University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Advanced Programming 3
Coursework Number	2
Deadline	**Time:** 16:30 **Date:** 27 November 2013
% Contribution to final course mark	10%
Solo or Group ✓	Solo ✓ Group
Anticipated Hours	10
Submission Instructions	Described in section 5, page 4, of the attached handout.
Please Note: This Coursework cannot be Re-Done	

Concurrent include file Crawler

Due at 16:30 on Wednesday, 27 November 2013, worth 10% of course

1 Requirement

Large-scale systems developed in C tend to include a large number of ".h" files, both of a system variety (enclosed in ‹ ›) and non-system (enclosed in " "). As we have seen in the first AP3 lecture, use of the make utility is a convenient way to record dependencies between source files, and to minimize the amount of work that is done when the system needs to be rebuilt. Of course, the work will only be minimized if the Makefile exactly captures the dependencies between source and object files.

Some systems are extremely large (for example, the Homework database system that Sventek's group has developed involves 25,000+ lines of code in 159 files). It is difficult to keep the dependencies in the Makefile correct as many people concurrently make changes, even using subversion. Therefore, there is a need for a program that can crawl over source files, noting any #include directives, and recurse through files specified in #include directives, and finally generate the correct dependency specifications.

#include directives for system files (enclosed in ‹ ›) are normally NOT specified in dependencies. Therefore, our system will focus on generating dependencies between source files and non-system #include directives (enclosed in " ").

For very large software systems, a singly-threaded application to crawl the source files may take a very long time. The purpose of this assessed exercise is to develop a concurrent include file crawler in Java, exploiting the concurrency features of the Java language.

2 Specification

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

-Idir   indicates a directory to be searched for any include files encountered

file.ext source file to be scanned for #include directives; ext must be c, y, or l

The usage string is: java –classpath . includeCrawler [-Idir] … file.ext …

The application must use the following environment variables 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.

CPATH – if this is defined, it contains a list of directories separated by ':'; these directories are to be searched for files specified in #include directives; if it is not defined, then no additional directories are searched beyond the current directory and any specified by -Idir flags.

For example, if CPATH is "/home/user/include:/usr/local/group/include" and if "-Ikernel" is specified on the command line, then when processing

#include "x.h"

x.h will be located by searching for the following files in the following order
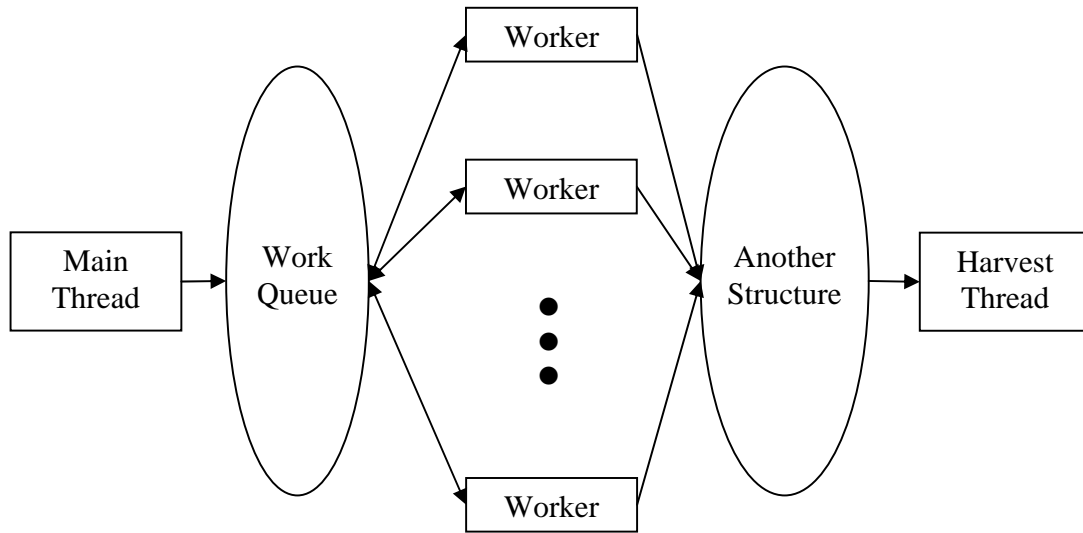
```
./x.h
kernel/x.h
/home/user/include/x.h
/usr/local/group/include/x.h
```

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.

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

I have provided you with a working, singly-threaded includeCrawler written in C; note that no attempt has been made to prevent memory leaks in this implementation. I have also provided extensive comments in includeCrawler.c with respect to the design of the application, along with three other modules that are needed to implement the program.[1]

---

[1] After you have built the includeCrawler executable from includeCrawler.c, the following commands should yield no output: % cd test; ../includeCrawler *.y *.l *.c | diff - output

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 pulling data 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 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 main thread determines 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 will indicate if it is thread-safe or not. Be sure to choose the right one[s]. 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 can encapsulate the unsafe class with thread-safe structures; you should look at java.util.concurrent first before building your own.

I have provided a sample folder with .c and .h files on Moodle, as well as the correct output that one should obtain from all versions of your program when applied to that folder.

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.

The marking scheme is appended to this handout.

5 Submission

You are to create a .tgz archive containing the following files:

- build.sh – a bash script that compiles all of your source files in the current directory

- includeCrawler.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.pdf or report.txt – a report in PDF or text file format describing the state of your solution

Assuming that your matric number is 1234567, your archive should be named 1234567.tgz. Other than the files listed above, there should be **NO** other files in the archive, and there should be **NO** folder names prefixing the names of the files. If I execute the following command, I should see a listing similar to the following:

```
$ tar –ztvf 1234567.tgz
-rwxr-xr-x Joe/None           10 2011-10-24 16:19 build.sh
-rw-r--r-- Joe/None          354 2011-10-24 16:19 includeCrawler.java
-rw-r--r-- Joe/None           18 2011-10-24 16:19 report.txt
```

You will submit your solutions electronically by attaching the .tgz file to an email to joseph.sventek@glasgow.ac.uk (Glasgow) or frankie.cha@glasgow.ac.uk (Singapore).

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 (Glasgow) or complete a paper "Own Work" form (Singapore).

We will be checking for collusion; better to turn in an incomplete solution that is your own than a copy of someone else's work. We have very good tools for detecting collusion. The penalty for collusion is to receive an H(0) for the assignment.

Marking Scheme for AP3, Exercise 2

Your submission will be marked on a 100 point scale. We 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 includeCrawler.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.

I have described three marking schemes 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	includeCrawler (+ other classes, if provided) 12 for workable solution (looks like it should work) 4 correct argument processing 2 for successful compilation 2 for successful compilation with no warnings 4 correct search path used for finding #include files 4 reasonable, concurrency-safe library class used for Work Queue 4 reasonable, concurrency-safe library class used for Another Structure 4 if it works correctly with the files in the test folder 4 if it works correctly with the files in an unseen folder of files

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

Points	Description
10	Your report – honestly describes the state of your submission
65	includeCrawler (+ other classes, if provided)
	24 for workable solution (looks like it should work)
	4 correct argument processing
	2 for successful compilation
	2 for successful compilation with no warnings
	4 correct search path used for finding #include files
	4 reasonable, concurrency-safe library class used for Work Queue
	4 reasonable, concurrency-safe library class used for Another Structure
	6 efficient mechanism for determination when worker thread has finished
	4 if it works correctly with the files in the test folder
	4 if it works correctly with the files in an unseen folder of files
	7 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	includeCrawler (+ other classes, if provided)
	36 for workable solution (looks like it should work)
	4 correct argument processing
	2 for successful compilation
	2 for successful compilation with no warnings
	4 correct search path used for finding #include files
	4 reasonable, concurrency-safe library class used for Work Queue
	4 reasonable, concurrency-safe library class used for Another Structure
	10 efficient mechanism for determination when worker threads have finished
	4 if it works correctly with the files in the test folder
	4 if it works correctly with the files in an unseen folder of files
	8 runtime performance with 1 worker on test folder is similar to single threaded implementation
	8 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 is offensive. 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 only part of the solution looks workable, then you will be awarded a portion of the points in that category.