

CS 414 : Operating Systems

UNIVERSITY OF VIRGINIA
Department of Computer Science

Spring 2008

Topic 6: Semaphore Implementation

- Readings for this topic: Section 6.5.2 and 6.4
- No existing hardware implements P&V directly. Why?
- 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. Any drawbacks?
- Uniprocessor solution: disable interrupts.

```
typedef struct {  
    int count;  
} SEMAPHORE;  
  
P(s)  
SEMAPHORE *s;  
  
    while (true) {  
        Disable interrupts;  
        if (s->count > 0) {  
            s->count -= 1;  
            Enable interrupts;  
            return;  
        }  
        Enable interrupts;  
    }  
}
```

V(s)

*SEMAPHORE *s;*

Disable interrupts;

s->count += 1;

Enable interrupts;

- What is wrong with this code?
- Modification: add a queue of waiting processes to the semaphore. On failed P, add to queue. On V, remove from queue.

typedef struct {

int count;

queue q;

} SEMAPHORE;

P(s)

*SEMAPHORE *s;*

{

while (true) {

Disable interrupts;

if (s->count > 0) {

s->count -= 1;

Enable interrupts;

return;

}

Add process to s->q

Enable interrupts;

Redispatch

}

}

```

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

```

- Is this solution correct?
- 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. Why not?
- Is busy-waiting unavoidable in multiprocessor systems?
- Most machines provide some sort of atomic *read-modify-write* instruction. Read existing value, store back in one atomic operation.
 - 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.

- Read-modify-writes may be implemented directly in memory hardware, or in the processor by refusing to release the memory bus.
- 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;
}

```

- Is this solution correct?
- Is it busy-waiting?
- Why do we still have to disable interrupts in addition to using test and set?
- What if we change the order of Disable interrupts and while (TAS (s->t) != 0)?
- Important point: implement some mechanism once, very carefully. Then always write programs that use that mechanism. Layering is very important.