CSCE 314: Programming Languages

Dr. Dylan Shell

Java Concurrency

The World is Concurrent

Concurrent programs:

- more than one activities execute simultaneously (concurrently)
- no interference between activities, unless specially programmed to communicate

A big portion of software we use is concurrent

- OS: IO, user interaction, many processes, . . .
- Web browser, mail client, mail server, . . .
- Think about the Internet!

Why should we care?

- Several application areas necessitate concurrent software
- Concurrency can help in software construction:
  - organize programs into independent parts
- Concurrent programs can run faster on parallel machines
- Concurrent programs promote throughput computing on CMT/CMP machines

Myths and Truths

- **Myth: concurrent programming is difficult**
- **Truth: concurrent programming is very difficult**
- **In particular: state and concurrency mix poorly**
- **Truth #2: Concurrent programming can be easy -- at least depending on the tools and programming languages used**
  - In pure languages (or the pure segments of those) with referentially transparent programs, no difficulty: concurrency can be (largely) ignored while reasoning about program behavior
- **Declarative/pure languages aren't mainstream. Imperative langs. with threads as their main model for concurrency dominate**
- **World is concurrent, many applications have to model it somehow.**

Language Support for Concurrency

How languages provide means to program concurrent programs varies:

C, C++: concurrency features not part of the language, but rather provided in (standard) libraries

- In Java, concurrency features partially part of the language and partially defined in its standard libraries (Java concurrency API)
- In Erlang, Oz, threads, futures, etc. integral part of the language

Next: mechanics of Java's low level concurrency feature - threads

Threads

- Thread is an independently executed unit of a program
- The JVM takes care of scheduling threads, typically each active thread gets a small amount of processing time in its turn, with rapid switching between threads
- In other words: Programmer does not control how much of which thread gets executed when (preemptive scheduling)
- In a system with more than one processing units, threads may execute in parallel

Threads vs. Processes

Process

1. self-contained execution environment
2. own memory space
3. one Java application, one process (not always true)

Thread

1. at least one per process
2. shares resources with other threads in the process, including memory, open files
3. every (Java) program starts with one thread (+ some system threads for GC etc.)
4. concurrency is attained by starting new threads from the main thread (recursively)

Running Threads

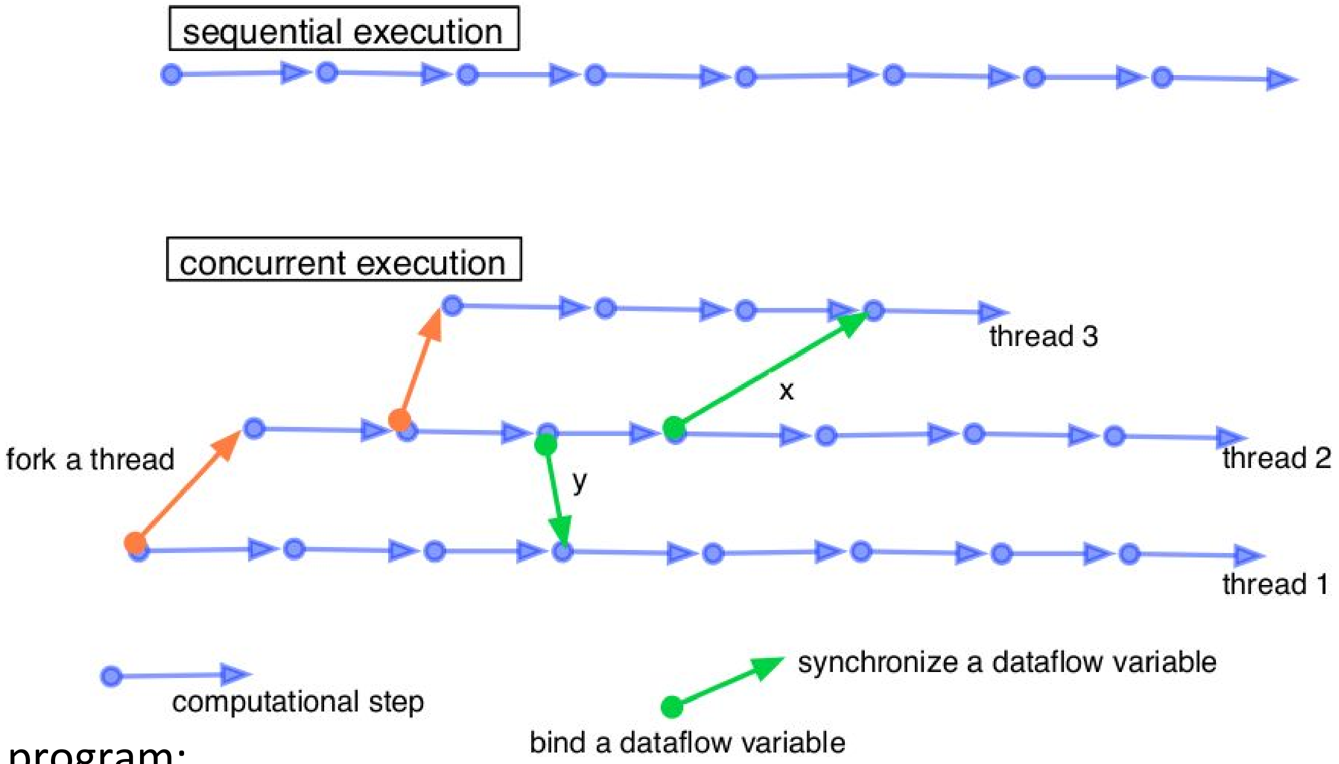
```
public interface Runnable {
   void run();
}
public class MyRunnable implements Runnable {
   public void run() {
      // task here . . .
   }
}
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Examples : GreetingRunnable.java

Key points of the example:

- In presence of side-effects, different interleavings of tasks may produce different results
- A situation where the result of a computation may vary based on the order of the execution of tasks of the computation is called a race condition (or race hazard)
    - A race hazard exists when two threads can potentially modify the same piece of data in an interleaved way that can corrupt data.
- One of the sources of difficulty of concurrent programming
- Absence of side-effects means that race conditions cannot occur (makes “purity” of a language a desirable property)

Causal Order



Concurrent program:

- All execution states of a given thread are totally ordered
- Execution states of the concurrent program as a whole are partially ordered

Extending Thread

Task for a thread can be specified also in a subclass of Thread

```
public class MyThread extends Thread {
   public void run() { . . . // task here
   }
}
Thread t = new MyThread();
t.start();
```

Benefits of using `Runnable` instead:

- It does not identify a task (that can be executed in parallel) with a thread object
- Thread object typically bound with the OS's thread
- `Runnable` is an interface, a class implements Runnable could extend another class
- Many runnables can be executed in a single thread for better efficiency, e.g., with thread pools

Aside: Thread Pools

- Thread pools launch a fixed number of OS threads and keeps them alive
- Runnable objects executed in a thread pool executes in one of those threads (in the first idle one)
- Thread pools commonly used to improve efficiency of various server applications (web servers, database engines, etc.)

```
GreetingRunnable r1 = new GreetingRunnable("Hi!");
GreetingRunnable r2 = new GreetingRunnable("Bye!");
ExecutorService pool = Executors.newFixedThreadPool(MAX_THREADS);
pool.execute(r1);
pool.execute(r2);
```

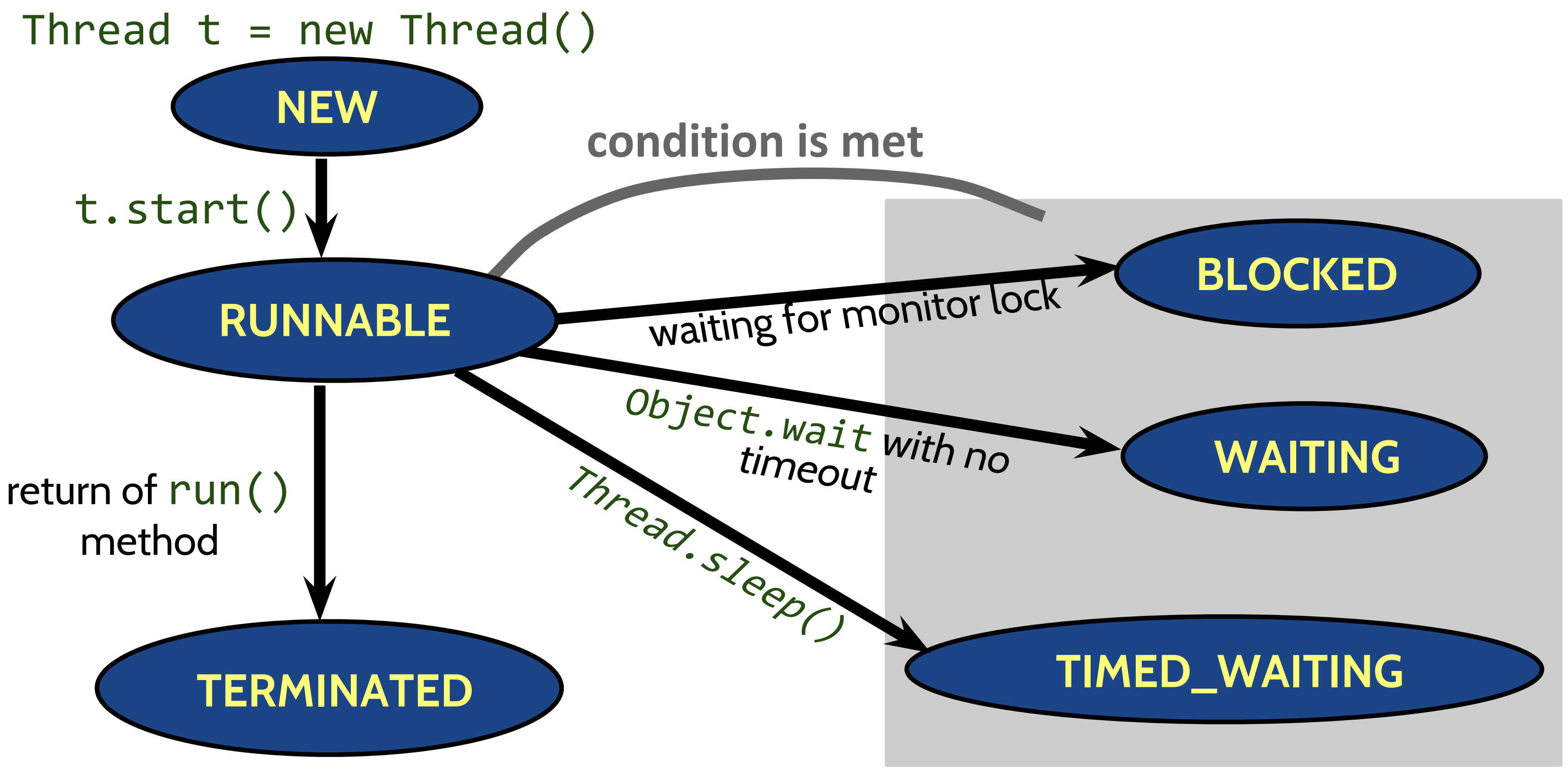
Stopping Threads

- Threads stop when the `run` method returns
- They can also be stopped via *interrupting* them
  - E.g., new HTTP GET request on a web server, while several threads are still processing the previous request from the same client
- Call to the `interrupt()` method of a thread sets the interrupted flag of the thread (Examining the flag with `Thread.interrupted()` clears it)
- Thread itself decides how to (and whether it should) stop - typically stopping is preceded by a clean-up (releasing resources etc.)
- Convention: entire body of run method protected by try-catch
- Note: `Thread.stop()` is deprecated as too dangerous
  - Example: InterruptRunnable.java

Thread States

A thread can be in one of the following states:

- **new**: just created, not yet started
- **runnable**: after invoking `start()`. Not scheduled to run yet
- **running:** executing
- **blocked**: waiting for a resource, sleeping for some set period of time. When condition met, returns back to runnable state
- **dead**: after return of `run` method. Cannot be restarted.



Thread t = new Thread()
NEW
t.start()
RUNNABLE
condition is met
BLOCKED
waiting for monitor lock
Object.wait with no timeout
WAITING
Thread.sleep()
TIMED_WAITING
return of run() method
TERMINATED

Synchronization

Thread Safety

- Some software element is *thread-safe* if it is guaranteed to exhibit correct behavior while executed concurrently by more than one thread
- A definition geared towards OO, and the ideology of design of Java concurrency features:
  - Fields of an object or class always maintain a valid state (class invariant), as observed by other objects and classes, even when used concurrently by multiple threads.
  - Postconditions of methods are always satisfied for valid preconditions.

Race Condition Example

- Object `account` is shared among several threads
- First thread reads account’s balance; second thread preempts, and updates the balance; first thread updates the balance as well, but based on incorrect old value.
- `deposit` and `withdraw` methods’ postconditions are not guaranteed to hold (what are their postconditions?)

```
public void deposit(double amount) {
   balance = balance + amount;
   . . .
}
```

Race Condition Example (Cont.)

- Removing the long sleeps will not help
- Pre-empting occurs at byte/native code level, and does not respect Java’s expression/statement boundaries
- Note: Local variables, function parameters, return values stored in thread’s own stack, only have to worry about instance variables and objects on the heap

Synchronization With Locks

- Lock object guards a shared resource
- Commonly a lock object is an instance variable of a class that needs to modify a shared resource:

```
public class BankAccount {
    public BankAccount() {
        balanceChangeLock = new ReentrantLock();
        . . .
    }
    . . .
    private Lock balanceChangeLock;
}
```

Synchronization With Locks (Cont.)

Code manipulating the shared resource guarded with a lock

```
public class BankAccount {
    public BankAccount() {
        balanceChangeLock = new ReentrantLock();  . . .
    }
    private Lock balanceChangeLock;
}
```

```
balanceChangeLock.lock();
// manipulate balance here
balanceChangeLock.unlock();
```

better →

```
balanceChangeLock.lock();
try {
 // manipulate balance here
}
finally {
balanceChangeLock.unlock();
}
```

Example

```
public void deposit(double amount) {
  balanceChangeLock.lock();
  try {
    System.out.println("Depositing " + amount);
    double nb = balance + amount;
    System.out.println("New balance is " + nb);
    balance = nb;
  } finally {
    balanceChangeLock.unlock();
  }
}
```

Could be improved - critical sections should be as short as possible.

Lock Ownership

- Thread owns the lock after calling `lock()`, if another thread does not own it already
- If lock owned by another thread, scheduler deactivates thread that tries to `lock`, and reactivates periodically to see if lock not owned anymore
- Ownership lasts until `unlock()` called
- “Reentrant” lock means the thread owning a lock can lock again (e.g., calling another method using the same lock to protect its critical section)

Per Method Synchronization

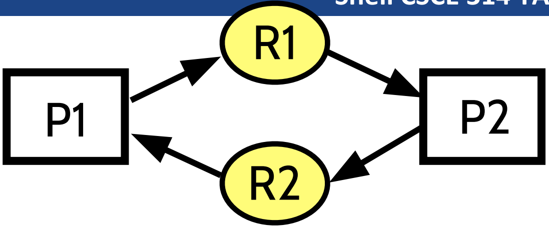
- Java ties locks and synchronization: *object locks* and *synchronized methods*
- The granularity may not always be desirable. Example:

```
public class BankAccount {
   public synchronized void deposit(double amount) {
      System.out.println("Depositing " + amount);
      double nb = balance + amount;
      System.out.println("New balance is " + nb);
      balance = nb;
   }
   public synchronized void withdraw(double amount) { . . . }
}
```

- Synchronized methods automatically wraps the body into

```
lock; try {body} finally { unlock }
```

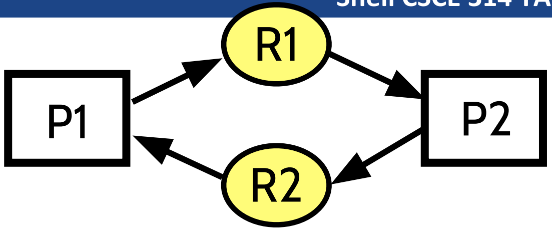
Deadlock



- Our bank account allows overdraft. Attempts to remedy:

```
if (account.getBalance() >= amount) account.withdraw(amount);
```

Deadlock

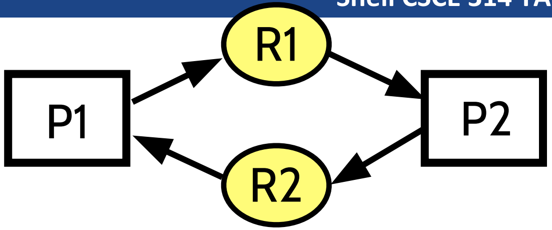


- Our bank account allows overdraft. Attempts to remedy:

```
if (account.getBalance() >= amount) account.withdraw(amount);
```

- Does not work, thread may be preempted between test of balance and withdrawing.
- Next attempt..

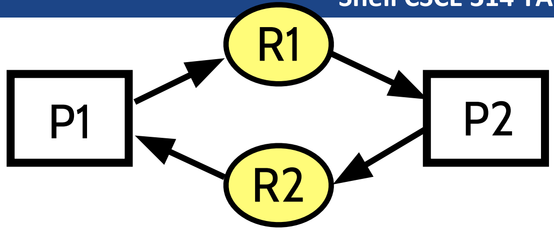
Deadlock



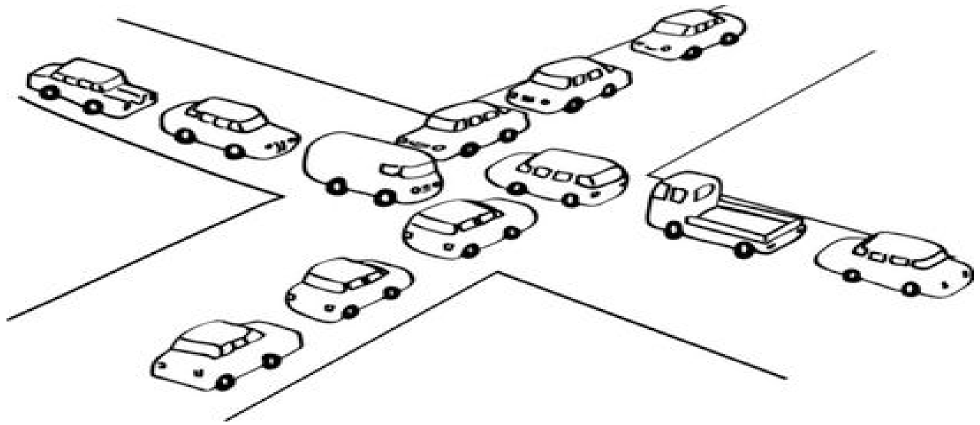
- Next attempt

```
public void withdraw(double amount) {
   balanceChangeLock.lock();
   try {
      while (balance < amount) {} // wait balance to grow
      double nb = balance - amount;
      balance = nb;
   } finally {
      balanceChangeLock.unlock();
   }
}
```

Deadlock



A situation where two or more processes are unable to proceed because each is waiting for one of the others to do something.



“When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.” -- Statute passed by the Kansas State Legislature, early in the 20th century

Condition Objects

Condition object allows a temporary release of a lock

```
public class BankAccount {
  public BankAccount() {
    balance = 0;
    balanceChangeLock = new ReentrantLock();
    sufficientFundsCondition = balanceChangeLock.newCondition();
  }
  public void withdraw(double amount) {
    balanceChangeLock.lock()
    try {
      while (balance < amount) sufficientFundsCondition.await();
       . . .
    } finally { balanceChangeLock.unlock(); }
  }
  private Lock balanceChangeLock;
  private Condition sufficientFundsCondition;
}
```

Condition Objects

Condition object allows a temporary release of a lock

```
public class BankAccount {
  ...
  public void withdraw(double amount) {
    balanceChangeLock.lock()
    try {
      while (balance < amount) sufficientFundsCondition.await();
       . . .
    } finally { balanceChangeLock.unlock(); }
  }
  private Lock balanceChangeLock;
  private Condition sufficientFundsCondition;
}
```

Current thread blocks, other threads can proceed

Current thread unblocked by a call to signalAll(), a notification to all threads blocked with a particular condition object

signalAll

```
public void deposit(double amount) {
  balanceChangeLock.lock();
  try {
    . . .
    sufficientFundsCondition.signalAll();
  } finally { balanceChangeLock.unlock(); }
}
```

- Notification with `signalAll` means: something has changed, it is worthwhile to check if it can proceed
- `signalAll` must be called while owning the lock bound to the condition object

Memory Consistency Errors

- Processors can exercise *out-of-order* execution. Compilers can generate code out-of-order.
- Processors have a variety of levels of caches: different processors have a different view of memory at a given time.
- This can lead to memory consistency errors
- Assume `y == 0, x == true`

```
// thread 1
while (x);
System.out.print(y);
```

```
// thread 2
y = 1;
x = false;
```

It is actually possible that the above prints 0.

Sequential Consistency

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

- Java (or C++) memory model does not guarantee sequential consistency.
- Program order only respected within a thread.
- Special rules by "Event A happens-before event B" relation are the guarantees of ordering of events between threads:
  - Action A followed by action B in the same thread,
  - actions before start of a thread happen before actions in the thread,
  - unlock/lock,
  - write of volatile field happens-before subsequent reads.

Summary

- Concurrent programming is very difficult when mutable state is in place
- A ton of idioms, “best practices”, but still many problems
- Does not easily scale beyond a few collaborating threads
- In particular, lock-based programs do not compose
    - E.g., assume a container class that has thread-safe insert and delete. It is not possible to write a thread-safe “delete item x from container c1 and add it to container c2” (see Harris et al.: “Composable Memory Transactions”)
- We do not yet know how to best program concurrent programs
- Several alternative approaches to lock-based concurrency exist such as non-blocking concurrency and software transactional memory

Summary

- Java’s concurrency support has been changing
- Atomic primitives only added in SDK 5
- Reflects the status quo: better programming models are being researched and developed, no fully satisfactory solution
- Gradually gaining better understanding, e.g, of transactions (e.g., Moore and Grossman, “High-Level Small-Step Operational Semantics for Transactions,” POPL2008)
- Expect more changes in the future. Even if transactions wrt. memory will be solved, how about wrt. external interaction through disk or other storage