CSE 451: Operating Systems

Section 5
Synchronization

Project 1 Recap

★ Tips:
★ Check flags with & not ==
★ Use constants for printed strings
  #define PROMPT “CSE451Shell>”
★ Use errno/perror(3) for error detection
★ To make grading easier:
  ★ Preserve the build hierarchy/commands
  ★ Check your files before turnin!

Project 2.a is almost due

★ Remember to write more test cases!
★ Writeups:
  ★ Design decisions & alternative implementations:
    give them some real thought
  ★ Be mindful of what you use as a resource (and how much)
  ★ We expect you to research, but we expect you to fumble around a little too
Synchronization

Disabling/enabling interrupts

Thread A:
- disable_irq()
- critical_section()
- enable_irq()

Thread B:
- disable_irq()
- critical_section()
- enable_irq()

- Prevents context-switches during execution of critical sections
- Sometimes necessary
- Many pitfalls

Synchronization support

- Processor level:
  - Disable/enable interrupts
  - Atomic instructions (test-and-set)

- Operating system level:
  - Special variables: mutexes, semaphores, condition variables

- Programming language level:
  - Monitors, Java synchronized methods

Processor support

- Atomic instructions:
  - test-and-set
  - compare-exchange (x86)

- Use these to implement higher-level primitives
  - E.g. test-and-set on x86 (given to you for part 4) is written using compare-exchange
**Processor support**

*Test-and-set using compare-exchange:

```c
compare_exchange(lock_t *x, int y, int z):
  if(*x == y)
    *x = z;
    return y;
  else
    return *x;
}
test_and_set(lock_t *lock) {
  ???
}
```

**Semaphores**

*Semaphore = a special variable

* Manipulated atomically via two operations
  * P (wait): tries to decrement semaphore
  * V (signal): increments semaphore
* Has a queue of waiting threads
  * If execute wait() and semaphore is available, continue
  * If not, block on the waiting queue
  * signal() unblocks a thread on queue

**Project 2: preemption**

*Think about where synchronization is needed

* Start inserting synchronization code
  * disable/enable timer interrupts
  * atomic_test_and_set

```c
10/28/10
```
Mutexes

* A binary semaphore (semaphore initialized with value 1)
* A lock that waits by blocking, rather than spinning

Aside: kernel locking

* Can we use mutexes inside our kernel?
* Sometimes...
* Spinlocks more common than semaphores/mutexes in Linux
* Reader-writer locks (rwlocks):
  * Allow multiple readers or single writer
  * Good idea?
  * [http://lwn.net/Articles/364583/](http://lwn.net/Articles/364583/)
**Condition variables**

* Let threads block until a certain event occurs (rather than polling)
* Associated with some logical condition in program
  ```c
  while (x <= y) {
    sthread_user_cond_wait(cond, lock)
  }
  ```

**Operations:**
* wait: sleep on wait queue until event happens
* signal: wake up one thread on wait queue
  * Explicitly called when event/condition has occurred
* broadcast: wake up all threads on wait queue

**Example synchronization problem**

* Late-Night Pizza
  * A group of students study for CSE 451 exam
  * Can only study while eating pizza
  * If a student finds pizza is gone, the student goes to sleep until another pizza arrives
  * First student to discover pizza is gone orders a new one
  * Each pizza has 5 slices
Late-night pizza

* Each student thread executes the following:
  ```
  while (must_study) {
    pick up a piece of pizza;
    study while eating the pizza;
  }
  ```

Late-night pizza

* Synchronize student threads and pizza delivery thread
* Avoid deadlock
* When out of pizza, order it exactly once
* No piece of pizza may be consumed by more than one student

Semaphore/mutex solution

* Shared data:
  ```
  semaphore_t pizza;    //Number of available pizza resources;
                      //init to 0
  semaphore_t deliver;  //init to 1
  int num_slices = 0;
  mutex_t mutex;        //guards updating of num_slices
  ```

student_thread {
  while (must_study) {
    wait(pizza);
    acquire(mutex);
    num_slices--;
    if (num_slices==0)
      signal(deliver);
    release(mutex);
    study();
  }
}

delivery_guy_thread {
  while (employed) {
    wait(deliver);
    make_pizza();
    acquire(mutex);
    num_slices=S;
    release(mutex);
    for (i=0;i<S;i++)
      signal(pizza);
  }
}
Condition variable solution

* Shared data:
  - int slices=0;
  - bool has_been_ordered;
  - Condition order;  // an order has been placed
  - Condition deliver;  // a delivery has been made
  - Lock mutex;  // protects "slices"; associated with both Condition variables

Monitors

* An object that allows one thread inside at a time
* Contain a lock and some condition variables
  - Condition variables used to allow other threads to access the monitor while one thread waits for an event to occur

```cpp
Student() {
    while(diligent) {
        mutex.lock();
        if (slices > 0) {
            slices--;
        } else {
            if (!has_been_ordered){
                order.signal(mutex);
                has_been_ordered = true;
            } deliver.wait(mutex);
        }
        mutex.unlock();
        Study();
    }
}
```

```cpp
DeliveryGuy() {
    while(employed) {
        mutex.lock();
        order.wait(mutex);
        makePizza();
        slices = S;
        has_been_ordered = false;
        mutex.unlock();
        deliver.broadcast();
    }
}
```


Monitors in Java

✦ Each object has its own monitor

Object o

✦ The Java monitor supports two types of synchronization:
  ✦ Mutual exclusion
    synchronized(o) { ... }
  ✦ Cooperation
    synchronized(o) { o.wait(); }
    synchronized(o) { o.notify(); }

Semaphores vs. CVs

Semaphores          Condition variables
✦ Used in apps      ✦ Typically used in monitors
✦ wait() does not always block the caller ➢ wait() always blocks caller
✦ signal() either releases a blocked thread, if any, or increases semaphore counter ➢ signal() either releases a blocked thread, if any, or the signal is lost forever