Multicore Computing Lecture 2

Chunheng Luo

August 30, 2016

1 Implementing Threads in Java

1. Extend the Thread interface → start
2. Implement Runnable/Callable → create threads → start

   [Example: FooBar.java]
3. Executor services

   Synchronous vs. asynchronous multithreading When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

   - Synchronous execution
     Using the "join" method: blocking construct
     Wait for all running threads to finish using join, and then start new threads.

     [Example: Fibonacci.java]
   - Asynchronous execution
     - Using ExecutorService and ThreadPool
       Submit threads to a thread pool, which manages the execution of all threads. Asynchronous execution is supported. Each submitted thread returns an object of "future" type, which

represents the output of the thread that has not been available at the moment. The "get" method of the "future" method waits until the thread finishes and then retrieves the final result.
[Example: Fibonacci2.java]

- Using RecursiveTask *to "fork" and "join" threads*
  The method "fork" arranges to asynchronously execute the task in the pool the current task is running in. [Example: Fibonacci3.java]

```
protected Integer compute() {
  if ((n == 0)||(n == 1 )) return 1;
  Fibonacci3 f1 = new Fibonacci3(n - 1);
  f1.fork();
  Fibonacci3 f2 = new Fibonacci3(n - 2);
  return f2.compute() + f1.join();
}
/*
 * By directly compute f2 first , the result of
 * Fibonacci3(n-2) is reused in the computation
 * of Fibonacci3(n-1), which is done during f1.join()
*/
```

2 Amdahl's Law

P: fraction of the work that can be done in parallel
n: number of cores
T_s: time on sequential processor
T_p: time on multicore machine

$$T_p \geq (1-P)T_s + \frac{PT_s}{n}$$

The speedup is

$$\frac{T_s}{T_p} \leq \frac{1}{1 - P + \frac{P}{n}}$$

3 Mutual Exclusion

If any core is in a crital section (CS), all other cores should not be in the critical section

Example: two cores are both doing x:=x+1, they cannot do it at the same time. (Wrong values can be wirtten into the mem otherwise.)

Some tentative solutions:

- Attemp1.java
  Several threads can "lock the door" simultaneously, which violates the mutual exclusion requirement.
- Attemp2.java
  If all threads request CS simultaneously, none will enter CS. That is called a deadlock.
- Attemp3.java
  If all threads assign the variable "turn" simultaneously, a deadlock occurs.

Canonical solution: Peterson Algorithm

```
/* For core P0 ------------------------------------------------ */
wantCS[0] = true;   // Raise the request to enter CS
turn = 1;           // Let the other core go first
while ( wantCS[1] && turn == 1 ) NO_OP();
{CS}
wantCS[0] = false;

/* For core P1 ------------------------------------------------ */
wantCS[1] = true;   // Raise the request to enter CS
```

```
turn = 0;              // Let the other core go first
while ( wantCS[0] && turn == 0 ) NO_OP();
{CS}
wantCS[1] = false;
```

Proof:

1. Deadlock-free

   Deadlock
   ⇒ (wantCS[1] && turn == 1) && (wantCS[0] && turn == 0)
   But turn cannot be 1 and 0 at the same time ⇒ conflict

2. Mutual exclusion

   Assume both are in CS, and let the value of turn be 1.
   Then at some point before this, turn = 0.
   Then at some point further before, wantCS[1] = true happens.
   Now we have (wantCS[1] && turn == 1) ⇒ P0 cannot enter CS.

   This informal proof does not seem to be correct. The formal proof (Dijkstra's proof) should be in the course material, and can be found on the Internet.