Student name : Eman Asfour

Student number :1200206

Section :4

# Part 1: Process Management

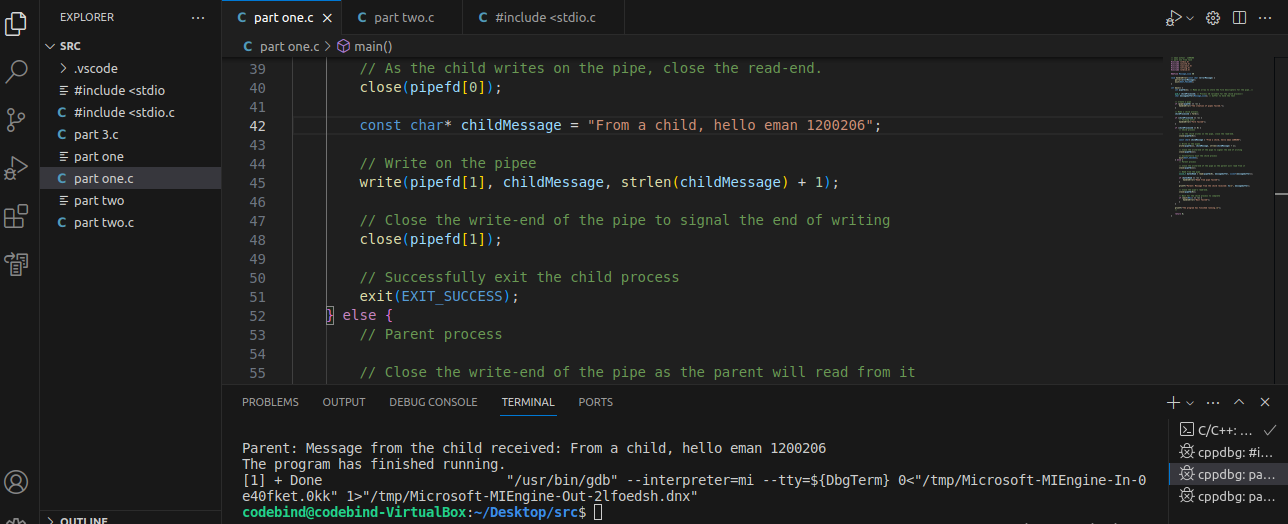


Figure 1: the output for part 1

This C program demonstrates a simple interaction between a parent and child process. The parent creates a child using a `fork ()`, and they communicate through a pipe. The child sends a message to the parent, which reads and displays it. The code includes error handling for potential issues and ensures proper cleanup. It illustrates the fundamental concepts of process creation and communication straightforwardly using pipes.

# **Part 2: Multithreaded Processing**

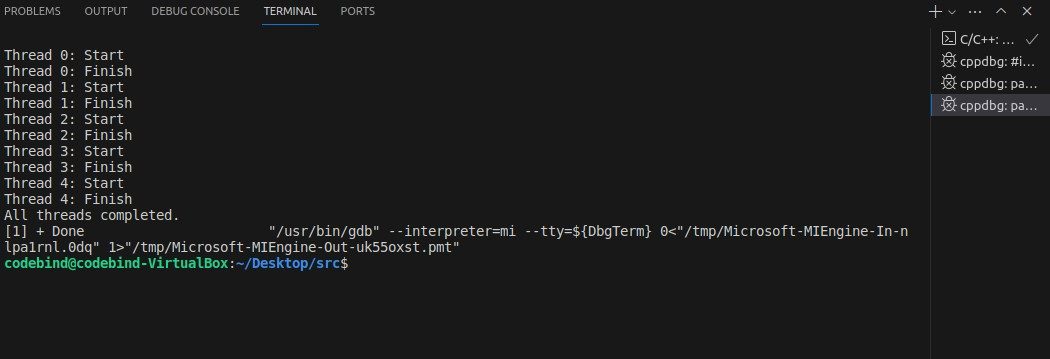


Figure 2: the output part 2

The output you provided shows that each thread is starting and finishing in sequence, which is expected behavior for the simple threaded program you've created. The program is creating seven threads, each performing some work within the “threadFunc” function.

The sequence of "Start" and "Finish" for each thread indicates that the threads are being created, executed, and completed in the order you've launched them. This order is not guaranteed, and it can vary based on the operating system's thread scheduler.

The last line [1] + Done is not related to your program's output. It appears to be a message from the shell or terminal where you ran the program. It typically indicates that a background job (process ID 1 in this case) has been completed.

In summary, the output is normal for a program that creates and runs multiple threads, and the [1] + Done message is unrelated to your program's output.

# **Part 3: Performance Measurement**

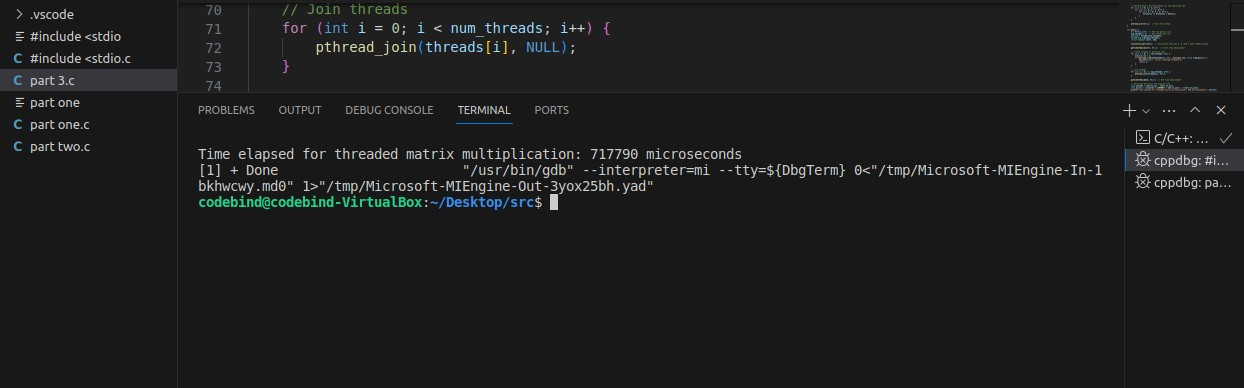


Figure 3: The output of part 3

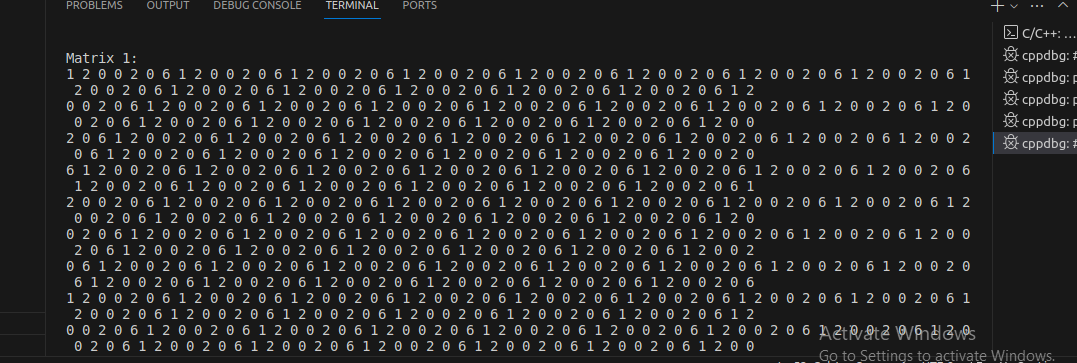


Figure 4: the matrix A

# Part 4: Thread Management

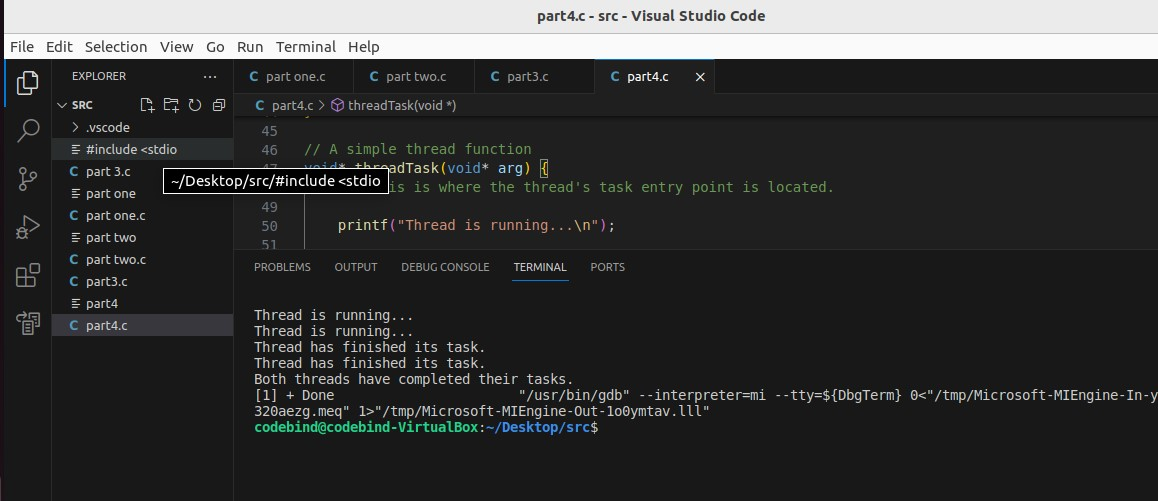


Figure 5: the output for part 4

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#include <pthread.h>

// Thread function prototype

void\* threadTask(void\* arg);

int main() {

// B- Program with Multiple Child Processes

for (int numProcesses = 2; numProcesses <= 4; ++numProcesses) {

for (int i = 0; i < numProcesses; ++i) {

pid\_t childProcessId = fork();

if (childProcessId == -1) {

perror("Fork failed");

return 1;

}

if (childProcessId == 0) {

// Child process

printf("Child process %d (PID: %d) is running...\n", i + 1, getpid());

// Perform some task in each child process

\_exit(0); // Exit the child process

}

}

// Wait for all child processes to complete

for (int i = 0; i < numProcesses; ++i) {

wait(NULL);

}

printf("All child processes have completed for %d processes.\n\n", numProcesses);

}

// C- Program with Multiple Joinable Threads

pthread\_t threads[2];

// Create a joinable thread

if (pthread\_create(&threads[0], NULL, threadTask, NULL)) {

perror("Failed to create joinable thread");

return 1;

}

// Create a detached thread

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

if (pthread\_create(&threads[1], &attr, threadTask, NULL)) {

perror("Failed to create detached thread");

pthread\_attr\_destroy(&attr);

return 1;

}

// Destroy the thread attribute object, as it is no longer needed

pthread\_attr\_destroy(&attr);

// Join the joinable thread

if (pthread\_join(threads[0], NULL)) {

perror("Failed to join thread");

return 1;

}

// Give the detached thread some time to complete

sleep(2);

printf("Both threads have completed their tasks.\n\n");

// D- Program with Multiple Detached Threads

pthread\_t detachedThreads[2];

// Create detached threads

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

int result = pthread\_create(&detachedThreads[i], (i == 0) ? NULL : &attr, threadTask, NULL);

if (result != 0) {

perror("Failed to create thread");

return 1;

}

}

// Destroy the thread attribute object

pthread\_attr\_destroy(&attr);

// Give detached threads some time to complete

sleep(2);

printf("Both detached threads have completed their tasks.\n");

return 0;

}

// A simple thread function

void\* threadTask(void\* arg) {

printf("Thread is running...\n");

sleep(1); // Simulate a task

printf("Thread has finished its task.\n");

return NULL;

}

Edit the code as messge in ritaj

In the initial stages of my learning journey, I encountered challenges such as configuring my BIOS for virtualization to install Ubuntu. As I delved into Ubuntu, I gained insights into its functionality and the process of downloading and installing programs. In our recent task, establishing communication between processes seemed daunting initially, but I successfully tackled it by implementing pipes and managing parent and child processes in my code. Additionally, while working on multithreading with Pthreads, I confronted matrix multiplication issues due to its large size. I addressed this problem by optimizing the code and resolving syntax errors. To enhance performance, I explored time measurement techniques using gettimeofday [1] and discovered that joinable threads outperformed detached threads. Through this journey, I refined my programming skills and problem-solving abilities. [2] [3]