EE 360P: Concurrent and Distributed Systems
Assignment 3

Instructor: Professor Vijay Garg (email: garg@ece.utexas.edu)

Deadline: March 7, 2021

This homework contains a theory part (Q1-Q2) and a programming part (Q3). Both the parts must be uploaded to the canvas before the deadline. The assignment should be done in teams of two.

1. **(10 pts)** Some applications require two types of accesses to the critical section – *read* access and *write* access. For these applications, it is reasonable for multiple read accesses to happen concurrently. However, a *write* access cannot happen concurrently with either a *read* access or a write access. Modify Lamport's mutex algorithm for such applications.

2. **(10 pts)**

   (a) Extend Lamport's mutex algorithm to solve k-mutual exclusion problem which allows at most k processes to be in the critical section concurrently.

   (b) Extend Ricart and Agrawala's mutex algorithm to solve the k-mutual exclusion problem.

3. **(80 pts)** The goal of this assignment is to learn client server programming with TCP and UDP sockets. You are required to implement a server and a client for an online book system of a library. The system should function with both TCP as well as UDP connections. The server has an input file which represents all books this library has. There is a single server, but multiple clients may access the server concurrently. The server must be multithreaded. Every client accepts only the following commands:

   (a) `set-mode t|u` – sets the protocol for communication with the server. The protocol is specified by the letter parameter `u` or `t` where `u` sets the communication mode to UDP, and `t` sets the communication mode to TCP. The default mode of communication is UDP. The server responds with the message:'The communication mode is set to TCP' or 'The communication mode is set to UDP'.

   (b) `begin-loan <user-name> <book-name>` – inputs the name of a user and book. The client sends this command to the server using the communication protocol given by the current mode. If the library has all the copies of this book lent out, the server responds with the message: 'Request Failed - Book not available'. If the library does not have the book, the server responds with message: 'Request Failed - We do not have this book'. Otherwise, the borrow request succeeds and the server replies with a message: 'Your request has been approved, `<loan-id> <user-name> <book-name>`'. The output parameter `<loan-id>` is a unique identifier to the loan generated by the server each time a book is borrowed. A sensible approach is to have the server assign the first loan an identifier of '1' and increment each time a loan is completed. The server should also update the library's inventory of available books.

(c) `end-loan <loan-id>` – return the book associated with the loan identified by `<loan-id>`. If there is no existing loan with the id, the response is: '`<loan-id>` not found, no such borrow record'. Otherwise, the server replies: '`<loan-id>` is returned' and updates the inventory.

(d) `get-loans <user-name>` – list all active loans associate with a user. If no loans are found for the user, the system responds with the message: 'No record found for `<user-name>`'. Otherwise, list all records of active loans for the user as `<loan-id> <book-name>`. Note that, you should print one line per loan record.

(e) `get-inventory` – lists all available books in the library. For each book, you should show '`<book-name> <quantity>`'. Note that, even if there is no copies left, you should print the book with quantity `0`. In addition, you should print one line per book.

(f) `exit` – inform server to stop processing commands from this client and print inventory to inventory file named as "inventory.txt." If there is an existing file with that name then that file is overwritten.

You can assume that the servers and clients always receive consistent and valid commands from users. The server reads the initial book inventory from the input file. Also, the book name is sufficient to uniquely represent a book and every user name is just a single word. Each line in the input file shows the name of the book and its quantity. You are provided with a sample command file for a client in which each line contains a command that needs to be executed. All the return messages after each command should be written into the file named **out_clientID.txt** in the current directory, e.g, client 1 should write to out_1.txt. If there is no return message from server, don't output anything. The following shows an example of the input file, command file and output file

---

input file:
"The Letter" 1
"The Great Gatsby" 15
"Divergent" 10
"The Count of Monte Cristo" 17

command file:
set-mode t
begin-loan Mike "The Letter"
begin-loan Jack "The Letter"
begin-loan Lucy "Divergent"
get-loans Mike
end-loan 1
get-inventory
exit

output file:
The communication mode is set to TCP
Your request has been approved, 1 Mike "The Letter"
Request Failed - Book not available
Your request has been approved, 2 Lucy "Divergent"
1 "The Letter"
1 is returned

"The Letter" 1
"The Great Gatsby" 15
"Divergent" 9
"The Count of Monte Cristo" 17

inventory file:
"The Letter" 1
"The Great Gatsby" 15
"Divergent" 9
"The Count of Monte Cristo" 17

Note: the book name includes quotation marks. Your program should run this way: on one terminal, run **java BookServer input-file** as the server program. Open other terminals to run clients. For the client, run **java BookClient command-file clientId**. The output file should be **out_clientId.txt**. For example, client 1 will run **java BookClient command-file1 1**. Its output will be **out_1.txt**. Client 2 will run **java BookClient command-file2 2**. Its output will be **out_2.txt**. For ease in grading, please keep your server program and client program in the same directory. The server program must be named BookServer and the client program must be named BookClient. After executing your server and clients program, the following files should be generated in that directory: one inventory.txt file for the server and one out_clientId.txt file for each client. There will be two types of test cases: sequential and concurrent. For sequential test cases, only one client(client 1) executes commands. We will compare your out_clientId.txt file(out_1.txt for client 1) with our expected one. For concurrent test cases, two clients will execute commands concurrently. We will only compare your final inventory.txt with our expected one. We will just use diff command to compare your output file and the expected one, so please be careful with your output file format. Please zip and name the source file as **EID1_EID2.zip**. No package name in your source file. Make sure you test your program on a Linux machine (for example the ECE Linux servers – see `https://wikis.utexas.edu/display/eceit/Connecting+to+Linux+Application+Servers` for connection instructions).