Multiprocessing and Multithreading 10

Introduction

This chapter describes concurrent programming concepts and practices that are suitable to Solaris threads. A multi-tasked application that communicates large amounts of data between its processes can benefit from multithreading. Much of the communication between processes that requires extensive use of system resources is done implicitly in a multithreaded application through memory that is global to all of the threads.

Computers with more than one processor provide multiple simultaneous points of execution. Multiple threads are an efficient way for application developers to utilize the parallelism of the hardware.

A good example of this is a process that does matrix multiplication. A thread can be created for each available processor. The threads each repeatedly compute a unique element of the result matrix by repeatedly doing the appropriate vector multiplication.

Terminology

The following terms are used in discussions of multiprocessing and multithreading:

Multiprocessing (MP) is the ability of the operating system to run on two or more CPUs. In modern CPU designs using shared main memory, all the CPUs share main memory and a common image of the operating system kernel.

A *thread of control* or *thread* is a sequence of instructions executed by a program. In the traditional UNIX model, a process is a thread. Threads execute independently of each other and share a common address space.

A traditional UNIX process has a single thread of control. A thread has a program counter and a stack to keep track of local variables and return addresses. A multithreaded UNIX process is no longer a thread of control in itself; instead it is associated with one or more threads. There is generally no way to predict how the instructions of different threads are interleaved, though they have execution priorities that can influence the relative speed of execution.

In a system that uses *multithreading*, or a *multithreaded system*, two or more threads share a common UNIX process. Threads share the process instructions and much of its data. Threads share most of the process resources: open files, signals, user data.

Solaris 2.x Kernel Features

Lightweight Processes (LWPs)

A major feature of the new kernel is its support of multiple kernel-supported threads of control, called *lightweight processes* (LWPs), in any user process, sharing the address space of the process and other resources, such as open files. LWPs are discussed in further detail later in this chapter.

Interrupts

Interrupts are also handled by kernel threads. The kernel synchronizes with interrupt handlers via normal thread synchronization primitives. If an interrupt thread encounters a locked synchronization variable, it blocks and allows the critical section to clear.

Locking

Solaris 2.x is designed with a relatively "fine grained" locking strategy to take advantage of as many processors as possible. Each kernel subsystem has a locking strategy designed to allow a high degree of concurrency for frequent operations. Infrequent operations are usually coarsely locked with simple mutual exclusion.

BSD UNIX File System (UFS)

The BSDUNIX file system processes multiple read operations on the same or different regular files concurrently. The actual degree of parallelism for these operations is limited by the device drivers, buffer cache, and the Virtual Memory system. Write operations or directory operations are processed in parallel only for different files.

Kernel Data Structures

In the traditional kernel, the `user` and `proc` structures contained all kernel data for the process. Processor data was held in global variables and data structures. This has changed in the Solaris 2.x kernel. The restructured kernel must separate this data into data associated with each LWP and its kernel thread, the data associated with each process, and the data associated with each processor.

Running Uniprocessor Code on a Multiprocessing System

Software does not have to be rewritten for end users to see a performance gain with multiprocessing. However, not all applications are equally suited to this technique. Computing tasks that are structured as a single, large UNIX process may not benefit from multiprocessing systems because these tasks must run sequentially on only one CPU, leaving the others idle.

Avoiding Code Changes

Recompiling is not necessary when taking your executable file from a single processor machine to a multiprocessor. The executable will run on either type of system.

Because Solaris 2.x is a fully symmetric kernel with an internal architecture based on threads, end users will see an increase in performance on their single-processor systems. The kernel takes advantage of multiple threads to asynchronously perform I/O and execute daemons. The synchronization locks used in the kernel have also been optimized to execute quickly and consume little memory.

Improving Software Performance on Multiprocessing Systems

Executables that are built on a uniprocessor may be run on a multiprocessing system and vice-versa.

There are several ways to make software perform better and utilize system resources more efficiently. The Solaris 2.x kernel employs many of these techniques. Code changes could affect the way the software runs on a uniprocessing system, but the changes will only improve performance.

Utilizing End-User-Level Threads

Multiprocessing is a side benefit of having a multithreaded kernel and having applications utilize end-user-level threads. Threads provide the best combination of speed, concurrent functionality, and kernel resource utilization. On multiprocessor systems, threads of control are capable of running concurrently on different processors.

Making Drivers MT-Safe

Configuring the module as MT-safe allows the module to take maximum advantage of the concurrency of the underlying hardware. This additional performance comes at a cost. It is your responsibility to fully understand the relationship between all portions of the module and private data within that module and to use the synchronization primitives available to maintain the integrity of these private data structures.

Multithreaded Code and Multiprocessors

Multiprocessing hardware is not necessary for the implementation of a multithreaded application.

The reasons for supporting multiple threads of control in SunOS fall into two categories: those motivated by multiprocessor hardware, and those motivated by application concurrency. You can exploit multiprocessors to varying degrees depending on how much the uniprocessor base is modified. In the simplest case, only separate end-user processes can run on the additional processors; the applications are unchanged. To allow a single application to use multiple processors, the application must be restructured.

The second reason for multiple threads of control is application concurrency. Many applications are best structured as several independent computations. A database system may have many user interactions in progress while at the same time performing several file and network operations. A window system can treat each widget as a separate entity. A network server may indirectly need its own service (and therefore its own thread of control) to handle requests. In each case, although you can write the software as one thread of control moving from request to request, you can simplify the code by writing each request as a separate sequence and letting the language, library and operating system handle the interleaving of the different operations.

These examples indicate the opportunities for software developers porting to Solaris to exploit powerful hardware and build complex applications and services with this technology. The examples show that the user model for multiple threads of control must support a variety of applications and environments. The architecture should, where possible, use current programming paradigms and preserve software compatibility.

As is true of many system services today, the programmer's view of the multiple threads of control service is not always identical to what the kernel implements. The software view is created by a combination of the kernel, runtime libraries and the compilation system. This approach increases the portability of applications and systems by hiding some details of the implementation, while providing better performance, by allowing library code to do some work without involving the kernel.

Multithreading Considerations

Multithreading programming techniques are not appropriate for some applications. For example, compute-bound programs do not benefit from multithreading when run on uniprocessor hardware. For some I/O bound programs, the system may already provide the required degree of I/O concurrency, such as read-ahead and write-behind to regular files in the primary file system. Compute-bound programs with coarse-grained parallelism can benefit from multithreading when run on multiprocessor hardware. While the overhead involved in synchronizing and context switching threads is significantly smaller than using processes, it is not zero. Programs in which each thread does not execute enough code between synchronizations or context switches will have performance problems.

Even appropriate multithreaded applications can be difficult to design and to debug. The shared memory and process state of a multithreaded program can be very intimate. Because of the ease of access to shared data and other process resources, programming with multiple threads requires more care and discipline than does single-threaded programming.

End-User-Level Threads

A traditional UNIX process has a single thread of control. A multithreaded UNIX process is no longer a thread of control in itself, instead it is associated with one or more threads. Threads execute independently. In general, the number or identities of threads that an application process chooses to apply to a problem are invisible from outside the process. Threads can be viewed as execution resources that may be applied to solving the problem at hand.

Threads share the process instructions and most of its data. A change in data by one thread can be seen by the other threads in the process. Threads also share most of the operating system state of a process. Each sees the same open files. For example, if a thread opens a file, another thread can read it. Because threads share so much of the process state, threads can affect each other in sometimes surprising ways. Programming with threads requires more care and discipline than ordinary programming because there is no system enforced protection between threads.

Each thread may make arbitrary system calls and interact with other processes in the usual ways. Some operations affect all the threads in a process. For example, if one thread calls `exit()`, all threads are destroyed. Other system services have new interpretations; for example, a floating point overflow trap applies to a particular thread, not the whole program.

Synchronization Primitives

The architecture provides a variety of synchronization facilities to allow threads to cooperate in accessing shared data. The synchronization facilities include mutual exclusion (*mutex*) locks, read-write locks, condition variables, and semaphores. For example, a thread that wants to update a variable might block waiting for a mutual exclusion lock held by another thread that is already updating it. To support different frequencies of interaction and different degrees of concurrency, several synchronization mechanisms with different semantics are provided.

As shown in Figure 10-1, threads in different processes can synchronize with each other via synchronization variables placed in shared memory, even though the threads in different processes are generally invisible to each other. Synchronization variables can also be *mapped* in files and have lifetimes beyond that of the creating process. For example, a file can be created that contains data base records. Each record can contain a mutual exclusion lock variable that controls access to the associated record. A process can map the file and a thread within it can release the lock and unmap the file. Once the lock has been acquired, if any thread within any process mapping the file attempts to acquire the lock that thread will block until the lock is released.

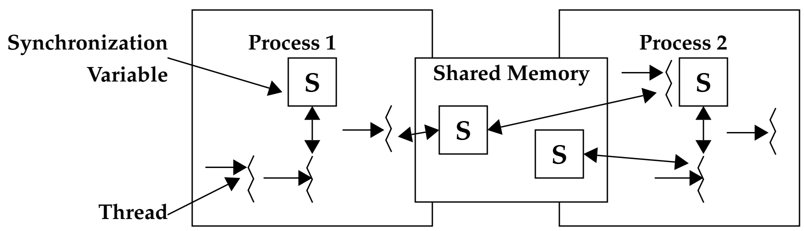

Figure 10-1 Synchronization Variables

Synchronization Primitives and Memory

In older multiprocessors, access to memory by the processors is strictly multiplexed. This means that a processor executing program code reads or writes memory in the order specified by the code. All memory operations of all processors in the system appear to happen in some global order, although the operation histories of different processors are interleaved arbitrarily. The memory operations of such machines are called *strongly ordered*. In this environment, threads can synchronize using ordinary memory operations. For example, a producer thread and a consumer thread can synchronize access to a circular data buffer. The following header could accompany a code example that uses Threads 1 and 2 as shown in Table 10-1:

```
int rdptr = 0;
int wrptr = 0
data_t buf[BUFSIZE];
```

Table 10-1 Producer/Consumer Example for Strongly Ordered Memory

Thread 1	Thread 2

`while (work_to_do) {`
`buf[wrptr] = produce();`
`wrptr = (wrptr + 1) %`
`BUFSIZE;`
`while (wrptr == rdptr)`
`;`
`}`
	`while (work_to_do) {`
	`while (rdptr == wrptr)`
	`;`
	`consume(buf[rdptr]);`
	`rdptr = (rdptr + 1) %`
	`BUFSIZE;`
	`}`

In current multiprocessors, the ordering is relaxed for greater performance. So, if one processor stores values in location A and then in location B, other processors loading data from location B and then location A may see the new value of B but the old value of A. The memory operations of such machines are called *weakly ordered*. On these machines the circular buffer technique shown in Table 10-1 fails because you may see the new value of `wrptr` but the old value of the data in the buffer.

In such machines, synchronization is only achieved through the use of special instructions that force an order on memory operations. Most high-level language compilers generate only ordinary memory operations to take advantage of the increased performance. They usually cannot determine when memory operation order is important and generate the special ordering instructions. They rely on the programmer to correctly use synchronization primitives to ensure that modifications to a location in memory are ordered with respect to modifications and access to the same location in other threads. Access to read-only data need not be synchronized. The resulting program is said to be *data-race free*.

Synchronization is still important even when accessing a single primitive variable, such as an integer. On machines where the integer may not be aligned to the bus data width or be larger than the data width, a single memory load may require multiple memory cycles. This means that it may be possible for some parts of the integer to have an old value while other parts have a newer value. On some processor architectures, such as the SPARC architecture, this cannot happen, but portable programs cannot rely on this.

In summary, a portable multithreaded program *must* use the synchronization primitives to synchronize data access. It cannot rely on modifications to memory being observed by other threads in the order written in the program or even on modification of a single variable being seen atomically.

Threads API

Most of the interfaces available to threads are those that are available to UNIX processes in a single-threaded UNIX environment. As mentioned previously, some of those interfaces have different implications in a multithreaded environment, but the intent is to provide "UNIX semantics" as the ordinary programming model. This section describes some of the additional interfaces needed to create and manage threads.

It is important to remember that `errno` is uniquely allocate to each thread. This allows threads to make simultaneous system calls without interfering with each other's instance of `errno`. The thread and LWP interfaces do not rely on a per-thread `errno`. These new interfaces return zero on success and a positive value, which is `errno` when they fail.

The interface syntax follows.

Code Example 10-1 Threads API

```
int     thr_create(void *stack_base, size_t
        stack_size,void *(*start_routine) (void *),
        void *arg, long flags,
        thread_t *new_thread);
size_t  thr_min_stack(void);
int     thr_setconcurrency(int new_level);
int     thr_getconcurrency(void);
void    thread_exit(void *status);
int     thr_suspend(thread_t target_thread);
int     thr_continue(thread_t target_thread);
int     thr_sigsetmask(int how, const sigset_t *set, sigset_t *oset);
int     thr_setprio(thread_t target_thread, int pri);
int     thr_getprio(thread_t target_thread, int *pri);
            thread_t thr_self(void);
int     thr_kill(thread_t target_thread, int sig);
void    thr_yield(void);
int     thr_keycreate(thread_key_t *keyp, void (*destructor)(void *value));
int     thr_setspecific(thread_key_t keyp, void *value);
int     thr_getspecific(thread_key_t keyp, void **valuep);
int     thr_join(thread_t wait_for, thread_t *departed, void **status);
void    mutex_init(mutex_t *mp, int type, void *arg);
void    mutex_destroy(mutex_t *mp);
void    mutex_lock(mutex_t *mp);
int     mutex_trylock(mutex_t *mp);
int     mutex_unlock(mutex_t *mp);
int     cond_init(cond_t *cvp, int type, int arg);
int     cond_destroy(cond_t *cvp);
int     cond_wait(cond_t *cvp, mutex_t *mp);
int     cond_timedwait(cond_t *cvp, mutex_t *mp, timestruc_t *abstime);
```

Code Example 10-1 Threads API (Continued)

```
int     cond_signal(cond_t *cvp);
int     cond_broadcast(cond_t *cvp);
int     sema_init(sema_t *sp, unsigned int count, int type, void *arg);
int     sema_destroy(sema_t *sp);
int     sema_wait(sema_t *sp);
int     sema_trywait(sema_t *sp);
int     sema_post(sema_t *sp);
int     rwlock_init(rwlock_t *rwlp, int type, void *arg);
int     rwlock_destroy(rwlock_t *rwlp);
int     rw_rdlock(rwlock_t *rwlp);
int     rw_wrlock(rwlock_t *rwlp);
int     rw_unlock(rwlock_t *rwlp);
int     rw_tryrdlock(rwlock_t *rwlp);
int     rw_trywrlock(rwlock_t *rwlp);
```

For specific information regarding each API, consult the MT man pages.

Writing MT-Safe Libraries

SunOS 5.x contains the kernel support for multiple threads of control in a single process address space. In the SunOS multithreaded architecture, threads are lightweight abstractions implemented by the threads library. The library controls how threads are scheduled onto lightweight processes (LWPs) which are the independent execution entities within the process and are supported by the kernel. This allows many hundreds of threads to exist in a process while the number of LWPs can be tailored to the actual concurrent need for system resources. The overall architecture is shown in Figure 10-2.

.

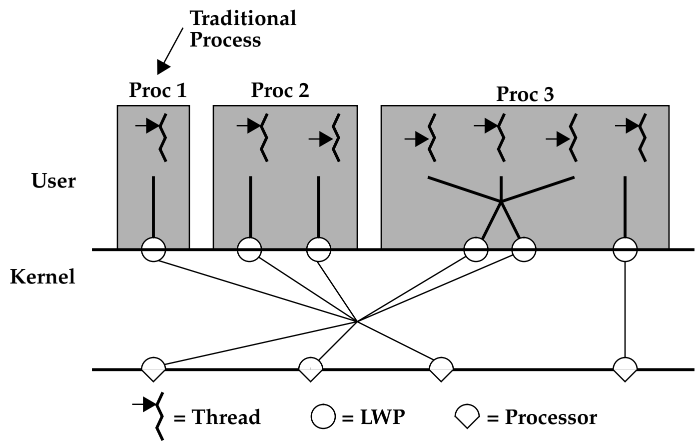

Figure 10-2 SunOS 5.x MT Architecture

In many cases an application need not be aware of the number of LWPs used because the library ensures that enough LWPs are created to prevent a total loss of concurrency when runnable threads are present. However, this may not be the optimal number for performance. When required, the size of the pool of LWPs used to schedule threads can be controlled by the application. Threads can also be bound to LWPs when there is some aspect of an LWP that is required by a thread, such as a system-wide, realtime priority.

MT Libraries

The general goal of converting existing libraries to a multithreaded environment is to provide correct operation when a library interface is entered by more than one thread simultaneously, such as when the library is MT-safe. In addition, it is usually desirable that long operations such as I/O not block other threads from using the library while the operation completes. In libraries that are not computationally intensive (most system libraries), it is much less important to allow many processors to execute library code simultaneously. In some libraries the interfaces rely on global data that will not work correctly in a multithreaded environment and must be changed.

Reentrant System Interfaces

POSIX 1003.4a has defined reentrant versions of the POSIX 1003.1 system interfaces. In most cases the interfaces are either defined to be reentrant or use synchronization to protect access to shared data. A good example of the latter is `malloc()`. Different threads can simultaneously enter `malloc()` and the implementation provides enough synchronization so that threads can allocate memory for a common heap.

In some cases the interface is inherently non-reentrant. An example of this is `getpwnam()`. This interface returns a pointer to a static data area. If a second thread enters `getpwnam()` before the thread that called `getpwnam()` first has completely consumed the entry in the static buffer, the entry could be overwritten. One solution is to put the buffer in a thread specific data area. This allows threads to call `getpwnam()` independently. This approach has the disadvantage that a data area must be allocated for each thread that uses the interface.

An alternative approach is to define a new, reentrant interface to these functions. For functions that return pointers to static data areas, the interface may be changed to have the caller pass in pointers to the memory in which the results can be stored. This is the approach taken by POSIX.

For example, POSIX defines a new interface, `getpwnam_r()`, which takes three additional arguments; a pointer to a struct `passwd` for the result, a buffer in which strings pointed to by the returned struct `passwd` are placed, and the size of the supplied buffer. This approach keeps the per-thread storage to a minimum, and allows the calling function to manage the required memory as appropriate.

In some cases where a new interface has been defined, the old interfaces will remain. They are still usable provided they are either called from a single thread or the application provides the appropriate locking before calling any of these routines.

Some interfaces can be made reentrant, but the overhead involved in hiding the required locking beneath the interface is too great. An example is the `stdio` library function `putc()`. By default, `putc()` is implemented with the required locking of the I/O buffers. However, the overhead of locking and unlocking the I/O buffers for each character can be too great in some situations. POSIX defines three new interfaces—

- `flockfile()`
- `funlockfile()`
- `putc_unlocked()`

to help in these situations. The first two interfaces serialize multiple access to a file. Once the file is locked, `putc_unlocked()` outputs characters until the file is unlocked. This allows the application to control the locking granularity to suit its needs.

Threads to LWP Mapping

Threads are implemented using LWPs. LWPs within a process execute threads as shown in Figure 10-3. An LWP chooses a thread to run by locating the thread state in process memory (a). After loading the registers and assuming the identity of the thread, the LWP executes the thread's instructions (b). If the thread cannot continue, or if other threads should be run, the LWP saves the state of the thread back in memory (c). The LWP can now select another thread to run (d).

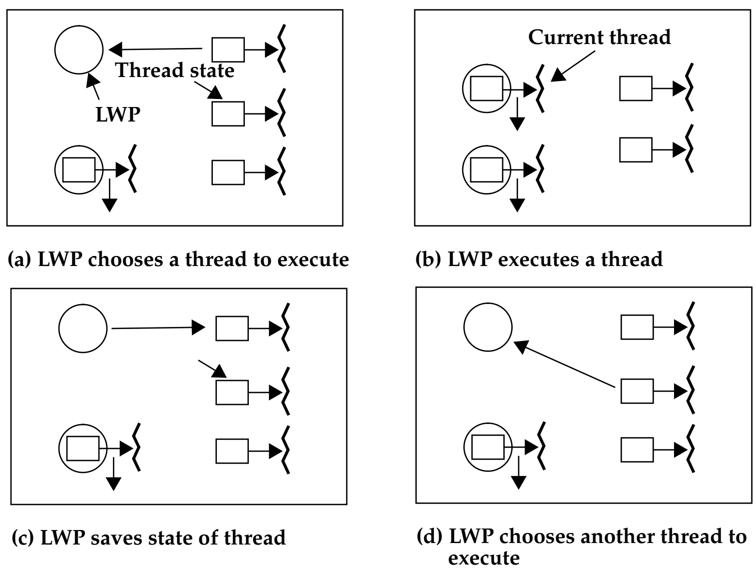

Figure 10-3 LWPs Running Threads

When a thread needs to access a system service by performing a kernel call, taking a page fault, or to interact with threads in other processes, it does so using the LWP that is executing it. The thread needing the system service remains bound to the LWP that is executing it until the system call is completed. If a thread needs to interact with other threads in the same process, it can do so without involving the operating system. As Figure 10-3 shows, switching from one thread to another occurs without the kernel knowing about it. Much as the UNIX `stdio` library routines (such as `fopen()` and `fread()`) are implemented using the UNIX system calls (`open()` and `read()`), the thread interface is implemented using the LWP interface, and for many of the same reasons.

An LWP may also have some capabilities that are not exported directly to threads, such as a special scheduling class. A programmer can take advantage of these capabilities while still retaining use of all the thread interfaces and capabilities (e.g., synchronization) by specifying that the thread is to remain permanently bound to an LWP.

Threads are the primary interface for application parallelism. Few multithreaded programs will use the LWP interface directly, but it is sometimes important to know that it is there. Most software developers can program using the threads interface and let the library take care of mapping threads onto the kernel primitives. The decision of how many LWPs should be created to run the threads can be left to the library or may be specified by the programmer.

Programming with User Threads vs. LWPs

You might wonder why it is necessary to have two interfaces that are so similar. The multithreaded architecture must meet a variety of different expectations. Some programs have large amounts of logical parallelism, such as a window system that provides each widget with one input handler and one output handler. Other programs need to map their parallel computation onto the actual number of processors available. In both cases, programs want full access to the system services.

Threads are implemented by the library and are not known to the kernel. Thus threads may be created, destroyed, or activated without involving the kernel. LWPs are implemented by the kernel. If a thread wants to read from a file, the kernel needs to be able to switch to other processing when the LWP blocks in the file system code waiting for the I/O to finish. The kernel has to preserve the state of the `read` operation and continue it when the I/O interrupt arrives. However, if each thread were always known to the kernel, it would have to allocate kernel data structures for each one and get involved in context switching threads even though most thread interactions involve threads in the same process. In other words, kernel supported parallelism (LWPs) is relatively expensive compared to threads. Having all the threads supported by the kernel would cause applications, such as the window system, to be much less efficient. Although the window system may be best expressed as a large number of threads, only a few of the threads ever need to be active—that is, require kernel resources other than virtual memory—at the same instant.

Sometimes having more threads than LWPs is a disadvantage. A parallel array computation divides the rows of its arrays among different threads. If there is one LWP per processor, but multiple threads per LWP, each processor spends overhead switching between threads. Your code may run faster if you know there is one thread per LWP because you can divide the rows among a smaller number of threads and reduce the number of thread switches. By specifying that each thread is permanently bound to its own LWP, you can write thread code that is really LWP code, much like locking down pages turns virtual memory into real memory.

A mixture of threads that are permanently bound to LWPs and unbound threads is also appropriate for some applications. An example of this would be some realtime applications that want some threads to have system-wide priority and realtime scheduling, while other threads can attend to background computations in the default timesharing scheduling class.

By defining both levels of interface in the architecture, we make clear the distinction between what the developer sees and what the kernel provides. Most application developers program using threads and do not think about LWPs. When it is appropriate to optimize the behavior of the program, you can tune the relationship between threads and LWPs. This allows developers to structure their application assuming extremely lightweight threads while bringing the appropriate degree of kernel-supported concurrency to bear on the computation. To some degree, when you program using threads, you can think of LWPs used by the application as the degree of real concurrency that the application requires.

Why Have Both Threads and LWPs?

Solaris 2.x must provide a variety of system services for a variety of software developers, many of whom have different expectations. Some developers want a large number of logical threads, independent of how many CPUs the hardware can provide. Others, such as database vendors, have a critical need to map their parallel computations onto the actual number of processors available.

Logically separating threads and LWPs provides flexibility. You can choose to use the resources best suited for your applications without excessive use of system resources.

Kernel Threads and User Threads

A *kernel thread* is the fundamental entity that is scheduled and dispatched onto one of the CPUs of the system. Kernel threads are fully preemptive and may be scheduled by any of the scheduling classes in the system, including the realtime (fixed priority) class. Because all other execution entities are built using kernel threads, they represent a fully preemptive, realtime “nucleus” within the kernel.

Kernel threads use synchronization primitives that support protocols for preventing priority inversion, so a thread's priority is determined by which activities it is impeding by holding locks as well as by the service it is performing.

SunOS 5.x uses kernel threads to provide asynchronous writes to disk, servicing STREAMS queues and callouts. This removes various diversions in the idle and trap code and replaces them with independently scheduled threads. This increases potential concurrency (these activities can be handled by other CPUs), and also gives each asynchronous activity a priority so it can be appropriately scheduled.

Even interrupts are handled by kernel threads. The kernel synchronizes with interrupt handles via normal thread synchronization primitives. If an interrupt thread encounters a locked synchronization variable, it blocks and allows the critical section to clear.

A major feature of Solaris 2.x is its support for end-user-level execution resources, called LWPs. These LWPs are implanted as kernel threads with some extra state to support the user level model of multithreading. It should be noted that kernel threads are not LWPs, however LWPs are always kernel threads.

LWPs are created within a process and share all process resources like address space and open file descriptors. Because LWPs are really implemented by the kernel threads, they are scheduled by the kernel onto CPUs, as shown in the following figure.

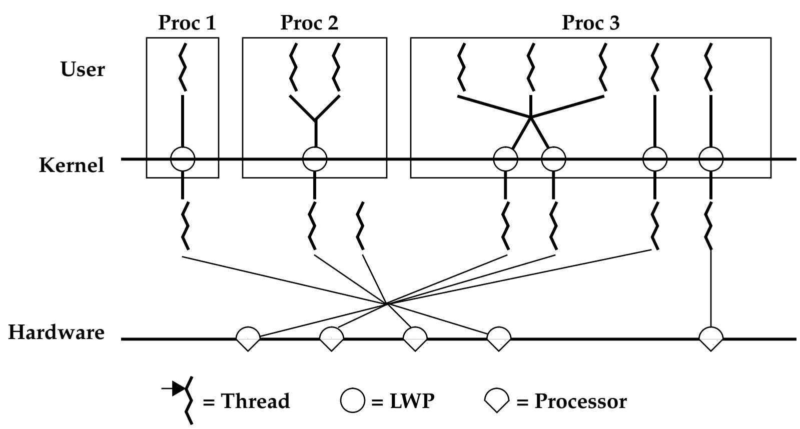

Figure 10-4 Multi-Thread Architecture Examples

An end-user-level library uses LWPs to implement end-user-level threads. These threads are scheduled at user-level and are switched by the library to any of the LWPs belonging to the process. End-user threads can also be bound to a particular LWP. Separating end-user-level threads from the LWP allows the end-user thread library to quickly switch between end-user threads without entering the kernel. In addition, it allows an end-user's process to have thousands of threads without overusing the kernel resources. A thread can be identical to an LWP if it is created bound to an LWP. LWPs have a few extra properties that are only accessible if the thread is bound.

Realtime Priority to Threads

The realtime class (RT) can be applied to a whole process or to one or more bound threads in a process. Bound threads that are classified realtime can be assigned priorities individually or jointly. The scheduler always dispatches the highest priority realtime process or bound thread. It preempts a lower priority process or bound thread when a higher priority process or bound thread can be run. A preempted bound thread is placed at the head of its level queue. A realtime bound thread retains control of a processor until it is preempted, it suspends, or its realtime priority is changed. Processes and bound threads in the RT class have absolute priority over processes in the TS class. Given the latency of the system, in recognizing interrupts and dispatching the highest priority process, the worst case behavior can be calculated.

A new bound thread inherits the scheduling class of the parent process or bound thread. An RT class inherits the parent's time slice, whether finite or infinite. A bound thread with a finite time slice runs until it terminates, blocks (for example, to wait for an I/O event), is preempted by a higher priority realtime process that can be run, or the time slice expires. a bound thread with an infinite time slice ceases execution only when it terminates, blocks, or is preempted.

The `priocntl()` system call sets the realtime priority and time quantum of one or more processes. The `priocntl()` call affects the attributes of all realtime class bound threads in the process at a time.

Debugging MT Programs

Common Bugs

Multithreaded programming differs enough from conventional synchronous programming that it is worthwhile to identify some of the most common oversights that can result in a bug:

- Accessing global memory without the protection of a synchronous mechanism.
- Deadlocks caused by two threads trying to acquire rights to the same pair of global resources in alternate order; one thread controls the first resource, the other controls the second resource, and neither can proceed until the other gives up.
- Hidden gaps in synchronization protection. This is caused when a segment of code that is protected by a synchronous mechanism contains a call to a function that frees and then reacquires the synchronization mechanism before it returns to the caller. The result that appears to the caller is that the global data has been protected when it actually has not been protected.

Using adb

The commands that have been added to adb for use with multithreaded programs are shown in Table 10-2.

Table 10-2 Commands Added to adb

Command	Function
pid:A	Attach to process # pid. This stops the process and all its LWPs.
:R	Detach from process. This resumes the process and all its LWPs.
$L	List all active LWPs in the stopped process.
n:l	Switch focus to LWP # n.
$l	Show the LWP currently focused on.
num:i	Ignore signal number num.

An example of the use of these commands is:

1. Determine the process identifier of the multithreaded program you want to debug.
2. Invoke adb on the program.
3. Type pid:A to attach to the hung process.
4. Type $L to list all the LWPs.
5. If there is more than one LWP, type n:l to switch to each LWP, and determine its state at the time the process stopped using standard adb commands such as $c and $r. For example, typing 3:l switches focus to LWP #3. Type $c to see the stack trace for LWP #3, and type $r to see its registers.

You can have adb ignore signals by typing num:i. If the signal number num is received, the signal is passed to the process being debugged but is ignored by adb.

You can bind all the threads in a multithreaded program so a thread and an LWP are synonymous. Then when you use adb, each thread is accessible using the preceding commands.

MT dbx

Consult the man page for dbx for a listing of the new commands within dbx which support multithreading.

Multithreading Example Programs

The example code entitled "MT Advantage", included on the floppy disk, shows the performance advantages of multithreaded technology in a client/server application even on uniprocessor hardware. The client application simply retrieves three different spot oil prices from three remote services. For the purpose of this demonstration, these services are located on one demonstration system.

When Start Request is pressed at the bottom of each client, that client's indicator lights change color. Red signifies that no connection has been established with that service, yellow indicates that a connection has been opened and the data transfer is in progress, and green indicates that the data transfer has completed.

In the traditional single threaded application, the client sequentially moves from one service to the next, establishing a network connection, transferring the data, and posting it on the display. The yellow indicator moves from left to right as connections are made and spot oil prices are received from each remote service. The concurrent execution graph is updated only after each request is completed, because single threaded applications cannot interrupt a data transfer in progress.

The multithreaded application never displays a red indicator. All lights immediately turn yellow as separate threads are created to transfer data from each service. Because a separate thread is processing graphical user interface events, the display constantly monitors execution and updates the graph. Also, all buttons continue functioning throughout the entire request.

Bibliography

SunOS Multithread Architecture, Part Number 91033-002, M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, SunSoft, Inc., Mountain View, CA.

Writing Multithreaded Code in Solaris 2.0, Part Number 91003-002, Steve Kleiman, Bart Smaalders, Dan Stein, Devang Shah, SunSoft Inc., Mountain View, CA.

Beyond Multiprocessing... Multithreading the SunOS Kernel, Part Number 91003-002, J.R. Eykholt, S.R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, D. Williams, SunSoft, Inc., Mountain View, CA.

Guide to Multithread Programming, Part Number 801-3176-03, SunSoft, Inc., Mountain View, CA.