-*- mode: org -*-
#+STARTUP: indent

* <2011-09-26 Mon>: Lec: Intro to concurrency
Students read text, and do homework assignment
Last lecture didn't cover device interrupts

* Plan: concurrency
** SMP
** Devices
** locks
** interrupts
** more sophisticated techniques (e.g., lock-free later in term)

* abstract SMP architecture
** processors with one shared memory
** devices
** interrupts processed in parallel

* Race conditions
** multiple processes adding to the disk queue
** simplified code:

```
struct List {
  int data;
  struct List *next;
};

List *list = 0;

insert(int data) {
  List *l = new List;
  l->data = data;
  l->next = list;  // A
  list = l;        // B
}
```

Whoever wrote this code probably believed it was correct, in the sense that if the list starts out correct, a call to insert() will yield a new list that has the old elements plus the new one. And if you call insert(), and then insert() again, the list should have two new elements. Most programmers write code that's only correct under serial execution -- in this case, insert() is only correct if there is only one insert() executing at a time.

What could go wrong with concurrent calls to insert(), as might happen on a multiprocessor? Suppose two different processors both call <code>insert()</code> at the same time. If the two processors execute A and B interleaved, then we end up with an incorrect list. To see that this is the case, draw out the list after the sequence A1 (statement A executed by processor 1), A2 (statement A executed by processor 2), B2, and B1.

* Goals
serialize inserts on same list

serialize inserts & deletes on same list
ops on different lists in parallel

* Popular tool: lock
** lock protects an invariant

** Avoiding the list race:
Lock list_lock;   // one per list

```
insert(int data){
  List *l = new List;
  l->data = data;

  acquire(&list_lock);

  l->next = list ; // A
  list = l;        // B

  release(&list_lock);
}
```

*** code between acquire/release: critical section, or atomic section

** Return to device driver
*** One lock for all disk devices
*** Invariants
The disk hardware can only execute one read or write at a time.
Only one process should be inserting or deleting from ide_queue at a time.
Only one process should be commanding the IDE hardware (via inb/outb instructions) at a time.
*** iderw: why no locks around first instruction of the function?
*** Device driver has its own concurrency invariants:
processor shouldn't write/read buffer when disk is using it
how such invariants enforced? (answer: by careful programming...)

** How hard is it to produce the list race
*** let's try with stressfs
*** must read because logging system serializes writes

* Implementing lock and release
** use atomic instructions (e.g., xchg)
For example, the x86 <tt>xchg %eax, addr</tt> instruction
does the following:
freeze other CPUs' memory activity for address addr
temp := *addr
*addr := %eax
%eax = temp
un-freeze other memory activity
** xv6 lock implementation
instruction reordering
** lock is a *spin lock*

* Design issues
** Granularity (e.g., BKL)

Should we have one lock for kernel (this year your JOS will do this ...)
Should we have one lock per device? (ide)
** Spinning versus sleeping
Spin locks not good for waiting until device driver returns
xv6: separate primitive to go to sleep
some kernel automatically switch to sleeping
** locks and interrupts
ide disk generates interrupt when disk operation completes
interrupt causes ideintr to run
ideintr acquires lock?
  deadlock?
  give interrupt handler lock? (recursive locks)
xv6: critical sections have no interrupts turned on
** locks and modularity
*** locks are part of the spec of a module (e.g., idestart assumes caller got lock)

```
  move(l1, l2) {
    e = del(l1)
    insert(l2, e)
  }
```

various problems: there is a time when it is observable that e is neither list

```
  move(l1, l2) {
    acquire(l1.lock);
    acquire(l2.lock);
    e = del(l1)
    insert(l2, e)
    release(l1.lock)
    release(l2.lock)
  }
```

*** recursive locks are a bad idea
ideintr should certainly not use that instead of disabling interrupts!
** lock ordering
iderw: sleep acquires plock
--> never acquire plock and then ide_lock

6.828 Operating System Engineering
Fall 2012