CS-377 Parallel Programming
Final Assignment

Eliot Cowley

May 16, 2015

1 The S'mores Problem

Before starting this problem, my first decision was whether to use Ruby/Rinda or Java. I decided to go with Java since I was more familiar with it and preferred it in general. Having only done the one Producer/Consumer problem in Rinda, I was still a little hazy on how it all worked, and was more comfortable with synchronized methods in Java.

As with most assignments, I found it helpful to first sketch out how I was going to structure my program before doing any actual coding. At first, I was confused as to why I needed four threads–couldn't I just have three threads representing the children waiting to eat, with one synchronizing monitor? But then I realized that I needed a `Chooser` thread that would be waiting to choose ingredients and signaling the correct `Child`.

So I organized my program into four classes: the main `S'mores` class that would initialize the threads and monitor; the `Child` class, which would try to pick up ingredients and make s'mores; the `Chooser` class, which would choose ingredients to put down; and the `Monitor` class, which would synchronize the shared variables and make sure that deadlock and race conditions were avoided.

Having decided on the structure of my program, I began adding fields and methods to my classes. `Child` would have an index between 0 and 2 (I used constants so the program would be scalable), a reference to the `Monitor`, and an `int` representing the number of s'mores it has eaten. As for methods, it would have a `pickUpIngredients` method that would go through the monitor to try to pick up the ingredients off the table. It would also have a `run` method that would be that thread's lifecycle. Finally, it would have a `delay` method that would make the thread sleep for a random number of milliseconds determined by another method, `randomInt`. A `Child`'s lifecycle would consist of trying to pick up ingredients, making a s'more, delaying for a few milliseconds, and then repeating.

The `Chooser` class's fields would include a reference to the `Monitor` and an `int` representing the index of the `Child` he has chosen that turn. It would have a `run` method, as well as `putDownIngredients` and `choose` methods, which would go through the `Monitor`. The life cycle of the `Chooser` would consist of choosing a `Child` who would eat a s'more that turn and putting down the correct ingredients on the table.

The `Monitor` class would have several fields, including: an `int` representing the index of the child who will be making a s'more that turn; a `boolean` representing whether there are ingredients currently on the table; an array of `int`s representing the ingredients on the table; and a constant representing the number of children who will be eating s'mores in order to make the program scalable. It would have a synchronized method `putDownIngredients` which would take the index of the eater as a parameter, and would wait for there to be no ingredients on the table before inserting ingredients into the array, putting down only the ingredients of the two non-eater children. It would also have a synchronized method `pickUpIngredients` which would also take an eater as a parameter, however this method would wait both for there to be ingredients on the table and for the current eater to be equal to the caller thread. There would also be a synchronized `choose` method which randomly chose a number between 0 and 2 and assigned that index as the eater. Finally there would be a `getName` method that took an integer as input and returned a String representing the ingredient of that index. The `Child` and `Chooser` threads would all call their methods through this `Monitor` to avoid deadlock and race conditions.

My solution works because any time a thread needs to access or modify a shared variable, it does so through synchronized methods in a monitor. This way, no two threads are stuck fighting for the same resource. I cannot say with certainty that this solution is fair, however after extensive testing I have found that each child generally eats the same amount of s'mores and no one child frequently eats the most.

2 The Water Molecule Problem

For this problem, I decided that to simplify things, the values that would be passed along channels would be integers. I declared 0 to be `hydrogen`, 1 `oxygen`, and 2 `water`. I made the two `hydrogen` processes and one `oxygen` process continuously output their respective data to both the `water` process and `print.stream` process. The `water` process waits until it has two `hydrogen`s and an `oxygen` before it passes a `water` to `print.stream`. `print.stream` alternates over all four processes, choosing them arbitrarily and writing to the screen when something is produced. It then delays for a certain amount of milliseconds before continuing, so the user can more easily read the output.

The program is run in parallel wherever possible. The `hydrogen` and `oxygen` processes output to the `water` and `print.stream` processes in parallel. The `water` process reads in `hydrogen` and `oxygen` in parallel. Additionally, the main process `waterMolecule` runs the `hydrogen`, `oxygen`, `water`, and `print.stream` processes in parallel. There were a couple of blocks that I couldn't run in parallel, however–for instance, `water` can only output to `print.stream` once it has read in two `hydrogen`s and an `oxygen`, so the output statement was done sequentially; `print.stream` chooses processes in an `ALT` block and then pauses afterwards, which is also done sequentially. These sequential blocks were necessary to ensure the program's accuracy.

There are a couple of things about the output of `print.stream` that seem odd, but make sense. First, sometimes `water` molecules are printed out before two `hydrogen`s and an `oxygen` have been. However, this does not mean that the atoms required for a `water` molecule have not been created yet. In fact, they may have already been created but are just waiting on `ALT` to choose them to print to the screen. Because there is a delay after each print, there is time for `water` molecules to be created, so `ALT` may arbitrarily choose to print out a `water` molecule that is waiting before it prints out the atoms. I proved this by running my program without the pause after printing, and it always printed out at least two `hydrogen`s and one `oxygen` before printing out a `water`. If I wanted to make for more accurate printing, I could have put a delay in the water process to make sure that it gets formed and then printed after the atoms had.

Second, the output always appears to be the same, even though technically `print.stream` is choosing processes "arbitrarily" with `ALT`. There is always a start-up section of output that is different from the rest, which then settles into a pattern of `water`, `oxygen`, `hydrogen`, `hydrogen`. I got a similar result, albeit with different patterns, when I switched the order of the guards. Additionally, when I tried using `PRI ALT`, it eventually settled into choosing processes from highest to lowest priority. The reason for `ALT` choosing processes with seeming regularity is likely because it is being fair, giving each process a chance to execute in order to avoid starvation. Therefore, `ALT` is not *completely* arbitrary, though it never gives unequal preference to any one process. Deadlock is avoided because there is no contention for resources; the `hydrogen` and `oxygen` processes produce integers for both `water` and `print.stream` equally. For the above reasons, my solution is correct.