323746016
Operating Systems – Exercise 3

Synchronization

Submission & General Guidelines

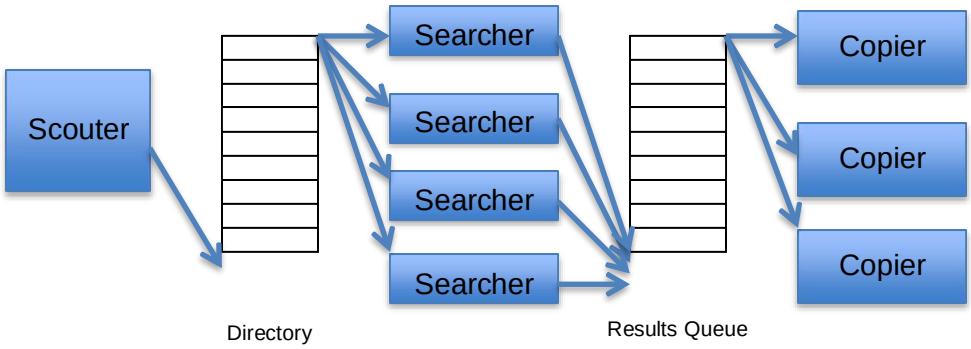
- Submission deadline is **03/06/2020, 23:55** Moodle server time
- Submit your answers in the course website only as single ex3-YOUR_ID.**zip (**e.g. ex3-012345678.zip), containing:
    - Ex3.**pdf**
    - **All Java files**

- 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 (40 points)

In this part, we will create a multithreaded search utility. The utility will allow searching for files that contain a given pattern in the name of the file, for all files with a specific extension in a root directory. Files with a specific extension that contain this pattern in their name 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 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.
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 each file name for containing the given pattern and if it has the correct extension. Files that contain the pattern and 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.
Usage of the main method from command line goes as follows:

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

For example:

```
> 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.
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

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 insufficient privileges).

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

Part 2 (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 line for clues he left in the text. We will be searching for characters constructing 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 _____
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. Read about thread pools and why to use them in the following link: https://www.geeksforgeeks.org/thread-pools-java/
        iv. Create a fixed thread pool of size x using the Executors.newFixedThreadPool(x) API
        v. Submit x workers to the thread pool using the submit method, this will cause every worker to be executed by a thread from the pool. each worker should handle a different data set from the partition
        vi. Wait until the executor service finishes by using the shutdown and awaitTermination API.
    c. Run the main method and record the time it took to execute without threads and with threads (Time measurement is already implemented)
3. Without threads execution time: 65
4. With threads execution time: 22

What would happen if we increase the number of threads and partitions to 500. Will that improve the performance?

No

Why?

, because there will be a lot of unnecessary overhead like system calls and context switches. As we saw in Class at a certain point the graph of time starts going back up because of overhead

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* (lock 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

Yes , lets check with threads T1 and T2 can they both be in the Critical section?
Lets say T1 entered first the Critical section then before Fetch_And_Lock lock = 0 and after Lock = 1, so it can go past the while loop .
If by contradiction T2 is also in the critical section than lock = 0 before T1 finnished fetch and add

Which is a contradiction that Fetch_aNd_add is atomic

b. Does the algorithm satisfy Deadlock Freedom?

Yes / No

Yes, there is No CIrculer Dependancy because there is one lock. That everyone is competing over

c. Does the algorithm satisfy Starvation Freedom?

Yes / No

No, example lets say thread T1 finnished the critical section and there was a context switch
Before Lock was initialized to zero , so now Lock != 0 so no one will get past the while
Which ,eans that no one will be able to get to the critical section

a. Does the algorithm suffer from busy-waiting?
yes/No
Yes, there is a while loop that keeps running till Lock = 0, therefore there is busy waiting.

Part 4 (20 points)

Answer whether each question is true / false and explain!

a. As Threads do not share the stack, communication among them must be done by IPC ways such as Pipes or Signals.

True / False

False, because they share all the other memory they can communicate through global variables

b. A race condition can only occur on a computer with more than one core as it happens only when we can execute more than a single instruction at a time.

True / False

False, there can be multible threads on a single core, that run at different times
But through context switches are competing
For the same resources

c. In order to improve the throughput of a CPU bound application, it is better to increase the number of threads as much as possible

True / False

false , , because there will be a lot of unnecessary overhead like system calls and context switches.
That the threads will wait for

d. Semaphores, as opposed to Monitors, guarantee deadlock freedom

True / False
false, there are cases where semaphores do not guarantee deadlock freedom we have seen

one like that in the recitation