CMSC 330: Organization of Programming Languages

Java 1.5, Ruby, and MapReduce

Reminders & Announcements

- Project 4 due Monday, July 12 at midnight!
- Project 5 will be posted today
  - Due Wednesday, July 21 at midnight
- Quiz #3 next Friday, July 16
  - Multithreading, garbage collection, type systems/names/bindings
- Final Exam in only two weeks!
  - Friday, July 23
  - Everything, but with emphasis on topics toward the end of the course

Last Lecture

- Multithreading introduction
- Java threads review
  - Topics from CMSC 132
- Producer / consumer pattern
- Conditions
  - wait( )
  - notifyAll( )

This Lecture

- 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 */
}
class ReentrantLock implements Lock { ... }
```

- **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

Synchronization Example (Java 1.5)

```java
public class Example extends Thread {
    private static int cnt = 0;
    static 
        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 Class (Java 1.5)

```java
class ReentrantLock implements Lock { ... }
```

- **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
static int count = 0;
static Lock l = new ReentrantLock();
void inc() {
    l.lock();
    count++;
    l.unlock();
}

void returnAndInc() {
    int temp;
    l.lock();
    temp = count;
    inc();
    l.unlock();
}
```

- Example
  - returnAndInc( ) can acquire lock and invoke inc( )
  - inc( ) can acquire lock without having to worry about whether thread already has lock
Explicit condition variable objects

- Condition variable \( C \) is created from a Lock object \( L \) by calling \( L.\text{newCondition()} \)
- 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

interface Lock {
  Condition newCondition(); ...
}

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

Condition Interface (Java 1.5)

Calling \( \text{await()} \) 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 \( \text{signalAll()} \) w/ lock held

- Resumes all threads in condition’s wait set
- Threads must reacquire lock
  - Before continuing (returning from \( \text{await()} \))
  - Enforced automatically; you don’t have to do it

Producer / Consumer Solution (Java 1.5)

```java
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean bufferReady = false;
Object buffer;

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

Object consume() {
  lock.lock();
  while (!bufferReady)
    consumers.await();
  Object o = buffer;
  bufferReady = false;
  consumers.signalAll();
  lock.unlock();
}
```

Uses single condition per lock (as in Java 1.4)

Producer / Consumer Solution (Java 1.5)

```java
Lock lock = new ReentrantLock();
Condition producers = lock.newCondition();
Condition consumers = lock.newCondition();
boolean bufferReady = false;
Object buffer;

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

Object consume() {
  lock.lock();
  while (!bufferReady)
    consumers.await();
  Object o = buffer;
  bufferReady = false;
  producers.signalAll();
  lock.unlock();
}
```

Uses 2 conditions per lock for greater efficiency
Ruby Threads – Thread Creation

Create thread using Thread.new

• New method takes code block argument
  t = Thread.new { …body of thread… }
  t = Thread.new (arg) { | arg | …body of thread… }
• Join method waits for thread to complete
  t.join

Example

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 # new lock
myCondition = myLock.new_cond # new condition
myLock.synchronize {
  myCondition.wait_while { y > 0 } # wait as long as y > 0
  myCondition.wait_until { x >= 0 } # wait as long as x >= 0
}
myLock.synchronize {
  myCondition.broadcast # wake up all waiting threads
}

Parking Lot Example

require "monitor.rb"
class ParkingLot
def initialize # initialize synchronization
  @numCars = 0
  @myLock = Monitor.new
  @myCondition = @myLock.new_cond
end
def addCar # ...
  end
def removeCar # ...
  end
def # ...
  end
end

Parking Lot Example

```ruby
def addCar
  # do work not requiring synchronization
  @myLock.synchronize {
    @myCondition.wait_until { @numCars < MaxCars }
    @numCars = @numCars + 1
    @myCondition.broadcast
  }
end

def removeCar
  # do work not requiring synchronization
  @myLock.synchronize {
    @myCondition.wait_until { @numCars > 0 }
    @numCars = @numCars - 1
    @myCondition.broadcast
  }
end
```

```ruby
garage = ParkingLot.new
valet1 = Thread.new {
  # valet 1 drives cars into parking lot
  while …
    # do work not requiring synchronization
    garage.addCar
  end
}
valet2 = Thread.new {
  # valet 2 drives car out of parking lot
  while …
    # do work not requiring synchronization
    garage.removeCar
  end
}
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)
  - Ruby can also abort all threads (better for debugging)
    > Set Thread.abort_on_exception = true

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, typsetting, …
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

Parallel Language Extensions

- MPI: expressive, portable, but
  - Hard to partition data and get good performance
    - Temptation is to hardcode 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

- \((\text{map } f \text{ list})\)
  - Unary operator

- \((\text{map square } '(1 2 3 4))\)
  - Binary operator

- \((\text{reduce } + ' (1 4 9 16) 0)\)
  - \(\left(\frac{1 + 4 + 9 + 16}{0}\right)\)
  - 30

- \((\text{reduce } + (\text{map square } '(1 2 3 4)) 0)\)

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 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)”

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
bob 1  run 1
run 1  see 2
see 1  spot 1
spot 1  throw 1
```

Execution

Parallel Execution

Key: no implicit dependencies between map or reduce tasks
### Model is Widely Applicable

**MapReduce Programs in Google Source Tree 2004**

<table>
<thead>
<tr>
<th>Time</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mar 2003</td>
<td>1000 GB</td>
</tr>
<tr>
<td>May 2003</td>
<td>2000 GB</td>
</tr>
<tr>
<td>Jul 2003</td>
<td>3000 GB</td>
</tr>
<tr>
<td>Sep 2003</td>
<td>4000 GB</td>
</tr>
<tr>
<td>Nov 2003</td>
<td>5000 GB</td>
</tr>
<tr>
<td>Jan 2004</td>
<td>6000 GB</td>
</tr>
<tr>
<td>Mar 2004</td>
<td>7000 GB</td>
</tr>
<tr>
<td>May 2004</td>
<td>8000 GB</td>
</tr>
<tr>
<td>Jul 2004</td>
<td>9000 GB</td>
</tr>
<tr>
<td>Sep 2004</td>
<td>10000 GB</td>
</tr>
</tbody>
</table>

Example uses:
- distributed grep
- term-vector / host
- document clustering
- distributed sort
- web link-graph reversal
- web access log stats
- inverted index construction
- machine learning
- statistical machine translation

---

### The Programming Model Is Key

- Simple control makes dependencies evident
  - 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

---

### Compared to Dedicated Supercomputers

- According to Wikipedia, in 2006 Google used
  - 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