

Lab 2: stay calm and trust the process

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.

Required reading/reference: <https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-api.pdf>

1.0.1 WARM UP

The `fork()` system call creates a new process by duplicating the calling process. The new process is referred to as the **child process**, and the calling process is referred to as the **parent process**. Please refer to the man pages for more information about fork: [fork\(\) man page](#)

Listed below is a program using the `fork` system call —

```
#include<stdio.h>
#include<unistd.h>
int main(){
    fork();
    fork();
    fork();
    fork();
    printf("Hello world\n"); return 0;
}
```

- a. How many times would the string **Hello world** be printed?
- b. How many new processes would be created other than the original/first process?
- c. Draw the process hierarchy of all the processes during execution, i.e., parent-child relationships.

Verify your answers by compiling and executing the program.

1.0.2 WARM UP

Listed below is a program using the `fork` system call —

```
#include<stdio.h>
#include<unistd.h>

int main(){
    int a=10;
    int pid=fork();
    if(pid==0){
        a++;
    }else{
        a--;
    }
    printf("%d\n, a");
    return 0;
}
```

- a. On execution, how many times will the value of variable **a** be printed?
- b. What do the value(s) of the variable **a** printed by this program infer (regarding data of parent and child processes)?

1.a.

Write a program **1a.c** which creates a child process using `fork` and prints the PID (process id) of the parent process id and its own process id, for both the parent and child processes.

Sample Output:

```
user@users-linux:~$ ./1a
P1 PID: 190498, PPID: 190235
P2 PID: 190499, PPID: 190498
```

Which process is the parent of the process corresponding to the program **1a**.

1.b.

Write a program **1b.c** which takes an integer **k** as command line argument and creates **k child processes**, with each printing its kth-index and its PID.

Sample Output:

```
user@users-linux:~$ ./1b 4
Child process 1, PID: 190247
Child process 2, PID: 190248
Child process 3, PID: 190249
Child process 4, PID: 190250
```

1.c.

Write a program **1c.c** which takes an integer **k** as command line argument and creates **k child processes**, with the **parent process terminating only after all its child processes terminate**. Each child process should sleep for 5 seconds while the parent has no sleep statement.

New system calls — `wait`, `waitpid`

Sample Output:

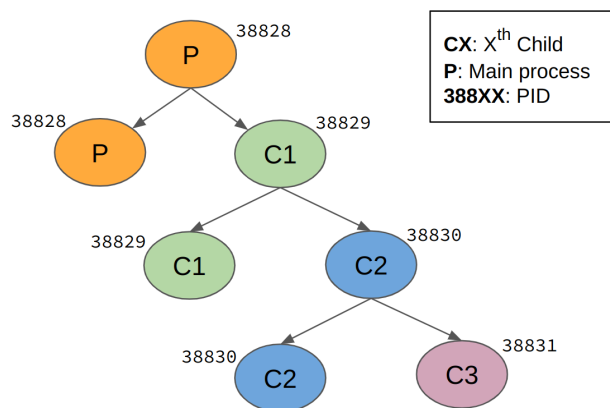
```
user@users-linux:~$ ./1c 3
190401: Parent process, PID: 190401
190402: Child process 1, PID: 190402, PPID: 190401
190403: Child process 2, PID: 190403, PPID: 190401
190404: Child process 3, PID: 190404, PPID: 190401
190401: PID: 190402 exited
190401: PID: 190403 exited
190401: PID: 190404 exited
```

Note: The exit messages should be printed from the parent process after the child process terminates. *Each print statement starts with PID of the process printing the message.*

1.d

Write a program **1d.c** that takes an integer **k** as a command-line argument and creates **k nested child processes**. As shown in the diagram below, a child process(C2) created by another child process(C1) is said to be a **nested child** of the main process(P).

Each child process should terminate before its parent process, ensuring the parent process waits for its child to finish before it exits.



The diagram visually represents the hierarchy, with arrows pointing from parent to child implying a fork by the parent process, and shows the process IDs to track the lineage of processes.

References: [fork\(\) man page](#), [wait\(\) man page](#)

Sample Output:

```
user@users-linux:~$ ./1d 3
38828: Number of children: 3
38828: Main Process's PID is: 38828
38829: PID: 38829 PPID: 38828 created
38830: PID: 38830 PPID: 38829 created
38831: PID: 38831 PPID: 38830 created

38830: PID: 38831 PPID: 38830 exited
38829: PID: 38830 PPID: 38829 exited
38828: PID: 38829 PPID: 38828 exited

38828: Main Process(PID: 38828) exiting
```

Note: Each print statement starts with PID of the process printing the message.

1.e

Write a program `1e.c` that acts as a simple command-line calculator. The calculator program should prompt the user for a command (`min`, `max` or `sum`), and for each command instance create a child process using `fork()` to execute the command, and wait for the child process to complete. The child process should have logic to execute these **3** commands. If the command cannot be found then it should display **Command not found** message and continue to prompt the user for another command. The calculator should be able to exit with command `exit`.

Format of the arguments given to the calculator:

```
>>> command N arg1 arg2 arg3 ... argN    // N is the number of arguments
```

References: [fork\(\) man page](#), [wait\(\) man page](#)

Sample Output:

```
user@users-linux:~$ ./1e
>>> max 5 1 2 3 4 5
5
>>> min 3 9 6 10
6
>>> sum 2 15 17
32
>>> invalid_command
Command not found
>>> exit
user@users-linux:~$
```

Note: To parse input arguments to the calculator, you may want to use the C string library.

1.f

Write a program `1f.c` that takes an **executable-file-name** at command line argument, then it creates a child process using `fork()`, the child process then calls `exec` system call (e.g., `execv` or `execp`) to execute the file taken as command line argument. If the file does not exist then the program should print “**Executable file not found**” to the output terminal.

In the given example, executable-file-name is `hello`, and is in the same directory as the executable `1f` and prints a **Hello, World!** message.

References: [exec\(\) man page](#)

Sample Output:

```
user@users-linux:~$ ./1f hello
1f PID: 38929
PID: 38931, PPID: 38929
Hello, World!
user@users-linux:~$ ./1f invalid_name
1f PID: 38963
Executable file not found
```

Note: `execv()` returns only if an error has occurred, else the statements after `execv()` will not be executed.

1.g

Redo task **1.e** using the `exec` system call.

Create 3 executable files `min`, `max` and `sum` which take command line arguments in the given format.

```
user@users-linux:~$ ./max N arg1 arg2 arg3 ... argN
user@users-linux:~$ ./min N arg1 arg2 arg3 ... argN
user@users-linux:~$ ./sum N arg1 arg2 arg3 ... argN
```

Write a program `1g.c` by modifying the `1e.c` such that after creating a child process using the `fork()` system call, the child process should use the `exec` system call with proper arguments. For any command other than `min`, `max` and `sum` the program should print a “**Command not found** message” and prompt again for another command. The command `exit` should quit the program..

Example `execv()` invocation of the program name `sum`

```
char *args[] = {"sum", "3", "1", "2", "3"};
execv("sum", args);
```

Sample Output:

```
user@users-linux:~$ ./1g
>>> max 5 1 2 3 4 5
5
>>> min 3 9 6 10
```

```
6
>>> sum 2 15 17
32
>>> invalid_command
Command not found
>>> exit
user@users-linux:~$
```

A file descriptors detour

References: [link1](#) [link2](#)

In UNIX-based OSES, each process has a **per-process file descriptor table** as part of the OS state to keep track of all the open files of the process (The fd table is an array of pointers to the global file table, and the fd is an index to an array element in this array). Further, each entry in the file descriptor table points to a **global file table**. The global file table contains actual metadata about the file (an example would be where the file is actually located on disk, the current offset for file operations, the number of file descriptor references to the file, etc.).

Entries at index 0, 1, and 2 in each process's file descriptor table normally point to the **STDIN** or **standard input** (input from your terminal/console), **STDOUT** or **standard output** (output to your terminal/console), and **STDERR** or **standard error** (output to your terminal/console) files respectively. These file descriptors are known as standard file descriptors.

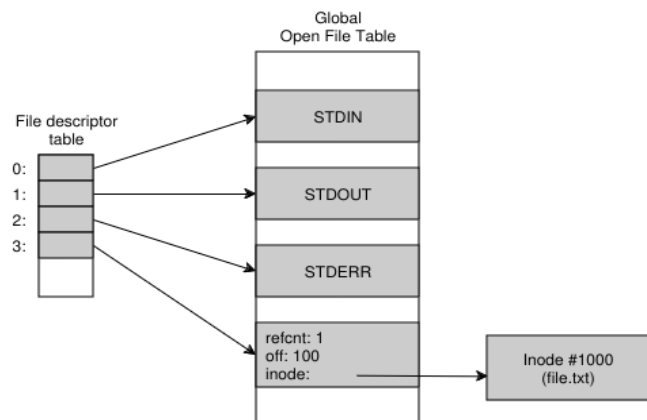
But why? Why do standard file descriptors exist in UNIX systems' processes, and where are they typically used in a program's execution?

As it turns out, the first process that is created (init process) opens these files, and then on these file descriptors are inherited from parent processes upon `fork()`. Interestingly, all interactions that are done with routines such as `printf()`, `scanf()`, which in turn use the `write` and `read` system calls, happen through these files.

Consider the following sample program

```
int main(){
    char buffer1[100];
    int fd = open("file.txt", O_RDONLY);
    read(fd, buffer1, 100);
    close(fd);
    return 0;
}
```

The diagram shows the state of the file descriptor table and global file table after the `read` system call has been performed and before the file is closed using the `close` system call.



2.0.1 WARM UP

To gain a better understanding of the file descriptor table's dynamics, let's walk through a simple example. We will use the `warmup.txt` file (10 bytes in size), which is included with the lab's auxiliary files.

The sample program provided opens the `warmup.txt` file twice using the `open()` system call, storing the file descriptors in `fd1` and `fd2`, respectively. The program then reads the contents of the file into three memory regions — `buffer1`, `buffer2`, and `buffer3`, using the `read()` system call.

Additionally, the program closes the file descriptor associated with standard input (indexed at 0) using the `close()` system call, and subsequently reopens `warmup.txt` storing this new file descriptor in `fd3`. After completing all operations, the program closes all open files using the `close()` system call.

References: [open\(\) man page](#), [close\(\) man page](#), [read\(\) man page](#)

```
#include<stdio.h>
#include<unistd.h>
int main(){
    char buffer1[10], buffer2[10], buffer3[10];

    int fd1 = open("warmup.txt", O_RDONLY);
    int fd2 = open("warmup.txt", O_RDONLY);

    read(fd1, buffer1, 5);
    read(fd1, buffer2, 5);
    read(fd2, buffer3, 10);

    close(0); // Closing STDIN
    int fd3 = open("file.txt", O_RDONLY);

    close(fd1);
    close(fd2);
    close(fd3);
    return 0;
}
```

- a. What will be the values of `fd1`, `fd2`, and `fd3`?
- b. What will be the contents of `buffer1`, `buffer2`, and `buffer3`?
- c. Draw a picture similar to the one in the box above to show the state before the `close(fd1)` statement is called.

You can verify your answers by compiling and executing the program via `gdb`.

2.0.2 WARM UP

This program shows how a fd table is shared when a parent process creates a child process via the `fork` system call. In the program, the parent first opens a file and stores the file descriptor in `fd1`, it then creates a child process, opens the file again, for which it stores the file descriptor in `fd2`, and then waits for it to complete. The child process adjusts the current offset via a call to `lseek()` and then exits. Finally, the parent process, after waiting for the child process, checks the current offset and prints out its value.

```
#include<stdio.h>
#include<unistd.h>
int main(){
    int fd1 = open("abc.txt", O_RDONLY);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd1, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        int fd2 = open("abc.txt", O_RDONLY);
        (void) wait(NULL);
        printf("parent: offset %d\n", (int) lseek(fd1, 0, SEEK_CUR));
    }
    return 0;
}
```

- What do you think will be the value of `fd1` in parent and child?
- What will be the value of `fd2` in the parent?
- What will be the output of the `printf()` statements and why?
- Draw a picture similar to the one in the box above to show the state before the `wait(NULL)` statement is called.

Hint: There will be two file descriptor tables and one global table. Some of the entries will be shared.

You can verify your answers by compiling and executing the program via `gdb`.

2.a.

Implement a simplified version of the **cat** command (name it **2a.c**) using the **fork** system call to create the child process that reads contents from STDIN or a file and writes them to STDOUT using system calls **read** and **write** (**NOT** **printf** and **scanf**). If the program is run with no arguments, it should read from standard input (STDIN) till a new line character and output to standard output (STDOUT), if a filename is mentioned at the command line, the program should read from till the end of the file and write to STDOUT.

Do not use the shell's **cat** command from within your program.

Sample Output:

abc.txt : abc (content of abc.txt file)

```
user@users-linux:~$ ./2a abc.txt
```

```
The only true limit is the one you set for yourself - Master Shifu
```

```
user@users-linux:~$ ./2a
```

```
>>> OS is critical for world peace!
```

```
OS is critical for world peace!
```

```
>>>
```

Press **ctrl+d** to signal to close the program.

2.b.

Following a similar approach to the **cat** command implementation, implement a simplified version of the **cp** command that copies the contents of a source file to a destination file using system calls like **open**, **read**, **write**, and **close**.

The program (**2b.c**) should read and write only to file descriptors 0 and 1 (similar to the **cat** program), but the fd's should be set up such that read/write happens on the files specified.

Hint:

The **close** system call deletes a file descriptor from the per-process fd table and a subsequent call to **open** system call use and return the smallest available value of file descriptor index.

```
user@users-linux:~$ ./2b abc.txt xyz.txt
```

```
user@users-linux:~$ ./2a abc.txt
```

```
The only true limit is the one you set for yourself - Master Shifu
```

```
user@users-linux:~$ ./2a xyz.txt
```

```
The only true limit is the one you set for yourself - Master Shifu
```

2.c.

Write a program `2c.c` which performs the following tasks:

- Use the `dup` system call to duplicate the file descriptor for standard output (`STDOUT_FILENO`) and standard input (`STDIN_FILENO`)
- Use the duplicated file descriptor for standard input to read the input and output the read content to the terminal.
- Use the `dup2` system call to redirect standard output to a file (`dup.txt`).
Note that this involves opening the file `dup.txt`, closing `STDOUT_FILENO` and using `dup2`.
Write "**I am re-directed output!**" to the file using the `STDOUT_FILENO`.
- Again use `dup2` to restore the original standard output and write "**I am original output!**" to the terminal using the `STDOUT_FILENO`.

References: [dup man page](#)

Sample Output

```
user@users-linux:~$ ./2c
>>> DECS
you have entered: DECS
I am original output!
user@users-linux:~$ ./2a dup.txt
I am re-direction output!
```

There is a next page to this document!

pipes — plumbing for the flow of information between processes

A **pipe** is a mechanism for inter-process communication using file descriptors.

man page of the **pipe** system call is [here](#).

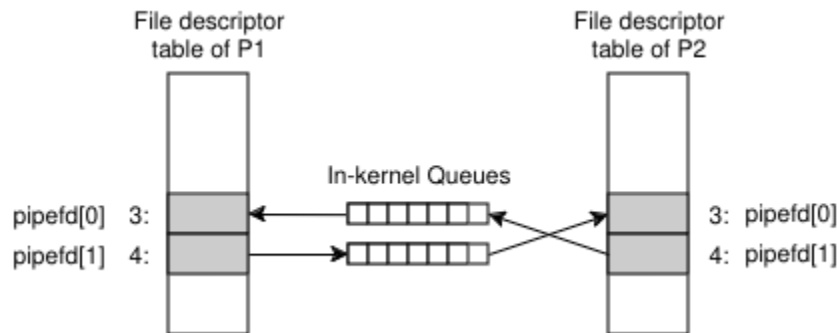
```
int pipe(int pipefd[2]);
```

This above call initializes a pipe and allocates two file descriptors via the arguments passed to the call.

The pipe system call initializes the 0th file descriptor **pipefd[0]** as the read end of the pipe and **pipefd[1]** as the write end of the pipe as shown in the figure.

After the pipe is set up, all writes using via **fd[1]** are connected to an in-kernel pipe (i.e., queue) and can be read using the **fd[0]** file descriptor.

When used along with the **fork** system call, pipes enable Inter-Process Communication (IPC).



2.d.

Write a program **2d.c** that creates two child process and connects the standard output of the first child to the standard input of the second child using the **pipe()** system call. The first child should take an input from the standard input and find the square of the value, which should be written to the standard output (the pipe's write end). The second child should take the squared value from the standard input (the pipe's read end), find the square root of the same, and print the original value in the standard output.

Sample Output:

```
user@users-linux:~$ ./2d
Process A (pid: 1234): Input value of x: 4
Process B (pid: 1235): Got value from A: 16
Process B (pid: 1235): Original value of x: 4
```

2.e.

Write a program to process the file `bigdata.txt` and determine how many times the following words appear, regardless of the case: **the, of, and, a, to** in the data file.

A basic approach would be to write a program that sequentially scans the entire file and counts each word's occurrences. However, a more efficient method involves parallel processing using multiple processes. Write a program named `2e.c` that takes a number as a command-line to specify the number of parallel processes the main process will set up to process the data file (`bigdata.txt`). Each of the child processes works on a part of the file, if there are `n` child processes each of them is assigned $1/n$ th of the data file for processing.

The parent process which part of the data file a child process is responsible for via a dedicated pipe, by sending a message in the following format:

`<start-offset>,<size of data chunk>`

For example, if there are two child processes and the file is 10 bytes long, the message format for each of the child processes will be —

`0, 5` — for one child process and

`5, 5` — for the second child process

Each child process will seek to its designated position in the file and read the assigned portion of the file to perform the word count. Each process should maintain a histogram for the occurrences of the specified words. Once the computation is complete, the child processes should send the computed values back to the parent process through the same pipe.

A child process will convey the count values for each of the words in the following format:

`<the-count>,<of-count>,<and-count>,<a-count>,<to-count>`

After receiving the results from all child processes, the parent process should aggregate the word counts and display the final histogram. Additionally, the program should measure and print the total time taken for the computation. Experiment with different numbers of child processes (1, 2, 4, 8) and observe the impact on performance.

All inter-process communication must be done using the pipe system call.

Sample Output:

```
user@users-linux:~$ ./2e 2
the: 4354
of: 6235
and: 1234
a: 5598
to: 7824
Computation time: 12s
```

Note: If a process tries to read before something is written to the pipe, the process is blocked until something is written and is available in the pipe.

Submission Guidelines

- All submissions via Moodle.
- Place all the files in your submission directory, with the directory name being your roll number.
- Warmup's questions should be answered in a text or pdf file named `warmup.txt` or `warmup.pdf`
- Tar and gzip the directory using the command
`$ tar -zcvf <rollno_lab2>.tar.gz <rollno_lab2>`

Please use this command for creating your submission!

For example; if your roll number is 123456789, then the directory name will be **123456789_lab2**, and submission will be **123456789_lab2.tar.gz**

Everything should be in lower case. And use only `.tar.gz` (~~`.tar.xz`, `.zip`, `.gz`, `.xz`, etc.~~)

- The tar should contain the following files in the following directory structure:

```
<rollnumber_lab2>/
| ____ 1a.c
| ____ 1b.c
| ____ 1c.c
| ____ 1d.c
| ____ 1e.c
| ____ 1f.c
| ____ 1g.c
| ____ 2a.c
| ____ 2b.c
| ____ 2c.c
| ____ 2d.c
| ____ 2e.c
| ____ warmup.txt
```

- **Deadline: 21 August 2024, 11.59 pm. (via Moodle)**
-