Concurrent include file crawler

Due at 11:59pm on Friday, 9 June 2017

1 Background

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 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 git or 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 make 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, exploiting the concurrency features of PThreads.

2 Specification

You are to create a file named include_crawler.c. Your program 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: ./include_crawler [-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

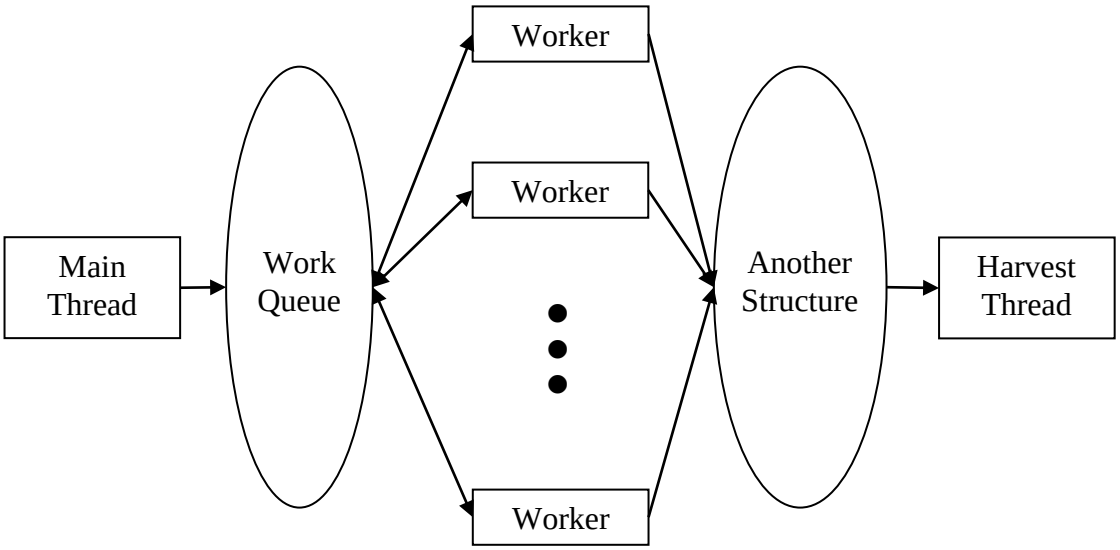
#include "x.h"

x.h will be located by searching for the following files in the following order, stopping as soon as one is found.

./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 to speed up the processing. Since the Work Queue and the Another Structure are shared between threads, you will need to use PThread'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; for each file retrieved from the Work Queue, scan the file for #include "…" lines; if found, add the included filename to the Work Queue and update the dependency for the file being processed 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 include_crawler written in Python; I have also provided linkedlist.py, as the program needs to use a LinkedList class. I have also provided design.pdf, that provides extensive comments about the design of the application.

After you have a singly-threaded C version working correctly[1], 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 the worker threads have finished (without busy waiting and without loss of concurrency) so it can harvest the information.

4 Hints

I have also provided a directory (named test) with a number of .y, .l, and .c files. The starting archive places this directory in the same directory as include_crawler.py. If you execute the following commands:

```
$ cd test
$ python3 ../include_crawler.py *.y *.l *.c | diff - output
```

you should see no output – i.e., it has produced the same dependencies as those found in the file named output.

This same technique can be used to test your single-threaded include_crawler.c, and your multi-threaded include_crawler.c.

4.1 Python3 installation in your Arch Linux virtual machine

The Arch Linux image provided to you for your virtual machine does ***not*** have python3 installed. You must, therefore, perform the following commands in order to be able to execute include_crawler.py:

```
# Download https://www.python.org/ftp/python/2.6.1/Python-3.6.1.tgz
# using Firefox; it will be downloaded to ~/Downloads
$ cd /usr/local/src
$ sudo mv ~/Downloads/Python-3.6.1.tgz .
$ sudo tar -zxvf Python-3.6.1.tgz
$ cd Python-3.6.1
$ sudo ./configure  # generates Makefile tuned to your system
$ sudo make         # makes python image and libraries
$ sudo make test    # takes awhile (7+ mins), runs 405 tests
$ sudo make install # places python3 in /usr/local/bin – make sure
                    # this is in your search path
```

5 Developing Your Code

You ***must*** develop your code in Linux running inside the virtual machine image provided to you. This gives you the benefit of taking snapshots of system state right before you do something

[1] You should feel free to use one or more of the abstract data types available from https://github.com/jsventek/ADTs that correspond to the data structures used in the Python version.

potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state.

You should use your Bitbucket GIT repositories for keeping track of you programming work. As a reference, you can perform the command line steps below to create a new project directory and upload it to your uoregon-cis415 repository.

```
% cd /path/to/your/uoregon-cis415
% mkdir extracredit
% echo "This is a test file." >extracredit/testFile.txt
% git add extracredit
% git commit –m "Initial commit of extracredit project"
% git push –u origin master
```

Any subsequent changes or additions can be saved via the add, commit, and push commands.

6 Submission[2]

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

- Makefile[3] – a makefile that compiles all of your source files in the current directory and produces an executable named include_crawler
- include_crawler.c – the source file as described above
- (other .c and .h files) – if your source file depends upon any other source files that you have defined or imported, you must include them here
- report.pdf or report.txt – a report in PDF or text file format describing the state of your solution

You will submit your solutions electronically by uploading a gzipped tar archive via Canvas.

Your TGZ archive should be named "<duckid>-extracredit.tgz", where "<duckid>" is your duckid.

These files should be contained in a folder named "<duckid>". Thus, if you upload "jsventek-extracredit.tgz", then we should see the following when we execute the following command:

```
% tar –ztvf jsventek-extracredit.tgz
-rw-rw-r-- joe/None    1234 2015-03-30 16:30 jsventek/Makefile
-rw-rw-r-- joe/None    5125 2015-03-30 16:37 jsventek/include_crawler.c
-rw-rw-r-- joe/None  629454 2015-03-30 16:30 jsventek/workqueue.h
-rw-rw-r-- joe/None  629454 2015-03-30 16:30 jsventek/workqueue.c
-rw-rw-r-- joe/None  629454 2015-03-30 16:30 jsventek/anotherstruct.h
-rw-rw-r-- joe/None  629454 2015-03-30 16:30 jsventek/anotherstruct.c
-rw-rw-r-- joe/None  629454 2015-03-30 16:30 jsventek/report.pdf
```

[2] If the TGZ archive is not in the correct format, ***I will simply not mark your submission***. See the section in the Project 0 handout for the appropriate way to create the TGZ archive if you need a refresher.

[3] You must test your submission on the Linux virtual machine. If your code does not compile there, or does not link there, ***I will simply not mark your submission***.

Each of your source files must start with an “authorship statement”, contained in C comments, as follows:

- state your name, your duckid, and the title of the assignment (CIS 415 Extra Credit)
- state either “This is my own work.” or “This is my own work except that …”, as appropriate.

Marking Scheme for CIS 415 Extra Credit Project

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.

You must be sure that your code works correctly on the Arch Linux virtual machine, regardless of which platform you use for development and testing. Leave enough time in your development to fully test on the virtual machine before submission.

Points	Description
10	Your report – honestly describes the state of your submission
90	include_crawler.c (+ other files, 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 C implementation (mine) 8 runtime performance on unseen 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.