8 Lecture: Concurrency and Synchronization, cont.

Outline:

Announcements
LWP
From last time: Review of Busy Waiting (Software only)
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:

Event	Subject	Due Date			Notes
asgn2	LWP	Mon	Jan 26	23:59	
asgn3	dine	Wed	Feb 4	23:59	
lab03	problem set	Mon	Feb 9	23:59	
midterm	stuff	Wed	Feb 11		
lab04	scavenger hunt II	Wed	Feb 18	23:59	
asgn4	/dev/secret	Wed	Feb 25	23:59	
lab05	problem set	Mon	Mar 9	23:59	
asgn5	minget and minls	Wed	Mar 11	23:59	
asgn6	Yes, really	Fri	Mar 13	23:59	
final (sec01)		Fri	Mar 20	10:10	
final (sec03)		Fri	Mar 20	13:10	

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

- "remember" function pointers

```
typedef void (*lwpfun)(void *); /* type for lwp function */
```

- Lab02
  - `testfloppy.img` is outside of minix, in the host filesystem. Inside, it's `/dev/fd0`.
  - Don't just click the little 'X' to stop minix

8.2 LWP

- The scheduler is totally separable. You can test it with `libPLN.a`.
- My demo code is in `~pn-cs453/Given/Asgn2`. It's not a particularly good test, but it's fun, `~pn-cs453/demos/tryAsgn2` will give it more of a workout.

- There are four pointers in the `context` structure. Two are reserved for the scheduler, two for the library. Use them for anything of nothing.

  Consider the value of:

  ```
  #define sched_one next
  ```

- The thread you create in `lwp_create()` will be loaded when one of `lwp_yield()`, `lwp_start()`, or `lwp_exit()` chooses it and passes its context to `swap_rfiles()`.

- It is loaded in the middle of `swap_rfiles()`, so it will go through `swap_rfiles()'s leave; ret` endgame. Figure out where you want to wind up, and work your way backwards through those instructions to see what you want your context to look like when it's loaded.

- the secret of `lwp_exit()`.

- Remember stacks are upside-down.

- Remember: all zeros is not a valid floating point state. Use the initializer.

8.3 From last time: Review of Busy Waiting (Software only)

Methods discussed:

- Hope for the best
- Disabling Interrupts
- Lock Variables (Software Only)
- Strict Alternation
- Peterson's Solution

The problem: lack of atomicity.

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

```
enter:  tsl r1,lock     leave:  move lock,#0
        cmp r1,#0               ret
        jne enter
        ret
```

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

```
#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 13: 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 `wake()`s `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.

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 14 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 14.

```
#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 14: A semaphore-based producer-consumer implementation

8.7.1 Monitors: (Hoare 1974, Brinch Hansen 1975)

Recall the senisitivity 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 15.

8.8 More Interprocess Communication

Ok, so, looking at our mechanisms that work

Spin Locks:	Waste Time
Semaphores:	Complicated (easy to get wrong)
Monitors:	Require language support

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