USENIX Association

Proceedings of BSDCon ’03

San Mateo, CA, USA
September 8–12, 2003

Reasoning about SMP in FreeBSD

Jeffrey Hsu
The FreeBSD Project

Abstract

While the subject of SMP locking primitives has been well covered in the literature [Val][Schm][Bald][Leh], there has been relatively little discussion on the rationale and process behind the application of these locking primitives. This leads to an inverted problem where the bulk of the work in making a kernel SMP-safe lies above the locking primitives, yet there is little guidance on how and where to lock in the individual subsystems. In this paper, we will discuss our experiences with SMP locking in the FreeBSD kernel and illustrate some of the reasoning concerning the placement or non-placement of SMP locks in the kernel. We hope this will aid other developers in locking up the remaining subsystems and in understanding the locks that are already in place. We start with an overview of general locking strategies followed by many examples of race conditions caused by faulty SMP reasoning and give solutions for correctly locking up the affected piece of code. All our examples are taken from actual committed versions of the FreeBSD source.

1. Global Reasoning

Our first guideline pertains to the difference between global reasoning and local reasoning. To paraphrase a famous quote about distributed systems[Lamp], SMP locking is where some code that you didn't even know existed can break your own local code.

Guideline #1: Think globally.

For SMP to work properly, all the affected code must adhere to the same locking strategy. A single piece not under the necessary lock can render all the other locks useless. For example, one common race condition concerns read-modify-write operations. A race window exists between the time a processor does the read and before it does the write, whereby another processor does a write.

```
processor 1            processor 2
read
modify
                       read
                       modify
                       write

write    ← loses processor 2's write!
```

The most frequently used solution for this race is to place locks around all the read-modify-writes of this variable. However, missing a lock can mysteriously result in a write being lost. We see this race in rev 1.114 of kern_descrip.c between the fcntl() function and the flock() function, both of which perform read-modify-write operations on the f_flag field, but only one of which holds the mutex lock.

```
fcntl():

        ...
        case F_SETFL:
        ...
        fp->f_flag &= ~FNONBLOCK;

flock():
        ...
        FILE_LOCK(fp);
        fp->f_flag |= FHASLOCK;
        FILE_UNLOCK(fp);
```

Here, even though the flock() code takes care to perform its operation while holding the mutex lock for a file structure, a fcntl() operation could sneak in and modify some other, totally different, bit in the f_flag field, causing the result of the flock() to be lost!

```
fcntl                  flock
read
modify
                       lock
                       read
                       modify
                       write
                       unlock

write    ← loses flock() result!
```

Furthermore, the result of either operation can be lost due to the missing lock, not just the place where the lock is missing, as is illustrated by the following sequence of operations:

```
              fcntl              flock
                                 lock
                                 read
                                 modify
              read
              modify
              write

    loses fcntl() result!  →     write
                                 unlock
```

In this case, both operations occur in the same file in the well-examined kern/ directory, so this bug should have been easy to detect. However, there are uses and assignments of the f_flag field in far off places like dev/streams/streams.c which need to be examined for potential races.

2. Understand the underlying code

A closely related principle to knowing all the places where a field is used is to understand what the underlying code does. Only by understanding what the subsystem is trying to accomplish can a proper locking strategy be devised. Many cases of improper SMP locking can be traced back to a misunderstanding about what the underlying code does.

Guideline #2: Understand the code to be locked.

For example, in rev 1.79 of uipc_usrreq.c, file descriptor table locks were added in the following code in unp_attach():

```
FILEDESC_LOCK(curproc->p_fd);
unp->unp_rvnode = curthread->td_proc->p_fd>fd_rdir;
FILEDESC_UNLOCK(curproc->p_fd);
```

Figure 1

This example locks a single read statement of the word-sized fd_rdir field. A word-sized memory read operation is already an atomic operation [Schimmel. Neither the fd_rdir nor the unp_rvnode fields are accessed in the rest of the routine. As we shall see later on, placing locks around a single read statement as is done here is rarely the right thing to do. But more importantly, the field that is being updated, unp_rvnode, is obsolete and is not used anywhere else in the code! So, the correct thing to do here should have been simply to delete this line, rather than wrapping a useless assignment inside a mutual exclusion lock. Here is a case where knowing what the unp_rvnode field is used for or not used for would have resulted in leaving out the file descriptor table lock altogether. An application of the first guideline would have helped in discovering this.

A proper approach to SMP locking starts with asking what the underlying subsystem is trying to do, then asking which of these operations involve races, and finally, which type of locks are appropriate to close these races. This is in marked contrast to a bottoms-up approach of locking individual statements considered in isolation, which does not work.

3. Naive SMP Locking

While the first principle requiring locks to be globally applied would seem to dictate that everything should be wrapped inside locks, this is not the case. In fact, a naive "wrap all accesses inside locks" approach is rarely the right thing to do and leads to unnecessary locking where locks are not needed and missing locks where locks are needed. The places where this approach gets a lock right, it's by accident. So, this guideline pertains with what not to do.

Guideline #3: Don't simply wrap all accesses inside locks

We already saw a consequence of this in Figure 1, where file descriptor table locks were placed around a single atomic read of a field in the file descriptor table structure. Another example of an unnecessary lock is in rev 1.79 of uipc_usrreq.c,

```
FILEDESC_LOCK(td->td_proc->p_fd);
vattr.va_mode =
  (ACCESSPERMS & ~td->td_proc->p_fd-
>fd_cmask);
FILEDESC_UNLOCK(td->td_proc->p_fd);
```

which is an atomic read of the fd_cmask field with no other reads of any other file descriptor table structure fields nearby.

Corollary: Don't lock single atomic memory reads.

Consider the following generic structure,

```
        struct something {
                int field;
        int otherfield;
    } *p;
```

There are few cases where

```
    LOCK(p)
    x = p->field
    UNLOCK(p)
```

can protect anything.

If someone else is writing to p->field, then that write either occurs before or after this read. Even with locks, both of the following two cases are possible.

```
    processor 1             processor 2
    LOCK(p)
    modify p->field
    UNLOCK(p)

                            LOCK(p)
                            x = p->field
                            UNLOCK(p)
```

or

```
    processor 1             processor 2
    LOCK(p)
    x = p->field;
    UNLOCK(p)
                            LOCK(p)
                            modify p->field
                            UNLOCK(p)
```

So the lock doesn't help determine an order here. The variable x could hold either the new or the old value of p->field. The read of this field might as well be unlocked. (The same argument applies to the store memory barrier effect of the LOCK() and UNLOCK() operations --- the store could occur either before or after the read, so it has no effect on determining the value read.)

But, one case where locks would be required is if the field temporarily holds a value that no one else is supposed to see and the writer, operating with the lock held, will store a valid value before releasing his lock. In this case, both the writer and reader need to hold the lock before accessing this field.

The situation is different if multiple fields in the structure were being read and we wanted to guarantee their read values were mutually consistent. Here, the reader would hold a lock around the multiple read statements and the writer would hold a lock across the multiple write statements. Then both readers and writers would always see a consistent picture of the fields. So, in the case, the correct course of action here is to expand the scope of the lock to cover all the nearby reads without releasing the lock in between. Then it is not a lock around a single read statement, but around multiple statements.

Finally, one might lock a structure in order to guarantee that it does not get freed while its fields are being used, but this is not the case for the examples mentioned. Locking to avoid deallocation during use usually requires some other lock to protect the initial call to LOCK(). Often, this problem is better solved with a reference count, a scheme that we will discuss next.

This section gives some valuable guidelines on what needs to be locked and why. Don't put a lock around everything just to put a lock around everything. As shown, that strategy can go wrong by having locks that cover too few statements, cover statements which don't need to be covered, and gives a false sense of security concerning the ordering between read and write operations. The goal in SMP locking is not to serialize everything through mutex locks, but to allow as much parallelism as possible while maintaining SMP safety.

4. Reference Counting Strategy

One of these SMP safety issues concerns guarding against deallocation. Reference counting is a scheme frequently used in SMP systems to protect against deallocation while an object is still in use. When a reference to an object is returned or stored, the reference count for that object is incremented. When a reference is no longer needed, the count is decremented, and if zero, the object is freed. The reference count manipulation must be performed under a mutual exclusion lock to avoid race conditions involving the count. These rules must be followed uniformly for the protection to work and to avoid memory leaks.

A disadvantage of the reference counting strategy is that a lock is typically acquired and released twice in the process of incrementing and decrementing the reference count. These count manipulation operations are an example of a read-modify-write operation mentioned earlier. (An alternative is to use the atomic operations in <sys/atomic.h>, but those are not applicable to reference counting scenarios where some set of actions must be performed atomically. They also incur the same amount of lock overhead at the machine level.)

5. Protect The Initial Reference

One of the problems encountered when implementing a reference counting scheme is how to protect the initial reference. Before locking an object to increment its reference count, we must ensure that object doesn't get freed before it is locked.

```
get pointer to object
                              ← obj freed here!
lock(ptr->mtx)
increment ptr->refcnt
unlock(ptr->mtx)
```

This problem is a particularly thorny one and can be addressed in the FreeBSD kernel in a number of ways, all of which involve reasoning inductively from some base case. For example, system calls occur within a process context, so system call code can safely lock the proc structure first before validating and acquiring a substructure lock. In the absence of such a natural base case, a global lock can be used in its place.

Another solution involves the transfer of a reference count lock protected by already holding an existing count or sole reference. For example, on initial allocation of a structure and before any references to that structure are made accessible, the allocation routine increments the reference count to 1. The calling routine can then store the reference in some other structure, taking care to increment the count as necessary. Any code that later accesses that reference knows by induction that a visible reference is a valid one.

6. Guard Against Deallocation

The deallocation during use problem looks like

```
processor 1                  processor 2
get obj
var1 = obj->field1
                             free(obj)
var2 = obj->field2  ←  use after free!
```

Holding the lock is one strategy to guard against deallocation during use.

```
processor 1                  processor 2
lock(obj)
      waits for unlock →     lock(obj)
uses of obj
unlock(obj) ← indicates obj no longer in use
                             free(obj)
```

The strategy of holding the lock to guard against deallocation during use must be used in conjuction with another strategy to guard the initial reference. The advantages of holding the lock to guard against deallocation is that it is simpler and lower overhead than reference counting, which requires a lock to atomically increment the reference count and another lock to atomically decrement it. It does, however, limit concurrency. With reference counting, simultaneous use of an object is allowed.

```
processor 1                  processor 2
lock obj
increment refcnt
unlock obj
                             lock obj
                             increment refcnt
                             unlock obj

use obj ← simultaneous use → use obj

lock obj
decrement refcnt
unlock obj
                             lock obj
                             decrement refcnt
                             unlock obj
```

An example of a place where mutexes are used to guard against deallocation is the inpcb lock in the networking stack. Here, due to the serial nature of the operations performed while holding an inpcb lock, only one processor can access the structure at a time anyways, so holding the lock to guard against deallocation rather than obtaining a reference count does not artificially limit parallelism.

7. Protect Linked Lists

Linked list traversal and linked list manipulation must be performed under a lock. This lock must be common to all the readers and writers of this linked list. One example of faulty locking is in rev 1.90 of sys_pipe.c, which uses the pipe lock in many places to protect knote list traversal, for example, in filt_pipedetach():

```
PIPE_LOCK(cpipe);
SLIST_REMOVE(&cpipe->pipe_sel.si_note,
                  kn, knote, kn_selnext);
PIPE_UNLOCK(cpipe);
```

Unfortunately, the knote() function in kern_event.c, which walks the knote list, is called without holding the pipe lock, leading to a race condition between the two operations. This bug is an example of not holding the same SMP lock for list traversal and list manipulation. This can be addressed, after tracing the calls of the knote() function back to pipeselwakeup(), by ensuring that pipeselwakeup() is always called with the pipe lock held. Alternatively, a new lock can be introduced to protect the knote list and acquired during both list traversal and list manipulation.

If a list will be traversed by more than one thread simultaneously, it may pay to use a shared sx lock [Bald], as is done for the some of the process lists. From the pfind() routine in kern_proc.c,

```
        sx_slock(&allproc_lock);
        LIST_FOREACH(p, PIDHASH(pid), p_hash)
                ...
        sx_sunlock(&allproc_lock);
```

Here, the allproc_lock shared lock is used to protect the linked list pointer stored in the p_hash field. This is a shared lock, so multiple threads can be executing this code in pfind() simultaneously. When modifying this field, an exclusive lock must be obtained to lock out other readers and writers, as is done, for example, in exit1() in kern_exit.c:

```
        sx_xlock(&allproc_lock);
        LIST_REMOVE(p, p_hash);
        sx_xunlock(&allproc_lock);
```

Unfortunately, shared locks are many times more expensive than simple mutexes, so they should only be used when lock profiling indicates lock contention for a given lock. Since most locks are not contested, using a shared lock rarely pay off.

One technique used in the FreeBSD code to leave open the option to switch from a fast simple mutex to a slow shared sx lock is to use macro definitions for the locking. We see this in net/if_var.h,

```
#define IFNET_WLOCK()     mtx_lock(&ifnet_lock)
#define IFNET_WUNLOCK()
                          mtx_unlock(&ifnet_lock)
#define IFNET_RLOCK()      IFNET_WLOCK()
#define IFNET_RUNLOCK() IFNET_WUNLOCK()
```

where shared read locks are differentiated from exclusive write locks in the code, but the two types of lock usage are both defined as the same simple mutex lock. This allows for an easy switch to a shared sx lock if lock profiling later determines that this is a heavily contested lock and the code usage exhibits a strong multiple-readers single-writer pattern. The amount of lock contention will depend on the application mix being run as well as the number of processors in the system.

8. Lock-Free Synchronization

There are many opportunities to exploit lock-free synchronization techniques in an SMP setting. The FreeBSD kernel does not, as yet, use many of these techniques. One technique that is currently deployed is the use of a generation count on a structure. This is a count that is incremented each time a structure is modified or freed. The generation count is remembered before an unlocked operation and checked at the end of an operation. If the generation count hasn't changed, then the structure was not modified during the operation. For example, this technique is used to avoid locking structures that are copied out to user-land for sysctls.

9. Lock Ordering and Deadlocks

Lock ordering considerations pervade much of the locking in the FreeBSD kernel.

There are four necessary conditions [Silb] for deadlock:

1. mutual exclusion
2. hold and wait
3. no preemption
4. circular wait

Breaking any one of these conditions is sufficient to guarantee that no deadlock can occur. The most commonly used approach and the one FreeBSD has chosen by design is to order the locks.

This means that locks are acquired in a particular order and if a lock is required while a higher numbered lock is held, the higher numbered lock is released and lock acquisition proceeds anew from the top. The witness facility automatically tracks lock ordering and warns if it detects locks being acquired out of order.

10. The dreaded "Could sleep while holding lock" warning

Blocking memory allocations are detected by witness and commonly reported by FreeBSD-current users. There are several common strategies for dealing with this. One is to allocate before acquiring mutex. Another is to allocate after releasing mutex. Sometimes a blocking malloc can be avoided by copying into local variables. Finally, if all else fails, use non-blocking allocation.

11. Related Work

There has been much work in the past concerning the formal semantics of programs. Some of this targets formal reasoning about concurrent programs [Lamp2]. In general, with notable exceptions such as [Sav], few of these techniques have not been applied to programs as large as the FreeBSD operating system.

12. Summary and Future Work

Many of the subsystems still remain to be locked up. An analysis of what the subsystem does, the inherent race conditions, and the proper locking strategy should precede placing actual locks in the code. That all the bugs discussed here were found in committed code indicates that we should focus on SMP safety and completeness. After that has been accomplished, then we can turn out attention towards lock profiling, tuning, and reorganizing code and data structures to better take advantage of the SMP environment.

We have gone over some of the techniques used in locking up the subsystems in the FreeBSD kernel. Both examples of correct as well as incorrect SMP locking and the reasoning behind both were explored. The common question of what needs to be locked and what does not need to be locked was illustrated in a number of contexts within the kernel. We hope this aids developers in understanding the current locking employed in the FreeBSD SMP kernel and in locking up the remaining modules.

13. References

[Bald] John Baldwin, Locking in the Multithreaded FreeBSD Kernel, Proceedings of the BSDCon 2002 Conference, Usenix, 2002.

[Lamp] Leslie Lamport, "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." CACM, June 1992.

[Lamp2] Leslie Lamport, “A Temporal Logic of Action”, ACM Transactions on Programming Languages and Systems, May 1994.

[Leh] Greg Lehey, Improving the FreeBSD SMP Implementation, Proceedings of FREENIX Track: 2001 USENIX Annual Technical Conference, Usenix, 2001.

[Sav] Stefan Savage, Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs, ACM Transactions on Computer Systems, Vol 15, No 4, Nov 1997.

[Schm] Curt Schimmel, UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers, Addison-Wesley, 1994.

[Silb] Silberschatz, Operating System Concepts, 6th Edition, John Wiley & Sons, 2001.

[Val] Vahalia, UNIX Internals: The New Frontiers, Prentice Hall, 1995.