Multithreading in the Solaris™ Operating Environment

A Technical White Paper

Table of Contents

Introduction **1**
A Decade of Change 1
Responding to Change 2
Simply a Better Implementation 2
Scalable Threaded Applications 2
Performance, Availability, and Ease of Use 3
The Next Logical Step Forward 3
Significant Gains for Java™ Benchmarks 3
Overview 3
A Brief History of Threads **5**
Breaking Moore's Law 5
Asymmetric Multiprocessors 6
An Asymmetric Symmetric Multiprocessor 6
Scalable Multiprocess Applications 6
Smashing the Monolith 6
Back to the Drawing Board 6
Beyond Multiprocessing 6
Two-Tiered Advantages 8
Opening Software Foundations 8
A Symmetric Symmetric Multiprocessor 8
Reliable, Scalable Threads For All 8
Setting the Standards 8
Pthreads or UI Threads? 8
The Choice Is Yours 9
A Level Playing Field 9
Looking Forward 9
Recent Innovations 9
Preemption Control 10
Scheduler Activation 10
Adaptive Mutexes 10
More Mutex Types 11
Yet Another Memory Allocator 12

Breaking the 32-Bit Barrier 12
Scalability Challenges 13
Threads and Signals Don’t Mix 13
Managing Concurrency 14
_schedlock Considerations 14
An Alternate Implementation 15
Two Libsthread 15
Bound Threads for All 15
Improved Quality 15
Dependable Signals 16
Better Scalability 16
Adaptive Mutexes Revisited 16
Just Plain Simple 16
Moving On 17
What’s New in the Solaris 9 Operating Environment? 18
MxN Implementation Retired 18
Adaptive Mutexes Revisited 18
Adaptive Mutex Unlock 19
Uncontended Mutex Tuning 19
User-Level Sleep Queues 19
The Issue of Fairness 20
Faster Thread-Specific Data 20
Yet Another Memory Allocator Enhancement 20
Per-Process Timers 20
Thread Local Storage 21
Conclusion 21
Bibliography 22
Life Before the Solaris 9 Operating Environment 23
Starting With Solaris 8 23
Thread Synchronization 23
Thread-Specific Data 24
Prior to Solaris 8 24
Are You Hitting _schedlock? 24
Concurrency Issues 24
Signal Reliability 24
Ping-Pong Performance 25
The Basic Idea 25
A Simple Game 25
A Complex Tournament 25
Implementation 26
Baseline Measurement 26
Small-Scale Comparison 27
Test 1: Old libthread, default settings 27
Test 2: Old libthread, increased concurrency 27
Kernel Context Switch Comparison 28
Test 3: New libthread, single CPU 28

©2002 Sun Microsystems, Inc.

Test 4: Old libthread, bound threads, single CPU 29
Multiprocessor Scalability Comparison 29
Test 5: New libthread, four games 29
Test 6: Old libthread, four games, concurrency hint 30
Test 7: Same again, but with shared mutexes 30
So, You Want Thousands of Threads? 31
Test 8: New libthread 31
Test 9: Old libthread at its best? 31
Test 10: Old libthread at its best 32
What Does Ping-Pong Teach Us? 32
Ping Pong Source 33

Chapter 1

Introduction

Multithreading is a popular programming and execution model that allows multiple threads to exist within the context of a single process, sharing the process' resources but able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. However, perhaps the most interesting application of the technology is when it is applied to a single process to enable parallel execution on a multiprocessor system.

Multithreading is not new to the Solaris™ Operating Environment (OE). Threads have played an important part in enabling Sun to deliver successful, scalable, multiprocessor systems. The threading capabilities in the Solaris OE are continually being improved, and the Solaris 9 Operating Environment contains innovations for multithreaded applications — chiefly, the adoption of a highly tuned and tested "1:1" thread model in preference to the historic "MxN" implementation.

The goal of this paper is to describe the new threads implementation and encourage its use. It begins by providing historical context and recent developments. A technical audience with practical experience of multithreading in the C or Java™ programming languages or similar environments is assumed. For additional reading on the subject, *"Programming with Threads"* is highly recommended for novice and expert alike.

A Decade of Change

Many changes occurred during the first ten years that Sun provided support for multithreaded applications:

- System size (in terms of CPUs and memory) saw exponential growth
- Multithreaded programming techniques evolved and matured
- Application programming interfaces for threads became standardized

 ©2002 Sun Microsystems, Inc.

- Threads became ubiquitous in modern operating systems
- Use of the inherently threaded Java programming language became widespread
- Threaded applications became the norm, not the exception

Clearly, the most significant development has been moving multithreading from theory into practice. Along the way, the multithreaded computing landscape has changed. Factors which seemed important 10 years ago may be irrelevant today, and issues which were once considered on the fringe are now commanding the full attention of developers.

Responding to Change

The Solaris OE has always moved with the times, and a great deal has been done to improve the scalability of threads within the kernel. This has led to exciting, and sometimes surprising, innovations. One such innovation is the move away from the original MxN model to a 1:1 implementation.

Simply a Better Implementation

The Solaris 8 OE introduced an alternate implementation based on the 1:1 model which leveraged the improvements in kernel threading directly at the process level. The 1:1 implementation was simpler to develop, and its code paths were generally more efficient than those of the old implementation. In the Solaris 8 OE, the MxN implementation remains the default. Users have tested the new 1:1 implementation, and many have adopted it. Again, this is not to say that a good implementation of the MxN model is impossible, but simply that a good 1:1 implementation is probably sufficient.

Note – This paper does not attempt a discussion of the relative merits of the MxN and 1:1 threading models. The basic thesis is that the quality of an implementation is often more important.

Scalable Threaded Applications

New programming technologies often go through an early period of upheaval as developer communities begin to separate theory from practice. Initial naivete is tempered by realism and experience. Witness the history of the Java™ programming language: the initial focus on browsers and not-so-thin-client applications has matured, broadened, and shifted to embrace just about every level of computing — from consumer products to the data center. Multithreading has seen a similar evolution, and one of the shifts in emphasis has been away from “threads for everything” to “threads for scalability.”

For example, developers used to believe that a threads stack was an ideal place to store application session state. However, experience has proved that it is generally better to hold session state in well-designed data structures, and deploy a worker thread pool to deliver scalable, predictable performance.

Note – Here, "better" is used in terms of performance. It is much easier to understand a program in which all state is represented as threads stack state. This also makes tools like pstack(1) very useful for debugging. But such implementations do not scale well as the number of threads increase. For instance, a Web application that provides a dedicated thread for every user session will suffer cache and memory management issues as its mean working set increases with load.

Performance, Availability, and Ease of Use

Extensive testing and production use of the new implementation has delivered significant gains in application performance (an Oracle OLAP workload test showed up to a 400-percent performance improvement[1], for example). Meanwhile, the implementation is proving to be more robust and intuitive than its predecessor. Applications are benefitting from increased availability, and developers must deal with fewer caveats and exceptions to expected behavior. Within Sun, the new implementation is high quality and easy to maintain. It should provide a sound foundation for further enhancements to the threading capabilities of the Solaris OE.

The Next Logical Step Forward

The Solaris 8 OE alternate implementation has proven so successful that in the Solaris 9 software it is not longer the alternate implementation — it is the *only* implementation. The vast majority of applications are expected to benefit from this step forward.

Significant Gains for Java™ Benchmarks

Much of the early work on the new threads library was targeted at improving Java™ application performance. SPECjbb2000[2] is the current server-side, multithreaded Java benchmark of choice, and the new 1:1 threads implementation has performed well:

- In February 2002, Sun submitted a new world-record SPECjbb2000 result using a 72-way Sun Fire™ 15K system and the Solaris 8 07/01 Release alternate implementation.
- On October 18, 2001, Sun submitted another leading result using a 24-way Sun Fire 6800 system and the Solaris 8 02/02 alternate implementation (which is essentially a back-port of the enhanced 1:1 implementation from the Solaris 9 OE).

Overview

The remainder of this paper is divided into two chapters:

- *A Brief History of Threads* - A survey of major milestones in the development of multithreading in the Solaris Operating Environment, from versions 2.0 to 8. While providing useful history of key developments, this section also explains the benefits of upgrading from earlier Solaris releases to a more current version.
- *What's New in the Solaris 9 OE?* - A brief look at the enhancements in the Solaris 9 multithreading environment.

1. Test was done by Solaris Operating Environment engineering team with 5 users each running 24 clients with 1000 transactions

2. SPECjbb2000 is benchmark from the Standard Performance Evaluation Corporation (SPEC). Competitive claims reflect results published on www.spec.org as of January 28, 2002.

 ©2002 Sun Microsystems, Inc.

In addition to the titles recommended in the bibliography, the following appendices should prove useful — even fun — reading:

- *Appendix B: Life Before the Solaris 9 OE* - Handy hints for those who are eager to adopt Solaris 9 software as soon as possible, but must improve threads performance and availability in the interim.
- *Appendix C: Ping Pong Performance* - A brief look at a benchmark that can be used to compare basic locking and context switch performance between various threads implementations.
- *Appendix D: Ping Pong Source* - The full Pthreads source of the Ping Pong benchmark.

Chapter 2

A Brief History of Threads

In the 1980s, most UNIX® systems had a single CPU. The operating system gave an illusion of parallelism, with the scheduler doing its best to share limited processing resources between processes. Various interprocess communication facilities (pipes, shared memory, semaphores, and so on) made it possible to build complex applications using multiple cooperating processes. Some work was done on purely user-level multithreading (such as early DCE threads implementations and liblwp in Solaris 1 software). However, with only a single CPU to share, these applications could use multiprocessing and threading as convenient coding models, but not for scalability. Applications needed to grow faster than improvements in CPU technology would allow.

Breaking Moore's Law

Gordon Moore predicted, back in 1964, that the density of silicon integrated circuit logic would follow the curve: $2^{(\text{year} - 1962)}$ bits per square inch. This approximates to a doubling of processor speed every 18 months, and has proven to be close to reality. It could be argued that Moore's Law is a self-fulfilling prophecy. Less controversial was the assertion that the demand for processing power would soon outgrow the capabilities of a single microprocessor.

By the early 1990s, the race to build affordable, scalable, multiprocessor UNIX systems had begun. However, a major issue that had to be resolved was the monolithic UNIX kernel. Uniprocessor systems require very little protection against concurrent execution in the kernel other than the occasional masking of interrupts. Yet to build scalable multiprocessor systems, it would be necessary to allow not just concurrent, but *parallel* execution within the kernel.

 ©2002 Sun Microsystems, Inc.

Asymmetric Multiprocessors

Some companies addressed this challenge by building asymmetric systems, in which all kernel processing was carried out by a dedicated CPU. Others attempted to protect critical regions of kernel code with mutual exclusion locks, but discovered it was difficult to find a level of locking that was safe and could deliver scalable performance. As shall be seen later, there is a subtle but important distinction between code and data locking.

An *Asymmetric* Symmetric Multiprocessor

When Sun shipped its first multiprocessor system (the SPARCserver® 600MP series), the then-monolithic Solaris 1.0 kernel was made multiprocessor safe by the use of a single lock. This treated the whole kernel as a single critical region — any one CPU could enter the kernel, but others would have to spin, doing no useful work — until the kernel was free. From a hardware point of view, the 600MP was a fully symmetric multiprocessor design. However, from a software perspective, the original single-lock kernel made it effectively asymmetric.

Scalable Multiprocess Applications

Despite these simple beginnings, some applications were able to achieve significant degrees of scalability on multiprocessor hardware. Generally, such applications used the multiprocess, shared-memory model to exploit the underlying parallel nature of the hardware. The secret of their success lay in avoiding contention for the kernel and its resources. Hardware support for atomic test-and-set operations could even be used to implement user-level synchronization via shared memory — without the need to involve the kernel. These techniques would later become important in the design of scalable multithreaded applications.

Smashing the Monolith

Sun addressed the problem of the monolithic kernel with a major rewrite. The goal was to build an all-new, scalable, preemptable, multithreaded kernel. Instead of attempting to identify critical regions in the existing code, the focus turned to protecting data and avoiding contention for it. This principle translates directly to the design of scalable multithreaded applications, as well.

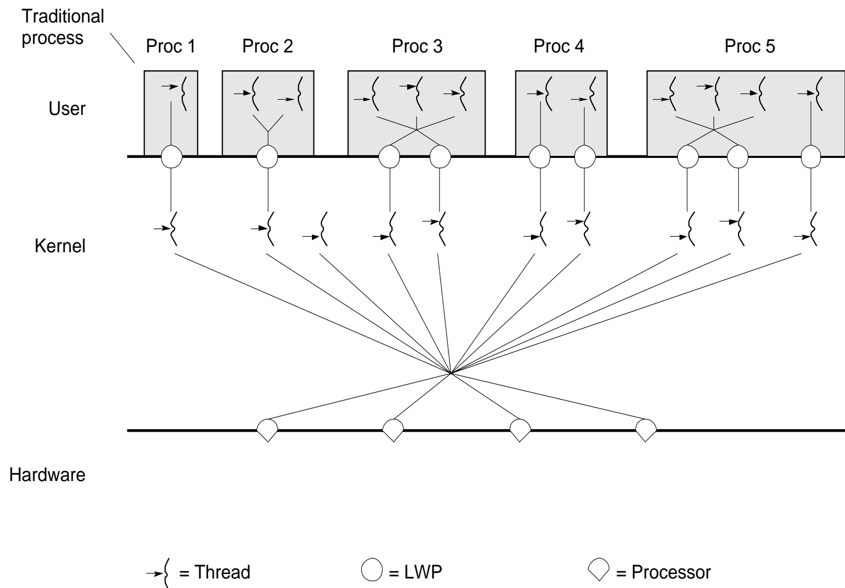
Back to the Drawing Board

The kernel code and its data structures were redesigned with efficient threading in mind. At the same time, many data structures that had previously been statically allocated were reworked for dynamic allocation. Threads were executed strictly on the basis of priority, with special attention being given to issues such as priority inversion. Device interrupts got the thread treatment, too. Whereas the old kernel protected itself by disabling interrupts for extended periods, the new kernel converted interrupts to runnable threads, and reduced the need for interrupt masking.

Beyond Multiprocessing

The goal was never *just* to keep threads safely tucked away inside the kernel to deliver a scalable platform. Applications needed to go beyond multiprocessing, and so the famous “Doctor Ruth” Solaris threads architecture diagram was conceived ().

Figure 2-1: Solaris threads architecture



Space does not permit a full explanation of the diagram (see *Programming with Threads*), but these points should be noted:

- Threads exist in two domains: application threads in the User domain; and kernel threads in the Kernel domain.
- An application thread is an independent sequence of program code execution within a user process.
- The point at which all the lines meet in the Kernel is the kernel's thread scheduler. Kernel threads are the only schedulable entity the kernel knows.
- Applications threads are not necessarily visible to the kernel. Instead, the kernel sees light-weight processes (LWPs) as the only schedulable entities within a process.
- Each LWP is associated with a kernel thread, but not every kernel thread is associated with an LWP. For example, note the third kernel thread (daemon) from the left in the diagram.
- A process can only run in parallel on as many processors as it has LWPs.
- A process can only enter the kernel concurrently as many times as it has LWPs.
- Processes 2 and 3 represent an MxN application threads scheduling model. Here, the point at which lines meet in process is a user-level threads scheduler. This allows a process to contain many more application threads than it does LWPs — the former are multiplexed over the latter.
- Process 4 represents a 1:1 scheduling model. Here, each application thread has its own LWP, and the kernel is used to schedule all application threads.
- Process 5 is a hybrid: it contains some threads scheduled according to an MxN model, and one thread according to a 1:1 model.
- The diagram does not illustrate *processor binding* or *processor sets*, as these had not been implemented at the time the diagram was drawn. In any case, these topics are not covered in detail here (see *Scalable Realtime in the Solaris Operating Environment*).

 ©2002 Sun Microsystems, Inc.

Two-Tiered Advantages

The first advantage of a two-tiered architecture is that application thread synchronization can be done either via the kernel (using system calls), or at the user level (leveraging atomic test-and-set facilities in the multiprocessor hardware). A system call may take many hundreds or even thousands of instructions, but a simple compare-and-swap operation takes just one (although this may involve a considerable number of clock cycles to complete).

The other advantage of a two-tiered implementation is that the cost of context switching in a user-level scheduler is considerably less expensive than with the kernel's scheduler. In the early days, it was expected that user-level scheduling would be orders of magnitude faster than kernel scheduling.

The first advantage has proven to be significant, but the second advantage has not. Building a user-level scheduler that works well in tandem with the kernel's scheduler is a significant challenge. However, the kernel's ability to efficiently schedule many threads has improved.

Opening Software Foundations

Solaris 2.0 software (released in July, 1992) did not actually include the necessary library support for threaded applications, and it did not support multiprocessor hardware. However, its preemptable, multithreaded kernel laid the foundation for a scalable, multithreaded, realtime, multiprocessor environment that would take UNIX into the next millennium.

A *Symmetric* Symmetric Multiprocessor

Solaris 2.1 software (December, 1992) saw the first support for multiprocessor hardware. The 600MP was now a truly symmetric multiprocessor system, and was soon joined by the SPARCstation® 10 workstation. However, the Solaris Operating Environment still did not support threaded applications.

Reliable, Scalable Threads For All

Solaris 2.2 software (May, 1993) included the first supported "libthread." Solaris software-based applications could now be explicitly coded to use threads. The implementation supported both MxN and 1:1 threading models, even in the same process (For example, process 5 in Figure 2-1:, above). The multithreaded kernel also began to show its potential as the SPARCcenter® 2000 server was released — although at this stage, a maximum of just eight (out of a potential twenty) CPUs were supported. Full 20-way support was soon to arrive, with Solaris 2.3 software (November, 1993). By the time Solaris 2.4 software (December, 1994) appeared, the major scalability issues associated with the rewrite had been resolved.

Setting the Standards

One issue facing early adopters of application threading was the lack of a standard set of application programming interfaces (APIs). For UNIX environments, two competing standards were to emerge: Pthreads and UI.

Pthreads or UI Threads?

When "libthread" first appeared in Solaris 2.2 software, it provided a set of APIs that later formed the basis of the UNIX International (UI) threads standard — part of System V Interface Definition, Issue 4.

Meanwhile, the IEEE Portable Operating System Interface (POSIX) draft standard P1003.4a (Pthreads) was in the works, and eventually emerged as POSIX.1c-1995.

Note – Reference is sometimes made to Solaris threads. Often, the implication is that multi-threading in the Solaris Operating Environment involves inevitable lock-in to proprietary technology. The reality, however, is that the Solaris OE provides a choice of two open industry standards for multithreading.

The Choice Is Yours

Solaris 2.5 software (November, 1995) included an implementation of the new Pthread APIs in addition to the older, UI-thread APIs. Both sets of APIs were implemented inside libthread. A filter library, libpthread, was used to expose the Pthread APIs independently of libthread.

A Level Playing Field

Both Pthreads and UI threads utilize the same concepts, although their terminology, function prototypes, and calling conventions differ. For the most part, the Solaris Operating Environment implements both sets of APIs using common code. Sometimes a Pthread function is implemented as a wrapper around the UI function — and vice versa. A few APIs have completely independent implementations. There is no real performance differential between equivalent APIs, except where the style of a particular API might lend itself to a more efficient implementation.

Looking Forward

Over the past ten years, developers have learned to cope with many different threads APIs and implementations. Some have adopted a single standard and looked to compatibility libraries for portability. Others have developed their own thread libraries, implementing them by using whatever APIs are available on the target platform.

Most developers and systems vendors have adopted Pthreads as the API set of choice for multithreaded programming. However, a significant number of developers still prefer UI threads. Sun believes Pthreads hold the most potential for the future, but remains committed to providing UI threads APIs. The similarity between the API sets means that improvements in the implementation of Pthreads will benefit UI threads, as well.

Recent Innovations

Solaris 2.5 software was something of a watershed for threads in the Solaris Operating Environment. The implementation was robust, and provided two sets of standard APIs. Consequently, the number of developers writing threaded applications for the Solaris Operating Environment increased dramatically — just in time to take advantage of Sun's new generation of servers based on the UltraSPARC® processor. Threads were going new places, and it soon became apparent that the implementation would need further tuning. Solaris 2.5.1 software (March, 1996) provided support for multiprocessor UltraSPARC systems, yet included the same threads implementation as the Solaris 2.5 software.

 ©2002 Sun Microsystems, Inc.

Preemption Control

Solaris 2.6 software (August, 1997) addressed a number of issues relating to the visibility of user-level threads to the kernel. One issue was that the kernel had no knowledge of user-level thread synchronization. An application thread can acquire a mutex using a simple, atomic test-and-set instruction, without troubling the kernel. In well-designed, scalable threaded applications, mutexes are held only for very short durations. It is not a good idea to preempt a thread that is holding mutex, since other threads may end up having to wait for the mutex holder to run again.

Solaris 2.6 software introduced the option of a per-LWP data structure in a region of shared memory that is private between the process and the kernel. This provides an efficient mechanism whereby an application thread can hint to the kernel's scheduler whenever preemption would be undesirable.

The preemption control mechanism works as follows:

- An application thread sets the "don't preempt me" flag in the shared data structure corresponding to its LWP and, presumably, acquires a lock.
- If the scheduler wants to preempt the LWP at this point, it will see the flag and defer preemption. To avoid abuse of this privilege, the scheduler sets a "you owe me" flag, to inform the application thread that it will be preempted if it doesn't yield soon.
- The application thread releases the mutex and clears the "don't preempt me" flag. If the "you owe me" flag is set, the application thread must yield in order to be sure of being able to use the privilege again in the future.

See the `schedctl_init(3SCHED)` manpage for further details.

Scheduler Activation

Prior to Solaris 2.6 software, the kernel used a special signal, SIGWAITING, to inform the threads library that all LWPs were blocked in the kernel. This gave the library the opportunity to create another LWP so it could to continue to run other, nonblocking threads. In Solaris 2.6 software, this mechanism was augmented by the preferential use of a "door upcall." Essentially, this involves the kernel being able to call into the user-level thread scheduler to adjust the number of LWPs in the process' pool of LWPs. This door mechanism is more efficient than a signal, but if necessary, Solaris 2.6 software falls back to using the SIGWAITING mechanism.

Adaptive Mutexes

As already stated, one advantage of a two-tier implementation is that application thread synchronization can be carried out using atomic test-and-set instructions, without the need to make costly system calls.

In reality, this is a little more complex than it sounds. The question is, what to do when an atomic test-and-set instruction fails to acquire the desired mutex? One approach is to keep retrying until the mutex is acquired, but this would consume a CPU, making it unavailable to other runnable threads. Another approach is to try again later after a sleep in the kernel. Or better still, ask the kernel to acquire the lock and sleep until it is ready.

As mentioned before, in well-designed threaded applications, mutex locks are held for very short periods of time. In such cases, spinning for a lock only makes sense if the lock holder is running on another CPU. In Solaris 2.6 software, this information became available through the same per-LWP data structures used to facilitate preemption control.

With this mechanism, the only constraint is that the thread holding a lock must be in the same process as the thread waiting for it, since these per-LWP data structures are private between each individual process and the kernel.

Solaris 2.6 software onwards implements adaptive locking for mutexes initialized as:

- PTHREAD_MUTEX_PRIVATE (Pthreads)
- USYNC_THREAD (UI threads)

The following pseudocode illustrates the adaptive mutex lock implementation:

```
if (test-and-set succeeds)
                return /* lock acquired */

spin count = 0
while (lock holder on cpu AND spin count < max spins) {
                if (lock not still held) /* ordinary load */
                    if (test-and-set succeeds)
                        return /* lock acquired */

                increment spin count
}

mutex lock system call /* always acquired the lock */
return /* lock acquired */
```

Note – It is not a good idea to spin using an atomic test-and-set instruction, since this could impact other processors on the same system bus. Thus, an ordinary load instruction is used until there is a good chance of a test-and-set instruction successfully acquiring the lock. This allows any processors spinning on the lock to share the cache line with the lock owner until the owner reclaims the line and releases the lock.

More Mutex Types

Solaris 7 software (October, 1998) added two new types of mutex lock for the Pthreads programmer:

- PTHREAD_MUTEX_ERRORCHECK
- PTHREAD_MUTEX_RECURSIVE

As the pthread_mutex_lock(3THR) manpage explains, the default mutex type of PTHREAD_MUTEX_NORMAL provides no protection against a thread either deadlocking itself or attempting to unlock a lock which is not held. The manpage continues:

> "If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.
>
> If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to 1. Every time a thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches 0, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex that is unlocked, an error will be returned."

 ©2002 Sun Microsystems, Inc.

The use of these new mutex types in freshly designed code is considered a weakness. Data structures should be designed for minimal contention, with locks being held for the shortest possible time. However, these new mutex types have some merit when attempting retrofit threading to old, monolithic applications.

As already hinted, it is possible for two processes to synchronize via mutexes held in shared memory. However, since the first stage to acquiring a mutex is an atomic test-and-set instruction, it is possible for a process to die leaving the lock held. Any other thread waiting in the kernel for this lock would block forever, since the kernel has no visibility of the lock ever being acquired.

For the UI threads programmer, the `USYNC_PROCESS_ROBUST` mutex type was introduced. It returns an error to the waiter if the lock holder dies. Pthreads has no notion of a robust mutex. To address this, the Solaris Operating Environment includes the nonportable (NP) extension `PTHREAD_MUTEX_ROBUST_NP`.

Yet Another Memory Allocator

Memory allocation (`malloc()`, `free()` *et al* in C; `new` and `delete` in C++) is a huge subject. Every programmer has a favorite heap manager. A major reason for the diversity is that it is not possible to implement a good, all-round solution. It is one thing to make `malloc()` thread safe, quite another to make it scalable. At the same time, without threads, the fast and simple `malloc()` could be slowed down by unneeded locks. So, Solaris 7 software saw the introduction of "mtmalloc." This provided a more scalable heap allocator that was plug-compatible with the libc implementation and could be selected at runtime.

Breaking the 32-Bit Barrier

Of course, the most significant innovation in Solaris 7 software was the introduction of 64-bit address spaces. Until then, 32-bit addressing limited each process to just four gigabytes (GB) of virtual memory. For threaded applications the issue was more acute, since thread stacks had to share the four-gigabyte process address space with program text and data. By default, each thread stack has an adjacent "red zone" — an unmapped region of at least one page that will trap stack overflows. Although red zones do not consume physical memory, they do reduce the amount of address space available for other purposes. With 64-bit virtual addressing, space is no longer a constraint. A threaded process could use all 64 UltraSPARC CPUs of a fully configured Starfire™ server, but now it could address nearly all 64 GB of the system's RAM, as well. However, it wasn't until Solaris 8 software (February, 2000) that the kernel could leverage the 64-bit address space for kernel thread stacks. In fact, until Solaris 8 software, a system-wide limit of 512 megabytes (MB) was enforced.

Each kernel thread also has a stack and a red zone. The kernel thread stack size is configurable. In 32-bit kernels on UltraSPARC systems, it defaults to eight kilobytes (KB), although Veritas file system users increase this to 16 KB. In 64-bit kernels, the default kernel thread stack size is 16 KB. For UltraSPARC systems, the red zone is a single 8-KB page in size, but has no physical memory associated with it (red zones are holes designed to catch stack overflows). All kernel thread stacks are allocated in the "kernel pageable segment" known as "segkp", but this is fixed at 512 MB for all 32-bit kernels and the 64-bit kernel of Solaris 7 software. So, with 16-KB stacks (each with an eight-KB red zone) there is a limit of approximately 21,000 stacks (a maximum of 21,000 LWPs system-wide), consuming about 340 MB of physical memory. From Solaris 8 software onwards, the 64-bit kernel allows segkp to be sized from 512 MB to 24 GB, with the default being two gigabtyes (sufficient for more than 87,000 kernel thread stacks/LWPs).

Scalability Challenges

So far, so good. However, as developers started building more complex threaded applications, and a growing number of users began deploying these applications on large, symmetric multiprocessor hardware configurations, the Solaris Operating Environment user threads implementation revealed certain scalability constraints. Implementing MxN thread scheduling correctly and with scalability was a complex challenge.

Threads and Signals Don't Mix

UNIX signals have some things in common with threads. In fact, Process 2 in the Doctor Ruth diagram (Figure 2-1:) could be used to describe a traditional, single-threaded UNIX process with a signal handler. Indeed, the sigaction(2) API could be explained in terms of specifying a thread to be run when some event occurs. However, signal handlers run in the context of the interrupted thread and use its stack.

Signals introduce similar issues of concurrent data access to those seen in threaded applications. For this reason, many library functions are "Async-Signal-Unsafe." Despite their similarities, threads and signals are sufficiently different to make the implementation of both within the same process rather complex. A mix of threads and signals adds a level of complexity for application developers, as well. Many application bugs arise out of writing code that is either MT-Unsafe, Async-Signal-Unsafe, or both.

It is important to draw a distinction between "synchronous" and "asynchronous" signals. A synchronous signal is generated as the result of something the signalled thread has *just* done (for example, SIGSEGV is delivered to a thread that has just violated a memory protection). An asynchronous signal interrupts the signalled thread at some arbitrary point in its execution (for example, SIGALRM from a timer that has expired, or SIGUSR1 from another process or thread).

The implementation of synchronous signals in a multithreaded environment is not difficult. The signal can simply be delivered straight to the LWP on which the thread that triggered the signal is running. However, asynchronous signals are a different matter in an MxN implementation, since the only application thread that has the signal unmasked may not be currently running on an LWP.

The old MxN implementation attempted to solve the asynchronous signal issue by introducing a dedicated LWP, Asynchronous Signal LWP (ASLWP), into the process model. Whenever the kernel wants to deliver an asynchronous signal to a process and ASLWP is present, the kernel will notify ASLWP instead of attempting to deliver the signal directly to an LWP. Once ASLWP sees a suitable application thread running on an LWP, it asks the kernel to redirect the pending signal to that LWP. Not only is this very complex to implement correctly, but it also means that ASLWP can become a bottleneck where there is a high volume of signals.

 ©2002 Sun Microsystems, Inc.

Note – A useful technique for threaded applications is to eliminate asynchronous signal delivery altogether. Instead of installing sigaction(2) signal handlers, dedicated threads can be made to wait for specific signals using sigwait(2) or sigwaitinfo(2). This effectively turns asynchronous signals into synchronous signals, and can provide better scalability.

Managing Concurrency

A major issue for MxN implementations is finding the optimum value of "N" for a population of "M" application threads. In the Solaris Operating Environment, it is the question of how many LWPs to provide within the process. LWPs give a process access to CPU resources and facilitate access to kernel services. A useful LWP is either running on a CPU, or waiting in the kernel for something to happen. An LWP with nothing useful to do needs to be parked in the kernel or destroyed, otherwise it consumes CPU cycles for no benefit. At any point in time, the number of CPUs available to a process, and the number of kernel events it can be waiting for, depends on how many LWPs it has. In theory, any thread that is runnable should be mapped to a runnable LWP, and any thread that must wait for a kernel event should be blocked on an LWP, which itself is blocked in the kernel. Therefore, the scalability of a threaded application depends on managing the number of LWPs.

In the early days when LWPs were considered an expensive resource, it was believed that a process should automatically manage its concurrency by creating or destroying LWPs as conditions dictated. It was also thought advantageous to have a user-level scheduler switching from one thread to another. The trouble is that in all but the simplest "co-routine" scenarios, one thread does not usually stop running and make another runnable.

Automatic concurrency management has not quite delivered the anticipated benefits. Building an MxN implementation that does a good job of managing concurrency across many types of application is hard. Even in those idealistic early days, it was soon acknowledged that an application might need to override the best intentions of the implementation, and so the thr_setconcurrency(3THR) API was born.

Meanwhile, threads developers were discovering that bound threads (those where each application thread has its own LWP) were becoming scalable. So programmers increasingly adopted the worker pool model. At one time, it was thought that thread stacks were the natural place to store application state. But it became apparent that threads delivered greater scalability when they were either running on a CPU, or blocked in the kernel waiting for some event to occur. In such cases, application state could be kept in well-designed data structures, and a pool of worker threads deployed to process this data. The improved scalability was due to better locality of reference caused by allowing both the user and kernel thread stacks to be used more intensively.

_schedlock Considerations

Correctness and performance are often held in tension, but performance without correctness is worthless. In threaded environments, data must be protected in a way that does not impede scalability. Such systems sink or swim on the merits of their locking strategies.

The current Solaris MxN implementation sometimes displays poor scalability due to an over-contended internal lock, _schedlock. As its name implies, it is related to the user-level thread scheduler. When process private application locks become highly contended, _schedlock can become a limiting bottleneck. The usual strategy for fixing such situations is to implement a

better locking strategy, often involving lock “break up.” This approach would probably work for the _schedlock issue. However, when other issues (including signals, concurrency management, current thread coding trends, and advances in kernel thread scalability) are taken into account, an alternate implementation would be a better choice.

An Alternate Implementation

Solaris 8 software includes not one user threads implementation, but two. The new, alternate libthread is 100-percent plug-compatible with the original libthread (including support for UI threads and Pthreads). Dynamic linking meant that it was possible to select a threads implementation at runtime. This made it very easy for Sun, independent software vendors, and end users to test the alternate version, without the need to recompile or relink applications.

Two Libsthread

The old threads implementation resided in /usr/lib:

```
/usr/lib/libthread.so.1                         (32-bit)
/usr/lib/sparcv9/libthread.so.1                 (64-bit)
```

The new implementation was in /usr/lib/lwp:

```
/usr/lib/lwp/libthread.so.1                     (32-bit)
/usr/lib/lwp/sparcv9/libthread.so.1             (64-bit)
```

The Pthreads filter library (libpthread) used whichever libthread was selected, and so remained in /usr/lib:

```
/usr/lib/libpthread.so.1                        (32-bit)
/usr/lib/sparcv9/libpthread.so.1                (64-bit)
```

To select the alternate libthread at runtime, dynamic linker search path could be set as follows:

```
LD_LIBRARY_PATH=/usr/lib/lwp                    (32-bit)
LD_LIBRARY_PATH_64=/usr/lib/lwp/sparcv9         (64-bit)
```

To build a binary that would default to the alternate libthread, specify a runpath at link time:

```
cc -mt ... -R /usr/lib/lwp                      (32-bit)
cc -mt ... -R /usr/lib/lwp/sparcv9              (64-bit)
```

Fuller details can be found in the `threads(3THR)` manpage.

Bound Threads for All

The most important innovation on the alternate libthread was to discard the MxN model and replace it with a simple 1:1 implementation. Every application thread now had its own LWP. Although this may sound drastic, it simply made all application threads behave as “bound” or “system scheduling scope” threads in the old libthread implementation. The new libthread ignored PTHREAD_SCOPE_PROCESS for Pthreads, or created every UI thread implicitly THR_BOUND. (Many threaded application developers had already discovered this with the old libthread.)

Improved Quality

The alternate libthread implementation had almost exactly half the number of lines of code as the standard libthread. Without the user-level thread scheduler, the code was much simpler, and

 ©2002 Sun Microsystems, Inc.

soon proved to be of a higher quality. Most bugs filed against the old implementation were simply not reproducible with the alternate implementation.

Dependable Signals

With no unbound threads to worry about, ASLWP was no longer necessary. Each thread could now store its signal mask directly in its LWP, so the kernel was able to deliver signals directly without needing to know any inside information. The complexities of the ASLWP implementation had been the source of many issues with signal delivery. With the demise of ASLWP, all these issues disappeared.

Better Scalability

The alternate implementation abolished _schedlock and its scalability constraints. The new 1:1 model also meant that concurrency management was a thing of the past, since thr_setconcurrency(3THR) was obsolete. Over time, the alternate libthread has proved to be more scalable than the old MxN implementation.

Adaptive Mutexes Revisited

Despite the advances with adaptive mutexes in the old libthread, it was often found that _schedlock issues eliminated the benefit of knowing whether a lock holder was on CPU or not. Consequently, the alternate libthread could use a simpler mutex implementation. The only optimization was to check that there was more than one CPU before spinning.

Just Plain Simple

The complexities associated with the MxN implementation are gone. Before, just linking with libthread added complexity to a process. Consider the following example:

```
#include <unistd.h>
#include <stdio.h>

int
main()
{
        (void) printf("Hello World!\n");
        (void) pause();
        return (0);
}
```

If this is compiled and linked with the old libthread, utilities like truss(1) and pstack(1) show the following four LWPs:

- The primary thread (as expected)
- ASLWP to handle all signal delivery
- A reaper thread to clean up detached threads
- An LWP waiting for the kernel's door upcall

With the alternate threads library, there is no such side effect associated with linking in the library. A single-threaded process and a multithreaded process with one thread behave the same way. There are no hidden LWPs.

This makes it possible for single-threaded processes to take advantage of some of the thread library's APIs, without themselves needing to be MT-Safe. For example, a single-threaded process

may want to use mutex locks and condition variables to communicate with other single-threaded or multithreaded processes via shared memory. It can do this by using the Pthread APIs available via part of the alternate libpthread implementation, and by initializing the objects PTHREAD _PROCESS_SHARED (USYNC_PROCESS in the UI threads case). Prior to the alternate threads implementation, this would have caused extra LWPs to be created, making the process effectively multithreaded.

Moving On

This paper has surveyed the major developments in Solaris threading over the past decade, showing how the Solaris Operating Environment has moved with the times as application threading methods, developer demands, and hardware platforms have evolved.

The next chapter looks at new features in Solaris 9 software. However, due to the nature of update releases, some of these features are also available for the alternate libthread in Solaris 8 software updates, including:

- Improved performance for thread-specific data (Solaris 8 10/01 Release)
- Improved adaptive locking (Solaris 8 02/02 Release)
- User-level sleep queues (Solaris 8 02/02 Release)

Chapter 3

What's New in the Solaris 9 Operating Environment?

The improved quality and scalability of the alternate thread library implementation supported by Solaris 8 software means that Sun can now focus on taking threads forward, rather than on maintaining an old implementation.

MxN Implementation Retired

The Solaris 9 Operating Environment does not include an alternate threads implementation. The old MxN implementation has been gracefully retired and replaced with an enhanced version of the 1:1 implementation. The /usr/lib/lwp directories which held the alternate library in Solaris 8 software have been preserved, but these now simply link back to /usr/lib.

Extensive testing indicates that this decision should cause no issues for existing applications. Most applications are expected to benefit from this move and binary compatibility is preserved.

Adaptive Mutexes Revisited

The Solaris 9 software libthread implements the optimization that was introduced into the old libthread in Solaris 2.6 software — a thread waiting for a process private mutex will spin only if the lock holder is currently running on a CPU. The process shared case remains unchanged. If there is more than one CPU, the thread will always spin before blocking.

Adaptive Mutex Unlock

A further optimization is the provision of adaptive mutex unlocking. When a thread is about to release the mutex it is holding, it looks to see if there are any waiters spinning. If so, it releases the lock and spins itself for a short while to see if the lock is acquired by another thread. This boosts performance because the thread releasing the lock no longer needs to enter the kernel to wake up a waiter.

Uncontended Mutex Tuning

An important maxim in software engineering is that "correctness is a constraint, while performance is a goal." The simplicity of the new threads implementation makes it easier to deliver correctness, and allows more time to be devoted to performance tuning. Many libraries must be MT-Safe to perform well in single-threaded cases. Often, MT-Safe means the addition of locking, yet these locks will never be contended in single-threaded applications. Considerable effort has gone into improving the uncontended case. Well-designed threaded applications will also benefit, since well-designed threaded applications avoid contention.

User-Level Sleep Queues

Perhaps the most significant new feature in the Solaris 9 libthread is the addition of user-level sleep queues. In the past, if a user-level synchronization needed to block, a call was made to an object-specific system call. For example, if a thread were unable to acquire a mutex after spinning, it would call _lwp_mutex_lock(2). Or, if a thread was blocking on a condition variable, it might call _lwp_cond_timedwait(2).

The drawback with these object-specific system calls is that the kernel must guarantee that the correct semantics are applied to each object. However, for intra-process synchronization (PTHREAD_PROCESS_PRIVATE or USYNC_THREAD) the kernel only needs to provide an efficient LWP suspend-and-resume mechanism, since the correct semantics could be maintained at the user level. This is what the Solaris 9 Operating Environment does, using these new private system calls:

- _lwp_park(2)
- _lwp_unpark(2)
- _lwp_unpark_all(2) (used for condition variable broadcasts)

These system calls are significantly faster than their object-specific predecessors (which, of course, still must be used when PTHREAD_PROCESS_SHARED or USYNC_PROCESS synchronization is not resolved at the user level).

The user-level sleep queues require some internal mutex protection. This is achieved with simple spin locks using the preemption control mechanism which was described as a recent innovation in Chapter 2.

The queues respect thread priority, but implement a near last-in-first-out (LIFO) queueing policy for threads of the same priority. That is, most of the time a thread will be placed at the head of the queue, but in order to prevent starvation, it will occasionally be placed at the rear of the queue.

 ©2002 Sun Microsystems, Inc.

The Issue of Fairness

The near-LIFO nature of the new user-level sleep queues raises the issue of fairness. LIFO queueing is considered undesirable, due to the inevitable increase in variance it introduces in response times. However, none of the threads API standards mandate a particular queueing policy. Indeed, the documentation sometimes warns the programmer not to depend on an implementation-specific policy.

It is important to remember that the issue is very low-level synchronization. Developers should be aware of the actual queueing requirements of an application, and design data structures and algorithms to meet them. In general, highly contended data structures are not well designed.

LIFO queueing is the natural extension of optimizations such as adaptive mutex locking and unlocking. An adaptive mutex is inherently unfair. However, this unfairness can deliver a performance win, because the running thread is already on CPU and has warmed the cache.

These low-level enhancements can deliver real benefits to well-written applications. The near-LIFO policy provides a safety net for applications that have been presumptuously coded for first-in-first-out (FIFO) queueing, and might otherwise suffer from thread starvation.

Faster Thread-Specific Data

As more programmers have attempted to retrofit threads to existing monolithic applications, the need for fast thread-specific data (TSD) has become a major issue. The Solaris Operating Environment includes basic support for both Pthread and UI thread TSD APIs:

- pthread_key_create(3THR) or thr_keycreate(3THR)
- pthread_setspecific(3THR) or thr_setspecific(3THR)
- pthread_getspecific(3THR) or thr_getspecific(3THR)

These allow for per-process global variables to be replaced with per-thread global storage. The very mention of the word global was once sufficient to raise scorn in some circles. Consequently, although the implementation of TSD was correct, it was not optimized for performance. Today, the need for efficient thread specific data is undeniable, and the Solaris 9 threads library has been optimized accordingly.

Note – This is an instance in which a Pthread API is implemented more efficiently than its UI thread equivalent. Since pthread_getspecific(3THR) returns the address of the data or NULL, it can be optimized further than thr_getspecific(3THR). The latter returns zero or an error number, passing the address via a pointer-to-pointer argument.

Yet Another Memory Allocator Enhancement

The mtmalloc, introduced in Solaris 7 software, has been further tuned and improved for the Solaris 9 OE.

Per-Process Timers

Prior to Solaris 9 software, the `setitimer(2)` manpage admits to implementing per-LWP timers. The Solaris 8 manpage commits to fixing this bug in the next release, which was done, and timers are implemented per process.

Thread Local Storage

The improved libthread implementation gives Sun a solid foundation for future enhancements to multithreading in the Solaris Operating Environment. For example, Solaris 9 software already includes linker and thread library support for thread local storage (TLS). Unlike thread-specific data (TSD), which requires explicit function calls to implement per-thread global data, TLS could be implemented using variable qualifiers at compile time.

Conclusion

Sun was an early supporter of multithreaded applications, and has spent the last decade refining the Solaris Operating Environment threads implementation. The success of the Sun™ server product family is partly due to this work. Today, many applications benefit from Solaris threads, and with the Solaris 9 Operating Environment, performance has never been better.

Appendix A

Bibliography

1. *Scalable Realtime in the Solaris Operating Environment - A Technical Whitepaper,* Sun Microsystems, 2001
2. *Solaris 8 Online Manual*, Sun Microsystems, 2000
3. Butenhof, *Programming with POSIX Threads*, Addison Wesley, 1997
4. Gallmeister, *POSIX.4 - Programming for the Real World*, O'Reilly, 1995
5. Kleiman, Shah, and Smaalders, *Programming with Threads*, Prentice Hall, 1996
6. Lewis and Berg, *Multithreaded Programming with Pthreads*, Prentice Hall, 1998
7. Mauro and McDougall, *SOLARIS Internals*, Prentice Hall, 2001
8. Nichols, Buttlar, and Farrell, *Pthreads Programming*, O'Reilly, 1996
9. Robbins and Robbins, *Practical UNIX Programming*, Prentice Hall, 1996

Appendix B

Life Before the Solaris 9 Operating Environment

It is hoped that the preceding material will be sufficient to entice even the most tardy of technology adopters to adopt Solaris 9. However, some organizations require time to roll out new configurations and may have to live with the foibles and shortcomings of older releases in the interim. The following information is offered on the understanding that it is not a justification for delaying the adoption of the Solaris 9 Operating Environment.

Starting With Solaris 8

Most of the recent enhancements to Solaris threads are available in Solaris 8. Simply using the alternate threads implementation and should help threaded applications become more robust and scalable. This is also an excellent way to prove application readiness for Solaris 9.

Thread Synchronization

When a programmer is using the alternate implementation, has process private (PTHREAD_PROCESS_PRIVATE or USYNC_THREAD) synchronization objects, and truss(1) is revealing system calls to: _lwp_mutex_lock(2), _lwp_cond_wait(2), _lwp_cond_timedwait(2), or _lwp_sema_wait(2), the application could benefit from the improved adaptive mutexes and user-level sleep queues available in Solaris 8 Update 7.

 ©2002 Sun Microsystems, Inc.

Thread-Specific Data

If an application makes heavy use of thread-specific data, consider Solaris 8 Update 6, which has a much improved implementation. This applies only to the alternate libthread.

Prior to Solaris 8

Short of keeping up to date with patches and kernel updates, there is little end-users can actively do to improve thread performance and reliability. However, developers may find the following useful.

Are You Hitting _schedlock?

There is no definitive way to tell diagnose a _schedlock issue, but if truss(1) shows heavy _lwp_mutex_lock(2) traffic at one address in libthread's uninitialized data segment, _schedlock is a fairly likely culprit. The programmer can determine if this is the case by comparing the address indicated by truss(1) with the output of pmap(1) for the process in question.

Note – If pmap-F is used, pmap(1) can be run on a process while tracing it with truss(1).

If a _schedlock issue is suspected, and the application uses process private (PTHREAD_PROCESS_PRIVATE or USYNC_THREAD) mutex locks, consider marking the mutexes process shared (PTHREAD_PROCESS_SHARED or USYNC_PROCESS). This has the effect of bypassing _schedlock altogether and also turns off the adaptivity of the mutexes.

Caution – This workaround involves lying to the threads library about the scope of the locks. Remember to undo this change before using the alternate libthread in Solaris 8 or the new libthread in Solaris 9.

For _schedlock issues with C++ code, or code which makes heavy use of malloc(3C), it is possible that memory allocation contention is the main problem. From Solaris 7 onwards, try using /usr/lib/libmtmalloc.so.1 or a third-party memory allocator such as SmartHeap or Hoard. Usually, this can be achieved without having to recompile or relink the application — it is often possible to make the switch using LD_LIBRARY_PATH or LD_PRELOAD.

Concurrency Issues

Experimenting with thr_setconcurrency(3THR) is not recommended. Instead, use bound threads (PTHREAD_SCOPE_SYSTEM or THR_BOUND).

Signal Reliability

Use bound threads (PTHREAD_SCOPE_SYSTEM or THR_BOUND) to ensure that a suitable LWP can always be found for signal delivery. See also the note about avoiding sigaction(2) altogether in *Threads And Signals Don't Mix*.

Appendix C

Ping-Pong Performance

The Ping-Pong benchmark was devised to test the quality of Solaris threads library implementations. In theory, the tests are slightly biased in favour of good implementations of the MxN model. Implementations able to resolve contention and perform context switching efficiently should excel.

The Basic Idea

The test at the center of Ping-Pong consists of a pair of threads in lock-step synchronization. For each iteration, each thread is alternately blocked and unblocked by the other. Only four mutex locks are required for synchronization. In theory, an MxN should have a distinct advantage, since there is very little time when both threads are runnable. One would expect user-level thread scheduling over a single LWP to produce good results in this case.

A Simple Game

The game continues until a specified number of iterations has been completed by each thread. Both the number of lock/unlock operations and the number of context switches should be twice the number of iterations.

A Complex Tournament

To make things even more interesting, Ping-Pong supports the playing of multiple concurrent games within a single process. Scalability can thus be investigated with respect to both the number of threads and the number of CPUs deployed.

Implementation

Ping-Pong consists of 340 lines of C implemented using the Pthreads APIs. A full listing is provided in Appendix D. Ping-Pong is fully configurable at runtime. The following parameters can be specified:

- The number of concurrent games (the default is one)
- The number of iterations per games (the default is one million)
- The thread scheduling scope (the default is PTHREAD_SCOPE_PROCESS)
- The mutex visibility mode (the default is PTHREAD_PROCESS_PRIVATE)
- An optional sleep per iteration (to avoid swamping CPU resources)
- An optional thread stack size (to allow thousands of threads in 32-bit mode)
- An optional thread concurrency hint (for the MxN model)

Baseline Measurement

The baseline measurement is of the new implementation on a 16-processor, 400-MHz Starfire system domain running the Solaris 9 Operating Environment. The default parameters are used, with verbose output selected.

```
weaver$ ptime pp32 -v

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 1
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 0
stacksize       (-S) = 0

2 threads initialised in 3ms
1 games completed in 6798ms

real        6.818
user       12.613
sys         0.296
weaver$
```

Analysis: The 6798-ms elapsed time corresponds to ~7 s per iteration. That is, as one thread is unblocked by the other, the “context switch” time must be less than four s. Of course, there is no real-level context switch. Each thread simply spins in the adaptive mutex code until it is unblocked by its partner. Notice the elapsed time (”real”) is roughly half the total consumed user CPU time.

Small-Scale Comparison

This section investigates the difference in performance between old and new implementations for the twin-thread case on multiprocessor hardware using a twin-CPU processor set.

Test 1: Old libthread, default settings

```
weaver$ ptime pp32 -v

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 1
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 0
stacksize       (-S) = 0

2 threads initialised in 0ms
1 games completed in 13881ms

real       13.903
user       13.888
sys         0.013
weaver$
```

Analysis: The MxN scheduler has chosen to deploy just one LWP. On face value, this is quite a smart move. The benefit is that all contention is now being resolved without involving the kernel. However, the elapsed time is more than twice that of the baseline achieved with the new implementation, and almost eight percent more CPU resource is consumed. This result reveals that a true user-level context switch can still be quite costly, relative to spinning in an adaptive mutex.

Test 2: Old libthread, increased concurrency

```
weaver$ ptime pp32 -v -c 2

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 1
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 2
stacksize       (-S) = 0

2 threads initialised in 0ms
1 games completed in 90922ms

real     1:30.986
user       55.879
sys      1:04.417
weaver$
```

 ©2002 Sun Microsystems, Inc.

Analysis: It is clear that more than one CPU was used, but to no avail. For some reason the old implementation's adaptive mutex optimization is not kicking in. This begins to demonstrate the cost of synchronization via the kernel.

Kernel Context Switch Comparison

The simplest way to force the new implementation to synchronize via the kernel is to restrict it to one CPU. In fact, this will totally eliminate the benefit of any mutex spinning in either implementation. With a single CPU, it is possible to compare the efficiency of the old _lwp_mutex_lock() and new _lwp_park() mechanisms.

Test 3: New libthread, single CPU

```
weaver$ ptime pp32 -v

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 1
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 0
stacksize       (-S) = 0

2 threads initialised in 3ms
1 games completed in 43779ms

real       43.799
user       21.383
sys        18.693
weaver$
```

Analysis: Although this is 6.5 times slower than the baseline, it is only three times slower than the old implementation in Test 1 (which included true user-level context switching). It is therefore hard to support the idea that user-level context switching is orders of magnitude faster than going via the kernel. It is also important to remember that in scalable threaded applications, there is a great deal of processing between thread synchronization points.

Test 4: Old libthread, bound threads, single CPU

```
weaver$ ptime pp32 -v -s system

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 1
sleepms         (-z) = 0
pthread_scope   (-s) = system
pthread_process (-p) = private
concurrency     (-c) = 0
stacksize       (-S) = 0

2 threads initialised in 1ms
1 games completed in 59231ms

real       59.250
user       27.283
sys        28.949
weaver$
```

Analysis: Again, the new implementation is faster than the old, but it is interesting to compare the "sys" CPU time for this and the previous test. The old _lwp_mutex_lock() mechanism consumes 55 percent more CPU than the new _lwp_park() mechanism. Kernel synchronization costs have been significantly improved.

Multiprocessor Scalability Comparison

The next series of tests are designed to see how well CPU resources can be utilized to deliver scalable performance. These tests attempt to use an eight-CPU processor set for maximum scalability.

Test 5: New libthread, four games

```
weaver$ ptime pp32 -v -n 4

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 4
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 0
stacksize       (-S) = 0

8 threads initialised in 4ms
4 games completed in 6901ms

real        6.925
user       51.109
sys         0.322
weaver$
```

Analysis: As expected, the new implementation does very well. It manages to use all the CPUs, and continues to leverage adaptive mutex synchronization.

Test 6: Old libthread, four games, concurrency hint

```
weaver$ ptime pp32 -v -n 4 -c 8

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 4
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 8
stacksize       (-S) = 0

8 threads initialised in 3ms
4 games completed in 350930ms

real     5:50.976
user     4:24.881
sys     18:46.027
weaver$
```

Analysis: This result underlines the concurrency and _schedlock issues of the old implementation. It would, of course, have been better to let the library do its own thing — use just one LWP — because at least this would have resulted in no more than 100 seconds total elapsed time. One might guess that four LWPs would be optimum — but unfortunately, the MxN scheduler is not smart enough to multiplex one pair of threads per LWP.

Test 7: Same again, but with shared mutexes

```
weaver$ ptime pp32 -v -n 4 -c 8 -p shared

PING-PONG CONFIGURATION:

target          (-i) = 1000000
ntables         (-n) = 4
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = shared
concurrency     (-c) = 8
stacksize       (-S) = 0

8 threads initialised in 3ms
4 games completed in 75792ms

real     1:15.836
user     1:28.432
sys      4:41.162
weaver$
```

Analysis: While this result is not nearly as good as the new implementation is capable of attaining, it demonstrates one strategy for attacking the _schedlock issue. Of course, this is at the cost of lying to the implementation, and may have a negative consequences for better implementations, as previously pointed out.

So, You Want Thousands of Threads?

Ping-Pong is able to deploy a very large number of threads. The following examples compare the old and new implementations when playing 5,000 short games of 100 iterations each, equalling 10,000 threads. This makes it necessary to specify smaller thread stacks (32 KB, in this case).

Test 8: New libthread

```
weaver$ ptime pp32 -v -n 5000 -i 100 -S 32768

PING-PONG CONFIGURATION:

target          (-i) = 100
ntables         (-n) = 5000
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 0
stacksize       (-S) = 32768

10000 threads initialised in 15677ms
5000 games completed in 12308ms

real       28.362
user       19.457
sys      1:32.072
weaver$
```

Analysis: It is clear that thread creation is a significant overhead, but creating 10,000 LWPs in under 16 seconds is nothing to be ashamed of. The kernel's thread scheduler copes reasonably well with 10,000 runnable threads.

Test 9: Old libthread at its best?

```
weaver$ ptime pp32 -v -n 5000 -i 100 -S 32768

PING-PONG CONFIGURATION:

target          (-i) = 100
ntables         (-n) = 5000
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = private
concurrency     (-c) = 0
stacksize       (-S) = 32768

10000 threads initialised in 1973ms
5000 games completed in 43140ms

real       45.134
user       43.955
sys         1.174
weaver$
```

 ©2002 Sun Microsystems, Inc.

Analysis: By rights, this should be where an MxN implementation wins hands down. But again, there is only one LWP. It is hardly surprising that thread creation is so quick. However, this is unacceptable, since it would provide zero scalability for a real workload on multiprocessor hardware.

Test 10: Old libthread at its best

```
weaver$ ptime pp32 -n -n 5000 -i 100 -S 32768 -c 8 -p shared

PING-PONG CONFIGURATION:

target          (-i) = 100
ntables         (-n) = 5000
sleepms         (-z) = 0
pthread_scope   (-s) = process
pthread_process (-p) = shared
concurrency     (-c) = 8
stacksize       (-S) = 32768

10000 threads initialised in 25180ms
5000 games completed in 10080ms


real       35.649
user       13.222
sys      1:27.149
weaver$
```

Analysis: Notice the thread creation time. This is because the use of shared mutexes results in the implementation needing to dump each thread in the kernel. Thus, despite the concurrency hint of eight, the old threads library ends up creating thousands of LWPs. This is the reason shared mutexes were used was to avoid the _schedlock issue.

What Does Ping-Pong Teach Us?

The Ping-Pong benchmark underlines the benefits of the new implementation of libthread, and accentuates the issues with the old libthread. It also demonstrates useful strategies for working around the limitations of the old implementation when the new implementation is not available.

Ping-Pong has many runtime modes. The experiments shown here represent a small subset of potentially useful experiments. Much more useful data could be generated. Remember, Ping-Pong is an extreme case. Many real world applications will not see the same scale of variation in performance. And Ping-Pong does not demonstrate the other benefits (such as improved signal handling) of the new implementation.

Appendix D

Ping Pong Source

```
/*
 * pp.c: ping-pong threads benchmark
 *
 * cc -mt -xO4 pp.c -o pp32 -lpthread -lrt
 * cc -mt -xarch=v9 -xO4 pp.c -o pp64 -lpthread -lrt
 */

#include <pthread.h>
#include <thread.h>
#include <stdlib.h>
#include <strings.h>
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <sys/mman.h>

/* a ping-pong player */
typedef struct {
        int                     table;
        int                     player;
        int                     count;
        pthread_mutex_t         blocks[2];
        pthread_t               thread;
} player_t;

/* a ping-pong table on which a rally is played */
typedef struct {
        int                     target;
        int                     sleepms;
```

 ©2002 Sun Microsystems, Inc.

```
        player_t                players[2];
        char                   pad[40]; /* avoids false cache
sharing */
} table_t;

/* a barrier used to pthread_processronise and time players */
typedef struct {
        pthread_mutex_t         mx;
        pthread_cond_t          cv;
        int                     target;
        int                     count;
} barrier_t;

/* global arena of ping-pong tables */
static table_t                  *tables;

/* global lock used to create a bottleneck */
static pthread_mutex_t          bottleneck;

/* player pthread_processonisation */
static barrier_t               setup_barrier; /* all players
ready */
static barrier_t                begin_barrier; /* all games
begin */
static barrier_t                end_barrier;   /* all games
ended */

/* global pthread attributes - must call attr_init() before
using! */
static pthread_mutexattr_t      mxattr;
static pthread_condattr_t       cvattr;
static pthread_attr_t           ptattr;

/* forward references */
static void *player(void *arg);
static void setup_tables(int n, int target, int sleepms);
static void attr_init(int pthread_process, int sched, long
stacksize);
static void barrier_init(barrier_t *b, int target);
static void barrier_wait(barrier_t *b);

/* for getopt(3C) */
extern char                     *optarg;
extern int                      ptind;

/* verbose output flag */
static int                      verbose = 0;

int
main(int argc, char *argv[])
{
        int                     c;
        hrtime_t                t0;
        int                     ntables = 1;
        int                     target = 1000000;
        int                     sleepms = 0;
        int                     concurrency = 0;
        int                     pthread_scope =
PTHREAD_SCOPE_PROCESS;
        int                     pthread_process =
```

```
PTHREAD_PROCESS_PRIVATE;
        long                    stacksize = 0L;
        int                     errflg = 0;

        while ((c = getopt(argc, argv, "?vi:n:z:p:s:c:S:")) !=
EOF)
        switch (c) {
        case '?':
                errflg++;
                continue;
        case 'v':
                verbose++;
                continue;
        case 'i':
                target = atoi(optarg);
                continue;
        case 'n':
                ntables = atoi(optarg);
                continue;
        case 'z':
                sleepms = atoi(optarg);
                continue;
        case 'p':
                if (strcmp(optarg, "shared") == 0) {
                     pthread_process = PTHREAD_PROCESS_SHARED;
                } else if (strcmp(optarg, "private") == 0) {
                      pthread_scope = PTHREAD_PROCESS_PRIVATE;
                } else {
                        errflg++;
                }
                continue;
        case 's':
                if (strcmp(optarg, "system") == 0) {
                        pthread_scope = PTHREAD_SCOPE_SYSTEM;
                } else if (strcmp(optarg, "process") == 0) {
                       pthread_scope = PTHREAD_SCOPE_PROCESS;
                } else {
                        errflg++;
                }
                continue;
        case 'c':
                concurrency = atoi(optarg);
                continue;
        case 'S':
                stacksize = atol(optarg);
                continue;
        default:
                errflg++;
        }

        if (errflg > 0) {
                (void) printf("usage: pp [-v] [-i <target>] [-
n <ntables>]"
                     " [-z <sleepms>]\n"
                     "[-p private|shared] [-s
process|system]\n"
                     "[-c <concurrency>] [-S <stacksize>]\n");
                exit(1);
        }
```

```
        if (verbose > 0) {
                (void) printf(
                    "\nPING-PONG CONFIGURATION:\n\n"
                    "target          (-i) = %d\n"
                    "ntables         (-n) = %d\n"
                    "sleepms         (-z) = %d\n"
                    "pthread_scope   (-s) = %s\n"
                    "pthread_process (-p) = %s\n"
                    "concurrency     (-c) = %d\n"
                    "stacksize       (-S) = %ld\n\n",
                    target,
                    ntables,
                    sleepms,
                   (pthread_scope == PTHREAD_SCOPE_PROCESS) ?
                        "process" : "system",
                (pthread_process == PTHREAD_PROCESS_PRIVATE) ?
                        "private" : "shared",
                    concurrency,
                    stacksize);
        }

        /* best to do this first! */
        attr_init(pthread_process, pthread_scope, stacksize);

        /* initialise bottleneck */
        (void) pthread_mutex_init(&bottleneck, &mxattr);

        /* initialise pthread_processronisation and timing
points */
        barrier_init(&setup_barrier, (2 * ntables) + 1);
        barrier_init(&begin_barrier, (2 * ntables) + 1);
        barrier_init(&end_barrier, (2 * ntables) + 1);

        /* should not be needed - sigh! */
        if (concurrency > 0) {
                (void) thr_setconcurrency(concurrency);
        }

        /* initialise all games */
        t0 = gethrtime();
        setup_tables(ntables, target, sleepms);

        /* wait for all players to be ready */
        barrier_wait(&setup_barrier);

        if (verbose) {
                (void) printf("%d threads initialised in
%lldms\n",
                 ntables * 2, (gethrtime() - t0) / 1000000LL);
        }

        /* start all games */
        t0 = gethrtime();
        barrier_wait(&begin_barrier);

        /* wait for all games to complete */
        barrier_wait(&end_barrier);

        if (verbose) {
              (void) printf("%d games completed in %lldms\n",
```

```
ntables,
                    (gethrtime() - t0) / 1000000LL);
        }

        return (0);
}

/*
 * build and populate the tables
 */
static void
setup_tables(int n, int target, int sleepms)
{
        int                     i, j;
        int                     res;

        tables = (void *) mmap(NULL, n * sizeof (table_t),
            PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANON, -
1, 0L);

        if (tables == (table_t *) (-1)) {
                exit(1);
        }

        for (i = 0; i < n; i++) {
                tables[i].target = target;
                tables[i].sleepms = sleepms;
                for (j = 0; j < 2; j++) {
                        tables[i].players[j].table = i;
                        tables[i].players[j].player = j;
                        tables[i].players[j].count = 0;
                        (void) pthread_mutex_init(
                          &(tables[i].players[j].blocks[0]),
&mxattr);
                        (void) pthread_mutex_init(
                          &(tables[i].players[j].blocks[1]),
&mxattr);
                        res =
pthread_create(&(tables[i].players[j].thread),
                            &ptattr, player,
&(tables[i].players[j]));
                        if (res != 0) {
                                (void) printf("pthread_create
= %d!\n", res);
                                exit(1);
                        }
                }
        }
}

/*
 * a ping-pong player
 */
static void *
player(void *arg)
{
        player_t                *us, *them;
        table_t                 *table;
        struct timespec         ts;
```

 ©2002 Sun Microsystems, Inc.

```
        us = (player_t *) arg;
        table = &(tables[us->table]);
        them = &(table->players[(us->player + 1) % 2]);

        barrier_wait(&setup_barrier);

        /* player 0 always serves */
        if (us->player == 0) {
                (void) pthread_mutex_lock(&(them->blocks[0]));
                (void) pthread_mutex_lock(&(them->blocks[1]));
                barrier_wait(&begin_barrier);

                /* serve! */
              (void) pthread_mutex_unlock(&(them->blocks[0]));
        } else {
                (void) pthread_mutex_lock(&(them->blocks[0]));
                barrier_wait(&begin_barrier);
        }

        while (us->count < table->target) {
                /* wait to be unblocked */
             (void) pthread_mutex_lock(&(us->blocks[us->count
% 2]));

                /* block their next + 1 move */
                (void) pthread_mutex_lock(
               &(them->blocks[(us->count + us->player) % 2]));

                /* let them block us again */
                (void) pthread_mutex_unlock(&(us->blocks[us-
>count % 2]));

                /* unblock their next move */
                (void) pthread_mutex_unlock(
                    &(them->blocks[(us->count + us->player +
1) % 2]));

                us->count++;

                if (table->sleepms == -1) {
                       (void) pthread_mutex_lock(&bottleneck);
                     (void) pthread_mutex_unlock(&bottleneck);
                } else if (table->sleepms > 0) {
                        ts.tv_sec = table->sleepms / 1000;
                       ts.tv_nsec = (table->sleepms % 1000) *
1000000;
                        (void) nanosleep(&ts, NULL);
                }
        }

        barrier_wait(&end_barrier);

        return (NULL);
}

/*
 * simple, non-spinning barrier wait mechinism
 */
static void
barrier_wait(barrier_t *b)
```

```
{
        (void) pthread_mutex_lock(&b->mx);
        b->count++;
        if (b->count >= b->target) {
                (void) pthread_mutex_unlock(&b->mx);
                (void) pthread_cond_broadcast(&b->cv);
                return;
        }
        while (b->count < b->target) {
                (void) pthread_cond_wait(&b->cv, &b->mx);
        }
        (void) pthread_mutex_unlock(&b->mx);
}

/*
 * initialise a barrier (ok to reinitialise object if no
waiters)
 */
static void
barrier_init(barrier_t *b, int target)
{
        (void) pthread_mutex_init(&b->mx, &mxattr);
        (void) pthread_cond_init(&b->cv, &cvattr);
        b->target = target;
        b->count = 0;
}

/*
 * initialise the global pthread attributes
 */
static void
attr_init(int pthread_process, int pthread_scope, long
stacksize)
{
        (void) pthread_mutexattr_init(&mxattr);
        (void) pthread_mutexattr_setpshared(&mxattr,
pthread_process);
        (void) pthread_condattr_init(&cvattr);
        (void) pthread_condattr_setpshared(&cvattr,
pthread_process);
        (void) pthread_attr_init(&ptattr);
        (void) pthread_attr_setscope(&ptattr, pthread_scope);
        if (stacksize > 0) {
                (void) pthread_attr_setstacksize(&ptattr,
stacksize);
        }
}
```

Copyright 2002 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, Solaris, Java, Starfire, and Sun Fire are service marks, trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

SPECjbb2000 is benchmark from the Standard Performance Evaluation Corporation (SPEC). Competitive claims reflect results published on www.spec.org as of January 28, 2002. For the latest SPECjbb2000 results visit http://www.spec.org/osg/jbb2000.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

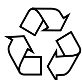
Copyright 2002 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, Californie 94303 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l’utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l’autorisation préalable et écrite de Sun et de ses bailleurs de licence, s’il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l’Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d’autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, Solaris, Java, Starfire, et Sun Fire sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d’autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d’autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

CETTE PUBLICATION EST FOURNIE “EN L’ETAT” ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N’EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L’APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU’ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S’APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.



Please
Recycle

Adobe PostScript

Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, CA 94303-4900 USA **Phone** 800 786-7638 or +1 512 434-1577 **Web** sun.com

We make the net work.

Sun Worldwide Sales Offices: Africa (North, West and Central) +33-13-067-4680, Argentina +5411-4317-5600, Australia +61-2-9844-5000, Austria +43-1-60563-0, Belgium +32-2-704-8000, Brazil +55-11-5187-2100, Canada +905-477-6745, Chile +56-2-3724500, Colombia +571-629-2323, Commonwealth of Independent States +7-502-935-8411, Czech Republic +420-2-3300-9311, Denmark +45 4556 5000, Egypt +202-570-9442, Estonia +372-6-308-900, Finland +358-9-525-561, France +33-134-03-00-00, Germany +49-89-46008-0, Greece +30-1-618-8111, Hungary +36-1-489-8900, Iceland +354-563-3010, India–Bangalore +91-80-2298989/2295454; New Delhi +91-11-6106000; Mumbai +91-22-697-8111, Ireland +353-1-8055-666, Israel +972-9-9710500, Italy +39-02-641511, Japan +81-3-5717-5000, Kazakhstan +7-3272-466774, Korea +822-2193-5114, Latvia +371-750-3700, Lithuania +370-729-8468, Luxembourg +352-49 11 33 1, Malaysia +603-21161888, Mexico +52-5-258-6100, The Netherlands +00-31-33-45-15-000, New Zealand–Auckland +64-9-976-6800; Wellington +64-4-462-0780, Norway +47 23 36 96 00, People's Republic of China–Beijing +86-10-6803-5588; Chengdu +86-28-619-9333; Guangzhou +86-20-8755-5900; Shanghai +86-21-6466-1228; Hong Kong +852-2202-6688, Poland +48-22-8747800, Portugal +351-21-4134000, Russia +7-502-935-8411, Singapore +65-6438-1888, Slovak Republic +421-2-4342-94-85, South Africa +27 11 256-6300, Spain +34-91-596-9900, Sweden +46-8-631-10-00, Switzerland–German 41-1-908-90-00; French 41-22-999-0444, Taiwan +886-2-8732-9933, Thailand +662-344-6888, Turkey +90-212-335-22-00, United Arab Emirates +9714-3366333, United Kingdom +44-1-276-20444, United States +1-800-555-9SUN or +1-650-960-1300, Venezuela +58-2-905-3800

5/02 FE1811-0