CMSC 330: Organization of Programming Languages

Multithreading

Multiprocessors

- 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
Concurrency

Important & pervasive topic in CS
Currently covered in
• CMSC 132 – object-oriented programming II
  ➢ Java threads, data races, synchronization
• CMSC 216 – low level programming / computer systems
  ➢ C threads
• CMSC 411/430 – architectures / compilers
  ➢ Instruction level parallelism
• CMSC 412 – operating systems
  ➢ Concurrent processes
• CMSC 424 – database design
  ➢ Concurrent transactions
• CMSC 433 – programming language technologies
  ➢ Advanced synchronization and parallelization
• CMSC 451 – algorithms
  ➢ Parallel algorithms

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, …
Computation Abstractions

A computer

Processes vs. Threads

Processes do not share data

Threads share data within a process
The schedulers do most of the heavy lifting.

But they don’t know the semantics of what the threads are actually doing.

---

So, What Is a Thread?

- **Fundamental unit of execution**
  - All programs have at least one thread (main)

- **Implementation view**
  - A program counter and a stack
  - Heap and static area are shared among all threads
Implementation View

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

Programming Processes

- Process creation is expensive
  - Stack, heap, PC, code, OS state
- Highly scalable
  - Virtually unlimited
- Processes may reside on separate processors
  - Sharing memory typically too expensive
- Message-passing programming paradigm
  - I/O streams, sockets, network, files
  - Cooperation key to communication (send/recv)
Programming Threads

- Thread creation is inexpensive
  - Stack, program counter, scheduler state
- Traditionally, they don’t scale well (10s of threads)
  - Multicore architectures relax this limitation
- Threads reside on same physical processor
  - So memory sharing is cheap
- Shared-memory programming paradigm
  - Everything except thread local variables are shared
  - Threads communicate via shared data
  - Synchronization used to avoid data races

What Is the Big Deal About Threads?

- Conventional wisdom: threads are hard
- Main reason: non-determinism
  - Different threads execute at different speeds
  - This leads to unpredictable results
  - When you program with threads, you have to be ready for unexpected results
- The goal of the PL designer is to make this easier
Programming Languages & Threads

- **Old: libraries**
  - pthreads
  - Could use different libraries for different properties

- **New: primitives**
  - Java, Ruby, OCaml
  - Can utilize special keywords, syntax
  - Not dependent on operating system

Which is better?

Example

- x = 0 initially. Then these threads are executed:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = x;</td>
<td>z = x;</td>
</tr>
<tr>
<td>x = y+1;</td>
<td>x = z+2;</td>
</tr>
</tbody>
</table>

- What is the value of x afterward?

  3 1 2
Data Races

- That was an example of a data race
  - Threads are “racing” to read, write x
  - The value of x depends on who “wins” (3, 1, 2)
- Languages rarely specify who wins data races
  - The outcome is nondeterministic
- So programmers restrict certain outcomes
  - Synchronization with locks, condition variables
- And they often mess up
  - Deadlocks, bugs that are hard to track down…

Thread API Concepts

- Thread management
  - Creating, killing, joining (waiting for) threads
  - Sleeping, yielding, prioritizing
- Synchronization
  - Controlling order of execution, visibility, atomicity
  - Locks: Can prevent data races, but watch out for deadlock!
  - Condition variables: supports communication between threads
- Most languages have similar APIs, details differ
Java – Creating Threads

- Thread.create(Runnable r)
  - Or subclass the Thread class
  - Java makes it hard to create threads that access local variables (since it does not have closures)

- In practice, there are better ways
  - Use thread pools, to separate the idea of creating a thread from creating a (Runnable) task
    - May have N threads execute M > N jobs

- We’ll stick with the simple idea here

Thread Creation Example

```java
public class MyT implements Runnable {
    public void run() {
        ...
        // particular work for this thread
    }
}

Thread t = new Thread(new MyT());  // create thread
t.start();  // begin running thread
...
// thread executing in parallel
t.join();  // waits for thread to exit
```
Locks

Language designers limit non-determinism by introducing concurrency-control constructs
- Make some parts deterministic = more predictable
- Trade-off: reducing concurrency can reduce performance

Common concurrency-control feature: locks
- They “guard” shared resources that shouldn’t be accessed by more than one thread at a time

The gist
- At most one owner at a time
- If someone else owns it, you block
- When you’re done with it, release ownership

Java Locks (1.4)

Objects each have an associated lock

Use synchronized keyword to acquire lock
- Code blocks – synchronized (o) { … } // lock for Object o
- Methods – synchronized foo() { … } // lock for this

Thread blocks when lock held
- Thread returns when lock is finally acquired
- May deadlock if threads try to acquire each other’s lock

Locks sometimes referred to as mutexes
Why Locks?

- #1 concern: prevent data races
- Patterns of use:
  - Enforce atomicity of shared data
    - 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
- Synchronization also used to ensure ordering and visibility
  - The last is due to memory models in modern arch’s

Synchronization Example (Java 1.4)

```java
public class Example extends Thread {
    private static int cnt = 0;
    public void run() {
        synchronized (this) {
            int y = cnt;
            cnt = y + 1;
        }
    }
}
```

**Acquires** the lock associated w/ current object; only succeeds if lock not held by another thread, otherwise blocks

**Releases** the 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

Producer / Consumer Idea

- If buffer is partially full, producer or consumer can run
  - Producer produces input: c b a
  - Consumer consumes input

- If buffer is empty, only producer can run
  - Producer produces input

- If buffer is full, only consumer can run
  - Consumer consumes input: e d c b a
Broken Producer/Consumer Example

```java
boolean valueReady = false;
Object value;

void produce(object o) {
    synchronized (this) {
        while (valueReady)
            ;
        value = o;
        valueReady = true;
    }
}

Object consume() {
    synchronized (this) {
        while (!valueReady)
            ;
        Object o = value;
        valueReady = false;
        return o;
    }
}

Threads wait with lock held – no way to make progress
```

Broken Producer/Consumer Example

```java
boolean valueReady = false;
Object value;

void produce(object o) {
    synchronized (this) {
        while (valueReady)
            ;
        value = o;
        valueReady = true;
    }
}

Object consume() {
    synchronized (this) {
        while (!valueReady)
            ;
        Object o = value;
        valueReady = false;
        return o;
    }
}

valueReady accessed without a lock held – data race
Inefficient Producer/Consumer Example

```java
boolean valueReady = false;
Object value;

Constantly acquiring / releasing lock — busy wait

void produce(Object o) {
    boolean done = false;
    while (!done) {
        synchronized (this) {
            if (!valueReady) {
                value = o;
                valueReady = true;
                done = true;
            }
        }
    }
}

Object consume() {
    Object o = null;
    boolean done = false;
    while (!done) {
        synchronized (this) {
            if (valueReady) {
                o = value;
                valueReady = false;
                done = true;
            }
        }
    }
    return o;
}
```

Solving Producer / Consumer Problem

- Difficult to use locks directly
  - Very hard to get correct (or efficient) solution
  - Problems often very subtle
- Proper approach — use signaling
- Common signaling scenario
  1. You get the lock
  2. You realize it’s no good to you yet (buffer is empty)
  3. Go to sleep: “wake me up when there’s work to do”
- Virtually every threading model supports this
  - Condition variables
  - Wait / notify
Condition Variables

- Each condition variable represents those threads waiting for a condition to become true
  - Implemented, at least conceptually, as a wait set associated with the condition variable

- Since different threads access the variable at once, we must protect the wait set contents with a lock

Condition Variables in Java 1.4

- Use `synchronized` on object to get associated lock

```
Object o
```

```
  o’s lock
```

```
  o’s wait set
```

- Objects also have an associated condition variable (and thus a wait set)
Wait and NotifyAll

- o.wait()
  - Must hold lock associated with o
  - Release that lock
    - And no other locks
  - Adds this thread to wait set for lock
  - Blocks the thread

- o.notifyAll()
  - Must hold lock associated with o
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - This is part of the function; you don't need to do it explicitly

Producer/Consumer Example

```java
boolean valueReady = false;
Object value;

void produce(Object o) {
    synchronized (this) {
        while (valueReady)
            wait();
        value = o;
        valueReady = true;
        notifyAll();
    }
}

Object consume() {
    synchronized (this) {
        while (!valueReady)
            wait();
        Object o = value;
        valueReady = false;
        notifyAll();
        return o;
    }
}
```
Using Conditions Correctly

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

- Avoid holding other locks when waiting
  - `wait()` only gives up lock on object you are waiting on
  - Reduces possibility of deadlock

---

Broken Producer/Consumer Example

```java
boolean valueReady = false;
Object value;

void produce(Object o) {
    synchronized (this) {
        if (valueReady)
            wait();
        value = o;
        valueReady = true;
        notifyAll();
    }
}

Object consume() {
    synchronized (this) {
        if (!valueReady)
            wait();
        Object o = value;
        valueReady = false;
        notifyAll();
        return o;
    }
}
```

- Illegal access if multiple producers or consumers
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

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

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 lock not held by another thread, otherwise blocks

**Releases** the lock
ReentrantLock Class (Java 1.5)

- 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

---

Reentrant Lock Example

```java
class ReentrantLock implements 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
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

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)

```
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();
    return o;
}
```

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

Producer / Consumer Solution (Java 1.5)

```
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();
    return o;
}
```

- Uses 2 conditions per lock for greater efficiency