COMP 322: Fundamentals of Parallel Programming

Lecture 27: Safety and Liveness Properties, Java Synchronizers, Dining Philosophers Problem

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/
Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock (see slide 8) instead of synchronized. Your goal is to write a correct implementation that never deadlocks, unlike the buggy version below (which can deadlock). Assume that each Account object already contains a reference to a ReentrantLock object dedicated to that object e.g., from.lock() returns the lock for the from object. Sketch your answer below using pseudocode.

1. public void transferFunds(Account from, Account to, int amount) {
2.     while (true) {
3.         // assume that trylock() does not throw an exception
4.         boolean fromFlag = from.lock.trylock();
5.         if (!fromFlag) continue;
6.         boolean toFlag = to.lock.trylock();
7.         if (!toFlag) { from.lock.unlock(); continue; }
8.         try {
9.             from.subtractFromBalance(amount);
10.            to.addToBalance(amount); break;
11.         } finally {
12.             from.lock.unlock(); to.lock.unlock();
13.         }
14.     } // while
15. } // while
Is this a linearizable execution for a FIFO queue, q?

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke q.enq(x)</td>
<td>Invoke q.enq(y)</td>
</tr>
<tr>
<td>1</td>
<td>Return from q.enq(x)</td>
<td>Work on q.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>3</td>
<td>Invoke q.deq()</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Work on q.deq()</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Return y from q.deq()</td>
<td></td>
</tr>
</tbody>
</table>

No! q.enq(x) must precede q.enq(y) in all linear sequences of method calls invoked on q. It is illegal for the q.deq() operation to return y.
Outline

- Safety and Liveness
- Java Synchronizers
- Dining Philosophers Problem
Safety vs. Liveness

• In a concurrent setting, we need to specify both the safety and the liveness properties of an object

• Need a way to define
  — Safety: when an implementation is functionally correct (does not produce a wrong answer)
  — Liveness: the conditions under which it guarantees progress (completes execution successfully)

• Examples of safety
  • Data race freedom is a desirable safety property for parallel programs (Module 1)
  • Linearizability is a desirable safety property for concurrent objects (Module 2)
Liveness

• Liveness = a program’s ability to make progress in a timely manner

• Termination ("no infinite loop") is not necessarily a requirement for liveness
  
  • some applications are designed to be non-terminating

• Different levels of liveness guarantees (from weaker to stronger) for tasks/threads in a concurrent program
  
  1. Deadlock freedom
  2. Livelock freedom
  3. Starvation freedom
  4. Bounded wait
1. Deadlock-Free Parallel Program Executions

- A parallel program execution is **deadlock-free** if no task’s execution remains incomplete due to it being blocked awaiting some condition.

- Example of a program with a deadlocking execution:

```java
// Thread T1
public void leftHand() {
    synchronized(obj1) {
        synchronized(obj2) {
            // work with obj1 & obj2
            ...
        }
    }
}

// Thread T2
public void leftHand() {
    synchronized(obj2) {
        synchronized(obj1) {
            // work with obj2 & obj1
            ...
        }
    }
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
  - Three constructs that can lead to deadlock in HJlib: async await, finish w/ actors, explicit phaser wait (instead of next)
  - There are many constructs that can lead to deadlock cycles in other programming models (e.g., thread join, synchronized, locks in Java)
2. Livelock-Free Parallel Program Executions

- A parallel program execution exhibits *livelock* if two or more tasks repeat the same interactions without making any progress (special case of nontermination)

- Livelock example:

```java
// Task T1
incrToTwo(AtomicInteger ai) {
    // increment ai till it reaches 2
    while (ai.incrementAndGet() < 2);
}

// Task T2
decrToNegTwo(AtomicInteger ai) {
    // decrement ai till it reaches -2
    while (a.decrementAndGet() > -2);
}
```

- Many well-intended approaches to avoid deadlock result in livelock instead

- Any HJlib program that uses only Module 1 features, and is data-race-free, is guaranteed to be livelock-free (may be nonterminating in a single task, however)
3. Starvation-Free Parallel Program Executions

- A parallel program execution exhibits *starvation* if some task is repeatedly denied the opportunity to make progress
  - Starvation-freedom is sometimes referred to as “lock-out freedom”
  - Starvation is possible in HJ programs, since all tasks in the same program are assumed to be cooperating, rather than competing
    - If starvation occurs in a deadlock-free HJ program, the “equivalent” sequential program must be non-terminating (infinite loop)

- Classic source of starvation for OS threads: “Priority Inversion”
  - Thread A is at high priority, waiting for result or resource from Thread C at low priority
  - Thread B at intermediate priority is CPU-bound
  - Thread C never runs (because its priority is lower than B’s priority), hence thread A never runs
  - Fix: when a high priority thread waits for a low priority thread, boost the priority of the low-priority thread
Related Concept: Progress Conditions for shared resources

- A resource is said to be \textit{obstruction}-free if it is deadlock-free
- A resource is said to be \textit{lock-free} if it is livelock-free and deadlock-free
- A resource is said to be \textit{wait-free} if it is starvation-free, livelock-free, and deadlock-free
  - Wait-free $\Rightarrow$ every thread/task will eventually get an opportunity to make progress, i.e., to access the shared resource
  - Question: how to bound the wait duration?
4. Bounded Wait

- A parallel program execution exhibits bounded wait if each task requesting a resource should only have to wait for a bounded number of other tasks to “cut in line” i.e., to gain access to the resource after its request has been registered.

- If bound = 0, then the program execution is fair
Outline

- Safety and Liveness
- Java Synchronizers
- Dining Philosophers Problem
Key Functional Groups in java.util.concurrent (j.u.c.)

- **Atomic variables**
  - The key to writing lock-free algorithms

- **Concurrent Collections:**
  - Queues, blocking queues, concurrent hash map, ...
  - Data structures designed for concurrent environments

- **Locks and Conditions**
  - More flexible synchronization control
  - Read/write locks

- **Executors, Thread pools and Futures**
  - Execution frameworks for asynchronous tasking

- **Synchronizers:** Semaphore, Latch, Barrier, Exchanger
  - Ready made tools for thread coordination
j.u.c Synchronizers --- common patterns in HJ’s phaser construct

- Class library includes several state-dependent synchronizer classes
  - `CountDownLatch` – waits until latch reaches terminal state
  - `Semaphore` – waits until permit is available
  - `CyclicBarrier` – like barriers in HJlib forall loops
  - `Phaser` – inspired by Habanero phasers
  - `FutureTask` – like futures in HJlib
  - `Exchanger` – waits until two threads rendezvous (special synchronization)

- These typically have three main groups of methods
  - Methods that block until the object has reached the right state
    - Timed versions will fail if the timeout expired
    - Many versions can be cancelled via interruption
  - Polling methods that allow non-blocking interactions
  - State change methods that may release a blocked method

- **WARNING:** synchronizers should only be used in Java threads, not HJlib tasks, since they can cause the HJlib runtime system to deadlock
CountDownLatch

- A counter that releases waiting threads when it reaches zero
  - Allows one or more threads to wait for one or more events
  - Initial value of 1 gives a simple gate or latch

  ```java
  CountDownLatch(int initialValue)
  ```

- `await()`: wait until the counter is zero
  - `await()` is what differentiates a CountDownLatch from an AtomicInteger

- `countDown()`: decrement the counter if > 0

- Query: `getCount()`

- Very simple but widely useful
  - Replaces error-prone attempts with data races
Example: using j.u.c.CountDownLatch to implement finish for Java threads

- Problem: Run N tasks concurrently in N threads and wait until all are complete

— Use a CountDownLatch initialized to the number of threads

```
public static void runTask(int numThreads, final Runnable task)
    throws InterruptedException {
    final CountDownLatch done = new CountDownLatch(numThreads);
    for (int i=0; i<numThreads; i++) {
        Thread t = new Thread() {
            public void run() {
                try {
                    task.run();
                } finally { done.countDown();}
            }
        };
        t.start();
    }
    done.await(); // wait for all threads to finish
```
Semaphores

- Conceptually serve as “permit” holders
  - Construct with an initial number of permits
    - \texttt{acquire()}: waits for permit to be available, then “takes” one, i.e., decrements the count of available permits
    - \texttt{release()}: “returns” a permit, i.e., increments the count of available permits
  - But no actual permits change hands
    - The semaphore just maintains the current count
    - Thread performing \texttt{release()} can be different from the thread performing \texttt{acquire()}

- “fair” variant hands out permits in FIFO order
- Useful for managing bounded access to a shared resource
public class BoundedBlockingList {
    final int capacity;
    final ConcurrentLinkedList list = new ConcurrentLinkedList();
    final Semaphore sem;
    public BoundedBlockingList(int capacity) {
        this.capacity = capacity;
        sem = new Semaphore(capacity);
    }

    public void addFirst(Object x) throws InterruptedException {
        sem.acquire(); // blocks until a permit is available
        try { list.addFirst(x); }
        catch (Throwable t){ sem.release(); rethrow(t); } // only performed on exception
    }

    public boolean remove(Object x) {
        if (list.remove(x)) { sem.release(); return true; }
        return false;
    }

    ... } // BoundedBlockingList
Outline

• Safety and Liveness

• Java Synchronizers

• Dining Philosophers Problem
  — Acknowledgments
    – CMSC 330 course notes, U. Maryland
    – Dave Johnson (COMP 421 instructor)
The Dining Philosophers Problem

Constraints
- Five philosophers either eat or think
- They must have two forks to eat (chopsticks are a better motivation!)
- Can only use forks on either side of their plate
- No talking permitted

Goals
- Progress guarantees
  - Deadlock freedom
  - Livelock freedom
  - Starvation freedom
  - Maximum concurrency (no one should starve if there are available forks for them)
1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
  5.   while(true) {
  6.     Think ;
  7.     Acquire forks;
  8.       // Left fork = fork[p]
  9.       // Right fork = fork[(p-1)%numForks]
 10.     Eat ;
 11.   } // while
12.} // forall
Solution 1: using Java’s synchronized statement

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.   while(true) {
6.     Think ;
7.     synchronized(fork[p])
8.       synchronized(fork[(p-1)%numForks]) {
9.         Eat ;
10.    }
11. }
12. } // while
13. } // forall
Solution 2: using Java’s Lock library

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.   while(true) {
6.     Think ;
7.     if (!fork[p].lock.tryLock()) continue;
8.     if (!fork[(p-1)%numForks].lock.tryLock()) {
9.       fork[p].lock.unlock(); continue;
10.   }
11.  }
12.  Eat ;
13.  fork[p].lock.unlock();fork[(p-1)%numForks].lock.unlock();
14. } // while
15.} // forall
Solution 3: using HJ’s isolated statement

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.   while(true) {
6.     Think ;
7.     isolated {
8.       Pick up left and right forks;
9.       Eat ;
10.   }
11. } // while
12.} // forall
Solution 4: using HJ’s object-based isolation

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. forall(point [p] : [0:numPhilosophers-1]) {
5.   while(true) {
6.     Think ;
7.     isolated(fork[p], fork[(p-1)%numForks]) {
8.       Eat ;
9.     }
10. } // while
11. } // forall
Solution 5: using Java’s Semaphores

1. int numPhilosophers = 5;
2. int numForks = numPhilosophers;
3. Fork[] fork = ... ; // Initialize array of forks
4. Semaphore table = new Semaphore(4, true);
5. for (i=0;i<numForks;i++) fork[i].sem = new Semaphore(1, true);
6. forall(point [p] : [0:numPhilosophers-1]) {
   7.   while(true) {
      8.     Think ;
   9.     table.acquire(); // At most 4 philosophers at table
10.    fork[p].sem.acquire(); // Acquire left fork
11.    fork[(p-1)%numForks].sem.acquire(); // Acquire right fork
12.    Eat ;
13.    fork[p].sem.release(); fork[(p-1)%numForks].sem.release();
14.    table.release();
15.  } // while
16.} // forall

“true” parameter creates a semaphore that guarantees fairness