### Synchronization issues

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

### Locks

- All objects can be locked
- Only one thread can hold a lock on an object
  - other threads block until they can acquire it
- If your thread already holds a lock on an object
  - can lock it a second time
  - object not unlocked until both locks released
- No way to only attempt to acquire a lock

### Synchronized methods

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

### Synchronized statement

- **synchronized** *(obj)* `{ statements }*
- Obtains a lock on **obj** before executing statements in block
- Releases lock once block completes
- Provides finer grained control than synchronized method
- Allows locking arguments to a method

### Synchronization example

```java
class SyncTest extends Thread {
    String msg;
    public SyncTest(String s) {
        msg = s;
        start();
    }
    public void run() {
        synchronized (SyncTest.class) {
            System.out.print("[" + msg + "]");
            try { Thread.sleep(1000); }
            catch (InterruptedException e) {};
            System.out.println("]");
        }
    }
    public static void main(String[] args) {
        new SyncTest("Hello");
        new SyncTest("Synchronized");
        new SyncTest("World");
    }
}
```

### Wait and Notify

- Must be called inside **synchronized** method or block of statements (on the synchronized object)
  - **a.wait()**
    - gives up lock(s) on **a**
    - adds thread to wait set for **a**
    - suspends thread
  - **a.wait(int m)**
    - limits suspension to **m** milliseconds
Wait and Notify (cont.)

- **a.notify()** resumes one thread from a’s wait list
  - and removes it from wait set
  - no control over which thread
- **a.notifyAll()** resumes all threads on a’s wait list
  - resumed threads must reacquire lock before continuing
- **wait** doesn’t give up locks on any other objects
  - e.g., acquired by methods that called this one

Producer/Consumer Example

// Won’t work. Needs to handle
// InterruptedException

```java
public class ProducerConsumer {
  private boolean ready = false;
  private Object obj;

  public ProducerConsumer() {
  }

  public ProducerConsumer(Object o) {
    obj = o;
    ready = true;
  }

  synchronized void produce(Object o) {
    while (ready) wait();
    obj = o;
    ready = true;
    notifyAll();
  }

  synchronized Object consume() {
    while (!ready) wait();
    ready = false;
    notifyAll();
    return obj;
  }
}
```

Changed example

```java
synchronized void produce(Object o) {
  synchronized (empty) {
    try {
      return obj;
    } catch (InterruptedException e) {
    }
  }
  obj = o;
  full = true;
  synchronized (full) {
    full.notify();
  }
}
```

A Better Solution

```java
void produce(Object o) {
  while (bufFull) {
    synchronized (empty) {
      try {
        empty.wait();
      } catch (InterruptedException e) {
      }
    }
    obj = o;
    bufFull = true;
    synchronized (full) {
      full.notify();
    }
  }
}
```

notify() vs. notifyAll()

- Very tricky to use notify() correctly
  - notifyAll() much safer
- Need:
  - all waiters are equal
  - each notify only needs to wake up 1 thread
  - handle InterruptedException correctly

InterruptedException Example

- Threads t1 and t2 are waiting
- Thread t3 performs a notify
  - thread t1 is selected
- Before t1 can acquire lock, t1 is interrupted
- t1’s call to wait throws InterruptedException
  - t1 doesn’t process notification
  - t2 doesn’t wake up
Handling InterruptedException

```java
synchronized (this) {
    while (!ready) {
        try { wait(); }
        catch (InterruptedException e) {
            notify();
            throw e;
        } // do whatever
    }
}
```

Deadlock

- Quite possible to create code that deadlocks
  - Thread 1 holds lock on A
  - Thread 2 holds lock on B
  - Thread 1 is trying to acquire a lock on B
  - Thread 2 is trying to acquire a lock on A
  - Deadlock!
- Not easy to detect when deadlock has occurred
  - other than by the fact that nothing is happening

A common multi-threading bug

- Threads might cache values
- Obtaining a lock forces the thread to get fresh values
- Releasing a lock forces the thread to flush out all pending writes
- `volatile` variables are never cached
- `sleep(...)` doesn’t force fresh values
- Many compilers don’t perform these optimizations
  - but some do (Hotspot?)
- Problem might also occur with multiple CPUs

Guidelines to simple/safe multi-threaded programming

- Synchronize access to shared data
- Don’t hold a lock on more than one object at a time
  - could cause deadlock
- Hold a lock for as little time as possible
  - reduces blocking waiting for locks
- While holding a lock, don’t call a method you don’t understand
  - e.g., a method provided by someone else, especially if you can’t be sure what it locks

Guidelines (cont.)

- Have to go beyond these guidelines for more complex situations
  - but need to understand threading and synchronization well
- Recommended book for more details:
  - *Concurrent Programming in Java*, by Doug Lea