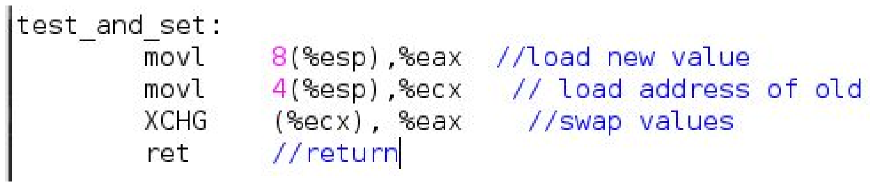
Jacob Anderson
ECE 492 Operating Systems
Project 3 Report

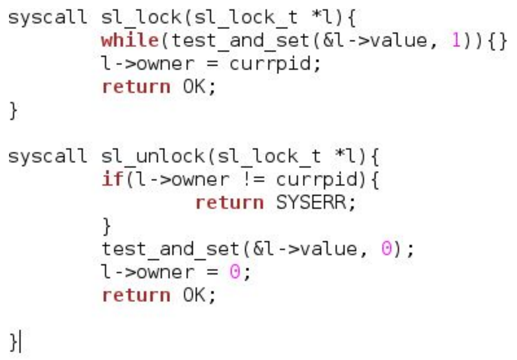
Test and Set



```
test_and_set:
        movl    8(%esp),%eax  //load new value
        movl    4(%esp),%ecx   // load address of old
        XCHG    (%ecx), %eax    //swap values
        ret     //return
```

The test and set function was very simple. The assembly code simply loads in the new value into eax and the address of the old into ecx and swaps the value at eax and the value at the address of ecx.

Spinlock



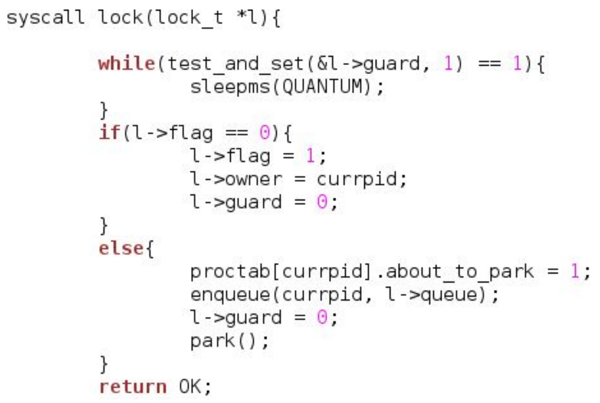
```
syscall sl_lock(sl_lock_t *l){
        while(test_and_set(&l->value, 1)){}
        l->owner = currpid;
        return OK;
}

syscall sl_unlock(sl_lock_t *l){
        if(l->owner != currpid){
                return SYSERR;
        }
        test_and_set(&l->value, 0);
        l->owner = 0;
        return OK;

}
```

Spinlock was also quite simple. The data structure sl_lock_t only has two entries: the value and the owner. When the lock is taken test and set sets the value to 1 and will only be set back to zero during the unlock. If the lock is taken and another thread tries to acquire it the thread will loop continuously in the while lock in sl_lock. Everything seems to work as intended.

Lock

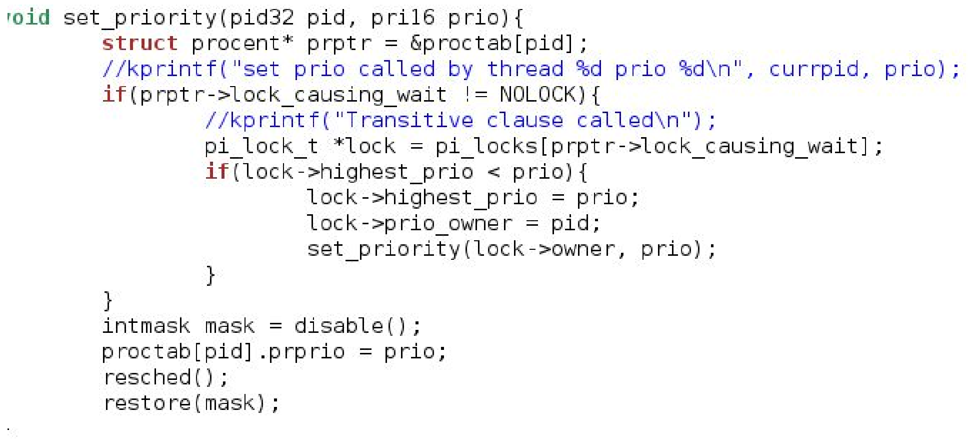


```
syscall lock(lock_t *l){

        while(test_and_set(&l->guard, 1) == 1){
                sleepms(QUANTUM);
        }
        if(l->flag == 0){
                l->flag = 1;
                l->owner = currpid;
                l->guard = 0;
        }
        else{
                proctab[currpid].about_to_park = 1;
                enqueue(currpid, l->queue);
                l->guard = 0;
                park();
        }
        return OK;
```

Lock implements some new features compared to spinlock. There is a guard that stops two threads from modifying the lock variables at the same time by using the test_and_set function. The lock also uses the park and unpark functions along with a about_to_park flag to allow the thread to be suspended while waiting to acquire a lock. This is possible by having the locks keep a queue of threads who want to acquire the lock and take them out in FIFO fashion. Everything seems to work as intended.

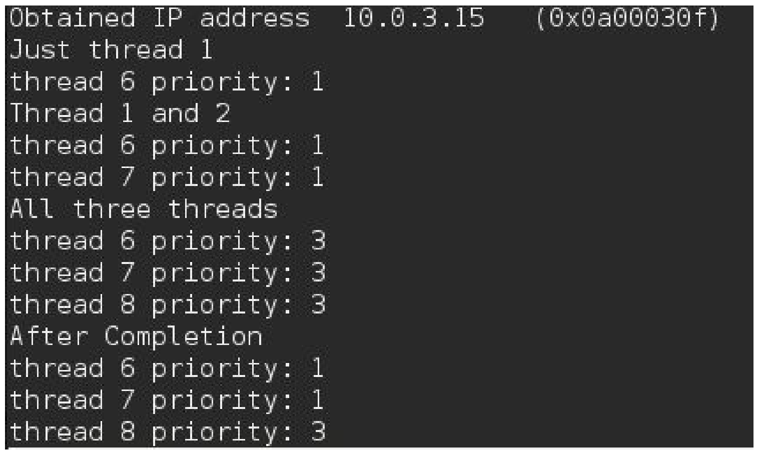
PiLock



```
oid set_priority(pid32 pid, pri16 prio){
        struct procent* prptr = &proctab[pid];
        //kprintf("set prio called by thread %d prio %d\n", currpid, prio);
        if(prptr->lock_causing_wait != NOLOCK){
                //kprintf("Transitive clause called\n");
                pi_lock_t *lock = pi_locks[prptr->lock_causing_wait];
                if(lock->highest_prio < prio){
                        lock->highest_prio = prio;
                        lock->prio_owner = pid;
                        set_priority(lock->owner, prio);
                }
        }
        intmask mask = disable();
        proctab[pid].prprio = prio;
        resched();
        restore(mask);
```

Pi lock keeps the same functionality as lock but adds priority inheritance. The pi_lock_t data structure has 3 new entries: its index in the pi_lock array, its highest priority and the thread that gave it its highest priority. Whenever a thread is queued and its priority is higher than the highest priority of the lock, those values are changed and the set priority is called to change the priority of the thread holding the lock. If that thread is waiting for another lock then its checked if that locks highest priority is lower than the passed priority. If it is that locks highest priority is set to the passed priority and the owner is set to the thread being set. Then setpriority is called recursively to check for locks owned by the third thread and so on until there is not a lock held. When a thread releases a lock and it was the holder of the highest priority then a function is called to find the next highest priority. All threads return to their original priority upon releasing a lock. Everything seems to work as intended.

Main-pi



```
Obtained IP address  10.0.3.15   (0x0a00030f)
Just thread 1
thread 6 priority: 1
Thread 1 and 2
thread 6 priority: 1
thread 7 priority: 1
All three threads
thread 6 priority: 3
thread 7 priority: 3
thread 8 priority: 3
After Completion
thread 6 priority: 1
thread 7 priority: 1
thread 8 priority: 3
```

The idea behind my test is that I would have three threads and two locks for them to share. First, Thread 1 acquires lock 1 then thread two acquires lock 2. Then lock 1 will attempt to acquire lock 2 and main will print the priorities of the two processes. At this point they should be unchanged. Then thread 3 will try to acquire lock 1 and this will change the priority of both thread 1 and thread 2 showing the transitive property of priority inheritance. Once the threads have relinquished their locks they will return to their original priority. This all seemed to work perfectly. Main-py was just left as main in the submission.