**LINUX**

**SYSTEM**

**PROGRAMMING**

**Que-1. Implement a program to copy source file into destination file. If the destination file exists, it should not overwrite. The program optionally takes a flag –f, which implies overwrite, if destination file exists.**

**- Usage of the application is as shown**

**$ copy [ -f] source-file dest-file**

**ALGORITHM:**

1. Include necessary header files:

- Include the `<iostream>` header for input and output operations.

- Include the `<fstream>` header for file input and output.

- Include the `<string>` header for string manipulation.

2. Start the `main` function:

- The `main` function is the entry point of the program and takes command-line arguments.

3. Check the number of command-line arguments:

- If there are fewer than 3 or more than 4 command-line arguments, display an error message and return 1 to indicate an error.

- Usage: `program\_name [-f] source-file dest-file`

4. Check for the '-f' flag:

- Initialize a boolean variable `forceOverwrite` to `false`.

- If there are 4 arguments and the second argument is "-f", set `forceOverwrite` to `true`.

5. Determine source and destination file names:

- Initialize `sourceFileName` and `destFileName` based on whether the `-f` flag was provided.

6. Check if the destination file already exists:

- Open the destination file for reading using an `ifstream`.

- If the file exists and the `-f` flag is not provided, display an error message and return 1.

7. Open the source file for reading:

- Open the source file using an `ifstream` in binary mode.

- If the file cannot be opened, display an error message and return 1.

8. Open the destination file for writing:

- Open the destination file using an `ofstream` in binary mode.

- If the file cannot be opened, display an error message and return 1.

9. Copy the contents of the source file to the destination file:

- Use `sourceFile.rdbuf()` to read the contents of the source file and write them to the destination file using `destFileOut`.

10. Close the files:

- Close both the source and destination files.

11. Display a success message:

- If everything was successful, display a message indicating that the file was copied successfully.

12. Return 0 to indicate successful execution of the program.

**CODE:**

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

int main(int argc, char\* argv[]) {

// Check if the correct number of command-line arguments are provided.

if (argc < 3 || argc > 4) {

cerr << "Usage: " << argv[0] << " [-f] source-file dest-file" << endl;

return 1;

}

// Determine if the '-f' flag is provided.

bool forceOverwrite = false;

if (argc == 4 && string(argv[1]) == "-f") {

forceOverwrite = true;

}

// Determine source and destination file names.

string sourceFileName = forceOverwrite ? argv[2] : argv[1];

string destFileName = forceOverwrite ? argv[3] : argv[2];

// Check if the destination file already exists and the -f flag is not provided.

ifstream destFile(destFileName);

if (destFile.good() && !forceOverwrite) {

cerr << "Destination file already exists. Use -f flag to overwrite." << endl;

return 1;

}

// Open the source file for reading.

ifstream sourceFile(sourceFileName, ios::binary);

if (!sourceFile) {

cerr << "Error opening source file: " << sourceFileName << endl;

return 1;

}

// Open the destination file for writing.

ofstream destFileOut(destFileName, ios::binary);

if (!destFileOut) {

cerr << "Error opening destination file: " << destFileName << endl;

return 1;

}

// Copy the contents of the source file to the destination file.

destFileOut << sourceFile.rdbuf();

// Close the files.

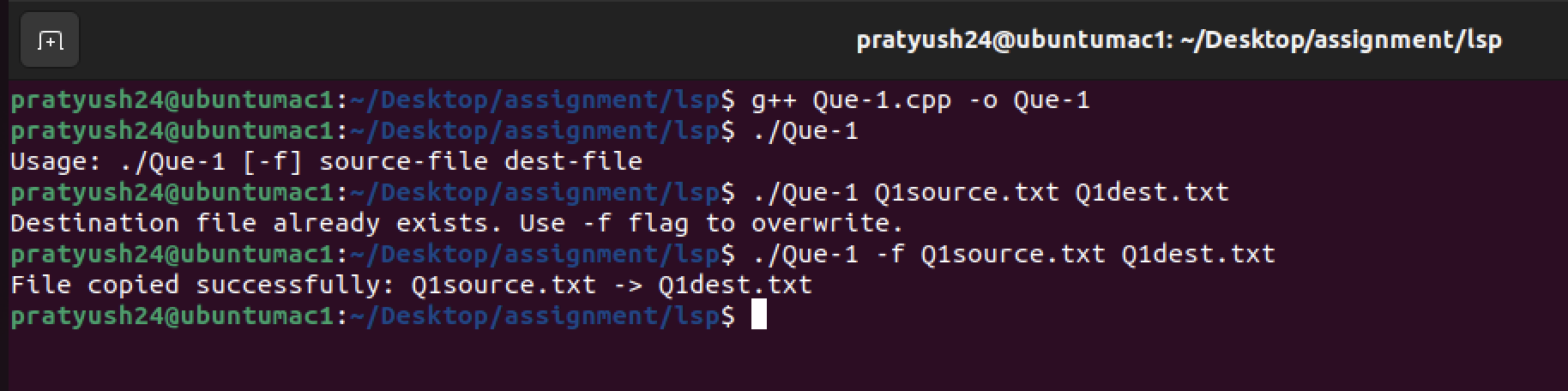
sourceFile.close();

destFileOut.close();

cout << "File copied successfully: " << sourceFileName << " -> " << destFileName << endl;

return 0;

}



**Que-2 Write a program which performs the following: - opens a file for reading - duplicates the file descriptor to descriptor 0. Subsequently the program reads from the standard input and writes to the standard output. What is the observation made?**

**ALGORITHM:**

1. Import the necessary libraries:

- `#include <iostream>`: Import the input-output stream library.

- `#include <fstream>`: Import the file stream library.

- `#include <string>`: Import the string library.

- `#include <cstring>`: Import the C string library.

- `#include <cerrno>`: Import the C error handling library.

- `#include <cstdlib>`: Import the C standard library.

- `#include <unistd.h>`: Import the Unix standard library for POSIX functions.

- `#include <fcntl.h>`: Import the file control library.

2. Define the `main` function:

- `int main(int argc, char\* argv[])`: Define the main function that takes command-line arguments.

3. Check the number of command-line arguments:

- `if (argc != 2)`: Check if the number of command-line arguments is not equal to 2.

4. Display an error message for incorrect usage:

- `cerr << "Usage: " << argv[0] << " filename" << endl;`: Print an error message to the standard error stream (`cerr`) indicating the correct usage of the program.

5. Return with an exit code indicating failure:

- `return EXIT\_FAILURE;`: Exit the program with a failure exit code.

6. Get the filename from the command-line argument:

- `const char\* filename = argv[1];`: Assign the second command-line argument (the filename) to the `filename` variable.

7. Open the file for reading:

- `int fileDescriptor = open(filename, O\_RDONLY);`: Use the `open` function to open the file specified by `filename` in read-only mode and store the file descriptor in `fileDescriptor`.

8. Check for errors while opening the file:

- `if (fileDescriptor == -1)`: Check if the `open` function returned -1, indicating an error in opening the file.

9. Print an error message with `perror`:

- `perror("Error opening file");`: Print an error message, including the system error message related to the last function call, using `perror`.

10. Return with an exit code indicating failure:

- `return EXIT\_FAILURE;`: Exit the program with a failure exit code.

11. Duplicate the file descriptor to descriptor 3:

- `int dupDescriptor = dup2(fileDescriptor, 3);`: Use `dup2` to duplicate the `fileDescriptor` to file descriptor 3 and store the result in `dupDescriptor`.

12. Check for errors while duplicating the file descriptor:

- `if (dupDescriptor == -1)`: Check if the `dup2` function returned -1, indicating an error in duplicating the file descriptor.

13. Print an error message with `perror`:

- `perror("Error duplicating file descriptor");`: Print an error message, including the system error message related to the last function call, using `perror`.

14. Close the original file descriptor:

- `close(fileDescriptor);`: Close the original file descriptor since it has been duplicated and is no longer needed.

15. Read from the duplicated file descriptor (descriptor 3) in a loop:

- `char buffer[1024];`: Declare a character array `buffer` to store data read from the file.

- `ssize\_t bytesRead;`: Declare a variable to store the number of bytes read.

- `while ((bytesRead = read(3, buffer, sizeof(buffer))) > 0)`: Read data from file descriptor 3 into the `buffer` with a maximum read size of `sizeof(buffer)` until the end of the file is reached.

16. Write the read data to standard output (stdout):

- `write(STDOUT\_FILENO, buffer, bytesRead);`: Write the data read from the file into the standard output (stdout).

17. Clean up by closing both file descriptors:

- `close(fileDescriptor);`: Close the original file descriptor.

- `close(dupDescriptor);`: Close the duplicated file descriptor.

18. Return with an exit code indicating success:

- `return EXIT\_SUCCESS;`: Exit the program with a success exit code.

**CODE:**

#include <iostream>

#include <fstream>

#include <string>

#include <cstring>

#include <cerrno>

#include <cstdlib>

#include <unistd.h>

#include <fcntl.h>

using namespace std;

int main(int argc, char\* argv[]) {

// Check if the correct number of command-line arguments are provided.

if (argc != 2) {

cerr << "Usage: " << argv[0] << " filename" << endl;

return EXIT\_FAILURE;

}

const char\* filename = argv[1];

// Open the file for reading

int fileDescriptor = open(filename, O\_RDONLY);

// Check if there was an error opening the file

if (fileDescriptor == -1) {

perror("Error opening file");

return EXIT\_FAILURE;

}

// Duplicate the file descriptor to descriptor 3

int dupDescriptor = dup2(fileDescriptor, 3);

// Check if there was an error duplicating the file descriptor

if (dupDescriptor == -1) {

perror("Error duplicating file descriptor");

close(fileDescriptor);

return EXIT\_FAILURE;

}

// Read from the duplicated file descriptor (descriptor 3)

char buffer[1024];

ssize\_t bytesRead;

while ((bytesRead = read(3, buffer, sizeof(buffer))) > 0) {

// Write to the standard output (stdout)

write(STDOUT\_FILENO, buffer, bytesRead);

}

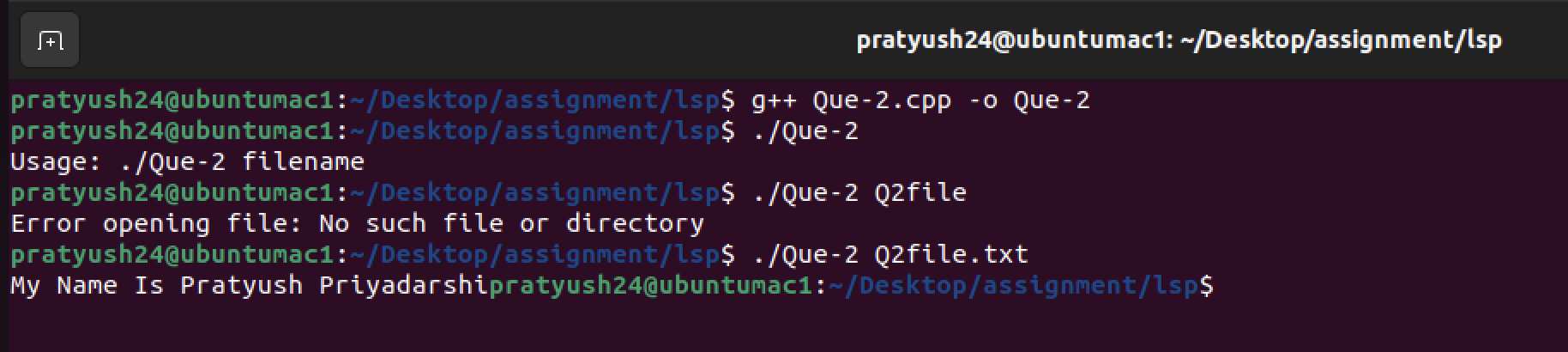
// Clean up

close(fileDescriptor);

close(dupDescriptor);

return EXIT\_SUCCESS;

}



**Que-3 Consider the following code**

**fd1 = open ("abc", O\_RDONLY);**

**fd2 = dup(fd1);**

**read (fd1, buf, 100); | --> R1**

**read (fd2, buf, 10); | --> R2**

**read (fd1, buf, 10); | --> R3**

**read (fd2, buf, 100); | --> R4**

**what is the offset from where read is done for the statements R3 and R4?**

**ALGORITHM:**

1. Initialize two file descriptors `fd1` and `fd2` for the same file "abc".

2. Call `read(fd1, buf, 100);` (R1):

- Read 100 bytes from the file associated with `fd1` starting from its current file offset.

- Advance the file offset for `fd1` by 100 bytes.

3. Call `read(fd2, buf, 10);` (R2):

- Read 10 bytes from the file associated with `fd2` starting from its current file offset.

- Since `fd2` was duplicated from `fd1`, also advance the file offset for `fd1` by 10 bytes to maintain consistency.

4. Call `read(fd1, buf, 10);` (R3):

- Read 10 bytes from the file associated with `fd1` starting from its current file offset.

- Advance the file offset for `fd1` by an additional 10 bytes, making it 120 bytes ahead of its initial position.

5. Call `read(fd2, buf, 100);` (R4):

- Read 100 bytes from the file associated with `fd2` starting from its current file offset.

- Since `fd2` was duplicated from the updated `fd1` (with a file offset advanced by 10 bytes in R3), also advance the file offset for `fd1` by 100 bytes to maintain consistency.

6. After the read operations, the file offsets for both `fd1` and `fd2` have advanced:

- The file offset for `fd1` is now 220 bytes ahead of its initial position.

- The file offset for `fd2` is now 100 bytes ahead of its initial position.

**Que-4 Write an application named mycat which provides functionality of cat**

**$ ./mycat - should read from stdin and write to stdout**

**$ ./mycat abc - should read file abc and write to stdout. Implementation should use filecopy() function as per the description given below**

**int filecopy (int infd, int outfd, int bufsize);**

**Copies contents of file given by descriptor infd into the file represented by descriptor outfd. Internally uses dynamically allocated buffer of size bufsize. Should free the buffer before returning from the function. Should return 0 on successful copy; otherwise returns –1**

**ALGORITHM:**

1. \*\*Include Libraries\*\*:

- Include the necessary C++ libraries such as `<iostream>`, `<fstream>`, `<string>`, `<cstdlib>`, `<cstring>`, `<cerrno>`, and `<unistd.h>` for file operations.

2. \*\*Namespace Declaration\*\*:

- Declare that you will be using the `std` namespace for simplifying access to standard C++ library functions.

3. \*\*File Copy Function\*\* (`int filecopy(int infd, int outfd, int bufsize)`):

- Define a function `filecopy` that takes three arguments:

- `infd`: Input file descriptor.

- `outfd`: Output file descriptor.

- `bufsize`: Size of the buffer used for copying data.

- Allocate memory for a character buffer (`buffer`) of size `bufsize`.

- Check if buffer allocation was successful. If not, print an error message and return -1.

- Initialize a variable `bytesRead` to store the number of bytes read from the input file descriptor.

- While data can be read from `infd`, do the following:

- Read data from `infd` into the `buffer`.

- Write the read data to the `outfd`.

- Check if the write operation was successful. If not, print an error message, free the memory used by `buffer`, and return -1.

- Free the memory used by `buffer`.

- Check if there was an error while reading from `infd`. If so, print an error message and return -1.

- Return 0 to indicate success.

4. \*\*Main Function\*\* (`int main(int argc, char\* argv[])`):

- Check the number of command-line arguments. If not equal to 2, print a usage message and return EXIT\_FAILURE.

- Get the input file name from the command-line argument.

- Open the input file for reading (`infd = open(filename, O\_RDONLY)`).

- Check if the file opening operation was successful. If not, print an error message using `perror` and return EXIT\_FAILURE.

- Call the `filecopy` function to copy data from `infd` to `STDOUT\_FILENO` (standard output) with a buffer size of 1024.

- Check if the `filecopy` function returned an error. If so, print an error message, close the input file descriptor, and return EXIT\_FAILURE.

- Close the input file descriptor.

- Return EXIT\_SUCCESS to indicate successful execution.

**CODE:**

#include <iostream>

#include <fstream>

#include <string>

#include <cstdlib>

#include <cstring>

#include <cerrno>

#include <unistd.h>

#include <fcntl.h>

using namespace std;

// Function to copy contents from one file descriptor to another

int filecopy(int infd, int outfd, int bufsize) {

char\* buffer = new char[bufsize];

if (buffer == nullptr) {

cerr << "Error allocating buffer" << endl;

return -1;

}

ssize\_t bytesRead;

while ((bytesRead = read(infd, buffer, bufsize)) > 0) {

// Write the read data to the output file descriptor

ssize\_t bytesWritten = write(outfd, buffer, bytesRead);

// Check if the write operation was successful

if (bytesWritten != bytesRead) {

cerr << "Error writing to output" << endl;

delete[] buffer;

return -1;

}

}

delete[] buffer;

// Check if there was an error while reading from the input file descriptor

if (bytesRead == -1) {

cerr << "Error reading input" << endl;

return -1;

}

return 0;

}

int main(int argc, char\* argv[]) {

if (argc != 2) {

// Print usage message if the number of arguments is incorrect

cerr << "Usage: " << argv[0] << " <input-file>" << endl;

return EXIT\_FAILURE;

}

const char\* filename = argv[1];

// Open the input file for reading

int infd = open(filename, O\_RDONLY);

if (infd == -1) {

// Print an error message using perror if opening the file fails

perror("Error opening input file");

return EXIT\_FAILURE;

}

if (filecopy(infd, STDOUT\_FILENO, 1024) == -1) {

cerr << "Error copying from input file to stdout" << endl;

close(infd);

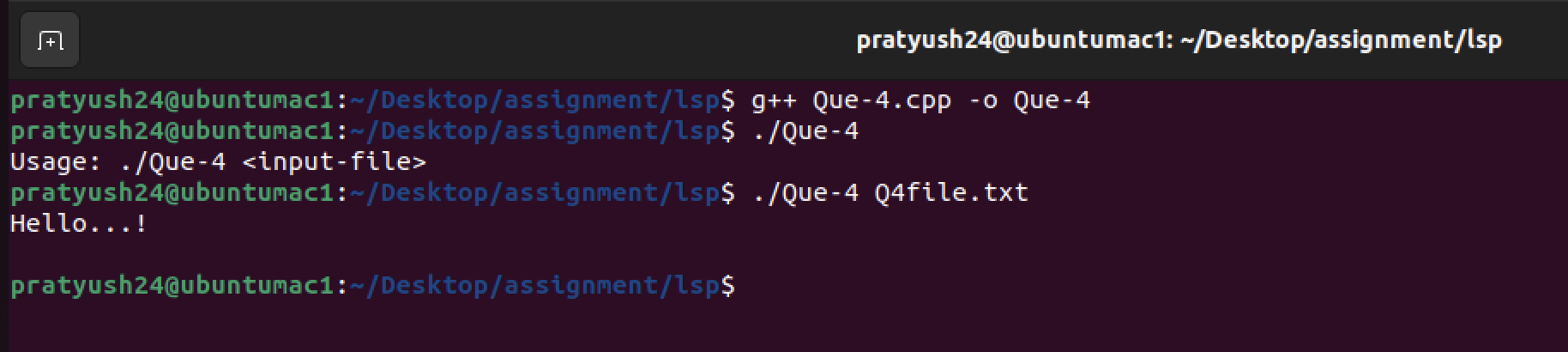
return EXIT\_FAILURE;

}

close(infd);

return EXIT\_SUCCESS;

}



**Que-5 Write a program which forks a process and the parent waits for the child to terminate and prints the exit status, considering the following cases**

**- Child terminates without a call to exit()**

**- Child terminates calling exit(n)**

**- Child terminates calling return(n)**

**hint: Try with values n = 0, n > 0 and n < 0 for argument n**

**ALGORITHM:**

1. Include necessary header files:

- `iostream` for input and output operations.

- `cstdlib` for standard library functions and exit status.

- `unistd.h` for Unix system calls like `fork()`.

- `sys/wait.h` for waiting and checking child process status.

2. Define the `main` function.

3. Create a variable `childPid` to store the process ID (PID) of the child process.

4. Use `fork()` to create a new process:

- If `fork()` returns a negative value, there was an error. Print an error message and exit with a failure status code.

- If `fork()` returns 0, this code is running in the child process.

- If `fork()` returns a positive value, this code is running in the parent process, and the positive value is the PID of the child process.

5. In the child process:

- Print a message indicating that the child is terminating without calling `exit()`.

- The child process implicitly exits at this point.

6. In the parent process:

- Declare an integer variable `childStatus` to store the exit status of the child process.

7. Use `wait(&childStatus)` to wait for the child process to terminate and store its status in `childStatus`.

8. Check the exit status of the child process using `WIFEXITED(childStatus)`:

- If `WIFEXITED(childStatus)` is true, the child process terminated normally.

9. If the child terminated normally:

- Check if the child exited with status 0 (`WEXITSTATUS(childStatus) == 0`), and print a message indicating it exited using `exit(0)`.

- Otherwise, print a message indicating it exited with a non-zero status using `exit(n)`.

10. If the child did not terminate normally (e.g., due to a signal):

- Check if `WIFSIGNALED(childStatus)` is true, indicating that the child was terminated due to a signal.

11. Print a message indicating that the child terminated due to a signal.

12. The `main` function returns `EXIT\_SUCCESS` to indicate successful execution.

**CODE:**

#include <iostream>

#include <cstdlib>

#include <unistd.h>

#include <sys/wait.h>

int main() {

pid\_t childPid = fork();

if (childPid < 0) {

perror("Fork error");

return EXIT\_FAILURE;

} else if (childPid == 0) {

// Child process

// Case 1: Child terminates without a call to exit()

std::cout << "Child (PID: " << getpid() << ") is terminating without exit()" << std::endl;

// Implicitly exit here

} else {

// Parent process

int childStatus;

wait(&childStatus);

if (WIFEXITED(childStatus)) {

// Child terminated normally

// Case 2: Child terminates calling exit(n)

if (WEXITSTATUS(childStatus) == 0) {

std::cout << "Child (PID: " << childPid << ") terminated using exit(0)" << std::endl;

} else {

std::cout << "Child (PID: " << childPid << ") terminated using exit(" << WEXITSTATUS(childStatus) << ")" << std::endl;

}

} else if (WIFSIGNALED(childStatus)) {

// Child terminated due to a signal

// Case 3: Child terminates calling return(n) or due to a signal

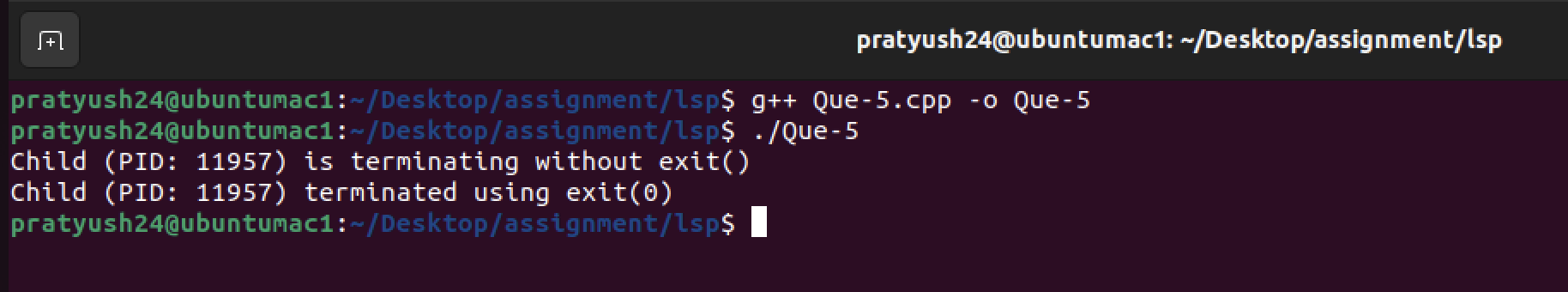
std::cout << "Child (PID: " << childPid << ") terminated due to a signal" << std::endl;

}

}

return EXIT\_SUCCESS;

}



**Que-6 Write an application that creates n child processes each sleeping for random amount of time. Use waitpid(), to see that the parent process waits for each of the child processes to terminate and prints the pid of the child process and its exit status**

**ALGORITHM:**

1. Include necessary libraries:

- Include the iostream library for input/output.

- Include cstdlib for standard C library functions.

- Include unistd.h for system calls like fork and sleep.

- Include sys/types.h and sys/wait.h for process-related functions.

- Include ctime for generating random numbers based on time.

2. Define the `main` function:

- Check if the command-line arguments are not equal to 2. If not, display a usage message and exit with failure status.

- Parse the number of child processes (n) from the command-line argument.

3. Validate the value of n:

- Check if n is less than or equal to 0. If it is, display an error message and exit with failure status.

4. Seed the random number generator using the current time.

5. Create child processes:

- Use a for loop to create n child processes.

- Inside the loop, use `fork()` to create a new child process.

- Check if `fork()` returns a negative value (indicating an error). If so, display an error message and exit with failure status.

- In the child process (when `fork()` returns 0), generate a random sleep time between 1 and 10 seconds using `std::rand()`.

- Display a message indicating the child's PID and sleep time.

- Sleep for the generated sleep time using the `sleep` function.

- Exit the child process with its number as the exit status.

6. In the parent process:

- Use a for loop to wait for the termination of each child process.

- Use `waitpid` to wait for any child process to terminate (`-1` as the first argument).

- Retrieve the child process's termination status.

- Check if the child process exited normally using `WIFEXITED`.

- If it did, display a message indicating the child's PID and exit status using `WEXITSTATUS`.

7. Exit the main function with a successful status.

**CODE:**

#include <iostream>

#include <cstdlib>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <ctime>

int main(int argc, char\* argv[]) {

if (argc != 2) {

std::cerr << "Usage: " << argv[0] << " <number of child processes>" << std::endl;

return EXIT\_FAILURE;

}

int n = std::atoi(argv[1]);

if (n <= 0) {

std::cerr << "Number of child processes should be a positive integer" << std::endl;

return EXIT\_FAILURE;

}

std::srand(time(NULL));

// Create child processes

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

pid\_t childPid = fork();

if (childPid < 0) {

perror("Fork error");

return EXIT\_FAILURE;

} else if (childPid == 0) {

// Child process

int sleepTime = std::rand() % 10 + 1; // Sleep for a random time between 1 and 10 seconds

std::cout << "Child (PID: " << getpid() << ") sleeping for " << sleepTime << " seconds" << std::endl;

sleep(sleepTime);

exit(i); // Exit with child number as exit status

}

}

// Parent process

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

int childStatus;

pid\_t terminatedChildPid = waitpid(-1, &childStatus, 0);

if (WIFEXITED(childStatus)) {

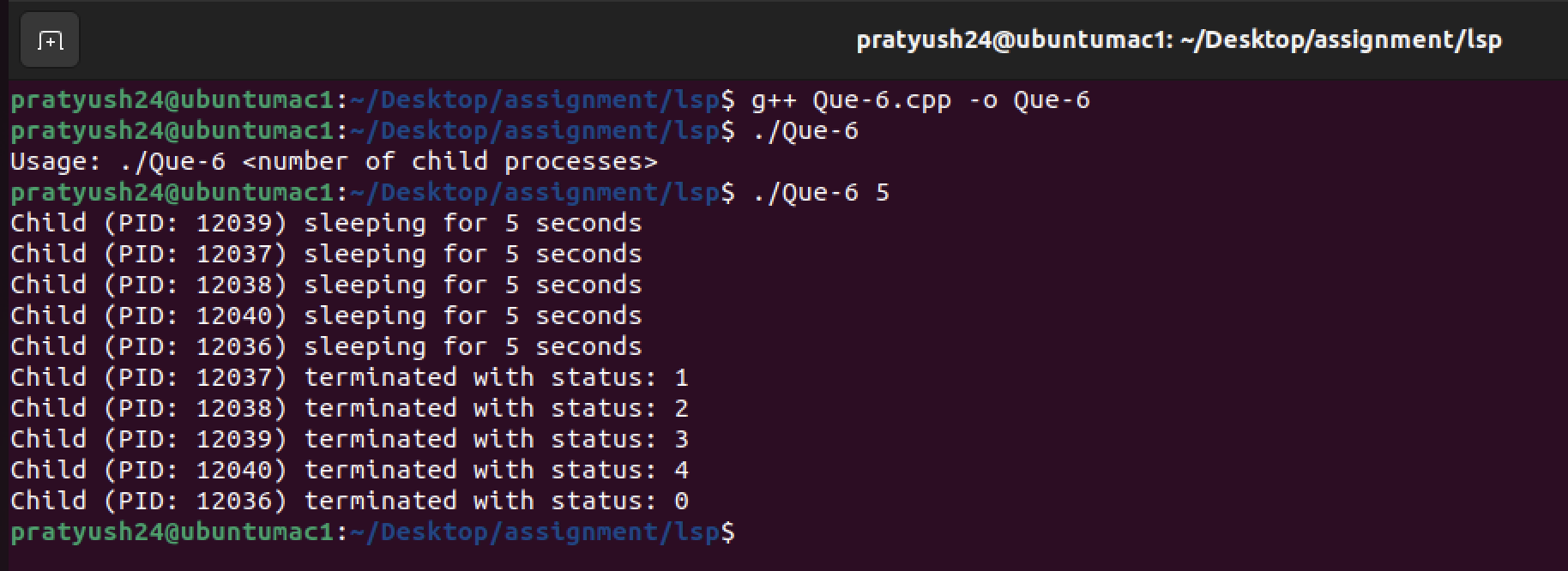
std::cout << "Child (PID: " << terminatedChildPid << ") terminated with status: " << WEXITSTATUS(childStatus) << std::endl;

}

}

return EXIT\_SUCCESS;

}

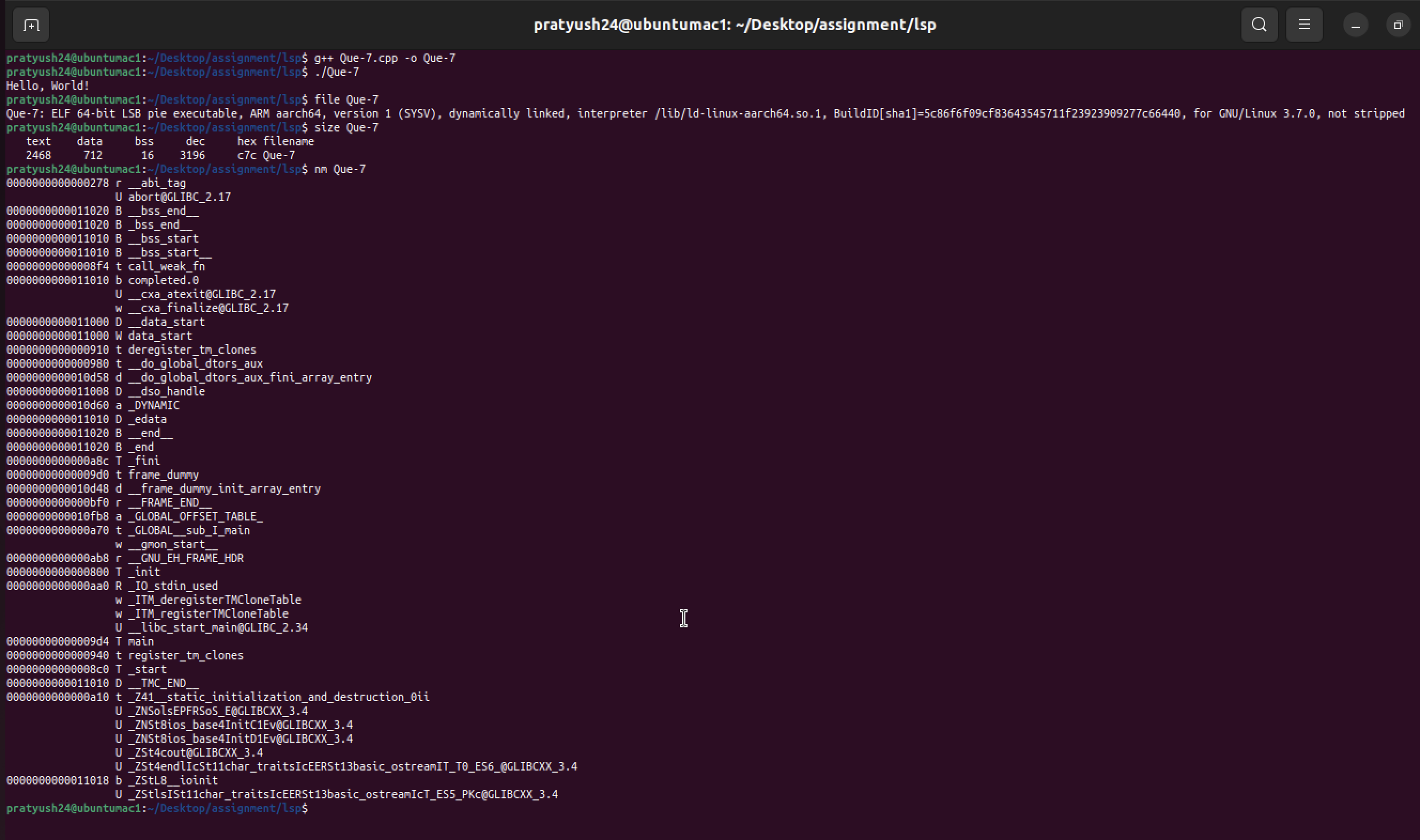


**Que-7 On the executables of the programs that you have already created run the following commands:**

**a. file**

**b. size**

**c. nm**



**Que-8 Write a program with local, global and static variables, and with dynamic memory allocation (malloc) and analyses the memory maps in proc**

**ALGORITHM:**

1. Include necessary header files:

- Include the `<iostream>` header for input and output operations.

- Include `<cstdlib>` for functions like `malloc` and `free`.

- Include `<cstdio>` for C-style input and output operations.

- Include `<cstring>` for string manipulation functions.

2. Declare and initialize global variables:

- Declare an integer global variable `global\_var` and initialize it to 42.

3. Declare a static variable:

- Declare a static integer variable `static\_var` and initialize it to 7.

4. Define the `main` function:

- The main function is the entry point of the program.

5. Declare and initialize a local variable:

- Declare an integer local variable `local\_var` and initialize it to 14.

6. Perform dynamic memory allocation:

- Declare a pointer to an integer called `dynamic\_array`.

- Use `std::malloc` to allocate memory for an array of 5 integers (5 \* sizeof(int)).

- Check if the allocation was successful (if `dynamic\_array` is `nullptr`).

- If allocation fails, print an error message and return `EXIT\_FAILURE`.

7. Initialize the dynamically allocated array:

- Use a `for` loop to populate the `dynamic\_array` with values (0, 2, 4, 6, 8).

8. Output variable values:

- Print the values of:

- `global\_var` using `std::cout`.

- `static\_var` using `std::cout`.

- `local\_var` using `std::cout`.

9. Output the dynamic array values:

- Print the values stored in the `dynamic\_array` using a `for` loop.

10. Free dynamically allocated memory:

- Use `std::free` to release the memory allocated for `dynamic\_array`.

11. Return from `main`:

- Return `EXIT\_SUCCESS` to indicate that the program has executed successfully.

**CODE:**

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <cstring>

// Global variable

int global\_var = 42;

// Static variable

static int static\_var = 7;

int main() {

// Local variable

int local\_var = 14;

// Dynamic memory allocation

int\* dynamic\_array = static\_cast<int\*>(std::malloc(5 \* sizeof(int)));

if (dynamic\_array == nullptr) {

std::cerr << "Memory allocation failed" << std::endl;

return EXIT\_FAILURE;

}

for (int i = 0; i < 5; i++) {

dynamic\_array[i] = i \* 2;

}

std::cout << "Global variable: " << global\_var << std::endl;

std::cout << "Static variable: " << static\_var << std::endl;

std::cout << "Local variable: " << local\_var << std::endl;

std::cout << "Dynamic array: ";

for (int i = 0; i < 5; i++) {

std::cout << dynamic\_array[i] << " ";

}

std::cout << std::endl;

// Free dynamically allocated memory

std::free(dynamic\_array);

return EXIT\_SUCCESS;

}



**Que-9 Define a signal handler that catches SIGINT, SIGTERM and SIGQUIT, prints the signal it has received. Test the program that it functions as desired.**

**ALGORITHM:**

1. Include necessary header files:

- Include the iostream library for input and output.

- Include the csignal library for signal handling.

- Include the unistd.h library for using the `sleep` function.

2. Define a signal handler function:

- Create a function named `signalHandler` that takes an integer `signum` as its parameter.

- Inside the `signalHandler` function, use a `switch` statement to check the value of `signum`:

- If `signum` is equal to `SIGINT`, print "Received SIGINT signal."

- If `signum` is equal to `SIGTERM`, print "Received SIGTERM signal."

- If `signum` is equal to `SIGQUIT`, print "Received SIGQUIT signal."

- If `signum` is not any of the above, print "Received unknown signal" followed by the value of `signum`.

3. Define the main function:

- In the `main` function:

- Register the `signalHandler` function to handle signals `SIGINT`, `SIGTERM`, and `SIGQUIT` using the `signal` function.

- Print the process ID (PID) using `getpid()` and display it as "PID: [PID value]".

- Display "Waiting for signals..." to indicate that the program is ready to receive signals.

- Enter a while loop that continues indefinitely (`while (true)`) to keep the program running.

- Inside the loop, use the `sleep(1)` function to pause the program for one second at a time.

4. Exit the program:

- The program will continue running until it receives a signal. When it receives a signal, the `signalHandler` function will execute, and the program will exit gracefully.

**CODE:**

#include <iostream>

#include <csignal>

#include <unistd.h>

// Signal handler function

void signalHandler(int signum) {

switch (signum) {

case SIGINT:

std::cout << "Received SIGINT signal" << std::endl;

break;

case SIGTERM:

std::cout << "Received SIGTERM signal" << std::endl;

break;

case SIGQUIT:

std::cout << "Received SIGQUIT signal" << std::endl;

break;

default:

std::cout << "Received unknown signal: " << signum << std::endl;

break;

}

}

int main() {

// Register the signal handler for SIGINT, SIGTERM, and SIGQUIT

signal(SIGINT, signalHandler);

signal(SIGTERM, signalHandler);

signal(SIGQUIT, signalHandler);

std::cout << "PID: " << getpid() << std::endl;

std::cout << "Waiting for signals..." << std::endl;

// Keep the program running to catch signals

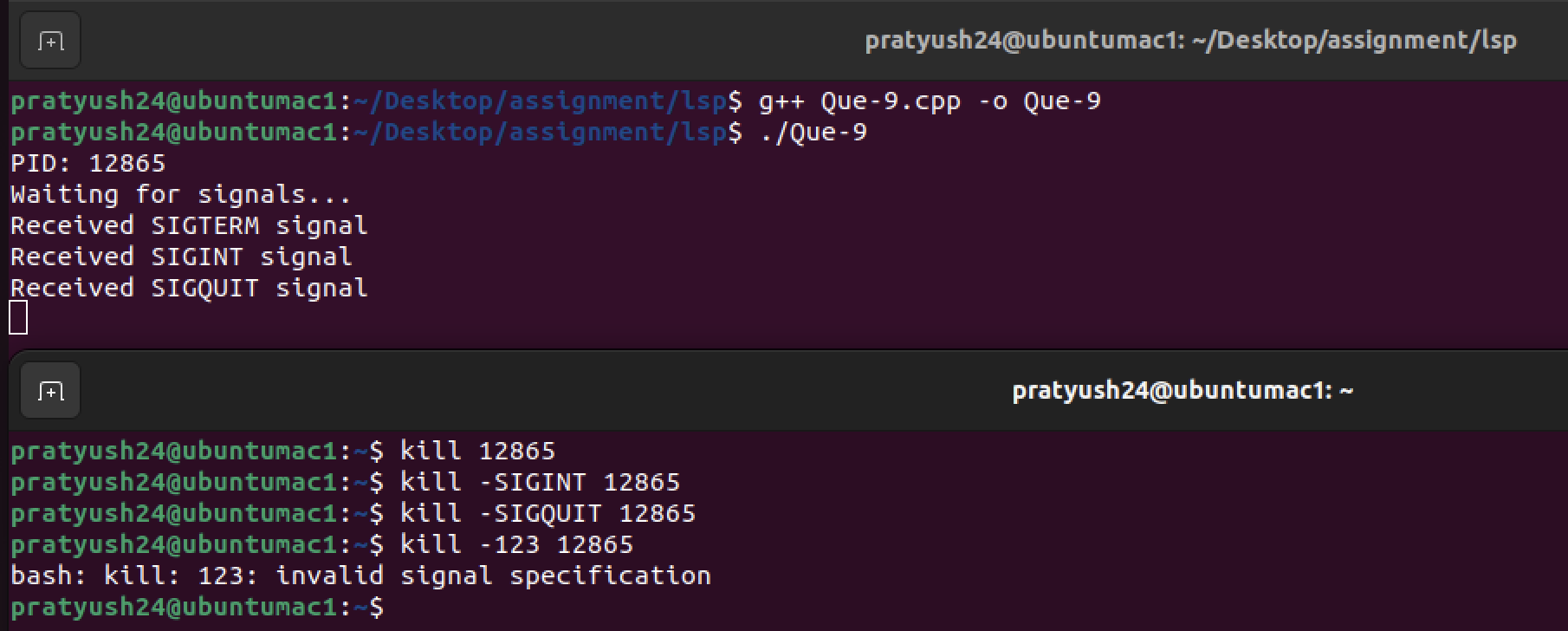
while (true) {

sleep(1);

}

return EXIT\_SUCCESS;

}



**Que-10 Write a program which meets the following requirements.**

**Parent process registers a signal handler for SIGINT, which prints process ID and the signal which the handler received. Parent process creates two child processes both run in infinite loop. Parent process sends SIGINT to both child processes. Parent process sends SIGINT to itself. Run the program and observe the behavior.**

**Note: Use POSIX sigaction to register signal handler.**

**ALGORITHM:**

1. Import necessary libraries:

- Include standard input/output library (`iostream`) for console output.

- Include the library for handling signals (`csignal`) and process management (`unistd.h`, `sys/types.h`, `sys/wait.h`).

2. Define a signal handler function `sigintHandler` to handle the SIGINT signal. It prints a message indicating the signal received and the process ID.

3. In the `main` function:

a. Initialize a `sigaction` structure `sa` to configure the signal handler.

b. Set the `sa\_handler` field of the `sigaction` structure to the `sigintHandler` function.

c. Clear the signal mask using `sigemptyset`.

d. Set `sa\_flags` to 0.

e. Register the `SIGINT` signal handler using `sigaction`. If it fails, print an error message and return with a failure status.

4. Print the parent process ID using `getpid()`.

5. Create the first child process (`child1`) using `fork()`. If `fork()` returns 0, it means this code is executing in the child process (Child 1). In Child 1:

a. Print the Child 1 process ID.

b. Enter an infinite loop.

6. Similarly, create the second child process (`child2`) using `fork()`. If `fork()` returns 0, it means this code is executing in the child process (Child 2). In Child 2:

a. Print the Child 2 process ID.

b. Enter an infinite loop.

7. Sleep for 1 second to give the child processes some time to start.

8. Send the SIGINT signal to both child processes (`child1` and `child2`) using `kill()` to simulate an interruption.

9. Send the SIGINT signal to the parent process itself using `kill()`.

10. Wait for both child processes to terminate using `waitpid()`.

11. Return with a successful exit status.

**CODE:**

#include <iostream>

#include <csignal>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

// Signal handler function

void sigintHandler(int signum) {

std::cout << "Received SIGINT (Signal: " << signum << ") in process " << getpid() << std::endl;

}

int main() {

struct sigaction sa;

sa.sa\_handler = sigintHandler;

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = 0;

if (sigaction(SIGINT, &sa, nullptr) == -1) {

perror("Error registering signal handler");

return EXIT\_FAILURE;

}

std::cout << "Parent process ID: " << getpid() << std::endl;

// Create first child process

pid\_t child1 = fork();

if (child1 == 0) {

// Child 1 process

std::cout << "Child 1 process ID: " << getpid() << std::endl;

while (true) {

// Infinite loop

}

}

// Create second child process

pid\_t child2 = fork();

if (child2 == 0) {

// Child 2 process

std::cout << "Child 2 process ID: " << getpid() << std::endl;

while (true) {

// Infinite loop

}

}

sleep(1); // Give child processes a moment to start

// Send SIGINT to both child processes

kill(child1, SIGINT);

kill(child2, SIGINT);

// Send SIGINT to the parent process itself

kill(getpid(), SIGINT);

// Wait for child processes to terminate

waitpid(child1, nullptr, 0);

waitpid(child2, nullptr, 0);

return EXIT\_SUCCESS;

}

