Lab02

E/19/166

Jayathunga W.W.K.

Exercise 1.1

a. **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.

b. **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

a.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    FILE *file;
    char ch;

    if (argc != 2) {
        printf("Usage: mycat filename\n");
        return 1;
    }

    file = fopen(argv[1], "r");
    if (file == NULL) {
        printf("Cannot open file \n");
        return 0;
    }

    ch = fgetc(file);
    while (ch != EOF) {
        putchar(ch);
        ch = fgetc(file);
    }

    fclose(file);
    return 0;
}
```

b.

```c
// Include the necessary libraries for file control and Unix standard functions
#include <fcntl.h>
#include <unistd.h>

// Define a buffer size of 1024 bytes
#define BUF_SIZE 1024

// Main function with command line arguments
int main(int argc, char *argv[]) {

    // Declare variables for source file, target file, and read status
    int sourceFile, targetFile, readStatus;
    // Declare a buffer of size BUF_SIZE
    char buffer[BUF_SIZE];

    // Open the source file in read-only mode
    sourceFile = open(argv[1], O_RDONLY);
    // If the source file cannot be opened, return 1
    if (sourceFile == -1) {

        return 1;

    }

    // Open or create the target file in write-only mode, create if it doesn't exist, truncate it otherwise
    targetFile = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);

    // If the target file cannot be opened or created, close the source file and return 1
    if (targetFile == -1) {

        close(sourceFile);
        return 1;

    }

    // Start the copy process
    while ((readStatus = read(sourceFile, buffer, BUF_SIZE)) > 0) {

        // Write the read content to the target file
        // If the number of written bytes doesn't match the read bytes, close both files and return 1
        if (write(targetFile, buffer, readStatus) != readStatus) {
            close(sourceFile);
            close(targetFile);
            return 1;

        }

    }

    // After the copy process, close the source file
    close(sourceFile);
    // Close the target file
    close(targetFile);

    // If everything is successful, return 0
    return 0;
}
```

Exercise 2.1

a. 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.

b. 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.

c. Unnamed pipes cannot be used to communicate between unrelated processes due to the following reasons:

1. 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.

2. 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.

3. 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.

d.

```c
#include <stdio.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>
#include <sys/wait.h>

void capitalize(char *str) {
    for (int i = 0; str[i] != '\0'; i++) {
        str[i] = toupper((unsigned char) str[i]);
    }
}

int main() {
    int pipe_parent_to_child[2];
    int pipe_child_to_parent[2];
    pid_t pid;
    int status;

    // Create pipes
    if (pipe(pipe_parent_to_child) == -1) {
        perror("pipe");
        return 1;
    }
    if (pipe(pipe_child_to_parent) == -1) {
        perror("pipe");
        return 1;
    }

    pid = fork();
```

```c
    if (pid < 0) {
        perror("fork");
        return 1;
    }

    if (pid > 0) { // Parent process
        close(pipe_parent_to_child[0]); // Close read end of parent-to-child pipe
        close(pipe_child_to_parent[1]); // Close write end of child-to-parent pipe

        char input_str[128];
        printf("Enter a string: ");
        fgets(input_str, 128, stdin);
        // Remove the newline character from the input string
        input_str[strcspn(input_str, "\n")] = 0;

        // Send the string to the child
        write(pipe_parent_to_child[1], input_str, strlen(input_str) + 1);
        close(pipe_parent_to_child[1]); // Close write end of parent-to-child pipe

        // Read the capitalized string from the child
        char output_str[128];
        int count = read(pipe_child_to_parent[0], output_str, 128);
        output_str[count] = '\0';
        close(pipe_child_to_parent[0]); // Close read end of child-to-parent pipe

        printf("Capitalized string: %s\n", output_str);

        wait(&status); // Wait for the child process to finish
    } else { // Child process
        close(pipe_parent_to_child[1]); // Close write end of parent-to-child pipe
        close(pipe_child_to_parent[0]); // Close read end of child-to-parent pipe

        char buff[128];
        int count = read(pipe_parent_to_child[0], buff, 128);
        buff[count] = '\0';
        close(pipe_parent_to_child[0]); // Close read end of parent-to-child pipe

        // Capitalize the string
        capitalize(buff);

        // Send the capitalized string back to the parent
        write(pipe_child_to_parent[1], buff, strlen(buff) + 1);
        close(pipe_child_to_parent[1]); // Close write end of child-to-parent pipe

        return 0; // Child process exits
    }

    return 0; // Parent process exits
}
```

Exercise 3.1

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
    int pipe_fd[2];
    pid_t pid;

    // Create the pipe
    if (pipe(pipe_fd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork the process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) { // Child process
        // Close the read end of the pipe
        close(pipe_fd[0]);

        // Redirect standard output to the write end of the pipe
        dup2(pipe_fd[1], STDOUT_FILENO);
```

```
        // Close the write end of the pipe after duplicating
        close(pipe_fd[1]);

        // Execute the ls command
        execlp("ls", "ls", (char *) NULL);

        // If execlp fails
        perror("execlp");
        exit(EXIT_FAILURE);
    } else { // Parent process
        // Close the write end of the pipe
        close(pipe_fd[1]);

        // Read from the read end of the pipe
        char buffer[1024];
        ssize_t count;

        // Wait for the child process to complete
        wait(NULL);

        // Read the output from the pipe and print it
        while ((count = read(pipe_fd[0], buffer, sizeof(buffer)-1)) > 0) {
            buffer[count] = '\0'; // Null-terminate the string
            printf("%s", buffer);
        }

        // Close the read end of the pipe
        close(pipe_fd[0]);
    }

    return 0;
}
```

Exercise 3.2

a. 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.

b. 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.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/**
 * Executes the command "cat fixtures | grep <input_country>".
 * Basic error checking is included for most system calls
 *
 * @version 0.2 05/08/2015
 */

#define INPUTFILE "fixtures"

/* function prototypes */
void die(const char*);

int main(int argc, char **argv)
{
    int pipefd[2];
    pid_t pid;

    if (argc < 2)
    {
        printf("%s: missing operand\n", argv[0]);
        printf("Usage: %s <search_term in %s>\n", argv[0], INPUTFILE);
        exit(EXIT_FAILURE);
    }
```

```
    char *cat_args[] = {"cat", INPUTFILE, NULL};
    char *grep_args[] = {"grep", "-i", argv[1], NULL};

    // make a pipe (fds go in pipefd[0] and pipefd[1])
    if (pipe(pipefd) == -1)
        die("pipe()");

    pid = fork();
    if (pid == (pid_t)(-1))
        die("fork()");

    if (pid == 0)
    {
        // child gets here and handles "grep <search_term>"

        // Close standard input
        close(0);

        // replace standard input with input part of pipe
        if (dup2(pipefd[0], 0) == -1)
            die("dup2()");

        // close unused half of pipe
        close(pipefd[1]);

        // execute grep
        if (execvp("grep", grep_args) == -1)
            die("execvp()");
        exit(EXIT_SUCCESS);
    }
    else
    {
        // parent gets here and handles "cat INPUTFILE"

        // close standard output
        close(1);

        // replace standard output with output part of pipe
        if (dup2(pipefd[1], 1) == -1)
            die("dup2()");

        // close unused input half of pipe
        close(pipefd[0]);

        // close the original pipefd[1] now that it's duplicated
        close(pipefd[1]);

        // execute cat
        if (execvp("cat", cat_args) == -1)
            die("execvp()");

        exit(EXIT_SUCCESS);
    }
}

/* A better way to Die (exit) */
void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}
```

C.

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

/**
 * Executes the command "cat fixtures | grep <search_term> | cut -b 1-9".
 */

void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

int main(int argc, char **argv)
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <search_term>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // arguments for commands
    char *cat_args[] = {"cat", "fixtures", NULL};
    char *grep_args[] = {"grep", argv[1], NULL};
    char *cut_args[] = {"cut", "-b", "1-9", NULL};

    // make 2 pipes (cat to grep and grep to cut); each has 2 fds
    int pipes[4];
    // sets up 1st pipe
    if (pipe(pipes) == -1)
        die("pipe()");

    // sets up 2nd pipe
    if (pipe(pipes + 2) == -1)
        die("pipe()");

    // we now have 4 fds:
    // pipes[0] = read end of cat->grep pipe (read by grep)
    // pipes[1] = write end of cat->grep pipe (written by cat)
    // pipes[2] = read end of grep->cut pipe (read by cut)
    // pipes[3] = write end of grep->cut pipe (written by grep)

    // fork the first child (to execute cat)
    if (fork() == 0)
    {
        // replace cat's stdout with write part of 1st pipe
        dup2(pipes[1], 1);

        // close all pipes (very important!); end we're using was safely copied
        close(pipes[0]);
        close(pipes[1]);
        close(pipes[2]);
        close(pipes[3]);

        // Execute the cat command
        execvp("cat", cat_args);
        die("execvp(cat)");
```

```
    }
    else
    {
        // fork second child (to execute grep)
        if (fork() == 0)
        {
            // replace grep's stdin with read end of 1st pipe
            dup2(pipes[0], 0);

            // replace grep's stdout with write end of 2nd pipe
            dup2(pipes[3], 1);

            // close all ends of pipes
            close(pipes[0]);
            close(pipes[1]);
            close(pipes[2]);
            close(pipes[3]);

            // execute grep command
            execvp("grep", grep_args);
            die("execvp(grep)");
        }
        else
        {
            // fork third child (to execute cut)
            if (fork() == 0)
            {
                // replace cut's stdin with input read of 2nd pipe
                dup2(pipes[2], 0);
                // close all ends of pipes
                close(pipes[0]);
                close(pipes[1]);
                close(pipes[2]);
                close(pipes[3]);

                // execute cut command
                execvp("cut", cut_args);
                die("execvp(cut)");
            }
        }
    }

    // only the parent gets here and waits for 3 children to finish
    // It's a good idea to close all our pipes (the parent needs none!)
    // before waiting for your children!
    close(pipes[0]);
    close(pipes[1]);
    close(pipes[2]);
    close(pipes[3]);

    // wait for children
    for (int i = 0; i < 3; i++)
        wait(NULL);

    return 0;
}
```

Exercise 4.1

a. 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.

b. Sender.c

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define FIFO_PATH "/tmp/fifo"

void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

int main() {
    int fd;
    char input[1024];
    char output[1024];

    // Create the named pipe (FIFO) if it doesn't exist
    if (mkfifo(FIFO_PATH, 0666) == -1) {
        perror("mkfifo");
    }

    // Get input from the user
    printf("Enter a string: ");
    if (fgets(input, sizeof(input), stdin) == NULL) {
        die("fgets");
    }
```

```c
    // Remove newline character if present
    input[strcspn(input, "\n")] = '\0';

    // Open the FIFO for writing
    fd = open(FIFO_PATH, O_WRONLY);
    if (fd == -1) {
        die("open");
    }

    // Write the input string to the FIFO
    if (write(fd, input, sizeof(input)) == -1) {
        die("write");
    }

    // Close the FIFO
    close(fd);

    // Open the FIFO for reading the capitalized string
    fd = open(FIFO_PATH, O_RDONLY);
    if (fd == -1) {
        die("open");
    }

    // Read the capitalized string from the FIFO
    if (read(fd, output, sizeof(output)) == -1) {
        die("read");
    }
```

```c
    // Print the capitalized string
    printf("Capitalized string: %s\n", output);

    // Close the FIFO
    close(fd);

    // Remove the named pipe (FIFO)
    unlink(FIFO_PATH);

    return 0;
}
```

Receiver.c

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define FIFO_PATH "/tmp/fifo"

void die(const char *msg) {
    perror(msg);
    exit(EXIT_FAILURE);
}

int main() {
    int fd;
    char input[1024];
    char output[1024];
    FILE *fp;

    // Create the named pipe (FIFO) if it doesn't exist
    if (mkfifo(FIFO_PATH, 0666) == -1) {
        perror("mkfifo");
    }

    // Open the FIFO for reading
    fd = open(FIFO_PATH, O_RDONLY);
    if (fd == -1) {
        die("open");
    }
```

```c
    // Read the input string from the FIFO
    if (read(fd, input, sizeof(input)) == -1) {
        die("read");
    }

    // Close the read end of the FIFO
    close(fd);

    // Use tr command to convert the string to uppercase
    snprintf(output, sizeof(output), "echo \"%s\" | tr '[:lower:]' '[:upper:]'", input);

    // Open a pipe to the tr command
    fp = popen(output, "r");
    if (fp == NULL) {
        die("popen");
    }

    // Read the output from the tr command
    if (fgets(output, sizeof(output), fp) == NULL) {
        die("fgets");
    }

    // Remove newline character if present
    output[strcspn(output, "\n")] = '\0';

    // Close the pipe
    pclose(fp);
```

```c
    // Open the FIFO for writing the capitalized string
    fd = open(FIFO_PATH, O_WRONLY);
    if (fd == -1) {
        die("open");
    }

    // Write the capitalized string to the FIFO
    if (write(fd, output, sizeof(output)) == -1) {
        die("write");
    }

    // Close the write end of the FIFO
    close(fd);

    return 0;
}
```