

# Gai Ashkenazy - 204459127

## Operating Systems – Exercise 3

### Synchronization

#### 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 shakespeare.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: 2064

4. With threads execution time: 720

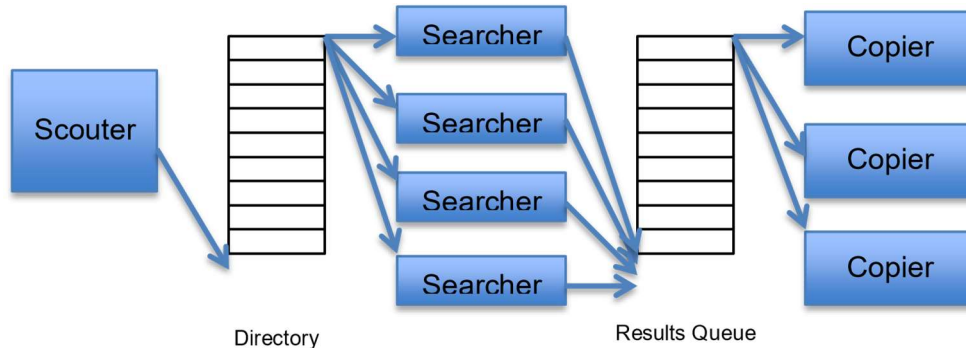
5. What would happen if we increase the number of threads and partitions to 100. Will that improve the performance?

No, first creating 100 threads by itself takes a long time, and second, we are using all available cores in this process so creating more thread will not be effective. So, because we create 100 threads and the are not used efficiently it will decrease performance.

## 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 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 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.

This class is responsible to list 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 to see if the file name contains the pattern given. Files that contain the pattern are enqueued to the results queue (to be copied).

- 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 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 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

Assume towards a contradiction that there are two threads, T1 and T2 in the critical section, this means that w.l.o.g T1 entered first, so lock is 1, now if T2 entered the critical section lock must be 0, which means that T1 exited the critical section, contradiction. Hence the algorithm provides mutual exclusion.

b. Does the algorithm satisfy Deadlock Freedom?

Yes / No

At any given moment lock is 0 or not, so at anytime any thread could either enter the loop or skip over it to the CS, there is a single resource, lock, no circular dependency.

c. Does the algorithm satisfy Starvation Freedom?

Yes / No

Counter example: if T1 runs all the way to the CS, and T2 is in the while loop, now say T1 sets lock to 0, and after that T2 that is in the while loop sets lock to 1, now T1 enters the while loop, because lock is 1, now both T1 and T2 are in the while loop. They cannot change lock to 0 so they will be in starvation.

- d. Does the algorithm suffer from busy-waiting?

Yes / No

While a thread is in the CS other threads will be waiting in the while loop always performing the Fetch and Add operation, and setting lock to 1 operation.

#### 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**

Threads of the same process share the code, data and files so they communicate by only reading the data, IPC signals like kill are a system call so in addition to reading the data they need to do a content switch to the kernel mode.

- b. A race condition can only occur on a computer with one CPU (or core) due to many context switches.

True / **False**

Race condition can occur when there are two (or more) threads that access the same data, this can happen on a computer with two CPU's also. Also when there are more processes than cores a race condition can happen.

- c. Using Semaphores ensured that a program will not deadlock

True / **False**

We saw an example of this in class, if we have two semaphores A, B and we have two threads running, if T1 runs Down(A) and after that T2 runs Down(B) and now the next line of code for T1 is Up(B) and for T2 is Up(A) we are in a deadlock, semaphores are not a silver bullet.

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

Process 3:

`down(S3); up(S1);`

Process 2:

`down(S2); up(S1);`

Process 1:

`while(true) { down(S1); print("me!"); up(S2); up(S3); }`

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

**True / False, it will be printed 3 times, it is not clear if the question asks about 2 or 3 or both 2 or 3.**

The only process that can run at the beginning is P1, the others are blocked. Now in any case P1 can 3 times.

An example of a run that will print 3 time: If P1 runs once on the loop the second time it will stop in down(S1) now let P2 run and after that P1 to continue then P3 will run and then P1 again, this is 3 times.

It cannot be printed less than 3 times or more then 3, because in each run we will perform up(S2) and up (S3) at least once, and we will perform up(S1) 2 times.

- 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**

The line can be printed 2 times, if P1 starts to run all the way to up(S3), now let P2 and P3 run all the way to the end of their code, because S1 is a binary semaphore its value will be 1, which means it has only one more time to run the print command, so in total the print command will run only twice.