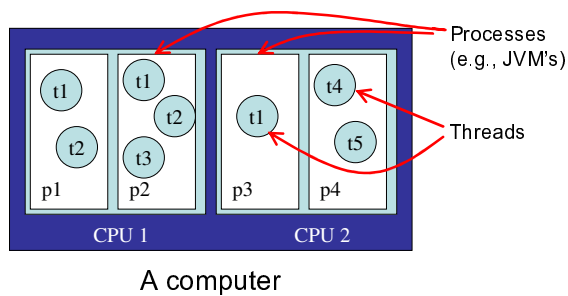


CMSC 330: Organization of Programming Languages

Multithreading

Computation Abstractions



CMSC 330

3

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

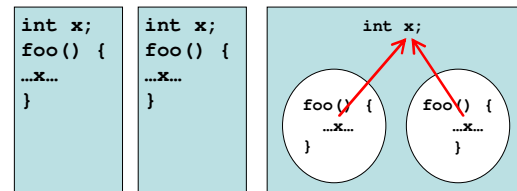


106K processor IBM BlueGene/L

CMSC 330

2

Processes vs. Threads



Processes do not share data

Threads share data within a process

CMSC 330

4

So, What Is a Thread?

Conceptually

- Parallel computation occurring within a process

Implementation view

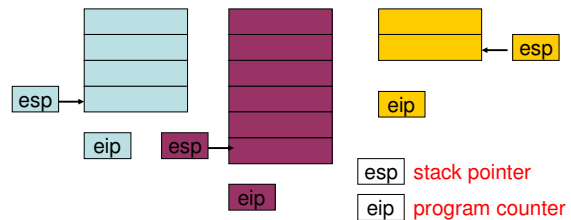
- A program counter and a stack
- Heap and static area are shared among all threads

- All programs have at least one thread (main)

CMSC 330

5

Implementation View



Per-thread stack and instruction pointer

- Saved in memory when thread suspended
- Put in hardware esp/eip when thread resumes

CMSC 330

6

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)

CMSC 330

7

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)

CMSC 330

8

Programming Languages & Threads

- ▶ Threads are available in many languages
 - C, C++, Java, Ruby, OCaml...
- ▶ In older languages (e.g., C and C++), threads are a platform specific add-on
 - Not part of the language specification
 - Implemented as code libraries (e.g., pthreads)
- ▶ In newer languages (e.g., Java, Ruby), threads are part of the language specification
 - Not dependent on operating system
 - Can utilize special keywords, syntax

CMSC 330

9

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

CMSC 330

10

Java Threads Review (cont.)

- ▶ Synchronization
 - Locks ensure exclusive access
 - > Lock associated w/ every Java object
 - Use **synchronized** keyword to acquire lock
 - > Code blocks – `synchronized (o) { ... }` // lock for Object o
 - > Methods – `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

CMSC 330

11

New Java Thread Topics

- ▶ Lock interface
 - lock()
 - unlock()
- ▶ ReentrantLock class
- ▶ Condition interface
 - await()
 - signalAll()

CMSC 330

12

Lock Interface (Java 1.5)

```
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

CMSC 330

13

Lock Synchronization Example

```
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

Releases the lock

CMSC 330

14

ReentrantLock Class (Java 1.5)

```
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

CMSC 330

15

Reentrant Lock Example

```
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

CMSC 330

16

Producer / Consumer Problem

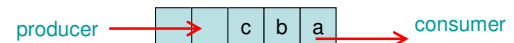
- ▶ Suppose we are communicating with a shared variable
 - E.g., a fixed size buffer holding messages
- ▶ One thread **produces** input to the buffer
- ▶ One thread **consumes** data from the buffer
- ▶ Rules
 - Producer can't add input to the buffer if it's full
 - Consumer can't take input from the buffer if it's empty

CMSC 330

17

Producer / Consumer Idea

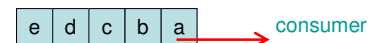
- ▶ If buffer is partially full, producer or consumer can run



- ▶ If buffer is empty, only producer can run



- ▶ If buffer is full, only consumer can run



CMSC 330

18

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

CMSC 330

19

Broken Producer / Consumer Code

```
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

CMSC 330

20

Broken Producer / Consumer Code

```
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

CMSC 330

21

Bad Producer / Consumer Code

```
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

CMSC 330

22

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

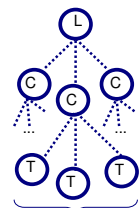
```
interface Lock { Condition newCondition(); ... }
interface Condition {
    void await();
    void signalAll(); ... }
}
```

CMSC 330

23

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



CMSC 330

24

Producer / Consumer Solution

```
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();
}
```

CMSC 330

25

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

CMSC 330

26

Broken Producer / Consumer Code

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

```
void produce(Object o) {
    lock.lock();
    if (valueReady) {
        ready.await();
    }
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
}

Object consume() {
    lock.lock();
    if (!valueReady) {
        ready.await();
    }
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
}
```

- ▶ Illegal access if **multiple** producers or consumers

CMSC 330

27

Producer / Consumer in Java 1.4

```
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;

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

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

Can use `wait()` & `notifyAll()` in conjunction w/ `synchronized`

synchronizes on lock for `this` waits using lock and wait set for `this`

CMSC 330

28

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

CMSC 330

29

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
}
```

CMSC 330

30

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`

CMSC 330

31

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
}
```

CMSC 330

32

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
}
```

CMSC 330

33

CMSC 330: Organization of Programming Languages

Review of Java Threads (Topics Covered in CMSC 132)

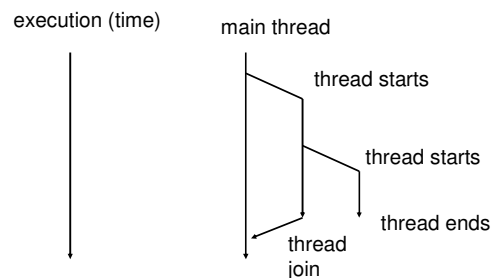
Java Threads

- ▶ Every application has at least one thread
 - The “main” thread, started by the JVM to run the application’s `main()` method
- ▶ `main()` can create other threads
 - Explicitly, using the `Thread` class
 - Implicitly, by calling libraries that create threads as a consequence
 - > RMI, AWT/Swing, Applets, etc.

CMSC 330

35

Thread Creation



CMSC 330

36

Creating Threads in Java

▶ Runnable Approach

1. Define class implementing Runnable interface

```
public interface Runnable {  
    public void run();  
}
```

2. Put work to be performed in run() method

3. Create instance of the “worker” class

4. Create thread to run it

- Create Thread object
 - Pass worker object to Thread constructor
- Or hand the worker instance to an executor
 - Alternative methods for running threads

CMSC 330

37

Creating Threads in Java (cont.)

▶ Example

```
public class MyT implements Runnable {  
    public void run() {  
        ... // work for thread  
    }  
}  
Thread t = new Thread(new MyT()); // create thread  
t.start(); // begin running thread  
... // thread executing in parallel
```

CMSC 330

38

Example Code – Alarms

▶ Goal

- Set alarms which will be triggered in the future

▶ Input

- Time t (seconds) and message m

▶ Result

- We'll see m printed after t seconds

CMSC 330

39

Example Code – Synchronous Alarms

```
while (true) {  
    System.out.print("Alarm> ");  
  
    // read user input  
    String line = b.readLine();  
    parseInput(line); // sets timeout  
  
    // wait (in secs)  
    try {  
        Thread.sleep(timeout * 1000);  
    } catch (InterruptedException e) { }  
    System.out.println(""+timeout+" "+msg);  
}
```

like phone calls

thrown when another thread calls interrupt

CMSC 330

40

Thread Example

```
public class AlarmRunnable implements Runnable {  
    private String msg = null;  
    private int timeout = 0;  
  
    public AlarmRunnable(String msg, int time) {  
        this.msg = msg;  
        this.timeout = time;  
    }  
  
    public void run() {  
        try {  
            Thread.sleep(timeout * 1000);  
        } catch (InterruptedException e) { }  
        System.out.println(""+timeout+" "+msg);  
    }  
}
```

CMSC 330

41

Thread Example (cont.)

```
while (true) {  
    System.out.print("Alarm> ");  
  
    // read user input  
    String line = b.readLine();  
    parseInput(line);  
    if (m != null) {  
        // start alarm thread  
        Thread t = new Thread(  
            new AlarmRunnable(m, tm) );  
        t.start();  
    }  
}
```

CMSC 330

42

Notes – Passing Parameters

- ▶ `run()` doesn't take parameters
- ▶ We "pass parameters" to the new thread by storing them as private fields
 - In the `Runnable` object
 - Example
 - > Time to wait & message to print in the `AlarmRunnable` class

CMSC 330

43

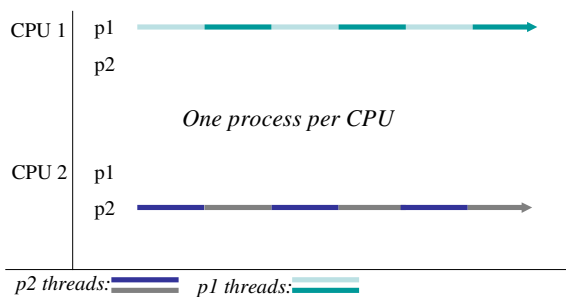
Concurrency

- ▶ 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

CMSC 330

44

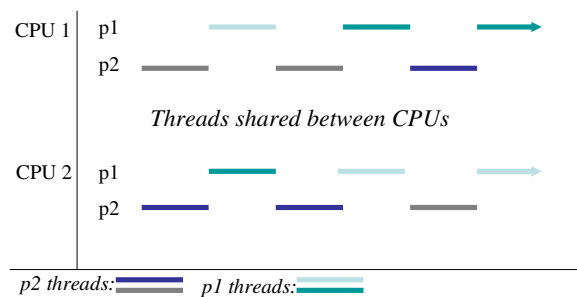
Scheduling Example (1)



CMSC 330

45

Scheduling Example (2)



CMSC 330

46

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

CMSC 330

47

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

CMSC 330

48

Data Race Example

```
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();
    }
}
```

CMSC 330

49

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

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

CMSC 330

50

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.

CMSC 330

51

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

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

CMSC 330

52

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
y = 1

T1 finishes. T2 executes, grabbing the global counter value into its own y.

CMSC 330

53

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
y = 0
y = 1

T2 executes, storing its incremented cnt value into the global counter.

CMSC 330

54

Data Race Example – 2nd Try

```
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.

CMSC 330

55

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.

CMSC 330

56

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

y = 0

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

CMSC 330

57

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

y = 0

T2 executes, storing the incremented cnt value.

CMSC 330

58

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

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)!

CMSC 330

59

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

CMSC 330

60

Question

- ▶ If instead of

```
int y = cnt;
cnt = y+1;
```
- ▶ We had written

```
• cnt++;
```
- ▶ Would the result be any different?
- ▶ Answer: NO!
 - Don't depend on your intuition about atomicity

CMSC 330

61

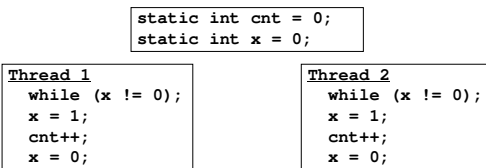
Question

- ▶ If you run a program with a race condition, will you always get an unexpected result?
 - No! It depends on the scheduler, i.e., which JVM you're running, and on the other threads/processes/etc, that are running on the same CPU
- ▶ Race conditions are hard to find

CMSC 330

62

What's Wrong with the Following?



- ▶ 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

CMSC 330

63

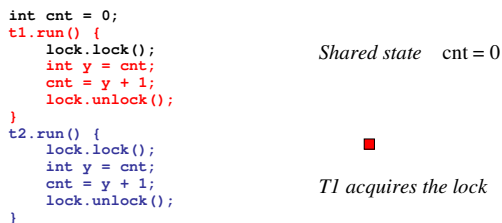
Synchronization

- ▶ 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

CMSC 330

64

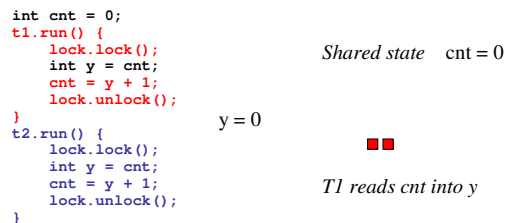
Applying Synchronization



CMSC 330

65

Applying Synchronization



CMSC 330

66

Applying Synchronization

```
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 = 0

y = 0



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

CMSC 330

67

Applying Synchronization

```
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 = 1

y = 0



*T1 runs, assigning
to cnt*

CMSC 330

68

Applying Synchronization

```
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 = 1

y = 0



*T1 releases the lock
and terminates*

CMSC 330

69

Applying Synchronization

```
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 = 1

y = 0



*T2 now can acquire
the lock.*

CMSC 330

70

Applying Synchronization

```
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 = 1

y = 0



T2 reads cnt into y.

y = 1

CMSC 330

71

Applying Synchronization

```
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



*T2 assigns cnt,
then releases the lock*

y = 1

CMSC 330

72

Different Locks Don't Interact

```
static int cnt = 0;
static Lock l =
    new ReentrantLock();
static Lock m =
    new ReentrantLock();

void inc() {
    l.lock();
    cnt++;
    l.unlock();
}
```

```
void inc() {
    m.lock();
    cnt++;
    m.unlock();
}
```

- ▶ This program has a race condition
 - Threads only block if they try to acquire a lock held by another thread

CMSC 330

73

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

CMSC 330

74

Example

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

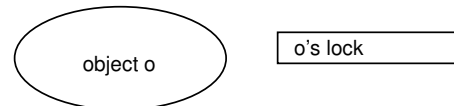
void f() throws Exception {
    synchronized (o) {
        FileInputStream f =
            new FileInputStream("file.txt");
        // Do something with f
        f.close();
    }
}
```

- Lock associated with `o` acquired before body executed
 - Released even if exception thrown

CMSC 330

75

Discussion



- ▶ An object and its associated lock are different!
 - Holding the lock on an object does not affect what you can do with that object in any way
 - Ex:


```
synchronized(o) { ... } // acquires lock named o
o.f(); // someone else can call o's methods
o.x = 3; // someone else can read and write o's fields
```

CMSC 330

76

Example: Synchronizing on this

```
class C {
    int cnt;

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

```
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

CMSC 330

77

Example: Synchronizing on this (cont.)

```
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

CMSC 330

78

Example: Synchronizing on this (cont.)

```
class C {
    int cnt;

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

```
C c1 = new C();
C c2 = new C();
```

```
Thread 1
c1.inc();
```

```
Thread 2
c2.inc();
```

- Does this program have a data race?
 - No, threads acquire different locks, but they write to different objects, so that's ok

CMSC 330

79

Synchronized Methods

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

```
class C {
    int cnt;

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

```
class C {
    int cnt;

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

CMSC 330

80

Synchronized Methods (cont.)

```
class C {
    int cnt;

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

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

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
c.dec();
```

- Data race?
 - No, both acquire same lock

CMSC 330

81

Synchronized Static Methods

- Warning: Static methods lock class object
 - There's no this object to lock

```
class C {
    static int cnt;

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

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

```
C c = new C();
```

```
Thread 1
c.inc();
```

```
Thread 2
C.dec();
```

CMSC 330

82

What Can Be Synchronized?

- code blocks
- methods
 - subclasses do not inherit synchronized keyword
 - interface methods cannot be declared synchronized
- NOT fields
 - but you could write synchronized accessor methods
- NOT constructors
 - but you could include synchronized code blocks
- objects in an array

CMSC 330

83

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?

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

Thread 1          Thread 2
l.lock();         m.lock();
m.lock();         l.lock();
...              ...
m.unlock();       l.unlock();
l.unlock();       m.unlock();
```

CMSC 330

84

Deadlock (cont.)

- ▶ 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

CMSC 330

85

Threads – Thread States

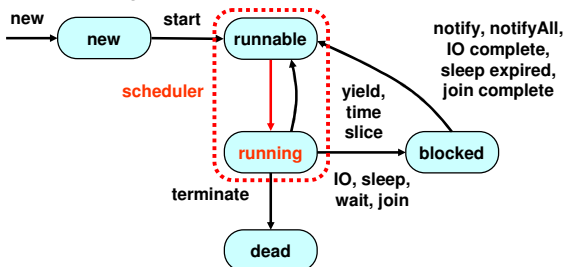
- ▶ Java thread can be in one of these states
 - New – thread allocated & waiting for start()
 - Runnable – thread can begin execution
 - ▶ Running – thread currently executing
 - Blocked – thread waiting for event (I/O, etc.)
 - Dead – thread finished
- ▶ Transitions between states caused by
 - Invoking methods in class Thread
 - ▶ new(), start(), yield(), sleep(), wait(), notify()...
 - Other (external) events
 - ▶ Scheduler, I/O, returning from run()...

CMSC 330

86

Threads – Thread States

- ▶ State diagram



Running is a logical state → thread only knows it's in runnable state

CMSC 330

87

Daemon Threads

- ▶ Definition: Threads which run unattended and perform periodic functions, generally associated with system maintenance.
- ▶ void setDaemon(boolean on)
 - Marks thread as a daemon thread
 - Must be set before thread started
- ▶ By default, thread acquires status of thread that spawned it
- ▶ Program execution terminates when no threads running except daemons

CMSC 330

88

Tradeoffs

- ▶ Threads can increase performance
 - Parallelism on multiprocessors
 - Multitasking can overlap computation and I/O
- ▶ Natural fit for some programming patterns
 - Event / transaction processing
 - Simulations of large systems
- ▶ Disadvantages
 - Increased complexity
 - ▶ Correctness (data races), liveness (deadlock), composition
 - Higher resource usage

CMSC 330

89

Another Case of Deadlock

```

static Lock l = new ReentrantLock();
void f () throws Exception {
    l.lock();
    FileInputStream f = new FileInputStream("f.txt");
    // Do something with f
    f.close();
    l.unlock();
}
  
```

- ▶ Lock l not released if exception thrown
 - Likely to cause deadlock some time later

CMSC 330

90

Solution – Use Finally

```
static Lock l = new ReentrantLock();
void f () throws Exception {
    l.lock();
    try {
        FileInputStream f = new FileInputStream("f.txt");
        // Do something with f
        f.close();
    }
    finally { // Executed regardless of try exit
        l.unlock();
    }
}
```

- ▶ Or just use **synchronized** block
 - `synchronized (o) { ... }`
 - Releases lock for `o` automatically upon exit
 - Even if exception is thrown in block

CMSC 330

91

Deadlock & Starvation

- ▶ Deadlock occurs when no thread can run
 - Because all threads are waiting to acquire locks
 - Also known as **circular wait**
 - Occurs intermittently depending on thread scheduler
 - Thus very difficult to find & debug
 - Detecting potential deadlock is undecidable in general
- ▶ Starvation occurs when some thread cannot run
 - Thread is waiting but unable to acquire lock
 - Because other thread(s) never release lock

CMSC 330

92

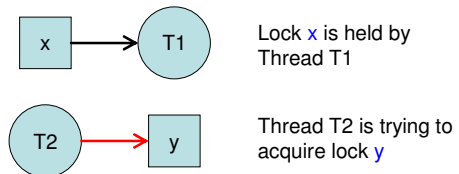
Wait Graph

- ▶ Approach for detecting deadlock at run time
 - After deadlock has occurred
 - Used in operating systems, databases
- ▶ Tracks
 - Threads waiting for locks
 - Locks held by threads

CMSC 330

93

Wait Graph

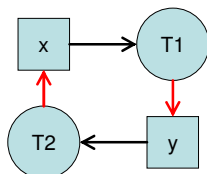


Deadlock occurs when there is a **cycle** in the wait graph

CMSC 330

94

Wait Graph Example



Lock **x** is held by T1
Lock **y** is held by T2
T1 is trying to acquire lock **y**
T2 is trying to acquire lock **x**

CMSC 330

95