UNIVERSITY OF CALIFORNIA AT BERKELEY
College of Engineering
Department of Electrical Engineering and Computer Sciences
Computer Science Division

CS 162
Fall 1988

John Ousterhout

Topic 5: Semaphore Implementation

- Readings for this topic:
  - Toy OS notes, Chapter 4.
  - Silberschatz/Peterson: Section 3.4.2.

- 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. *Remind that the only way the dispatcher regains control is through interrupts or through explicit process requests. Where there is parenthetical stuff below, leave space on board, add parenthetical stuff in when eliminating busy-waiting. Blank lines get filled in with multiprocessor solution.*

```
typedef struct {
    int count;
```

```
    (queue q;)
    (1 line of space here)
    } SEMAPHORE;

P(s)
SEMAPHORE *s;
{
    while (1) {
        Disable interrupts;
        (1 line of space here)
        if (s->count > 0) {
            s->count-= 1;
            (1 line of space here)
            Enable interrupts;
            return;
        }
        (Add process to s->q)
        Enable interrupts;
        (redispatch)
    }
    (1 line of space here)
}
```

```
V(s)
SEMAPHORE *s;
{
    Disable interrupts;
    (1 line of space here)
    (if (s->q empty) {)
    s->count += 1;
    (} else {)
        (Remove first process from s->q)
        (Wake it up)
    (})
    (1 line of space here)
    Enable interrupts;
}
```

- *Lecture 9 ended here.*

- What is wrong with this code? *Busy-waiting, may deadlock if process priorities are wrong.*

- *In the modifications below make sure that the final result ends up looking as close as possible to the multiprocessor solution that will be explained next.*

- Step 1: when P fails, put process to sleep; on V just wakeup everybody, processes all try P again. *Be careful about putting to sleep: must not re-enable interrupts or V can sneak in!*

- Is this ‘‘busy-waiting’’? *Yes and no. V is expensive if there are many processes waiting, but eventually they’ll all go back to sleep again. If*

nothing else happens in the system, then this process won't use up any resources. On the other hand, the process could use an arbitrarily large amount of resources before it finally finishes (of course, that's unlikely).

- Step 2: label each process with semaphore it's waiting for, then just wakeup relevant processes.

- Step 3: just wakeup a single process.

- Step 4: add a queue of waiting processes to the semaphore. On failed P, add to queue. On V, remove from queue. *Eliminate while loop.*

- There are several tradeoffs implicit here: how many processes in the system, how much queueing on semaphores, storage requirements, etc. The most important thing is to avoid busy-waiting.

- *The solution used here for semaphores is almost identical to what's in the Toy Operating System synchronization kernel.*

- 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 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 processes from getting a 0-to-1 transition simultaneously.
- Test and set is tricky to use, since you can't get at it from HLLs. Typically, use a routine written in assembler.

- Read-modify-writes may be implemented directly in memory hardware (e.g. IBM S/360), or in the processor by refusing to release the memory bus (PDP-11). *Has to be done in some central place, so it's not possible for somebody to sneak around the back.*

- Using test and set to implement semaphores in a multiprocessor: For each semaphore, keep a test-and-set integer in addition to the semaphore integer and the queue of waiting processes.

```
typedef struct {
    int count;
    queue q;
    int t;
    } SEMAPHORE;

P(s)
SEMAPHORE *s;
{
    Disable interrupts;
```

```
        while (TAS(s->t) != 0) /* do nothing */;
        if (s->count > 0){
            s->count = s->count-1;
            s->t = 0;
            Enable interrupts;
            return;
        }
        Add process to s->q;
        s->t = 0;
        Redispatch;
    }

    V(s)
    SEMAPHORE *s;
    {
        Disable interrupts;
        while (TAS(s->t) != 0) /* do nothing */;
        if (s->q empty) {
            s->count += 1;
        } else {
            Remove first process from s->q;
            Wake it up;
        }
        s->t = 0;
        Enable interrupts;
    }
```

- Why do we still have to disable interrupts in addition to using test and set?

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

- *Lecture 10 ended here.*