Topic 7: Semaphore Implementation

Reading: 5.8
Next reading: 5.2-5.6

- No existing hardware implements P&V directly. They involve some sort of CPU scheduling and it's not clear that scheduling decisions should be made in hardware anyway. Thus semaphores must be implemented in software using a lower-level atomic operations 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. The only way the dispatcher regains control is through interrupts or direct invocation. Interrupts are not lost if they are off. Enable/disable routines probably in assembly code.

- Simple Solution:

```
typedef struct  {
    int value;
} Semaphore;

P(Semaphore *s) {
    while(1) {
        int_disable();
        if (s->value > 0) {
            s->value--;
            break;
        }
        int_enable(); //enabled, not restored
    }
    int_restore();
}

V(Semaphore *s) {
    int_disable();
    s->value++;
    int_restore();
}
```

- What happens if we move the int_disable/int_enable outside the loop?
- Note that P must enable interrupts, not simply restore them. Why?

Step 1: when count is 0, put process to sleep; on V just wake up everybody, processes all check count again. Be careful about putting to sleep: must not re-enable interrupts before process is blocked or V can sneak in!

Is this “busy-waiting”?

Step 2: label each process with semaphore it’s waiting for, then just wake up relevant processes.

Step 3: just wake up a single process. Maybe unfair.

Step 4: add a queue of waiting processes to the semaphore, wake up FCFS. Fair.

- On failed P, add to queue. On V, remove from queue.
- Queue could be implemented in process table by combining Steps 2 and 3 and keeping track of the order of the waiting processes.

```
typedef struct  {
    int count;
    queue q;
} Semaphore;

P(Semaphore *s) {
    while(1) {
        int_disable();
        if (s->count > 0) {
            s->count--;
            break;
        }
        Move process from ready queue to s->q
        int_restore();
        dispatcher();
    }
    int_restore();
}

V(Semaphore *s) {
    int_disable();
    s->count++;
    if (s->q not empty) {
        Move first process from s->q to ready queue
        dispatcher();
    }
    int_restore();
}
```