Operating Systems – Exercise 3

Synchronization

Submission & General Guidelines

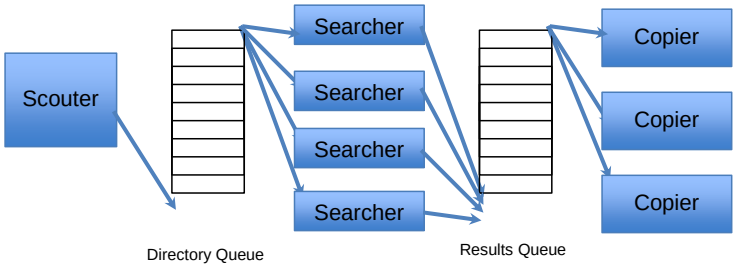
- Submission deadline is **03/06/2020, 23:55** Moodle server time
- Submit your answers in the course website only as single ex3-YOUR_ID.**zip (**e.g. ex3-012345678.zip), containing:
    - **All Java files**

- Place your name and ID at the top of every source file.
- No late submission will be accepted!
- Please give concise answers, but make sure to explain them.
- Write **clean code** (readable, documented, consistent, ...)

Part 1 (60 points)

In this part, we will create a multithreaded search utility. The utility will allow searching for all files with a specific extension in a root directory. Files with the specific extension will be copied to a specified directory.

The application consists of two queues and three groups of threads:



The attached JavaDoc contains detailed explanation for each class in the application. Please read it carefully and follow the APIs as defined in it.
(To open the attached JavaDoc open the file index.html inside the directory doc)

A. Write the class SynchronizedQueue
   This class should allow multithreaded enqueue/dequeue operations.
   The basis for this class is already supplied with this exercise. You have to complete the empty methods according to the documented API and also follow **TODO** comments.
   For synchronization you may either use monitors or semaphores, as presented in recitation.

This class uses Java generics. If you are not familiar with this concept you may read the next link - https://www.tutorialspoint.com/java/java_generics.htm
Of course, many other resources about the uses of this concept are out there in the web.

B. Write the class Scouter that implements Runnable.
This class is responsible for listing all directories that exist under the given root directory. It enqueues all directories into the directory queue.
There is always only one scouter thread in the system.

C. Write the class Searcher that implements Runnable.
This class reads a directory from the directory queue and lists all files in this directory. Then, it checks for each file name if it has the correct extension. Files that have the correct extension are enqueued to the results queue (to be copied).

D. Write the class Copier that implements Runnable.
This class reads a file from the results queue (the queue of files that contains the output of the searchers), and copies it into the specified destination directory.

E. Write the class DiskSearcher.
This is the main class of the application. This class contains a main method that starts the search process according to the given command lines.
The main class also contains another SynchronizedQueue<String> for a different purpose – to save the execution milestones for each thread. i.e., Every action performed by threads should be written down as a String in the queue. It should work as follows:
The Global shared SynchronizedQueue contains strings :
SynchronizedQueue<String> milestonesQueue.
PLEASE NOTE - The program should receive a "flag" argument (essentialy a boolean), that indicates whether the milestonesQueue and written to or not (in that case, it should be set to null). If not, the program executes without any writings and the queue is set to null (the constructor of the all threads receives null in it's parameter place).
. If the flag is "true", then the milestonesQueue must be

For every important action made in the program (scouting, searching, copying), there will be a String in the queue, as follows:
The first Audit will always be - "General, program has started the search"
For every scouted directory, the scouter will enqueue to the queue the string - "Scouter on thread id X: directory named Y was scouted"
For every searched directory the searcher will enqueue to the queue the string - "Searcher on thread id X: file named Y was found"
For every copied file, the copier will enqueue to the queue the string - "Copier from thread id X: file named Y was copied".
Please note that every thread has it's unique ID. The whole process should help you with the debugging of the program.
Make sure there is no race conditions or other synchronization mistakes when writing your code.

Before finishing, the main class should print all entries of the queue, with indexes showing first to last actions made in the program.

Formatted: Font: (Default) Arial, Font color: Text 1, Complex Script Font: Arial

Formatted: Font: (Default) Arial, Font color: Text 1, Complex Script Font: Arial

Usage of the main method from command line goes as follows:

> java DiskSearcher <boolean of milestoneQueueFlag> <file-extension> <root directory> <destination directory>
<# of searchers> <# of copiers>

> java DiskSearcher solution txt C:\OS_Exercises C:\temp 10 5

This will run the search application to look for files with the string "solution" inside their name and that have an extension of txt, in the directory C:\OS_Exercises and all of its subdirectories. Any matched file will be copied to C:\temp. The application will use 10 searcher threads and 5 copier threads.

* Of course, other directory paths are fine.

Specifically, it should:

1. Start a single scouter thread (and make sure it writes to the milestonesQueue)
2. Start a group of searcher threads (number of searchers as specified in arguments, and make sure it writes to the milestonesQueue)
3. Start a group of copier threads (number of copiers as specified in arguments, and make sure it writes to the milestonesQueue)
4. Wait for scouter to finish.
5. Wait for searcher and copier threads to finish.

Formatted: Font: (Default) Arial, Font color: Text 1, Complex Script Font: Arial

F.

Create a time measurement that starts at the beginning of the program and ends in the last command executed (pay attention that you must make sure you start the counting at the beginning of the program and that the counting ends after all threads are finished).
Next, run the program twice, once with 1 thread for each role (scouter which always runs only as 1 thread, but also 1 thread for searcher and copier). The second time, run the program with 5 threads for searcher and copier each.

Write down the total execution time, as shown by the time measurement you implemented.

_____
_____
_____

What would happen if we increased the number of threads and partitions to 500 for each class? Will that improve the performance?

_____
_____
_____

why?

_____
_____
_____

Guidelines:

1. Read the attached JavaDoc. It contains a lot of information and tips.
   You must follow the public APIs as defined in the attached JavaDoc!
2. Use the attached code as a basis for your exercise. Do not change already-written code. Just add your code.
3. To list files or directories under a given directory, use the File class and its methods listFiles() and listfiles(FilenameFilter).
   Note that if for some reason these methods fail, they return null. You may ignore such failures and skip them (they usually occur because of insufficient privileges).
4. If you have a problem reading the content of a file, skip it.

Part 2 (20 points)

Due to COVID19 restrictions, for adhering to the purple badge (Israeli regulations for operating businesses during COVID19), only take-aways are allowed from restaurants and up to 10 people can wait inside the restaurant for the take-away. In our case, each restaurant has one chef.

The chef serves the customers one after the other, so that at any given time the chef prepares the order of a single customer. When the customers arrive at the restaurant and the chef is busy preparing an order (of another customer), the customer will wait in the restaurant if in the restaurant there are at most 9 other customers (not including the customer himself). If there are 10 customers waiting inside the restaurant, the customer will leave the restaurant and will not be served.

The operating of the restaurant is implemented as follows: the chef's task is implemented as a single process running the 'chef' function. Each customer is implemented in a separate process executing the 'customer' function. When the chef completes the preparation of the customer's order (that is, completes the 'prepareMeal() function), the customer takes the order (implemented in function getMeal()) and the chef picks another waiting customer to handle his order.

We shall examine the solution in the following criteria:

1. Avoiding contagion – only one customer gets the order simultaneously (i.e. running the code of 'getMeal()').
2. Progress – if customer A waits for an order, there will be a customer that will get an order in a finite time (not necessarily customer A).
3. Efficiency – the chef does not waist CPU resources if there is no customer waiting for an order or receiving an order.
4. No Starvation – if a customer enters the restaurant, the customer will get his order in a finite time (in function 'getMeal()') or leaves the restaurant without being served.
5. Adhering to the purple badge – at most 10 customers are waiting in the restaurant. A customer is defined as 'waiting' if he/she is in the execution of the code marked between 'start of waiting block' to 'end of waiting block'.

a. Consider the following solution:

```
Initialization of shared variables:

semaphore customers initialized to 0; // shared among processes

semaphore   chef initialized to 10; // shared among processes

semaphore   meal initialized to  0; // shared among processes
```

```
void chef (void) {

  while(TRUE) {

            down (customers);
            prepareMeal();
           up(meal);
            up (chef);
        }

}

void customer (void) {

        \\ start of waiting block - beginning of the code where processes are waiting for service

      up(customers);
      down(chef);
      \\ end of waiting block - end of the code where processes are waiting for service
      down(meal);
      getMeal();

}
```

Formatted: Font: (Default) Calibri, 11 pt, Font color: Text 1, Complex Script Font: Calibri, 11 pt

Formatted: Font: (Default) Calibri, 11 pt, Font color: Text 1, Complex Script Font: Calibri, 11 pt

Assume the semaphores are not fair.

For each of the following properties, mark if it holds or not and explain.

· Avoiding contagion: yes / no

Explain:

_____
_____
_____
_____

· Progress: yes / no

Explain:

_____
_____
_____
_____

· Efficiency: yes / no

Explain:

_____
_____
_____
_____

· No starvation: yes / no

Explain:

_____
_____
_____
_____

· Adhering to the purple badge: yes / no

Explain:

_____
_____
_____
_____

b. second solution offered -

Initialization of shared variables :

```
Int         cnt_customers                   initialized to 0

semaphore_ customers initialized to 0; // shared among processes

semaphore   chef initialized to 10; // shared among processes

semaphore   meal initialized to  0; // shared among processes

void chef (void) {

  while(TRUE) {

          down(customers);

            prepareMeal();.

          up(meal);

          up (chef);

  }

}

void customer (void) {

           if (cnt_customers>=10) {

                    // leave shop - the customer leaves the restaurant without being served


                    return;

                  }

           cnt_customers = cnt_customers+1;

           // start of waiting block - beginning of the code where processes are waiting for service
           up (customers);

           down(chef);

          // end of waiting block - beginning of the code where processes are waiting for service
           down(meal);

           getMeal();

           cnt_customers = cnt_customers-1;                    }
```

In this section, assume the semaphores are fair.

For each of the following properties, mark if it holds or not and explain.

- Avoiding contagion: yes / no

Explain:

_____
_____
_____
_____

- Progress: yes / no

Explain:

_____
_____
_____
_____

- Efficiency: yes / no

Explain:

_____
_____
_____
_____

- No starvation: yes / no

Explain:

_____
_____
_____
_____

- Adhering to the purple badge: yes / no

Explain:

_____
_____
_____
_____

Part 3 (20 points)

Answer whether each question is true / false and explain!

a. Busy waiting is happening every time there's a sleeping process, waiting to be awakened.

b. Semaphores can provide the same functionality as mutex locks. True / False Yes,

c. Assuming we need to run 2 concurrent tasks, the only benefit concurrent threads have on concurrent processes is a much more efficient communication between them.

d. Before the multi-core processors emerged, semaphores, monitors and other locks' implementations were not needed.