

Re: How to do proper locking

Source: <http://unix.derkeiler.com/Mailing-Lists/FreeBSD/hackers/2005-08/0117.html>

From: Hans Petter Selasky (*hselasky_at_c2i.net*)

Date: 08/06/05

To: freebsd-hackers@freebsd.org

Date: Sat, 6 Aug 2005 01:39:55 +0200

On Friday 05 August 2005 19:29, John Baldwin wrote:

> On Thursday 04 August 2005 04:53 pm, Hans Petter Selasky wrote:

>> On Thursday 04 August 2005 20:15, John Baldwin wrote:

>>> On Thursday 04 August 2005 12:50 pm, Hans Petter Selasky wrote:

>>>> On Thursday 04 August 2005 15:53, John Baldwin wrote:

>>>>> On Thursday 04 August 2005 07:40 am, Hans Petter Selasky wrote:

>>>>>> On Wednesday 03 August 2005 19:21, John Baldwin wrote:

>>>>>>> On Tuesday 02 August 2005 06:23 pm, Hans Petter Selasky wrote:

>>>>>>>> Hi,

>>>>>>>>>

>>>>>>>>> I am looking for a safe way to access structures that can be

>>>>>>>>> freed. The solution I am looking for must not:

>>>>>>>>>

>>>>>>>>> - hinder scalability

>>>>>>>>> - lead to use of a single lock

>>>>>>>>> - lead to lock order reversal

>>>>>>>

>

>> This is only going to work if you are detaching. But consider the

>> following:

>>

>> FOO_LOCK(sc);

>> callout_stop();

>> callout_start();

>> FOO_UNLOCK(sc);

>>

>> Your solution is only spinning halfway around. What I say is that you

>> need some more parameters passed to callbacks, so that one can check if

>> it has been cancelled before proceeding.

>

> Err, there is no callout_start(), but callout_reset() will just reschedule

> a callout if it's already pending. The consumer of the callout can

> maintain his own side state that he can check in his callout to handle a

> state change in between scheduling the callout and the callout running.

Yes, right, but I want this state variable checking to be centralized.

freebsd-hackers: Re: How to do proper locking

On the spark I can think of the following two situations where the state must always be checked:

- before calling a callback (like `d_read / d_write / d_ioctl / timeouts / usbd_callback ...`)
- after returning from sleep

```
> > Now consider my proposal. There is no more need for
> > dev_refthread/dev_relthread. Instead the callback, "d_read()" will do the
> > checking. This consist of:
> >
> > d_read()
> > {
> > mtx_lock(args->mtx);
> >
> > if(args->refcount_copy == atomic_read(args->refcount_p))
> > {
> > /* valid */
> > }
> >
> > mtx_unlock(args->mtx);
> >
> > return;
> > }
> >
> > Per "devfs_read_f()" one has got to lock/unlock one time. It involves two
> > refcount reads and no refcount writes.
> >
> > Well, isn't this 200 percent faster?
>
> Maybe, but another thing you need to consider is "maintenance" overhead.
> Device drivers, especially, should be the simplest parts of the kernel to
> implement because we want to minimize the potential for screw up in that
> area. Having to trust that the same code is going to be duplicated 40 or
> 50 times without any errors versus having it done once in a centralized
> place where it breaks everyone if it is broken is just insane. Otherwise,
> we might as well go write the whole kernel in assembly so we can tweak
> every last bit out of it. :)
```

Yes, you are right, but the problem is, that for most callback systems in the kernel, there is no mechanism that will pre-lock some custom mutex before calling the callback.

I am not speaking about adding lines to existing code, but to add one extra parameter to the setup functions, where the mutex that should be locked before calling the callback(s) can be specified. If it is NULL, Giant will be used.

The setup functions I have in mind are for example: `"make_dev()", "bus_setup_intr()", "callout_reset()" ...` and in general all callback systems that look like these.

OLD:

```
make_dev(devsw, minor, uid, gid, perms, fmt...)
```

NEW:

```
make_dev(devsw, mtx, minor, uid, gid, perms, fmt...)
```

This increment only refcount system that I want, will be setup automatically by the setup functions, and code is saved in the device drivers. Destruction functions will increment the refcount and so invalidate the old copies of the refcounts, all centralized.

Here is my idea of how this can be implemented for the filesystem mounted on /dev :

Struct cdev will have a new field of type callback_args:

```
struct cdev {  
    ...  
  
    struct callback_args si_callback_args;  
  
    ...  
};
```

Maybe we can trim down the callback_args structure to something like this: And maybe we can pick a better name for it.

```
struct callback_args {  
    struct mtx *mtx;  
    u_int32_t *p_refcount;  
    u_int32_t refcount;  
};
```

To update the kernel you make a simple script that searches for "make_dev()" and adds one NULL parameters after the first parameter. Look over all the substitutions to catch any exceptions and that is it. Update the manual. We don't have to rewrite any drivers at all. Keep the code like it is, no changes are needed at all!!!

Then one has to update "devfs_read_f()":

```
devfs_read_f()  
{  
    This first part should be done once, and the result should be stored in some  
    file descriptor that is owned by the thread and needs no locking.
```

```
    mtx_lock(global);  
  
    struct callback_args cba = dev->si_callback_args;
```

freebsd-hackers: Re: How to do proper locking

```
mtx_unlock(global);
```

This is the second part, and as a good rule, the caller is doing the locking. There is only one problem and that is that the callback can call `tsleep/msleep`. So what we do here is that we extend "struct thread" to contain another field, "struct callback_args *td_cba_p", which we use to pass a hint to `tsleep/msleep`, about that we are holding a lock:

```
mtx_lock(cba.mtx);

if(cba.refcount_copy == atomic_read(cba.refcount_p))
{
    /* valid */
    curthread->td_cba_p = &cba; /* pass hint to next tsleep/msleep */
    error = xxx->d_read();

    if(curthread->td_cba_p == NULL)
    {
        return error; /* lock already exited */
    }

    curthread->td_cba_p = NULL;
}
else
{
    error = ENXIO;
}

mtx_unlock(cba.mtx);

return error;
}
```

Then we have to update `msleep()`:

```
int
new_msleep()
{
    struct callback_args *args = curthread->td_cba_p;
    int error;

    if(args)
    {
        /* we are holding a lock that we need to exit */

        error = old_msleep(... &args->mtx ...);

        /* here we can save a lot of code by checking
         * that things are still present like when called,
         * before returning !
         */
    }
}
```

freebsd-hackers: Re: How to do proper locking

```
if(args->refcount_copy == atomic_read(args->refcount_p))
{
    // EGONE is a new error enum, that serves this
    // special case only
    //
    error = EGONE;
}
else
{
    error = old_msleep();
}
return error;
}
```

Another consequence of this is that our device driver does not have to contain a single line about "mtx_lock()"/"mtx_unlock()". And then one is saved the worries about having to exit a lock before returning.

```
my_read()
{
    int error;

    error = tsleep()
```

Look at this. Here we can handle signal interruption, tsleep timeout and device removal in a single check.

```
    if(error) return error;

    return 0;
}
```

Then we can also call uiomove() without having to worry about exiting any locks. And if anything goes wrong, copyout() will return EGONE, and the function is no longer accessing any freed memory. Holding a lock while calling uiomove() should be allowed, because the lock is preventing the memory uiomove() is writing to/from, from being freed. Then one can actually call any function that can sleep, without worrying about exiting any locks. The only thing is that we have to make a note that these functions can exit the currently held lock or maybe we should specify currently active "callback_args".

So this is just backwards:

```
mtx_unlock();

ptr = malloc();

mtx_lock();
```

freebsd-hackers: Re: How to do proper locking

In 99 percent of the cases malloc will never sleep, and if the call takes some time, we pass a hint to malloc and to this unlock-/lock-ing centralized. That makes the code cleaner. Now how about this:

```
ptr = malloc();

if(ptr == NULL)
{
    /* no memory or EGONE */

    if(check_refcount(curthread) == OK)
    {
        we can still access any softc
    }
    else
    {
        just return, nothing more to do
    }
}
```

But in the general case I assume that one can just return immediately if one didn't get the memory that was needed, assuming that the destructor will clean up the softc, hence we are exiting in the middle of a program, with perhaps loose ends.

If the malloc() routine finds out that it has to do a lengthy memory search, then it can exit the callback arguments pointed to by "curthread->td_cba_p" and enter those arguments again, checking the refcount, before returning. This way we will never see a single line of mtx_lock/mtx_unlock in the code, in most cases.

Well, is it not better to have locking centralized, than decentralized: The functions that sleep must exit/enter the callback args by default.

So what do you think about this way of allowing us to hold locks while calling functions that can sleep?

Assuming there are "n" callbacks in the kernel. Then there is "2*n" too many lock/unlock lines in the kernel, than strictly necessary. That is a rough calculation. The number might be a little lower, hence not all callbacks need locking.

What do you think about having this "check some value before proceeding" also when returning from sleep, centralized?

—HPS

freebsd-hackers@freebsd.org mailing list

<http://lists.freebsd.org/mailman/listinfo/freebsd-hackers>

To unsubscribe, send any mail to "freebsd-hackers-unsubscribe@freebsd.org"