Symmetric Multiprocessing in Solaris 2.0

Steven Kleiman, Jim Voll, Joe Eykholt, Anil Shivalingiah, Dock Williams,
Mark Smith, Steve Barton, Glenn Skinner

SunSoft Inc.
Mountain View, California

Abstract

SunOS 5.0 is the operating system component of the Solaris 2.0 environment. SunOS 5.0 is a dynamically loaded, fully preemptable, real-time, symmetric multiprocessing kernel that supports multiple threads of control within a single application process. The kernel not only runs equally on all processors within a tightly coupled shared memory multiprocessor, but in addition, it has sufficient locking so that kernel processing can efficiently utilize multiple processors. We describe the overall architecture of the SunOS 5.0 kernel and the locking strategy of several major kernel subsystems to illustrate the concurrency available.

1: SunOS 5.0 overview

SunOS 5.0 is the operating system component of the Solaris 2.0 environment. It is composed of separate modules, such as device drivers, file systems, and individual system calls which are dynamically loaded into the kernel on an as-needed basis. The core of SunOS 5.0 is a real-time nucleus that supports kernel threads of control (threads). All control flows are threads, including interrupts. The kernel is preemptable. The few non-preemption points (e.g. thread switch) are of bounded length. Kernel threads are also used to support multiple threads of control, called lightweight processes (LWPs) within a single UNIX process [5]. Kernel threads are dispatched in priority order on the pool of available processors. The overall architecture is shown in Figure 1.

The threads programming model in the kernel is similar to threads in a user program. The relevant analogy is: a user thread is executed by a LWP in much the same way as a kernel thread is executed by a processor.

There is one kernel thread for each LWP. This kernel thread is used to run in the kernel when the LWP performs a system call. Other kernel threads do not have LWPs associated with them and are used for various purposes. For example; handling interrupts, executing STREAMS service procedures, or providing NFS service. Since the kernel is fully preemptable many of the same state transitions occur running on a uniprocessor as running on a multiprocessor.

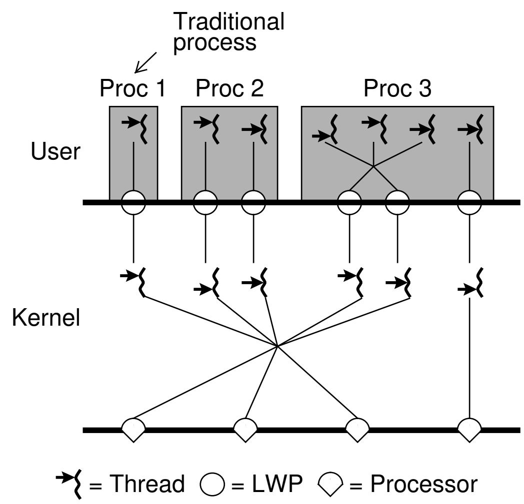

Figure 1: SunOS 5.0 architecture

`Synchronization`

Kernel threads synchronize via a variety of synchronization primitives, such as:

- Mutual exclusion (mutex) locks
- Condition variables
- Counting semaphores
- Multiple readers, single writer (readers/writer) locks

These primitives are very similar to the equivalent user thread primitives.

The mutex and writer locks support a priority inheritance protocol [3] which prevents lower priority threads from blocking higher priority threads (priority inversions).

1.1: Interrupts

Interrupts cause the current thread to be preempted and an interrupt thread to be dispatched. The interrupt trap code is responsible for ensuring that the interrupted thread is correctly passivated, the processor is at the correct interrupt level, and the correct vector to the second level interrupt handler (device interrupt handler) is called on a new thread stack.

In many implementations, there will be some interrupt level (usually the clock interrupt level) above which interrupts are not handled as threads. These interrupts are usually the shortest and most time critical ones (i.e. firmware). The interrupt handlers at these levels are machine dependent and use machine dependent mechanisms to synchronize. Some possible mechanisms are; circular buffers, redispatching the interrupt at a lower level, and spin locks with appropriate processor interrupt level changes.

Further details of the kernel threads implementation can be found in [6]. Further details of the real-time aspects of SunOS 5.0 can be found in [3].

2: Symmetric multiprocessing

SunOS 5.0 is intended to run on uniprocessors and tightly coupled shared memory multiprocessors. The kernel currently assumes all processors are equivalent. Processors select kernel threads from the queue of runnable kernel threads. If a particular multiprocessor implementation places an asymmetric load on the processors (e.g. interrupts) the kernel will nonetheless schedule threads to processors as if they were equivalent.

In general, all processors see the same data in memory. This model is relaxed, somewhat, in that memory operations issued by a processor may be delayed or reordered when viewed by other processors. In this environment, shared access to memory must be protected by synchronization primitives. The exception is that single, primitive data items may be read or updated atomically (e.g. all the bytes in an `int` change at the same time).

The shared memory is assumed to be symmetrical. Thus the kernel currently does not ensure that processes scheduled on a particular processor are placed in a particular piece of memory that is faster to access from that processor.

It is possible for a kernel to run "symmetrically" on a multiprocessor yet not allow more than one processor to execute kernel code. This is clearly not a strategy that scales well with increasing numbers of processors. SunOS 5.0 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. In general, access to data items are protected by locks as opposed to entire routines. Infrequent operations are usually coarsely locked with simple mutual exclusion. Overall, SunOS 5.0 has several hundred distinct synchronization objects statically, and can have many thousands of synchronization objects dynamically. The remainder of this paper discusses the concurrency goals and locking strategy of several major kernel subsystems.

3: Virtual memory

The virtual memory system consists of four subsystems: address spaces, segment drivers, the page cache, and the hardware address translation (`hat`) layer [2]. An address space is subdivided into a set of segments each of which represents a chunk of virtual memory. The main segment type is called `segvn`. `Segvn` segments are fully pageable and are used to memory-map files (`vnodes` [4]).

Physical pages become associated with segments through page faults. If a segment is backed by a `vnode` pointer, then the `vnode`'s `get_page()` operation will find the page in the page cache and make it available to the segment. Otherwise an empty page is obtained from the page cache and an I/O operation is started to bring in the appropriate data. The segment will call the `hat` layer to set up an MMU translation for this page.

The `hat` layer manages the hardware tables that translate a virtual address within an address space to a physical address. The address space contains a pointer to all of its hardware translations.

3.1: Concurrency

Page faults within an address space or among different address spaces may happen in parallel. Mapping and unmapping memory (`mmap()` and `munmap()`) can proceed in parallel between different address spaces. The page cache may be searched in parallel, but page cache misses and page cache free list operations proceed serially. Hardware address translation maintenance proceeds serially.

3.2: Address spaces and segments

A readers/writer lock is associated with each address space. The read lock prevents segments from being added or deleted, and prevents the address space from disappearing. This lock doesn't prevent the contents of the address space structure from changing. A mutex lock per address space is used to protect the contents of the address space structure. The readers/writer lock on the address space does not prevent changes to the segments in the address space.

When a fault occurs, the address space is read locked. This prevents the segment where the fault happened from disappearing. Instead the page is looked up in the page cache. If the page isn't in the cache, then a new page is allocated and marked "in transit". Subsequent faults to this page will block waiting for the I/O to complete.

3.3: Page cache

The page cache consists of physical pages that are associated with a `vnode` and a offset. The `vnode` and offset pair is referred to as a page's identity. A page can hold onto its identity by acquiring a shared version of a shared/exclusive lock. This is very similar to a readers/writer lock except that it is implemented with higher level locks so that the page structure itself can be freed even when a thread is waiting for it. Unlocked pages are reusable.

A page with an identity is also attached to a list associated with its `vnode`. When a page is reused, the old identity must be discarded. This requires the page to be removed from the cache and the `vnode` list.

A readers/writer lock, called `page_idlock`, is used to protect the name (`vnode` plus page offset within the `vnode`) of each page, the hash list for looking up a page given its name, and the list of pages associated with each `vnode`. The write version of the lock is taken whenever the identity of a page must change. The read version of the lock is taken when searching for a page given its name. This allows cache hits to be serviced in parallel. A cache miss, however, write locks the `page_idlock`. This causes access to the page cache to be single threaded up until the write lock is released.

A free list of unreferenced pages or pages with no identity is also maintained. A page is allocated from the free list when a cache miss happens. A page is placed on the free list when a page loses its identity. A single mutex protects the contents of the page cache's free list.

3.4: Hardware address translation

The hardware address translation (`hat`) layer has been extended to allow multiple MMUs to map system pages. Each MMU in the system has its own driver. A single, global mutex is used to maintain consistency in the `hat` layer, including all the MMU drivers. This mutex also protects the fields in the address space and page structure that the `hat` layer maintains.

3.5: Page out daemon

The page out daemon consists of two kernel threads. One thread scans physical pages looking for unused pages. When the scanning thread finds an unused unmodified page, it frees it. If the page has been modified (dirty), it is placed on a list of pages to be cleaned. The second thread reads this list and starts the I/O to copy the modified data to disk. When the data has been written, the page is placed on the free list.

The use of the second thread allows scanning to continue even when page cleaning must wait due to lack of some resource. The page out daemon threads only runs when the amount of free pages declines below a threshold value.

4: Drivers

SunOS 5.0 supports two types of drivers; MT-unsafe and MT-safe. In this context, the term "MT-safe" means a driver that has been explicitly modified to protect itself against possible multiple threads of execution within the driver. All old drivers are MT-unsafe.

4.1: MT-unsafe drivers

MT-unsafe drivers are restricted to execute on one processor in the system at any one time. All MT-unsafe drivers share a single mutual exclusion lock, `unsafe_driver`. The lock is acquired before any unsafe driver entry point, including interrupt handlers, is called. Mutual exclusion between the driver and the interrupt handler is accomplished via this mutex.

A suite of compatibility routines, such as `sleep()`, and `wakeup()`, is provided which emulates routines in the non-multi-threaded kernel. Some of these routines may actually cause an interrupt handler thread to block on an internal system mutex, but this is not noticed by the driver.

4.2: MT-safe drivers

MT-safe drivers explicitly protect themselves from reentrance via the normal MT synchronization mechanisms. The interrupt handlers for MT-safe drivers are run as separate threads and use the mutual exclusion mechanisms (i.e. they can block). Thus, MT-safe drivers should not explicitly raise the processor interrupt priority level, as doing so does not lock out interrupts serviced on other processors.

MT-safe drivers are responsible for all mutual exclusion. To allow the drivers maximum flexibility, there are no implicit calls to mutual exclusion primitives. For example, drivers can supply a mutex for each minor device. Thus the degree of concurrency available is determined by the device driver itself.

Drivers that are critical for system performance have been converted to be MT-safe.

4.3: Mutual exclusion and interrupts

The device interrupt handler and upper level driver may use normal MT mutual exclusion primitives to prevent multiple entry to critical sections. If the interrupt handler finds a mutex lock held, it will block. Note that if the interrupt thread attempting the lock interrupts the thread holding the lock, the interrupt thread will block and allow the interrupted thread to run. Priority inheritance still works in this situation, so threads holding locks required by an interrupt handler will inherit a very high priority. This scheme allows MT synchronization to work even in the uniprocessor case. Also, there is little or no need for a driver to manipulate the processor interrupt level.

4.4: Device driver interface (DDI)

The kernel presents a SVR4 ES/MP Driver/Kernel Interface (DKI) [7] compliant interface to drivers. The synchronization primitives supplied in this interface are intended for a kernel that does not treat interrupts as threads. The interface has a variety of mutual exclusion primitives that manipulate the processor interrupt level and/or spin and/or sleep. These primitives map directly to the single SunOS 5.0 mutual exclusion primitive and any processor interrupt level information is ignored.

5: STREAMS

In this document, the kernel data structures, system call interface code, and utility routines implementing STREAMS are referred to as the STREAMS f*ramework*. The goal of the framework design is to allow multiple processors to be applied to a single STREAMS module for separate Streams (connections) and to multiple modules within the same Stream. These types of concurrency are referred to as horizontal and vertical parallelism [1]. Both types are supported.

The design supports modules that have been explicitly made MT-safe, modules that have not been made MT-safe, and two additional concurrency paradigms in which the framework supplies some automatic locking. Non-critical STREAMS modules can run with little or no change, at the cost of concurrency, using either the unsafe or one of the automatic locking paradigms. Critical STREAMS drivers either use the automatic locking or use explicit finer grain locking to get maximum performance.

5.1: STREAMS service scheduling

The STREAMS framework dedicates a small number of threads to executing the queue of scheduled STREAMS service routines. These threads have priority lower than that of a thread executing an interrupt, or a system call on behalf of a user, but higher than that of most non-real-time threads running in user mode. Since service routines do not sleep, *ncpu* background service threads (where *ncpu* is the number of actual processors in the system), represent the maximum possible amount of parallelism that the system can apply to service processing.

The use of multiple threads for STREAMS service procedures allows several STREAMS modules to execute concurrently as well as allowing several independent Streams within a STREAMS module to execute concurrently. Service routines are guaranteed that requests are not reentered.

5.2: MT-safe STREAMS modules

Streams modules are similar to device drivers. Certain properties of existing uniprocessor STREAMS, such as atomicity of writes, and ordering of messages in a Stream, rely upon the implicit kernel monitor. In a multi-threaded kernel, however, races are possible where they were not before. These races must be prevented by use of the thread synchronization and locking mechanisms, where these properties (as well as data consistency) are to be preserved.

The measures required to make a STREAMS module MT-safe depend to some degree on the specific protocol that the module implements. For instance, some modules do not care if messages become disordered, since the messages have already been stamped with a sequence number, and will be re-assembled into their sequence by some other module. Other modules must maintain the order of data that pass into them (e.g. tty drivers). If more than one thread is in the process of delivering a message on a Stream it is impossible to say which thread will deliver its message first unless explicit synchronization is imposed, since each thread is progressing at a rate that is unknown with respect to any other thread. Mutex locks do not preserve the order in which threads attempt the mutex. Therefore the order of the messages the threads are carrying through the Stream must be preserved by some extrinsic data structure, such as the module's queue, or by single-threading with some higher level lock.

Other kinds of conversion problems are generic to all protocols. For example, if a lock is purely local to a module instance, it should not be held when the module is exited. Exiting the module includes calling into the succeeding module. The danger here is that a STREAMS module putting messages to a succeeding module may be reentered when successive modules echo, as shown in Figure 2. This recursive entry into the earlier module needs to be able to acquire any local locks to complete its processing without danger of deadlock.

5.3: Unsafe STREAMS modules

MT-unsafe STREAMS modules run with minimal source code change. However, only one thread in the entire system can execute such code at any one time. When the framework detects that an unsafe module is about to be entered it acquires the `unsafe_driver` mutex. Similarly, when an unsafe module returns or calls info a safe module, the framework drops the `unsafe_driver` mutex.

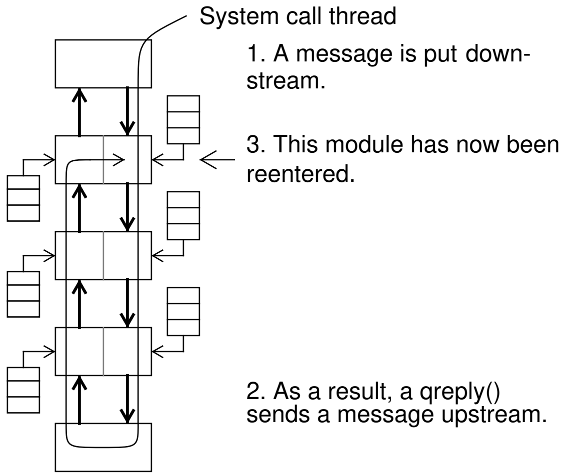

Figure 2: Reentering STREAMS modules

5.4: Queue-pair safe STREAMS modules

Some STREAMS modules have no shared data between instances of a Stream, since each instance is independent. However, within a single instance, data is shared between the input and output Streams. These modules are considered queue-pair safe. The framework can automatically acquire a lock associated with both the input and output queues for the module instance whenever the module is entered. This allows multiple processors to execute module code for different Streams. The processors will serialize whenever more than one tries to enter a particular module instance either from the input or output sides.

The line discipline processing module, `ldterm`, uses this concurrency paradigm. The input and output sides of `ldterm` must communicate, for example, to do echo processing.

Even when the module shares some data between instances, the framework can handle the bulk of the locking automatically while the module handles only those cases where data is shared.

5.5: Queue safe STREAMS modules

Some STREAMS modules not only do not share data between instances of a stream, but also share no data between the input and output sides of a single stream instance. These modules are considered queue safe, and the framework can automatically acquire a lock associated with each module queue. This allows multiple processors to execute module code for different Streams and on the input and output sides of a single instance. The processors will serialize whenever more than one tries to enter a particular module instance from the input side or more than one tries to enter from the output sides.

The pseudo-terminal packet module, `pckt`, uses this concurrency paradigm. In `pckt`, the input and output processing is completely autonomous.

6: BSD fast file system (UFS)

The BSD 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 VM system. Write operations or directory operations are processed in parallel only for different files.

6.1: Inodes

Each file or directory is represented in memory by an in-memory . A readers/writer lock, `i_rwlock`, protects the data and attributes of an `inode`. A readers lock is taken whenever file data is read through the `read()` system call, directories are searched or read, or attributes (e.g. file size) are read. A writer lock is taken whenever a file is written through the `write()` system call or directories or file attributes are to be modified. This provides atomic updates with respect to these operations, as required by UNIX semantics. Thus all `read()` and `write()` system calls to a particular file are blocked while a `write()` system call is in progress. Multiple `read()` system calls can happen in parallel. Operations to memory-mapped files are not protected by `i_rwlock`, and may cause data to change or I/O operations to happen on a file, while `i_rwlock` is held.

A second readers/writer lock, `i_contents`, is used by most UFS `vnode` entry points to protect the contents of the in-memory `inode` and any indirect blocks, but excluding file data and the access time field.

The `inode` accessed, modified, and changed times, and the `inode` flags are protected by a per-`inode` mutex lock, `i_tlock`. This lock is used for short periods to synchronize updates to the accessed time even though the `inode` has several active readers.

6.2: Recursive vnode operation entry

UFS uses the virtual memory (VM) system to cache and manipulate file data. In some cases, this requires the VM system to reenter UFS while servicing a UFS request. The problem with this circularity is that it is possible to end up calling some `vnode` entry point routines in UFS already holding an `inode` `i_contents` lock.

The solution to this problem is to unlock `i_contents` whenever a page fault may occur in the UFS code and reacquire the lock afterwards. Since another file system operation may have occurred during the time the lock was dropped, any relevant state information at the time the lock was dropped must be reevaluated after re-obtaining the lock (if necessary).

6.3: Directory manipulation

Directory lookup is done under a readers lock on `i_rwlock` and directory changes are done under a writer lock on `i_rwlock`.

6.4: Inode cache

The `inode` cache holds `inodes` representing the most frequently used files, directories, and devices. It is used to look up particular `inodes` given the `inode` number. During `inode` lookup, if the desired `inode` is not found in the cache, a free or least recently used `inode` is reclaimed and the desired `inode` is fetched from disk into the new `inode` structure. A free list of least recently used `inodes` is maintained so that stale `inodes` can be recycled.

A readers/writer lock, `icache_lock`, is used to protect the `inode` cache. This lets multiple readers search in parallel. The cache, however, can only be modified when holding a writer lock. The free list is protected by a single mutex lock, `ifree_lock`.

When searching for an `inode` a readers version of `icache_lock` is taken. If the `inode` is found it is marked as referenced. If the `inode` is not found, the free list is locked to get an unused `inode`. If no unused `inode` exists a new one is allocated.

6.5: Super blocks and cylinder groups

Most of the data in the super block is read only and does not require a lock to access it. However the fields that have to do with changeable file system state (e.g. modified, summary information) are protected by the `vfs_lock` of the associated VFS structure.

A cylinder group structure contains bit maps for all the free `inodes` and blocks associated with a group of cylinders. The cylinder group information is also protected by `vfs_lock`.

6.6: Asynchronous writes

Many of the file systems in SunOS 5.0 employ separate threads to handle asynchronous write requests. Most of the processing time associated with a write request is spent gathering dirty pages, setting up the I/O request, and obtaining locks. This work can be deferred to the file system asynchronous write threads, allowing the thread that issued the write to return sooner. On a multiprocessor this has the additional advantage that another processor can service the asynchronous write requests while the generating thread continues to run.

7: References

[1] A. Garg. "Parallel STREAMS: a Multiprocessor Implementation", Proceedings of the Winter 1990 USENIX Conference.

[2] R.A. Gingell, J.P. Moran, W.A. Shannon. "Virtual Memory Architecture in SunOS", Proceedings of the Summer 1987 USENIX Conference.

[3] S. Khanna, M. Sebree, J. Zolnowski. "Realtime Scheduling in SunOS 5.0", accepted, Winter 1992 USENIX Conference.

[4] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", Proceedings of the Summer 1986 USENIX Conference.

[5] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks. "SunOS Multi-thread Architecture", Proceedings of the Winter 1991 USENIX Conference.

[6] SunSoft Inc. "Multithreading Techniques used in the SunOS 5.0 Kernel", To be submitted, Summer 1992 USENIX Conference.

[7] UNIX System Laboratories. "UNIX System V Release 4 ES/MP Multiprocessing Detailed Specifications".