UCB CS 162 - Operating Systems and Systems Programming "Lecture 7 - Synchronization 2- Semaphores, Lock Implementation"

Lecture 7 - Synchronization 2: Semaphores, Lock Implementation

This lecture gives an intuition about how locks are actually implemented in an operating system. We want an interface which lets us acquire a lock on a critical section of code, run the critical section, and then release the lock.

The simple implementation is described by the acquire and release pseudocode given below -

```
int value = FREE;   // Flag describing if the critical section is being executed or not
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread to sleep;
        go to sleep();      // careful!
    }
    else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (thread is sleeping on this lock) {
        wake up thread;
    }
    else {
        value = FREE;
```

```
    }
    enable interrupts;
}
```

A small problem with this implementation is in the `Acquire()` code, where if the `value` is `BUSY`, we put the thread (that is trying to enter the critical section) to sleep, and then we go to sleep ourselves, without ever re-enabling interrupts!

This problem is solved by following the convention that whichever thread wakes up after we go to sleep re-enables interrupts.

Another problem with this implementation is that we have to run this code at the kernel level, since it involves disabling and enabling interrupts. This means that the user level code has to go into the kernel by making a system call in order to acquire and release a lock.

Making a system call is around 25x costlier than a normal function call. So if we have hundreds of threads all acquiring and releasing locks, this can make the computer unusably slow. Therefore, we need a locking implementation that can acquire and release locks without having to go into the kernel.