BKL: One Lock to Bind Them All

Rick Lindsley
IBM Linux Technology Center
ricklind@us.ibm.com

Dave Hansen
IBM Linux Technology Center
haveblue@us.ibm.com

Abstract

One ring to rule them all
One ring to find them
One ring to bring them all
And in the darkness bind them.

— *The Fellowship of the Ring*,
J.R.R. Tolkien

When the topic turns to the Big Kernel Lock (BKL), the comparison to Tolkien's one Ring comes naturally. The BKL was among the first locks to be created for the Linux(R) kernel, and many other locks were developed either to complement or replace instances of it. Despite this, coders are reluctant to reduce or eliminate the usage of the BKL so while it may not rule them all, it continues, in a performance sense, to "bind them all."

Once the varied uses of the BKL are understood, the BKL can safely be replaced by other lock mechanisms, which are more appropriate for each instance. The difficulty lies in identifying these distinct instances, determining what protection is provided by the BKL in each, and carefully replacing the BKL without perturbing the rest of the system. In this paper, we examine the history of the BKL, review recent efforts to replace and remove it, and outline the work remaining. The One Lock need not rule nor bind the others any longer.

1 Introduction

Careless locking throughout the Linux kernel adds unneeded complexity and decreases performance. With the introduction of Robert Love's changes[1] to implement a preemptive kernel in 2.5.4, the effects of poor locking now can affect SMP and uniprocessor machines alike. Locks such as the Big Kernel Lock (BKL) have multiple uses and can be confusing to use correctly.

As a result of its overuse, many instances of the BKL subtly intertwine, causing a single `lock_kernel()` call to have several protective and often unrecognized effects. Until recently, for example, the BKL protected list operations on a webcam list in the CPiA driver and would lock out the NFS kernel daemon thread while these operations were being performed. These two activities always executed exclusively, when absolutely no exclusion was necessary. In determining how best to release the BKL's hold over the rest of the kernel, it is useful to examine not only how it is currently used, but how it came to be used that way.

2 History of the BKL

The BKL originated with Linux's first attempts to support SMP. The patch for 1.3.26 shows

[1] Patches available at
`http://www.kernel.org/pub/linux/kernel/people/rml/killbkl/llseek/`

the first signs of the BKL's declaration, but it was not actually used until 1.3.31. The lock was simply a bit which was set whenever a CPU was in kernel context. If another CPU attempted to enter the kernel at the same time, it spun. The net effect was to allow only one process in the kernel at a time. This was a time well before the `spin_lock()` functions, so the authors implemented this spinning behavior themselves in the `ENTER_KERNEL` assembly macro. At this point, the lock was acquired exclusively in `ENTER_KERNEL` and released in `EXIT_KERNEL`; no device drivers or kernel subsystems explicitly interacted with it.

1.3.54 brought with it the now-familiar `lock_kernel()` and `unlock_kernel()` functions defined in C. This opened up the way for code other than the kernel entry code to use the BKL. For all of 1.3 and 2.0, this code was limited to kernel daemons: `bdflush`, `kswapd`, and `nfsd`. With the new C definitions, 1.3.54 also introduced one of the BKL's most striking features: the ability to be held recursively. In that code, the lock's spin loop will terminate if the current processor already holds the lock:

```
while (set_bit(0,
  (void *)&kernel_flag)) {
    if (proc==active_kernel_processor)
      break;
      <snip...>
}
active_kernel_processor = proc;
kernel_counter++;
```

This feature made the BKL more obviously a processor lock rather than a process lock. A single process was not prevented from grabbing it multiple times, but other processors were blocked. It also greatly simplified the programmer's task: there was no worry about deadlocks with yourself. In cases where a function's caller holds the lock, the second `lock_kernel()` will never spin waiting for release.

```
lock()      // spin until acquired
func() {
   lock()   // kernel_counter++;
   unlock() // kernel_counter--;
}
unlock()    // kernel_counter--;
            // (and release if
            // kernel_counter == 0)
```

However, this convenient feature invites abuse. With the threat of deadlocks removed, programmers can take the lock "just to be safe," and there is no penalty for not diligently checking or commenting code. The penalty falls on the inheritors of this code, when they ask, "What is this guarding?" and try to remove the BKL.

The 1.3/2.0 development period saw only very limited spreading of the BKL. As 2.0 development continued, the BKL was added in only one more place, and that was for another kernel daemon.

The BKL as we know it today (a spinlock) was introduced in 2.1.23. The old `ENTER_KERNEL` and `EXIT_KERNEL` semantics were replaced by `lock_kernel()` and `unlock_kernel()` calls around critical regions. At this point, only Sparc and i386 had generic spinlock mechanisms, which meant that, besides semaphores, the BKL offered the only SMP mutex mechanism. The scope of this change in Linux's SMP support is evident from counting how many times `lock_kernel()` is called; 2.1.22 had 9 calls of `lock_kernel()` and `unlock_kernel()` while 2.1.23 had 761!

It might appear that the 2.1.23 patch is the root of all evil. But it did add a very important feature: kernel concurrency. Before this point, no two tasks could be running in the kernel at

once. The modern BKL is perhaps not as one-sided as it first looks, as it was the price to pay for kernel concurrency at the time. The current BKL removal process is just a continuation of this effort to allow more kernel concurrency.

3 Current state of the BKL

In 2.4.18, `lock_kernel()` is invoked over 500 times in about 290 files. Determining why it is invoked in those files is a little tricky, since comments are rarely present. Sometimes it's not clear that even the authors understood why it was needed; they appear to have invoked it either because the code they were copying from invoked it, or simply because they feared angering the ancient gods of coding by omitting it.

Semantically, we find that in the 2.4.18 kernel the BKL is used primarily in the following areas:

- `release` (or `close`) routines
- `open` routines
- `mmap` routines
- `ptrace` system call
- file system code
- module protection

Functionally, however, the intended use in each case is far less clear. Most of the uses can only be inferred from code inspection, because usually the users of the BKL did not create comments describing their changes. After some research into the 2.4.18 code and old change logs, however, it is possible to hazard an educated guess at the uses.

`release` (or `close`) routines. These routines are called when a file descriptor is closed for the last time. At one time, the BKL was held across the release call, and when that call was removed and responsibility for acquiring the lock was pushed down into those routines, many authors did not have the time, knowledge, or inclination to determine whether it was truly needed. Many of them remain today, even though most are unnecessary, as we'll see later.

`open` routines. These routines are called when a file descriptor is first created (opened). As with the release routines, the BKL was originally held across the generic open call and thus was held for all devices upon entry to their open code. When the code acquiring the BKL was removed, it became the responsibility of the open routines to acquire it. To avoid breaking any code relying on the ability to acquire the BKL, the patch modified each and every open routine to grab the BKL itself, and left it to the driver owners to take it out if it was unnecessary. Unfortunately, many driver owners chose not to spend time determining whether it was necessary.

`mmap` routines. These routines are called when a `mmap` call is made against the file descriptor to map some portion of the underlying data into memory. While the details of what is being mapped vary with the device, it seems to have been generally accepted that the BKL needed to be held in order to accomplish it. This remains one of the more mysterious uses of the BKL, and will be high on the list for future work.

`ptrace` system call. The BKL appears to be used to lock down the important fields of a process while `ptrace()` (which is architecture-specific) manipulates them.

file system code. In the file system code, it's harder to discern a general pattern of usage. Frequently, the BKL seemed to protect various file-system-specific data structures, as

well as some VFS structures. As noted above, file system code exploded with BKL usage in 2.1.23, and since then authors have slowly been weeding it out. What's left is, in general, the code hardest to fathom or the most sensitive to changes, and extricating the BKL from this code requires thorough knowledge of the file system being operated on. Some very recent work has made ext2 much less dependent on the BKL.[2]

module protection. The BKL is used to protect the module list in kernel/module.c. This use of the BKL has the unusual distinction of at least being consistent and well-defined. If it were not for the remote possibility of an unexpectedly positive interaction with other BKL usages elsewhere in the kernel, this mechanism could be replaced with a simple spin lock. As it is, it needs to be inspected as closely and understood as well as any other BKL usage before taking any action.

4 Why does it matter?

It could be argued that "if it ain't broke, don't fix it" and that efforts to reduce or eliminate the BKL are not only difficult in many cases, but pointless. The reference to the module code, above, would be a prime example.

The BKL is not viewed as an obstacle for many benchmarks or workloads. Certainly on uniprocessors, many other concerns are of higher precedence. But when the One Lock does obstruct some task or benchmark, it is a daunting task to remove that roadblock. As mentioned earlier, there are few comments or other documentation to explain why the BKL is used in any particular spot, let alone how it might be excised. Further, because the BKL is used in so many places, the problem may not lie in the region in which it is contended for.

Imagine this scenario. Function `foo()` grabs the BKL 500 times during a particular workload and holds it for 100ms each. Function `bar()`, on the other hand, attempts to grab it 100000 times, but seems constantly thwarted, waiting an average of 35ms 70% of the time. When it does finally get it, it holds it for an average of 10ms before releasing it. One could mistakenly conclude that reducing or eliminating the BKL in `bar()` would make the contention problem go away - and indeed it would. But the real problem probably lies in holding the lock an incredibly long time in `foo()`, thus holding up the many instances of `bar()`.

Does one fix `foo()` to hold the lock a shorter period of time, or does one fix `bar()` to acquire it less? It's bad practice to hold a spinlock a long time in `foo()`, but then it seems optimization may be needed in `bar()` to reduce the number of times locking is required or even the number of times the function is called. Determining the correct answer really requires that both functions be well understood in purpose and scope. In a given instance, the answer may be one or the other, or even both or neither. (Both functions may be completely unnecessary upon closer inspection. Equally possible would be that neither may be able to change their behavior. For example, if `foo()` must hold the BKL while calling a proprietary function or performing some hardware operation it has no control over, it may not be able to reduce its hold time. Similarly, the high number of calls to `bar()` may be a necessary evil that keeps an interface definition clean and more easily supported at the expense of a seemingly high number of function calls.)

Now recall that the BKL is used in hundreds of routines, interacting in thousands of ways, and you've answered the question of why its

[2] Patches (for 2.5 only) are available at `http://linus.bkbits.net:8080/linux-2.5/cset@1.290`

widespread use and misuse does matter.

5 Recent work

There has already been a great deal of work in 2.5 to remove the BKL where it isn't necessary. Happily, most of the fixes to remove the BKL are dirt-simple (once the arduous investigation to prove their simplicity is completed!) and provide significant, measurable benefits.

5.1 do_exit()

Perhaps the best example of this is `do_exit()`. Lockmeter data from that most-trusted of all benchmarks, the kernel build, showed `do_exit()` holding the BKL for an average of 8ms, with a maximum hold time of 55ms. That is an eternity for a cpu whose time is simply wasted spinning. On first examination of this code, removing the BKL appears difficult because so many complex data structure manipulations are done here. However, upon closer examination, it is evident that many of the functions called under the BKL in `do_exit()` already have their own locking implemented.

So what is the BKL really guarding? Linus Torvalds himself mentioned on the linux-kernel mailing list[3] (LKML) that, even in 2.4, few of these functions actually still need the BKL. The strategy for removal was simple: only hold the BKL around the functions where it is really necessary. In this case, `sem_exit()` and `disassociate_ctty()` appeared to be the most likely candidates for still needing the BKL. After a suggestion from Linus, the BKL was moved into those two functions, and out of `do_exit()` itself. The amount of time the lock was held went from 8ms on average to only *5.5us* in the worst case!

However, this fix was not without its price. Shortly after the initial `do_exit()` patch went into 2.5, posts on LKML reported OOPSes during boot whenever preemption was enabled. Replacing the `lock_kernel()` in `do_exit()` fixed the problem, as did a `preempt_disable` (which, in a preemptive kernel, all `spin_lock()` calls do implicitly). While the task is exiting, `exit_notify()` sets `current->state` to `TASK_ZOMBIE`. However, if a preemption point occurs after the state is set, the return from preemption code `sets current->state` to `TASK_RUNNING`. This makes the previously "zombied" process eligible to run again, instead of being cleaned up. The `schedule()` at the end of `do_exit()`, which is never meant to return, ends up returning.

The fix is to note the task's state when it was preempted and make sure not to make it runnable again if it was exiting when it was preempted. `do_exit()` is a prime example of why it is very dangerous to derive protection from a lock without realizing why, especially from the BKL.

5.2 `release()` removal

Many device drivers' `release` and `open` functions try to guarantee that only one open can be done on the device at a time:

```
static int opened = 0;
open()
{
    if( opened )
       return -EBUSY;
    ... do opening stuff
    opened = 1;
}
```

[3] Mailing List Archive: `http://marc.theaimsgroup.com/?l=linux-kernel&m=101484620020622&w=2`

```
release()
{
    opened = 0;
}
```

This works fine on a uniprocessor machine. However, on SMP systems a race can allow two processes to open the device simultaneously, as demonstrated in the C code in Figure 5.2.

Currently, this is not a problem for character or block devices. The VFS code holds the BKL over the calls to all char and block open functions:

```
int chrdev_open(struct inode *inode,
                struct file * filp)
{
  ...
  if (filp->f_op->open != NULL) {
     lock_kernel();
     ret =
      filp->f_op->open(inode,filp);
     unlock_kernel();
  }
  ...
  return ret;
}
```

There is similar code for block devices, but misc devices are not afforded this protection. Also, as good practice, devices should never depend on the layers above them to protect against races in their own code especially when they depend on protection provided by the BKL. They should be even more careful to avoid depending on the BKL's protection without realizing it (see `do_exit()` example).

Before the `release()` removal patches, many drivers' open/release combinations looked like this:

```
static int opened = 0;

// implicit from VFS code
// for char/block
lock_kernel();
open()
{
   if( opened )
      return -EBUSY;
   ... do opening stuff
   opened = 1;
}
unlock_kernel();

release()
{
   lock_kernel();
   opened = 0;
   unlock_kernel();
}
```

In almost all of these cases, the fix is simple: just use atomic bit operations. No spinlocks or semaphores are needed; just a simple bit operation. Now the functions are safe to use in block, char, or misc devices because they don't rely on VFS to do any locking for them.

```
static int opened;
open()
{
   if( test_and_set_bit(0,&opened) )
      return -EBUSY;
   // I'm the only opener
   ... do opening stuff
   // success
}

release()
{
   clear_bit(0,&opened);
}
```

6 Notable work, and kudos

Valiant, ongoing efforts by many code warriors to reduce or eliminate the BKL are worth mentioning. This is not a complete list, of course, but just some recent efforts that have had a significant impact.

```
open                              open
{                                 {
  if( opened )
     return -EBUSY;                 if( opened )
                                       return -EBUSY;
  // I'm the only opener            // Uh-oh, we got in before opened
  ... do opening stuff              // was set and there are two of us!
  opened = 1;   // success          opened = 1;
}                                 }
```

Figure 1: Open Race

VFS has had a difficult struggle with the BKL. The UNIX(R) "everything is a file" approach forces perfection from filesystem code. VFS is also a place where concurrency is important, so locking must be done carefully and kept finely grained so as not to adversely affect performance. The BKL's presence in so many other pieces of code has made its presence in VFS code troublesome.

Al Viro has done a noteworthy job of freeing VFS from dependency on the BKL and shifting the responsibility of locking into the underlying code.[4] This underlying code is always closer to the task at hand and can make more well-informed, finely-grained decisions than VFS can. He has also worked on documenting filesystem locking. The BKL still plays a big part in VFS and Al has done a good job of documenting that role in Documentation/filesystems/Locking. He has also documented the locking changes in Documentation/filesystems/porting.

Richard Gooch has been an exemplary advocate for pushing the BKL out of devfs.[5] On one occasion, he responded within minutes of a patch being posted which pushed the BKL into some devfs code, down from the VFS layer. A couple days later, he posted a patch purging the BKL from his subsystem because it derives absolutely no protection from the BKL.

7 Future Work

There are some specific areas that need to be addressed, and some were mentioned above: the `mmap` code, the `ptrace` code, and the file system code, for example. Stalwart, knowledgable code warriors are always sought for this sort of effort.

However, you can join the fellowship and be a BKL Eliminator even without knowledge in these areas. Even if you have no time to eliminate the BKL from existing code, you, as a coder, can help prevent it from proliferating by adopting simple standards for yourself:

- Never submit code that adds the BKL, anywhere.
- If you need exclusion in your driver, provide it yourself with your own lock.
- If you need to sleep while holding a lock, use a semaphore.
- If you still think you need the BKL, ask somebody else first.

If you *are* responsible for code that still uses the BKL, make an effort not to expand where

[4] Patches are available (for 2.5) at `http://linux.bkbits.net:8080/linux-2.5/user=viro`

[5] Patches are available (for 2.5) at `http://marc.theaimsgroup.com/?l=linux-kernel&m=101787867929523`

it is used. The next time you go to rewrite a big chunk of your code, think about the BKL and start imagining ways to remove it.

A running scorecard for each release can be found at `http://lse.sourceforge.net/lockhier/index.html`. Listed there are versions of a locking reference document for recent releases of both 2.4 and 2.5—in particular, outlining where the BKL is still found. Here is a summary for 2.5.8-pre3:

Top users of BKL in 2.5.8-pre3
(excluding filesystems)

1689	.
254	drivers
177	arch
113	sound
107	sound/oss
71	include
71	drivers/usb
47	drivers/char
39	drivers/isdn
32	include/linux
29	arch/sparc64
28	arch/sparc/kernel
28	arch/sparc
23	drivers/usb/core
22	kernel
19	drivers/usb/media
19	arch/alpha/kernel
19	arch/alpha
17	arch/sparc64/solaris
16	arch/ppc64/kernel

Top users of BKL in 2.5.8-pre3
(filesystems only)

1026	fs
76	fs/reiserfs
67	fs/coda
61	fs/ext3
58	fs/intermezzo
58	fs/hfs
54	fs/nfs
51	fs/hpfs
44	fs/affs
43	fs/udf
35	fs/autofs
32	fs/ufs
30	fs/smbfs
28	fs/jffs
24	fs/ntfs
22	fs/bfs
21	fs/vfat
19	fs/ncpfs
19	fs/autofs4
18	fs/qnx4

All maintainers with a subsystem listed above should take a hard look at their code. In most cases, the code which uses the BKL does not actually need it. Understandably, developers are reluctant to change code which they do not have great knowledge about. As the maintainer, you are the authority , you do have intimate knowledge of the code, and you can intelligently and safely remove the BKL.

Perhaps the only generic use for the BKL which is spreading is the use around calls to `daemonize()`. Many of them, like this use from `ibmphp_hpc.c`, hold the BKL for short periods of time.

```
lock_kernel ();
daemonize ();
reparent_to_init ();
...
unlock_kernel ();
```

However, there are many other cases where the same operations are performed without the

BKL. The authors would be very interested to receive any information that readers can add about this use with `daemonize()`.

In general, the kernel would be a better place if Linus would never accept another patch with `lock_kernel()` in it. Truth is, this is not likely to happen anytime soon. But an increased awareness among the Linux community can be almost as useful as eradication in achieving this goal. Until the BKL is reduced to one or zero usages, there will remain fear and uncertainty when it is encountered, and the One Lock will continue its hold over all of us.

8 Acknowledgements

IBM is a registered trademark of International Business Machines Corporation in The United States and other countries.

Linux is a registered trademark of Linus Torvalds.

UNIX is a registered trademark of the Open Source Group.

The opinions expressed are those of the authors, and do not necessarily reflect the opinions of the IBM Corporation.

Proceedings of the
Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*