EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 2: August 30

Lecturer: Vijay Garg *Scribe: Rohan Nagar*

2.1 Puzzle (Continued)

Recall the puzzle from last time. We have an array of n integers and we want to find the maximum value in the array. Our best attempt so far was the *all pair comparison*, with execution time equal to $O(1)$ and work done equal to $O(n^2)$. Think of it as you have n people each with n helpers. They each can figure out themselves if they are the max value beacuse each of the n helpers can look at the other n numbers and determine if that number is bigger or smaller.

The problem with this algorithm is the work that needs to be done. It requires n^2 processors.

Let us introduce a new algorithm to improve on the amount of work done. We can divide the n integers into $\sqrt{n}$ groups of $\sqrt{n}$. Then, we find the maximun value in each group, and then finally one more step to find the maximum value overall. This still gives us an execution time of $O(1)$, but cuts the work down to $O(n^{\frac{3}{2}})$. All of our solutions so far are listed in the table below.

Algorithm	T	W
Sequential	$O(n)$	$O(n)$
Binary Tree	$O(log(n))$	$O(n)$
All Pair Comparison	$O(1)$	$O(n^2)$
Group-Based	$O(1)$	$O(n^{\frac{3}{2}})$

Question Can you find another solution that improves further on the amount of work done?

2.2 Creating Threads (in Java)

There are three ways to work with threads in Java.

- Extend Thread → call start()
- Implement Runnable or Callable → create new Thread → call start()
- Use Executor Service

2.2.1 Sample Code

All of the sample code examples are available on Dr. Garg's class Github page.

- FooBar.java → Implementing Runnable

  - Motivation for Implementing Runnable: Say, for example, that you already have a class that you want to keep the functionality of. You want to extend this class, but in Java you cannot extend multiple classes. Therefore, instead of extending Thread, you should implement Runnable so that you can still extend the class that you want functionality from.
  - To implement Runnable, override the run() method.
- Fibonacci.java → Waiting with join()
  - join() is used for waiting on a thread. The method is a blocking construct and will stop execution of the current thread to wait for the other thread to finish execution.
  - The join() method call must be surrounded in a try/catch block.
- Fibonacci2.java → Using ExecutorService
  - Think in terms of tasks, not threads. ExecutorService manages threads for you and helps you avoid the overhead associated with creating and running threads.
  - Faster because the system knows best how to manage its resources based on how many cores it has. It is also able to re-use the same thread for multiple tasks instead of throwing each one away and creating a new one.
  - Use submit() to add a task to the ExecutorService thread pool.
- Fibonacci3.java → Extend RecursiveTask
  - In a recursive setting, we will create many threads and run out of memory because threads have to wait for the recursive calls.

2.2.2 Asynchronous Execution

In asynchronous execution, you don't wait for a task to complete or do the work yourself. You wait for 'someone else' to do the subroutine work, and then continue doing something else. When you need the result from that work, then you wait on the task.

One way to do this in Java is to use the Future type. This type has a class method get(). When called, it blocks if the value is not yet set, and returns the value when done with executing the task.

If you want a return value from a thread, then be sure to implement Callable.

2.3 Amdahl's Law

Question Is there a fundamental limit on how much speed-up I can get in a program?

Solution Let p be the fraction of work that can be parallelized. Then $1 - p$ is the fraction that cannot be parallelized. Let n be the number of cores on the machine. Let T_s be the time taken on a sequential processor. Let T_p be the time taken on a multicore machine. Then:

$$T_p \geq (1-p)T_s + \frac{pT_s}{n}$$

$$speedup = \frac{T_s}{T_p} \leq \frac{1}{1-p+\frac{p}{n}}$$

2.4 Mutual Exclusion

Definition Critical Section - A section of the code that can cause race conditions if the code is interleaved.

An example of this could be the line of code $x = x + 1$; If two threads are both in the process of executing this line of code at the same time, problems could arise. Since this breaks down into a load, write, and store instruction, both threads could load the value 0 and add one, then then store that value. In this case, x would be the value 1 and not 2, which may be the expected value.

We want critical sections to be mutally exclusive. If any processor is in the critical section, then no others should be executing that section.

2.4.1 Peterson's Algorithm

This is an algorithm to implement the mutual exclusion construct. The key in this solution is that it is a 'polite' solution - each thread sets the turn variable to the other thread.

P0

```
wantCS[0] = true;
turn = 1;
while (wantCS[1] && turn == 1);
// CS
wantCS[0] = false;
```

P1

```
wantCS[1] = true;
turn = 0;
while (wantCS[0] && turn == 0);
// CS
wantCS[1] = false;
```

Question Does this satisfy Deadlock Freedom?

Proof A deadlock can happen in this code if both while loops are true at the same time.

$$DEADLOCK = wantCS[1] \;\&\&\; turn == 1 \;\&\&\; wantCS[0] \;\&\&\; turn == 0$$

But, we know that $p \wedge q \Rightarrow p$.

$$turn == 1 \wedge turn == 0$$

But, we know that the turn variable cannot be both 0 and 1. This gives us a contradiction, and thus this code is free of deadlock.

Question Does this satisfy Mutual Exclusion?

Informal Argument Assume both threads are in the critical section. This means that both excuted the assignment statement of the variable turn. Let turn = 1. This means that the assignment turn = 0 happened before turn = 1. But, P0 checks that turn = 1 and $wantCS[1]$ = true. This means that P0 could not have entered the critical section. This is a contradiction.

Dijkstra's Proof Let $trying[0]$ be true when the execution point is at the while statement. Let it become false when entering the critical section.

$$H(0) = wantCS[0] \wedge ((turn = 1) \vee ((turn = 0) \wedge trying[1]))$$
$$H(1) = wantCS[1] \wedge ((turn = 0) \vee ((turn = 1) \wedge trying[0]))$$

Based on the above definitions, when P0 gets to the while loop, $H(0) = true$. When P1 gets to the while loop, $H(1) = true$. Let us show that P0 cannot falsify $H(1)$. Then by symmetry, P1 cannot falsify $H(0)$.

Let us look at each part of the statement in turn.

- $wantCS[1]$ - P0 does not touch this variable, so it cannot make it false.
- $turn = 0$ - This value can be changed to $turn = 1$, but then the second part of the statement will become true. This is because when you change the value of turn, you have reached the while loop. This means that $trying[0] = true$ and $turn = 1$.
- $turn = 1 \wedge trying[0]$ - This cannot be falsified because it is equivalent to falsifying $wantCS[1] \wedge turn = 1 \wedge trying[0]$. The first two predicates in that statement are the same condition as the while loop. Due to the entry protocol, we cannot make $trying[0] = false$.

From here, we can assume that both P0 and P1 are in the critical section, and then manipulate the statements to show a contradiction such as $turn = 0 \wedge turn = 1$. This is left as a exercise.

References

[1] V.K. Garg Introduction to Multicore Computing