Additional Kernel Locking Options

Advanced Embedded Linux Development

with **Dan Walkes**

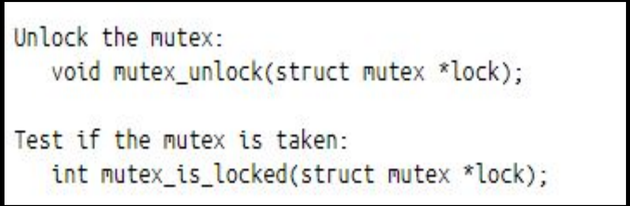
University of Colorado **Boulder**

Learning objectives:

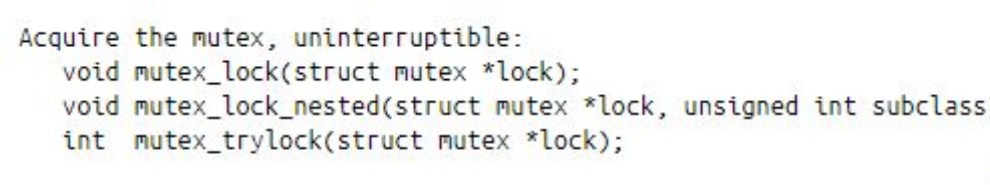
Introduce Kernel Mutexes, Reader/Writer Semaphores, and Completions

Mutex()

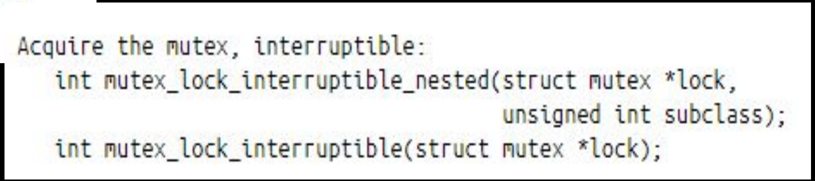
- Since the book was written, the kernel has added a mutex type for the purposes used in scull
  - See https://elixir.bootlin.com/linux/v5.3.6/source/Documentation/locking/mutex-design.rst and http://lwn.net/Articles/164802/



```
Unlock the mutex:
   void mutex_unlock(struct mutex *lock);

Test if the mutex is taken:
   int mutex_is_locked(struct mutex *lock);
```



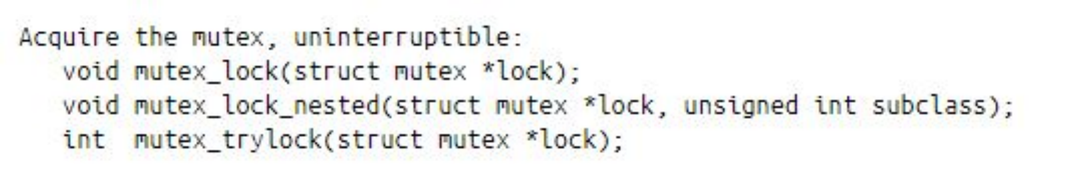
```
Acquire the mutex, uninterruptible:
   void mutex_lock(struct mutex *lock);
   void mutex_lock_nested(struct mutex *lock, unsigned int subclass);
   int  mutex_trylock(struct mutex *lock);
```



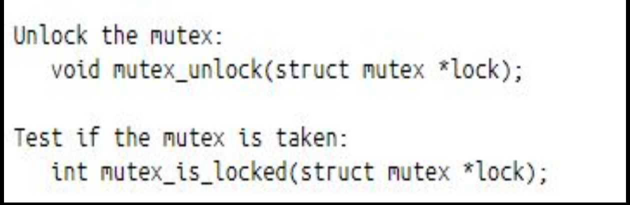
```
Acquire the mutex, interruptible:
   int mutex_lock_interruptible_nested(struct mutex *lock,
                                       unsigned int subclass);
   int mutex_lock_interruptible(struct mutex *lock);
```

Linux Device Drivers 3rd Edition Chapter 5

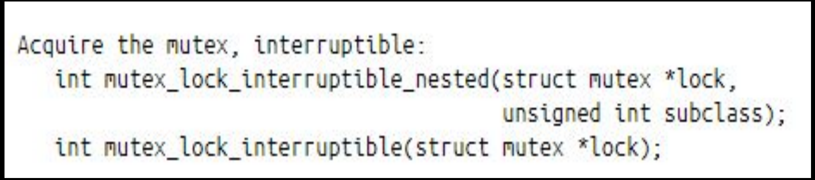
Mutex() map to Semaphore



```
Acquire the mutex, uninterruptible:
   void mutex_lock(struct mutex *lock);
   void mutex_lock_nested(struct mutex *lock, unsigned int subclass);
   int  mutex_trylock(struct mutex *lock);
```



```
Unlock the mutex:
   void mutex_unlock(struct mutex *lock);

Test if the mutex is taken:
   int mutex_is_locked(struct mutex *lock);
```

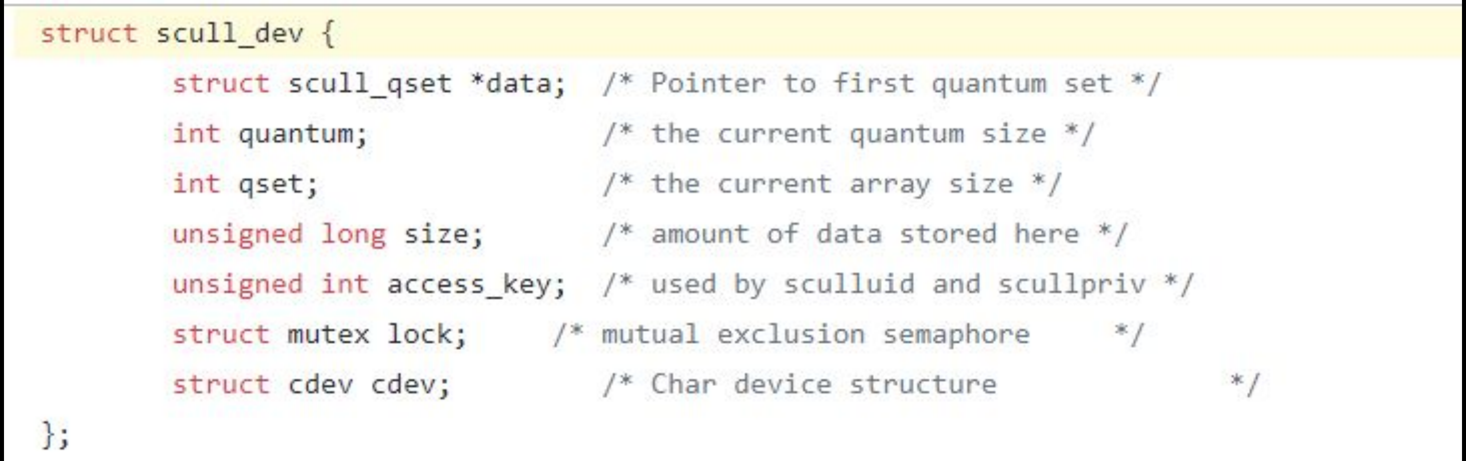


```
Acquire the mutex, interruptible:
   int mutex_lock_interruptible_nested(struct mutex *lock,
                                       unsigned int subclass);
   int mutex_lock_interruptible(struct mutex *lock);
```

- mutex_lock -> down
- mutex_trylock -> down_trylock
- mutex_lock_interruptible -> down_interruptible
- mutex_unlock -> up
- nested -> used for multiple locks, ordering between

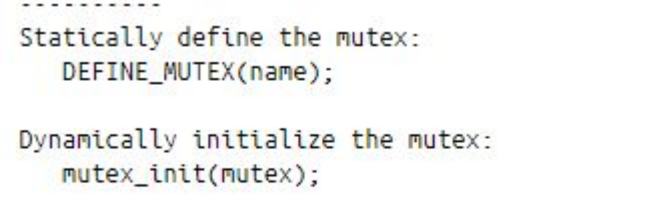
Linux Device Drivers 3rd Edition Chapter 5
https://elixir.bootlin.com/linux/latest/source/Documentation/locking/mutex-design.txt
https://elixir.bootlin.com/linux/latest/source/Documentation/locking/lockdep-design.txt

Mutex use in scull



```
struct scull_dev {
        struct scull_qset *data;  /* Pointer to first quantum set */
        int quantum;              /* the current quantum size */
        int qset;                 /* the current array size */
        unsigned long size;       /* amount of data stored here */
        unsigned int access_key;  /* used by sculluid and scullpriv */
        struct mutex lock;      /* mutual exclusion semaphore     */
        struct cdev cdev;         /* Char device structure              */
};
```

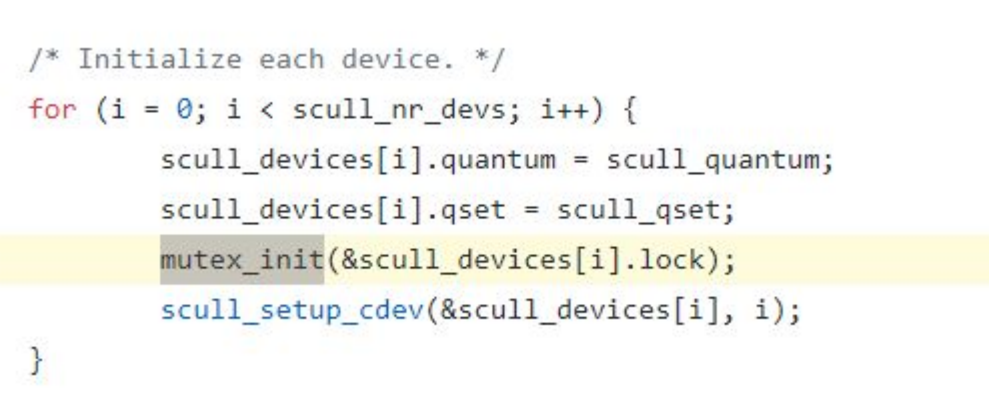
- Which initialization method (DEFINE_MUTEX or mutex_init()) would you guess scull uses?
  - ○ mutex_init() runtime



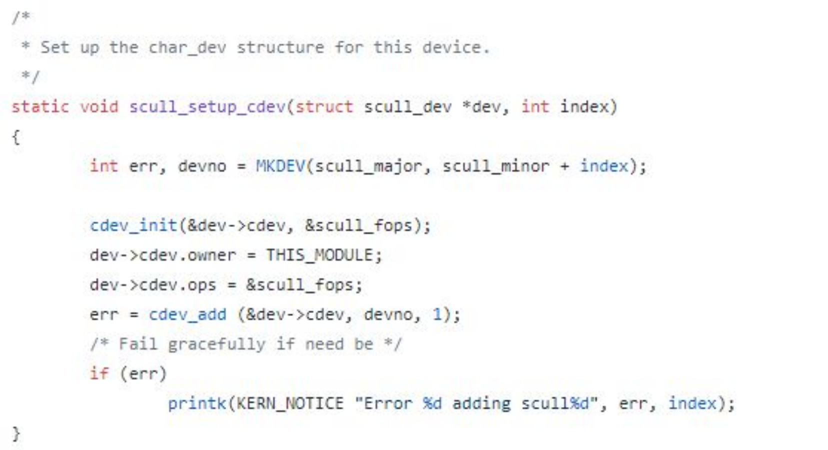
```
-----------
Statically define the mutex:
   DEFINE_MUTEX(name);

Dynamically initialize the mutex:
   mutex_init(mutex);
```

Linux Device Drivers 3rd Edition Chapter 5
https://github.com/cu-ecen-5013/ldd3/blob/master/scull/scull.h#L87

Mutex Use in Scull



```
/* Initialize each device. */
for (i = 0; i < scull_nr_devs; i++) {
        scull_devices[i].quantum = scull_quantum;
        scull_devices[i].qset = scull_qset;
        mutex_init(&scull_devices[i].lock);
        scull_setup_cdev(&scull_devices[i], i);
}
```

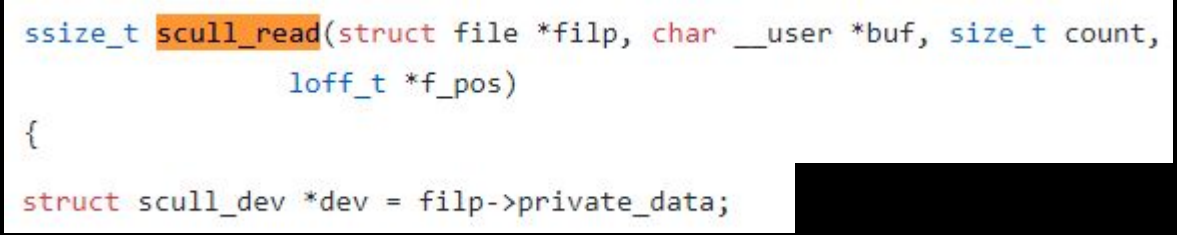


```
/*
 * Set up the char_dev structure for this device.
 */
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
        int err, devno = MKDEV(scull_major, scull_minor + index);

        cdev_init(&dev->cdev, &scull_fops);
        dev->cdev.owner = THIS_MODULE;
        dev->cdev.ops = &scull_fops;
        err = cdev_add (&dev->cdev, devno, 1);
        /* Fail gracefully if need be */
        if (err)
                printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

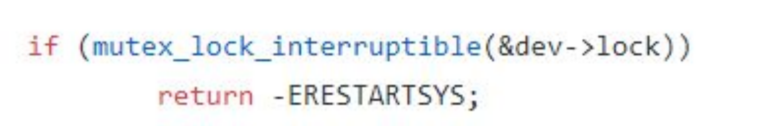
- What does mutex_init necessarily happen before?
  - Notifying the kernel about the object in cdev_add()
    - “No object can be made available to the kernel before it can function properly”
  - Happens during module_init() function

Linux Device Drivers 3rd Edition Chapter 5
https://github.com/cu-ecen-5013/ldd3/blob/master/scull/main.c#L653

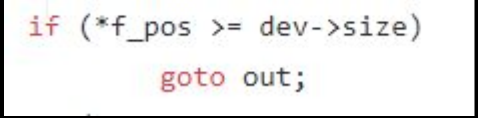
Handling Mutex Lock Failures



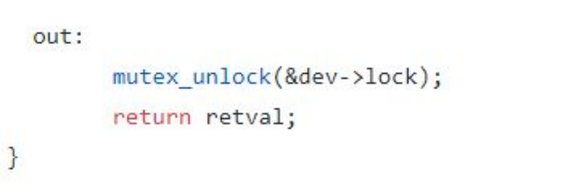
```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count,
                loff_t *f_pos)
{

struct scull_dev *dev = filp->private_data;
```



```
if (mutex_lock_interruptible(&dev->lock))
        return -ERESTARTSYS;
```



```
if (*f_pos >= dev->size)
        goto out;
```



```
out:
        mutex_unlock(&dev->lock);
        return retval;
}
```

- Use -ERESTARTSYS when retrying the operation is the right thing to do (undo any user visible changes)
- Use -EINTR when you can’t undo previous changes
- Balance each lock with unlock (including error case)

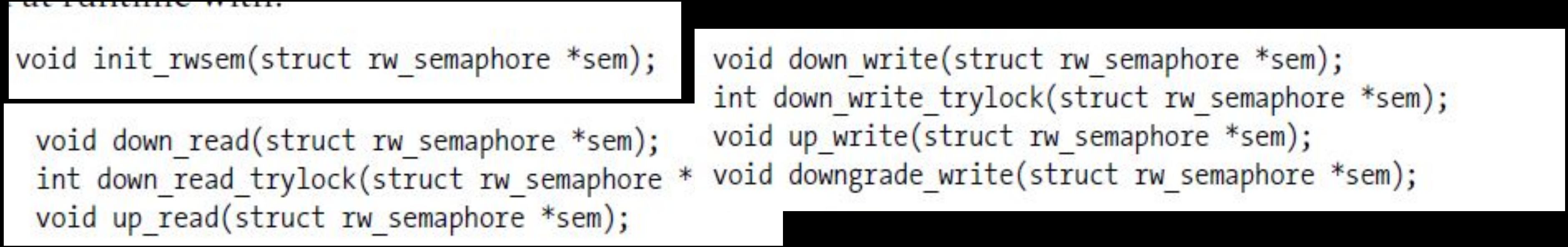
Linux Device Drivers 3rd Edition Chapter 5
https://github.com/cu-ecen-5013/ldd3/blob/master/scull/main.c#L293

Reader/Writer Semaphores

- How many threads can read a non-atomic variable concurrently when writes to the variable are blocked?
  - Infinite
- How many threads can read a non-atomic variable concurrently when writes to the variable are occurring?
  - 0
- Does every thread need an exclusive lock to access a variable?
  - Only writer threads need exclusive access.
  - An infinite number of reader threads can access as long as writer threads are blocked.

Linux Device Drivers 3rd Edition Chapter 5

Reader/Writer Semaphores



```
void init_rwsem(struct rw_semaphore *sem);
```

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *
void up_read(struct rw_semaphore *sem);
```

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

- down_write_trylock/down_read_trylock return 1 instead of 0 on success (different from convention)
- downgrade_write converts a write lock to a read lock
- Writers get priority
- Use when write access is rarely required, held briefly.

Linux Device Drivers 3rd Edition Chapter 5

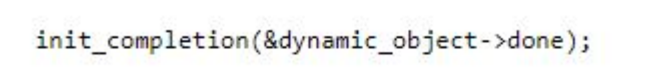
Completions

- Wait for some activity to complete
- Can use a semaphore for this.
    - Semaphores are optimized for “available” case
- Completions designed for the “not available” case
    - Will be the case when you just started a thread and want to wait for it to complete.
- complete/complete_all() support use in interrupt handlers.

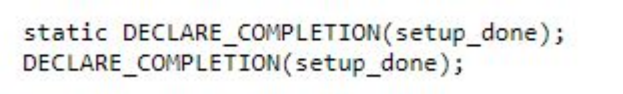
Linux Device Drivers 3rd Edition Chapter 5
https://www.kernel.org/doc/Documentation/scheduler/completion.txt

Completions

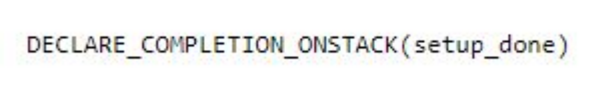
- “Any time you think of using yield() or some quirky msleep(1) loop to allow something else to proceed, you probably want to look into using one of the wait_for_completion*() calls and complete() instead.”



```
init_completion(&dynamic_object->done);
```

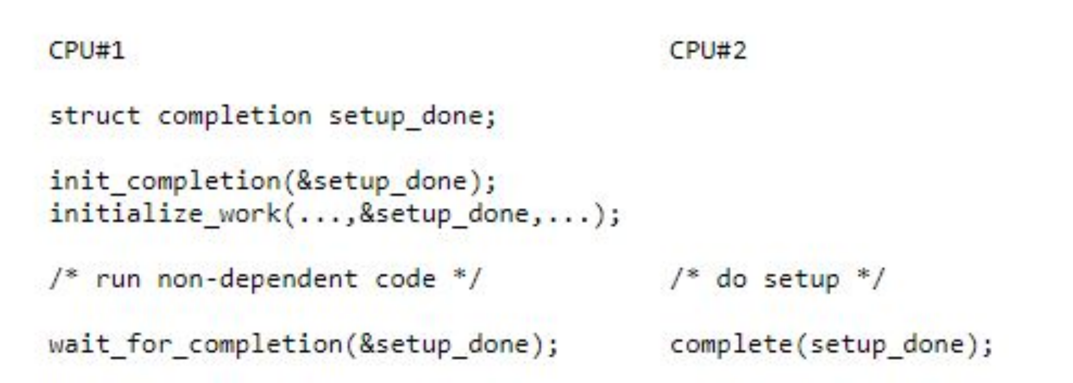


```
static DECLARE_COMPLETION(setup_done);
DECLARE_COMPLETION(setup_done);
```



```
DECLARE_COMPLETION_ONSTACK(setup_done)
```

Worker Thread



```
CPU#1                                    CPU#2

struct completion setup_done;

init_completion(&setup_done);
initialize_work(...,&setup_done,...);

/* run non-dependent code */             /* do setup */

wait_for_completion(&setup_done);        complete(setup_done);
```

Define and initialize the completion

Pass it to a worker that runs in a different thread

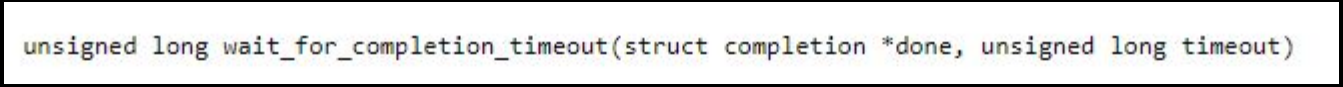
Wait for the worker to signal completion

Linux Device Drivers 3rd Edition Chapter 5

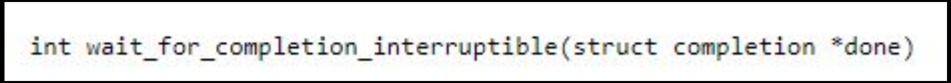
https://www.kernel.org/doc/Documentation/scheduler/completion.txt

Completions

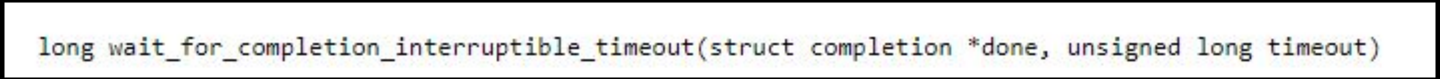
- Wait for completion variants



```
unsigned long wait_for_completion_timeout(struct completion *done, unsigned long timeout)
```

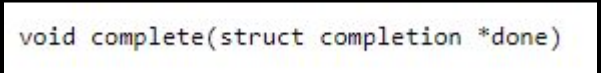


```
int wait_for_completion_interruptible(struct completion *done)
```

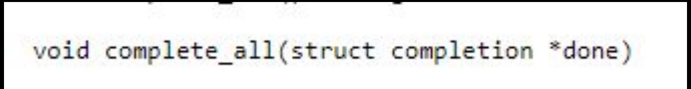


```
long wait_for_completion_interruptible_timeout(struct completion *done, unsigned long timeout)
```

- Completion Variants
  - complete_all - current *and* future waiters



```
void complete(struct completion *done)
```



```
void complete_all(struct completion *done)
```

Linux Device Drivers 3rd Edition Chapter 5
https://www.kernel.org/doc/Documentation/scheduler/completion.txt