Lab#3 – Message Queues

Objective

In this lab, you will build a series of software components to illustrate the usefulness and benefits of message queues.

Message Queues

Message queues are essential in building:

- large and loosely connected distributed systems,
- systems that can grow over time,
- distributed logging system for performance monitoring.

Tasks

Task #1: Implementing a simple message queue in Java

A message queue can be roughly described by the following interface:

```
interface MessageQueue<T> {
T take();
void put(T m);
}
```

These methods are self-explanatory.

For this task, you are to implement a simple producer/consumer program.

1. Construct a MessageQueue<Line> q1, where Line is a class shown below. You are highly recommend to utilize the *java.util.concurrent.BlockingQueue* class to implement the message queue. (Hint: BlockingQueue already satisfies the interface.)
2. The producer scans through a given file, and ***put***s each line as a message to the message queue q1. At the end of the file, the producer **MUST PUT** a special END-OF-FILE message. The **Line** class has a field that indicates if it is an end-of-file.
3. The consumer ***take***s the messages from q1, and checks for the words in the line that are at most *k* edit-distance away from a query word.
4. The consumer saves the lines matching the query word (with <= k edit-distance) to a given output file.

You will find the following utility classes useful.

```
Class Line {
```

```
String content;
long lineNumber;
public Line(String content, long lineNumber) { ... }
bool isEnd() {
      return lineNumber < 0;
}
}

Class FileIterator implements Iterable<Line> {
FileIterator(String filename, int repeat) {...}
}

Class Util {
public static String[] words(String lineContent) {...}
public static int editDistance(String s1, String s2) {...}
}
```

You are recommended to follow the following class structure for the producer and consumer, and the main executable class.

```
Class LineProducer implements Runnable {
public  LineProducer(FileIterator  input,  MessageQueue<Line>  q1)
{...}
/**
 * puts strings from q1
*/
public void run() {...}
}

Class LineConsumer implements Runnable {
public LineConsumer(MessageQueue<Line> q1, String outputFileName)
{
      ...
}
// starts the consumption of strings from q1
public void run() {…}
}

class Program {
public static void main(String[] args) {
      String inputFileName  = args[0];
      int    repeat         = Integer.parseInt(args[1]);
      String outputFileName = args[2];

      // setup the data flow
      MessageQueue<Line> q1 = ...;
      FileIterator lines = new FileIterator(inputFileName,repeat);
      LineProducer p1 = new LineProducer(lines, q1);
      LineConsumer c1 = new LineConsumer(q1, outputFileName);

      // place the producer and consumer in thread containers
```

```
            List<Thread> threads = new ArrayList<Thread>();

            threads.add(new Thread(p1));
            threads.add(new Thread(c1));

            long start = System.currentTimeMillis();

            // start the threads
            for(Thread t : threads) {
                  t.start();
            }

            // wait for the threads to complete
            for(Thread t : threads) {
                  t.join();
            }

            long duration = System.currentTimeMillis() - start;

            System.out.println("Total Duration: " + duration + " ms.");
      }
}
```

Presentation to the TA:

Show the TA the running programs.

Task #2: Multiple working consumers

In this task, you will extend the system with multiple consumers to maximally utilize all the available CPUs.

Queues are highly functional in helping distributing work among multiple works, and then collect them, *all without locking*.

You will need to make some modifications to the pipeline. The LineConsumer still consumes from q1, but it also places its output to another queue q2.

```
Class LineConsumer implements Runnable {
public LineConsumer(MessageQueue<Line> q1, MessageQueue<Line> q2)
{...}
public void run() {
        ...
}
}
```

You will need to implement a new consumer that consumes the results from q2, and write them to an output file.

```
Class ResultConsumer implements Runnable {
public ResultConsumer(MessageQueue<Line> q2,
       String outputFileName) {...}
public void run() {
        ...
}
}
```

The executable class will also need to be modified to construct two queues, q1 and q2, and multiple consumer threads.

```
Class Program2 {
public static void main(String[] args) {

        String inputFileName  = args[0];
        int    repeat         = Integer.parseInt(args[1]);
        String outputFileName = args[2];
        int    numWorkers     = Integer.parseInt(args[3]);

        FileIterator        lines = ... ;
        MessageQueue<Line> q1 = ... ;
        MessageQueue<Line> q2 = ... ;

        List<Thread> threads = new ArrayList<Thread>();

        // create the producer thread
        threads.add(new Thread(new LineProducer(lines, q1)));

        // create the LineConsumer threads
```

```
            for(int i=0; i < numWorkers; i++)
                  threads.add(new Thread(new LineConsumer(q1, q2)));

            // create the ResultConsumer threads
            threads.add(new Thread(new ResultConsumer(q2,
                                      outputFileName)));

            ...
      }
}
```

Presentation to the TA:

Show the TA the running programs, with multiple consumers.

Task #3: Extending the functionality of message queues

Message queues are **great** because they can inspect every message passing through, and optionally perform additional functionalities.

Extend your implementation of message queue to filter out any line containing the phrase "distributed systems".

Presentation to the TA:

Show the TA the running programs as above with the additional scenarios where some messages have lines containing "distributed systems" and show that those lines get filtered out.

Lab Report:

Write in your lab report the answers to the following questions.

For **Task #1**:

1. Use your system specific CPU monitoring tool. What is the maximal CPU utilization over all your CPUs? For Ubuntu Linux, you can install *htop* command to monitor multiple CPU usages.

2. If multiple consumers are created, what would be the potential problem if they all write to the same file without synchronization?

For **Task #2**:

1. Repeat the CPU utilization measurement for various numWorkers values (for example: 2, 4, 6, 8, 10).
2. Measure the total processing time for various numWorkers values (for example: 2, 4, 6, 8, 10). What is the optimal setting of numWorkers? Explain.

General question:

1. How would you suggest to implement message queues that can connect **two** Java programs running on different machines, connected by the Internet?

Submitting lab report :

Submit your lab report on Blackboard by the day of the lab session (11:59PM).