SAHYADRI
COLLEGE OF ENGINEERING & MANAGEMENT
An Autonomous Institution

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:

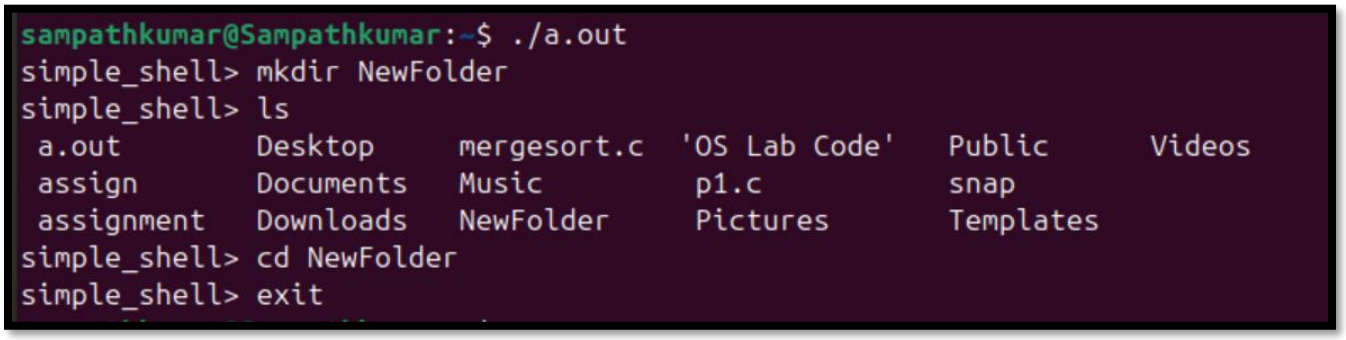
-> Write a simple shell program that can execute user commands.

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

```c
#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.

```c
#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:

```
sampathkumar@Sampathkumar:~$ ./a.out
FCFS Scheduling:
Average Waiting Time: 12.00
Average Turnaround Time: 19.25

SJF Scheduling:
Average Waiting Time: 8.75
Average Turnaround Time: 16.00

Priority Scheduling:
Average Waiting Time: 10.25
Average Turnaround Time: 17.50

Round Robin Scheduling (Quantum 3):
Average Waiting Time: 15.50
Average Turnaround Time: 22.75
```

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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
```

```c
#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:

```
sampathkumar@Sampathkumar:~$ ./a.out
Child 0 received message: Hello from parent
Child 1 received message: Hello from parent
Child 2 received message: Hello from parent
Parent received message: Hello from child 0
Parent received message: Hello from child 2
Parent received message: Hello from child 1
```

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.