Lab02

# E/19/166

# Jayathunga W.W.K.

**Exercise 1.1**

1. **O\_WRONLY:** This flag opens the file for writing, retains the existing file contents, and puts the file pointer at the start of the file. Any writes overwrite existing content.

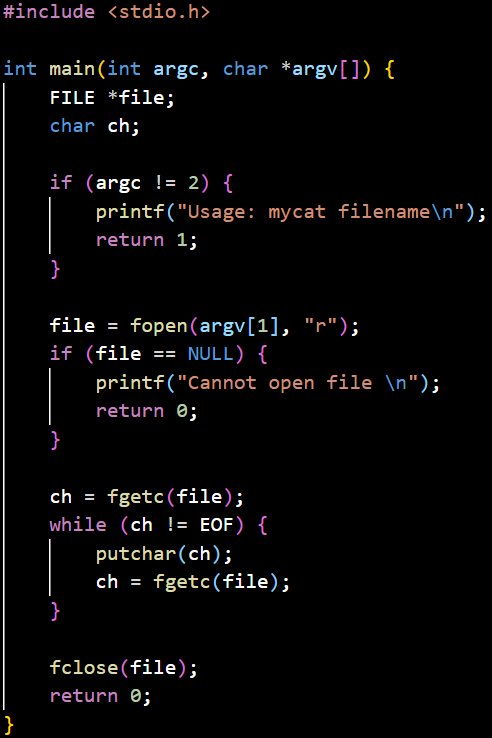
**O\_APPEND:** This flag causes writes to append to the end of the file instead of overwriting at the start. This flag is persistent. If we move the cursor elsewhere to read data, it’s always repositioned to the end of the file before each write.

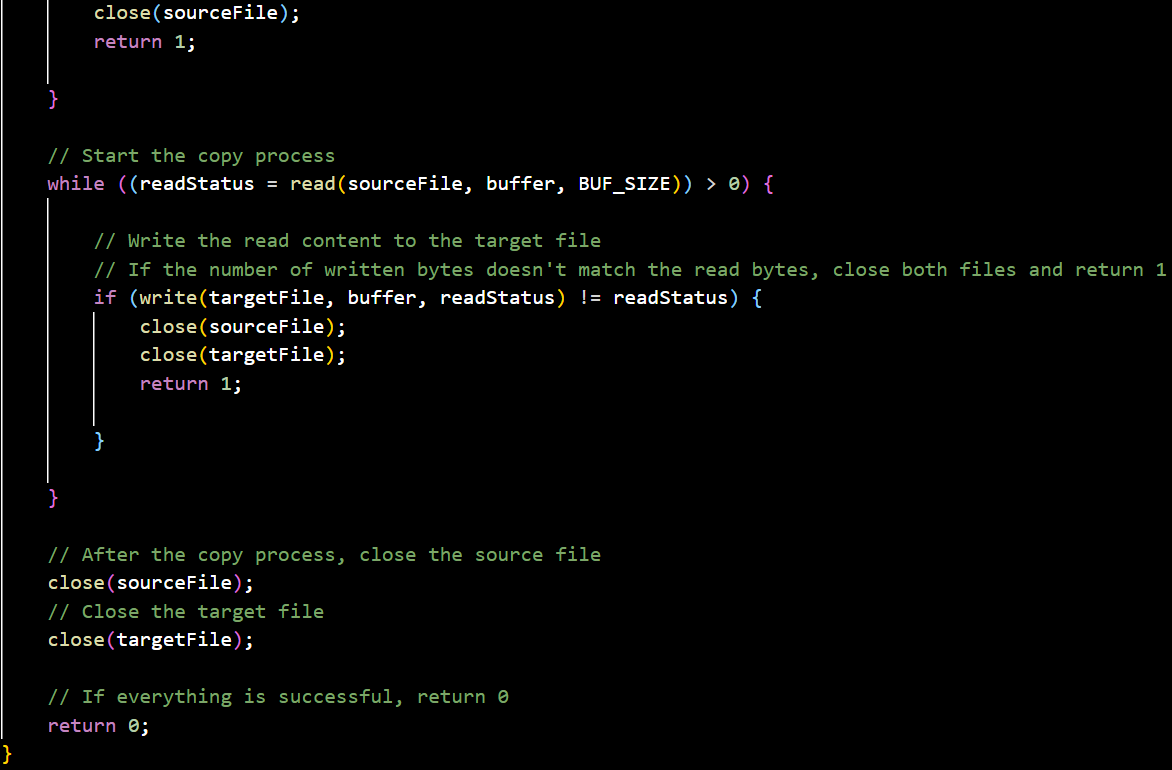
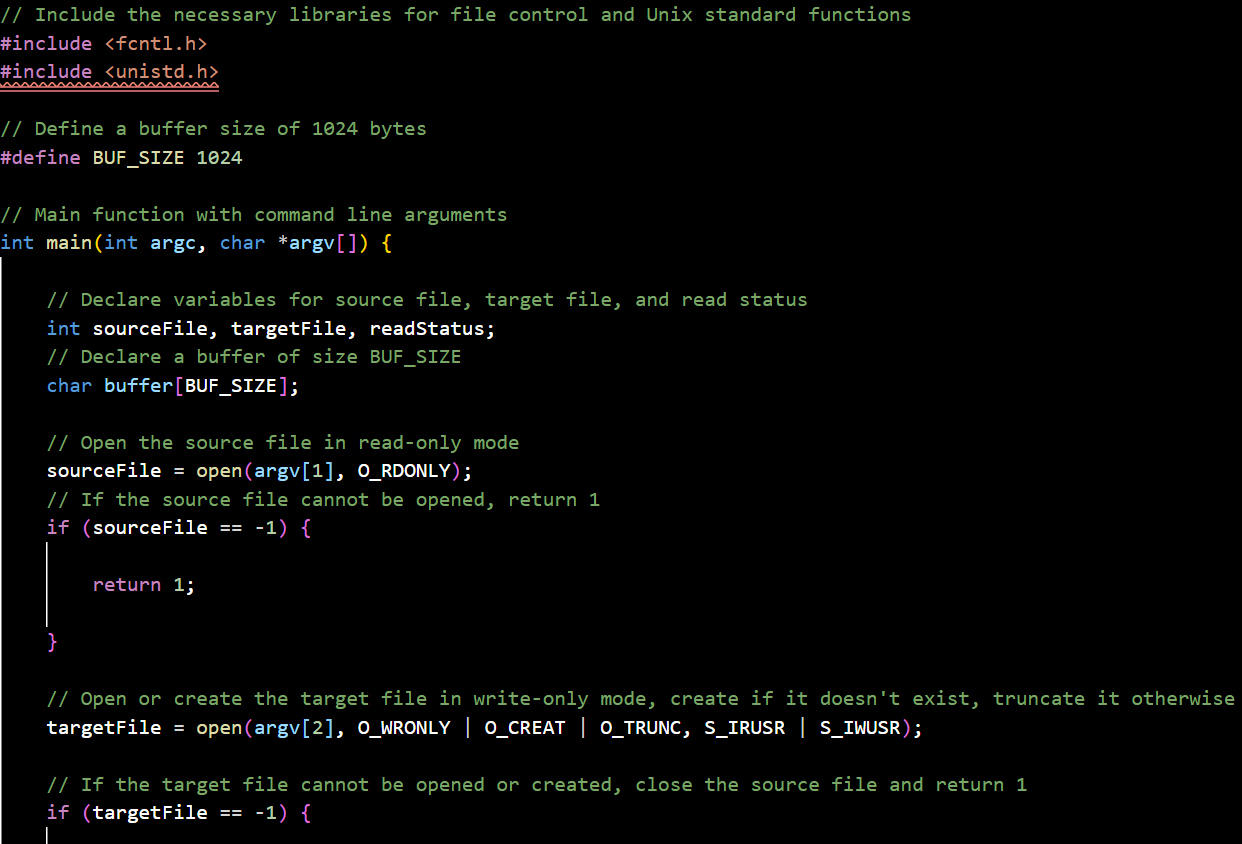
**O\_CREAT:** This flag is used to create a file if it does not exist.

1. **S\_IRUSR:** This mode sets read permission for the owner of the file. On many systems, this bit is 0400.

**S\_IWUSR:** This mode sets write permission for the owner of the file. Usually, this bit is 0200.

**Exercise 1.2**





**Exercise 2.1**

1. writes ‘count’ bytes from the buffer ‘buff’ directly to the standard output (usually the terminal). This system call bypasses any additional formatting or interpretation that higher-level functions like ‘printf’ might apply, making it suitable for raw data output.
2. Using a single unnamed pipe for bidirectional communication is generally not possible due to the inherent design of pipes. A pipe, created with the pipe(int pipefd[2]) function, is unidirectional. This means data can only flow in one direction—from the write-end (pipefd[1]) to the read-end (pipefd[0]).

Attempting to use the same pipe for both reading and writing in both directions would lead to confusion and possible deadlocks, as there’s no built-in mechanism to distinguish whether data is meant for reading or writing.

1. Unnamed pipes cannot be used to communicate between unrelated processes due to the following reasons:
2. Creation and Inheritance Mechanism:

**Pipe Creation:**

Unnamed pipes are created using the pipe(int pipefd[2]) system call, which provides two file descriptors: one for reading and one for writing.

**File Descriptor Inheritance:**

These file descriptors are only available in the creating process and its children.

When a process forks, the child process inherits the file descriptors from the parent process, allowing both processes to use the pipe for communication.

1. File Descriptor Scope:

**Process-Specific:**

File descriptors in Unix-like systems are specific to the process in which they are created. They are not globally accessible and cannot be shared directly with unrelated processes.

**Lack of a Common Handle:**

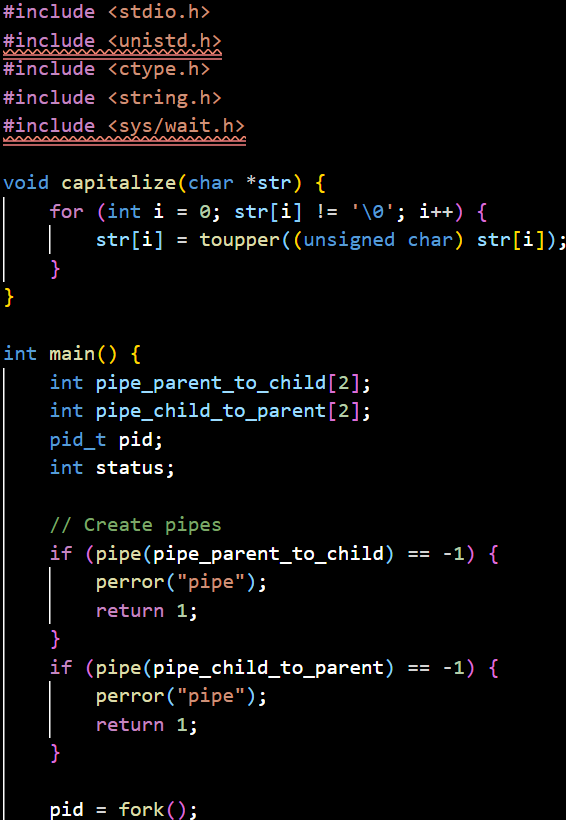
Unnamed pipes lack a global identifier or handle that can be passed to other unrelated processes to access the pipe.

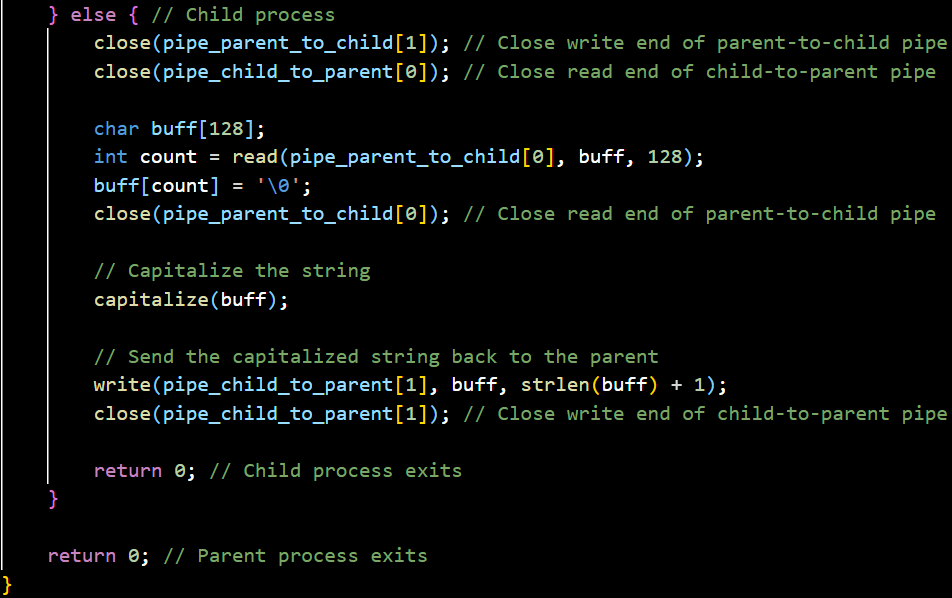
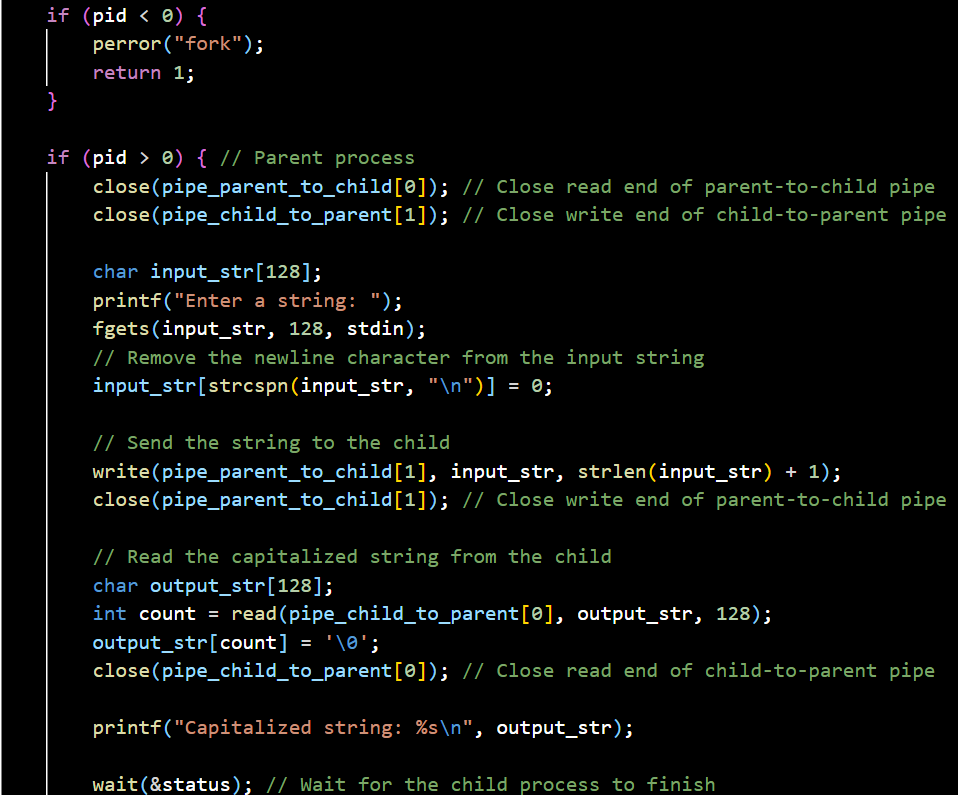
1. Inter-Process Communication (IPC) Requirements:

**Shared Memory or Named Objects:**

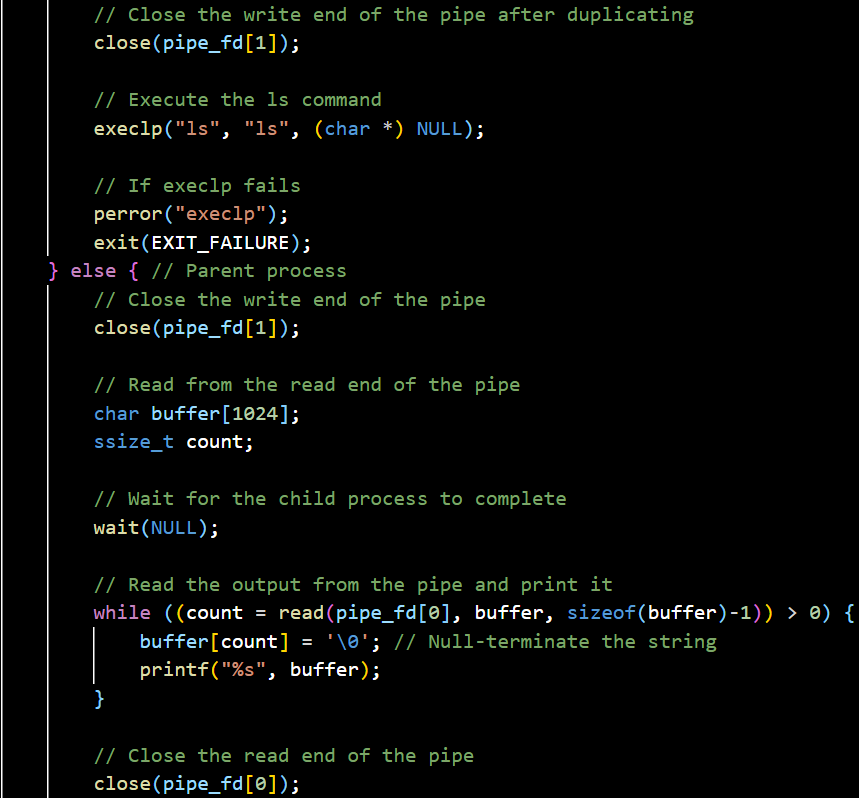
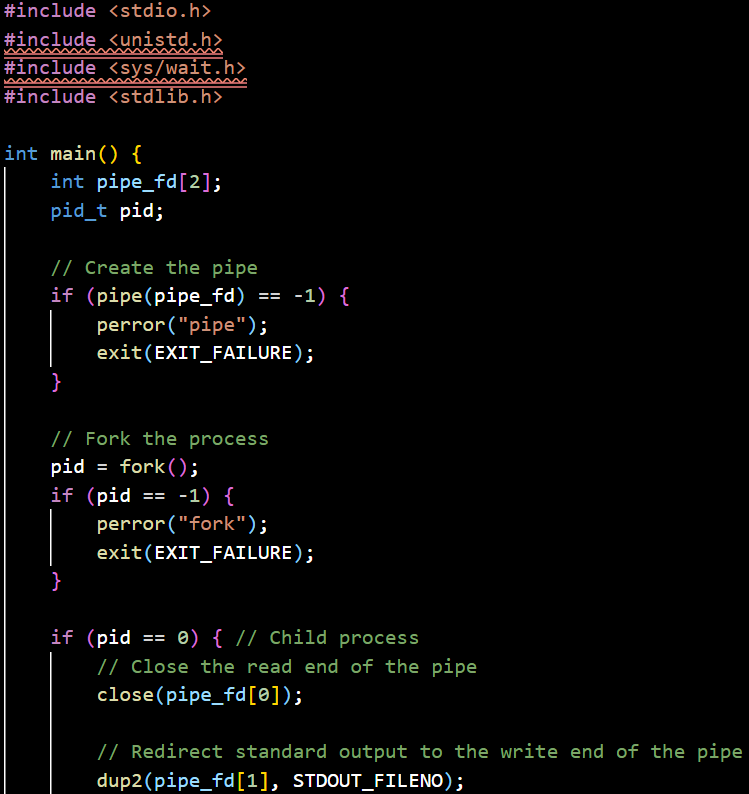
Unrelated processes typically need to use named IPC mechanisms such as named pipes (FIFOs), message queues, shared memory, or sockets.

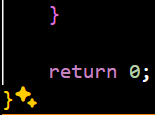
Named pipes (created using mkfifo) provide a file path that unrelated processes can use to open and communicate through the pipe.





**Exercise 3.1**





**Exercise 3.2**

1. In the line dup2(out, 1); from the provided program, the 1 represents the file descriptor for the standard output (stdout). File descriptors are integer handles used by the operating system to access files or input/output streams. By convention, the first three file descriptors are:

0: Standard input (stdin)

1: Standard output (stdout)

2: Standard error (stderr)

So, when we call dup2(out, 1);, we are duplicating the file descriptor out and making it the new file descriptor for stdout. This means that any output that would normally go to the terminal (standard output) will instead be written to the file referred to by out, which in this case is the file out opened earlier in the code with the open system call.

1. i. dup()

**Syntax:** int dup(int oldfd);

**Functionality:** Duplicates the file descriptor oldfd, returning a new file descriptor that refers to the same open file description.

**Behavior:** The new file descriptor returned by dup() is the lowest-numbered unused file descriptor.

**Use Case:** When we need to duplicate a file descriptor but don't care what the new descriptor number will be.

dup2()

**Syntax:** int dup2(int oldfd, int newfd);

**Functionality:** Duplicates the file descriptor oldfd to newfd, closing newfd first if it is already open.

**Behavior:** Guarantees that newfd will be the new file descriptor referring to the same open file description as oldfd.

**Use Case:** When we need to duplicate a file descriptor to a specific number, such as redirecting standard input/output or error streams.

**Necessity of Both Functions**

dup() and dup2() serve different purposes and both are necessary because:

dup() Use Case:

When we don't need a specific file descriptor number.

When we want to get a new file descriptor for a file and we are not concerned about the specific number.

Example: Duplicating a file descriptor in a scenario where we simply need another handle to the same file without affecting standard I/O streams.

dup2() Use Case:

When we need to duplicate a file descriptor to a specific file descriptor number, which is crucial for redirection of standard streams (stdin, stdout, stderr).

When we need precise control over which file descriptor is used, such as replacing standard I/O file descriptors in processes.

Example: Redirecting standard output to a file or pipe, as seen in the provided program where dup2(pipefd[1], 1) is used to redirect stdout to the pipe.

ii. Yes, there is a significant error in the provided code. The main issue lies in how the parent process and child process handle the closing of standard output and standard input respectively. Specifically, the error occurs when the parent process attempts to execute the cat command after closing its standard output.

**Identified Error**

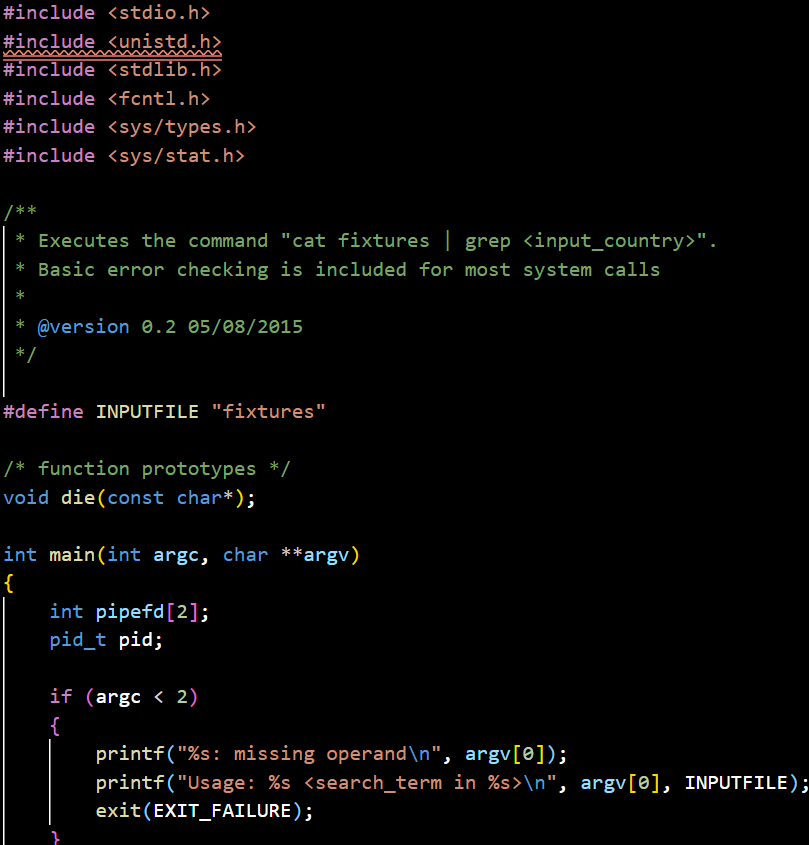
The parent process closes standard output (file descriptor 1) before duplicating pipefd[1] to it using dup(pipefd[1]). However, the parent process never actually closes the pipefd[1] after duplicating it. This is an issue because pipefd[1] should be closed after it has been duplicated to standard output. This can cause the pipe to remain open for writing, which means the grep command in the child process may not receive an end-of-file (EOF) signal, potentially causing it to hang.

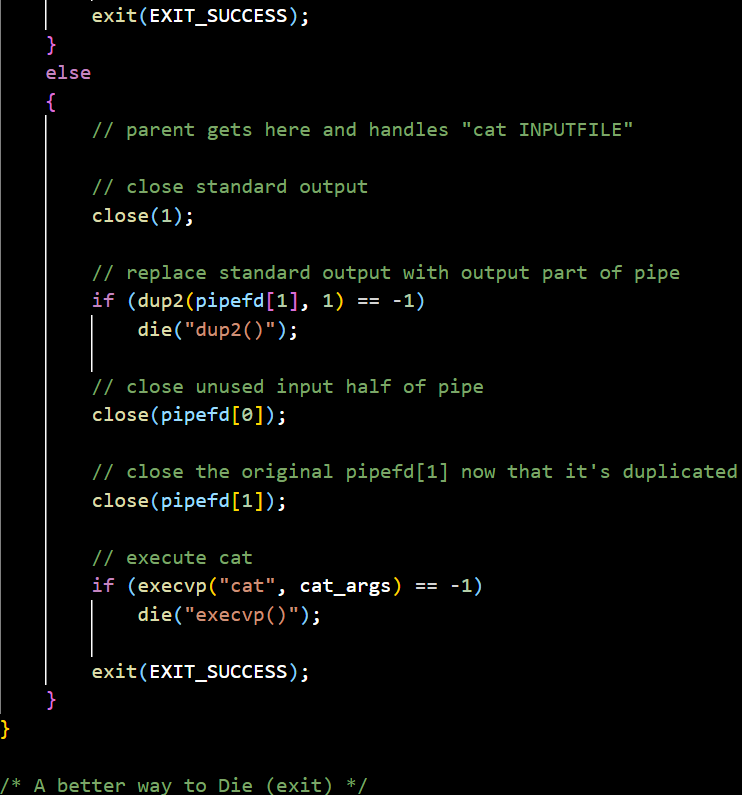
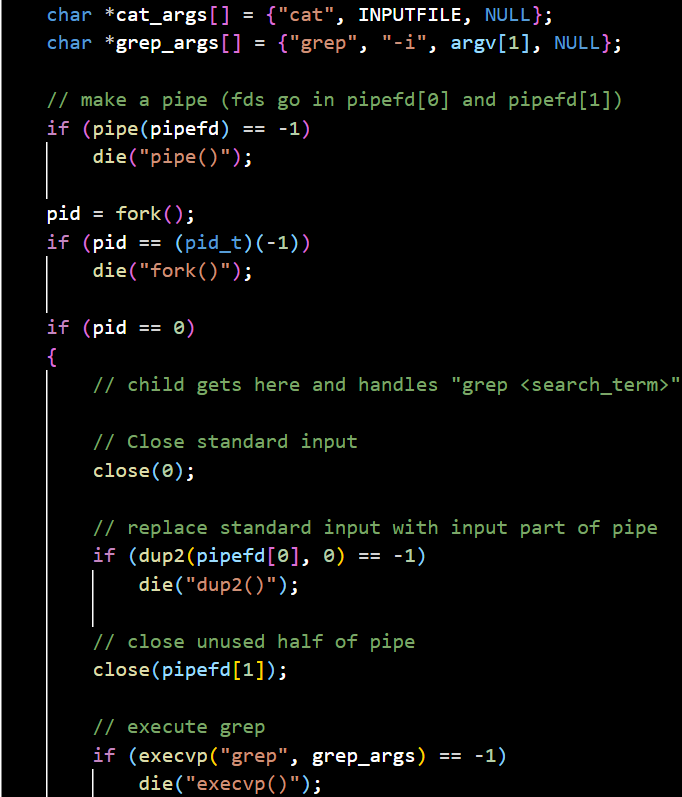
**Additional Errors and Recommendations**

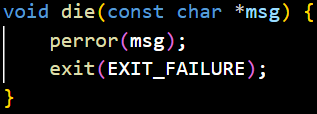
Use of dup instead of dup2: It is generally better practice to use dup2 when we need to duplicate a file descriptor to a specific descriptor number. Using dup2 makes the code clearer and safer.

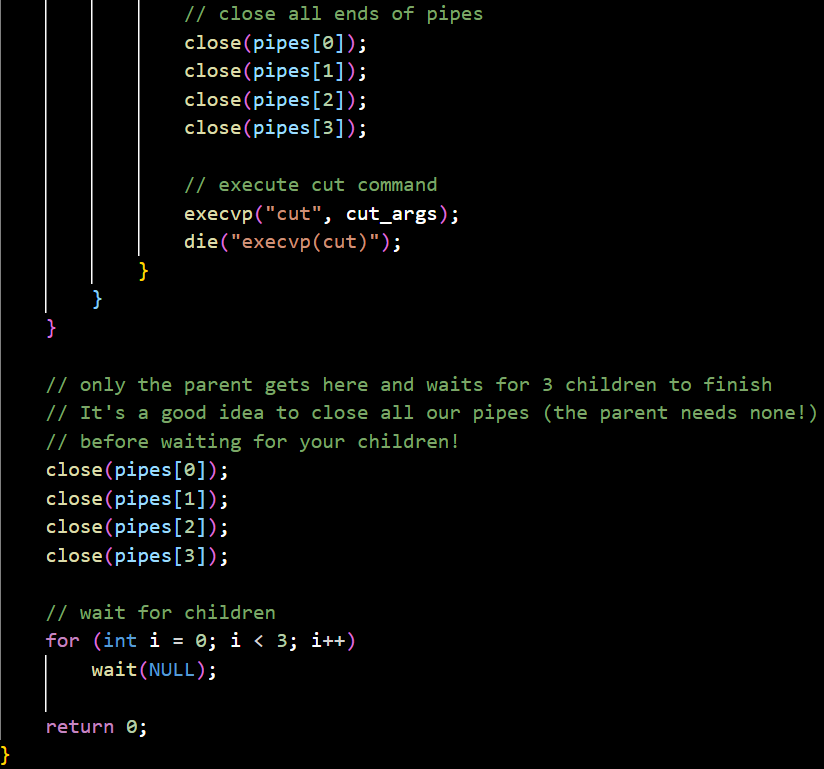
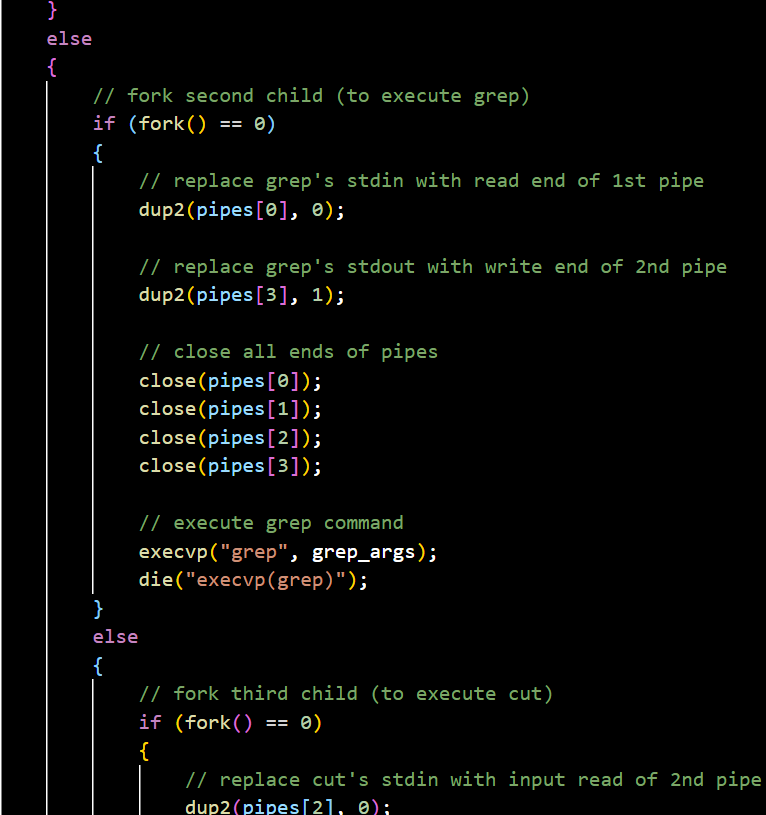
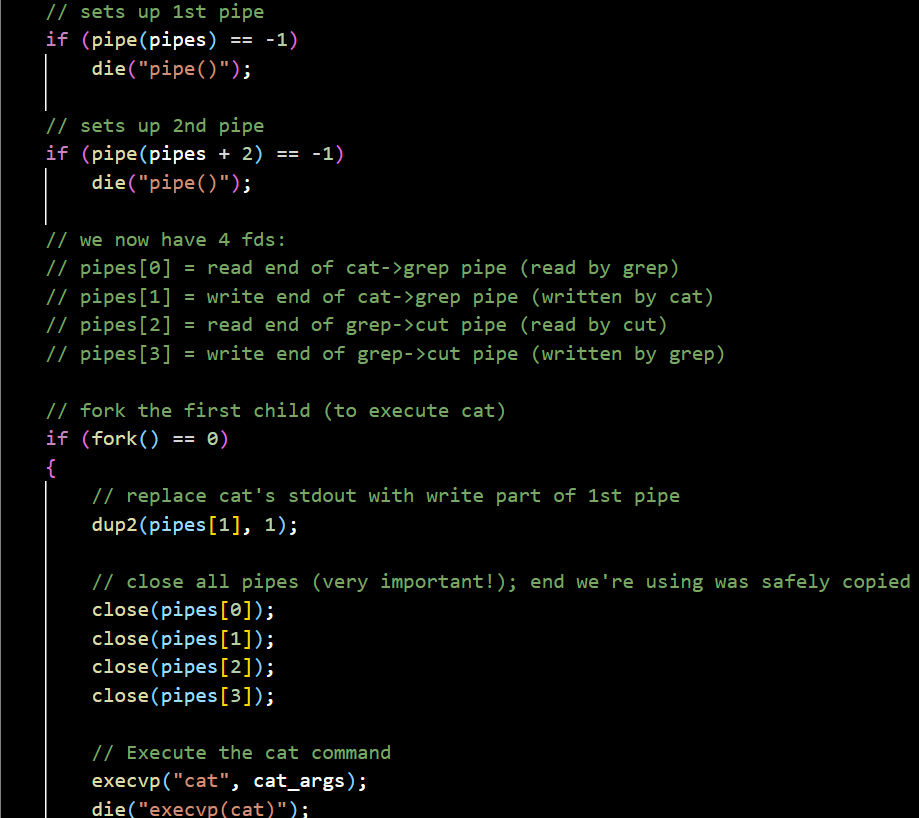
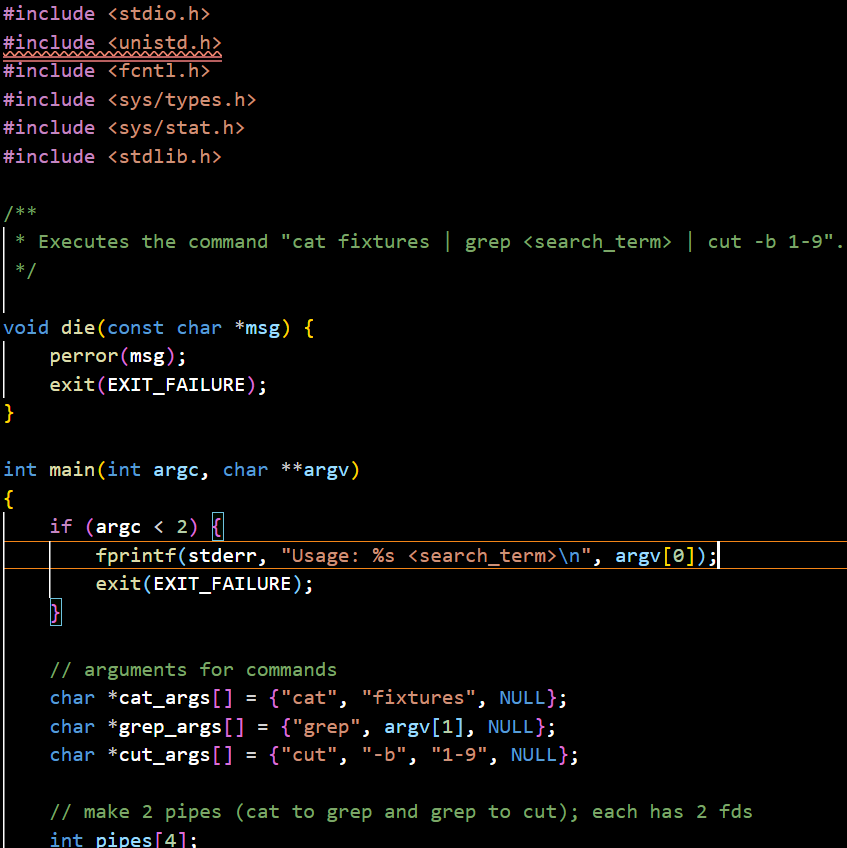
Error handling consistency: Although basic error checking is included, it would be better to ensure all system calls are consistently checked for errors.

iii.









**Exercise 4.1**

1. **Test 1: Commenting out mkfifo(fifo, 0666); in the Reader**

Comment out mkfifo(fifo, 0666); in the reader:

// mkfifo(fifo, 0666);

Run the writer first, then run the reader:

Writer Output: The writer might block indefinitely on the open(fifo, O\_WRONLY) call because the named pipe does not exist. The writer is trying to open a non-existent file for writing, which results in blocking until a reader creates the pipe.

Reader Output: When run after the writer, the reader will fail immediately because it tries to open a named pipe that doesn't exist, resulting in an error.

Run the reader first, then run the writer:

Reader Output: The reader will fail immediately because it tries to open a named pipe that doesn't exist.  
Writer Output: If the reader fails and exits, the writer, when subsequently run, will also block indefinitely on the open(fifo, O\_WRONLY) call.

**Test 2: Commenting out mkfifo(fifo, 0666); in the Writer**

Comment out mkfifo(fifo, 0666); in the writer:

// mkfifo(fifo, 0666);

Run the writer first, then run the reader:

Writer Output: The writer will fail immediately because it tries to open a named pipe that doesn't exist.

Reader Output: When run after the writer, the reader will also fail because the named pipe still doesn't exist.

Run the reader first, then run the writer:

Reader Output: The reader might block indefinitely on the open(fifo, O\_RDONLY) call because the named pipe does not exist. The reader is trying to open a non-existent file for reading, which results in blocking until a writer creates the pipe.

Writer Output: If the reader blocks indefinitely, the writer will never get a chance to run, and when it does, it will fail because the pipe was never created.

**Why This Happens**

Named Pipe Creation: The mkfifo system call creates a named pipe (FIFO) in the filesystem. This call must be made before any process attempts to open the pipe for reading or writing. If the named pipe does not exist, any attempt to open it will fail (if the process is the first to open for reading) or block indefinitely (if the process is the first to open for writing).

Blocking Behavior: The open system call on a named pipe is blocking if it is the first call to open for writing or reading, and the corresponding reading or writing end has not yet been opened. This blocking behavior ensures synchronization between the reader and writer but can lead to indefinite blocking if the pipe does not exist.

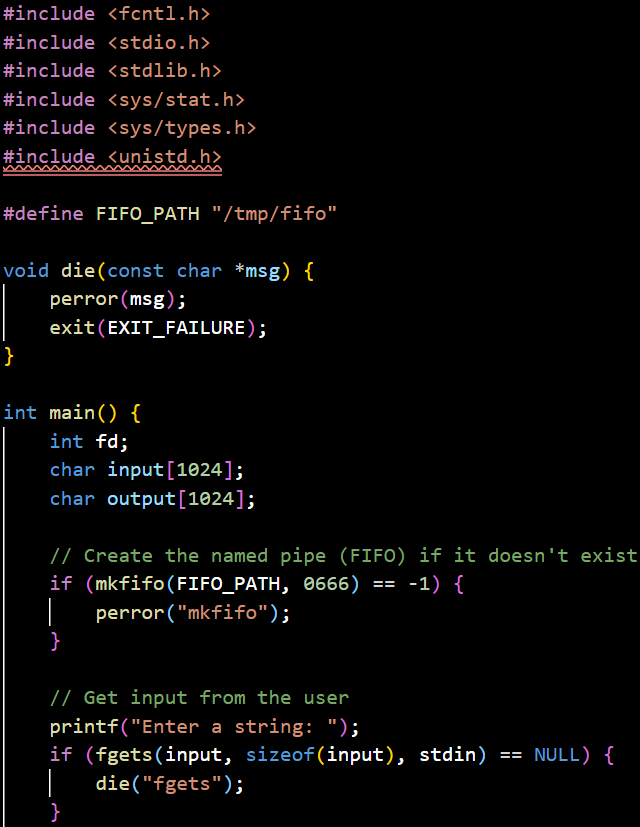
**Debugging Nightmare**

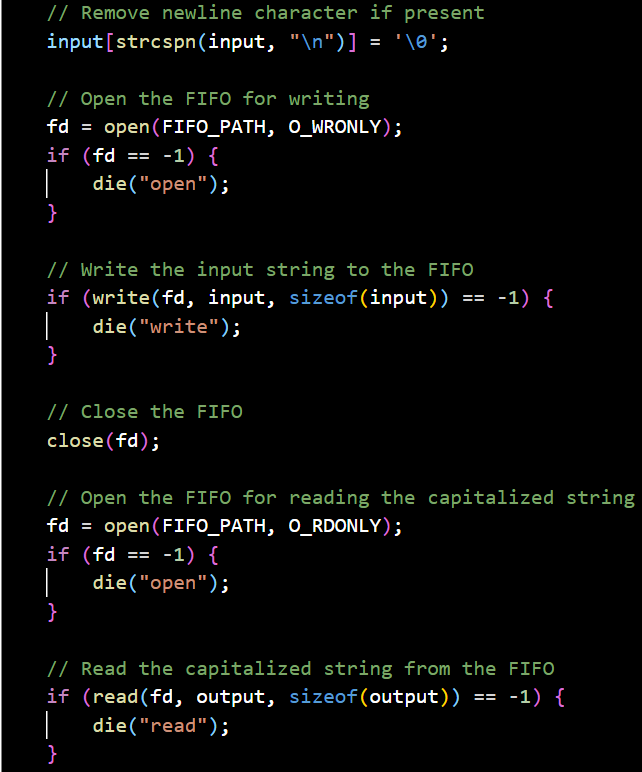
Inconsistent State: If the mkfifo call is omitted, the state of the filesystem is inconsistent with the expectations of the program. The named pipe is not created, leading to failures or indefinite blocking, which is difficult to trace back to the missing mkfifo call.

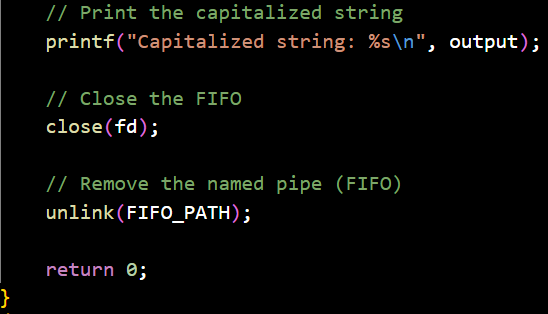
Intermittent Failures: Depending on the order of execution (reader first or writer first), the program might fail immediately or block indefinitely. This inconsistent behavior can be confusing and time-consuming to debug.

Silent Errors: If the pipe is not created and the process blocks indefinitely, there might be no immediate indication of what went wrong, especially in a larger program with multiple points of failure.

1. Sender.c







Receiver.c

