Section 5

Synchronization primitives

(Many slides taken from Winter 2006)

Announcements

- Assignment grades online
  - Please check them and report bugs to TAs
- Mailing list for TAs:
  - cse451-tas@cs.washington.edu
  - Faster response time
- Late project 0s were not downgraded
  - Next ones will!
- Hand homework 2 back
- Any questions?? (project 2, class, midterm)

Synchronization

<table>
<thead>
<tr>
<th>High-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitors</td>
</tr>
<tr>
<td>Java synchronized method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS-level support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special variables – mutex, futex, semaphor, condition var</td>
</tr>
<tr>
<td>Message passing primitives</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low-level support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable/enable interrupts</td>
</tr>
<tr>
<td>Atomic instructions (test_and_set)</td>
</tr>
</tbody>
</table>

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
  - E.g. to prevent further interrupts during interrupt handling
- Many problems
  - E.g., an interrupt may be shared
  - How does it work on multi-processors?

Hardware 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:
    ```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 *)
    ```
Looking ahead: preemption
- You can start inserting synchronization code
  - disable/enable interrupts
  - atomic_test_and_set
- Where would you use these?

Synchronization

<table>
<thead>
<tr>
<th>High-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitors</td>
</tr>
<tr>
<td>Java synchronized method</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS-level support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special variables – mutex, futex, semaphore, condition var</td>
</tr>
<tr>
<td>Message passing primitives</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low-level support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable/enable interrupts</td>
</tr>
<tr>
<td>Atomic instructions</td>
</tr>
</tbody>
</table>

- Used to implement higher-level sync primitives (in the kernel typically)
- Why not use in apps?

Semaphore review
- Semaphore = a special variable
  - Manipulated atomically via two operations:
    - P (wait)
    - V (signal)
- Has a counter = number of available resources
  - P decrements it
  - V increments it
- Has a queue of waiting threads
  - If execute wait() and semaphore is free, continue
  - If not, block on that waiting queue
- Signal() unblocks a thread if it's waiting
- Mutex is bi-value semaphore (capacity 1)

Condition Variable
- A "place" to let threads wait for a certain event to occur while holding a lock
- It has:
  - Wait queue
  - Three functions: wait, signal, and broadcast
    - wait – sleep until the event happens
    - signal – event/condition has occurred. If wait queue nonempty, wake up one thread, otherwise do nothing
    - Do not run the woken up thread right away
    - FIFO determines who wakes up
  - Broadcast – just like signal, except wake up all threads
  - In part 2, you implement all of these
- Typically associated with some logical condition in program

Condition Variable (2)
- cond_wait(sthread_cond_t cond, sthread_mutex_t lock)
  - Should do the following atomically:
    - Release the lock (to allow someone else to get in)
    - Add current thread to the waiters for cond
    - Block thread until awoken
  - Read man page for pthread_cond_[wait|signal|broadcast]
  - Must be called while holding lock! -- Why?

Semaphores vs. CVs
Semaphores vs. CVs

**Semaphores**
- Used in apps
- `wait()` does not always block the caller
- `signal()` either releases a blocked thread, if any, or increases sem. counter.

**Condition variables**
- Typically used in monitors
- `wait()` always blocks caller
- `signal()` either releases blocked thread(s), if any, or the signal is lost forever.

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

Sample synchronization problem

**Late-Night Pizza**
- A group of students study for cse451 exam
- Can only study while eating pizza
- Each student thread executes the following:
  - `while (must_study) {
    pick up a piece of pizza;
    study while eating the 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 S slices.

Semaphore / mutex solution

```c
Semaphore / mutex solution

shared data:
semaphore_t pizza; (counting sema, init to 0, represent number of available pizza resources)
semaphore_t deliver; (init to 1)
int num_slices = 0;
mutex_t mutex; (init to 1) // guard updating of num_slices

Student { 
  while (must_study) {
    P(pizza);
    acquire(mutex);
    num_slices--;
    if (num_slices==0)
      // took last slice
      V(deliver);
    release(mutex);
    study();
  }
}

DeliveryGuy { 
  while (employed) {
    P(deliver);
    make_pizza();
    acquire(mutex);
    num_slices=S;
    release(mutex);
    for (i=0; i < S; i++)
      V(pizza);
  }
}
```

Condition Variable Solution

```c
Condition Variable Solution

int slices=0;
Condition order, deliver;
Lock mutex;
bool has_been_ordered = false;

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();
  }
}

DeliveryGuy { 
  while (employed) {
    mutex.lock();
    order.wait(mutex);
    make_pizza();
    mutex.unlock();
    if( num_slices==0) // took last slice
      V(deliver);
    release(mutex);
    deliver.broadcast();
  }
}
```

Monitors: preview

- One thread inside at a time
- Lock + a bunch of condition variables (CVs)
- CVs used to allow other threads to access the monitor while one thread waits for an event to occur
Monitors in Java

- Each object has its own monitor
  
  ```java
  Object o
  ```

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