University of Glasgow | School of Computing Science

Assessed Coursework

Course Name	Systems Programming H			
Coursework Number	2: Concurrent Dependency Discoverer			
Deadline	Time:	17:00	Date:	30st November 2018
% Contribution to final course mark	10%			
Solo or Group ✓	Solo	✓	Group	
Anticipated Hours	15			
Submission Instructions	**As Specified in Section 4 below.**			
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://studentltc.dcs.gla.ac.uk/

for all coursework

Concurrent Dependency Discoverer

1 Requirement

Large-scale systems developed in C and C++ tend to include a large number of **".h"** files, both of a system variety (enclosed in `< >`) and non-system (enclosed in `" "`). The `make` utility and `Makefiles` are 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.

Many software systems are extremely large. It is difficult to keep the dependencies in the `Makefile` correct as many people concurrently make changes. Therefore, you will develop a program that can crawl over source files, noting any `#include` directives, and recurs 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 **C or C++**.

2 Specification

You are to create a program named `dependencyDiscoverer` based on a single-threaded template provided in either C or C++. The `main()` function 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: `./dependencyDiscoverer [-Idir] file1.ext [file2.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

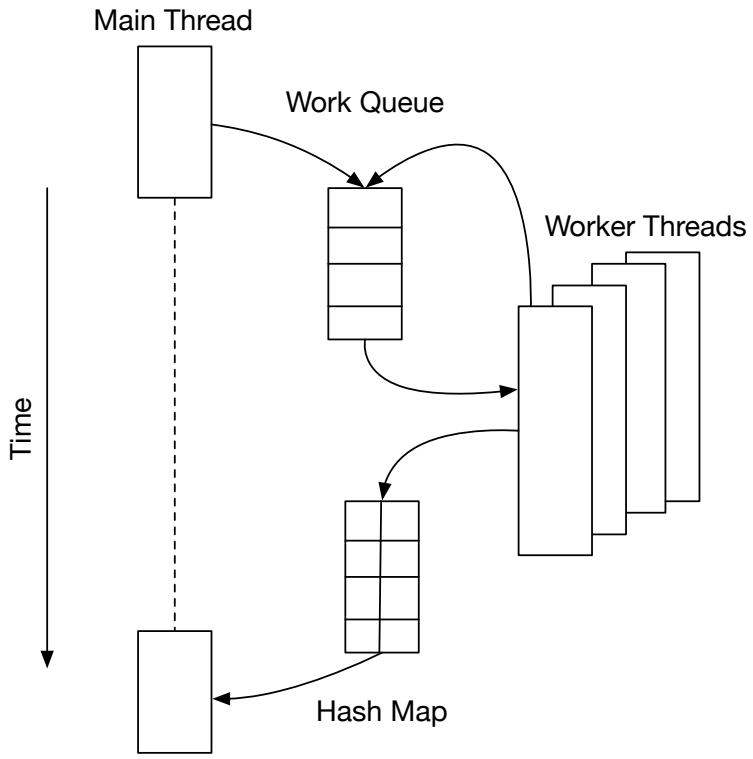
```
#include "x.h"
```

`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 should follow a structure as shown in the figure below.



The `Main Thread` manages the execution of the multiple `Worker Threads`. They communicate via two data structures: a `Work Queue` and a `Hash Map`. It should be possible to adjust the number of worker threads to process the accumulated work in order to speed up the processing. Since the `Work Queue` and the `Hash Map` are shared between threads, you will need to use concurrency control mechanisms to implement appropriate conditional critical regions around the shared data structures.

3.1 How to proceed

You are provided with a working, singly-threaded `dependencyDiscoverer` written in C or C++; note that no attempt has been made to prevent memory leaks in the C implementation. There are extensive comments in the source files which explain the design of the application in more detail.

You should build the program with the provided Makefile and you can then test it by running

```
cd test; ../dependencyDiscoverer *.y *.l *.c
```

This should produce an output identical to the provided output file, so that the following command should yield no output when the correct output is produced:

```
cd test; ../dependencyDiscoverer *.y *.l *.c | diff - output
```

The single threaded implementation is divided into three phases (highlighted in the `main` function):

1. populate the `Work Queue`
2. process the `Work Queue`, placing the processed data in the `Hash Map`
3. harvest the data in the `Hash Map`, printing out the results

You should choose one version – either the C or the C++ version – to start from to develop a multi-threaded version. You should first develop a version which still uses only a single thread but protects the accesses to the shared data structures with appropriate synchronization mechanisms.
For the C version you can modify the implementation of the data structures directly.
For the C++ version you can create your own data structures which encapsulate the used C++ standard containers by creating a `struct` which stores the container as a member alongside the synchronization utilities and provides a similar interface to the container while providing appropriate synchronization.

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. You will have to design a solution so that the main thread can determine that all of the worker threads have finished (without busy waiting) so it can harvest the information.

Use the documentation at `en.cppreference.com` to learn about the usage of the C++ standard containers used in the C++ implementation.

4 Submission

You will submit your solutions electronically via moodle.

- all source file of your solution
- a `Makefile` which is configured to compile your solution with a call to `make` and which produces your program which must be called `dependencyDiscoverer`
  (The provided `Makefile` in the template is configured to behave like this)
- `report.pdf` – a report in PDF as specified below.

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

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

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 software plagiarism, e.g. JPLAG or MOSS.

4.1 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 compiled either manually by a sequence of commands, or by a Makefile

   (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 -p` 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 -p ./dependencyDiscoverer -Itest
test/*.c test/*.l test/*.y > temp
real   0.28
user   0.02
sys    0:00.14
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 for 3 executions with different numbers of threads *on one of the UoG lab machines.* Compute the median elapsed time. 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_	1	2	3	4	6	8
THREADS	Elapsed Time	Elapsed Time	Elapsed Time	Elapsed Time	Elapsed Time	Elapsed Time
Execution 1						
Execution 2						
Execution 3						
Median						

5 Marking Scheme

Your submission will be marked on a 100-point scale. There is substantial emphasis on **WORKING** submissions, and you will note that a large fraction of the points is reserved for this aspect. It is to your advantage to ensure that whatever you submit compiles, links, 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.

Points	Description
10	Your report – accurately, clearly and honestly describes the state of your submission
90	dependencyDiscoverer (+ other source files, if provided) workable solution (looks like it should work): 16 marks if sequential 21 marks if 1 worker thread 40 marks if multiple workers 4 for successful compilation with no warnings 10 appropriate thread safety implemented for Working Queue and Hash Map 10 efficient mechanism for determination when worker threads have finished: 0 if sequential 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: 0 if sequential 10 sound experimentation with different numbers of threads, and analysis of the results: 0 if sequential or 1 worker thread.

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.