Implementing Concurrency For An ML-based Operating System

by

Albert C. Lin

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1998

Author .....

Department of Electrical Engineering and Computer Science
February 3, 1998

Certified by .....

Olin Shivers
Research Scientist
Thesis Supervisor

Accepted by .....

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Eng.

Implementing Concurrency For An ML-based Operating System

by

Albert C. Lin

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 1998, in partial fulfillment of the
requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this paper I describe the design, implementation, and features of ML/OS, an operating system with an embedded ML compiler. ML/OS supports a continuation-based thread model of concurrency with non-blocking, interrupt-driven input/output. By embedding the ML compiler into the operating system, ML/OS attempts to eliminate levels of abstraction that are present in traditional interactions between compilers and operating systems. By using a continuation-based scheduler, I demonstrate the use of advanced programming language features such as continuations and type safety in system-level programming.

Thesis Supervisor: Olin Shivers
Title: Research Scientist

Acknowledgments

Without the assistance and support of many people, this research would not have been possible.

First and foremost, I'd like to thank Olin Shivers, my research supervisor, for his guidance and support. His ideas, technical assistance, inspiration, humor, and near-infinite patience have seen this project through from beginning to end.

Lorenz Huelsbergen and John Reppy, both of Bell Labs (Lucent Technologies), have been invaluable in providing technical assistance for the mass of code that is SML/NJ and CML. Their willingness to answer questions and assist in debugging my code was enormously helpful.

Jay Lepreau and the rest of the Flux OS Toolkit team at the University of Utah were extremely helpful and cooperative in getting the OSKit up and running on the AI Lab machines. Their technical help, example programs, and code updates made the OSKit a truly useful toolkit for OS research.

I'd like to thank everyone named James Tetazoo for being so inspiring, interesting, supportive, and cool. In particular, I'd like to thank hxl for everything; Martin, Kathy, and Kathy for listening to me; and Srikar for talking to me.

Finally, I'd like to thank my parents and grandparents for their love and patience. From the FORTRAN coloring book[15] to my higher education and everything in between, their enthusiasm and support have helped me tremendously.

Contents

1 Introduction **5**

2 Background **6**
2.1 Concurrency 6
2.2 ML 6
2.3 Related Projects 7

3 Overview **9**
3.1 Components 9
3.1.1 SML/NJ 10
3.1.2 OSKit 11
3.1.3 GRUB 12
3.2 ML/OS 13

4 Implementation **15**
4.1 SML/NJ Modifications 15
4.1.1 C Libraries 15
4.1.2 Memory Management 16
4.2 OSKit Modifications 17
4.2.1 Nonblocking I/O 17
4.3 Putting It All Together 18
4.3.1 Heap Conversion 18
4.3.2 Trap/Signal Interface 19
4.3.3 Alarm Clock Facility 21
4.3.4 Interrupt-Driven I/O 22
4.4 CML 23
4.5 Application: Session Manager 25
4.5.1 Limitations 27

5 Results **28**
5.1 Advanced Language Features 28
5.2 Streamlining the OS 28
5.3 Continuations And Concurrency 29
5.4 Drawbacks/Difficulties 30
5.4.1 SML/NJ 30
5.4.2 ML/OS 31

6 Conclusions **32**
6.1 Future Directions 32

A ML/OS Session Manager **33**

1 Introduction

There has been considerable research in the area of advanced programming languages, especially functional languages such as Scheme, ML, and Haskell. There have also been a number of projects focusing on embedding programming languages into operating systems. The product of this project, ML/OS, is an investigation into the potential benefits and effects of combining the features of advanced languages and language-based operating systems. Specifically, I examine the feasibility of implementing a low-level operating-system mechanism, concurrency, using high-level advanced-language features such as higher-order procedures and continuations.

The Background section introduces the concepts behind ML and ML/OS. Concurrency is discussed, as are the features of the ML programming language. In addition, I mention other advanced languages, previous work in the field, and related projects.

The Overview discusses the existing software components that were used to build ML/OS. It explains the rationale behind using SML/NJ and the OSKit in building the operating system. It also introduces the concurrency model and other features of ML/OS.

The Implementation section discusses in detail the process of building ML/OS. The numerous modifications to SML/NJ and OSKit are motivated and explained. The steps involved in putting all the pieces together are laid out, and a sample application for ML/OS is discussed.

The Results section examines the various benefits and drawbacks of ML/OS. It discusses the effects of using an advanced language and having the compiler and operating system in close interaction. It also lists some drawbacks and avenues for improvement within ML/OS.

The Conclusions section draws a few summary conclusions about ML/OS. It also proposes a number of directions for future research based on ML/OS.

2 Background

2.1 Concurrency

Concurrency refers to the simultaneous progress of independent computations within a computer and the syntactic constructs used to express and control these computations. Over the years, people have used a number of different approaches to implementing concurrency or multitasking in computer systems. Some of the various concurrency mechanisms include threads and multi-processing. Threads[5] are independent pieces of program execution sharing the same address space within a machine. Multi-processing is the practice of using multiple physical processors to do computations in parallel. There has been extensive research into different methods of task switching and thread management as well as issues such as shared memory and avoiding deadlock.

2.2 ML

The primary implementation language used in this project is ML (short for Meta-Language). Originally developed as a part of the Edinburgh LCF theorem-proving system[12], ML is a programming language with a number of features that make it an attractive candidate for advanced language research. Among these features are polymorphism, type checking and inference, safety, and a formally defined semantics.

ML is a *functional programming language.* A functional language is a programming language in which function application is the primary means of computation. Functional languages are also characterized by their treatment of functions as first class values and minimization of side effects.

One of the most distinctive features of ML is its *strong polymorphic type system.* The type system ensures that values of incompatible types aren't mixed, and a type inference algorithm assigns to expressions of unspecified type the most general types possible that maintain type-safety. ML is notable in that it supports type inference in the face of polymorphic functions. For example, ML is able to correctly

type-check an application of the polymorphic list `map` function. Type checking and inference allow ML to detect type errors during compilation. Detecting these errors at compile-time instead of at run-time catches many common mistakes and speeds the development cycle.

ML features a *formally defined semantics.* By referring to *The Definition of Standard ML*[17], one can reason precisely about ML program behavior and prove certain properties of ML programs. The semantics also provide an unambiguous definition of the language for programmers, compiler writers, and language designers.

ML is a *safe language.* This means that valid programs (as defined by the semantics) written in ML cannot cause the run-time system to fail catastrophically. The type system, exception handlers, and memory management ensure that it is impossible for programs to do things like read from unallocated memory or perform pointer arithmetic. As a result, distinct ML programs can run concurrently in the same address space without needing run-time boundaries to prevent them from interfering with each other. ML provides the safety of processes and the efficiency of threads.

Other powerful features of ML include *higher-order procedures*, a *garbage collector*, and a *module system.* Higher-order procedures in ML, as in Scheme and other functional languages, provide a powerful means of abstraction and a way to easily capture and name computations. The garbage collector relieves the programmer from having to deal with memory management issues. The module system aids in developing effective programming abstractions.

The specific implementation of ML used in ML/OS is version 109.30 of Standard ML of New Jersey (SML/NJ)[2, 3].

2.3 Related Projects

The project most closely related to this one is the Fox project[7, 14] at CMU. Though the Fox project members have not implemented an operating system, they have written an entire TCP/IP stack in ML to investigate the feasibility of using ML as a language for systems programming. In addition, they have written a number of

network applications, including a WWW server, in ML.

A number of other existing projects also examine the effects of embedding a language into an operating system and providing OS services in an advanced programming language. The Lisp Machine project[4, 13] featured both hardware and software that supported an operating system with Lisp as its primary interface. The SPIN[27] operating system allows applications to execute partially in the kernel's address space via kernel extensions and the safety properties of the Modula-3[19] programming language. The Sting[21] operating system uses the Scheme programming language and is based on virtual processors and lightweight threads. The Inferno[16] operating system features modules, threads, type-checking, and a type-safe language, Limbo.

The idea of using safe languages in operating systems is not new. The IBM 801[22] featured a single address space and provided a language, PL.8, that featured a safe subset. Similarly, the Alto was another single address space system programmed in a safe subset of the Mesa[18] programming language.

3 Overview

The goal of this research project was not to create a novel operating system, compiler, or advanced language, but rather to investigate the process and effects of combining the three into a single tightly integrated unit. The interactions between any two of these components are interesting in their own right; this research examines those interactions as well as the characteristics of the entire system.

Given the complexity of even the most basic operating systems, building an entirely new operating system from the ground up is a daunting task even for experienced OS hackers. In fact, the point of an operating system is to provide a simple interface to complex hardware, and it's the OS implementor that needs to know the full details of the underlying machine. Also, the complexity of most compilers and especially those for advanced languages makes it worthwhile to use an existing compiler instead of writing a new one, which would be a project unto itself.

In order to contain the scope of this project, pre-existing components were utilized to build ML/OS. Fortunately, there are tools available to assist in building operating systems as a test bed for new ideas in operating system research. ML/OS is built from the Flux OS Toolkit[9, 10], the SML/NJ compiler[2, 3], the GRUB boot loader[6], and original code. The Concurrent ML (CML) package[23, 24] was ported and used as the base concurrency model of ML/OS. In addition to putting these various packages together into one system, a number of different I/O options were added to ML/OS.

3.1 Components

The use of existing code provided a way to quickly obtain a running kernel to which interesting extensions could be added. Since the component packages used are fairly complete and mature, it ensured that large portions of the code would be reasonably functional and stable as well. Although it was not quite as simple as would have been expected, interfacing the various components was still considerably less effort than writing both a compiler and an operating system from scratch.

3.1.1 SML/NJ

Standard ML of New Jersey is a compiler and runtime system for ML developed at AT&T Bell Laboratories and Princeton University. It is now a joint project between AT&T Research, Bell Laboratories (Lucent Technologies), Princeton University, and Yale University. ML/OS incorporates most of the SML/NJ run-time system and ML heap.

SML/NJ was used as the compiler for ML/OS for a number of reasons. It is one of the most mature implementations of ML available; though the last "official" release was in 1993, current working releases are available. In the course of development, ML/OS was modified to use several different "current" versions of SML/NJ. The source code is freely available, making it possible to modify the compiler. The development team is sizeable, and the developers were very helpful in providing assistance in navigating and understanding the source code.

SML/NJ is also perhaps the most full-featured of any existing ML implementation. There are many auxiliary packages included with SML/NJ, including ML-Lex, ML-Yacc, a compilation manager (CM), Concurrent ML (CML), and a graphical user interface module (eXene). The SML/NJ modules made it easier to add functionality to ML/OS as it was being developed.

One of the most interesting features of SML/NJ is that it runs without a stack: all procedure frames are allocated in the heap, and heap allocation/deallocation is very inexpensive. One result of stackless operation is that higher-order functions and continuations can be created cheaply. The fact that continuations are inexpensive in SML/NJ enables their use as a primitive operating system concurrency mechanism.

SML/NJ consists of a run-time system, written in C, that loads a heap image from disk, containing compiled ML code and other ML data structures. The heap image contains a designated top-level procedure for the run-time system to invoke upon loading. The SML/NJ heap is largely operating-system independent, while the run-time system varies from platform to platform. Under a standard operating system environment, SML/NJ is run by starting the run-time system, which loads

the heap from disk and invokes the designated top-level procedure.

3.1.2 OSKit

The Flux OS Toolkit (OSKit) is a set of basic operating-system components developed at the University of Utah that is intended be a tool for OS researchers[9, 10]. It consists of several modular libraries that can be used separately or together in building an operating system. These libraries include a memory manager, a minimal C library, and a set of device drivers adapted from Linux and FreeBSD. The OSKit also includes documentation and example kernels, making it very easy to quickly compile a working kernel. The most useful parts of the OSKit were the low-level kernel-support library, the memory-management library, and the minimal C library.

The kernel-support library provides a basic environment for the kernel by installing default handlers for hardware traps and providing primitive keyboard input and screen output. It also provides functions for low-level hardware manipulation such as switching processor modes and setting up page tables. The library enables an OS implementor to get a kernel up and running without having to learn every single hardware detail or knowing exactly how to set up interrupt handlers and switch processor modes.

The memory-management library frees the developer from having to worry too much about memory, if that is not the focus of the research. It provides a flexible list-based memory manager that can deal with the memory-allocation issues found in typical operating system kernels. The memory manager is implemented at a fairly low level; for example, it doesn't record the sizes of allocated blocks of memory. However, the OSKit provides a simple implementation of `malloc()` and `free()` based on the memory-management library. SML/NJ has a built-in memory manager that allocates large blocks of memory from the operating system and uses its own high-performance system for allocating and managing ML data structures such as records, lists and procedure activation frames. As a result, it does not need a very complex operating-system interface for managing memory, and the provided `malloc()` and `free()` interface of the OSKit is adequate for the needs of ML/OS.

The minimal C library implements a subset of the standard POSIX C library. It provides many standard string, memory, and output functions to the kernel developer, so writing kernel code becomes similar to writing standard user code. For example, primitive I/O is provided in the form of `printf()`, `putchar()`, and `getchar()`. The OSKit C library enables the easier porting of existing applications into the kernel, and it saves the OS developer from having to rewrite many of the standard library functions.

Another very useful feature of the OSKit is its serial-line debugging capabilities. The OSKit can be used to build a kernel that can send gdb-readable debugging packets over a serial port. By using the remote debugging capabilities of gdb and a serial cable connected to a test machine, serial-line debugging enables source-level debugging of the running ML/OS kernel. This allows the developer to use all of the functionality of gdb or a compatible debugger to debug the kernel. Without this feature of the OSKit, debugging ML/OS would have been via the exceedingly cumbersome method of strategically inserted calls to `printf()` and `halt()`.

3.1.3 GRUB

Another important piece of software used in ML/OS is GRUB[6], the GRand Unified Bootloader. Written by Erich Boleyn, GRUB is a versatile boot loader for a large variety of operating systems. It contains code that can read most common filesystems, such as those used by Linux, FreeBSD, Windows, and DOS. As a result, the kernel can be compiled in a FreeBSD environment, then later booted directly from the BSD filesystem by GRUB. This sped up the development cycle, as the ML/OS kernel could be compiled on a development machine and then copied to a test machine for booting. The filesystem support of GRUB eliminated the need to dedicate a disk partition to ML/OS and write the kernel into that partition for testing.

GRUB also has support for so-called boot modules. Boot modules are pieces of non-kernel data that can be passed to the operating system as part of the boot sequence. A number of operating systems, such as Mach, use boot modules to load device drivers and other pieces of functionality that aren't in the kernel. By sepa-

rating the boot modules from the kernel, the operating system has more flexibility and convenience in deciding what modules to load. Also, loading boot modules at boot time is more efficient than compiling them into the kernel, then finding and separating the relevant data after booting.

In the early stages of ML/OS development, the SML/NJ heap image was linked into the ML/OS kernel, and the SML/NJ run-time system was modified to read the heap image directly from memory instead of disk. Later in the development process, the heap image was separated from the kernel and treated as a GRUB boot module. The run-time system was also changed to locate the boot module in memory and read the heap image from the resulting location. Treating the heap image as a boot module makes the ML/OS compilation and testing process much more efficient, as it separates the compilation processes for the run-time system and the heap image. Changes can be made to the relatively small run-time system without the need to regenerate the heap image. Similarly, changes to ML code in the heap image do not have to be linked into the run-time system before booting.

Overall, GRUB provides a great deal of flexibility and convenience when booting the ML/OS kernel. Its support for reading directly from common filesystems helped speed up the development cycle, as did its boot-module capabilities.

3.2 ML/OS

ML/OS combines the functionality of SML/NJ and the OSKit and adds some of its own. It takes the SML/NJ run-time system and embeds it into an OSKit-based kernel. At bootup, the ML/OS run-time system relocates the ML heap image in memory and executes it. ML/OS runs entirely in physical memory; virtual memory and other memory abstractions are not currently supported. Modifications were made to both SML/NJ and the OSKit; functions were written in C and ML to smoothly interface between the two systems. ML/OS adds an interrupt-driven I/O mechanism and a port of CML, the concurrency module for SML/NJ. The concurrency model of CML makes use of higher-order procedures and explicit continuation handling. By using CML in SML/NJ as the concurrency mechanism for the en-

tire OS, ML/OS investigates the effects of implementing basic OS functions in an embedded advanced language.

4 Implementation

Creating an operating system with an embedded ML compiler required modifying SML/NJ and the OSKit, interfacing the two packages, and adding features to the new system. Standard ML of New Jersey was modified to work within the environment of the OSKit, which is markedly different from the Unix or Windows environment under which it is written to operate. The OSKit was extended to work with SML/NJ. Various interface functions were written in order for the system to compile as one unit, and the Concurrent ML package was ported to ML/OS. Finally, extra I/O functionality and a sample application were written to demonstrate the capabilities of ML/OS.

4.1 SML/NJ Modifications

Standard ML of New Jersey expects to be run as an application in a full-featured OS with the standard components available from an OS: memory management, processes, a filesystem, networking, etc. Since the OSKit is a toolkit for building operating systems that only provides a basic kernel environment, a number of changes were made to SML/NJ in the process of building ML/OS.

4.1.1 C Libraries

The run-time system of SML/NJ has a set of C libraries that allows ML functions to use Unix system calls. They provide the means to perform system calls such as `read()`, `write()`, `fork()`, `pause()`, `select()`, and `gethostbyaddr()` from within ML. Since many of these Unix system calls are not provided by the OSKit's minimal C library, the corresponding functions in the C libraries were replaced with dummy functions. The type signatures of the affected functions had to be changed to match the type of the dummy functions. For example, the ML invocation of `exec` changed from a function of type `(string * string list) -> 'a` to a dummy function of type `string -> unit`. All functions in the C libraries that were not explicitly needed for the basic functionality of ML/OS were replaced with dummy functions of type

`string -> unit`.

As functionality was added to ML/OS, some of the C functions were added back to the SML/NJ C library and made available to ML. Needed functions such as `read()`, `select()`, `pause()` and `gettime()` were later implemented using the OSKit.

4.1.2 Memory Management

The memory-management library of SML/NJ was modified to work with OSKit. The SML/NJ memory-management library provides for allocation of memory blocks by using one of a number of different underlying mechanisms depending on the host operating system. The OSKit provides a basic memory management library but no higher-level functions such as `mmap()` or `vm_allocate()`. Instead of using these functions, the SML/NJ library was modified to use `malloc()` and `free()`. The SML/NJ run-time system requests a large block of memory once at boot time and manages ML memory allocation requests internally, so the way memory is allocated to the run-time system is not very important. Thus, using `malloc()` is sufficient for the purposes of ML/OS.

SML/NJ manages memory via garbage collection and open-coded sequences for allocation. Since there is no stack, all allocations are done on the heap. SML/NJ can very efficiently allocate memory by simply moving its heap pointer and allowing the stop-and-copy garbage collector to reclaim unused memory. In fact, it can allocate and initialize n words in approximately $n+2$ instructions. In order for this to work, though, the allocations need to be atomic, otherwise an interrupt in the middle of a memory allocation could leave the heap in an inconsistent state, corrupting the heap and possibly crashing the system.

SML/NJ achieves atomicity by deferring all interrupt handling until the ends of basic blocks. If an interrupt occurs in the middle of a basic block, it is noted and the limit pointer (representing the end of available memory) is set to be equal to the heap pointer (representing the end of allocated memory). After the limit and heap pointers are adjusted, the interrupted code is resumed without servicing the

interrupt. SML/NJ does a garbage collection check on every basic block boundary; if the heap pointer and the limit pointer are equal, it appears as if there is no remaining free memory and a garbage collection is necessary. When the garbage collector is invoked, it first checks whether or not there are signals pending. If there are no signals pending, the garbage collector proceeds as usual. If there are pending signals, the limit pointer is restored to its original value, and the appropriate interrupt handler is called.

In short, signals arriving in the middle of basic blocks are deferred, and the limit pointer is temporarily adjusted so as to "fake out" the end-of-basic-block garbage-collection check, which cedes control to the garbage collector, which checks for pending signals and calls a signal handler if necessary. Deferring interrupts until basic-block boundaries ensures that heap allocation operations are atomic. A full explanation of this technique can be found in a paper by Reppy[25].

4.2 OSKit Modifications

The OSKit required relatively little modification for use as part of ML/OS, since it was designed to be used for OS projects. The parts of the OSKit used most heavily in ML/OS were the kernel support library, the minimal C library, and the memory management library. Aside from early compilation problems, the OSKit was a highly useful and relatively trouble-free component of ML/OS[9].

4.2.1 Nonblocking I/O

One thing that was added to the OSKit was an improved keyboard input function. The simple `direct_cons_getchar()` function provided by the OSKit would, when called, loop until a key was available from the hardware keyboard buffer, then return the value of the key. Having the entire operating system go into a tight loop while waiting for keyboard input is not desirable behavior, especially if it is supporting concurrency and there are other threads waiting to run.

In preparation for supporting concurrency, the keyboard input function was split

into two C functions. One function, `direct_cons_trygetchar()`, is a polling input function that immediately returns the next character in the keyboard buffer or -1 if there is no input available. The second function is a replacement for the original `direct_cons_getchar()` that uses the polling function to read characters. The advantage of the polling version of the keyboard input function is that it returns without looping, and the system doesn't necessarily block while waiting for keyboard input.

Granted, the new `direct_cons_getchar()` is not much more useful than the original, but it gives the system the option of using `direct_cons_trygetchar()` instead to immediately return -1 if there is no input available instead of waiting indefinitely for an input character. The introduction of interrupt-driven I/O for concurrency (described later) is even more efficient and eliminates the need for any of the original keyboard input functions.

4.3 Putting It All Together

Given the OS facilities provided by the OSKit and the environment that SML/NJ expects to run in, the job of ML/OS is to integrate both into a functional whole. This is primarily a task of smoothing out differences between interfaces and filling in functionality gaps. In addition to just merging SML/NJ and the OSKit, ML/OS also provides non-blocking and interrupt-driven I/O.

4.3.1 Heap Conversion

The SML/NJ run-time system expects to read the initial heap image from disk, but ML/OS has no filesystem. To enable the reading of the heap the ML heap image was linked directly into the ML/OS kernel. This was done by developing a utility that takes an arbitrary data file and converts it to an a.out-format object file that contains the data in the original file and defines a linker symbol to be the address of this data. When linked with other object files, the resulting executable can access the data from the original file as an array of characters from the converted object

Type	Examples
SML exceptions & interrupts	Div, Overflow, Subscript
Unix synchronous & asynchronous signals	SIGFPE, SIGALRM, SIGHUP
x86 exceptions & interrupts	div0, timer, overflow, double fault

Figure 1: Interrupt & exception types

file. An unmodified SML/NJ was used to produce a heap image which was then converted to an object file and linked with the ML/OS run-time system to produce the final, bootable ML/OS kernel.

SML/NJ was modified to read directly from the array in physical memory, instead of trying to read the initial heap image from disk. Linking the heap image directly into the kernel eliminated the need to develop a filesystem for ML/OS, which would have been a significant research effort by itself. Later, the heap image was separated from the run-time system and loaded as a boot module by GRUB. This made development even more convenient, as the 65KB run-time system could be repeatedly recompiled without having to also link the 10MB heap image every time.

4.3.2 Trap/Signal Interface

One thing that is not provided by the OSKit's minimal C library is a signal facility. However, the SML/NJ run-time system is written to deal with exceptional circumstances such as division by zero and integer overflow by using Unix signal handlers that invoke the SML/NJ exception system. At the interface between SML/NJ and the OSKit, ML/OS needs to resolve the differences between what SML/NJ expects and what the OSKit provides. Figure 1 illustrates the various types of interrupts and exceptions that ML/OS must deal with.

The OSKit's kernel-support library provides functions that interact with the hardware at a low level. It deals with machine interrupts and exceptions, and it provides direct access to the x86's Interrupt Descriptor Table (IDT), which controls how hardware interrupts are handled. The OSKit uses the x86 hardware's ability to jump to different addresses on interrupts and provides the ability to call different

C functions on interrupts. It installs one handler in the IDT for each hardware exception. Each handler, written in x86 assembly language[8], pushes its exception number onto the stack[1] and jumps to a common handler. The handler fetches a C function from a table, indexed by the exception number, and transfers control to it using the C function-call linkage. By default, all hardware exceptions result in a call to `trap_dump_panic()` and subsequent shutdown by the OSKit. Optionally, the OS implementor can install handlers that call predefined C functions for more graceful handling of interrupts.

SML/NJ, on the other hand, expects to be run in a Unix (or Windows) environment with full signal generation and delivery facilities. It uses `signal()` or `sigaction()` to install C signal handlers that interact with ML to properly deal with exceptions. The SML/NJ run-time system is designed to map Unix signals to SML interrupts and exceptions. For asynchronous interrupts, it installs signal handlers that adjust the heap pointer as described earlier. The difference between synchronous exceptions and asynchronous interrupts is that interrupts require the run-time system to construct a continuation before entering the signal handler, so the interrupted computation can be resumed if desired after the handler completes.

ML/OS manages these different interfaces by registering an exception handler with the OSKit that creates a "signal" if possible and delivers it to SML/NJ if appropriate. ML/OS installs a default handler, `MLOSTrapHandler()`, for exceptions. The handler is a C function that maps hardware exceptions to Unix signal values, where possible. These values don't represent actual signals because the OSKit doesn't support signals, but they are presented to SML/NJ as if the equivalent Unix signal had been generated. Currently, only the divide by zero, overflow, and debug exceptions are handled by ML/OS. The first two are passed to the SML/NJ run-time system by the ML/OS, and the debug exception is passed to a debugging function; others such as exception 6 ("invalid opcode") and exception 8 ("double fault") are

[1]This is a bounded-length stack used by the C code in the OSKit and the SML/NJ run-time system. ML/OS as a whole runs "stackless" because all ML procedure frames are allocated on the ML heap.

ML/OS Layer	Requires	Provides
CML	ML interrupts	concurrency
SML/NJ run-time	Unix signals with C handlers	ML interrupt & exception system
ML/OS interface code	x86 interrupts & exceptions with C handlers	"Unix signals" with C handlers
OSKit	hardware	x86 interrupts & exceptions with C handlers
hardware		x86 interrupts & exceptions

Figure 2: Interrupt & exception handling in ML/OS

indicative of more serious problems and cannot be mapped to a valid SML/NJ signal. The remaining exceptions are unhandled and cause the OSKit to panic and shut down. Figure 2 summarizes the roles of the various parts of ML/OS when dealing with interrupts and exceptions.

Remote debugging is achieved via exception 1. If debugging is enabled, the ML/OS exception handler passes the exception to the gdb handler, `gdb_trap()`, before doing anything else. The gdb handler is a C function that is capable of using the gdb serial debugging protocol to send debugging information over the serial port. Once done, the gdb handler returns a success code, and the ML/OS handler does not need to do any further work. If debugging is not enabled, the gdb handler returns a failure code, and the exception is passed like any other exception to the rest of the ML/OS exception handler, which attempts to translate the exception number into an equivalent signal for SML/NJ.

4.3.3 Alarm Clock Facility

Implementing preemptive concurrency requires a way of setting a system timer and acting when the timer runs out. In Unix, this is done by installing a handler for the SIGALRM signal and setting a system alarm clock with the `setitimer()` system call. For ML/OS, instead of implementing general signals, a simplified alarm-clock mechanism was implemented.

The OSKit provides a way to install C functions as handlers for hardware interrupts, so ML/OS installs a handler, `Tick()`, for the timer interrupt, which happens every 100 ms. The handler, when called, updates a global system clock and a private internal timer. If there is an alarm clock active, it checks to see if the internal timer has run out, and if so calls an auxiliary function. This second function emulates the SML/NJ signal handler, which enqueues the "signal," sets various ML-related state variables, and adjusts the limit pointer if necessary to alert ML of a pending signal.

To set the alarm clock, the SML/NJ run-time system calls the `setitimer()` function from the smlnj-signals C library. This alarm-clock mechanism gives ML/OS a coarse-grained timing mechanism and a way to deliver the equivalent of a SIGALRM signal to SML/NJ. By installing an interrupt handler for the hardware clock interrupt and interfacing it directly to the SML/NJ run-time system, ML/OS bypasses the levels of abstraction that are present in a standard Unix implementation of signal-based alarm clocks.

4.3.4 Interrupt-Driven I/O

Interrupt-driven input was implemented to improve the efficiency of ML/OS; instead of polling the keyboard buffer for input once every scheduling quantum, the system is notified whenever input becomes available. A function similar to the original `direct_cons_getchar()` was written and installed as the OSKit handler for a hardware-generated keyboard interrupt. For ML/OS, interrupt-based I/O acts on two levels, both driven by the installed handler.

At the kernel level, a data structure is maintained to hold keyboard input. This software input buffer holds character codes for keys that have been entered at the keyboard as well as their shift, control, and alt states (i.e., "ctrl-c" is stored differently than "c"). The installed handler inserts characters into this buffer, and `direct_cons_getchar()` was replaced with a function that simply returns either the first character in the buffer or a return value denoting an empty buffer.

At the ML level, the installed handler calls an ML function, `IOInterrupt()`, in the SML/NJ run-time system similar to the one that is called when a system timer

expires. This function delivers a simulated SIGUSR1 signal to SML/NJ, indicating that input has become available. An ML signal handler can be installed to respond to the I/O notification and deal with the input appropriately; if there is input, it will be invoked when program execution reaches the next basic-block boundary.

4.4 CML

ML/OS uses Concurrent ML (CML) as its basic concurrency mechanism. Concurrent ML is a module that provides concurrency in SML/NJ. CML programs are composed of threads, which perform sequential evaluation of ML expressions. Threads are implemented using the first-class continuations of SML/NJ, and preemption is implemented using its asynchronous interrupt system. Synchronous communication and synchronization between threads is done by sending messages via data structures called channels. In addition to providing CML, ML/OS adds support for nonblocking and interrupt-driven I/O.

CML threads communicate via typed channels, which are the standard means of inter-thread communication. Channels are created with the `channel()` function, and communication is performed using `accept` and `send`. Channel transmission is synchronous, so threads block on `accept` and `send` calls until a message arrives or another thread is ready to accept a sent message, respectively.

Threads in CML can also synchronize on an event or a nondeterministic choice of events using the `sync` function. Events are first-class values in CML, and different types of events are implemented by different modules within CML. There are channel send and receive events, timeout events, I/O events, degenerate events (`always`, `never`), and ways to manipulate events (`choose`, `wrap`) to make them more useful. The specific semantics of the various events are described in the CML manual.[23]

Figure 3 shows a sample CML program. The function `proc()` creates a channel and spawns a thread that sends two messages over the channel. Since communication is synchronous, the spawned thread blocks until the original thread calls `CML.recv`. After the second call to `recv`, both threads finish.

At the core of CML is the scheduler, which manages threads by setting an alarm

```
fun proc () = let
          val ch = CML.channel()
          in
            CML.spawn (fn () => (
                CML.send(ch, "hi\n");
                CML.send(ch, "bye\n")));
            Debug.sayDebug (CML.recv ch);
            Debug.sayDebug (CML.recv ch)
          end

fun test () = RunCML.doit (proc, SOME(Time.fromSeconds 1))
```

Figure 3: Example CML function

clock timer to regulate time quanta and by maintaining a run queue of threads ready to run. Threads are represented and stored on the various scheduler queues as their continuations in combination with auxiliary information such as their unique thread ID. When the current time quantum has expired or the current thread blocks, an ML trap handler saves the interrupted computation as a continuation and passes the thread information to the scheduler. The use of continuations for representing threads means that resuming a thread is done by simply throwing to the stored continuation, and suspending a thread is done by saving the current continuation and throwing to the `schedulerHook` continuation.

The timeout manager deals with threads that are blocked and waiting on a timeout event. When a thread blocks on a timeout, the scheduler moves it to the sorted timeout queue. After each time quantum or when CML has nothing else to do (i.e., when `pause()` is called), the timeout queue is checked to determine if any threads can proceed. If so, the eligible threads are moved to the run queue. The timeout manager is accessed via the `pollTime()` function, which does the checking and enqueueing of ready threads.

Similarly, the I/O manager handles threads that are blocked while waiting for input or output events. After each time quantum, the `pollIO()` function is called to determine whether or not any threads can be moved to the run queue. The

functionality of `pollIO()` was later moved to an interrupt handler when interrupt-driven I/O was added to ML/OS.

Porting CML to ML/OS was relatively straightforward. In fact, it compiled without any major modifications other than added debugging output. Adding interrupt-driven I/O to CML involved modifying the OSKit's keyboard-interrupt handler to emulate the alarm clock interface, so that a simulated SIGUSR1 signal is delivered to SML/NJ when a key is pressed. Registering an ML I/O handler provided the means to call `pollIO()` when and only when a key is pressed. When ML/OS boots, it automatically calls `RunCML.doit()`, which starts the scheduler and a predefined top-level function.

4.5 Application: Session Manager

After ML/OS was implemented, an application was written to demonstrate its capabilities. Since the only input and output methods were from the keyboard and to the screen, respectively, there was not much choice in the kinds of applications that could be usefully implemented. If networking or a graphical user interface are added in the future, other more complex applications could be written, such as a World Wide Web server or a windowing system. For this project, a simple session manager was written to demonstrate the functionality of ML/OS.

A session manager, as the name implies, allows the user to interact with and control several "sessions" simultaneously. In the ML/OS session manager, each session is represented by a CML thread. A session can be a read-eval-print loop, an interactive thread, or a thread that produces output. The session manager demonstrates how ML/OS can manage the input and output needs of several different, concurrent threads.

The behavior of the ML/OS session manager is loosely modeled after that of other existing session managers. One example of a process that manages different sessions is a Unix shell. A shell allows a user to do several things. The user can start or stop a process by typing a command name at the shell prompt or hitting a "kill" character such as ctrl-C, respectively. Processes can be suspended and resumed via

appropriate command keys. Also, the user can list all active processes. At any given point, the user can interact with a maximum of one process. Processes are referred to by their unique process ID or by a shell-defined naming convention (e.g., `42`, `%1`, `%2`, `%emacs`, etc).

Another example of a session manager is the method of switching between virtual consoles in Linux and similar operating systems. By using various keystrokes, the user can switch between a number of independent virtual consoles. There is a similar notion here of a "suspended" session; if a process is waiting for input on one virtual console and the user is in another one, it waits/blocks until input becomes available.

The ML/OS session manager manages each session as a separate thread and controls how keyboard input is received by each process. Similarly to a Unix shell, all of the threads send output to the screen simultaneously, but the user can only send keyboard input to one thread at a time. Managing keyboard input is done by assigning a CML channel to each thread on its creation and having the session manager pass keyboard input to the "foreground thread" via these channels. Each thread is called with its channel as an argument, so it knows where to get input.

At any point, the user can switch to manager mode by hitting a predefined "escape key," similar to the suspend key in a Unix shell. In the manager mode, the user can list the threads, start new threads, designate another thread as the foreground thread, or shut down the manager altogether. The code for the session manager can be found in Appendix A.

The session manager consists of a thread list, a main loop, a job list, a thread ID list, and various utility functions. The thread list is an associative list of predefined threads and their names. While the manager is running, the user can spawn a thread from the thread list, and the associated name of the thread is used for identification when listing active threads.

The job list and thread ID lists are a data structures used by the session manager to keep track of currently active threads. The job list is a list of thread names and their corresponding CML channels, with a type of `(string * char CML.chan) list ref`. The session manager uses the job list to list threads and redirect user

input to the current thread. The thread ID list is a list of thread IDs which is used by the session manager to synchronize on and properly handle thread death. Newly spawned threads are added to the job list, and their thread IDs are added to the thread ID list.

The main loop is a thread that repeatedly blocks, waiting for user input or thread death. After an event happens (`KEYPRESS` or `THREAD_DEATH`), the main loop performs the appropriate action. If there is a thread death, it removes the dead thread from the job and thread ID lists. Otherwise, it acts according to the current session manager state, which is either `MANAGER` or `THREAD of int`. If the current state is `MANAGER`, the key is interpreted as a command for the manager thread. The user can list threads, spawn a thread, make a thread current, or shut down the session manager. If the current state is `THREAD(n)`, the key is sent to the n-th thread. The user can switch between the `MANAGER` and `THREAD` states via the "escape" and various "foreground" keys. When the session manager is shut down, CML halts and all the spawned threads are killed.

4.5.1 Limitations

The session manager is not very advanced, and there are limitations to its capabilities. Currently, only threads from the predefined thread list can be spawned, though an appropriately modified real-eval-print loop (modified to take input from a CML channel) could be defined, put on the thread list, and used to dynamically spawn new threads.

5 Results

ML/OS exhibits many of the expected benefits of a tightly coupled operating system and compiler, even though it is not a "production-quality" OS. The elimination of any distinction between kernel threads and user threads reduces the overhead of system calls. The use of CML as a system-wide concurrency mechanism takes advantage of the inexpensive first-class continuations of the SML/NJ compiler. Also, ML/OS inherits all the advanced language features of ML.

5.1 Advanced Language Features

The use of SML/NJ as the compiler in ML/OS provides both the operating system and the user with all the benefits of the ML programming language and SML/NJ, such as higher-order procedures and garbage collection. All the type-safety and polymorphism of ML is available, as is the ability to directly create, manipulate, and save continuations. The presence of these features in ML/OS allows it to examine the benefits and drawbacks of using high-level programming language concepts for low-level system programming.

5.2 Streamlining the OS

One of the major benefits of moving the compiler into the kernel is that it reduces much of the overhead involved with running user applications in a typical operating system. The safety property of ML allows user threads and system threads to coexist in the same address space without any danger of unintentionally interfering with each other. The single address space eliminates the need to do costly context switches to handle system calls or perform other kernel functions.

Since there is no intermediate layer between an application and the hardware, events such as hardware timer and keyboard interrupts can be handled more efficiently than in a standard operating system. Since the OSKit doesn't provide a generic signal mechanism, hardware interrupts are forwarded to SML/NJ after a relatively short function call, instead of having the operating system save the context

and switch into kernel mode to process the interrupt. The way interrupts are handled allows threads to receive and respond to notification of these significant system events with a minimum of interference and system overhead.

5.3 Continuations And Concurrency

Continuations are used as the system-wide concurrency mechanism in ML/OS for a number of reasons. Continuations are first-class values in SML/NJ, and they are a natural way of expressing concurrency. More importantly, the creation of continuations is inexpensive, allowing for efficient switching between threads of execution. Using a relatively high-level construct such as a continuation to implement a low-level OS function such as concurrency demonstrates some of the benefits of close interaction between the compiler, the operating system, and an advanced language.

Continuations can be explicitly created, manipulated, and saved in SML/NJ and ML/OS. They are an easy way of encapsulating the current state; instead of manually saving the program state and switching contexts, the ML programmer can just save the current continuation and throw to another saved continuation. Since all memory allocation is done on the heap, there is no stack to save, and memory management issues are left to the garbage collector. As a result, continuations can be created and used efficiently and cheaply enough to be effectively used for concurrency in ML/OS.

The use of continuations for concurrency in ML/OS is interesting because it demonstrates one way in which the compiler and operating system closely interact in ML/OS. Continuations are supported at all levels of ML/OS, from the CPS representation used by the SML/NJ run-time system to the use of continuations to represent threads in the implementation of CML. By implementing concurrency in a language more advanced than assembly or C, ML/OS shows that it is possible to use higher-order procedures, type safety, and continuations to implement low-level operating system mechanisms. By using CML as its concurrency mechanism, ML/OS takes what was originally designed as a module for a compiler and pushes it down into one of the lowest and most important functions of an operating system.

5.4 Drawbacks/Difficulties

5.4.1 SML/NJ

Working with Standard ML of New Jersey proved to be much more difficult than expected. The sheer amount of code, sparse documentation, and periodic releases of new versions combined to made it difficult to effectively integrate SML/NJ into ML/OS. The assistance provided by Lorenz Huelsbergen and John Reppy, of the SML/NJ development team, was essential to understanding the run-time system and effectively embedding it into ML/OS.

Standard ML of New Jersey is a large software package. There are about 144,000 lines of code in over 1,000 source files. Even with a well laid-out directory structure and comments in the code, it was often difficult to determine exactly what a given function was doing and how it fit into the larger structure of the system.

Since the last official release of SML/NJ was in 1993, the development releases were not very well-documented. The paucity of documentation isn't surprising, considering that these were intermediate development releases and that the SML/NJ team was probably busier programming than documenting. However, the lack of documentation (and obsolescence of old documentation) made it almost always necessary to read the appropriate source files in order to determine the type of a function or the use of a particular module.

The frequent release cycle was sometimes difficult to deal with. Granted, accepting rapid change is part of the decision to use development releases, but sometimes a new release would fix a serious bug and would have to be integrated into ML/OS. Module interfaces and function names would sometimes change between releases, requiring the modification of already-written ML/OS code. The frequency of releases would not have been so bad if the locations of important functions such as callcc had remained stable.

Also, debugging ML programs was awkward at best. Traditional C debuggers could not be used for finding ML bugs since the run-time system cannot be single-stepped while ML code is executing. ML's type-checking catches many errors, but

run-time debugging is restricted to inserted debugging output. Overall, the lack of debugging tools in SML/NJ was one of its biggest drawbacks.

5.4.2 ML/OS

The main drawback of ML/OS in its current state is the fact that it is still a prototype system and is missing a number of important OS components. Devices other than the keyboard and screen are not supported; currently, input is only received from the keyboard and output is only sent to the screen or a serial port. There is no networking or filesystem support, though work is being done towards adding networking in the future.

Potential problems may also arise when additional OS features are added to ML/OS; certain algorithms may have to be reevaluated or rewritten entirely. Techniques developed for user programs don't always work at the operating-system level or at the hardware level. For example, the way SML/NJ handles asynchronous signals, by deferring them until a basic-block boundary, would no longer function correctly with the addition of virtual memory. If part of the current basic block has been swapped out, the operating system can no longer simply defer the handling of a page fault until the end of the block, because it will never get that far unless the needed page is retrieved from disk.

6 Conclusions

ML/OS is interesting for a number of reasons. It is an implementation of an advanced language embedded into an operating system. It demonstrates the benefits of using such a language to do system-level programming in a type-safe manner. ML/OS shows how a continuation-based thread model can be used as the base concurrency mechanism for an operating system. It is an example of how existing components can be used to build an operating system more efficiently and productively than attempting to implement one from scratch. Finally, it incorporates all these ideas into a single operating system, combining ideas from compiler, programming-language, and OS research.

6.1 Future Directions

There are a number of directions for further research using ML/OS. The addition of a filesystem and networking would make it an ideal platform for experiments in implementing device drivers and network protocols using the features of an advanced language. Also, development of a persistent store for ML/OS would be an interesting attempt to match an advanced filesystem with an advanced language.

A ML/OS Session Manager

```
(* Session Manager *)

(* command specifics:
 *
 * '0' to bring up session manager
 * '1'-'9' to fg thread 1-9
 * 'l' to list jobs in manager
 * 'a','b',... to start a thread
 * '`' to shutdown everything
 *)

structure PF = Posix.FileSys;
structure PIO = Posix.IO;
val pd = OS.IO.pollDesc(PF.fdToIOD(PF.stdin))
val pi = OS.IO.pollIn(case pd of SOME(foo) => foo);
val stdInReadEvt = OS.IO.pollEvt([pi]);

datatype manager_state = MANAGER | THREAD of int

datatype key_type =
    CMD_MANAGER                          (* == CMD_BG *)
  | CMD_SHUTDOWN
  | CMD_LIST_JOBS
  | CMD_NEW_JOB of int
  | CMD_FG of int
  | CHAR of char

datatype event_type =
    KEYPRESS
  | THREAD_DEATH of CML.thread_id
```

```
(* example threads: *)

fun thread1 channel =
    (* prints out "loop" 5000 times *)
    let
        fun loop 0 = print "done\n"
          | loop n = (print "loop\n"; loop (n-1))
    in
        loop 5000
    end

fun thread2 channel =
    (* repeats every input char 75 times, except q *)
    let
        fun rptch (0, ch) = print "\n"
          | rptch (n, ch) = (print (Char.toString ch);
                             rptch (n-1, ch))

        fun rptstr (0, str) = print "\n"
          | rptstr (n, str) = (print str; rptstr (n-1, str))

        fun foo #"q" = rptstr (50, "q rules! ")
          | foo ch = rptch (75, ch)

        fun loop () =
            (foo (CML.recv(channel)); (* wait for input, repeat it *)
             loop ())
    in
        loop ()
    end

 fun thread3 channel =
    (* echoes input *)
    let
        fun loop () = (print (Char.toString (CML.recv(channel)));
                       loop ())
    in
        loop ()
    end
```

```
fun thread4 channel =
    (* prints out "hack" 300 times *)
    let
        fun loop 0 = print "done\n"
          | loop n = (print "hack\n"; loop (n-1))
    in
        loop 300
    end

fun thread5 channel =
    (* prints out "lose" 300 times *)
    let
        fun loop 0 = print "done\n"
          | loop n = (print "lose\n"; loop (n-1))
    in
        loop 300
    end

val thread_list = [("looploop", thread1),
                   ("repeat75", thread2),
                   ("echo", thread3),
                   ("hackloop", thread4),
                   ("loseloop", thread5)];
```

```
fun SessionManager () = let

  (* state variables *)

  val current_state = ref MANAGER  (* manager state: MANAGER or THREAD(n) *)
  val current_job = ref 0          (* session manager thread id *)

  val job_list: (string * char CML.chan) list ref =
        ref (("dummy", CML.channel()) :: nil)
  val thread_id_list: CML.thread_id list ref =
        ref ((CML.getTid ()) :: nil)
  val sm_tid = hd (!thread_id_list)

  (* utility functions *)

  (* find: 'a list * 'a -> int: returns pos of item in list or -1 *)
  fun find (nil, tid) = ~1
    | find (h :: tl, tid) = (if (h=tid) then 0 else
                                let
                                    val f = find (tl,tid)
                                in
                                    (if (f<0) then f else 1+f)
                                end)

  (* dropnth: 'a list * int -> 'a list: removes the nth element from the list *)
  fun dropnth (nil, n) = raise Subscript
    | dropnth (h :: tl, 0) = tl
    | dropnth (h :: tl, n) = (if (n<0)
                              then raise Subscript
                              else h :: dropnth (tl, n-1))

  (* xlate_key : char -> key_type *)
  fun xlate_key ch =
      (case ch of
           #"0" => CMD_MANAGER
         | #"l" => CMD_LIST_JOBS
         | #"`" => CMD_SHUTDOWN
         | _ => if Char.isDigit(ch) then
                  CMD_FG (Char.ord(ch) - Char.ord(#"0"))
                else if ch >= #"a" andalso ch <= #"f" then
                  CMD_NEW_JOB (Char.ord(ch) - Char.ord(#"a"))
                else
                  CHAR ch
  (* end case *))

  fun listjobs nil = print "SM: CMD_LIST_JOBS: end of list\n"
    | listjobs ((jobname, channel) :: r) =
        (print ("SM: CMD_LIST_JOBS: " ^ jobname ^ "\n"); listjobs r)
```

```
(* main session manager loop *)

fun loop () =
    (print "SM: waiting\n";
     let
       val event =
           CML.sync(CML.choose
                    ((CML.wrap (stdInReadEvt, fn x => KEYPRESS )) ::
                     (map (fn t => (CML.wrap (CML.joinEvt t,
                                              fn foo => THREAD_DEATH t)))
                          (!thread_id_list))))
       val curtid = hd (!thread_id_list)
       val _ = print "SM: something happened: "
     in
       (case event of
          KEYPRESS => (* deal with keypress *)
              let
                val key = Char.chr(Word8.toInt(Word8Vector.sub
                                               (PIO.readVec(PF.stdin, 1),0)))
              in
                (print " KEYPRESS\n";
                 print ("\nSM: read key: " ^  (Char.toString key) ^ "\n");
                 (case !current_state of
                      THREAD(int) =>
                        (print "SM: current_state = THREAD\n";
                         case xlate_key(key) of
                             CMD_MANAGER => (* background the process *)
                                            (print "SM: got CMD_MANAGER\n";
                                             current_state := MANAGER)
                           | _ => let
                                    val (tname,tch) =
                                        List.nth(!job_list, !current_job)
                                  in
                                    (print ("SM: sending key to " ^
                                            tname ^ "\n");
                                     CML.send (tch, key))
                                  end
                         (* end case xlate_key *))
```

```
          | MANAGER =>
             (print "SM: current_state = MANAGER\n";
              case xlate_key(key) of
                  CMD_SHUTDOWN =>
                      (print "SM: got CMD_SHUTDOWN\n";
                       RunCML.shutdown())
                | CMD_LIST_JOBS =>
                      (print "SM: got CMD_LIST_JOBS\n";
                      listjobs(tl(!job_list)))
                | CMD_NEW_JOB thread_num =>
                      if (thread_num >= List.length(thread_list)) then
                      print "SM: CMD_NEW_JOB: invalid thread\n"
                      else
                          let
                            val newch = CML.channel()
                            val (tname,tproc) =
                                List.nth(thread_list, thread_num)
                          in
                            (print "SM: CMD_NEW_JOB: start\n";
                             job_list := !job_list @ [(tname, newch)];
                             thread_id_list := (!thread_id_list) @
                                    [CML.spawn(fn () =>
                                               ((tproc newch); ()))];
                             current_job := length(!job_list)-1;
                             current_state := THREAD(!current_job);
                             print "SM: CMD_NEW_JOB: done\n")
                          end
                | CMD_FG thread_num =>
                          (print "SM: got CMD_FG\n";
                           (if (thread_num >= List.length(!job_list))
                                then print "SM: CMD_FG: invalid job\n"
                            else
                                (current_job := thread_num;
                                 current_state := THREAD thread_num)))
                | _ => print "SM: error, invalid key\n"
            (* end case xlate_key *))
        (* end case current_state *));
        loop ())
end
```

```
      | THREAD_DEATH tid => (* deal with thread death *)
            let
              val _ = print "THREAD DEATH\n"
              val pos = find (!thread_id_list, tid) (* better be > 0 *)
            in
              (case !current_state of
                   THREAD(i) => (if (pos = i) then
                                      (print "SM: current thread died\n";
                                       current_job := 0;
                                       current_state := MANAGER)
                                 else (if (pos < i) then
                                          (print "SM: adjusting current state\n";
                                           current_job := !current_job - 1;
                                           current_state := THREAD(i-1))
                                       else
                                         ()))
                 | MANAGER => ()
               (* end case current_state *));
               job_list := dropnth (!job_list, pos);
               thread_id_list := dropnth (!thread_id_list, pos);
               print "SM: dead thread removed\n";
               loop ()
            end
 (* end case event *))
 end)
in
    loop ()
end
```

```
(* function to invoke session manager *)

fun RunSessionManager () =
    RunCML.doit (SessionManager, SOME(Time.fromMilliseconds(500)));

(* function to export to heap *)

fun main2 (s:string, sl: string list): OS.Process.status = (
 print "SM: starting\n";
 RunSessionManager ();
 print "SM: done\n";
 OS.Process.success
)
```

References

[1] Andrew W. Appel.
Compiling with Continuations.
Cambridge University Press, 1992.

[2] Andrew W. Appel and David B. MacQueen.
A Standard ML compiler.
Functional Programming Languages and Computer Architecture, Vol. 274, pages 301–324, G. Kahn, editor, Springer-Verlag, 1987.

[3] Andrew W. Appel and David B. MacQueen.
Standard ML of New Jersey.
Third International Symposium on Programming Language Implementation and Logic Programming, pages 1–13, J. Maluszynski and M. Wirsing, editors, Springer-Verlag, New York, 1991.

[4] A. Bawden, G. Greenblatt, J. Holloway, T. F. Knight, D. Moon, and D. Weinreb.
The Lisp Machine.
Artificial Intelligence—An MIT Perspective, Vol. 2, Patrick Winston, editor, The MIT Press, 1979.

[5] Andrew D. Birrell.
An introduction to programming with threads.
Research Report 35, Digital Equipment Corporation Systems Research Center, Palo Alto, California, 1989.

[6] Erich Boleyn.
GRUB – GRand Unified Bootloader.
`http://www.uruk.org/~erich/grub/`

[7] Eric Cooper, Robert Harper, and Peter Lee.
The Fox Project: Advanced development of systems software.
Technical Report CMU-CS-91-178, CMU School of Computer Science, Pittsburgh, Pennsylvania, 1991.

[8] John H. Crawford and Patrick P. Gelsinger.
Programming the 80386.
SYBEX, 1987.

[9] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers.
The Flux OSKit: A substrate for OS and language research.
Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, 1997.

[10] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner.
The Flux OS toolkit: Reusable components for OS implementation.
Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, 1997.

[11] Bryan Ford and Erich Boleyn.
Multiboot standard, Version 0.6.
`http://www.uruk.org/~erich/grub/boot-proposal.html`

[12] M. J. Gordon, A. J. Milner, and C. P. Wadsworth.
Edinburgh LCF.
Springer-Verlag, New York, 1979.

[13] R. D. Greenblatt, T. F. Knight, J. Holloway, D. A. Moon, and D. L. Weinreb.
The Lisp Machine.
Interactive Programming Environments, pages 326–352, D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, McGraw-Hill, New York, 1984.

[14] Robert Harper and Peter Lee.
Advanced languages for systems software: The Fox project in 1994.
Technical Report CMU-CS-94-104, CMU School of Computer Science, Pittsburgh, Pennsylvania, 1994.

[15] Roger E. Kaufman.
A Fortran Coloring Book.
The MIT Press, 1978.

[16] Lucent Technologies Inc.
Inferno: la Commedia Interattiva.
`http://inferno.lucent.com/inferno/infernosum.html`

[17] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen.
The Definition of Standard ML (Revised).
The MIT Press, 1997.

[18] James G. Mitchell, William Maybury, and Richard Sweet.
Mesa language manual. Version 5.0.
Technical Report CSL-79-3, Xerox Corporation Palo Alto Research Center, Palo Alto, California, 1979.

[19] Greg Nelson, editor.
Systems Programming with Modula-3.
Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[20] Larry C. Paulson.
ML for the Working Programmer.
Cambridge University Press, 1990.

[21] James Philbin.
An overview of the Sting operating system.
Proceedings of the 4th NEC Software Conference, 1992.

[22] George Radin.
The 801 minicomputer.
Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, pages 39–47, Palo Alto, California, 1982.

[23] John H. Reppy.
Concurrent programming with events–The Concurrent ML manual.
Technical report, Cornell University Department of Computer Science, Ithaca, New York, 1990.

[24] John H. Reppy.
CML: A higher-order concurrent language.
Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, pages 293–305, 1991.

[25] John H. Reppy.
Asynchronous signals in Standard ML.
Technical report TR 90-1144, Cornell University Department of Computer Science, Ithaca, New York, 1990.

[26] John H. Reppy.
First-class synchronous operations in Standard ML.
Technical report TR 89-1068, Cornell University Department of Computer Science, Ithaca, New York, 1989.

[27] Emin Gun Sirer, Stefan Savage, Przemyslaw Pardyak, Greg DeFouw, Mary Ann Alapat, and Brian Bershad.
Writing an operating system using Modula-3.
Appeared in the *Workshop on Compiler Support for System Software*, 1996.

[28] Guy L. Steele, Jr.
RABBIT: A Compiler for Scheme (A Study in Compiler Optimization).
Master's thesis, Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, 1978.

[29] W. Richard Stevens.
Advanced Programming in the Unix Environment.
Addison-Wesley, 1992.