CMSC 330: Organization of Programming Languages

Threads
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

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

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

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

This code can deadlock...  
-- when will it work?  
-- when will it deadlock?
Deadlock (cont’d)

• 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
Wait Graphs

Thread T1 holds lock \( l \)

Thread T2 attempting to acquire lock \( m \)

Deadlock occurs when there is a cycle in the graph
Wait Graph Example

T1 holds lock on l
T2 holds lock on m
T1 is trying to acquire a lock on m
T2 is trying to acquire a lock on l
Another Case of Deadlock

static Lock l = new ReentrantLock();

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

• l not released if exception thrown
  – Likely to cause deadlock some time later
Solution: Use Finally

```java
class Solution {
  static Lock l = new ReentrantLock();

  void f() throws Exception {
    l.lock();
    try {
      FileInputStream f =
        new FileInputStream("file.txt");
      // Do something with f
      f.close();
    } finally {
      // This code executed no matter how we
      // exit the try block
      l.unlock();
    }
  }
}
```
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
Example

```java
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
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 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
Example: Synchronizing on this (cont’d)

```java
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.
Example: Synchronizing on this (cont’d)

```java
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
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++;
        }
    }
}
```

```java
class C {
    int cnt;

    synchronized void inc() {
        cnt++;
    }
}
```
Synchronized Methods (cont’d)

```java
class C {
    int cnt;

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

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

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

• Data race?
  – No, both acquire same lock
Synchronized Static Methods

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

```java
class C {
    static int cnt;

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

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

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

• When multiple threads share a CPU...
  – When should the current thread stop running?
  – What thread should run next?
• A thread can voluntarily `yield()` the CPU
  – Call to yield may be ignored; don’t depend on it
• *Preemptive schedulers* can de-schedule the current thread at any time
  – Not all JVMs use preemptive scheduling, so a thread stuck in a loop may *never* yield by itself. Therefore, put `yield()` into loops
• Threads are de-scheduled whenever they block (e.g., on a lock or on I/O) or go to sleep
Thread Lifecycle

• While a thread executes, it goes through a number of different phases
  – **New**: created but not yet started
  – **Runnable**: can run on a free CPU
  – **Running**: currently executing on a CPU
  – **Blocked**: waiting for I/O or on a lock
  – **Sleeping**: paused for a user-specified interval
  – **Terminated**: completed
Which Thread to Run Next?

• Look at all runnable threads
  – A good choice to run is one that just became unblocked because
    • A lock was released
    • I/O became available
    • It finished sleeping, etc.

• Pick a thread and start running it
  – Can try to influence this with `setPriority(int)`
  – Higher-priority threads get preference
  – But you probably don’t need to do this
Some Thread Methods

- **void interrupt()**
  - Interrupts the thread
- **void join() throws InterruptedException**
  - Waits for a thread to die/finish
- **static void yield()**
  - Current thread gives up the CPU
- **static void sleep(long milliseconds) throws InterruptedException**
  - Current thread sleeps for the given time
- **static Thread currentThread()**
  - Get Thread object for currently executing thread
Example: Threaded, Sync Alarm

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

    // read user input
    String line = b.readLine();
    parseInput(line);

    // wait (in secs) asynchronously
    if (m != null) {
        // start alarm thread
        Thread t = new AlarmThread(m, tm);
        t.start();
        // wait for the thread to complete
        t.join();
    }
}
```
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
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
    • 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