University of Glasgow | School of Computing Science

# Assessed Coursework

| | | | | |
|---|---|---|---|---|
| **Course Name** | **Advanced Programming 3** | | | |
| **Coursework Number** | **2** | | | |
| **Deadline** | **Time:** | **4:30pm** | **Date:** | **4th December 2014** |
| **% Contribution to final course mark** | **10%** | | | |
| **Solo or Group** ✓ | **Solo** | ✓ | **Group** | |
| **Anticipated Hours** | **15** | | | |
| **Submission Instructions** | **Submit via Moodle; see attached for details** | | | |
| **Please Note: This Coursework cannot be Re-Done** | | | | |

## Code of Assessment Rules for Coursework Submission

Deadlines for the submission of coursework which is to be formally assessed will be published in course documentation, and work which is submitted later than the deadline will be subject to penalty as set out below.

The primary grade and secondary band awarded for coursework which is submitted after the published deadline will be calculated as follows:

(i) in respect of work submitted not more than five working days after the deadline
   a. the work will be assessed in the usual way;
   b. the primary grade and secondary band so determined will then be reduced by two secondary bands for each working day (or part of a working day) the work was submitted late.

(ii) work submitted more than five working days after the deadline will be awarded Grade H.

Penalties for late submission of coursework will not be imposed if good cause is established for the late submission. You should submit documents supporting good cause via MyCampus.

**Penalty for non-adherence to Submission Instructions is 2 bands**

**You must complete an "Own Work" form via**
**https://webapps.dcs.gla.ac.uk/ETHICS for all coursework**
**UNLESS submitted via Moodle**

# Concurrent file system Crawler

# 1 Requirement

Disks attached to laptops and desktops have become absurdly large. This generally leads users to accumulate large numbers of files in a large number of directories – i.e. rather than manage the use of disk space, it is easier to simply create more directories, retaining older versions of files.

Some operating systems have addressed this by continuously indexing the names of files in the file system, and then providing the ability to query the index – e.g. Google desktop. Server system generally do not continuously index file names, instead providing one or more applications for searching the file system for files whose names match the desired patterns. Linux, for example, provides the find application for general traversal of the file system – see the man page for more details.

The syntax for the find application is extremely difficult to master. You are already familiar with the filename patterns that are understood by bash. The goal for this assignment is to design and build a high-performance application that will recursively crawl through a set of specified directories, looking for filenames that match a specified bash pattern.

For very large software systems, a singly-threaded application to crawl the directories may take a very long time. The purpose of this assessed exercise is to develop a concurrent file system crawler in Java, exploiting the concurrency features of the Java language, to rapidly, recursively crawl through the nominated directories, looking for filenames that match a specified pattern.

# 2 Specification

You are to create a class named fileCrawler, in a source file named fileCrawler.java. The static void main() method of this class must understand the following arguments:

| | |
|---|---|
| pattern | indicates a bash pattern that filenames must match |
| directory | directory to be recursively crawled for matching filenames |

The usage string is: java –classpath . fileCrawler pattern [directory] …

If a directory is not specified, the application must search in the current directory, ".".

The application must use the following environment variable when it runs:

CRAWLER_THREADS – if this is defined, it specifies the number of worker threads that the application must create; if it is not defined, then two (2) worker threads should be created.

## *2.1 Bash pattern refresher*

When you type a command to bash, it splits what you have typed into words. It scans each word for the characters '*', '?', and '['. If one of these characters appears, then the word is regarded as a pattern, and replaced with an alphabetically sorted list of filenames in the specified directory matching the pattern. Your fileCrawler will look for these patterns recursively in the specified directories. Note that when you specify pattern to fileCrawler, you will have to enclose it in single quotes to prevent bash from interpreting it – e.g., '*.c'.[1]

Any character that appears in a pattern, other than the special characters described below, matches itself.

The special pattern characters have the following meanings:

* Matches any string, including the null string.

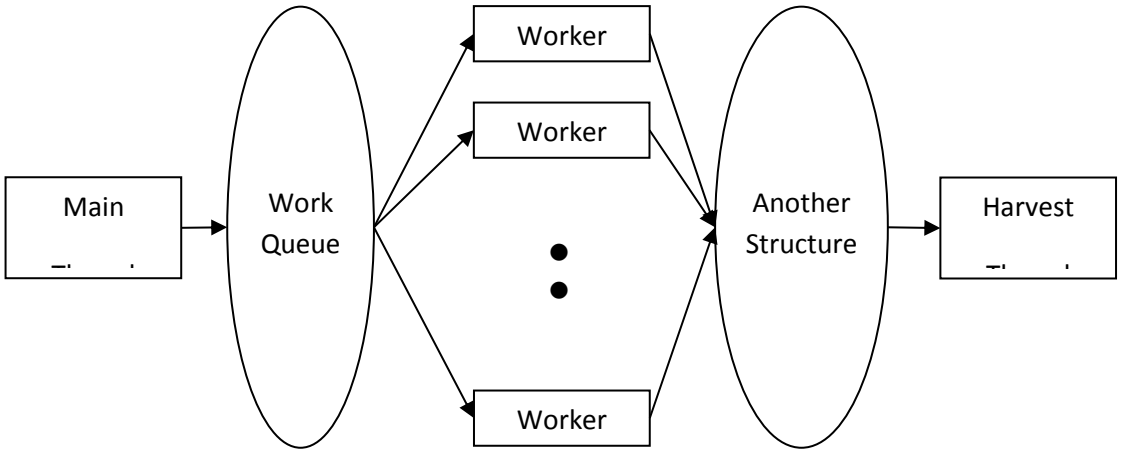
? Matches any single character.

[...] Matches any one of the enclosed characters. A pair of characters separated by a hyphen denotes a range expression; any character that sorts between those two characters, inclusive, is matched. If the first character following the '[' is a '^', then any character **not** enclosed is matched.

[1] Note that if you are building and testing your program using Java on Windows, you will have to doubly escape the bash patterns when invoking java, as in "'*.c'". The code in Section 6 is designed to strip off such leading and trailing apostrophes.

# 3 Design and Implementation

The concurrent version of the application naturally employs the manager/worker pattern. An abstract version of this is shown in the figure below.




The Harvest Thread is often identical to the Main Thread, thus the Main Thread is the manager. It should be possible to adjust the number of worker threads to process the accumulated work queue in order to speed up the processing. Since the Work Queue and the Another Structure are shared between threads, you will need to use Java's concurrency control mechanisms to implement appropriate [conditional] critical regions around the shared data structures.

For this assignment, the Main Thread writes directory names into the Work Queue; concurrently, the Worker Threads each retrieve directories from the Work Queue; for each directory retrieved, the Worker Thread scans the entries in the directory; for each such entry that is not itself the name of a directory, it attempts to match the pattern against the filename; if it matches, it places the fully-qualified filename into Another Structure.

## *3.1 How to proceed*

The first thing most programmers do when developing a manager/worker application is to first build the application with only a single worker thread. In fact, one can design a singly-threaded program that assumes three phases:

1. populate the Work Queue
2. process the Work Queue, placing the processed data in the Another Structure
3. harvest the data in the Another Structure, printing out the results

You are provided with a working, singly-threaded fileCrawler written in C. There are extensive comments in fileCrawler.c outlining the design of the application, along with other modules that are needed to implement the program.

After you have a singly-threaded Java version working correctly, then you should move to a version in which there is a single worker thread concurrently retrieving directories from the Work Queue and placing data into the Another Structure. This will require thread-safe data structures, and that you determine an efficient way for the worker thread to determine when there are no more directories to be added to the Work Queue, and for the main thread to determine when the worker thread has finished, so that it can harvest the data in the Another Structure.

After you have this version working, it should be straightforward to obtain the number of worker threads that should be created from the CRAWLER_THREADS environment variable, and create that many worker threads. Again, the tricky part will be how the worker threads determine that there are no more directories to be added to the Work Queue, and for the main thread to determine that all of the worker threads have finished (without busy waiting) so it can harvest the information.

# 4 Hints and Options

You may not have written any Java programs that access environment variables before. The following simple program shows you how to do so.

```
public class Env {
    public static void main (String[] args) {
        for (String env: args) {
            String value = System.getenv(env);
            if (value != null) {
                System.out.format("%s=%s%n", env, value);
            } else {
                System.out.format("%s is not assigned.%n", env);
            }
        }
    }
}
```

The documentation for each Java collection class indicates if it is thread-safe or not. Be sure to choose the right one. If you cannot find one with the appropriate functionality that is also thread-safe, then you can create a subclass which does implement thread-safety, or you can encapsulate the unsafe class with thread-safe structures; you should look at `java.util.concurrent` first before building your own.

You are provided a sample directory tree populated with files on Moodle, as well as the correct output that one should obtain from all versions of your program when applying different patterns to that directory. The file `"patterns"` lists the mapping from `bash` pattern to output file.

## *4.1 Submission Options*

As with assessed exercise 1, you have the option of submitting a less than complete implementation of this exercise. Your options are as follows:

1. You may turn in a singly-threaded Java implementation of the Crawler; it must use thread-safe data structures between the phases. If you select this option, you are constrained to 50% of the total marks.
2. You may turn in an implementation which supports a single worker thread in addition to the main/manager thread. If you select this option, you are constrained to 75% of the total marks.
3. You turn in an implementation that completely conforms to the specification in section 2 above. If you select this option, you have access to 100% of the total marks.

A marking guide is appended to this handout, indicating how the marks will be apportioned between different aspects of the solution.

# 5 Submission

You will submit your solutions electronically by submitting the following files on Moodle:

- `build.sh` – a bash script that compiles all of your source files in the current directory
- `fileCrawler.java` – the source file as described above
- (other `.java` files) – if your source file depends upon any other classes that you have defined, you must include them here
- `report.doc` – this can also be `report.pdf` or `report.txt` – a report in Word, PDF, or text file format describing the state of your solution

Each of your source files must start with an "authorship statement" as follows:

- state your name, your login, and the title of the assignment (AP3 Exercise 2)
- state either "This is my own work as defined in the Academic Ethics agreement I have signed." or "This is my own work except that …", as appropriate.

You must complete an "Own Work" form via https://webapps.dcs.gla.ac.uk/ETHICS.

Assignments will be checked for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. There are very good tools for detecting collusion.

# 6 Regular Expressions in Java

The single threaded C version of the program provided in the starting TGZ archive on Moodle has an abstract data type for regular expressions; fileCrawler.c has a function, cvtPattern, which converts a bash expression to a RegEx expression.

The following simple Java class shows how to achieve the same thing using the regular expression classes provided in Java. Parts of this code may be of some use to you in your development.

```
import java.util.regex.*;
import java.util.*;

/**
 * class showing how to translate from bash pattern to RegEx pattern
 * and how to then use the resulting Regex pattern to match input from System.in
 */

public class Regex {

    // converts bash pattern to Regex string
    // due to vagaries of parameter expansion on Windows, this code will
    // strip off leading and trailing apostrophes (') from `str'
    //
    // the RegEx string is generated as follows
    // '^' is put at the beginning of the string
    // '*' is converted to ".*"
    // '.' is converted to "\."
    // '?' is converted to "."
    // '$' is put at the end of the string

    public static String cvtPattern(String str) {
        StringBuilder pat = new StringBuilder();
        int start, length;

        pat.append('^');
        if (str.charAt(0) == '\'') {  // double quoting on Windows
            start = 1;
            length = str.length() - 1;
        } else {
            start = 0;
            length = str.length();
        }
        for (int i = start; i < length; i++) {
            switch(str.charAt(i)) {
            case '*': pat.append('.'); pat.append('*'); break;
            case '.': pat.append('\\'); pat.append('.'); break;
            case '?': pat.append('.'); break;
```

```
                default:  pat.append(str.charAt(i)); break;
                }
            }
            pat.append('$');
            return new String(pat);
        }

        // simple main program - expects bash pattern in Arg[0]
        // after creating the RegEx string and constructing a Pattern,
        // it then attempts to match each line read from System.in against
        // the pattern, printing each matching line on System.out

        public static void main(String Arg[]) {
            // convert bash pattern to RegEx pattern, print it out
            String pattern = Regex.cvtPattern(Arg[0]);
            System.out.print(Arg[0] + " --> ");
            System.out.println(pattern);
            // create a scanner to read System.in
            Scanner sc = new Scanner(System.in);
            // compile RegEx pattern into Pattern object
            Pattern p = Pattern.compile(pattern);
            // for each line of input
            while (sc.hasNext()) {
                String line = sc.nextLine();
                // create a matcher against that line of input
                Matcher m = p.matcher(line);
                // if it matches the pattern, print it out
                if (m.matches())
                    System.out.println(line);
            }
        }
    }
```

# 7 Traversing directories in Java

The single-threaded C version of the program provided in the starting TGZ archive uses opendir(), readdir(), and closedir() methods from the C runtime to traverse directories. The following simple Java class shows how to achieve the same thing using Java facilities. Parts of this code may be of some use to you in your development.

```
import java.io.* ;

/**
 * A simple class to print out a directory tree, depth first
 *
 */
public class DirectoryTree {
    /**
     * Works on a single file system entry and
     * calls itself recursively if it turns out
```

```
     * to be a directory.
     * @param name    The name of a directory to visit
     */
    public void processDirectory( String name ) {
        try {
            File file = new File(name);  // create a File object
            if (file.isDirectory()) {      // a directory - could be symlink
                String entries[] = file.list();
                    if (entries != null) {      // not a symlink
                        System.out.println(name);  // print out name
                        for (String entry : entries ) {
                            if (entry.compareTo(".") == 0)
                                continue;
                            if (entry.compareTo("..") == 0)
                                continue;
                            processDirectory(name+"/"+entry);
                        }
                    }
            }
        } catch (Exception e) {
            System.err.println("Error processing "+name+": "+e);
        }
    }

    /**
     * The program starts here.
     * @param args The arguments from the command line
     */
    public static void main( String args[] ) {
        // Create an object of this class
        DirectoryTree dt = new DirectoryTree() ;

        if( args.length == 0 ) {
            // If there are no arguments, process the current directory
            dt.processDirectory( "." ) ;
        } else {
            // Else process every argument sequentially
            for( String arg : args ) {
                dt.processDirectory( arg ) ;
            }
        }
    }
}
```

# 8 Marking Scheme for AP3, Exercise 2

Your submission will be marked on a 100 point scale. I place substantial emphasis upon **WORKING** submissions, and you will note that a large fraction of the points are reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles and runs correctly. The information returned to you will indicate the number of points awarded for the submission, and will also inform you the band associated with that number of points. Note that the mapping from points to bands is different for Designated Degree students than for Honours, so you should not be surprised if the same number of points yields different bands when comparing notes with your classmates.

You must be sure that your code works correctly on the lab 64-bit Linux systems, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the lab machines before submission.

As indicated in the handout, you can choose to turn in three forms of fileCrawler.java:

- if it is strictly single-threaded, only 50 of the 100 total points are available to you
- if it consists of a main thread and a single worker thread, only 75 of the 100 total points are available to you
- if it enables multiple worker threads, all 100 total points are available to you.

The three marking schemes are described below, one for each possible choice.

The single-threaded marking scheme is as follows:

| Points | Description |
|---|---|
| 10 | Your report – honestly describes the state of your submission |
| 40 | <u>fileCrawler (+ other classes, if provided)</u><br>12 for workable solution (looks like it should work)<br>4 correct argument processing<br>2 correct processing of CRAWLER_THREADS<br>6 for successful compilation (no warnings)<br>8 reasonable, concurrency-safe classes: Work Queue and Another Structure<br>8 if it works correctly with seen and unseen directories |

The two-threaded (Main+Worker) marking scheme is as follows:

| Points | Description |
|---|---|
| 10 | Your report – honestly describes the state of your submission |
| 65 | <u>fileCrawler (+ other classes, if provided)</u><br>24 for workable solution (looks like it should work)<br>4 correct argument processing<br>2 correct processing of CRAWLER_THREADS<br>4 for successful compilation (no warnings)<br>8 reasonable, concurrency-safe classes: Work Queue and Another Structure<br>3 efficient mechanism for determination when no more directories<br>3 efficient mechanism for determination when worker thread has finished<br>8 if it works correctly with seen and unseen directories<br>9 runtime performance is similar to single threaded implementation |

The multi-threaded (Main+multiple Workers) marking scheme is as follows:

| Points | Description |
|---|---|
| 10 | Your report – honestly describes the state of your submission |
| 90 | <u>fileCrawler (+ other classes, if provided)</u><br>36 for workable solution (looks like it should work)<br>4 correct argument processing<br>2 correct processing of CRAWLER_THREADS<br>4 for successful compilation (no warnings)<br>8 reasonable, concurrency-safe classes: Work Queue and Another Structure<br>5 efficient mechanism for determination when no more directories<br>5 efficient mechanism for determination when worker thread has finished<br>8 if it works correctly with seen and unseen directories<br>9 runtime performance with 1 worker on test folder is similar to single threaded implementation<br>9 runtime performance on test folder first improves, then degrades as number of threads is increased |

Several things should be noted about the marking schemes:

- Your report needs to be honest. Stating to me that everything works and then finding that it won't even compile offends me. The 10 points associated with the report are probably the easiest 10 points you will ever earn as long as you are honest.
- If your solution does not look workable, then the points associated with successful compilation and lack of compilation errors are **not** available to you.
- The points associated with "workable solution" are the maximum number of points that can be awarded. If I deem that only part of the solution looks workable, then you will be awarded a portion of the points in that category.