COEN 346 OPERATING SYSTEMS – WINTER 2023

Programming Assignment # 3: OS Simulator for the Producer-Consumer Problem

Deadline: April 4th, 2023

Objectives:

Develop a practical understanding of inter-process communication and synchronization through the implementation of a producer-consumer problem using sockets in Java.

Description

In this assignment, you will implement a simulation of the classic producer-consumer problem using Java sockets (see the end for a definition). The simulation will involve multiple clients acting as producers or consumers, and an OS server that will schedule their execution. A process can act as either a consumer or a producer. Consumers read items from the buffer. Producers store items in the buffer. The buffer is bounded, so there is a limited number of items that can be stored. The processes run indefinitely. You need to ensure that there are no race conditions while accessing the buffer, by creating the necessary classes or instructions.

The base code consists of five Java classes: ProcessServer, OSSimulator, Buffer, Process, and ProducerClient.

- The ProcessServer class creates a server socket that listens for client connections. When a client connects, the server creates a new process and adds it to a queue.
- The OSSimulator class extends the Thread class and implements a scheduling algorithm that selects the next process to execute.
- The Process class represents a client process that communicates with the server over sockets. Finally, the ProducerClient class is a client that sends requests to the server to produce items.

Processes can run one of the following instructions:

- Process.NUM_ITEMS: Requests the number of items currently in the buffer.
- Process.NEXT_ITEM: Requests the position of the next item to be read from the buffer.
- Process.GET_ITEM_POS: Requests the next item from the buffer.
- Process.TERMINATE: Informs the OS to terminate the process.

In the current implementation, clients (producers and consumers) communicate with the server (OS simulator) using a simple text-based protocol. The protocol consists of the following steps:

- When a client (producer or consumer) connects to the server, the server assigns it a unique ID and sends the ID back to the client.
- The client waits for a "RUN" signal from the server.

- Once the client receives the "RUN" signal, it sends a request to the server for an operation to be performed on the shared buffer. The client can send one of four requests: "getNumItems", "getItemInPos", "getNextItemPos", or "terminate".
- The server performs the requested operation and sends the result back to the client.
- The client repeats steps 3-4 as necessary, until it is done producing or consuming.
- The client sends a "terminate" request to the server when it is finished.
- The server removes the client from its queue of active processes.
- This protocol ensures that multiple clients can communicate with the server without interfering with each other, as each client is assigned a unique ID and communicates with the server sequentially.

Your tasks:

1. Run the ProcessServer class and the ProducerClient class and understand how they communicate. Run multiple clients: to do so on IntelliJ you need to:
   Click Run->Edit Configurations. For the Producerclient file, click on Modify options and select **Allow multiple instances**. Click on Apply.
   You can also run the server on one machine and the client on a different one. To do so, you will need to change the client code
2. Make the necessary modifications to assign a unique ID to each process. You can use the process id allocator from programming assignment # 1.
3. Modify the ProducerClient class and Implement the ConsumerClient class.
4. Modify the Process class: You will need to modify the Process class to handle requests from both the ProducerClient and the ConsumerClient. The class should have separate methods to handle requests to produce and consume items from the buffer.
5. Modify the OSSimulator class: You will need to modify the scheduling algorithm in the OSSimulator class to select the next process to execute using a first come first served algorithm. You can use programming assignment # 2 for this.
6. Make the necessary modifications to use synchronization to avoid race conditions when adding or removing items from the buffer. You may add new instructions to the process class.
7. The ProcessServer class can remain unchanged as it only accepts client connections and schedules them for execution.
8. Make all other necessary modifications so that you can ensure safe access to the buffer: consumers wait if there are no items, producers wait if the buffer is full. Finally, the count variable should not be in inconsistent states.
9. Test your implementation with multiple producer and consumer clients running concurrently. Verify that the buffer remains consistent and that there are no race conditions.

BONUS (+10 points): Let the user select the operations that the client will send to the server by typing them in the console.

Why we are using sockets:

Definition: A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.
Sockets provide a simple and efficient way to communicate between different processes running on different machines.
In this simulation, the server is simulating an operating system and the clients represent processes that run on this operating system. The clients send instructions to the server to execute and the server has an image of each process (that also enables the communication with them). The server will listen on a specific port for client connections, and clients will connect to the server using a socket. Once the connection is established, the clients can send requests to the server and receive responses over the socket. This makes it easy to implement the producer-consumer simulation while controlling the number of instructions a process runs on its time slice.
You can implement any synchronization technique to avoid race conditions, such as locks or semaphores. You will need to add the instructions required for this.