7 Lecture: Concurrency and Synchronization

Outline:

Announcements
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:

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.

- Warning about stack overflow/etc. E.g. WNOHANG or WUNTRACED. RTFM
- See gradesheet snapshot
- Lab02:
  - Choosing an emulator: Pick one that works for you. (Pick VirtualBox)
  - Be sure virtualization is turned on in your BIOS.
- "remember" function pointers

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

- Partners: Only turn in one assignment/lab with both names.
- GDB hints: display, conditional breakpoints

7.2 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.

Process A	**Process B**
(1) Read `qhead`	
	(2) Read `qhead`
	(3) Update array
(4) Update array	
	(5) Increment Index
(6) Increment index	
(7) Write `qhead`	
	(8) Write `qhead`

Not so good.

7.2.1 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).

7.3 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.

7.4 Mutual Exclusion

How can we ensure mutual exclusion?

7.5 Busy Waiting: Software Only

For busywaiting 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();
  }
  ```

7.6 Peterson's Solution

A better solution: Peterson's Solution.

The first safe software solution was proposed by Dekker and published by Dijkstra in 1965 ("Cooperating Sequential Processes," in *Programming Languages*, London: Academic Press, 1965.)

Peterson came up with a better solution in 1981.
See Figure 12.

Advantages:	Works. Does not require strict alternation.
Disadvantages:	Busy waiting.

```
#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 12: Peterson's solution for mutual exclusion