HEWLETT PACKARD

Intraprocess Concurrency Under UNIX

Scott B. Marovich
Software and Systems Laboratory
HPL-91-02
March, 1991

concurrent/parallel
software,HP-UX,
IEEE 1003.4,
operating systems,
POSIX, threads, UNIX

This paper describes a coroutine implementation of the *Threads Extension for Portable Operating Systems,* defined in proposed IEEE Standard No. 1003.4. Packaged as an HP-UX subroutine library, it allows multiple flows of control in a single program and provides inexpensive, simulated concurrency under a traditional, uniprocessor operating-system. We analyze problems of designing such a mechanism for UNIX-style operating-systems and give performance measurements for two HP computers. The presentation is for advanced programmers who wish to use this software or build similar facilities, and it assumes knowledge of UNIX system programming.

Contents

1 Introduction 1
2 Design Problems and their Solutions 2
2.1 Execution Stacks 3
2.2 Operating-System Errors 4
2.3 Asynchrony and Process Suspension 5
2.4 Signals and Timing 8
2.5 Packaging Issues 10
2.6 Application Debugging 13
2.7 Problems of the IEEE Proposal 15
2.8 Open Problems 17
3 Performance 19
3.1 Measurement Conditions 20
3.2 Process-Context Switching Time 21
3.3 IEEE 1003.4 Metrics 21
4 Discussion and Conclusions 31
5 Acknowledgements 32
6 References 33
Appendix A: Operating-System Stub Replacements 37
Appendix B: Coroutine-Switching Marginal Costs 39
Appendix C: Process-Context Switching Time 43

List of Tables

Table 1. Process-Context Switching Time (μS) 21
Table 2. Yield Time, Not Busy (μS) 25
Table 3. Yield Time, Busy 25
Table 4. Scheduler Overhead (μS) 26
Table 5. Mutex Lock/Unlock Time, No Contention (μS) 27
Table 6. Mutex Lock/Unlock Time, Long-Term Contention 27
Table 7. Lock Manipulation Time (μS) 27
Table 8. Condition Signal Time (μS) 28
Table 9. Condition Broadcast Time (μS) 29
Table 10. Condition Signal Time, Outside Critical Section 29
Table 11. Condition Signal Time, Inside Critical Section 29
Table 12. Event Manipulation Time (μS) 29
Table 13. Event + Lock Manipulation Time (μS) 30
Table 14. Condition Signal Time, Outside Critical Section 30
Table 15. Condition Signal Time, Inside Critical Section 30
Table 16. Event + Lock Manipulation Time (μS) 31
Table 17. Process-Context Switching Time (μS) 44

1 Introduction

As computer users seek faster performance by distributing parts of an application among several central processing units (CPUs), exploiting opportunities for concurrency has become an important principle for organizing software. As a result, there is also an interest in low-cost simulations of concurrency using a single CPU under a traditional operating-system. It is often thought that conventional multiprogramming is too costly for this purpose, due to the time required for process-context switching [1], so there has been much recent discussion of what are called *light weight* processes: multiple, independent flows of activity within a task that share some of its hardware-state information, such as the content of memory. By reducing the amount of information that distinguishes each flow, which must be saved in memory and restored to the CPU whenever one is interrupted by another, it is thought that shared hardware can be multiplexed among such flows much faster than among traditional operating-system processes. How best to do this is a topic of much current research.

The MACH operating-system treats a flow of activity (which its literature calls a *thread of control,* or just a *thread*) as a fundamental abstraction that is represented by internal data structures and manipulated by new operating-system services [26]. We use MACH terminology and refer to some of its ideas, but we describe an alternate implementation using coroutines, managed by a subroutine library. Within an application program, our mechanism provides an inexpensive simulation of concurrency on a uniprocessor computer, using an unmodified, traditional operating-system. Our library was built and measured on HP 9000 Series 300 and 800 computers, under Version 7.0 of HP-UX™, Hewlett-Packard Company's version of the UNIX™ operating-system, using techniques that apply broadly to other versions of UNIX. It presents the same application program interface (API) that is proposed for IEEE Standard No. 1003.4 [15][1].

Using coroutines to simulate concurrency is not a new idea [10][2], but using an industry-standard API is. Not all details of the IEEE proposal are final, but because many vendors seem likely to support it, understanding its provisions is important. The work described here is a trial implementation to assess construction problems, to

---

™ HP-UX is a trade mark of Hewlett-Packard Company. UNIX is a trade mark of UNIX System Laboratories, Inc.

1. The most recent version of the proposed thread facility at the time of writing is a supplementary chapter (or an appendix?), distributed as a separate, unapproved draft, numbered P1003.4a and entitled: *Threads Extensions for Portable Operating Systems,* Draft 4 (10 August 1990). Assuming eventual incorporation into the full document, and despite possible ambiguity at present, we use number 1003.4 to identify its thread facility without expecting confusion, since we discuss no other part of the standard.

2. The subject has an extensive history and literature, covering parallel computation and discrete-time simulation for many programming languages and operating-systems. Recent examples for the *C* language under UNIX include [3,4,6,7,8,16 (or 25, Ch. 6)].

compare the performance of our system with that of ordinary multiprogramming, and to provide Hewlett-Packard software developers with a way to experiment with the proposed facilities.

Section 2 analyzes some problems of designing a general-purpose, library-based coroutine mechanism for UNIX and how this implementation solves them. Many of these techniques have undoubtedly been discovered before, but they seem to be inadequately documented in earlier literature. Section 3 gives performance measurements, including metrics required by the IEEE proposal, and it compares the speed of thread-context switching to that of ordinary process-context switching. Section 4 discusses the implications of our results.

2 Design Problems and their Solutions

Our coroutine system was built for HP 9000 Series 370 and 835 computers, two engineering workstations with different architectures but comparable performance[3]. Optional features of the IEEE proposal that it supports include: a thread stack size attribute, real-time scheduling, and *per* process signals. However, reentrant versions of *C* Library subroutines are not yet supported (*cf.* §2.8.1). Our code is packaged as an augmented copy of the *C* Library, and it is initialized by a modified version of the library's start-up module, **crt0**. The package also replaces about forty system-call interface subroutines, both in the *C* Library and one for NetIPC™, an optional X.25 software product.

The system runs under HP-UX Version 7.0, a combination of A.T.&T. UNIX System V, Release 3, with features of 4.2BSD and 4.3BSD UNIX, a fact which substantially increases both the number of operating-system services that the coroutine package must emulate (*cf.* §2.3) and the implementation effort. The IEEE proposal defines new POSIX operating-system services altered semantics when an application has more than one thread, that HP-UX does not support but to which it provides equivalent functionality for single-thread programs in a different way. In these cases, we try to extend HP-UX functionality in the spirit of POSIX.

Our package contains about one-hundred modules. Stubs and other machine-dependent subroutines are coded in assembly language, while the rest are coded in *C*. The total implementation effort for two computer systems described here was about three engineer-months.

The next sections present additional details while they describe the project's chief design problems and their solutions.

---

3. Series 835 is about twice as fast as Series 370 in CPU-intensive, standard bench marks but it is only a little faster in memory-intensive programs.

™ NetIPC is a trade mark of Hewlett-Packard Company.

2.1 Execution Stacks

Each thread of control, or coroutine, in a process requires its own "push-down stack" for private arguments, local variables, and machine-state information that must be saved when its execution is interrupted, so an implementation must include a memory allocation mechanism for these stacks. The mechanism should be transparent enough that a programmer need not worry about stack overflow nor be called upon to predict a stack's maximum size, but it is difficult to satisfy this requirement under UNIX.

Ordinarily, a UNIX process has a single execution stack at a fixed virtual-memory address that holds contiguous procedure-activation records (stack *frames*). The operating-system automatically detects stack overflow and extends this memory region accordingly. The fact that UNIX can treat only a single region in this way becomes a problem if a process has more than one thread of control, and there are two obvious solutions.

The most convenient and reliable alternative uses the automatic mechanism by time-sharing the memory region: A coroutine implementation copies the content of a thread's stack to the shared region before it resumes a thread's execution, and it copies the region's content back to a unique "save area" when it suspends the thread's execution. This method is robust and makes stack management transparent to an application program, but thread-context switching can be slow; in fact, the time is unbounded, because the amount of memory copied—twice!—is proportional to the stack's size at that moment.

A faster, but risky alternative is to ignore the operating-system's automatic mechanism by allocating several distinct memory regions for thread stacks, and to choose among them by reloading a "stack pointer" CPU register during each thread-context switch. This is faster than copying a stack's content, but overflow cannot easily be detected, so each region must be large enough to prevent it. Moreover, a stack cannot easily be relocated were enlargement required, because it may contain addresses of other variables in the region, which would require detection and adjustment, so it is a practical necessity to specify fixed region sizes in advance. The IEEE proposal includes a mechanism for programmers to optionally do so, but they usually do not have enough information for accurate estimates, so an implementation must pick a reasonable default size. If an enormous size is chosen in order to minimize the risk of disaster, a program that creates a large number of threads can exhaust the amount of secondary storage available for virtual-memory paging, since UNIX employs the "banker's algorithm" [12] to preallocate space for each process' main-memory data.

MACH solves this problem using a *lazy evaluation* strategy [27]: Assigning main memory and secondary storage to virtual-memory pages is deferred until a process tries to use them, whereupon it is done in small increments, upon demand[4]. When

only a little stack space is used, many large memory-address ranges can be reserved at low cost. Unfortunately, this solution is unavailable under most versions of UNIX.

A different problem arises if threads are used to implement programming language features like ADA *tasks* [2][5], which may share variables that are declared in a common, non-global lexical scope, and which a compiler usually allocates on a thread's execution stack. In most block-structured languages, these variables are accessed using a display or a chain of static-environment pointers [20], which refers to memory locations in a thread's stack but is not usually modified during thread-context switching. If stack frames are allocated contiguously (as in most versions of UNIX, for fast subroutine calls), storage for these variables is duplicated in each sharing thread's stack. This is harmless—if wasteful—when stacks occupy distinct, non-copied memory regions, since a display or static-environment pointer can refer to only one of the copies. But if a single memory region is multiplexed among stacks and a thread updates a shared variable, the next thread-context switch will copy the new value to that thread's "save area" without propagating it to those of other sharing threads. If any of the latter then reads the variable, an obsolete copy of the value will be retrieved, violating the semantics of "shared" memory! Noncontiguous stack-frame allocation is an alternate solution to this problem, but few versions of UNIX provide it.

Hence, the choice of stack allocation strategy under UNIX is a trade-off between between convenience with robustness, and speed. The author built two versions of the coroutine package described here in order to compare execution times using (non)copied stacks; the results are described in §3.

2.2 Operating-System Errors

When an application program requests an operating-system service and an error occurs, UNIX returns a code number in a global variable named **errno**. If a process contains several threads that request services concurrently, **errno** effectively becomes a shared variable; however, the implementation must still arrange that error codes are somehow routed to the proper threads. The IEEE proposal suggests several solutions:

- Change all operating-system services to return the code number using either a signal or an extra argument, passed "by reference" to each service. Except perhaps in new UNIX implementations, downward-compatibility requirements make this unreasonable.

---

4. This can lead to deadlocks if a process tries to use more memory than is available at that moment, but MACH's builders assume that this circumstance is rare, so they deliberately sacrifice robustness for speed.

5. The IEEE proposal does not require such support, but it strongly encourages it.

- Reimplement **errno** as an explicit, *per* thread variable. The proposal recognizes that this requires special support from a compiler, linker, and/or virtual-memory system.
- If application programs are coded in *C*, use the compiler's *cpp*(1)[6] macro preprocessor to redefine **errno** as a function which reads an implicit, *per* thread variable. This is a language-specific kludge, but it is probably the most practical solution for a multiprocessor computer.

In a uniprocessor simulation of concurrency, there is a much simpler solution: during a thread-context switch, just save the value of a global **errno** variable on the suspended thread's stack and restore an earlier value from the resumed thread's stack. This is like the LISP implementation technique called *shallow binding* [5], and it is the method we use.

Another, less serious problem is that the IEEE proposal introduces a new POSIX error code that is not supported by HP-UX. Named **ENOTSUP**, it denotes an attempt to use a defined, optional facility that is unimplemented, unconfigured, temporarily disabled, or otherwise unavailable. To avoid changing too many standard HP-UX files, we temporarily use the header file **<pthread.h>** to assign code number 47, the next code after those defined in **<errno.h>**, the definition's proper location. Any future product based on our work will have a permanent definition in the latter file.

2.3 Asynchrony and Process Suspension

A uniprocessor simulation of concurrency exhibits no speed-up unless it can overlap computation with an activity, such as I/O, that would ordinarily cause a process to be suspended. Because a process can only be suspended by the operating-system in response to a service request, the simulation must change the way in which such a request is made.

The most common cause of suspension is an I/O request, since UNIX I/O is synchronous by default. For instance, a programmer who requests a *read*(2) service ordinarily expects the operation to be finished when control of the CPU returns from the corresponding subroutine call, so if no input is immediately available, the operating-system suspends the program—and may resume another—while waiting for it. UNIX programs can overlap computation by requesting asynchronous I/O, in which the operating-system starts a data transfer but returns control immediately—then is

6. This form of citation identifies a topic in a numbered section of the on-line UNIX *Reference Manual* in the same way as other literature about this operating-system. Most dialects of UNIX are sufficiently alike that any version of the manual may be consulted. Similar generic citations are used throughout this paper.

queried for completion later—but the UNIX mechanism for this is clumsy, so programmers rarely use it.

Other operating-system services also cause a process to be suspended. For instance, *wait*(2) blocks until a subprocess finishes execution, and *pause*(2) blocks until a signal is received (from a process invoking *kill*(2)). HP-UX also includes UNIX System V inter-process communication (IPC) facilities involving semaphores and message queues, whereby one process can be suspended pending certain actions by another. Altogether, some thirty HP-UX service requests can trigger suspension.

But if a process has several threads of control and one of them requests one of these services, we do not wish to suspend the entire process, but only the invoking thread. If the operating-system immediately returns control of the CPU, the process can resume another thread while the service is performed. Thus, from a process' point of view, I/O and similar services should be performed asynchronously and concurrently, but the default behavior presented to each thread should be synchronous[7]. A coroutine simulation must therefore mediate and transform a thread's operating-system requests in order to suspend and resume threads, plus do whatever "bookkeeping" is necessary to preserve this illusion, while avoiding process suspension when it has threads that can execute.

2.3.1 Stub Replacement

Since a coroutine package consists of library subroutines in an application program's virtual-memory address space, which must communicate with the operating-system in the same way as other application code, how can the package alter operating-system requests? Our method of intercepting and transforming these relies on a trick.

Most computers use special machine instructions to enter an operating-system, which can only be coded in assembly language. In order to let calls be written in a higher-level language, the UNIX *C* Library provides a mediating "stub" subroutine for each service, containing the necessary instruction(s). By replacing these stubs, one can intercept and transform all operating-system request except perhaps those written in assembly language, but these are rare under UNIX. Appendix A lists the stubs replaced by our implementation. There are two replacement techniques that make it fairly transparent to an application programmer.

One method uses systematic renaming: For each stub subroutine **S**, the coroutine package includes a replacement stub **S′** with the same parameters, which either transforms a caller's arguments before invoking **S** or performs an alternate action.

7. If a thread explicitly requests asynchronous service, the request should be honored by passing it directly to the operating-system.

Application program references to **S**, references are then systematically changed to **S′**. This is less difficult than it sounds, since programs written in *C*, the most popular language under UNIX, can be automatically transformed using the compiler's *cpp*(1) macro preprocessor. The Concert Multithread™ Architecture (CMA) [8] uses this technique. Its limitations are inherent in the mechanism: Macro calls must conform to the syntactic idiosyncracies of a particular language and compiler technology, text substitutions may occur in unexpected places—causing obscure errors, and mediating subroutines may unexpectedly usurp names that a programmer intends for other purposes. A second problem is inefficiency, since the technique introduces extra subroutine calls between an application program and the library stubs. A third problem is that it demands recompiling all program modules that refer to the candidate stub, which may preclude other subroutine libraries or proprietary code for which no source text is available. On the other hand, this technique is reasonably portable when programs are coded in *C*, and it sometimes allows another library to be reused without modification.

The second method is to change (a copy of) the *C* Library. A coroutine package could theoretically be isolated in a separate library that supplements and overrides the *C* Library, but the UNIX linker makes this inconvenient (*cf.* §2.5.), so merging them is a practical necessity. One difficulty is that stubs written in assembly language are not portable; hence, the same can be said of modifications. The coding of UNIX system-call stubs is sufficiently arcane that there is also a risk of incompatibility if the original and replacement stubs are written by different vendors and the originals, usually undocumented, are poorly understood. However, this technique makes a coroutine package easier to use with a variety of programming languages, it preserves application program efficiency, and because application programs do not need recompilation (they do need relinking), not all modules need be available in source-program form; therefore, our package uses this method.

2.3.2 Simulation Mechanism

The basic mechanism used by our system to schedule asynchronous I/O is the *select*(2) operating-system service, which returns sets of UNIX File Descriptors that are ready for service by an application program and which can optionally suspend the process for a time, pending I/O completion or signal delivery. The package implicitly opens all of a program's files for asynchronous I/O, but it uses a bit vector to note File Descriptors for which synchronous I/O must be simulated. Since *fcntl*(2) requests can alter this data, they too must be intercepted and modified. An internal, "scheduler" subroutine repeatedly polls the operating-system using *select*(2), and it

---

™ Concert Multithread is a trade mark of Digital Equipment Corporation.

either dispatches an executable thread after suspending another, or else it asks that the entire process be suspended until some thread(s) can proceed.

Some synchronous operating-system services are simulated in other ways. The scheduler is periodically interrupted by UNIX *Alarm Clock* (**SIGALRM**) signals[8] for preemptive *round robin* scheduling (*time slicing*), and the package's internal "handler" subroutine then does some extra work:

UNIX System V IPC facilities cannot indicate a need for attention using *select*(2), so a process must query them in other ways. Stubs for these services have a loop that conditionally suspends a client thread on a global "timing queue" so that the operating-system is asynchronously polled after each Alarm Clock signal, until it indicates that the client's request was serviced.

A synchronous *select*(2) request is simulated in the same way. The implementation maintains a global "I/O queue", plus bit vectors of File Descriptors for each thread suspended on it. When the package's scheduler calls *select*(2), it asks the operating-system about the union of all of these File Descriptors and it resumes any thread(s) awaiting information about those ready for service. A thread may be removed from the queue and resumed sooner if the Alarm Clock signal handler detects expiration of a time-out interval, optionally specified by a thread when it calls *select*(2) (*cf.* §2.4 for more about timing).

The *pause*(2) and *wait*(2) operating-system services ordinarily return when a process receives certain signals. Like *select*(2), they are simulated using a global "signal queue", plus a bit vector *per* thread, except that when a thread is suspended in this queue, its vector indicates the signal(s) it awaits.

Signals also create other problems, which we consider next.

2.4 Signals and Timing

In addition to scheduling threads and simulating operating-system services, a coroutine package must intercept UNIX signals for a further reason.

The IEEE proposal divides signals into two classes. *Synchronous* signals indicate conditions attributable to a particular thread, such as division by zero, floating-point arithmetic errors, attempted execution of an invalid machine instruction, memory reference errors, or an error in an operating-system service request. The proposal requires that these signals be delivered to the thread that causes them. All other signals, termed *asynchronous*, indicate conditions that cannot be attributed to a particular thread[9], and the proposal lets them be delivered to any thread—most

8. *Cf. alarm*(2) and *signal*(5).

9. In our implementation, for example, an Alarm Clock signal is considered asynchronous.

conveniently, to whichever one is executing when a process takes delivery. But, what if all threads are suspended then? If an application program's signal handler expects to use the *C* Library **longjmp()** procedure[10] to resume execution at a predetermined recovery point, and if an underlying coroutine package uses the stack-copying method of thread-context switching (§2.1), the package risks disaster unless it either ignores the signal or else forcibly installs the stack of the thread whose *setjmp*(3C) call established the recovery point. Of course, ignoring a signal can induce application program errors, but identifying an arbitrary thread that previously called *setjmp*(3C) may be impossible, so our package implements a compromise: it installs the stack belonging to a process' initial thread[11] in order to handle an asynchronous signal.

To meet all of the needs described here and in the previous section, our coroutine package arranges with the operating-system to receive signals internally, where it filters them, performs simulation steps, then calls any handler(s) an application program may have requested. The latter requires mediating any program requests that (un)install handlers or change a process' signal *mask*[12]. Our package preserves a process' initial handlers and mask, asks the operating-system to deliver all signals to it, then simulates whatever modification of the initial state a program requests. Since HP-UX supports several interfaces to its signal management facilities, seven *C* Library stubs must be replaced for this purpose alone.

An application program may request delivery of Alarm Clock signals, which the coroutine package also needs, so our system must simulate certain timing facilities by mediating operating-system requests that manage them, including *alarm*(2), *getitimer*(2), and *setitimer*(2). The *setitimer*(2) service normally allows a program to enable or disable clock signals at times specified, with microsecond resolution if the underlying hardware allows it. Of course, our package's scheduler cannot perform a thread-context switch so quickly and, in fact, the scheduler's overhead is likely to be intolerable unless its time-slice interval for thread preemption is considerably longer than the thread-context switching time. The package therefore requests Alarm Clock signal delivery at an interval equal to the greatest common divisor of a time slice and any interruption interval an application program may have requested by calling

```
setitimer(ITIMER_REAL,...)
```

We then use software counters to simulate an interruption of the scheduler every s signals and an interruption of the application program every u signals, where s and u are ratios of the respective intervals to the g.c.d. Currently, we implement a degenerate case in which s is 1, effectively quantizing application requests in multiples of a

---

10. *Cf. setjmp*(3C).

11. In *C*, this is usually the thread that executes an application's **main()** program.

12. *Cf. sigvector*(2).

time slice, which is 200 mS. If an application program's operating-system requests indicate that it wishes to ignore Alarm Clock signals, our package's internal handler continues to receive them but merely stops relaying them to the application.

The package must forestall delivery of Alarm Clock signals to avoid interruption while it manipulates scheduling queues and other internal data structures. One way to do this is by invoking *sigblock*(2) or a similar operating-system service around such *critical sections*[13] but since these are short and numerous, the execution-time cost is high. Our package uses a faster solution: A lock variable in the application program's virtual-memory address space is set and cleared around critical sections, and it is tested by the package's signal handler. When the lock is set, Alarm Clock signals are considered to be logically "masked", so the package's handler dismisses them without action. Most modern computers, including the two described here, have machine instructions to manipulate such locks atomically, at high speed. An alternate technique, used in the μSystem [7], is to locate all of a coroutine package's code in a single, contiguous, virtual-memory address range, so that a signal handler can perform a quick range check to decide whether an interrupted instruction belongs to the package or to application code. But segregating modules in this way either requires compiler and linker support that is not available under all versions of UNIX, or else it requires that a coroutine package be pre-linked into a single module, which negates the "pay only for what you use" code size benefit of making it a library.

It can argued more generally that delivery of *all* asynchronous signals should be masked during an internal critical section, since an application program's signal handler might call some other subroutine defined in the IEEE proposal (*e.g.*, **pthread_yield()**) that manipulates data structures perhaps left in an inconsistent state by an interruption. Although Alarm Clock signals always recur, other signals may not, so a thread implementation cannot afford to ignore them. But since delivery cannot be postponed without costly operating-system intervention, our package relays other asynchronous signals to an application program even during an internal critical section, thereby admitting a possible source of error in the interest of speed.

A final problem created by internal critical sections is that ignoring Alarm Clock signals induces delay in the package's simulation of UNIX timing facilities.

2.5 Packaging Issues

Any mechanism that simulates operating-system facilities should be transparent to an application programmer; it should be as general and easy to use as the mechanism it replaces, and it should support as many programming languages[14] with as few

13. Dijkstra's terminology [12].

14. The IEEE proposal currently defines only a *C* language binding, but we anticipate that other bindings will be specified in time.

surprises as possible. This objective dictates not only coding choices, but also packaging decisions that are compatible with common UNIX programming tools and procedures. Since our coroutine implementation is a collection of subroutines, the most obvious package is an archive library file[15] that supplements and overrides the standard *C* Library. However, this is inconvenient for two reasons; one of them involves a design weakness of the UNIX linker, *ld*(1), and the other concerns package initialization.

To understand them, one must appreciate that most UNIX *C* programs are built using the convenient *cc*(1) command, which automatically compiles and links a program's modules, arranges to link *C* Library subroutines, prefixes a special **crt0**[16], library module that initializes and calls a **main()** program, optionally appends a debugger-support module, and substitutes library modules as necessary when support for execution profiling is requested. A "cc" command may specify a list of supplementary and overriding libraries to which the *C* Library is automatically appended, and the linker searches these libraries for unresolved module references in the order listed.

2.5.1 Linker Limitation

A restriction imposed by almost all versions of the UNIX linker is that a list of libraries is searched only once, in the forward direction. An unresolved reference in a library module must therefore be satisified either by:

- a module that is already linked, or
- another module in the same library, or
- a module in a library that appears later in the list.

Backward searches for additional library modules are not allowed. This creates a problem when one wishes to override operating-system stubs in the *C* Library, always last in a list, since other *C* Library modules refer to these stubs as well. Such internal references cannot be linked to replacements in a preceding library. In order to replace a stub, a programmer must either guarantee that a reference to it occurs earlier in the library list or else identify the replacement module, extract it from its library, and explicitly request that it be linked. Either alternative is clumsy and error-prone.

As an example, consider a frequently used *C* Library subroutine in the Standard I/O package such as **getchar()**[17], which calls a hierarchy of subroutines in the same

---

15. *Cf. ar*(1) and *ar*(4).

16. The module is named for and initializes "*C* run-time (location) 0", the virtual-memory address at which the *exec*(2) operating-system service starts a new program.

17. *Cf. getc*(3S).

library, ultimately invoking the *read*(2) operating-system service. It should not be necessary for programmers to know of this dependency, but since our coroutine package must replace the *read*(2) stub (in order to overlap input with other activity; *cf.* §2.3), **getchar()** won't work if our system were packaged in a separate library, unless a programmer knows enough about both packages' internal organization to explicitly link the replacement stub. Similar scenarios could be constructed for many other library subroutines, all equally annoying.

In order to solve the problem, either the linker's library-search algorithm must be changed or else the coroutine package must be merged with standard subroutines in one library. The first option offers the greatest generality and convenience, but it requires redesigning the UNIX linker. Most other operating-systems' linkers offer the functionality we need by doing library searches iteratively, but this entails more execution time and/or memory than a "one pass" algorithm. The UNIX linker's designer thought that such a feature would rarely be necessary, so it was omitted in order to improve performance in simple cases. Unfortunately, adding the capability takes more time than we could spare, so we did not pursue the matter.

Directly merging our coroutine package into the standard *C* Library seemed inconsiderate because it imposes a small, but gratuitous performance penalty on ordinary, single-thread applications, as well as some implementation restrictions, so we copied the library under a different name and merged our package into the copy. To compile and link a multiple-thread application, a user of the *cc*(1) command must specify the copy as the last (or only) library in a list, and the standard *C* Library will then be ignored by the linker. A user who invokes *ld*(1) directly must specify one library or the other but need not request both.

2.5.2 Package Initialization

Our coroutine package requires run-time initialization that, for the sake of transparency and convenience, should be hidden from an application programmer and should occur before an application's **main()** program is entered. These functions include:

- Allocate and initialize memory for internal data structures.
- Create (data structures representing) a process' initial thread of control.
- Reopen a process' standard input, output, and error streams for implicitly asynchronous I/O.
- Ask the operating-system to install the package's signal-handling subroutines and to deliver all signals.
- Start Alarm Clock signals.

This code also invokes the *sigstack*(2) operating-system service to request that signal handlers be called using a special, internal stack, so that they can safely alter the stack of an interrupted thread.

Initialization code could potentially be invoked in one of several ways; for instance, the package could require that a programmer call an initialization procedure explicitly, or it could arrange that each call of a user-visible subroutine implicitly tests (and conditionally sets) an internal variable, indicating that initialization has been accomplished. The first option is inconvenient, and its accidental omission can cause errors, while the second is inefficient. Our implementation therefore calls an initialization procedure from the *C* Library start-up module, **crt0**. To avoid linking our package with ordinary, single-thread applications, while also avoiding two versions of this module, we use a linker-dependent trick[18].

The start-up module calls the initialization procedure indirectly through a "pointer" variable, but only if the variable's value is not zero. The variable is defined in the start-up module, but it is not initialized there; instead, the variable is redeclared in the module containing the initialization procedure, where it is bound to the procedure's entry-point address. (It is guaranteed that the initialization procedure cannot begin at address 0, since that is where the start-up module begins.) Moreover, the latter module defines all of the package's statically-allocated variables, so cross-module references insure that the linker will incorporate it in any application program that includes other modules of the package. If a program does not use the package, then there are no references to its data, so the initialization module is omitted and the linker silently makes the pointer's value zero, thereby insuring that no call is attempted. But if the initialization module must be linked in order to satisfy external references, the linker merges its (non-zero) pointer value into the resulting binary file, making it accessible to the *C* Library start-up module, which insures that the initialization procedure will be called. In other words, the pointer is treated much like a conditionally initialized FORTRAN **COMMON** variable.

2.6 Application Debugging

Debugging a multiple-thread application program under a conventional operating-system is difficult[19], because most debuggers assume that a program contains only one flow of control and are not designed to distinguish among several. In fact, designing a debugger for multiple threads is hard because:

- All debuggers require implementation-specific knowledge of how a program and its data structures are represented, and they usually require operating-system assistance to set and clear "break points", detect unexpected memory "store" instructions, "single step" through individual machine instructions, *etc*.

---

18. The idiosyncracy making it possible is apparently undocumented, perhaps because it is so much a part of system programming "folklore" that no formal description is thought necessary. The author discovered it by reading Carnegie-Mellon University's MACH code, but this is probably not the original source. A documented attribution is welcomed.

19. General problems of debugging multiple-thread applications have been discussed in other contexts; *e.g.*, [9].

- The queries that a multiple-thread debugger should be able to answer and the most appropriate interface through which to ask questions and receive answers are topics of current research.

There are few standards in this area yet, so most application programmers must use an ordinary, single-thread debugger and resort to detailed knowledge of a thread mechanism's internal implementation. This section describes several peculiarities of our package that make debugging difficult.

Each thread in our implementation is represented by a unique, internal data structure, and there is a statically-allocated, global variable that contains the address of one of these structures, identifying the thread in execution at that moment. (If there is no such thread, the variable's value is zero.) When an application program stops at a debugger "break point", a programmer may examine the variable to identify a stopped thread, and a debugger that supports "conditional break points" can test the variable to determine whether a particular thread should stop there. This technique is crude, and it would be more elegent to absorb such functionality into the debugger itself, but we did not have enough time.

The fact that our package is periodically interrupted by Alarm Clock signals (§2.3.2), together with a property of Motorola MC68000-series CPUs, badly disrupts the operation of debugger "break points" and "single step" commands on Series 300 computers. The "single step" problem arises because:

- This feature is usually implemented using the CPU's trace facility [19, Ch. 8], which involves setting a bit in the machine's Status Register so that a hardware exception is raised before the next User Mode instruction is executed.
- Alarm Clock signals occur every 200 mS, so when a programmer tries to "single step" a stopped program, a signal delivery to the process is nearly always pending.

As a result, the next User Mode instruction to be executed is the one that begins the coroutine package's signal handler, not the one before which the program stopped. Unless a programmer is prepared to "single step" through the entire signal handler, which takes so long that many further Alarm Clock signals occur in the meantime(!), it is impossible to make forward progress through the rest of an application. What is needed is a way to disable the trace facility between memory addresses occupied by signal-handler instructions, but the *ptrace*(2) operating-system service, through which it is manipulated, has no such option.

Debugger "break points" evoke a similar problem. These are implemented by temporarily replacing the machine instruction at the desired stopping point with a **TRAP** instruction, which raises a hardware exception. In response to this, the operating-system stops the program until the debugger issues a *ptrace*(2) command to restart it, and it delivers a Trap signal (**SIGTRAP**) to the debugger, informing it of the halt. If a programmer then types a debugger command to continue, the original instruction is restored and execution is resumed. But if delivery of an Alarm Clock signal to the

program is pending by the time a command is entered (as it usually is, since these signals arrive faster than most programmers can type!), a "race" condition can occur between signal deliveries to the two processes, and by the time the debugger learns of the **TRAP** instruction, the application program has halted at the beginning of the package's Alarm Clock signal handler. Since the debugger "remembers" where it installed the **TRAP** instruction, it becomes quite confused when the program appears not to have stopped there. Fortunately, there is a simple solution: When the coroutine package first registers its signal handlers with the operating-system, using the *sigvector*(2) service, it asks that delivery of Trap signals be masked while Alarm Clock signals are handled. Since a debugger can correctly determine a **TRAP** instruction's address after an Alarm Clock signal handler returns control of the CPU to the point of interruption, "break points" will then work as expected. This solution's main disadvantage is that "break points" deliberately set in the package's (or an application program's) Alarm Clock signal handler are ignored, but this seems a small price to pay.[20]

A final nuisance caused by periodic Alarm Clock signals is that most UNIX debuggers default to typing a message, and perhaps stopping a program, when any signal is delivered to it, so users of our package may be deluged by repeated messages unless special commands are issued to suppress them and continue execution. If a debugger has an ability to read a user-specified file of "start-up" commands, this mechanism can provide an effective solution.

2.7 Problems of the IEEE Proposal

The description of POSIX thread extensions in proposed IEEE Standard No. 1003.4 is intended to be an implementation-neutral, behavioral specification that is complete except for conditions that it explicitly describes as "implementation dependent" or "undefined". But our implementation has weaknesses that result from inadequate specification, and this section discusses them. Bear in mind that our work is based upon a draft version of the proposal, and subsequent revisions may render this discussion obsolete.

2.7.1 Language Bindings

The proposal currently defines an API only for the *C* language, although HP-UX supports other languages in which concurrent applications could reasonably be programmed. Our package defines all of its public functions as subroutines, not just

20. It may be worth noting that neither of these foibles affects Series 800 computers. They lack an MC68000-style CPU tracing facility, so the "single step" feature must be simulated using a "break point" with a one-instruction span, and the architecture-dependent problem described above does not arise. Debugger confusion over "break point" addresses is also unlikely, since these machines have a two-instruction CPU pipeline with delayed **Branch** instructions, so a signal delivered during a trap instruction leaves more information in the pipeline to distinguish the address of the trap from the address where control transfers next.

cpp(1) macros, so that other languages can be supported merely by creating additional interface files, which define the package's public data types and named constants in each language. For *C* there is a file named **<sys/pthread.h>**, which redefines several functions as in-line macros for greater efficiency, but this mechanism does not support other languages. Present implementors of facilities like these must temporarily define (parts of) their interfaces without the benefit of standardization, but we hope that the IEEE will soon specify other language bindings.

2.7.2 Process State and Non-Initial Threads

For compatibility with ordinary, single-thread UNIX programs, the proposal requires that a process' initial thread of control be able to change the process' state[21] using the usual operating-system services, but it does not specify whether non-initial threads are allowed to do so. Since a process' state is shared by all of its threads, there are numerous possibilities for deadlock; for example, one thread might inadvertently mask delivery of signals that other threads await. For greater robustness in such cases, our coroutine implementation prohibits certain changes of a process' signal-handling state by non-initial threads; specifically, altering its signal mask or certain bits that choose between A.T.&T. UNIX System V and 4.3BSD UNIX signal-handling semantics (HP-UX supports both). If a non-initial thread requests an operating-system service that would ordinarily affect these, that service's replacement stub returns the Invalid Argument (**EINVAL**) error code and ignores the request.

Perhaps the most troublesome state changes are caused by *fork*(2). The proposal only requires (§8.3.1.2) that:

> If a multi-threaded process calls *fork*(2), the new process contains a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources.

In the most straightforward implementation for multiple threads, a UNIX *fork*(2) service would duplicate all threads of control, which is probably not what most programmers intend, so it seems incumbent upon an implementation to end all but one thread in a process' clone. Since operating-system services later requested by a clone must behave in the traditional way for (at least) the initial thread, an implementation must either transform a clone's copy of the *fork*(2)-ing thread into something resembling the parent process' initial thread, or else it must prevent *fork*(2) from being called by a non-initial thread. Our package would require a complicated transformation of internal data structures in order to implement the first option, and it is not clear whether application programs need or should pay for this generality, so we chose the second option: if a non-initial thread calls *fork*(2), the Deadlock (**EDEADLK**) error code is returned and the request is ignored.

---

21. *Cf. exec*(2) and *fork*(2) for a description of a UNIX process' state.

It may be noted parenthetically that the 4.2BSD UNIX *vfork*(2) operating-system service will no longer be supported by HP-UX after Version 7.0, so our implementation treats *vfork*(2) as a synonym of *fork*(2), rather than trying to emulate their differences.

2.7.3 Asynchronous Exceptions

We have already discussed some of the proposal's requirements concerning signal delivery (§2.4), but there is yet another problem: If an application program wants a particular thread to take delivery of an asynchronous signal (e.g., to respond to the "attention" key of a user's console, using the Interrupt (**SIGINT**) signal), there is no sanctioned mechanism except to make the thread wait, using the proposed *sigwait*(2) service. Evidently there is no way for a thread to do useful work pending interruption for a recovery action.

The proposal defines a set of **pthread_cleanup_*()** functions that allow application-specific termination processing when one thread is cancelled by another (using a proposed **pthread_cancel()** function) or a thread voluntarily exits. It has been suggested that this could be used as a more general mechanism, by letting a thread's clean-up routine **longjmp()** to a recovery point that effectively rescinds termination in favor of further activity. If the exception giving rise to such cancellation and recovery should be an asynchronous signal, then presumably an ordinary UNIX signal-handling function, executing in the context of some thread (which one?), must identify the application program's designated handler-thread (how? Using a global variable?) in order to cancel it (can the signal-handling thread possibly cancel itself?). It is not clear whether the proposal allows any of this, and one can imagine enough pathological cases to warrant careful specification, but it is apparent that the result of such an interpretation is not pretty.

2.8 Open Problems

It should now be apparent that an effective coroutine simulation of concurrency requires emulating a large number of operating-system services outside the operating-system, which prompts the general question: how effective can such a strategy be? Of the problems created by our implementation, we have described several that are solved, and we now discuss some that are not.

2.8.1 Reentrant Library Subroutines

When an application program has more than one thread of control, global and statically-allocated variables implicitly become shareable data, so a programmer must consider the need for mutual exclusion and locking. This kind of recoding is fairly easy if one can alter the source program text, but it is almost impossible if it involves libraries originally built for single-thread use that may be available only in binary form, possibly from different vendors. The IEEE proposal recognizes that the *C* Library must especially be reentrant, and it prescribes changes for many of its subroutine interfaces. Realizing these intentions for an existing version of UNIX

demands only routine coding effort (for the most part), but quite a lot of it, and we have not been able to devote the time it requires. Hence, users of our coroutine package must know *a priori* which library subroutines use internal, statically-allocated variables, and they must arrange to call these from only one thread at a time.

Problems of this nature often involve file buffers or other data maintained by the Standard I/O package, and these can often be solved by organizing a program such that each file is manipulated by only one thread of control. Fortunately the standard error stream, which must often be used by several threads, is unbuffered by default, so in our uniprocessor implementation several threads can write messages on it without corrupting internal data structures, although displayed texts may be intermingled.

In general, requiring application programmers to circumvent internal implementation problems is unattractive, and a better solution is needed.

2.8.2 Shared Device Information

As discussed in §2.3.2, our coroutine package implicitly opens all of an application program's File Descriptors for asynchronous I/O, including those connected to its standard input, output, and error streams. When the standard input stream is connected to the teletypewriter (TTY) console controlling a process—the normal case under UNIX—the author observed a surprising consequence: If a Kill signal (**SIGKILL**) is delivered to the process in order to halt errant behavior, the user's command session is involuntarily terminated and the console is abruptly logged off the computer! An investigation revealed the following sequence of events.

When a UNIX process' standard input stream is connected to a controlling TTY, the open File Descriptor is shared with the process (*e.g.*, a Shell) that spawned the program, which transitively inherited the File Descriptor from a command session's original "log-in" Shell. Moreover, information describing whether file operations should be performed (a)synchronously appears to be stored in the operating-system's device driver module on a *per* device basis, not *per* process. In other words, this information is implicitly shared by all processes connected to the device. But in the circumstance at hand, all other sharing programs expected I/O to be synchronous (this is normal under UNIX) and none of them, especially the log-in Shell, were prepared to cope with asynchronous I/O. Now, it happens that when an application program issues a synchronous *read*(2) request from a TTY that (some other process) is operating asynchronously and no characters have been typed on the console, the operating-system returns a code indicating "success" but specifies that 0 Bytes were transferred. However, a Shell treats a 0-Byte transfer as an "end of input" condition, so it ends the command session, logs the console "off", then exits!

Neither the application program nor any other process can prevent this behavior by intercepting the Kill signal and switching the TTY to a synchronous mode, because **SIGKILL** is especially designed to halt errant programs, so the operating-system does not let them respond in any other way.

Evidently this problem arises because the operating-system unexpectedly shares device-state information. The author does not know whether this behavior is a "bug" or a "feature", nor whether it is unique to HP-UX or common to other versions of UNIX. It is likewise unknown whether the same weakness exists in UNIX driver code for other devices. Further investigation and a solution are needed.

2.8.3 Scheduling CPU-Intensive Processes

From an operating-system point of view, overlapping an application program's computation and I/O can be seen as a technique to increase the process' average CPU utilization; however, because part of an operating-system's job is to balance competing demands for resources, including CPU time, its process-scheduling algorithm may frustrate an application programmer's intent. The UNIX scheduler was particularly designed for I/O-intensive, time-sharing applications, and it expects programs to mix computation with I/O in order to overlap and mingle them. The scheduler computes performance metrics for each process, and it penalizes those deemed "uncooperative" by awarding them lower priorities for execution; CPU-time "hogs" receive an especially low priority. Thus, the main reason to use multiple of threads of control in a program is in direct conflict with the scheduler's objective.

The MACH operating-system solves this problem by treating threads as schedulable entities, not processes. Unfortunately, there are few ways to influence most UNIX systems' schedulers except to recode them. However, HP-UX reserves a range of high execution priorities for designated "real time" processes, and it provides both an operating-system service and a command interface[22] to assign them. This provides an effective solution for multiple-thread programs but not one that is portable to other versions of UNIX.

3 Performance

The IEEE proposal is intended to support real-time applications, so the most important requirement of any implementation is that it be fast; consequently, the proposal defines a set of performance metrics by which implementations should be evaluated. It is also important to compare the speed of thread-context switching to that of ordinary, process-context switching in order to decide whether the new mechanisms offer better support for concurrent applications than the old ones.

The coroutine package we have described was measured on HP 9000 Series 370 and 835 computers, and this section presents the results.

---

22. *Cf. rtprio*(2) and *rtprio*(1), respectively.

3.1 Measurement Conditions

Execution-time measurements of particular mechanisms defined in the proposal were made using test programs, coded in *C*. These and our package's own *C*-language modules were compiled by requesting as much code optimization as our compilers can provide. Each test program repeats its measurement inside a loop, then it averages these values over the number of iterations specified by the tester. The programs were executed under HP-UX Version 7.0 with their file-system *sticky* bits turned on[23], so that the time to load their machine instructions into memory could be ignored after an initial execution. Subsequent executions comprised the experimental trials, and each program was run at least ten times.

To prevent other programs from interrupting a test, the operating-system was placed in its *single user* state, and most other programs were halted. Several "daemon" programs built into the operating-system cannot be halted, so they were rendered quiescent in other ways: A Network Interrupt Service Request program (**netisr**) that receives messages through a computer's Local Area Network (LAN) hardware was stopped by disconnecting the LAN cable. A program that transfers processes from the main memory to secondary storage (**swapper**) and another that replaces virtual-memory pages (**pagedaemon**) were blocked by installing enough memory to obviate their functions.

Despite these precautions, our measurements were periodically interrupted by the operating-system's scheduler, which cannot be stopped. Our solution was to discard wildly deviant data that appeared to span interruptions, and to retain only trials yielding minimal, consistent results, which were assumed to represent periods between scheduling activity. Obviously, some subjective judgment was necessary. Each measurement that we report is computed from ten consistent, not necessarily consecutive trials.

The Series 370 time base is a crystal-controlled, 250 kHz real-time clock, accurate to two parts *per* million. It drives a 32-bit, recycling binary counter that is mapped into a test program's virtual-memory address space for fast access. The Series 835 time base is a 15 MHz clock, controlled by a ceramic resonator accurate to 0.05%[24]; it also drives a 32-bit counter that is implemented as a CPU register for even faster access.

---

23. *Cf. chmod*(2).

24. Despite its low accuracy, the resonator is highly stable and inexpensive. The operating-system keeps accurate time by correcting measured values in software; every computer's resonator is measured at the factory, then an individual calibration constant is computed and stored in a programmable, read-only memory (PROM) integrated circuit (IC). Unfortunately, HP-UX does not make this constant available to application programs, so we ignored it.

3.2 Process-Context Switching Time

Because literature about multiple threads often assumes that they behave as "light weight" operating-system processes, in the sense that thread-context switching is thought to be much faster than process-context switching, we decided to test this assumption by making reference measurements of process-context switching time. This parameter must usually be measured indirectly, and the methods are controversial; Appendix C discusses some of the problems and describes the technique we used. Table 1 summarizes our results.

Table 1. Process-Context Switching Time (μS)

CPU	Time
370	584.2
835	135.7

These measurements should be compared to thread yield times presented in §3.3.6.

3.3 IEEE 1003.4 Metrics

The next sections present measurements prescribed by the IEEE proposal; more information about each metric can be found in the part of Draft 4 cited in each subsection's title.

To evaluate metrics involving thread-context switching, two variants of our coroutine package were tested: one in which threads' execution stacks are copied to and from a shared memory region, and one in which the stacks reside in separate memory regions (*cf.* §2.1). Since copying time is proportional to a stack's size, measurements of the first variant are divided into two parts: a fixed cost, expressed in microseconds, and a marginal cost, expressed in microseconds *per* kilo-Byte (1024 Bytes) copied. The latter component was expected to depend mainly on a computer's CPU speed and memory bandwidth, including cache performance, and one can predict "best-case" marginal costs knowing a copy-alogrithm's machine-instruction representation and some hardware design information. Appendix B shows the analysis for the computers we measured; the marginal costs of copying memory are predicted to be 125-140 μS/kB for Series 370 and 104 μS/kB for Series 835.

In IEEE metrics affected by thread-context switching, test programs were timed using ½-, 1-, 2-, and 4-kB stack sizes, and the observed marginal cost of stack copying is taken to be the slope of a best-fitting line for each metric, calculated using the method of least squares. The observed fixed cost is taken to be that line's intercept; *i.e.*, the projected execution time had the stacks been empty. These numbers are reported below, and our observed marginal costs should be compared to predictions above.

3.3.1 Granularity of Parallelism, Light and Heavy Loads (P§3.4)

The proposal specifies two computational metrics to characterize the speed-up of parallelism on a multiprocessor computer. Since our implementation only simulates

parallelism on a single CPU, no computational speed-up can be expected, so these metrics do not apply.

3.3.2 Once-Only Overhead (P§3.4)

A proposed function, **pthread_once()**, together with statically-allocated "flag" variables, arrange that application-specific initialization code is executed at most once, even when requested by several threads. This is useful since an implementation may not be able to guarantee which thread runs first, and by coding such an initialization reqest in all of them, a programmer need not worry about it. However, the cost of the extra requests must be low.

In our coroutine package, a "flag" variable is a Byte that is initialized to zero when an application program is loaded into memory. For Series 370 computers, a *C*-language macro expands **pthread_once()** into in-line code, equivalent to the following assembly language:

```
        TST.B variable
        BEQ.B label1
        CLR.L D0      ; Return "success" if already called
        BRA.B label2
label1  MOV.B #1,variable
        (Call initialization function)
label2  ...
```

Only the first four of these instructions are executed more than once; if they and the "flag" variable are in the computer's memory caches (*cf.* Appendix B), their execution time is calculated[25] to be fourteen 30-nS CPU clock cycles, or 420 nS.

The code for Series 835 is equivalent to:

```
        ADDIL    L'variable-$global$,DP
        LDB      R'variable-$global$(0,1),register
        (1 instruction-cycle delay)
        COMB,=,N register,0,label1
        MOVB,TR  0,RET0,label2     ; Return "success" if already called
        (1 instruction-cycle delay)
label1  LDI      1,register
        STB      register,R'variable-$global$(0,1)
        (Call initialization function)
label2  ...
```

Again, only the first four instructions are executed more than once, but because this machine's **Branch** and **Load** instructions require two instruction cycles each, the sequence's execution time is six 67-nS instruction cycles, or 400 nS.

---

25. *Cf.* [19, Ch. 11] for information about how to calculate Motorola CPU instruction times.

3.3.3 Self-Identification Overhead (P§3.4)

A proposed function, **pthread_self()**, returns a unique identifier of the calling thread. This function should be fast because the identifier may be needed by other subroutines defined in the proposal. As noted earlier, our implementation represents an identifier by four Bytes, and it contains a statically-allocated global variable, holding the identity of the thread controlling the CPU at any moment. A *C*-language macro expands **pthread_self()** into in-line code, which loads this variable's value into a CPU register. The code for Series 370 is a Longword **MOVE** instruction that requires two CPU-clock cycles if the variable is in a cache, or 60 nS.

The code for Series 835 consists of two instructions, equivalent to:

```
ADDIL L'variable-$global$,DP
LDW   R'variable-$global$(0,1),reg
{1 instruction-cycle delay}
```

which require three instruction cycles, or 200nS.

3.3.4 Run-Time Stack Memory Allocation (P§3.4)

A thread attribute that is defined by the proposal lets a programmer specify the minimum size of that thread's execution stack. Since creating a large number of threads could exhaust available memory, an implementation must document its stack allocation mechanism's granularity and limits.

Our package allocates stack space in multiples of a virtual-memory page, using the *C* Library *malloc*(3C) function. A page is 4096 Bytes for Series 370 or 2048 Bytes for Series 835; consequently, a request for 16384, 32768, or 65536 Bytes would be satisfied exactly, but a request for 1024 Bytes would be exceeded by four- or twofold, respectively.

The amount of memory available for threads' stacks is limited only by the amount of data storage available to application programs executing on these computers. This, in turn, is determined by the operating-system, the CPU architecture, the amount of secondary storage installed for virtual-memory paging, and the demand by other processes when storage is requested. For Series 370, the limit could be as large as 4 GB, less the size of the requesting program's machine instructions. For Series 835, it could be as large as 1 GB except in the stack-copying variant of our package (*cf.* §2.1): HP-UX Version 7.0 limits a process' stack to the 384 MB virtual-memory address range from 68000000_{16} to 80000000_{16}, so no thread's stack may exceed this size.

3.3.5 Maximum Number of Threads (P§3.4)

The proposal requires documentation of any limit to the number of simultaneously unterminated (active) threads, simultaneously undetached[26] threads, or the total

number threads created during a program's execution. Our implementation is limited by its memory management strategy for certain *per* thread internal data structures.

Each thread is represented by a descriptor variable occupying a block of storage in an application program's virtual-memory address space, the address of which constitutes the thread identifier returned by **pthread_self()**. Most subroutines defined in the proposal that accept a thread identifier argument must validate it during each call. For speed, our package allocates thread descriptors from a *zone*, a contiguous, fixed-length pool of storage blocks that is located at a known virtual-memory address. Since a descriptor's size is constant, verifying that an argument refers to a descriptor is then a simple calculation.

Our scheme's main disadvantage is that the number of descriptors—hence, the number of simultaneously active threads—is a fixed value that must be established before the zone's storage is allocated, when an application program begins. This value also limits the number of simultaneously undetached threads. Our implementation chooses a default value but provides a *C*-language data-declaration macro to let the choice be overridden when a **main()** program module is compiled. The limit may not be chosen dynamically, and the current default value is sixteen. Since a descriptor is recycled when a thread terminates, the number of threads that may be created during a program's execution is unlimited except by this concurrency constraint. The IEEE proposal includes no override mechanism like ours, so it must be considered "implementation dependent" even though quite portable among applications coded in *C*.

3.3.6 Thread Yield Time (P§4.4)

A proposed function, **pthread_yield()**, lets a thread of control voluntarily relinquish control of its CPU; hence, timing this call when another thread can execute measures thread-context switching time. Some implementations of the IEEE proposal may optimize performance by avoiding an unnecessary self context-switch when only one thread can execute on a CPU, so two cases must be considered: when there is no other executable thread (*Yield Time, Not Busy*), and when there is (*Yield Time, Busy*).

3.3.6.1 Yield Time, Not Busy

Table 2 shows the execution time of **pthread_yield()** when only one thread can run, averaged over ten trials. Each trial includes 100,000 calls.

---

26. *Cf.* the proposed function, **pthread_detach()**.

Table 2. Yield Time, Not Busy (μS)

CPU	Stack Copied?	
	No	Yes
370	4.0	3.9
835	2.19	2.19

The difference between Series 370 figures is less than one instruction-cycle time and is much less than one measurement-clock period, so the discrepancy is considered to be an artifact of the measurement technique that is not statistically significant.

3.3.6.2 Yield Time, Busy

Table 3 shows the execution time of **pthread_yield()** when two threads can run, averaged over ten trials. Each trial includes 1,000 calls.

Table 3. Yield Time, Busy

CPU	Stack Copied?	
	No	Yes
370	53.8 μS	53 μS + 136 μS/kB
835	22.2 μS	25.5μS + 108 μS/kB

Comparing the fixed costs in Table 3 to process-context switch times in Table 1, we see that when threads of control are implemented as coroutines instead of operating-system processes, context-switching is up to eleven times faster on Series 370 computers and up to six times faster on Series 835 computers.

This is encouraging, but the fact that the speed-up is only about an order of magnitude suggests that the popular view of coroutines as "light weight" processes is overly optimistic; "medium weight" seems more accurate. These data suggest that process-context switching in a modern computer can be faster than is commonly perceived. Turning the comparison around, the fact that these process-context switching times are within an order of magnitude of what can be achieved with coroutines suggests that the latter's implementation restrictions may not be worth their modest speed improvement, and that despite the conceptual advantage of threads as an organizing principle for individual programs, ordinary operating-system processes may still be the most practical vehicle for concurrent applications.

Table 3 also implies that the cost of stack-copying exceeds the fixed cost of a thread-context switch when the average stack depth exceeds about 400 Bytes for Series 370 or 240 Bytes for Series 835. Because stack space is consumed by local variables and CPU-state information that is saved during nested subroutine calls, we may take this as a crude complexity measure of work done by a thread and say loosely, in this sense, that the copying method of stack allocation (§2.1) is adequate when threads do simple tasks, but the cost of copying stacks dominates overhead as threads' activities become more complex.

3.3.7 Scheduling Overhead (P§4.4)

The IEEE proposal prescribes a mechanism by which a programmer can establish a scheduling policy for each thread, and one of the choices, noted earlier, is a preemptive *round robin* policy using priority levels and "time slicing". We noted that this entails periodic interruption of a process in order to determine whether a thread-context switch should be forced, and that our implementation does the job in an Alarm Clock signal handler, executed every 200 mS. Obviously, the overhead of these interruptions should be as low as possible, and the proposal specifies that it be measured.

Our handler's execution time is very short in comparison to a time slice, so measuring enough calls to compute a reasonably precise average takes a long time. Therefore we made a faster measurement, using a kind of simulation: A test program repeatedly invoked our handler in a "dummy" environment, which was initialized to guarantee that no rescheduling decisions were made; then the time *per* call, averaged over a large number of calls, was taken to be the handler's overhead. Table 4 shows the results, averaged over ten of these trials. Each trial includes 5,000 calls.

Table 4. Scheduler Overhead (μS)

CPU	Stack Copied?	
	No	Yes
370	4.7	4.9
835	2.85	2.85

This method does not include operating-system overhead incurred to interrupt an application program, both because it is hard to measure and to preserve comparability, UNIX ordinarily accounts separately for "system time" incurred on a process' behalf, regardless of whether it covers one thread or many. In any event, the overhead is small.

3.3.8 Time to (Un)Lock a Mutual-Exclusion Variable (P§5.4)

The IEEE proposal defines facilities to manage locks in shared memory, called *mutex* variables[27], so that threads can mutually exclude concurrent access to shared resources. The proposed functions, **pthread_mutex_[un]lock()**, set and release a lock on behalf of a calling thread. If a thread tries to set a lock that another thread has already set, the second thread is suspended until the first releases the lock. These functions can therefore entail thread-context switching, so one expects delays proportional to thread yield times (Table 3) plus an extra fixed cost to manipulate the lock.

27. Like Dijkstra's *semaphores* [12].

Implementations may optimize performance by avoiding an unnecessary self context-switch when only one thread can execute on a CPU, so two cases must be considered: when no thread awaits a lock's release (*Mutex Lock/Unlock Time, No Contention*), and when one does (*Mutex Lock/Unlock Time, Long Term Contention*). The proposal specifies a further computational metric (*Mutex Lock/Unlock Time, Short Term Contention*) to characterize the cost of contention on a multiprocessor computer, but it does not apply to this implementation.

3.3.8.1 Mutex Lock/Unlock Time, No Contention

Table 5 shows the time to release and set a lock when only one thread contends for it, averaged over ten trials. Each trial includes 5,000 paired calls of **pthread_mutex_[un]lock()**.

Table 5. Mutex Lock/Unlock Time, No Contention (μS)

CPU	Stack Copied?	
	No	Yes
370	10.4	10.7
835	12.76	12.32

The small differences as a function of stack-management technique reflect several instructions different between the two variants' execution paths, which are not considered statistically significant.

3.3.8.2 Mutex Lock/Unlock Time, Long-Term Contention

Table 6 shows elapsed time when one thread releases a lock and a second, waiting thread sets it, all averaged over ten trials. Each trial includes 1,000 paired calls of **pthread_mutex_[un]lock()**.

Table 6. Mutex Lock/Unlock Time, Long-Term Contention

CPU	Stack Copied?	
	No	Yes
370	66.1	68 μS + 130 μS/kB
835	35.7	41.5μS + 108 μS/kB

Comparing the marginal costs in Table 3 to those in Table 6 shows that our implementation causes a single thread-context switch. Subtracting the fixed costs in Table 3 from those in Table 6 gives the direct cost of manipulating the lock variable, shown in Table 7.

Table 7. Lock Manipulation Time (μS)

CPU	Stack Copied?	
	No	Yes
370	12.3	15.0
835	13.7	16.0

3.3.9 Time to Signal/Broadcast a Condition (P§5.4)

The IEEE proposal defines additional facilities to synchronize threads, using event queues called *condition* variables. A proposed function, **pthread_cond_wait()**, suspends a calling thread in a queue that is associated with an argument condition variable until an occurrence of the corresponding event is declared. A function named **pthread_cond_signal()** declares such an occurrence when it is called with the same argument, by resuming execution of the (single) highest-priority waiting thread. A similar function named **pthread_cond_broadcast()** resumes all waiting threads. An additional function, named **pthread_cond_timedwait()**, behaves like **pthread_cond_wait()** with an added deadline-time argument; it returns an error status if the calling thread resumes after the deadline has passed.

A thread wishing to wait for an event must call one of the latter two functions with a locked mutex variable. An implementation implicitly releases the lock when it suspends the thread, then resets it on the thread's behalf prior to resumption. If the lock is set by another thread when an occurrence of the event is declared, resumption is delayed until the suspended thread can once more be given control of the lock.

All of these functions entail thread-context switching, perhaps more than once, so one expects delays proportional to thread yield times (Table 3) plus an extra fixed cost to manipulate condition- and mutex variables. When two threads share a lock for mutual exclusion from a critical section of code, the number of context switches may depend upon whether one of them signals the associated condition inside or outside the critical section, which the other may be waiting to enter (signalling outside the critical section should exhibit better performance), so both cases must be considered. Moreover, implementations may optimize performance by avoiding an unnecessary self context-switch when only one thread can execute on a CPU, so these must be sub-divided into two further cases: when no thread awaits the event, and when one does. In all four cases, measurements are made using at most two threads.

3.3.9.1 Condition Signal/Broadcast Time, No Waiting Threads

Table 8 shows elapsed time to signal a condition that no thread awaits, averaged over ten trials. Each trial includes 5,000 calls to **pthread_cond_signal()**.

Table 8. Condition Signal Time (μS)

CPU	Stack Copied?	
	No	Yes
370	3.6	3.6
835	1.81	1.81

Table 9 shows elapsed time to broadcast a condition that no thread awaits, averaged over ten trials. Each trial includes 5,000 calls to **pthread_cond_broadcast()**.

Table 9. Condition Broadcast Time (μS)

CPU	Stack Copied?	
	No	Yes
370	3.3	3.3
835	2.64	2.64

3.3.9.2 Condition Signal Time, Untimed Threads Waiting

Table 10 shows elapsed time to signal a condition that another thread awaits, outside any critical section and without timing (*i.e.*, the waiting thread called **pthread_cond_wait()**), averaged over ten trials. Each trial includes 1,000 calls to **pthread_cond_signal()**.

Table 10. Condition Signal Time, Outside Critical Section

CPU	Stack Copied?	
	No	Yes
370	78.0	78 μS + 132 μS/kB
835	40.5	44.2μS + 108 μS/kB

Table 11 shows comparable data when the condition is signalled inside a critical section, guarded by a shared lock.

Table 11. Condition Signal Time, Inside Critical Section

CPU	Stack Copied?	
	No	Yes
370	196	197 μS + 396 μS/kB
835	94.4	105.7μS + 325 μS/kB

Comparing the marginal costs in Tables 10 and 11 to those in Table 3 shows that our implementation causes one thread-context switch in the the first case and three in the second. Subtracting the fixed costs in Table 3 from those in Table 10 gives the direct cost of manipulating the condition variable, shown in Table 12.

Table 12. Event Manipulation Time (μS)

CPU	Stack Copied?	
	No	Yes
370	24.2	25
835	18.3	18.7

Subtracting three times the fixed costs in Table 3 from those in Table 11 gives the combined cost of manipulating the condition variable plus releasing and setting the shared lock an extra time. We can separate these components using Table 12, with the result shown in Table 13.

Table 13. Event+Lock Manipulation Time (μS)

CPU	Stack Copied?	
	No	Yes
370	24.2+10.0	25.0+10.0
835	18.3+ 9.5	18.7+10.5

The fact that second components of these figures are in good agreement with Table 5 verifies the consistency of these computations.

3.3.9.3 Condition Signal Time, Timed Threads Waiting

Table 14 shows elapsed time to signal a condition that another thread awaits, outside any critical section and with timing (*i.e.*, the waiting thread called **pthread_cond_timedwait()**), averaged over ten trials. Each trial includes 1,000 calls to **pthread_cond_signal()**.

Table 14. Condition Signal Time, Outside Critical Section

CPU	Stack Copied?	
	No	Yes
370	169.3	164 μS + 138 μS/kB
835	100.7	99.1 μS + 116 μS/kB

Table 15 shows comparable data when the condition is signalled inside a critical section, guarded by a shared lock.

Table 15. Condition Signal Time, Inside Critical Section

CPU	Stack Copied?	
	No	Yes
370	275	288 μS + 400 μS/kB
835	155.5	169.7μS + 332 μS/kB

Comparing the marginal costs Tables 14 and 15 to those in Table 3 shows that, like untimed waiting, our implementation causes one thread-context switch in the first case and three in the second.

Our implementations of **pthread_cond_wait()** and **pthread_cond_timedwait()** differ in that the latter queries the operating-system for its time of resumption using the *gettimeofday*(2) service, then it calculates whether the caller's deadline has passed using a double-precision integer comparison. Subtracting the fixed costs in Tables 10 and 11 from those in Tables 14 and 15, respectively, then averaging, shows that this extra work costs about 87 μS for Series 370 and 60 μS for Series 835.

Subtracting the cost of deadline checking and three times the fixed costs in Table 3 from the fixed costs in Table 15 gives a second estimate of the combined cost to manipulate the condition variable, plus release and set the shared lock, which is shown in Table 16.

Table 16. Event+Lock Manipulation Time (μS)

CPU	Stack Copied?	
	No	Yes
370	24.2+ 2.8	25.0+17.0
835	18.3+10.5	18.7+14.4

These data show more variation than Table 13 because they represent differences between larger numbers that are more difficult to measure consistently.

4 Discussion and Conclusions

We have described a library-based coroutine simulation of concurrent programming facilities prescribed by proposed IEEE Standard No. 1003.4, which provides most of the proposal's functionality on a uniprocessor using an unmodified version of UNIX. Our package is easy to use with traditional programming tools and a variety of languages, although it would be even more transparent without the UNIX linker's "one pass" library search restriction. We noted additional functionality, such as reentrant library routines, that could be provided with more time and effort. Our implementation can support ordinary, single-thread UNIX applications, with some restrictions.

We discussed implementation problems due to ambiguities and weaknesses in the IEEE proposal, as well as its requirement of compatibility with ordinary, single-thread POSIX programs. Most of the latter difficulties, involving treatment of a process' initial thread, can be solved with additional effort, but questions involving asynchronous-exception handling and additional language bindings beg further work by the standardization committee.

A practical simulation of concurrency within application programs must transparently emulate operating-system services, and our system shows that this is possible to a considerable degree; however, there are limits—especially when execution speed is important. Time-dependent program behavior cannot be simulated with great accuracy or precision if the emulation package must share the UNIX signal mechanism for internal scheduling purposes. It is hard to allow delivery of asynchronous signals while the package's internal data structures are in flux without incurring the cost of frequent operating-system calls to mask signals around critical sections. Simulating certain synchronous operating-system services entails repeated polling, which is inefficient.

Perhaps the most obvious problem is the time needed to switch threads' execution stacks unless automatic overflow detection and expansion are sacrified, due to the need to copy stack contents in a traditional, "one stack *per* process" UNIX environment. Speed could be obtained with relative—though not absolute—safety if large, disjoint regions of virtual memory could be cheaply allocated for separate stacks, taking advantage the fact that these are often sparsely used. Following the lead of

MACH, some versions UNIX now provide such a facility, so a portable solution may eventually be possible.

Measurements of our package show that execution speeds of critical functions can be up to an order of magnitude faster than comparable operating-system services, which are traditionally used to manage concurrent processes. Some recent discussion of "light weight" processes suggests that the gains should be far greater [1], so such modest improvements are a surprising and perhaps disappointing result. It happens not because coroutines are slow, but because modern computers' operating-systems are unexpectedly fast. In other words, commentators' expectations may be unrealistic because they reflect hardware that is obsolete.

If the amount of speed-up we observed should prove typical of other CPUs, this has implications for what can be expected of new operating-systems, such as MACH, which implement threads as internal primitives. Since their thread management services are unlikely to be much faster than a coroutine package on the same computer (and we hope that they will be no slower than existing process management services!), we conjecture that one should expect gains approximating the geometric mean of the extremes we have measured; that is, speed-ups of only about 2-4 times relative to ordinary operating-system processes or (equivalently) losses of 2-4 times relative to coroutines. Future CPU designs that reduce the cost of context-switching between an operating-system and an application program will narrow this performance gap even further and may reduce the appeal of threads as an efficient software abstraction.

But even modest improvements should not be gainsaid in the light of another benefit. It is widely acknowledged that programming languages should provide better mechanisms to express concurrency for today's application problems, but after nearly a quarter-century of effort, there are still very few standards; hence, programmers continue to rely upon that traditional mechanism of abstraction, the subroutine. Because even so primitive a device is too important to abandon, threads of control managed through subroutine interfaces seem likely to become increasingly important, and we hope that implementations like ours will promote new, concurrent applications, based upon the emerging IEEE standard.

5 Acknowledgements

The work of an industrial research laboratory is usually a group effort, and I have gratefully drawn upon the knowledge of many fellow employees in several Divisions of the Hewlett-Packard Company. Special help was provided by Kevin Ackley, Deborah Caswell, Tim Connors, Bart Sears, Rob Seliger, and Bob Shaw.

6 References

1. Anderson, Thomas E., Edward D. Lazowska, and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. on Computer Systems 38,* 12 (Dec 1989) 1631-1644.

2. Reference Manual for the Ada Programming Language. United States of America Department of Defense Military Standard No. MIL-STD-1815A-1983.

3. Apollo Computer, Inc. Concurrent Programming Support (CPS) Reference Manual. Order No. 010233, Jun 1987.

4. Bailes, Paul A. A Low-Cost Implementation of Coroutines for *C. Software; Practice and Experience 15,* 4 (Apr 1985) 379-395.

5. Baker, Jr., Henry G. Shallow Binding in LISP 1.5. *Comm. ACM 21,* 7 (Jul 1978) 565-569.

6. Binding, Carl. Cheap Concurrency in *C. SIGPLAN Notices 20,* 9 (Sep 1985) 21-26.

7. Buhr, Peter A., and Richard A. Stroobosscher. The μSystem; Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software; Practice and Experience 20,* 9 (Sep 1990) 929-964.

8. Butenhof, David, Robert Conti, Paul Curtin, and Webb Scales. *Concert Multithread Architecture,* Draft 1.1. Digital Equipment Corporation (unpublished technical report), 28 Feb 1990.

9. Caswell, Deborah L., and David L. Black. Implementing a MACH Debugger for Multithreaded Applications. *Proc. USENIX Assoc. Winter Tech. Conf. and Exhibition* (22-26 Jan 1990) 25-39.

10. Conway, Melvin E. Design of a Separable Transition-Diagram Compiler. *Comm. ACM 6,* 7 (Jul 1963) 396-408.

11. Dahms, David. *Banana Senior Processor Board; External Reference Specification.* Hewlett-Packard Company, Fort Collins Systems Division Drawing No. A-98562-66516-12, 1 Jan 1987.

12. Dijkstra, Edsger W. Cooperating Sequential Processes, in F. Genuys, ed. *Programming Languages* (New York: Academic Press Inc., 1968), 43-112.

13. Hargis, Jeffrey G., and John J. Murzyn. *Top Gun; Theory of Operation.* Hewlett-Packard Company, Fort Collins Systems Division Drawing No. A-09850-66516-9, 20 May 1988.

14. *HP 3000 and 9000 Series Computers; Precision Architecture Instruction Set Reference Manual,* 3rd Ed. Hewlett-Packard Company Part No. 5952-0510, Sep 1989.

15. Institute of Electrical and Electronic Engineers. (Proposed) Standard No. 1003.4, *Realtime Extensions for Portable Operating Systems,* Draft 9. IEEE Computer Society, Technical Committee on Operating Systems, Joint Technical Committee Work Item No. JTC1.22.21.2, 1 Dec 1989.

16. Kepecs, Jonathan. Lightweight Processes for UNIX Implementation and Applications. *Proc. USENIX Assoc. Summer Tech. Conf. and Exhibition* (11-14 Jun 1985) 299-308.

17. Leffler, Samuel, Michael Karels, and M. Kirk McKusick. *Measuring and Improving the Performance of 4.2BSD.* University of California at Berkeley, Department of Electrical Engineering and Computer Science, Computer Science Research Group (unnumbered technical report), 14 May 1984.

18. Meyer, Thomas O., Russell C. Brockmann, Jeffrey G. Hargis, John Keller, and Floyd E. Moore. New Midrange Members of the Hewlett-Packard Precision Architecture Computer Family. *Hewlett-Packard J. 40,* 3 (Jun 1989) 18-25.

19. Motorola, Inc. *MC68030 Enhanced 32-Bit Microprocessor User's Manual,* 3rd Ed. (Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1990).

20. Randell, Brian, and L. J. Russell. *ALGOL 60 Implementation* (New York: Academic Press, Inc., 1964).

21. Raveling, Paul A. Personal communcation describing the author's Portable UNIX Benchmark Suite, written at Information Sciences Institute, University of Southern California, 1989.

22. Roesner, Arlen L. *Top Gun; External Reference Specification.* Hewlett-Packard Company, Fort Collins Systems Division Drawing No. A-A1035-90300-9, 13 Sep 1988.

23. Severin, Janet, and Janet Lehl. *Wolverine; External Reference Specification.* Hewlett-Packard Company, Fort Collins Systems Division Drawing No. A-98579-66515-12, 23 Sep 1988.

24. Severin, Janet, and Janet Lehl. *Wolverine; Theory of Operation.* Hewlett-Packard Company, Fort Collins Systems Division Drawing No. A-98579-66515-9, 23 Sep 1988.

25. *System Services Overview.* Sun Microsystems, Inc. Part No. 800-1753-10, May 1988.

26. Tevanian Jr., Avadis, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. MACH Threads and the UNIX Kernel; The Battle for Control. *Proc. USENIX Assoc. Summer Tech. Conf. and Exhibition* (8-12 Jun 1987) 185-198.

27. Tevanian, Jr., Avadis. Architecture Independent Virtual Memory Management for Parallel and Distributed Environments. Ph.D. Diss., Department of Computer Science, Carnegie Mellon University, 1987.

Appendix A: Operating-System Stub Replacements

In order to overlap a process' computation and I/O without suspending it when it has executable threads of control, the coroutine package described in this paper replaces these operating-system stub subroutines in the HP-UX *C* Library:

accept	getitimer	select	sigsetmask
alarm	msgrcv	semop	sigsuspend
close	msgsnd	send	sigvector
connect	open	sendto	socket
creat	pause	setitimer	wait
dup	read	sigaction	wait3
dup2	readv	sigblock	waitpid
fcntl	recv	sigpause	write
fork	recvfrom	sigprocmask	writev

The package also replaces these stubs in HP's NetIPC Library:

ipccontrol ipccreate ipcrecv ipcrecvcn ipcsend

Appendix B: Coroutine-Switching Marginal Costs

When coroutines' execution stacks time-share a common virtual-memory address range, and stack contents are copied between it and unique "save areas" (*cf.* §2.1), thread-context switching time is proportional to the rate at which computer hardware copies data from one memory address to another. This, in turn, depends upon a machine's instruction-cycle time, its memory cache design, and the amount of data copied by each instruction of an algorithm. Since a cache is usually shared by all processes executing on a computer, its performance is determined by their combined memory access patterns; but when only one process executes, it is possible to predict the maximum copy rate—hence, the marginal cost of thread-context switching—knowing the algorithm's machine-instruction representation and some hardware design information. This appendix shows the analysis for the HP 9000 Series 370 and 835 computers measured in this study.

Note that a copy algorithm must be executed twice during each thread-context switch: once to save the suspended coroutine's stack, and again to restore that of the resumed coroutine.

B.1 HP 9000 Series 370

A Series 370 CPU is a Motorola MC68030 IC [19], clocked at a 33 MHz rate. The chip contains separate, 256-Byte, first-level instruction and data caches; both are direct-mapped using virtual-memory addresses. The instruction cache provides fast access to a copy algorithm's machine instructions as long as the operating-system does not interrupt, since both on-chip caches must then be flushed; however, the data cache is ineffectual when copying large blocks of memory. This computer also includes a 64 kB, off-chip, second-level cache combining instructions and data, which is direct-mapped using physical addresses and can be read at the maximum CPU speed. Data enter this cache from the main memory in four-Longword (sixteen-Byte) units, but they are written back only a Longword at a time, using a *write through* policy; thus, the copy rate for large blocks is bounded by the speed of writing to the main memory. This happens partly because the CPU's printed circuit board was designed to be a plug-in replacement for earlier, Series 350 computers [23,24], memory systems of which operate at only 25 MHz [11]. A memory **read** cycle takes three 40-nS memory clock periods, or 120 nS, and a **write** cycle takes seven periods, or 280 nS. If data to be copied are all in the second-level cache, a **write** from the cache to the main memory can entirely overlap a succeeding **read** in the opposite direction, so the **read** time can be ignored but the *write through* policy demands that the **write** time itself be included in our calculations. The buss bar between the cache and main memory includes a "first in, first out" (FIFO) buffer holding a Longword of data to be written, so an isolated **write** delays the CPU for only three 30-nS clock pulses, but successive **write**s take either eight or nine clock pulses apiece, depending upon the relative phase of the CPU- and memory clocks.

The copy algorithm is an unrolled loop, consisting of sixty-four Longword **MOVE** instructions and a **DBF** instruction[28], which copies 256 Bytes *per* iteration like the following assembly language:

```
        MOVE.L #(No. of 256-Byte segments, less 1),D0
label   MOVE.L (A0)+,(A1)+ ; Repeat 64 times
        ...
        DBF    D0,label
```

This is the largest power-of-two expansion fitting entirely in the first-level instruction cache, and it effectively quantizes data transfers in 256-Byte segments. When only one segment is copied, it fits entirely in the first-level data cache, so instruction-time tables in [19, Ch. 11] indicate that each **MOVE** instruction takes at least seven clock pulses and the **DBF** instruction takes at least three. Thus, the total execution time is at least 451 clock pulses/iteration, and the marginal cost of thread-context switching is at least:

$$2 \times \frac{451\ clock\ pulses/iteration \times 4\ iterations/kB}{33\ clock\ pulses/\mu S} = 109\ \mu S/kB$$

If more than one segment is copied, this calculation must be altered to include main memory **write** time. If writing a Longword takes eight clock pulses, then each iteration requires 515 pulses and the marginal cost is 125 μS/kB. If writing a Longword takes nine clock pulses, then each iteration requires 579 pulses and the marginal cost is 140 μS/kB. Notice that our measurements, reported in §3, lie about halfway between these two estimates.

B.2 HP 9000 Series 835

The Series 835 CPU implements Hewlett-Packard Company's proprietary Precision Architecture Reduced Instruction Set Computer (PA-RISC) technology [13,14,18,22]. It executes at a peak rate of 15 million instructions *per* second (MIPS) and has only one level of memory cacheing. This computer has separate, 64-kB instruction- and data caches that are two-way set associative, using virtual-memory addresses. Since the operating-system and each application program occupy unique ranges of a 48-bit virtual-memory address space, the caches may contain information for several processes at once, without needing to flush it during process-context switches. This makes them as fast as physical-address caches. Data are transferred between the caches and main memory in thirty-two Byte units using a *write back* policy, so copy rates for large blocks are bounded by transfer rates between the data cache and the CPU. Writing to the cache takes two instruction cycles; in the first cycle, the cache identifies the location to be updated and retrieves its current content; in the second cycle, new data are written. Reading from the cache also takes two instruction cycles,

28. Decrement a Data Register, then branch if its value was not zero.

but for a different reason: the cache retrieves data during the first cycle, but it is not copied to the CPU until the second cycle. Since the two actions are done by separate hardware, successive **read**s can be overlapped (*pipelined*), but successive **write**s cannot; thus, a **Store** instruction effectively takes twice as long as a **Load**.

The copy algorithm is an unrolled loop, consisting of sixteen **Load Word** (into a General Register) instructions, sixteen **Store Word** (from a General Register) instructions, and a conditional **Add Immediate and Branch** instruction, whic copies 64 Bytes *per* iteration like the following assembly language:

```
        LDI     {No. of 64-Byte segments},R1
label   LDW     60(0,src),R3
        LDW     56(0,src),R4
        ...
        LDW     4(0,src),R17
        LDWM    64(0,src),R18
        STW     R3,60(0,dst)
        ...
        STW     R17,4(0,dst)
        ADDIB,> -1,R1,label    ; Delayed "Branch" executes
        STWM    R18,64(0,dst)  ;   AFTER last "Store"!
```

This is the largest power-of-two expansion that can use all of the available General Registers, and it effectively quantizes data transfers in 64-Byte segments. Since the overlapped **Load** and **Branch** instructions effectively take one instruction cycle each and the **Store** instructions take two, the total execution time is 49 instruction cycles/iteration. Thus, the marginal cost of thread-context switching is:

$$2 \times \frac{49\ instruction\ cycles/iteration \times 16\ iterations/kB}{15\ instruction\ cycles/\mu S} = 104\ \mu S/kB$$

Notice that our measurements, reported §3, are about 4-12% greater than this figure.

Appendix C: Process-Context Switching Time

Some versions of UNIX have a *swtch*(2) operating-system service that lets an application program relinquish control of its CPU in favor of another process, and this provides a direct, measurable interface to the process-context switching mechanism. But the execution time of this mechanism must usually be measured indirectly, and the best method is a subject of debate. This appendix describes our technique.

An approximation used in at least two popular "bench mark" suites [17,21] employs two "clone" processes, produced by the *fork*(2) operating-sytem service, that repeatedly exchange 1-Byte messages through a pair of pipes, as shown in this *C*-language fragment:

```
int NSWITCH,  /* No. of context switches */
    PIPE[2];  /* File Descriptors of (bidirectional) pipe */
char MESSAGE; /* Arbitrary data */
...
while (--NSWITCH >= 0)
  {
    [Process 1]                    [Process 2]
    write(PIPE[0],&MESSAGE,1);     read(PIPE[0],&MESSAGE,1);
    read(PIPE[1],&MESSAGE,1);      write(PIPE[1],&MESSAGE,1);
  }
```

If these are the only two processes executing on a computer and their I/O is performed synchronously (the default behavior under UNIX), control of the CPU alternates between them 2 × NSWITCH times; hence, their combined "system time"[29] divided by 2 × NSWITCH is often taken as the mean process-context switching time. But this method has two flaws.

First, the measurement includes I/O activity, and the CPU time to accomplish it is not necessarily negligible. Its contribution can be eliminated by timing a "reference" loop that uses a unidirectional pipe in a single process, then deducting this measurement from the two-process result.

The second defect is more subtle. Given that the purpose of the measurement is to compare process- and thread-context switching, one must consider that the latter occurs directly, while the former is indirect. That is, process-context switching occurs only through operating-system intervention, in which control of the CPU passes from the execution context of an application program to that of the operating-system, then back to that of another application program, implying machine-state information is actually swapped twice. Moreover, the amount of information exhanged during this passage through the operating-system often greatly exceeds the amount needed to multiplex the CPU among coroutines in a single process, so the time it takes is not

29. Reported by the *times*(2) operating-system service.

necessarily negligible. We argue that this component should be measured separately and half of it should be deducted from the two-process measurement.

This could be done by timing a "null" operating-system service, if one existed, but measuring a very simple service is a good substitute. We used *getpid*(2), which returns the identification number of a calling process, because most versions of UNIX implement it by loading a pre-calculated value from a memory location into a CPU register (usually, in one machine instruction).

Hence, our operational definition of process-context switching time is:

Time to switch between two processes (as above)
- Time for a single process to perform equivalent I/O
- Half the time for a single process to call *getpid*(2)

Measurements were made under the conditions described in §3.1 and averaged over ten trials, where each trial includes 10,000 process-context switches. Table 17 shows the result.

Table 17. Process-Context Switching Time (μS)

CPU	I/O+Switch Time	- I/O Time	-½ *getpid*(2) Time	= Switch Time
370	1356.8	758.4	28.4	584.2
835	499.0	362.1	2.4	135.7