PA3

Re-submit Assignment

Due Sunday by 11:59pm **Points** 100 **Submitting** a file upload **File Types** txt

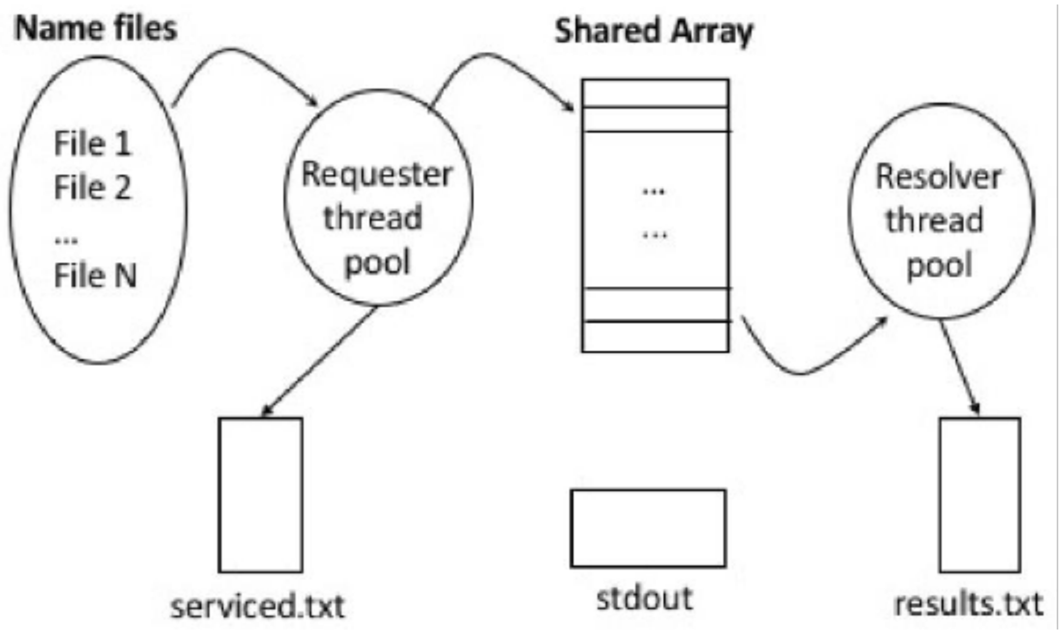
Programming Assignment Three

Due Date and Time: Sunday, November 1st, 2020 @11:59PM

Make sure you download PA3.zip before writing any code

Introduction

In this assignment you will develop a multi-threaded application, written in C, that resolves domain names to IP addresses. This is similar to the operation performed each time you access a new website in your web browser. The application will utilize two types of worker threads: requesters and resolvers. These threads communicate with each other using a shared array, also referred to as a bounded buffer. The diagram below outlines the desired configuration:



We will provide some number of input files that contain one hostname per line. Your program will process each of these files using a single requester thread per file. The number of concurrently executing requester threads is determined by a command line argument and may be less than or equal to the total number of input files. A requester thread will read a single file line-by-line, place the hostname into the shared array, and log this hostname into the file **./serviced.txt**.

You will also create some number of resolver threads, again determined by a command line argument, that will remove hostnames from the shared array, lookup the IP address for that hostname, and write the results to the file **./results.txt**.

Once all the input files have been processed the requester threads will terminate. Once all the hostnames have been looked up, the resolver threads will terminate and the program will end. Just prior to termination, your program will print (to standard out) the total time that it took to process all the data.

Man Page for Multi-Lookup

```
NAME

multi-lookup - resolve a set of hostnames to IP addresses

SYNOPSIS

multi-lookup <# requester> <# resolver> <requester log> <resolver log> [ <data file> ... ]

DESCRIPTION

The file names specified by <data file> are passed to the pool of requester threads which place i
nformation into a shared data area. Resolver threads read the shared data area and find the corre
sponding IP address.

<# requesters> number of requestor threads to place into the thread pool.

<# resolvers> number of resolver threads to place into the thread pool.

<requester log> name of the file into which all the requester status information is written.

<resolver log> name of the file into which all the resolver status information is written.

<data file> filename to be processed. Each file contains a list of host names, one per line, that
are to be resolved.
```

Requester Threads

Your application will take as input the number of requester threads. These threads service a set of text files as input, each of which contains a list of hostnames. Each name that is read from each of the files is placed on the shared array. If a requester thread tries to write to the array but finds that it is full, it should block until a space opens up in the array. After servicing an input file, a requester thread checks if there are any remaining input files left to service. If so, it requests one of the remaining files next. This process goes on until all input files have been serviced. If there are no more input files remaining, the thread writes the number of files it serviced in a new line in **./serviced.txt** in the following format:

```
Thread <thread id> serviced ### files.
```

To get the id of a thread on Linux systems, use **gettid( )**, or more preferably **pthread_self()** for a POSIX compliant implementation.

Resolver Threads

The second thread pool is comprised of a set of resolver threads. The resolver thread consumes the shared array by taking a name off the array 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
```

If the resolver is unable to find the IP address for a hostname, it should leave the field after the comma blank. If a resolver thread tries to read from the array but finds that it is empty, it should block until there is a new item in the array or all names in the input files have been serviced.

Synchronization and Deadlock

Your application should synchronize access to shared resources and avoid any deadlock or busy wait. You should use some combination of **mutexes, semaphores and/or condition variables** to meet this requirement. There are at least three shared resources that must be protected: the **shared array, serviced.txt** and **results.txt**. <u>**None of these resources is thread-safe by default.**</u>

Ending the Program

Your program must end after all names in each input 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. Just prior to exiting, your program must print the total runtime to standard out:

```
./multi-lookup: total time is 2.420237 seconds
```

Use the **gettimeofday( )** **(https://man7.org/linux/man-pages/man2/gettimeofday.2.html)** library function for this purpose.

Limits

You should 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. You should define these as macros in the appropriate .h file(s).

- **ARRAY_SIZE:** 20 Elements. This is the size of the shared array used by the requester and resolver threads to communicate.
- **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)
- **MAX_RESOLVER_THREADS**: 10 Threads. This is an upper-limit. Your program may also handle more
- **MAX_REQUESTER_THREADS**: 5 Threads. This is an upper-limit. Your program may also handle more
- **MAX_NAME_LENGTH**: 1025 Characters, including the 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)
- **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)

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 contains the hostname, followed by a comma, followed by a newline 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.

What's included in PA3.zip?

Some **files** are included with this assignment for your benefit:

- **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.
- **Makefile**: A simple makefile that will allow you to compile and link your implementation. We've also included an option that will bundle your code into a file suitable for submission to Canvas. Simply type **make submit** and follow the prompts.
- **input/names*.txt**: A set of sample input files to test your program with.

External Resources

You may use the following libraries to complete this assignment:

- Any functions listed in the provided util.h
- The C Standard Library (**stdlib.h** **(https://man7.org/linux/man-pages/man0/stdlib.h.0p.html)**)
- The C String Library (**string.h** **(https://man7.org/linux/man-pages/man3/string.3.html)**)
- Linux **pthread** **(https://man7.org/linux/man-pages/man7/pthreads.7.html)** and **semaphore** **(https://man7.org/linux/man-pages/man7/sem_overview.7.html)** libraries
- The Standard I/O Library (**stdio.h** **(https://man7.org/linux/man-pages/man3/stdio.3.html)**)

You will not be allowed to use pre-existing thread-safe queue or file i/o libraries. One of the goals of this assignment is to teach you how to make non-thread-safe resources thread-safe. Specifically, when you are interacting with these data resources, you will need to look up whether the functions you want to use are thread safe or not. You will find a helpful page **here** **(http://man7.org/linux/man-pages/man7/attributes.7.html)** that formally defines MT-safe code. You can look for this safety value on man pages for the functions you are interested in using. If you are unsure about using a particular library or function, please ask.

What you must submit

To receive credit, you must submit the following items to Canvas by the due date. Please use **make submit** to prepare your code for submission. The following files are expected:

- **multi-lookup.c**: Your program, conforming to the above
- **multi-lookup.h**: A header file containing prototypes for any function you write as part of your program
- Any additional .c or .h files that you've authored for this application
- **Makefile**: The provided makefile that builds your program
- **README**: A readme briefly describing any .c or .h files included in your solution
- **performance.txt**: Run your program over the files provided in the input directory for the following scenarios:
  - 1 requester thread and 1 resolver thread
  - 1 requester thread and 3 resolver threads
  - 3 requester threads and 1 resolver thread
  - 3 requester threads and 3 resolver threads
  - 5 requester threads and 5 resolver threads
- For each of the above scenarios, provide the following information in a file named performance.txt. Use the following example to format your file:

```
Number of requester threads is 4
Number of resolver threads is 5
Thread 0a5f4700 serviced 8 files
Thread 08df1700 serviced 7 files
Thread 09df3700 serviced 7 files
Thread 095f2700 serviced 8 files
./multi-lookup: total time is 8.824480 seconds
```

Grading

Implementation: **50%** of your grade will be based on the performance of the implementation that you provide, and the quality of your submitted code. We will test your implementation on a number of test cases.

Interview: **50%** of your grade will be based on your interview. You will be expected to explain your work and answer additional questions. You are expected to understand all concepts pertaining to the assignment.

To receive full credit your program **must**:

- meet all requirements outlined in this writeup
- build with “-Wall” and “-Wextra” enabled, producing no errors or warnings
- run without leaking any memory, as measured using ***valgrind***
- not use pre-existing thread safe queues or i/o functions
- not use global variables for worker threads, instead pass parameters via pthread_create()'s ***void *arg*** parameter

If your code does not compile and run in the standard Ubuntu VM, you will not get any points for the Implementation part!

Valgrind

To verify that you do not leak memory, the TAs will use *valgrind* to test your program. Your VM should already have *valgrind.* If not, use the following command to install it:

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

To use *valgrind* to evaluate your program, simply prepend it to your usual invocation of multi-lookup, for example:

```
valgrind ./multi-lookup 5 5 serviced.txt results.txt input/names1*.txt
```

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

```
==21728== Memcheck, a memory error detector
==21728== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==21728== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==21728== Command: ./multi-lookup 5 5 serviced.txt results.txt input/names11.txt input/names12.tx
t input/names13.txt input/names14.txt input/names15.txt input/names1.txt
==21728==
./multi-lookup: total time is 2.420237 seconds
==21728==
==21728== HEAP SUMMARY:
==21728==     in use at exit: 0 bytes in 0 blocks
==21728==   total heap usage: 1,723 allocs, 1,723 frees, 9,394,035 bytes allocated
==21728==
==21728== All heap blocks were freed -- no leaks are possible
==21728==
==21728== For counts of detected and suppressed errors, rerun with: -v
==21728== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

For comparison, here's output from valgrind when run with a "leaky" program:

```
==22400== HEAP SUMMARY:
==22400==     in use at exit: 7,380 bytes in 618 blocks
==22400==   total heap usage: 8,115 allocs, 7,497 frees, 45,674,418 bytes allocated
==22400==
==22400== LEAK SUMMARY:
==22400==    definitely lost: 7,380 bytes in 618 blocks
==22400==    indirectly lost: 0 bytes in 0 blocks
==22400==      possibly lost: 0 bytes in 0 blocks
```

```
==22400==    still reachable: 0 bytes in 0 blocks
==22400==         suppressed: 0 bytes in 0 blocks
==22400== Rerun with --leak-check=full to see details of leaked memory
```

References

https://man7.org/linux/man-pages/man7/pthreads.7.html (https://man7.org/linux/man-pages/man7/pthreads.7.html)

https://man7.org/linux/man-pages/man7/sem_overview.7.html (https://man7.org/linux/man-pages/man7/sem_overview.7.html)

https://www.valgrind.org/docs/manual/mc-manual.html (https://www.valgrind.org/docs/manual/mc-manual.html)

Modular programming in C (https://www.icosaedro.it/c-modules.html)