CMSC330

Multithreading
• Description
  – Multiple processing units (multiprocessor)
  – From single microprocessor to large compute clusters
  – Can perform multiple tasks in parallel simultaneously

Intel Core 2 Quad 6600
32 processor Pentium Xeon
106K processor IBM BlueGene/L
Computation Abstractions

A computer
Processes vs. Threads

Processes do not share data

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

• Conceptually
  – Parallel computation occurring within a process

• Implementation view
  – A program counter and a stack
  – Heap and static area are shared among all threads

• All programs have at least one thread (main)
  – Child threads are forked, and then join with main thread
Programming Threads

- Thread creation is inexpensive
- Threads reside on same physical processor
- Threads share memory, resources
  - Except for local thread variables
- Shared-memory programming paradigm
  - Threads communicate via shared data
  - Synchronization used to avoid data races
- Limited scalability (10’s of threads)
Programming Processes

• Process creation is expensive
  – Request to operating system
• Processes may reside on separate processors
• Processes do not share memory
• Message-passing programming paradigm
  – Messages using I/O streams, sockets, network, files
• Processes must cooperate to communicate
  – Actions performed to send and receive data
• Highly scalable (1000’s of processors)
Java Threads Review
(from CMSC 132)

• **Thread** class & **Runnable** interface
  – Used to create / manipulate threads

• Run-time scheduler
  – Preemptive / non-preemptive
  – Thread states (new, runnable, blocked, dead)

• Data race
  – Concurrent accesses to same shared object
    • Where at least one access is a write
  – Result may change depending on thread schedule
  – Very difficult to detect & correct
Java Threads Review (cont.)

• Synchronization
  – Locks ensure exclusive access
    • Lock associated w/ every Java object
  – Use **synchronized** keyword to acquire lock
    • Code blocks –  
      ```java
      synchronized (o) { ... }
      // lock for Object o
      ```
    • Methods –  
      ```java
      synchronized foo( ) { ... }
      // lock for this
      ```
  – Thread blocks when trying to acquire locked lock
    • Thread returns when lock is finally acquired
    • May **deadlock** if threads try to acquire each other’s lock
Concurrency

- A concurrent program
  - Program with multiple threads that may be active at the same time

- Might run on one CPU (multi-tasking)
  - The CPU alternates between running different threads
  - The scheduler takes care of the details
    - Switching between threads might happen at any time

- Might run in parallel on a multiprocessor machine
  - May have multiple threads per CPU
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

• In multithreaded programs, communication is achieved by sharing data
  – In Java, different threads may access the heap simultaneously
  – But the scheduler might interleave threads arbitrarily
  – Problems can occur if we’re not careful
Data Race

• Definition
  – Concurrent accesses to same shared variable, where at least one access is a write

• Properties
  – Order of accesses may change result of program
  – May cause intermittent errors, very hard to debug
public class Example extends Thread {
    private static int cnt = 0; // shared state
    public void run() {
        int y = cnt;
        cnt = y + 1;
    }
    public static void main(String args[]) {
        Thread t1 = new Example();
        Thread t2 = new Example();
        t1.start();
        t2.start();
    }
}
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

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

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

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

*T1 executes again, storing its value of } y + 1 \text{ 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 \( \text{cnt} = 1 \)

\( \text{cnt} = 1 \)

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  cnt = 2

T2 executes, storing its incremented cnt value into the global counter.
Data Race Example – 2\textsuperscript{nd} Try

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

*Shared state*  
\[ cnt = 0 \]

*Start: both threads ready to run. Each will increment the global count.*
Data Race Example – 2\textsuperscript{nd} Try

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

Shared state \hspace{1cm} cnt = 0

T1 executes, grabbing the global counter value into its own y.
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;        y = 0
}
t2.run() {
    int y = cnt;
    cnt = y + 1;        y = 0
}

Shared state    cnt = 0

T1 is preempted. T2 executes, grabbing the global counter value into its own y.
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;  \textit{Shared state} \hspace{1cm} \textit{cnt} = 1
}
t2.run() {
    int y = cnt;
    cnt = y + 1;  \hspace{1cm} \textit{T2 executes, storing the}
    \hspace{1cm} \textit{incremented cnt value.}  \hspace{1cm} \textit{y} = 0
}
static int cnt = 0;

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

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

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

\(T2\) 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
    - Execute 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!
  
  – Don’t depend on your intuition about atomicity
What’s Wrong with the Following?

```c
static int cnt = 0;
static int x = 0;
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while (x != 0);</code></td>
<td><code>while (x != 0);</code></td>
</tr>
<tr>
<td><code>x = 1;</code></td>
<td><code>x = 1;</code></td>
</tr>
<tr>
<td><code>cnt++;</code></td>
<td><code>cnt++;</code></td>
</tr>
<tr>
<td><code>x = 0;</code></td>
<td><code>x = 0;</code></td>
</tr>
</tbody>
</table>

• Threads may be interrupted after the `while` but before the assignment `x = 1`  
  – Both may think they “hold” the lock!
• This approach is called **busy waiting**  
  – Consumes lots of processor cycles
Synchronization

• Definition
  – Coordination of events with respect to time

• Properties
  – Can eliminate data races in multithreaded programs
  – Overhead → excessive use reduces performance

• Mechanisms
  – Different in each programming language
  – Look at examples in Java
Synchronized

• This pattern is really common
  – Acquire lock, do something, release lock under any circumstances after we’re done
    • Even if exception was raised etc.

• Java has a language construct for this
  – `synchronized (obj) { body }`
    • Every Java object has an implicit associated lock
  – Obtains the lock associated with `obj`
  – Executes `body`
  – Release lock when scope is exited
    • Even in cases of exception or method return
Example

```java
static Object o = new Object();

void f() throws Exception {
    synchronized (o) {
        FileInputStream f =
            new FileInputStream("file.txt");
        // Do something with f
        f.close();
    }
}
```

– Lock associated with `o` acquired before body executed
  • Released even if exception thrown
Example: Synchronizing on this

```java
class C {
    int cnt;

    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
}
```

```java
C c = new C();
Thread 1
c.inc();
Thread 2
c.inc();
```

• Does this program have a data race?
  – No, both threads acquire locks on the same object before they access shared data
Example: Synchronizing on this (cont.)

```java
class C {
    int cnt;

    void inc() {
        synchronized (this) {
            cnt++;
        }
    }

    void dec() {
        synchronized (this) {
            cnt--;
        }
    }
}
```

C c = new C();

Thread 1
c.inc();

Thread 2
c.dec();

• Data race?
  - No, threads acquire locks on the same object before they access shared data
Synchronized Methods

• Marking method as synchronized same as synchronizing on this in body of the method
  – The following two programs are the same

```java
class C {
    int cnt;
    
    void inc() {
        synchronized (this) {
            cnt++;
        }
    }
}
```

```java
class C {
    int cnt;
    
    synchronized void inc() {
        cnt++;
    }
}
```
New Java Thread Topics

• Lock interface
  – lock( )
  – unlock( )

• ReentrantLock class

• Condition interface
  – await( )
  – signalAll( )
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`
- **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
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();
    }
    ...
}
ReentrantLock Class (Java 1.5)

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

- **Reentrant lock**
  - Can be reacquired by same thread by invoking `lock()`
  - 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
Reentrant Lock Example

```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();
}
```

- `returnAndInc()` can acquire lock and invoke `inc()`
- `inc()` can acquire lock without having to worry about whether thread already has lock
Producer / Consumer Problem

- Suppose we are communicating with a shared variable
  - E.g., 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
The Dining Philosophers Problem

- **Philosopher**
  - Thinks & eats

- **To eat**
  - Must have two forks
  - Can only use forks on either side of plate

- **Goal**
  - Avoid deadlock
  - Avoid starvation

- **Fancy version of producer / consumer**
Broken Producer / Consumer Code

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(Object o) {
    lock.lock();
    while (valueReady);
    value = o;
    valueReady = true;
    lock.unlock();
}

Object consume() {
    lock.lock();
    while (!valueReady);
    Object o = value;
    valueReady = false;
    lock.unlock();
}
```

- Threads wait with lock held – deadlock
Broken Producer / Consumer Code

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(Object o) {
    while (valueReady);
    lock.lock();
    value = o;
    valueReady = true;
    lock.unlock();
}

Object consume() {
    while (!valueReady);
    lock.lock();
    Object o = value;
    valueReady = false;
    lock.unlock();
}
```

- valueReady accessed without a lock held – data race
Bad Producer / Consumer Code

```java
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;

void produce(Object o) {
    while (true) {
        lock.lock();
        if (!valueReady) {
            value = o;
            valueReady = true;
        }
        lock.unlock();
    }
}

Object consume() {
    while (true) {
        lock.lock();
        if (valueReady) {
            Object o = value;
            valueReady = false;
        }
        lock.unlock();
    }
}

- Constantly acquiring / releasing lock – busy wait
```
Solving Producer / Consumer Problem

- Difficult to use locks directly
  - Very hard to get right
  - Problems often very subtle
- Another approach – use **Condition** interface
  - Condition is created from **Lock** object
  - Allows threads to sleep while waiting to acquire lock
  - Can wake up sleeping threads before releasing lock

```java
interface Lock {
  Condition newCondition(); ...
}
interface Condition {
  void await();
  void signalAll(); ...
}
```
Condition (Java 1.5)

- 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
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){
        ready.await();
    }
    Object o = buffer;
    bufferReady = false;
    ready.signalAll();
    lock.unlock();
}
```
Await and SignalAll Gotcha's

- **await() must** be called in a while loop
  - Conditions may not be met when await returns
  - Some other thread may have awoken first
    - And changed condition (e.g., consumed item in buffer)

- Avoid holding other locks when waiting
  - **await()** only gives up lock on the object you are waiting on
  - Reduces possibility of deadlock
Key Ideas

• Multiple threads can run simultaneously
  – Either truly in parallel on a multiprocessor
  – Or can be scheduled on a single processor
    • A running thread can be pre-empted at any time

• Threads can share data
  – In Java, only fields can be shared
  – Need to prevent data races
    • Rule of thumb 1: You must hold a lock when accessing shared data
    • Rule of thumb 2: You must not release a lock until shared data is in a valid state
  – Overuse use of synchronization can create deadlock
    • Rule of thumb: No deadlock if only one lock
Deadlock theory

• 4 necessary conditions:
  – Mutual exclusion of a resource
  – Hold and wait
  – No preemption
  – Circular wait
Deadlock and Halting Problem

- Detecting the possibility of a deadlock before it occurs is generally undecidable!
  - Epic fail!
- What can we do?
  - Deadlock prevention
  - Deadlock avoidance
    - Only grant resources if going to be in safe state