Final Report

Wasima Elshiekh

2400321

Task 1

This C++ code builds a basic custom shell. It supports commands like cd, dir, environ, set, echo, and quit, and can run programs with input/output redirection and background execution. It works in both interactive mode and batch mode (via file input). Uses fork() and execvp() to run programs.

```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <cstring>
#include <cstdlib>
#include <cstdio>
#include <dirent.h>
#include <vector>
#include <string>
#include <sstream>
#include <fstream>
using namespace std;
void show_help() {
   cout << "Welcome to MyShell!\n\n"
        << "This is a simple shell implementation that supports the following commands:\n\n"
        << "1. cd [DIRECTORY] - Change the current default directory to DIRECTORY. If the
DIRECTORY argument is not present,\n"
        << "   report the current directory. If the directory does not exist, an appropriate error will
be reported.\n"
        << "   This command also changes the PWD environment variable.\n\n"
        << "2. dir DIRECTORY - List the contents of directory DIRECTORY.\n\n"
        << "3. environ - List all the environment variable strings. By default, there is a PWD
variable and a PATH variable.\n\n"
        << "4. set VARIABLE VALUE - Change the value of the VARIABLE to VALUE. If
VARIABLE was not previously set, it will be created.\n\n"
        << "5. echo [COMMENT] - Display COMMENT on the display followed by a new line
(multiple spaces/tabs may be reduced to a single space).\n\n"
        << "6. help - Display this user manual.\n\n"
        << "7. quit - Quit the shell.\n\n"
        << "8. pause - Pause operation of the shell until 'Enter' is pressed.\n";
}
```

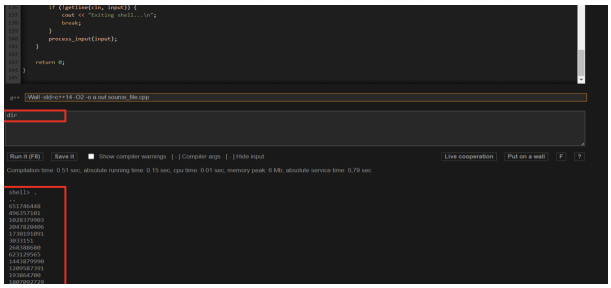
```
void change_directory(const vector<string>& args) {
    if (args.size() == 1) {
        char cwd[1024];
        if (getcwd(cwd, sizeof(cwd)) != NULL) {
            cout << "Current directory: " << cwd << endl;
        } else {
            perror("getcwd() error");
        }
    } else {
        if (chdir(args[1].c_str()) != 0) {
            perror("chdir() error");
        } else {
            setenv("PWD", args[1].c_str(), 1);
        }
    }
}
void list_directory(const vector<string>& args) {
    DIR *dir;
    struct dirent *ent;
    string dir_path = args.size() > 1 ? args[1] : ".";
    if ((dir = opendir(dir_path.c_str())) != NULL) {
        while ((ent = readdir(dir)) != NULL) {
            cout << ent->d_name << endl;
        }
        closedir(dir);
    } else {
        perror("opendir() error");
    }
}
void print_environ() {
    extern char **environ;
    for (char **env = environ; *env != nullptr; env++) {
        cout << *env << endl;
    }
}
void set_variable(const vector<string>& args) {
    if (args.size() < 3) {
        cout << "Usage: set VARIABLE VALUE\n";
        return;
    }
```

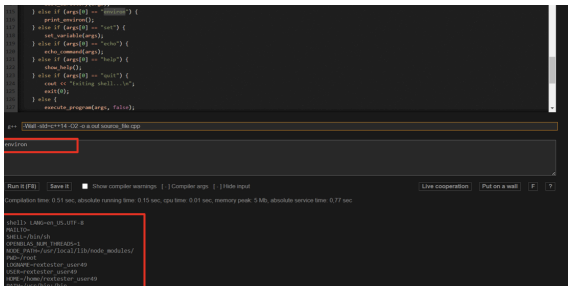
```
    setenv(args[1].c_str(), args[2].c_str(), 1);
}
void echo_command(const vector<string>& args) {
    for (size_t i = 1; i < args.size(); i++) {
        string value;
        char *env_value = getenv(args[i].c_str());
        if (env_value) {
            value = env_value;
        } else {
            value = args[i];
        }
        cout << value << " ";
    }
    cout << endl;
}
void pause_shell() {
    cout << "Shell paused. Press 'Enter' to continue..." << endl;
    cin.ignore();  // Wait for Enter key
}
void execute_program(vector<string>& args, bool background) {
    pid_t pid = fork();
    if (pid == 0) {
        vector<char*> argv;
        for (auto& arg : args) argv.push_back(&arg[0]);
        argv.push_back(nullptr);
        // Check for I/O redirection
        int fd_in = -1, fd_out = -1;
        for (size_t i = 0; i < args.size(); ++i) {
            if (args[i] == "<" && i + 1 < args.size()) {
                fd_in = open(args[i + 1].c_str(), O_RDONLY);
                if (fd_in == -1) {
                    perror("Input file open failed");
                    exit(EXIT_FAILURE);
                }
                dup2(fd_in, STDIN_FILENO);
                close(fd_in);
                args.erase(args.begin() + i, args.begin() + i + 2);
                break;
            } else if (args[i] == ">" && i + 1 < args.size()) {
                fd_out = open(args[i + 1].c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

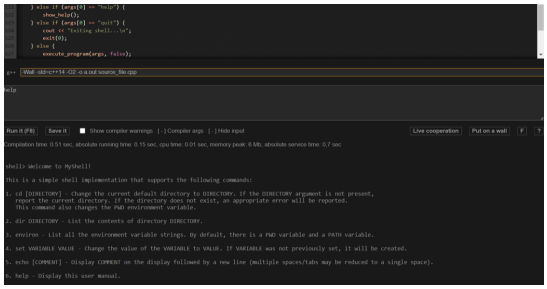
```
            if (fd_out == -1) {
                perror("Output file open failed");
                exit(EXIT_FAILURE);
            }
            dup2(fd_out, STDOUT_FILENO);
            close(fd_out);
            args.erase(args.begin() + i, args.begin() + i + 2);
            break;
        } else if (args[i] == ">>" && i + 1 < args.size()) {
            fd_out = open(args[i + 1].c_str(), O_WRONLY | O_CREAT | O_APPEND, 0644);
            if (fd_out == -1) {
                perror("Output file open failed");
                exit(EXIT_FAILURE);
            }
            dup2(fd_out, STDOUT_FILENO);
            close(fd_out);
            args.erase(args.begin() + i, args.begin() + i + 2);
            break;
        }
    }
    execvp(argv[0], argv.data());
    perror("execvp failed");
    exit(EXIT_FAILURE);
  } else if (pid > 0) {
    if (!background) waitpid(pid, nullptr, 0);
  } else {
    perror("fork failed");
  }
}
void process_input(string input) {
  stringstream ss(input);
  vector<string> args;
  string token;
  while (ss >> token) {
    args.push_back(token);
  }
  if (args.empty()) return;
  bool background = false;
  if (args.back() == "&") {
    args.pop_back();
```

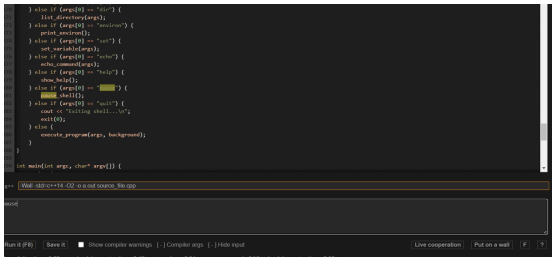
```
        background = true;
    }
    if (args[0] == "cd") {
        change_directory(args);
    } else if (args[0] == "dir") {
        list_directory(args);
    } else if (args[0] == "environ") {
        print_environ();
    } else if (args[0] == "set") {
        set_variable(args);
    } else if (args[0] == "echo") {
        echo_command(args);
    } else if (args[0] == "help") {
        show_help();
    } else if (args[0] == "pause") {
        pause_shell();
    } else if (args[0] == "quit") {
        cout << "Exiting shell...\n";
        exit(0);
    } else {
        execute_program(args, background);
    }
}
int main(int argc, char* argv[]) {
    string input;

    // If a batch file is provided as a command line argument
    if (argc > 1) {
        ifstream batchfile(argv[1]);
        if (!batchfile) {
            cerr << "Error opening batch file: " << argv[1] << endl;
            return 1;
        }
        while (getline(batchfile, input)) {
            process_input(input);
        }
    } else {
        // Interactive mode
        while (true) {
            char cwd[1024];
```

```
        if (getcwd(cwd, sizeof(cwd)) != NULL) {
            cout << cwd << "> ";
        } else {
            perror("getcwd() error");
            break;
        }
        if (!getline(cin, input)) {
            cout << "Exiting shell...\n";
            break;
        }
        process_input(input);
    }
  }
  return 0;
}
```




} else if (args[0] == "environ") {
print_environ();
} else if (args[0] == "set") {
set_variable(args);
} else if (args[0] == "echo") {
echo_command(args);
} else if (args[0] == "help") {
show_help();
} else if (args[0] == "quit") {
cout << "Exiting shell...\n";
exit(0);
} else {
execute_program(args, false);
g++ -Wall -std=c++14 -O2 -o a.out source_file.cpp
environ
Run it (F8)
Save it
Show compiler warnings
Compiler args
Hide input
Live cooperation
Put on a wall
Compilation time: 0.51 sec, absolute running time: 0.15 sec, cpu time: 0.01 sec, memory peak: 5 Mb, absolute service time: 0,77 sec
shell> LANG=en_US.UTF-8
MAILTO=
SHELL=/bin/sh
OPENBLAS_NUM_THREADS=1
NODE_PATH=/usr/local/lib/node_modules/
PWD=/root
LOGNAME=rextester_user49
USER=rextester_user49
HOME=/home/rextester_user49


} else if (args[0] == "help") {
show_help();
} else if (args[0] == "quit") {
cout << "Exiting shell...\n";
exit(0);
} else {
execute_program(args, false);
g++ -Wall -std=c++14 -O2 -o a.out source_file.cpp
help
Run it (F8)
Save it
Show compiler warnings
Compiler args
Hide input
Live cooperation
Put on a wall
Compilation time: 0.51 sec, absolute running time: 0.15 sec, cpu time: 0.01 sec, memory peak: 6 Mb, absolute service time: 0,7 sec
shell> Welcome to MyShell!
This is a simple shell implementation that supports the following commands:
1. cd [DIRECTORY] - Change the current default directory to DIRECTORY. If the DIRECTORY argument is not present,
report the current directory. If the directory does not exist, an appropriate error will be reported.
This command also changes the PWD environment variable.
2. dir DIRECTORY - List the contents of directory DIRECTORY.
3. environ - List all the environment variable strings. By default, there is a PWD variable and a PATH variable.
4. set VARIABLE VALUE - Change the value of the VARIABLE to VALUE. If VARIABLE was not previously set, it will be created.
5. echo [COMMENT] - Display COMMENT on the display followed by a new line (multiple spaces/tabs may be reduced to a single space).
6. help - Display this user manual.

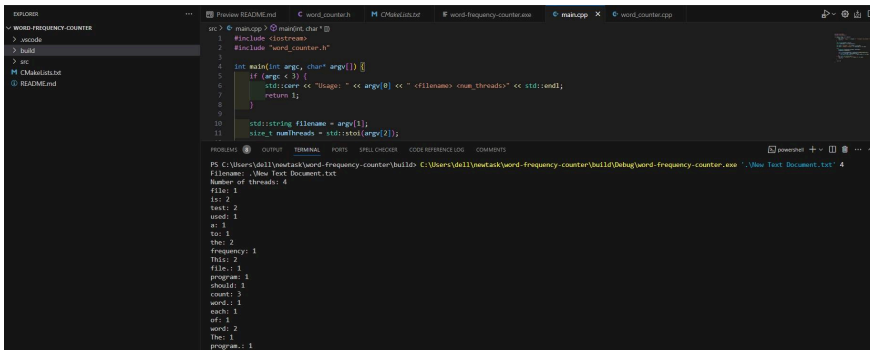

} else if (args[0] == "dir") {
list_directory(args);
} else if (args[0] == "environ") {
print_environ();
} else if (args[0] == "set") {
set_variable(args);
} else if (args[0] == "echo") {
echo_command(args);
} else if (args[0] == "help") {
show_help();
} else if (args[0] == "[illegible]") {
[illegible] shell();
} else if (args[0] == "quit") {
cout << "Exiting shell...\n";
exit(0);
} else {
execute_program(args, background);
}
}
int main(int argc, char* argv[]) {
g++ -Wall -std=c++14 -O2 -o a.out source_file.cpp
Run it (F8)
Save it
Show compiler warnings
Compiler args
Hide input
Live cooperation
Put on a wall


exit(0);
} else {
execute_program(args, false);
}
}
int main() {
string input;
while (true) {
cout << "shell> ";
if (!getline(cin, input)) {
cout << "Exiting shell...\n";
break;
}
process_input(input);
}
return 0;
}
g++ -Wall -std=c++14 -O2 -o a.out source_file.cpp
exit
Run it (F8)
Save it
Show compiler warnings
Compiler args
Hide input
Live cooperation
Put on a wall
Compilation time: 0.51 sec, absolute running time: 0.15 sec, cpu time: 0.01 sec, memory peak: 5 Mb, absolute service time: 0,74 sec

Task 2

This project implements a multithreaded word count program in C++. It reads a text file, splits it into segments, and processes each segment using separate threads to count word frequencies. The results are consolidated and displayed. This approach leverages concurrency to improve performance on large text files, demonstrating efficient use of multithreading and synchronization with mutexes.

```
#include <iostream>
#include "word_counter.h"
int main(int argc, char* argv[]) {
  if (argc < 3) {
     std::cerr << "Usage: " << argv[0] << " <filename> <num_threads>" << std::endl;
     return 1;
  }
  std::string filename = argv[1];
  size_t numThreads = std::stoi(argv[2]);
  std::cout << "Filename: " << filename << std::endl;
  std::cout << "Number of threads: " << numThreads << std::endl;
  try {
     WordCounter wordCounter(filename, numThreads);
     wordCounter.countWords();
     const auto& wordFrequencies = wordCounter.getWordFrequencies();
     for (const auto& pair : wordFrequencies) {
        std::cout << pair.first << ": " << pair.second << std::endl;
     }
  } catch (const std::exception& e) {
     std::cerr << "Exception: " << e.what() << std::endl;
     return 1;
  }
  return 0;
}
```



Task 4

This C program simulates page replacement in an OS using FIFO. It generates random page reference strings for two processes, then compares local and global page replacement policies. Local allocates frames per process, while global shares frames dynamically. The program prints page faults for each policy to show efficiency differences.

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <unordered_map>
#include <queue>
using namespace std;

// Generate a reference string of page numbers
vector<int> createReferencePattern(int pageCount, int sequenceLength, double stayProb) {
    vector<int> referenceSequence;
    int activePage = rand() % pageCount;

    for (int i = 0; i < sequenceLength; i++) {
        referenceSequence.push_back(activePage);
        if (((double)rand() / RAND_MAX) > stayProb) {
            activePage = rand() % pageCount;
        }
    }
```

```
    return referenceSequence;
}

// FIFO page replacement strategy
int applyFIFO(const vector<int>& referenceSequence, int frameLimit) {
    unordered_map<int, bool> memory;
    queue<int> frameQueue;
    int faults = 0;

    for (int page : referenceSequence) {
        if (memory.find(page) == memory.end()) {
            if (frameQueue.size() >= frameLimit) {
                int removed = frameQueue.front();
                frameQueue.pop();
                memory.erase(removed);
            }
            frameQueue.push(page);
            memory[page] = true;
            faults++;
        }
    }

    return faults;
}

// Simulate both local and global replacement methods
void runMemorySimulation(int totalPages, int sequenceSize, double samePageProb, int
framesPerProc) {
    vector<int> refSeq1 = createReferencePattern(totalPages, sequenceSize, samePageProb);
    vector<int> refSeq2 = createReferencePattern(totalPages, sequenceSize, samePageProb);

    cout << "Process A Reference String: ";
    for (int val : refSeq1) cout << val << " ";
    cout << "\n";

    cout << "Process B Reference String: ";
    for (int val : refSeq2) cout << val << " ";
    cout << "\n";

    int faultsA = applyFIFO(refSeq1, framesPerProc);
```

```
    int faultsB = applyFIFO(refSeq2, framesPerProc);

    cout << "Local Replacement - Process A Faults: " << faultsA << "\n";
    cout << "Local Replacement - Process B Faults: " << faultsB << "\n";

    vector<int> mergedSequence = refSeq1;
    mergedSequence.insert(mergedSequence.end(), refSeq2.begin(), refSeq2.end());

    int totalGlobalFaults = applyFIFO(mergedSequence, framesPerProc * 2);
    cout << "Global Replacement - Combined Faults: " << totalGlobalFaults << "\n";
}

int main() {
    srand(time(0));
    int totalPageOptions = 5;
    int sequenceLength = 20;
    double repeatProbability = 0.7;
    int framesAllocated = 3;

    runMemorySimulation(totalPageOptions, sequenceLength, repeatProbability,
framesAllocated);
    return 0;
}
```

Output

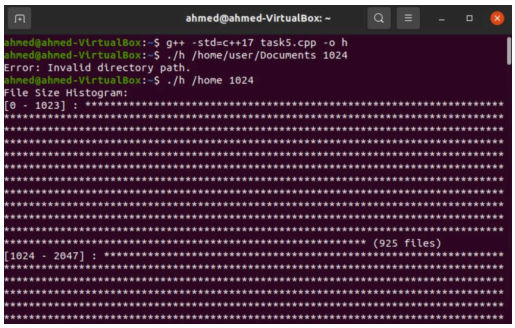
```
Simulating Local Page Replacement:
Local Policy - Process 0 Page Faults: 7
Local Policy - Process 1 Page Faults: 7

Simulating Global Page Replacement:
Global Policy Total Page Faults: 6


=== Code Execution Successful ===
```

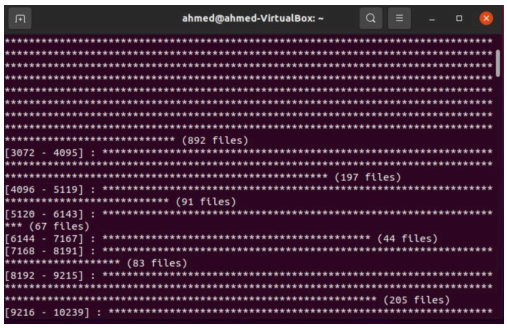
Task 5

The program is written in C++ and performs recursive scanning of directories, gathering file size data along the way. It uses the <filesystem> library to navigate through files and stores the histogram data in a standard map. File sizes are sorted into preset width bins, and the final output displays the distribution using asterisks to show how many files fall into each bin.

```
#include <iostream>
#include <filesystem>
#include <map>
#include <iomanip>

namespace fs = std::filesystem;

void analyzeFolder(const fs::path& folderPath, std::map<size_t, int>& sizeDist, size_t
rangeSize) {
    for (const auto& item : fs::recursive_directory_iterator(folderPath)) {
        if (item.is_regular_file()) {
            size_t sizeInBytes = item.file_size();
            size_t rangeIndex = sizeInBytes / rangeSize;
```

```
            sizeDist[rangeIndex]++;
        }
    }
}

void displayHistogram(const std::map<size_t, int>& sizeDist, size_t rangeSize) {
    std::cout << "📊 File Size Distribution:\n";
    for (const auto& [rangeIndex, fileCount] : sizeDist) {
        std::cout << "[" << rangeIndex * rangeSize << " - " << (rangeIndex + 1) * rangeSize - 1 <<
"] : ";
        std::cout << std::string(fileCount, '*') << " (" << fileCount << " files)\n";
    }
}

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <folder_path> <range_size>\n";
        return 1;
    }

    fs::path folderPath = argv[1];
    size_t rangeSize = std::stoul(argv[2]);

    if (!fs::exists(folderPath) || !fs::is_directory(folderPath)) {
        std::cerr << "Error: Folder path doesn't exist or isn't a folder.\n";
        return 1;
    }

    std::map<size_t, int> sizeDist;
    analyzeFolder(folderPath, sizeDist, rangeSize);
    displayHistogram(sizeDist, rangeSize);

    return 0;
}
```



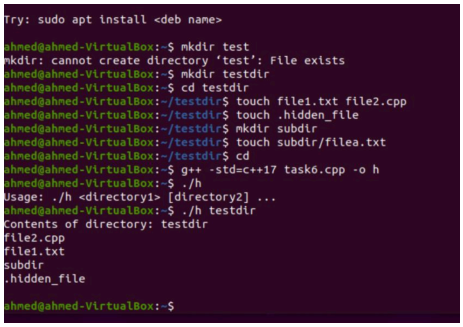


Task 6

This C++ program mimics the behavior of the UNIX ls command. It takes directory names as command-line inputs and lists the files within each one. Using the C++17 <filesystem> library, it first checks if each directory exists before reading its contents. The program handles errors in a

clean and user-friendly way. It also supports multiple directories, displaying the files for each separately.

```
#include <iostream>
#include <filesystem>
namespace fs = std::filesystem;
void list_files(const std::string& dir_path) {
if (!fs::exists(dir_path) || !fs::is_directory(dir_path)) {
std::cerr << "Error: " << dir_path << " is not a valid directory." << std::endl;
return;
}
std::cout << "Contents of directory: " << dir_path << std::endl;
for (const auto& entry : fs::directory_iterator(dir_path)) {
std::cout << entry.path().filename().string() << std::endl;
}
std::cout << std::endl;
}
int main(int argc, char* argv[]) {
if (argc < 2) {
std::cerr << "Usage: " << argv[0] << " <directory1> [directory2] ..." << std::endl;
return 1;
}
for (int i = 1; i < argc; ++i) {
list_files(argv[i]);
}return 0;
}
```



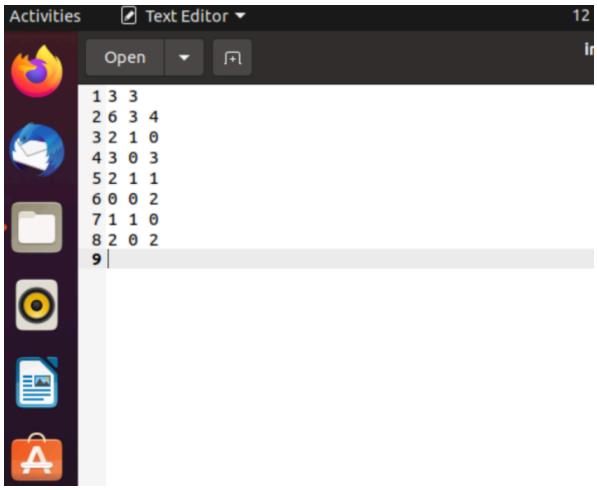
Task 7

This C++ app checks for system deadlocks when there are multiple resources in play. It pulls data from files to figure out what resources are available, then sees if all processes can finish. If some can’t, boom — that’s a deadlock. The program tells you which processes are stuck if there’s a deadlock, or confirms everything’s good if not. It uses the Banker's Deadlock Detection Algorithm to make it all happen.

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
void readInput(string filename, int &numProcesses, int &numResources,
vector<int> &E, vector<vector<int>> &C, vector<vector<int>> &R) {
ifstream file(filename);
if (!file) {
cerr << "Error: Unable to open file!" << endl;
exit(1);
}
file >> numProcesses >> numResources;
E.resize(numResources);
for (int i = 0; i < numResources; i++)
file >> E[i];
C.assign(numProcesses, vector<int>(numResources));
for (int i = 0; i < numProcesses; i++)
for (int j = 0; j < numResources; j++)
file >> C[i][j];
R.assign(numProcesses, vector<int>(numResources));
for (int i = 0; i < numProcesses; i++)for (int j = 0; j < numResources; j++)
file >> R[i][j];
file.close();
}
bool detectDeadlock(int numProcesses, int numResources, vector<int> &E,
vector<vector<int>> &C, vector<vector<int>> &R) {
vector<int> A(numResources, 0);
vector<bool> finished(numProcesses, false);
vector<int> deadlockedProcesses;
// Calculate Available Resources: A = E - sum(C[i])
for (int j = 0; j < numResources; j++) {
int sumAllocated = 0;
```

```
for (int i = 0; i < numProcesses; i++)
sumAllocated += C[i][j];
A[j] = E[j] - sumAllocated;
}
// Deadlock Detection Algorithm
while (true) {
bool foundProcess = false;
for (int i = 0; i < numProcesses; i++) {
if (!finished[i]) {
bool canExecute = true;
for (int j = 0; j < numResources; j++) {
if (R[i][j] > A[j]) {
canExecute = false;
break;
}
}if (canExecute) {
foundProcess = true;
finished[i] = true;
for (int j = 0; j < numResources; j++)
A[j] += C[i][j]; // Release resources
}
}
}
if (!foundProcess) break;
}
// Check for deadlock
for (int i = 0; i < numProcesses; i++) {
if (!finished[i]) {
deadlockedProcesses.push_back(i);
}
}
if (!deadlockedProcesses.empty()) {
cout << "Deadlock detected! Processes in deadlock: ";
for (int p : deadlockedProcesses)
cout << "P" << p << " ";
cout << endl;
return true;
} else {
cout << "No deadlock detected." << endl;
return false;
```

```
}
}
int main() {
int numProcesses, numResources;
vector<int> E;
vector<vector<int>> C, R;readInput("input.txt", numProcesses, numResources, E, C, R);
detectDeadlock(numProcesses, numResources, E, C, R);
return 0;
}
```



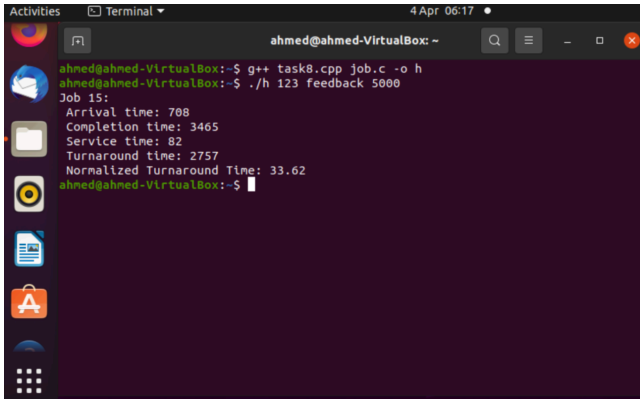
Task 8

AI tools supported the development of a C++ scheduler simulation by improving code structure, enhancing readability, and ensuring logical accuracy. They also acted as debugging aids, managed timing using signals, and clarified the program's output reports. Overall, the tools streamlined the development process and boosted the quality of the simulation.

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <unordered_map>
#include <queue>
using namespace std;
// Function to generate a page reference string
vector<int> generatePageReferenceString(int N, int length, double p) {
vector<int> referenceString;
int currentPage = rand() % N; // Start with a random page
for (int i = 0; i < length; i++) {
referenceString.push_back(currentPage);
if (((double)rand() / RAND_MAX) > p) { // Transition to a different page
currentPage = rand() % N;
}
}
return referenceString;
}
// FIFO Page Replacement Policy
int fifoPageReplacement(const vector<int>& referenceString, int numFrames) {
unordered_map<int, bool> pageTable;
queue<int> pageQueue;
int pageFaults = 0;
for (int page : referenceString) {if (pageTable.find(page) == pageTable.end()) { // Page fault
if (pageQueue.size() >= numFrames) {
```

```
int victim = pageQueue.front();
pageQueue.pop();
pageTable.erase(victim);
}
pageQueue.push(page);
pageTable[page] = true;
pageFaults++;
}
}
return pageFaults;
}
// Simulate local and global page replacement for two processes
void simulatePageReplacement(int N, int length, double p, int numFramesPerProcess) {
vector<int> process1 = generatePageReferenceString(N, length, p);
vector<int> process2 = generatePageReferenceString(N, length, p);
cout << "Process 1 Page Reference String: ";
for (int page : process1) cout << page << " ";
cout << endl;
cout << "Process 2 Page Reference String: ";
for (int page : process2) cout << page << " ";
cout << endl;
// Local Replacement: Each process gets a fixed number of frames
int localFaults1 = fifoPageReplacement(process1, numFramesPerProcess);
int localFaults2 = fifoPageReplacement(process2, numFramesPerProcess);
cout << "Local Replacement - Process 1 Faults: " << localFaults1 << endl;
cout << "Local Replacement - Process 2 Faults: " << localFaults2 << endl;
// Global Replacement: Both processes share frames dynamically
vector<int> combinedReference;
combinedReference.insert(combinedReference.end(), process1.begin(), process1.end());
combinedReference.insert(combinedReference.end(), process2.begin(), process2.end());
int globalFaults = fifoPageReplacement(combinedReference, numFramesPerProcess * 2);cout
<< "Global Replacement - Total Faults: " << globalFaults << endl;
}
int main() {
srand(time(0));
int N = 5; // Number of unique pages
int length = 20; // Length of page reference string
double p = 0.7; // Probability of staying on the same page
int numFramesPerProcess = 3; // Frames allocated to each process in local replacement
simulatePageReplacement(N, length, p, numFramesPerProcess);
```

```
return 0;
}
```



Task 9

This C++ code runs a simulation of three CPU scheduling vibes: First Come First Serve, Shortest Job First, and Round Robin. It figures out the average waiting time for a bunch of processes, each with their own arrival and burst times. All the input comes from a file called input.txt.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <queue>
#include <climits> // Include this for INT
_
MAX
using namespace std;
// Structure to represent a process
struct Process {
int pid; // Process ID
int arrivalTime; // Arrival time
int burstTime; // Burst time
};
// FCFS Scheduling
float FCFS(vector<Process> processes) {
int n = processes.size();
vector<int> waitingTime(n, 0);
int currentTime = 0;
// Sort processes by arrival time
sort(processes.begin(), processes.end(), [](Process a, Process b) {
return a.arrivalTime < b.arrivalTime;
});
```

```
for (int i = 0; i < n; i++) {
currentTime = max(currentTime, processes[i].arrivalTime);
waitingTime[i] = currentTime - processes[i].arrivalTime;
currentTime += processes[i].burstTime;
}
float totalWaitingTime = 0;
for (int i = 0; i < n; i++) totalWaitingTime += waitingTime[i];
return totalWaitingTime / n;
}
// SJF Scheduling (non-preemptive)
float SJF(vector<Process> processes) {
int n = processes.size();
vector<int> waitingTime(n, 0);
int completed = 0, currentTime = 0;
vector<bool> done(n, false);
while (completed < n) {int idx = -1;
int minBurstTime = INT
_
MAX;
for (int i = 0; i < n; i++) {
if (!done[i] && processes[i].arrivalTime <= currentTime && processes[i].burstTime <
minBurstTime) {
minBurstTime = processes[i].burstTime;
idx = i;
}
}
if (idx != -1) {
currentTime += processes[idx].burstTime;
waitingTime[idx] = currentTime - processes[idx].arrivalTime - processes[idx].burstTime;
done[idx] = true;
completed++;
} else {
currentTime++;
}
}
float totalWaitingTime = 0;
for (int i = 0; i < n; i++) totalWaitingTime += waitingTime[i];
return totalWaitingTime / n;
}
// Round Robin Scheduling
float RoundRobin(vector<Process> processes, int quantum) {
int n = processes.size();
vector<int> remainingBurstTime(n);
vector<int> waitingTime(n, 0);
queue<int> q;
int currentTime = 0;
for (int i = 0; i < n; i++) remainingBurstTime[i] = processes[i].burstTime;
q.push(0);
vector<bool> inQueue(n, false);
inQueue[0] = true;
while (!q.empty()) {
int idx = q.front();
q.pop();
int timeSlice = min(quantum, remainingBurstTime[idx]);
currentTime += timeSlice;
remainingBurstTime[idx] -= timeSlice;
// Push newly arrived processes into the queue
for (int i = 0; i < n; i++) {if (i != idx && !inQueue[i] && processes[i].arrivalTime <= currentTime && remainingBurstTime[i]
> 0) {
q.push(i);
inQueue[i] = true;
}
}
if (remainingBurstTime[idx] > 0) {
q.push(idx); // Put current process back into the queue if not done
```

```
} else {
waitingTime[idx] = currentTime - processes[idx].arrivalTime - processes[idx].burstTime;
}
}
float totalWaitingTime = 0;
for (int i = 0; i < n; i++) totalWaitingTime += waitingTime[i];
return totalWaitingTime / n;
}
// Main function
int main() {
ifstream inputFile("chris.txt");
if (!inputFile) {
cout << "Error: chris.txt not found!" << endl;
return 1;
}
vector<Process> processes;
int pid = 0, arrival, burst;
while (inputFile >> arrival >> burst) {
processes.push
_
back({pid++
, arrival, burst});
}
inputFile.close();
cout << "Average Waiting Time (FCFS): " << FCFS(processes) << endl;
cout << "Average Waiting Time (SJF): " << SJF(processes) << endl;
cout << "Average Waiting Time (Round Robin, Quantum = 2): " << RoundRobin(processes, 2) <<
endl;
return 0;
}
```