LAB 4

CS 361: Systems Programming / Spring 2023

Description

In this lab session, you will explore:

- Function `open`, which allows you to open and create files and obtain a file descriptor
- Function `close`, which allows you to close previously open file descriptors
- Functions `read` and `write`, which allow you to read from and write to file descriptors
- Functions `dup` and `dup2`, which allows you to duplicate and redirect input/output
- Function `pipe`, which allows you to create a unidirectional channel of communication backed by file descriptors
- Standard file descriptors available to every process

Please read this document carefully and follow the instructions on the last section to complete this lab session. When you answered all the questions, please show your work to the TA.

open

Function `open` allows you to create a new file descriptor backed by a file on disk. The signature of function `open` is:

```
int open(const char *pathname, int flags, mode_t mode);
```

Where:

- pathname: the path to the file on disk that you want to open (e.g., "/tmp/out.txt")
- flags: options to configure how the file descriptor should be open
  - You need to specify exactly one of these options:
    - `O_RDONLY`: open the file for reading only
    - `O_WRONLY`: open the file for writing only
    - `O_RDWR`: (use of this flag is discouraged) open the file for reading and writing only
  - You can specify many of these options:
    - `O_APPEND`: write to a file by adding new data to the end, past the existing data already in the file

        - `O_TRUNC`: write to a file by overwriting its contents. The file will be made empty immediately after open.
        - `O_CREAT`: create a new file if the path specified does not exist
            - You have to specify which mode to create a file when using this option
    - You can combine many flags with the bitwise-or operator
        - E.g., `O_WRONLY | O_APPEND` create a file descriptor that will open a file on disk for writing only, and the written data will be added to the end of the existing contents of that file.
- mode: when using `O_CREAT` you need to pass options to specify the mode in which the file will be created
    - You have to select one user, group, and others option (i.e., one column from the table below).
    - You can combine multiple modes with bitwise OR
        - E.g., `S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH` allows the user to read and write to this file, the group to read, and others to read

	User	Group	Others
Execute	`S_IXUSR`	`S_IXGRP`	`S_IXOTH`
Read	`S_IRUSR`	`S_IRGRP`	`S_IROTH`
Write	`S_IWUSR`	`S_IWGRP`	`S_IWOTH`

- Function open returns:
    - A positive number, indicating the file descriptor created
    - A negative number, indicating that an error occurred.
        - Function perror will print the error to the terminal
            - E.g., `perror("Error when opening file");`

close

Function close closes an existing file descriptor. It has the following signature:

```
int close(int fd);
```

- The single argument is the file descriptor to close, indicated by an integer (e.g., returned by open)
- After calling close, it is not possible to use that file descriptor any further
    - Creating a new file descriptor through open may reuse the same integer of a previously closed file descriptor
- This function returns zero on success, -1 when it fails to close the file
    - Function `perror` can print the error to the terminal

read / write

Function `read` and allow to read from / write to open file descriptors. Their signature is as follows:

```
ssize_t  read(int fd,       void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

These functions take the following arguments:

- `fd`: which file descriptor to read from or write to. The file descriptor should be created with the correct flags (e.g., `read` must use file descriptors created by open with flags `O_RDONLY` or `O_RDWR`)
- `buf`:
  - `read`: a pointer to allocated memory where the data read will be stored
  - `write`: a pointer to memory containing the data to be written
- `count`:
  - `read`: maximum number of bytes to read and store in the memory pointed to by `buf`
  - `write`: maximum number of bytes to write starting from the memory pointed to by `buf`
- These functions return an integer indicating the number of bytes read or written
  - Positive number: these bytes were read and stored in buf / written from buf
  - Zero: when reading, indicates that we reached the end of the file
    - Reading from this file descriptor again will result in an error
  - Negative number: an error occurred when reading / writing
    - Function `perror` can print the error to the terminal
- Examples:
  - Read 10 bytes from file descriptor 0

```
int fd = open("/path/to/file", O_RDONLY);
char * buffer = // allocate at least 10 bytes of memory
int count = read(fd, buffer, 10);
if (count == 0) // handle end of file
else if (count < 0) // handle error
else if (count < 10) // less data was read than requested
```

  - Write 10 bytes to file descriptor 1

```
int fd = open("/path/to/file", O_WRONLY | O_APPEND);
char * buffer = "123456789\n";
                    // memory at least 10 bytes long to be written
int count = write(fd, buffer, 10);
if (count < 0) // handle error
else if (count < 10) // less data was written than requested
```

dup

Function `dup` allows you to duplicate an existing file descriptor into a new file descriptor. Its signature is as follows:

```
int dup(int fd);
```

On success, it duplicates the file descriptor provided and returns a new small positive integer, which is the new file descriptor. On error, it returns -1. You can check the cause of the error with function `perror`.

For instance, the following program prints "Hello CS361" to the output:

```
int fd = dup(1);
close(1);
char buffer[] = "Hello CS361";
write(fd, buffer, sizeof(buffer));
```

This function is good to keep file descriptors from being closed when redirecting input as described below. After redirection, `dup` 'ed file descriptors can be restored easily.

dup2

Function `dup2` allows to redirect one file descriptor to another. Its signature is as follows:

```
int dup2(int oldfd, int newfd);
```

This function takes the following arguments:

- `oldfd`: file descriptor that will replace another existing file descriptor
- `newfd`: file descriptor that will be replaced such that `oldfd` and `newfd` reference the same file. The replaced file descriptor is closed.

It returns the new file descriptor if successful. Otherwise, it returns -1 and function `perror` can print the error to the terminal.

For instance, consider the following program:

```
int fd1 = open("hello.txt", O_WRONLY | O_TRUNC);
int fd2 = open("goodbye.txt", O_WRONLY | O_TRUNC);

int res = dup2(fd1, fd2);
if (res < 0) perror("Error when performing dup2");

char buffer[] = "Hi there!";
write(fd2, buffer, sizeof(buffer));
```

This program writes `Hi there!` to file `hello.txt`

Pipe

Function `pipe` creates a unidirectional channel of communication. Example:

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

int read_fd = fds[0];
int write_fd = fds[1];
```

Function `pipe` creates two file descriptors and stores it in the int array passed as argument. Position 0 holds the read side of the pipe, position 1 holds the write side of the pipe:

```
char buffer1[] = "CS361";
char buffer2[10];

write(write_fd, buffer1, sizeof(buffer1));
read(read_fd, buffer2, sizeof(buffer2));
```

The program above results in the string `CS361` being written to buffer1 and then read into buffer2.

Note that the file descriptors created by pipe survive fork. In this case, pipe can be used to establish communication between parent and child:

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

int read_fd = fds[0];
int write_fd = fds[1];

if (fork() != 0) {
  close(write_fd);
  char buffer[20];
  read(read_fd, buffer, 20);
} else {
  close(read_fd);
  write(write_fd, "Hi from child", 13);
}
```

The program above results in the child process sending the message "Hi from child" to the parent process via the pipe. It is always a good idea to close the end of the pipe that the process will not use.

Standard file descriptors

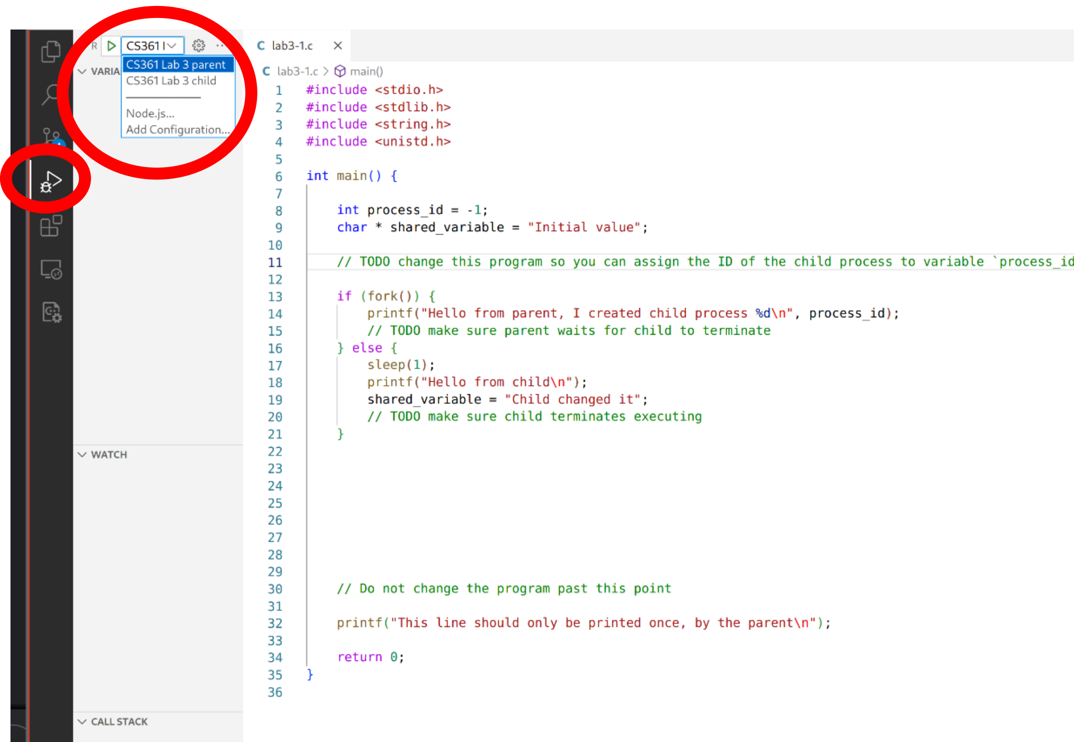
Every process inherits all open file descriptors from its parent on fork. All processes have the following file descriptors by default:

0. Standard in, reading from this file descriptor reads from the console
1. Standard out, writing to this file descriptor write to the console
2. Standard error, writing to this file descriptor also writes to the console

These file descriptors can be redirected using `dup2`.

Guide

1. Accept the invitation for Lab 3 on Github classroom: https://classroom.github.com/a/ymgGheq2
2. Import the Github repository created to your machine using vscode, as explained in Assignment 0
3. Make sure that you can launch a terminal inside vscode via menus: Terminal > New Terminal
4. The repository has three files that you need: `lab4-1.c`, `lab4-2.c`, `lab4-3.c`.
5. Type `make` on the terminal and then run file lab3-1 by typing `./lab4-1` on the terminal. Observe the output. Repeat for files lab4-2 and lab4-3.
6. Similarly to Lab 3, you can run files under the debugger, using both the parent and child configurations depicted below.



7. Modify file `lab4-1.c` to answer Question 1.
8. Modify file `lab4-2.c` to answer Questions 2 and 3.

9. Modify file `lab4-3.c` to answer Question 4.
10. Show your work to the TA, commit and push your changes to the repo created in Step 2.

Questions

You can write the answers to your questions to file `ans.txt` in your repository, it then makes it easy to show your work to the TA.

1. How did you change lab4-1.c so that running it creates file `out.txt` (or overwrites it if it exists), such that the contents of the file are the duplicated contents of file `in.txt` ?

```
CS361 is awesome
CS361 is awesome
```

2. How did you change lab4-2.c so that it repeats what it reads from the terminal twice on the terminal, and twice on file out.txt.
3. How would you change lab4-2.c so that it does not output anything to the terminal? Feel free to comment out any line, including lines that you are not allowed to modify.
4. How did you change lab4-3.c so that the parent receives a message from the child, and then prints it to the terminal?

Extra / Optional

1. How would you change lab4-2.c such that it forks and prints the message it receives from the child to the terminal and to file out.txt, instead of reading it from the terminal?

Grading

Show your UIC card to the TA when you enter the lab, or type your UIN on the chat when joining remotely. Stay in the session until you show your work, or until the TA announces that the lab is over.

- You have to remain present for the whole lab to get attendance, which you can then use to resubmit Assignment 2.
- You can leave early after showing your work to the TA (answers to all questions). In this case, you will get a 5% bonus in Assignment 2.