SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

[CS205] Operating Systems Concepts with Android

Assignment #3

Purpose of this exercise

Upon successful completion of this exercise students should be able to do the following:

1. Perform multi-threaded programming with the Java programming language.
2. Apply concepts of concurrency with appropriate protections of synchronization mechanisms and mutual exclusion.

Overview

Some students claim that I'm Kim Korean BBQ restaurant at School of the Arts located near our school is the best Korean buffet around! We're not sure about that (you have to judge for yourself!), but nothing will stop us from mixing kimchi and japchae with a bit of Java!

In this assignment you will develop a Java-based simulation of such a restaurant to explore multi-threaded programming and operating system concepts such as *concurrency*, *synchronization*, and *mutual exclusion*.

Your program will simulate the interactions between waiters, chefs, and the main thread - the restaurant. The whole endeavour will be controlled by a set of configuration parameters provided in a file (see the sample configuration file provided). The program must record its activities in a semi-structured plain-text log file (see the sample log file provided) to enable automated testing.

Requirements

The objective of this assignment is twofold.

Firstly, you must demonstrate competency in implementing multi-threaded Java programs that apply discussed concepts of *concurrency*, *synchronization*, and *mutual exclusion* to simulate a restaurant environment in line with the requirements stated below in a way that allows your programs to avoid *race conditions* and *deadlocks*.

Secondly, this assignment lets you practice reading file-based configuration, and writing semi-structured plain-text logs from Java programs.

Main thread

Performs the following operations:

- Reads a configuration file.
- Initializes *Order placement queue* and *Prepared order queue*.
- Starts worker threads: *waiter threads* and *chef threads*.
- Coordinates execution and graceful termination of all started threads.

Worker threads

The program creates two types of worker threads:

- *Waiter threads* - as many threads as the configuration value `Number of waiters`. Each:
    - Places orders by taking `Time of placement of a single order (milliseconds)` before adding them to *Order placement queue*.
    - Retrieves prepared orders from the *Prepared order queue* and serves them taking `Time of serving of a single order (milliseconds)`.
- *Chef threads* - as many threads as the configuration value `Number of chefs`. Each:
    - Retrieves placed orders from the *Order placement queue*.
    - Prepares orders taking `Time of preparation of a single order (milliseconds)`.
    - Adds prepared orders to the *Prepared order queue*.

Queues

The program creates two queues:

- *Order placement queue* - a thread-safe queue for waiters to place orders and chefs to retrieve them for preparation. The capacity is the `Size of the order placement queue`.
- *Prepared order queue* - a thread-safe queue for chefs to add prepared orders and waiters to retrieve them for serving. The capacity is the `Size of the prepared order queue`.

Additional considerations

While creating the program, make sure to implement also the following requirements:

- **Logging** - each action (`Order Placed`, `Order Prepared`, `Order Served`) must be recorded in a log file.
- **Synchronization** - all operations must use appropriate mechanisms to facilitate thread-safety: avoiding race conditions and deadlocks, and ensuring mutual exclusion.
- **Termination** - the simulation should run according to the configuration and terminate gracefully once all of the `Number of orders` are completely processed: prepared and served.
- **Single file implementation** - the entire solution must be implemented in the Java programming language (JDK17 or newer) as one or more classes all residing in a single source file *restaurant.java*.
- **Documentation** - at the top of *restaurant.java* include an extensive documentation comment that justifies your design choices, focusing on concurrency controls and implementation of used synchronization mechanisms. Additional comments in the source code are encouraged.

Instructions

Submission

To complete the assignment, write and test Java code implementing the aforementioned requirements. Do not forget the documentation comments! Your code must not contain any unnecessary external references, or third-party libraries (standard headers are permitted), and compile in a typical Java environment (JDK17) with the following compilation flags:

```
javac -d ../bin restaurant.java
```

For your reference, you can review the `build.sh` scripts provided with the examples from the in-class laboratory exercise.

Submit the code as a single *ZIP archive*. It must contain only the source code file named *restaurant.java*, and no other files. The code must be compliable without warnings, and stable in execution to get full marks for the assignment.

Grading

This assignment will contribute **10%** to the overall course grade. All students are required to submit their solutions individually before the due date indicated in eLearn.

Assessment criteria are the following:

1. **Functionality** (50%): a correct implementation meeting all the requirements.
2. **Log accuracy** (30%): a log file reflecting a correct sequence of actions and adherence to concurrency principles.
3. **Code quality** (10%): readability, comments, and consistent adherence to accepted coding standards.
4. **Documentation** (10%): clarify and precision of the extensive documentation comment that you must include at the top of the file.

Late submissions will be graded one full letter grade down (i.e. from A- to B-) for each started period of 12 hours of delay.

Students may request for extensions should they provide valid formal reasons with documented evidence to justify their case. Such cases will be handled on a case-by-case basis. Request for extension after the deadline will be accepted only in extraordinary cases.

Honor code

SMU expects students to abide by the honor code and The SMU Student Code of Conduct. Specifically, students are expected to submit only their own work and not submit answers or code that have not been created by students themselves.

This is an individual assignment. While you are encouraged to discuss with your classmates, submitted code must be your own work. No direct assistance from generative AI tools are allowed.

Appendixes

Sample log file

The name of a file must be `log.txt`.

Each line contains exactly one entry that uses the following format:

```
'['TimeStamp'] 'ThreadType' 'ThreadId': 'Action' - Order 'orderId
```

The `Timestamp` must represent the epoch time in *milliseconds*.

Example:

```
[1703657540220] Waiter 0: Order Placed - Order 0
[1703657541241] Waiter 0: Order Placed - Order 1
[1703657542243] Waiter 0: Order Placed - Order 2
[1703657543249] Waiter 0: Order Placed - Order 3
[1703657544255] Waiter 0: Order Placed - Order 4
[1703657545222] Chef 0: Order Prepared - Order 0
[1703657545262] Waiter 0: Order Placed - Order 5
[1703657546266] Waiter 0: Order Placed - Order 6
[1703657547273] Waiter 0: Order Placed - Order 7
[1703657548277] Waiter 0: Order Placed - Order 8
[1703657549281] Waiter 0: Order Placed - Order 9
[1703657550229] Chef 0: Order Prepared - Order 1
[1703657552292] Waiter 0: Order Served - Order 0
[1703657554299] Waiter 0: Order Served - Order 1
[1703657555231] Chef 0: Order Prepared - Order 2
[1703657557239] Waiter 0: Order Served - Order 2
[1703657560235] Chef 0: Order Prepared - Order 3
[1703657562247] Waiter 0: Order Served - Order 3
[1703657565251] Chef 0: Order Prepared - Order 4
[1703657567259] Waiter 0: Order Served - Order 4
[1703657570254] Chef 0: Order Prepared - Order 5
[1703657572261] Waiter 0: Order Served - Order 5
[1703657575257] Chef 0: Order Prepared - Order 6
[1703657577261] Waiter 0: Order Served - Order 6
[1703657580265] Chef 0: Order Prepared - Order 7
[1703657582272] Waiter 0: Order Served - Order 7
[1703657585269] Chef 0: Order Prepared - Order 8
[1703657587274] Waiter 0: Order Served - Order 8
[1703657590275] Chef 0: Order Prepared - Order 9
[1703657592283] Waiter 0: Order Served - Order 9
```

Sample configuration file

The name of a file must be `config.txt`.

Each line contains exactly one unsigned integer, optionally followed by one or more whitespaces (tabs or space characters), optionally followed by a hash symbol (`#`) with a comment text. The parsing logic should follow these rules:

- There should be no empty line at the end of the file.
- The hash symbol (if present) along with command should be ignored to the end of the line.
- The whitespaces should be ignored.
- The integer should be interpreted as a value for one of settings ordered as in the example.
- Any line that does not follow this format should result in termination of the program.

Example:

```
1       # Number of chefs
1       # Number of waiters
10      # Number of orders
1000    # Time of placement of a single order (milliseconds)
5000    # Time of preparation of a single order (milliseconds)
2000    # Time of serving of a single order (milliseconds)
10      # Size of the order placement queue
2       # Size of the prepared order queue
```