Programming Assignment 1
(Use of System Calls)
CS962: Operating System Principles
2023-24 Q3
eMasters in Cyber Security
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Due Date: 07-02-2024 23:55

This programming assignment makes you acquainted with the working principle of system calls. You need to have a Linux system for this assignment, those who have other operating systems can use a Virtual Machine with Linux. In Task-1 and Task-2, you need to establish a communication channel between two processes to achieve different functionalities as described in each task. In Task-3, you need to implement a utility to find disk space used by a directory. Template codes for tasks are placed under the PA1 folder in —PA1/Task-1, PA1/Task-2, and PA1/Task-3 directories in **PA1.zip**. Please go through the `README` file to understand the code base structure and build process of the assignment.

Task 1: Lets Do Some Calculation !!!! Client-Server Calculator [30 Points]

In this task, you need to implement a client-server based calculator. The client can submit any number of queries (i.e. mathematical expressions) to the server. The server will evaluate the client's queries one by one and send back the answer as a response.

In this task, the main process will call fork to create a child process where child process acts as the server and the parent (main process) acts as client. The server will respond to all queries submitted by the client until it receives a specific terminating word (mentioned below) as a query from the client. After receiving the specific terminating word, the server will close the communication channel and exits. You need to use pipe to establish a communication channel between client and server.

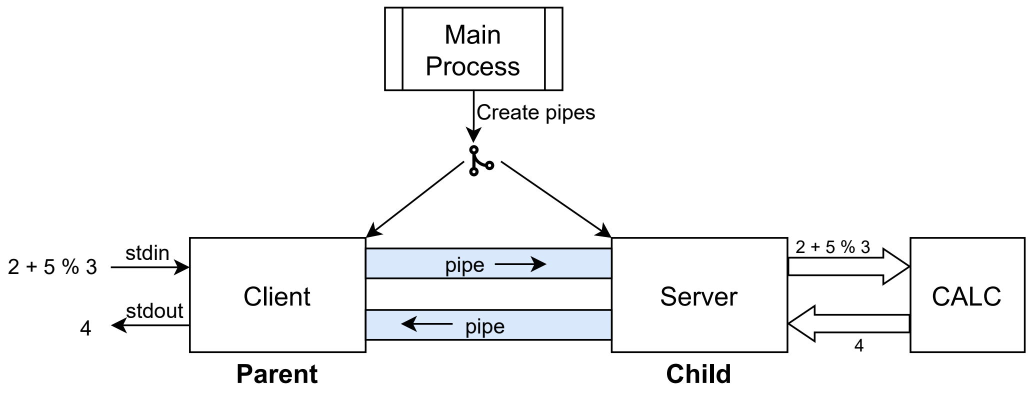


Figure 1: Workflow of client-server based calculator.

Figure 1 shows the workflow of this client-server based calculator using pipes. The client (parent) should read expression (or may be terminating word) from standard input device, and will display result received from the server on standard output device as shown in the figure.

Requirement specifications for this task are as follows:

1. Input expression is always space separated, i.e. blank space between operand and operator.
2. Input expression always starts with an operand.
3. Input expression can have at most 20 operands (i.e., it can have at most 19 operators).
4. Operands in input expression may be an integer or float/double number.
5. Input expression can contain any combination of these five operators viz. (i) Addition (+), (ii) Subtraction (–), (iii) Multiplication (*), (iv) Division (/), and modulo (%).
6. When server performs calculation, it needs to follow **BODMAS** rule where division, multiplication and modulo operators are at same precedence (say precedence-1) and addition and subtraction are at precedence-2. According to BODMAS, server should first perform precedence-1 operations followed by precedence-2 operations. Among same precedence operations, server should perform operation from left to right.
7. When client receives **END** word from STDIN, it should send a termination signal to the server to terminate it properly.
8. On receiving the result of mathematical expressions from the server, client should print "**RESULT:** " followed by result. As an example, if client receive **5** from the server, then it should display "**RESULT: 5**" on STDOUT.

Example of Input & Output:

```
[INPUT]  1 + 5 * 2 - 1 + 10
[OUTPUT] RESULT: 20
[INPUT]  5 + 1 - 2 * 5 + 10
[OUTPUT] RESULT: 6
[INPUT]  END
```

Implementation

- Inside **Task-1** folder, you will find four C files namely—(i) **task1_calc.c**, (ii) **task1_calculate.c**, (iii) **task1_client.c**, and (iv) **task1_server.c**.
- You need to establish the communication channel and invoke the client and server function at the appropriate location inside the main function available in **task1_calc.c**.
- The client and server functionality need to be implemented in files **task1_client.c** and **task1_server.c**, respectively.
- In **task1_calculate.c** file, implement the calculate function that the server will invoke whenever required.
- Run **make task1** command from the root folder of this assignment to get binary file named **task1_calc**. Use this binary to test the functionality of your implementation.

You can use below mentioned APIs to implement this part of the assignment. Refer to man page of these APIs to know about their usage.

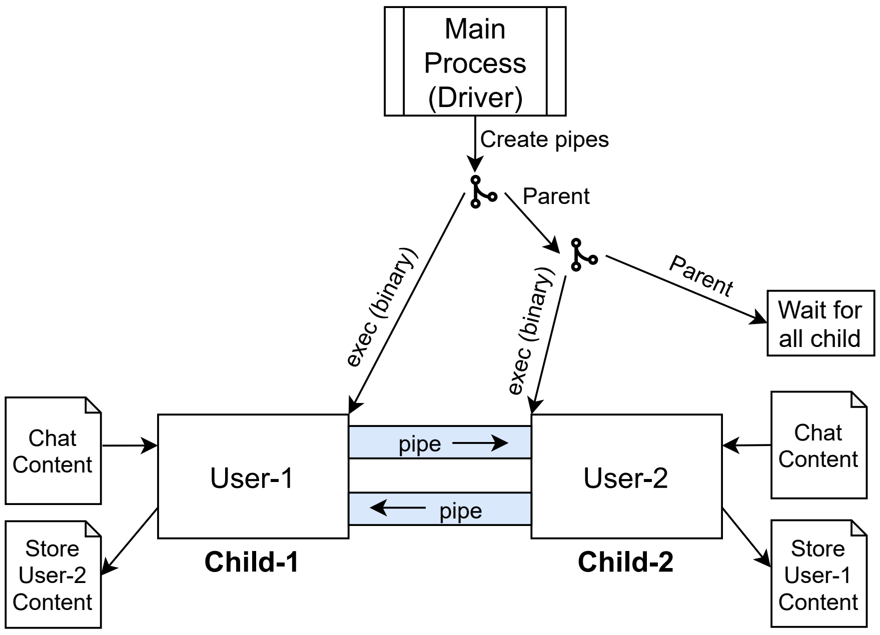


Figure 2: Flow of establishing chat communication between two user.

`pipe`	`write`	`exit`
`close`	`sprintf`	`sscanf`
`wait`	`strtok`	
`read`	`strcmp`	

Task 2: Lets Chit-Chat !!! Chat Between Two Users [30 points]

In this task, you need to implement chat communication between two users using pipe. The main process acts as the driver and creates two child processes to facilitate communication between them using pipes. Each child processes (i.e. user-1 and user-2) exec the same "user program" binary to start the conversation between them. User program reads the chat content which has to be communicated to the other user from the **chat content** file (provided) and writes the chat content received from other user to the **store content** file. Each line in chat content can be a maximum of 500 characters.

Figure 2 shows the workflow of this chat communication. One of the users will be the **initiator** based upon the first line (**keyword: initiate**) in the **chat content** file. The initiator user will start the chat by sending next line (actual message) after initiate keyword in the **chat content** file. Similar to the initiator, any user can be the **terminator** of chat communication. Terminator will have **keyword: bye** in the chat content file or there is no more content left for communication in the **chat content** file.

As part of this task, you need to implement both driver and user program. The driver is responsible for fork & exec the user programs and establishing the communication channels between them. The user program is responsible for reading the chat messages and writing the replies as mentioned above. The driver program should accept 4 command line arguments as follows.

./task2_driver <user1_content> <user1_store> <user2_content> <user2_store>

Implementation

- Navigate to the **Task-2** folder of code base, there are two C files, **task2_driver.c** and **task2_user.c**, to implement the driver and user program, respectively.

- After implementation, run the **make task2** command from the root folder of this assignment. You will get two binary files at the root folder, namely **task2_driver** and **task2_user (inside Task-2**

folder), corresponding to the driver and user programs.

- You need to exec **task2_user** inside the driver program two times to mimic two users.
- Use the **task2_driver** binary as mentioned above to test the correct functionality of your implementation.
- You will find two additional text files for testing purposes, namely— **chat-1.txt** and **chat-2.txt**, inside the **Task-2** folder, which holds the sample **chat contents** for two users.

You can use below mentioned APIs to implement this part of the assignment. Refer to man page of these APIs to know about their usage.

`pipe`	`fopen`	`strcmp`	`exec`
`close`	`perror`	`exit`	`strcpy`
`wait`	`getline`	`sscanf`	`sprintf`
`read`	`free`	`strlen`	`fclose`
`write`	`fputs`	`atoi`	

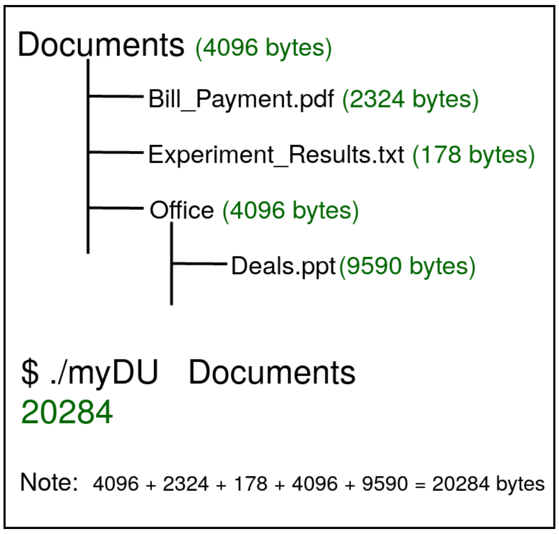
Task 3: Directory Space Usage [40 points]

In this question, you have to write a program (`myDU.c`) that finds the space used by a directory (including its files, sub-directories, files in sub-directories, sub-sub directories etc.). Let's call this directory as the *Root* directory.

Syntax

```
$./myDU <relative path to a directory>
```

Example


Documents (4096 bytes)
Bill_Payment.pdf (2324 bytes)
Experiment_Results.txt (178 bytes)
Office (4096 bytes)
Deals.ppt (9590 bytes)

```
$ ./myDU   Documents
20284
```

Note: 4096 + 2324 + 178 + 4096 + 9590 = 20284 bytes

Figure 3: Example to illustrate the use of directory space usage finding utility

Figure 3 shows the structure of a directory called **Documents** which is the designated *Root* directory in this example. This directory contains files such as **Bill_Payment.pdf**, **Experiment_Results.txt** and a sub-directory called **Office**. The Sub-directory **Office** contains a file named **Deals.ppt**. To find the size of the **Documents** directory, its name is passed as **$./myDU Documents**. Your utility is expected to print the total size of the contents of the passed root directory in bytes **(For eg: 20284)**. Note that, this is inclusive of directory sizes.

Output

Only print the size of the *Root* directory in bytes (Refer to Figure 3)

Detailed instructions

To make the calculation process more efficient, we propose a method where different sub-directories under the *Root* directory will be processed by different processes. The exact working is detailed in the following points.

- Assume that there are N immediate sub-directories under the *Root* directory. For each immediate child sub-directory under the provided *Root* directory, your program must create a new process P_i (i will range from 1 to the total number of immediate sub-directories under *Root*). Each child process P_i should find the size of the i^{th} child sub-directory (including all files and directories under it and the size of the sub-directory itself) and pass this information back to the parent process. Parent process should find the size of the files immediately under it along with the *Root* directory size. Finally, parent process will find the sum of all sizes and print the final output.

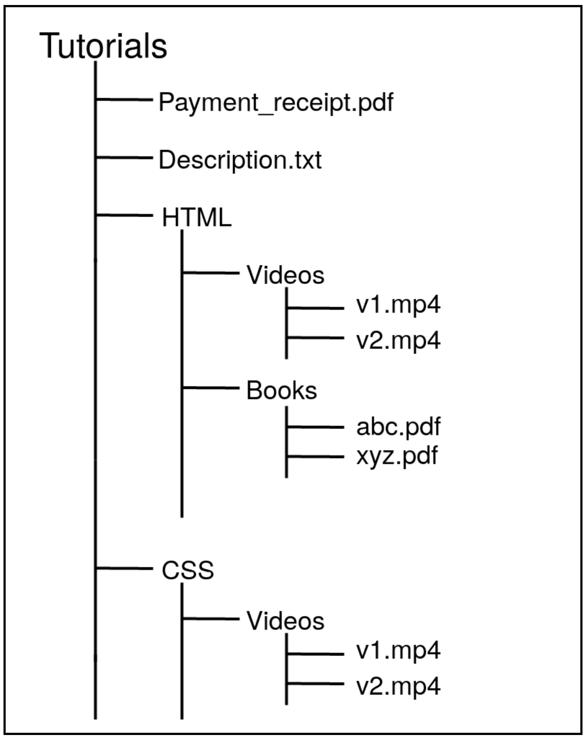


Figure 4: Sample directory structure

For example: In Figure 4, there are two immediate sub-directories (**HTML, CSS**) under the *Root* directory (**Tutorials**). In this case, the parent process should create two child processes, say, P_1 and P_2. P_1 should calculate the size of the sub-directory (**HTML**) which is a sum of sizes of all files and directories under **HTML** including the size of **HTML** itself. Similarly, P_2 should calculate size of the directory **CSS**. Both P_1 and P_2 should return the result back to the parent using pipes. Finally, the parent process would find the size of the files immediately under it (**Payment_receipt.pdf, Description.txt**) and the size of the **Tutorials** directory to report the final result.

- Your program should *only* use pipes (pipe() system call) to communicate between parent process and the child processes. There is no restriction on the number of pipes to be used.

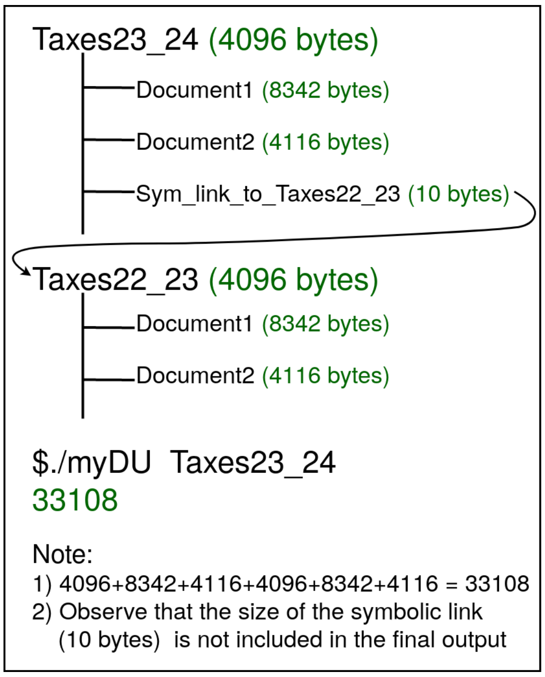


Figure 5: Example to illustrate the handling of symbolic links by ./myDU utility

- A symbolic link is a special type of file in Linux that points to another file or directory. Symbolic links can be present anywhere in the *Root* directory tree. You should resolve the symbolic links and find the size of the file/directory pointed by a symbolic link instead of reporting the size of the symbolic link file (Refer figure 5). It can be assumed that a symbolic link will never point to itself recursively. For example, in Figure 5, `Taxes22_23` directory will not contain a symbolic link that points to the `Taxes23_24` directory or directly to the `Sym_link_to_Taxes22_23` symbolic file in the `Taxes23_24` directory.

- It is the responsibility of the parent process to find the size of the file/directory pointed by a symbolic link which is present immediately under *Root* directory. For example, in Figure 5 parent process will not create any child process. However, if a symbolic link is present in a sub-directory, it should be processed by the child process handling the sub-directory.

- During testing, only relative path of the *Root* directory will be passed to the `./myDU` utility. It can be assumed that the size of the directory path generated during testing will not exceed 4096 characters.

- You can assume that the total size of the *Root* directory will fit in a 8-byte integer type (i.e., *unsigned long* on 64-bit machines)

Error handling

In case of any error, print **"Unable to execute"** as output.

System calls and library functions allowed

```
- fork               - malloc
- exec* family       - free
- pipe                   - stat
- opendir                - lstat
- readdir                - readlink
- closedir               - strlen
- read                   - open
- write                  - close
- strcpy                 - strcat
- strcmp                 - strto* family
- ato* family            - wait/waitpid
- printf family          - exit
- dup                    - dup2
```

Testing

Refer to README document

Deliverable:

1. Source code of Task-1, Task-2 and Task-3 which you have implemented. Document your code properly. The source code of each task should be placed on the same place as provided in the code base. **Please do not change PA1 code structure as we will be using automated scripts for testing**.

2. Feel free to discuss about the assignment among your friends, instructor and TAs. However, at the end of the day, we want you to do the implementation by yourself.

3. Write a `README.txt` file containing what you have discussed/with whom/any other sources that you have referred to do the assignment. If you have used any scripts/code from your friends, do mention that as well.

4. You have to submit zip file named `your_roll_number.zip` (for example: `12345.zip`) containing **only** the following files in specified folder format:

```
12345.zip
|
|------ 12345
              |------- Task-1
              |            |----- task1_calc.c
              |            |----- task1_calculate.c
              |            |----- task1_client.c
              |            |----- task1_server.c
              |-------- Task-2
              |            |----- task2_driver.c
              |            |----- task2_user.c
              |-------- Task-3
              |          |----- myDU.c
```

We will accept your submission via emasters portal only, and any other submission mode is strictly prohibited such as submitting via email.

All the best ..... PS: Start Early.