ITEC 1630
Week 8: Multithreading

Yves Lespérance
Readings: Horstmann Ch. 23

Multithreading

• When multiple threads/processes run concurrently/in parallel
• Often timesliced, but may run in parallel on a multiprocessor
• Many applications

Multithreading basics

• Package tasks to be run by threads in classes that implement the Runnable interface
• Put code performing task in the run() method
• Create a thread and use it to execute the task

The class Thread

• To create a thread:
  Thread t = new Thread(runnableObject);
  Or Thread t = new Thread();
• To start running it: t.start();
• To make it go to sleep:
  Thread.sleep(milliseconds);
• See Greetings e.g.
Stopping/interrupting a thread

- To interrupt a thread: `t.interrupt();`
- Thread decides how/when to stop
- Can use `Thread.interrupted()` to check if interrupted
- If thread is interrupted while sleeping, `InterruptedException` is thrown; catch it and wind up the task

Race conditions

- When several threads are using shared data structures, they must take turns to ensure the data is not corrupted
- Such corruption may occur if the threads are interleaved in a very particular way
- This is called a *race condition*
- See `BankAccountThreadTester` without lock e.g.

Using Locks

- To ensure mutual exclusion when accessing the data structure, use a lock
- Create a lock: `l = new ReentrantLock();`
- To get the lock (before accessing the data): `l.lock();`
- To release the lock (after accessing the data): `l.unlock();`
- See `BankAccountThreadTester` with lock e.g.

synchronized methods

- Another way to ensure mutually exclusive access to data and do synchronization
- Only one thread can be executing a `synchronized` method at any one time
- Each object has a built-in lock which is acquired when entering a `synchronized` method and released when leaving it
- See modified `BankAccount` with `synchronized` e.g.
deadlock

• Deadlock occurs when a thread acquires a lock and then must wait for another thread to do some work before proceeding, but where the second thread needs the lock to proceed
• E.g. withdraw waits for balance to increase while holding lock
• E.g. t1 has resource1 and needs resource2 to proceed while t2 has resource2 and needs resource1

Waiting and signaling

• Can be used to do advanced synchronization
• A thread waits on a condition (e.g. balance > 0) and another thread signals when the condition becomes true
• To create a condition: Condition c = alock.newCondition();
• To start waiting on a condition: c.await();
• To signal that a condition has become true: c.signalAll() or c.signal()

Waiting and signaling

• Waiting threads are blocked and will not be considered for execution until the condition is signaled
• The lock must still be released before they can run
• See modified BankAccount with wait/signal
• Can also be done with an object's built-in lock and condition: wait() to wait and notifyAll() or notify() to signal