Using Threads in Interactive Systems: A Case Study

Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch and Mark Weiser

Xerox PARC
3333 Coyote Hill Road
Palo Alto, California 94304

Correspondence should be addressed to:

Mark Weiser
Xerox PARC
3333 Coyote Hill Road
Palo Alto, California 94304

weiser@xerox.com

Using Threads in Interactive Systems: A Case Study

Abstract. We describe the results of examining two large research and commercial systems for the ways that they use threads. We used two methods: reading the code and doing microsecond analysis of interthread events. We identify ten different paradigms of thread usage: *defer work, general pumps, slack processes, sleepers, one-shots, deadlock avoidance, rejuvenation, serializers, encapsulated fork* and *exploiting parallelism*. While some, like *defer work*, are well known, others have not been previously described. Most of the paradigms cause few problems for programmers and help keep the resulting system implementation understandable. The *slack process* paradigm is both particularly effective in improving system performance and particularly difficult to make work well. We observe that thread priorities are difficult to use and may interfere in unanticipated ways with other thread primitives and paradigms. Finally, we glean from the practices in this code several possible future research topics in the area of thread abstractions.

1. Introduction

Threads sharing an address space are becoming more widely available in popular operating systems such as Solaris 2.x, OS2/2.x, Windows NT and SysVR4 [Powell91][Custer93]. Xerox PARC's *Cedar* programming environment and Xerox's *STAR*, *ViewPoint*, *GlobalView for X Windows* and *DocuPrint* products have used lightweight threads for over 10 years [Smith82][Swinehart86]. This paper reports on an inspection and analysis of the code developed at Xerox in an attempt to detect common paradigms and common mistakes of programming with threads. Our analysis is both static, from reading many lines of ancient and modern modules, and dynamic, using a number of tools we built for detailed thread inspection. We distinguish two systems that developed independently although most of our remarks apply to both. One, originally a research system but now underlying a number of products as well, we call Cedar. The other, a product system originally derived from the same base as Cedar but relatively unconnected to it for over ten years, we call GVX.

We believe that the systems we examined are the largest and longest-used thread-based interactive systems in everyday use in the world. They are both in use and under continual development. They contain approximately 2.5 million lines of code, in 10,000 modules written by hundreds of people, declaring more than 1000 monitors and monitored record types and over 300 condition variables. They mostly run as very large (working sets of 10's of megabytes) shared-memory, multi-application systems. They have been ported to a variety of processors and multiprocessors, including the .25 to 5 MIPS Xerox D-machines in the 1980's and 5 to 100 MIPS processors of the early 1990's. Our examination of them reveals the kinds of thread facilities and practices that the programmers of these systems found useful and also the kinds of thread programming mistakes that even this experienced community still makes.

Our analysis is subject to two unavoidable biases. First, the programmers of these systems may not be representative -- for instance, one community consisted primarily of PhD researchers. Second, any group of people develop habits of work--idioms--that may be unrelated to best practice, but are simply "how we do it here". Although our report draws on code written by two relatively independent communities, comparing and contrasting the two is beyond the scope of this paper.

We examined the systems running on the Portable Common Runtime on top of Unix [Weiser89]. PCR provides the usual primitives: threads sharing a single address space, monitor locks, condition variables, fork and join. Section 2 describes the primitives in more detail.

Although these systems do run on multiprocessors, this paper emphasizes the role of threads in program structuring rather than how they are used to exploit multiprocessors [Owicki89].

This paper is not, for the most part, a statistical analysis of thread behavior. Instead of using the aggregate view provided by statistics we choose the microscopic view of individual paradigms and critical path performance. For example, the time between when a key is pressed and the corresponding glyph is echoed to a window is very important to the usability of these systems. Developing an understanding of how threads are interacting in accomplishing this task helps to understand the observed performance.

Information for this paper came from two sources. While the system is running we gather microsecond-resolution information about events in the threads and the Unix kernel. Section 3 provides summaries of the data and a micro-behavior example. Section 7 describes the tools we used to gather and present the data. To develop the database of static uses of threads we used *grep* to locate all uses of thread primitives and then read the surrounding code. This reading led us to further searching for uses of modules that provide specialized access to threads (for example, the Cedar package PeriodicalFork). This reading and understanding eventually led to the classifications of thread paradigms described in Section 4.

Sections 5 and 6 present some engineering lessons -- both for implementors using threads and for implementors of thread systems -- from this study. We conclude with some suggestions for future work and a request for more detailed descriptions of large systems.

2. Thread model

Lampson and Redell describe the Mesa language's thread model and provide rationale for many of the design choices [Lampson80]. Here, we summarize the salient features as used in our systems.

The Mesa thread model supports multiple, light-weight, pre-emptively scheduled threads that share an address space. The FORK operation creates a new thread to carry out the FORK's procedure-invocation argument. FORK returns a thread value. The JOIN operation on a thread value returns the value returned by the corresponding FORK's procedure invocation. A thread may be JOINed at most once. If a thread will not be JOINed it should be DETACHed, which tells the thread implementation that it can recover the resources of the thread when it terminates.

The language provides *monitors* and *condition variables* for synchronizing thread activities. A monitor is a set of procedures, or *module*, that share a mutual exclusion lock, or *mutex*. The mutex protects any data managed by the module by ensuring that only a single thread is executing within the module at any instant. Other threads wanting to enter the monitor are enqueued on the mutex. The Mesa compiler automatically inserts locking code into monitored procedures. A variant on this scheme, associating locks with data structures instead of with modules, is occasionally used in order to obtain finer grain locking. Condition variables (CVs) give more explicit control of thread scheduling. Each CV represents a state of the module's data structures (a *condition*) and a queue of threads waiting for that condition to become true. A thread uses the WAIT operation on a CV if it has to wait until the condition holds. WAIT operations may time out depending on the timeout interval associated with the CV. A thread uses NOTIFY or BROADCAST to signal waiting threads that the condition has been achieved. The compiler enforces the rule that CV operations are only invoked with the monitor lock held. The WAIT operation atomically releases the monitor lock and adds its calling thread to the CV's wait queue. NOTIFY causes a single thread that is waiting on the CV's wait queue to become runnable--*exactly one waiter wakens* behavior. (Note that some thread packages define their analog of NOTIFY to have *at least one waiter wakens* behavior [Birrell91].) BROADCAST causes all threads that are waiting on the CV to become runnable. In either case, threads must compete for the monitor’s mutex before reentering the monitor.

Unlike the monitors originally described by Hoare [Hoare74], the Mesa thread model does not guarantee that the condition associated with a CV is satisfied when a WAIT completes. If BROADCAST is used, for example, a different thread might acquire the monitor lock first and change the state of the program. Therefore a thread is responsible for rechecking the condition after each WAIT. Thus, the prototypical use of WAIT is inside a WHILE loop that checks the condition, *not* inside an IF statement that would only check the condition once. Programs that obey the "WAIT only in a loop" convention are insensitve to whether NOTIFY has *at least one waiter wakens* behavior or *exactly one waiter wakens* behavior as described above. Indeed, under this convention BROADCAST can be substituted for NOTIFY without affecting program correctness, so NOTIFY is just a performance hint.

Threads have *priorities* that affect the scheduler. The scheduler runs the highest priority runnable thread and if there are several runnable threads at the highest priority then round-robin is used among them. If a system event causes a higher priority thread to become runnable, the scheduler will *preempt* the currently running thread, even if it holds monitor locks. There are 7 priorities in all, with the default being the middle priority (4). Typically lower priority is used for long running, background work, while higher priority is used for threads associated with devices or aspects of the user interface, keeping the system responsive for interactive work. A thread’s initial priority is set when it is created. It can change its own priority.

The timeslice interval and the CV timeout granularity in the current implementation are each 50 milliseconds. The scheduler runs at least that often, but also runs each time a thread blocks on a mutex, waits on a CV, or calls the YIELD primitive. The only purpose of the YIELD primitive is to cause the scheduler to run (but see discussion later of YieldButNotToMe). The scheduler takes less than 50 microseconds to switch between threads on a Sparcstation-2.

3. Dynamic thread behavior

Section 3.1 describes the large-scale behavior of our thread systems: how many threads are there, how often are threads created and destroyed, how often are monitor locks and condition variables used. It is intended to give the reader an intuitive feel for the physical and temporal scale of the system.

Section 3.2 provides a microscopic look at a thread misbehavior. It is intended to introduce the notion of microscopic examination and to provide an example for discussion later in the paper.

3.1 Macro-behavior

One of the original motivations for our work was a desire to understand the dynamic behavior of user-level threads. For this purpose, we constructed an instrumented version of PCR that measured the number of threads in the system, thread lifetimes, the run length distribution of threads and the rate at which monitor locks and condition variables are used. Analysis of the data obtained from this instrumented system led to the realization that there were a number of consistent patterns of thread usage, which led to the static analysis on which this paper is focused. To give the reader some background and context for this static analysis, we present a summary of our dynamic data below. This data is based on a set of benchmarks intended to be typical of user activity, including compilation, formatting a document into a page description language (like Postscript), previewing pages described by a page description language and user interface tasks (keyboarding, mousing and scrolling windows). All data was taken on a Sparcstation-2 running SunOS-4.1.3.

Looking at the dynamic thread behavior, we observed several different classes of threads. There were *eternal* threads that repeatedly waited on a condition variable and then ran briefly before waiting again. There were *worker* threads that were forked to perform some activity, such as formatting a document. Finally, there were short-lived *transient* threads that were forked by some long-lived thread, would run for a relatively short while and then exit.

A Cedar or GVX world uses a moderate number of threads. Consider Cedar first: an idle Cedar system has about 35 eternal threads running in it and forks a transient thread once a second on average. Keyboard activity can cause up to 5 thread forks per second, although most other user-interface activity causes much smaller increases in thread forking rates. While one of our benchmark applications (document formatting) employed large numbers of transient threads (forking 3.6 threads/sec.), the other two compute-intensive applications we examined caused thread-forking activity to *decrease* by more than a factor of 3. In all our benchmarks, the maximum number of threads concurrently existing in the system never exceeded 41, although users employ two to three times this many in everyday work. Transient threads are by far the most numerous resulting in an average lifetime for non-eternal threads that is well under 1 second.

An idle GVX world exhibits noticeably different behavior than just described. An idle system contains 22 eternal threads and forks no additional threads. In fact, no additional threads are forked for any user interface activity, be it keyboard, mouse, or windowing activity.

The Appendix contains a brief description of each eternal thread seen in the Cedar and GVX benchmarks.

Table 1: Forking and thread-switching rates

Cedar	Forks/sec	Thread Switches/sec
Idle Cedar	0.9	132
Keyboard input	5.0	269
Mouse movement	1.0	191
Window scrolling	0.7	172
Document formatting	3.6	171
Document previewing	1.6	222
Make program	0.3	170
Compile	0.3	135
GVX		
Idle GVX	0	33
Keyboard input	0	60
Mouse movement	0	34
Window scrolling	0	43

The rate at which a Cedar system switches among running threads varies from 130/sec. for an idle system to around 270/sec for a system experiencing heavy keyboard/mouse input activity. Thread execution intervals (the lengths of time between thread switches) exhibit a peak at about 3 milliseconds, with about 75% of all execution intervals being between 0 and 5 milliseconds in length. This is due to the very short execution intervals of most eternal and transient threads. A second peak is around 45 milliseconds, which is related to the PCR time-slice period, which is 50 milliseconds. Transient and eternal thread activity steals the first part of a timeslice with the remainder going to worker threads.

While most execution intervals are short, longer execution intervals account for most of the total execution time in our systems. Between 20% and 50% of the total execution time during any period is accumulated by threads running for periods of 45 to 50 milliseconds. We also examined the total execution time contribution as a function of thread priority. Only two patterns were evident: of the 7 available priority levels one wasn't used at all, and user interface activity tended to use higher priorities for its threads than did user-initiated tasks such as compiling.

GVX switches among threads at a decidely lower rate: an idle system switches only 33 times per second, while heavy keyboard activity will drive the rate up to 60/sec. The same bi-modal distribution of execution intervals is exhibited as in Cedar: between 50% and 70% of all execution intervals are between 0 and 5 milliseconds in length with a second peak around 45 milliseconds. Between 30% and 80% of the total execution time during any period is accumulated by threads running for periods of 45 to 50 milliseconds.

GVX's use of thread priorities was noticeably different than Cedar's. While Cedar's core of long-lived threads are relatively evenly distributed over the four "standard" priority values of 1 to 4, GVX sets almost all of its threads to priority level 3; using the lower two priority levels only for a few background helper tasks. Two of the five low-priority threads in fact never actually ran during our experiments. As with Cedar, one of the 7 priority levels is never used. However, while Cedar uses level 7 for interrupt handling and doesn't use level 5, GVX does the opposite. In both systems, priority level 6 gets used by the system daemon that does proportional scheduling. Cedar also uses level 6 for its garbage collection daemon.

One interesting behavior that our Cedar thread data exhibited was a variety of different forking patterns. An idle Cedar system forks a transient thread about once every 2 seconds. Each forked thread, in turn, forks another transient thread. Keyboard activity causes a transient thread to be forked by the command-shell thread for every keystroke. On the other hand, simply moving the mouse around causes no threads to be forked. Even clicking a mouse button (e.g. to scroll a window) causes no additional forking activity. (However, both keyboard activity and mouse motion cause significant increases in activity by eternal threads.) Scrolling a text window 10 times causes 3 transient threads to be forked, one of which is the child of one of the other transients.

Document formatting causes a great number of transient threads to be forked by the main formatting worker thread, whereas compilation and document previewing cause a moderate number of transient forks. While the compiler's and previewer's transient threads simply run to completion, each of the document formatter's transient threads fork one or more additional transient threads themselves. However, third generation forked threads do not occur. In fact, none of our benchmarks exhibited forking generations greater than 2. That is, every transient thread was either the child or grandchild of some worker or long-lived thread.

Checking whether a program needs recompiling (the Make program) does not cause any threads to be forked (the command-shell thread gets used as the main worker thread), except for garbage collection and finalization of collected data objects. Each of these activities causes a moderate number of first-generation transient threads to be forked.

Table 2: Wait-CV and monitor entry rates

Cedar	Wait-CVs/sec	% of CV that timeout	ML-enters/sec
Idle Cedar	121	82%	414
Keyboard input	185	48%	2557
Mouse movement	163	58%	1025
Window scrolling	115	69%	2032
Document formatting	130	72%	2739
Document previewing	157	56%	1335
Make program	158	61%	2218
Compile	119	82%	1365
GVX			
Idle GVX	32	99%	366
Keyboard input	38	42%	1436
Mouse movement	33	96%	410
Window scrolling	25	61%	691

The rate at which locking and condition variable primitives are used is another measure of thread activity. Table 2 shows the rates for each benchmark. The rate of waiting on CVs in Cedar ranged from 115/second to 185/second, with 50% to 80% of these waits timing out rather than receiving a wakeup notification. Monitors are entered much more frequently, reflecting their use to protect data structures (especially in reusable library packages). Entry rates varied from 400/second for an idle system to 2500/sec for a system experiencing heavy keyboard activity to 2700/second for document formatting. Contention was low, however, occuring on 0.01% to 0.1% of all entries to monitors.

For GVX, the rate of waiting on CVs ranged from 32/second to 38/second, with 42% to 99% of these waits timing out rather than receiving a wakeup notification. Monitors are entered at rates between 366/sec and 1436/sec. Interestingly, contention for monitor locks was sometimes significantly higher in GVX than in Cedar, occuring 0.4% of the time when scrolling a window and 0.2% of the time when heavy keyboard traffic was present.

Table 3: Number of different CVs and monitor locks used

Cedar	# CVs	# MLs
Idle Cedar	22	554
Keyboard input	32	918
Mouse movement	26	734
Window scrolling	30	797
Document formatting	46	1060
Document previewing	32	938
Make program	24	1296
Compile	36	2900
GVX		
Idle GVX	5	48
Keyboard input	7	204
Mouse movement	5	52
Window scrolling	6	209

Typically, most of the monitor/condition variable traffic is observed in about 10 to 15 different threads, with the worker thread of a benchmark activity dominating the numbers. The other active threads exhibit approximately equal traffic. The number of *different* monitors entered during the benchmarks varies from 500 to 3000 as shown in Table 3. In contrast, only about 20 to

50 different condition variables are waited for in the course of the benchmarks. GVX uses fewer monitors and CVs.

3.2 Micro-behavior

The dynamic information described in the previous section is not fully satisfying, because it fails to show details of individual threads, and is not sufficient to understand the behaviors we experienced in our large systems. For instance, our systems had subtle problems: sometimes less performance than we expected, sometimes large amounts of idle time in what should have been a compute-bound process, sometimes rare lockups of the system. For years none of these were frequent or annoying enough to warrant urgent attention. But eventually we felt we needed to understand them. The problems were not easily amenable to discovery via conventional debugging or profiling methods (which we tried) or dynamic statistics. We needed new tools that could show us detailed scheduling and process interaction.

When we built these tools, we found a fascinating world of microscopic thread behavior had opened up to us. We think that micro-visualization tools for understanding operating systems are underutilized and could be a source of considerable insight. In this section we illustrate some of this world through one of the examples that led us to look for the thread paradigms described later in this paper.

Good X window system performance when painting large regions with many requests requires batching and merging overlapping requests. Good interactive performance, such as keystroke echoing, requires that paint requests be sent to the server with very little delay. In the fall of 1992 Cedar was painting large regions too slowly and we did not know why. We saw more frequent X communication than we expected, even though we had a batching mechanism in place. What was going wrong? The answer is easy to see and explain with the proper tools.

3.2.1 Example of micro-behavior visualization

Figure 1 shows 100 milliseconds of micro-behavior. It focuses on two Unix processes, one of which is running two independent threads inside itself. The first Unix process is the X server. The other one is a Cedar *Virtual Processor* (VP), which runs the Portable Common Runtime (PCR), and so can be assigned to run any number of threads. For our example, we focus on two of those threads: a worker thread producing an image and an I/O thread that is buffering the image commands to the X server. (The I/O thread is an example of what we will call in the next section a "slack process").

In Figure 1 the top two horizontal lines represent the Unix processes, the two at the bottom represent the threads running within the second Unix process. Many other Unix processes and Cedar threads running at the same time are not shown.

Let’s look first at the two Unix processes. Where they show a slim horizontal grey line, they are idle, unscheduled by the Unix kernel. Where they show a wider red horizontal line, they have actually acquired a CPU. When they have a CPU, they may cause kernel activity, shown as short vertical strokes. For this figure we were interested in possible overhead caused by system calls and page faults, so only those kernel events are shown, page faults in red, system calls in blue. Figure 1 shows that the X window server acquired and lost the CPU eleven times in the 100 milliseconds; the VP ten times.

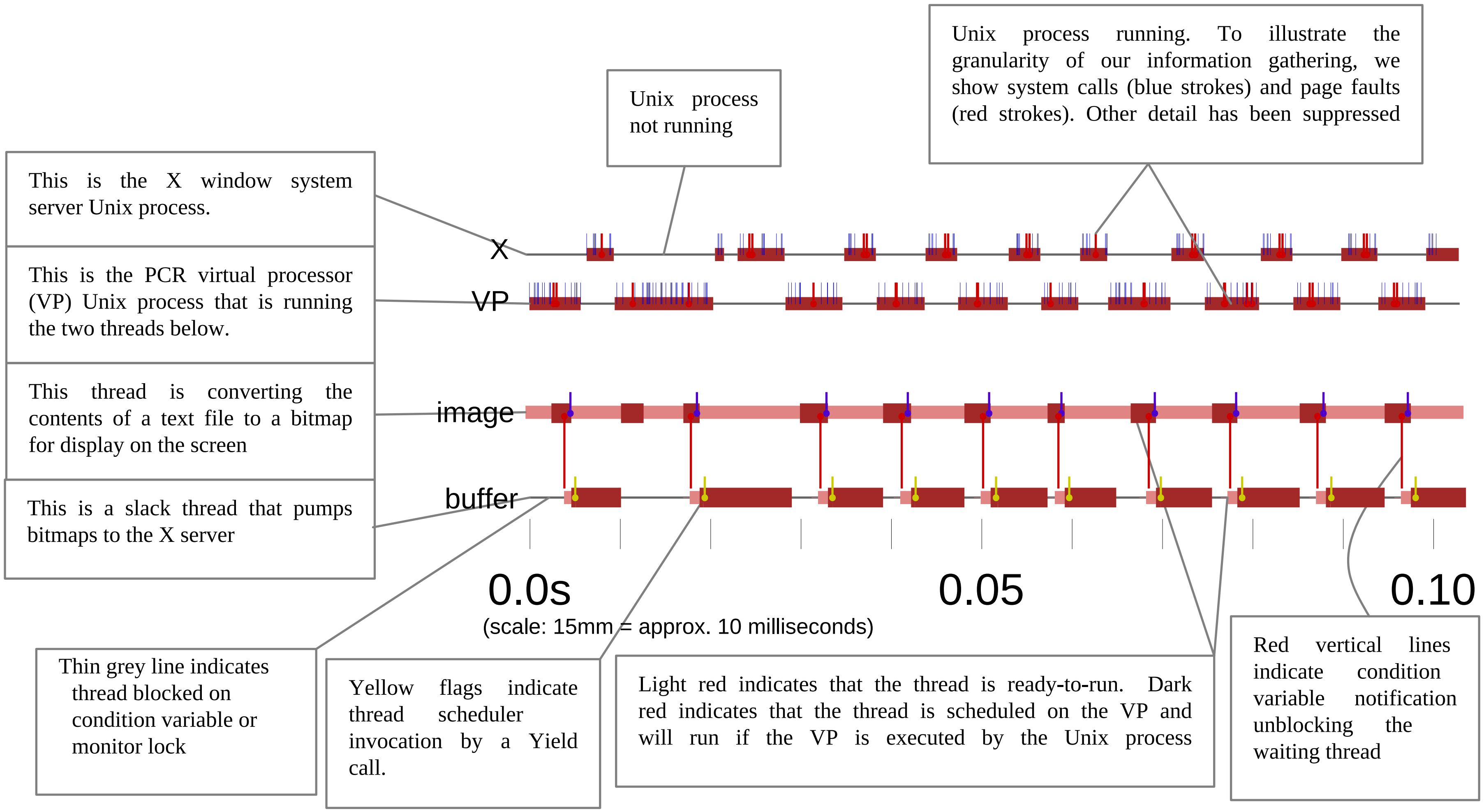


Figure 1
example of micro-visualization

Note: Our data collector reports "unknown state" for processes at the beginning of a capture interval. For clarity we added by hand the state color in two places: the first red "running" region in VP, and the pink "ready-to-run" region in image.

Now look at the two threads, the third and fourth lines in figure 1. Again a slim grey line shows they are idle. A broader pink line indicates the thread is ready to run. As with the Unix processes, a broad red area shows that the thread has actually acquired a virtual processor. Within the threads, again vertical strokes indicate "kernel" activity, in this case requests to the underlying thread mechanism running in the VP. Long vertical strokes from thread-to-thread indicate inter-thread communication, in this case notification via a condition variable that a thread should wake up. The sender of the communication has a dot at the beginning of the stroke. Short yellow strokes indicate a YIELD call, in which a thread offers to voluntarily give up the processor to anyone who may want it. Although PCR has a pre-emptive scheduler so yields are not strictly necessary, threads sometimes yield for performance reasons.

Finally, notice the relationship between the red areas in the VP, meaning it has acquired a real processor, and the red areas in the threads, meaning they have acquired the VP. A thread only can do real work when two conditions are met: it has acquired a VP, indicated by a red area in the thread, and the VP has itself acquired the real processor, indicated by red in the VP. Thus threads sometimes become red for quite a while, but are making no progress because their VP does not have the processor. This is particularly apparent in the second thread (fourth line) in figure 1.

3.2.2 A problem revealed by micro-behavior visualization

Figure 1a shows a 10 millisecond slice from Figure 1. Starting from the top left, the X server process is idle, the VP process is about to acquire the processor (indicated by turning red), the image thread is ready-to-run but not scheduled on the VP and the buffer thread is idle.

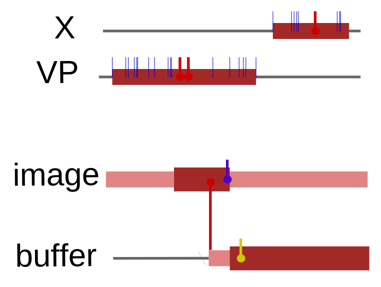


figure 1a

When the VP starts running, it eventually runs the image thread. After some work, the image thread tells the buffer thread that there is some image data to transmit by notifying a condition variable (CV). This notification (shown by the long red stroke from the image to buffer thread) immediately makes the buffer thread ready-to-run (shown by the pink color). The buffer thread does not run immediately upon being notified because the image thread is still holding the monitor lock. (See Section 6.1) When the image thread leaves the monitor (indicated by the blue stroke at the end of the dark red region), the buffer thread runs.

The buffer thread runs only briefly and then YIELDs, shown by the short yellow stroke. The YIELD does nothing, the buffer thread continues to run and begins to communicate to the X server, which we deduce from the fact that the X server now starts to run (top line). The buffer thread continues to run for some time and eventually this cycle repeats.

Why is that YIELD there and what is supposed to be happening? The buffer thread is supposed to be storing up paint requests, merging overlapping requests and sending them only occasionally to the X server. It does the YIELD when it determines that it should not yet communicate to the X server, instead offering to give up the processor to any other thread that desires it. In this case, the scheduler decided to continue running the buffer process so it proceeded to communicate with the X server anyway.

The YIELD by the buffer thread invokes the scheduler to choose the highest priority ready-to-run thread. The image thread is ready-to-run as shown by its pink color but it has lower priority than the buffer thread (deduced from the figure and confirmed in the source code). So in this case the YIELD in the buffer thread accomplishes nothing.

As can be seen in microcosm in Figure 1a, and repeatedly in Figure 1, this failure to yield means that every time the image thread wakes up and signals the buffer thread, the buffer thread runs and wakes up the X server. The failure to merge overlapping requests and the frequent thread and process switches are costly in performance.

We will return to this problem in Section 5.2.

4. Thread paradigms

Birrell provides a good introduction to some of the basic paradigms for thread use [Birrell91]. Here we go further into more advanced paradigms, their necessity and frequency in practice and the problems of performance and correctness entailed by these advanced usages.

Birrell suggests that forking a new thread is useful in several situations: to exploit concurrency on a multiprocessor (including waiting for I/O, where one processor is the I/O device); to satisfy a human user by making progress on several tasks at once; to provide network service to multiple clients simultaneously; and to defer work to a less busy time [Birrell91, p. 109].

We examined about 650 different code fragments that create threads. We gradually developed a collection of categories that we could use to explain how thread uses were similar to one another and how they differed. Our final list of categories is:

- defer work (same as Birrell's)
- pumps (components of Birrell's pipelines. He describes them for exploiting multiprocessing, but we saw them mostly used for structuring.)
- slack processes, a kind of specialized pump (new)
- sleepers and one-shots (common uses probably omitted by Birrell because synchronization problems in them are rare)
- deadlock avoiders (new)
- task rejuvenation (new)
- serializers, another kind of specialized pump (new)
- concurrency exploiters (same as Birrell's)
- encapsulated forks, which are forks in packages that capture certain paradigms and whose uses are themselves counted in the other categories.

These static categories complement the thread lifetime characterization in Section 3.1. The eternal threads tend to be sleepers, pumps and serializers with nothing to do. Worker threads are often work deferrers and transient threads are often deadlock avoiders. But the reader is cautioned that the static paradigm can't be predicted from the dynamic lifetime.

4.1 Defer work

Deferring work is the single most common use of forking in these systems. A procedure can often reduce the latency seen by its clients by forking a thread to do work not required for the procedure's return value. Sometimes work can be deferred to times when the system is under less load [Birrell91]. Cedar practice has been to introduce work deferrers freely as the opportunity to use them is noticed. Many commands fork an activity whose results will be reported in a separate window: control in the originating thread returns immediately to the user, an example of latency reduction for the human client. Some examples of work deferrers are:

- forking to print a document
- forking to send a mail message
- forking to create a new window
- forking to update the contents of a window

Some threads are themselves so critical to system responsiveness that they fork to defer almost any work at all beyond noticing what work needs to be done. These critical threads play the role of interrupt handlers. Forking the real work allows it to be done in a lower priority thread and frees the critical thread to respond to the next event. The keyboard-and-mouse watching process, called the Notifier, is such a critical, high priority thread in both Cedar and GVX.

4.2 Pumps

Pumps are components of pipelines. They pick up input from one place, possibly transform it in some way and produce it as output someplace else.[1] Bounded buffers and external devices are two common sources and sinks. The former occur in several implementations in our systems for connecting threads together, while the latter are accessed with system calls (read, write) and shared memory (raw screen IO and memory shared with an external X server).

Though Birrell suggests creating pipelines to exploit parallelism on a multiprocessor, we find them most commonly used in our systems as a programming convenience, another reflection of the uniprocessor heritage of the systems. This is also their primary use in the well-known Unix shell pipelines. For example, in our systems all user input is filtered through a pipeline thread that preprocesses events and puts them into another queue, rather than have each reader thread preprocess on demand. Neither approach necessarily provides a more efficient system, but the pipeline is conceptually simpler: tokens just appear in a queue. The programmer needs to understand less about the pieces being connected.

One interesting kind of pump is the slack process. A slack process explicitly adds latency to a pipeline in the hope of reducing the total amount of work done, either by merging input or replacing earlier data with later data before placing it on its output. Slack processes are useful when the downstream consumer of the data incurs high per-transaction costs. The buffer thread discussed in Sections 3.2 and 5.2 is an example of a slack process.

4.3 Sleepers and oneshots

Sleepers are processes that repeatedly wait for a triggering event and then execute. Often the triggering event is a timeout. Examples include: call this procedure in K seconds; blink the cursor in M milliseconds; check for network connection timeout every T seconds. Other common events are external input and service callbacks from other activities. For instance, our systems

[1]We use the term *pump*, rather than the more common *filter* because data transformation, ala filters, is just one of the things that pumps can do. In general, pumps control both the data transformation and the timing of the transfer. We also like the connotation of an active entity conveyed by *pump* as opposed to the passivity of *filter*.

use callbacks from the garbage collector to finalize objects and callbacks from the filesystem when files change state. These callbacks are removed from time-critical paths in the garbage collector and filesystem by putting an event in a work queue serviced by a sleeper thread. The client's code is then called from the sleeper.

Sleepers frequently do very little work before sleeping again. For instance, various cache managers in our systems simply throw away aged values in a cache then go back to sleep.

Another kind of sleeper is the garbage collector's background thread which cleans pages dirtied by other threads. If it gets too far behind in its work it could cause virtual memory thrashing by cleaning pages no longer resident in physical memory. Its rate of awakening must depend on the amount of page dirtying which depends on the workload of all the other threads in the system.

OneShots are sleeper processes that sleep for a while, run and then go away. This paradigm is used repeatedly in Cedar, for example, to implement guarded buttons of several kinds. (A guarded button must be pressed twice, in close, but not too close succession. They usually look like "~~Button~~" on the screen.) After a one-shot is forked it sleeps for an arming period that must pass before a second click is acceptable. Then it changes the button appearance from "~~Button~~" to "Button" and sleeps a second time. During this period a second click invokes a procedure associated with the button, but if the timeout expires without a second click, the one-shot just repaints the guarded button.

4.4 Deadlock avoiders

Cedar often uses FORK to avoid violating lock order constraints. The window manager makes heavy use of this paradigm. For example, after adjusting the boundary between two windows the contents of the windows must be repainted. The boundary-moving thread forks new threads to do the repainting because it already holds some, but not all of the locks needed for the repainting. Acquiring these locks would require unwinding the adjusting process far enough to release locks that would violate locking order, then reacquiring all the necessary locks in the right order. It is far simpler to fork the painting processes, unwind the adjuster completely and let the painters acquire the locks that they need in separate processes.

Another case of deadlock avoidance is forking the callbacks from a service module to a client module. Forking permits the service thread to proceed, eventually releasing locks it holds that will be needed by the client. The fork also insulates the service from things that may go wrong in the client callback. For instance, Cedar permits clients to register callback procedures with the garbage collector that are called to finalize (clean up) data structures. The finalization service thread forks each callback.

4.5 Task rejuvenation

Sometimes threads get into bad states, such as arise from uncaught exceptions or stack overflow, from which recovery is impossible within the thread itself. In many cases, however, cleanup and recovery is possible if a new "task rejuvenation" thread is forked. For uncaught errors, an exception handler may simply fork a new copy of the service. For stack overflow, a new thread is forked to report the stack overflow. Using threads for task rejuvenation can be tricky and is a bit counter-intuitive (This thread is in trouble. Ok let's make *two* of them!) However, it is a paradigm that adds significantly to the robustness of our systems and its use is growing. A recent addition is a task-rejuvenating FORK that was added to the input event dispatcher in Cedar. The dispatcher makes unforked callbacks to client procedures because (a) this code is on the critical path for user-visible performance and (b) most callbacks are very short (e.g. enqueue an event) and so a fork overhead would be significant. But not forking makes the dispatcher vulnerable to uncaught runtime errors that occur in the callbacks. Using task rejuvenation, the new copy of the dispatcher keeps running.

Task rejuvenation is a controversial paradigm. It's ability to mask underlying design problems suggests that it be used with caution.

4.6 Serializers

A serializer is a queue and a thread that processes the work on the queue. The queue acts as a point of serialization in the system. The primary example is in the window system where input events can arrive from a number of different sources. They are handled by a single thread in order to preserve their ordering. This same paradigm is present in most other window systems and in many cases it is the only paradigm. In the Macintosh, Microsoft Windows, and X programming models, for example, each application runs in a serializer thread that pulls events from a queue associated with the application's window.

4.7 Concurrency exploiters

Concurrency exploiters are threads created specifically to make use of multiple processors. They tend to be very problem-specific in their details. Since our systems have only relatively recently begun to run on multiprocessors we were not surprised to find very few concurrency exploiters in them.

4.8 Encapsulated forks

One way that our systems promote use of common thread paradigms is by providing modules that encapsulate the paradigms. This section describes three such packages used frequently in our systems.

DelayedFork and PeriodicalFork

DelayedFork expresses the paradigm of a one-shot. It calls a procedure at some time in the future. Although one-shots are common in our system, DelayedFork is only used in our window systems. (Its limited use might be because it appeared only recently.)

PeriodicalFork is simply a DelayedFork that repeats over and over again at fixed intervals. It encapsulates the sleeper paradigm where the wakeups are prompted solely by the passage of time.

MBQueue

The module MBQueue (the name means Menu/Button Queue) encapsulates the serializer paradigm in our systems. MBQueue creates a queue as a serialization context and a thread to process it. Mouse clicks and key strokes cause procedures to be enqueued for the context: the thread then calls the procedures in the order received.

We consider a queue together with a thread processing it an important building block of user interfaces. This is borne out by the fact that our system actually contains several minor variations of MBQueue. Why is there not a more general package? It seems that each instance adds serialization to a specific interface already familiar to the programmer. Furthermore, the serialization is often on a critical interactive performance path. Keeping a familiar interface to the programmer and reducing latency cause new variations to be preferred over a single generic implementation.

Miscellaneous

Many modules that do callbacks offer a *fork* boolean parameter in their interface, indicating whether or not the called-back procedure is to be called directly or in a forked thread. The default is almost always TRUE, meaning the callback will be forked. Unforked callbacks are usually intended for experts, because they make future execution of the calling thread within the module dependent on successful completion of the client callback.

4.9 Paradigm summary

Table 4 summarizes the absolute and relative frequencies of the paradigms in our systems. Note that in keeping with our emphasis on using threads as program structuring devices this is a static count. (Threads may be counted in more than one category because they change their behavior.) Some threads seem not to fit easily into any category. These are captured in the "Unknown or other" entries.

Table 4. Static Counts, Cedar and GVX

	Cedar		**GVX**	
Defer work	108	31%	77	33%
Pumps				
General pumps	48	14%	33	14%
Slack processes	7	2%	2	1%
Sleepers	67	19%	15	6%
Oneshots	25	7%	11	5%
Deadlock avoidance	35	10%	6	3%
Task rejuvenation	11	3%	0	0%
Serializers	5	1%	7	3%
Encapsulated fork	14	4%	5	2%
Concurrency exploiters	3	1%	0	0%
Unknown or other[2]	25	7%	78	33%
TOTAL	348	100%	234	100%

Our categories seem to apply well to other systems. For instance, Lampson and Redell describe the applications Pilot, Violet and Gateway (which although Mesa-based share no code with the systems we examined), but do not identify general types of threads [Lampson80]. Yet from the published description one can deduce the following:

Pilot: almost all sleepers.

Violet: sleepers, one-shots and work deferral.

Gateway: sleepers and pumps.

5. Issues in thread use

Given a modern platform providing threads, system builders have the option of using thread

[2]The large number of unknown threads in GVX is due to our unfamiliarity with this code, rather reflecting any significant difference in paradigm use.

primitives to accomplish tasks that they would have used other techniques to accomplish on other platforms. The designer must balance the modest cost of creating a thread against the benefits in structural simplification and concurrency that would accrue from its introduction. In addition to the cost of creating it, a thread incurs a modest ongoing cost for the virtual memory occupied by its stack. Our PCR thread implementation allocates virtual memory for the maximum possible stack size of each thread. If there is very little state associated with a thread this may be a very inefficient use of memory [Draves91].

5.1 Easy thread uses

Our programmers have become very adept at using the sleeper, oneshot, pump (in paths without critical timing constraints) and work deferrer paradigms. For the most part these require little interaction between threads beyond taking care to provide mutual exclusion (monitors) for shared data. Pump threads interact with threads on either side of them in pipelines, but the interactions generally follow well-known producer-consumer patterns. Using FORK to create sleeper threads has fallen into disfavor with the advent of PCR thread implementation: 100 kilobytes for each of hundreds of sleepers' stacks is just too expensive. The PeriodicalProcess for timeout driven sleepers and other sleeper encapsulations often can accomplish the same thing using closures to maintain the little bit of state necessary between activations.

Deadlock avoiders also are usually very simple, but the overall locking schemes in which they are involved are often very, very complicated (and far beyond the scope of this paper).

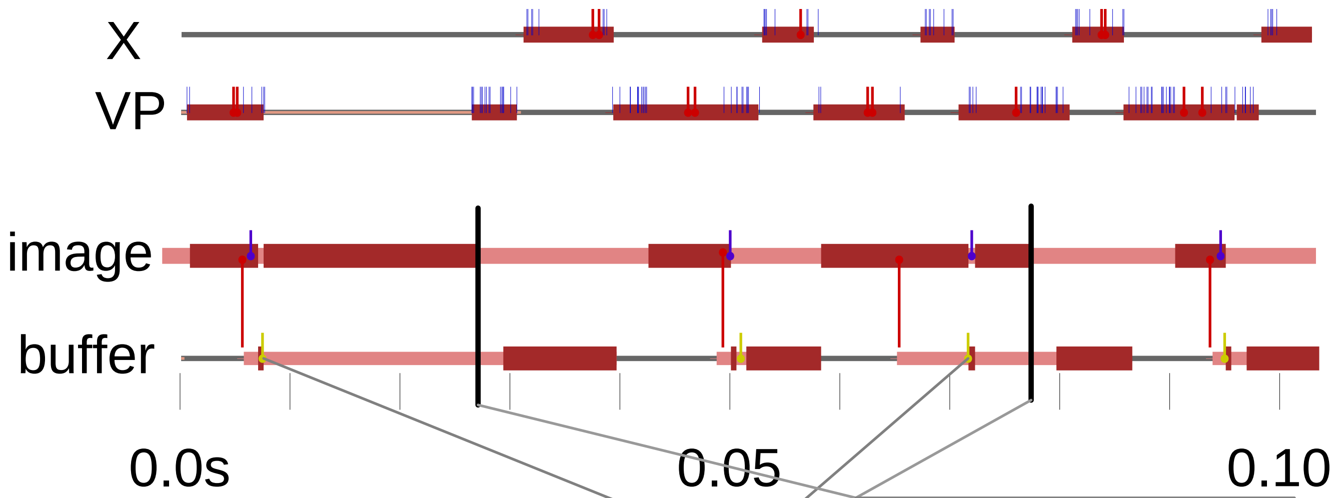
5.2 Hard thread uses

Some of the thread paradigms seem much more difficult. First, there is little guidance in the literature and in our experience for using concurrency exploiters in interactive systems. The arrival of relatively low-cost workstations and operating systems supporting multiprocessing offers new incentives to understand this paradigm better.

Second, designing slack processes and other pumps in paths with timing constraints continues to challenge both the slack process implementors and the PCR implementors. Bad performance in the pipelines that provide feedback for typing (character echoing) and mouse motion is immediately apparent to the user, yet improving the performance is often very difficult.

The buffer thread of Figure 1 and Section 3.2 is an example of a poorly performing slack process--though it's supposed to introduce slack in sending requests to X server, it is not doing so because when it tries to yield the processor, as the highest priority thread it gets it right back.

Fixing the problem by lowering the priority of the buffer thread is clearly wrong, since that could cause starvation of screen painting. We believe that the architecture of having a producer thread notify a consumer (buffer) thread periodically of work to do is a good one, so we did not consider changing the basic paradigm by which these two threads interact. Rewriting the buffer thread to simply sleep waiting for a timeout before sending its events doesn't work either, for reasons discussed in Section 6.3.



(scale: 15mm = approx. 10 milliseconds)

In this diagram YieldButNotToMe succeeds more often in handing the virtual processor (and the real processor) back to the image thread. Consequently there are fewer thread switches, fewer process switches, more merges in the buffer thread, and less total work done in the X server.

These black lines mark the ends of PCR scheduler quanta where YieldButNotToMe terminates and the higher priority buffer thread runs again.

Figure 2
improvement from YieldButNotToMe

Note: Our data collector reports "unknown state" for processes at the beginning of a capture interval. For clarity we added by hand the state color in three places: the first two red "running" regions in **VP**, and the pink "ready-to-run" region in **image**.

We fixed the immediate problem by creating a new yield primitive, called YieldButNotToMe which gives the processor to the highest priority ready thread *other than its caller,* if such a thread exists. Most of the time the image thread is the thread favored with the extra cycles and there is a big improvement in the system's perceived performance. However, the semantic compromise entailed in YieldButNotToMe and the uncertainty that the additional cycles will go to the correct thread suggest that the final solution to these problems still eludes us.

The improved result when the buffer thread uses YieldButNotToMe can be seen in Figure 2. Notice that now when the image thread notifies the buffer thread, the buffer thread usually runs for only a moment before yielding back to the image thread. The result is that the X server runs only five times in the same time interval. The fewer switches to the X server make the buffer thread more effective at doing rectangle combination, there is less time spent in thread switching and the image thread gets much more processor resource over the interval. The user experiences about a three-fold performance improvement.

Finally, even beyond the problems encountered with priorities in managing a pipeline, we found priorities to be problematic. Birrell describes a stable priority inversion in which a high priority thread waits on a lock held by a low priority thread that is prevented from running by a middle-priority cpu hog [Birrell91, pp. 99-100]. Like Birrell, we chose not to incur the implementation overhead of providing priority inheritance from blocked threads to threads holding locks. (Notice that the problem occurs for abstract resources such as the condition associated with a CV as well as for real resources such as locks: the thread primitives have little hope of automatically adjusting thread priority in such situations.) The problem is not hypothetical: we experienced enough real problems with priority inversions that we found it necessary to put the following two workarounds into our systems. First, for one particular kind of lock in the system, PCR does donate cycles from a blocked thread to the thread that is blocking it. This is done only for the per-monitor metalock that locks each monitor's queue of waiting threads. It is not done for monitors themselves, where we don't know how to implement it efficiently. Second, PCR utilizes a high-priority sleeper thread (which we call the SystemDaemon) that regularly wakes up and donates, using a directed yield, a small timeslice to another thread chosen at random. In this way we ensure that all ready threads get some cpu resource, regardless of their priorities.

5.3 Common mistakes

Our dynamic and static inspections of old code revealed occasional correctness and performance problems caused by improper thread usage. Two questionable practices stood out.

First, we saw many instances of WAIT code that did not recheck the predicate associated with the condition variable. Recall that proper use of WAIT when using Mesa monitors is

```
WHILE NOT (OK to proceed) DO WAIT cv END
```

not the

```
IF NOT (OK to proceed) THEN WAIT cv
```

which would be appropriate with Hoare's original monitors.

The IF-based approach will work in Mesa with sufficient constraints on the number and behavior of the threads using the monitor, but its use cannot be recommended. The practice has been a continuing source of bugs as programs are modified and the correctness conditions become untrue.

Second, there were cases where timeouts had been introduced to compensate for missing NOTIFYs (bugs), instead of fixing the underlying problem. The problem with this is that the system can become timeout driven--it apparently works correctly but slowly. Debugging the poor performance is often harder than figuring out why a system has stopped due to a missing NOTIFY. Of course, legitimate timeouts can mask an omitted NOTIFY as well.

5.4 When a fork fails

Sometimes an attempt to fork may fail for lack of resources. As with many other resource allocation failures it's difficult to plan a response. Earlier versions of the systems would raise an error when a FORK failed: the standard programming practice was to catch the error and to try to recover, but good recovery schemes seem never to have been worked out. The resulting code seems overly complex to no good end: the machinery for catching the error is always set up even though once an error is caught nobody really knows what to do about it. Memory allocation failures present similar problems.

Our more recent implementations simply wait in the fork implementation for more resources to become available, but the behaviors seen by the user, such as long delays in response or even complete unresponsiveness, go unexplained.

A number of techniques may be adopted to avoid running out of resources: unfortunately, the better we succeed at that, the less experience we gain with techniques for appropriately recovering from the situation.

5.5 On robustness in a changing environment

Two final comments on the archeology of thread-related bugs. First, we found many instances of timeouts and pauses with ridiculous values. These values presumably were chosen with some particular now-obsolete processor speed or network architecture in mind. For user interface-related timeouts, values based on wall-clock time are appropriate, but timeouts related to processor speeds, or more insidiously, to expected network server response times, are more difficult to specify simply for all time. This may be an area of future research. For instance, dynamically tuning application timeout values based on end-to-end system performance may be a workable solution.

Second, we saw several places where the correctness of threaded code depended on strong memory ordering, an assumption no longer true in some modern multiprocessors with weakly ordered memory [Frailong93][Sites93]. The monitor implementation for weak ordering can use memory barrier instructions to ensure that all monitor-protected data access is consistent, but other uses that would be correct with strong ordering will not work. As a simple example, imagine a thread that once a minute constructs a record of time-date values and stores a pointer to that record into a global variable. Under the assumptions of strong ordering and atomic write of the pointer value, this is safe. Under weak ordering, readers of the global variable can follow a pointer to a record that has not yet had its fields filled in. As another example, Birrell offers a performance hint for calling an intialization routine exactly once [Birrell91, p. 97]. Under weak ordering, a thread can both believe that the initializer has already been called, but not yet be able to see the initialized data.

5.6 Some contrasting experiences: multi-threading and X windows

The usual, single-threaded client interface to an X server is through Xlib, a library that translates an X client's procedural interface into the message-oriented interface of the X server. Xlib does the required translation and manages the I/O connection to the server, both for reading events and writing commands. We studied two approaches to using X windows from a multi-threaded client. One approach uses Xlib, modified only to make it thread-safe [Schmitmann93].

The other approach uses Xl, an X client library designed from scratch with multi-threading in mind [Jacobi92].

A major difference between the two approaches concerns management of the I/O connection to the server. Xl introduced a new serializing thread that was associated with the I/O connection. The job of this thread was solely to read from the I/O connection and dispatch events to waiting threads. In contrast, the ported library allowed any client thread to do the read and a monitor lock on the library provided serialization. There were two problems with this: priority inversion and honoring the clients' timeout parameter on the GetEvent routine. When a client thread blocks on the read call it holds the library mutex. A priority inversion could occur if the thread were preempted. Furthermore, it is not possible for other threads to timeout on their attempt to obtain the library mutex. Therefore, each read had to be done with a short timeout after which the mutex was released, allowing other threads to continue. In contrast, with the introduction of a reading thread, the client timeout is handled perfectly by the condition variable timeout mechanism and priority inversion can only occur during the short time period when a low-priority thread checks to see if there are events on the input queue.

A second benefit of introducing this thread concerns interaction between output requests and input events. The X specification requires that the output queue be flushed whenever a read is done on the input stream. This is done to ensure that any commands that might trigger a response are delivered to the server before the client waits on the reply. The ported library retained this behavior, but the short timeout on the read operations (to handle the problem described above) caused an excessive number of output flushes, defeating the throughput gains of batching requests. With the introduction of a reading thread, however, there is no need to couple the input and output together. The reading thread can block indefinitely and other mechanisms such as an explicit flush by clients or a periodic timeout by a maintenance thread ensure that output gets flushed in a timely manner.

Another difference in the two approaches is the introduction of an extra thread for the batching of graphics requests. Both systems do batching on a higher level to eliminate unnecessary X requests. Xl uses the slack process mentioned previously. It makes the connection to the server asynchronous in order to improve throughput, especially when performing many graphics operations. The Xlib port uses external knowledge when the painting is finished to trigger a flush of the batched requests. This limits asynchronous graphics operations and leads to a few superfluous flushes.

In summary, this example illustrates the benefit of introducing an additional thread to help manage concurrency and interactions with external I/O events.

6. Issues in thread implementation

6.1 Spurious lock conflicts

A spurious lock conflict occurs between a thread notifying a CV and the thread that it awakens. Birrell describes its occurence on a multiprocessor: the scheduler starts to run the notified thread on another processor while the notifying thread, still running on its processor, holds the associated monitor lock. The notifyee runs for a few microseconds and then blocks waiting for the monitor lock. The cost of this behavior is that of useless trips through the scheduler made by the notifyee's processor [Birrell91].

We observed this phenomenon in our micro-visualizations even on a uniprocessor, where it occurs when the waiting thread has higher priority than the notifying thread. Figure 3 illustrates the behavior before and after the notify implementation was fixed to correct the behavior. Since the Mesa language does not allow condition variable notifies outside of monitor locks, Birrell's technique of moving the NOTIFY out of the locked region is not applicable. In our systems the fix (defer processor rescheduling, but not the notification itself, until after monitor exit) was made in the runtime implementation. The changed implementation of NOTIFY prevents the problem both in the case of interpriority notifications and on multiprocessors.

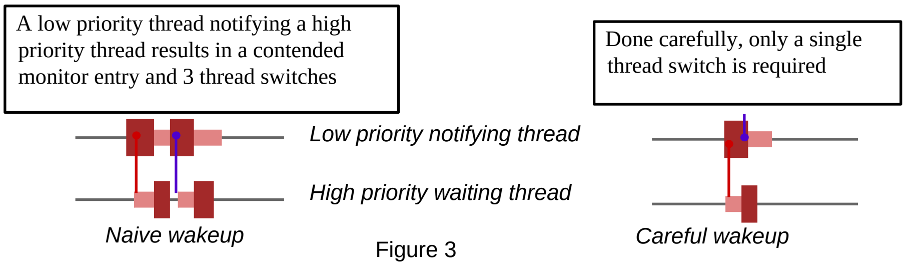


Figure 3

6.2 Priorities

PCR approximates a strict priority scheduler, by which we mean that if a process of a given priority is currently scheduled to run on a processor, no process with higher priority is ready to run. As we have seen in Section 5.2, strict priority is not a desirable model on which to run our client code: a model that provides some cpu resource to all runnable threads has proven necessary to overcome stable priority inversions. Similarly, the YieldButNotToMe and SystemDaemon hacks, also described above, violate strict priority semantics yet have proven useful in making our system perform well. The SystemDaemon hack pushes the thread model a bit in the direction of proportional fair-share scheduling (threads at each priority progress at a rate proportional to a function of the current distribution of threads among priorities), a model intuitively better suited to controlling long-term average behavior than to controlling moment-by-moment processor allocation to meet near-real-time requirements.

We do not regard this as a satisfactory state of affairs. These implementation hacks mean that the thread model is incompletely specified with respect to priorities, adversely affecting our ability to reason about existing code and to provide guidance for engineering new code. Priority inversions and techniques for avoiding them are the subjects of considerable research in the realtime computing context [Sha90][Pilling91]. We believe that someone should investigate these techniques for interactive systems and report on the result.

6.3 The effect of the time-slice quantum

Only after several months of study of Figure 2 did we realize something else it was showing us: the importance of the time-slice quantum. Figure 2 has been annotated by hand with a black stroke every 50 milliseconds denoting the end of the PCR time-slice. The end of a timeslice ends the effect of a YieldButNotToMe or a directed yield. In Figure 2 the lower priority image thread stops running and (after a brief gap) the buffer thread again runs. (The gap occurs when various threads, not shown, run briefly because they have timed out).

What we did not realize for a long time is that it is the 50 millisecond quantum that is clocking the sending of the X requests from the buffer thread. That is, the only reason this performs well is that the quantum is 50 milliseconds. For instance, if the quantum were 1 second, then X events would be buffered for one second before being sent and the user would observe very bursty screen painting. If the quantum were 1 millisecond, then the YieldButNotToMe would yield only very briefly and we would back to Figure 1 again.

Above we mentioned that it does not work to rewrite the buffer thread to sleep for a timed interval, instead of doing a yield. The reason is that the smallest sleep interval is the remainder of the scheduler quantum. Our 50 millisecond quantum is a little bit too long for snappy keyboard echoing and line drawing, both instances where immediate response is more important than the throughput improvement achieved by buffering. However, if the scheduler quantum were 20 milliseconds, using a timeout instead of a yield in the buffer thread would work fine.

We conclude that the choice of scheduler quantum is not to be taken lightly in the design of interactive thread systems, since it can severely affect the performance of different, correct, multiprogramming algorithms.

7. Visualization tools

An important tool in our study is the program that we use to examine the interactions between threads. The visualization system lets us see what threads are involved in performing a certain task and how they interact. Our visualization system consists of three major parts. The first two parts, HistorySpy and KTrace, gather information, while the third, ThreadsVis, displays the information.

HistorySpy collects detailed information about four kinds of events: thread creation (fork), thread transition to blocked state (blocked monitor entry, condition wait), thread transition to unblocked state (monitor exit, notify and broadcast) and thread transition to running state (chosen for execution by the thread scheduler). For each event the program counters on the call stack are gathered. The first few PCs in the callstacks are later used to identify the kind of transition and all the PCs are mapped to procedure names for inspection by the ThreadsVis user. The time overhead of HistorySpy is dominated by recording the call stacks, which takes about 45 microseconds per frame on a Sparcstation 2. Events have an average of 12 frames. A measured series of tasks took about 20% longer with HistorySpy turned on and gathering about 300 events/second. Note that events such as uncontended monitor entry and exit and notifies with no waiters do not cause transitions and are not recorded by HistorySpy.

KTrace is a similar system for monitoring Unix kernel events. It consists of a SunOS kernel modified to put events into a large in-memory ring buffer and a user process that empties the buffer. KTrace events are transition to idle, user process execution, interrupts, trap entry and exit, page faults, internal kernel sleeps and system calls. Detailed information about each type of event is also gathered, such as which process, which interrupt, the event slept upon and so on. (KTrace does not capture call stacks.) Tracing of different types of events can be turned on and off dynamically, to limit unnecessary overhead. A disabled trace point just costs a few instructions and an enabled trace point costs about 5 microseconds on a Sparcstation 2. This includes reading the microsecond hardware clock and adding a record to the circular trace buffer. The biggest source of overhead is the background job that runs periodically and copies the kernel’s trace buffer to disk. Programs use from 2% to 3% more CPU time when tracing is turned on, but on a uniprocessor they run 10% to 11% slower in terms of elapsed time because they compete with the process that empties the trace buffer. The same measurement on a multiprocessor shows only the 2-3% slowdown as expected.

The ThreadsVis program automatically converts the data from KTrace and HistorySpy into an interactive graphical form. Using a graphical editor with embedded buttons [Pier88][Bier92], the picture is easily panned, zoomed, or edited. Any item in the picture can be clicked to pop up additional information, including the callstacks for thread events. There are obvious advantages to viewing the trace data graphically and interactively, but a complete discussion of thread visualization techniques is beyond the scope of this paper. (The reader interested in parallel visualization systems might start with the references in [Halstead90].)

The PCRTrace package, used in gathering data for the statistics for Section 3.1, is a PCR instrumented to record events like thread switches, thread preemptions, IO operations performed, contended and uncontended monitor entry, etc. Events are recorded in an in-memory trace buffer and analyzed off-line.

8. Suggestions for future work

Two areas stand out as needing further work. First, micro-visualization of threads is very compute and memory intensive. We were not satisfied with either the performance or the user interface of our visualization tools, even when we ran them on 50 MIP multiprocessor RISC machines. We had two persistent user interface problems: managing the huge number of events in even a few seconds of visualization (our figures here were drastically cut down for purposes of presentation); and better relating the visualized events to what occured during event capture.

Second, we believe that work from the real-time scheduling community must be explored in the context of large, interactive systems. Both strict priority scheduling and fair-share priority scheduling seem to complicate rather than ease the programming of highly reactive systems.

9. Conclusions

We have analyzed two interactive computing systems that make heavy use of light-weight threads and have been in daily use for many years by many people. The model of threads both these systems use is one based on preemptable threads that use monitors and condition variables to control thread interactions. Both systems run on top of the Portable Common Runtime, which provides preemptable user-level threads based on a strict priority-based scheduling model.

Our analysis has focused on how people use threads for program structuring rather than for achieving multiprocessor performance. As such, we have focused more on a static analysis of program code, coupled with an analysis of thread micro-behavior, than on macroscopic thread runtime statistics.

These systems exemplify some paradigms that may be useful to the thread programmer who is ready for more advanced thread uses, such as slack processes, serializers, deadlock avoiders and task rejuvenators.

These systems also show that even very experienced communities may struggle at times to use threads well. Some thread paradigms using monitor mechanisms are easy for programmers to use; others, such as slack processes and priorities, challenge both application programmers and thread system implementors.

One of our major conclusions is a suggestion for future work in this area: there is still much to be learned from a careful analysis of large systems. The more unusual paradigms described in this paper, such as task rejuvenation and deadlock avoidance, arose from a relatively small community over a small number of years. There are likely other innovative uses of threads waiting to be discovered.

Reading code and microscopic visualization taught us new things about systems we had created and used over a ten year period. Even after a year of looking at the same 100 millisecond pictures we are seeing new things in them. To understand systems it is not enough to describe how things should be; one also needs to know how they are.

Acknowledgements

This work would not have been possible without the efforts of the hundreds of Cedar and GVX programmers who created these systems. Alan Demers created the PCR thread package and participated in hours of discussions regarding thread primitives and scheduling. John Corwin and Chris Uhler assisted us in building and using instrumented versions of GVX.

This work was supported by Xerox. Portions were also paid for by ARPA under contract DABT63-91-C-0027.

Bibliography

[Accetta86] Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, ‘‘Mach: A New Kernel Foundation for UNIX Development.’’ *Proceedings of the Summer 1986 USENIX Conference*, July 1986.

[Bier92] E. Bier. ‘‘EmbeddedButtons: Supporting Buttons in Documents.’’ *ACM Transactions on Information Systems,* 10(4), October 1992, pages 381-407.

[Birrell91] A. Birrell. ‘‘An Introduction to Programming with Threads.’’ in *Systems Programming with Modula-3,* G. Nelson editor. Prentice Hall, 1991, pp. 88-118.

[Custer93] H. Custer. *Inside Windows NT*. Microsoft Press, 1993.

[Draves91] R. Draves, B. Bershad, R. Rashid, R. Dean. ‘‘Using Continuations to Implement Thread Management and Communication in Operating Systems.’’ *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, in *Operating Systems Review*, 25(5), October 1991.

[Frailong93] J. Frailong, M. Cekleov, P. Sindhu, J. Gastinel, M. Splain, J. Price, A. Singhal. ‘‘The Next-Generation SPARC Multiprocessing System Architecture.’’ *Proceedings of COMPCON 93*.

[Halstead90] R. Halstead, Jr., D. Kranz. ‘‘A Replay Mechanism for Mostly Functional Parallel Programs.’’ *DEC Cambridge Research Lab Technical Report 90/6*, November 13, 1990.

[Jacobi92] Jacobi, C. ‘‘Migrating Widgets.’’ *Proceedings of the 6th Annual X Technical Conference* in *The X Resource*, Issue 1, January 1992, p. 157.

[Lampson80] B. Lampson, D. Redell. ‘‘Experience with Processes and Monitors in Mesa.’’ *Communications of the ACM*, 23(2), February 1980.

[Owicki89] Owicki, S. ‘‘Experience with the Firefly Multiprocessor Workstation.’’ *Research Report 51*, Digital Equipment Corp. Systems Research Center, September, 1989.

[Pier88] K. Pier, E. Bier, M. Stone. ‘‘An Introduction to Gargoyle: an Interactive Illustration Tool.’’ J.C. van Vliet (editor), *Document Manipulation and Typography, Proceedings of the Int’l Conference on Electronic Publishing, Document Manipulation and Typography*, Nice (France), April 20-22, 1988. Cambridge University Press, 1988, pp. 223-238.

[Pilling91] M. Pilling. ‘‘Dangers of Priority as a Structuring Principle for Real-Time Languages.’’ *Australian Computer Science Communications*, 13(1), February 1991, pp. 18-1 - 18-10.

[Powell91] M. Powell, S. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks. ‘‘SunOS Multi-thread Architecture.’’ *Proceedings of the Winter 1991 USENIX Conference*, Jan. 1991, pp. 65-80.

[Scheifler92] R. Scheifler, J. Gettys. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*. Third edition. Digital Press, 1992.

[Schmitmann93] C. Schmidtmann, M Tao, S. Watt. ‘‘Design and Implementation of a Multi-Threaded Xlib.’’ *Proceedings of the Winter 1993 Usenix Conference*, Jan. 1993, pp 193-204.

[Sha90] L. Sha, J. Goodenough. ‘‘Real-Time Scheduling Theory and Ada.’’ *IEEE Computer*, 23(4), April 1990, pp 53-62.

[Sites93] Sites, R. ‘‘Alpha AXP Architecture.’’ *CACM* 36(2), February, 1993, pp. 33-44.

[Smith82] D. Smith, C. Irby, R. Kimball, B. Verplank, E. Harslem. ‘‘Designing the STAR User Interface.’’ *BYTE Magazine*, (7)4, April 1982, pp. 242-282.

[Swinehart86] D. Swinehart, P. Zellweger, R. Beach, R. Hagmann. ‘‘A Structural View of the

Cedar Programming Environment.'' *ACM Transactions on Programming Languages and Systems*, (8)4, October, 1986.

[Weiser89] M. Weiser, A. Demers, C. Hauser. ''The Portable Common Runtime Approach to Interoperability.'' *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, in *Operating Systems Review*, 23(5), December 1989.

Appendix: Summary of Cedar and GVX eternal threads.

These threads are created when Cedar or GVX is started and they never exit. These one line descriptions give some sense of what long-lived threads are used for in these systems.

Cedar eternal thread descriptions:

0: Idler thread created by PCR to occupy its scheduler when nothing else is runnable.
1: Idler thread created by PCR to occupy its scheduler when nothing else is runnable.
2: Unknown sleeper thread.
3: Feedback window typescript thread.
4: Garbage collection daemon.
5: System daemon. Provides a proportional scheduling capability.
6: Finalization manager. Forks threads to handle finalization tasks when necessary.
7: Waits for Unix signals and feeds them to Cedar.
8: PFS file system cache sweeping thread.
9: Cache sweeper for Cedar's (distributed) naming facilities.
10: SunRPCAuth housekeeping thread.
11: Serializer thread for window painting operations.
12: Serializer thread for window painting operations.
13: Serializer thread for window painting operations.
14: Caret blinking thread.
15: Serializer thread for window input events.
16: Serializer thread for window input events.
17: Serializer thread for window input events.
18: Serializer thread for window input events.
19: Manager thread for ForkOps package (provides "hot" threads for periodically executing and delayed execution tasks).
20: Manager thread for CommTimer package (provides a timeout service).
21: Buffering thread for collecting bytes to be sent to the network.
22: SUN YP name cache sweeper thread.
23: Fake X server events generator (for when the X server is quiescent).
24: GC cursor thread. Waits until the GC-happening cursor should be shown and does so.
25: Unknown sleeper thread.
26: Serializer thread for typescript painting operations.
27: Input serialization thread.
28: Input event-to-function matcher thread.
29: Popup menu thread. Waits for popup requests and executes them.
30: X server connection input events thread.
31: Buffered X server connection output thread.
32: ForkOps thread, waiting for a task to execute.
33: Unknown.
34: ForkOps thread, waiting for a task to execute.
35: Commander typescript (equivalent to a Unix shell) thread.
36: Clock thread.
37: ForkOps thread, waiting for a task to execute.

GVX eternal thread descriptions:

0: Idler thread created by PCR to occupy its scheduler when nothing else is runnable.
1: PCR console reader thread.
2: System daemon.
3: Finalization manager.

4: GVX desktop events thread.
5: User abort thread. (Only the System Daemon runs at higher priority.)
6: Inactive session timeout thread.
7: Distributed sessions probe thread.
8: Unknown.
9: Display flusher thread.
10: Network listener thread.
11: Network listener thread.
12: Interrupt thread. (Only the System Daemon runs at higher priority.)
13: X server connection event reader thread.
14: X connection event matcher.
15: X event dispatcher.
16: Background event handler.
17: Input event-to-function matcher helper thread.
18: Input event-to-function matcher thread.
19: Unknown.
20: Forked events manager thread.
21: Free pages sweeper thread.
22: Cursor blinking thread.