COMSW4118-10-1 10/5/2017

● Synchronization (cont.)

▶ Semaphore (binary):

```
P(s)  { while (s ≤ 0); s--; }
V(s)  { s++; }
```

How can we implement ~~this~~ locking with this, so we don't disable interrupts for whole critical section, but just lock/unlock.

```
Init:
s=1;

Lock(s) {
    disable_int();
    while (s<=0);
    s--;
    enable_int();
}

Unlock(s) {
    disable_int();
    s++;
    enable_int();
}
```

BUT: problem is Lock(s) will still spin w/ interrupts disabled, so in a single CPU this will block forever.

COULD be fixed by enable/disable interrupts in while loop (to ~~give~~ check for interrupts) (PIC could even buffer some) BUT better

```
Lock(s) {
  disable
  while (s≤0) {
    enable
    disable
  }
  s--;
  enable
}
```

```
Lock(s) {
wloop:  while (s <=0);
        disable_int();
        if (s>=0) {
            s--;
            enable_int();
        } else {
            enable_int();
            goto wloop;
        }
}
```

BUT BEST IS TO WAIT (Synchronization is omitted from the code here)

```
P(s) {
    s--;
    if (s<0) {
        add process to wait queue;
        block;
    }
}

V(s) {
    s++;
    if (s≤0) {
        T = remove from wait queue;
        wakeup(T);
    }
}
```

(!) BUT, generally speaking, the busy wait implementation is preferred in practice!

BECAUSE it is shorter & simpler & faster & works better w/ no contention.

NOTE: the wait queue is shared, so you must use another lock to access it!

(!) The waiting is useful in a situation when there is a lot of lock contention!

- Usually, when you see the term semaphore, it usually refers to the blocking semaphore or mutex. ← ~~Expe~~ in Linux kernel
- The busy wait is referred to as spinlock.

▶ Linux prevents you from holding a spinlock if you need to block/sleep (i.e. can't call kmalloc while holding a spinlock) → then use a semaphore.

▶ **Read/Write Locks** - allow multiple readers, but only one writer (and no readers).

- Implementation w/ ordinary locks: use read_lock (for readers), use writelock (for writer)

```
read_lock(L) {
    lock(c);
    count++;
    if (count==1) {
        lock(L);
    }
    unlock(C);
}

read_unlock(L) {
    Lock(C);
    count --;
    if (count==0) {
        unlock (L);
    }
    unlock (C);
}

writelock(L) {
    Lock(L);
}

write_unlock {
    unlock(L);
}
```

NOTE: Writers can get starved out in this solution.

■