Programming Assignment 4:
DNS Name Resolution Engine

CSCI340 - Operating Systems
California State University, Chico
Due Date: Sunday, October 26, 2014 11:59pm
Adapted from University of Colorado at Boulder CSCI3753 Assignment

Fall 2014

1 Introduction

In this assignment you will develop a multi-threaded application that resolves domain names to IP addresses, similar to the operation preformed each time you access a new website in your web browser. The application is composed of two sub-systems, each with one thread pool: requesters and resolvers. The sub-systems communicate with each other using a bounded queue.

This type of system architecture is referred to as a Producer-Consumer architecture. It is also used in search engine systems, like Google. In these systems, a set of crawler threads place URLs onto a queue. This queue is then serviced by a set of indexer threads which connect to the websites, parse the content, and then add an entry to a search index. Refer to Figure 1 for a visual description.

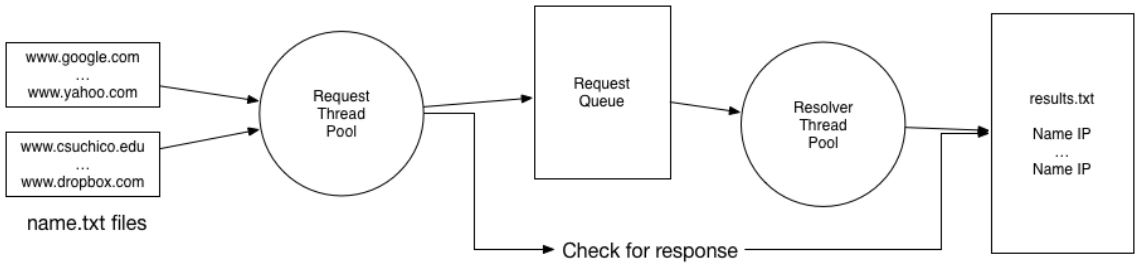

Figure 1: System Architecture

2 Description

2.1 Name Files

Your appliction will take as input a set of name files. Names files contain one hostname per line. Each name file should be serviced by a single requester thread from the requester thread pool.

2.2 Requester Threads

The requester thread pool services a set of name files, each of which contains a list of domain names. Each name that is read from each of the files is placed on a FIFO queue. If a thread tries to write to the queue but finds that it is full, it should sleep for a random period of time between 0 and 100 microseconds.

2.3 Resolver Threads

The second thread pool is comprised of a set of `THREAD_MAX` resolver threads. The resolver thread pool services the FIFO queue by taking a name off the queue and querying its IP address. After the name has been mapped to an IP address, the output is written to a line in the results.txt file in the following format:

```
www.google.com,74.125.224.81
```

2.4 Synchronization and Deadlock

Your application should synchronize access to shared resources and avoid deadlock. You should use mutexes and/or semaphores to meet this requirement. There are at least two shared resources that must be protected: the queue and the output file. Neither of these resources is thread-safe by default.

2.5 Ending the Program

Your program must end after all the names in each file have been serviced by the application. This means that all the hostnames in all the input files have received a corresponding line in the output file.

3 What's Included

Some files are included with this assignment for your benefit. You are not required to use these files, but they may prove helpful.

1. **queue.h** and **queue.c** These two files implement the FIFO queue data structure. The queue accepts pointers to arbitrary types. You should probably use the queue to store pointers to C-strings of hostnames. The requester threads should push these hostnames into the queue, and the resolver threads should obtain hostnames from the same queue.

Please consult the *queue.h* header file for more detailed descriptions of each available function.

2. **queueTest.c** This program runs a series of tests to confirm that the queue is working correctly. You may use it as an example of how to use the queue, or to test the functionality of the queue code that you are provided.

3. **util.c** and **util.h** These two files contain the DNS lookup utility function. This function abstracts away a lot of the complexity involved with performing a DNS lookup. The function accepts a hostname as input and generates a corresponding dot-formatted IPv4 IP address string as output.

   Please consult the *util.h* header file for more detailed descriptions of each available function.

4. **input/names*.txt** This is a set of sample name files. They follow the same format as mentioned earlier. Use them to test your program.

5. **results-ref.txt** This result file is a sample output of the IPs for the hostnames from all the *names*.txt* files used as input.

6. **lookup.c** This program represents an un-threaded solution to this assignment. Feel free to use it as a starting point for your program, or as a reference for using the utility functions and performing file i/o in C.

7. **pthread-hello.c** A simple threaded "Hello World" program to demonstrate basic use of the pthread library.

8. **Makefile** A GNU Make makefile to build all the code.

4 Additional Specifications

Many of the specifications for your program are embedded in the descriptions above. This section details additional specifications to which you must adhere.

4.1 Program Arguments

Your executable program should be named "multi-lookup". When called, it should interpret the last argument as the file path for the file to which results will be written. All proceeding arguments should be interpreted as input files containing hostnames in the aforementioned format.

An example call involving three input files might look like:

```
multi-lookup names1.txt names2.txt names3.txt result.txt
```

4.2 Limits

If necessary, you may impose the following limits on your program. If the user specifies input that would require the violation of an imposed limit, your program should gracefully alert the user to the limit and exit with an error.

- **MAX_INPUT_FILES**: 10 Files (This is an optional upper-limit. Your program may also handle more files, or an unbounded number of files, but may not be limited to less than 10 input files.)
- **MAX_RESOLVER_THREADS**: 10 Threads (This is an optional upper-limit. Your program may also handle more threads, or match the number of threads to the number of processor cores.)
- **MIN_RESOLVER_THREADS**: 2 Threads (This is a mandatory lower-limit. Your program may handle more threads, or match the number of threads to the number of processor cores, but must always provide at least 2 resolver threads.)
- **MAX_NAME_LENGTH**: 1025 Characters, including null terminator (This is an optional upper-limit. Your program may handle longer names, but you may not limit the name length to less than 1025 characters.)
- **MAX_IP_LENGTH**: INET6_ADDRSTRLEN (This is an optional upper-limit. Your program may handle longer IP address strings, but you may not limit the name length to less than INET6_ADDRSTRLEN characters including the null terminator.)

4.3 Error Handling

You must handle the following errors in the following manners:

- **Bogus Hostname**: Given a hostname that can not be resolved, your program should output a blank string for the IP address, such that the output file contines the hostname, followed by a comma, followed by a line return. You should also print a message to stderr alerting the user to the bogus hostname.
- **Bogus Output File Path**: Given a bad output file path, your program should exit and print an appropriate error to stderr.
- **Bogus Input File Path**: Given a bad input file path, your program should print an appropriate error to stderr and move on to the next file.

All system and library calls should be checked for errors. If you encounter errors not listed above, you should print an appropriate message to stderr, and then either exit or continue, depending upon whether or not you can recover from the error gracefully.

5 External Resources

You may use the following libraries and code to complete this assignment, as well as anything you have written for this assignment:

- Any functions listed in util.h
- Any functions listed in queue.h
- Any functions in the C Standard Library
- Standard Linux pthread functions
- Standard Linux Random Number Generator functions
- Standard Linux file i/o functions

If you would like to use additional external libraries, you must clear it with me first. You will not be allowed to use pre-existing thread-safe queue or file i/o libraries since the point of this assignment is to teach you how to make non-thread-safe resources thread-safe.

6 What You Must Provide

To receive full credit, you must submit the following items to Turnin by the due date.

- **multi-lookup.c**: Your program, conforming to the above requirements
- **multi-lookup.h**: A header file containing prototypes for any function you write as part of your program.

7 Extra Credit

There are a few options for receiving extra credit on this assignment. Completion of each of the following items will gain you 5 points of extra credit per item. In no case will the maximum score on this assignment exceed 110/100. If you alter any files other than `multi-lookup.c` and `multi-lookup.h` to accomplish the extra credit submit your solution to me via email by taring up your project folder and sending me the tar of files.

- **Multiple IP Addresses**: Many hostnames return more than a single IP address. Add support for listing an arbitrary number of addresses to your program. These addresses should be printed to the output file as additional comma-separated strings after the hostname. For example:

  ```
  www.google.com,74.125.224.81,76.125.232.80,75.125.211.70
  ```

You may find it necessary to modify code in the util.h and util.c files to add this functionality. If you do this, please maintain backwards compatibility with the existing util.h functions. This is most easily done by adding new function instead of modifying the existing ones.

- **IPv6 Support and Testing**: Add support for IPv6 IP addresses and find an IPv6 aware environment where you can test this support. Since IPv6 is relatively new, finding an environment for testing this support is probably harder than adding it. You must be able to demonstrate IPv6 support during your grading session to receive credit for this item. You may find it necessary to modify code in util.h and util.c to complete this item. If you do so, please maintain backward compatibility with the existing code.
- **Matching Number of Threads to Number of Cores**: Make your program dynamically detect the number of cores available on a system and set the number of resolver threads to take into account the number of cores.
- **Full-Loop Lookups**: Make each requester thread query the output file every 250ms to detect when each of its requests have been filled. Requester threads should print a message to the user with the IP address of each hostname, and should not exit until all of each threads requests have been satisfied.
- **Benchmarks**: Determine the ideal number of resolver threads for a given processor core count. Provide benchmark data backing up your determination. Include this documentation in your README.

8 Grading

To received full credit your program must:

- Meet all requirements elicited in this document
- Build with "-Wall" and "-Wextra" enabled, producing no errors or warnings.
- Run without leaking any memory, as measured using valgrind
- Document any resources you use to solve your assignment in the header comment of your file
- Include your name in the header comment of your file

This includes adhering to good coding style practices. To verify that you do not leak memory, I may use *valgrind* to test your program. To install *valgrind*, use the following command:

```
sudo apt-get install valgrind
```

And to use *valgrind* to monitor your program, use this command:

```
valgrind ./pa2main text1.txt text2.txt ..... textN.txt results.txt
```

Valgrind should report that you have freed all allocated memory and should not produce any additional warnings or errors.

You can write your code in any environment you like. But you have to make sure that your programs can be compiled and executed on Ubuntu 14.04.

9 References

Refer to your textbook and class notes for descriptions of producer/consumer and reader/writer problems and the different strategies used to solve them.

The Internet is also a good resource for finding information related to solving this assignment.

You may wish to consult the man pages for the following items, as they will be useful and/or required to complete this assignment. Note that the first argument to the "man" command is the chapter, insuring that you access the appropriate version of each man page. See example `man man` for more information.

- man 7 pthreads
- man 3 pthread_create()
- man 3 pthread_join()
- man 3 pthread_mutex_init()
- man 3 pthread_mutex_destroy()
- man 3 pthread_mutex_lock()
- man 3 pthread_mutex_unlock()
- man 3 fopen()
- man 3 fclose()
- man 3 fprintf()
- man 3 fscanf()
- man 3 usleep()
- man 3 random()
- man 3 perror()
- man 1 valgrind