PROGRAMMING IN THE UNIX ENVIRONMENT, DV1457/DV1578

—

LAB 2: C PROGRAMMING

Sai Prashanth Josyula
Blekinge Institute of Technology

September 12, 2022

The objective of this laboratory assignment is for you to practice the following concepts in the context of programming: processes and forking, pthreads, daemons, IO multiplexing, and socket programming.

The laboratory assignments should be conducted, solved, implemented, and presented in groups of two students!

Preparations

Read through these laboratory instructions carefully and make sure that you understand what you are supposed to do. If something is unclear, please contact the teaching assistants or the main teacher. You are expected to have acquired the following knowledge before the lab sessions:

- You are supposed to have basic knowledge about working in a Unix environment.
- You are supposed to have good knowledge and experience in C programming.
- You should be familiar with the operating systems concepts dealt with in this assignment.

The five main topics relevant to this assignment are dealt with in lectures 6–10 of the course and are as follows:
(i) processes and forking, (ii) pthreads, (iii) daemons, (iv) IO multiplexing, (v) inter process communication via sockets.
You are expected to actively find and learn concepts required to complete this assignment. You do not need to wait for the corresponding lecture to be completed before you incorporate those concepts in your code; you can go ahead, learn the concepts, and start using them already. Do not forget to consult the relevant sections in the course book whenever required.

Plagiarism and collaboration

You are expected to work in groups of two. Groups larger than two are not accepted. You are also not expected to work alone unless you have a good reason.

You are allowed to discuss problems and solutions with other groups. However, the code that you write must be your own and cannot be copied or downloaded from somewhere else, e.g., fellow students, the internet, etc. You can of course use the source code provided with the laboratory assignment. You may copy idioms and snippets from various sources. However, you must be able to clearly describe each part of your implementation. The submitted solution(s) should be developed by the group members only.

1 Your task and requirements

Your task is to develop a limited *math server* that can solve two specific mathematical problems: (i) matrix inversion, (ii) k-means clustering. You will also develop a *client* program that can communicate with the server by sending the problem data and receiving the computed results. Note that when you are developing and testing your code, you will work will the same physical

device (i.e., your computer) and simply run the compiled *server.c* and *client.c* programs from different terminals. However, keep in mind that it must be possible to execute your server and client programs from different computers in a network. In the real-world, the client programs will run on different computers while the server program is running on another computer.

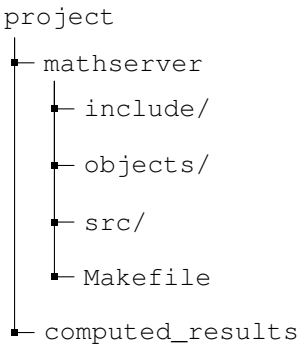
After you finish writing your code, the interaction between the server and a client should be as follows. The server program is run from a terminal with a *port number* as the command-line parameter (see Section 1.5 for more details). The client program is run from another terminal with the server's IP and port as the command-line parameters. Then, the connection between the server and the client processes is established. Let us call the two processes `server` and `client1`, respectively. The user should now be able to send the *problem data* to the server process through the client process (via the socket) using the command line. The server process receives the problem data from the client's socket, computes the solution to the problem, and sends the *result* back to the client process. The client process should give proper feedback to the user through the terminal whenever a file is successfully sent or received. Once a result is received, the user should be able to send the next problem's data to the server.

The server should support multiple clients at the same time. While the server is connected to a client (and likely solving its problem), another client should be able to connect to the server, send its problem data, and also receive its result. In other words, if the client program is run from yet another terminal with the server's IP and port as the command-line parameters, the connection between the server and this new client process should be established. Let us call this process `client2`. The server should simultaneously solve problems of `client1`, `client2`, and other clients that may want to connect with it.

The server and client programs must be written in C, compile using `gcc` and run in Ubuntu's latest LTS version. There shall be no software limitation in the number of served clients or concurrent requests.

1.1 Project structure

The basic folder structure for your project has to look as follows (you may add folders, if necessary):



The `mathserver` folder shall contain all your source files. Header files should be placed in the `include` folder, source code files should be in the `src` folder and object files, created during the project build, shall be placed in the `objects` folder. If you see it fit, you can further divide your header or source code files into separate folders within their respective folders. The `computed_results` folder is supposed to be the root directory of your math server. By default, your math server shall use this directory. Please keep in mind that it must be possible to execute your program on a different computer as well. Last but not the least, it must be possible to build your math server by issuing the `make` command within the `mathserver` folder, which will place the final executable in the `mathserver` folder.

1.2 Problems to be solved by your server

1.2.1 Matrix inversion

The inverse of a matrix is a compute-intensive operation that is used in many algorithms, e.g., linear regression, which is a machine learning algorithm. This will be our motivation to choose this as the operation to parallelize using `pthreads`. Our focus is on finding the inverse of a matrix using a method called Gauss-Jordan elimination. This method involves performing *elementary row operations* on the input matrix to obtain its inverse. Here is an illustration of matrix inversion using an example:

Problem: To find the inverse of an input matrix A of dimensions $n \times n$
(we will assume that the input matrix is always invertible)

For example, let us say that a user inputs A = $\begin{bmatrix} 3 & 6 & 9 \\ 2 & 8 & 15 \\ 7 & 3 & 8 \end{bmatrix}$. Here is how we calculate the inverse of matrix A.

$$A = IA$$

$$\begin{bmatrix} 3 & 6 & 9 \\ 2 & 8 & 15 \\ 7 & 3 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} A$$

Applying $R_1 \rightarrow R_1/3$ throughout

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 8 & 15 \\ 7 & 3 & 8 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} A$$

Applying $R_2 \rightarrow R_2 - 2R_1$ throughout

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 9 \\ 7 & 3 & 8 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ -\frac{2}{3} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} A$$

Applying $R_3 \rightarrow R_3 - 7R_1$ throughout

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 9 \\ 0 & -11 & -13 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ -\frac{2}{3} & 1 & 0 \\ -\frac{7}{3} & 0 & 1 \end{bmatrix} A$$

Applying $R_2 \rightarrow R_2/4$ throughout

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & \frac{9}{4} \\ 0 & -11 & -13 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ -\frac{1}{6} & \frac{1}{4} & 0 \\ -\frac{7}{3} & 0 & 1 \end{bmatrix} A$$

Applying $R_1 \rightarrow R_1 - 2R_2$ throughout

$$\begin{bmatrix} 1 & 0 & -\frac{3}{2} \\ 0 & 1 & \frac{9}{4} \\ 0 & -11 & -13 \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{2} & 0 \\ -\frac{1}{6} & \frac{1}{4} & 0 \\ -\frac{7}{3} & 0 & 1 \end{bmatrix} A$$

Applying $R_3 \rightarrow R_3 + 11R_2$ throughout

$$\begin{bmatrix} 1 & 0 & -\frac{3}{2} \\ 0 & 1 & \frac{9}{4} \\ 0 & 0 & \frac{47}{4} \end{bmatrix} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{2} & 0 \\ -\frac{1}{6} & \frac{1}{4} & 0 \\ -\frac{25}{6} & 11/4 & 1 \end{bmatrix} A$$

After doing similar operations to bring the last column of the matrix in the required form, the equation looks as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{19}{141} & -\frac{7}{47} & \frac{6}{47} \\ \frac{89}{141} & -\frac{13}{47} & -\frac{9}{47} \\ -\frac{50}{141} & \frac{11}{47} & \frac{4}{47} \end{bmatrix} A$$

$$I = \begin{bmatrix} \frac{19}{141} & -\frac{7}{47} & \frac{6}{47} \\ \frac{89}{141} & -\frac{13}{47} & -\frac{9}{47} \\ -\frac{50}{141} & \frac{11}{47} & \frac{4}{47} \end{bmatrix} A$$

Since the inverse of matrix A is a matrix which when multiplied by A gives the identity matrix I, the above obtained matrix is indeed the inverse of A. You may check that this solution is correct using this online calculator. You are provided with a sequential C program (`matrix_inverse.c`) that calculates the inverse of an input matrix A. Your task is to implement a parallel C program (using `pthreads`) that calculates the inverse of a matrix. You may compile and execute the given program

as follows:

```
$ gcc -w -O2 -o matinv matrix_inverse.c
$ ./matinv -n 4 -P 1 -I fast
Matrix Inverse

size      = 4x4
maxnum    = 15
Init          = fast
Initializing matrix...done

Inversed Matrix:
  0.27 -0.06 -0.06 -0.06
 -0.06  0.27 -0.06 -0.06
 -0.06 -0.06  0.27 -0.06
 -0.06 -0.06 -0.06  0.27
```

1.2.2 k-means clustering

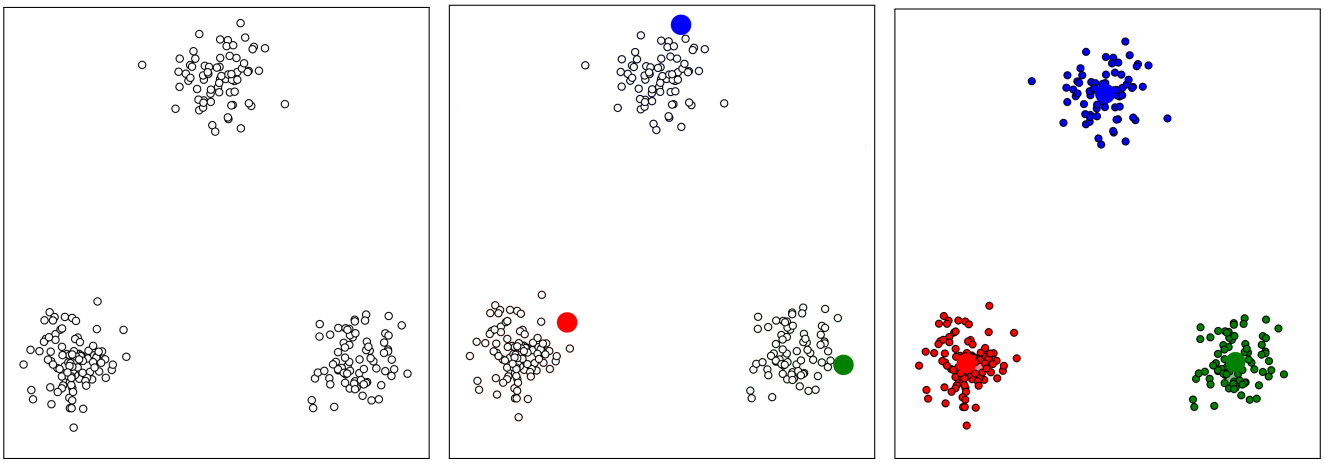
The k-means clustering is a simple algorithm that is used to group data points into a specified number of clusters. This popular machine learning algorithm is used in various applications such as computer vision. Algorithm 1 describes the steps constituting k-means clustering.

Algorithm 1: The k-means clustering algorithm

Data: Data points (D), Number of clusters (k)
Result: Cluster numbers (C) which specify the cluster number that each point belongs to

```
1 Randomly select k data points and set them as the k cluster centers (also called centroids);
2 while there are changes in cluster numbers C do
3     foreach data point in D do
4         Associate the data point with the nearest cluster ;      // Assigning a cluster to each point!
5     Recalculate the position of the k centroids using the mean value for each cluster of points.
```

Figure 1 shows how the k-means clustering algorithm groups a set of data points D into three clusters when $k = 3$ (for more visualizations, see this link). The clusters 0, 1, and 2 are indicated by the three colors red, green, and blue. Our algorithm's output will be an assignment of a cluster number to each data point.



(a) The data points D

(b) Initially assigned random centroids (shown as large red, green, and blue points)

(c) The final positions of the centroids and clusters after the algorithm concluded. The three colors represent three clusters.

Figure 1: k-means clustering algorithm for a given set of data points D and number of clusters = 3

You are provided with a sequential C program (`kmeans.c`) that implements the k-means algorithm. You can use the Python code provided in `kmeans-viz.py` to visualize the output of the sequential program. Your task is to implement a parallel C program (using `pthreads`) that calculates the kmeans of the input data file. You may compile and execute the given programs as follows:

```
$ gcc -w -O2 -o kmeans kmeans.c
$ ./kmeans
Read the problem data!
Number of iterations taken = 21
Computed cluster numbers successfully!
Wrote the results to a file!
$ python3 kmeans-viz.py
```

You may of course need to install `Python 3` and relevant packages on your system so that you are able to run the Python file `kmeans-viz.py`.

The source code files for the sequential programs are available on Canvas and are tested on Ubuntu 22.04. However, they should also work on any other Linux distribution that you may be using. You may compile your server program using the `-lpthread` flag as follows:

```
gcc -O2 -o server sourcefile1.c ... sourcefileN.c -lpthread
```

1.3 Concurrent Client Handling

The server should be able to efficiently handle client user sessions in parallel, i.e., if one user session is in progress, it should be possible for the server to handle another session concurrently. See Section 2 at the end of the document for various strategies. The intention is that your server should mimic the functionality of a real-world server. Note that your server must always be running once started and may only be terminated when `Ctrl+C` is pressed.

1.4 Process Management

The server may not create zombie processes, regardless of the server load (see the manual page for `ps`). See the course literature for information and tips on how to deal with zombie processes. You can create and terminate several user sessions in order to test your process management. Any dead processes will be visible using `ps`.

1.5 Command line options

The server must support at least the following command line options:

-h	Print help text
-p port	Listen to port number `port`.
-d	Run as a daemon instead of as a normal program. **(if implemented, see Section 3)**
-s strategy	Specify the request handling strategy: `fork`, `muxbasic`, or `muxscale` **(if implemented, see Section 3)**

Through the last option, it should be possible to select the request handling strategy. It shall be possible to run your server as a daemon, as well as as normal program. After starting as daemon, the server should print its process ID. The command line options should override the default value, which may have been specified by your implementation. If any of these options is not supported by your server, the program should exit and return 3 as its return value. Additionally, it should print the usage/help message.

Your `matinv-par.c` file must support all the command-line options supported by the sequential code provided to you.

Your `kmeans-par.c` file must support at least the following command line options:

-f file	Specify the file containing the problem data, e.g., `kmeans-data.txt`
-k clusters	Specify the number of clusters, e.g., `4`, `9`, etc.

2 Strategies for handling multiple connections

In this assignment, you will deal with two main strategies for handling multiple client connections. These strategies are forking and multiplexing. Let us call these two strategies *fork* and *mux* respectively. To pass the assignment, your server must give support for concurrent clients at least using `fork()`. Additionally, you may also implement one or two multiplexing strategies depending on the grade you are aiming for.

The general idea of the *fork* strategy is that the server creates a new child process, which is dedicated to handle the new client connection. Alternatively, one may also use threads instead of child processes to handle a new client connection. However, we will not do that in this assignment. You will instead use threads to parallelize the matrix inverse and k-means computations.

The *mux* strategy can be categorized into the two following strategies. The *muxbasic* strategy makes use of either `select()` or `poll()` system calls. The *muxscale* strategy makes use of either `epoll()`, or `kqueue()` to multiplex between a number of file descriptors. Using the *muxscale* strategy, your server will easily be able to serve thousands of clients, i.e., it will be scalable. When testing your code, we may use many clients and your server-client code is expected to work correctly for any parameters.

3 Presentation and Grading

The assignment should be presented in person with both group members in attendance on a lab slot. You will be required to show the code, let me run it and be prepared to discuss your work. The following are some test scenarios that will be used to grade your submission. You are expected to perform similar tests with your code before submitting it on Canvas. We will first start the server and perform the following tests.

1. Start a client, connect to server, and check whether the matrix inversion and k-means performed by the server are returning the correct results for various inputs.

   From the client's command line, you are expected to receive the results file returned from the server. The results returned by your server's parallel programs shall, of course, be in the *same* format as the output of the sequential programs provided to you.

2. Start a client, send some problem data to the server and compute some results. Meanwhile, start a new client, and see if it can also communicate with the server as intended. The existing client connections should function correctly despite new clients connecting to the server.

3. When using the *fork* strategy, your server should start multiple processes, each corresponding to a client. Thus, with this strategy, your server will handle multiple clients by forking a separate process for each client. Similarly, when using the *muxbasic* or *muxscale* strategy, your server should **not** start multiple processes.

You should demonstrate a clear understanding of the problem and detail your strategy of how to solve it, including drawbacks and advantages of the solution chosen. Your code should be clear, concise and to the point. The following list details, which of the requirements have to be implemented to get a certain grade.

Grade — D

- All requirements from Section 1.1 to Section 1.5 (including both sections)
- Implement support for handling concurrent clients using `fork()`.
  We will call this strategy *fork*.

Grade — C

- All requirements for Grade D
- Implement possibility to execute the server as daemon (see *-d* in Section 1.5)

Grade — B

- All requirements for Grade C
- Implement an additional strategy for concurrently handling multiple clients using `select()` or `poll()`.
  We will call this strategy *muxbasic*.

Grade — A

- All requirements for Grade B
- Implement an additional strategy for concurrently handling a large number of multiple clients. Use `epoll()` or `kqueue()`. We will call this strategy *muxscale*.

Note: Even if your implementation fulfils the requirements for the grade you were targeting, you might get a lower grade (lower by one), if your code is of bad quality. For this reason, there are no requirements for grade E. You can only get an E, if your implementation fulfils the requirements for grade D and your code is of bad quality. In this way, it is impossible to fail the assignment just because of poor code quality.

4 How the client and the server work together

When testing your code, you will need to carry out the following steps:

- Compile the server code and start the server program on some port in a terminal.
- The server process should now listen to the port. Whenever a client connection request arrives on the socket, the server should accept the connection, read the client's request, and return the result.
- Compile the client code and start the client program (from another terminal) with server IP and port as the command-line arguments.
- The client should now establish a connection with the server.
- The user enters the problem data from the client's terminal and their input is sent to the server via the connected socket.
- The server reads the user's input from the client socket, evaluates the command, and sends the result back to the client.
- The server should then wait for more input from the client.
- The client should display the server's reply to the user, and prompt the user for the next input, until the user terminates the client program with `Ctrl+C`.
- Meanwhile, when the client program is started from another terminal with the proper command-line arguments, it should be able to establish a connection with the server, send its problem to the server, and also receive the result.

The command line of the server looks as follows.

```
$ ./server -p 9999
Listening for clients...
```

The command line of a client looks as follows. After the server computes the result, it should send the solution to the client in the form of a file, which it should receive. Note that a solution file received by the client may not overwrite a previously received solution file.

```
$ ./client -ip 127.0.0.1 -p 9999
Connected to server
Enter a command for the server: matinvpar -n 8 -I fast -P 1
Received the solution: matinv_client1_soln1.txt
Enter a command for the server: matinvpar -n 2048 -I rand -P 1
Received the solution: matinv_client1_soln2.txt
Enter a command for the server: kmeanspar -f kmeans-data.txt -k 9
Received the solution: kmeans_client1_soln1.txt
```

Meanwhile the command line of the server will look as follows.

```
$ ./server 9999
Listening for clients...
Connected with client 1
Client 1 commanded: matinvpar -n 8 -I fast -P 1
Sending solution: matinv_client1_soln1.txt
Client 1 commanded: matinvpar -n 2048 -I rand -P 1
Sending solution: matinv_client1_soln2.txt
Client 1 commanded: kmeanspar -f kmeans-data.txt -k 9
Sending solution: kmeans_client1_soln1.txt
```

————*All the best!*————