Kenta Hasui

CS377: Spring 2015

Final Assignment Report

Part 1: S'Mores

Running the simulation and solution:

ruby *server.rb*

ruby *chooser.rb*

ruby *choc.rb*

ruby *marsh.rb*

ruby *graham.rb*

Language Choice

I chose to implement a solution to the Smores problem using Ruby/Rinda. In my opinion, Rinda tuples are a much more elegant and intuitive implementation of semaphores than Java synchronized methods/objects. Java has many different useful applications, but unfortunately concurrency is not its strongest suit. Ruby is a very powerful language that can handle scripting as well as object-oriented programming. Coupled with Rinda's robust tuple space support, Ruby was the obvious choice to code this solution in.

Tuple Space Setup and Semaphores

Unlike the bounded buffer problem, I did not need to set up the environment other than starting the Rinda server and instantiating the tuple space. The child processes do not need to

wait on a condition such as “empty”, as the producer process did in the producer/consumer or bounded buffer problem. I included several different semaphores for this problem:

Ingredient Semaphores: These represented the ingredients that are placed on the table to make Smores. They are implemented as three different tuples: [:Ingredient, “Marshmallows”], [:Ingredient, “Graham Crackers”], and [:Ingredient, “Chocolate]. All three semaphores have the same ‘:Ingredient’ symbol tag. These can all be considered binary semaphores.

Ready and *Finished Semaphores:* These semaphores allow messages to be passed between the chooser child and the eating children. Each child is uniquely identified by a string representing their supply of ingredients (“Marshmallows”, “Graham Crackers”, “Chocolate”). The chooser places two ingredients on the table, then puts a [:Ready, String] tuple into the tuple space. The String is one of the three ingredients. The chooser chooser then waits (blocks) for a [:Finished] tuple to be written into the tuple space. This only occurs when a child with ingredients takes the two other ingredients from the table (tuple space), creates a smore, eats it, and writes a [:Finished] tuple.

Files and Classes

server.rb: Starts up the server

chooser.rb: Creates a chooser process. It runs for 90 iterations (because 90 is divisible by 3). The chooser chooses an ingredient NOT to put on the table (implemented using Ruby’s built-in

rand() function). The chooser then places the other two ingredients on the table (by writing two Ingredient tuples to the tuple space) and signals to the appropriate child that they can start eating (by writing a Ready tuple to the tuple space). It then waits for the child to finish eating (by trying to take the Finished tuple from the tuple space).

eater.rb: Because all three children who are eating do almost the same exact thing, I created a Eater class to each of the three children. To instantiate an Eater object, we give it an ID representing the ingredient the child has: "Marshmallows," "Chocolate," or "Graham Crackers." The Eater object's main method is called eat_smores. This method loops forever, waiting (blocking) until it reads a Ready tuple with its ingredient from the tuple space -- [:Ready, "Marshmallows"], for example. It then prints messages indicating that it is eating a smore and prints out how many smores have been eaten so far. It then places a [:Finished] tuple in the tuple space to tell the chooser process that it is finished, and starts the loop again.

marsh.rb, choc.rb, graham.rb: Instantiations of the Eater objects. Simply creates a eater object and runs its eat_smores method.

Correctness

By making the chooser wait until a child is finished before choosing another ingredient, we prevent the chooser from placing more than one Ready tuple/semaphore in the tuple space at any given time and we prevent the eater files from placing more than one Finished tuple in the

tuple space at any given time. Thus we ensure that only one child will be making a smore at one time. In this manner we prevent race conditions for the shared candle resource through this passing-the-baton strategy.

Because the chooser.rb process will always start by placing a [:Ready] semaphore into the tuple space, we are guaranteed that one of the three children will stop blocking and execute its eat_smores body. Thus the only way that these processes will deadlock will be when there is faulty execution on the part of the user: they don't run one of the processes. If chooser.rb is not run, no child will ever be signaled. If one of the three eater processes is not run, then the chooser could signal to a child who is not executing. But if all the files are running, then these processes will not deadlock. Circular waiting and hold-and-wait do not hold simultaneously at any point in this program, so deadlock cannot occur. When the eater is in its critical section, there MUST be two Ingredient semaphores in the tuple space because the chooser places them there BEFORE it places the Ready semaphore. If the eater tries to enter its critical section before the chooser process places these two ingredient tuples, the eater will be blocked. Similarly, if the chooser tries to choose two new ingredients to place on the table before the previous child has finished eating, the chooser will be blocked.

The chooser may not choose truly randomly. It uses Ruby's built-in rand() method, so its true randomness is restricted by the implementation of the method. Starvation could technically happen for one of the eater processes if they never get chosen. But this is unlikely, given the pseudo-random nature of the choosing process. Rand() seems to have pretty good results in terms of distributions of smores to the three children: one child does not always eat all of the smores.

Part 2: Water Molecule

Running the simulation and solution:

To compile: kroc watermolecule.occ -lcourse

To run: ./watermolecule

Processes:

Hydrogen: A process that simulates the production of hydrogen atoms. It is made of one prefix process and two delta processes. While its output channels are ready to accept values, it continually outputs the integer value 1. Due to all processes being composed in parallel, there is no risk of deadlock within the Hydrogen process itself.

Oxygen: It is exactly like the Hydrogen process, but outputs the integer value 2 instead of 1. This was accomplished by setting the initial prefix value to 2.

Water: This process has three input channels: h1, h2, and o. These represent inputs from two hydrogen processes and one oxygen process, respectively. Water reads in values from the channels (in parallel, using three different variables to store them into) and then outputs the value 3. This simulates the three atoms being combined to create a water molecule. The three "read" operations are composed in parallel, so the inputs can be received in any order and still avoid deadlock.

print.stream: This process is a multiplexer: it reads from multiple channels (h1, h2, oxygen, and water) and prints to a single input channel. These channels represent inputs from two hydrogen processes, one oxygen process, and one water process. This process has local variables that count the number of each type of atom produced. The process initializes the three counter variables to 0 (in parallel) and then loops forever. It ALTs over its four read channels, printing each value as it becomes available. Unfortunately I was not able to compose the print statements in parallel, as changing the order of the printing would lead to gibberish output. However this process will not deadlock thanks to the ALT: as long as one of its four input channels have a value available, it will be able to execute its body. The SEQs within the alt will not cause deadlocks, as they only manipulate local variables and produce an output. They do not depend on any outside inputs or synchronization, so we avoid a hold-and-wait situation.

main: This process is the main process network. It is composed of two Hydrogen processes, one Oxygen process, one Water process, and one print.stream process. These processes are composed in parallel. No two processes read to the same channel and no two processes write to the same channel, so we avoid race conditions.

Freedom from Starvation:

The Hydrogen and Oxygen processes start the main process by writing values to their output channels. Print.stream receives these values and prints out the appropriate messages. The Water process also receives these values and writes its own value to print.stream. These processes cannot become starved because the channels are synchronous. For each of the

hydrogen or oxygen processes to keep producing data, both output channels must be ready. This means that the data must be used by both the Water and print.stream processes. So after the first one, a hydrogen or oxygen atom cannot be produced until a water molecule is produced and a message is printed to the output. Water cannot be produced until two hydrogen atoms and one oxygen atom is produced.

Freedom from deadlock:

I have already explained in the "Processes" section how each individual process avoids deadlock. The main process' network is composed in parallel, preventing circular waiting from becoming a problem. The order in which the outputs are produced does not matter. Each process will become blocked until the output channels are ready to be read into. The main process starts by the oxygen and hydrogen processes producing values, and will loop forever.