Adam Ryder

Operating Systems

Dr. Georgiev

Operating Systems Final Project – Producer Consumer

For my project I decided to work on the producer-consumer problem. This is a very well-known problem for concurrency so I decided to tackle it. It was difficult to design, but once I had everything planned out, it turned out to be quite enjoyable. The hardest part was setting up the server and getting the clients to talk to it successfully. After designing the server and client classes and getting them up and running, I decided to work out how they were going to interact with the buffer. I knew the buffer had to be synchronized correctly with the clients and producers or else I would have concurrency issues. So, I decided to use the wait() and notify() synchronized methods.

I used five classes, a consumer(client), producer, consumerMonitor (driver), ProducerServer (server), and Buffer. The buffer was a basic array full of integers to signify the resource buffer. The producerServer created the server and spawned the appropriate number of producers as declared by the user. Once a producer was spawned it would start to fill up the buffer with resources (integers). The consumerMonitor asked the user how many producers, consumers, and the size of the buffer. After that, the consumerMonitor would spawn the appropriate number of consumers. Once a consumer spawned it would establish a TCP connection with the server. Once a connection is established, the server would create a handler thread that would handle all interactions between the consumer (client) and server. The consumer would send messages to the server asking to consume a resource. The handler of the server would then interact with the buffer to consume a resource.

After implementing the wait and notify correctly within the buffer class I noticed that the output wasn’t what I was expecting. I wanted the consumers to be able to consume as many resources in a row without being interrupted by a producer, but that wasn’t what was happening. Every time a consumer took a resource from the buffer, the buffer would notify a producer and the producer would immediately replenish the buffer. So it was like the producers and consumers were taking turns accessing the buffer. This wasn’t really what I was looking for so I decided to look around for another way to solve this problem.

After some research I decided to use a blocking queue in the java concurrency package. This method was so easy to work with. I was amazed at how simple this idea was and immediately fell in love with it. I basically turned my buffer class into a blocking queue and kept all the other classes the same. This got rid of almost 100 lines of code. After implementing it into my code and working out all the bugs, my program starting running the way I wanted it to. The blocking queue uses only two methods to access the buffer, which is the take() and put() methods. These methods are much more safer because you don’t have to check to see if the queue is empty or full before using them. For example, when using the take() method, if there is nothing in the queue to take, the thread will automatically wait until there is something available to take. Same thing with the put(), if there is no room in the queue to put a resource, then the thread will automatically wait for room to become available.

So after working on this project, if I ever had to work on a concurrency problem I would most definitely use a blocking queue. The blocking queue is very efficient and easy to work with and is the best solution to avoiding dead locks or concurrency issues.