CMSC 330: Organization of Programming Languages

Threads

Synchronization

- Refers to mechanisms allowing a programmer to control the execution order of some operations across different threads in a concurrent program.
- Different languages have adopted different mechanisms to allow the programmer to synchronize threads.
- Java has several mechanisms; we'll look at locks first.

Locks (Java 1.5)

```
interface Lock {
  void lock();
  void unlock();
  ... /* Some more stuff, also */
}
class ReentrantLock implements Lock { ... }
```

- Only one thread can hold a lock at once
  - Other threads that try to acquire it *block* (or become suspended) until the lock becomes available
- *Reentrant lock* can be reacquired by same thread
  - As many times as desired
  - No other thread may acquire a lock until has been released same number of times it has been acquired

Avoiding Interference: Synchronization

```
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

Applying Synchronization

```
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 acquires the lock

Applying Synchronization

```
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

y = 0

T1 reads cnt into y

Applying Synchronization

```
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();
}
```

y = 0

Shared state cnt = 0

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

Applying Synchronization

```
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();
}
```

y = 0

Shared state cnt = 1

T1 runs, assigning to cnt

Applying Synchronization

```
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();
}
```

y = 0

Shared state cnt = 1

T1 releases the lock and terminates

Applying Synchronization

```
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();
}
```

y = 0

Shared state cnt = 1

T2 now can acquire the lock

Applying Synchronization

```
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();
}
```

y = 0

Shared state cnt = 1

T2 reads cnt into y.

y = 1

Applying Synchronization

```
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();
}
```

y = 0

Shared state cnt = 2

T2 assigns cnt, then releases the lock

y = 1

Different Locks Don't Interact

```
static int cnt = 0;
static Lock l =
    new ReentrantLock();
static Lock m =
    new ReentrantLock();

void inc() {
  l.lock();
  cnt++;
  l.unlock();
}
```

```
void inc() {
  m.lock();
  cnt++;
  m.unlock();
}
```

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

Reentrant Lock Example

```
static int cnt = 0;
static Lock l =
    new ReentrantLock();

void inc() {
  l.lock();
  cnt++;
  l.unlock();
}
```

```
void returnAndInc() {
  int temp;

  l.lock();
  temp = cnt;
  inc();
  l.unlock();
}
```

- Reentrancy is useful because each method can acquire/release locks as necessary
  - No need to worry about whether callers have locks
  - Discourages complicated coding practices

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

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

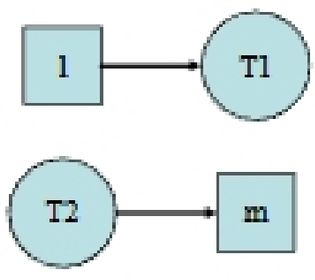
```
Lock l = new ReentrantLock();
Lock m = new ReentrantLock();
```

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

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

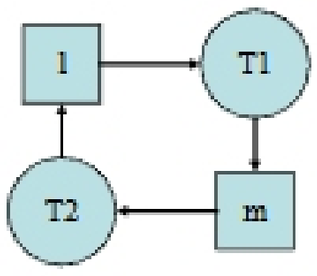


Thread T1 holds lock l

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