Seventh Assignment Report

<Sara Akbarzadeh><401222155>

CalculatePi

The program divides the calculation of the series into batches of terms, with each batch This code calculates the value of pi up to a specified number of decimal places using a multithreaded approach. It first creates an ExecutorService to manage the threads and divides the calculation into batches. For each batch, it creates a Callable object to calculate the sum of a portion of the pi series. The results of each Callable are stored in a List of Future objects. After all batches have been submitted, the thread pool is shut down and the results are combined to calculate the final value of pi. The calculated pi is then rounded to the desired precision and returned as a string.

The algorithm used to calculate pi in this code is the Leibniz formula for pi: pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + ...). The formula is approximated by summing the terms of the series up to a specified number of terms. The terms are calculated in the PiTask class, which implements the Callable interface. Each instance of the PiTask class is responsible for calculating the sum of a portion of the series. The start and end values of the portion are passed as parameters to the constructor of the PiTask class.

PrioritySimulator

The colors are represented by BlackThread, BlueThread, and WhiteThread. The goal is to execute all the BlackThreads, then all the BlueThreads, and finally all the WhiteThreads.

The program uses the CountDownLatch class to synchronize the execution of the threads. The blackLatch and blueLatch objects are used to signal the completion of the BlackThreads and BlueThreads, respectively. The addToList method is called by the threads to add a message to a shared list and count down the corresponding latch.

The ColorThread class is an abstract class that defines a printMessage method that takes a Message object and prints it along with the name of the current thread. The getMessage method is an abstract method that is implemented by the subclasses to return a message specific to each color.

The BlackThread, BlueThread, and WhiteThread classes are subclasses of ColorThread. They implement the run method to call the printMessage method with a Message object specific to each color.

The Runner class has a run method that takes the number of threads for each color as arguments. It creates and starts the BlackThreads, waits for them to complete using the blackLatch, creates and starts the BlueThreads, waits for them to complete using the blueLatch, creates and starts the WhiteThreads, and waits for all the threads to complete using the join method.

Finally, the Runner class has a getMessages method that returns the list of messages added by the threads.

Semaphore

The provided code shows an implementation of a synchronization problem where multiple threads (operators) need to access a shared resource with a maximum of two threads accessing it concurrently. The code uses a Semaphore to solve the problem.

The main method creates five operator threads and starts them. Each operator thread acquires a permit from the Semaphore before accessing the critical section, which is represented by the accessResource() method in the Resource class. Once an operator has finished accessing the resource, it releases the permit back to the Semaphore.

The Semaphore is initialized with two permits, which means that at most two threads can enter the critical section concurrently. If more than two threads try to acquire a permit at the same time, they will be blocked until one of the threads releases a permit back to the Semaphore.

The Operator class extends the Thread class and overrides the run() method to implement the behavior of the operator thread. The Resource class represents the shared resource that the threads need to access.

Overall, the provided code demonstrates an effective use of Semaphores to solve a concurrency problem where multiple threads need to access a shared resource with a limited capacity.