Bryan Arnold

CSE 4300

Programming Assignment 4

5/2/18

The changes that I made include the following. First, in synch.h I added two components to the lock struct. First, I added volatile int lock_acquired; to have a value to symbolize whether a lock is available. If the lock isn't taken, it is 0, if it is, it is 1. The next component I added to the lock struct was volatile struct thread *lock_acquire;. This was to symbolize the thread that is currently holding the lock. By adding these to the lock struct definition, it is easy to track when a lock is held or not, and which thread is currently holding it.

The next change I made was in synch.c. The first thing I added was in the lock_create function. I added initializations to the values I added to the lock struct in synch.h. lock->lock_acquired = 0; since the lock by default isn't being used, and lock->lock_acquire = NULL; since no thread is currently holding the lock.

For the lock_destroy function, I just added one statement after the kfree() calls to the lock name and lock. This is a just incase if the kfree(lock) doesn't set free the lock.

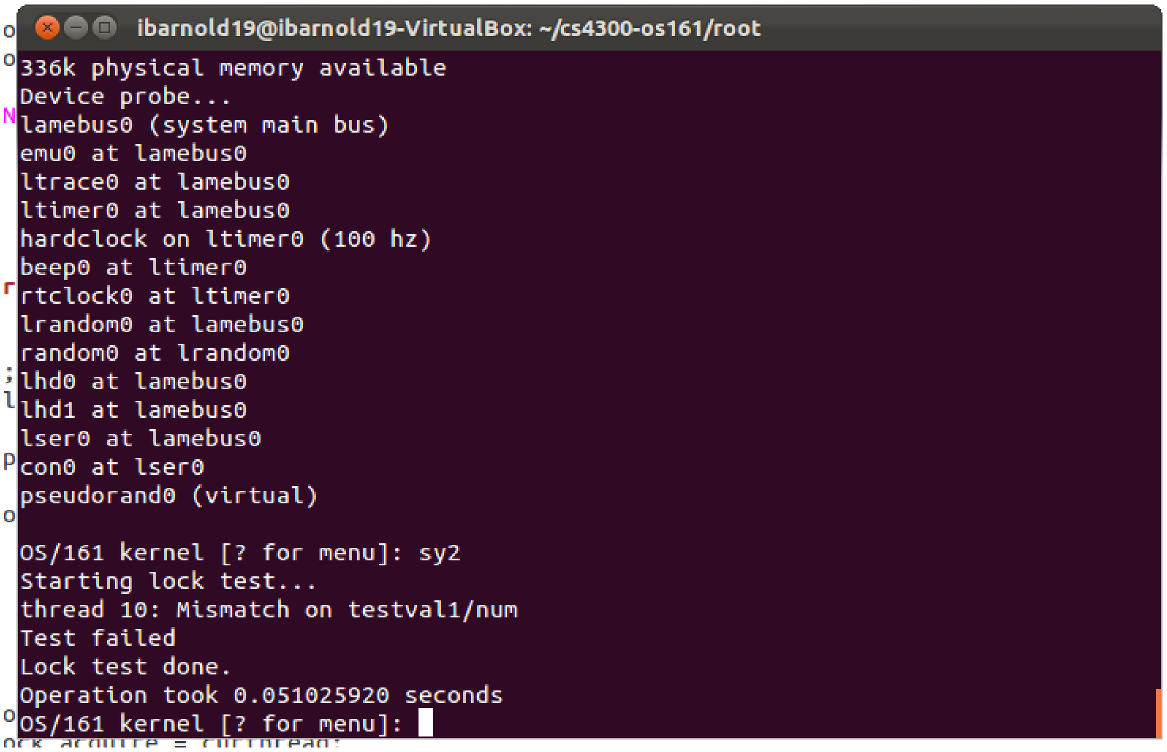
For void_acquire function, first I disabled all interrupts to allow for the threads to approach the lock. Then, I set up a while loop to check whether the lock is being held (lock->lock_acquired == 1), as well as if the current thread is the one holding the lock (lock->lock_acquire). If the second condition isn't true and the first is true, make the thread sleep indefinitely until it is woken up. After this loop, a thread can acquire the lock. I set the lock struct's members to lock->lock_acquired = 1;, to indicate it is taken, as well as setting the thread holding the lock to the current thread, lock->lock_acquire = curthread;. Finally, I reenable all interrupts.

For void_release function, first I disabled all interrupts to allow for the threads to release the lock. I set the lock's members to lock->lock_acquired = 0;, to show the lock isn't being held, and lock->lock_acquire = NULL; to indicate a thread isn't holding the lock. I then call thread_wakeup(lock) on the lock to wake up the next thread sleeping that wants to acquire the lock. Finally, I reenable all interrupts.

For the lock_do_i_hold function, all I do I simply return the value of the statement lock->lock_acquire == curthread. If this is 1, the current thread has the lock, if it is 0, the current thread doesn't have the lock.

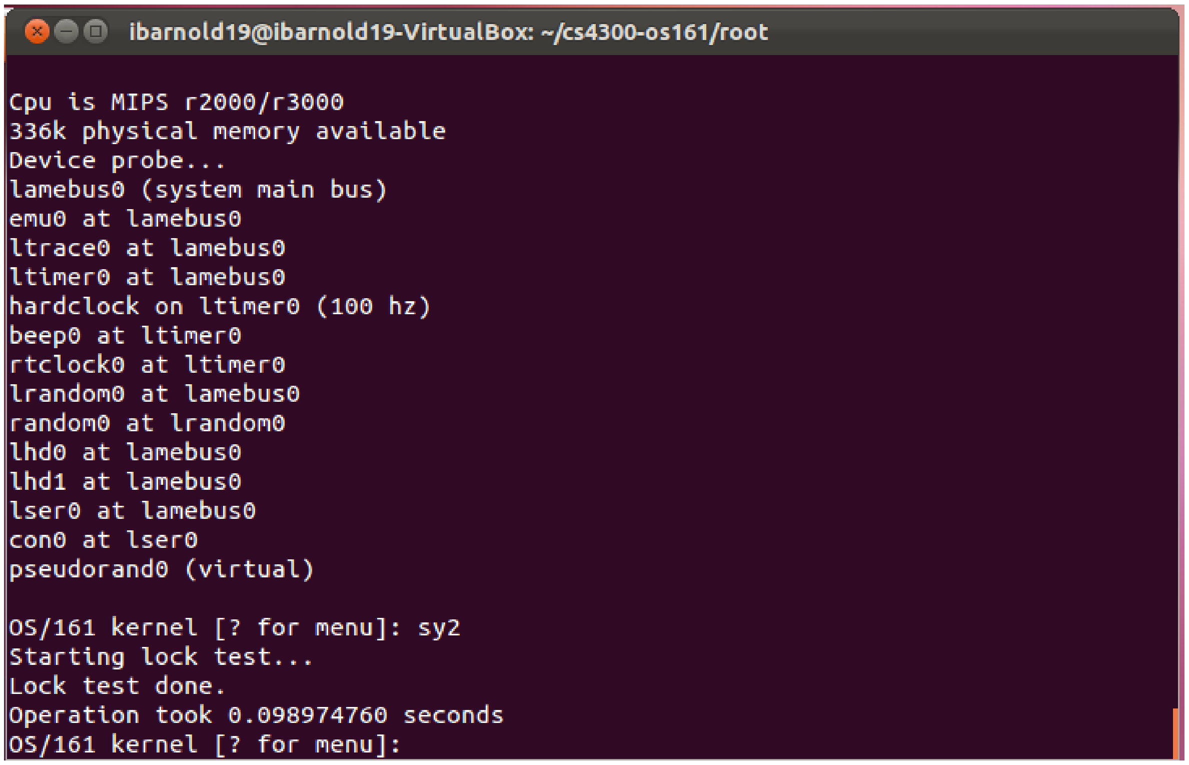
I didn't add any changes to the semaphore components of synch.h and synch.c as they weren't part of the assignment, as well as any condition variable components as they weren't part of the assignment. The assignment was only to implement sleeping locks.

Here are some screen shots of the out of running the tests for the lock. This is what happened when my implementation failed:


```
336k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
hardclock on ltimer0 (100 hz)
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0
pseudorand0 (virtual)

OS/161 kernel [? for menu]: sy2
Starting lock test...
thread 10: Mismatch on testval1/num
Test failed
Lock test done.
Operation took 0.051025920 seconds
OS/161 kernel [? for menu]:
```

This is when my lock implementation was successful:


```
Cpu is MIPS r2000/r3000
336k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
hardclock on ltimer0 (100 hz)
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0
pseudorand0 (virtual)

OS/161 kernel [? for menu]: sy2
Starting lock test...
Lock test done.
Operation took 0.098974760 seconds
OS/161 kernel [? for menu]:
```

This occurred when I implemented correctly as well as when I commented out the sections of code that allowed lock to work before code was put in (like (void) lock;).

Here is the source code for the files that I changed:

synch.h

```
/*
 * Header file for synchronization primitives.
 */

#ifndef _SYNCH_H_
#define _SYNCH_H_

/*
 * Dijkstra-style semaphore.
 * Operations:
 *     P (proberen): decrement count. If the count is 0, block until
 *                   the count is 1 again before decrementing.
 *     V (verhogen): increment count.
 *
 * Both operations are atomic.
 *
 * The name field is for easier debugging. A copy of the name is made
 * internally.
 */

struct semaphore {
     char *name;
     volatile int count;
};

struct semaphore *sem_create(const char *name, int initial_count);
void              P(struct semaphore *);
void              V(struct semaphore *);
void              sem_destroy(struct semaphore *);


/*
 * Simple lock for mutual exclusion.
 * Operations:
 *    lock_acquire - Get the lock. Only one thread can hold the lock
at the
 *                   same time.
 *    lock_release - Free the lock. Only the thread holding the lock
may do
 *                   this.
 *    lock_do_i_hold - Return true if the current thread holds the
lock;
 *                   false otherwise.
 *
 * These operations must be atomic. You get to write them.
 *
 * When the lock is created, no thread should be holding it. Likewise,
 * when the lock is destroyed, no thread should be holding it.
 *
 * The name field is for easier debugging. A copy of the name is made
 * internally.
 */

struct lock {

     char *name;
     volatile int lock_acquired; //This will be 0 if the lock isn't
taken, and 1 if it is taken.
     volatile struct thread *lock_acquire; //This thread is when a
lock is being held by a thread.

};

struct lock *lock_create(const char *name);
void         lock_acquire(struct lock *);
void         lock_release(struct lock *);
int          lock_do_i_hold(struct lock *);
void         lock_destroy(struct lock *);


/*
 * Condition variable.
 *
 * Note that the "variable" is a bit of a misnomer: a CV is normally
used
 * to wait until a variable meets a particular condition, but there's
no
 * actual variable, as such, in the CV.
 *
 * Operations:
 *    cv_wait      - Release the supplied lock, go to sleep, and,
after
 *                   waking up again, re-acquire the lock.
 *    cv_signal    - Wake up one thread that's sleeping on this CV.
 *    cv_broadcast - Wake up all threads sleeping on this CV.
 *
 * For all three operations, the current thread must hold the lock
passed
 * in. Note that under normal circumstances the same lock should be
used
 * on all operations with any particular CV.
 *
 * These operations must be atomic. You get to write them.
 *
 * These CVs are expected to support Mesa semantics, that is, no
 * guarantees are made about scheduling.
 *
 * The name field is for easier debugging. A copy of the name is made
 * internally.
 */

struct cv {
	char *name;
	// add what you need here
	// (don't forget to mark things volatile as needed)
};

struct cv *cv_create(const char *name);
void       cv_wait(struct cv *cv, struct lock *lock);
void       cv_signal(struct cv *cv, struct lock *lock);
void       cv_broadcast(struct cv *cv, struct lock *lock);
void       cv_destroy(struct cv *);

#endif /* _SYNCH_H_ */
```

synch.c

```
/*
 * Synchronization primitives.
 * See synch.h for specifications of the functions.
 */

#include <types.h>
#include <lib.h>
#include <synch.h>
#include <thread.h>
#include <curthread.h>
#include <machine/spl.h>

////////////////////////////////////////////////////////////
//
// Semaphore.

struct semaphore *
sem_create(const char *namearg, int initial_count)
{
	struct semaphore *sem;

	assert(initial_count >= 0);

	sem = kmalloc(sizeof(struct semaphore));
	if (sem == NULL) {
		return NULL;
	}
```

```
      sem->name = kstrdup(namearg);
      if (sem->name == NULL) {
            kfree(sem);
            return NULL;
      }

      sem->count = initial_count;
      return sem;
}

void
sem_destroy(struct semaphore *sem)
{
      int spl;
      assert(sem != NULL);

      spl = splhigh();
      assert(thread_hassleepers(sem)==0);
      splx(spl);

      /*
       * Note: while someone could theoretically start sleeping on
       * the semaphore after the above test but before we free it,
       * if they're going to do that, they can just as easily wait
       * a bit and start sleeping on the semaphore after it's been
       * freed. Consequently, there's not a whole lot of point in
       * including the kfrees in the splhigh block, so we don't.
       */

      kfree(sem->name);
      kfree(sem);
}

void
P(struct semaphore *sem)
{
      int spl;
      assert(sem != NULL);

      /*
       * May not block in an interrupt handler.
       *
       * For robustness, always check, even if we can actually
       * complete the P without blocking.
       */
      assert(in_interrupt==0);

      spl = splhigh();
      while (sem->count==0) {
            thread_sleep(sem);
      }
      assert(sem->count>0);
      sem->count--;
      splx(spl);
}

void
V(struct semaphore *sem)
{
      int spl;
      assert(sem != NULL);
      spl = splhigh();
      sem->count++;
      assert(sem->count>0);
      thread_wakeup(sem);
      splx(spl);
}

////////////////////////////////////////////////////////////
//
// Lock.

struct lock *
lock_create(const char *name)
{
      struct lock *lock;

      lock = kmalloc(sizeof(struct lock));
      if (lock == NULL) {
            return NULL;
      }

      lock->name = kstrdup(name);
      if (lock->name == NULL) {
            kfree(lock);
            return NULL;
      }

      //Initialize the values of the lock struct to default values

      lock->lock_acquired = 0; //Lock isn't being used, so 0.
      lock->lock_acquire = NULL; //Lock currently isn't held by any
thread or process.

      return lock;

}

void
lock_destroy(struct lock *lock)
{
      assert(lock != NULL);

      kfree(lock->name);
      kfree(lock);

      lock = NULL; //Back up destruction, just incase kfree() manages
to fail. Nothing happens if klfree                     succeeds.

}

void
lock_acquire(struct lock *lock)
{

      int spl;
      assert(lock != NULL);

      spl = splhigh(); //Disable all interrupts

      while(lock->lock_acquired == 1 && lock->lock_acquire !=
curthread){ //If the lock is held and it isn't
                           the current thread...

            thread_sleep(lock); //...make the thread sleep

      }


      lock->lock_acquired = 1; //Thread acquired lock
      lock->lock_acquire = curthread; //The current thread has the lock

      splx(spl); //Reenable all interrupts

      //(void)lock;

}

void
lock_release(struct lock *lock)
{

      int spl;
      assert(lock != NULL);

      spl = splhigh(); //Disable all interrupts

      lock->lock_acquired = 0; //Lock isn't held by the thread anymore
      lock->lock_acquire = NULL; //The current thread no longer holds
the lock, no thread does

      thread_wakeup(lock); //Wakeup the next thread waiting to get the
lock

      splx(spl); //Reenable all interrupts
      //(void)lock;

}

int
lock_do_i_hold(struct lock *lock)
{

      return (lock->lock_acquire == curthread); //If the current thread
holds the lock, return true

      //(void)lock;
      //return lock;

}

////////////////////////////////////////////////////////////
//
// CV


struct cv *
cv_create(const char *name)
{
      struct cv *cv;

      cv = kmalloc(sizeof(struct cv));
      if (cv == NULL) {
            return NULL;
      }

      cv->name = kstrdup(name);
      if (cv->name==NULL) {
            kfree(cv);
            return NULL;
      }

      // add stuff here as needed

      return cv;
}

void
cv_destroy(struct cv *cv)
{
      assert(cv != NULL);

      // add stuff here as needed

      kfree(cv->name);
      kfree(cv);
}
```

```
void
cv_wait(struct cv *cv, struct lock *lock)
{
      // Write this
      (void)cv;    // suppress warning until code gets written
      (void)lock;  // suppress warning until code gets written
}

void
cv_signal(struct cv *cv, struct lock *lock)
{
      // Write this
      (void)cv;    // suppress warning until code gets written
      (void)lock;  // suppress warning until code gets written
}

void
cv_broadcast(struct cv *cv, struct lock *lock)
{
      // Write this
      (void)cv;    // suppress warning until code gets written
      (void)lock;  // suppress warning until code gets written
}
```

synchtest.c

```
/*
 * Synchronization test code.
 */

#include <types.h>
#include <lib.h>
#include <synch.h>
#include <thread.h>
#include <test.h>
#include <clock.h>

#define NSEMLOOPS     63
#define NLOCKLOOPS    120
#define NCVLOOPS      5
#define NTHREADS      32

static volatile unsigned long testval1;
static volatile unsigned long testval2;
static volatile unsigned long testval3;
static struct semaphore *testsem;
static struct lock *testlock;
static struct cv *testcv;
static struct semaphore *donesem;

static
void
inititems(void)
{
	if (testsem==NULL) {
		testsem = sem_create("testsem", 2);
		if (testsem == NULL) {
			panic("synchtest: sem_create failed\n");
		}
	}
	if (testlock==NULL) {
		testlock = lock_create("testlock");
		if (testlock == NULL) {
			panic("synchtest: lock_create failed\n");
		}
	}
	if (testcv==NULL) {
		testcv = cv_create("testlock");
		if (testcv == NULL) {
			panic("synchtest: cv_create failed\n");
		}
	}
	if (donesem==NULL) {
		donesem = sem_create("donesem", 0);
		if (donesem == NULL) {
			panic("synchtest: sem_create failed\n");
		}
	}
}

static
void
semtestthread(void *junk, unsigned long num)
{
	int i;
	(void)junk;

	/*
	 * Only one of these should print at a time.
	 */
	P(testsem);
	kprintf("Thread %2lu: ", num);
	for (i=0; i<NSEMLOOPS; i++) {
		kprintf("%c", (int)num+64);
	}
	kprintf("\n");
	V(donesem);
}

int
```

```
semtest(int nargs, char **args)
{
	int i, result;

	(void)nargs;
	(void)args;

	inititems();
	kprintf("Starting semaphore test...\n");
	kprintf("If this hangs, it's broken: ");
	P(testsem);
	P(testsem);
	kprintf("ok\n");

	for (i=0; i<NTHREADS; i++) {
		result = thread_fork("semtest", NULL, i, semtestthread,
NULL);
		if (result) {
			panic("semtest: thread_fork failed: %s\n",
			      strerror(result));
		}
	}

	for (i=0; i<NTHREADS; i++) {
		V(testsem);
		P(donesem);
	}

	/* so we can run it again */
	V(testsem);
	V(testsem);

	kprintf("Semaphore test done.\n");
	return 0;
}

static
void
fail(unsigned long num, const char *msg)
{
	kprintf("thread %lu: Mismatch on %s\n", num, msg);
	kprintf("Test failed\n");

	lock_release(testlock);

	V(donesem);
	thread_exit();
}

static
void
locktestthread(void *junk, unsigned long num)
```

```
{
	int i;
	(void)junk;

	for (i=0; i<NLOCKLOOPS; i++) {
		lock_acquire(testlock);
		testval1 = num;
		testval2 = num*num;
		testval3 = num%3;

		if (testval2 != testval1*testval1) {
			fail(num, "testval2/testval1");
		}

		if (testval2%3 != (testval3*testval3)%3) {
			fail(num, "testval2/testval3");
		}

		if (testval3 != testval1%3) {
			fail(num, "testval3/testval1");
		}

		if (testval1 != num) {
			fail(num, "testval1/num");
		}

		if (testval2 != num*num) {
			fail(num, "testval2/num");
		}

		if (testval3 != num%3) {
			fail(num, "testval3/num");
		}

		lock_release(testlock);
	}
	V(donesem);
}


int
locktest(int nargs, char **args)
{
	int i, result;

	(void)nargs;
	(void)args;

	inititems();
	kprintf("Starting lock test...\n");

	for (i=0; i<NTHREADS; i++) {
		result = thread_fork("synchtest", NULL, i, locktestthread,
				      NULL);
		if (result) {
			panic("locktest: thread_fork failed: %s\n",
			      strerror(result));
		}
	}
	for (i=0; i<NTHREADS; i++) {
		P(donesem);
	}

	kprintf("Lock test done.\n");

	return 0;
}

static
void
cvtestthread(void *junk, unsigned long num)
{
	int i;
	volatile int j;
	time_t secs1, secs2;
	u_int32_t nsecs1, nsecs2;

	(void)junk;

	for (i=0; i<NCVLOOPS; i++) {
		lock_acquire(testlock);
		while (testval1 != num) {
			gettime(&secs1, &nsecs1);
			cv_wait(testcv, testlock);
			gettime(&secs2, &nsecs2);

			if (nsecs2 < nsecs1) {
				secs2--;
				nsecs2 += 1000000000;
			}

			nsecs2 -= nsecs1;
			secs2 -= secs1;

			/* Require at least 2000 cpu cycles (we're 25mhz) */
			if (secs2==0 && nsecs2 < 40*2000) {
			  kprintf("cv_wait took only %u ns\n", nsecs2);
			  kprintf("That's too fast... you must be "
				  "busy-looping\n");
			  V(donesem);
			  thread_exit();
			}
		}
		kprintf("Thread %lu\n", num);
```

```
            testval1 = (testval1 + NTHREADS - 1)%NTHREADS;

            /*
             * loop a little while to make sure we can measure the
             * time waiting on the cv.
             */
            for (j=0; j<3000; j++);

            cv_broadcast(testcv, testlock);
            lock_release(testlock);
      }
      V(donesem);
}

int
cvtest(int nargs, char **args)
{

      int i, result;

      (void)nargs;
      (void)args;

      inititems();
      kprintf("Starting CV test...\n");
      kprintf("Threads should print out in reverse order.\n");

      testval1 = NTHREADS-1;

      for (i=0; i<NTHREADS; i++) {
            result = thread_fork("synchtest", NULL, i, cvtestthread,
                                 NULL);
            if (result) {
                  panic("cvtest: thread_fork failed: %s\n",
                         strerror(result));
            }
      }
      for (i=0; i<NTHREADS; i++) {
            P(donesem);
      }

      kprintf("CV test done\n");

      return 0;
}
```