Overview

- **Java 1.5**
  - ReentrantLock class
  - Condition interface - `await()`, `signalAll()`
- **Ruby multithreading**
- **Concurrent programming**
  - Parallel applications & languages
  - MapReduce

---

### Lock Interface (Java 1.5)

```java
interface Lock {
    void lock();
    void unlock();
    ...
    /* Some more stuff, also */
}
```  

- **Explicit Lock objects**
  - Same as implicit lock used by synchronized keyword
- **Only one thread can hold a lock at once**
  - `lock()` causes thread to block (become suspended) until lock can be acquired
  - `unlock()` allows lock to be acquired by different thread

---

### ReentrantLock Class (Java 1.5)

```java
class ReentrantLock extends Object implements Lock {
    // ... more stuff ...
}
```  

- **Reentrant lock**
  - Can be reacquired by same thread by invoking `lock()`
    - Up to 2147483648 times
  - To release lock, must invoke `unlock()`
    - The same number of times `lock()` was invoked
- **Reentrancy is useful**
  - Each method can acquire/release locks as necessary
  - No need to worry about whether callers already have locks
  - Discourages complicated coding practices
    - To determine whether lock has already been acquired

---

### Synchronization Example (Java 1.5)

```java
public class Example extends Thread {
    private static int cnt = 0;
    Lock lock = new ReentrantLock();

    public void run() {
        lock.lock();
        int y = cnt;
        cnt = y + 1;
        lock.unlock();
    }
    ...
}
```  

- **Lock**, for protecting the shared state
- **Acquires** the lock; only succeeds if not held by another thread, otherwise blocks
- **Releases** the lock

---

**Reentrant Lock Example**

```java
static int count = 0;
static Lock l = new ReentrantLock();

void inc() {
    l.lock();
    count++;
    l.unlock();
}
```  

- **Example**
  - `returnAndInc()` can acquire lock and invoke `inc()`
  - `inc()` can acquire lock without having to worry about whether thread already has lock
Condition Interface (Java 1.5)

```java
interface Lock {
    Condition newCondition();
}

interface Condition {
    void await();
    void signalAll();
}
```

- **Explicit condition variable objects**
  - Condition variable C is created from a Lock object L by calling L.newCondition()
  - Condition variable C is then associated with L
- **Multiple condition objects per lock**
  - Allows different wait sets to be created for lock
  - Can wake up different threads depending on condition

Producer / Consumer Solution (Java 1.5)

```java
void produce(Object o) {
    lock.lock();
    while (!bufferReady)
        ready.await();
    buffer = o;
    bufferReady = true;
    ready.signalAll();
    lock.unlock();
}
```

- Uses single condition per lock (as in Java 1.4)

Condition – await() and signalAll()

- **Calling await() w/ lock held**
  - Releases the lock
  - But not any other locks held by this thread
  - Adds this thread to wait set for condition
  - Blocks the thread
- **Calling signalAll() w/ lock held**
  - Resumes all threads in condition’s wait set
  - Threads must reacquire lock
  - Before continuing (returning from await)
  - Enforced automatically; you don’t have to do it

Producer / Consumer Solution (Java 1.5)

```java
void produce(Object o) {
    lock.lock();
    while (!bufferReady)
        producers.await();
    buffer = o;
    bufferReady = true;
    producers.signalAll();
    lock.unlock();
}
```

- Uses 2 conditions per lock for greater efficiency

Overview

- **Java 1.5**
  - ReentrantLock class
  - Condition interface - await(), signalAll()
- **Ruby multithreading**
- **Concurrent programming**
  - Parallel applications & languages
  - MapReduce

Ruby Threads – Thread Creation

- **Create thread using Thread.new**
  - New method takes code block argument
    ```java
    t = Thread.new {
        body of thread...
    }
    ```
  - ```java
    t = Thread.new(args) {
        body of thread...
    }
    ```
  - **Join** method waits for thread to complete
    ```java
    t.join()
    ```
- **Example**
  ```java
  myThread = Thread.new {
      sleep 1
      # sleep for 1 second
      puts "New thread awake!"
      $stdout.flush
      # flush makes sure output is seen
  }
  ```
Ruby Threads – Locks

- **Monitor, Mutex**
  - Object intended to be used by multiple threads
  - Methods are executed with mutual exclusion
    - As if all methods are synchronized
  - Monitor is reentrant, Mutex is not
- **Create lock using Monitor.new**
  - Synchronize method takes code block argument
    ```
    require 'monitor.rb'
    myLock = Monitor.new
    myLock.synchronize {
      # myLock held during this code block
    }
    ```

Ruby Threads – Condition

- **Condition derived from Monitor**
  - Create condition from lock using new_cond
  - Sleep while waiting using wait_while, wait_until
  - Wake up waiting threads using broadcast
- **Example**
  ```
  myLock = Monitor.new
  myCondition = myLock.new_cond
  myLock.synchronize {
    myCondition.wait_until { y > 0 }
    myCondition.wait_until { x <= 0 }
  }
  myLock.synchronize {
    myCondition.broadcast
    # wake up all waiting threads
  }
  ```

Parking Lot Example

```ruby
require "monitor.rb"
class ParkingLot
  def initialize  # initialize synchronization
    @numCars = 0
    @myLock = Monitor.new
    @myCondition = @myLock.new_cond
  end
  def addCar
    # do work not requiring synchronization
    garage.addCar
    @myLock.synchronize {
      @myCondition.broadcast
      # wake up all waiting threads
    }
  end
  def removeCar
    # do work not requiring synchronization
    garage.removeCar
    @myLock.synchronize {
      @myCondition.wait_until { @numCars > 0 }
      @numCars = @numCars - 1
      @myCondition.broadcast
    }
  end
end
```

Parking Lot Example

```ruby
garage = ParkingLot.new
valet1 = Thread.new {  # valet 1 drives cars into parking lot
  while ...
    # do work not requiring synchronization
    garage.addCar
    @myLock.synchronize {
      @myCondition.broadcast
      # wake up all waiting threads
    }
  }
} valet2 = Thread.new {  # valet 2 drives car out of parking lot
  while ...
    # do work not requiring synchronization
    garage.removeCar
    @myLock.synchronize {
      @myCondition.wait_until { @numCars < MaxCars }  # new lock
      @numCars = @numCars + 1
      @myCondition.broadcast
    }
  }
} valet1.join  # returns when valet 1 exits
valet2.join  # returns when valet 2 exits
```

Ruby Threads – Difference from Java

- Ruby thread can access all variables in scope when thread is created, including local variables
  - Java threads can only access object fields
- **Exiting**
  - All threads exit when main Ruby thread exits
  - Java continues until all non-daemon threads exit
- **When thread throws exception**
  - Ruby only aborts current thread (by default)
  - Java aborts all threads (better for debugging)
    - Set Thread.abort_on_exception = true
Overview

- Java 1.5
  - ReentrantLock class
  - Condition interface - await(), signalAll()
- Ruby multithreading
- Concurrent programming
  - Parallel applications & languages
  - MapReduce

Multithreading (Java threads, pthreads)

- Portable, high degree of control
- Low-level and unstructured
  - Thread management, synchronization via locks and signals essentially manual
    - Blocking synchronization is not compositional, which inhibits nested parallelism
  - Easy to get wrong, hard to debug
    - Data races, deadlocks all too common

Parallelizable Applications of Interest

- Knowledge discovery: mine and analyze massive amounts of distributed data
  - Discovering social networks
  - Real-time, highly-accurate common operating picture, on small, power-constrained devices
- Simulations (games?)
- Data processing
  - NLP, Vision, rendering, in real-time
- Commodity applications
  - Parallel testing, compilation, typesetting, …

Parallel Language Extensions

- MPI – expressive, portable, but
  - Hard to partition data and get good performance
    - Temptation is to hardcore data locations, number of processors
  - Hard to write the program correctly
    - Little relation to the sequential algorithm
- OpenMP, HPF – parallelizes certain code patterns (e.g., loops), but
  - Limited to built-in types (e.g., arrays)
  - Code patterns, scheduling policies brittle

Two Directions To A Solution

- Start with clean, but limited, languages/abstractions and generalize
  - MapReduce (Google)
  - StreamIt (MIT)
  - Cilk (MIT)
- Start with full-featured languages and add cleanliness
  - Software transactional memory
  - Static analyzers (Locksmith, Chord, …)
  - Threaded Building Blocks (Intel)

Space of Solutions
Kinds of Parallelism

- Data parallelism
  - Can divide parts of the data between different tasks and perform the same action on each part in parallel
- Task parallelism
  - Different tasks running on the same data
- Hybrid data/task parallelism
  - A parallel pipeline of tasks, each of which might be data parallel
- Unstructured
  - Ad hoc combination of threads with no obvious top-level structure

MapReduce: Programming the Pipeline

- Pattern inspired by Lisp, ML, etc.
  - Many problems can be phrased this way
- Results in clean code
  - Easy to program / debug / maintain
    - Simple programming model
    - Nice retry / failure semantics
  - Efficient and portable
    - Easy to distribute across nodes

Map & Reduce in Lisp / Scheme

- (map f list)
- (map square '1 2 3 4)
  - (1 4 9 16)
- (reduce + (1 4 9 16) 0)
  - (+ 1 (+ 4 (+ 9 (+ 16 0 ) )))
  - 30
- (reduce + (map square '1 2 3 4) 0)

Count Words in Documents

- Input consists of (url, contents) pairs
- map(key=url, val=contents):
  - For each word w in contents, emit (w, "1")
- reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

MapReduce a la Google

- map(key, val) is run on each item in set
  - emits new-key / new-val pairs
- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output

Count, Illustrated

map(key=url, val=contents):
  For each word w in contents, emit (w, "1")
reduce(key=word, values=uniq_counts):
  Sum all "1"s in values list
  Emit result "(word, sum)"

see 1
bob 1
run 1
see 2
spot 1
throw 1
Execution

Parallel Execution

Model is Widely Applicable
MapReduce Programs In Google Source Tree 2004

The Programming Model Is Key

Compare to Dedicated Supercomputers

- Simple control makes dependencies evident
  - Can automate scheduling of tasks and optimization
    - Map, reduce for different keys, embarrassingly parallel
    - Pipeline between mappers, reducers evident
  - Map and reduce are pure functions
    - Can rerun them to get the same answer
    - In the case of failure, or
    - To use idle resources toward faster completion
  - No worry about data races, deadlocks, etc. since there is no shared state

- According to Wikipedia, in 2006 Google uses
  - 450,000 servers from 533 MHz Celeron to dual 1.4GHz Pentium III
  - 80GB drive per server, at least
  - 2-4GB memory per machine
  - Jobs processing 100 terabytes of distributed data

- More computing power than even the most powerful supercomputer