8 Lecture: More Synchronization

Outline:
- Announcements
- LWP?
  - From Last time: Busywaiting
  - Onwards: Busy Waiting: With Hardware Support
  - Reflection
    - sleep() and wakeup()
- Synchronization without busy waiting
- Semaphores
  - Monitors: (Hoare 1974, Brinch Hansen 1975)
- More Interprocess Communication

8.1 Announcements

- Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

- MEANINGFUL SYMBOLS (not: SIXTEEN)
- Don’t just click the little ‘X’ to stop minix
- function pointers
- schedulers
- context structure
- tryAsgn2
- Lab02: Distributed version should work for qemu, vmware. You can install your own, of course.

8.2 LWP?

- Winding up the stack
- Function pointers
- tryAsgn2
8.3 From Last time: Busywaiting

A better solution: Peterson’s Solution.

The first safe software solution was proposed by Dekker and published by Dijkstra in 1965 (“Co-operating Sequential Processes,” in Programming Languages, London: Academic Press, 1965.) Peterson came up with a better solution in 1981. See Figure 11.

**Advantages:** Works. Does not require strict alternation.

**Disadvantages:** Busy waiting.

```c
#define TRUE 1
#define FALSE 0
#define N 2 /* number of processes */

int turn; /* whose turn is it */
int interested[N]; /* all values initially FALSE */

void enter_region(int self) {
    /* self is 0 or 1 */
    int other; /* number of the other process */

    other = 1−self;

    interested[self] = TRUE; /* show interest */
    turn = self; /* try and claim the turn */
    while ((turn==self) && (interested[other]==TRUE)) /* dum–dee–dum */;
}

void leave_region(int self) { /* process who is leaving */
    interested[self]=FALSE;
}
```

Figure 11: Peterson’s solution for mutual exclusion

8.4 Onwards: Busy Waiting: With Hardware Support

Peterson’s soln is fairly complicated. What if we can have a little help.

- Test-and-set-lock instruction (TSL): Read a memory location and set a non-zero value to it.

**Advantages:** Atomic access guarantees simple correctness.

**Disadvantages:** Requires hardware support, still busy waiting

<table>
<thead>
<tr>
<th>enter</th>
<th>leave</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsl r1,lock</td>
<td>move lock,#0</td>
</tr>
<tr>
<td>cmp r1,#0</td>
<td>ret</td>
</tr>
<tr>
<td>jne enter</td>
<td></td>
</tr>
</tbody>
</table>
8.5 Reflection

The fundamental busywaiting problem: Wasting time.
That is, *the waiting process can actively prevent the event for which it is waiting.* (feeding the cat?) Remember the four desired principles:

1. No two processes may simultaneously enter the critical region.
2. No assumptions may be made about CPU speed of the number of CPUs
3. No process running outside of its critical section may run while another process is in its critical section.
4. No process should have to wait forever in its critical section.

Reconsider the problems with busywaiting:

- Except on a multiprocessor, busywaiting must waste time.
- Consider the priority inversion problem: a low-priority process is holding a lock that a high-priority process needs, but the low priority process can’t run while the high-priority one is waiting.

8.5.1 sleep() and wakeup()

sleep() goes to sleep until awakened
wakeup(process) wakes up a sleeping process

```c
#define N 100;
int count=0;

void producer(void) {
    while(TRUE) {
        produce();
        if ( count == N ) /* full */
            sleep();
        enter_item();
        count = count + 1;
        if ( count == 1 ) /* empty */
            wakeup (consumer);
    }
}

void consumer(void) {
    while(TRUE) {
        if ( count == 0 ) /* empty */
            sleep();
        remove_item();
        count = count - 1;
        if ( count == N-1 ) /* full */
            wakeup (producer);
        consume();
    }
}
```

Figure 12: A producer-consumer implementation with a race condition

Race conditions still: Wakeup might be missed:
When the buffer is empty:

1. Scheduler interrupt consumer() after the count==0 test, but before the sleep().
2. producer() produces an item, notes that count is now 1, and wakes consumer()
3. the `wake()` is lost because `consumer()` is not asleep.

4. `consumer()` is scheduled, then goes to sleep.

5. `producer()` continues to produce until the buffer is full, then goes to sleep.

Nobody ever wakes up.

### 8.6 Synchronization without busy waiting

#### 8.7 Semaphores

Generalized `sleep()` and `wakeup()` using a counter.

```c
p(counter) down(counter) decrement counter; wait if zero
v(counter) up(counter) increment counter; wakeup others if was zero
```

Must be atomic:

- usually done as a system call with disabled interrupts—note this is not the same as a long-term busywait.
- LWP can turn off signals to achieve the same effect.

Semaphores can be used for both (And these are different)

- mutual exclusion (binary semaphore)
- synchronization (initially N for producer-consumer)

Figure 13 shows a solution to the producer-consumer problem using semaphores.

Still must be careful: if you lock in the wrong order, you can deadlock. Consider the effect of reversing the downs in figure 13.

```c
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void) {
    while(TRUE) {
        produce();
        down(&empty);
        down(&mutex);
        enter_item();
        up(&mutex);
        up(&full);
    }
}

void consumer(void) {
    while(TRUE) {
        down(&full);
        down(&mutex);
        remove_item();
        up(&mutex);
        up(&empty);
        consume();
    }
}
```

Figure 13: A semaphore-based producer-consumer implementation
8.7.1 Monitors: (Hoare 1974, Brinch Hansen 1975)

Recall the sensitivity of semaphores to ordering we saw in the example of the other time: If the producer and consumer lock mutex and full/empty in the wrong order, they will deadlock. 

Monitors are a higher-level synchronization mechanism requiring programming-language support.

- A monitor is region of code where only one process can be active at a time. (enforced by the language).
- Communication between processes is done via condition variables with the primitives:
  
  wait(condition) wait until a signal. A process that blocks in the monitor releases other processes to enter.
  
  Note that this will be ok, because the release is voluntary.

  signal(condition) send a signal. To enforce the exclusion principle, a process that signals is required to leave the monitor immediately.
  
  A signal wakes one process waiting on that condition variable.

  These are very like sleep and wakeup, but automagically synchronized so counters are unnecessary.

A monitor-based solution to the producer-consumer problem is shown in Figure 14.

8.8 More Interprocess Communication

Ok, so, looking at our mechanisms that work

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Spin Locks</td>
<td>Waste Time</td>
</tr>
<tr>
<td>Semaphores</td>
<td>Complicated (easy to get wrong)</td>
</tr>
<tr>
<td>Monitors</td>
<td>Require language support</td>
</tr>
</tbody>
</table>

So what else can we do?
monitor ProducerConsumer
condition full,empty;
integer count;

procedure enter()
begin
if count = N then
wait(full);
enter_item();
count := count + 1;
if count = 1 then
signal(empty);
end;

procedure remove()
begin
if count = 0 then
wait(empty);
remove_item();
count := count - 1;
if count = N - 1 then
signal(full);
end;

count := 0;
end monitor

procedure consumer()
begin
while TRUE do
begin
ProducerConsumer.remove();
consume_item();
end
end;

procedure producer()
begin
while TRUE do
begin
produce_item();
ProducerConsumer.enter();
end
end;

Figure 14: A monitor-based solution to the producer-consumer problem.