Lab 4: the network is the computer

Instructions

- This course is on a **_no-plagiarism_** diet. All parties involved in plagiarism harakiri will be penalized referred to DADAC and penalized to the maximum extent.
- The Moss Detective Agency has agreed to conduct all investigations.
  https://theory.stanford.edu/~aiken/moss/
  Byomkesh, Sherlock, Phryne, Marple, and Hercule are on standby.
- Hardcoding expected output of execution in source code will yield **negative** marks.
- Generative AI (ChatGPT, Gemini, etc.) is your friend, but is not you!
  Analytical, critical and creative thinking are learning outcomes of this course and source code or any content via GenAI tools cannot be part of your submissions.

- **Note:** Submission guidelines to be strictly followed; otherwise, your submission will not count.
- Login with username your **CSE LDAP ID** or **labuser** on software lab machines for this lab.
- This file is part of the **lab2.tar.gz** archive which contains multiple directories with programs associated with the exercise questions listed below.
- Most questions of this lab are tutorial style and are aimed to provide an introduction to tools and files related to system and process information and control.

4a. network programming 101

Two auxiliary files `client.c` and `load-gen.sh` are provided for this section of the lab exercise.

`client.c`

- Sends a connection request to server
- After connecting, the client sends a character string "Hello" to the server and waits for the server to respond with "World".

`loadgen.sh`

- Is a script that generates load by creating multiple client processes.

Your task is to write code for a server process that —

1. Accepts client connections in an infinite loop.
2. Read a message `"Hello"` from a client and respond with a message `"World"` to the client.
3. Close the connection once the response is sent.

Server Implementations:

The task is to write code for the server with three different IO models and then measure server performance.

1. A Single-Threaded Server – `server.c`
   - Implement a simple server that continuously accepts connections in an infinite loop.
   - After accepting a connection, the server reads the message `"hello"` from the client and responds with `"world"`.
   - The connection is closed immediately after the response is sent.
2. A server using the `select()` system call — `server_select.c`
   - Modify the above server to use the `select()` system call for handling multiple connections simultaneously.
   - The server should monitor multiple sockets (if needed) using `select()` to know when data is available to be read from the client.
3. A server using the `epoll()`system call — `server_epoll.c`
   - Further, modify the server to use the `epoll()` system call for efficient I/O multiplexing.
   - The server should use the `epoll()` system call to handle multiple client connections more efficiently than `select()`.
4. Use the `loadgen.sh` script to generate different load levels/setups (#client processes) and measure and report the performance of the server.

   E.g., draw a plot of load (#clients) vs. completion time for the three different server implementations.

5. Limit the maximum length for the queue of pending connections of the servers to be a finite number (say 5 or 10).
   Determine the load (#clients) that results in a connection refused error for each of the servers.
   Hint: Check the man page for the `listen` system call.
   Prefix the new server implementations a `kserver.c` , `kserver_select.c` and `kserver_epoll.c`

6. Can you design an experiment to empirically demonstrate the performance of the `select()` and `epoll()` system call mechanisms.
   This will need extensions to the client program (name it **stress-client.c**) and the load generator scripts (name it `stress-loadgen.sh`) and possibly the server implementation as well.

4b. pthreads 101

0. Threads are fun!

We have looked at processes; now, let us look at threads. There are two programs named `processes.c` and `threads.c`. Both programs have the global variable **x**. In `processes.c,` **x** is incremented by 5 by the child process and then printed by both parent and child processes. Furthermore, in `threads.c`, two threads are created, **x** is incremented by 5 by a thread in the routine `foo()`, and both threads print the value of **x**.

Compile and run both programs and study outputs. Specifically, observe the process ids and thread ids in the output and explain the values of **x** printed by both programs in a **report**.

Note: You must use `-lpthread` flag while compiling your code containing the pthread library (e.g. gcc threads.c -o threads.o -lpthread)

1. Where did the money go?

Task 1A: Don't touch my Money

For this task, refer to the file `addmillion.c`; the file has a routine named `increment()`, which the bank uses to increment the amount of `account_balance` for each deposit (banks do not repeatedly add 1, demonstrating a flaw mentioned later in the question). A deposit can only be made of 1 million. If anyone goes to their bank and makes ten such deposits, they expect the account balance to be 10 million because the `account_balance` is incremented by a sequential program. However, now say, ten people at ten different banks make one deposit in one account, the `bank_balance` is still expected to be 10 million. The program `addmillion.c` simulates ten different transactions at ten different banks using ten threads. Compile and run the program.

Is the output 10 million?
No! Where did the money go?
This condition is called a **race condition**.
Now modify the `addmillion.c` as `addmillion10.c` and implement a locking mechanism using pthread mutexes and get the desired 10 million as the `account_balance`.

Task 1B: The Sweet Spot

Further, modify the `addmillion.c` to take the number of threads as a command line argument. Depending on the number of threads, your program should make a valid deposit of 2048 million (e.g., for two threads, each thread should deposit 1024 million each). Understand

how arguments are passed to threads and modify the `increment()` routine to take the number of million as input. (Create an outer loop for iterating over the for loop that is already present). Run the program with threads ranging from 2 1024 (in powers of 2) and analyze the time taken for each run. So does increasing the number of threads result in better performance?

Submit the plot of the time taken by the different runs and conclude the key takeaway from the task in the **report.** You should also submit the modified `addmillion.c` in your submission. **Note:**
- You should measure time from the start of `main()` to program exit **WITHIN** the program and report it in the **EXACT** format: `Time spent: <time_taken> ms` (Use this: https://github.com/remzi-arpacidusseau/ostep-code/blob/master/include/common.h)

    The following two are optional, you can use your own favourite method to plot the data.
- Use the bash script `analysis.sh` to generate the plot.
- Install the prerequisites using: `sudo apt install plotutils`

2. Task server with a thread pool

For this question, you need to refer to the file `taskqueue.c`; the file contains self-explanatory global variables which are updated inside the routine *processtask()*, the program takes input from a file `tasklist`, and each line in the `tasklist` is a task or the time for which there is no task. For each line:

   I. A processing task (p) is written as **'p 2'** where p stands for processing task and 2 is the burst time (in seconds) for the task.
  II. A waiting period (w) is written as **'w 1'**, where w stands for a waiting period, and 1 denotes the time (in seconds) for which there are no tasks.

For processing each task, the *processtask()* routine is called. The routine simulates a task by sleeping for the given task's burst time. With the given tasks in the `tasklist` file, the current sequential program will take 10 seconds. Moreover, it gives the following output:

```
λ@sl2-1 > ./taskqueue.o tasklist
The number of tasks are : 4
Task completed
Wait Over
Task completed
Task completed
8 2 1 2 3
λ@sl2-1 > ▯
```

In the output the last line denotes the final values of the global variables, after the processing is done by all processing tasks. Outputs format is as follows —
sum of the processing times;
number of processing instances that have are odd burst times
number of processing instances that have are even burst times;
the min/max of the processing times

Implement a **multi-threaded version** of this program to reduce the time to process all the tasks by implementing the following:

- The main thread reads the number of worker threads as a command line argument and creates a pool of worker threads.
- The main thread reads the tasks one by one from the file and enters the task into a queue.
- A free thread from the thread pool picks up the task and processes it.
- Once all tasks are complete, the main thread joins the other threads and the global variables are printed.

Note: The first line of the `tasklist` file contains the number of tasks, followed by a task or waiting period.
Testing: To test your multithreaded version use the file `tasklistmultithreaded` as input.

3. Multi-threaded Server

i. To get started, modify your server from **4a** to add multithreading capability. i.e. now you have a listener thread that accepts grading requests and creates a worker thread to process each request. A thread should be created for each request and should exit after serving "hello" with a "world" reply. The thread should directly write the response back to the client and then exit.

ii. In the previous version, a thread was created and destroyed for every request. In this design, first of all, there is an overhead of thread creation and destruction, and secondly, there is no control over how many threads are created.

Specifically: update your server to accept a thread pool size which is the number of threads the

server has active. This thread pool size is fixed.

```
$./server  <port>  <thread_pool_size>
```

- At the beginning, the server main thread creates the thread pool of size `thread_pool_size`. These threads will start and wait for requests.
- You will now need a shared queue in which the main thread queues the requests (connections). Whenever a worker thread is free it should pick up the next request from the shared queue. After that the functionality is the same as before, the thread directly sends the response to the client.
- You will need **mutexes** (for thread-safe access to the shared queue) and **condition variables** so that the threads are not constantly checking the queue in a 'busy loop'.

4c. cFlask - HTTP C-based flask server

This assignment is inspired by Flask — a Python-based web application framework.
more info:
- https://palletsprojects.com/p/flask/
- https://flask.palletsprojects.com/en/2.2.x/
- https://flask.palletsprojects.com/en/2.2.x/quickstart/#

Example #1
The following code block demonstrates a simple Flask example. It exposes the hello_world() function to the web which can be accessed at the '/' URL path. Flask, by default, spins a server on localhost listening for requests on port 5000 (default port number, but configurable).

```python
# Importing flask module in the project is mandatory
# An object of Flask class is our WSGI application.
from flask import Flask
# Flask constructor takes the name of
# current module (__name__) as argument.
app = Flask(__name__)
# The route() function of the Flask class is a decorator,
# which tells the application which URL should call
# the associated function.
@app.route('/')
# '/' URL is bound to the hello_world() function.
def hello_world():
    return 'Hello World'

# main driver function
if __name__ == '__main__':

# run() method of Flask class runs the application
```

```
        # on the local development server.
        app.run()
```

Example #2
The following code block demonstrates how to deal with GET query parameters. The code snippet below is able to parse a URL similar to /hello?name=Ashwin, and stores 'Ashwin' in the variable name

```
from flask import Flask
app = Flask(__name__)
@app.route('/hello')
def hello_name():
    name = request.args.get('name')
    return 'Hello %s!' % name


if __name__ == '__main__':
    app.run()
```

In this assignment, we will recreate some of Flask's functionality of adding new functions and exposing them to the web via a URL using libHTTP — an open-source HTTP library in C, of which the web server and HTTP parser is our interest.

LibHTTP: https://www.libhttp.org
LibHTTP download: https://github.com/lammertb/libhttp/archive/v1.8.zip
Download and extract source. Study examples in the examples dir, build and try them.

1.

The tasks of the first part of the assignment are to spin up a simple web server using the libHTTP library, parse HTTP requests (using the libHTTP library), and then finally execute the function corresponding to each request.
Your program should be designed to accept the following inputs …
1. Port number on which the server accepts requests
2. Number of threads in the server thread pool to serve requests

You are expected to submit 4 files to implement the above mentioned functionality:

1. **functions.h** - This file will contain the list of all URLs to be exposed and will associate each URL to an integer which will act as an index to access the actual function associated with the URL.
   Example listing:
   ```
   /              0
   /square        1
   /cube          2
   ```

```
/hello            3
/prime            4
/pingpong         5
```
2. **functions.c** - This file will contain implementations of all the functions which are to be exposed on the Web.
3. **functionslist.h** - This file will contain a function pointer array which will place each function implemented in the file functions.c at the index specified in the file functions.h
4. **cflask.c** - This file will contain code to instantiate the web server using the libHTTP library and will also code to parse the URL and parameters (if any) of the incoming request, and code to call the actual requested functionality.

Note: All request URLs will only be of length 1 for this part of the assignment.
(e.g.,: http://127.0.0.1:8080/square), and will not contain any query parameters.

2.

In the second part of the assignment, extend the implementation to handle arguments passed in the URL as either query parameters in a GET request or inside a request body of a POST request. You can choose any of the above techniques to handle HTTP arguments.

HTTP requests will require careful parsing to extract the arguments from the request and then be supplied to the function corresponding to the HTTP request.

E.g., http://127.0.0.1:8080/square?num=3
The URL request beyond '?' specifies the argument variable num and its value is 3. The corresponding function already expects this format.

3.
In the third part of the assignment we will increase the complexity of parsing request URLs by increasing the length of the request URLs to two.
An example of a request URL of length two along with query parameters is …
http://127.0.0.1:8080/arithmetic/square?num=3

A corresponding entry in the functions.h file for the above example as follows

```
/arithmetic/square        x   (x is the function id for this function)
```

4.

Build an in-memory representation (a search tree, or a hash-based store) to store the URL to function ID mappings … instead of reading this information from a file on each HTTP request.

5.

(a) The sample list of URLs are to be used for testing your implementation …

1. **/**
   This just returns a hello world message back to the user.
   This URL will not have any arguments.
2. **/square**
   This URL requests computing the square of a number which can be passed along with the URL as a query parameter like /square?num=3 (return the square of 3). If no query parameter is present then this function should return the value 1.
3. **/cube**
   This URL requests computing the cube of a number which can be passed along with the URL as a query parameter like /cube?num=7 (return the cube of 7). If no query parameter is present then this function should return the value 1.
4. **/helloworld**
   This URL maps to a function that takes a string argument as a query parameter, and returns the same string prefixed by hello back to the user. For example, /helloworld?str=Ashwin, returns back "Hello, Ashwin". In case of no query parameter, it simply returns "Hello" back to the user.
5. **/pingpong**
   This URL maps to a function that takes a string argument as a query parameter, and returns the same string back to the user. For example, /pingpong?str=cs695, returns back "cs695". In case of no query parameter, it simply returns "PingPong" back to the user.
6. **/arithmetic/prime**
   This URL maps to a function that takes a number as a query parameter and returns 'True' to the user if the number is a prime and 'False' if the number is not prime. In case of no query parameter, the function returns 'False' back to the user.
7. **/arithmetic/fibonacci**
   This URL maps to a function that takes a number k as a query parameter and returns the kth fibonacci number to the user. In case of no query parameter, the function returns 1 back to the user.
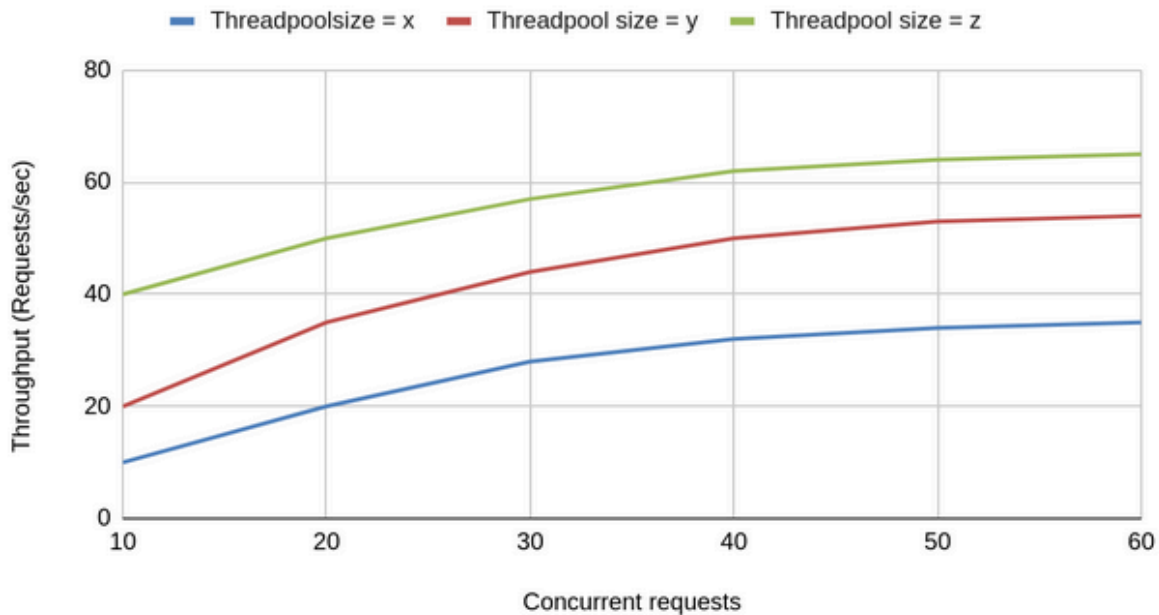
(b) Perform a load test of the server+function(s).

Pick a function (URL request) from the list above and increase the request rate and measure the average response time and throughput (#requests/second that were served).

- Plot these metrics as a function of Request rate on a graph for at least three different settings of the thread pool size.
  Make sure the plots have x and y axes labels and units.
  Hypothesis: throughput increases and then flattens at a certain load value.

- Repeat the above load test for at least 3 different functions of your choice and summarize your observations in a README file.

Load vs Throughput plot

Legend: — Threadpoolsize = x   — Threadpool size = y   — Threadpool size = z

Y-axis: Throughput (Requests/sec)
X-axis: Concurrent requests

You can use the tool Apache Benchmark or httperf to load test the cflask implementation. More info:
https://www.techrepublic.com/article/load-test-website-apache-bench/
https://github.com/httperf/httperf

Reporting of experiments

Experiment with ab as the default load generator and additionally, use any other load-generators that you know of or are familiar with or want to explore.
In any and all of the above cases the following is required for experimentation (and its reporting).

a. aim/purpose of the experiment
b. setup and execution details, metrics and independent parameter
c. hypothesis/expectation
d. observations from the data/plots
e. explanation of behavior and inferences

... repeat above for all individual experiments and for reasoning about a collection of experiments.

Submission Instructions

- All submissions are via moodle. Name your submission as **<rollnumber_lab4>.tar.gz**

  (e.g 190050096_lab4.tar.gz)

- The tar should contain the xv6 source code files in the following directory structure: **<rollnumber_lab4>/**
- Your code should be well commented and readable.
- `tar -czvf <rollnumber_lab4>.tar.gz <rollnumber_lab4>`

Deadline: — via moodle.