

**ADIGRAT UNIVERSITY**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**DEPARTMENT OF SOFTWARE ENGINEERING**

**GROUP ASSIGIMENT ON: SYSTEM PROGRAMMING**

**COURSE CODE:**

**MEMBERS OF THE GROUP**

**NAME ID-NUMBER**

**HAFTAY DESTA RET/10473/11**

**GUESH GOITOM RET/11771/11**

**HIRUT TESFAY RET/14555/11**

**LIDAY ERMIASE RET/14444/11**

**HAYAT TUEM RET/06456/10**

**G/TSADIK BRHANU RET/10585/11**

**Submitted to: Advisors: Ins. KIBROM R. (MSc)**

**Submission date: 18, January, 2016E.C**

Contents

[1. Introduction: 3](#_Toc159792316)

[2. Process Management 3](#_Toc159792317)

[3. IPC Mechanism: 3](#_Toc159792318)

[4. **Synchronization Mechanism:** 4](#_Toc159792319)

[5. Message Prossesing 4](#_Toc159792320)

[6. Out Put Of The C Programme 5](#_Toc159792321)

[**7. C Program (The Source Code)** 6](#_Toc159792322)

**A SHORT DESCRIPTION ABOUT OUR PRJECT**

# 1. Introduction:

This C program demonstrates a multi-process system for inter-process communication (IPC) using pipes.

It performs a simple addition of two numbers provided by the user.

It demonstrates inter-process communication (IPC) using pipes in C.

It involves three processes:

**Parent process:** Manages the overall execution and retrieves the combined result.

**First child process:** Collects two numbers from the user and sends them to the second child.

**Second child process:** Receives the numbers, performs their addition, converts the result to a binary string, and sends it back to the parent process.

# 2. Message Prossesing

**Main Process:**

Creates two pipes for communication.

Creates two child processes using fork().

**First Child Process:**

Gets two numbers from the user.

Sends the numbers to the second child process through the first pipe.

**Second Child Process:**

Receives the numbers from the first child process.

Adds the numbers.

Sends the result back to the parent process through the second pipe.

**Parent Process:**

Receives the result from the second child process.

Prints the result to the console.

Waits for both child processes to finish.

Terminate the process.

# 3. Process Management

**Process management** in an operating system involves the creation, scheduling, and termination of processes, which are instances of executing programs. It includes various tasks such as allocating system resources, managing process states, coordinating process execution, and ensuring proper synchronization and communication between processes.

The code utilizes fork() system call to create two child processes from the parent process. This allows for parallel execution of tasks.

# 4. IPC Mechanism:

Inter process communication (IPC) refers to the mechanisms and techniques used by processes to communicate and share data with each other. IPC allows processes to exchange information, coordinate their activities, and synchronize their operations. Common methods of IPC include shared memory, message passing, pipes, sockets, and signals.

The code uses **pipes** for inter-process communication (IPC).

**Pipes** are unidirectional communication channels that allow processes to exchange data.

**Reason:** Pipes are a simple and efficient mechanism for communication between related processes within the same system. They are suitable for transferring small amounts of data between processes without complex setup.

The code creates two pipes:

pipe1: Used for the first child to send numbers to the second child.

pipe2: Used for the second child to send the result back to the first child (and then to the parent).

# 5. **Synchronization Mechanism:**

The code relies on **implicit synchronization** based on the order of pipe reads and writes.

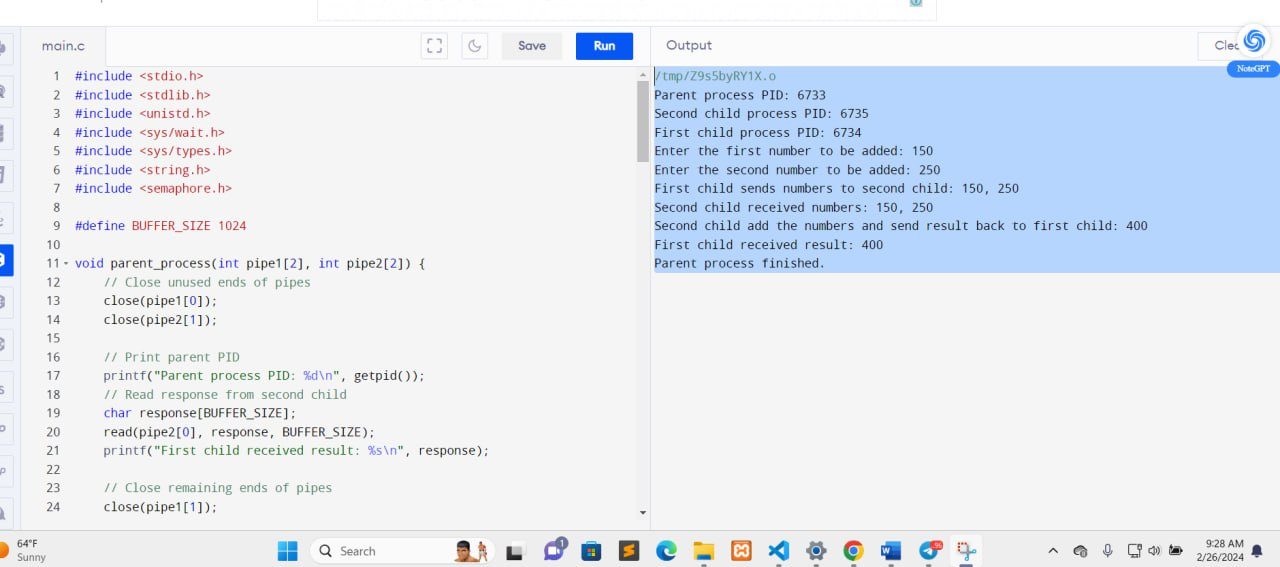
The first child process blocks on write(pipe1[1], numbers, sizeof(numbers)) until the second child process reads from pipe1[0].

Similarly, the second child process blocks on write(pipe2[1], binary\_result, strlen(binary\_result)+1) until the parent process reads from pipe2[0].

This ensures that the first child waits for the second child's response before both terminate.

**Buffer Size:** Manages data transfer between processes using a buffer of 1024 bytes.

# 6. Out Put Of The C Programme



# **7. C Program (The Source Code)**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <unistd.h>**

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

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

**#include <string.h>**

**#include <semaphore.h>**

**#include <ctype.h> // Added header for 'isdigit' and 'isspace'**

**#define BUFFER\_SIZE 1024**

**void parent\_process(int pipe1[2], int pipe2[2]) {**

**// Close unused ends of pipes**

**close(pipe1[0]);**

**close(pipe2[1]);**

**// Print parent PID**

**printf("Parent process PID: %d\n", getpid());**

**// Read response from second child**

**char response[BUFFER\_SIZE];**

**read(pipe2[0], response, BUFFER\_SIZE);**

**printf("parent process received result: %s\n", response);**

**// Close remaining ends of pipes**

**close(pipe1[1]);**

**close(pipe2[0]);**

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

**}**

**void first\_child\_process(int pipe1[2]) {**

**// Close unused end of pipe**

**close(pipe1[0]);**

**// Print first child PID**

**printf("First child process PID: %d\n", getpid());**

**// Get numbers from user with error handling**

**int numbers[2];**

**for (int i = 0; i< 2; i++) {**

**int valid = 0;**

**while (!valid) {**

**printf("Enter number %d to be added: ", i + 1);**

**if (scanf("%d", &numbers[i]) == 1) {**

**valid = 1; // Valid number entered, break loop**

**} else {**

**// Consume remaining invalid input**

**int c;**

**while ((c = getchar()) != '\n' && c != EOF);**

**printf("Invalid input. Please enter only numbers.\n");**

**}**

**}**

**}**

**// Send valid numbers to second child**

**write(pipe1[1], numbers, sizeof(numbers));**

**printf("First child sends numbers to second child: %d, %d\n", numbers[0], numbers[1]);**

**// Close remaining end of pipe**

**close(pipe1[1]);**

**exit(0);**

**}**

**void second\_child\_process(int pipe1[2], int pipe2[2]) {**

**// Close unused ends of pipes**

**close(pipe1[1]);**

**close(pipe2[0]);**

**// Print second child PID**

**printf("Second child process PID: %d\n", getpid());**

**// Receive numbers from first child**

**int numbers[2];**

**read(pipe1[0], numbers, sizeof(numbers));**

**printf("Second child received numbers: %d, %d\n", numbers[0], numbers[1]);**

**// Process numbers (addition)**

**int result = numbers[0] + numbers[1];**

**// Convert result to binary string**

**char binary\_result[BUFFER\_SIZE];**

**sprintf(binary\_result, "%d", result);**

**// Send the summation binary result back to Parent child**

**write(pipe2[1], binary\_result, strlen(binary\_result)+1);**

**printf("Second child add the numbers and send result back to parent child: %d \n", result);**

**// Close remaining ends of pipes**

**close(pipe1[0]);**

**close(pipe2[1]);**

**exit(0);**

**}**

**int main() {**

**int pipe1[2], pipe2[2];**

**pid\_t pid1, pid2;**

**// Create first pipe**

**if (pipe(pipe1) == -1) {**

**perror("pipe1");**

**exit(1);**

**}**

**// Create second pipe**

**if (pipe(pipe2) == -1) {**

**perror("pipe2");**

**exit(1);**

**}**

**// Create first child process**

**pid1 = fork();**

**if (pid1 == -1) {**

**perror("fork1");**

**exit(1);**

**} else if (pid1 == 0) {**

**// First child process**

**first\_child\_process(pipe1);**

**}**

**// Create second child process**

**pid2 = fork();**

**if (pid2 == -1) {**

**perror("fork2");**

**exit(1);**

**} else if (pid2 == 0) {**

**// Second child process**

**second\_child\_process(pipe1, pipe2);**

**}**

**// Parent process**

**parent\_process(pipe1, pipe2);**

**// Wait for both child processes to finish**

**waitpid(pid1, NULL, 0);**

**waitpid(pid2, NULL, 0);**

**return 0;**

**}**