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

![Multiprocessors Image](image_url)

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)

Implementation View

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

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

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 – 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

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 */
}
```

- 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

ReentrantLock Class (Java 1.5)

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

- Reentrant lock
  - Can be reacquired by same thread by invoking lock() (Up to 2^47483648 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

Lock Synchronization Example

```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 not held by another thread
- Releases the lock

Reentrant Lock Example

```java
static int count = 0;
static Lock l = new ReentrantLock();
void inc() {
    l.lock();
    count++;
    l.unlock();
}
```

- Example
  - 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

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
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();
    value = o;
    valueReady = true;
    lock.unlock();
}

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

Broken Producer / Consumer Code

- 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) {
    Object consume() {
        lock.lock();
        while (!valueReady); // Bad: valueReady not checked
        o = value;
        lock.lock(); // Bad: lock held on consumer
        while (valueReady);
        lock.unlock();
    }
}
```

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) {
    bufferReady = true;
    lock.unlock();
    ready.signalAll();
    buffer = o;
    while (bufferReady) {
        ready.await();
    }
}

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

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

Broken Producer / Consumer Code

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

void produce(Object o) {
    lock.lock();
    if (!valueReady)
        return o;
    ready.await();
    value = o;
    valueReady = true;
    lock.unlock();
    notifyAll();
}

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

Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel or 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
- Ruby Threads – Thread Creation

```ruby
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 output seen
}
```
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

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

```
myLock.synchronize {
  myLock = Monitor.new
  myCondition.wait_until { x != 0 }
}
```

```
require 'monitor.rb'
# myLock held during this code block
myCondition.wait_until { x != 0 }
myCondition.broadcast
```

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.

Thread Creation

```
execution (time)
```

```
main thread
thread starts
```

```
thread starts
thread joins
```

```
thread ends
```

CMSC 330: Organization of Programming Languages

Review of Java Threads
(Topics Covered in CMSC 132)
Creating Threads in Java

- **Runnable Approach**
  1. Define class implementing Runnable interface
  ```java
  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

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

```java
public class AlarmRunnable implements Runnable {
    public void run() {
        // start alarm
        try {
            Thread.sleep(timeout * 1000);
        } catch (InterruptedException e) { }
        System.out.println("timeout") + msg;
        parseInput(line); // sets timeout
    }

    public AlarmRunnable(String msg, int time) {
        this.msg = msg;
        this.timeout = time;
    }
}
```

Thread Example

```java
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;
        parseInput(line); // sets timeout
    }
}
```

Creating Threads in Java (cont.)

- **Example**
  ```java
  public class MyT implements Runnable {
      public void run() {
          ... // work for thread
      }
  }
  ```

  ```java
  Thread t = new Thread(new MyT()); // create thread
  t.start(); // begin running thread
  ... // thread executing in parallel
  ```

Example Code – Synchronous Alarms

```java
Example Code – Synchronous Alarms

while (true) {
    System.out.print("Alarm> ");

    try {
        Thread.sleep(timeout * 1000);
    } catch (InterruptedException e) { }
    System.out.println("timeout") + msg;
    parseInput(line); // sets timeout
}
```

Thread Example (cont.)

```java
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();
    }
}
```
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

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

Scheduling Example (1)

CPU 1
- p1
- p2

CPU 2
- p1
- p2

One process per CPU

p2 threads: p1 threads:

Scheduling Example (2)

CPU 1
- p1
- p2

CPU 2
- p1
- p2

Threads shared between CPUs

p2 threads: 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

- 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();
    }
}

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

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.

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

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

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

```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.

**Data Race Example**

```java
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**

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

T1 is preempted. T2 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;
}
t2.run() {
    int y = cnt;
    cnt = y + 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!
  ```java
  • Don't depend on your intuition about atomicity
  ```

What's Wrong with the Following?

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

Thread 1
while (x != 0);
  x = 1;
  cnt++;
  x = 0;

Thread 2
while (x != 0);
  x = 1;
  cnt++;
  x = 0;
```

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

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();
}
```

- `T1` acquires the lock

Synchronization

- **Definition**
  ```java
  • Coordination of events with respect to time
  ```
- **Properties**
  ```java
  • Can eliminate data races in multithreaded programs
  • Overhead → excessive use reduces performance
  ```
- **Mechanisms**
  ```java
  • Different in each programming language
  • Look at examples in Java
  ```

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();
}
```

- `T1` reads `cnt` into `y`

- `Shared state cnt = 0`
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 = 0
```

**T1 runs, assigning to cnt**

```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 = 1
```

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

```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 = 1
```

**T1 releases the lock and terminates**

```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
```

**T2 assigns cnt, then releases the lock**
Different Locks Don’t Interact

```java
static int cnt = 0;
static Lock l = new ReentrantLock();
static Lock m = new ReentrantLock();
void inc() {
  l.lock();
  cnt++;
  m.unlock();
}
void inc() {
  i.lock();
  cnt++;
  i.unlock();
}
```

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

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
    - Release lock when scope is exited
      - Even in cases of exception or method return

Example

```java
static Object o = new Object();
synchronized (o) {
  f.close(); // Do something with f
  Filef = new FileInputStream("file.txt");
}
```

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:
    ```java
    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
    ```

Example: Synchronizing on this

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

- 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--; // someone else can read and write cnt
    }
  }
}
c = new C();
```

- Data race?
  - No, 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++;
        }
    }
    synchronized void dec() {
        cnt--;
    }
}
```

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

Synchronized Methods (cont.)

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

- Data race?
  - No, both acquire same lock

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++;
        }
    }
}

C c1 = new C();
c2.inc();
Thread 1
C c2 = new C();
c1.inc();
Thread 2
```

Synchronized Static Methods

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

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

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

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

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

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

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

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

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()...

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

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

Another Case of Deadlock

```java
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
**Solution – Use Finally**

```java
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

**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

**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

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