COMP 322: Fundamentals of Parallel Programming

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

Lecture 35: Liveness and Progress Guarantees for Parallel Programs

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

Acknowledgments for Today's Lecture

- Maurice Herlihy and Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008.
  - Optional text for COMP 322
  - Slides and code examples extracted from http://www.elsevierdirect.com/companion.jsp?ISBN=9780123705914
- "Synchronization and Concurrency for User-level Systems", William N. Scherer III, Ph.D. Defense, U. Rochester, December 2005
- "The Java Tutorials --- Concurrency"
  - http://download.oracle.com/javase/tutorial/essential/concurrency
- "Introduction to Synchronization", Klara Nahrstedt, CS 241 Lecture 10, Spring 2007
  - www.cs.uiuc.edu/class/sp07/cs241/Lectures/10.sync.ppt

Announcements

- Homework 7 due by 5pm on Friday, April 22nd
  - Send email to comp322-staff if you're running into issues with accessing SUG@R nodes, or anything else

Desirable Properties of Parallel Program Executions

- Data-race freedom (Lecture 6)
- 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 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*. There are many mechanisms (e.g., locks) that can lead to deadlock cycles.
  - *No deadlock cycle possible with finish, isolated, phasers, and async's without await clauses*
    - *future async's and phased async's are fine*

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:
  - Source: http://stackoverflow.com/questions/1036364/good-example-of-livelock

```
// Thread 1
getLocks12(lock1, lock2) {
  lock1.lock();
  while (lock2.locked()) {
    // attempt to yield to other thread
    lock1.unlock(); wait(); lock1.lock();
  }  lock2.lock();
}
```

```
// Thread 2
getLocks21(lock2, lock1) {
  lock2.lock();
  while (lock1.locked()) {
    // attempt to yield to other thread
    lock2.unlock(); wait(); lock2.lock();
  }  lock1.lock();
}
```

- Many well-intended approaches to avoid deadlock result in livelock instead
- A practical heuristic (but not a guarantee) for avoiding livelock is to introduce randomization in distribution of requests
- Any data-race-free HJ program without isolated 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”
- Common source of starvation: adjustment of priorities
- Classic “Priority Inversion” problem
  - 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*

- Progress?
- Bounded Wait?

What's the difference?

- Progress?
  - If ***no process is waiting for a resource*** and several processes are requesting access to the resource, then access to the resource cannot be postponed indefinitely

- **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***

Related Concepts

- **A resource is said to be *wait-free* if it is starvation-free, livelock-free, and deadlock-free**
- **A resource is said to be *lock-free* if it is livelock-free and deadlock-free**
- **A resource is said to be *obstruction-free* if it is deadlock-free**