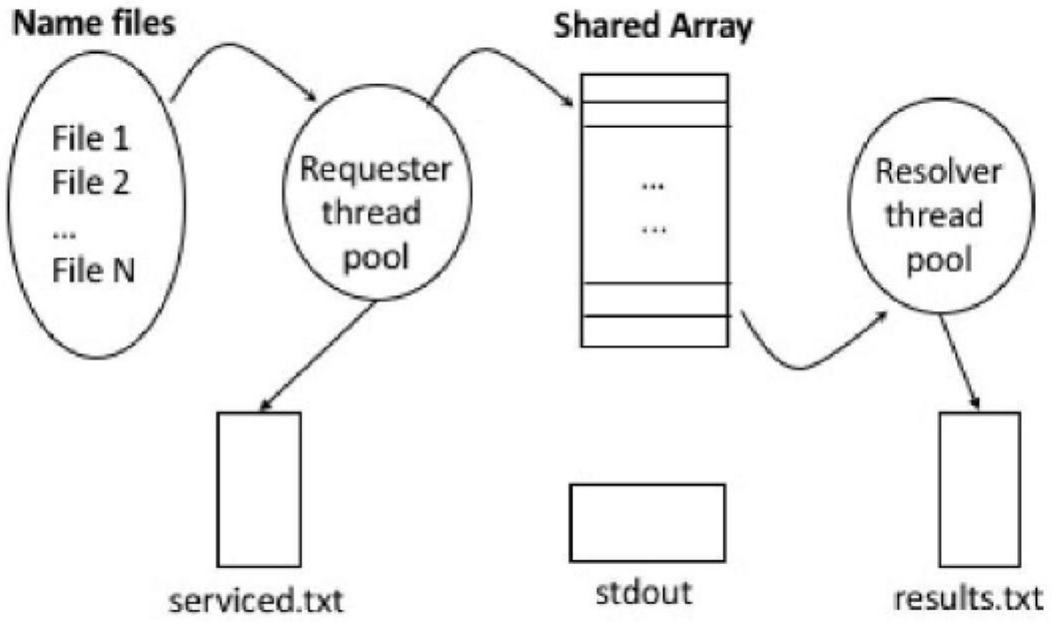
CSCI 3753: Operating Systems
Spring 2023
Programming Assignment Three

<u>Due Date and Time: 11:55 PM, Friday March 24, 2023</u>

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**, sometimes referred to as a bounded buffer. The diagram below outlines the desired configuration:



We will provide a 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, where each thread will read the file line-by-line, place the hostname into the shared array, and write this hostname into the requester logfile.

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 resolver logfile.

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.

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 information into a sh ared data area. Resolver threads read the shared data area and find the corresponding 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 requested hostnames are written
<resolver log> name of the file into which hostnames and resolved IP addresses are written
<data file> filename to be processed. Each file contains a list of host names, one per line, that are to be resolved

SAMPLE INVOCATION
```
./multi-lookup 5 5 serviced.txt resolved.txt input/names*.txt
```

SAMPLE CONSOLE OUTPUT
```
thread d2fa8700 serviced 6 files in 0.200960 seconds
thread d37a9700 serviced 6 files in 0.201183 seconds
thread d27a7700 serviced 6 files in 0.201345 seconds
thread d3faa700 serviced 6 files in 0.201562 seconds
thread d47ab700 serviced 6 files in 0.201787 seconds
thread d4fac700 resolved 177 hosts in 0.202039 seconds
thread d57ad700 resolved 89 hosts in 0.202088 seconds
thread d67af700 resolved 171 hosts in 0.202204 seconds
thread d6fb0700 resolved 32 hosts in 0.202244 seconds
thread d5fae700 resolved 137 hosts in 0.202215 seconds
./multi-lookup: total time is 0.202741 seconds
```

Requester Threads

Your application will take an argument for the number of requester threads. This pool of threads service a set of text files as input, each of which contains a list of hostnames. Each name read should be 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 to stdout 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. A resolver thread consumes the shared array by taking a name off the array and resolving its IP address. After the name has been mapped to an IP address, the output is written to a line in the resolver logfile in the following format:

```
google.com, 74.125.224.81
```

If the resolver is unable to find the IP address for a hostname, it should leave *NOT_RESOLVED* instead of an IP address:

```
glsdkjf.com, NOT_RESOLVED
```

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. If there are no more hostnames remaining, the thread writes the number of names it successfully resolved to stdout in the following format:

```
thread <thread id> resolved ### hostnames
```

Shared Array

The thread-safe shared array will require you to implement a solution to the Bounded Buffer Problem as discussed in lecture and the textbook. The shared array must be built on top of a single, contiguous, linear array of memory. You can organize this array as a queue (FIFO), circular queue, or stack (LIFO). We will not accept solutions that rely on linked lists, trees, dictionaries, etc. **You will receive a zero for the coding portion of the assignment if you don't use a linear array of memory.**

Synchronization, Deadlock & Starvation

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 several shared resources that must be protected: the shared array, the logfiles, stdout/stderr and argc/argv[]. **None of these resources are thread-safe by default.**

We will be measuring the runtime of your solution to determine if it's experiencing any sort of starvation. If you misuse a synchronization mechanism and essentially create a serialized solution (where your threads run in sequence rather than in parallel), your grade will be subject to a significant deduction.

Please also note that using delays, like calls to usleep(), to get synchronization to work are not robust and will incur a significant penalty.

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 will print the total runtime to standard out:

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

We recommend using the **gettimeofday( )** library function for this purpose.

Limits

You should define several C macros that impose the following limits on your program. If the user specifies input that would violate an imposed limit, your program should gracefully alert the user to the limit and exit with an error and an appropriate return value. You should define these macros in the corresponding .h file(s).

- **ARRAY_SIZE: 8 -** Number of elements in the shared array used by the requester and resolver threads to communicate
- **MAX_INPUT_FILES**: **100 -** Maximum number of hostname file arguments allowed
- **MAX_REQUESTER_THREADS**: **10 -** Maximum number of concurrent requester threads allowed
- **MAX_RESOLVER_THREADS**: **10 -** Maximum number of concurrent resolver threads allowed
- **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 -** INET6_ADDRSTRLEN is the maximum size IP address string util.c will return

Error Handling

Your program must accept arguments as detailed in the above manual page. Your program should deal with the following conditions:

- Missing arguments - terminate with a usage synopsis to stdout
- Arguments that are out of range (i.e. too many input files) - terminate with an error message to stderr
- If a log file exists and is writeable - you should overwrite it
- If a log file doesn't exist - it should be created by your program
- Missing or unreadable input files - print ***invalid file <filename>*** to stderr, then move onto 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. **Make sure your program returns an appropriate exit status**.

What's included in pa3.zip?

Some files in pa3.zip 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 string as input and returns a corresponding dot-formatted IPv4 IP address string
  - Please consult the util.h header file for more detailed descriptions of each available function
  - Do not modify these files, we will not be collecting them, and will build your code with our standard versions
- **Makefile**: A simple makefile to compile and link your implementation.
  - Type **make** to build your code, **make clean** to cleanup any extraneous .o files or executables
  - Type **make submit** and follow the prompts to bundle your code into a form suitable for submission to Canvas
  - You must use our makefile for both compilation and submission
  - We've included comments in the makefile on where to add any additional .c or .h files you've authored
- **input/names*.txt**: A set of sample input files to test your program with
  - You may copy or combine these files in order to test against more data

External Resources

You may use the following libraries/functions to complete this assignment:

- Any functions listed in the provided util.h
- The C Standard Library
- The C String Library
- Linux pThread and semaphore libraries
- The Standard I/O Library

- Any other functions we've explicitly directed you to use in this writeup

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 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 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 type **make submit** in your PA3 project directory to prepare your code for submission. You will be prompted for your identikey name, and when complete, the make script should produce a filename like *PA3-abcd1234.txt*. This is the file you will submit.

The following files will be bundled for submission:

- **Makefile**: The makefile we provided that builds your program
- **README**: Include any comments or instructions about your code here
- **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 and included in the appropriate section of the Makefile

Grading

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

- compile and run in our standard Cloud VM (Ubuntu 16.04)
- build with “-Wall” and “-Wextra” enabled, producing no errors or warnings
- run without leaking any memory (see the section on ***valgrind*** below)
- 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
- not use sleep() or other delays
- not experience deadlock or starvation

Valgrind

To verify that you do not leak memory, we 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.txt 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.htmlLinks to an external site.

https://man7.org/linux/man-pages/man7/sem_overview.7.htmlLinks to an external site.

https://www.valgrind.org/docs/manual/mc-manual.htmlLinks to an external site.

Modular programming in CLinks to an external site.