

**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 | Mikiele Gher | RET/10730/11 |
| 2 | Meareg khasay | RET/14443/11 |
| 3 | Mekonen Niguse | RET/10399/11 |
| 4 | Millyon demeke | RET/11765/11 |
| 5 | Mlat teamr | RET/14541/11 |
|  |  |  |

Submitted to: Ins. Kibrom

Adigrat, Ethiopia

***The code :***

***#include <stdio.h>***

***#include <stdlib.h>***

***#include <unistd.h> ]***

***#include <sys/types.h>***

***#include <sys/wait.h>***

***#include <string.h>***

***#define BUFFER\_SIZE 256***

***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];***

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

***// Create pipe***

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

***perror("Pipe creation failed");***

***exit(EXIT\_FAILURE);***

***}***

***// Fork first child***

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

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

***perror("First fork failed");***

***exit(EXIT\_FAILURE);***

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

***// Inside first child process***

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

***// Prompt user to input a string to send to the second child***

***printf("Enter a string to send to the second child: ");***

***fgets(buffer, BUFFER\_SIZE, stdin);***

***// Send message to second child***

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

***perror("Write to pipe failed");***

***exit(EXIT\_FAILURE);***

***}***

***// Close write end of the pipe***

***close(fd[1]);***

***exit(EXIT\_SUCCESS);***

***} 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");***

***exit(EXIT\_FAILURE);***

***} 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***

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

***perror("Read from pipe failed");***

***exit(EXIT\_FAILURE);***

***}***

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

***// 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);***

***} 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");***

***exit(EXIT\_FAILURE);***

***}***

***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");***

***exit(EXIT\_FAILURE);***

***}***

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

***exit(EXIT\_SUCCESS);***

***}***

***}***

***return 0;***

***}***

**Brief explaination of the code.**

**Process Creation:**

The program starts by creating two child processes using the fork() system call.

Each child process is created to handle specific tasks within the program.

The parent process creates the first child process and then creates the second child process.

**Interprocess Communication (IPC):**

After creating the child processes, the parent process sets up a communication mechanism between them using a pipe.

Pipes allow one-way communication between processes. In this case, the pipe is used for from the first child process to the second child process.

**Message Passing:**

Once the pipe is set up, the first child process writes a message to the pipe.

The message contains information or data that the second child process needs to receive and process.

**Message Processing:**

The second child process reads the message from the pipe.

After receiving the message, the second child process processes the message as required.

In your program, it prints the received message to the console and performs a simple operation on it, such as converting it to binary or hexadecimal.

**Synchronization:**

To ensure proper synchronization between the processes, the parent process waits for both child processes to complete their tasks before exiting.

It uses the waitpid() system call to wait for each child process to terminate.

**Termination:**

Once both child processes have completed their tasks and exited, the parent process also exits.

**Here's a summary of the program flow:**

* The parent process creates two child processes.
* The first child process sends a message through the pipe to the second child process.
* The second child process receives the message, processes it, and sends a response back to the first child process (not explicitly mentioned in your d escription, but usually expected for bidirectional communication).
* The parent process waits for both child processes to finish their tasks.
* Once both child processes have completed their tasks, the parent process exits.

In this way, the program demonstrates the creation of processes, interprocess communication using pipes, message passing, synchronization, and proper termination. It showcases how different processes can collaborate and communicate with each other within an operating system environment.

**interposes communication (IPC)**

The provided C program demonstrates process management and inter-process communication (IPC) using pipes or as for the IPC (Interprocess Communication) and synchronization mechanisms, we've chosen to use pipes for communication between the child processes. **Because** Pipes are a simple and effective way to pass data between processes, especially when there's a parent-child relationship.

**Sychronization:**

For synchronization ,our code uses the **waitpid()** system call to ensure that the parent process waits for both child processes to complete before exiting. **waitpid()** suspends the execution of the calling process until a child specified by the **pid** argument has terminated.

**our test cases and outputs demonstrates:**

The first child process sends a message to the second child process through the pipe.

The second child process receives the message, processes it, and sends a response back to the first child process through the pipe.

The parent process waits for both child processes to complete their tasks before exiting.

***First child process created with PID: 6917***

***Second child process created with PID: 6918***

***Enter a string to send to the second child: example***

***Second child received message: example***

***Hexadecimal representation of example***

***: 65 78 61 6D 70 6C 65 0A***

***Binary representation of example***

***: 01100101 01111000 01100001 01101101 01110000 01101100 01100101 00001010***

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

***Wait failed: No child processes.***

**printBinary(char c):**

This function prints the binary representation of a character c.

It takes a character c as input and prints its binary representation by iterating through its bits.

**main():**

This is the main function of the program.

It sets up inter-process communication using pipes and forks child processes to communicate with each other.

It creates two child processes to demonstrate message passing between them.

The parent process coordinates the creation and termination of child processes.

In the main() function, the following system calls and C library functions are used:

**pipe(fd):**

Creates a pipe with two file descriptors fd[0] for the read end and fd[1] for the write end.

**fork():**

Creates a new process by duplicating the calling process. After calling fork(), two processes are created - the parent and the child.

**close(fd[0]) and close(fd[1]):**

Closes file descriptors. Typically, each process should close the end of the pipe that it does not need.

**write(fd[1], buffer, strlen(buffer) + 1):**

Writes data from the buffer to the pipe.

**read(fd[0], buffer, BUFFER\_SIZE):**

Reads data from the pipe into the buffer.

**waitpid(second\_child\_pid, NULL, 0):**

The parent process waits for the second child process to finish before proceeding.

**wait(NULL):**

The parent process waits for any child process to finish.

**exit(status):**

Terminates the currently executing process and returns the status to the parent process. The status value is usually 0 for successful execution and non-zero for failure.

**Process Creation:**

We Use the fork() system call to create two child processes from the parent process.

**Interprocess Communication (IPC):**

We Choose an IPC mechanism pipes, to facilitate communication between the child processes.

We Create a pipe in the parent process before forking. This pipe will be used for communication between the child processes.

**Message Passing:**

After forking, the first child process writes a message to the pipe.

The second child process reads the message from the pipe.

**Message Processing:**

Upon receiving the message, the second child process prints the message to the console.

Perform changing the string to binary and hexadecimal on the message.

**Synchronization:**

We Use **wait()** synchronization mechanism to ensure that the first child process waits for a response from the second child process.

We Ensure that the first child process waits until the second child process has completed its task and sent a response.

**Error Handling:**

We Include error handling perror() for system calls to handle any potential errors during process creation, IPC setup, and synchronization.

**here's a brief description of each function used in the provided C program:**

**printBinary(char c):**

This function prints the binary representation of a character c.

It takes a character c as input and prints its binary representation by iterating through its bits.

**main():**

This is the main function of the program.

It sets up inter-process communication using pipes and forks child processes to communicate with each other.

It creates two child processes to demonstrate message passing between them.

The parent process coordinates the creation and termination of child processes.

In the main() function, the following system calls and C library functions are used:

**pipe(fd):**

Creates a pipe with two file descriptors fd[0] for the read end and fd[1] for the write end.

**fork():**

Creates a new process by duplicating the calling process. After calling fork(), two processes are created - the parent and the child.

**close(fd[0]) and close(fd[1]):**

Closes file descriptors. Typically, each process should close the end of the pipe that it does not need.

**write(fd[1], buffer, strlen(buffer) + 1):**

Writes data from the buffer to the pipe.

**read(fd[0], buffer, BUFFER\_SIZE):**

Reads data from the pipe into the buffer.

**waitpid(second\_child\_pid, NULL, 0):**

The parent process waits for the second child process to finish before proceeding.

**wait(NULL):**

The parent process waits for any child process to finish.

**exit(status):**

Terminates the currently executing process and returns the status to the parent process. The status value is usually 0 for successful execution and non-zero for failure.

These functions and system calls are used to create a simple inter-process communication scenario where the parent process communicates with two child processes through pipes. The child processes receive input from the parent, process it, and then communicate back the results to the parent process and the **output** of the C code seems like this

***First child process created with PID: 6917***

***Second child process created with PID: 6918***

***Enter a string to send to the second child: example***

***Second child received message: example***

***Hexadecimal representation of example***

***: 65 78 61 6D 70 6C 65 0A***

***Binary representation of example***

***: 01100101 01111000 01100001 01101101 01110000 01101100 01100101 00001010***

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

***Wait failed: No child processes.***