File System Multithreading in System V Release 4 MP

J. Kent Peacock – Intel Multiprocessor Consortium

ABSTRACT

An Intel-sponsored Consortium of computer companies has developed a multiprocessor version of System V Release 4 (SVR4MP) which has been released by UNIX System Laboratories. The Consortium's goal was to add fine-grained locking to SVR4 with minimal change to the kernel, and with complete backward compatibility for user programs. To do this, a locking strategy was developed which complemented, rather than replaced, existing UNIX synchronization mechanisms.

To multithread the file systems, some general locking strategies were developed and applied to the generic Virtual File System (VFS) and vnode interfaces. Of particular interest were the disk-based *S5* and *UFS* file system types, especially with respect to their scalability. Contention points were found and successively eliminated to the point where the file systems were found to be disk-bound. In particular, several file system caches were restructured using a low-contention, highly-scalable approach called a *Software Set-Associative Cache.* This technique reduced the measured locking contention of each of these caches from the 10-15% range to less than 0.1%.

A number of experimental changes to disk queue sorting algorithms were attempted to reduce the disk bottleneck, with limited success. However, these experiments provided the following insight into the need for balance between I/O and CPU utilization in the system: that attempting to increase CPU utilization to show higher parallelism could actually lower system throughput.

Using the *GAEDE* benchmark with a sufficient number of disks configured, the kernel was found to obtain throughput scalability of 88% of the theoretical maximum on 5 processors.

Introduction

The goal of the Consortium assembled by Intel was to produce a multiprocessor version of System V Release 4 in as short a time as possible. As such, there was no inclination to perform a radical restructuring of the kernel, nor to support user-level threads while the evolution of a consensus threads standard was incomplete.

There were two main performance goals for this effort: binary performance running the *GAEDE* benchmark [7] with respect to the uniprocessor system should degrade by no more than 5%; and the proportional increase of throughput should be at least 85% of the first processor for each processor brought online, up to 6 processors. The performance numbers were arrived at through negotiations with UNIX System Laboratories as their acceptance criteria. The 6 processor limit arose from considerations of how well the implementation was expected to scale, namely between 8 and 16 processors, and, more importantly, of how large a system was likely to be available for testing.

Previous Work

Many attempts have been made to adapt UNIX to run on multiprocessor machines. Companies such as AT&T, Encore, Sequent, NCR, DEC, Silicon Graphics, Solbourne and Corollary have all offered multiprocessor UNIX systems, though not all have described the changes made to the operating system in the literature. Bach [2] describes a multiprocessor version of System V released by AT&T. Encore has described several generations of their multithreading effort, first on MACH and then on OSF/1 in a series of papers [4, 11, 12, 15]. This work has the greatest similarity to that reported here, so an attempt has been made to make relevant comparisons to it throughout the paper. DEC has also published papers on their approach to multithreading their BSD-derived ULTRIX system [10, 21]. Ruane's paper on Amdahl's UTS multiprocessing kernel is the best precedent to the philosophy of the approach used by the Consortium [20]. Lately, NCR has discussed their parallelization efforts on System V Release 4 [5, 6]. As NCR has participated as a member of the Intel Consortium, this work influenced the Consortium's approach.

Locking Model

Multiprocessor locking implementations can be divided roughly into two camps: those who replaced the traditional *sleep-wakeup* UNIX synchronization with semaphores or other locking primitives, and those who opted to retain the *sleep-wakeup*

synchronization model and add mutual exclusion around the appropriate critical sections. Bach's [2] implementation is of the first type, with existing synchronization mechanisms replaced by Dijkstra semaphores. Ruane's paper [20] represents a sort of canonical description of the type of locking described most often by previous multithreaders using the mutual exclusion approach. The paper gives a very good discussion of some subtle synchronization issues relative to a pre-SVR4 environment, as well as some sound arguments against using Dijkstra semaphores.

Although a thorough description of the Consortium locking protocols is beyond the scope of this paper, there are a number of key features of the locking primitives which should be noted: Firstly, mutual exclusion locks are implemented so that any mutex locks held by a process are automatically released and reacquired across a context switch. This is an imitation of the implicit uniprocessor locking achieved by holding the processor in a non-preemptive kernel. It is necessary for proper *sleep-wakeup* operation for a lock protecting a sleep condition to be released after the sleeping process has established itself on the sleep queue. Most previous efforts have enhanced the *sleep* function to release a single mutex lock, usually specified as an extra argument to the sleep call. With automatic releasing of locks across context switches, sleep calls need not be changed. This means that much of the multithreading effort merely involves adding mutex locks around sections of code, even though they may sleep. Comparisons of many multithreaded files with the originals show this to be literally the only change required. This is an important feature when it is necessary to occasionally upgrade to ongoing releases of the underlying uniprocessor system.

The ability to release all of the locks acquired in the call stack relates to another feature of the locks, namely that recursive locking of each individual lock is allowed. SVR4 is designed in such a way that there are a number of object-oriented interfaces between sub-systems of the kernel, derived from SunOS [8]. These subsystems call back and forth to one another and create surprisingly deep recursive call stacks. (This happens particularly between virtual memory and file system modules.) Though providing considerable modularity, this feature makes it very difficult to establish assertions about which mutex locks might be held coming into any given kernel function. Allowing lock recursion and the automatic release of mutex locks allows a given function to deal only with its own locking requirements, without having to worry about which locks its callers hold, aside from deadlock considerations.

The primitives were also designed to be able to configure whether the caller spins or gives up the processor when the lock is not available. Locks acquired by interrupt routines must spin, whereas locks of the same class which might deadlock one another can avoid deadlock by sleeping. The deadlock avoidance comes from the fact that held locks are released when a lock requester sleeps. In addition, mutex locks may be configured as shared/exclusive locks, allowing multiple reader locks or a single writer lock to be held.

For purposes of the following discussion, it is useful to carefully define the difference between a *resource lock* and a *mutex lock*. Logically, a *resource lock* is a lock which protects a resource even when the locker is not running on a CPU. A *mutex lock*, on the other hand, only protects a resource when the locker is running on a CPU. Stated another way, a resource lock may be held across context switches, while a mutex lock would not be. A resource lock can be an actual locking primitive, such as a semaphore [2], but need not be. Implementing a resource lock as locking data protected by a mutex has been shown by Ruane to provide greater flexibility in locking than the semaphore approach [20]. It should be noted that there are resource locks already present in the uniprocessor code, for example, the B_BUSY flag bit in a disk buffer cache header or the ILOCKED flag in an inode structure. Protecting manipulations of these resource locks with mutexes is usually sufficient to generalize them to work on a multiprocessor. In fact, all of the instances of a given type of resource lock can be protected using a single mutex lock rather than a lock per instance, typically with very low contention on the mutex.

More detailed descriptions of the locking primitives and more detailed arguments in favor of using them over semaphores can be found elsewhere [5, 17].

File Systems Locking

The SVR4 file system implementation centers around two separate object-oriented interfaces which originated in SunOS [9]. One is the Virtual File System (VFS) interface, which consists of functions which perform file system operations, for example, mounting and unmounting. The other is the virtual node (vnode) interface, which allows operations on instances of files which reside in a virtual file system.

Most of the file system multithreading effort was spent developing a general multithreading model for these two interfaces. Although there are around 10 file system types in SVR4, only the two disk-based file systems are discussed: the UNIX File System (UFS) type, which is a derivative of the Berkeley Fast File System [13] via SunOS, and the S5 type, which is derived from the System V Release 3 file system. The strategies for multithreading these two file systems were basically the same.

VFS Locking

The VFS layer of the file system provides the support for an object-oriented interface to the various file system types available. This support includes maintaining the list of mounted file systems and resolving races between file system unmounts and pathname searches into a file system being unmounted. The multithreading of the VFS layer centers around a shared/exclusive mutual exclusion lock which serializes access to mount points during pathname lookup, mount and unmount operations. This lock protects the existing uniprocessor resource lock on each virtual file system. This approach seems to be essentially the same as that used by Encore in their multithreading of the Mach vnode-based file system [11].

Vnode Locking

The vnode layer of the kernel implements another object-oriented interface for operations on each file within a virtual file system of a given type. The focal point of these operations is the *vnode* structure, which is usually contained within a VFS-dependent node structure for each active vfs element in the system. The object-oriented interface consists of a number of macros prefixed with "VOP_" which call through a VFS-dependent function code table. With only a few exceptions, these macro calls represent the only path into the file system code. The interface was designed to support a set of operations which were atomic and mostly stateless, in order to support a stateless network file system, namely NFS [9]. In actual implementation, local file systems do retain some state necessary to implement normal UNIX file system semantics. For example, in a UFS or S5 filesystem, a VOP_LOOKUP operation, which does one stage of a pathname lookup, leaves the found file or directory in a locked state on return.

The granularity of locking desired at the vnode level is the individual vnode, which suggests a locking strategy whereby a lock on a vnode is obtained and released around almost every VOP call. Using this strategy provides a blanket level of essentially automatic vnode locking for most of the VOP functions.

Inode Locking

There are two main aspects to file system locking inside UFS and S5: locking of a given file inode while some operation is performed on it, and the locking of the file system inode cache during the lookup or freeing of an inode. Inode cache locking is discussed in a later section devoted to cache contention.

The per-inode lock is a resource lock which can remain held across blocking operations, such as disk I/O. The original uniprocessor implementation of the inode locking uses two separate lock flag bits in the inode, ILOCKED and IRWLOCKED. (Pre-SVR4 systems had only the ILOCKED flag.) The ILOCKED flag locks the inode during most operations which would access or change the contents of the inode itself, whereas the IRWLOCKED flag makes read or write operations atomic. This allows a long read or write operation to proceed without blocking a *stat* operation, for example. These locks are set and cleared by the ILOCK – IUNLOCK and IRWLOCK – IRWUNLOCK macro pairs. Both the ILOCKED and IRWLOCKED bits can be held at the same time by two different processes. In a multiprocessor, the processes must be prevented from running at the same time for correct emulation of the uniprocessor semantics. In order to accomplish this, the vnode lock must be held concurrently with each of the two resource lock flags. This is done by acquiring the vnode lock from within ILOCK and IRWLOCK and releasing it within IUNLOCK and IRWUNLOCK. The effect of this on return from a VOP function which does an ILOCK without an IUNLOCK, due to lock recursion, is to leave the vnode locked outside the VOP function. For example, the previously mentioned VOP_LOOKUP function returns with both ILOCKED set and the vnode lock held on the found directory or file. If the process does a context switch after the return, the vnode lock is released, but the resource lock, embodied in the flag bit, is not. Hence, the integrity of the inode is still protected, although a process holding the IRWLOCKED lock could run after acquiring the vnode lock. This behavior is the same as in the uniprocessor system. When the next VOP call is made which does the matching IUNLOCK on the inode, the extra vnode lock on the file is released, and the vnode is unlocked completely on return from the call.

This locking approach represents a different philosophy from the OSF/1 file system [12], where by design, no file system locks can be held on return from a VOP function. Since the uniprocessor model is preserved by our strategy, no additional races are introduced such that an error path could cause the lock not to be released. The only way for the vnode lock not to be released is for the inode lock not to be released, which would also qualify as a uniprocessor bug. Having the vnode lock visible at the VOP interface also allows the lock to be held around multiple VOP calls, which is made possible by the recursive property of the Consortium mutex locks. This is used in several cases to avoid a race with file locking on a vnode.

Because there is a lock on each vnode, the potential for deadlock exists when it is necessary to lock more than one vnode, as during pathname searches or some STREAMS operations. The solution to this problem is to have the lock *sleep* when it waits for a busy vnode lock. As previously discussed, this removes the deadlock possibility because all of the locks held by the locker are released. The one problem that this causes is that potential context-switches are introduced which were not

present before the vnode locking was added. One solution to this is to widen the scope of the vnode locking so that the preemption happens in a safe place. Locking around the VOP calls has been found to be sufficiently wide for almost all problem situations, since callers should conservatively assume that any VOP call might sleep. Additional vnode locking which is bound with inode lock manipulations is also safe, because the inode locking itself may sleep. The only place where the vnode lock's possible sleeping causes a problem is in the *iget* function where the inode has been found by a cache lookup. A sleep to wait for the vnode lock could allow the identity of the inode to change. The solution is simply to recheck after the vnode is locked that the inode obtained still matches the identity of the one searched for.

When an inode is heavily shared by many processes, the processes tend to queue up sleeping to wait for the inode lock. This represents an example of what has been called the *thundering herd* problem [5]. When an inode lock is released, the normal *wakeup* function makes all of the processes sleeping on the inode runnable. In a multiprocessor, this results in all but one of the processes finding the lock busy and going back to sleep. This takes $O(N^2)$ CPU time to process N sleepers. To alleviate this, a *wake1proc* version of *wakeup* has been used to awaken only one process when an inode lock is released. The process awakened using this approach has to assume more responsibility: if the process no longer wants the inode lock, it must do another *wake1proc* to pass the lock to another waiting process.

VN_HOLD/VN_RELE Locking

Vnodes represent shared resources in most cases, and can hence be referenced by a number of different processes in a completely asynchronous fashion. When a new pointer reference to a vnode is created, a reference count in the vnode is incremented by doing a VN_HOLD (which uses an atomic add operation in the multiprocessor case). When the caller is done with the vnode, it releases the reference by doing a VN_RELE operation. The VN_RELE does an atomic decrement of the reference count and if it becomes zero, calls VOP_INACTIVE, which does a file system dependent cleanup operation on the vnode.

In some file systems the vnode continues to exist in a quiescent state after the VOP_INACTIVE, such that it can be reclaimed by a future lookup operation. The problem with this is that the zero count state is used to signify the quiescent state in these file systems. But the count goes to zero before the inactivation is performed, so there is a serious race between the lookup operation and the inactivation. It is possible for another process to find the vnode and do a VN_HOLD followed by a VN_RELE before the inactivation is performed, resulting in two inactivation attempts. To solve this problem it is necessary to change the file systems not to use the zero count as the inactive indicator. For example, in the S5 and UFS file systems, the count is decremented again to –1 under the inode cache lookup lock to *truly* indicate the inactive state. On a lookup, which is also done holding the cache lookup lock, when the count for an inode is –1, the inode is reactivated and the count set back to zero before doing a VN_HOLD to count the new reference. This solution makes the state transitions between active and inactive states safe, while allowing the use of atomic operations for VN_HOLD and VN_RELE.

Encore describes using a per-vnode spin lock around the VN_HOLD/VN_RELE increment and decrement operations [11], but they do not reveal their solution to the above race condition.

Performance Tuning

A number of useful tools were available to the SVR4MP developers [6, 17]. These tools, together with techniques very similar to those described by Paciorek, *et al.* [15], allowed reasonably accurate characterization of the locking contention and scalability of the system and the detection of deadlocks.

Cache Contention

A number of different caches are used within all UNIX variants to enhance system performance. Logically, such caches are collections of objects each tagged with an identifier. Typical operations perform some actions on an individual cache object. These actions can often be quite self-contained and very scalable, since their locality is constrained. However, once the operation enters the domain of the cache itself, it can collide with other operations on different objects in the cache. Hence, attention is naturally drawn to these caches as places where processor contention can occur and needs to be relieved. Most of these caches have a similar structure, as illustrated in Figure 1. In Figure 1, the hash queues are an array of queue head structures, indexed by computing a hash function of an object identifier during a cache operation. Each square box represents an element of the cache, and those that are not marked "Busy" are chained in least-recently-used (LRU) order on a free list. To aid in discussing locking strategies for this organization it is useful to consider the three main operations: lookup, release and flush.

A cache lookup operation scans the appropriate hash queue for the given object identifier. If an element matching the identifier is found and is not busy, it is removed from the free list, marked busy and returned to the caller. If the matching element is busy, then the caller waits for it to be released by the current owner and then rescans the hash queue to make sure the identity of the matched element has not changed. If the identifier is not found, an

element is removed from the front of the free list and its current hash list and replaced on the hash queue of the searched-for identifier. The element is marked busy, its contents are then changed to reflect the new identity and it is returned to the caller. The release operation of a busy cache element puts a busy element back onto the free list and clears the busy mark.

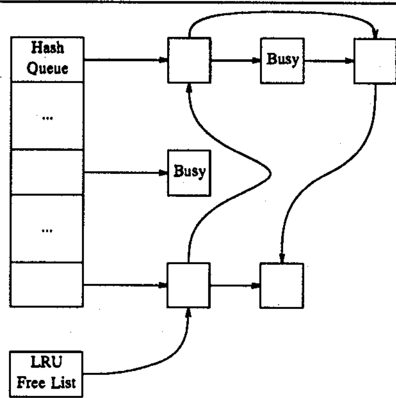

Figure 1: General Cache Organization

The flush operation runs periodically to clean elements on the free list so that they can be reused immediately when a cache miss occurs. The flush operation scans the list for items that need cleaning, removes them from the free list, cleans them and returns them to the free list when done. As an example, the buffer cache cleaning involves queueing modified disk blocks to be written to the disk. A very detailed description of this structure relative to the disk buffer cache can be found in Bach [1].

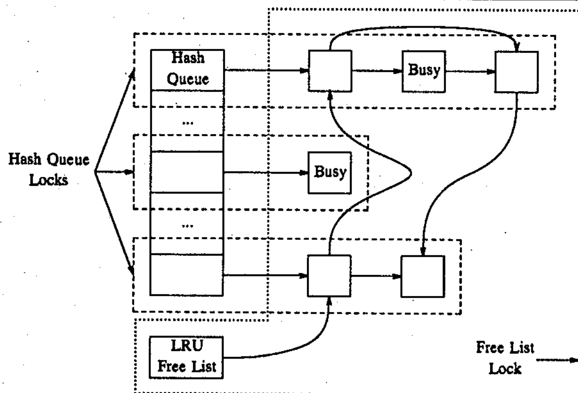

Figure 2: Separate Hash Queue and Free List Locks

The "intuitively obvious" way to lock this structure involves the use of three types of locks: a lock on each cache element, a lock for each hash queue and a lock to protect the free list, as shown in Figure 2. The dotted and dashed lines enclose the data structures protected by each of the hash queue and free list locks (the cache element locks are not shown). This locking strategy was used by Encore in their Mach/4.3BSD buffer cache locking, and has a number of deadlock and performance difficulties [4]. In terms of overhead, this locking requires obtaining at least three locks per lookup or release operation, including the per-element lock. A cache lookup miss may require obtaining two arbitrary hash queue locks, thus requiring a deadlock avoidance strategy to be implemented. More importantly, the free list lock is obtained for every operation, so it represents the limiting factor for the scalability of the cache, as discovered and reported by Encore.

A simplification of this approach lowers the locking overhead without sacrificing scalability (assuming short average hash queue size) by using only the single free list lock to protect all of the data structures, as shown in Figure 3. An early version of the Consortium locking for the buffer cache included a per-element lock along with the free list lock. It was noted that an element lock was almost always obtained while holding the free list lock, and was hence redundant. Removing the element lock reduced lock overhead by 50% and lowered contention also. Of course, the element lock can not be removed if it is a semaphore resource lock which replaces the busy mark on the cache element, as in the semaphore locking approach. In the Consortium case, the element lock was a mutex which only protected manipulations of fields within the data structure.

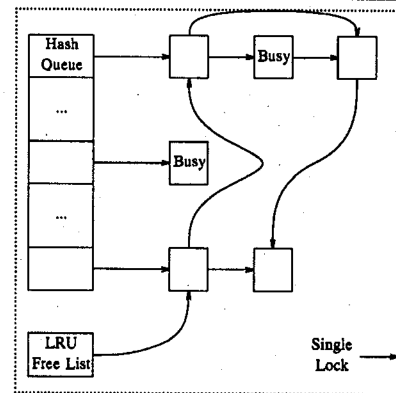

Figure 3: Single Lock for All Cache Data Structures

Unfortunately, the single lock still represents a contention point when the access rate to a cache is high. A solution to this problem is to fragment the free list into segments which are associated with each hash list and use a lock on each hash list to protect both the hash queue and the free list, as shown in Figure 4. Any time a free element is required, it is allocated from the free list associated with the hash queue being searched, so that each cache element is permanently bound to a fixed hash queue. Because of this, each hash queue is typically initialized to have a configurable, constant number of elements. This cache organization was dubbed a

Software Set-Associative Cache (SSAC) due to its logical resemblance to a hardware set-associative cache, and is described in detail elsewhere [17]. It represents the key technique used in multithreading the file system caches to obtain practically unlimited scalability. The description is extended here to show the application of the technique to a number of different caches and illustrate the problems that arose.

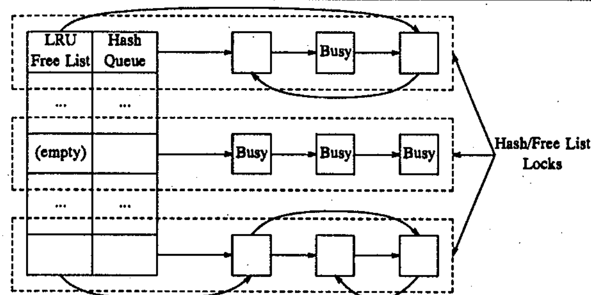

Figure 4: Software Set-Associative Cache Locking

A follow-on paper by Encore describing parallelization of a vnode-based file system [11] does not refer to the free list contention problem. The buffer cache locking is described as consisting of a lock on each hash queue, plus a lock inside each buffer. From this description it is not possible to determine how or even if they have solved the free list contention problem. The problem is likely moot to them, however, as their stated intention was ultimately to replace the buffer cache entirely with the Mach "No Buffer Cache" code from CMU.

File System Caches

While tuning the file systems, three SVR4 file system caches showed up as having significant levels of cache contention. The first of these caches is the *segmap* cache, which is used to map file pages into windows where they can be copied during read and write operations. The segmap cache replaces the buffer cache in SVR4 for most data read and write operations. The second cache is the buffer cache (a shadow of its former self), which is used to hold file system structural information, such as blocks of inodes. The last cache is the directory name lookup cache (DNLC), which is a cache of pathname component translations. The segmap cache exhibited 10-15% contention with only two processors running cache-bound file system I/O, while the buffer and DNLC caches had similar contention when the GAEDE and AIM 3 benchmarks were run on 5 CPUs. The SSAC technique was applied to these caches, with slightly different structures in each case. The locking contention for all of the caches was reduced to less than .1%. The different structures lead to some small problems which highlight the properties of the approach.

Segmap Cache

The *segmap* cache was changed first, as it represented the largest bottleneck. In this case, the hash queue and free list pointers in the cache structure were kept. The hash and free queues for each hash queue were set up at initialization to contain 4 cache elements each. The hash queue manipulations were simplified, since elements are never moved from one hash chain to another. This code change was very straightforward and took less than one day to accomplish. No side effects or difficulties were apparent after the change, and the contention was reduced as described. Segmap cache elements are not locked for exclusive use on return from the lookup operation, but a reference count is incremented. Hence, more than one process may actually use a cache entry. To protect these processes from one another, the hash queue lock for the queue containing an element is locked while fields in the cache element are manipulated. This avoids the necessity for a separate mutex lock to protect the data contents within the cache. Contention for this per-element locking is included in the less than .1% contention measurement.

Disk Buffer Cache

The application of the SSAC structure to the disk buffer cache proved to be somewhat more interesting and challenging due to some features of the SVR4 buffer cache code. Memory for the cache headers and buffers is not statically allocated at initialization, as it was in SVR3. Rather, the free list of headers grows as needed and the buffers themselves are dynamically allocated and freed to accommodate different buffer sizes. This made it difficult to configure a set of fixed-length buffer chains on the hash queues, so another approach was needed. In addition, the cache hashing function was found to be very poor. Most kernel hash queue arrays are configured to have 2^n entries so that the cheaper bitmask operation (tag & (2^n-1)) can be used instead of the modulus operation to fold the object identifier tag into a hash index. If the object identifiers are separated by a power of two, then only a subset of the hash queues are actually ever used. In the case of the buffer cache, a UFS file system with 4-Kbyte blocks generates object identifiers which are usually multiples of 8 (when converted to 512-byte physical block numbers), thus using only every 8th hash queue. This problem can be alleviated by using only 2^n-1 entries, which requires the more expensive modulus operation, but gives a better hashing distribution. (As another example of this, process structures were always allocated on 512-byte boundaries, so any *sleep* on the address of any proc structure always queued the sleeper on sleep hash queue 0.)

One of the desirable properties of a cache is that the hash queue remain fairly short, to reduce the search time required for a lookup. A traditional target has been to keep the average size of each hash

queue at around 4 buffers. When the free list for each hash queue was configured to be this size, it was discovered that the buffer cache code contains an inherent deadlock. Certain operations require the use of 2 or more buffers, which can deadlock when all of the buffers on a free list are used up by such operations. This deadlock could not be eliminated without radical restructuring of the code. So, to make its occurrence less likely, larger free lists were configured and shared among a number of hash queues, with a single mutex lock per free list. Each hash queue is bound at initialization to a particular free list, and buffers do move between hash queues, but only within those bound to the same free list. This approach provided the contention reduction desired, while increasing the probability of the deadlock only marginally.

Directory Name Lookup Cache

The DNLC cache changes were another variation on the SSAC theme. All of the free list and hash queue pointers were removed and the cache structured as a 2-dimensional array, with a vector of simple spin locks to lock each row of the cache. To maintain the LRU ordering of each cache line, a timestamp was added to each DNLC structure indicating the last time that the structure was accessed, with a 0 timestamp indicating a busy cache entry. When a cache search is attempted, the entry with the lowest non-zero timestamp is remembered and reused if the name is entered into the cache.

The semantics of this particular cache have some unusual features, most of which do not affect the SSAC structure. The lookup and enter operations are split into two distinct operations, where they are normally combined. A number of operations purge entries from the cache based on different characteristics, such as vnode identity, file system identity, just any old entry or the entire cache. The purge operation actually removes the entry, rather than cleaning it as in other flush-type operations. Because of this, a little more care needs to be taken in deciding which entries should be purged when merely reclaiming space. In most of the purge operations, the entire cache has to be scanned and all elements satisfying the search criterion must be removed. The division of free list locks in this case is an advantage, because each row of the cache is locked individually as the cache is searched, allowing other operations to proceed in parallel. On the other hand, the multiple locks do not allow the state of the entire cache to remain frozen during an operation. (Although this property does not present a problem in normal operation, a debugging function which counts all of the occurrences of a given vnode in the cache is no longer reliable.)

Since there is no single LRU free list, purging an element when any one will do has to be done carefully. The approach of traversing each cache row processing each element in turn works very well in the cleaning case (for example, the buffer cache), where the cleaned element is not reused immediately. In the purging case, it is a poor approximation to selecting a least-recently-used entry to purge. A better LRU approximation for the purging case is to select the least-recently-used entry from a cache row, purge it and then move onto the next row in the cache, cycling through the cache.

A potential problem was noticed in the hashing function for this cache. The same entry can be entered into the cache with different user permissions (credentials), so an often-used directory entry could be in the cache many times with different credentials for different users, all hashed onto the same queue. This could cause thrashing on that queue. The solution to the problem is to use the credential pointer in the hash calculation to distribute the occurrences of the name to a number of different hash queues. This illustrates a general property of the SSAC organization, which is that the goodness of the hashing function becomes more important when relatively short, fixed-sized cache rows are defined. In general, the shorter each row is, the faster the search can be, but the probability of thrashing within a row increases. Experience from hardware caches, where 4-way set associativity is considered adequate, suggests that 4 is a reasonable starting value for tuning the hash queue length.

The DNLC cache also has some other interesting behavior in its interactions with the UFS and S5 file systems. When a vnode is entered into the DNLC cache, it has a VN_HOLD placed on it. Quite often, the last reference to a UFS or S5 file is from the DNLC cache. Because of this, the *iget* function, which attempts to find a given file in the inode cache, calls the DNLC purge-any function when there are no more free inodes in the cache. This purging represents a hit-or-miss effort which is repeated until one of the file system's inodes whose last reference is from the DNLC cache is purged. If one is hit, the purge function calls VN_RELE to decrement the vnode reference count to zero, which then calls VOP_INACTIVE to release its inode. In the S5 file system, this is where the fun began: If the file being inactived was unlinked, then the inactivate routine called the *ifree* function to release the inode number back to the inode free list. If the call to *iget* was for a file being allocated (from *ialloc*), then the *ifree* function would deadlock trying to obtain the non-recursive mutex lock on the file system structure, which was held by *ialloc*. The solution to the problem was to release the mutex lock in *ialloc* before the call to *iget*. Unfortunately, this allowed a race where attempts could be made to allocate the same inode more than once, which had to be dealt with by detecting the already-allocated inode after return from *iget*. This example is cited to illustrate the (often unexpected) recursive flavor of the SVR4 file system code.

The Encore approach to locking the DNLC cache is the combination of hash and free list locks shown in Figure 2 [11]. As such, each operation acquires between 1 and 4 locks to accomplish an enter operation, whereas the SSAC approach always acquires only 1 row lock with lower contention than in the Encore case due to the fragmented free lists.

Inode Caches

The inode caches in the UFS and S5 file systems are additional candidates for the SSAC multithreading approach. However, a single lock has been used for the entire collection of inode hash queues and the inode free list in each file system. This strategy was chosen due to an insufficient level of measured contention. It could be argued that the lack of contention arises from the relative infrequency of file open or create operations relative to others, such as reading or writing. Also, the DNLC cache removes some of the inode lookup load caused by pathname searches. The Encore locking approach is the same as that used for their other caches, namely the use of a lock for each hash queue as well as a free list lock.

Disk Queue Tuning

Once the contention on the three caches was removed, a 2-CPU system with a single disk became essentially I/O-bound running the GAEDE benchmark. To investigate the cause of this, some performance measuring was added to the kernel which reported utilizations of all processors and the disk controller, as well as the size of the disk queue. These measurements were collected at each clock tick and averaged over a 1-second interval. Watching these statistics in real time revealed that the system was alternating between periods of processor saturation with little or no disk activity, and disk saturation with little or no processor activity. The disk saturation occurred when the disk queue size rose to several *hundred* requests in length. Unfortunately, it is theoretically possible in SVR4 for the size of the disk queue to be bounded only by the number of allocatable memory pages in the system.

The blocking of the CPUs during periods when the disk queues were very large was due to the fact that processes were blocked on synchronous I/O which was generated to update inode contents and directories. As an experiment, all of the synchronous inode update operations were changed to delayed writes to decouple them from the disk driver. This change improved the CPU scalability substantially, at the cost of a less consistent file system. On a two-processor system, the GAEDE benchmark could be transformed from mostly disk-bound to mostly CPU-bound with this change. (A kernel variable which turned this feature on or off was cynically dubbed the "benchmark flag", with the suggestion that at least some vendors actually have such a flag.) McVoy [14] proposed doing something similar to this, but with better file system consistency, by forcing the correct ordering of asynchronous operations to the disk for directory and inode updates. The change described here was not intended primarily as a file system enhancement, but rather as a tool to make the benchmark more CPU-bound to uncover locking contention in the file systems.

Another approach, which has been used successfully to increase write throughput of the buffer cache in SVR3 [16], was to queue synchronous requests near the front of the disk queue. This change gave only minimal benefit. It was found that requests were often queued asynchronously, but then a process would later decide that it wanted the data block. This suggested an approach whereby requests which were being waited for were dynamically promoted to the front of the disk queue, to allow waiting processes to become active sooner. Surprisingly, this change caused overall throughput of the benchmark to decline dramatically, because the disk queue never actually became empty, but increased monotonically in size.

It appears from this that balance among utilizations of resources is more important than optimizing the utilization of one class of resource, namely the CPUs. The segments of idle CPU time seem to be necessary to restore balance by lowering the effective request rate to the disk, allowing it to clear the accumulated backlog.

The real problem here is that there is a "high wall" between the disk driver and upper level file system layer. Once an asynchronous write request is passed to the disk driver, there is no current way to retrieve it until the operation is complete. This represents wasted I/O activity, since a process that waits for such a request is likely to modify the data just written. Similarly, the disk can be idle for long periods of time because the cache flushing code which could give the driver some work to do does not know that the disk is idle. When it does give it work to do, it tends to flood the disk queue with requests, which overloads the driver and slows or idles the CPUs.

Fixing these problems to even out the flow of cache copy-back traffic to the disk is one part of the solution to this problem. The other part of the solution is to reduce the write requirements of the file systems themselves, possibly by the use of log-structured file systems such as that implemented for the Sprite project [19].

Related Work

The closest work to this effort was done by Encore to multithread their vnode-based Mach file system and the OSF/1 file system [11, 12]. Their approach was actually quite different in a number of ways.

The most significant difference is that they chose to lock the file systems below the vnode layer, that is, there is no generic locking outside the file system. The SVR4MP approach was the exact opposite of this. The locking is provided in a generic sense across the vnode operation interface, with the vnode lock available as the mutual exclusion lock for all of the per-vnode data, including file system dependent data. The Encore approach replaced the inode ILOCKED flag with read/write resource locks, which extended the file semantics to allow true concurrency for file reading. The rationale for this extension was that some files are read frequently on a large system, and that performance could be enhanced by reducing lock contention on these files. During Consortium tuning, this type of contention was not encountered, probably due to our configurations being smaller in size than Encore's. Another advantage of locking below the vnode interface is that the file system writer has the flexibility to determine whether a read/write or simple mutex lock should be used, whereas the Consortium approach allows only mutex locking. On the other hand, the Consortium approach involves less semantic (i.e. code) changes, particularly since it is permissible for a vnode lock to be held around more than one VOP call when that is required. This was important due to the time-to-market and robustness constraints that the project faced.

Encore avoids contention by releasing the resource locks across any blocking operations, which allows a race between two simultaneous lookups of the same inode. This race requires a recheck of the hash queue whenever a new inode is read from disk. In the OSF/1 file system, the recheck is avoided by timestamping data structures, such as a hash queue, when they change and only rechecking a queue if its timestamp changes across a blocking operation. Much of the paper on OSF/1 describes the races introduced into directory operations by more permissive locking and how they were solved using timestamps. In the Consortium approach, the original resource locking semantics are preserved, keeping the ILOCKED or IRWLOCKED flags locked across blocking operations, even though the vnode mutex locks are released.

Another clear difference can be seen in the general cache locking strategy, which has been commented on throughout the paper. Practically all of their described cache locking fits the hash queue lock + free list lock + element lock model, as shown in Figure 2. The Consortium locking uses the single lock model shown in Figure 3, modified to use SSAC locking (Figure 4) where contention is too high.

Summary

The SVR4MP effort resulted in a system which largely met its goals. The degradation relative to a uniprocessor SVR4 kernel is very close to the 5% target, and the throughput scalability at 5 processors is 88% of the theoretical maximum. This is computed as 5-CPU Throughput / (5 * 1-CPU Throughput). Figure 5 is a graph which shows the actual increase in throughput as a function of the number of processors, as well as the ideal scalability. To achieve this goal given the disk-bound behavior described previously, it was necessary to configure 8 separate disk drives on 4 disk controllers in the benchmark system (with the "benchmark flag" turned off).

The file system multithreading effort resulted in a model which provides a lot of generic support for locking within each file system type. The vnode locking around VOP functions provides default locking protection for most file system functions, making individual file system multithreading easier.

Significant contention points in the file system caches were relieved by the application of the Software Set-Associative Cache structure. Once these significant locking bottlenecks were relieved, the file systems were found to be inherently disk bound by our benchmarks. Some disk queueing modifications were tried, with limited success and one dramatic failure. However, these modifications lead to insights about the importance of maintaining balance to achieve optimal throughput, rather than attempting to optimize the utilization of a single resource class.

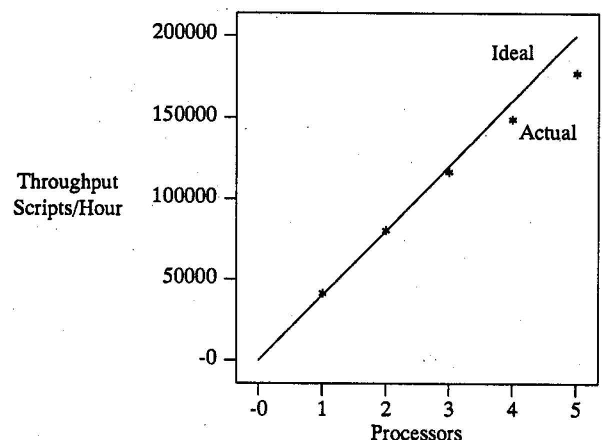

Figure 5: Scalability of the Gaede Benchmark

The Sprite Logging File System is reportedly an order of magnitude more efficient at using disks than existing file systems [19], running at CPU saturation while consuming only 17% of disk bandwidth in a test which is similar to GAEDE. This being the opposite of our situation, the marriage of multiprocessor technology with log-based file

systems appears to be quite desirable.

SVR4MP is available from UNIX Systems Laboratories as a source-code upgrade to System V Release 4. More information can be obtained by calling 1-800-828-UNIX.

Acknowledgments

This project has been a pleasure for us to be involved with, as most of the architecture team had at least two generations of multiprocessor design experience. A large number of people have contributed to the success of SVR4MP, most having written code: Mike Abbott, Sunil Bopardikar, Fiorenzo Catteneo, Calvin Chou, Ho Chen, Ben Curry, Jane Ha, Jim Hanko, Mohan Krishnan, John Litvin, Mani Mahalingam, Arun Maheshwari, Cliff Neighbors, Sandeep Nijhawan, Mark Nudelman, Lisa Repka, Sunil Saxena, K. M. Sherif, Moyee Siu, John Slice, Dean Thomas, Vijaya Verma, Fred Yang and Wilfred Yu. Special thanks are due to Wilfred Yu for the performance numbers quoted here. Thanks are also due to the reviewers for their helpful suggestions for improvements to this paper.

References

[1] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs NJ, 1986, pp. 38-59.

[2] M. Bach and S. Buroff, Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 64:1733-1749, October 1984.

[3] R. Barkley and T. P. Lee. A Dynamic File System Inode Allocation and Reclaim Strategy. Froceedings of the Winter 1990 USENIX Conference.

[4] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis. Froceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems, pp 105-126, October 1989. Also appeared as: Mach/4.3BSD: A Conservative Approach to Parallelization. *Computing Systems*, Vol. 3, No. 1, USENIX, Winter 1990.

[5] M. Campbell, Richard Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith and R. Wescott. The Parallelization of UNIX System V Release 4.0. Froceedings of the Winter 1991 USENIX Conference.

[6] M. Campbell, R. Holt and J. Slice. Lock Granularity Tuning Mechanisms in SVR4/MP. Froceedings of the Second Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II). USENIX, March 1991.

[7] S. Gaede. A Scaling Technique for Comparing Interactive System Capacities. Froceedings of the Conference of CMG XIII, December 1982.

[8] R. Gingell, J. Moran and W. Shannon. Virtual Memory Architecture in SunOS. Froceedings of the Summer 1987 USENIX Conference.

[9] S. Kleiman. Vnodes: An Architecture for Multiple File Systems in Sun UNIX. Froceedings of the Summer 1986 USENIX Conference.

[10] G. Hamilton and D. Conde. An Experimental Symmetric Multiprocessor ULTRIX Kernel. Froceedings of the Winter 1988 USENIX Conference.

[11] A. Langerman, J. Boykin and S. LoVerso. A Highly-Parallelized Mach-based Vnode Filesystem. Froceedings of the Winter 1990 USENIX Conference.

[12] S. LoVerso, N. Paciorek, A. Langerman and G. Feinberg. The OSF/1 UNIX Filesystem (UFS). Froceedings of the Winter 1991 USENIX Conference.

[13] M. K. McKusick, W. Joy, S. Leffler and R. Fabry. A Fast File System for UNIX. *Transactions on Computer Systems*, Vol. 2 No. 3, pp 181-197. ACM, 1984.

[14] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. Froceedings of the Winter 1991 USENIX Conference.

[15] N. Paciorek, S. LoVerso, A. Langerman. Debugging Operating System Kernels. Froceedings of the Second Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS II). USENIX, March 1991.

[16] K. Peacock. The Counterpoint Fast File System. Froceedings of the 1988 Winter USENIX Conference.

[17] J. K. Peacock, S. Saxena, D. Thomas, F. Yang and W. Yu. Experiences from Multithreading System V Release 4. Froceedings of the Third Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III). USENIX, March 1992.

[18] R. Rodriguez, M. Koehler, L. Palmer and R. Palmer. A Dynamic UNIX Operating System. Froceedings of the Summer 1988 USENIX Conference.

[19] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. Froceedings of the Thirteenth ACM Symposium on Operating Systems Principles. ACM SIGOPS, Vol. 25, No. 5, October 1991.

[20] L. M. Ruane. Process Synchronization in the UTS Kernel. *Computing Systems*, Vol. 3 No. 3, USENIX, Summer 1990.

[21] U. Sinkewicz. A Strategy for SMP ULTRIX. Froceedings of the Summer 1988 USENIX Conference.

Author Information

Kent Peacock has worked at Intel as a Consultant for 2 years as a member of the Intel Multiprocessor Consortium. Prior to that, he worked at Acer/Counterpoint developing a multiprocessor implementation of System V and the Acer/Counterpoint Fast File System, which is now part of SCO UNIX [16]. He has worked on design and implementation of 5 multiprocessor systems since 1978, and has dabbled in performance tuning, C compilers, multiprocessor debugging tools and graphics applications. He graduated from the University of Waterloo in Ontario, Canada with a Ph.D. in Computer Science in 1979, having previously completed a Master of Mathematics in Computer Science in 1975. In 1974, he graduated from the University of Manitoba, in Winnipeg, Canada, with a Bachelor of Science in Electrical Engineering. He can be reached via U. S. Mail at 1747 Fanwood Ct.;San Jose, CA 95133 and electronically at kentp@stps18.intel.com .