University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Advanced Programming H			
Coursework Number	2			
Deadline	Time:	4:30pm	Date:	3rd December 2015
% Contribution to final course mark	10%			
Solo or Group ✓	Solo	✓	Group	
Anticipated Hours	20			
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 Dependency Discoverer

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 APH 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 dependencyDiscoverer, in a source file named dependencyDiscoverer.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 . dependencyDiscoverer [-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"
```

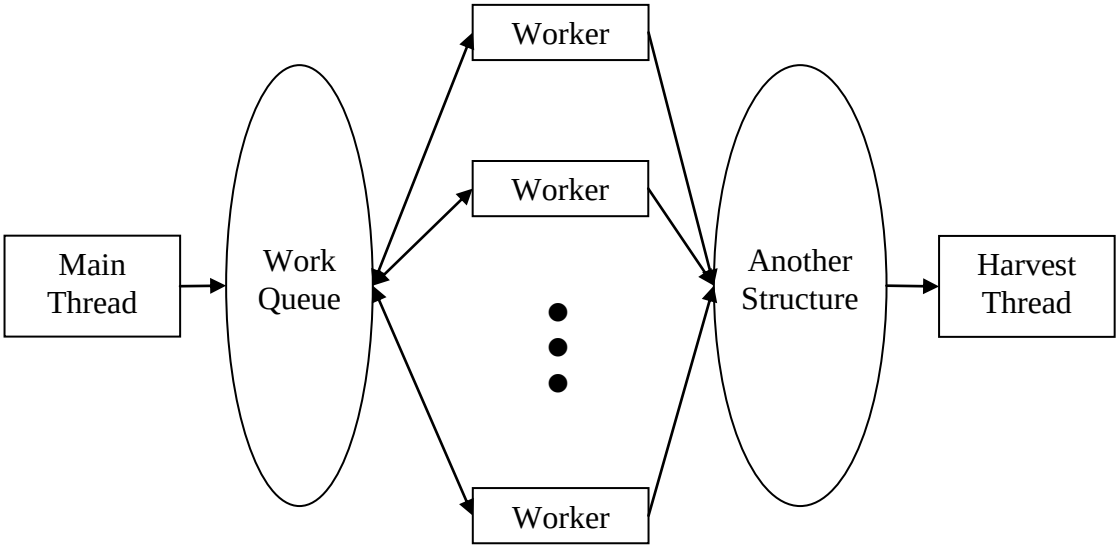
x.h will be located by searching for the following files in the following order

```
./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

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

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

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

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 will submit your solutions electronically by submitting a `.tgz` archive containing the following files on Moodle

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

- dependencyDiscoverer.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 – a report in PDF as specified below.

Assuming that your matric number is 1234567A, your archive should be named 1234567A.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.

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

- state your name, your login, and the title of the assignment (APH 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.

5.1 Report Contents

Your report should contain the following Sections.

1. **Status.** A brief report outlining the state of your solution, and indicating which of the single threaded, two threaded or multithreaded solutions you have provided. It is important that the report is accurate. For example it is offensive to report that everything works when it won't even compile.
2. **Build and sequential and 1 Thread runtimes.** A screenshot showing

   (a) The path where you are executing the program (i.e. pwd)

   (b) your crawler being built by your build.sh

   (c) the time to run your sequential crawler on all .c, .l and .y files in the test directory.

   (d) the time to run your threaded crawler (if you have one) with one thread.

   You'll need to use /usr/bin/time to obtain understandable times, and to pipe the output to a file to keep the screenshot manageable, e.g.

   ```
   bo720-4-02u(154 ^H)% /usr/bin/time java dependencyDiscoverer -Itest
   test/*.c test/*.l test/*.y > temp
   0.28user 0.02system 0:00.14elapsed 213%CPU (0avgtext+0avgdata
   38040maxresident)k
   0inputs+80outputs (0major+4767minor)pagefaults 0swaps
   bo720-4-02u(155 ^H)%
   ```

3. **Runtime with Multiple Threads.**

   3a Screenshot. A screenshot showing the path where you are executing the program (i.e. `pwd`), and the times to run the crawler with 1, 2, 3, 4, 6, and 8 threads on all .c, .l and .y files in the test directory.

   3b. Experiment with your multithreaded program, completing the following table of *elapsed* time and core utilization for 3 executions with different numbers of threads *on one of the UoG lab machines or UGS cloud instances*. Compute the median elapsed time and core utilization. To get reproducible results you should run on a lightly loaded machine.

   3c. Discussion. Briefly describe what you conclude from your experiment about (a) the benefits of additional cores for this input data (b) the variability of elapsed times.

CRAWLER_ THREADS	1		2		3		4		6		8	
	El. Time	Core Util.	El. Time	Core Util.	El. Time	Core Util.	El. Time	Core Util.	El. Time	Core Util.	El. Time	Core Util.
Execution 1												
Execution 2												
Execution 3												
Median												

6 Marking Scheme for APH, 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.

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 dependencyDiscoverer.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.

There are 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	dependencyDiscoverer (+ other classes, if provided) 12 for workable solution (looks like it should work) 4 correct argument processing 4 for successful compilation with no warnings 4 correct search path used for finding #include files 8 reasonable, concurrency-safe library class used for Work Queue and Another Structure 8 if it works correctly with the files in the test folder and 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	dependencyDiscoverer (+ other classes, if provided) 24 for workable solution (looks like it should work) 4 correct argument processing 4 for successful compilation with no warnings 4 correct search path used for finding #include files 8 reasonable, concurrency-safe library class used for Work Queue and Another Structure 6 efficient mechanism for determination when worker thread has finished 8 if it works correctly with the files in the test folder and an unseen folder of files 7 runtime performance is shown to be 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	dependencyDiscoverer (+ other classes, if provided) 36 for workable solution (looks like it should work) 4 correct argument processing 4 for successful compilation with no warnings 4 correct search path used for finding #include files 8 reasonable, concurrency-safe library class used for Work Queue and Another Structure 10 efficient mechanism for determination when worker threads have finished 8 if it works correctly with the files in the test folder and an unseen folder of files 8 runtime performance with 1 worker on test folder is shown to be similar to single threaded implementation 8 Sound experimentation with different numbers of threads, and analysis of the results.

Several things should be noted about the marking schemes:

- 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.