Runtime Performance Analysis of the M-to-N Scheduling Model

Bryan M. Cantrill

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-96-19
May 1996

Runtime Performance Analysis of the M-to-N Scheduling Model

Bryan M. Cantrill
Dept. of Computer Science
Brown University
`bmc@cs.brown.edu`

May 6, 1996

Abstract

The popular M-to-N thread scheduling model multiplexes many user-level threads on top of fewer kernel-level threads. While this model is designed to be scalable and efficient without excessive resource consumption, we isolate several elementary examples under which the M-to-N model exhibits highly nonintuitive performance. We use a runtime performance monitor for multithreaded programs which we have developed, ThreadMon, to explain the causes of the unexpected results. We conclude that the complexity and nondeterminism exported to the programmer make performance tuning to the intricate M-to-N model extremely difficult. Moreover, we show that the insulation of user-level scheduling from kernel-level scheduling can have undesirable side-effects.

1 Introduction

Since the approval of the POSIX threads standard [6], multithreaded programming has experienced an explosion of interest. Many applications have appreciated performance gains through multithreading; the popular Netscape web browser is a highly visible example of a program whose multithreading has an obvious and substantial performance impact. However, despite its hype, Netscape runs no faster on a multiprocessor than it runs on a uniprocessor. Netscape's bottleneck on the multiprocessor has nothing to do with poor implementation, but is rather a direct result of the scheduling model upon which it is implemented. As multiprocessors proliferate, the performance characteristics of the underlying thread scheduling model will become increasingly significant. In this paper, we perform runtime analysis of one of the more popular models, the M-to-N [8] model, as implemented on SunOS 5.5[1]. Our results were not what we expected; to explain the phenomena behind them, we use a runtime tool we developed, ThreadMon, to effectively identify some performance caveats of the model. Section 2 gives an overview of competing thread scheduling models, and Section 3 gives the specifics of the Solaris implementation. Section 4 motivates the need for runtime thread scheduler performance analysis tools, and discusses the specifics of the tool we developed. Section 5 gives our experiments and results, and Section 6 gives our conclusions.

2 Overview of Competing Threads Models

There are several competing threads models, all of which require varying degrees of kernel support and offer different advantages and disadvantages.

[1] SunOS 5.x and Solaris 2.x are equivalent

2.1 N-to-1 model

In a kernel which does not support multiple threads of control, multithreading can be implemented entirely as a user level library. These libraries, without the kernel's knowledge, schedule multiple threads of control onto the process's single kernel-level (or *hot*) thread. Thus, just as a uniprocessor provides the illusion of parallelism by multiplexing multiple processes on a single CPU, user-level threads packages provide the illusion of parallelism by multiplexing multiple user-level (or *cold*) threads on a single hot thread. Due to the many-to-one relationship between cold threads and hot threads, this is referred to as the *N-to-1* model [8]. There are several advantages to this model:

- Cheap synchronization. When a cold thread wishes to perform synchronization, the user-level thread library checks to see if the thread needs to block. If it does, then the library enqueues the thread on the synchronization primitive, dequeues a cold thread from the library's run queue, and switches the active thread. If it does not need to block, then the active thread continues to run. No system calls are required in either case.
- Cheap thread creation. To create a new thread, the threads library need only create a context (i.e. a stack and registers) for the new thread and enqueue it in the user-level run queue.
- Resource efficiency. Kernel memory isn't wasted on a stack for each user level thread. This allows for a number of threads limited only by the amount of virtual memory.
- Portability. Because user-level threads packages are implemented entirely with standard UNIX and POSIX library calls (e.g. with `getcontext(2)` and `setcontext(2)`), they are often quite portable.

However, the N-to-1 model does not come without a heavy price. Specifically:

- Nonpreemptable. Once one thread has begun making a system call, it cannot be preempted by the user-level library. This becomes a significant problem when the thread needs to block in the kernel for some reason (e.g. waiting for I/O). The blocking hot thread will block the entire process, even in the presence of runnable cold threads. While it adds significantly to implementation complexity, the library can circumvent this problem where asynchronous variants of system calls exist [7].
- Nonscalable. Multithreaded programs under the N-to-1 model will run no faster on multiprocessors than they run on uniprocessors. The single hot thread acts as a bottle neck, preventing optimal use of the multiprocessor.

Despite substantial disadvantages, the relative ease of implementation of N-to-1 threads packages has resulted in it being the most popular model to date. For example, the current versions of Netscape and Java achieve their multithreading strictly through user-level threads packages.

2.2 1-to-1 model

An obvious alternative to the N-to-1 model is to have every cold thread have its own hot thread (i.e. there is a *1-to-1* correspondence between cold threads and hot threads). This allows several advantages:

- Scalable. Because each hot thread is actually a different kernel-schedulable entity, multiple threads can run concurrently on different processors. Thus, multithreaded programs written under the 1-to-1 model can achieve linear or superlinear[2] speedup when migrated from uniprocessors to multiprocessors.

[2] A multithreaded application can see superlinear speedup due to cache and scheduling effects

- True parallelism. Unlike the N-to-1 model, threads blocking in the kernel do not impede process progress under the 1-to-1 model. When one hot thread blocks, the kernel simply deschedules it, and picks another thread off of the kernel-level run queue.

While the 1-to-1 model can represent a major performance win, it is not without its costs. Virtually all of the assets of the N-to-1 model are liabilities of the 1-to-1 model:

- Expensive synchronization. Because kernel threads require kernel involvement to be descheduled, hot thread synchronization will require a system call if the lock is not immediately acquired[3]. Estimates vary, but if a trap is required, synchronization will be from three to ten times more costly than for the N-to-1 case [9, 11].
- Expensive creation. Under the 1-to-1 model, every thread creation requires explicit kernel involvement and consumes kernel resources. The difference in creation cost depends on the specific implementation, but creating a hot thread is generally between three and ten times more expensive than creating a cold thread [11].
- Resource inefficiency. Every thread that the user creates requires kernel memory for a stack, as well as some sort of kernel data structure to keep track of it. Many parts of many kernels cannot be paged out to disk; the presence of hot threads will likely displace physical memory for applications.
- Implementation difficulty. Extensive kernel modifications are required to allow the user to have multiple hot threads per process. Many stock kernels associate a process entry directly with one runnable hot thread; separating these two entities is a nontrivial coding effort.

2.3 M-to-N model

In an attempt to combine these two models, some operating systems, notably Mach 3.0 [5], SVR4/MP and SunOS 5.x [9], provide both user-level threads and kernel threads to the programmer. User-level threads are multiplexed on top of kernel-level threads, which in turn are scheduled on top of processors. The kernel knows only about the kernel-level threads; it does not know of the multiplexing performed by the user-level scheduler. Due to the many-to-many relationship between cold threads and hot threads, this is referred to as the M-to-N model [8] (it is also referred to as the two-level model [2], the split model [11] and the LWP model). By taking a hybrid approach, this model aims to combine the advantages of both the N-to-1 model and 1-to-1 model, while minimizing those models' disadvantages.

The most important attribute of this approach is that it gives the programmer explicit control over the performance of the application. Threads which are critical can have a 1-to-1 relationship with an underlying hot thread, while a large number of less critical threads can be multiplexed on a single hot thread. Under the POSIX specification [6], this mapping is specified by the *contention scope* of each thread. Threads may have either a *system scope* (and are thus associated in a 1-to-1 fashion with underlying hot threads) or a *process scope* (in which case they're multiplexed on top of hot threads[4]). While the explicit control given the programmer is the M-to-N model's greatest asset, it is also its greatest liability; deep understanding of the characteristics of the model are required to effectively use it.

2.4 Scheduler Activations

While the M-to-N model offers some advantages over the N-to-1 and 1-to-1 models, it's not an entirely satisfactory solution. Anderson et. al. suggest a dramatically different approach using *scheduler actvia-tions* [1]. A scheduler activation is like an LWP in that it represents a kernel schedulable entity, but it is not

[3]This system call can sometimes be avoided on a multiprocessor by spinning for some small period before giving up and trapping

[4]POSIX leaves it up to the implementation to define the precise semantics of process-wide contention.

permanently associated to a single process. When the kernel wishes to schedule a specific process, it makes an *upcall* to user-level, passing the user-level library as many scheduler activations as there are processors available. If the user-level library has runnable threads, it may place them on the activations. When a thread blocks in the kernel, another upcall is made with the blocking activation. It is the responsibility of the user-level library to save the blocking state, and inform the kernel that the activation may be reused. Likewise, when a thread is unblocked, the owner process is informed of the unblocking thread and passed a new activation. This model appears to elegantly solve many of the problems of the M-to-N family of models; its greatest drawback is the frequent crossings of the user-kernel boundary.

3 Solaris Implementation of the M-to-N Model

3.1 Overview

SunOS 5.x implements the M-to-N thread scheduling model [9], and introduces a new set of vocabulary: A kernel thread in Solaris is referred to as a *lightweight process* (LWP), while a cold thread is simply a *thread*. Solaris implements both the system- and process-wide contention scopes of POSIX; threads in the system-wide contention scope are referred to as *bound threads*, and likewise threads in the process-wide contention scope are *unbound threads*. The Solaris threads package is designed to isolate the programmer as much as possible from the underlying LWP implementation.

3.2 User-level Thread Scheduling

3.2.1 User-level thread states

Living, unbound threads in Solaris may be in one five states: **Stopped**, **Blocked**, **Run queue**, **Dispatchable** or **On LWP**. A thread that has been suspended is **Stopped**, while a thread blocked on a synchronization primitive is **Blocked**. If a thread is runnable, but is not running on an LWP then it is either on the **Run queue**, or, if an LWP has been found to run the thread, it is **Dispatchable**. Once a runnable thread is picked up by an LWP, it is **On LWP**. While a thread cannot be actually running on a CPU unless it is **On LWP**, being **On LWP** does not imply that a thread is running on a CPU; the underlying LWP itself could be sleeping, waiting for a processor, etc.

3.2.2 Thread-LWP interaction

Solaris implements the multiplexing of user-level threads onto LWPs by maintaining a *pool* of LWPs. Any unbound thread may run on any LWP in the pool; when a thread is ready to run (i.e. in the user-level run queue) the user-level scheduler takes an LWP out of the pool and assigns it to run the newly-runnable thread (changing the thread's state to **On LWP**). This LWP will continue to run the thread until either a thread at a higher priority becomes runnable, or the thread blocks on a synchronization primitive. Thus, the user-level threads library is *nonpreemptive* when all threads have the same priority.

When an LWP is idle (i.e. the LWP is in the pool and no threads are runnable), the user-level scheduler *parks* it in the kernel. If a thread becomes runnable while LWPs are parked, the user-level scheduler will *unpark* one of the LWPs. Once an LWP is unparked, it dequeues and runs a cold thread from the user-level run queue.

3.3 LWP Pool Management

The size of the LWP pool has a critical impact on the performance of the M-to-N model: if the number of LWPs in the pool is nearly equal to the number of threads, the implementation will act much like the 1-to-1 model. Conversely, if there there are very few LWPs in the pool, the implementation will act like the N-to-1 model.

Of particular concern is the fear of deadlock with an excessively small pool: one thread may block on a resource in the kernel and go to sleep, and by so doing block the LWP needed to run the resource-holder. To solve this problem, the threads package makes a minimal guarantee to the threads programmer: progress will always be made. This is implemented through the use of the `SIGWAITING` signal. When the kernel realizes that all of a process's LWPs are blocked at the kernel-level, it drops a `SIGWAITING` on the process. Upon receipt of the signal[5], the user-level threads package in the process will make the decision to create a new LWP or not, based on the number of runnable threads. The `SIGWAITING` mechanism makes no guarantees as to optimal use of processors on a multiprocessor. Specifically, a process may have many more runnable user-level threads than it has LWPs, but will not receive a `SIGWAITING` until all LWPs are blocked. Thus, even if there are processors available and work to be done, the `SIGWAITING` mechanism does not guarantee that the number of LWPs won't be a performance bottleneck. The weakness of the guarantee made to the programmer is evident when a multithreaded program creates its threads with the default parameters: such a program will not scale to a multiprocessor[6] Thus, if the programmer wishes to use unbound threads and achieve speedup, (s)he is required to explicitly advise on the number of LWPs. This advice is given by either setting the `THR_NEW_LWP` bit during `thr_create(3T)`, or by setting the program's *concurrency* with `thr_setconcurrency(3T)`. The concurrency is defined to be the level of *anticipated* parallelism.

Just as too few LWPs can result in performance bottlenecks, too many LWPs wastes resources. If a program only rarely has many threads which are simultaneously runnable, the `SIGWAITING` mechanism could result in an excessively large LWP pool. To deal with the problem of the pool growing larger than it has to, a kludge was introduced: when an LWP is created, it starts to *age*. If that LWP has not been used for five minutes, then it is destroyed.

4 Performance Monitoring of the Solaris M-to-N Implementation

A characteristic of the M-to-N model is that programmer decisions can have substantial performance ramifications. The Solaris implementation, in particular, has an enormous number of variables which affect performance. In order to analyze the effectiveness of the model, we must have a precise way of viewing the effect that changes in these variables have.

4.1 Existing tools

Traditional performance debuggers (e.g. call profilers) offer little help; simply knowing *where* a thread spent its time does not aid in analysis of the model. While postmortem tracing tools such as `tnfview`[7] allow some performance analysis of specific programs, they offer little insight into the effectiveness of the model itself. Moreover, the sheer volume of data generated makes it difficult to spot detrimental anomalous performance behavior.

While existing tools make the model difficult to evaluate on a uniprocessor, they make its evaluation on a multiprogrammed multiprocessor virtually impossible. To perform this kind of analysis, *runtime* correlation of thread, LWP and CPU behavior is required. To this end, we implemented ThreadMon, a tool which graphically displays the runtime interactions in the Solaris implementation of the M-to-N scheduling model.

[5]The user-level threads package dedicates thread three as blocking for a `SIGWAITING`

[6]Without explicit control, there is one LWP in the pool and threads are created unbound by default.

[7]Available at `ftp://opcom.sun.ca/pub/tnf`

4.2 ThreadMon Overview

ThreadMon displays runtime information for each of user-level threads, LWPs and CPUs. ThreadMon provides not only the *state information* for each of these computational elements, but also the *mappings* between them: which thread is running on which LWP, and which LWP is running on which CPU. Thus, to a large degree, one can watch the scheduling decisions made by both the user-level threads package and the kernel, and view how those decisions affect thread state, LWP state, and most importantly, CPU usage. We have been able to use this tool to effectively analyze the decisions made by the M-to-N model.

Implementation details

To minimize probe effect, we did not want to display runtime data on the same machine as the monitored program. Thus, ThreadMon consists of two discrete parts: a *library side* which gathers data in the monitored program, and a remote *display side* which presents the data graphically.

In order to allow monitoring of arbitrary binaries, the library side is implemented as a *shared library.* Thus, to monitor a program, the user sets the `LD_PRELOAD` environment variable to point to the ThreadMon library. This will force ThreadMon to be loaded before other shared libraries [10] (including `libthread.so`). Once loaded, ThreadMon connects to the remote display side, and continues the program. As the program continues, ThreadMon wakes up every 10 milliseconds[8], gathers data, and forwards that data to the display side. The gathering of data at the 10 millisecond rate requires approximately ten percent of one CPU. In practice, we have found that this probe effect is not significant enough to drastically change a program's performance characteristics. However, for the skeptical, a nice fringe benefit of ThreadMon is its ability to monitor itself. By examining the thread and LWP which ThreadMon uses the probe effect can be precisely determined.

ThreadMon uses several OS services to perform data gathering:

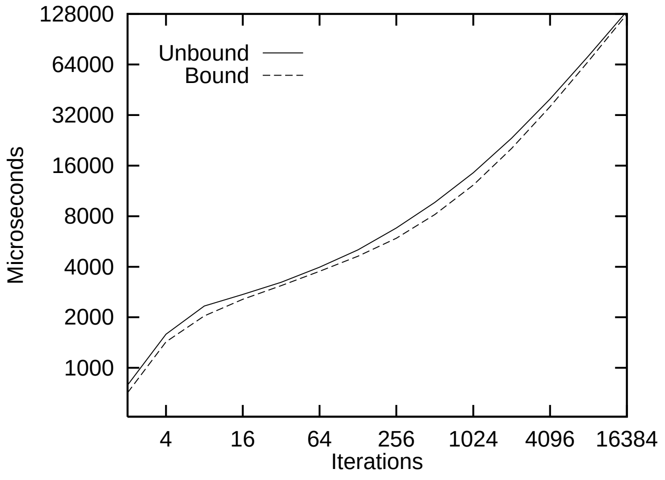
- Interpositioning. The most important data gathered by the library is done so by *interpositioning* between the user-level threads library and itself. That is, ThreadMon redefines many of the functions which the user-level threads library uses internally to change the state of threads and LWPs.
- Process file system [4]. The `/proc` filesystem offers a wealth of performance information. Specifically, `PIOCLUSAGE` is used to determine LWP states.
- Kernel statistics interface. The `kstat(7d)` interface is used to obtain CPU usage statistics.
- Trace Normal Form. Unfortunately, there is no existing OS service to determine the mappings between LWPs and CPU's. To get this information, we used the TNF kernel probes present in SunOS 5.5 and extrapolated the mapping information. For a variety of reasons, this extrapolation is extremely expensive. The TNF monitoring is off by default; when it is turned on, ThreadMon typically consumes fifty percent of one CPU on a four processor SuperSparc.
- `mmap(2)`ing of `/dev/kmem`. For some statistics, we have found it significantly faster to delve straight into kernel memory.[9]

5 Experimentation

We wished to especially analyze the performance of the M-to-N model on multiprocessors. In an effort to simulate commonly used techniques, our experiments analyzed the performance of the model using *barrier synchronization* [3]. All threads perform some amount of computation, and block until the others reach a common synchronization point. When the last thread reaches the barrier, all threads are released to perform

[8] The monitor exists as its own thread bound to its own LWP

[9] As the old adage goes, "When all else fails, open `/dev/kmem O_RDWR`."



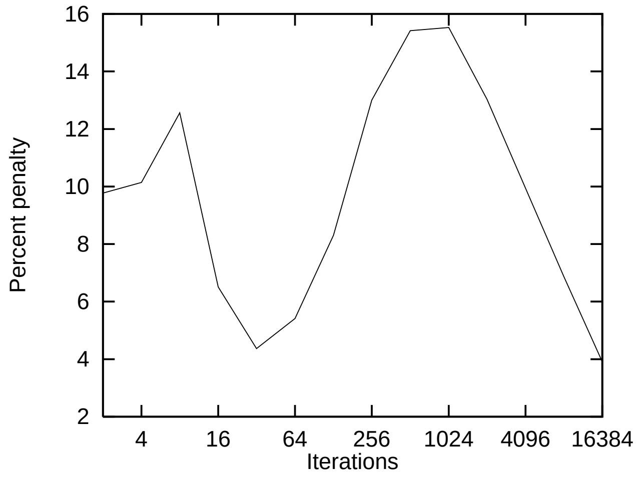


Figure 1: Performance for fine-grained barriers on a four-processor machine with four bound threads versus four unbound threads and an LWP pool of size four. NUM_ITERATIONS varies along the x-axis (see Appendix A for an explanation of this value). The figure on the right is extrapolated from the figure on the left, and represents the percentage penalty for using four unbound threads versus four bound threads.

more computation. This easily-understood and commonly used synchronization technique presented many of the performance issues of the M-to-N model. The barrier source used for the experimentation is given in figure 12. For purposes of this experimentation, we refer to a program which does a small amount of work between barrier synchronization as using *fine-grained* barriers. Likewise, a program which does a large amount of work between barriers is referred to as using *coarse-grained* barriers. The definitions used for purposes of these experiments is more precisely given in Appendix A.

5.1 Fine-Grained Barrier

On a multiprocessor machine with n processors, we felt that a multithreaded program with n bound threads should not perform any differently than a program with n unbound threads and an LWP pool of size n. To test this hypothesis, we ran our barrier (figure 12) on a four processor machine[10] with four unbound threads and an LWP pool size of four (set with `thr_setconcurrency(3T)`), and again with four bound threads. Our results were surprising: when the granularity was fine, the program using unbound threads ran slower than its equivalent using bound threads (figure 1). Moreover, we found that the precise difference was highly dependent upon the amount of work done in between barrier synchronization operations; as more work was done, the difference became increasingly small.

No amount of postulating could reveal a possible scenario which would result in this kind of performance degradation, so we monitored the program with ThreadMon. Before discussing the monitoring results, there are some caveats one should be aware of when interpreting the screen shots: in all of the examples, threads one through three are blocked for various reasons. This is entirely normal; thread one is the main thread, which spawns the worker threads and then goes to sleep. Thread two is the thread assigned to handling callouts, and is thus bound to an LWP; it spends virtually all of its life blocked in the kernel. Thread three is the thread assigned to the dynamic creation of LWPs (i.e. it handles `SIGWAITING`); it spends its life blocked at user-level. Finally, thread four is the bound thread which the monitor itself uses for reporting to the display side. Thus, all interesting behavior is seen by examining threads five and higher.

By running the program under ThreadMon the cause of the performance degradation, at least, became clear: in both versions, the working threads were dividing their time between being blocked in the barrier, and running their computation. ThreadMon revealed that in the unbound version, the threads were spending

[10] SPARCstation10/404ZX running SunOS 5.5

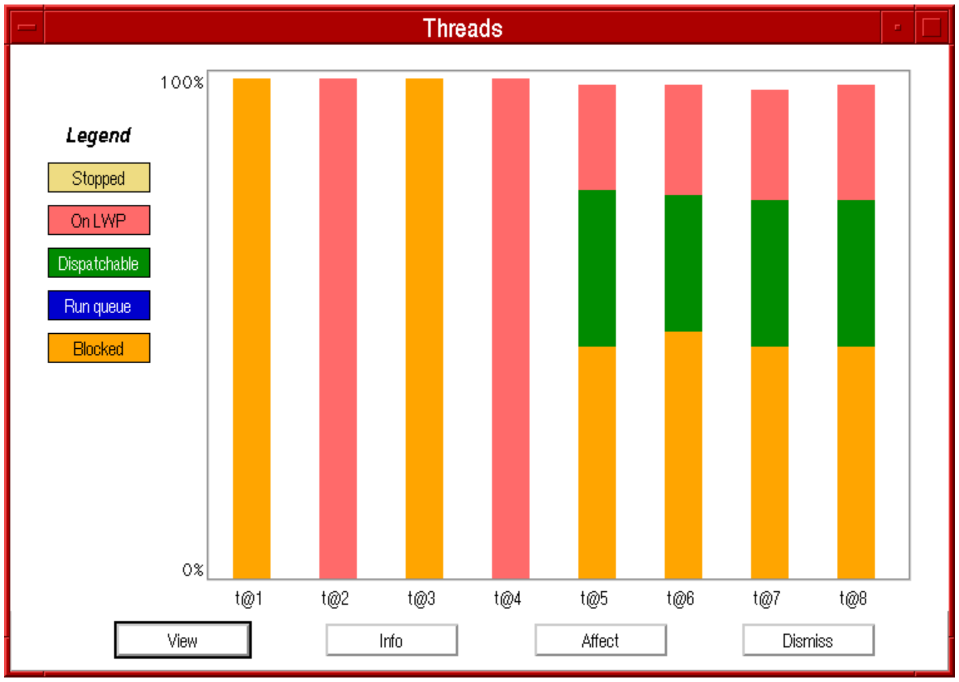


Figure 2: ThreadMon monitoring unbound version of fine-grained barrier example. Note the high time period spent **Dispatchable** by the working threads (threads five through eight). Threads one through four are blocked.

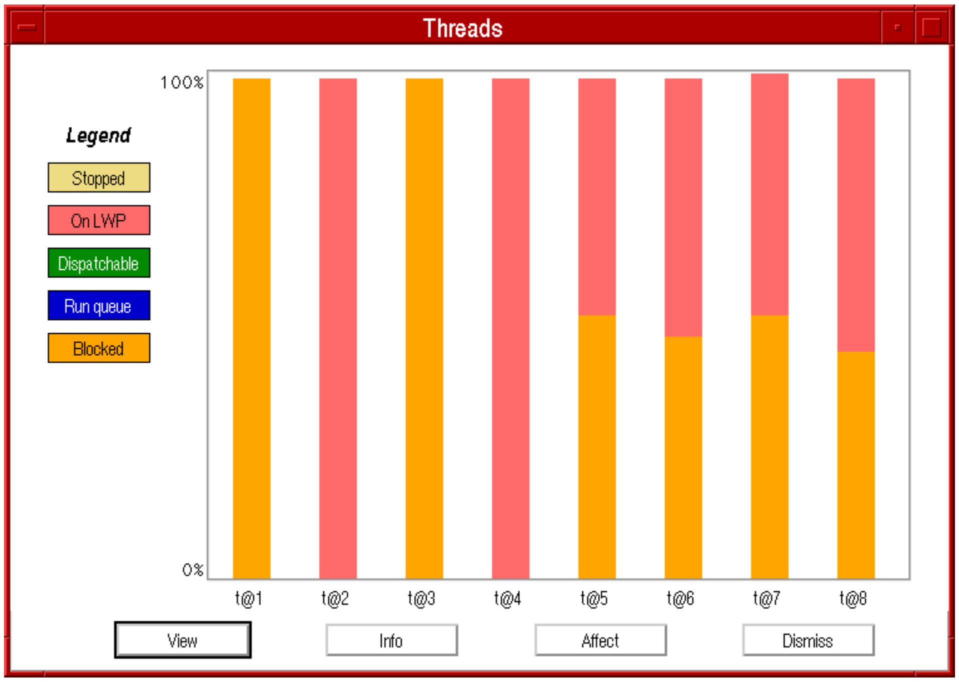


Figure 3: ThreadMon monitoring bound version of fine-grained barrier example. The time period spent **Dispatchable** has evaporated; the working threads are spending all of their time either on an LWP or blocked in the barrier.

a significant amount of time **Dispatchable** (figure 2). In the bound version, on the other hand, the time spent in this state disappeared; threads were only dividing their time between being **Blocked** and being **On LWP** (figure 3).

Theoretically, synchronization is cheap under the M-to-N model, in part because a blocking thread does not necessarily imply a blocking LWP; threads which need to block are separated from their underlying LWP, which then attempts to find other work. As mentioned in section 3.2.2, if there is no work for an LWP to do, the user-level scheduler will park it in the kernel. When a thread is made runnable, the user-level scheduler marks the thread as **Dispatchable**, and unparks an LWP. When an LWP returns from parking, it finds the **Dispatchable** thread which it needs to run, changes its state to **On LWP**, and runs it.

Our experiment demonstrated that this detachment and reattachment of threads from LWPs does not come for free; there is some latency between a thread being made **Dispatchable** and the LWP actually picking it up and running it. This latency is particularly exacerbated when contention is moderately high. The M-to-N model is touted as providing inexpensive synchronization [9, 2, 11, 8], but it appears that this is not always the case.

5.2 Coarse-grained barriers

We noticed that once the amount of work done between each barrier synchronization became nontrivial, the difference between n bound threads and n unbound threads with an LWP pool of size n disappeared. We wished to explore the performance of coarse-grained barriers when the number of threads exceeded the number of LWPs in the pool. Specifically, we believed that based on the monitoring of code from [8], the nonpreemptability of the user-level threads package would have nondesirable performance effects. Indeed, our results (figure 4) show that when the number of threads is not a multiple of the number of processors, the version using unbound threads experiences significant performance degradation.

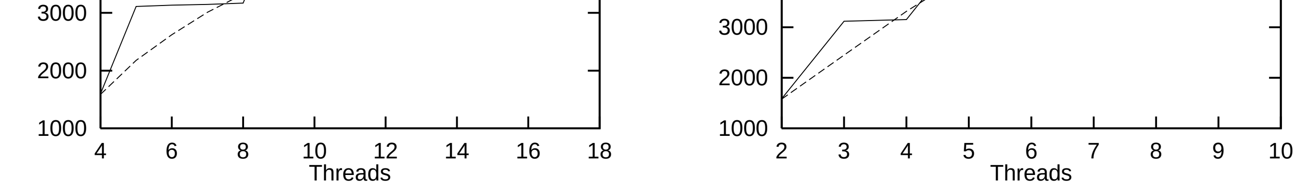


Figure 4: Coarse-grained barrier synchronization on a four-processor machine (left figure) and two-processor machine (right figure). The unbound version was started with as many LWPs as processors. When the number of threads is not a multiple of the number of CPUs, performance is worse than for the bound version. This problem is a result of the lack of time quantum preemption in the user-level threads package.

The cause of the problem was revealed by running the program under THREADMON. Figures 5 through 8 display THREADMON running the coarse-barrier example with five unbound threads and an LWP pool size of four on a four-processor machine.

The first interesting behavior we noticed was by examining the thread states: half the time, four workers were **On LWP** and one worker was on the **Run queue** (figure 5), while the other half of the time, four workers were **Blocked** while one worker was **On LWP** (figure 6). The phenomenon at work became more

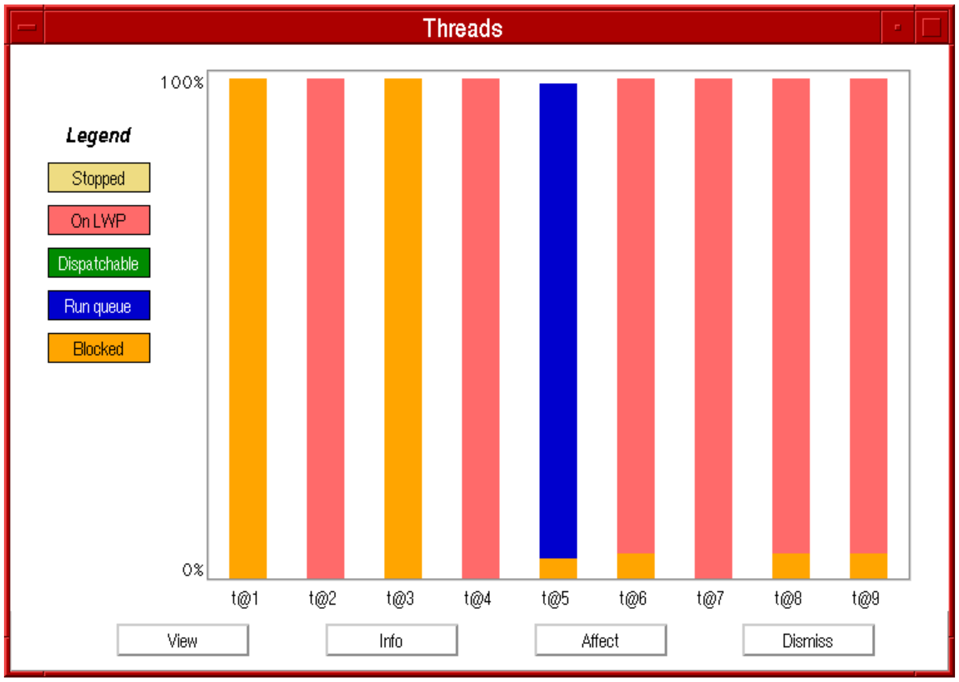


Figure 5: THREADMON monitoring unbound version of coarse-grained barrier example. Threads six, seven, eight and nine have been picked up by LWPs. Thread five is in the **Run queue**, waiting for an available LWP.

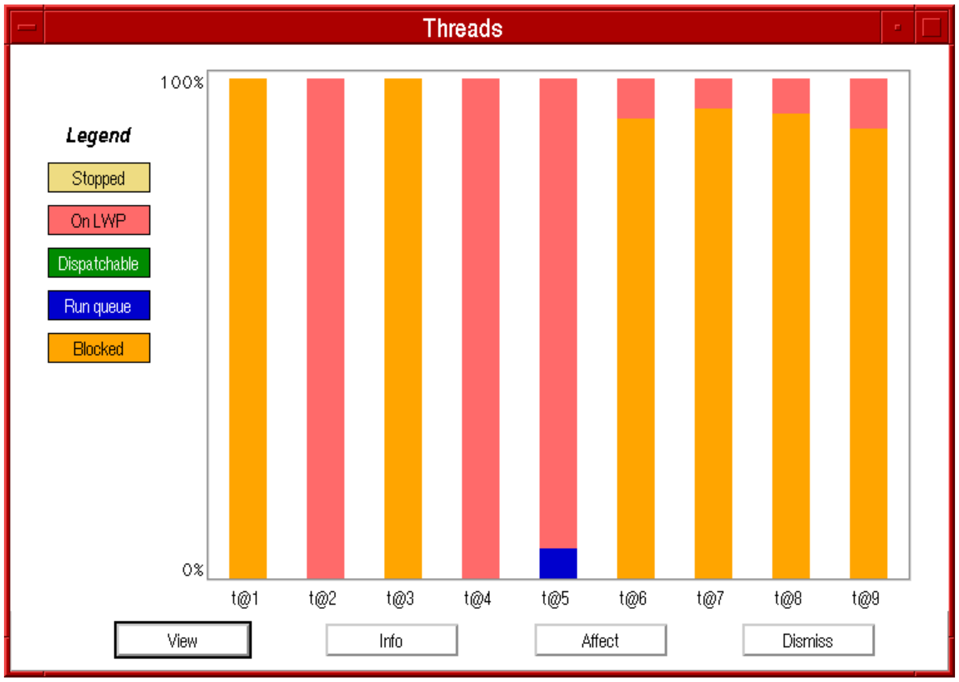


Figure 6: Threads six through nine have now completed, and are blocking on the barrier condition. Thread five, meanwhile, has been picked up by an LWP.

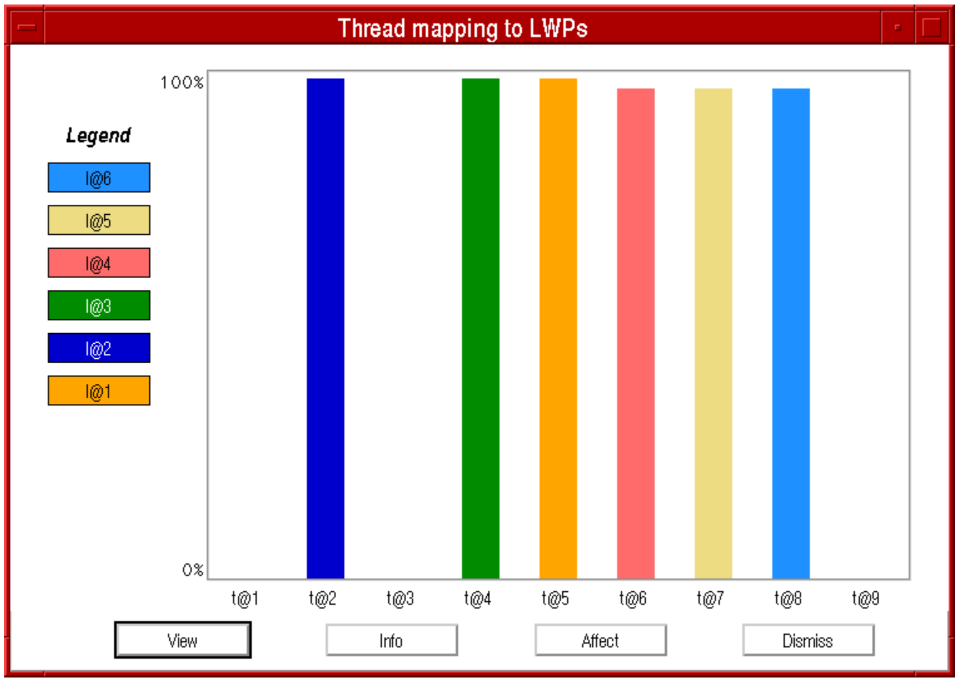


Figure 7: The mapping of threads to LWPs. Threads one and three are both **Blocked** at user-level. Threads two (callouts) and four (ThreadMon) are both bound to LWPs two and three, respectively. The four LWPs in the pool have picked up threads five through eight; thread nine is in the **Run queue**.

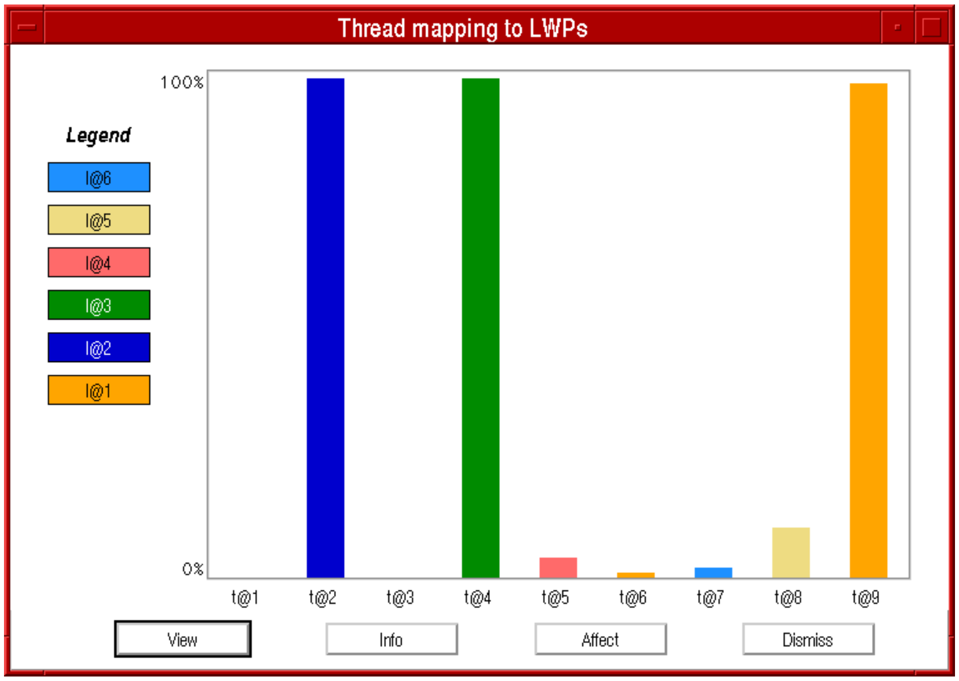


Figure 8: Threads five through eight have finished their work; LWP one has returned to pick up thread nine. The other threads are **Blocked**, waiting for thread nine to perform the computation before continuing.

clear by examining the mapping between threads and LWPs: half the time, four of the worker threads were being run by the four LWPs in the pool (figure 7), while the other half of the time, one LWP was running the remaining worker (figure 8). From this information, we were able to determine precisely what was happening: when the barrier was released, four of the five threads were picked up to run by the four LWPs in the pool (threads five, six, seven and eight in figure 7). Due to the lack of time-quanta based preemption at the the user-level, these four threads were run for the duration of the computation. Thus, when the entire computation finished some (perhaps significant) amount of time later, there was still one thread which had not yet begun to compute (thread nine in figure 7). Three LWPs then had to park, waiting for the one LWP which picked up the last thread to perform the computation (figure 8). As figure 4 shows, the time to complete the entire computation was effectively doubled. In the version using five bound threads, on the other hand, the kernel takes care of time slicing the LWPs as their time quanta expire. The bound version thus runs nearly[11] exactly 5/4 as long when running five threads as when running four threads.

5.3 Performance under load

It appeared, then when implementing a barrier which requires more threads than processors, one should be sure that the number of threads is a multiple of the number of processors. We wished to investigate this hypothesis, specifically focusing on machines under load. We ran another process which takes up one hundred percent of two CPU's. Our results, shown in figure 9, reveal that when another compute-intensive process is running, performance will overly degrade when threads are unbound. The reason for this degradation highlights one of the design tradeoffs of the M-to-N model. By multiplexing several user-level threads on a single LWP, the M-to-N model is able to allow cheap synchronization and thread creation. Creating concurrency without the kernel's knowledge has drawbacks, however. Specifically, the user-level threads package cannot inform the kernel that it *could* actually be running m threads in parallel; the kernel sees only the n LWPs. Thus, the kernel does not *fairly distribute computational resources*. Instead of giving the process computation time proportional to m, it gives the process computation time proportional to n. The version using unbound threads will starve on loaded machines; the version using bound threads scales perfectly.

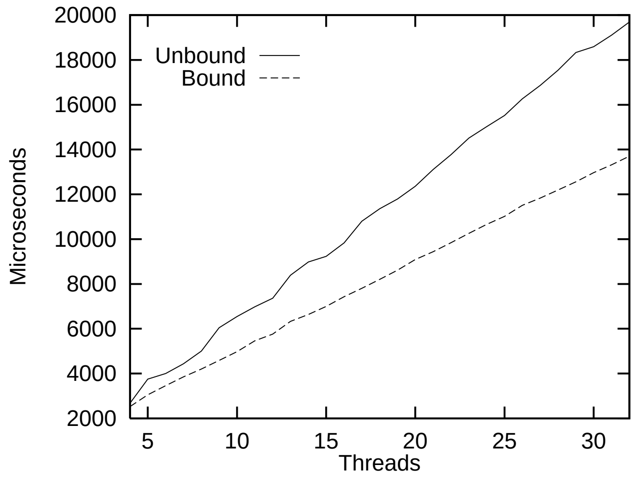


Figure 9: Performance of the coarse grained barrier on a four-processor machine when another process is consuming two CPU's. The unbound version has four LWPs in its LWP pool. Given the presence of the two other LWPs in the system, the unbound version is guaranteed to never get any more than 4/6 of the CPU resources, regardless of how many threads it has runnable. The amount of computing resources dedicated to the bound version, on the other hand, rises linearly with the number of threads.

[11]Barring scheduling overhead and cache effects, both of which are amortized in the coarseness of the barrier synchronization in figure 4.

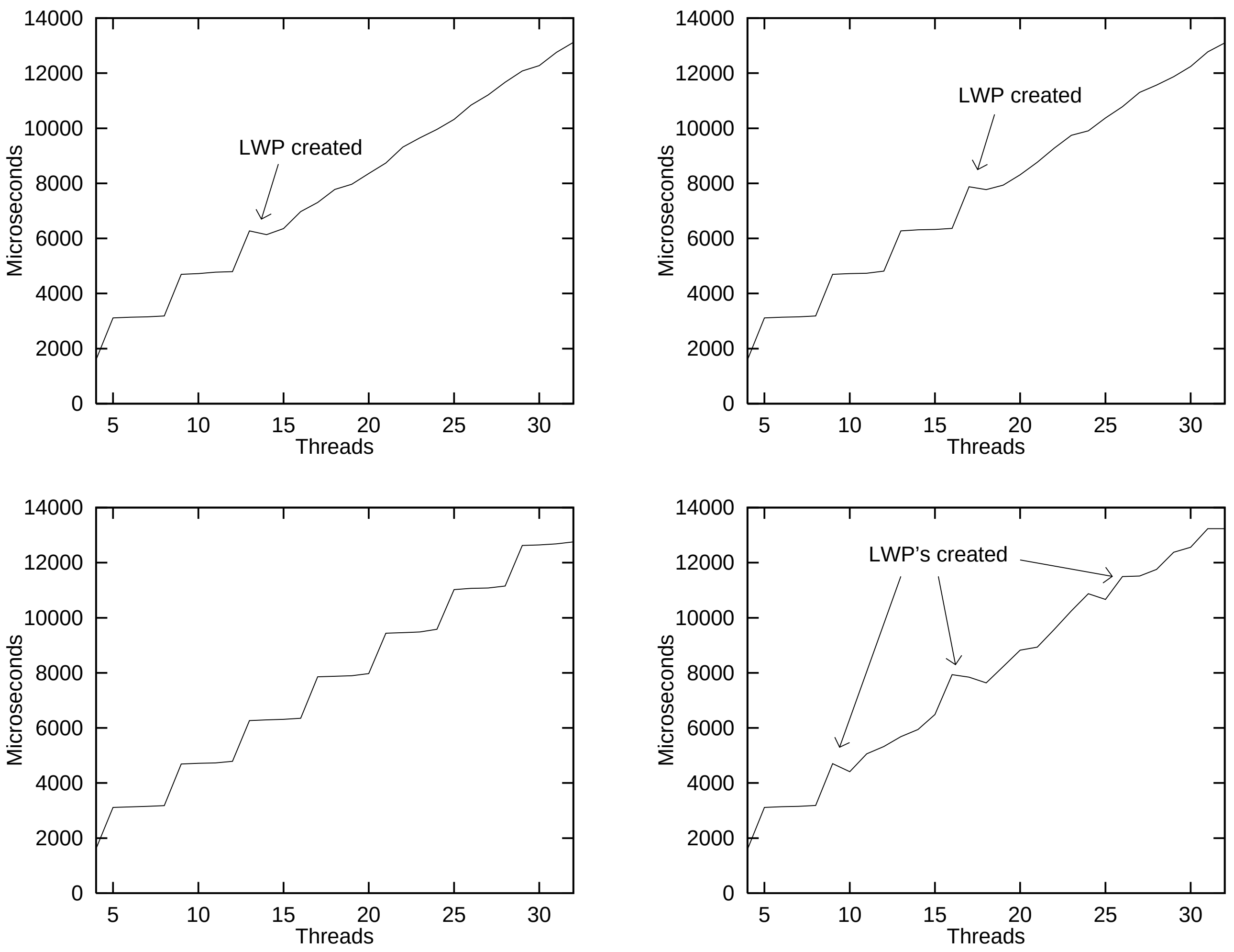


Figure 10: Four consecutive runs of the experiment in figure 4 showing the nondeterministic nature of the **SIGWAITING** mechanism. All experiments started with a pool of four LWPs on a four-processor SuperSparc. As shown, the number of additional LWPs created ranged from zero to three.

5.4 Nondeterministic performance

Disconcertingly, running our tests multiple times did not always generate the same results. At first we suspected deficiencies in our monitoring code or in the experimental conditions. It was puzzling, however, that our bound version never felt corresponding effects. Running the program under ThreadMon showed that something else entirely was the cause of the experimental differences. Specifically, the threads package would, every once in a while, act on a **SIGWAITING** from the kernel, and create another LWP. As figure 10 shows, the creation of an LWP had a significant impact on performance; the distinctive "step" shape completely disappeared. ThreadMon was very useful in determining exactly what was transpiring here; figure 11 shows an execution with ten unbound worker threads running on five LWPs. By watching the thread states, we observed that the threads were not hitting the barrier simultaneously. In figure 11 we can look for the thread that has been blocked the longest in order to determine who hit the barrier first. We can see this "staggering" of threads again in the order in which threads come off of the run queue. The staggering assures us that we won't get into the situation of figure 6 where all threads had completed the computation before the last one began. We are thus guaranteed much better performance with five LWPs than with four; this is shown clearly in figure 10.

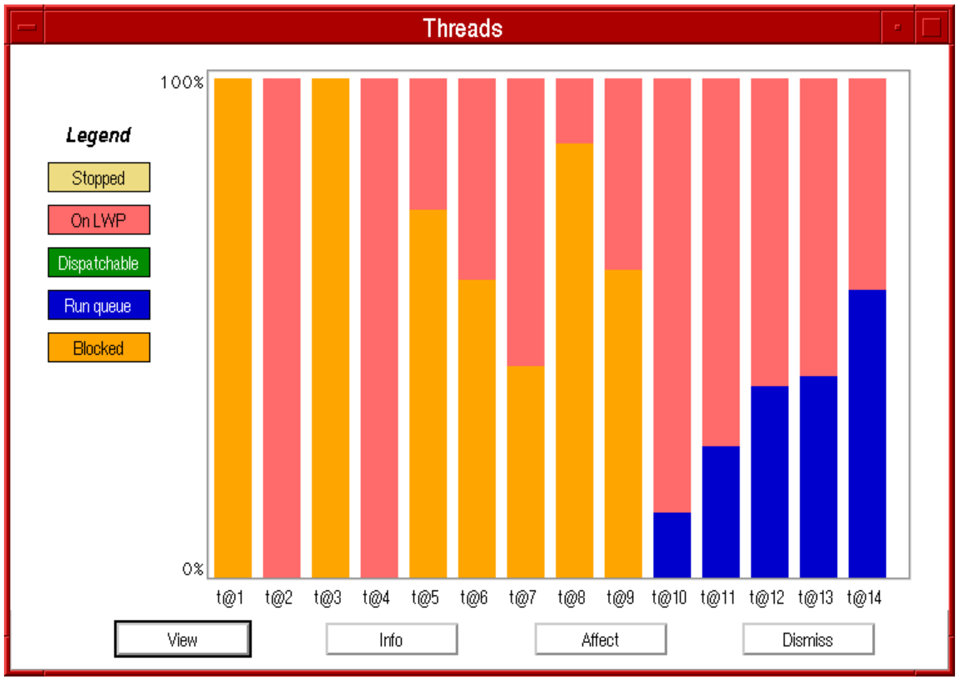


Figure 11: Here we see the staggering of execution which five LWPs on four CPUs creates. From the figure, we can infer that thread eight was the first to reach the barrier (it's been **Blocked** the longest), followed by threads five, nine, six and seven. By examining who was on the **Run queue** the least recently, we can further determine that the five LWPs running those threads picked up threads ten, eleven, twelve, thirteen and fourteen respectively.

6 Conclusions

The benefits of the M-to-N model are well known; the number of user-level threads is virtually unlimited, thread synchronization and creation times are relatively low, and the savvy programmer is afforded the ability to optimally use advanced architectures. To exploit the full advantages of the model, however, requires a deep understanding of its implementation. Using ThreadMon we were able to discover several scenerios in which the Solaris implementation of the M-to-N thread scheduling model acted in a way which was nonintuitive or nondeterministic. Specifically, we have shown that:

- Separating a cold thread from its underlying hot thread is not without performance overhead. We showed that there were cases in which this overhead could cause up to a fiteen percent performance hit.
- Lack of preemptability based on time quanta creates performance problems whenever performance is defined by the last thread to finish computation. Barrier synchronization is a common technique which has this attribute.
- Nondeterministic LWP creation can have an appreciable and unpredictable impact on performance. The nondeterministic nature of the `SIGWAITING` mechanism makes performance debugging difficult. Moreover, the guarantees provided by `SIGWAITING` are too weak to allow efficient use on a multiprocessor, and, more generally, the policies regarding LWP pool size are too conservative to scale.

Furthermore, we believe that we have demonstrated several important properties inherent to the M-to-N model:

- The insulation of user-level scheduling activity from kernel-level scheduling activity can have unintuitive performance effects. This is particularly evident when machines are heavily loaded; processes will only get time from the kernel relative to their number of LWPs, not their number of runnable threads.

- By introducing a second layer of scheduling, the M-to-N model makes performance tuning to the model exceedingly difficult. Moreover, the nonintuitive performance attributes of the model makes performance analysis necessary even for reasonably trivial applications.

The dominating trait of the M-to-N scheduling model is its complexity. Unfortunately, virtually all of its complexity is exported to the programmer. The net result is that programmers must have a complete understanding of the model and the inner workings of its implementation in order to be able to successfully tap its stengths and avoid its pitfalls.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principle*, pages 95–109. ACM SIGOPS, October 1991. Published in ACM Operating Systems Review Vol.25, No.5, 1991. Also published ACM Transactions on Computer Systems Vol.10 No.1, pp.53–70, Feb 92.

[2] Ben Catanzaro. *Multiprocessor System Architectures.* SunSoft Press, 1994.

[3] Ding-Kai Chen, Hong-Hen Su, and Pen-Chung Yew. The impact of synchronization and granularity in parallel systems. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990. Also as Tech report 942 - University of Illinois at Urbana-Champaign, Centre of Supercomputing Research and Development.

[4] Roger Faulkner and Ron Gomes. The process file system and process model in UNIX system V. In *USENIX Conference Proceedings*, pages 243–252, Dallas, TX, January 21-25 1991. USENIX.

[5] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. UNIX as an application program. In *USENIX Conference Proceedings*, pages 87–96, Anaheim, CA, Summer 1990. USENIX.

[6] Institute for Electrical and Electronic Engineers. *POSIX P1003.4a, Threads Extensions for Portable Operating Systems*, 1994.

[7] Thomas W. Doeppner Jr. Threads: A system for the support of concurrent programming. Technical Report CS-87-11, Brown University, Department of Computer Science, Providence, RI 02912, Jun 1987.

[8] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads.* SunSoft Press, 1996. ISBN: 0131723898.

[9] M. L. Powell, Steve R. Kleiman, Steve Barton, Devang Shah, Dan Stein, and Mary Weeks. SunOS multi-thread architecture. In *Proceedings of the Winter 1991 USENIX Technical Conference and Exhibition*, pages 65–80, Dallas, TX, USA, January 1991.

[10] Sun Microsystems. *SunOS 5.3 Linker and Libraries Manual*, 1993.

[11] Uresh Vahalia. *UNIX Internals: The New Frontiers.* Prentice Hall, 1996.

A Experimentation Implementation Details

```
void work (int thr_id) {
  for (;;) {
    /* the simulated work */
    int iter, i, j;
    for (iter = 0; iter < NUM_ITERATIONS; iter++) {
      for (i = 0; i < NUM_ROWS; i++)
        for (j = 0; j < NUM_COLS; j++)
          mat[thr_id][i][j]++;
    }

    /* acquire the mutex associated with the barrier */
    mutex_lock (&barrier_mtx);

    /* check to see if this is the last thread to arrive */
    if (++count == NUM_THREADS) {

      /* if it is clear the count */
      count = 0;

      /* ...time accouting is performed here... */

      /* finally kick all threads waiting */
      cond_broadcast (&barrier_cond);
    } else {
      /* Otherwise, wait to be kicked */
      cond_wait (&barrier_cond, &barrier_mtx);
    }
    mutex_unlock (&barrier_mtx);
  }
}
```

Figure 12: Barrier source

For the experiments using fine-grained barriers, NUM_ROWS and NUM_COLS were both set to two (the experiments varied NUM_ITERATIONS; see figure 1). For the experiments using coarse-grained barriers, NUM_ROWS and NUM_COLS were set to 500, and NUM_ITERATIONS was set to 10.