Operating Systems

Exercise 3 – Synchronization

General Guidelines:

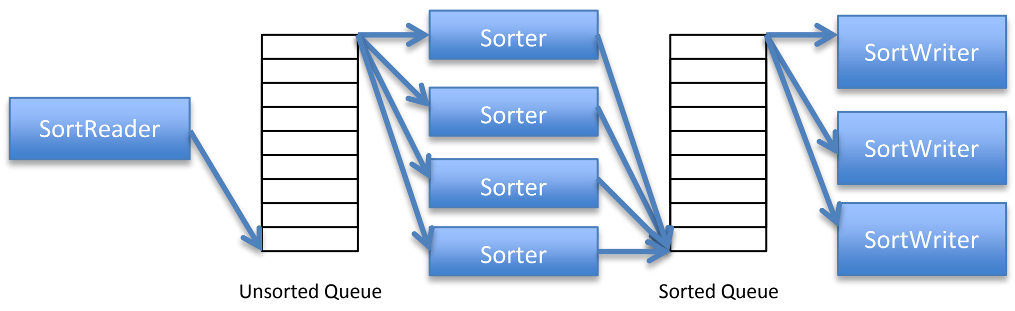
Submission deadline is **Wednesday, May 2, 23:59**
Submit your answers as PDF and Java files, packed into a single ZIP file.
Pack your files in a ZIP file named ex3-YourName-YourID.zip,
for example: ex3-JohnDoe-012345678.zip
Post the ZIP file in the submission page in course website

- No late submission will be accepted!
- You should work on your exercise by yourself. Misconducts will be punished harshly.
- Please give concise answers, but make sure to explain any answer.
- Document your code. Place your name and ID at the top of every source file, as well as in the PDF with the answers.

Part 1 (50 points)

In this part we will implement a multithreaded multi-file sort utility. This utility will be able to fetch files that contain sequences of numbers, sort them, and write the results to new files.

Input files of this utility are text files with some number (integer) at each row. The utility is started with some directory given, and it fetches all such files from this directory (according to their file extension, see below). Several sorting threads read file data from a shared queue (each thread reads data of another file), and sort it. When done, sorting threads enqueue the results to another queue, from which writing threads read these results in order to write them to disk.



A.
Write the class `SortTask`. This class represents a single file sort task. It should contain the integers array to sort, the source file path, and some simple methods as described in the attached JavaDoc API.

B.
Write the class `BoundedBuffer`. This class should allow multithreaded enqueue/dequeue operations. **The basis for this class is already supplied with this exercise** (in `src` directory). 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

C.
Write the class `SortReader extends Thread`. This class is responsible to read all files with extension `.unsorted` in the directory given to its constructor, and convert them to an array of integers. Then, it should create a corresponding `SortTask` object and enqueue it to the *unsorted queue*.
If this class has any problem with reading a file, it should display an informative error message and continue to other files.

To make sure that there is always at most one reader thread in the system, this class should be a *singleton*.
The basis for this class is already supplied with this exercise (in `src` directory). You have to complete the empty methods according to the documented API and also follow **TODO** comments.

D.
Write the class `Sorter implements Runnable`. This class reads a `SortTask` from the Unsorted Queue and sorts its data. Sorted result should be stored in the same place where unsorted data was. Then, it should enqueue the `SortTask` to the Sorted Queue.
(You may use the Arrays.sort method in order to perform the sort)

E.
Write the class `SortWriter implements Runnable`. This class reads a `SortTask` from the Sorted Queue, and writes its data to a file with the name specified in the `SortTask` object, but with a `.sorted` extension, instead of the `.unsorted` extension of the input files.

F.
Write the class `SortFiles`. 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 SortFiles directory num-sorters num-writers
```

For example:

```
> java SortFiles C:\to_sort 10 5
```

This will run the sort application to sort all files with extension `.unsorted` and write them sorted, to the same directory, with an extension `.sorted`. The application will use 10 sorter threads and 5 writer threads.
Specifically, it should:

- Start the single reader thread
- Start a group of sorter threads (number of sorters as specified in arguments)
- Start a group of writer threads (number of writers as specified in arguments)
- Wait for reader to finish
- Wait for sorters and writers threads to finish

Guidelines:

1. Read the attached JavaDoc (open `index.html` in `doc` directory). 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 method `listFiles(FilenameFilter)`.
4. You can use the attached set of 16 test files, in directory `test`.
5. Use 10 as a size for your bounded buffers. Test your program with more than 10 files.
6. To read textual data from a file you should read the file using a character stream. See this short tutorial about them: http://download.oracle.com/javase/tutorial/essential/io/charstreams.html
   More specifically, you should use a `BufferedReader` that wraps a `FileReader`. `FileReader` class lets you read characters from a file, and `BufferedReader` manages the buffering for you (to save system calls, like you learned in exercise 1).
   Then, you should read whole lines from the reader one after each other using the `readLine()` method of `BufferedReader`. While reading characters you should try to match the pattern (which is a string) to the characters you read.
   For writing text files, same instructions apply by using `BufferedWriter` *and* `FileWriter`*, using methods* `write(String)` *and* `newline()`.

<u>Part 2</u> (18 points)

The `swap(int *a, int *b)` operation performs a swapping of the values in two memory addresses `a` and `b`, in a single atomic operation.

The following code tries to solve the mutual exclusion problem using the `swap(…)` operation, for any number of threads:

```
// Shared variable:
int lock = 0;

// Local variables:
int value;
int id; // initialized as the number of thread (greater than 0)

// Entry code:
do {
    value = id;
    swap(&lock, &value);
} while (value != 0 && value != id);

// Critical Section

// Exit code:
lock = 0;
```

Prove (formally) or disprove (by a counterexample) the following claims:

A. The algorithm satisfies mutual exclusion

B. The algorithm satisfies deadlock freedom

C. The algorithm satisfies starvation freedom

Part 3: (32 points)

A barrier is a synchronization tool for multiple processes, which allows all participating processes to wait for each other at a specific point in the code, until all of them reach it, and then continue.

Below is an example for using a barrier in the code of some process, using a shared barrier `B`:

```
// Do part 1
wait_on_barrier(B);  // until all processes arrive at this point
// Do part 2
```

Note that in this example, no process will execute part 2 before all processes finish part 1.

Below are several suggestions for implementing a barrier. For each suggestion, answer whether it provides the required behavior, as explained above, and also what are the disadvantages of it, if there are any. Explain your answers!

A.
Barrier fields:
`count` – integer, initialized to the number of participating processes

```
wait_on_barrier(B):
    B.count--
    while (B.count > 0) { /* wait */ }
```

Does the suggested solution provide the requested barrier behavior?

What are the disadvantages of this solution, if any?

B.
Barrier fields:
`count` (like in the previously suggested code)
`monitor` – a monitor

```
wait_on_barrier(B):
    synchronized(B.monitor) {
        B.count--
        while (B.count > 0) { /* wait */ }
    }
```

Does the suggested solution provide the requested barrier behavior?

What are the disadvantages of this solution, if any?

C.

Barrier fields:

`count` (like in the previously suggested code)

`monitor` – a monitor

```
wait_on_barrier(B):
    synchronized(B.monitor) {
        B.count--
    }
    while (B.count > 0) { /* wait */ }
```

Does the suggested solution provide the requested barrier behavior?

What are the disadvantages of this solution, if any?

D.

Barrier fields:

`count` (like in the previously suggested code),

`monitor` – a monitor

```
wait_on_barrier(B):
    synchronized(B.monitor) {
        B.count--
        if (B.count > 0)
            B.monitor.wait()
        else:
            B.monitor.notifyAll()
    }
```

Does the suggested solution provide the requested barrier behavior?

What are the disadvantages of this solution, if any?