

**OPERATING SYSTEMS**

**Assignment**

**Process Management and Scheduling**

Submitted by

**Team Members**

**Sampath Kumar (4SF22CS175)**

**Rohan Shankar (4SF22CS167)**

**Nischal B K (4SF22CS123)**

Submitted to

**Prof. Prapulla G**

Assistant Professor

Department of CSE,SCEM

**Process Management and Scheduling**

**1. Implement a Simple Shell:**

-> Write a simple shell program that can execute user commands.

-> Implement basic commands like cd, ls, mkdir, and exit.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#define MAX\_LINE 1024

#define MAX\_ARGS 64

void parse\_input(char \*line, char \*\*args) {

    while (\*line != '\0') {

        while (\*line == ' ' || \*line == '\t' || \*line == '\n') {

            \*line++ = '\0';

        }

        \*args++ = line;

        while (\*line != '\0' && \*line != ' ' && \*line != '\t' && \*line != '\n') {

            line++;

        }

    }

    \*args = '\0';

}

void execute\_command(char \*\*args) {

    // Check for built-in commands

    if (strcmp(args[0], "cd") == 0) {

        if (args[1] == NULL) {

            fprintf(stderr, "cd: expected argument\n");

        } else {

            if (chdir(args[1]) != 0) {

                perror("cd failed");

            }

        }

        return;

    }

    pid\_t pid = fork();

    if (pid < 0) {

        perror("Fork failed");

        exit(1);

    } else if (pid == 0) {

        if (execvp(args[0], args) < 0) {

            perror("Exec failed");

            exit(1);

        }

    } else {

        wait(NULL);

    }

}

int main() {

    char line[MAX\_LINE];

    char \*args[MAX\_ARGS];

    while (1) {

        printf("simple\_shell> ");

        if (fgets(line, sizeof(line), stdin) == NULL) {

            break;

        }

        if (line[0] == '\n') {

            continue;

        }

        line[strlen(line) - 1] = '\0';  // Remove the newline character

        parse\_input(line, args);

        if (strcmp(args[0], "exit") == 0) {

            break;

        }

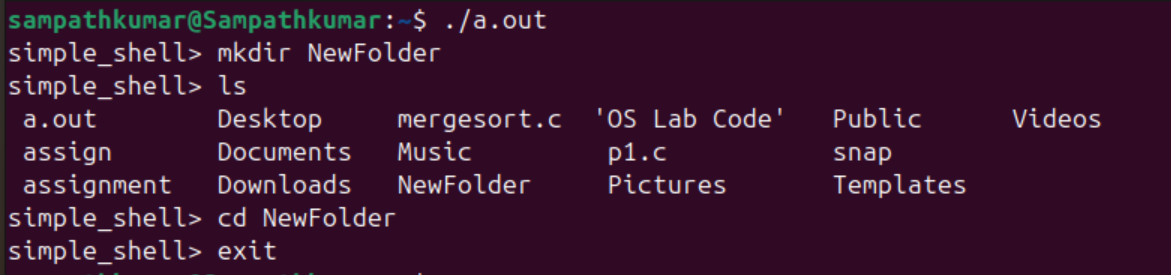
        execute\_command(args);

    }

    return 0;

}

**Output:**

****

**Explanation:**

1. **Includes and Definitions**:
   * Includes standard libraries for input/output, process control, and string manipulation.
   * Defines constants for the maximum line length and number of arguments.
2. **parse\_input Function**:
   * This function splits the input line into arguments.
   * It replaces spaces, tabs, and newlines with \0 to separate each argument and stores pointers to each argument in an array.
3. **execute\_command Function**:
   * This function creates a new process using fork().
   * The child process uses execvp to execute the command.
   * The parent process waits for the child to complete using wait.
4. **Main Function**:
   * Runs an infinite loop that prints a prompt and reads user input.
   * Calls parse\_input to split the input into arguments.
   * If the command is exit, the loop breaks and the shell exits.
   * Calls execute\_command to run the entered command.

**2. Simulate Scheduling Algorithms:**

-> Implement and simulate different CPU scheduling algorithms (FCFS, SJF, Priority, Round

Robin).

->Compare their performance based on average waiting time and turnaround time.

#include <stdio.h>

#include <stdlib.h>

typedef struct {

    int pid;       // Process ID

    int burst;     // Burst Time

    int priority;  // Priority

    int arrival;   // Arrival Time

} Process;

// Function to compare based on arrival time

int compare\_arrival(const void \*a, const void \*b) {

    return ((Process \*)a)->arrival - ((Process \*)b)->arrival;

}

// Function to compare based on burst time

int compare\_burst(const void \*a, const void \*b) {

    return ((Process \*)a)->burst - ((Process \*)b)->burst;

}

// Function to compare based on priority

int compare\_priority(const void \*a, const void \*b) {

    return ((Process \*)a)->priority - ((Process \*)b)->priority;

}

// FCFS Scheduling

void fcfs(Process \*proc, int n) {

    qsort(proc, n, sizeof(Process), compare\_arrival);

    int wait\_time[n], turnaround\_time[n];

    int total\_wait = 0, total\_turnaround = 0;

    wait\_time[0] = 0;

    for (int i = 1; i < n; i++) {

        wait\_time[i] = proc[i-1].burst + wait\_time[i-1];

    }

    for (int i = 0; i < n; i++) {

        turnaround\_time[i] = proc[i].burst + wait\_time[i];

        total\_wait += wait\_time[i];

        total\_turnaround += turnaround\_time[i];

    }

    printf("FCFS Scheduling:\n");

    printf("Average Waiting Time: %.2f\n", (float)total\_wait / n);

    printf("Average Turnaround Time: %.2f\n\n", (float)total\_turnaround / n);

}

// SJF Scheduling

void sjf(Process \*proc, int n) {

    qsort(proc, n, sizeof(Process), compare\_burst);

    int wait\_time[n], turnaround\_time[n];

    int total\_wait = 0, total\_turnaround = 0;

    wait\_time[0] = 0;

    for (int i = 1; i < n; i++) {

        wait\_time[i] = proc[i-1].burst + wait\_time[i-1];

    }

    for (int i = 0; i < n; i++) {

        turnaround\_time[i] = proc[i].burst + wait\_time[i];

        total\_wait += wait\_time[i];

        total\_turnaround += turnaround\_time[i];

    }

    printf("SJF Scheduling:\n");

    printf("Average Waiting Time: %.2f\n", (float)total\_wait / n);

    printf("Average Turnaround Time: %.2f\n\n", (float)total\_turnaround / n);

}

// Priority Scheduling

void priority\_scheduling(Process \*proc, int n) {

    qsort(proc, n, sizeof(Process), compare\_priority);

    int wait\_time[n], turnaround\_time[n];

    int total\_wait = 0, total\_turnaround = 0;

    wait\_time[0] = 0;

    for (int i = 1; i < n; i++) {

        wait\_time[i] = proc[i-1].burst + wait\_time[i-1];

    }

    for (int i = 0; i < n; i++) {

        turnaround\_time[i] = proc[i].burst + wait\_time[i];

        total\_wait += wait\_time[i];

        total\_turnaround += turnaround\_time[i];

    }

    printf("Priority Scheduling:\n");

    printf("Average Waiting Time: %.2f\n", (float)total\_wait / n);

    printf("Average Turnaround Time: %.2f\n\n", (float)total\_turnaround / n);

}

// Round Robin Scheduling

void round\_robin(Process \*proc, int n, int quantum) {

    int wait\_time[n], turnaround\_time[n], remaining\_burst[n];

    int total\_wait = 0, total\_turnaround = 0, time = 0;

    for (int i = 0; i < n; i++) {

        remaining\_burst[i] = proc[i].burst;

    }

    while (1) {

        int done = 1;

        for (int i = 0; i < n; i++) {

            if (remaining\_burst[i] > 0) {

                done = 0;

                if (remaining\_burst[i] > quantum) {

                    time += quantum;

                    remaining\_burst[i] -= quantum;

                } else {

                    time += remaining\_burst[i];

                    wait\_time[i] = time - proc[i].burst;

                    remaining\_burst[i] = 0;

                }

            }

        }

        if (done) {

            break;

        }

    }

    for (int i = 0; i < n; i++) {

        turnaround\_time[i] = proc[i].burst + wait\_time[i];

        total\_wait += wait\_time[i];

        total\_turnaround += turnaround\_time[i];

    }

    printf("Round Robin Scheduling (Quantum %d):\n", quantum);

    printf("Average Waiting Time: %.2f\n", (float)total\_wait / n);

    printf("Average Turnaround Time: %.2f\n\n", (float)total\_turnaround / n);

}

int main() {

    Process proc[] = {

        {1, 10, 3, 0},

        {2, 5, 1, 0},

        {3, 8, 2, 0},

        {4, 6, 4, 0}

    };

    int n = sizeof(proc) / sizeof(proc[0]);

    int quantum = 3;

    fcfs(proc, n);

    sjf(proc, n);

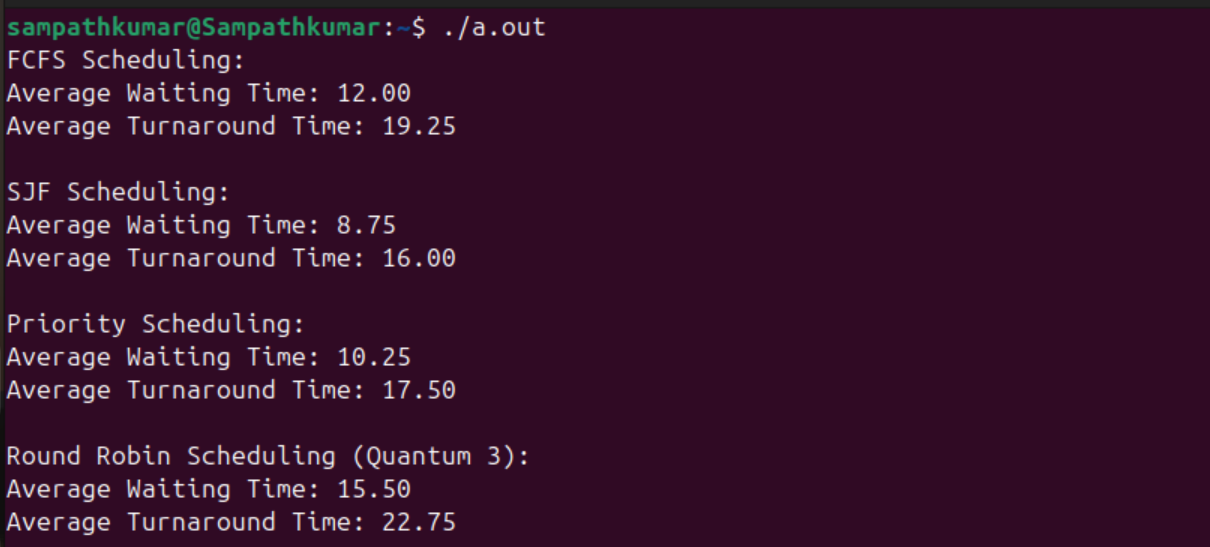
    priority\_scheduling(proc, n);

    round\_robin(proc, n, quantum);

    return 0;

}

**Output:**



### Explanation:

1. **Process Structure**:
   * The Process struct stores the process ID, burst time, priority, and arrival time.
2. **Comparison Functions**:
   * compare\_arrival, compare\_burst, and compare\_priority are used for sorting the processes based on different criteria using qsort.
3. **FCFS Scheduling**:
   * Processes are sorted by arrival time.
   * Waiting and turnaround times are calculated sequentially.
4. **SJF Scheduling**:
   * Processes are sorted by burst time.
   * Waiting and turnaround times are calculated sequentially.
5. **Priority Scheduling**:
   * Processes are sorted by priority.
   * Waiting and turnaround times are calculated sequentially.
6. **Round Robin Scheduling**:
   * The algorithm cycles through processes, giving each one a fixed quantum of time.
   * If a process's remaining burst time is less than the quantum, it completes in that cycle.
   * Waiting and turnaround times are calculated once all processes are done.
7. **Main Function**:
   * An array of Process structs is defined to represent the processes.
   * The scheduling functions are called, and results are printed.

**3. Create and Manage Child Processes:**

-> Write a program to create multiple child processes using fork().

-> Implement communication between parent and child processes using pipes.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/wait.h>

#define NUM\_CHILDREN 3

#define BUFFER\_SIZE 100

int main() {

    int pipefd[NUM\_CHILDREN][2];

    pid\_t pid[NUM\_CHILDREN];

    char buffer[BUFFER\_SIZE];

    const char \*message = "Hello from parent";

    // Create pipes and fork child processes

    for (int i = 0; i < NUM\_CHILDREN; i++) {

        if (pipe(pipefd[i]) == -1) {

            perror("pipe");

            exit(EXIT\_FAILURE);

        }

        pid[i] = fork();

        if (pid[i] < 0) {

            perror("fork");

            exit(EXIT\_FAILURE);

        }

        if (pid[i] == 0) { // Child process

            close(pipefd[i][1]); // Close write end of the first pipe

            read(pipefd[i][0], buffer, BUFFER\_SIZE);

            printf("Child %d received message: %s\n", i, buffer);

            close(pipefd[i][0]); // Close read end of the first pipe

            int pipefd\_response[2];

            if (pipe(pipefd\_response) == -1) {

                perror("pipe");

                exit(EXIT\_FAILURE);

            }

            pid\_t response\_pid = fork();

            if (response\_pid < 0) {

                perror("fork");

                exit(EXIT\_FAILURE);

            }

            if (response\_pid == 0) {

                close(pipefd\_response[0]); // Close read end of the response pipe

                snprintf(buffer, BUFFER\_SIZE, "Hello from child %d", i);

                write(pipefd\_response[1], buffer, strlen(buffer) + 1);

                close(pipefd\_response[1]); // Close write end of the response pipe

                exit(EXIT\_SUCCESS);

            } else {

                close(pipefd\_response[1]); // Close write end of the response pipe

                wait(NULL); // Wait for the response child to send the message

                read(pipefd\_response[0], buffer, BUFFER\_SIZE);

                close(pipefd\_response[0]); // Close read end of the response pipe

                printf("Parent received message: %s\n", buffer);

                exit(EXIT\_SUCCESS);

            }

        } else { // Parent process

            close(pipefd[i][0]); // Close read end of the first pipe

            write(pipefd[i][1], message, strlen(message) + 1);

            close(pipefd[i][1]); // Close write end of the first pipe

        }

    }

    // Wait for all child processes to finish

    for (int i = 0; i < NUM\_CHILDREN; i++) {

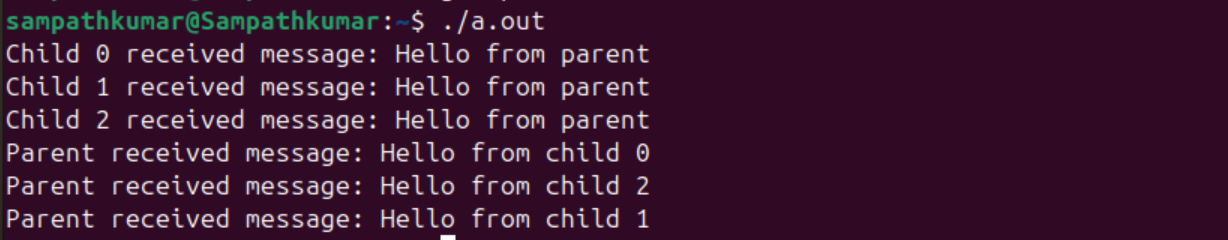
        waitpid(pid[i], NULL, 0);

    }

    return 0;

}

**Output:**



### Explanation

1. **Pipe Creation**:
   * We create a pipe for each child process to handle the communication between the parent and each child.
2. **Child Process**:
   * Each child process reads the message from the parent through the pipe.
   * Each child process then creates another pipe (pipefd\_response) for sending a response back to the parent.
   * The child writes the response to the parent through the new pipe and exits.
3. **Parent Process**:
   * The parent process writes the message to each child through the pipe.
   * After forking and sending the message to all children, the parent waits for each child to finish.
   * The parent then reads the responses from each child and prints them.