

Spring 2007 Operating Systems: Concurrency Java Candy Factory Simulation

Problem statement: The purpose of this exercise is to simulate a candy factory. Three ingredients are needed to make a candy bar: Chocolate, Nuts, and a Wrapper. Assume that there are three machines (Clients) that can make candy bars. Client 0 already has an unlimited supply of Chocolates; Client 1 already has an unlimited supply of Nuts; Client 2 already has an unlimited supply of Wrappers. Each Client takes a random amount of time to make a candy bar as soon as it has all three ingredients.

In addition, there is a machine called Agent with an unlimited supply of all three ingredients. The Agent puts two of the ingredients on a Table at the same time for any Client to use to make a candy bar. The Agent decides at random which two of the ingredients that it will put on the Table. Of course, if it puts down Nuts and Wrapper, only Client 0 would be able to use them. But as soon as a Client can use ingredients, it takes them (again, *at the same time*) and the Agent is immediately free—without waiting for any Client to finish!—to put down two other ingredients.

Write a Java program to use threads to simulate the activities of the Agent and the Clients. Do not assume that the machines cooperate.

For extra credit: Solve the problem for n Clients instead of three. In this case, Client number i would have the same infinite supply as Client number $i\%3$. For example, Client 23 would have an infinite supply of Wrappers, because $23\%3 == 2$, and Client 2 has Wrappers.

Background: This problem appears in a wide variety of forms in distributed processing. Here are three examples. Historically the Harpy and Hearsay projects' approach to speech understanding at Carnegie-Mellon University used an architecture similar to this. The Table then is called a Blackboard, and the Clients can remove stuff from the blackboard, but they can also read stuff from the blackboard without removing it, and then like our Agent they are actually allowed to put things on the blackboard. The Clients have specialized responsibilities in phonetics (sound processing), in syntax (parsing), in semantics (meaning), and in pragmatics (context of use). The 2004 state of the art about this approach is described by Deng and Huang¹.

Google uses a specialized file system with high redundancy for speed.² One Client (or group of Clients) has an unlimited supply of web pages; another Client (or group of Clients) has an unlimited supply of user requests; a third Client (or group of Clients) has an unlimited supply of updates to the data. Multiple Agents delegate responsibilities to these Clients, not in a deterministic way, but in a “first one ready gets the job” fashion.

Although Amazon.com is much more secretive about its system architecture than Google, you can imagine similarly an unlimited supply of books, of customer requests, and in this case a database of historical data, both about overall usage of the site in general, and usage by the specific customer in particular. Agents delegate to Client processors the work load based on which Client has most ready access to the specific resources needed.

Why did I capitalize several things in this question? As a hint. Some of those things are things that are likely to be Java classes in your simulation.

Distributed: Monday, March 26, 2007; **due: Wednesday, April 18, 2007.** (One week later than syllabus says, because administrative confusion about when School of MEB Scholarship Day would be resulted in my cancelling the final project in this course, which could not possibly be done in time for the week-earlier Scholarship Day.)

Your solution must meet the following **four specifications** for any well-tuned multi-threaded system:

- S1) Each participant must be simulated with a separate **Java thread**. The client 0, 1, and 2 threads may be three different classes, since each must have a different resource—or they may be taken from a single Client class, each object instance of which just happens to know what resource it needs. You'll need an Agent class. You'll need a driver class to gather information from the user and launch all the action. Each thread should

¹ Li Deng and Xuedong Huang, “Challenges in Adopting Speech Recognition,” *Communications of the ACM*, 47, 1, January 2004. Pp. 69–75. See especially p. 72.

² Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System,” <http://www.cs.rochester.edu/sosp2003/papers/p125-ghemawat.pdf>. Last accessed Jan. 26, 2007.

display what it is doing.

- S2) **Mutual exclusion.** Of course, you should not allow multiple clients to consume the same resources!
- S3) **Progress.** The clients should behave as independently and as generally as possible, requiring as little knowledge as possible about the other clients or about the Agent. Clients and Agent should communicate only through their shared resources. In that way, progress can be made. No scheduler should queue up waiting clients wanting to use a resource behind a client who doesn't want the resource. Turn taking must be based on the availability of resources, not on polling the participants in some pre-determined order. [I would solve the problem using one semaphore per resource. Java's `synchronized` uses built-in semaphores associated with classes, so you don't need to "roll your own" semaphores.]
- S4) **No starvation.** Since there are multiple clients who want the same resource, your solution should be fair to all the clients. Don't starve one of them. This implies **no deadlock** since two deadlocked threads are surely starving.

You must meet the following four criteria, and **DEFEND the fact that you meet these four criteria by answering all of these questions.**

C1) Did you solve the basic problem (70 points) with readable source code including comments with a good interface to the world showing what your solution is doing? Discuss the following:

- a. How did you enforce mutual exclusion in the use of the chocolate, nuts, and wrapper? For example, we don't want a client to pick up one of the items, then to have the agent replace both of the items with a pair of items only one of which is different, and then have another client go after that new pair before the first client has even had a chance to pick up the second item, which by now may be obsolete.
- b. How did you prevent deadlock? Which of the four criteria for deadlock does not hold, and how do you know? If the agent replaces things as soon as they are taken, could two clients both be after the same item?

C2) Did you implement a general solution (15 points)?

- a. How independent are the clients from each other and from the agent?
- b. How functional (general and complete) is your solution?
- c. (Extra credit) Does your solution work for N clients?

C3) Performance criteria (10 points).

- a. Fairness. Did you prevent indefinite postponement (starvation)? Argue that you did or didn't.
- b. Did you avoid busy waits? Argue that you did or didn't.

C4) Miscellaneous factors (5 points).

- a. How aesthetically pleasing is your solution? (Is your code elegant?)
- b. Do the clients and agent behave realistically? (The problem statement suggests that you use random numbers, as some Java examples in Chapter 7 do, to allow the times for one step in manufacturing a candy bar to vary.)
- c. How robust is your solution? Does it easily adapt to new situations? (Would multiple agents, for example, be easy to add, or varying numbers of clients of each kind?)
- d. How innovative is your solution? Did you try graphics instead of text? Did you document all outside sources? If you discuss this with other classmates or find code on the internet, did you credit your ideas and code? If this problem looks like one of the three classic problems which we discussed in class (bounded buffer, dining philosophers, or readers-writer), then Java code on the web might be of aid in solving this.