Operating Systems 2018-B
Assignment 5: Multithreaded DNS Name Resolution System

Deadline: Monday, 14/05/2018

This assignment should be submitted as a *.tar.gz* archive, with C++ and README files inside. The archive should not contain generated object or executable files. You should verify that the archive can be downloaded back from Moodle and that it extracts correctly.

The assignment should be submitted in groups of 2 students. Names and IDs of students in group must be written inside a README file in the archive, and *also* listed in the submission comment in Moodle.

Here is how you can create a flat submission archive of directory ex5 that contains files ex5/README, ex5/main.cpp, ex5/utils.cpp, and so forth:

```
tar -C ex5 -czvf ex5.tar.gz .
```

Questions about the assignment should be asked in course's Piazza site. Late submissions are penalized 10 points per day, and assignments can be submitted at most 5 days late. Special requests should be directed to your lab TA.

Read carefully till the end the instructions for this assignment.

Introduction

In this assignment, you will develop a multithreaded application that resolves domain names to IP addresses, similar to the operation performed 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 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. See Figure 1.

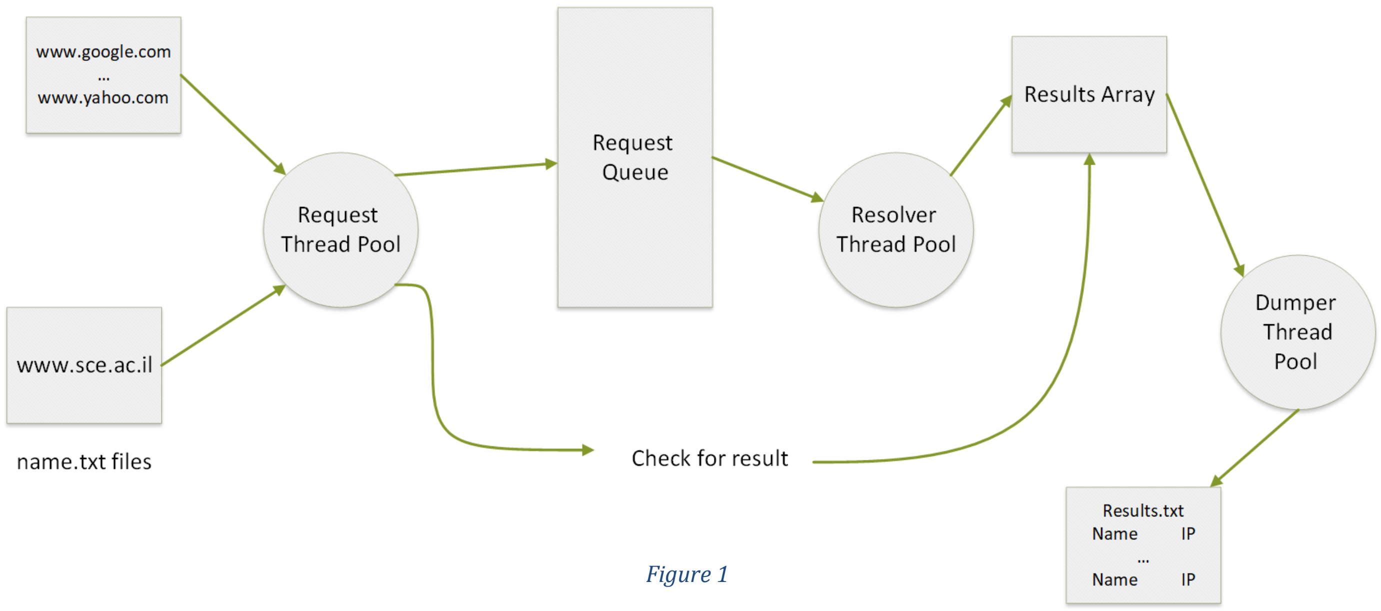

Figure 1

Task 1

General Description

Thread Pools are useful when you need to limit the number of threads running in your application at the same time. There is a performance overhead associated with starting a new thread, and each thread is also allocated some memory for its stack etc.

Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. Internally the tasks are inserted into a *Blocking Queue* which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.

Task Instructions

In this task, you will implement a thread pool which will be used in following tasks. A thread pool is a managed collection of threads that are available to perform tasks.

You can design your own thread pool structure as you wish, or you can adhere to the following thread pool skeleton:

- **Class Task** – Abstract class. Each of the tasks submitted to the thread pool would be derived from this class.
- **SafeQueue** – A safe implementation of a queue data structures. Ensures thread-safe access to queue elements. In other words, access to the queue should be via mutexes. In addition, a thread should wait if there are no tasks submitted now. Similarly, upon new task submission, all waiting threads should be notified.
  Note: It is recommended to design the queue with a *final* feature. A queue which is set as final prevents any new tasks to be submitted to.
- **ThreadPool** – The thread pool class, which includes a thread-safe queue for tasks and an array of working threads. Each working thread retrieves a task from the queue of tasks and executes that task. Upon completion retrieves the next task if there are any, or is blocked until new tasks arrive.

Task 2

General Description

Name Files

Your application will take as input a set of *name* files. Name files contain one hostname per line. Each name file should be serviced by a single requester thread from the requester thread pool, and further by a single resolver thread from the resolver thread pool.

Instructions

In this task, you should implement the requester thread pool and the resolver thread pool.

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 the name files, is placed into a FIFO queue. The FIFO queue is shared among threads. If a thread tries to write to the queue but finds that it is full, then it should sleep for a random period between 0 and 100 μs.

Each requester thread should detect, using condition variables, when each of its requests has 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 threads requests have been satisfied.

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 queries its IP address. After the name has been mapped to an IP address, the output is written to a shared array.

Task 3

Instructions

In this task, you should implement the dumper thread pool as described below.

Dumper Threads

A dumper thread from a thread pool should write results to a file in the following format:

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

Many hostnames return more than a single IP address. 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
```

General Instructions

(1) Each requester thread is responsible for one file, while each resolver thread is responsible for one domain name.

(2) The resolver threads can choose any domain name to resolve, independently of the requester threads.

(3) Additionally, a requester thread must know whether all of the domain names from the file it submitted were resolved.

(4) The requester thread is not released, or is not continued to the next task, until (3) is true.

Synchronization and Deadlock

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

Important: Notice that the console is not thread-safe either and you should assure synchronized access to the console.

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 input files have received a corresponding line in the output file.

Additional Specifications

You are provided with **libmyutil.so** and **util.h**, a shared library and a header file. 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.

In the **samples** directory, you can find

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.

results.txt This result file is a sample output of the IPs for the hostnames from the names1.txt file

Compilation Instructions

g++ -o *exefile FILES_TO_COMPILE* -std=c++11 -Wall -Wno-vla -pedantic -Os -lmyutil -L*PATH*

For example, in order to compile the accompanying test.cpp (assuming all files are in the same directory):

```
g++ -o test test.cpp -std=c++11 -Wall -Wno-vla -pedantic -Os  -lmyutil -L.
```

NOTE the . (dot) immediately after the -L argument. This argument states the path to the shared library, and . (dot) means current working dir.

Before running the compiled code, you should set the environment variable for the shared library loader. You can accomplish this for the running session of your terminal with:

```
export LD_LIBRARY_PATH=FULL_PATH_TO_SHARED_LIBRARY_LOCATION
```

For example, assume that all files including the shared library reside in /home/user/ex5:

```
export LD_LIBRARY_PATH=/home/user/ex5
```

You can also add this to your ~/.bashrc in order to set the variable for all sessions.

Note that, you cannot use class methods as arguments to pthread_create due to hidden *this* parameter. You must use either static class methods (which has no *this* parameter) or a plain ordinary function.

Program Arguments

When the program is called, it should interpret the last argument as the file path for the file to which results will be written. All preceding arguments should be interpreted as input files containing hostnames in the aforementioned format. An example call involving three input files might look like (assume the executable name is multi-lookup):

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

Limits

If necessary, you may impose the following limits on your program. If the user specifies input that would require 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**: INET_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 INET_ADDRSTRLEN characters including the null terminator.) This value is defined in inet.h in the Linux OS.

Error Handling

You must handle the following errors in the following manner:

• **Bogus Hostname**: Given a hostname that cannot 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 carriage return. You should also print a message to stderr alerting the user about 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.

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 function in the C Standard Library

• Standard Linux pthread functions

• Standard Linux file I/O functions

You are not allowed to use pre-existing thread-safe queue (including STL queue/vector) or file I/O libraries, since the point of this assignment is to teach you how to make non-thread-safe resources thread-safe.

If you would like to use additional external libraries, you must clear it with the TAs first.

What You Must Provide

To receive full credit, you must submit the following items to Moodle by the due date. Please, package the files into a single tar archive.

• multi-lookup.cpp: 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.

• README: A readme describing how to build and run your program.

• Any additional files necessary to build or run your program.

Good luck!