CMSC 330: Organization of Programming Languages

Multiprocessing
Multiprocessing

- **Multiprocessing:** The use of multiple parallel computations

- We are in the era of multiple cores...
  - Hyperthreading
  - Dual-core and quad-core consumer processors
  - Symmetric Multi-Processing (SMP) machines

- …and multiple nodes
  - Computational clusters
  - Grid and cloud computing
Technologies

• Multiple cores:
  – SIMD arithmetic and GPUs
  – Java and Ruby threads
  – POSIX threads
  – OpenMP

• Multiple nodes:
  – MPI
  – MapReduce
Amdahl’s Law

- Informally: The theoretical maximum speedup using multiprocessing is limited by a program’s sequential performance.
Computation Abstractions

A computer

Processes

Threads
Processes vs. Threads

Processes do not share data

Threads share data within a process
So, What Is a Thread?

• Conceptually: it is a parallel computation occurring within a process

• Implementation view: it’s a program counter and a stack. The heap and static area are shared among all threads

• All processes have at least one thread (main)
  – Programs vs. processes
Thread Creation

execution (time)

main thread

thread starts

thread starts

thread ends

thread join
Implementation View

- Per-thread stack and instruction pointer
  - Saved in memory when thread suspended
  - Put in hardware esp/eip when thread resumes
Threads vs. Processes

• Threads
  – Less expensive to create
  – Shared memory paradigm
  – Easier to program
  – Limited scalability (tens of threads)

• Processes
  – More expensive to create (system call)
  – Message-passing paradigm
  – Harder to program
  – Highly scalable (thousands of processes)
Programming Threads

• Threads are available in many languages
  – C, C++, Objective Caml, Java, SmallTalk …

• In many languages (e.g., C and C++), threads
  are a platform specific add-on
  – Not part of the language specification

• They're part of the Java language specification
• Thread and Monitor modules in Ruby
Ruby Threads

- **Create thread using** `Thread.new`
  - **New** method takes code block argument
    
    ```ruby
    t = Thread.new { ...body of thread... }
    t = Thread.new (arg) { | arg | ...body of thread... }
    ```
  - **Join** method waits for thread to complete
    ```ruby
t.join
    ```

- **Example:**
  ```ruby
  myThread = Thread.new { 
      sleep 1    # sleep for 1 second
      puts "New thread awake!"
      $stdout.flush # flush makes sure output is seen
  }
  ```
Thread Lifecycle

• While a thread executes, it goes through a number of different phases
  – **New**: created but not yet started
  – **Runnable**: can run on a free CPU
  – **Running**: currently executing on a CPU
  – **Blocked**: waiting for I/O or on a lock
  – **Sleeping**: paused for a user-specified interval
  – **Terminated**: completed
Which Thread to Run Next?

- Look at all runnable threads
  - A good choice to run is one that just became unblocked because
    - A lock was released (we’ll see this in a minute)
    - I/O became available
    - It finished sleeping, etc.

- Pick a thread and start running it
  - Higher-priority threads get preference
  - Handled by the system or VM scheduler
Scheduling

One process per CPU

CPU 1
- p1
- p2

CPU 2
- p1
- p2

p2 threads: [Blue]  p1 threads: [Teal]
Scheduling

Threads shared between CPUs

CPU 1
- p1
- p2

CPU 2
- p1
- p2

p2 threads: [diagram showing p2 threads]

p1 threads: [diagram showing p1 threads]
Concurrency and Shared Data

• Concurrency is easy if threads don’t interact
  – Each thread does its own thing, ignoring other threads
  – Typically, however, threads need to communicate with each other

• Communication is done by sharing data
  – Different threads may access the heap simultaneously
  – But the scheduler might interleave threads arbitrarily
  – Problems can occur if we’re not careful
Problem: Data Race

- Multiple processes may attempt to modify the same value at the same time, and their edits may conflict.
Data Race Example

static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

Start: both threads ready to run. Each will increment the global cnt.

Shared state   cnt = 0
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state  cnt = 0

y = 0

T1 executes, grabbing the global counter value into its own y.
Data Race Example

static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

Shared state  cnt = 1

T1 executes again, storing its value of y + 1 into the counter.
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

*Shared state* \( cnt = 1 \)

\[ y = 0 \]

\[ y = 1 \]

*T1 finishes.  T2 executes, grabbing the global counter value into its own y.*
Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

*Shared state*  \( \text{cnt} = 2 \)

\( y = 0 \)

\( y = 1 \)

*T2 executes, storing its incremented \( \text{cnt} \) value into the global counter.*
But When it's Run Again?
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Start: both threads ready to run. Each will increment the global count.

Shared state  cnt = 0
Data Race Example

static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

T1 executes, grabbing the global counter value into its own y.
Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

*Shared state* $cnt = 0$

$y = 0$

$T1$ is preempted. $T2$ executes, grabbing the global counter value into its own $y$. 

$y = 0$
Data Race Example

```java
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

Shared state  cnt = 1

T2 executes, storing the incremented cnt value.
```
Data Race Example

static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}

Shared state  cnt = 1

y = 0

T2 completes. T1 executes again, storing the incremented original counter value (1) rather than what the incremented updated value would have been (2)!
What Happened?

• Different schedules led to different outcomes
  – This is a *data race* or *race condition*

• A thread was preempted in the middle of an operation
  – Reading and writing `cnt` was supposed to be *atomic*—to happen with no interference from other threads
  – But the schedule (interleaving of threads) which was chosen allowed atomicity to be violated
  – These bugs can be extremely hard to reproduce, and so hard to debug
    • Depends on what scheduler chose to do, which is hard to predict
Question

• If instead of
  ```java
  int y = cnt;
  cnt = y+1;
  ```
• We had written
  ```java
  cnt++;
  ```
• Would the result be any different?
• Answer: NO!
  ```java
  – Don’t depend on your intuition about atomicity
  ```
Question

• If you run a program with a race condition, will you always get an unexpected result?
  – No! It depends on the scheduler and on the other threads/processes/etc, that are running on the same machine

• Race conditions are very hard to find
Synchronization

- Refers to mechanisms allowing a programmer to control the execution order of some operations across different threads in a concurrent program.
- Different languages have adopted different mechanisms to allow the programmer to synchronize threads.
- Primary mechanism: locks/mutexes
Locks (Java) and Mutexes (Ruby)

```ruby
class Lock
  void lock();
  void unlock();
end
```

- Ruby: Mutex class in Thread library
- Only one thread can hold a lock at once
  - Other threads that try to acquire it `block` (or become suspended) until the lock becomes available
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

*Shared state* \( \text{cnt} = 0 \)

*\( T1 \) acquires the lock*
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}

\[\text{Shared state} \quad \text{cnt} = 0\]

\[y = 0\]

\[\text{T1 reads cnt into } y\]
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

*Shared state*  \( \text{cnt} = 0 \)

\( y = 0 \)

*T1 is preempted.*
*T2 attempts to acquire the lock but fails because it’s held by T1, so it blocks*
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

**Shared state**  \( \text{cnt} = 1 \)

**y = 0**

**T1 runs, assigning to cnt**
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

Shared state  \( \text{cnt} = 1 \)

\( y = 0 \)

\( T1 \) releases the lock and terminates
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}

T2 now can acquire the lock.

Shared state cnt = 1

cnt = 1

y = 0
Applying Synchronization

```java
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}

t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

Shared state \(cnt = 1\)

\[y = 0\]

\(T2\) reads \(cnt\) into \(y\).

\[y = 1\]
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
t2.run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
}
```

- **Shared state** $cnt = 2$
- $y = 0$
- $y = 1$
- **T2 assigns cnt,**
  then releases the lock
Another Problem: Deadlock

- **Deadlock** occurs when no thread can run because all threads are waiting for a lock
  - No thread running, so no thread can ever release a lock to enable another thread to run

This code can deadlock…
-- when will it work?
-- when will it deadlock?

```java
Lock l = new Lock();
Lock m = new Lock();

Thread 1
l.lock();
m.lock();
...
m.unlock();
l.unlock();

Thread 2
m.lock();
l.lock();
...
l.unlock();
m.unlock();
```
Deadlock (cont’d)

• Some schedules work fine
  – Thread 1 runs to completion, then thread 2

• But what if...
  – Thread 1 acquires lock $l$
  – The scheduler switches to thread 2
  – Thread 2 acquires lock $m$

• Deadlock!
  – Thread 1 is trying to acquire $m$
  – Thread 2 is trying to acquire $l$
  – And neither can, because the other thread has it
Wait Graphs

Thread T1 holds lock $l$

Thread T2 attempting to acquire lock $m$

Deadlock occurs when there is a cycle in the graph
**Wait Graph Example**

T1 holds lock on l
T2 holds lock on m
T1 is trying to acquire a lock on m
T2 is trying to acquire a lock on l
Deadlock Conditions (Coffman)

• Mutual Exclusion
  – At least one resource must be non-sharable

• Hold and Wait
  – At least one process must be simultaneously holding and requesting resources

• No Pre-emption
  – The operating system cannot (or will not) break the deadlock by killing a process

• Circular Wait
  – Wait graphs...
Dealing with deadlock

• Ignore it
  – The “ostrich algorithm”

• Detect it and recover from it
  – Kill or roll back a process
  – Re-allocate a resource

• Avoid it
  – Don’t allow resource acquisition if it will lead to deadlock (Banker’s algorithm)

• Prevention
  – Remove all possibility of one of the Coffman conditions
Classic: Dining Philosophers Problem

- Philosophers either eat or think
- They must have two forks to eat
- Can only use forks on either side of their plate
- No talking!
- Avoid deadlock and starvation!
Bad Dining Philosophers Solution 1

• Philosophers all pick up the left fork first

• Deadlock!
  – all are holding the left fork and waiting for the right fork
Bad Dining Philosophers Solution 2

- Philosophers all pick up the left fork first
- Philosophers put down a fork after waiting for 5 minutes, then wait 5 minutes before picking it up again
- Starvation!
Possible Solutions

• The waiter solution
  – Third party arbiter (scheduler)
  – Each thread requests permission before acquiring a resource

• The resource hierarchy solution
  – Impose ordering on resources
  – Must obtain resources in order
  – Most practical solution
  – Sometimes hard to know in advance
Dining Philosophers Solution

• Number the philosophers
• Start by giving the fork to the philosopher with lower number. Initially, all forks are dirty.
• When a philosopher wants both forks, he sends a message to his neighbors
• When a philosopher with a fork receives a message: if his fork is clean, he keeps it, otherwise he cleans it and gives it up.
• After a philosopher eats, his forks are dirty. If a philosopher had requested his fork, he cleans it and sends it.
Dining Philosophers Example

Each philosopher begins with the forks shown.

All are dirty.
Dining Philosophers Example

Philosopher 2 sends a message to philosopher 1 that he wants his fork.

Their shared fork is dirty, so philosopher 1 cleans it and sends it.
Dining Philosophers Example

Philosopher 2 eats!

While he is eating philosopher 3 requests their shared fork.

Philosopher 2 is done eating, so his forks become dirty.
Dining Philosophers Example

Philosopher 2 is done eating, so he honors philosopher 3’s request and cleans the fork and sends it.

Philosopher 3 eats!
Philosophers Implementation Needs

• Wait until notified about something by another philosopher
  – stay hungry until you have two forks
  – hold onto your fork until your neighbor needs it

• Send a message to a philosopher and have it processed at a later time
  – multiple philosophers can send messages to one
  – when philosopher done eating he should process all

… and here’s another problem with these needs…
Producer/Consumer Problem

• Suppose we are communicating with a shared variable
  – E.g., some kind of a fixed size buffer holding messages

• One thread *produces* input to the buffer
• One thread *consumes* data from the buffer

• Rules:
  – producer can’t add input to the buffer if it’s full
  – consumer can’t take input from the buffer if it’s empty
Producer / Consumer Idea

If the buffer is partially full, producer or consumer can run:

```
producer                      consumer
```

If the buffer is empty, only the producer can run:

```
producer                      consumer
```

If the buffer is full, only the consumer can run:

```
producer                      consumer
```
Needed Solution

• Need a way of having threads “wait” on a resource
• Also need a way to “notify” waiting threads when they can wake up
• This is usually called a “monitor”
Ruby Locks

- **Monitor, Mutex**
  - Objects 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

```ruby
require 'monitor.rb'
myLock = Monitor.new
myLock.synchronize {
  # myLock held during this code block
}
```
Ruby Conditions

- 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

```ruby
myLock = Monitor.new
myCondition = myLock.new_cond
myLock.synchronize {
  myCondition.wait_while { y > 0 }
  myCondition.wait_until { x != 0 }
}
myLock.synchronize {
  myCondition.broadcast
}
```
Parking Lot Example

```ruby
require "monitor.rb"

class ParkingLot
  def initialize
    @numCars = 0
    @myLock = Monitor.new
    @myCondition = @myLock.new_cond
  end

  def addCar
    ...
  end

  def removeCar
    ...
  end
end
```

CMSC 330
Parking Lot Example

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

CMSC 330
Parking Lot Example

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
valet2.join
Ruby vs. Java Threads

• 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`
Review

• Multiprocessing
  – Processes vs. threads

• Problems
  – Data races/conditions
  – Deadlock
  – Dining philosophers
  – Producers/consumers

• Tools
  – Locks/mutexes
  – Monitors
Aside: Functional Programming

• Pure functional languages
  – No side effects
  – No memory access
  – No data races!
• Much easier to parallelize functional programs
What’s Next?

• Extension to existing language...
  – Universal Parallel C (UPC)
  – High-Performance Fortran (HPF)
  – POSIX threads
• ...or an entirely new language?
  – Chapel (Cray)
  – X10 (IBM)

• No clear consensus yet