Introduction to Java Threads and the java.lang.Thread class

• Execution of a Java program begins with an instance of Thread created by the Java Virtual Machine (JVM) that executes the program’s main() method.

• Parallelism can be introduced by creating additional instances of class Thread that execute as parallel threads.

```java
public class Thread extends Object implements Runnable {
    Thread() { ... } // Creates a new Thread
    Thread(Runnable r) { ... } // Creates a new Thread with Runnable object r
    void run() { ... } // Code to be executed by thread
    // Case 1: If this thread was created using a lambda
    // then that object’s run method
    // Case 2: If this class is subclassed, the
    // in the subclass is called
    void start() { ... } // Causes this thread to
    void join() { ... } // Wait for this thread to die
    void join(long m) // Wait at most m milliseconds for thread to die
    static Thread currentThread() // Returns currently executing thread
    ...
}
```

A lambda can be passed as a Runnable
start() and join() methods

• A Thread instance starts executing when its start() method is invoked
  — start() can be invoked at most once per Thread instance
  — As with async, the parent thread can immediately move to the next statement after invoking t.start()

• A t.join() call forces the invoking thread to wait till thread t completes.
  — Lower-level primitive than finish since it only waits for a single thread rather than a collection of threads
  — No restriction on which thread performs a join on which thread, so it is possible to create a deadlock cycle using join() even when there are no data races
Two-way Parallel Array Sum using Java Threads

1. // Start of main thread
2. sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3. Thread t1 = new Thread(() -> {
4.     // Child task computes sum of lower half of array
5.     for(int i=0; i < X.length/2; i++) sum1 += X[i];
6. });
7. t1.start();
8. // Parent task computes sum of upper half of array
9. for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. // Parent task waits for child task to complete (join)
11. t1.join();
12. return sum1 + sum2;
1. // Start of Task T0 (main program)
2. sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
3. finish(() -> {
4.   async(() -> {
5.     // Child task computes sum of lower half of array
6.     for(int i=0; i < X.length/2; i++) sum1 += X[i];
7.   });
8.   // Parent task computes sum of upper half of array
9.   for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
10. });
11. // Parent task waits for child task to complete (join)
12. return sum1 + sum2;

Compare with Two-way Parallel Array Sum using HJ-Lib’s finish & async API’s
HJlib runtime uses Java threads as workers

- HJlib runtime creates a small number of worker threads in a *thread pool*, typically one per core.
- Workers push async's/continuations into a logical work queue:
  - when an async operation is performed
  - when an end-finish operation is reached
- Workers pull task/continuation work item when they are idle.

```java
final int numThreads = numWorkerThreads();
```

Static & instance fields are *shared* among tasks.
Locking guarantees in Java

- It is preferable to use java.util.concurrent.atomic or HJlib isolated constructs, since they cannot deadlock.

- Locks are needed for more general cases. Basic idea is for JVM to implement `synchronized(a) <stmt>` as follows:
  1. Acquire lock for object `a`
  2. Execute `<stmt>`
  3. Release lock for object `a`

- The responsibility for ensuring that the choice of locks correctly implements the semantics of isolation lies with the programmer.

- The main guarantee provided by locks is that only one thread can hold a given lock at a time, and the thread is blocked when acquiring a lock if the lock is unavailable.
Implementation of Java synchronized statements/methods

- Every object has an associated lock
- “synchronized” is translated to matching monitorenter and monitorexit bytecode instructions for the Java virtual machine
  - monitorenter requests “ownership” of the object’s lock
  - monitorexit releases “ownership” of the object’s lock
- If a thread performing monitorenter does not gain ownership of the lock (because another thread already owns it), it is placed in an unordered “entry set” for the object’s lock
Locks

- Use of monitor synchronization is just fine for most applications, but it has some shortcomings
  - Single wait-set per lock
  - No way to interrupt or time-out when waiting for a lock
  - Locking must be block-structured
    - Inconvenient to acquire a variable number of locks at once
    - Advanced techniques, such as hand-over-hand locking, are not possible

- Lock objects address these limitations
  - But harder to use: Need `finally` block to ensure release
  - So if you don’t need them, stick with `synchronized`

Example of hand-over-hand locking:
- L1.lock() ... L2.lock() ... L1.unlock() ... L3.lock() ... L2.unlock() ...
java.util.concurrent.locks.Lock interface

1. interface Lock {
2.     // key methods
3.     void lock(); // acquire lock
4.     void unlock(); // release lock
5.     boolean tryLock(); // Either acquire lock (returns true), or return false if lock is not obtained.
6.     // A call to tryLock() never blocks!
7.
8.     Condition newCondition(); // associate a new condition
9. }

dependencies: java.util.concurrent.locks.lock interface is implemented by java.util.concurrent.locks.ReentrantLock class
Simple ReentrantLock() example

- Used extensively within `java.util.concurrent`
  ```java
  final Lock lock = new ReentrantLock();
  ...
  lock.lock();
  try {
    // perform operations protected by lock
  } catch (Exception ex) {
    // restore invariants & rethrow
  }
  finally {
    lock.unlock();
  }
  ```

- Must manually ensure lock is released

  ==> Importance of including call to `unlock()` in finally clause!
What if you want to wait for shared state to satisfy a desired property? (Bounded Buffer Example)

1. public synchronized void insert(Object item) { // producer
2. // TODO: wait till count < BUFFER SIZE
3. ++count;
4. buffer[in] = item;
5. in = (in + 1) % BUFFER SIZE;
6. // TODO: notify consumers that an insert has been performed
7. }

9. public synchronized Object remove() { // consumer
10. Object item;
11. // TODO: wait till count > 0
12. --count;
13. item = buffer[out];
14. out = (out + 1) % BUFFER SIZE;
15. // TODO: notify producers that a remove() has been performed
16. return item;
17.}
The Java wait() Method

- A thread can perform a `wait()`:
  1. the thread releases the object lock
  2. thread state is set to blocked
  3. thread is placed in the wait set

- Causes thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

- Should always be used in a loop
Entry and Wait Sets

![Diagram of entry and wait sets]

- **Entry Set**: A set of processes that hold the lock and are allowed to enter.
- **Object Lock**: The lock that is protected.
- **Owner**: The process that currently owns the lock.
- **Wait Set**: A set of processes that are waiting for the lock to become available.

Processes acquire the lock and release it when they complete their work or enter the wait set if the lock is not available.
The notify() Method

When a thread calls `notify()`, the following occurs:

1. selects an arbitrary thread \( T \) from the wait set
2. moves \( T \) to the entry set
3. sets \( T \) to Runnable

\( T \) can now compete for the object’s lock again
Multiple Notifications

- `notify()` selects an arbitrary thread from the wait set.
  - This may not be the thread that you want to be selected.
  - Java does not allow you to specify the thread to be selected.

- `notifyAll()` removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.

- `notifyAll()` is a conservative strategy that works best when multiple threads may be in the wait set.
What if you want to wait for shared state to satisfy a desired property? (Bounded Buffer Example)

1. public synchronized void insert(Object item) { // producer
2.     while(count == buffer.length()) wait();
3.     ++count;
4.     buffer[in] = item;
5.     in = (in + 1) % BUFFER SIZE;
6.     notify();
7. }

9. public synchronized Object remove() { // consumer
10.    Object item;
11.    while(count == 0) wait();
12.    --count;
13.    item = buffer[out];
14.    out = (out + 1) % BUFFER SIZE;
15.    notify();
16.    return item;
17.}
java.util.concurrent.locks.Condition interface

• Can be allocated by calling ReentrantLock.newCondition()
• Supports multiple condition variables per lock
• Methods supported by an instance of condition
  —void await() // NOTE: like wait() in synchronized statement
    – Causes current thread to wait until it is signaled or interrupted
    – Variants available with support for interruption and timeout
  —void signal() // NOTE: like notify() in synchronized statement
    – Wakes up one thread waiting on this condition
  —void signalAll() // NOTE: like notifyAll() in synchronized statement
    – Wakes up all threads waiting on this condition
• For additional details see
  —http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html
BoundedBuffer Example using Two Conditions: full and empty

1. class BoundedBuffer {
2.     final Lock lock = new ReentrantLock();
3.     final Condition full = lock.newCondition();
4.     final Condition empty = lock.newCondition();
5. 
6.     final Object[] items = new Object[100];
7.     int putptr, takeptr, count;
8. 
9.     ...
BoundedBuffer Example using Two Conditions: full and empty (contd)

1. public void put(Object x) throws InterruptedException
2. {
3.   lock.lock();
4.   try {
5.     while (count == items.length) full.await();
6.     items[putptr] = x;
7.     if (++putptr == items.length) putptr = 0;
8.     ++count;
9.     empty.signal();
10.   } finally {
11.     lock.unlock();
12.   }
13. }
BoundedBuffer Example using Two Conditions:
full and empty (contd)

1. public Object take() throws InterruptedException
2. {
3.     lock.lock();
4.     try {
5.         while (count == 0) empty.await();
6.         Object x = items[takeptr];
7.         if (++takeptr == items.length) takeptr = 0;
8.         --count;
9.         full.signal();
10.        return x;
11.    } finally {
12.        lock.unlock();
13.    }