Operating Systems – Exercise 3

Synchronization

Submission & General Guidelines

- Submission deadline is **01/5/2019, 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:
    - Ex3.**pdf**
    - **All Java files**

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

Part 1 (20 points)

This part aims to show performance improvements using threads.
We will be going through a file that contains all of Shakespeare's literature several times and we are going to search each lines for clues he left in the text, we will be searching for characters that consist the word – "operating system" – 'o' , 'p', 'e' 'r' 'a' 't' 'i' 'n' 'g' 's' 'y' 'm'.
if at least one of the characters exists in a line, the counter is increased (This part is already implemented in the files provided)
Guidelines:

1. Download the shakespear.txt file from https://drive.google.com/open?id=1gp6C5qvBw46aGbEpqqSDmUDxS6fhYvij
2. Use the Main.java and the worker.java files provided and implement the following:
    a. Implement the getLinesFromFile() method.
    Read the Shakespeare.txt file from C:\Temp\Shakespeare.txt and save its lines in an array list of Strings. (Read about File API in Java)
    b. Implement the workWithThreads method.
        i. Get the number of available cores:
        int x = Runtime.getRunTime().availableProcessors();
        ii. Partition the lines collection into x
        data sets (you can use the List's sublist API)
        iii. Create x Threads that will activate the run method of the worker, each thread should handle a different data set from the partition
        iv. Wait until all thread finish.
    c. Run the main method and observe the different time it took to execute without threads and with threads (Time measurement is already implemented

3. Without threads execution time: _____
4. With threads execution time: _____
5. What would happen if we increase the number of threads and partitions to 100. Will that improve the performance?

   _____

   _____

   Why?

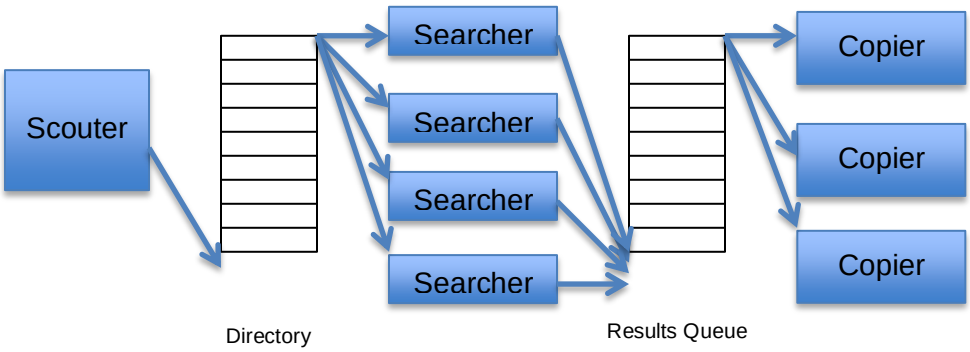
   _____

   _____

Part 2 (40 points)

In this part, we will create a multithreaded search utility. The utility will allow searching for files that contain some given pattern, under some given root directory. Files that contain this pattern will be copied to some specified directory.

Our 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 <u>index.html</u> inside the directory <u>doc</u>)

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.
   <u>For synchronization</u> you may either use monitors or semaphores, as learned in recitation.
   This class uses Java generics. If you are not familiar with this concept you may read the first few pages of this tutorial: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

B. Write the class Scouter that implements Runnable.
   <u>This class is responsible to list all directories that exist under the given root directory. It enqueues all directories into the directory queue.</u>
   <u>There is always only one scouter thread in the system.</u>

C. Write the class Searcher that implements Runnable.
<u>This class reads a directory from the directory queue and lists all files in this directory. Then, it checks each file to see if the file name contains the pattern given. Files that contain the pattern are enqueued to the results queue (to be copied).</u>

D. Write the class Copier implements Runnable.
This class reads a file from the results queue (the queue of files that contains the pattern), 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.
Usage of the main method from command line goes as follows:

```
> java DiskSearcher <filename-pattern> <root directory> <destination directory>
                    <# of searchers> <# of copiers>
```

For example:

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

This will run the search application to look for files with the string "solution" inside them, 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.
Specifically, it should:

- Start a single scouter thread
- Start a group of searcher threads (number of searchers as specified in arguments)
- Start a group of copier threads (number of copiers as specified in arguments)
- Wait for scouter to finish
- Wait for searcher and copier threads to finish

<u>Guidelines:</u>

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 insufficient privileges).
4. If you have a problem reading the content of a file, skip it.

Part 3 (20 points)

1. (20 points) Prove or provide a detailed counter example.

(20 points)

Fetch_and_add(&p, inc) is an atomic function which reads the value from location *p* in memory, increments it by *inc* and returns the value of *p* before the change.

Fetch_and_add(&p, inc):

{val=*p; *p=val + inc; return val;}

Given the following solution to the critical section problem, which uses a member called *lock* which is initialized to 0:

```
while(1)
{
[Remainder Code]
while(Fetch_and_add(lock, 1) != 0)
{
lock = 1;
}
[Critical Section]
lock = 0;
}
```

Prove, or provide a detailed counter example:

a. Does the algorithm provide Mutual Exclusion?

Yes / No

b. Does the algorithm satisfy Deadlock Freedom?

Yes / No

c. Does the algorithm satisfy Starvation Freedom?

Yes / No

_____

_____

d. Does the algorithm suffer from busy-waiting?
Yes/No

_____

_____

_____

Part 4 (20 points)

Answer whether each question is true / false and explain!

a. Inter process communication (IPC) Signals is more efficient than communication between threads of the same process.
True / False

_____

_____

b. A race condition can only occur on a computer with one CPU (or core) sue to many context switches.
True / False

_____

_____

c. Using Semaphores ensured that a program will not deadlock
True / False

_____

_____

d. Three processes execute the following code using three counting semaphores which are initialized to the following values: S1=1, S2=0, S3=0

<u>Process 3:</u>
down(S3); up(S1);

<u>Process 2:</u>
down(S2); up(S1);

<u>Process 1:</u>
while(true) { down(S1); print("me!"); up(S2); up(S3); }

The line "me!" will be printed 2 or 3 times
True / False

e. The three processes (of clause d) now execute the same code using three binary semaphores that are initialized to the following values: S1=1, S2=0, S3=0
The line “me!” will be printed 3 times
True / False