Concurrent Programming

Introduction and Advice

- Read the Birrell paper
  - excellent introductory paper
  - promotes understanding the material
  - abstract content with direct application
    - limited and rather outdated concrete technical content
- I will concentrate on Java here, but the same applies to other systems

Concurrent Programming

- Most thread programming nowadays is monitor-style programming (e.g., Java threads, PThreads, threads in OS kernels)
- Monitor style programming has two components:
  - locks/mutexes (lock)
  - condition variables (wait, signal, broadcast)
- Mapping of abstract to concrete:
  - Java:
    - lock -> synchronized
    - wait -> wait
    - signal -> notify
    - broadcast -> notifyAll
  - PThreads:
    - lock -> pthread_mutex_lock ...
    - pthread_mutex_unlock
    - wait, signal, broadcast -> pthread_cond_{wait, signal, broadcast}

Different Models

- In Java every object can be a mutex/condition variable
  - better way to think of it: every object is associated with a unique mutex and condition variable
- In other systems you need to explicitly create mutex/condition variables
  - E.g.,
    Mutex m; ... Lock(m) { ... }
- Thread creation is mostly uninteresting
  - in Java: threads are instances of class Thread, they begin execution when their start method is called

Mutex Example

```java
class List {
  public synchronized int insert(int i) {
    [BODY] }
}
```

same as

```java
class List {
  public int insert(int i) {
    synchronized(this) [BODY] }
}
```

- Mutexes are used to control access to shared data
  - only one thread can execute inside a synchronized clause
  - other threads who try to enter synchronized code, are blocked until the mutex is unlocked
Condition Variables

- Condition variables are used to wait for specific events (especially for long waits)
  - free memory is getting low, wake up the garbage collector thread
  - 10,000 clock ticks have elapsed, update that window
  - new data arrived in the I/O port, process it

- Each condition variable is associated with a single mutex
  - In Java, each mutex is also associated with a single condition (ugly, ugly, ugly!)

- `wait` atomically unlocks the mutex (as many times as needed) and blocks the thread

- `notify` awakes some blocked thread
  - the thread is awoken inside `wait`
  - tries to lock the mutex (maybe many times)
  - when it (finally) succeeds, it returns from the `wait`

Condition Variable Example

class Buffer {
    Port port;
    public synchronized void consume() {
        while (port.empty())
            wait();
        process_data(port.first_data());
    }
    public synchronized void produce() {
        port.add_data();
        notify();
    }
}

Use of Mutexes and Condition Variables

- We’ll talk about programming suggestions, common errors, and good idioms
- Advice: read examples in paper and absorb at your own pace

Deadlocks

Examples:

- A locks M1, B locks M2, A blocks on M2, B blocks on M1
- Similar examples with condition variables and mutexes

Techniques for avoiding deadlocks:

- Fine grained locking
- Two-phase locking: acquire all the locks you’ll ever need up front, release all locks if you fail to acquire any one
  - very good technique for some applications, but generally too restrictive
- Order locks and acquire them in order (e.g., all threads first acquire M1, then M2)

Using Condition Variables

Recall our example:

class Buffer {
    Port port;
    public synchronized void consume() {
        while (port.empty())
            wait();
        process_data(port.first_data());
    }
    public synchronized void produce() {
        port.add_data();
        notifyAll();
    }
}

Why use `while` instead of `if`? (think of many consumers, simplicity of coding producer)

- `notifyAll` is then safe to use in place of `notify`
The Golden Rules

- Most problems with concurrent programming are very simple oversights! People forget to access shared variables in locks, forget to signal when a condition changes, etc.

**The golden rules:**

1. Shared data should always be accessed through a single mutex (easy in Java: just make non-public in a class)
2. Think of a boolean condition (expressed in terms of program variables) for each condition variable. Every time the value of the boolean condition may have changed, call notifyAll for the condition variable - only call notify when you are absolutely certain that *any* and *only* one waiting thread can enter the critical section
3. Globally order locks, acquire in order in all threads

Monitor-Style Programming

- Armed with mutexes and condition variables, you can implement *any* kind of critical section
  
  ```
  CS.enter(); [controlled code] CS.exit();
  ```

- General pattern:

```java
class CS {
    [shared data]
    public synchronized void enter() {
        while (![condition])
            wait();
        [change shared data to reflect in_CS]
        notify as needed
    }

    public synchronized void exit() {
        [change shared data to reflect out_of_CS]
        notify as needed
    }
} 
```

Example: Readers/Writers Locking

```java
class RWLock {
    int readers;
    public RWLock() { readers = 0; }
    public synchronized void enter_read() {
        while (readers == -1)
            wait();
        readers++;
    }
    public synchronized void exit_read() {
        readers--;
        if (readers == 0)
            notify();
    }
    public synchronized void enter_write() {
        while (readers != 0)
            wait();
        readers = -1;
    }
    public synchronized void exit_write() {
        readers = 0;
        notifyAll();
    }
}
```

Comments on Readers/Writers Example

- Invariant: readers >= -1
- Note the use of notifyAll
- Single condition variable for phase changes - ugly, ugly, ugly, and inefficient!
- Note that a writer signals all potential readers and one potential writer. Not all can proceed, however (*spurious wake-ups*)
- Unnecessary lock conflicts may arise (especially for multiprocessors):
  - both readers and writers signal condition variables while still holding the corresponding mutexes
  - notifyAll wakes up many readers that will contend for a mutex
  - can do a single notify, then have a reader notify next reader