

**Adigrat University**

**College Of Engineering and Technology**

**Department Of Software Engineering**

**Group assignment G2**

**Course name: System Programing**

**Course code: SENG 4044**

|  |  |  |
| --- | --- | --- |
| **#** | **Name** | **Id no** |
| 1 | Mikiale g/her | RET10631/11 |
| 2 | Mearg kahsay | RET14487/11 |
| 3 | Mlat Teamr | RET14541/11 |
| 4 | Milion Demeke | RET/11 |
| 5 | Mekonen nguse | RET10343/11 |
|  |  |  |

Submitted to: Ins. Kibrom

Adigrat, Ethiopia

feb 13, 2024

process management and inter-process communication (IPC) using pipes. Process management and interprocess communication (IPC) are fundamental aspects of operating systems, especially in multitasking environments. Here are their definitions:

Process Management:

Process management involves all the activities related to **creating, scheduling, terminating**, and **managing** processes within an operating system. A process can be defined as an instance of a running program. Process management ensures that processes are properly managed, allowing for efficient utilization of system resources and providing an environment where multiple processes can run concurrently without interfering with each other. Key aspects of process management include process creation, process scheduling, process synchronization, and process communication.

Process management involves creating, scheduling, terminating, and managing processes within an operating system. In this program:

1, The main function creates multiple processes using the fork() system call.

2, Each process executes its own code independently from the parent process.

3, The parent process waits for child processes to complete using the wait() and waitpid() system calls.

4, Process IDs (PIDs) are used to differentiate between parent and child processes.

Inter-Process Communication (IPC):

Interposes communication refers to the **mechanisms** and **techniques** used by processes to **communicate** and **share** data with each other. In a multitasking operating system, multiple processes may need to cooperate or exchange information to accomplish tasks. IPC provides a way for processes to **send** and **receive** messages, **share memory**, or **synchroniz**e their activities. Common IPC mechanisms include message passing, shared memory, semaphores, pipes, and sockets. These mechanisms enable processes to coordinate their activities and exchange data efficiently, facilitating collaboration and interactivity in multi-process systems.

Inter-process communication enables processes to exchange data and coordinate their activities. In this program:

1, Pipes are used for IPC between the parent and child processes.

2, The pipe() system call creates a unidirectional communication channel.

3, The write() and read() system calls are used to send and receive data through the pipe.

4, Communication between processes allows them to synchronize their activities and share information.

Functions Used in the Program:

Here's a brief description of each function used in the provided C program: **1, printBinary(char c):**

Description: This function prints the binary representation of a character. Parameters: c - The character whose binary representation is to be printed.

Functionality: It uses bitwise operations to extract each bit of the character c and prints them one by one.

**2, main():**

Description: The main function of the program.

Functionality: It declares variables for pipe file descriptors and a buffer for message passing.

Creates a pipe using the pipe() system call.

Forks two child processes using the fork() system call.

Handles parent and child processes accordingly, facilitating communication via pipes.

Manages process synchronization using waitpid() and wait() system calls.

**3, printf(), fprintf(), fgets(), strlen():**

**Description:** Standard C library functions for formatted printing, printing to the standard error, reading a string from the standard input, and getting the length of a string, respectively.

Functionality: Used for input/output operations and string manipulation within the program.

**4, perror():**

**Description**: Prints a descriptive error message to the standard error output.

**Functionality**: Used to print error messages for failed system calls, with additional information about the error.

**5,exit():**

**Description**: Terminates the program execution immediately.

**Functionality**: Used to exit the program, indicating whether the execution was successful or failed.

**6, pipe(), fork(), close(), write(), read():**

**Description**: System calls for creating pipes, forking processes, closing file descriptors, writing to file descriptors, and reading from file descriptors, respectively.

**Functionality**: Used for inter-process communication through pipes.

**7, waitpid(), wait():**

**Description:** System calls for waiting for child processes to change state.

**Functionality**: Used to synchronize the execution of the parent process with the child processes.

Output:

**=> The program prompts the user to input a message.**

**=> The message is sent from the first child process to the second child process through the pipe.**

**=> The second child process receives the message, tokenizes it, and prints hexadecimal and binary representations of each token.**

**=> The parent process waits for the second child process to complete and then exits.**

the out put is

**\*\*\* Parent Process \*\*\***

**First child process created with PID: 15768**

**Enter name, age, and sex separated by spaces: Second child process created with PID: 15769**

**m 1 f**

**Second child received message: m 1 f**

**Hexadecimal representation of m: 6D**

**Binary representation of m: 01101101**

**Hexadecimal representation of 1: 31**

**Binary representation of 1: 00110001**

**Hexadecimal representation of f**

**: 66 0A**

**Binary representation of f**

**: 01100110 00001010**

**Parent process received response from second child.**

**Wait failed: No child processes**

Conclusion:

The program demonstrates the use of process management and inter-process communication mechanisms in a multitasking environment. It illustrates how processes can communicate and coordinate their activities using pipes in a Unix-like operating system. Understanding these concepts is crucial for developing efficient and scalable software applications that leverage multiprocessing and IPC capabilities.

**the code is**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <string.h>

#define BUFFER\_SIZE 256

// Function to print binary representation of a character

void printBinary(char c) {

for (int i = 7; i >= 0; --i) {

printf("%d", (c >> i) & 1);

}

}

int main() {

// File descriptors for the pipe

int fd[2];

// Buffer for message passing

char buffer[BUFFER\_SIZE];

// Print a header indicating the parent process

printf("=== Parent Process ===\n");

// Create pipe

if (pipe(fd) == -1) {

perror("Pipe creation failed"); // Print error message if pipe creation fails

exit(EXIT\_FAILURE); // Exit the program with failure status

}

// Fork first child

pid\_t first\_child\_pid = fork();

if (first\_child\_pid == -1) {

perror("First fork failed"); // Print error message if fork fails

exit(EXIT\_FAILURE); // Exit the program with failure status

} else if (first\_child\_pid == 0) {

// Inside first child process

close(fd[0]); // Close unused read end of the pipe

// Prompt user to input name, age, and sex

printf("Enter name, age, and sex separated by spaces: ");

fgets(buffer, BUFFER\_SIZE, stdin); // Read user input

// Send message to second child through the pipe

if (write(fd[1], buffer, strlen(buffer) + 1) == -1) {

perror("Write to pipe failed"); // Print error message if write to pipe fails

exit(EXIT\_FAILURE); // Exit the program with failure status

}

// Close write end of the pipe

close(fd[1]);

exit(EXIT\_SUCCESS); // Exit the child process with success status

} else {

// Inside parent process

printf("First child process created with PID: %d\n", first\_child\_pid);

// Fork second child

pid\_t second\_child\_pid = fork();

if (second\_child\_pid == -1) {

perror("Second fork failed"); // Print error message if fork fails

exit(EXIT\_FAILURE); // Exit the program with failure status

} else if (second\_child\_pid == 0) {

// Inside second child process

close(fd[1]); // Close unused write end of the pipe

// Read message from first child through the pipe

if (read(fd[0], buffer, BUFFER\_SIZE) == -1) {

perror("Read from pipe failed"); // Print error message if read from pipe fails

exit(EXIT\_FAILURE); // Exit the program with failure status

}

printf("Second child received message: %s\n", buffer); // Print the received message

// Tokenize input string

char \*token;

token = strtok(buffer, " ");

while (token != NULL) {

// Process each token

// Convert token to hexadecimal representation

printf("Hexadecimal representation of %s: ", token);

for (int i = 0; token[i] != '\0'; ++i) {

printf("%02X ", token[i]);

}

printf("\n");

// Convert token to binary representation

printf("Binary representation of %s: ", token);

for (int i = 0; token[i] != '\0'; ++i) {

printBinary(token[i]);

printf(" ");

}

printf("\n");

// Get the next token

token = strtok(NULL, " ");

}

// Close read end of the pipe

close(fd[0]);

exit(EXIT\_SUCCESS); // Exit the child process with success status

} else {

// Inside parent process

printf("Second child process created with PID: %d\n", second\_child\_pid);

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

close(fd[1]); // Close write end of the pipe

// Wait for the second child to complete

if (waitpid(second\_child\_pid, NULL, 0) == -1) {

perror("Waitpid failed"); // Print error message if waitpid fails

exit(EXIT\_FAILURE); // Exit the program with failure status

}

printf("Parent process received response from second child.\n");

// Wait for both children to complete

if (wait(NULL) == -1 || wait(NULL) == -1) {

perror("Wait failed"); // Print error message if wait fails

exit(EXIT\_FAILURE); // Exit the program with failure status

}

printf("Parent process exiting.\n");

exit(EXIT\_SUCCESS); // Exit the parent process with success status

}

}

return 0; // This line should never be reached, as the program exits using exit()

}