Operating Systems Lab Report

Muhammad Shafeen
Student ID: 22P-9278

October 11, 2024

Lab 8: IPC with Pipes

In this lab, we explore inter-process communication (IPC) using pipes, focusing on creating pipes in C and using them to communicate between parent and child processes.

0.1 Task 1: View the Contents of the Pipe Array

The first task involves creating a pipe and viewing the contents of the pipe's file descriptors.

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

int main() {
    int pfd[2]; // file descriptors array
    pipe(pfd);  // create pipe

    printf("pfd[0] = %d, pfd[1] = %d\n", pfd[0], pfd[1]);
    // Print pipe file descriptors
}
```

Solution:

The file descriptors for the pipe are printed using the printf statement. The two numbers correspond to the read and write ends of the pipe. After running this code, you will see two file descriptor values, typically '3' and '4', which represent the pipe's read and write ends respectively.

0.2 Task 2: Reading from the Pipe and Viewing `aString` Contents

This task involves creating a child process that writes to a pipe and a parent process that reads from it. The contents of `aString` before and after the read operation are examined.

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

```
int main() {
    int pid;           // Process ID for fork
    int pfd[2];        // Pipe file descriptors
    char aString[20]; // Buffer for the parent to store data from the pipe

    pipe(pfd);         // Create a pipe

    pid = fork();      // Fork a child process

    if (pid == 0) {    // Child process
        close(pfd[0]); // Close the read end of the pipe
        write(pfd[1], "Hello", 5); // Write "Hello" to the pipe
        close(pfd[1]); // Close the write end after writing
    } else {           // Parent process
        close(pfd[1]); // Close the write end of the pipe

        printf("Before read: %s\n", aString); // Print aString before reading
        read(pfd[0], aString, 5); // Read from the pipe into aString
        aString[5] = '\0'; // Null-terminate the string
        printf("After read: %s\n", aString); // Print aString after reading

        close(pfd[0]); // Close the read end after reading
    }
}
```

Solution:

In the parent process, `aString` is initially empty before the `read()` call. After the `read()` call, it contains "Hello", which was written to the pipe by the child process.

0.3 Task 3: Closing Unnecessary Ends of the Pipe

The goal of this task is to close unnecessary ends of the pipe, ensuring the child only writes to the pipe and the parent only reads from it.

```
if (pid == 0) {    // Child process
    close(pfd[0]); // Close the read end of the pipe
    write(pfd[1], "Hello", 5); // Write "Hello" to the pipe
    close(pfd[1]); // Close the write end after writing
} else {           // Parent process
    close(pfd[1]); // Close the write end of the pipe
    read(pfd[0], aString, 5); // Read from the pipe into aString
    close(pfd[0]); // Close the read end after reading
}
```

Solution:

Closing unnecessary pipe ends ensures proper communication and prevents resource leakage. The child process only needs to write, so the read end is closed. Similarly, the parent process only needs to read, so the write end is closed.

0.4 Task 4: `ls | wc` Command using Pipes

This task simulates the shell command `ls | wc`, where the output of the `ls` command is passed to the `wc` command through a pipe.

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

int main() {
    int pfd[2]; // Pipe file descriptors

    pipe(pfd);  // Create a pipe

    if (fork() == 0) {    // Child process
        close(pfd[1]);    // Close the write end
        dup2(pfd[0], 0);  // Redirect stdin to the read end of the pipe
        close(pfd[0]);    // Close the read end after duplication
        execlp("wc", "wc", (char *)0);  // Replace process with 'wc'
    } else {              // Parent process
        close(pfd[0]);    // Close the read end
        dup2(pfd[1], 1);  // Redirect stdout to the write end of the pipe
        close(pfd[1]);    // Close the write end after duplication
        execlp("ls", "ls", (char *)0);  // Replace process with 'ls'
    }

    return 0;
}
```

Solution:

The parent process runs the `ls` command, which sends its output to the pipe. The child process reads from the pipe and runs the `wc` command to count the number of lines, words, and bytes.

Conclusion

In this lab, we explored how to use pipes for inter-process communication in C. We learned how to set up pipes, communicate between parent and child processes, close unnecessary ends, and simulate shell commands like `ls | wc`. By using pipes, processes can share data efficiently while maintaining separate memory spaces.