Operating Systems

Mini Report

This project written in C++ simulates a classic bar interaction between students and a bartender. Students take from a barrel of beer until that store is empty. Once the store is empty, the bartender is awoken to refill the beer barrel. There are certain restrictions. The barrel can hold up to fifty beers; after taking beer, students drink and think for a random amount of time before going back for more beer; the bartender will awake only three times.

This simulation can only be effectively and efficiently executed using a multithreaded approach. Where multiple student threads run the student functions while a single thread plays the bartender role.

The commands to run the simulation are listed below:

Open a terminal window

“cd” to the project directory

Build command: g++ -pthread Project.cpp –o Project

Run command: ./Project

Pause command: Enter

Resume command: Enter

Exit command: Ctrl+c

**The student thread**

The student thread will run two main functions: get\_serving and drink\_and\_think;

**get\_serving()**

The get\_serving function is designed to simulate taking beer from the barrel. The student thread will decrement the amount of light beer stored in the barrel by a randomly generated number as long as its capacity is more than 0. If however, the student requests more beer, than what is available they will get only what is available. Because we will have multiple threads - students and the bartender – accessing a shared resource – the barrel – this could give rise to concurrency issues such as, race condition or deadlock. To ensure synchronisation of the threads we will use three multithreading techniques and algorithms.

The first is mutual exclusion. Mutual exclusion is a technique that allows us to lock certain resources thereby ensuring only one thread has access to it at any given time. We will use the C++ mutex type for this lock.

The second technique used is bounded waiting. This algorithm puts a cap on the number of times a thread will allow other threads to execute their critical section before it gets a chance to do so. With bounded waiting we track the number of ‘cycles’ that a thread has been waiting for. If that number exceeds a predetermined max we steer execution to the starving thread. In this application we set a limit of two times the number of threads that exist. Before the execution of the critical section of any student thread we check the wait time of all threads. If there is a thread with such a wait time and that thread is not the thread currently trying to execute the critical section, execution will be passed. This will repeat until the thread that is being starved gets a chance to execute. If, however, the current thread is cleared to execute, we increase the wait count for all other threads and reset the wait counter for the current thread to zero.

The third technique, progress, is inherent in the design of the critical sections. The mutex lock is only used to lock the critical section. This means that no code exists outside the critical section that would block its execution.

In the case of the barrel being empty the function will trigger a condition\_variable that has an attached mutex which will pause all student threads until the refill\_barrel() function is executed. This is bars the student thread from the beer barrel resource while the bartender thread works thereby avoiding dead lock. Once a student finishes drinking the process is considered terminated.

**drink\_and\_think()**

Once the get\_serving function successfully takes beers from the barrel it puts the thread into the *Drinking-thinking* state. This allows the drink and think function to run.

The drink\_and\_think() is a simple one. It simulates the time taken for a student to drink and their beers and think by simply sleeping the thread. The sleep time is randomly generated to give a more real world feel to the simulation.

**Data structures**

The application uses a class (Student) to track a student thread. The std::vector data structure is used to store the list of Student threads in the simulation. This vector is iterated over to start all student threads after the application is run. It is iterated over again to display the Process control block of each thread when the application is pause.

**The bartender**

The bartender thread is a much simpler one than student. It is stored in a Bartender class. The bartender thread uses the same mutex lock as the student thread to ensure that the barrel resource is only being used by a single thread at a time. A condition variable keeps track of whether the barrel is empty. An empty barrel will trigger a pausing of all students and a waking of the bartenders thread running the refill\_barrel function. Once the barrel has been refilled the bartender thread releases the lock.