7 Lecture: Concurrency and Synchronization

Outline:
Announcements
LWP
Problems with parallelism: Race Conditions
  Race condition, defined
Critical Sections
Mutual Exclusion
Busy Waiting: Software Only
From last time: Review of Busy Waiting
Peterson’s Solution

7.1 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 23:59</td>
<td></td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7 23:59</td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8 10:10 (in 03-201)</td>
<td></td>
</tr>
</tbody>
</table>

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

• Warning about stack overflow/etc. E.g. WNOHANG or WUNTRACED. RTFM
• “remember” function pointers

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

• Lab02: Choosing an emulator: Pick one that works for you.
• Be sure virtualization is turned on.

7.2 LWP
• Note that the scheduler is totally separable
• Pointers already provided
• How the scheduler works
  The lwp scheduler is a structure that holds pointers to five functions. These are:

    void init(void) This is to be called before any threads are admitted to the scheduler. It’s to allow the scheduler to set up. This one is allowed, to be NULL, so don’t call it if it is.

    void shutdown(void) This is to be called when the lwp library is done with a scheduler to allow it to clean up. This, too, is allowed, to be NULL, so don’t call it if it is.

    void admit(thread new) Add the passed context to the scheduler’s scheduling pool and also adds it to the global lwp.tlist.

    void remove(thread victim) Remove the passed context from the scheduler’s scheduling pool and from the global thread list.
thread next(void)  Return the next thread to be run or NO_THREAD if there isn’t one. You’ll probably have to write a helper function along the lines of tid2context() to get the actual context pointer.

Changing schedulers will involve initializing the new one, pulling out all the threads from the old one (using next() and remove()) and admitting them to the new one (with admit()), then shutting down the old scheduler.

7.3 Problems with parallelism: Race Conditions

As soon as there any shared resources (even the filesystem), there can be problems.

Example: Print spooler

Consider a print spooler where files to be printed are placed in spots in an array indexed by qhead. The process to add an element looks something like: array[qhead++]=name. This breaks into three stages: read, update, write.

Consider the effect of the following interleaving.

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Read qhead</td>
<td>(2) Read qhead</td>
</tr>
<tr>
<td>(4) Update array</td>
<td>(3) Update array</td>
</tr>
<tr>
<td>(6) Increment index</td>
<td>(5) Increment Index</td>
</tr>
<tr>
<td>(7) Write qhead</td>
<td>(8) Write qhead</td>
</tr>
</tbody>
</table>

Not so good.

7.3.1 Race condition, defined

A race condition is any situation where the precise ordering of a series events affects the (correctness of the) outcome of the entire process.

The term is usually only applied where processes are reading or writing some shared data and where correctness is at stake.

Race conditions manifest themselves as nondeterministic behavior (usually leading to strange behavior at inopportune times).

7.4 Critical Sections

Critical sections require mutual exclusion.

For good solution we wish to maintain the following four 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.
7.5 Mutual Exclusion

How can we ensure mutual exclusion?

7.6 Busy Waiting: Software Only

For busy waiting to work, all contention must be among time-sliced processes or with true concurrency. (Why)

These solutions do not rely on hardware support to work correctly.

Options:

- Eliminate concurrency: That'll fix it.
- Hope for the best?
- Disabling Interrupts

**Advantages:** Foolproof?

**Disadvantages:**
- Gives user (fool?) too much power. Blocks everything. (Decapitation will cure a headache.)

What about a multiprocessor?

This is used in the kernel.

- Lock Variables (Software Only)

  ```
  while ( TRUE ) {
    while ( lock ) /* twiddle */;
    lock = 1;
    critical_things();
    lock = 0;
    noncritical_things();
  }
  ```

  **Advantages:** Allows finer-grained synchronization
  
  **Disadvantages:** **Doesn’t work** (race-conditions)

- Strict Alternation: use a variable to say whose turn it is, have each thread set it explicitly on the way out

  **Advantages:** Works.
  
  **Disadvantages:** Significantly reduces parallelism. What if a process forgets? What if a process is slow?

  ```
  Process A
  while ( TRUE ) {
    while ( turn != 0 ) /* twiddle */;
    critical_things();
    turn = 1;
    noncritical_things();
  }
  ```

  ```
  Process B
  while ( TRUE ) {
    while ( turn != 1 ) /* twiddle */;
    critical_things();
    turn = 0;
    noncritical_things();
  }
  ```