CS/TR-89-37.3

Multi-threaded Processes in CHORUS/MIX

François Armand, Frédéric Herrmann, Jim Lipkis, Marc Rozier

Chorus Systèmes
6, avenue Gustave Eiffel, F-78182, Saint-Quentin-En-Yvelines, France
Tel: +33 1 30 64 82 00, Fax: +33 1 30 57 00 66, Email: mg@chorus.fr

ABSTRACT

Interest in concurrent programming in recent years has spurred development of ‘‘threads’’, or ‘‘lightweight processes’’, as an operating system paradigm. UNIX-based systems have been especially affected by this trend because the smallest unit of CPU scheduling in UNIX, the process, is a rich and expensive software entity with a private memory address space. In this article we examine performance constraints affecting concurrent programs, including real-time applications, in order to understand and evaluate the demand for a new scheduling model. Although performance criteria differ sharply among various application domains, we conclude that a single thread model can provide efficient concurrent execution in a general-purpose operating system.

We describe the design considerations behind the thread-management facilities of CHORUS/MIX, a UNIX-compatible operating system built for distributed, real-time, and parallel computing. Mechanisms for processor scheduling and inter-thread synchronization must satisfy the needs of each of these three categories of concurrency. Extension of the traditional UNIX interface to the multi-threaded environment is an area of particular delicacy. CHORUS/MIX adopts novel approaches for signal handling and other UNIX facilities so as to ensure a smooth transition from sequential to concurrent semantics in applications.

KEYWORDS: concurrent programming, threads, performance, multiprocessing, real time, UNIX

Appeared in the Proceedings of EEUG Spring’90 Conference, Munich, Germany, 23-27 April 1990, pp.1-13.

1. Introduction

Recent years have seen an increase in the role of concurrent programming, a traditional tool for applications in which concurrency arises naturally, such as simulations, servers in distributed systems, and programs which interact with humans or hardware devices. With the recent availability of multiprocessors, parallel programming has become an important technique for increasing the speed of computation. Concurrency is most often invoked through programming of multiple threads of control that access a shared memory context. A variety of general- and special-purpose programming languages provide concurrency in their semantics.

Operating systems become involved in the support of concurrent programming for two reasons. First, applications are often written in low-level programming languages like C or assembler which do not support concurrency. Creation and management of execution threads must be arranged through run-time services of the operating system or system-level libraries. Second, and more important, multiprocessor hardware and real-time applications each impose requirements on the scheduling and resource management facilities provided by the operating system. The nature of the interaction between supervisory software and user programs, in light of these new requirements, has become a major topic of both academic and commercial operating system research in recent years. Potential solutions are constrained by considerations of compatibility, portability, and standardization. New features must be introduced into existing operating systems through graceful and non-disruptive extensions. It is important to distinguish the *interface* provided by a system from the *implementation* of the corresponding features. Ideally, a single interface – that is, a coherent set of operating system facilities – should support both conventional and concurrent programming, with or without real-time constraints, on uniprocessors and multiprocessors. The *implementation* of these facilities will vary according to the underlying architecture and specific system goals. Nonetheless, a uniform interface requires negotiation of conflicting performance requirements in several areas.

In this article we describe design issues and alternatives for thread management, including inter-thread communication, synchronization, scheduling, and exception handling. Many of these functions play prominent roles in concurrent language semantics, as well as in operating system design. Ada, for example, includes semantics for task creation, synchronization and exception handling. We will ignore language and programming issues, however, so as to concentrate on the operating system services needed to *support* concurrent programming environments. Focusing primarily on UNIX[†] derivatives, we consider some of the approaches taken in existing multithreading systems, including CMU's Mach[1,2], DEC's Topaz[3], and the SunOs Lightweight Process Library[4]. We present the thread interface designed for CHORUS/MIX, a UNIX-compatible real-time operating system which builds on the experience of earlier systems. The goal in CHORUS/MIX was to design a coherent set of thread management features which address the performance requirements of both multiprocessor and real-time programs in a unified manner.

2. Performance requirements and alternatives

Scheduling and resource management in the UNIX environment are defined in terms of the *process*, a rich software entity which incorporates a stream of instruction execution, a memory address space, an exception-handling environment, a set of access capabilities, and other information about the executing user program. Process creation and deletion are expensive functions, as is CPU context switching among running processes. Good performance in concurrent applications will be impossible to the extent that these functions are frequently invoked. An often-proposed solution is the addition of a new scheduling entity called a ''lightweight process'' or a ''thread'', which represents an executing stream (and perhaps a stack) but does not contain an

[†]UNIX is a registered trademark of AT&T.

address space or other resources. (This is the approach taken in Mach, Topaz, Amoeba and other systems.) Programs arrange concurrency by invoking multiple threads, all sharing a process' address space. This seems attractive in the UNIX world because existing program development tools, designed to compile and link sequential programs for a monolithic address space, can be extended to multi-threading without change, albeit with some awkward limitations (which we discuss in later sections).

However, adding a new unit of scheduling constitutes a fundamental change to the heart of the system interface, with repercussions in many areas. Extending signal handling, memory and resource management, and process control features into this new framework requires resolution of a number of difficult issues. Before taking this path, then, it is worthwhile to examine the performance issues that lead to the demand for lightweightness, and to consider alternative approaches. We will look closely at two application domains which are highly sensitive to scheduler performance – computations with fine-grain parallelism, and real-time systems.

Requirements for multiprocessing. Performance of computation-intensive programs on multiprocessors is affected by two related factors.

- *Fine granularity.* As multiprocessors become available with increasing numbers of processing elements, the challenge of partitioning the computational work of an application to fully exploit the hardware resources becomes greater. Amdahl's Law states that the number of processors that can be effectively used to reduce the elapsed execution time of a computation is limited by the percentage of the program's execution that is spent in serial (or less parallel) sections. Serial execution is often a consequence of the high overhead of parallelization. If the time required for initiating and terminating parallel execution is much greater than that of the computations to be performed, then the extra available processors are of no use. The desire to parallelize fine-grain operations leads to a demand for very inexpensive fork/join functions. Fine granularity occurs in a wide variety of situations, including object-oriented programming and concurrent functional languages as well as numerical computation.
- *Flexible semantics.* In practice, the fineness of granularity attainable is a consequence of the semantics of the concurrent threads. In Ada, tasks are subject to priority-based preemption and elaborate rules for exception handling and termination. The run-time system may incur considerable bookkeeping expense when tasks are created or destroyed. By contrast, most Fortran systems with MIMD multiprocessing extensions create simple threads for parallel DO loops which must run to completion (cannot block or switch processors) and have little semantic baggage. Initiation of parallel execution requires a very small number of instructions in common cases. Other concurrent languages and tasking libraries used from C or assembler occupy the area between these two extremes.

In order to support a wide range of task granularities and adapt to various semantic requirements, thread management for multiprocessing is generally implemented in user mode – usually in general or language-specific libraries – rather than in the operating system. An apparent drawback is that two separate schedulers must be implemented, one at the system level and one in user level code. However, no centralized thread scheduler can alone handle the granularity needs that arise in parallel programming systems. Creation and deletion of threads in Mach and Topaz require between five hundred and one thousand instructions on a DEC Vax computer, according to measurements made by their respective designers[2,5]. The semantics of most concurrent languages allow threads to be created directly, without system intervention, far more cheaply. This is particularly true in the case of Fortran parallel DO loops, where fine granularity is often crucial. Thus reliance on supervisor-level threads limits the utility of a system for other than coarse-grain parallelism.

Requirements for real-time. Robotics, process control, and other real-time domains pose the following requirements.

- *Fast response.* A key metric of a real-time system is the maximum time required between the occurrence of a hardware or software event and the execution of the first instruction of application code that handles the event.
- *Scheduling control.* Real-time systems often rely on thread priorities to control CPU scheduling. For example, a thread that interacts with a physical process will be subject to tighter timing requirements, and thus assigned a higher priority, than one that is spooling or analyzing data. The thread semantics should include a system-wide priority level which may be changed by the user.

  An alternative, *deadline scheduling*, has been proposed in the research literature and used in specialized environments. Here, information concerning the real-time tasks' timing characteristics and deadline requirements is communicated to the scheduler instead of task priorities.
- *Determinism.* Response should be predictable as well as fast, since designers must accommodate the upper bounds on software operation timing. Thread priorities must be strictly enforced, by preemption when necessary. Thus at each moment in time, the highest-priority ready thread is active in execution. Extraneous context switches, interrupts, and other asynchronous activities can prevent tight upper bounds on time requirements.

Less demanding applications (i.e., without real-time constraints) may nonetheless make use of priorities or related features. In the next section we will consider ramifications on system design posed by these requirements.

Alternatives for thread management. There are three common approaches for thread management.

(1) *User mode threads within a process.*

A library package which multiplexes a single UNIX-style process to implement multiple threads can provide a useful concurrent programming environment. Threads execute as coroutines, with optional timer-based preemption. While the operating system kernel need know nothing about the threads' existence, concurrency can be increased if asynchronous kernel operations are provided. While an I/O system call is being processed or a page fault is being resolved on behalf of one thread, this enhancement would allow other threads to execute within the same process. UNIX signals can be used to inform the thread scheduler of the completion of the asynchronous operation. The SunOS Lightweight Process (LWP) Library is an example of this approach, though without general asynchronous system calls or asynchronous page fault resolution. Other implementations exist within specialized or language-specific contexts.

Threads within a single process cannot make use of multiple processors, nor can they satisfy the requirements of real-time applications. Within these limits, however, this is an attractive approach because it minimizes disruption to the operating system interface.

(2) *User mode threads scheduled across multiple processes.*

We concluded earlier in this section that parallel computation on multiprocessors is most effectively implemented with threads managed and scheduled in user mode. A mechanism quite similar to that of the previous paragraph suffices. Multiple processes – at least as many as the number of hardware processors to be used – are needed, instead of only one. These processes must provide access to some amount of common memory. In all other respects, use of ''heavyweight'' processes is acceptable because they will be created and destroyed very rarely, usually only at the beginning and end of program execution. Various

UNIX versions have supported shared memory regions among processes for years – though sometimes in a manner which limits usability – and this is still a relatively minor extension to underlying process model. Although processes retain their individual address spaces, large amounts (in theory, all) of the address space may contain shared regions. Controlled sharing of other process attributes, such as file access capabilities, can be provided as well but is not strictly necessary[6].

The processes are used as thread executors. Let us first imagine that each of the processes created by a parallel program is locked onto one processor and will never be preempted, by virtue of a special provision in the system scheduler. The various threads of the user program are then scheduled onto these process/processor pairs, just as in a standard operating system processes are scheduled onto a processor. A queue of executable threads is maintained in shared memory and accessed by each process/processor, executing the code of the user-level thread scheduler between executions of user program threads. The scheduler code might be loaded from a thread-management library or emitted by a parallel-language compiler. Scheduling overhead is minimized, and task granularity is limited only by the parallel language semantics and the machine architecture. Use of shared regions for thread-specific storage allows threads to context switch among processes with low overhead, if necessary.

This is an attractive approach for general computation on multiprocessors, and it has been used in both commercial and research systems (for example, the Sequent Balance/Symmetry[7] and NYU Ultracomputer[6]). Because the lightweightness of the scheduling entity provided by the operating system is unimportant, fundamental revisions to the operating system paradigms are unnecessary. It may be beneficial, however, to add some provision for grouping the processes that act as thread executors for one program. Processes are not in general locked onto processors, and so process scheduling becomes an issue. A process aggregate or ‘‘container’’ would be useful so that processes within one program might be scheduled onto processors together. Another motivation arises in distributed systems that support process migration. For high-bandwidth communication over shared memory, the collection of cooperating processes should remain on the same (multiprocessor) node.

Real-time requirements, however, remain unsatisfied by user-managed threads.

- Because the central scheduler is not aware of individual threads, it cannot recognize or enforce thread priorities. A user-mode thread scheduler might implement priorities, but only at the level of an individual job. Deadline schemes also require centralized thread scheduling.
- Events recognized within the operating system kernel affect thread state. A hardware interrupt could cause a blocked high-priority thread to become runnable, but the system process scheduler cannot place that thread into execution.
- Both of these problems might be addressed by binding selected high-priority threads to specific processes and thus using process priorities to simulate system-level thread priorities. But we would begin to lose the advantage of lightweight threads. Context switching among processes is expensive on many architectures because address mapping hardware must be updated and often caches must be flushed.

While user mode threads provide a simple and useful paradigm for multiprocessing, they do not appear to be able to support the combination of prioritized threads and tight response time requirements that arise in real-time applications.

(3) *Threads scheduled by the operating system.*

Thus real-time considerations mandate the use of threads implemented and managed through the central system scheduler. In CHORUS/MIX, the memory address space and resource ownership functions remain with the process, but a process may contain an arbitrary number of threads, and threads rather than processes are scheduled onto hardware processors. With no intermediate thread-management layer in user mode, the system scheduler can enforce thread priorities directly and provide the deterministic scheduling required for real-time.

On multiprocessors, in the absence of real-time constraints, system-level threads may be used as executors of the parallel program activities which are created and scheduled in user mode. A multi-threaded process acts as a ‘‘container’’ of the system threads which participate in execution of a concurrent program. We conclude that a single interface model can effectively address the needs of several different categories of concurrent applications.

3. Threads in CHORUS and CHORUS/MIX

CHORUS® is a family of operating systems based on a minimal real-time nucleus which provides low-level services for distributed processing and communications[8]. The nucleus can be scaled to run on a variety of hardware configurations, including embedded boards, multicomputer and multiprocessor configurations, networked workstations, and dedicated servers. CHORUS operating systems are built as sets of independent, dynamically-loadable servers that rely on the generic services provided by the nucleus, i.e., thread scheduling, network transparent inter-process communication (IPC), optional virtual memory management, and real-time event handling. CHORUS-V3, the current version, was developed by Chorus Systèmes and has been commercially available since early 1989. Earlier versions were designed and implemented by the Chorus research project at INRIA between 1979 and 1986. Work on UNIX integration and compatibility in CHORUS began in 1984.

The physical support for a CHORUS system consists of a set of *sites*, interconnected by a communication network. A site is a tightly coupled grouping of physical resources: one or more processors, memory, and attached I/O devices. There is one CHORUS nucleus per site.

The *actor* is the logical unit of distribution of processing and of collection of resources in a CHORUS system. Actors in CHORUS are similar to *tasks* in Mach. An actor constitutes an execution environment, including a protected address space, for one or more *threads*. Each actor is tied to a single site. Within a site, threads of multiple running actors are scheduled by the nucleus as independent activities. Thus multiple threads of an actor may run in parallel on multiprocessor sites. Threads of the same actor may communicate and synchronize through shared memory. In addition, CHORUS offers message-based facilities which allow any thread to communicate and synchronize with any other, whether within the same actor, across actors, or across sites. Message exchange under CHORUS IPC may be either asynchronous or by demand-response, also called *remote procedure call* (RPC).

The CHORUS/MIX operating system is composed of a CHORUS-V3 nucleus in combination with a set of subsystem servers that implement a System V-compatible UNIX. Each UNIX process is implemented by one CHORUS actor; hence the multi-thread nucleus model extends naturally into the UNIX layer. The traditional UNIX services are augmented in CHORUS/MIX with facilities for distributed, parallel, and real-time computing. In order to distinguish the thread interface provided by the CHORUS nucleus (used primarily by subsystem programmers) from the thread interface provided by CHORUS/MIX (used by UNIX applications programmers), CHORUS/MIX threads will be called *u_threads* (short for ‘‘UNIX threads’’) in the remainder of this paper.

® CHORUS is a registered trademark of Chorus systèmes

The u_thread management interface has been defined to satisfy two major objectives:

- to provide a low-level, generic interface which can satisfy a variety of needs, including support of existing thread interfaces and language-dependent packages. This was also one of the CHORUS nucleus objectives, and as a result the thread management interfaces at the nucleus level and UNIX level in CHORUS are quite similar.
- to have minimal impact on the syntax and semantics of UNIX system calls so that existing mono-threaded programs can easily become multi-threaded programs.

The basic u_thread management interface includes primitives for creating and deleting u_threads, suspending and resuming their execution, modifying their priorities, obtaining the identification of the current thread, and obtaining the CPU context (register values, etc.) of a blocked thread. All of these services are low-level and incorporate a minimum of semantic assumptions. Policies regarding stack management and ancestor/descendant relationships among u_threads, for example, are left to higher-level library routines which are provided to make u_thread usage easier for programmers.

4. Synchronization

Threads running on a multiprocessor can coordinate directly through shared memory, using hardware synchronization primitives like *test-and-set* or *fetch-and-add* when necessary. Nonetheless, operating system services play an important role in synchronization. In this section we consider two basic forms of synchronization, mutual exclusion and event notification. More advanced coordination functions can be built on top of these, or through use of other system facilities. Ada-style synchronous rendezvous, for example, can be implemented within an actor (or process) using shared memory, or in a more general manner using the CHORUS RPC facility. Functions described in this section are identical in the CHORUS nucleus and in CHORUS/MIX.

Performance goals for synchronization functions again differ sharply between the two application domains of parallel computation and real-time.

- *Multiprocessing.* Most important is to minimize the overhead of synchronization. In particular, ''easy'' operations like obtaining a lock which is already free should require only a few machine instructions.
- *Real-time.* Overhead is still a consideration, but determinism is again crucial for real-time. When one thread releases a lock, it is assumed that the highest-priority waiting thread is scheduled. Some systems go further and allow a releasing thread to designate a specific target thread (of the same or higher priority) that is guaranteed to execute immediately.

Overhead on multiprocessors can be minimized through *busy-waiting* synchronization, in which a thread tests a condition or a lock in shared memory, and, if unavailable, continues testing as long as necessary. This is especially effective if the synchronization is satisfied immediately or very soon, because no context switching or other software overhead is introduced. However, when resources are not available busy-waiting can waste processor time, tie up the memory subsystem, and even lead to deadlock in some cases. Therefore we may want to switch the waiting thread off of its processor and queue it until the condition is satisfied. This is known as *blocking* synchronization. Various hybrids of busy-waiting and blocking have been proposed in order to combine the performance advantages of each[9].

In CHORUS, synchronization functions and implementation strategies for multiprocessing are left to user-level libraries and execution environments. The operating system kernel merely maintains the thread state (ready vs. blocked) and queues blocked threads, thus maintaining flexibility for the user layer to optimize for specific requirements or hardware features. As we shall see, the kernel interface is designed to ease the job of writing synchronization library code that is free of *race conditions*, timing-sensitive errors that can unpredictably cause the synchronization to go awry.

For real-time purposes, however, synchronization must interface directly with the thread scheduler. Hence we also provide a set of synchronization functions directly in the operating system nucleus. Only blocking functions are provided, since busy-waiting is incompatible with deterministic scheduling.

4.1. Semaphores

The *binary semaphore* or *lock* provides mutual exclusion for protecting the integrity of shared data. A generalization, the *counting semaphore*, allows a fixed number (not necessarily one) of concurrent accesses to a resource. Support for these functions presents two demands to the CHORUS nucleus. First, we must support semaphores directly in the nucleus, so that real-time constraints may be met. Second, we need a race-free mechanism by which user-level semaphores or other synchronization routines may cause threads to be blocked and restarted by the nucleus. However, we can address both with a single nucleus facility.

The *P* and *V* (counting semaphore *obtain* and *release*) functions were defined by Dijkstra as follows[10]. The semaphore variable *sem* contains an integer counter.

P(sem) – *atomically:* decrement the counter, check the result
– if the counter value has become negative,
block the current thread
– return

V(sem) – *atomically:* increment the counter, check the result
– if the counter value remains less than one,
awaken the waiting thread with the highest priority
– return

The function of the P and V operations depends on how the counter is initialized. If it is set to one, then P and V implement mutual exclusion. P is executed on entry to a critical section, V on exit, and the result is that no more than one thread can execute in the critical section at a time.

However, if we associate a private semaphore with each thread, and initialize the counters to zero, then P and V can be used to perform the functions ‘‘block the current thread’’ and ‘‘release the previously-blocked thread’’, respectively. When any user-level synchronization routine checks a condition, perhaps busy-waits, and then determines that it is time to free the processor to execute a different thread, it invokes P on its private semaphore. Later, when the condition has been satisfied, the routine can resume the suspended thread with V on that thread’s private semaphore. Use of the counting semaphore here lends robustness in the face of unforeseen timing circumstances on multiprocessors. Suppose that a decision has been made that the current thread must block but it has not yet performed its P operation, and meanwhile a second thread satisfies the relevant condition and calls V. We must ensure that when the P is finally executed, the first thread will not be made to wait forever for a corresponding V that will never occur. In fact, the V is effective regardless of the order of P and V. In this example it will increment the counter to one, and the subsequent P will immediately return without waiting. Thus the counting semaphore included in the nucleus for direct use in real-time situations also serves to provide reliable thread control for general user synchronization.

4.2. Event notification

Often threads must synchronize because one activity cannot logically proceed until it obtains data, or notification of an internal or external event, or other status information from a second thread. Mutual exclusion alone cannot provide event notification. For synchronizing access to discrete resources that occur in finite allotments, e.g. space in bounded buffers, counting semaphores can be used for ‘‘notify when not full’’ and ‘‘notify when not empty’’. But more general forms of event synchronization are useful in many applications. In the following subsections we describe two of the most popular mechanisms for event notification, and consider some of their strengths and weaknesses.

4.2.1. Condition variables. Topaz, the SunOS LWP Library, and the Mach ‘‘C Threads’’ library all provide some variation of Hoare’s *monitors* with *condition variables*[11] for event notification. A monitor is a package of functions protected with a binary semaphore. Under Hoare’s discipline, each thread performing an operation on a shared object must first enter the corresponding monitor (i.e., lock the semaphore). Each condition variable declared within a monitor is associated with some logical condition which may affect threads’ ability to continue processing. When a thread in the monitor needs to wait for a condition to be satisfied, it issues a *condition_wait*, thus suspending the current thread and also releasing the semaphore. Later, when another thread in the monitor satisfies the condition, it may issue either *condition_signal* to wake one waiting thread or *condition_broadcast* to awaken all threads currently blocked on the specified condition variable. Awakened threads automatically re-enter the monitor before proceeding. Thus the shared program objects which make up the condition being awaited are protected at all times with mutual exclusion.

The power of this paradigm – as well as its drawbacks – stem from the coupling of event notification with mutual exclusion. By defining composite functions as *atomic*, we can solve synchronization problems that would arise in use of more primitive functions. This arises in two areas.

- *Avoiding deadlock on condition_wait.*
  By specifying that the *condition_wait* operation release the monitor (semaphore) *atomically*, we can avoid an area of potential deadlock. If the *condition_wait* and semaphore release were independent nonatomic operations, then they would have to be done in some sequence. If the *condition_wait* comes first, then the semaphore would remain locked, and no other thread would ever be able to enter the monitor to satisfy the awaited condition. If the semaphore release comes first, then there would be a danger of a race condition similar to that discussed in Section 4.1: an intervening thread could issue the *condition_signal* before our original thread issues its *condition_wait*. Condition variables contain no counters, and no other form of memory, so the eventual wait will indeed wait forever because it has missed the corresponding signal. In lieu of keeping information in the condition variable, atomicity is used to solve the problem.

  The Topaz and Mach implementations of condition variables on multiprocessors guarantee atomicity in this case.

- *Deterministic notification on condition_signal*
  A second atomicity issue can effect real-time performance. Experience suggests that when a thread waiting on a condition is signalled, that thread is ready to do useful or critical work and should be scheduled before the signalling thread resumes and before other threads (of the same priority) are allowed to enter the monitor. Assuming that a thread issuing *condition_signal* or *condition_broadcast* holds the monitor semaphore, then it is important that the semaphore be released *atomically* with the signal or broadcast operation, then relocked before the thread proceeds. Otherwise, we find that neither of the two possible orderings allow optimal scheduling. If the semaphore is released first, then other threads waiting for the monitor may enter before the target thread is made executable, thus delaying its progress. If the condition signal takes effect first, then the signalled thread will awaken only to block immediately in attempting to re-enter the monitor. The penalty is several extra context switches.

  Some specialized real-time monitors enforce a guarantee that a signalled thread will execute immediately on invocation of the signal operation. Thus no intervening threads can enter the monitor to tamper with the logical condition that led to the issuance of the signal. The condition variable interface is sufficiently powerful to implement this stronger functionality if desired.

However, no general-purpose thread system known to the authors includes an implementation of condition variables that enforces *any* form of atomicity on condition signal or broadcast.

Monitors and condition variables have serious limitations in other areas. One problem is immediately apparent: the monitor imposes serialization which may be logically unnecessary and may be detrimental to good performance on a multiprocessor. Designers of large parallel computers strive to avoid the need for mutual exclusion, using specialized hardware features like *fetch-and-add* in combination with software mechanisms. On such systems it is possible that a number of threads test a condition fully in parallel without threatening the integrity of shared objects. If the condition is satisfied, the threads proceed in parallel, otherwise they all block until a condition-satisfied broadcast allows them to resume – again, all in parallel. Using the condition variable interface described, there is no way to program the synchronization without spurious serialization.

Even where mutual exclusion *is* required, it may not always fit the model underlying the monitor/condition variable paradigm. Device drivers in CHORUS reside in subsystems, not in the nucleus, and event notification between interrupt routines and user-level threads is sometimes required. But mutual exclusion in this case is based on hardware interrupt masking, rather than software semaphores, and so condition variables are not usable without loss of performance. Further, there may be situations in which a condition is logically coupled to multiple semaphores rather than just one. Finally, condition variables may be difficult to use and understand in complex programs. There are subtle interactions between the condition and semaphore that can give rise to unexpected bugs or performance problems. (Birrell[12] discusses experiences in this area.)

Thus monitors and condition variables do not appear adequate for general-purpose thread synchronization.

4.2.2. Latching Events. Another popular event notification scheme involves synchronization objects which we shall call *latching events*. These differ from condition variables in that each event contains a persistent state which has two possible values, *OCCURRED* and *NOTOCCURRED*. Three operations are defined on latching events. Here *ev* is an event variable.

event_wait(ev)
– if *ev* is in state *NOTOCCURRED*, enqueue the thread and wait
– return

event_signal(ev)
– if *ev* is in state *NOTOCCURRED*,
change the state to *OCCURRED* and awaken all threads enqueued on *ev*.
– return

event_reset(ev)
– if *ev* is in state *OCCURRED*, change the state to *NOTOCCURRED*
– return

When an event is signaled, it stays signaled until reset. The deadlock problem in the *condition_wait* operation above has no analog here, since it makes no difference whether an event_wait is followed or preceded by its corresponding event_signal. Otherwise latching events are similar in operation and usage to the broadcast mode of condition variables. However, this is an independent, primitive mechanism. No linkage to any other synchronization mechanism is involved, though events may be combined with semaphores or other synchronization devices by the programmer as required. Latching events are more flexible than condition variables, and avoid problems of spurious serialization. However they have no analog to the signal mode of condition variables, and thus cannot entirely supplant the latter as a general notification device.

CHORUS includes sufficient tools to allow implementation of condition variables, latching events, or other synchronization devices at the subsystem or user level. The scheduler does not currently provide such a feature directly. Unfortunately, little experience exists to guide the selection of synchronization primitives in a general-purpose operating system interface. The situation is particularly chaotic in real-time systems, because there is no consensus either on synchronization functions, scheduling paradigms, or the relationship between the two. Current real-time monitors tend to provide a multitude of *ad hoc*, redundant facilities for thread coordination. Investigation and experimentation are continuing in the search for software devices that can address this problem.

5. Thread Scheduling and Responsiveness

5.1. Scheduling modes

Real-time applications require deterministic thread scheduling which can be controlled by the user; time-sharing environments, on the other hand, require time-slicing and dynamic adjustment of priorities for good response times. The CHORUS nucleus provides both modes.

Scheduling is preemptive: at any given time, the running thread is always the ready thread with the highest priority. Scheduling decisions are based on *absolute* thread priorities within a system-wide range of priority values. The absolute priority is calculated as the sum of the priority of the owning process and the *relative* priority of the u_thread within the process. Relative priorities are useful for tuning of priorities among various applications and system components without disturbing the scheduling of the u_threads within a component.

Above a threshold priority known as SLICE_PRIO, scheduling within a priority level is first-in first-out (FIFO); a thread that yields control or is preempted is placed at the end of the queue for its priority. FIFO scheduling provides determinism. Combined with the capability to alter thread priorities at any time, this gives real-time users full control over scheduling both within a program and across the system.

Below the SLICE_PRIO threshold, threads are subject to time-slicing. CHORUS/MIX arranges that all u_threads are within this range by default. Thus real-time application programmers must raise the priorities of their u_threads explicitly to avoid time-slicing. Threads of the UNIX subsystem servers in CHORUS/MIX execute above SLICE_PRIO and are not subject to time-slicing. However, a range of priority values is available above that of the UNIX servers. This allows real-time application u_threads to execute at higher priorities than the server threads and thus to preempt them.

5.2. Interruptability

Real-time responsiveness in traditional UNIX systems is limited by the fact that system calls are noninterruptable. A high-priority activity might be delayed for a period equal to the longest-running system call before it can respond to an event, even though the issuer of the system call executes at a low priority.

The subsystem servers which implement UNIX system calls in CHORUS/MIX are themselves multi-threaded; thus every system call is interruptable at any point. Real-time threads need never wait for completion of a system call or any other function in another process. Shortly (section 6) we will mention a restriction on concurrent execution of certain short-running UNIX system calls *within* a process. This is an implementation artifact, however, and can be modified if performance problems result.

6. Adapting UNIX System Functions

In several respects, UNIX system services exploit the fact that memory, resource ownership, and signal delivery are tied to the unit of CPU scheduling. In adapting these services for multi-thread processes, we would like to preserve the existing interface and function to as great a degree as possible. Maximizing compatibility will minimize the difficulties faced both in converting old programs and writing new programs to use concurrency.

In some areas compatibility is problematical, but the basic UNIX process management functions extend readily. The *fork* primitive creates a child process, but with only one u_thread, corresponding to the u_thread in the parent process that executed the *fork*. *exec* overlays the current process memory with a new program, which begins execution as a mono-threaded process. Other related functions retain their traditional semantics or are adapted in minor ways.

Concurrently-executing system calls within a process could interfere with each other in such a way as to violate the semantics of some or all. However, serializing system calls in each process would limit concurrency to an unacceptable degree. CHORUS/MIX multiplexes system calls within a process according to the following policies.

- A u_thread executing a system call never prevents other u_threads from running in user mode.
- Execution of extended (non-UNIX) CHORUS/MIX system calls relating to CHORUS IPC or u_thread management can be completely multiplexed
- UNIX system calls that wait for I/O or other external events, including *read*, *write*, *open*, *pause*, *wait*, etc.) may be multiplexed with any other system call after the calling u_thread has blocked. (These are precisely the system calls which may be interrupted by signals in UNIX.) Thus multiple u_threads may invoke concurrent I/O operations.
- Only one u_thread in a process may be actively in execution of a UNIX system call at any moment. This restriction is imposed to protect the semantics of the UNIX emulation. The result is that system calls which affect the entire process, like *fork* and *exec*, and some others which execute very quickly, like *getpid* (obtain process id), are serialized.

We mentioned previously that system calls invoked by different processes can execute concurrently without limitation.

6.1. Signals – Exceptions and Asynchronous Events

UNIX uses *signals* to manage both synchronous exceptions caused by errors in user code (i.e., divide by zero), and for asynchronous events such as expiration of a time interval or completion of an asynchronous I/O operation. Most signals are initiated by the operating system kernel, but user processes may also send signals to each other using a standard system call. Delivery of a signal acts like a software analog of a processor interrupt. The normal flow of code is suspended and a user-specified *signal handler* routine begins execution in the same process context (address space). If the handler returns to the system, execution of the original program resumes where it left off. Some signal handlers instead branch elsewhere in the program, usually to a caller of the interrupted routine, and resume program execution directly. (The standard C library includes a package of routines, called setjmp and longjmp, for non-local branching that can ‘‘unwind’’ the call stack.) The user program may choose to ignore a certain signal instead of providing a handler. If it does neither, then receipt of a signal terminates the process in most cases.

The introduction of multi-threaded processes mandates a reconsideration of signal handling. This has proved to be the thorniest area in extending the UNIX semantics, especially as pertains to asynchronous signals (as opposed to program exceptions). Traditionally, signals are sent to processes, and signal handlers are declared on a per-process basis. Which thread should receive delivery of a signal directed to a multi-thread process? More fundamentally, one might question whether signals are an appropriate mechanism at all in a multi-threaded environment. The argument is that asynchronous interrupts are confusing, difficult, and bug-prone in user programs, especially when locks or other synchronization facilities are being used by the interrupted code. In many cases, signals can be replaced through use of multiple concurrent threads. Instead of initiating an asynchronous operation, performing other tasks concurrent with the operation, and awaiting the signal at completion, one can create a concurrent thread to perform the operation using simpler synchronous system functions which return control only after the operation is complete. Further, thread-to-thread notification and abort can be provided with a mechanism that is less intrusive than an asynchronous interrupt.

Signals play an important role in UNIX and must be extended for multi-thread programs in a compatible and graceful way. Nonetheless some designers have attempted to minimize reliance on asynchronous interrupts in user programs. In both Topaz and the SunOS LWP Library, a single thread is designated to receive signals of a particular type on behalf of all threads in the process. Signal-receptor threads are dedicated to this purpose; they do not perform computational work as well. Thus worker threads are immune from interruption, though exception handlers may be defined to handle synchronous program errors. In the future, Mach will use a similar mechanism. Currently, signals in Mach are handled pseudo-randomly by any thread in the process[2].

CHORUS/MIX adopts a new approach to signal delivery and management. The goal is simple: *each signal should be processed by (i.e., on the stack of) the u_threads which have indicated that they want to process that signal.* Each thread has its own signal context (signal handlers, blocked signals, etc.) and system calls used to manage this information affect only the calling thread. This design arose from considerations of common signal usage in existing mono-threaded programs, and the desire to keep the same semantics in multi-threaded programs.

- Many applications use the longjmp non-local branch to abort a computation loop on receipt of a signal. This can work correctly only if the handler executes in the thread which is to be interrupted.
- When a terminal user types the control sequence which aborts the current foreground process, a certain type of signal is sent to that process. In a multi-threaded process, where one thread writes to or reads from the terminal while other threads perform computation, the programmer should be able to decide which thread(s) will process this signal: certainly the thread that interacts with the terminal, but possibly some of the other threads as well.

Furthermore, the per-thread signal design seems to ease programmer difficulties and maximize flexibility.

- If signal handling were assigned to only one u_thread at a time, the programmer would have to manage the complexity of redispatching caught signals to other u_threads.
- While asynchronous interrupts are sometimes difficult to manage, concurrency also increases the level of complexity, and in fact the issues are similar. The programmer must deal with asynchronous access to shared variables, and the resulting race condition bugs, in both cases. Programming a simple signal handler can be much more straightforward than coordinating threads, especially in a section of program that is not otherwise using concurrency.
- Signals provide a low-overhead, minimalist mechanism that can be used to implement any desired semantics for a specific language or programming environment. Standard library signal handlers may transform a delivered signal into a message, create a new thread to respond to an asynchronous events, or pass on the signal to another thread, as desired.

In order to determine which u_threads should receive signal delivery, CHORUS/MIX distinguishes two types of signals:

(1) *Signals for which the identity of the target u_thread is non-ambiguous.* These are signals corresponding to synchronous exceptions, and also those issued in response to a system call issued by a specific u_thread. Signals for expiration of a timer interval or completion of an I/O operation are in this category. In these cases the signal is sent only to the u_thread concerned. If that thread has not declared a handler or opted to ignore the signal, the default action (usually termination) is taken for the entire process.

(2) *Signals which are logically sent to an entire process.* For some purposes a process is viewed as a single entity. Signals sent by device drivers (e.g., the keyboard ‘‘abort process’’ sequence), and signals directed to a process by a user program, are broadcast to all u_threads of the target process. Each u_thread which has declared a signal handler for the particular signal will receive delivery. If no u_thread either declares a handler or ignores the signal, the default action is again taken relative to the entire process.

Thus signal semantics and usage extend smoothly from the traditional UNIX mechanisms to the multi-thread environment of CHORUS/MIX. The per-u_thread signal status adds to the size of the state information that must be initialized and maintained for each u_thread. However, no extra expense is added to the u_thread context switch, so the real-time responsiveness of the system is not compromised.

Signals are used to perform one new function in CHORUS/MIX: an inter-thread interrupt facility. Any u_thread may send a signal to another u_thread of the same process using a new system call. Designers of previous thread systems have preferred to introduce new facilities for inter-thread notification. The *alert* facility of Topaz, for example, effectively sets a flag which is then polled by various coordination functions in the target thread. However, the signal interface already exists and cannot be suppressed. It is simpler to provide a single mechanism than to add a new concept whose semantics would complicate the signal semantics. Again, individual programming environments may adopt their own conventions for inter-thread interrupts.

6.2. Thread context

A final compatibility problem affects UNIX programs and libraries that are written in C. Variables in C programs which are declared as global or static are bound to a single storage location in a process. Hence in a multi-threaded process, those variables become globally shared. However, they are sometimes used in a manner that requires a separate instance of the variable in each thread. The best-known example in UNIX is the *errno* variable, which reports error codes returned from the most recently executed system call. Other examples occur in library packages which must maintain state information between successive calls in a thread. The C language makes no provision for thread-private instances of global or static variables. (Automatic variables are thread-private because each thread has its own private stack.)

There are a number of ways of addressing this deficiency.

- A different language, with support for private thread context, might be used.
- A pointer to a u_thread-private data area might be passed as an argument to all routines, either explicitly or implicitly as with C++ member functions. This solution is awkward and potentially expensive.
- Private u_thread information could be maintained at a fixed location in the process address space, which is remapped on each context switch. As discussed earlier, this could add considerable expense to the context switch and sacrifice much of the advantage of threads, especially in real-time applications.

- A nucleus facility could be invoked on each access to obtain the current thread identifier, which could then be combined with a hashing scheme to obtain the address of thread-private storage.
- A hardware register might be reserved to hold the address of this private area. This is not feasible on all processors, and in any case requires that compilers be modified.

The solution in CHORUS/MIX has been to extend the processor context with a set of *software registers*. These are implemented in the CHORUS nucleus; one register is provided per privilege level, as is done for stack pointers. The software registers are saved and restored at each context switch. In the usual implementation, with two privilege levels (user and supervisor), the added cost on context switch is equivalent to a copy of four pointers. Each u_thread may read and modify the value of the software register at the corresponding privilege level. Where possible, the reading of these software registers is implemented inline, without a trap to the system.

Several independent routines in a library may need thread-private static storage within a single thread. Hence a package is provided in the C library to manage multiple uses of an individual software register. Macros and inline functions are used to minimize expense and facilitate modification of library routines for multi-threaded use. No modification at all is required in user programs which merely access *errno* or similar standard variables.

7. Conclusion

Extending a general-purpose operating system to support concurrent programming requires careful attention both to performance requirements and to the compatibility and usability of the system interface. Within the framework of a distributed and parallel environment, the particular design goals for thread support in CHORUS/MIX were the following.

- *To satisfy real-time needs.* u_threads are implemented and scheduled by the operating system according to standard requirements of real-time computing: deterministic and responsive behavior with respect to priorities, scheduling, and synchronization. The CHORUS/MIX design differs substantially from that of other general-purpose thread-management systems.
- *To achieve a graceful integration with standard UNIX semantics.* Multi-threaded CHORUS/MIX processes are still UNIX processes; each standard UNIX feature has been extended in a coherent manner, as required. The major focus was on signal handling and parallel invocation of system calls. Our signal handling design is deterministic and gives the user full control over signal behavior in a multi-threaded program. This design is again a departure from earlier systems that augment UNIX with concurrency features.
- *To support parallel architectures.* u_threads can be used along with user-mode scheduling to provide efficient parallelization of a wide range of granularities of tasks in compute-intensive applications.

Work is still in progress in two important areas. First, we must develop low-cost synchronization methods which allow programs to scale to large number of processors while satisfying the determinism requirements of real-time programs. Finally, we have entirely omitted the crucial area of debugging support in this article. Operating system features in support of debuggers (including remote debuggers) for multi-thread programs are currently under investigation and experimentation.

8. Acknowledgements

Many thanks to Vadim Abrossimov, Michel Gien, Marc Guillemont, Marc Maathuis, Will Neuhauser and Francois Saint Lu who have contributed, each with a particular skill, to the CHORUS/MIX u_thread management specification and implementation.

References

1. Eric C. Cooper and Richard P. Draves, ''C Threads,'' Technical Report, Carnegie Mellon University, Pittsburg, PA (March 1987).
2. Avadis Jr. Tevanian, Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michäel W. Young, ''Mach Threads and the Unix Kernel : The Battle for Control,'' Technical Report CMU-CS-87-149, Carnegie Mellon University, Pittsburg, PA (August 1987).
3. Paul R. McJones and Garret F. Swart, ''Evolving the UNIX System Interface to Support Multithreaded Programs,'' Technical Report 21, DEC Systems Research Center, Palo Alto, CA (September 1988).
4. Sun microsystems, ''Lightweight Processes Library,'' SunOS Release 4.0 Reference Manual, Sun microsystems, Mountain View, CA (November 1987).
5. Garret Swart and Roy Levin, ''An Introduction to Threads and CMA,'' Presentation to IEEE POSIX P1003.4 Realtime Extension for Portable Operating System, DEC Systems Research Center, Palo Alto, California (January, 1989).
6. Jan Edler, Jim Lipkis, and Edith Schonberg, ''Process Management for Highly Parallel UNIX Systems,'' *Proc. USENIX Workshop on UNIX and Supercomputers*, (Sept. 1988).
7. Bob Beck and Dave Olien, ''A Parallel Process Model,'' *Proc. USENIX*, pp. 83-101 (September, 1987).
8. Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser, ''CHORUS Distributed Operating Systems,'' *Computing Systems Journal* **1**(4) pp. 305-370 The Usenix Association, (December 1988).
9. John K. Ousterhout, ''Scheduling Techniques for Concurrent Systems,'' *Proc. 3rd International Conf. on Distributed Systems*, pp. 22-30 (1982).
10. E.W. Dijkstra, ''Cooperating Sequential Processes,'' in *Programming Languages*, ed. F. Genuys,Academic Press, New York (1968).
11. C.A.R. Hoare, ''Monitors: An Operating System Structuring Concept,'' *Communications of the ACM* **17**(10)(October, 1974).
12. Andrew D. Birrell, ''An Introduction to Programming with Threads,'' Technical Report 35, DEC – SRC, Palo Alto, CA (January 89).