COMP 322: Fundamentals of Parallel Programming

Lecture 24: Safety and Liveness Properties, Introduction to Java Threads

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

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 correct
  - Liveness: the conditions under which it guarantees progress

- Data race freedom is a desirable safety property for most parallel programs
- Linearizability is a desirable safety property for most concurrent objects

Desirable Properties of Parallel Program Executions

- Data-race freedom
- Termination
  - But some applications are designed to be non-terminating
- Liveness = a program's ability to make progress in a timely manner
- Different levels of liveness guarantees (from weaker to stronger)
  - —Deadlock freedom
  - —Livelock freedom
  - —Starvation freedom
  - —Bounded wait

Terminating Parallel Program Executions

- A parallel program execution is terminating if all sequential tasks in the program terminate
- Example of a nondeterministic data-race-free program with a nonterminating execution

```
p.x = false;
finish {
  async { // S1
     boolean b = false; do { isolated b = p.x; } while (! b);
  }
  isolated p.x = true; // S2
} // finish
```

- Some executions of this program may be terminating, and some not
- Cannot assume in general that statement S2 will ever get a chance to execute if async S1 is nonterminating e.g., consider case when program is run with one worker (-places 1:1)

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

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await ( left ) right.put(rightBuilder()); // Task1
 async await ( right ) left.put(leftBuilder()); // Task2
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
  – Three constructs that can lead to deadlock in HJ: async await, finish + actors, explicit phaser wait (instead of next)
  —There are many mechanisms that can lead to deadlock cycles in other programming models (e.g., locks)

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:

```
// Task 1
incrToTwo(AtomicInteger ai) {
  // increment ai till it reaches 2
  while (ai.incrementAndGet() < 2);
}
```

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

- Many well-intended approaches to avoid deadlock result in livelock instead
- Any data-race-free HJ program without isolated/atomic-variables/actors is guaranteed to be livelock-free (may be nonterminating in a single task, however)

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
- Classic source of starvation: “Priority Inversion” problem for OS threads
  - 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, 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

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

Do you understand deadlock, livelock, starvation, and unbounded wait? Let’s give Worksheet #24 a try.

Bounded Wait Mutual Exclusion Progress

Oversimplifying Assumptions

- **Bounded Wait?**
  - A process requesting access to a resource should only have to wait for a bounded number of other processes to access the resource that requested access after it

Classification of Parallel Programming Models

- Library approaches
  - —POSIX threads
  - —Message-Passing Interface (MPI)
  - —MapReduce frameworks
- Pseudocomment “pragma” approaches
  - —OpenMP
- Language approaches
  - —Habanero-Java
  - —Unified Parallel C
  - —Co-Array Fortran
  - —Chapel
  - —X10
  - —. . .

==> Java takes a library approach with a little bit of language support (synchronized keyword)

Closures

- Library-based approaches to parallel programming require interfaces in which computations can be passed as data
- Recall that a closure is a first-class function with free variables that are bound in function's lexical environment e.g., the anonymous lambda expression in the following Scheme program is a closure

```
; Return a list of all books with at least THRESHOLD copies sold.
(define (best-selling-books threshold)
  (filter
    (lambda (book)
      (>= (book-sales book) threshold))
   book-list))
```

- Note that the value of free variable threshold is captured when the lambda expression is defined

HJ Asyncs and Closures

- **The body of an HJ async task is a parameter-less closure that is both created and enabled for execution at the point when the async statement is executed**
- **An async captures the values of free variables (local variables in outer scopes) when it is created**
  - **—e.g., variable len in Listing 1 below**

```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
finish {
  // Compute sum1 (lower half) and sum2 (upper half) in parallel
  int len = X.length
  async for(int i=0 ; i < len/2 ; i++) sum1 += X[i]; // Task T2
  async for(int i=len/2 ; i < len ; i++) sum2 += X[i]; // Task T3
}
//Task T1 waits for Tasks T2 and T3 to complete
int sum = sum1 + sum2; // Continuation of Task T1
```

Listing 1: Two-way Parallel ArraySum in HJ

java.lang.Runnable interface

- **Any class that implements java.lang.Runnable must provide a parameter-less run() method with void return type**
- **Lines 3-7 in Listing 2 show the creation of an instance of an anonymous inner class that implements the Runnable interface**
- **The computation in the run() method can be invoked sequentially by calling r.run()**
  - **—We will see next how it can be invoked in parallel**

```
1  . . .
2  final int len = X.length;
3  Runnable r = new Runnable() {
4    public void run() {
5      for(int i=0 ; i < len/2 ; i++) sum1 += X[i];
6    }
7  };
8  . . .
```

Listing 2: Example of creating a Java Runnable instance as a closure

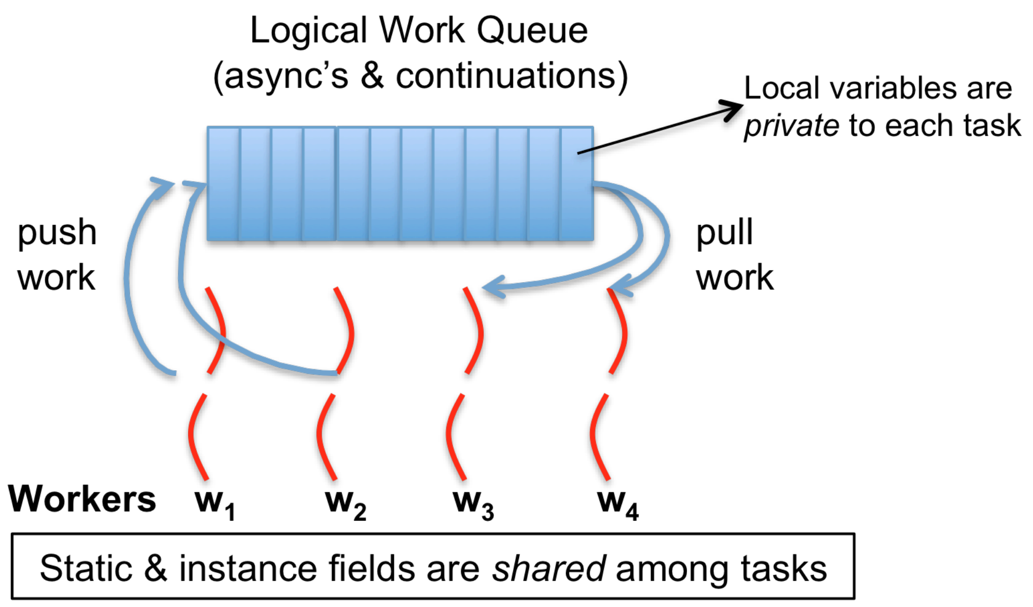
java.lang.Thread class

- **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 thread
   // Case 1: If this thread was created using a Runnable object,
   //         then that object's run method is called
   // Case 2: If this class is subclassed, then the run() method
   //         in the subclass is called
  void start() { ... } // Causes this thread to start execution
  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
  . . .
}
```

Listing 3: java.lang.Thread class

HJ runtime uses Java threads as workers …


Logical Work Queue
(async's & continuations)
Local variables are *private* to each task
push work
pull work
Workers
w_1
w_2
w_3
w_4
Static & instance fields are *shared* among tasks

- HJ runtime creates a small number of worker threads, 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

… because programming directly with Java threads can be expensive

k	$t_s(k)$	$t_1^{wf}(k)$	$t_1^{hf}(k)$	$t_1^{ws}(k)$	Java-thread(k)
1	0.11	0.21	0.22		
2	0.22	0.44	2.80		
4	0.44	0.88	2.95		
8	0.90	1.96	3.92	335	3,600
16	1.80	3.79	6.28		
32	3.60	7.15	10.37		
64	7.17	14.59	19.61		
128	14.47	28.34	36.31	2,600	63,700
256	28.93	56.75	73.16		
512	57.53	114.12	148.61		
1024	114.85	270.42	347.83	22,700	768,000

Fork-Join Microbenchmark Measurements (execution time in micro-seconds)

Two ways to specify computation for a Java thread

1. Define a class that implements the Runnable interface and pass an instance of that class to the Thread constructor in line 3 of slide 15
   - It is common to create an instance of an anonymous inner class that implements Runnable for this purpose. In this case, the Runnable instance defines the work to be performed, and the Thread instance identifies the worker that will perform the work.
2. Subclass Thread and override the run() method. This is usually inconvenient in practice because of Java's single-inheritance constraint.

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()
  - No notion of an Immediately Enclosing Finish in Java threads
  - No propagation of exceptions to parent/ancestor threads

Two-way Parallel ArraySum using Java threads

```
// Start of Task T1 (main program)
sum1 = 0; sum2 = 0; // Assume that sum1 & sum2 are fields (not local vars)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final int len = X.length;
Runnable r1 = new Runnable() {
  public void run(){ for(int i=0 ; i < len/2 ; i++) sum1 += X[i];}
};
Thread t1 = new Thread(r1);
t1.start();
Runnable r2 = new Runnable() {
  public void run(){ for(int i=len/2 ; i < len ; i++) sum2 += X[i];}
};
Thread t2 = new Thread(r2);
t2.start();
// Wait for threads t1 and t2 to complete
t1.join(); t2.join();
int sum = sum1 + sum2;
```

Worksheet #24: Liveness Guarantees

Name 1: ______________________ Name 2: ______________________

```
/** Atomically adds delta to the current value.
  *
  * @param delta the value to add
  * @return the previous value
  */
 public final int getAndAdd(int delta) {
     for (;;) {
         int current = get();
         int next = current + delta;
         if (compareAndSet(current, next))
              // commit
              return current;
      }
  }
```

Assume that multiple tasks call getAndAdd() repeatedly in parallel. Can this implementation of getAndAdd() lead to a) deadlock, b) livelock, c) starvation, or d) unbounded wait? Write and explain your answer below