

## Synchronization Topics

- Locks
- **synchronized** statements and methods
- **wait** and **notify**
- Deadlock

## Locks

- Any Object subclass has (can act as) a lock
- Only one thread can hold the lock on an object
  - other threads block until they can acquire it
- If your thread already holds the lock on an object
  - can lock many times
  - Lock is released when object unlocked the corresponding number of times
- No way to only attempt to acquire a lock
  - Either succeeds, or blocks the thread

## Synchronized statement

- **synchronized (obj) { statements }**
- Obtains the lock on **obj** before executing statements in block
- Releases the lock when the statements block completes

CMSC 433, Fall 2002 - Michael Hicks

50

## Recasting earlier example

```
public class State {
    public int cnt = 0;
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        int y = s.cnt; ← Unsynchronized access to
        s.cnt = y + 1;   shared data!
    }
}
public void main(String args[]) {
    State s = new State();
    MyThread thread1 = new MyThread(s);
    MyThread thread2 = new MyThread(s);
    thread1.start(); thread2.start();
}
}
```

CMSC 433, Fall 2002 - Michael Hicks

51

## Adding Synchronization

```
public class State {
    public int cnt = 0;
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        synchronized (s) { Uses s as a lock, forces
            int y = s.cnt; exclusive access
            s.cnt = y + 1;
        }
    }
    public void main(String args[]) {
        State s = new State();
        MyThread thread1 = new MyThread(s);
        MyThread thread2 = new MyThread(s);
        thread1.start(); thread2.start();
    }
}
```

CMSC 433, Fall 2002 - Michael Hicks

52

## Synchronized methods

- A method can be synchronized
  - add **synchronized** modifier before return type
- Obtains the lock on object referenced by **this**, before executing method
  - releases lock when method completes
- For a **static synchronized** method
  - locks the class object

CMSC 433, Fall 2002 - Michael Hicks

53

## Synchronization example

```
public class State {
    private int cnt = 0;
    public int synchronized incCnt(int x) {
        cnt += x;
    }
    public int synchronized getCnt() { return cnt; }
}
public class MyThread extends Thread {
    State s;
    public MyThread(State s) { this.s = s; }
    public void run() {
        s.incCnt(1)
    }
}
public void main(String args[]) {
    State s = new State();
    MyThread thread1 = new MyThread(s);
    MyThread thread2 = new MyThread(s);
    thread1.start(); thread2.start();
}
}
```

*Synchronization occurs in State object itself, rather than in its caller.*

CMSC 433, Fall 2002 - Michael Hicks

54

## Synchronization Style

- Design decision
  - Internal synchronization (class is thread-safe)
    - Have a stateful object synchronize itself (e.g. with synchronized methods)
  - External synchronization (class is thread-compatible)
    - Have callers perform synchronization before calling the object
- Can go both ways:
  - Thread-safe: Random
  - Thread-compatible: ArrayList, HashMap, ...

CMSC 433, Fall 2002 - Michael Hicks

55

## Condition Variables

- Want access to shared data, but only when some condition holds
  - Implies that threads play different *roles* in accessing shared data
- Examples
  - Want to read shared variable *v*, but only when it is non-null
  - Want to insert myself in a datastructure, but only if it is not full

CMSC 433, Fall 2002 - Michael Hicks

56

## CVs: Use Wait and Notify

*To wait for a condition to become true:*

```
synchronized (obj) {  
    while (condition does not hold)  
        obj.wait();  
    ... perform appropriate actions  
}
```

*To notify waiters that a condition has changed:*

```
synchronized (obj) {  
    ... perform actions that change condition  
    obj.notify();  
    or obj.notifyAll();  
}
```

CMSC 433, Fall 2002 - Michael Hicks

57

## Producer/Consumer Example

```
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;
        value = null;
        notifyAll();
        return o;
    }
}
```

CMSC 433, Fall 2002 - Michael Hicks

58

## Wait and Notify

- Must be called inside **synchronized** method or block of statements
- **a.wait()**
  - releases the lock on **a**
    - But not any other locks acquired by this thread
  - adds the thread to the *wait set* for **a**
  - blocks the thread
- **a.wait(int m)**
  - limits wait time to **m** milliseconds (but see below)

CMSC 433, Fall 2002 - Michael Hicks

59

## Wait and Notify (cont.)

- **a.notify()** resumes *one* thread from **a**'s wait set
  - no control over which thread
- **a.notifyAll()** resumes *all* threads on **a**'s wait set
- resumed thread(s) must reacquire lock before continuing

CMSC 433, Fall 2002 - Michael Hicks

60

## Attempting to Refine Sync

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

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

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

*No guarantee about  
which thread wakes  
up*

CMSC 433, Fall 2002 - Michael Hicks

61

## A Better Solution?

```
public class ProducerConsumer {
    private boolean valueReady = false;
    private Object value;
    private Object fullLock=new Object(), emptyLock=new Object();

    synchronized void produce(Object o) {
        while (valueReady) {
            synchronized (emptyLock) {
                emptyLock.wait();
            }
        }
        value = o; valueReady = true;
        synchronized (fullLock) { fullLock.notify();}
    }
}
```

CMSC 433, Fall 2002 - Michael Hicks

62

## A Better Solution?

```
synchronized Object consume() {
    while (!valueReady) {
        synchronized (fullLock) {
            fullLock.wait();
        }
    }
    valueReady = false;
    Object o = value;
    value = null;
    synchronized (emptyLock) { emptyLock.notify();}
    return o;
}
}
```

*Can you see the problem?*

CMSC 433, Fall 2002 - Michael Hicks

63

This document was created with Win2PDF available at <http://www.daneprairie.com>.  
The unregistered version of Win2PDF is for evaluation or non-commercial use only.