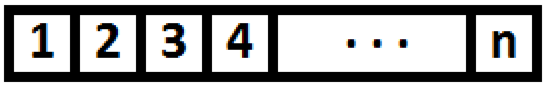
EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 2: August 30

Lecturer: Vijay Garg *Scribe: Doyoung Kim*

2.1 Review of Last Lecture



Let's say there is an array with n unique, unsigned integers. If you want to find the max of array, what kind of algorithm would you use to find the max with fastest time and with least amount of work?

2.1.1 Basic Sequential Algorithm

How about good ol' sequential algorithm? First, assume that integer on index 0 is the max. (Let's say we store the value in the variable called *tempMax*) Then, compare *tempMax* with another integer on the next index. Only if 'another integer' is greater than *tempMax*, simply replace *tempMax*'s value with 'another integer'. Continue incrementing the index until the end. This algorithm has execution time of $O(n)$ and work load of $O(n)$.

2.1.2 Binary Algorithm

Another way is to compare integers by pairs. First, compare integers by pairs; first thread compares index 0 and 1, second thread compares index 2 and 3, and so on. Next, compare greater-of-twos by pairs; if index 0 was greater than index 1 and index 3 was greater than index 2, compare index 0 and index 3. Repeat until you get the max. This algorithm has execution time of $O(log(n))$ and work load of $O(n)$.

2.1.3 All Pair Comparison Algorithm

Now, what if you were given huge amounts of threads, let's say n^2? Now, make threads to check all possible, unique combinations and change the value to 0, if value is less than other value. After one run, there will be only one non-zero value and that is the max. This algorithm has execution time of $O(1)$, but work load of $O(n^2)$.

2.2 Ways to Create Threads in Java

There are multiple ways to create threads in Java:

- Extend **Thread** -> Overwrite method **run()** -> Call method **start()** (ex. Fibonacci.java)
- Implement **Runnable** -> Overwrite **run()** -> Create Thread -> Call **start()** (ex. FooBar.java)

- Implement **Callable** -> Overwrite **call()** -> Use **ExecutorService** class to start (ex. Fibonacci2.java)
- Extend **RecursiveTask** -> Overwrite **compute()** -> Use **ForkJoinPool** class (ex. Fibonacci3.java)

Note that **Thread** and **Runnable**'s **run()** do not return anything, while **Callable**'s **call()** and **RecursiveTask**'s **compute()** do.

ExecutorService object lets program to handle the threads, rather than programmer manually handling the threads. Combination of **RecursiveTask** and **ForkJoinPool** also make program to handle the threads. Unlike **ExecutorService**, **RecursiveTask** enables threads to work on other job as waiting for other value(s).

Threads could be **join**ed and those threads wait until all joined threads complete their tasks. **Future** class could be also used, instead of **join**. **Future** class lets thread to run and stops only when program needs computed value.

2.3 Amdahl's Law

Let's say, we have large tasks to complete. If we can define **p**, a fraction of the work that can be parallelized, we can calculate the limit on speedup for this task by Amdahl's Law. Say, **n** is number of cores, $\mathbf{T_p}$ is time on multicore machine, and $\mathbf{T_s}$ is time on sequential process. By Amdahl's Law, we get following formula:

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

Which can be used to derive function for **Speedup**:

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

2.4 Mutual Exclusion

Suppose there are two threads that are doing same task, using same variable x with initial value of 0 :

$\mathbf{T_1}$	$\mathbf{T_2}$
$x = x + 1$	$x = x + 1$

If those two tasks are executed at the same time, what will x be after the execution?

Answer is 1. Why not 2? It is because this process is not atomic. Because both threads were executed at the same time without variable locking or any check, when they started to run, they both see the initial value of x, which is 0. Both tasks, then, will add 1 to the value they saw, and store it to variable x. Conditions such as this, where more than one threads may read and write on same variable at the same time, is called **critical section**. To avoid *critical section*, code has to be executed atomically.

2.4.1 Door Problem

Assume there are two threads, P_1 and P_2 trying to use same 'door'.

$\mathbf{P_0}$	$\mathbf{P_1}$
loop	loop
—– Entry —–	—– Entry —–
Critical Section (CS)	Critical Section (CS)
—– Exit —–	—– Exit —–
Not Critical Section (NCS)	Not Critical Section (NCS)
Go back to loop	Go back to loop

If two threads are executed at the same time, how can we prevent *critical section*?

How about using a variable to keep track of the door (Ex. Attempt1.java)? In this process, each thread checks whether door is open or not by looking at boolean variable *openDoor*. While *openDoor* is false, wait until it becomes true. If *openDoor* is true, set it to false and then execute code. After all necessary lines are executed, set *openDoor* back to true. This attempt does not work, because threads may check *openDoor* at the same time before any one thread sets *openDoor* to false.

How about using two variables instead of one (Ex. Attempt2.java)? In this process, there is a boolean array *wantCS* and each index of *wantCS* is assigned to all threads (P_0 gets index 0 and P_1 gets index 1). A thread sets its corresponding *wantCS* to true, right before it checks whether other thread wants or not, This attempt does not work, because if both threads set their *wantCS* to true, none of both threads threads will be able to get past while loop. This situation, which all threads are stuck in some part of the code, is called **deadlock**.

How about manually setting which thread goes first (Ex. Attempt3.java)? In this process, P_0 will execute its code first and then let P_1 to execute. After P_1 executes its code, it will let P_0 to execute. This attempt does work, but since threads don't know whether other thread needs to execute or not, there may be unnecessary waiting time.

2.4.2 Peterson's Algorithm

Peterson's Algorithm (Ex. PetersonAlgorithm.java) uses approaches of Attempt2 and Attempt3 (*wantCS* and *turn*). This process gives each thread a turn, but also checks whether that thread wants to execute the code or not.

Check for Deadlock Freedom

$$\begin{aligned} \mathit{Deadlock} &\equiv (\mathit{wantCS}[1] \wedge \mathit{turn} == 1) \wedge (\mathit{wantCS}[0] \wedge \mathit{turn} == 0) \\ &\Rightarrow \mathit{turn} == 1 \wedge \mathit{turn} == 0 \\ &\Rightarrow \mathit{false} \end{aligned}$$

Check for Mutual Exclusion
Let's add another boolean array, *trying*. Only when thread is checking for while-loop conditions, corresponding index of *trying* becomes true. Otherwise, it is false.

Consider predicate H, where:

$$H(0) \equiv \mathit{wantCS}[0] \wedge [(\mathit{turn} == 1) \vee ((\mathit{turn} == 0) \wedge \mathit{trying}[1])]$$

$$H(1) \equiv \mathit{wantCS}[1] \wedge [(\mathit{turn} == 0) \vee ((\mathit{turn} == 1) \wedge \mathit{trying}[0])]$$

In here, P_0 cannot falsify $H(1)$. Because only P_1 can change the value of wantCS[1], $H(1)$ will be falsified only if $\mathit{wantCS}[1] \wedge \mathit{turn} == 1 \wedge \mathit{trying}[0]$ holds true. When turn is set to 1 by P_0, trying[0] is false. From symmetry, P_1 also cannot falsify $H(0)$.

Now, check for the mutual exclusion:

$$\begin{aligned} CriticalSection \equiv \neg trying[0] \wedge H(0) \wedge \neg trying[1] \wedge H(1) \\ \Rightarrow turn == 0 \wedge turn == 1 \\ \Rightarrow false \end{aligned}$$

References

[1] V. K. Garg, Introduction to Multicore Computing, pp. 21-22.