Multi-Threaded STREAMS 13

MT STREAMS Overview

This chapter describes how to multi-thread a STREAMS driver or module. It covers the necessary conversion topics so that new and existing STREAMS modules and drivers will run in the multi-threaded kernel. We will be looking mostly at STREAMS specific multi-threading issues and techniques. Refer also to the *Writing Device Drivers* manual.

SunOS 5.x is a fully multi-threaded operating system able to make effective use of the available parallelism of a symmetric shared-memory multiprocessor computer. All kernel subsystems have been multi-threaded: scheduler, virtual memory, file systems, block/character/STREAMS I/O, networking protocols, and device drivers.

MT STREAMS requires you to use some new concepts and terminology. These concepts apply not only to STREAMS drivers, but to all device drivers in SunOS. For more a complete description of these terms, see *Writing Device Drivers*. Additionally, see Chapter 2, “Overview of STREAMS" of this guide for definitions, and Chapter 5, “Messages" for elements of MT drivers.

As an overview, you will need to understand the following terms and ideas.

Thread	sequence of instructions executed within context of a process
Lock	mechanism for restricting access to data structures
Single Threaded	restricting access to a single thread
Multi Threaded	allowing two or more threads access
Multiprocessing	two or more CPUs concurrently executing the OS

Concurrency	simultaneous execution
Preemption	suspending execution for the next thread to run
Monitor	portion of code that is single threaded
Mutual Exclusion	exclusive access to a data element by a single thread
Condition Variables	kernel event synchronization primitives
Counting Semaphores	memory based synchronization mechanism
Readers/Writer Locks	data lock allowing one writer and many readers
Callback	upon specific event, call module function

MT STREAMS Framework

The STREAMS framework consists of the Stream head, STREAMS utility routines, and documented STREAMS data structures. The STREAMS framework allows multiple kernel threads to concurrently enter and execute within each module. There may be multiple threads actively executing within the `open`, `close`, `put`, and `service` procedures for each queue within the system.

A goal of SunOS 5.x is to preserve the interface and flavor of STREAMS to shield module code as much as possible from the impact of migrating to the multi-threaded kernel. The majority of the locking is hidden from the programmer and performed by the STREAMS kernel framework. As long as module code uses the standard, documented programmatic interfaces to shared kernel data structures (such as `queue_t`, `mblk_t`, and `dblk_t`), it will not have to explicitly lock these framework data structures.

A second goal is to make it simple to write MT SAFE modules. The framework accomplishes this by providing the MT STREAMS perimeter mechanisms for controlling and restricting the concurrency in a STREAMS module. See the section ”MT SAFE Modules".

The DDI/DKI entry points (`open`, `close`, `put`, and `service` procedures) plus certain callback procedures (scheduled with `qtimeout`, `qbufcall`, or `qwriter`) are termed *synchronous* entry points. All other entry points into a module are termed *asynchronous*. Examples of the latter are hardware interrupt routines, `timeout`, `bufcall`, and `esballoc` callback routines.

STREAMS Framework Integrity

The STREAMS framework guarantees the integrity of the STREAMS framework data structures, such as `queue_t`, `mblk_t`, and `dblk_t`, assuming the module conforms to the DDI/DKI thus does not directly access global operating system data structures nor facilities not described within the Driver-Kernel Interface.

The `q_next` and `q_ptr` fields of the `queue_t` structure will not be modified by the system while a thread is actively executing within a synchronous entry point. The `q_next` field of the `queue_t` structure could change while a thread is executing within an asynchronous entry point.

As in previous SunOS releases, a module must not call another module's `put` or `service` procedures directly. The DDI/DKI routines `putnext`(), `put`(), and others in Section 9F must be used to pass the message to another queue. Calling another module's routines directly circumvents the design of the MT STREAMS framework and can yield unknown results.

When making your module MT SAFE, the integrity of private module data structures must be ensured by the module itself. Knowing the boundaries of what the framework supports is critical in deciding what you must provide yourself. The integrity of private module data structures can be maintained by either using the MT STREAMS perimeters to control the concurrency in the module, by using module private locks, or by a combination of the two.

Message Ordering

The STREAMS framework guarantees the ordering of messages along a stream if all the modules in the stream preserves message ordering internally. This ordering guarantee only applies to messages that are sent along the same stream and produced by the same source.

The STREAMS framework does not guarantee that a message has been seen by the next `put` procedure when `putnext`(), `qreply`() return.

13

Your MT Options

There are two MT configuration options available to a module (or driver):

- MT SAFE
- MT UNSAFE

MT SAFE modules

For MT SAFE mode it is possible to use MT STREAMS perimeters to restrict the concurrency in a module or driver to e.g.:

- Per module single-threading
- Per queue-pair single-threading
- Per queue single-threading
- Per queue or per queue-pair single-threading of the `put` and `service` procedures with per module single-threading of the `open` and `close` routines.
- Unrestricted concurrency in the `put` and `service` procedures with the ability to restrict the concurrency when handling messages that modify data.
- Completely unrestricted concurrency.

We recommend that you initially implement your module and configure it to be per-module single-threaded, and increase the level of concurrency as needed. The section ”Sample Multi-threaded Device Driver" provides a complete example of using a per-module perimeter, and the section ”Sample Multi-threaded Module with Outer perimeter" provides a complete example with a higher level of concurrency.

MT SAFE modules can use different MT STREAMS perimeters to restrict the concurrency in the module to a concurrency that is natural given the data structures that the module contains, thereby removing the need for module private locks. A module that requires unrestricted concurrency can be configured to have no perimeters. Such modules have to use explicit locking primitives to protect their data structures. While such modules can exploit the maximum level of concurrency allowed by the underlying hardware platform, they are more complex to develop and support. See the section on ”MT SAFE Modules using Explicit Locks".

Independent of the perimeters, there will be at most one thread allowed within any given queue's service procedure.

MT UNSAFE modules

MT UNSAFE mode for STREAMS modules is temporarily supported as an aid in porting SVR4 modules. MT UNSAFE might not be supported in future versions of the operating system. See the section on "MT UNSAFE Modules" on page 300 for details.

Preparing to Port

When modifying a STREAMS driver to take advantage of the multi-threaded kernel, a level of MT-safeness is selected according to:

- The desired degree of concurrency
- The natural concurrency of the underlying module
- The amount of effort/complexity required

Note that much of the effort in conversion is simply determining the appropriate degree of data sharing and the corresponding granularity of locking. The actual time spent configuring perimeters and/or installing locks should be much smaller than the time spent in analysis.

To port your module, you must understand the data structures used within your module as well as accesses to those data structures. 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 MT STREAMS perimeters (or the synchronization primitives available) to maintain the integrity of these private data structures.

It is your responsibility to explicitly restrict access to private module data structures as appropriate to ensure the integrity of these data structures. You must use the MT STREAMS perimeters to restrict the concurrency in the module so that the parts of the module that modify module private data is single threaded with respect to the parts of the module that read the same data. Alternatively to the perimeters, you can use the synchronization primitives available (mutex, condition variables, readers/writer, semaphore) to explicitly restrict access to module private data appropriate for the operations within the module on that data.

The first step in multi-threading a module or driver is to analyze the module, breaking the entire module up into a list of individual operations and the private data structures referenced in each operation. Part of this first step is deciding upon a level of concurrency for the module. Ask yourself which of these operations can be multi threaded and which must be single threaded. Try to find a level of concurrency that is "natural" for the module and that matches one of the available perimeters (or alternatively, requires the minimal number of locks) and that has a simple and straightforward implementation. Avoid additional complexity. Avoid the desire to overly multi-thread the module at this point. Simple is better at this stage.

Typical questions to be answered are:

1. what data structures are maintained within the module?
2. what types of accesses are made to each field of these data structures?
3. when is each data structure accessed destructively (written) and when is it accessed non-destructively (read)?
4. which operations within the module should be allowed to execute concurrently?
5. is per-module single-threading appropriate for the module?
6. is per queue-pair or per queue single-threading appropriate?
7. what are the message ordering requirements?

Examples of natural levels of concurrency are:

- A module, where the `put` procedures read as well as modify module global data can be configured to be per module single-threaded using a per module inner perimeter.
- A module, where all the module private data is associated with a queue (or a read/write pair of queues) can be configured to be single-threaded for each queue (or queue pair) using the corresponding inner perimeter.
- A module where most of the module private data is associated with a queue (or a queue pair), but that in addition has some module global data which is mostly read, can be configured with a queue (or queue pair) inner perimeter plus an outer perimeter. The module can then use `qwriter()` to protect the sections where it modifies the module global data.

- A module that falls in one of the above categories, but requires higher concurrency for certain message types while not requiring message ordering, can be configured as one of the above perimeters with the addition of specifying shared inner perimeter access for the `put` procedures. The module can then use `qwriter()` when messages are handled in the `put` procedures that require exclusive access at the inner and/or outer perimeter.
- A hardware driver can use an appropriate set of inner and outer perimeters to restrict the concurrency in the `open`, `close`, `put`, and `service` procedures. Together with explicit synchronization primitives, these drivers restrict the concurrency when accessing the hardware registers in interrupt handlers etc. Such drivers need to be aware of the issues listed in the section ”MT SAFE Modules using Explicit Locks".

Porting to SunOS 5.x

When porting a SunOS 4.x STREAMS module or driver to SunOS 5.x, the module should be examined with respect to the following areas:

- SunOS 5.x Device Driver Interface (DDI/DKI).
- SunOS 5.x MT Design

For portability and correct operation, each module must adhere to the SunOS DDI/DKI. Several facilities available in previous releases of SunOS have changed and may take different arguments or provide different side-effects or may no longer exist in SunOS 5.x. The module writer should carefully review the module with respect to the DDI/DKI.

Each module that accesses underlying Sun-specific features included within SunOS should conform to the Device Driver Interface. The SunOS 5.x DDI defines the interface used by the device driver to register device hardware interrupts, access device node properties, map device slave memory, and establish and synchronize memory mappings for DVMA (Direct Virtual Memory Access). These areas are primarily applicable to hardware device drivers. Refer to the Device Driver Interface Specification within the *Writing Device Drivers* for details on the 5.x DDI and DVMA.

The kernel networking subsystem in SunOS 5.X is STREAMS based. Datalink drivers which used the `ifnet` interface in SunOS 4.x must be converted to use DLPI for SunOS 5.X. Refer to the *Data Link Provider Interface, Revision 2* specification.

13

After reviewing the module for conformance to the SunOS 5.x DKI and DDI specifications, the module writer should be able to consider the impact of multi-threading on the module.

MT SAFE Modules

We recommend that your MT SAFE modules use perimeters and avoid using module private locks. Should you opt to us-e module private locks you need to read the section ”MT SAFE Modules using Explicit Locks" in addition to this section.

MT STREAMS perimeters

Note – The support for MT STREAMS perimeters and related interfaces (`qwriter`, `qwait`, `qtimeout`, and `qbufcall`) was new to SunOS 5.3. These interfaces are subject to minor change based on further experience using these facilities.

For the purpose of controlling and restricting the concurrency for the synchronous entry points, the STREAMS framework defines two MT *perimeters*.The STREAMS framework provides the concepts of *inner* and *outer* perimeters. A module can be configured either to have no perimeters, to have only an inner or an outer perimeter, or to both an inner and outer perimeter. For inner perimeters there are different scope perimeters to choose from. Unrestricted concurrency can be obtained by configuring no perimeters.

Figure 13-1 and figure 13-2 are examples of inner perimeters, and figure 13-3 shows multiple inner perimeters inside an outer perimeter.

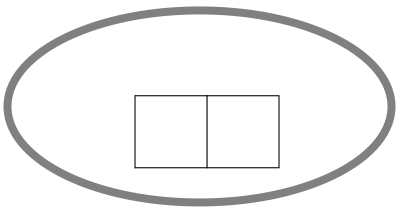

Figure 13-1 Inner perimeter spanning a pair of queues. (D_MPTQAIR)

Both the inner and outer perimeters act as readers/writer locks allowing multiple readers or a single writer. Thus, each perimeter can be entered in two modes: shared (reader) or exclusive (writer). By default all synchronous entry points enter the inner perimeter exclusively and the outer perimeter shared.

The inner and outer perimeters are entered when one of the synchronous entry points is called and the perimeters are retained until the call returns from the entry point. Thus, for example, the thread does not leave the perimeter of one module when it calls `putnext()` to enter another module.

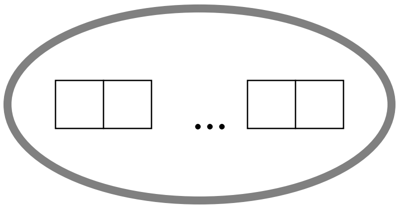

Figure 13-2 Inner perimeter spanning all queues in a module. (D_MTPERMOD)

When a thread is inside a perimeter and it calls `putnext()` (or `putnextctl()` etc.), it is possible that the thread will "loop around" through other STREAMS modules and try to re-enter a put procedure inside the original perimeter. If this re-entry conflicts with the earlier entry (e.g. if the first entry has exclusive access at the inner perimeter), the STREAMS framework will defer the re-entry while preserving the order of the messages attempting to enter the perimeter. Thus, `putnext()` will return without the message having been passed to the `put` procedure and the framework will pass in the message to the `put` procedure when it is possible to enter the perimeters.

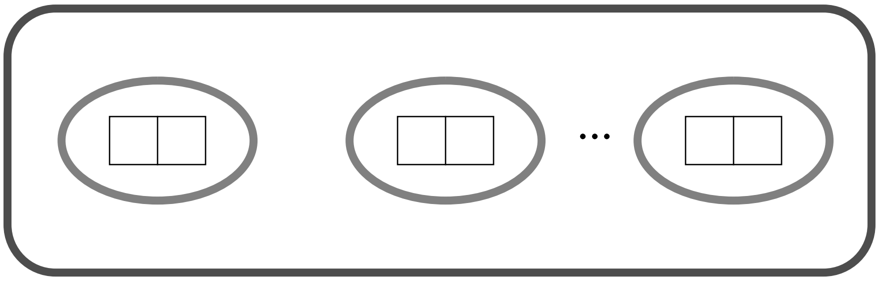

Figure 13-3 Outer perimeter spanning all queues in a module with inner perimeters spanning each pair of queues. (D_MTOUTPERIM combined with D_MTQPAIR)

The optional outer perimeter spans all queues in a module (see figure 13-3)

Perimeter options

There are several flags that are used to specify the perimeters. These flags fall into three categories:

- Define the presence and scope of the inner perimeter
- Define the presence of the outer perimeter (which can have only one scope)
- Modify the default concurrency for the different entry points

The inner perimeter is controlled by these mutually exclusive flags:

- `D_MTPERMOD`: The module has an inner perimeter that encloses all the module's queues.
- `D_MTAPAIR`: The module has an inner perimeter around each read/write pair of queues.
- `D_MTPERQ`: The module has an inner perimeter around each queue.
- None of the above: The module has no inner perimeter.

The presence of the outer perimeter is configured using:

- `D_MTOUTEPERIM`: In addition to any inner perimeter (or none), the module has an outer perimeter that encloses all the module's queues. This can be combined with all the inner perimeter options except `D_MTPERMOD`.

Recall that by default all synchronous entry points enter the inner perimeter exclusively and enter the outer perimeter shared. This behavior can be modified in two ways:

- `D_MTOCEXCL`: The framework invokes the `open` and `close` procedures with exclusive access at the outer perimeter (instead of the default shared access at the outer perimeter.)
- `D_MTPUTSHARED`: The framework invokes the `put` procedures with shared access at the inner perimeter (instead of the default exclusive access at the inner perimeter.)

MT configuration

To configure the driver as being MT SAFE, the `cb_ops` and `dev_ops` data structures must be initialized. This code must be in the header section of your module. For more information, see the example program in the section "Sample Multi-threaded Device Driver", the code sample in Appendix E, "Configuration" and `cb_ops(9S)` and `dev_ops(9S)`.

The driver is configured to be MT SAFE by setting the `cb_flag` field to `D_MP`. It also configures any MT STREAMS perimeters by setting flags in the `cb_flag` field. (See `mt-streams(9F)`.) The corresponding configuration for a module is done using the `f_flag` field in the `fmodsw` data structure.

qprocson()/qprocsoff()

The routines `qprocson`() and `qprocsoff`() respectively enable and disable the `put` and `service` procedures of the queue pair. Prior to the call to `qprocson`, and after the call to `qprocsoff`, the module's `put` and `service` procedures are disabled; messages flow around the module as if it were not present in the Stream.

The `qprocson`() routine must be called by the first open of a module, but only after allocation and initialization of any module resources on which the `put` and `service` procedures depend. The `qprocsoff`() routine must be called by the `close` routine of the module before deallocating any resources on which the `put` and `service` procedures depend.

Note – To avoid deadlocks, modules should not hold private locks across the calls to qprocson() or qprocsoff().

qtimeout()/qbufcall()

The timeout() and bufcall() callbacks are asynchronous, that is, they are not tracked by the STREAMS framework. For a module using MT STREAMS perimeters, this implies that the timeout and bufcall callback functions execute outside the scope of the perimeters. This makes it complex for the callbacks to synchronize with the rest of the module.

To make timeout and bufcall functionality easier to use for modules with perimeters, there are additional interfaces that use synchronous callbacks. These routines are qtimeout(9F), quntimeout(9F), qbufcall(9F), and qunbufcall(9F). When using these routines, the callback functions are executed inside the perimeters, i.e. with the same concurrency restrictions as the put and service procedures.

qwriter()

Modules can use the qwriter(9F) function to upgrade from shared to exclusive access at a perimeter. For example, a module with an outer perimeter can use qwriter() in the put or service procedures to upgrade to exclusive access at the outer perimeter. A module where the put procedures run with shared access at the inner perimeter (D_MTPUTSHARED) can use qwriter() in the put or service procedures to upgrade to exclusive access at the inner perimeter.

Note – Note that qwriter() cannot be used in the open or close procedures. If a module needs exclusive access at the outer perimeter in the open and/or close procedures, it has to specify that the outer perimeter should always be entered exclusively for open and close (using D_MTOCEXCL).

The STREAMS framework guarantees that all deferred qwriter callbacks associated with a queue have executed before the module's close routine is called for that queue.

For an example of a driver using `qwriter()` see the section “”Sample Multi-threaded Module with Outer perimeter".

qwait()

A module that uses perimeters and must wait in its `open` or `close` procedure for a message from another STREAMS module has to wait outside the perimeters; otherwise the message would never be allowed to enter its `put` and `service` procedures. This is accomplished by using the `qwait()` interface. See `qwriter(9F)` for an example.

Asynchronous Callbacks

Interrupt handlers and other asynchronous callback functions require special care by the module writer, since they can execute asynchronously to threads executing within the module `open`, `close`, `put`, and `service` procedures.

For modules using perimeters, we recommend using `qtimeout` and `qbufcall` instead of `timeout` and `bufcall`, since the `qtimeout` and `qbufcall` callbacks are synchronous and consequently introduce no special synchronization requirements.

Since a thread can enter the module at any time, the module writer is responsible for ensuring that the asynchronous callback function acquires the proper private locks before accessing private module data structures and then releases these locks before returning. It is the responsibility of the module writer to cancel any outstanding registered callback routines before the data structures on which the callback routines depend are deallocated and the module closed.

- For hardware device interrupts, this involves disabling the device interrupts.
- Outstanding callbacks from `timeout()` and `bufcall()` must be cancelled by calling `untimeout()` and `unbufcall()`.
- Outstanding callbacks from `esballoc()`, if associated with a particular Stream, must be allowed to complete before the module close routine deallocates those private data structures on which they depend.

Note – The module cannot hold certain private locks across calls to `untimeout()` or `unbufcall()`. These locks are those which the module's `timeout()` or `bufcall()` callback functions acquire. See section “”MT SAFE Modules using Explicit Locks".

Close Race Conditions

Since the callback functions are by nature asynchronous, they may be executing or about to execute at the time the module close routine is called. It is the responsibility of the module writer to cancel all outstanding callback and interrupt conditions before deallocating those data structures or returning from the close routine.

The callback functions scheduled with `timeout()` and `bufcall()` are guaranteed to have been cancelled by the time `untimeout()` and `unbufcall()` return. The same is true for `qtimeout()` and `qbufcall()` by the time `quntimeout()` and `qunbufcall()` return. You must also take responsibility for other asynchronous routines, including `esballoc()` callbacks and hardware as well as software interrupts.

Module unloading and esballoc

The STREAMS framework prevents a module/driver text from being unloaded while there are open instances of the module or driver. If a module does not cancel all callbacks in the last `close` routine it has to refuse to be unloaded.

This is an issue mainly for modules and drivers using `esballoc` since `esballoc` callbacks cannot be cancelled. Thus modules and drivers using `esballoc` have to be prepared to handle calls to the `esballoc` callback free function after the last instance of the module or driver has been closed.

Modules and drivers can maintain a count of outstanding callbacks. They can refuse to be unloaded by having their `_fini()` routine return `EBUSY` if there are outstanding callbacks.

Use of q_next

The `q_next` field in the `queue_t` structure can be dereferenced in `open`, `close`, `put` and `service` procedures as well as the synchronous callback procedures (scheduled with `qtimeout`(), `qbufcall`(), and `qwriter`()).

All other module code, such as interrupt routines, `timeout`() and `esballoc`() callback routines, cannot dereference `q_next`. Those routines have to use the "next" version of all functions, that is, use e.g. `canputnext`() instead of dereferencing `q_next` and using `canput`().

MT SAFE Modules using Explicit Locks

Although we recommend you use MT STREAMS perimeters you have the option of using explicit locks either instead of perimeters or in order to augment the concurrency restrictions provided by the perimeters.

Caution – Explicit locks can not, in general, be used to preserve message ordering in a module due to the risk of reentering the module. Use MT STREAMS perimeters to preserve message ordering.

All four types of kernel synchronization primitives are available to the module writer: mutexes, readers/writer locks, semaphores, and condition variables. Since `cv_wait`() implies a context switch, it can only be called from the module's `open` and `close` procedures, which are executed with valid process context. It is the responsibility of the module writer to use the synchronization primitives provided to protect accesses and ensure the integrity of private module data structures.

Constraints when using locks

When adding locks in a module it is important to observe these constraints:

- Avoid holding module private locks across calls to `putnext`() etc., since the module might be reentered by the same thread that called `putnext`(), causing the module to try to acquire a lock that it already holds. This can cause kernel panic.

- Do not hold module private locks, acquired in `put` or `service` procedures, across the calls to `qprocson()` or `qprocsoff()`. Doing this will cause deadlock, since `qprocson()` and `qprocsoff()` wait until all threads leave the inner perimeter.
- Similarly, do not hold locks, acquired in the `timeout` and `bufcall` callback procedures, across the calls to `untimeout` or `unbufcall`. Doing this will cause deadlock, since `untimeout` and `unbufcall` wait until an already executing callback has completed.

The first restriction makes it very hard to use module private locks as a means of preserving message ordering. MT STREAMS perimeters is the preferred mechanism to preserve message ordering.

Preserving message ordering

Module private locks cannot be used to preserve message ordering, since they cannot be held across calls to `putnext()` (and the other messages that pass routines to other modules). The alternatives for preserving message ordering are:

- Use MT STREAMS perimeters.
- Pass all messages through the `service` procedures. The `service` procedure can drop the locks before calling `putnext()`, `qreply()` etc. without reordering messages, since the framework guarantees that at most one thread will execute in the `service` procedure for a given queue.

The use of perimeters is preferred since there is a performance penalty associated with using `service` procedures.

MT UNSAFE Modules

Most USL DDI/DKI compliant STREAMS drivers and modules can run without any source changes.

Note – This is not recommended nor even 100% applicable, since doing this might jeopardize performance or cause inoperability. These exceptions are usually due to specific implementation issues. It is expected that unsafe modules will run approximately as fast as they would have done on a uniprocessor.

Note – SunOS supports an MT UNSAFE mode for STREAMS modules as an aid in porting modules. This feature should be considered a transition aid and may not be supported in future releases of the operating system. It is strongly recommended that all STREAMS modules and drivers are converted to be MT SAFE.

All MT UNSAFE code within the system runs single threaded, meaning there is no concurrency in the MT UNSAFE code. Only one executing thread is allowed within the MT UNSAFE code at any one time, with the exceptions described below. While the thread executing within the MT UNSAFE code can be preempted at any time, no other thread will be allowed entry into the MT UNSAFE code.

Modifying UNSAFE Drivers

By default, all STREAMS modules and drivers are considered MT UNSAFE unless configured into the system as MT SAFE (“`D_MP`”).

Unsafe drivers run with only the minimum of modification. They run under the general unsafe driver monitor, which implies that at any time, only one processor in the entire system is executing unsafe driver code. Thus, such modules do not gain any performance advantage by being run in a multiprocessor environment. Since these modules hold the mutex lock controlling entry to this monitor, they should not block for long periods, except by calling `sleep`(), which will transparently release and re-acquire the mutex for the caller.

Unsafe drivers are also the only kernel code that can call `sleep`() without catastrophic results. In general, such code will not explicitly block for any other reason other than `sleep`(), since the pre-MT kernel contained no locks.

Some module code cannot run safely as MT UNSAFE. Modules that access data shared by other modules must be converted, unless all other modules sharing such data is themselves unsafe. Also, modules that access safe modules by means other than `putnext`() and the like must be modified. This includes modules that call the `put` procedure of an other module directly.

Caveats

Preemption

The following events will allow the current thread to block, and another thread to enter the MT UNSAFE module, thus preempting the current thread:

- calling another module's `put` procedure via `putnext`, `putctl`, `qreply`, ...
- `sleep`()
- `delay`()
- `strlog`()
- `cmn_err`()

Once entered, a thread within an MT UNSAFE module is allowed to execute until it returns or until it calls one of the these routines. Other threads may have been allowed to execute within the module during the interim between calling one of the above routines and it returning. Consequently, the MT UNSAFE module must be prepared to save state across this preemptable point and revalidate private state information when the routine returns. This is not necessary if the module returns immediately after calling one of the above preemptable routines.

Asynchronous Callbacks

The MT STREAMS framework automatically restricts access into the MT UNSAFE module from all entry points. In addition to the MT UNSAFE module DKI entry points, the framework also blocks asynchronous callback routines entry into the module if a thread is currently active within the module until that thread exits the module. The following sources of asynchronous entry into the MT UNSAFE module are monitored by the framework and are not allowed to preempt a thread executing within the MT UNSAFE module:

- timeout
- bufcall
- esballoc
- software interrupt service routine
- delay

- device hardware interrupt service routine

Just like MT SAFE modules, MT UNSAFE modules have to cancel all outstanding callbacks in their `close` routine. See "Close Race Conditions" on page 298.

Interrupt Handlers

As described earlier, the framework singly threads all MT UNSAFE code within the system. The interrupt service routine is not called by the framework until any thread actively executing MT UNSAFE code within the system has exited the MT UNSAFE code. Therefore the MT UNSAFE module may not spin-wait for a hardware interrupt, since this interrupt handler is not called until the thread exits the module.

Sharing Data Structures

Modules that share some data structure(s) must be configured as either all MT UNSAFE or all MT SAFE. Mixing of module configurations is not allowed, since this would allow entry by multiple threads into the module.

New facilities

MT UNSAFE modules cannot use the regular synchronization primitives (such as mutexes and condition variables). Instead of condition-variables they have to use `sleep`() and `wakeup`().

MT UNSAFE modules cannot use `put(9F).`

Old Facilities

This section describes routines your unsafe module may call and how these translate into the new MT interfaces. Some translations are one-for-one, just using a new call in place of the older one. Others require new ways of viewing the problem and new techniques to solve them.

spl

Traditionally, modules have used the DKI `spl` routines to set the interrupt priority level of a processor to block certain hardware device interrupts. The intent of this was to block hardware interrupt preemption during a particular module operation so that the operation would effectively be atomic.

Prior to SunOS 5.x, only one active thread was allowed within the kernel at any one time. The only form of preemption in a pre-SunOS 5.x kernel came from device interrupts. Therefore using `spl` was a simple and effective method of single-threading in a pre-SunOS 5.x kernel.

In SunOS 5.x this is no longer true. The `spl` routines block only one form of preemption, those that arise directly from device hardware interrupts, and do not prevent preemption by other threads. Since the `spl` routines affect only one of the processors within the MP system, the device interrupt is not masked on other processors within the system. This can allow the hardware interrupt to be taken by any of the other processors in an MP system.

Use of the `spl` routines is restricted to MT UNSAFE modules only. The `spl` routines are not useful for MT UNSAFE drivers since the MT UNSAFE driver's interrupt handler will not be called as long as there is an active thread within the module.

MT SAFE modules should use the MT STREAMS perimeters or mutex, readers-writer, semaphore, and condition variable synchronization primitives instead of `spl` to prevent possible preemption of a non-atomic operation. MT SAFE modules must not call `spl`.

sleep/wakeup

In SunOS 5.x the functionality of `sleep/wakeup` is implemented via condition variables. The replacement for `sleep`() and `cv_signal`() is `cv_wait ()`, while `cv_broadcast`() replace `wakeup`(). See *Writing Device Drivers* for details on using condition variables.

Since only the module `open` and `close` routines have user process context, the `cv_wait`() primitive can be called only from the module `open` and `close` routines. The `cv_signal`() and `cv_broadcast`() primitives can be called by the module at any time since they do not require valid user process context.

Modules that use MT STREAMS perimeters have to use `qwait()` instead of `cv_wait()` in order to allow their `put` or `service` procedures to be called while they are waiting.

Use of the routines `sleep()` and `wakeup()` are restricted to MT UNSAFE modules only, and should not be used my MT SAFE modules. MT SAFE modules should use condition variables or `qwait()` for this purpose.

Sample Multi-threaded Device Driver

Below is a sample multi-threaded, loadable, STREAMS pseudo-driver. The driver MT design is the simplest possible based on using a per module inner perimeter. Thus at most one thread can execute in the driver at any time. In addition, a `qtimeout()` synchronous callback routine is used. Note that the driver cancels any outstanding `qtimeout()` callback by using `quntimeout()` in the close routine. See "Close Race Conditions" on page 298.

Code Example 13-1 Sample Multi-threaded, Loadable, STREAMS Pseudo-Driver

```
/*
 * Example SunOS 5.x multi-threaded STREAMS pseudo device driver.
 * Using a D_MTPERMOD inner perimeter.
 */

#include        <sys/types.h>
#include        <sys/errno.h>
#include        <sys/stropts.h>
#include        <sys/stream.h>
#include        <sys/strlog.h>
#include        <sys/cmn_err.h>
#include        <sys/modctl.h>
#include        <sys/kmem.h>
#include        <sys/conf.h>
#include        <sys/ksynch.h>
#include        <sys/stat.h>
#include        <sys/ddi.h>
#include        <sys/sunddi.h>

/*
 * Function prototypes.
 */
static      int xxidentify(dev_info_t *);
static      int xxattach(dev_info_t *, ddi_attach_cmd_t);
```

```
static      int xxdetach(dev_info_t *, ddi_detach_cmd_t);
stati   int xxgetinfo(dev_info_t *,ddi_info_cmd_t,void *,void**);
**);
static      int xxopen(queue_t *, dev_t *, int, int, cred_t *);
static      int xxclose(queue_t *, int, cred_t *);
static      int xxwput(queue_t *, mblk_t *);
static      int xxwsrv(queue_t *);
static      void xxtick(caddr_t);

/*
 * Streams Declarations
 */
static struct module_info xxm_info = {
    99,                 /* mi_idnum */
    "xx",               /* mi_idname */
    0,                  /* mi_minpsz */
    INFPSZ,             /* mi_maxpsz */
    0,                  /* mi_hiwat */
    0                   /* mi_lowat */
};

static struct qinit xxrinit = {
    NULL,               /* qi_putp */
    NULL,               /* qi_srvp */
    xxopen,             /* qi_qopen */
    xxclose,            /* qi_qclose */
    NULL,               /* qi_qadmin */
    &xxm_info,          /* qi_minfo */
    NULL                /* qi_mstat */
};

static struct qinit xxwinit = {
    xxwput,             /* qi_putp */
    xxwsrv,             /* qi_srvp */
    NULL,               /* qi_qopen */
    NULL,               /* qi_qclose */
    NULL,               /* qi_qadmin */
    &xxm_info,          /* qi_minfo */
    NULL                /* qi_mstat */
};

static struct streamtab xxstrtab = {
    &xxrinit,                   /* st_rdinit */
    &xxwinit,                   /* st_wrinit */
```

```
    NULL,                       /* st_muxrinit */
    NULL                        /* st_muxwrinit */
};

/*
 * define the xx_ops structure.
 */

static          struct cb_ops cb_xx_ops = {
    nodev,                      /* cb_open */
    nodev,                      /* cb_close */
    nodev,                      /* cb_strategy */
    nodev,                      /* cb_print */
    nodev,                      /* cb_dump */
    nodev,                      /* cb_read */
    nodev,                      /* cb_write */
    nodev,                      /* cb_ioctl */
    nodev,                      /* cb_devmap */
    nodev,                      /* cb_mmap */
    nodev,                      /* cb_segmap */
    nochpoll,                   /* cb_chpoll */
    ddi_prop_op,                /* cb_prop_op */
    &xxstrtab,                  /* cb_stream */
    (D_NEW|D_MP|D_MTPERMOD)     /* cb_flag */
};

static struct dev_ops xx_ops = {
    DEVO_REV,                       /* devo_rev */
    0,                              /* devo_refcnt */
    xxgetinfo,                      /* devo_getinfo */
    xxidentify,                     /* devo_identify */
    nodev,                          /* devo_probe */
    xxattach,                       /* devo_attach */
    xxdetach,                       /* devo_detach */
    nodev,                          /* devo_reset */
    &cb_xx_ops,                     /* devo_cb_ops */
    (struct bus_ops *)NULL /* devo_bus_ops */
};

/*
/*
 * Module linkage information for the kernel.
 */
static struct modldrv modldrv = {
```

```
    &mod_driverops,    /* Type of module. This one is a driver */
    "xx",              /* Driver name */
    &xx_ops,           /* driver ops */
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    NULL
};

/*
 * Driver private data structure. One is allocated per Stream.
 */
struct xxstr {
    struct      xxstr *xx_next;/* pointer to next in list */
    queue_t     *xx_rq;        /* read side queue pointer */
    int         xx_minor;      /* minor device # (for clone) */
    int         xx_timeoutid;  /* id returned from timeout() */
};

/*
 * Linked list of opened Stream xxstr structures.
 * No need for locks protecting it since the whole module is
 * single threaded using the D_MTPERMOD perimeter.
 */
static struct xxstr    *xxup = NULL;


/*
 * Module Config entry points
 */

_init(void)
{
    return (mod_install(&modlinkage));
}

_fini(void)
{
    return (mod_remove(&modlinkage));
}

_info(struct modinfo *modinfop)
```

```
{
    return (mod_info(&modlinkage, modinfop));
}

/*
 * Auto Configuration entry points
 */

/*
 * Identify device.
 */
static int
xxidentify(dev_info_t *dip)
{
    if (strcmp(ddi_get_name(dip), "xx") == 0)
            return (DDI_IDENTIFIED);
    else
            return (DDI_NOT_IDENTIFIED);
}

/*
 * Attach device.
 */
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    /*
     * This creates the device node.
     */
    if (ddi_create_minor_node(dip, "xx", S_IFCHR,
            ddi_get_instance(dip), DDI_PSEUDO, CLONE_DEV)
            == DDI_FAILURE) {
                return (DDI_FAILURE);
    }

    ddi_report_dev(dip);
    return (DDI_SUCCESS);
}

/*
 * Detach device.
 */
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
```

```
{
    ddi_remove_minor_node(dip, NULL);
    return (DDI_SUCCESS);
}

/* ARGSUSED */
static int
xxgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg,
    void **resultp)
{
    dev_t dev = (dev_t) arg;
    int instance, ret;

    devstate_t *sp;
    state *statep;
    instance = getminor(dev);

    switch (infocmd) {
    case DDI_INFO_DEVT2DEVINFO:
    if ((sp = ddi_get_soft_state(statep,
        getminor((dev_t) arg))) != NULL) {
            *resultp = sp->devi;
            ret = DDI_SUCCESS;
    } else
             *result = NULL;
    break;

    case DDI_INFO_DEVT2INSTANCE:
            *resultp = (void *)instance;
            ret = DDI_SUCCESS;
            break;

    default:
            ret = DDI_FAILURE;
            break;
    }
    return (ret);
}

static
xxopen(rq, devp, flag, sflag, credp)
queue_t        *rq;
dev_t          *devp;
int            flag;
```

```
int             sflag;
cred_t          *credp;
{
    struct xxstr *xxp;
    struct xxstr **prevxxp;
    minor_t     minordev;

    /*
     * If this Stream already open - we're done.
     */
    if (rq->q_ptr)
            return (0);

    /*
     * Determine minor device number.
     */
    prevxxp = &xxup;
    if (sflag == CLONEOPEN) {
            minordev = 0;
            for (; (xxp = *prevxxp) != NULL;
                    prevxxp = &xxp->xx_next) {
                if (minordev < xxp->xx_minor)
                        break;
                minordev++;
            }
            *devp = makedevice(getmajor(*devp), minordev);
    } else
            minordev = getminor(*devp);

    /*
     * Allocate our private per-Stream data structure.
     */
    if ((xxp = kmem_alloc(sizeof (struct xxstr),
            KM_SLEEP)) == NULL) {
            return (ENOMEM);
    }

    /*
     * Point q_ptr at it.
     */
    rq->q_ptr = WR(rq)->q_ptr = (char *) xxp;

    /*
     * Initialize it.
```

```
     */
    xxp->xx_minor = minordev;
    xxp->xx_timeoutid = 0;
    xxp->xx_rq = rq;

    /*
     * Link new entry into the list of active entries.
     */
    xxp->xx_next = *prevxxp;
    *prevxxp = xxp;

    /*
     * Enable xxput() and xxsrv() procedures on this queue.
     */
    qprocson(rq);

    return (0);
}

static
xxclose(rq, flag, credp)
queue_t         *rq;
int             flag;
cred_t          *credp;

{
    struct      xxstr           *xxp;
    struct      xxstr           **prevxxp;

    /*
     * Disable xxput() and xxsrv() procedures on this queue.
     */
    qprocsoff(rq);
    /*
     * Cancel any pending timeout.
     */
     xxp = (struct xxstr *) rq->q_ptr;
     if (xxp->xx_timeoutid != 0) {
         (void) quntimeout(rq, xxp->xx_timeoutid);
         xxp->xx_timeoutid = 0;
     }
    /*
     * Unlink per-Stream entry from the active list and free it.
     */
```

```
    for (prevxxp = &xxup; (xxp = *prevxxp) != NULL;
         prevxxp = &xxp->xx_next)
             if (xxp == (struct xxstr *) rq->q_ptr)
                     break;
    *prevxxp = xxp->xx_next;
    kmem_free (xxp, sizeof (struct xxstr));

    rq->q_ptr = WR(rq)->q_ptr = NULL;

    return (0);
}

static
xxwput(wq, mp)
queue_t         *wq;
mblk_t          *mp;
{
    struct xxstr   *xxp = (struct xxstr *)wq->q_ptr;

    /* do stuff here */
    freemsg(mp);
    mp = NULL;

    if (mp != NULL)
        putnext(wq, mp);
}

static
xxwsrv(wq)
queue_t         *wq;
{
    mblk_t              *mp;
    struct xxstr        *xxp;

    xxp = (struct xxstr *) wq->q_ptr;

    while (mp = getq(wq)) {
            /* do stuff here */
            freemsg(mp);

            /* for example, start a timeout */
            if (xxp->xx_timeoutid != 0) {
                /* cancel running timeout */
                (void) quntimeout(wq, xxp->xx_timeoutid);
```

```
                }
                xxp->xx_timeoutid = qtimeout(wq, xxtick, (char *)xxp,
                            10);
        }
}

static void
xxtick(arg)
    caddr_t arg;
{
    struct xxstr *xxp = (struct xxstr *)arg;

    xxp->xx_timeoutid = 0;      /* timeout has run */
    /* do stuff */

}
```

Sample Multi-threaded Module with Outer perimeter

Below is a sample multi-threaded, loadable, STREAMS module. The module MT design is a relatively simple one based on a per queue-pair inner perimeter plus an outer perimeter. The inner perimeter protects per-instance data structure (accessed through `the q_ptr` field) and the module global data is protected by the outer perimeter. The outer perimeter is configured so that the `open` and `close` routines have exclusive access to the outer perimeter. This is necessary since they both modify the global linked list of instances. Other routines that modify global data is run as `qwriter`() callbacks giving them exclusive access to the whole module.

```
/*
 * Example SunOS 5.x multi-threaded STREAMS module.
 * Using a per queue-pair inner perimeter plus an outer perimeter.
 */

#include       <sys/types.h>
#include       <sys/errno.h>
#include       <sys/stropts.h>
#include       <sys/stream.h>
#include       <sys/strlog.h>
#include       <sys/cmn_err.h>
#include       <sys/kmem.h>
```

```
#include        <sys/conf.h>
#include        <sys/ksynch.h>
#include        <sys/modctl.h>
#include        <sys/stat.h>
#include        <sys/ddi.h>
#include        <sys/sunddi.h>

/*
 * Function prototypes.
 */
static      int xxopen(queue_t *, dev_t *, int, int, cred_t *);
static      int xxclose(queue_t *, int, cred_t *);
static      int xxwput(queue_t *, mblk_t *);
static      int xxwsrv(queue_t *);
static      void xxwput_ioctl(queue_t *, mblk_t *);
static      int xxrput(queue_t *, mblk_t *);
static      void xxtick(caddr_t);

/*
 * Streams Declarations
 */
static struct module_info xxm_info = {
    99,                 /* mi_idnum */
    "xx",               /* mi_idname */
    0,                  /* mi_minpsz */
    INFPSZ,             /* mi_maxpsz */
    0,                  /* mi_hiwat */
    0                   /* mi_lowat */
};

static struct qinit xxrinit = {
    xxrput,             /* qi_putp */
    NULL,               /* qi_srvp */
    xxopen,             /* qi_qopen */
    xxclose,            /* qi_qclose */
    NULL,               /* qi_qadmin */
    &xxm_info,          /* qi_minfo */
    NULL                /* qi_mstat */
};

static struct qinit xxwinit = {
    xxwput,             /* qi_putp */
    xxwsrv,             /* qi_srvp */
    NULL,               /* qi_qopen */
```

```
    NULL,                  /* qi_qclose */
    NULL,                  /* qi_qadmin */
    &xxm_info,             /* qi_minfo */
    NULL                   /* qi_mstat */
};

static struct streamtab xxstrtab = {
    &xxrinit,                     /* st_rdinit */
    &xxwinit,                     /* st_wrinit */
    NULL,                         /* st_muxrinit */
    NULL                          /* st_muxwrinit */
};

/*
 * define the fmodsw structure.
 */

static         struct fmodsw xx_fsw = {
    "xx",                         /* f_name */
    &xxstrtab,                    /* f_str */
    (D_NEW|D_MP|D_MTQPAIR|D_MTOUTPERIM|D_MTOCEXCL) /* f_flag */
};

/*

/*
 * Module linkage information for the kernel.
 */
static struct modlstrmod modlstrmod = {
    &mod_driverops,    /* Type of module; a STREAMS module */
    "xx module",       /* Module name */
    &xx_fsw,           /* fmodsw */
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modlstrmod,
    NULL
};

/*
 * Module private data structure. One is allocated per Stream.
 */
struct xxstr {
```

```
    struct      xxstr *xx_next;/* pointer to next in list */
    queue_t     *xx_rq;        /* read side queue pointer */
    int         xx_timeoutid;  /* id returned from timeout() */
};

/*
 * Linked list of opened Stream xxstr structures and other module
 * global data. Protected by the outer perimeter.
 */
static struct xxstr    *xxup = NULL;
static int some_module_global_data;


/*
 * Module Config entry points
 */
int
_init(void)
{
    return (mod_install(&modlinkage));
}
int
_fini(void)
{
    return (mod_remove(&modlinkage));
}
int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}


static int
xxopen(rq, devp, flag, sflag, credp)
queue_t         *rq;
dev_t           *devp;
int             flag;
int             sflag;
cred_t          *credp;
{
    struct xxstr *xxp;
    /*
     * If this Stream already open - we're done.
```

```
     */
    if (rq->q_ptr)
               return (0);

    if (sflag != MODOPEN)
        return (EINVAL);

    /*
     * D_MTOCEXCL implies that the open and close routines have
     * exclusive access to the module global data structures.
     */

    /*
     * Allocate our private per-Stream data structure.
     */
    if ((xxp = kmem_alloc(sizeof (struct xxstr),
           KM_SLEEP)) == NULL) {
           return (ENOMEM);
    }

    /*
     * Point q_ptr at it.
     */
    rq->q_ptr = WR(rq)->q_ptr = (char *) xxp;

    /*
     * Initialize it.
     */
    xxp->xx_rq = rq;
    xxp->xx_timeoutid = 0;

    /*
     * Link new entry into the list of active entries.
     */
    xxp->xx_next = xxup;
    xxup = xxp;

    /*
     * Enable xxput() and xxsrv() procedures on this queue.
     */
    qprocson(rq);

    return (0);
}
```

```
static int
xxclose(rq, flag, credp)
queue_t         *rq;
int             flag;
cred_t          *credp;

{
    struct      xxstr          *xxp;
    struct      xxstr          **prevxxp;

    /*
     * Disable xxput() and xxsrv() procedures on this queue.
     */
    qprocsoff(rq);
    /*
     * Cancel any pending timeout.
     */
     xxp = (struct xxstr *) rq->q_ptr;
     if (xxp->xx_timeoutid != 0) {
         (void) quntimeout(WR(rq), xxp->xx_timeoutid);
         xxp->xx_timeoutid = 0;
     }
    /*
     * D_MTOCEXCL implies that the open and close routines have
     * exclusive access to the module global data structures.
     */
    /*
     * Unlink per-Stream entry from the active list and free it.
     */
    for (prevxxp = &xxup; (xxp = *prevxxp) != NULL;
         prevxxp = &xxp->xx_next)
               if (xxp == (struct xxstr *) rq->q_ptr)
                      break;
    *prevxxp = xxp->xx_next;
    kmem_free (xxp, sizeof (struct xxstr));

    rq->q_ptr = WR(rq)->q_ptr = NULL;
    return (0);
}

static int
xxrput(wq, mp)
queue_t         *wq;
```

```
mblk_t          *mp;
{
    struct xxstr   *xxp = (struct xxstr *)wq->q_ptr;

    /*
     * Do stuff here. Can read "some_module_global_data" since we
     * have shared access at the outer perimeter.
     */
    putnext(wq, mp);
}

/* qwriter callback function for handling M_IOCTL messages */
static void
xxwput_ioctl(wq, mp)
queue_t         *wq;
mblk_t          *mp;
{
    struct xxstr   *xxp = (struct xxstr *)wq->q_ptr;

    /*
     * Do stuff here. Can modify "some_module_global_data" since
     * we have exclusive access at the outer perimeter.
     */
    mp->b_datap->db_type = M_IOCNAK;
    qreply(wq, mp);
}

static
xxwput(wq, mp)
queue_t         *wq;
mblk_t          *mp;
{
    struct xxstr   *xxp = (struct xxstr *)wq->q_ptr;

    if (mp->b_datap->db_type == M_IOCTL) {
        /* M_IOCTL will modify the module global data */
        qwriter(wq, mp, xxwput_ioctl, PERIM_OUTER);
        return;
    }
    /*
     * Do stuff here. Can read "some_module_global_data" since we
     * have shared access at the outer perimeter.
     */
    putnext(wq, mp);
```

```
}

static
xxwsrv(wq)
queue_t         *wq;
{
    mblk_t              *mp;
    struct xxstr        *xxp;

    xxp = (struct xxstr *) wq->q_ptr;

    while (mp = getq(wq)) {
        /*
         * Do stuff here. Can read "some_module_global_data" since
         * we have shared access at the outer perimeter.
         */
        freemsg(mp);

        /* for example, start a timeout */
        if (xxp->xx_timeoutid != 0) {
            /* cancel running timeout */
            (void) quntimeout(wq, xxp->xx_timeoutid);
        }
        xxp->xx_timeoutid = qtimeout(wq, xxtick, (char *)xxp,
                                     10);
    }
}

static void
xxtick(arg)
    caddr_t arg;
{
    struct xxstr *xxp = (struct xxstr *)arg;

    xxp->xx_timeoutid = 0;      /* timeout has run */
    /*
     * Do stuff here. Can read "some_module_global_data" since we
     * have shared access at the outer perimeter.
     */
}
```