Asynchronous Signals in Standard ML[*]

John H. Reppy

TR 90-1144
August 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853

[*] This work was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862

Asynchronous Signals in Standard ML[*]

John H. Reppy
Cornell University
jhr@cs.cornell.edu

August 1, 1990

Abstract

We describe the design, implementation and use of a mechanism for handling asynchronous signals, such as user interrupts, in the New Jersey implementation of Standard ML. Providing this kind of mechanism is a necessary requirement for the development of real-world application programs. Our mechanism uses first-class continuations to represent the execution state at the time at which a signal occurs. It has been used to support pre-emptive scheduling in concurrency packages and for forcing break-points in debuggers, as well as for handling user interrupts in the **SML/NJ** interactive environment.

1 Introduction

Programs normally receive communication from the outside world via input operations. This method of communication is inherently synchronous: there is no way for the outside world to force the program to accept communication. But sometimes it is necessary to communicate asynchronously; for example, if the user wants to interrupt execution, or if the operating system needs to inform a program that its terminal connection has been lost. Most operating systems provide a mechanism for asynchronously *signaling* a program in these situations. For example, on UNIX systems, when a user types the break character (e.g., control-C), the terminal driver sends a `SIGINT` signal to the process attached to the terminal. Under UNIX a program can establish a *handler* that the operating system will call when a given signal occurs. The signal handler is, in effect, a limited co-routine of the main program. Most programs use the default signal handlers, but some applications require specialized handlers. For example, an editor will save the edited state of a file when the terminal connection is lost (signified by `SIGHUP` on UNIX). Providing a signal handling mechanism is a necessary requirement for implementing programs such as editors.

The **SML** definition ([MTH90]) includes a weak mechanism for handling asynchronous interrupts generated from the keyboard (e.g., `SIGINT` on UNIX systems). When the user types the break character, the exception `Interrupt` is raised. This exception is primarily used to force control back up to the top-level read-eval-print loop, and it has a number of problems:

[*]This work was supported, in part, by the NSF and ONR under NSF grant CCR-85-14862

- *Lack of robustness:* it is not possible to write a robust `Interrupt` handler. Multiple `SIGINT`s can create race conditions in an exception handler that handles `Interrupt`, and, in addition, there is no mechanism for masking interrupts. The race conditions are a direct result of using exceptions, which are otherwise a synchronous control-flow mechanism, to support asynchronous control-flow.
- *Lack of generality:* there is no support for other kinds of signals, such as hang-ups or interval timer interrupts. Real-world applications must be able to deal with these situations.
- *Lack of flexibility:* there is no way to reliably resume execution at the point where the signal occurred. A general mechanism should provide a way to restart after a signal.

We have designed and implemented a general-purpose signal mechanism for the New Jersey implementation of **SML**that addresses these concerns.

Standard ML of NJ (**SML/NJ**) is a publicly available implementation of **SML** developed by Dave MacQueen and Andrew Appel, with contributions from a number of other people[AM87]. It runs on a variety of UNIX-based systems and has a highly portable run-time system[Appel90]. **SML/NJ** *supports first-class* continuations as an extension[DHM]. Our mechanism treats asynchronous signals as continuation producing operations. A signal handler is a function from continuations to continuations: it takes take the current continuation at the time of the signal and returns a, possibly different, continuation. Because of the problems with asynchronous exceptions, we have chosen to replace the use of the exception `Interrupt` by our mechanism. This means that exceptions are always synchronous in **SML/NJ**, which has advantages for compiling and reasoning about the language. This paper describes the mechanism and its implementation, and describes some applications[1].

2 Continuations in SML/NJ

SML/NJ uses a continuation-passing style (CPS) code generator, which produces high quality code[AJ89]. The use of CPS in the compiler allows the easy introduction of first-class continuations into the language. Unlike in **Scheme**, continuations are statically typed[DHM]; they have the polymorphic type "α `cont`." There are two built-in operations on continuations:

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

A τ `cont` is the type of a function representing the rest of the program with a formal parameter of type τ. Continuations are created using `callcc` (call with current continuation) and are applied using `throw`. A simple example is the expression:

[1] This paper faithfully describes the mechanism as provided in August 1, 1990 release **SML/NJ** (version 0.62).

```
callcc (fn (k : int cont) => (throw k 5; 6)) + 7
```

The variable `k` is bound to the `int cont` that adds `7` to its argument; the `throw` applies `k` to `5`, giving `12`. Continuations provide a natural mechanism for implementing co-routines[Wand80], such as signal handlers.

A subtlety of the continuation mechanism is the interaction between `callcc` and exceptions. Normally, exception handlers are dynamically scoped, but `callcc` binds the current exception handler into the continuation it passes to its argument. To illustrate, consider the following example:

```
exception Foo
val (f : unit -> 'a) = (fn () => raise Foo)
val (g : unit -> 'a) = throw (
      callcc (fn k => (callcc (fn k' => throw k k'); raise Foo)))
fun h x = ((x ()) handle Foo => 1)
```

Applying `h` to `f` will produce the value `1`; applying it to `g` will produce an uncaught exception `Foo`. This is because `g` raises `Foo` in the exception context in which it was bound, instead of in the context of `h`'s handler.

3 ML signal handling

Our approach to supporting asynchronous signals in **ML** is to view them a *continuation producing* operations. When a signal occurs, the run-time system captures the current **ML** state as a continuation, which we call the *resumption continuation*, and passes it to a signal handler. The signal handler returns a new continuation with which the run-time system resumes **ML**. As with the UNIX signal mechanism, the interrupted thread and signal handler are co-routines.

3.1 `System.Signals`

The structure `System.Signals` in the **SML/NJ** pervasive environment provides a low-level interface to the **ML** signal handling (see figure 1). There are a number of signals, corresponding to a subset of the UNIX signals[UNIX] plus a signal generated after garbage collections (`SIGGC`). A signal may be either be ignored or caught. The function `setHandler` is used to install a handler for a given signal, and the function `inqHandler` is used to get the current handler. A value of `NONE` for the handler in these operations specifies an ignored signal. The function `maskSignals` is used to turn signal masking on and off. This operation is cumulative, so that multiple masking operations will nest.

An **ML** signal handler has the type:

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

```
signature SIGNALS =
  sig
    datatype signal
      = SIGHUP | SIGINT | SIGQUIT | SIGALRM | SIGTERM | SIGURG
      | SIGCHLD | SIGIO | SIGWINCH | SIGUSR1 | SIGUSR2
      | SIGTSTP | SIGCONT (* not yet supported *)
      | SIGGC
    val setHandler : (signal * ((int * unit cont) -> unit cont) option)
                         -> unit
    val inqHandler : signal -> ((int * unit cont) -> unit cont) option
    val maskSignals : bool -> unit
  end
```

Figure 1: Signature `SIGNALS`

It takes a count of pending signals[2] and the interrupted thread's resumption continuation as arguments. The signal that is to be handled is not given as an argument, but is implicit in the choice of handler called. The return value of the handler is the continuation with which execution is resumed. All signals are masked while the handler is executing: if a signal occurs, then it is delayed until the handler returns. This is why the handler returns a continuation instead of directly throwing to it. If a signal handler raises an exception, instead of returning normally, it is treated as a run-time system error, and **SML/NJ** will exit with an *uncaught exception* error.[3].

In order to write robust signal handlers, there needs to be some mechanism for atomic actions: i.e., actions that cannot be interrupted by a signal. We guarantee that signal handlers will execute atomically. Signals are masked upon entry to the handler and unmasked when the handler returns. Signals that occur during execution of a handler are postponed until they are unmasked. Signals may also be masked by calling the `maskSignals` function with the value `true`. Note that it is *not* possible to unmask signals during the execution of a signal handler, although one can turn masking on. To avoid delaying the servicing of a signal unnecessarily, it is good practice to write short and simple signal handlers. If an application requires a more complicated and time consuming handler action, then techniques similar to those of section 4.1 should be used.

3.2 Exceptions

The signal handler executes in a different exception handler context than that of the interrupted thread. For this reason, the use of `callcc` to build the return value of the handler requires care. For example, consider the following handler:

[2] The pending signal count is necessary because delays in the handling of a signal can result in multiple occurrences of the signal before being handled.

[3] A higher-level signal interface would presumably implement something more tolerant.

```
fun handler (_, resumek) =
      callcc (fn k1 => (
        callcc (fn k2 => (throw k1 k2));
        doSomething();
        throw resumek ()))
```

The function `doSomething` is called in the exception handler context bound by the outer `callcc`, which is the context in which the handler was called. If `doSomething` raises an unhandled exception, it will cause **SML/NJ** to terminate with an uncaught exception error.

3.3 The default handlers

The structure `Signals` defines default handlers for some signals. Table 1 lists the signals with a short description and the default handler action provided by the structure `Signals`. Most of these signals correspond to standard UNIX signals[UNIX]. The signals `SIGTSTP` and `SIGCONT` are currently unsupported, but will be implemented in the near future. `SIGGC` is generated following every garbage collection.

signal	**default action**	**description**
`SIGHUP`	quit	hangup
`SIGINT`	raise Interrupt	interrupt (e.g., control-C)
`SIGQUIT`	quit	quit (e.g., control-\)
`SIGALRM`	ignore	alarm clock (interval timer)
`SIGTERM`	quit	software termination signal
`SIGURG`	ignore	urgent condition present on a socket
`SIGCHLD`	ignore	child status has been changed
`SIGIO`	ignore	I/O is possible on a descriptor
`SIGWINCH`	ignore	window changed
`SIGUSR1`	ignore	user-defined signal 1
`SIGUSR2`	ignore	user-defined signal 2
`SIGTSTP`	n.a.	currently unsupported
`SIGCONT`	n.a.	currently unsupported
`SIGGC`	ignore	garbage collection

Table 1: Default signal actions

3.4 Handling `SIGINT`

As we noted above, we implement a non-standard policy for handling user interrupts. We decided on this policy after struggling with a version of this mechanism that supported asynchronous exceptions (such as `Interrupt`)[Reppy90].

Our approach is to eliminate asynchronous exceptions, including the exception `Interrupt`. Although this means a variance with the definition, it is a fairly minor one. The definition states:

> If the evaluation of a *topdec* yields an exception (for instance because of a `raise` expression or external intervention) then the result of executing the program "*topdec ;*" is the original basis together with the state which is in force when the exception is generated.
> ([MTH90], page 64)

We believe that the *intent* here is to specify a policy with respect to the top-level read-eval-print loop. This policy can be easily implemented without the exception `Interrupt`. The read-eval-print loop captures its state at the beginning of each iteration as a continuation. The `SIGINT` handler throws to this continuation, which restores the original basis. By banishing asynchronous exceptions we make reasoning about programs that use exceptions more tractable, as well as allowing the compiler more latitude in optimizing away exception handlers.

4 Applications

The interface provided by structure `Signals` is low-level, but general enough to provide the basis for more sophisticated mechanisms. In this section we illustrate the use of oure mechanism by describing a number of applications and programming techniques.

4.1 Masking signals

Our mechanism allows all signals to be masked; but there is no mechanism for masking individual signals. In this section, we describe a signal handling package that allows a mask to be attached to each handler. This mask specifies a set of signals to be masked during the handler's execution. We have the following interface:

```
type handler = (int * unit cont) -> unit cont
type mask = signal list
val install : (signal * (handler * mask) option) -> handler option
```

`Install` installs a signal handler and associated signal mask, returning the previous handler.

In order to provide this finer grain masking of signals, the installed signal handlers are run outside the context of the handlers provided by `System.Signals` (recall that those handlers mask all signals). When a signal occurs, the **ML** signal handler sets the handler mask and returns a continuation that will call the installed handler. When the installed handler returns, the signal mask is reset and control is thrown to the returned continuation. If a masked signal occurs, it is added to a list of pending signals. This list is checked when the installed handler returns and, if there are pending signals, then the returned continuation is passed to the installed handler of the first signal on the pending list. This implementation requires about 100 lines of **ML** code.

4.2 Concurrency

First-class continuations provide an attractive basis for implementing light-weight threads[Wand80]. The continuations of **SML/NJ** have been used to implement co-routine packages (e.g. [Reppy89, Ramsey90]), but providing pre-emptive scheduling requires additional run-time system support. This was one of the major reasons for developing the signal handling mechanism described in this paper.

Shared data-structures, such as the process ready queue, must be accessed atomically with respect to process switches. To insure this, we use a simple scheme to mark the beginning and end of critical regions (see figure 2)[4]. We assume that threads are represented by `unit` continuations,

```
val atomicLevel = ref 0
val signalPending = ref false
fun atomicBegin () = (atomicLevel := !atomicLevel + 1)
fun atomicEnd () = let val level = !atomicLevel - 1
      in
        if (!signalPending andalso (level = 0))
          then callcc (fn k => (
            enqueue k;
            let val next = dequeue ()
            in
              atomicLevel := 0;
              signalPending := false;
              throw next ()
            end))
          else
            atomicLevel := level
      end
fun alrmHandler (_, k : unit cont) =
      if (!atomicLevel > 0)
        then (signalPending := true; k)
        else (enqueue k; dequeue ())
```

Figure 2: Pre-emptive scheduling with atomic regions

and that we have functions `enqueue` and `dequeue` to manipulate the queue of ready threads. If a signal occurs in an atomic region, then the handler sets the `signalPending` flag and doesn't switch threads. When the thread leaves the critical region, it will note the pending fault and relinquish control. The pre-emptive scheduler is quite simple; if the current thread is in a critical region, then the `signalPending` flag is set and the current thread is resumed, otherwise the current thread is placed in the ready queue and another thread is dispatched.

[4] To simplify this presentation, we only concern ourselves with `SIGALRM`.

4.3 Debugging

Tolmach and Appel have built a replay debugger for **SML/NJ**[TA90]. They use source-to-source transformations in the compiler to instrument the user's code. The debugger uses a software counter to keep track of the current "time." The instrumentation code increments this counter and compares it against a "target" time at important points, called *events*, in the code (e.g., binding sites and function entry-points). When the current time matches the target time, then control is transferred to the debugger thread (the debugger and user program are co-routines). One thing that debuggers should provide is a way for the user to asynchronously force a break-point in the execution of the program. With our signal mechanism, this function is easy to provide. The following handler for `SIGINT` will force a break-point at the next debugger event, by resetting the target time and then resuming the user's thread.

```
fun sigintHandler (_, k) = (targetTime := !currentTime+1; k)
```

Of course, a more sophisticated implementation is possible that would allow programs that handle signals (such as concurrent programs) to be debugged. The debugger would provide its own implementation of the `Signal` structure, which would intercept interesting signals.

5 Implementation

There are two levels to the implementation: the underlying run-time support for mapping UNIX signals to **ML** signals, and the implementation of the structure `System.Signals`. Following a quick introduction to the **SML/NJ** run-time system, we describe the implementation bottom-up. Then there is a discussion of the implementation of robust I/O, followed by some performance measurements.

5.1 The SML/NJ run-time system

The **SML/NJ** run-time system is described in [Appel90]), but it has been extensively modified to support signals. We describe the relevant features here.

The **SML/NJ** compiler uses *continuation-passing style* (CPS) code generation[AJ89]. Unlike other CPS-based compilers ([KKR+86] for example), **SML/NJ** has no run-time stack; the function return continuations are heap allocated. The generated code is also heap allocated, so special tagging techniques are used to allow the garbage collector to deal with program counters and code objects.

Because there is no run-time stack, the **SML/NJ** code generator uses a register model. The **ML** state consists of a three special-purpose registers and a machine dependent number of root registers. Other registers may be used as temporaries, but these are not visible to the run-time system. Five of the root registers have special meaning: one is the program counter, one holds the current exception handler context, and the other three are used in the standard calling convention. There are two kinds of functions: a two-argument function has the standard argument and closure registers as parameters; a three-argument function also has the standard return continuation register as a parameter. Two-argument functions are functions that never return, such as return continuations and the continuations produced by `callcc`.

A **C** structure, called the *state vector*, is used to hold the **ML** state in the run-time system. Two assembly routines provide the interface between **C** and **ML** code. The run-time system invokes **ML** code by loading the state vector and calling `restoreregs`, which loads the machine registers from the state vector and jumps to the given program counter. When **ML** code requires a service from the run-time system, it loads a global variable `request` with a request code, and calls `saveregs`, which saves the **ML** state and returns to the **C** code that called `restoreregs`. The `restoreregs`/`saveregs` protocol is essentially a co-routine switch.

SML/NJ has a simple, but efficient, semi-generational garbage collector[Appel89a]. Allocation is also fast: it is open-coded and only requires a couple more instructions than object initialization. A heap limit check is generated at the entry point of each *code tree*[5]. If there is not enough free space for the maximum possible allocation in the code tree, then a *garbage collection trap* (GC-trap) occurs[6]. The entry-point of a code tree is proceeded by an object descriptor word, thus the program counter at the trap point is a valid heap pointer. Handling a GC-trap involves the following steps:

1. The run-time 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 coded routine `saveregs`.

2. `Saveregs` saves the ML state in the state vector and passes control up 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 jumps to the trap location.

5.2 Run-time support for signals

The major problem with handling an asynchronous signal is to avoid corrupting the heap. The **ML** signal handler cannot be dispatched immediately, since the current thread may be in the middle of an allocation. There are a number of ways to deal with this problem. One approach is to have the run-time system recognize this situation and emulate the instruction stream until the allocation is

[5] A *code tree* (or *extended basic block*) is an acyclic set of blocks with one entry-point and one, or more, exits.

[6] The GC-trap is just a UNIX signal.

complete[MK87, Appel89b]. This approach has the substantial disadvantage that it requires an emulator for every different architecture, reducing portability[7].

Instead, we use an approach similar to the one used by **Argus** for light-weight context switches[LCJS87]: we synchronize signals with "safe" points in the code. The heap limit checks at the code tree entry-points are a convenient choice for the synchronization points; by delaying a signal until the next check we can guarantee safety. When an asynchronous signal occurs, we adjust the limit register to force a garbage collection trap on the next limit check. The run-time system recognizes this as really being a signal, and passes it to the **SML** signal handler[8]. This technique has the additional advantage that it doesn't incur any additional run-time overhead on normal execution.

More specifically, let us consider the case of a signal occurring during the execution of **ML** code. When a signal occurs, the following steps are taken:

1. The run-time routine `sighandler` catches the signal. It records the signal and resumes execution of **ML** code via an assembly routine that adjusts the heap limit register.
2. The **ML** code executes until the next heap limit check, which will trap.
3. `Ghandle` recognizes that the GC-trap is really a signal trap, and sets `request` to `REQ_SIGNAL`. It saves the faulting program counter and returns control to the assembly coded routine `saveregs`.
4. As with a GC-trap, `saveregs` saves the ML state in the state vector and passes control up to `run_ml`.
5. `Run_ml` allocates a resume continuation using the saved state, builds an argument tuple and calls the **ML** signal handler (via `restoreregs`).
6. The **ML** handler deals with the signal and then returns a resume continuation. The return continuation of the **ML** handler is a piece of assembly code that calls `saveregs` with `request` set to `REQ_SIG_RETURN`.
7. `Run_ml` calls the resume continuation (via `restoreregs`).

Of course, another signal can occur while we are in the process of handling a signal, so it is necessary to provide some concurrency control. We use a small collection of global variables for this purpose (see figure 3). The `inML` flag is set by `restoreregs` and cleared by `saveregs`; it marks when execution is in **ML** code. The flag `handlerPending` is set by `sighandler` (step 1) to note that a handler trap is pending. This flag is cleared by `ghandle` (step 3), which sets the

[7] **SML/NJ** is currently supported for 5 different machine architectures, so this is a major concern.

[8] The run-time system checks on the available memory and, if necessary, does a garbage collection prior to invoking the **ML** signal handler.

```
int   inML;           /* This flag is set when we are executing ML code.  */
int   handlerPending; /* This flag is set when a handler trap is pending, */
                      /* and cleared when the handler trap occurs.        */
int   inSigHandler;   /* This flag is set when a handler trap occurs and  */
                      /* is cleared when the ML handler returns.          */
int   maskSignals = 0;/* When set, signals are masked.                    */
int   NumPendingSigs; /* This is the total number of signals pending.     */
```

Figure 3: Concurrency control globals

inSigHandler flag. The inSigHandler flag is cleared by restoreregs (step 7), after the signal has been handled. If either handlerPending or inSigHandler is set when a signal occurs, then sighandler records the signal, but does not adjust the heap limit register or set the handlerPending flag. The UNIX signal handlers sighandler and ghandle are installed with all signals masked, so they execute atomically. This atomicity, coupled with the two-phase structure of handling signals, guarantees that any signal that occurs during signal handling will be postponed until after the first signal is handled. The flag maskSignals is used to implement signal masking outside of a signal handler.

If a signal occurs during execution of **C** code (e.g., during a garbage collection), then we record it and resume execution. The assembly routine restoreregs checks for pending signals just prior to resuming **ML** execution. Figure 4 contains the Motorola MC680x0 code for this operation After setting the inML flags, this code tests for pending signals. If there are any, then it checks

```
_restoreregs:
      ...
      addql #1,_inML              /* note that we are executing ML code */
      tstl  _NumPendingSigs       /* check for pending signals */
      jeq   call_ml
      tstl  _maskSignals          /* are signals masked? */
      jne   call_ml
      tstl  _inSigHandler         /* are we currently handling a signal? */
      jne   call_ml
      addql #1,_handlerPending    /* note that a handler trap is pending */
      clrl  d5                    /* force a trap on the next limit check */
call_ml:
      jmp   a5@                   /* jump to the ML code */
```

Figure 4: Returning to **ML**

to see if it is in the second phase of signal handling, and, if not, it starts the first phase of signal handling. Since starting the first phase is an idempotent operation, it doesn't matter if a signal occurs at this point.

5.3 The implementation of `System.Signals`

The implementation of `System.Signals` is fairly straight-forward. At the core of the implementation is the root **ML** signal handler, which is called by the run-time code. A global array is maintained, that records the current status of each signal:

```
datatype sig_sts
  = ENABLED of ((int * unit cont) -> unit cont)
  | DISABLED
val sigvec : sig_sts array
```

The functions `setHandler` and `inqHandler` update the array appropriately. If a call to `setHandler` makes a disabled signal enabled, or visa-versa, then the run-time system is notified by a call to the **C** function `enablesig`[9]. The code for the root **ML** signal handler is given in figure 5. The signal handler should never get called on a disabled signal, since the run-time system blocks them. A global counter is used to keep track of the nesting level of signal masking. A call to `maskSignals` increments or decrements this counter depending on the argument. On a `0`–`1` (`1`–`0`) transition, a call is made to a **C** routine that sets (clears) the `maskSignals` flag.

```
fun sigHandler (code, count, resume_k) = (
      case (InLine.subscript(sigvec,  code))
       of DISABLED => resume_k
        | (ENABLED handler) => handler (count, resume_k))
```

Figure 5: Root **ML** signal handler

5.4 Signals and I/O operations

SML/NJ implements in and out streams as a buffered I/O library written in **ML**. The run-time system provides an interface to the `read` and `write` system calls[UNIX], which are used by the I/O library. The implementation of the stream operations raise a number of problems with respect to signals:

- An input operation, such as reading the line from the terminal, can block indefinitely (i.e., until the user hits the "return" key), which conflicts with the desire to have a user interrupt to force control back to the top-level prompt.
- A signal that occurs during an I/O operation may cause inconsistencies between the buffer and the operating system. This results in input being lost and output being printed twice.

[9] The run-time system also shadows the status of **ML** signals, in order that the state can be restored for exported images.

- A signal that occurs during an I/O operation may cause inconsistencies between the user program and the buffer.

Our approach to implementing I/O solves the first two problems, and the third problem can easily be solved, using signal masking, in applications that require it.

To understand the subtleties of this interaction, consider the case of signal occurring during a call to `read`. The signal handler[10] has two options: it can either interrupt the operation or resume it. Unfortunately, neither option will work. If the signal occurs immediately following the completion of the `read`, then interrupting it will result in data loss. On the other hand, if the signal occurs just prior to the initiation of the `read`, then indefinite blocking may occur. Another problem with interrupting an I/O operation is that we may need to to restart it (e.g., if the signal is `SIGALRM` and it causes a context switch).

To address these problems, we divide the buffered I/O operations into two phases. First we wait for the operation to be ready, and then we actually do the operation. The first phase is idempotent, and so can restarted safely. Once the I/O operation is ready, we make the `read`/`write` system call and update the buffers. To illustrate, the **ML** code to fill an input buffer looks something like

```
fun filbuf (INSTRMfilid, buf, pos, len, ...) = (
      in_wait filid;
      protect (fn _ => (pos := 0; len := read(filid, buf, bufsize))) ())
```

The `protect` function (figure 6) is used to guarantee that the buffer and operating system have a consistent view of the file.

```
fun protect f x = let
      open System.Signals
      val _ = maskSignals true
      val y = (f x) handle ex => (maskSignals false; raise ex)
      in
        maskSignals false;
        y
      end
```

Figure 6: Protect a function call

The run-time system provides *wait for input/output* operations to **ML**. These operations are unique among the run-time routines in that `sighandler` will interrupt them. Figure 7 gives pseudo **C** code for the wait for input routine. The `ioWaitFlag` is used to mark that we are waiting for I/O. If it is set when a signal occurs, then `sighandler` does a `longjmp` to the waiting routine, which restores the **ML** state vector to the value it had when the input wait routine was called by **ML**.

[10] This is the **C** function `sighandler`, not an **ML** handler.

```
if (setjmp(env) == 0) {
    ioWaitFlg = 1;
    if (NumPending Sigs == 0) {
        wait for input to be available
    } else {
      /* A signal was already pending */
        restore the ML state vector
    }
    ioWaitFlg = 0;
} else {
  /* A signal occured while waiting */
    restore the ML state vector
}
```

Figure 7: Wait for input

The **ML** signal handler is then called with a resumption continuation that will retry the I/O wait operation. If a signal occurs before we set the `ioWaitFlag`, then we also restore the **ML** state. If a signal occurs after the `ioWaitFlag` has been cleared, then the wait operation completes.

5.5 Performance

Signals such as `SIGINT` occur infrequently enough that performance is not a major concern, but for concurrency packages that use `SIGALRM` to provide pre-emption, signal handling overhead is an issue. Table 2 contains measurements of the overhead of handling `SIGALRM` signals in **ML**. Our benchmark is a simple program that doesn't do any allocation. We measured the time it took to run with the interval timer turned off and the time needed with the timer on, and used the difference in these times to compute the overhead. The relative speed column in table 2 relates the time it took to run the benchmark (without signals) on the various systems. The Sun systems were all running SunOS 4.0.3, the DECstation was running Ultrix V2.2, and the NeXT was running release 1.0. The relatively high cost of signal handling on the NeXT machine is probably because the NeXT's

System	Relative Speed	μS per signal	Overhead (50 signals/sec)
Sun 4/490	7.3	350	1.7%
Sun SPARCstation 1+	4.6	530	2.6%
DECstation 3100	4.2	420	2.1%
Sun SPARCstation 1	3.8	600	3.0%
Sun 4/280	3.2	460	2.3%
NeXT	1.3	2,400	12%
Sun 3/60	1.0	1,300	6.3%

Table 2: Signal handling costs

implementation of UNIX signal handling is built on top of the MACH exception mechanism.

6 Future work

We know of a few remaining problems with the current implementation; the following is a brief discussion of some of these.

6.1 Incremental garbage collection

One problem area in our implementation is the interaction between signals and the garbage collector. A signal that occurs during execution of **C** code in the run-time kernel cannot be safely handled, but must be delayed until the next heap-limit check in ML code. Most of the run-time routines are quick enough so that this isn't a problem, but, unfortunately, a garbage collection can take several seconds[11], during which several hundred timer signals can occur. One solution would be to use a full generational collector, such as those described in [Ungar84] and [WM89]. We suspect, however, that this will still be inadequate for concurrent applications with real-time responsiveness requirements. What is really needed is a way to interleave small amounts of garbage collection work with program execution. The **Pegasus** garbage collector has this property, but its heap architecture involves substantial overhead on allocation and object accesses[NR87]. Another approach is to use page-protection tricks[Shaw87, AEL88], but these techniques may be too expensive on some systems and not possible on others. We are exploring these, and some other, ways of addressing the problem.

6.2 Unimplemented signals

The current implementation does not support the signals `SIGTSTP` and `SIGCONT`, which are generated as a result of suspending and resuming the process. Because these signals cannot be masked, the assumption that `sig_handler` and `ghandle` are atomic with respect to signals doesn't hold. This means that supporting these signals will require some cleverness.

6.3 MACH support

As the benchmarks attest, the performance under MACH leaves something to be desired. The problem is that we are using the UNIX signal mechanism, which is not directly supported by the MACH kernel. Providing an implementation based directly on the MACH exception ports will improve performance substantially.

[11] About 2 ms. of CPU time per kilobyte of live data on a Sun SPARCstation. The real-time delays are often much worse, because of paging.

Acknowledgements

Bill Aitken made the suggestion that **SML** ought to have a signal handling mechanism. Andrew Appel helped with implementing the heap limit checks, and discussions with Andrew Appel, Dave MacQueen and Norman Ramsey helped in cleaning up the interface.

References

[AEL88] Appel, A.W., J.R. Ellis and K. Li. “Real-time concurrent collection on stock multiprocessors,” *Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation,* June 1988, pp. 11-20.

[Appel89a] Appel, A.W. “Simple generational garbage collection and fast allocation,” *Software – Practice and Experience*, V 19, Nr. 2, February 1989, pp. 171-183.

[Appel89b] Appel, A.W. “Allocation without locking,” *Software – Practice and Experience*, V 19, Nr. 7, July 1989, pp. 703-705.

[Appel90] Appel, A.W. “A runtime system,” to appear in the *Journal of Lisp and Symbolic Computation*, 1990.

[AJ89] Appel, A.W. and T. Jim. “Continuation-passing, closure-passing style,” *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, January 1989, pp. 293-302.

[AM87] Appel, A.W. and D.B. MacQueen. “A standard ml compiler,” *Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science 274, Springer-Verlag, September 1987, pp. 301-324.

[DHM] Duba, B., R. Harper and D.B. MacQueen. “Type-checking first-class continuations,” *in preparation*.

[MTH90] Milner, R., M. Tofte and R. Harper. *The definition of standard ml,* The MIT Press, Cambridge, Mass., 1990.

[KKR+86] Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin and N. Adams. “Orbit: an optimizing compiler for scheme,” *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction*, July 1986, pp. 219-233.

[LCJS87] Liskov, B., D. Curtis, P. Johnson and R. Scheifler. “Implementation of Argus,” *Proceedings of the Eleventh ACM Symposium on Operating System Principles,* November 1987, pp. 111-122.

[MK87] Moss, J.E.B. and W.H. Kohler. “Concurrency features for the trellis/owl language,” *ECOOP’87,* Lecture Notes in Computer Science 276, Springer-Verlag, pp. 171-180.

[NR87] North, S.C. and J.H. Reppy. “Concurrent garbage collection on stock hardware,” *Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science 274, Springer-Verlag, September 1987, pp. 113-133.

[Ramsey90] Ramsey, N. “Concurrent programming in ml,” *Technical Report CS-TR-262-90,* Department of Computer Science, Princeton University, April 1990.

[Reppy89] Reppy, J.H. “First-class synchronous operations in standard ml,” *Technical Report TR 89-1068*, Computer Science Department, Cornell University, December 1989.

[Reppy90] Reppy, J.H. “Asynchronous signals in standard ml (**SML/NJ** version 0.56),” *unpublished draft*, May 1990.

[Shaw87] Shaw, R.A. “Improving garbage collection performance in virtual memory,” *Technical Report CSL-TR-87-323*, Computer Systems Laboratory, Stanford, March 1987.

[TA90] Tolmach, A.P. and A.W. Appel. “Debugging standard ml without reverse engineering,” *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, June 1990, pp. 1-12.

[Ungar84] Ungar, D. “Generation scavenging: a non-disruptive high performance storage reclamation algorithm,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 1984, pp. 157-167.

[UNIX] *UNIX programmer's reference manual*, 4.3 Berkeley software distribution, April 1986.

[Wand80] Wand, M. “Continuation-based multiprocessing,” *Conference Record of the 1980 Lisp Conference*, August 1980, pp. 19-28.

[WM89] Wilson, P.R. and T.G. Moher, “Design of the opportunistic garbage collector,” *Proceedings of OOPSLA'89*, October 1989, pp. 23-35.