Threads and Synchronization
Nov 9, 2004
(thanks to Doug Lea for some slides)

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;
        }
    }
}

int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

Applying Synchronization

Avoiding Interference: Synchronization

int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}

Applying Synchronization

Lock, for protecting the shared state
Acquires the lock; Only succeeds if not held by another thread
Releases the lock

Shared state cnt = 0
T1 acquires the lock
T1 reads cnt into y
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 0

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

T1 releases the lock and terminates

T2 now can acquire the lock.

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;  y = 0
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 1
Applying Synchronization

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state: \(cnt = 1\)

T2 reads \(cnt\) into \(y\).

```java
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state: \(cnt = 2\)

T2 assigns \(cnt\), then releases the lock.

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 a thread already holds the lock on an object
  - The thread can reacquire the same lock many times
    - Locks are reentrant
  - Lock is released when object unlocked the corresponding number of times
- No way to only attempt to acquire a lock
  - ...in Java 1.4
  - 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 statement block completes
  - Either normally, or due to a return, break, or exception being thrown in the block
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 for the class
    - Accessible directly, e.g. Foo.class
  - Not the same as this!

Synchronization Example

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

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, …

Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called deadlock
```
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            synchronized (A) {
                ...
            }
        }
    }
}
T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```
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

Deadlock: Wait graphs

A
T1
Thread T1 holds lock A

T2
B
Thread T2 attempting to acquire lock B

Deadlock occurs when there is a cycle in the graph

Wait graph example

A
T1
T2
B
T1 holds lock on A
T2 holds lock on B
T1 is trying to acquire a lock on B
T2 is trying to acquire a lock on A

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 interference
    • Synchronization is one way, but not the only way
  – Overuse use of synchronization can create deadlock
    • Violation of liveness
Guaranteeing Safety

- Ensure objects are accessible only when in a **consistent** and appropriate state
  - All invariants are maintained
  - Presents subclass obligations
- Use locks to enforce this
  - 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

Guaranteeing Liveness

- Ensuring availability of services
  - Called methods eventually execute
- Ensuring progress of activities
  - Managing resource contention
  - Freedom from deadlock
  - Fairness
  - Fault tolerance

Producer/Consumer Design

- Suppose we are communicating with a shared variable
  - E.g., some kind of a buffer holding messages
- One thread *produces* input to the buffer
- One thread *consumes* data from the buffer
- How do we implement this?
  - Use wait and notify

Producer/Consumer Example

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

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

    synchronized Object consume() throws InterruptedException {
        while (!valueReady) wait();
        valueReady = false;
        Object o = value;
        value = null; // why do we do this?
        notifyAll();
        return o;
    }
}
```
Wait and Notify

- Both must be called while lock is held on `a`
- `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

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
  - Java performs the reacquire automatically