

**OPERATING SYSTEMS PROJECT REPORT**

**REAL TIME PROCESS SCHEDULING**

**TEAM MEMBERS:**

K MANEESH RAM 23BCE5018

ABINAYAA M B 23BCE1913

E GOBI 23BCE1105

**SLOT: F1+TF1**

**ABSTRACT:**

Process scheduling is a fundamental concept in operating systems, where various scheduling algorithms determine the execution order of processes. This project implements and evaluates three scheduling algorithms: **First Come First Serve (FCFS)**, **Round Robin (RR)**, and **Shortest Job First (SJF)**. The program allows interactive process configuration, dynamically tracks process execution, and calculates key performance metrics like turnaround time and waiting time. The goal is to provide a simulation framework for understanding the efficiency and behavior of different scheduling strategies under various scenarios.

**INTRODUCTION**:

The CPU scheduling problem is a critical aspect of operating systems, directly influencing system performance and user experience. Effective scheduling minimizes process waiting time, optimizes CPU utilization, and ensures timely response. This project implements three fundamental algorithms:

1. **FCFS**: Processes execute sequentially based on arrival time, which is simple but can lead to the convoy effect.
2. **RR**: Allocates a fixed time quantum to processes, improving fairness and response for time-sharing systems.
3. **SJF**: Selects processes with the shortest burst time first, minimizing average turnaround time at the cost of complexity in burst time estimation.

The program focuses on simulating these algorithms, generating process metrics, and visualizing the scheduling process through Gantt charts. The use of system calls such as mmap, fork, and waitpid enables precise emulation of real-world scheduling scenarios.

**METHODOLOGY:**

##### **Tasks Performed**

**1. Interactive Process Setup:**

* Designed an interface for users to select process types and provide arrival and burst times.
* **Task types included File Write, Console Echo, Compute, and Database Write, reflecting common real-world workloads.**

**2. Process Sorting:**

* Implemented a sorting mechanism to arrange processes by arrival time for sequential execution.

**3. Implementation of Scheduling Algorithms:**

* **FCFS**:
  + Executed processes in order of arrival.
  + Turnaround time, waiting time, and completion time were calculated upon process termination.
* **RR**:
  + Introduced a time-sharing model using time quantum for fair CPU allocation.
  + Used preemption to cycle through processes until all were completed.
* **SJF**:
  + Dynamically selected the process with the shortest burst time.
  + Executed non-preemptively to minimize average turnaround time.

**4. Metrics Calculation and Visualization:**

* Computed key metrics:
  + **Turnaround Time**: Completion Time - Arrival Time
  + **Waiting Time**: Turnaround Time - Burst Time
* Displayed a Gantt chart to illustrate process execution order and durations.

**5. Use of System Calls and Shared Memory:**

* Integrated mmap for shared memory to manage real-time process states across parent and child processes.
* Leveraged fork for process creation and waitpid to monitor child processes.

##### **Execution Flow**

1. User Input: Number of processes, process types, arrival times, and burst times.
2. Algorithm Selection: FCFS, RR (with time quantum), or SJF.
3. Process Simulation: Processes executed based on selected scheduling strategy.
4. Metrics Reporting: Performance metrics and Gantt charts generated for analysis.

**SOURCE CODE:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/time.h>

#include <string.h>

#include <time.h>

#include <signal.h>

#include <sys/mman.h>

#include <sqlite3.h>

#define MAX\_PROCESSES 10

#define MS\_PER\_SECOND 1000

#define MAX\_GANTT\_WIDTH 100

*/\* Structure to track execution events for Gantt chart visualization*

*\* Records the task type, process name, and timing information*

*\* for each process execution segment*

*\*/*

typedef struct {

    char task\_type[20]; *// Type of task being executed*

    char process\_name[5]; *// Name of the process (e.g., P1, P2)*

    double start\_time; *// Start time of execution segment*

    double end\_time; *// End time of execution segment*

} ExecutionEvent;

*/\* Structure to maintain process information and state*

*\* Contains all necessary attributes for process scheduling and execution*

*\* including timing metrics and process identification*

*\*/*

typedef struct {

    pid\_t pid; *// Process ID*

    char process\_name[5]; *// Process name identifier*

    int arrival\_time; *// Time when process arrives*

    double burst\_time; *// Total CPU time needed*

    double remaining\_time; *// Remaining execution time*

    double completion\_time; *// Time when process completes*

    double waiting\_time; *// Total time spent waiting*

    double turnaround\_time; *// Total time in system*

    char task\_type[20]; *// Type of task*

    void (\*task\_function)(); *// Pointer to task implementation*

    int is\_active; *// Flag for process state*

    int first\_run; *// Flag for first execution*

} Process;

*/\* Structure for inter-process communication and control*

*\* Used to manage process execution and scheduling decisions*

*\* Shared between parent and child processes*

*\*/*

typedef struct {

    volatile sig\_atomic\_t should\_run; *// Control flag for process execution*

    volatile sig\_atomic\_t progress; *// Track task progress*

    volatile sig\_atomic\_t quantum; *// Time quantum for scheduling*

    int isPreemptive; *// Flag for preemptive scheduling*

} ProcessControl;

ProcessControl\* process\_control;

ExecutionEvent events[1000];

int event\_count = 0;

*/\* Records an execution event for Gantt chart visualization*

*\* Stores timing and process information for each execution segment*

*\*/*

void record\_event(const char\* task\_type, const char\* process\_name, double start\_time, double end\_time) {

    if (event\_count < 1000) {

        strcpy(events[event\_count].task\_type, task\_type);

        strcpy(events[event\_count].process\_name, process\_name);

        events[event\_count].start\_time = start\_time;

        events[event\_count].end\_time = end\_time;

        event\_count++;

    }

}

*/\* Generates and prints a Gantt chart visualization*

*\* Shows the execution timeline of all processes*

*\*/*

void print\_gantt\_chart() {

    if (event\_count == 0) return;

    printf("\nGantt Chart:\n\n");

    int \*block\_widths = (int \*)malloc(event\_count \* sizeof(int));

    for (int i = 0; i < event\_count; i++) {

        int time\_digits = snprintf(NULL, 0, "%.0f", events[i].end\_time);

        block\_widths[i] = strlen(events[i].process\_name) > time\_digits ?

                         strlen(events[i].process\_name) : time\_digits;

        block\_widths[i] += 4;

    }

    printf(" ");

    for (int i = 0; i < event\_count; i++) {

        for (int j = 0; j < block\_widths[i]; j++) printf("-");

        printf(" ");

    }

    printf("\n");

    printf("|");

    for (int i = 0; i < event\_count; i++) {

        int padding = (block\_widths[i] - strlen(events[i].process\_name)) / 2;

        int extra\_pad = (block\_widths[i] - strlen(events[i].process\_name)) % 2;

        for (int j = 0; j < padding; j++) printf(" ");

        printf("%s", events[i].process\_name);

        for (int j = 0; j < padding + extra\_pad; j++) printf(" ");

        printf("|");

    }

    printf("\n");

    printf(" ");

    for (int i = 0; i < event\_count; i++) {

        for (int j = 0; j < block\_widths[i]; j++) printf("-");

        printf(" ");

    }

    printf("\n");

    printf("0");

    for (int i = 0; i < event\_count; i++) {

        char time\_str[20];

        snprintf(time\_str, sizeof(time\_str), "%.0f", events[i].end\_time);

        int time\_digits = strlen(time\_str);

        int spaces = block\_widths[i] + 1 - time\_digits;

        for (int j = 0; j < spaces; j++) printf(" ");

        printf("%s", time\_str);

    }

    printf("\n");

    free(block\_widths);

}

*/\* Gets current system time in milliseconds*

*\* Used for timing and scheduling calculations*

*\*/*

double get\_current\_time() {

    struct timeval tv;

    gettimeofday(&tv, NULL);

    return tv.tv\_sec \* 1000.0 + tv.tv\_usec / 1000.0;

}

*/\* Implements milliexplain the second-precision sleep*

*\* Used for controlled process delays*

*\*/*

void sleep\_ms(int milliseconds) {

    usleep(milliseconds \* 1000);

}

*/\* Task implementation: File writing operation*

*\* Writes sequential lines to an output file*

*\*/*

void task\_file\_write() {

    FILE \*fp = fopen("output.txt", "a");

    if (fp != NULL) {

        struct timespec start, current;

        clock\_gettime(CLOCK\_MONOTONIC, &start);

        for(int i = process\_control->progress; i < 1000; i++) {

            clock\_gettime(CLOCK\_MONOTONIC, &current);

            double elapsed = (current.tv\_sec - start.tv\_sec) \* 1000.0 +

                           (current.tv\_nsec - start.tv\_nsec) / 1000000.0;

            if (elapsed >= process\_control->quantum) {

                process\_control->progress = i;

                fclose(fp);

                raise(SIGSTOP);

                fp = fopen("output.txt", "a");

                if (fp == NULL) break;

                clock\_gettime(CLOCK\_MONOTONIC, &start);

            }

            fprintf(fp, "Process %d writing line %d\n", getpid(), i);

            fflush(fp);

            usleep(1000);

        }

        fclose(fp);

    }

}

*/\* Task implementation: Console output operation*

*\* Prints sequential lines to console*

*\*/*

void task\_console\_echo() {

    int total\_lines = 100;

    struct timespec start, current;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    for(int i = process\_control->progress; i < total\_lines; i++) {

        clock\_gettime(CLOCK\_MONOTONIC, &current);

        double elapsed = (current.tv\_sec - start.tv\_sec) \* 1000.0 +

                        (current.tv\_nsec - start.tv\_nsec) / 1000000.0;

        if (elapsed >= process\_control->quantum) {

            process\_control->progress = i;

            raise(SIGSTOP);

            clock\_gettime(CLOCK\_MONOTONIC, &start);

        }

        printf("Process %d echoing line %d\n", getpid(), i);

        fflush(stdout);

        usleep(10000);

    }

    process\_control->progress = 0;

}

*/\* Task implementation: CPU-bound computation*

*\* Performs intensive mathematical calculations*

*\*/*

void task\_compute() {

    struct timespec start, current;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    long total\_iterations = process\_control->quantum == \_\_DBL\_MAX\_\_ ? 50000000 : 500000000;

    long long sum = 0;

    volatile long i = process\_control->progress;

    for (long j = 0; j < i; j++) {

        sum += j;

    }

    for (; i < total\_iterations; i++) {

        sum += i;

        if (i % 1000 == 0) {

            clock\_gettime(CLOCK\_MONOTONIC, &current);

            double elapsed = (current.tv\_sec - start.tv\_sec) \* 1000.0 +

                           (current.tv\_nsec - start.tv\_nsec) / 1000000.0;

            if (elapsed >= process\_control->quantum) {

                process\_control->progress = i + 1;

                printf("Process %d suspended at iteration %ld, current sum: %lld\n",

                       getpid(), i, sum);

                fflush(stdout);

                if (!process\_control->isPreemptive){

                    exit(0);

                    break;

                }

                raise(SIGSTOP);

                clock\_gettime(CLOCK\_MONOTONIC, &start);

            }

        }

        if (i % 10000000 == 0) {

            printf("Process %d computed sum up to %ld: %lld\n", getpid(), i, sum);

            fflush(stdout);

        }

    }

    printf("Process %d completed computation. Final sum: %lld\n", getpid(), sum);

    fflush(stdout);

}

*/\* Task implementation: Database write operation*

*\* Performs sequential database record insertions*

*\*/*

void task\_db\_write() {

    int total\_records = 90;

    struct timespec start, current;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    for(int i = process\_control->progress; i < total\_records; i++) {

        clock\_gettime(CLOCK\_MONOTONIC, &current);

        double elapsed = (current.tv\_sec - start.tv\_sec) \* 1000.0 +

                        (current.tv\_nsec - start.tv\_nsec) / 1000000.0;

        if (elapsed >= process\_control->quantum) {

            process\_control->progress = i;

            raise(SIGSTOP);

            clock\_gettime(CLOCK\_MONOTONIC, &start);

        }

        printf("Process %d adding Record %d\n", getpid(), i);

    sqlite3 \*db;

    char \*err\_msg = 0;

    int rc;

    pid\_t pid;

    rc = sqlite3\_open("os\_project.db", &db);

    if (rc != SQLITE\_OK) {

        fprintf(stderr, "Cannot open database: %s\n", sqlite3\_errmsg(db));

        sqlite3\_close(db);

        return;

    }

    const char \*create\_table\_sql =

        "CREATE TABLE IF NOT EXISTS student ("

        "id INTEGER, "

        "name TEXT, "

        "age INTEGER);";

    rc = sqlite3\_exec(db, create\_table\_sql, 0, 0, &err\_msg);

    if (rc != SQLITE\_OK) {

        fprintf(stderr, "SQL error: %s\n", err\_msg);

        sqlite3\_free(err\_msg);

        sqlite3\_close(db);

        return;

    }

    srand(time(NULL));

    pid = getpid();

    sqlite3\_stmt \*stmt;

    const char \*insert\_sql = "INSERT INTO student (id, name, age) VALUES (?, ?, ?);";

    rc = sqlite3\_prepare\_v2(db, insert\_sql, -1, &stmt, 0);

    if (rc != SQLITE\_OK) {

        fprintf(stderr, "Preparation failed: %s\n", sqlite3\_errmsg(db));

        sqlite3\_close(db);

        return;

    }

    char name[50];

    snprintf(name, sizeof(name), "name\_%d", pid);

    int age = 10 + rand() % 11;

    sqlite3\_bind\_int(stmt, 1, i);

    sqlite3\_bind\_text(stmt, 2, name, -1, SQLITE\_STATIC);

    sqlite3\_bind\_int(stmt, 3, age);

    rc = sqlite3\_step(stmt);

    if (rc != SQLITE\_DONE) {

        fprintf(stderr, "Insertion failed: %s\n", sqlite3\_errmsg(db));

    }

    sqlite3\_finalize(stmt);

    sqlite3\_close(db);

        usleep(10000);

    }

    process\_control->progress = 0;

}

*/\* Measures actual burst time for a given task*

*\* Creates test process and measures execution time*

*\*/*

double measure\_burst\_time(void (\*task\_function)()) {

    double start\_time = get\_current\_time();

    pid\_t pid = fork();

    if (pid == 0) {

        task\_function();

        exit(0);

    } else if (pid > 0) {

        int status;

        waitpid(pid, &status, 0);

        double end\_time = get\_current\_time();

        return (end\_time - start\_time);

    }

    return 0;

}

*/\* Initializes a process structure with given parameters*

*\* Sets up process attributes and measures burst time*

*\*/*

void initialize\_process(Process\* p, void (\*task\_function)(), const char\* task\_name,

                       int arrival\_time\_ms, int process\_num) {

    p->task\_function = task\_function;

    strcpy(p->task\_type, task\_name);

    sprintf(p->process\_name, "P%d", process\_num);

    p->arrival\_time = arrival\_time\_ms;

    printf("Measuring burst time for %s (%s)...\n", p->process\_name, task\_name);

    p->burst\_time = measure\_burst\_time(task\_function);

    p->remaining\_time = p->burst\_time;

    p->completion\_time = 0;

    p->waiting\_time = 0;

    p->turnaround\_time = 0;

    p->pid = 0;

    p->is\_active = 0;

    p->first\_run = 1;

    printf("Measured burst time for %s (%s): %.2f ms\n", p->process\_name, task\_name, p->burst\_time);

}

*/\* Creates a new process using fork()*

*\* Initializes child process for task execution*

*\*/*

void create\_process(Process\* p) {

    pid\_t pid = fork();

    if (pid == 0) {

        p->task\_function();

        exit(0);

    } else if (pid > 0) {

        p->pid = pid;

    }

}

*/\* Implements First-Come-First-Serve scheduling algorithm*

*\* Executes processes in order of arrival*

*\*/*

void fcfs(Process processes[], int n) {

    printf("\nExecuting FCFS Scheduling...\n");

    event\_count = 0;

    process\_control = mmap(NULL, sizeof(ProcessControl),

                         PROT\_READ | PROT\_WRITE,

                         MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

    if (process\_control == MAP\_FAILED) {

        perror("mmap failed");

        return;

    }

    double current\_time = 0;

    for (int i = 0; i < n; i++) {

        if (current\_time < processes[i].arrival\_time) {

            sleep\_ms(processes[i].arrival\_time - current\_time);

            current\_time = processes[i].arrival\_time;

        }

        if (processes[i].first\_run) {

            processes[i].first\_run = 0;

        }

        printf("Starting %s at time %.2f ms\n", processes[i].process\_name, current\_time);

        process\_control->should\_run = 1;

        process\_control->progress = 0;

        process\_control->quantum = processes[i].burst\_time;

        process\_control->isPreemptive=0;

        double start\_time = current\_time;

        create\_process(&processes[i]);

        int status;

        struct timespec start\_ts, current\_ts;

        clock\_gettime(CLOCK\_MONOTONIC, &start\_ts);

        while (1) {

            pid\_t result = waitpid(processes[i].pid, &status, WNOHANG);

            if (result > 0) break;

            clock\_gettime(CLOCK\_MONOTONIC, &current\_ts);

            usleep(1000);

        }

        clock\_gettime(CLOCK\_MONOTONIC, &current\_ts);

        double elapsed = (current\_ts.tv\_sec - start\_ts.tv\_sec) \* 1000.0 +

                        (current\_ts.tv\_nsec - start\_ts.tv\_nsec) / 1000000.0;

        current\_time += elapsed;

        record\_event(processes[i].task\_type, processes[i].process\_name, start\_time, current\_time);

        processes[i].completion\_time = current\_time;

        processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

        processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

        printf("Completed %s at time %.2f ms\n", processes[i].process\_name, current\_time);

    }

    munmap(process\_control, sizeof(ProcessControl));

}

*/\* Implements Round Robin scheduling algorithm*

*\* Executes processes with time quantum-based preemption*

*\*/*

void round\_robin(Process processes[], int n, int time\_quantum\_ms) {

    printf("\nExecuting Round Robin Scheduling (Time Quantum: %d ms)...\n", time\_quantum\_ms);

    event\_count = 0;

    process\_control = mmap(NULL, sizeof(ProcessControl),

                         PROT\_READ | PROT\_WRITE,

                         MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

    if (process\_control == MAP\_FAILED) {

        perror("mmap failed");

        return;

    }

    process\_control->should\_run = 1;

    process\_control->quantum = time\_quantum\_ms;

    process\_control->isPreemptive=1;

    double current\_time = 0;

    int completed = 0;

    int\* terminated = (int\*)calloc(n, sizeof(int));

    double\* first\_execution\_time = (double\*)calloc(n, sizeof(double));

    for (int i = 0; i < n; i++) {

        first\_execution\_time[i] = -1;

    }

    struct sigaction sa;

    sa.sa\_handler = SIG\_IGN;

    sigemptyset(&sa.sa\_mask);

    sa.sa\_flags = 0;

    sigaction(SIGSTOP, &sa, NULL);

    while (completed < n) {

        int work\_done = 0;

        for (int i = 0; i < n; i++) {

            if (terminated[i] || processes[i].arrival\_time > current\_time) {

                continue;

            }

            if (processes[i].remaining\_time <= 0) {

                if (!terminated[i]) {

                    terminated[i] = 1;

                    completed++;

                    processes[i].completion\_time = current\_time;

                }

                continue;

            }

            work\_done = 1;

            if (first\_execution\_time[i] == -1) {

                first\_execution\_time[i] = current\_time;

            }

            if (processes[i].pid == 0) {

                if (processes[i].first\_run) {

                    processes[i].first\_run = 0;

                    process\_control->progress = 0;

                }

                pid\_t pid = fork();

                if (pid == 0) {

                    struct sigaction child\_sa;

                    child\_sa.sa\_handler = SIG\_DFL;

                    sigemptyset(&child\_sa.sa\_mask);

                    child\_sa.sa\_flags = 0;

                    sigaction(SIGSTOP, &child\_sa, NULL);

                    processes[i].task\_function();

                    exit(0);

                } else if (pid > 0) {

                    processes[i].pid = pid;

                }

            }

            double exec\_time = (processes[i].remaining\_time > time\_quantum\_ms) ?

                              time\_quantum\_ms : processes[i].remaining\_time;

            printf("Executing %s for %.2f ms at %.2f (Progress: %d)\n",

                   processes[i].process\_name, exec\_time, current\_time,

                   process\_control->progress);

            double start\_time = current\_time;

            struct timespec start, now;

            clock\_gettime(CLOCK\_MONOTONIC, &start);

            kill(processes[i].pid, SIGCONT);

            int process\_stopped = 0;

            while (!process\_stopped) {

                clock\_gettime(CLOCK\_MONOTONIC, &now);

                double elapsed = (now.tv\_sec - start.tv\_sec) \* 1000.0 +

                               (now.tv\_nsec - start.tv\_nsec) / 1000000.0;

                if (elapsed >= exec\_time) {

                    kill(processes[i].pid, SIGSTOP);

                    process\_stopped = 1;

                }

                int status;

                pid\_t result = waitpid(processes[i].pid, &status, WNOHANG | WUNTRACED);

                if (result > 0) {

                    if (WIFSTOPPED(status)) {

                        process\_stopped = 1;

                        elapsed = elapsed > exec\_time ? exec\_time : elapsed;

                    } else if (WIFEXITED(status)) {

                        process\_stopped = 1;

                        processes[i].remaining\_time = 0;

                        terminated[i] = 1;

                        completed++;

                        processes[i].completion\_time = current\_time + elapsed;

                    }

                }

                if (!process\_stopped) {

                    usleep(1000);

                }

            }

            double actual\_exec\_time = (now.tv\_sec - start.tv\_sec) \* 1000.0 +

                                    (now.tv\_nsec - start.tv\_nsec) / 1000000.0;

            actual\_exec\_time = actual\_exec\_time > exec\_time ? exec\_time : actual\_exec\_time;

            current\_time += actual\_exec\_time;

            processes[i].remaining\_time -= actual\_exec\_time;

            record\_event(processes[i].task\_type, processes[i].process\_name,

                        start\_time, current\_time);

        }

        if (!work\_done) {

            double next\_arrival = \_\_DBL\_MAX\_\_;

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

                if (!terminated[i] && processes[i].arrival\_time > current\_time) {

                    next\_arrival = (next\_arrival < processes[i].arrival\_time) ?

                                  next\_arrival : processes[i].arrival\_time;

                }

            }

            if (next\_arrival != \_\_DBL\_MAX\_\_) {

                current\_time = next\_arrival;

            }

        }

    }

    for (int i = 0; i < n; i++) {

        processes[i].turnaround\_time = processes[i].completion\_time - processes[i].arrival\_time;

        processes[i].waiting\_time = processes[i].turnaround\_time - processes[i].burst\_time;

    }

    free(first\_execution\_time);

    free(terminated);

    munmap(process\_control, sizeof(ProcessControl));

}

*/\* Implements Shortest Job First scheduling algorithm*

*\* Executes processes ordered by burst time*

*\*/*

void sjf(Process processes[], int n) {

    printf("\nExecuting Shortest Job First Scheduling...\n");

    event\_count = 0;

    process\_control = mmap(NULL, sizeof(ProcessControl),

                         PROT\_READ | PROT\_WRITE,

                         MAP\_SHARED | MAP\_ANONYMOUS, -1, 0);

    if (process\_control == MAP\_FAILED) {

        perror("mmap failed");

        return;

    }

    double current\_time = 0;

    int completed = 0;

    int\* completed\_processes = (int\*)calloc(n, sizeof(int));

    while (completed < n) {

        int shortest = -1;

        double shortest\_burst = \_\_DBL\_MAX\_\_;

        for (int i = 0; i < n; i++) {

            if (!completed\_processes[i] &&

                processes[i].arrival\_time <= current\_time &&

                processes[i].burst\_time < shortest\_burst) {

                shortest\_burst = processes[i].burst\_time;

                shortest = i;

            }

        }

        if (shortest == -1) {

            double next\_arrival = \_\_DBL\_MAX\_\_;

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

                if (!completed\_processes[i] && processes[i].arrival\_time > current\_time) {

                    if (processes[i].arrival\_time < next\_arrival) {

                        next\_arrival = processes[i].arrival\_time;

                    }

                }

            }

            if (next\_arrival != \_\_DBL\_MAX\_\_) {

                sleep\_ms(next\_arrival - current\_time);

                current\_time = next\_arrival;

            }

            continue;

        }

        printf("Starting %s at time %.2f ms\n",

               processes[shortest].process\_name, current\_time);

        if (processes[shortest].first\_run) {

            processes[shortest].first\_run = 0;

        }

        process\_control->should\_run = 1;

        process\_control->progress = 0;

        process\_control->quantum = processes[shortest].burst\_time;

        process\_control->isPreemptive=0;

        double start\_time = current\_time;

        create\_process(&processes[shortest]);

        int status;

        struct timespec start\_ts, current\_ts;

        clock\_gettime(CLOCK\_MONOTONIC, &start\_ts);

        while (1) {

            pid\_t result = waitpid(processes[shortest].pid, &status, WNOHANG);

            if (result > 0) break;

            clock\_gettime(CLOCK\_MONOTONIC, &current\_ts);

            usleep(1000);

        }

        clock\_gettime(CLOCK\_MONOTONIC, &current\_ts);

        double elapsed = (current\_ts.tv\_sec - start\_ts.tv\_sec) \* 1000.0 +

                        (current\_ts.tv\_nsec - start\_ts.tv\_nsec) / 1000000.0;

        current\_time += elapsed;

        record\_event(processes[shortest].task\_type,

                    processes[shortest].process\_name,

                    start\_time, current\_time);

        processes[shortest].completion\_time = current\_time;

        processes[shortest].turnaround\_time = processes[shortest].completion\_time -

                                            processes[shortest].arrival\_time;

        processes[shortest].waiting\_time = processes[shortest].turnaround\_time -

                                         processes[shortest].burst\_time;

        completed\_processes[shortest] = 1;

        completed++;

        printf("Completed %s at time %.2f ms\n",

               processes[shortest].process\_name, current\_time);

    }

    free(completed\_processes);

    munmap(process\_control, sizeof(ProcessControl));

}

*/\* Prints detailed statistics for all processes*

*\* Shows timing metrics and generates Gantt chart*

*\*/*

void print\_stats(Process processes[], int n) {

    printf("\nProcess Statistics:\n");

    printf("%-8s %-8s %-12s %-12s %-12s %-12s %-12s %-12s\n",

           "Process", "PID", "Task", "Arrival", "Burst", "Completion", "Turnaround", "Waiting");

    printf("---------------------------------------------------------------------------------------\n");

    double avg\_turnaround = 0, avg\_waiting = 0;

    for (int i = 0; i < n; i++) {

        if (processes[i].waiting\_time<0){

            processes[i].waiting\_time=0;

        }

        printf("%-8s %-8d %-12s %-12d %-12.2f %-12.2f %-12.2f %-12.2f\n",

               processes[i].process\_name,

               processes[i].pid,

               processes[i].task\_type,

               processes[i].arrival\_time,

               processes[i].burst\_time,

               processes[i].completion\_time,

               processes[i].turnaround\_time,

               processes[i].waiting\_time);

        avg\_turnaround += processes[i].turnaround\_time;

        avg\_waiting += processes[i].waiting\_time;

    }

    printf("\nAverage Metrics:\n");

    printf("Turnaround Time: %.2f ms\n", avg\_turnaround/n);

    printf("Waiting Time: %.2f ms\n", avg\_waiting/n);

    print\_gantt\_chart();

}

*/\* Handles process type selection and initialization*

*\* Provides user interface for process configuration*

*\*/*

void select\_process(Process\* p, int index) {

    printf("\nAvailable process types for P%d:\n", index + 1);

    printf("1. File Write\n");

    printf("2. Console Echo\n");

    printf("3. Compute\n");

    printf("4. Add record to database\n");

    int choice;

    printf("Select process type (1-4): ");

    scanf("%d", &choice);

    int arrival\_time\_ms;

    printf("Enter arrival time (in milliseconds): ");

    scanf("%d", &arrival\_time\_ms);

    switch(choice) {

        case 1:

            initialize\_process(p, task\_file\_write, "file\_write", arrival\_time\_ms, index + 1);

            break;

        case 2:

            initialize\_process(p, task\_console\_echo, "console\_echo", arrival\_time\_ms, index + 1);

            break;

        case 3:

            initialize\_process(p, task\_compute, "compute", arrival\_time\_ms, index + 1);

            break;

       case 4:

            initialize\_process(p, task\_db\_write, "db\_write", arrival\_time\_ms, index + 1);

            break;

        default:

            printf("Invalid choice! Defaulting to compute process.\n");

            initialize\_process(p, task\_compute, "compute", arrival\_time\_ms, index + 1);

    }

}

*/\* Sorts processes array by arrival time*

*\* Used to order processes for scheduling*

*\*/*

void sort\_processes\_by\_arrival\_time(Process processes[], int n) {

    for (int i = 0; i < n - 1; i++) {

        for (int j = 0; j < n - i - 1; j++) {

            if (processes[j].arrival\_time > processes[j+1].arrival\_time) {

                Process temp = processes[j];

                processes[j] = processes[j+1];

                processes[j+1] = temp;

            }

        }

    }

}

int main() {

    int n;

    printf("Enter number of processes (max %d): ", MAX\_PROCESSES);

    scanf("%d", &n);

    if(n <= 0 || n > MAX\_PROCESSES) {

        printf("Invalid number of processes!\n");

        return 1;

    }

    Process processes[MAX\_PROCESSES];

    printf("\nSelect process types and arrival times:\n");

    for(int i = 0; i < n; i++) {

        select\_process(&processes[i], i);

    }

    printf("\nInitial Process Set:\n");

    printf("Process\tTask Type\tArrival Time (ms)\tMeasured Burst Time (ms)\n");

    for(int i = 0; i < n; i++) {

        printf("%s\t%s\t\t%d\t\t\t%.2f\n",

               processes[i].process\_name,

               processes[i].task\_type,

               processes[i].arrival\_time,

               processes[i].burst\_time);

    }

    printf("\nChoose scheduling algorithm:\n");

    printf("1. First Come First Serve (FCFS)\n");

    printf("2. Round Robin (RR)\n");

    printf("3. Shortest Job First (SJF)\n");

    int choice;

    scanf("%d", &choice);

    switch(choice) {

        case 1: {

            Process fcfs\_processes[MAX\_PROCESSES];

            memcpy(fcfs\_processes, processes, sizeof(Process) \* n);

            sort\_processes\_by\_arrival\_time(fcfs\_processes, n);

            fcfs(fcfs\_processes, n);

            print\_stats(fcfs\_processes, n);

            break;

        }

        case 2: {

            int time\_quantum\_ms;

            int valid=0;

            while (!valid){

            printf("Enter time quantum (in milliseconds): ");

            scanf("%d", &time\_quantum\_ms);

            if (time\_quantum\_ms>0)

            valid=1;

            else

            printf("Invalid time quantum.\n");

            }

            Process rr\_processes[MAX\_PROCESSES];

            memcpy(rr\_processes, processes, sizeof(Process) \* n);

            sort\_processes\_by\_arrival\_time(rr\_processes, n);

            round\_robin(rr\_processes, n, time\_quantum\_ms);

            print\_stats(rr\_processes, n);

            break;

        }

        case 3: {

            Process sjf\_processes[MAX\_PROCESSES];

            memcpy(sjf\_processes, processes, sizeof(Process) \* n);

            sort\_processes\_by\_arrival\_time(sjf\_processes, n);

            sjf(sjf\_processes, n);

            print\_stats(sjf\_processes, n);

            break;

        }

        default:

            printf("Invalid choice!\n");

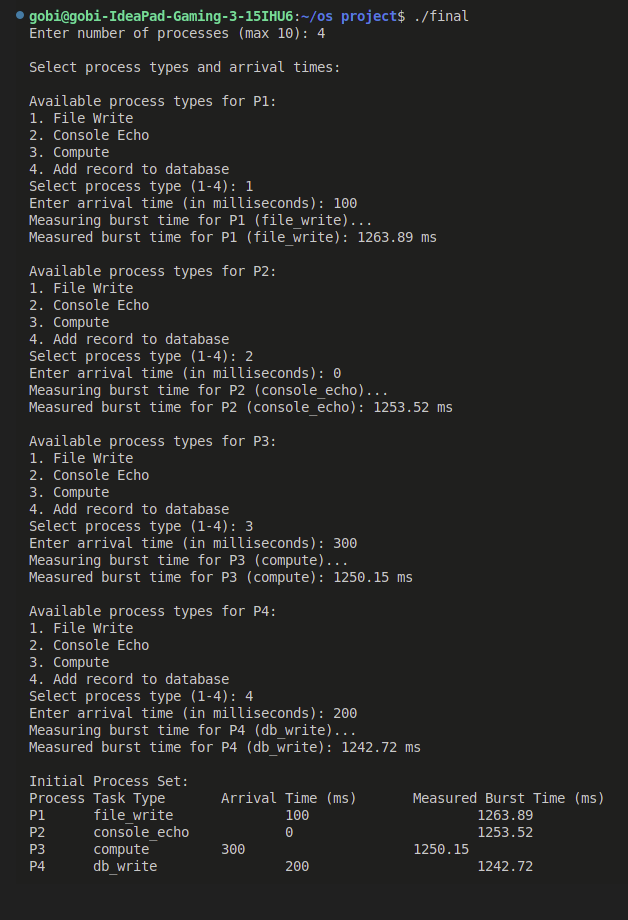
            return 1;

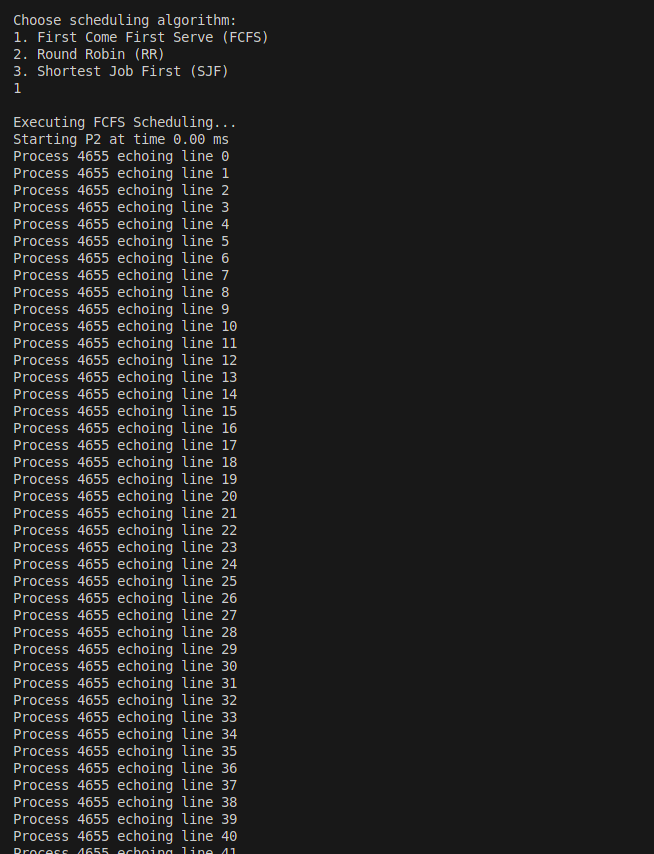
    }

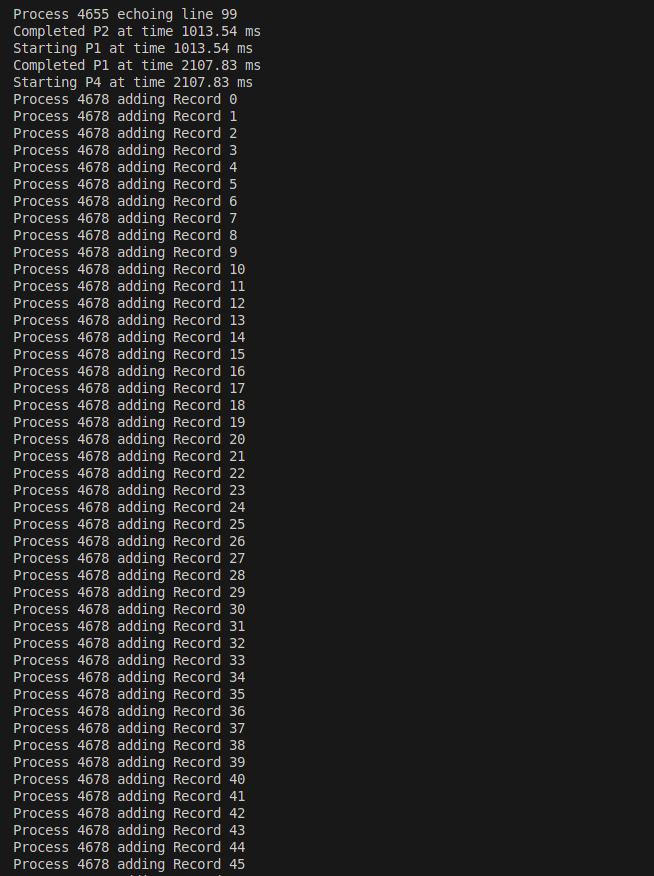
    return 0;

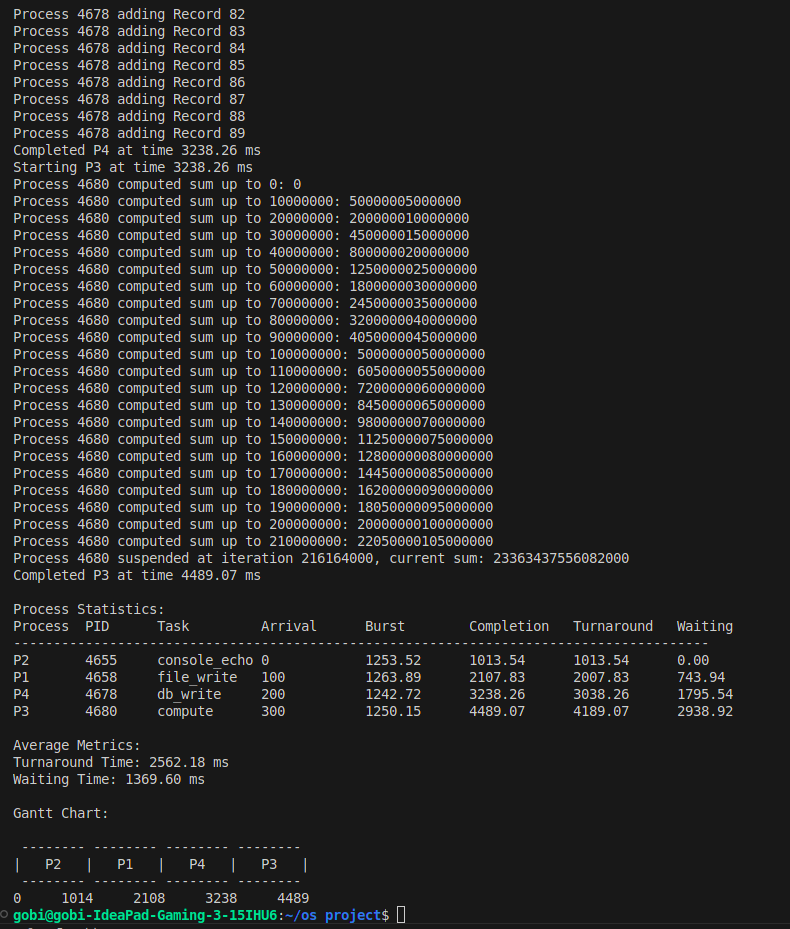
}

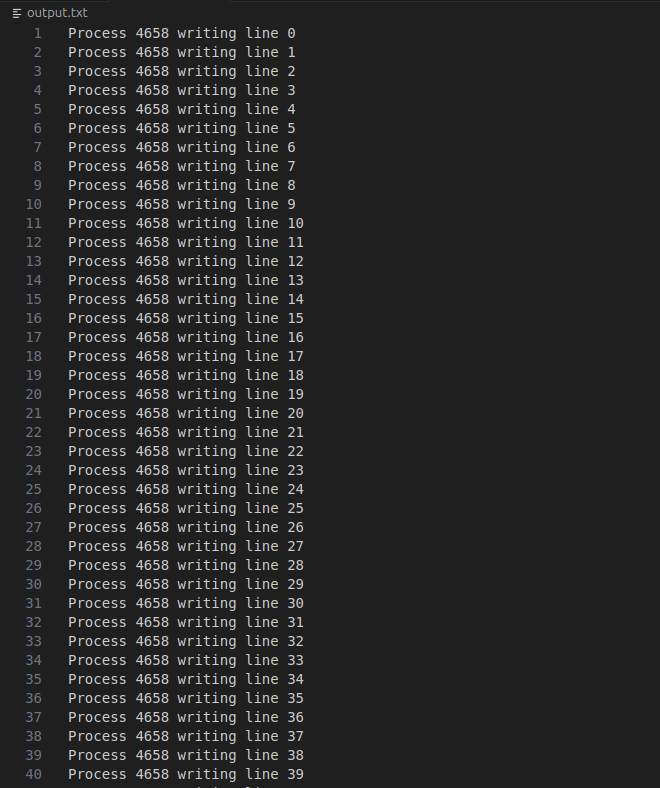
**IMPLEMENTATION:**

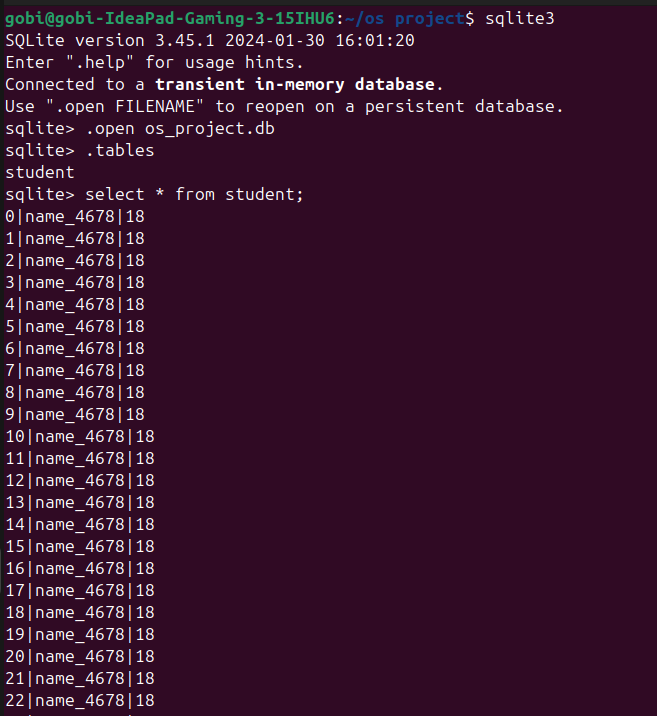
****

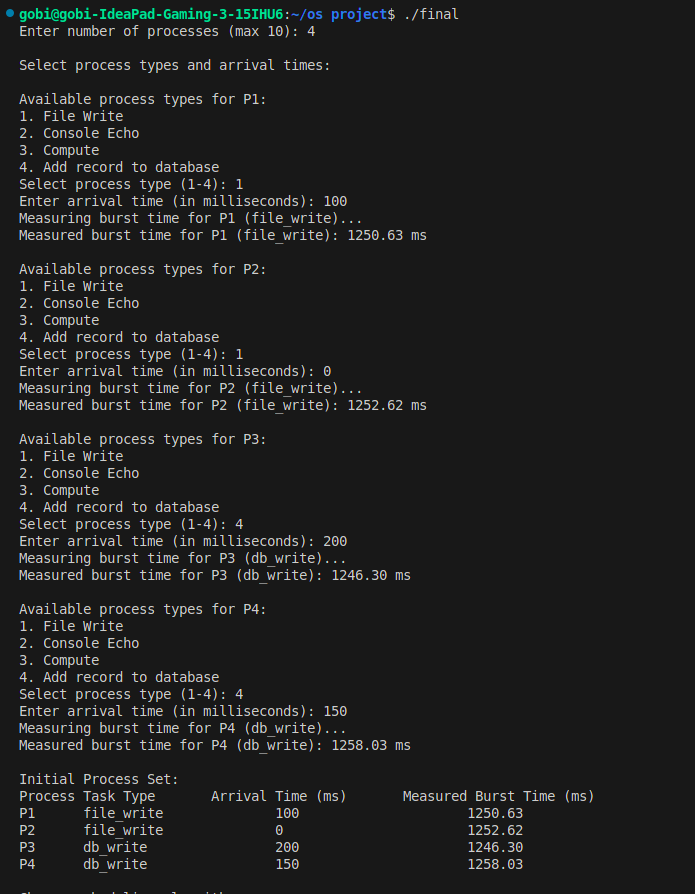


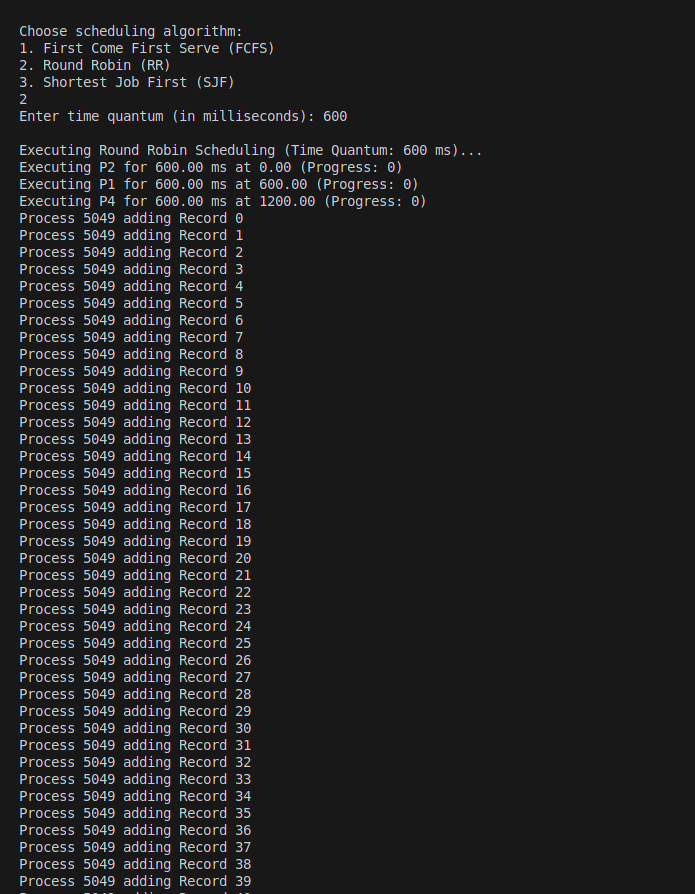


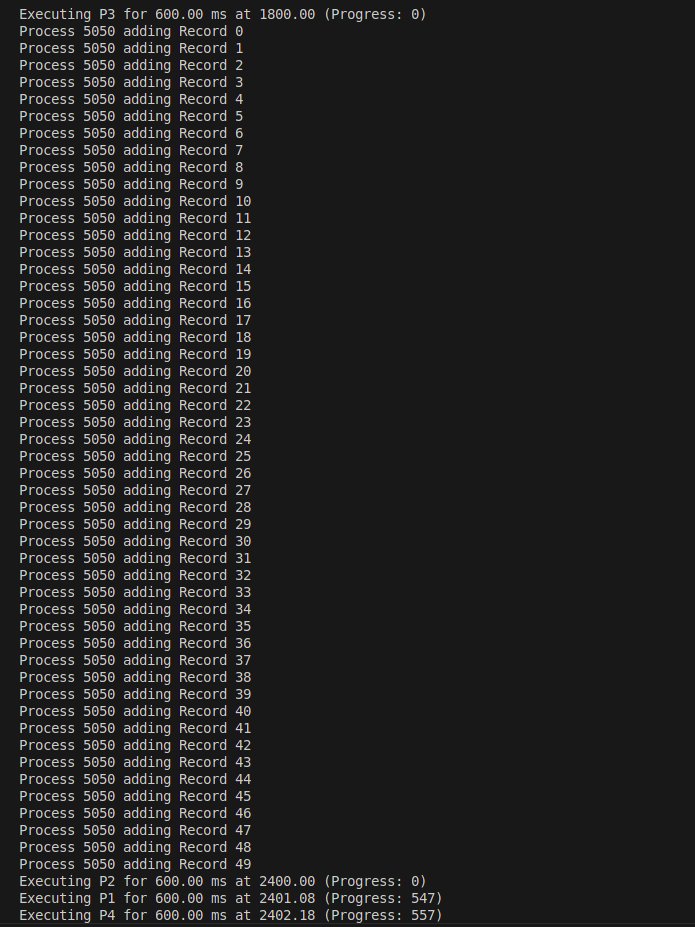


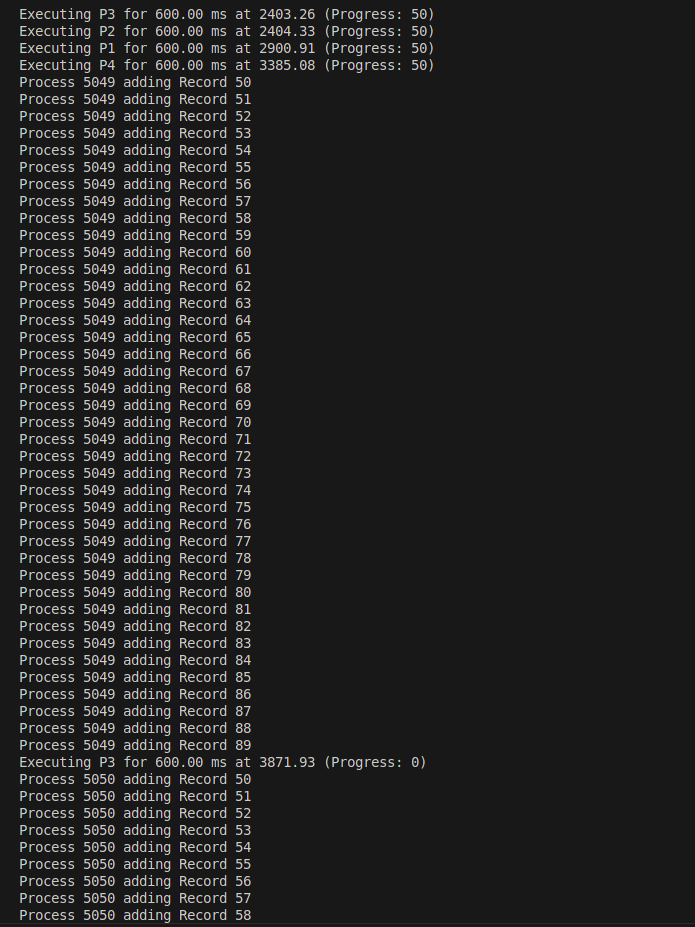


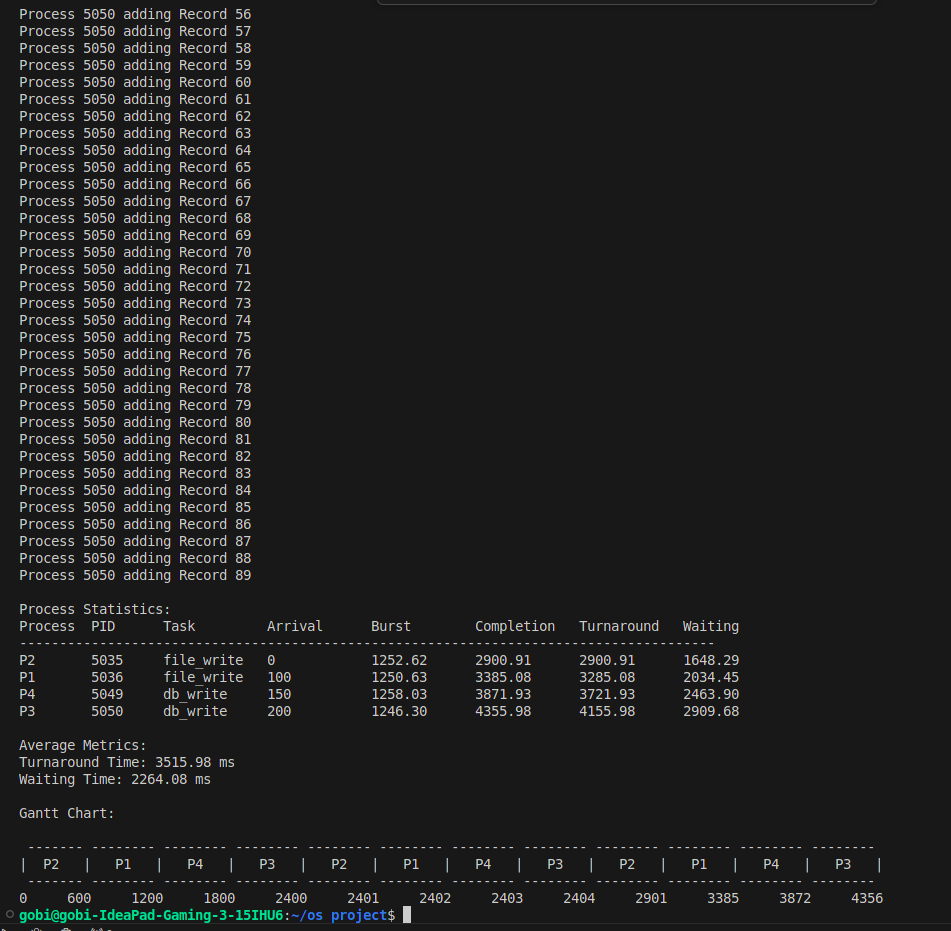


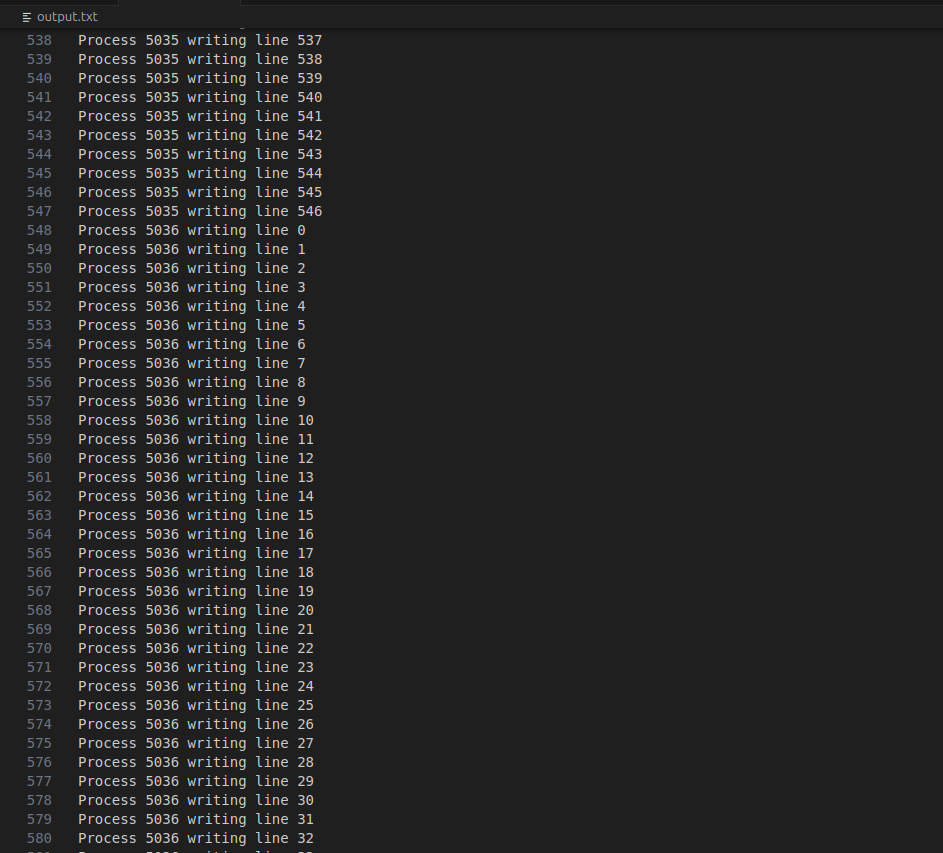


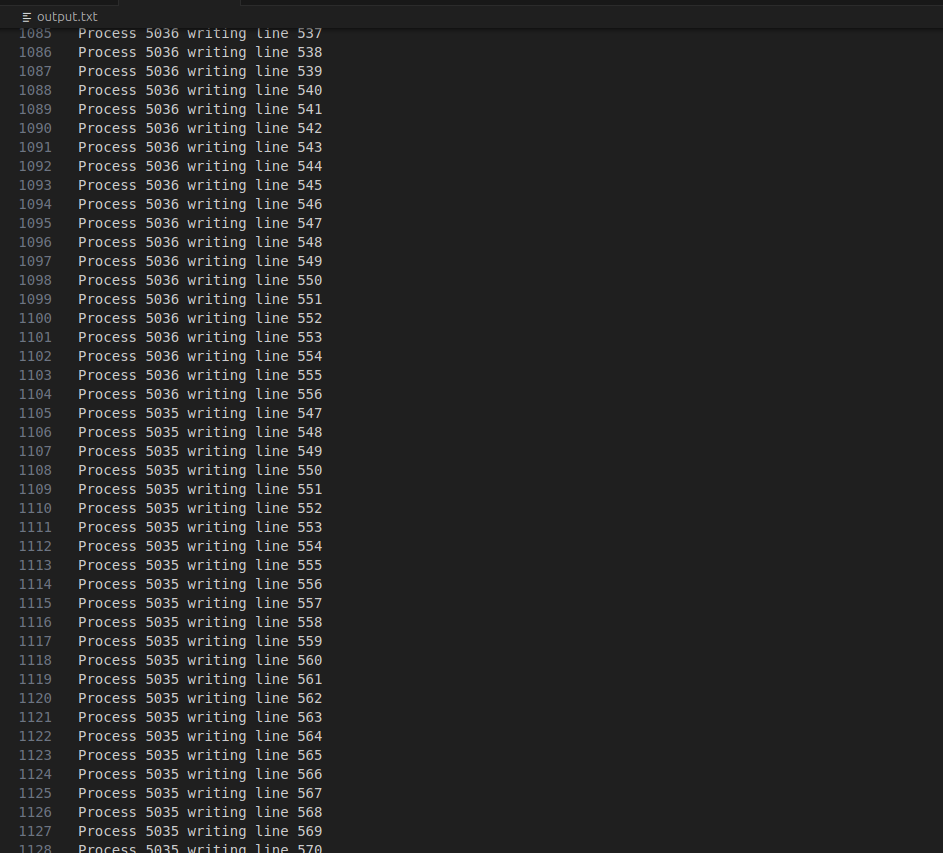


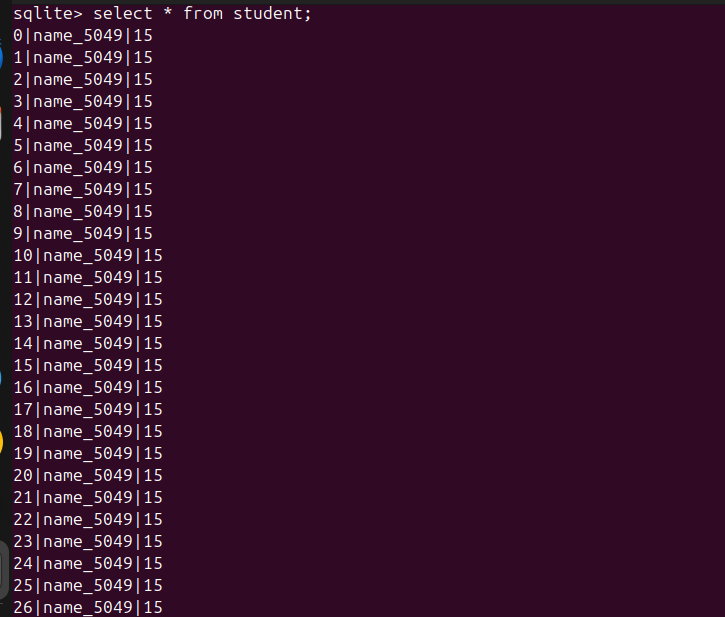


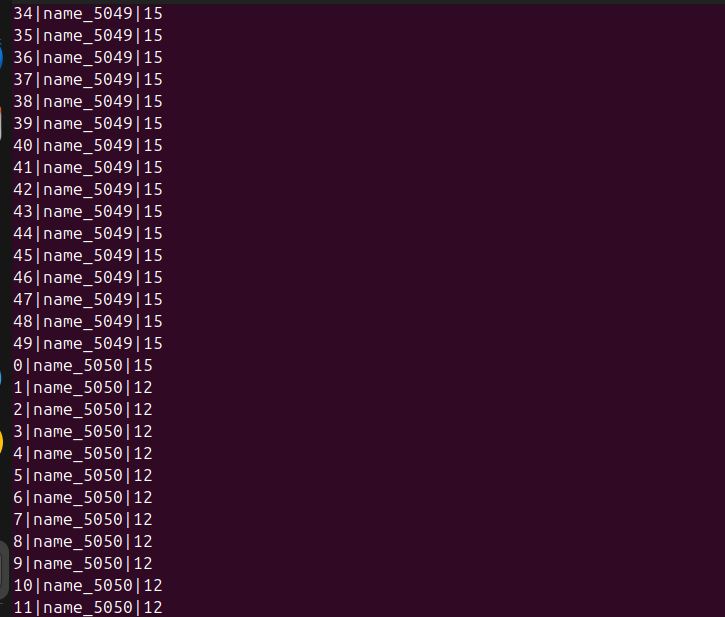


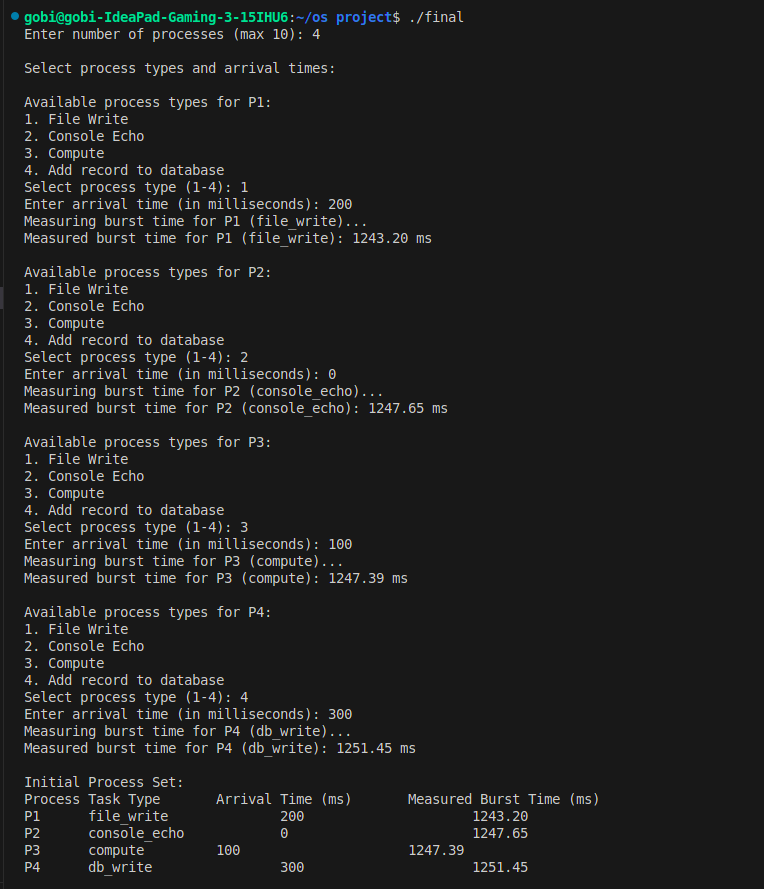


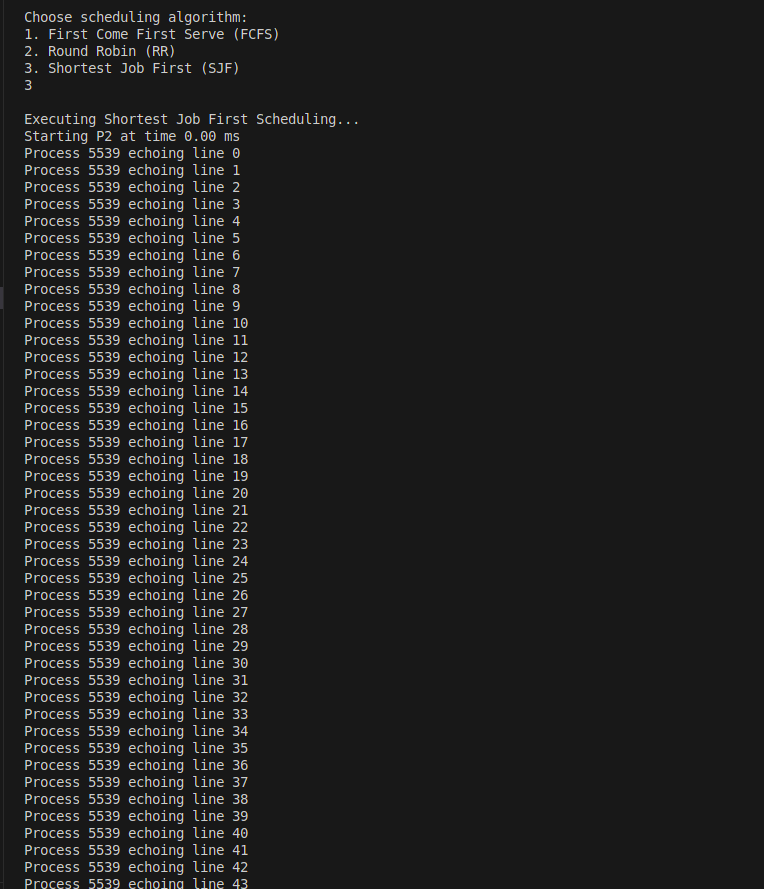


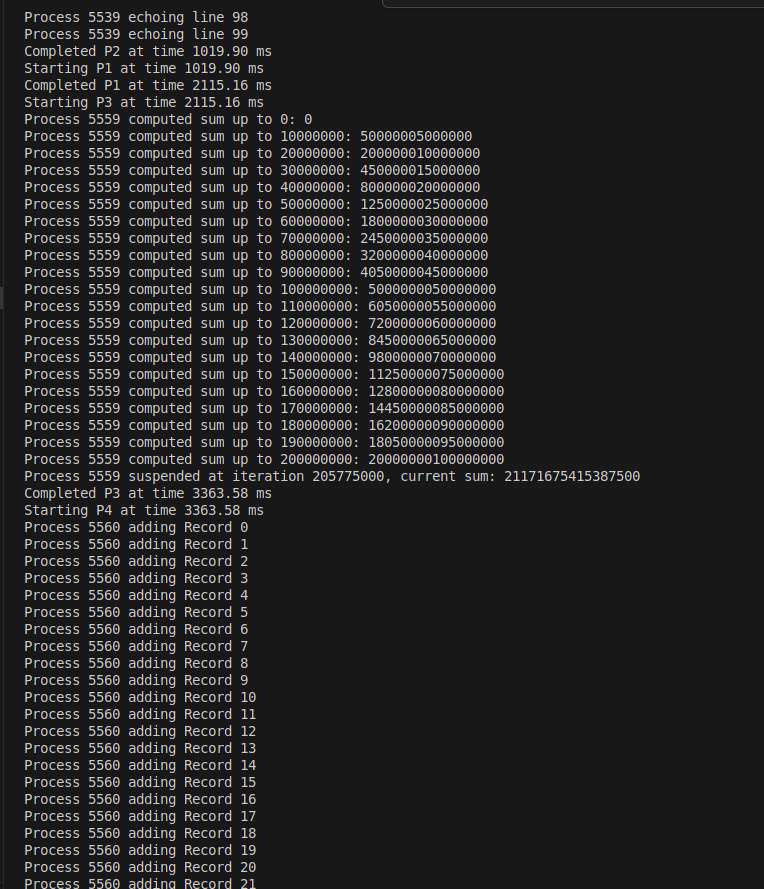


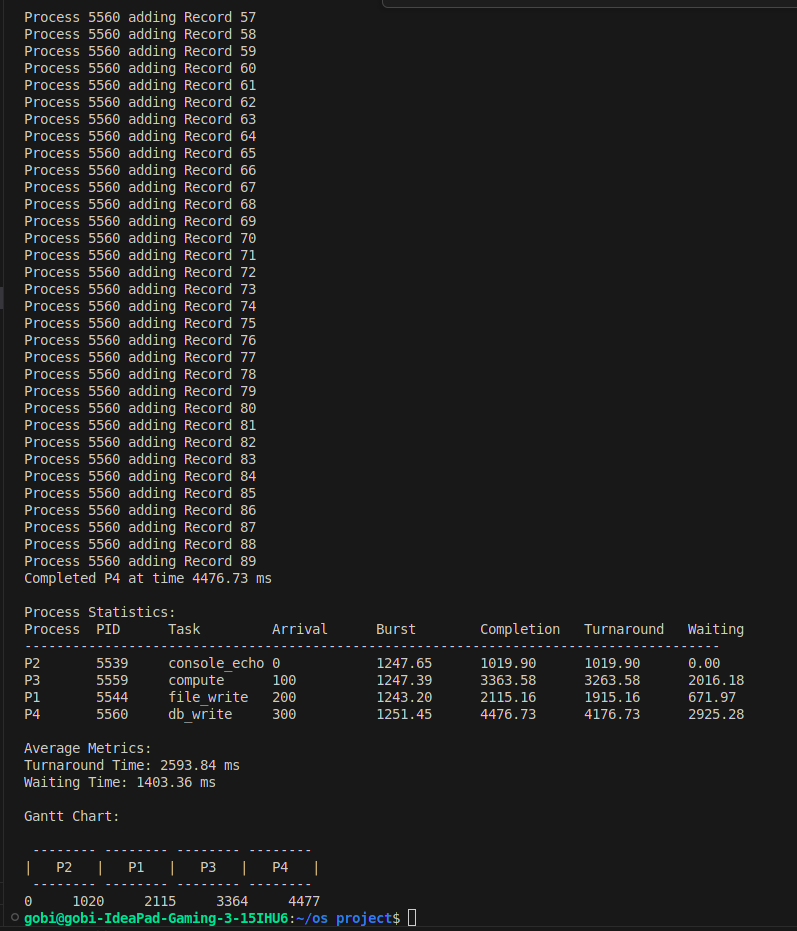


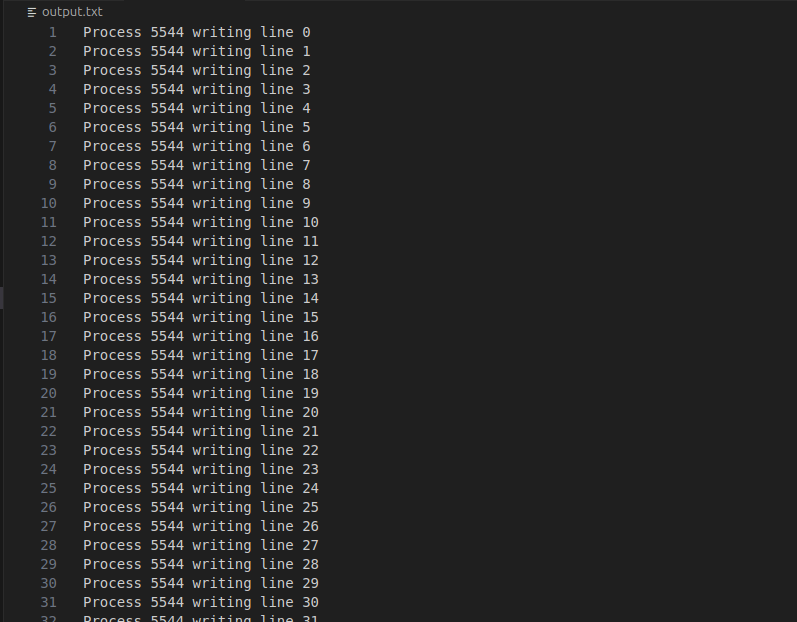


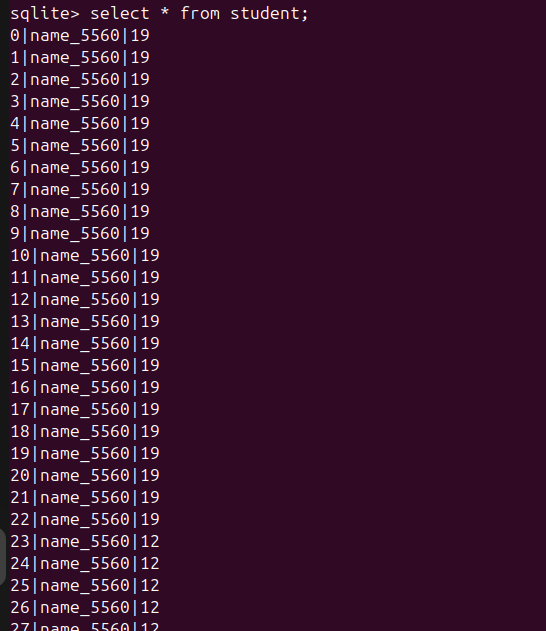












**CONCLUSION:**

This project successfully implements and evaluates three key process scheduling algorithms. The modular design and interactive interface make it an effective tool for analyzing scheduling behavior under various scenarios. **FCFS** demonstrates simplicity but inefficiency for varied workloads. **RR** balances fairness and responsiveness, making it ideal for time-sharing systems. **SJF** excels in optimizing turnaround time but requires precise burst time knowledge. The inclusion of real-world tasks and metrics visualization enhances the practical value of this simulation framework.

##### **Key Achievements:**

* Simulated real-world tasks with system-level APIs (mmap, fork, waitpid).
* Implemented three scheduling algorithms with detailed performance analysis.
* Developed metrics and visualization tools for intuitive understanding.

**REFERENCES:**

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts. Wiley.
2. Stallings, W. (2018). Operating Systems: Internals and Design Principles. Pearson.
3. Tanenbaum, A. S., & Bos, H. (2015). Modern Operating Systems. Pearson.
4. Linux man pages: mmap, fork, waitpid.