7 Semaphore Implementation

No existing hardware implements P&V directly. They all involve some sort of scheduling and it's not clear that scheduling decisions should be made in hardware (layering). Thus semaphores must be built up in software using some lower-level synchronization primitive provided by hardware.

Need a simple way of doing mutual exclusion in order to implement P's and V's. We could use atomic reads and writes, as in the "too much milk" problem, but these are very clumsy.

Uniprocessor solution: disable interrupts. Recall that the only way the dispatcher regains control is through interrupts or through explicit requests.

```
typedef struct {
    int count;
    queue q; /* queue of threads waiting on this semaphore */
} Semaphore;

void P(Semaphore s)
{
    Disable interrupts;
    if (s->count > 0) {
        s->count -= 1;
        Enable interrupts;
        return;
    }
    Add(s->q, current_thread);
    sleep(); /* re-dispatch */
    Enable interrupts;
}




void V(Semaphore s)
```

```
{
    Disable interrupts;
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* put thread on the ready queue */
    }
    Enable interrupts;
}
```

What do we do in a multiprocessor to implement P's and V's? Can't just turn off interrupts to get low-level mutual exclusion.

- Turn off all other processors?
- Use atomic read and write, as in "too much milk"?

In a multiprocessor, there will have to be busy-waiting at some level: can't go to sleep if don't have mutual exclusion.

Most CISC machines provide some sort of atomic *read-modify-write* instruction. Read existing value, store back in one atomic operation. E.g. Test and set (implemented initially by IBM, later by many others). Set value to one, but return OLD value. Use ordinary write to set back to zero.

Using test and set for mutual exclusion: It's like a binary semaphore in reverse, except that it doesn't include waiting. 1 means someone else is already using it, 0 means it's OK to proceed. Definition of test and set prevents two threads from getting a 0-to-1 transition simultaneously.

```
int lock;
..
while (TAS(&lock, 1) != 0);
..
critical section
..
```

lock = 0;

Modern RISC machines don't provide read-modify-write instructions. Instead, most of them provide a weaker mechanism that does not guarantee atomicity, but detects interference.

- *load-linked* instruction (ldl): Loads a word from memory and sets a per-processor flag associated with that word (usually stored in the cache).
- store operations to the same memory location (by any processor) reset all processor's flags associated with that word.
- *store-conditionally* instruction (stc): Stores a word iff the processor's flag for the word is still set; indicates success or failure.

Using ldl/stc for mutual exclusion:

```
int lock;
..
while (ldl(&lock) != 0 || !stc(&lock, 1));
..
critical section
..
lock = 0;
```

Using ldl/stc to implement semaphores in a multiprocessor: For each semaphore, keep an integer (lock) in addition to the semaphore integer and the queue of waiting threads.

```
typedef struct {
    int lock; /* initially 0 */
    int count;
    queue q; /* queue of threads waiting on this semaphore */
```

```
} Semaphore;

void P(Semaphore s)
{
    Disable interrupts;
    while (ldl(s->lock) != 0 || !stc(s->lock, 1));
    if (s->count > 0) {
        s->count -= 1;
        s->lock = 0;
        Enable interrupts;
        return;
    }
    Add(s->q, current_thread);
    s->lock = 0;
    sleep(); /* re-dispatch */
    Enable interrupts;
}



void V(Semaphore s)
{
    Disable interrupts;
    while (ldl(s->lock) != 0 || !stc(s->lock, 1));
    if (isEmpty(s->q)) {
        s->count += 1;
    } else {
        thread = RemoveFirst(s->q);
        wakeup(thread); /* put thread on the ready queue */
    }
    s->lock = 0;
    Enable interrupts;
}
```

Why do we still have to disable interrupts in addition to using the lock with ldl/stc?

Important point: implement some mechanism once, very carefully. Then always write programs that use that mechanism. Layering is very important.