COMP 322: Fundamentals of Parallel Programming

Lecture 24: Java Threads,Java synchronized statement

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

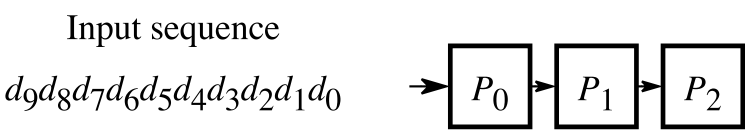
http://comp322.rice.edu

Worksheet #23: Analyzing Parallelism in an Actor Pipeline

Consider a three-stage pipeline of actors (as in slide 5), set up so that P0.nextStage = P1, P1.nextStage = P2, and P2.nextStage = null. The process() method for each actor is shown below. Assume that 100 non-null messages are sent to actor P0 after all three actors are started, followed by a null message. What will the total WORK and CPL be for this execution? Recall that each actor has a sequential thread.

Solution: WORK = 300, CPL = 102

...




```
protected void process(final Object msg) {
    if (msg == null) {
      exit(); //actor will exit after returning from process()
    } else {
      doWork(1); // unit work
    }
    if (nextStage != null) {
      nextStage.send(msg);
    }
} // process()
```

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.

```
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 thre
   // Case 1: If this thread was
   //         then that object's run method
   // Case 2: If this class is subclassed, t
   //         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
    - Declaring thread references as final does not help because the new() and start() operations are separated for threads (unlike futures, where they are integrated)

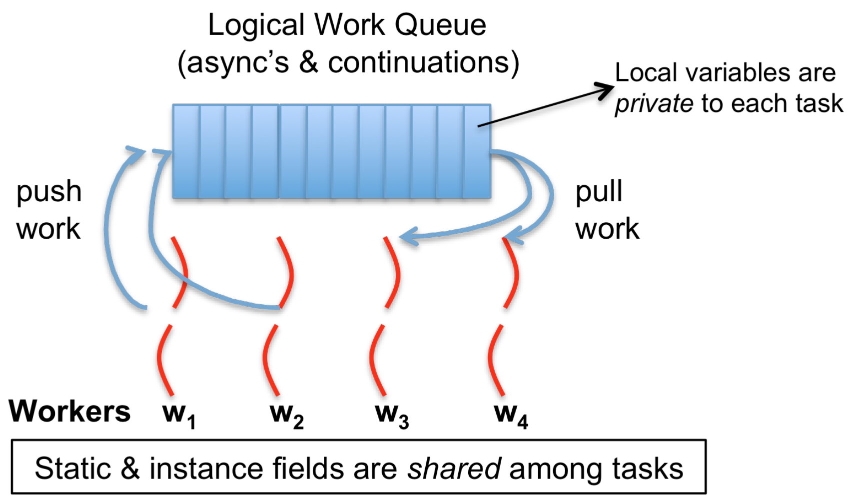
Two-way Parallel Array Sum using Java Threads

```
// Start of main thread
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
Thread t1 = new Thread(() -> {
    // Child task computes sum of lower half of array
    for(int i=0; i < X.length/2; i++) sum1 += X[i];
  });
t1.start();
// Parent task computes sum of upper half of array
for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
// Parent task waits for child task to complete (join)
t1.join();
return sum1 + sum2;
```

Compare with Two-way Parallel Array Sum using HJ-Lib's finish & async API's

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish(() -> {
  async(() -> {
    // Child task computes sum of lower half of array
    for(int i=0; i < X.length/2; i++) sum1 += X[i];
  });
  // Parent task computes sum of upper half of array
  for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
});
// Parent task waits for child task to complete (join)
return sum1 + sum2;
```

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

Objects and Locks in Java — synchronized statements and methods

- Every Java object has an associated lock acquired via:
  - `synchronized` statements
    - ```
      synchronized( foo ) { // acquire foo's lock
         // execute code while holding foo's lock
      } // release foo's lock
      ```
  - `synchronized` methods
    - ```
      public synchronized void op1() { // acquire 'this' lock
         // execute method while holding 'this' lock
      } // release 'this' lock
      ```
- Java language does not enforce any relationship between the object used for locking and objects accessed in isolated code
  - If same object is used for locking and data access, then the object behaves like a monitor
- Locking and unlocking are automatic
  - Locks are released when a synchronized block exits
    - By normal means: end of block reached, return, break
    - When an exception is thrown and not caught

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.

Deadlock example with Java synchronized statement

- The code below can deadlock if `leftHand()` and `rightHand()` are called concurrently from different threads
  - Because the locks are not acquired in the same order

```
public class ObviousDeadlock {
   . . .
   public void leftHand() {
     synchronized(lock1) {
       synchronized(lock2) {
         for (int i=0; i<10000; i++)
            sum += random.nextInt(100);
       }
     }
   }
```

```
   public void rightHand() {
     synchronized(lock2) {
       synchronized(lock1) {
         for (int i=0; i<10000; i++)
           sum += random.nextInt(100);
       }
     }
   }
}
```

Deadlock avoidance in HJ with object-based isolation

- HJ implementation ensures that all locks are acquired in the same order
- ==> no deadlock

```
public class ObviousDeadlock {
   . . .
   public void leftHand() {
     isolated(lock1,lock2) {
       for (int i=0; i<10000; i++)
          sum += random.nextInt(100);
     }
   }
}
```

```
   public void rightHand() {
     isolated(lock2, lock1) {
       for (int i=0; i<10000; i++)
         sum += random.nextInt(100);
     }
   }
}
```

Dynamic Order Deadlocks

- There are even more subtle ways for threads to deadlock due to inconsistent lock ordering
  - Consider a method to transfer a balance from one account to another:

```
public class SubtleDeadlock {
       public void transferFunds(Account from,
                                 Account to,
                                 int amount) {
           synchronized (from) {
               synchronized (to) {
                   from.subtractFromBalance(amount);
                   to.addToBalance(amount);
               }
           }
       }
   }
```

  - What if one thread tries to transfer from A to B while another tries to transfer from B to A ?

    Inconsistent lock order again – Deadlock!

Avoiding Dynamic Order Deadlocks

- The solution is to induce a lock ordering
  - Here, uses an existing unique numeric key, `acctId`, to establish an order

```
public class SafeTransfer {
    public void transferFunds(Account from, Account to, int amount) {
        Account firstLock, secondLock;
         if (fromAccount.acctId == toAccount.acctId)
             throw new Exception("Cannot self-transfer");
         else if (fromAccount.acctId < toAccount.acctId) {
             firstLock = fromAccount;
             secondLock = toAccount;
         }
         else {
             firstLock = toAccount;
             secondLock = fromAccount;
         }
         synchronized (firstLock) {
           synchronized (secondLock) {
              from.subtractFromBalance(amount);
              to.addToBalance(amount);
           }
        }
    }
  }
```

Java’s Object Locks are Reentrant

- Locks are granted on a per-thread basis
  - Called reentrant or recursive locks
  - Promotes object-oriented concurrent code
- A synchronized block means execution of this code requires the current thread to hold this lock
  - If it does — fine
  - If it doesn’t — then acquire the lock
- Reentrancy means that recursive methods, invocation of super methods, or local callbacks, don’t deadlock

```
public class Widget {
   public synchronized void doSomething() { ... }
}
public class LoggingWidget extends Widget {
   public synchronized void doSomething() {
       Logger.log(this + ": calling doSomething()");
       super.doSomething();  // Doesn't deadlock!
    }
 }
```

Worksheet #24: Java Threads and Data Races

Name: ______________________________ Net ID: ________________

1) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using start() and join() operations.

2) Write a sketch of the pseudocode for a Java threads program that exhibits a data race using synchronized statements

BACKUP SLIDES START HERE

Monitors

- One definition of monitor is a thread-safe class, object, or module that uses wrapped mutual exclusion in order to safely allow access to a method or variable by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: At each point in time, at most one thread may be executing any of its methods. Using a condition variable(s), it can also provide the ability for threads to wait on a certain condition (thus using the above definition of a "monitor"). For the rest of this article, this sense of "monitor" will be referred to as a "thread-safe object/class/module".

- Source: https://en.wikipedia.org/wiki/Monitor_(synchronization)

How to convert a sequential library to a monitor in HJ vs. Java?

HJ approach:

- Use object-based isolation to ensure that each call to a public method is isolated on "this" e.g.,

  public void add(...) { isolated(this) { .... } }

- Can also use general isolated statement, but that is overkill e.g.,

  public void add(...) { isolated { .... } }

Java approach:

- Use Java's synchronized statement instead of object-based isolation e.g.,

  public void add(...) { synchronized(this) { .... } }

  or equivalently

  public synchronized void add(...) { .... }

- Both HJ and Java programs can use specialized implementations of monitors available in java.util.concurrent
  - ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArraySet

Unit 7.1: Java Threads (Recap)

- 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.

```
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 thr
   // Case 1: If this thread was cr
   //          then that object's run method
   // Case 2: If this class is subclassed, t
   //          in the subclass is called
  void start() { ... } // Causes this thread to st
  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

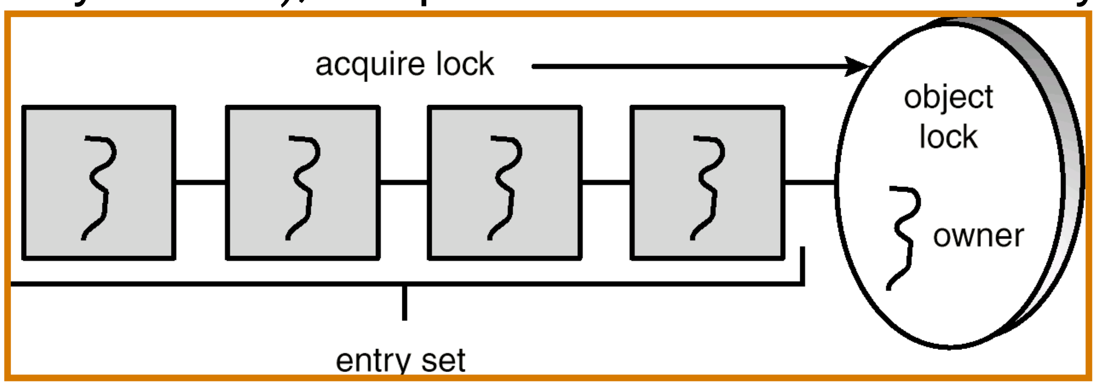
Listing 3: java.lang.Thread class

Unit 7.2: Objects and Locks in Java --- synchronized statements and methods (Recap)

- Every Java object has an associated *lock* acquired via:
  - `synchronized` statements
    - synchronized(foo) { // acquire foo's lock
// execute code while holding foo's lock
} // release foo's lock
  - `synchronized` methods
    - public synchronized void op1() { // acquire 'this' lock
// execute method while holding 'this' lock
} // release 'this' lock
- Java language does not enforce any relationship between object used for locking and objects accessed in isolated code
  - If same object is used for locking and data access, then the object behaves like a monitor
- Locking and unlocking are automatic
  - Locks are released when a synchronized block exits
    - By normal means: end of block reached, return, break
    - When an exception is thrown and not caught

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



Monitors – a Diagrammatic summary

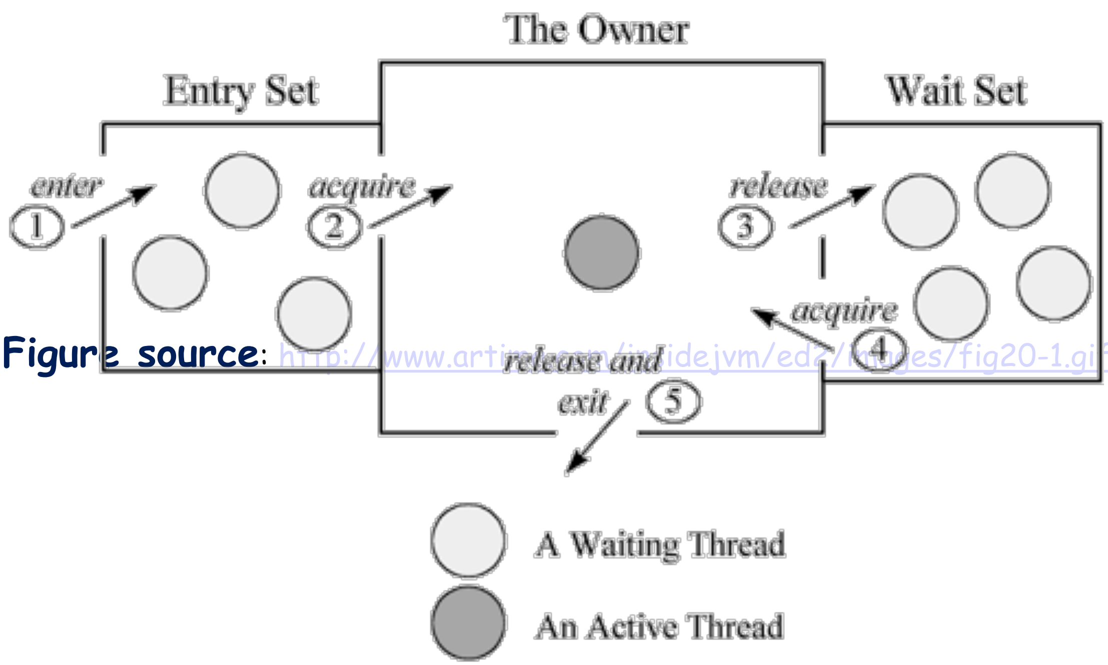


Figure source: http://www.artima.com/insidejvm/ed2/images/fig20-1.gif

Figure 20-1. A Java monitor.

What if you want to wait for shared state to satisfy a desired property? (Bounded Buffer Example)

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

public synchronized Object remove() { // consumer
  Object item;
   // TODO: wait till count > 0
  --count;
  item = buffer[out];
  out = (out + 1) % BUFFER SIZE;
  // TODO: notify producers that a remove() has been performed
  return item;
}
```

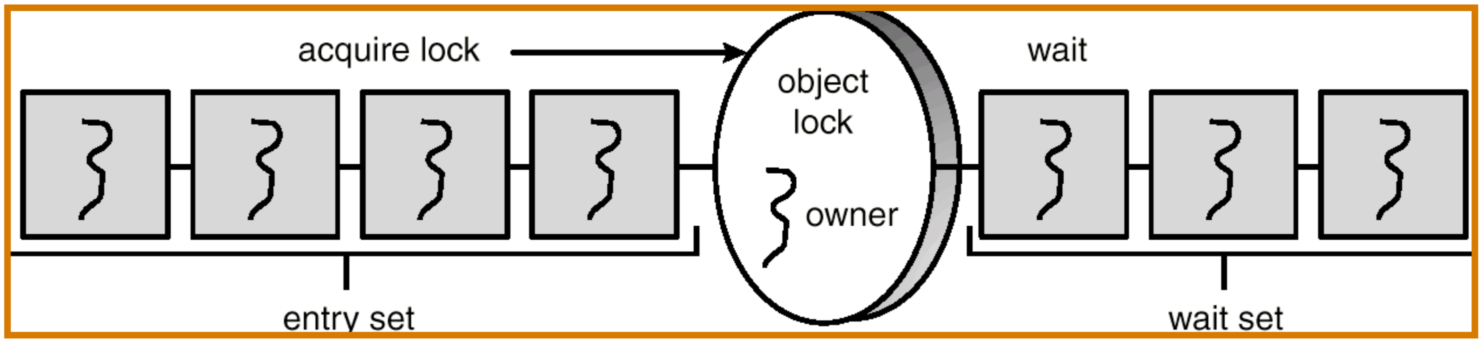
The Java wait() Method

- A thread can perform a wait() method on an object that it owns:
  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.
- Since interrupts and spurious wake-ups are possible, this method should always be used in a loop e.g.,

```
synchronized (obj) {
    while (<condition does not hold>)
        obj.wait();
    ... // Perform action appropriate to condition
}
```

- Java's wait-notify is related to "condition variables" in POSIX threads

Entry and Wait Sets



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

insert() with wait/notify Methods

```
public synchronized void insert(Object item) {
    while (count == BUFFER SIZE) {
            try {
                    wait();
            }
            catch (InterruptedException e) { }
    }
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    notify();
}
```

remove() with wait/notify Methods

```
public synchronized Object remove() {
    Object item;
    while (count == 0) {
                    try {
                            wait();
                    }
                    catch (InterruptedException e) { }
    }
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    notify();
    return item;
}
```

Complete Bounded Buffer using Java Synchronization

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER SIZE = 5;
    private int count, in, out;
    private Object[] buffer;
    public BoundedBuffer() { // buffer is initially empty
                count = 0;
                in = 0;
                out = 0;
                buffer = new Object[BUFFER SIZE];
    }
    public synchronized void insert(Object item) { // See previous slides
    }
    public synchronized Object remove() { // See previous slides
    }
}
```

Worksheet #27: Entry and Wait Sets

Name: ______________________ Netid: ______________________

Consider a bounded buffer implementation using synchronized() statements as discussed in the lecture, with BUFFER_SIZE = 1. Assume that there are multiple producers (P0, P1, ...) and multiple consumers (C0, C1, …). Can you create a "livelock" scenario using only notify() and wait(), but not notifyAll()? i.e., a scenario in which threads are repeatedly inserted and removed from entry/wait sets without allowing a ready producer/consumer to make progress.