads, with appropriate wrapper functions to provide synchronization between the producer and consumers of the results. This example shows that a "join" protocol need not be provided by the threads module.

A *cobegin* ... *coend* control structure can be implemented trivially in terms of the `future` and `touch` functions:

```
val cobegin : (unit -> unit) list -> unit

fun cobegin fns =
    app touch (map (fn f => future f ()) fns)
```

The `cobegin` function takes a list of procedures, starts each one executing in its own thread, and then waits for them all to finish before returning.

3.3 Channels

We can use the thread interface to define a simple buffered message passing system, as shown in Figure 5. We use bounded ($n = 1$) buffers with mutual exclusion to facilitate the communication. The `create` operation creates a new typed channel.[2] The `put` and `get` operations allow one to send and receive values through a channel. Note that a `get` operation blocks until the buffer is non-empty, while the `put` operation blocks until the buffer is non-full.

Figure 6 shows how we can simulate remote procedure call (RPC) using channels. An RPC-value consists of an α channel used for input, and a β channel used for output. The `accept` operation takes an RPC-value and an $\alpha \rightarrow \beta$ function, gets an α value from its input channel, applies the function to that value, and puts the result of type β in its output channel. The `call` operation takes an RPC-value and an α value, sends the value to the acceptor, waits until a β value is sent back, and returns this value as the result of the operation.

3.4 Other Constructs

It is possible to implement many other concurrency constructs using the thread interface, including but not limited to:

- CSP guarded commands [23], the Ada rendezvous and select [37], and variants such as Charlesworth's multiway rendezvous [14]. (See Section B.3 for an implementation of rendezvous and select.)
- Reppy's first-class synchronous operations (*events*) [30].

[2] Since we use `refs` to implement channels, we are forced to use a "weak" type variable in the specification.

```
signature CHANNEL =
    sig
        type ’a T
        val create  : unit -> ’1a T
        val put     : ’a T -> ’a -> unit
        val get     : ’a T -> ’a
    end

functor Channel (Thread : THREAD) : CHANNEL =
    struct
        datatype ’a T = CHAN of { here  : Thread.condition,
                                  gone  : Thread.condition,
                                  value : ’a option ref }

        fun create () =
            let val m = Thread.mutex ()
            in
                CHAN { here  = Thread.condition m,
                       gone  = Thread.condition m,
                       value = ref NONE }
            end

        fun put (CHAN { here, gone, value }) v =
            let fun put’ () =
                case !value of
                    SOME _ => (Thread.wait gone;
                               put’ ())
                  | NONE   => (value := SOME v;
                               Thread.broadcast here)
            in
                Thread.with_condition gone put’
            end

        fun get (CHAN { here, gone, value }) =
            let fun get’ () =
                case !value of
                    SOME v => (value := NONE;
                               Thread.broadcast gone;
                               v)
                  | NONE   => (Thread.wait here;
                               get’ ())
            in
                Thread.with_condition here get’
            end
    end
```

Figure 5: Buffered channels

```
signature RPC =
    sig
        type ('a, 'b) T
        val create : unit -> ('1a, '1b) T
        val accept : ('2a -> 'b) -> ('2a, 'b) T -> unit
        val call   : ('a, '2b) T -> 'a -> '2b
    end

functor RPC (Channel : CHANNEL) : RPC =
    struct
        datatype ('a, 'b) T = T of ('a Channel.T * 'b Channel.T)

        fun create () = T (Channel.create (), Channel.create ())

        fun accept f (T (inChan, outChan)) =
            Channel.put outChan (f (Channel.get inChan))

        fun call (T (outChan, inChan)) value =
            (Channel.put outChan value; Channel.get inChan)
    end
```

Figure 6: Remote procedure call using channels

Of these, Reppy's events are by far the most difficult to implement, due primarily to their generality. Basically, an event is a synchronous value that may be "invoked" by a `sync` operation. Example event producing functions include `transmit` and `receive`, which synchronize with communication over channels, and `wait`, which synchronizes with the termination of a thread. The `choose` operation takes a list of events and produces a new event, representing a non-deterministic choice among the events in the list. When `sync` is applied to an event, the base events that comprise it are polled[3] to see if any are immediately satisfiable. If so, then one such base event is chosen and the corresponding synchronization (and communication, if appropriate) takes place. If no base events are immediately satisfiable, then the thread is blocked until one is satisfiable, at which point the corresponding synchronization takes place.

The similarities between events and other languages' select constructs are apparent, but there are three main differences:

1. Events are values, while select statements are, of course, just statements. This implies that no single thread "owns" an event.
2. Events may be composed at run time using `choose`, whereas the form of select statements is fixed at compile time.
3. "Output" event values are allowed in arguments to `choose`, whereas most select statements do not allow output commands in guards.

[3]When polled, each event returns one of three indications: *ready, any,* or *blocked.* The *any* status is used to indicate a satisfiable event that should only be chosen if there are no *ready* events. Events that are *blocked* will never be chosen.

Each of these differences makes implementing events quite difficult. (Reppy gives a coroutine implementation [31], but many of the difficulties arise in the presence of true parallelism.) In particular, the last problem has been a topic of quite a bit of research [8, 34, 35]. Nevertheless, we have been able to implement events using the thread interface [27].

The fact that so many higher-level constructs can be efficiently implemented in terms of the thread interface (with help from SML's first-class functions, polymorphic types, and module system) reinforces our belief that we have chosen a good set of primitives. The advantage of our low-level approach is that it can be implemented efficiently on both uniprocessors and shared-memory multiprocessors, as we will show in the remainder of this paper. It can then be used as the basis for multiple higher-level facilities, all of which will interoperate. In a large system, for example, a module that uses futures can be composed with another module that uses synchronous events and channels. The alternative—providing a high-level mechanism as the sole means of expressing concurrency—is much less attractive: it may not be natural for all applications, and implementing other paradigms efficiently in terms of it may be difficult (consider using a rendezvous simply to acquire a lock!)

4 Simulating Concurrent Threads

In the following sections, we give some details regarding two implementations of the thread interface that we have developed purely in SML/NJ. The first is a coroutine version that uses SML/NJ's first-class continuations to multiplex control. The second is also a continuation-based version, but uses UNIX signals and SML/NJ's asynchronous signal-handling facility to provide preemption. Both implementations provide the functionality needed for simulating concurrency on uniprocessors.

4.1 Using Continuations to Simulate Concurrency

A continuation of some expression is a function that takes the result of the expression and computes the "rest of the program". SML/NJ provides continuations as abstract types with the following signature [18]:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw  : 'a cont -> ('a -> 'b)
```

The `callcc` operation is used to create a `cont` that can be applied using the `throw` operation.

Wand is generally credited with showing how to simulate concurrency using first-class continuations [38]. The key idea is that continuations represent the state of a computation; since they are first-class, they can be stored in a data structure (such as a queue) and invoked at a later time.

Thus, to provide coroutines in SML/NJ, we can map a thread directly onto a continuation. When a thread must block (*e.g.,* in `await`), we can capture its continuation using `callcc`, store the continuation appropriately, and invoke it using `throw` at some later time when the thread is no longer blocked.

To provide a concrete example, Figure 7 gives a simplified implementation of the `fork` operation. The full coroutine implementation is given in Appendix A. The `running_queue` holds continuations for all threads that are not blocked (except for the currently executing thread).

```
fun fork child =
    callcc (fn parent =>
                (enqueue parent running_queue;
                 child ();
                 throw (dequeue running_queue) ()))
```

Figure 7: Implementing `fork`

```
fun alarm_handler (_, k) =
    if (in_atomic_region ()) then
         (signal_occurred := true; k)
    else
         (enqueue running_queue k;
          dequeue running_queue)
```

Figure 8: Context switching signal handler

When we `fork` a new `child`, we capture the `parent` thread's continuation, enqueue it on the `running_queue`, and invoke the `child`. When the `child` completes, we dequeue a thread from the `running_queue` and invoke its continuation.

Implementing a coroutine version of the full thread interface using SML/NJ's continuations is quite simple, as evidenced by the size of our code (154 lines including white space).

4.2 Using Signals for Preemption

There are certain advantages to coroutines: simplicity, lack of race conditions, and repeatable interleaving. However, a major disadvantage of any coroutine implementation is the lack of *preemption* among threads. Without preemption, threads may be "starved" from doing any work since a thread could run quite a long time before coming to any synchronization point. This sort of behavior is particularly undesirable in interactive programs.

Fortunately, SML/NJ provides the mechanisms needed to turn our coroutine implementation into a preemptively-scheduled uniprocessor threads package. In particular, we are provided with user-programmable *asynchronous signal handlers* [32]. We can use one such handler to catch UNIX timer signals and trigger a context switch.

An SML/NJ signal handler has the type:

```
(int * unit cont) -> unit cont
```

Different signal handlers may be installed for different signals. The first argument of the handler indicates the number of times the signal has been received before the handler was called. Signals are masked when a handler is executing, so it is possible that a signal could have been sent multiple times before the corresponding handler is executed. The second argument of the handler is the current continuation of the computation that was taking place at the time the signal was received.[4]

[4] This is not entirely accurate: SML/NJ records the signal when it is received, but lets the computation continue until it reaches a convenient point at which a continuation can be captured. See Reppy's paper [32] for a complete description.

The handler should return a continuation to be invoked as its result.

To facilitate preemptive context switching, we use the UNIX `SIGALRM` signal (set to go off at some appropriate interval such as 20 msec) and a handler similar to the one found in Figure 8. When a `SIGALRM` is received, the handler will be passed the thread that was running as a continuation `k`. A check is done to see if `k` was in some "atomic" region (*e.g.,* doing a test-and-set) when the signal was received. If so, instead of doing the context switch, the signal is recorded, and `k` is returned so it may complete its atomic operation before switching contexts. If the thread was not in an atomic region when the signal was received, `k` is enqueued on the `running_queue`. Then, another thread is dequeued and returned as the result continuation to be invoked.

5 A Multiprocessor Implementation of Threads

The continuation-based implementations of the thread interface are simple and portable, but they have two disadvantages. The first is that calls to the operating system to perform a service such as I/O will block all threads until the service is complete. It is possible for certain operations to do a non-blocking system call (such as UNIX's `select`) before performing the blocking operation, but other common operations (such as a page fault) have no such "hooks". The second disadvantage is that there is no provision in SML/NJ to specify that computation should actually take place concurrently. Consequently, we cannot take advantage of parallelism on multiprocessor architectures.

To address these two disadvantages, we have modified the SML/NJ system to support a multi-processor implementation of the thread interface. In the following sections, we give a high-level description of these modifications and some details regarding the threads implementation built on top of the system.

5.1 Mach

The Mach operating system "provides a set of low-level, language-independent primitives for manipulating threads of control" [1]. Mach also provides novel memory management facilities and inter-process communication. Combined with the UNIX BSD server, which allows BSD binaries to be run on top of Mach [20], these facilities provide an attractive operating system platform on which to build shared-memory, parallel programming languages. Consequently, we have chosen Mach as the foundation for our multiprocessor SML system, SML/Mach.

Since Mach is an ongoing research project at Carnegie Mellon, and SML/NJ is an ongoing research project at AT&T Bell Laboratories and Princeton (among others), we chose as one of our goals to have "minimal impact" on the SML/NJ system. In particular, we decided to concentrate our work on modifying the runtime system of SML/NJ and avoid touching the compiler. As a result, we expect to be able to keep up with and take advantage of new developments in both SML/NJ and Mach. Furthermore, it should be possible to integrate, with minimal modifications, other compiler-oriented research such as the SML to C compiler [36].

5.2 The SML/NJ Runtime

The SML/NJ runtime system is described elsewhere [5], but we will point out some of the highlights that are relevant to this paper.

The runtime system for SML/NJ is written in C and provides a coroutine interface to ML code. When ML requires a runtime service (*e.g.,* I/O), it sets a global variable `request` to a value indicating which service is desired, saves its register set in a global state vector, loads the C registers from the C stack[5] and begins to execute the C code that will provide the service. When the runtime service is complete, the C registers are saved on the C stack, the ML registers re-loaded, and execution of ML code continues. Two assembly language routines, `saveregs` and `restoreregs`, handle the machine-specific task of crossing this ML/C boundary.

One of the most important services provided by the runtime system is garbage collection. SML/NJ uses a simple (but efficient) two-generation, copying collector [4]. Allocation is inlined by the compiler, making it quite fast. At the entrance of each *code tree,*[6] a check is made to see if enough heap space exists for the maximum amount of allocation that the code tree might do. If there is not enough space, a trap instruction is used to initiate garbage collection. The runtime system catches the exception caused by the GC-trap and performs the following steps [32].

1. The runtime routine `ghandle` catches the trap and records the program counter of the trap location in the state vector, sets `request` to `REQ_GC` and returns control to the assembly code routine `saveregs`.
2. `Saveregs` saves the ML state in the state vector and passes control to `run_ml`.
3. The garbage collector is then run, using the state vector as the root set.
4. After garbage collection, `run_ml` calls `restoreregs`, which loads the machine registers from the state vector, and returns to the trap location.

5.3 Support for Thread Creation

Given the organization of the SML/NJ runtime, the primary obstacle to providing support for multiple Mach threads running ML code is the ML/C boundary. One of two approaches could be taken: either have each thread run ML code only and treat the C runtime as a server, or allow each Mach thread to execute both ML and C runtime code. Obviously, the latter approach is more attractive, since no synchronization must take place. Consequently, our implementation takes this approach.

To allow each Mach thread to execute both ML and C runtime code, it is necessary that each thread have its own state vector as well as its own C runtime stack. In fact, each thread must have its own copy of the `request` variable and many other variables that are unique, “global” variables in the SML/NJ system.

To provide this functionality, we divide the UNIX stack segment for the entire process into sub-stacks, one for each thread. The sub-stacks are aligned in such a way that by masking a thread’s stack pointer appropriately, we can determine the base of the thread’s sub-stack. A special routine, `thread_self`, does this masking.

[5]Contrary to some published claims [5, 32], SML/NJ does use a runtime stack. However, the stack is only used by the C code that comprises the runtime system. The ML code does all of its allocation (including closures) in the ML heap and does not use the stack.

[6]According to Reppy [32], “a *code tree* or *extended basic block* is an acyclic set of blocks with one entry point and one or more exits.”

A thread's state vector and "global" variables are stored at the base of its sub-stack.[7] Since the ML code does not use the stack (and fortunately does not use the stack pointer register), `thread_self` can be called at any time to gain access to the per-thread information. Therefore, routines such as `saveregs` and `restoreregs` were modified to use `thread_self` to locate their calling thread's state vector.

Thus, to create a Mach thread to execute some ML code, we take the following steps:

1. Obtain a stack for the new thread.
2. Obtain a portion of the allocation area for the thread. (This is explained in Section 5.4.)
3. Place an initialized thread-state vector at the base of the stack. The vector will contain the address of the ML code to execute, the continuation, the closure, etc. It will also have `request` set to `REQ_RETURN`.
4. Make a `thread_create` call to Mach.
5. Make a call to a machine-specific routine, `MLthread_setup`, which sets up the thread to execute the C routine `run_ml`. This is done by calling the Mach `thread_set_state` routine to initialize the program counter and other registers.
6. Call the Mach `thread_resume` routine. At this point, the Mach kernel will schedule the thread to run.

When the new thread starts running, it will call the `run_ml` routine which will check `request`, see that it is set to `REQ_RETURN`, and "return" to the appropriate ML code. It does so by making a call to the modified `restoreregs` as explained in the previous section.

5.4 Heap Management

Once a Mach thread is running, it needs to be able to allocate memory from the ML heap. There are basically two approaches we can take towards allocation: have each thread share the allocation area and acquire a lock on the current heap-limit pointer before doing an allocation, or divide the allocation area among the threads.

There are two main reasons why the latter approach is more attractive. First, the extra overhead of acquiring a lock and updating a shared limit pointer would be unacceptable, since allocations are quite frequent in ML. (The SML/NJ compiler dedicates a register to hold the limit pointer; on a typical RISC machine, changing to a shared limit pointer would add 4 memory operations to the single register operation currently required.) Second, since SML/NJ generates inlined allocation code, we would have to modify the compiler to support the former approach.

Consequently, we give each thread a separate portion of the allocation area. Each thread has its own heap-limit pointer, allocation is still inlined, no synchronization is necessary for allocation, and no changes are needed in the compiler. In addition, each thread's heap can have pointers into other threads' heaps, so the partitioning is invisible to the SML programmer.

Now that a Mach thread can allocate memory, we need to be able to garbage collect (GC) it. There are many approaches we can take towards GC in a multi-threaded system, but our self-imposed constraint of modifying only the runtime system limits our options. As a first cut, we have

[7]This is basically the same approach taken by the Mach C Threads package [15].

Master Code

1. The thread that caused the GC trap enters the trap handler and acquires a lock on a `gc_master` variable.

2. If `gc_master` is already set, the thread executes the **Slave** code. Otherwise, the thread designates itself as the master by setting `gc_master` appropriately.

3. The master stops all threads that are running and sets their heap-limit pointers so that they will each call the GC handler upon entrance to their next code tree.

4. The master releases the `gc_master` lock and waits until all other threads that were running have entered the GC handler.

5. The master gathers all of the threads' roots and calls the garbage collector.

6. When the garbage collector returns, the master re-divides the heap among the threads, acquires the `gc_master` lock, gives each thread its roots back, and allows them to continue.

7. The master clears `gc_master` and releases the lock. It then continues with its ML code.

Slave Code

1. The slave releases the lock on `gc_master`.

2. The slave tells the master that it is ready for the GC and passes its roots to the master.

3. The slave waits for the master to signal that the GC is done, at which point it receives its new roots.

4. The slave continues with its ML code.

Figure 9: Synchronization for GC

chosen to stop all threads when a single thread exhausts its allocation area. When the threads have synchronized, we gather their roots, and call the same copying collector that is used in the SML/NJ system.

Recall from Section 5.2 that when a code tree is entered, a heap limit check is done and a GC trap occurs if there is not enough space. We use this facility to synchronize our threads. The details of the synchronization appear in Figure 9. Special care must be taken to ensure that deadlock does not occur, especially for threads that are blocked at the time of the GC.

5.5 Support for Locking and Synchronization

To support `mutex` locks, we have added two assembly language routines to the runtime system, `try_acquire` and `release`, which operate on ML `int ref` values. These routines are machine dependent, but essentially translate into atomic "test-and-set" and "clear" operations.[8]

To support `condition` variables, we have added three routines to the runtime system. The

[8] This is actually quite difficult to implement on MIPS machines that do not provide an atomic test-and-set instruction. We are indebted to Alessandro Forin of the Mach project for providing a solution for uniprocessors [19].

first, thread_wait_port, returns a representation of a Mach port that the thread can use to block itself. The second, thread_wait, is used by a thread when it wishes to block. This is accomplished by performing a Mach msg_receive on its wait_port. The third routine, thread_signal, takes a representation of a Mach port and signals the corresponding thread. This is accomplished by doing a Mach msg_send on the port.

These routines, together with the modifications mentioned in the two previous sections, are all the support that was needed to implement the thread interface for Mach multiprocessors. Currently, SML/Mach runs on VAX, MIPS, SPARC, and 680x0 based machines.

5.6 Virtualizing Mach Threads

In our current implementation, SML/Mach provides only a *fixed* number of Mach threads for use by the programmer. The number of Mach threads is determined when the SML/Mach system begins execution. Our intent was to map one Mach thread (or any fixed number) onto each processor, and use continuations to multiplex ML threads on individual processors. In the degenerate case of a uniprocessor, this should be equivalent to our non-Mach coroutine implementation.

There are several reasons why we undertook this approach. The first reason is that it leads to fewer modifications to the runtime system. For instance, instead of allocating a runtime stack for each thread, we can divide the single UNIX stack segment into n pieces, where n is the number of Mach threads.

The second reason is that it is quite simple to "virtualize" the fixed number of Mach threads using continuations. Essentially, each continuation-based thread is placed in a shared run-queue. Each Mach thread executes an infinite loop as follows:

1. Block until a continuation is available and atomically dequeue it.
2. Invoke the continuation. If an exception is raised, catch it, and print it to the terminal.
3. Go to step 1.

Note that if a continuation-thread becomes blocked on a mutex or condition, and that mutex or condition is later garbage collected, the continuation and hence the logical thread will also be garbage collected. Since the thread could never have been awakened, this is precisely what should happen.

The third reason we decided to virtualize Mach threads using continuations is that we claim context switching of SML/NJ continuation threads is cheaper than context switching of Mach threads. To substantiate this claim, we measured the wall clock time of two threads doing context switches on a DECstation 3100. Essentially the same benchmark was run for both C Threads running under Mach 2.5 and SML coroutine threads running on the SML/NJ version 0.65 system. The results of this benchmark are summarized in Figure 10. It is apparent that context switching for SML/NJ continuations takes less time than context switching for Mach threads.

The primary reason that Mach threads are more expensive than continuations is the need to go through the kernel to do an operation such as thread_switch. Another reason why continuations are cheaper than Mach threads is that the SML/NJ system uses *continuation-passing style* code generation and allocates all closures on the heap [3]. Thus, continuation creation consists of merely allocating and initializing a closure. No migration from a stack to the heap needs to take place.

C Mach Threads	86 μsec
SML Coroutine Threads	39 μsec

Figure 10: Context Switch Times on DECstation 3100

System	*Lock/Unlock*	*Signal*	*Handoff*
C Coroutines	2.0 μsec	0.7 μsec	49.2 μsec
C/Mach	9.4 μsec	0.7 μsec	215.4 μsec
SML Coroutines	14.4 μsec	10.9 μsec	307.9 μsec
SML Coroutines-2	10.9 μsec	5.7 μsec	270.4 μsec
SML Preemptive	64.6 μsec	43.1 μsec	625.0 μsec
SML/Mach	107.3 μsec	56.8 μsec	1379.2 μsec

Figure 11: Comparison of C and SML Threads on DECstation 3100

6 Future Work

There are many other issues that we would like to address concerning the addition of threads to Standard ML. In the following two sections, we present some of these issues and discuss possible approaches.

6.1 Optimizations

As stated before, one of our goals was to have minimal impact on the SML/NJ system. However, this has kept us from pursuing paths that could lead to better performance. Figure 11 gives some comparison between C Mach threads, C coroutine threads, and the different implementations of SML threads. All benchmarks were performed on a 20 Mbyte DECstation 3100 running Mach 2.5 and SML/NJ version 0.65. The *Lock/Unlock* column gives the time needed to do a (successful) `try_acquire` followed by a release. The *Signal* column gives the time needed to do `signal` on a condition that has no threads waiting on it. The *Handoff* column gives the time needed for two threads to `acquire`, `await`, update a shared flag, `signal` the other thread, and `release`. The SML Coroutines-2 implementation is essentially the SML Coroutines implementation with queue operations inlined by hand. The SML Preemptive implementation was run with a 10 msec interval between context switches. While not quite a fair comparison[9], it is still apparent that we have quite a way to go before we have a production quality threads package.

Many of the inefficiencies in the SML threads packages are by-products of SML/NJ's treatment of `ref` cells, which is itself a by-product of the generational copying collection scheme employed.

[9]For instance, the SML times include the cost (and benefits) of garbage collection.

Essentially, reading a mutable object can cost twice as much in SML/NJ as C (due to indirection), while updating a mutable object can cost up to 5 times as much (due to the need to log the update.)

The main inefficiency for SML/Mach, however, is that a call to one of the thread routines (*e.g.,* `fork`, `acquire`, `release`, etc.) will cross the ML/C boundary, do the operation, and then re-cross the ML/C boundary. Each crossing of the ML/C boundary constitutes a context switch in and of itself. By adding new primitive operations (*e.g.,* `try_acquire`, `release`, etc.) to the SML/NJ *compiler,* we can eliminate the need to cross this boundary for most of the routines. Furthermore, this will allow the simple routines to be inlined by the compiler.

While providing a fixed number of Mach threads has eased our implementation, we really should provide a facility for forking any number of Mach threads dynamically. To do this, we would have to modify stack allocation and the assignment of allocation areas to threads. The former problem seems quite easy to handle given Mach's virtual memory capabilities, and in fact seems to work directly on most architectures. However, on some architectures (notably the Sun-3), this fragments the address space and requires a rewrite of the routines that export an executable image file.

We feel that a change should be made in the assignment of allocation areas to threads, regardless of whether we choose to allow dynamic Mach thread creation. We plan to modify SML/Mach so that each thread is given a small, fixed-size chunk of allocation area. When a thread exhausts a chunk, a check will be made if more chunks are available. If not, then we will have to delay the thread until a GC occurs. If a chunk is available, then we can give it to the thread and let it continue.

This strategy would have the advantage that a GC could be delayed until every chunk was used. The current system does a GC whenever one thread exhausts its allocation area. We also feel that the chunk strategy would provide a better interface for different GC algorithms. However, this strategy has the disadvantage that intra-chunk fragmentation can be quite high. Tuning the size of the chunks will be quite important.

We are interested in exploring concurrent, incremental garbage collection as proposed by Appel, Ellis and Li [6]. The Mach virtual memory primitives have been shown to provide the functionality necessary to implement such a collector [16, 29]. An incremental collector will be necessary for developing interactive and real-time programs in SML.

Finally, more *interaction* is desirable between Mach and the SML/Mach runtime system concerning the availability of processors and physical memory. For instance, if the operating system knew that it was going to take away a processor, it could inform the SML/Mach runtime, as suggested by Anderson *et al.* [2]. The runtime could then decide to adjust the number of Mach threads to keep the proper thread to processor ratio. As another example, we already use *mutex handoffs*[10] to facilitate context switching [11]. This *user-level scheduling* approach is an ongoing research topic being explored by the Mach project and we hope to take advantage of any "hooks" they expose.

6.2 Shortcomings

One potential shortcoming of the thread interface is the inability to communicate asynchronously with a thread. To understand why such a facility is desirable, consider the following scenario: A thread is forked in an interactive system to invert a 100×100 matrix. During the middle of the inversion, the user decides that he or she does not need the inverted matrix and presses the "cancel" button. However, unless the matrix inversion routine was written in such a way that it periodically

[10] If a thread is about to block because it failed to acquire a lock, it tells Mach to deschedule it and gives a "hint" that the owner of the lock should be run. Thus, the processor is *handed off* from one thread to another.

checks whether or not it has been canceled, the computation will continue.

To address this shortcoming, we could add an *alert* mechanism to the interface, as in Modula-2+ [9]. Perhaps the best way to do so is to provide `Alert` as a signal producing function and use Reppy's signal handlers to catch the alert. This would require that a handle for a thread be returned so the user could indicate which thread was to be alerted. It would also require that signal handlers be definable on a per-thread basis.

Unfortunately, the whole notion of UNIX signals in multi-threaded programs is ill-defined. The problem is that certain signals should be "broadcast" to all threads, while some signals should only be sent to a single thread. For instance, it would seem desirable in an interactive system for `SIGQUIT` (quit) to be broadcast to all threads. However, the `SIGFPE` (floating point error) signal should only be sent to the thread that caused the error.

Mach defines a general *exception* mechanism that can be used to emulate UNIX signals and provides a precise semantics even in the presence of concurrency [12]. Presently, we use this mechanism to catch a GC trap and plan to extend our use to handle the entire range of exceptions that Mach provides.[11]

A rather serious shortcoming of the thread interface is the lack of integration with the Definition of SML. In fact, it is quite interesting that our rather modest changes to SML/NJ's runtime have completely invalidated the Definition. It would seem desirable to model the dynamic semantics of SML/Mach using some concurrency model such as CCS or CSP. This is certainly a rich topic for researchers to pursue and we hope that SML/Mach will provide a concrete reference point.

7 Summary and Conclusions

We have presented an interface for multiple threads of control for Standard ML, including provisions for locking, synchronization, and per-thread state. One of the primary advantages of the interface is that the concepts introduced are similar to those in threads packages that have been demonstrated to be of practical use.

One might argue that using the store to communicate is contrary to the philosophy of SML, since the language is "mostly functional". However, we have shown how the interface can be used to build higher-level concurrency constructs and mechanisms such as Multilisp-style futures and synchronous message passing. Furthermore, we argue that building these abstractions in terms of threads provides efficient implementations of these mechanisms for uniprocessors and shared-memory multiprocessors.

However, having worked with the thread interface extensively, we have found that it is best to structure one's program around these higher-level constructs, only "dipping down" into the thread interface when efficiency is called for.

We have presented three implementations of the interface. All of the implementations make use of SML/NJ's first-class continuations and provide a concrete justification for their inclusion in the language. Our multiprocessor Mach-based implementation was built with the goal of having "minimal impact" on the NJ system. As a result, we expect to keep up with the rapid developments of both SML/NJ and Mach. We have pointed out some of the problems with our approach, as well as possible solutions.

[11] Using the native Mach exception handling facilities instead of the full emulation of UNIX signals should also provide better performance.

8 Acknowledgments

We would like to thank Bob Harper, Peter Lee, and Jeannette Wing for their help in designing the threads interface. We would also like to thank Anurag Acharya, Andrew Appel, Andrzej Filinski, Bob Harper, and David Tarditi with their help concerning SML and in particular, SML/NJ. Finally, thanks to Brian Bershad, David Black, David Detlefs, Alessandro Forin, and Mary Thompson for their help concerning Mach.

References

[1] Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young.
Mach: A new kernel foundation for UNIX development.
In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy.
Scheduler activations: Effective kernel support for the user-level management of parallelism.
Technical Report 90-04-02, Department of Computer Science and Engineering, University of Washington, April 1990.
Revised October 1990.

[3] Andrew W. Appel.
Continuation-passing, closure-passing style.
In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 293–302, January 1989.

[4] Andrew W. Appel.
Simple generational garbage collection and fast allocation.
Software—Practice & Experience, 19(2):171–183, February 1989.

[5] Andrew W. Appel.
A runtime system.
Journal of Lisp and Symbolic Computation, 3(4):343–380, November 1990.

[6] Andrew W. Appel, John R. Ellis, and Kai Li.
Real-time concurrent collection on stock multiprocessors.
In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.
Also published as *SIGPLAN Notices*, 23(7).

[7] Andrew W. Appel and David B. MacQueen.
A Standard ML compiler.
In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.
Volume 274 of *Lecture Notes in Computer Science*.

[8] A. J. Bernstein.
Output guards and nondeterminism in communicating sequential processes.
ACM Transactions on Programming Languages and Systems, 2(2):234–238, April 1980.

[9] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin.
Synchronization primitives for a multiprocessor: A formal specification.
In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 94–102, November 1987.
Published as *Operating Systems Review,* 21(5).

[10] Andrew D. Birrell.
An introduction to programming with threads.
Research Report 35, DEC Systems Research Center, January 1989.

[11] David L. Black.
Scheduling support for concurrency and parallelism in the Mach operating system.
Technical Report CMU-CS-90-125, School of Computer Science, Carnegie Mellon University, 1990.

[12] David L. Black, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young.
The Mach exception handling facility.
Technical Report CMU-CS-88-129, Computer Science Department, Carnegie Mellon University, April 1988.

[13] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson.
Modula-3 report (revised).
Research Report 52, DEC Systems Research Center, November 1989.

[14] A. Charlesworth.
The multiway rendezvous.
ACM Transactions on Programming Languages and Systems, 9(3):250–267, July 1987.

[15] Eric C. Cooper and Richard P. Draves.
C Threads.
Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[16] David L. Detlefs.
Concurrent garbage collection for C++.
Technical Report CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, May 1990.

[17] Digital Equipment Corporation.
Concert Multithread Architecture, March 1989.

[18] Bruce F. Duba, Robert Harper, and David B. MacQueen.
Typing first-class continuations in ML.
In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, 1991.
To appear.

[19] Alessandro Forin.
Test-and-set instructions for MIPS.
School of Computer Science, Carnegie Mellon University, Mach Operating System source file `mips/lock.s`.

[20] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid.
Unix as an application program.
In *Proceedings of the USENIX Summer Conference*, pages 87–95, Anaheim, CA, June 1990. USENIX.

[21] Robert H. Halstead, Jr.
Multilisp: A language for concurrent symbolic computation.
ACM Transactions on Programming Languages and Systems, 7(4):501–538, October 1985.

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

[23] C. A. R. Hoare.
Communicating sequential processes.
Communications of the ACM, 21(8):666–677, August 1978.

[24] IEEE Computer Society.
Threads Extension for Portable Operating Systems.
Technical Committee on Operating Systems, December 1990.
Draft P1003.4a/D5 for work item JTC1.22.21.2.

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

[26] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, 1990.

[27] J. G. Morrisett.
Implementing events in ML+Threads.
Venari Note 5, School of Computer Science, Carnegie Mellon University.

[28] Norman Ramsey.
Concurrent programming in ML.
Technical Report CS-TR-262-90, Department of Computer Science, Princeton University, April 1990.

[29] Richard F. Rashid, Avadis Tevanian, Jr., Michael W. Young, David B. Golub, Robert V. Baron, David L. Black, William Bolosky, and Jonathan J. Chew.
Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures.
IEEE Transactions on Computers, C-37(8):896–908, August 1988.

[30] J. H. Reppy.
Synchronous operations as first-class values.
In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 250–259, June 1988.
Also published as *SIGPLAN Notices*, 23(7).

[31] J. H. Reppy.
First-class synchronous operations in Standard ML.
Technical Report TR 89-1068, Department of Computer Science, Cornell University, December 1989.

[32] J. H. Reppy.
Asynchronous signals in Standard ML.
Technical Report TR 90-1144, Department of Computer Science, Cornell University, August 1990.

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

[34] F. B. Schneider.
Synchronization in distributed programs.
ACM Transactions on Programming Languages and Systems, 4(2):125–148, April 1982.

[35] A. Silberschatz.
Communication and synchronization in distributed systems.
IEEE Transactions on Software Engineering, 5(6):542–546, November 1979.

[36] David Tarditi, Anurag Acharya, and Peter Lee.
No assembly required: Compiling Standard ML to C.
Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, November 1990.

[37] United States Department of Defense.
Reference Manual for the Ada Programming Language, February 1983.
U.S. Government Printing Office, ANSI/MIL-STD-1815A-1983.

[38] Mitchell Wand.
Continuation-based multiprocessing.
In *Proceedings of the 1980 LISP Conference*, pages 19–28, 1980.

A Coroutine Implementation of SML Threads

This section presents a coroutine-based implementation of the thread interface given in Figure 2, using continuations in Standard ML of New Jersey. The `Thread` functor is parameterized by a polymorphic queue module with the following signature:

```
signature QUEUE =
    sig
        type 'a T
        exception Deq
        val create : unit -> '1a T
        val enq : 'a T -> 'a -> unit
        val deq : 'a T -> 'a
        val len : 'a T -> int
        val contents : 'a T -> 'a list
    end
```

The `Queue` structure is used for the run queue, mutex queues, and condition queues.

```
functor CoThread (Queue : QUEUE) : THREAD =
    struct
        (********************)
        (* Per-thread state. *)
        (********************)

        type env = unit ref

        datatype 'a var = VAR of (env * 'a) list ref

        exception Undefined

        fun new_env () = ref ()

        val current_env = ref (new_env ())

        fun var () = VAR (ref [])

        fun find _ [] = raise Undefined
          | find env ((e, a) :: rest) =
            if e = env then a else find env rest

        fun get (VAR v) = find (!current_env) (!v)

        fun replace env [] a = [(env, a)]
          | replace env ((pair as (e, _)) :: rest) a =
            if e = env then (e, a) :: rest
            else pair :: replace env rest a
```

```
fun set (VAR v) a = (v := replace (!current_env) (!v) a)

(*********************************************)
(* Thread creation, destruction, and scheduling. *)
(*********************************************)

datatype thread = THREAD of unit cont * env

fun thread k = THREAD (k, !current_env)

val run_queue : thread Queue.T = Queue.create ()

fun reschedule thread = Queue.enq run_queue thread

exception Deadlock

fun run_next () =
    let val THREAD (k, env) = Queue.deq run_queue
        handle Queue.Deq => raise Deadlock
    in
        current_env := env;
        throw k ()
    end

fun exit () = run_next ()

fun block queue =
    callcc (fn k => (Queue.enq queue (thread k);
                     run_next ()))

fun yield () = block run_queue

fun fork f =
    callcc (fn k =>
            (reschedule (thread k);
             current_env := new_env ();
             f () handle exn =>
                 print ("Unhandled exception " ^
                        System.exn_name exn ^
                        " raised in thread.\n");
             run_next ()))

(***************)
(* Mutex locks. *)
(***************)
```

```
datatype mutex =
    MUTEX of bool ref * thread Queue.T

fun mutex () =
    MUTEX (ref false, Queue.create ())

fun try_acquire (MUTEX (held, _)) =
    if not (!held) then
        (held := true; true)
    else
        false

fun acquire (m as MUTEX (held, q)) =
    if try_acquire m then
        ()
    else
        block q

fun release (MUTEX (held, q)) =
    reschedule (Queue.deq q)
    handle Queue.Deq => held := false

fun with_mutex m body =
    let val result = (acquire m;
                      body () handle exn =>
                          (release m; raise exn))
    in
        release m;
        result
    end

(***************)
(* Conditions. *)
(***************)

datatype condition =
    CONDITION of mutex * thread Queue.T

fun condition m =
    CONDITION (m, Queue.create ())

fun mutex_of (CONDITION (m, _)) = m

val with_condition = with_mutex o mutex_of

fun awaken (CONDITION (m as MUTEX (_, mq), cq)) =
    let val thread = Queue.deq cq
    in
```

```
            if try_acquire m then
                reschedule thread
            else
                Queue.enq mq thread
        end

    fun repeat f = (f (); repeat f)

    fun signal c =
        awaken c handle Queue.Deq => ()

    fun broadcast c =
        repeat (fn () => awaken c) handle Queue.Deq => ()

    fun wait (CONDITION (m, q)) =
        (release m; block q)

    fun await c test =
        if test () then
            ()
        else
            (wait c; await c test)
end
```

B Further Examples

The following sections give further examples that make use of the thread interface.

B.1 Reader/Writer Locks

```
signature RW_LOCK =
    sig
        type T
        val create          : unit -> T
        val read_with_lock  : T -> (unit -> 'a) -> 'a
        val write_with_lock : T -> (unit -> 'a) -> 'a
    end
```

Reader/writer locks allow multiple readers or a single writer access to some state. The following functor implements reader/writer locks.

```
functor RW_Lock (Thread : THREAD) : RW_LOCK =
    struct
        datatype T = RW of { free       : Thread.condition,
                             num_readers: int ref,
                             write      : bool ref }

        fun create () = RW { free        = Thread.condition (Thread.mutex ()),
                             num_readers = ref 0,
                             write       = ref false }

        fun rw_lock_read (RW { free, num_readers, write }) =
            Thread.with_condition free
            (fn () =>
             (Thread.await free (fn () => not (!write));
              inc num_readers))

        fun rw_unlock_read (RW { free, num_readers, ... }) =
            Thread.with_condition free
            (fn () => (dec num_readers;
                       if !num_readers = 0 then
                           Thread.broadcast free
                       else
                           ()))

        fun rw_lock_write (RW { free, num_readers, write }) =
            Thread.with_condition free
            (fn () => (Thread.await free
                        (fn () => !num_readers = 0 andalso not (!write));
                       write := true))
```

```
    fun rw_unlock_write (RW { free, num_readers, write }) =
         Thread.with_condition free
         (fn () => (write := false;
                    if !num_readers = 0 then
                        Thread.broadcast free
                    else
                        ()))

    fun with_lock lock_fn unlock_fn lock f =
        let val result = (lock_fn lock;
                          f () handle exn =>
                              (unlock_fn lock; raise exn))
        in
            unlock_fn lock;
            result
        end

    val read_with_lock =
        with_lock rw_lock_read rw_unlock_read

    val write_with_lock =
        with_lock rw_lock_write rw_unlock_write
end
```

B.2 Asynchronous Channels

In order to implement an approximation of Ada's rendezvous and select constructs, we first define a `CHANNEL` signature that allows asynchronous sends and asynchronous receives.

```
signature ASYNC_CHANNEL =
    sig
        include CHANNEL

        exception GetNow
        val get_now  : 'a T -> 'a
        val get_wait : 'a T -> unit
    end
```

The `put` operation puts an object into a channel and returns immediately. The `getWait` operation waits until some object has been put into the channel. The `getNow` operation gets a value from a channel if a value is available, otherwise the exception `GetNow` is raised. A functor that implements this channel interface is given below:

```
functor AsyncChannel (Thread : THREAD) : ASYNC_CHANNEL =
    struct
        datatype 'a T = CHAN of { here  : Thread.condition,
                                  gone  : Thread.condition,
                                  value : 'a option ref }

        fun create () =
            let val m = Thread.mutex ()
            in
                CHAN { here  = Thread.condition m,
                       gone  = Thread.condition m,
                       value = ref NONE }
            end

        fun put (CHAN { here, gone, value }) v =
            let fun put' () =
                case !value of
                    SOME _ => (Thread.wait gone;
                               put' ())
                  | NONE   => (value := SOME v;
                               Thread.broadcast here)
            in
                Thread.with_condition gone put'
            end

        fun get (CHAN { here, gone, value }) =
            let fun get' () =
                case !value of
```

```
                SOME v => (value := NONE;
                           Thread.broadcast gone;
                           v)
              | NONE   => (Thread.wait here;
                           get' ())
        in
            Thread.with_condition here get'
        end

    fun get_wait (CHAN { here, gone, value }) =
        let fun get_wait' () =
            case !value of
                SOME _ => ()
              | NONE => (Thread.wait here;
                         get_wait' ())
        in
            Thread.with_condition here get_wait'
        end

    exception GetNow

    fun get_now (CHAN { here, gone, value }) =
        let fun get_now' () =
            case !value of
                SOME v  => (value := NONE;
                            Thread.signal gone;
                            v)
              | NONE => raise GetNow
        in
            Thread.with_condition gone get_now'
        end
end
```

B.3 Rendezvous and Select

The following signature defines a first-class `rendezvous` type along with a first-class `select` type. The former is analogous to an Ada entry, while the latter is analogous to an Ada select.

```
signature RENDEZVOUS =
  sig
    type ('a, 'b) rendezvous
    val rendezvous : ('1a -> '1b) -> ('1a, '1b) rendezvous

    type ('a, 'b) arm
    val arm : ((unit -> bool) * ('a, 'b) rendezvous) -> ('a, 'b) arm

    type ('a, 'b) select
    val select : (('a, 'b) arm) list -> ('a, 'b) select

    val accept : ('a, 'b) select -> unit
    val call : ('a, '2b) rendezvous -> 'a -> '2b
  end
```

The `Rendezvous` functor below implements the `RENDEZVOUS` signature. Each rendezvous value consists of a function and an input and output channel. Each select value is a list of test functions (guards) and corresponding rendezvous values.

The `call` operation sends a value on the input channel of the `rendezvous` to some receiver. The receiver will use the value to compute some new value and return the result on the output channel.

The `accept` forks a thread to handle each "arm" of the select. Each thread evaluates its guard to determine if it should attempt to do communication. If so, it waits until a value has arrived from a caller on the input channel. It then looks to see if any other thread has already finished its rendezvous. If so, the thread quietly dies (aborts). Otherwise, the thread grabs the value from the input channel, uses its rendezvous function to compute the output value, and places the output value in the output channel. It then sets a shared flag so that other threads associated with the `accept` will abort.

```
functor Rendezvous (structure Thread : THREAD
                    structure Channel : ASYNC_CHANNEL) : RENDEZVOUS =
  struct
    structure Thread = Thread

    datatype ('a, 'b) rendezvous =
        RV of (('a -> 'b) * ('a Channel.T) * ('b Channel.T))

    fun rendezvous f = RV (f, Channel.create (), Channel.create ())

    datatype ('a, 'b) arm = ARM of ((unit -> bool) * ('a,'b) rendezvous)

    val arm = ARM
```

```
  datatype ('a, 'b) select = SEL of ('a, 'b) arm list

  val select = SEL

  fun call (RV (_, outChan, inChan)) v =
      (Channel.put outChan v;
       Channel.get inChan)

  exception EmptySelect

  fun threadMap f = map (fn x => Thread.fork (fn () => (f x)))

  fun accept (SEL []) = raise EmptySelect
    | accept (SEL l) =
      let val selectUnfinished = ref true
          val selectFinished = Thread.condition (Thread.mutex ())
          fun helper (ARM (test, (r as RV (f, inChan, outChan)))) =
              let fun deposit () =
                  if !selectUnfinished then
                      (Channel.put outChan (f (Channel.getNow inChan));
                       selectUnfinished := false;
                       Thread.signal selectFinished)
                  else ()
              in
                  if test () then
                      (Channel.getWait inChan;
                       Thread.with_condition selectFinished deposit
                       handle Channel.GetNow =>
                           (helper (ARM ((fn () => true), r))))
                  else ()
              end
      in
          Thread.with_condition selectFinished
          (fn () => (threadMap helper l;
                     Thread.wait selectFinished))
      end
end
```