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 pthreads
  - 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
    - *Moral successor to ideas discussed in this lecture and the next*
  - 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 (e.g., JVM’s)

Threads
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
Per-thread stack and instruction pointer

- Saved in memory when thread suspended
- Put in hardware esp/eip when thread resumes
Programming Languages & Threads

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

- **New: primitives**
  - Java, Ruby, OCaml
  - Can utilize special keywords, syntax
  - Better integration into assumptions of the language
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 less expensive
  - Stack, program counter, scheduler state
- Threads may reside on same physical processor
  - So memory sharing is cheap
- Modern architectures can scale threads well
  - Depends on implementation; e.g., hundreds of thousands of Erlang threads
- 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 of scheduling
  - Different threads execute at different speeds
    - Actions on memory are interleaved
  - This leads to unpredictable results
  - When you program with threads, you have to consider all possible ways that threads can interact
- The goal of the PL designer is to make this easier
Example

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

\[
\begin{align*}
T1 & \quad y = x; \\
T2 & \quad z = x; \\
& \quad x = y+1; \\
& \quad x = z+2;
\end{align*}
\]

- What is the value of x afterward? 3 1 2

\[
\begin{align*}
T1 & \quad y = x; \\
T2 & \quad x = y+1; \\
& \quad z = x; \\
& \quad x = z+2;
\end{align*}
\]

\[
\begin{align*}
T1 & \quad y = x; \\
T2 & \quad z = x; \\
& \quad x = y+1; \\
& \quad x = z+2;
\end{align*}
\]

\[
\begin{align*}
T1 & \quad y = x; \\
T2 & \quad z = x; \\
& \quad y = x; \\
& \quad x = y+1;
\end{align*}
\]

\[
\begin{align*}
T1 & \quad y = x; \\
T2 & \quad z = x; \\
& \quad y = x; \\
& \quad x = y+1;
\end{align*}
\]
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
  - Leading to 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

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 Intrinsic Locks

- Objects each have an associated intrinsic 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 held at a time
- Synchronization also used to ensure ordering and visibility
  - The last is due to memory models in modern arch’s
Synchronization Example

public class Example {
    private int cnt = 0;
    public void increment() {
        synchronized (this) {
            int y = cnt;
            cnt = y + 1;
        }
    }
    ...
}

Acquires the lock associated with the current object; only succeeds if lock not held by another thread, otherwise blocks.

Releases the lock.
Synchronization Example, retold

```java
public class Example {
    private int cnt = 0;
    public synchronized void increment() {
        int y = cnt;
        cnt = y + 1;
    }
    ...
}
```

*Acquires* the lock associated with current object on entering the method

*Releases* the lock on method exit
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

- If buffer is empty, only producer can run

- If buffer is full, only consumer can run
Broken Producer/Consumer Example

boolean valueReady = false;
Object value; // one-place buffer

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

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

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

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 only
  - Very hard to get correct (or efficient) solution
  - Problems 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
    - Operations: *Wait* and *notify*
Condition Variables

- A condition variable represents a set of threads waiting for a condition to become true
  - Implemented, at least conceptually, as a wait set

- Since different threads may access the variable at once, we protect the wait set with a lock
  - Thus avoiding possible data races
Condition Variables and Intrinsic Locks

- **synchronized accesses intrinsic lock**

- Objects also have an intrinsic condition variable (and thus a wait set)

```plaintext
Object o

o's intrinsic lock

o's wait set
```
Wait and NotifyAll

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

- o.notifyAll()
  - Must hold o’s intrinsic lock
  - Resumes all threads on o’s wait set
  - Those threads will reacquire lock before continuing
    - This is part of the function; you don’t need to do it explicitly
Producer/Consumer Example

boolean valueReady = false;
Object value;

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

Object consume() {
    synchronized (this) {
        while (!valueReady)
            this.wait();
        Object o = value;
        valueReady = false;
        this.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)

- Prefer `notifyAll()` to the alternative, `notify()`

- Avoid holding (other) locks when waiting
  - `wait()` only gives up lock on object you are waiting on
  - Reduces possibility of deadlock
    - Called Nested Monitor Lockout
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
Notify vs. NotifyAll

- Notify( ) is like notifyAll( ), but wakes up a single thread on the wait set, not all threads
  - Can be more efficient, since if you have a lot of threads waiting, most will simply fail to reacquire the lock, wasting effort

- But, easy to use notify( ) incorrectly
  - Leading to a kind of deadlock
Broken Producer/Consumer Example

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

void produce(Object o) {
    synchronized (this) {
        while (valueReady)
            wait();
        value = o;
        valueReady = true;
        notify();
    }
    BAD: Could wake up another producer
}
```

```java
Object consume() {
    synchronized (this) {
        while (!valueReady)
            wait();
        Object o = value;
        valueReady = false;
        notify();
        return o;
    }
    BAD: Could wake up another consumer
}
```

- Notify only wakes up one thread – could be wrong kind
Lock Interface (Java 1.5 and later)

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

- **Explicit Lock objects**
  - `ReentrantLock` implements `Lock`
    - same behavior as an intrinsic lock

- **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 Object lock = new Object();
    public void run() {
        synchronized (lock) {
            int y = cnt;
            cnt = y + 1;
        }
    }
}
Synchronization, with explicit Locks

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
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
Reentrant Lock Example

Example

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

```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();
}
```
Condition Interface (Java 1.5 and later)

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 `signallAll()`

- 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 `signallAll()` 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();
    return o;
}
```

- Uses single condition per lock (like intrinsics)
Producer / Consumer Solution

```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();
    return o;
}

- Uses 2 conditions per lock for greater efficiency
```
Producer / Consumer Solution

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

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

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

- Wakes up only one thread: More efficient, still!
Note about unlock

- The prior examples were slightly simplified

```java
void produce(Object o) {
    lock.lock();
    try {
        if(bufferReady)
            producers.await();
        buffer = o;
        bufferReady = true;
        consumers.signal();
    } finally {
        lock.unlock();
    }
}
```

- Need to consider the possibility of exceptional exit
  - Handled automatically for intrinsic locks, when leaving synchronized blocks

- Use **finally** to ensure that lock is released