OS Practice Assignment 1

---

1 CPU Scheduling Algorithms

Real-world processes are often a mix of CPU and I/O bursts. This question explores how a scheduler can improve system utilization by overlapping I/O and CPU work.

1.1 Part A: Timeline Analysis with I/O

Consider two processes:

- **Process A (I/O-Bound):** Needs 10s of CPU, then 20s of I/O, then another 10s of CPU.
- **Process B (CPU-Bound):** Needs 40s of CPU.

Both processes arrive at t=0. Assume a simple, non-preemptive scheduler that runs a process until it either completes or issues an I/O request.

1. Draw Timeline 1 (No Overlap): Create a timeline diagram showing the usage of the CPU and the Disk if Process A is scheduled first. Show when each resource is busy or idle. Calculate the total time taken to complete both jobs.
2. Draw Timeline 2 (Overlap): Create a new timeline diagram showing how scheduling the I/O-bound process (Process A) first allows for the overlapping of I/O and CPU execution, improving resource utilization. Calculate the new total time.
3. Analyze: Briefly explain why the second schedule results in better system performance.

No overlap (CPU sits idle during A's I/O)

Assumption for this timeline: after A starts and issues I/O, the scheduler does not schedule B while A's I/O is happening, CPU stays idle until A completely finishes. (This is a pessimistic / badly-utilising schedule used to illustrate the difference.)

Events (by time):

- t = 0 → 10 : CPU runs A (first CPU burst = 10 s)
- t = 10 → 30 : Disk runs A's I/O (20 s). CPU is idle in this schedule.
- t = 30 → 40 : CPU runs A (second CPU burst = 10 s), A completes at t = 40.
- t = 40 → 80 : CPU runs B (CPU-bound = 40 s). B completes at t = 80.
- Disk is idle after t = 30.

CPU: [A: 0 - 10] [idle: 10 - 30] [A: 30 - 40] [B: 40 - 80]
Disk: [-: 0 - 10] [A_IO: 10 - 30] [-: 30 - 80]
Total completion time (makespan):
Compute step-by-step:

- A total work = 10 (CPU) + 20 (I/O) + 10 (CPU) = 40 s.
- B CPU = 40 s.
- Because CPU was idle during A's I/O, total time = A finishes at 40, then B 40 → 40 + 40 = 80 s.

Answer (Timeline 1): 80 seconds.

Timeline 2 — Overlap (good scheduling: schedule A first, then run B while A does I/O) Assumption for this timeline: when A issues I/O at t=10, the CPU is free to run other ready processes (here B). So B runs on the CPU while A's I/O proceeds on the disk, which creates overlap and better utilisation.
Events (by time):

- t = 0 → 10 : CPU runs A (first CPU burst = 10 s)
- t = 10 → 30: Disk runs A's I/O (20 s) AND CPU runs B simultaneously (B needs 40 s).
  - So B runs from t = 10 → 50 on the CPU.
  - A's I/O completes at t = 30, but A still needs its final 10 s of CPU.
- t = 30 → 50 : Disk idle; CPU still running B until t = 50.
- t = 50 → 60: CPU runs A's final CPU burst (10 s). A finishes at t = 60. B finished at t = 50.
- All done at t = 60.

timeline:
CPU: [A: 0 - 10] [B: 10 - 50] [A: 50 - 60]
Disk: [-: 0 - 10] [A_IO: 10 - 30] [-: 30 - 60]
Total completion time (makespan): Compute carefully:

- CPU busy segments in this sequence: 10 (A first) + 40 (B) + 10 (A second) = 60 s of CPU time, but because some CPU work overlaps the disk, the wall-clock finish time = 60 s.
- Another way: B runs from 10→50, and A needs last CPU 10 s after that → finish at 50+10 = 60 s.

Answer (Timeline 2): 60 seconds.
Why Timeline 2 is better:

- Timeline 2 overlaps the disk-bound work (A's 20s I/O) with useful CPU work (B's CPU), so both resources (CPU and Disk) are busy for more of the elapsed time.
- In Timeline 1, the CPU sits idle for 20 s (10→30) while the disk is busy; that is wasted CPU capacity and lengthens the total completion time from 60 s → 80 s.
- Overlapping I/O and CPU reduces the makespan and increases throughput — the system finishes both jobs faster (60 s vs 80 s) and has better resource utilization (less idle CPU time).

1.2 Part B: Simulating a Scheduler with I/O

- **Input:** Each process will now have an alternating sequence of CPU and I/O burst times (e.g., P1: [CPU 10, I/O 20, CPU 10]).
- **Logic:**
  - Maintain three states for a process: Ready, Running, and Blocked (for I/O).
  - When a running process needs I/O, it moves to the Blocked state for the duration of its I/O burst. The CPU is then assigned to the next process in the Ready queue.
  - When a process completes its I/O, it moves back to the Ready queue.
- **Output:** The program should print a log of events (e.g., "Time 10: P1 moved to Blocked", "Time 10: P2 scheduled on CPU") and the final completion time for all processes.
- **Test:** Verify your program's logic using the scenario from Part A.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_PROCESSES 10
#define MAX_BURSTS 20
#define MAX_LOG 1000
// Process states
typedef enum {
    READY,
    RUNNING,
    BLOCKED,
    FINISHED
} State;
```

```c
// Process structure
typedef struct {
    char name[10];
    int bursts[MAX_BURSTS]; // Alternating CPU and I/O burst times
    int burst_count;        // Total number of bursts
    int burst_index;        // Current burst index
    int time_left;          // Time left in current burst
    State state;            // Current state
    int completion_time;    // When process finished
} Process;
Process processes[MAX_PROCESSES];
int process_count = 0;
// Log structure
char event_log[MAX_LOG][100];
int log_count = 0;
// Log an event with timestamp
void log_event(int time, const char *msg) {
    if (log_count < MAX_LOG) {
        snprintf(event_log[log_count++], 100, "Time %d: %s", time, msg); } }
// Add a process to the system
void add_process(const char *name, int *bursts, int burst_count) {
    if (process_count < MAX_PROCESSES) {
        Process *p = &processes[process_count++];
        strcpy(p->name, name);
        memcpy(p->bursts, bursts, burst_count * sizeof(int));
        p->burst_count = burst_count;
        p->burst_index = 0;
        p->time_left = bursts[0];
        p->state = READY;
        p->completion_time = -1;  } }
// Find the next ready process (FCFS)
int find_next_ready() {
    for (int i = 0; i < process_count; i++) {
        if (processes[i].state == READY)
            return i;  }
    return -1; }
// Check if all processes are finished
int all_finished() {
    for (int i = 0; i < process_count; i++) {
        if (processes[i].state != FINISHED)
            return 0; }
    return 1; }
int main() {
    int mode;
```

```c
printf("Select I/O mode (1 = overlapping, 0 = non-overlapping): ");
scanf("%d", &mode);
int n;
printf("Enter number of processes: ");
scanf("%d", &n);
process_count = 0;
for (int i = 0; i < n; i++) {
    char name[10];
    int bursts[MAX_BURSTS];
    int burst_count = 0;
    printf("Enter name for process %d: ", i + 1);
    scanf("%s", name);
    printf("Enter burst times for %s (space separated, end with
    -1): ", name);
    int val;
    while (burst_count < MAX_BURSTS) {
        scanf("%d", &val);
        if (val == -1) break;
        bursts[burst_count++] = val;  }
    add_process(name, bursts, burst_count);  }
int time = 0;
int running = -1; // Index of running process
if (mode == 0) {
    // Strict non-overlapping: finish one process completely
    before starting the next.
    for (int i = 0; i < process_count; i++) {
        Process *p = &processes[i];
        log_event(time, "Scheduler starts process");
        while (p->state != FINISHED) {
            if (p->burst_index % 2 == 0) { // CPU Burst
                char msg[100];
                snprintf(msg, 100, "%s starts CPU burst (%d
                units)", p->name, p->time_left);
                log_event(time, msg);
                time += p->time_left;
                p->time_left = 0;
                snprintf(msg, 100, "%s finishes CPU burst",
                p->name);
                log_event(time, msg);
            } else { // I/O Burst
                char msg[100];
                snprintf(msg, 100, "%s starts I/O burst (%d units),
                CPU is idle", p->name, p->time_left);
                log_event(time, msg);
```

```
                time += p->time_left;
                p->time_left = 0;
                snprintf(msg, 100, "%s finishes I/O burst", p->name);
                log_event(time, msg);   }
            p->burst_index++;
            if (p->burst_index < p->burst_count) {
                p->time_left = p->bursts[p->burst_index];
            } else {
                p->state = FINISHED;
                p->completion_time = time;
                char msg[100];
                snprintf(msg, 100, "%s has finished all bursts
                at time %d", p->name, time);
                log_event(time, msg); } }  }
    } else {
     // Overlapping I/O mode
     while (!all_finished()) {
         // 1. Update blocked processes (I/O)
         for (int i = 0; i < process_count; i++) {
             Process *p = &processes[i];
             if (p->state == BLOCKED) {
                 p->time_left--;
                 if (p->time_left == 0) {
                     p->burst_index++;
                     if (p->burst_index < p->burst_count) {
                         p->time_left = p->bursts[p->burst_index];
                         p->state = READY;
                         char msg[50];
                         snprintf(msg, 50, "%s completed I/O,
                         moved to Ready", p->name);
                         log_event(time, msg);
                     } else {
                          p->state = FINISHED;
                          // CORRECTED: A process finishes at
                          the end of a time slice.
                          p->completion_time = time + 1;
                          char msg[50];
                          snprintf(msg, 50, "%s finished at time
                          %d", p->name, time + 1);
                          log_event(time, msg);
                          } }   }  }
         // 2. If CPU is free, schedule next ready process
         if (running == -1 || processes[running].state != RUNNING) {
             running = find_next_ready();
```

```c
            if (running != -1) {
                processes[running].state = RUNNING;
                char msg[50];
                snprintf(msg, 50, "%s scheduled on CPU",
                processes[running].name);
                log_event(time, msg); } }
        // 3. Run the process on CPU
        if (running != -1 && processes[running].state == RUNNING) {
            Process *p = &processes[running];
            p->time_left--;
            if (p->time_left == 0) {
                p->burst_index++;
                if (p->burst_index < p->burst_count) {
                    // Next burst is I/O
                    p->time_left = p->bursts[p->burst_index];
                    p->state = BLOCKED;
                    char msg[50];
                    snprintf(msg, 50, "%s completed CPU burst,
                    moved to Blocked for I/O", p->name);
                    log_event(time, msg);
                    running = -1;
                } else {
                    // Process finished
                    p->state = FINISHED;
                    // CORRECTED: A process finishes at the end
                    of a time slice.
                    p->completion_time = time + 1;
                    char msg[50];
                    snprintf(msg, 50, "%s finished at time %d",
                    p->name, time + 1);
                    log_event(time, msg);
                    running = -1; } } }
        time++;  }  }
    // Print event log
    printf("\n--- Event Log ---\n");
    for (int i = 0; i < log_count; i++) {
        printf("%s\n", event_log[i]); }
    // Print completion times
    printf("\n--- Completion Times ---\n");
    for (int i = 0; i < process_count; i++) {
        printf("%s: %d\n", processes[i].name, processes[i].completion_time);
    printf("\nTotal simulation time: %d\n", time);
    return 0; }
```

Output:-

Select I/O mode (1 = overlapping, 0 = non-overlapping): 1
Enter number of processes: 2
Enter name for process 1: p1
Enter burst times for p1 (space separated, end with -1): 10 20 10 -1
Enter name for process 2: p2
Enter burst times for p2 (space separated, end with -1): 40 -1

— Event Log —

Time 0: p1 scheduled on CPU
Time 9: p1 completed CPU burst, moved to Blocked for I/O
Time 10: p2 scheduled on CPU
Time 29: p1 completed I/O, moved to Ready
Time 49: p2 finished at time 50
Time 50: p1 scheduled on CPU
Time 59: p1 finished at time 60

— Completion Times —

p1: 60
p2: 50

Total simulation time: 60

gcc scheduler_with_io.c -o scheduler_with_io && ./scheduler_with_io
Select I/O mode (1 = overlapping, 0 = non-overlapping): 0
Enter number of processes: 2
Enter name for process 1: p1
Enter burst times for p1 (space separated, end with -1): 10 20 10 -1
Enter name for process 2: p2
Enter burst times for p2 (space separated, end with -1): 40 -1

— Event Log —

Time 0: Scheduler starts process
Time 0: p1 starts CPU burst (10 units)
Time 10: p1 finishes CPU burst
Time 10: p1 starts I/O burst (20 units), CPU is idle
Time 30: p1 finishes I/O burst
Time 30: p1 starts CPU burst (10 units)
Time 40: p1 finishes CPU burst
Time 40: p1 has finished all bursts at time 40
Time 40: Scheduler starts process
Time 40: p2 starts CPU burst (40 units)
Time 80: p2 finishes CPU burst
Time 80: p2 has finished all bursts at time 80

— Completion Times —
p1: 40
p2: 80
Total simulation time: 80

1.3 Part A: Programming Component

You will implement the following scheduling algorithms:

- First-In-First-Out (FIFO)
- Shortest Remaining Time First (SRTF)

Your program must accept input in the following format:
Process_ID Arrival_Time Number_of_CPU_Bursts
CPU_Burst_1 IO_Duration_1 CPU_Burst_2 IO_Duration_2 ... CPU_Burst_n
Example:
P1 0 3
4 2 3 3 2

(Process P1 arrives at time 0, has 3 CPU bursts [4, 3, 2], and two I/O bursts [2, 3])

1. Maintain two queues:
   (a) Ready queue for processes waiting to run on CPU.
   (b) I/O queue for processes waiting for I/O.
2. Implement scheduling:
   (a) When a process finishes a CPU burst and has more bursts left, move it to the I/O queue.
   (b) When the I/O completes, move the process back to the ready queue.
   (c) When a process has no more bursts, mark it as completed.
3. Track and display the following:
   (a) CPU Utilization percentage = (Total busy CPU time / Total simulation time) $\times$ 100
   (b) Timeline showing when:
      i. CPU is running, which process
      ii. I/O devices are busy
      iii. CPU is idle

```c
#include <stdio.h>
#include <string.h>

#define MAX_PROCESSES 10
#define MAX_BURSTS 10

// Process structure
struct Process {
    char id[10];
    int arrival;
    int bursts[MAX_BURSTS];
    int io[MAX_BURSTS];
    int num_bursts;

    // For simulation
    int current_burst;
    int remaining_time;
    int io_time;
    int state; // 0=not_arrived, 1=ready, 2=running, 3=io, 4=finished
    int ready_time;

    // Metrics
    int completion;
    int turnaround;
    int waiting;
    int response;
    int first_run;
};

struct Process processes[MAX_PROCESSES];
int num_processes = 0;

// Input function
void get_input() {
printf("Enter number of processes: ");
    scanf("%d", &num_processes);

    printf("\nInput format:\n");
    printf("Process_ID Arrival_Time Number_of_CPU_Bursts\n");
    printf("CPU_Burst_1 IO_Duration_1 CPU_Burst_2 IO_Duration_2
    ... CPU_Burst_n\n");
    printf("Example: P1 0 3\n");
    printf("          4 2 3 3 2\n\n");
```

```c
    for (int i = 0; i < num_processes; i++) {
        printf("--- Process %d ---\n", i + 1);
        printf("Enter Process_ID Arrival_Time Number_of_CPU_Bursts: ");
        scanf("%s %d %d", processes[i].id,
        &processes[i].arrival, &processes[i].num_bursts);

        printf("Enter CPU_Burst_1 IO_Duration_1 CPU_Burst_2 IO_Duration_2 ...
        for (int j = 0; j < processes[i].num_bursts; j++) {
            scanf("%d", &processes[i].bursts[j]);
            if (j < processes[i].num_bursts - 1) {
                scanf("%d", &processes[i].io[j]);
            }
        }

        // Initialize
        processes[i].current_burst = 0;
        processes[i].remaining_time = processes[i].bursts[0];
        processes[i].io_time = 0;
        processes[i].state = 0; // not arrived
        processes[i].ready_time = 0;
        processes[i].first_run = -1;
    }
}

// Find next FIFO process
int find_fifo() {
    int earliest = -1;
    int min_time = 999999;

    for (int i = 0; i < num_processes; i++) {
        if (processes[i].state == 1 && processes[i].ready_time < min_time)
        { min_time = processes[i].ready_time;
            earliest = i;
        }
    }
    return earliest;
}

// FIFO simulation
void simulate() {
    int time = 0;
    int running = -1;
    int total_cpu_time = 0;
```

```
printf("\n--- Starting FIFO Simulation ---\n");

while (1) {
    // Check arrivals
    for (int i = 0; i < num_processes; i++) {
        if (processes[i].state == 0 && processes[i].arrival <= time) {
            processes[i].state = 1; // ready
            processes[i].ready_time = time;
            printf("Time %d: %s has arrived and is Ready\n",
            time, processes[i].id);
        }
    }

    // Update I/O processes
    for (int i = 0; i < num_processes; i++) {
        if (processes[i].state == 3) { // in I/O
            processes[i].io_time--;
            if (processes[i].io_time == 0) {
                processes[i].current_burst++;
                if (processes[i].current_burst <
                processes[i].num_bursts) {
                    processes[i].state = 1; // ready
                    processes[i].ready_time = time + 1;
                    processes[i].remaining_time =
                    processes[i].bursts[processes[i].current_burst];
                    printf("Time %d: %s completes I/O and is
                    Ready for CPU burst %d (%d units)\n",
                           time + 1, processes[i].id, processes[i].curren
                } else {
                    processes[i].state = 4; // finished
                    processes[i].completion = time;
                    processes[i].turnaround = processes[i].completion -
                    processes[i].arrival;
                    printf("Time %d: %s has finished all
                    its bursts\n", time, processes[i].id);
                }
            }
        }
    }

    // Schedule if CPU idle
    if (running == -1) {
        running = find_fifo();
        if (running != -1) {
```

```
                processes[running].state = 2; // running
                if (processes[running].first_run == -1) {
                    processes[running].first_run = time;
                    processes[running].response = time -
                    processes[running].arrival;
                }
            }
        }

        // Execute running process
        if (running != -1) {
            processes[running].remaining_time--;
            total_cpu_time++;

            // Check if burst completed
            if (processes[running].remaining_time == 0) {
                if (processes[running].current_burst <
                processes[running].num_bursts - 1) {
                    // Go to I/O
                    processes[running].state = 3; // I/O
                    processes[running].io_time =
                    processes[running].io[processes[running].current_burst];
                    printf("Time %d: %s finishes CPU burst %d, moves to I/O f

                           time + 1, processes[running].id, processes[running
                } else {
                    // Process finished
                    processes[running].state = 4; // finished
                    processes[running].completion = time + 1;
                    processes[running].turnaround = processes[running].comple
                    printf("Time %d: %s has finished all its bursts\n", time
                }
                running = -1;
            }
        }

        time++;

        // Check if all finished
        int all_done = 1;
        for (int i = 0; i < num_processes; i++) {
            if (processes[i].state != 4) {
                all_done = 0;
                break;
```

```
            }
        }
        if (all_done) {
            printf("Time %d: All processes have completed. Simulation ending.
            break;
        }
    }

    // Calculate waiting times
    for (int i = 0; i < num_processes; i++) {
        int total_cpu = 0, total_io = 0;
        for (int j = 0; j < processes[i].num_bursts; j++) {
            total_cpu += processes[i].bursts[j];
            if (j < processes[i].num_bursts - 1) {
                total_io += processes[i].io[j];
            }
        }
        processes[i].waiting = processes[i].turnaround - total_cpu - total_io
    }

    printf("--- FIFO Simulation Complete ---\n");

    // Display results
    printf("\n==================== FIFO RESULTS ====================\n");
    printf("Total Simulation Time: %d\n", time);
    printf("CPU Busy Time: %d\n", total_cpu_time);
    printf("CPU Utilization: %.2f%%\n\n", (double)total_cpu_time / time * 100
        printf("Process metrics:\n");
    for (int i = 0; i < num_processes; i++) {
        printf("%s - Turnaround %d, Waiting %d, Response %d\n",
               processes[i].id, processes[i].turnaround, processes[i].waiting
    }
}

int main() {
    printf("=== FIFO CPU Scheduler ===\n");
    get_input();
    simulate();
    return 0;
}
```

Output :-

=== FIFO CPU Scheduler ===

Enter number of processes: 3

Input format:
Process_ID Arrival_Time Number_of_CPU_Bursts
CPU_Burst_1 IO_Duration_1 CPU_Burst_2 IO_Duration_2 ... CPU_Burst_n
Example: P1 0 3
4 2 3 3 2

— Process 1 —
Enter Process_ID Arrival_Time Number_of_CPU_Bursts: P1 0 3
Enter CPU_Burst_1 IO_Duration_1 CPU_Burst_2 IO_Duration_2 ... CPU_Burst_n: 4 2 3 3 2
— Process 2 —
Enter Process_ID Arrival_Time Number_of_CPU_Bursts: P2 1 2
Enter CPU_Burst_1 IO_Duration_1 CPU_Burst_2 IO_Duration_2 ... CPU_Burst_n: 6 4 4
— Process 3 —
Enter Process_ID Arrival_Time Number_of_CPU_Bursts: P3 3 3
Enter CPU_Burst_1 IO_Duration_1 CPU_Burst_2 IO_Duration_2 ... CPU_Burst_n: 5 3 2 2 1

— Starting FIFO Simulation —
Time 0: P1 has arrived and is Ready
Time 1: P2 has arrived and is Ready
Time 3: P3 has arrived and is Ready
Time 4: P1 finishes CPU burst 1, moves to I/O for 2 units
Time 6: P1 completes I/O and is Ready for CPU burst 2 (3 units)
Time 10: P2 finishes CPU burst 1, moves to I/O for 4 units
Time 14: P2 completes I/O and is Ready for CPU burst 2 (4 units)
Time 15: P3 finishes CPU burst 1, moves to I/O for 3 units
Time 18: P3 completes I/O and is Ready for CPU burst 2 (2 units)
Time 18: P1 finishes CPU burst 2, moves to I/O for 3 units
Time 21: P1 completes I/O and is Ready for CPU burst 3 (2 units)
Time 22: P2 has finished all its bursts
Time 24: P3 finishes CPU burst 2, moves to I/O for 2 units
Time 26: P3 completes I/O and is Ready for CPU burst 3 (1 units)
Time 26: P1 has finished all its bursts
Time 27: P3 has finished all its bursts
Time 27: All processes have completed. Simulation ending.
— FIFO Simulation Complete —

================== FIFO RESULTS ===================
Total Simulation Time: 27
CPU Busy Time: 27
CPU Utilization: 100.00%

Process metrics: P1 — Turnaround 26, Waiting 12, Response 0
P2 — Turnaround 21, Waiting 7, Response 3

P3 — Turnaround 24, Waiting 11, Response 7

1.4 Part B: Performance Analysis

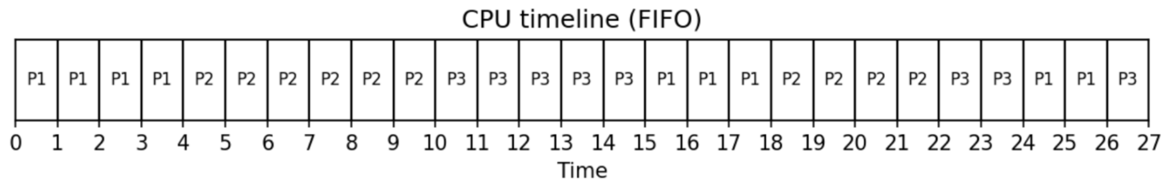
Consider the Table 1:

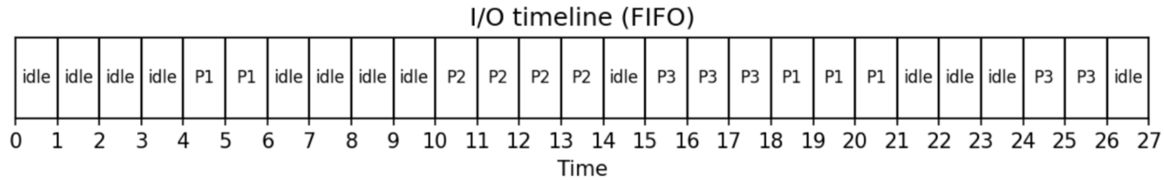
1. Simulate this workload under SJF and RR.
2. For each algorithm, calculate:
   (a) CPU Utilization
   (b) Turnaround Time (completion time - arrival time) for each process
   (c) Waiting Time (time spent in ready queue) for each process
   (d) Response Time (time until first scheduled on CPU) for each process
3. Identify which algorithm achieves the highest CPU utilization and explain why.

Table 1:

Process	**Arrival Time**	**CPU Bursts**	**I/O Bursts**
P1	0	[4, 3, 2]	[2, 3]
P2	1	[6, 4]	[4]
P3	3	[5, 2, 1]	[3, 2]

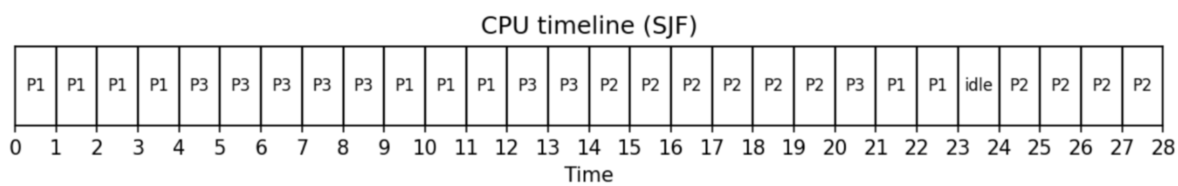
FIFO

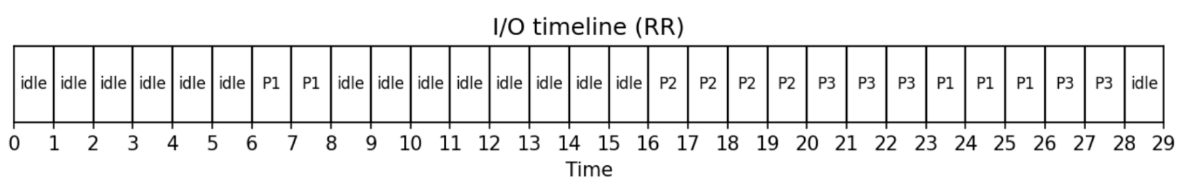




- CPU Utilization: **100.00%**
- Process metrics:
  - P1 — Turnaround **26**, Waiting **12**, Response **0**
  - P2 — Turnaround **21**, Waiting **7**, Response **3**
  - P3 — Turnaround **24**, Waiting **11**, Response **7**

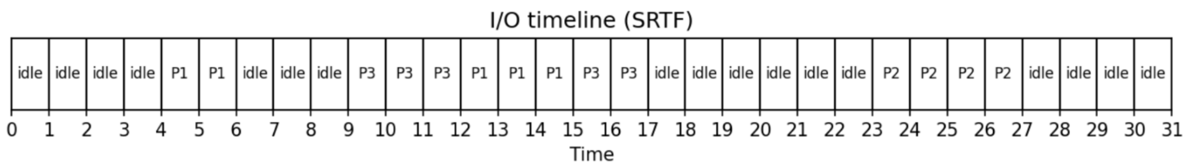
SJF (non-preemptive shortest next CPU burst)

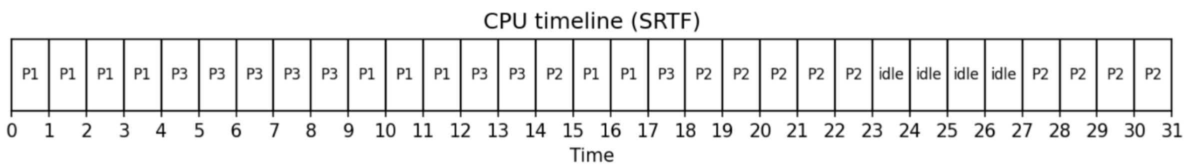




- CPU Utilization: **96.43%**
- Process metrics:
  - P1 — Turnaround **23**, Waiting **9**, Response **0**
  - P2 — Turnaround **27**, Waiting **13**, Response **13**
  - P3 — Turnaround **18**, Waiting **4**, Response **1**

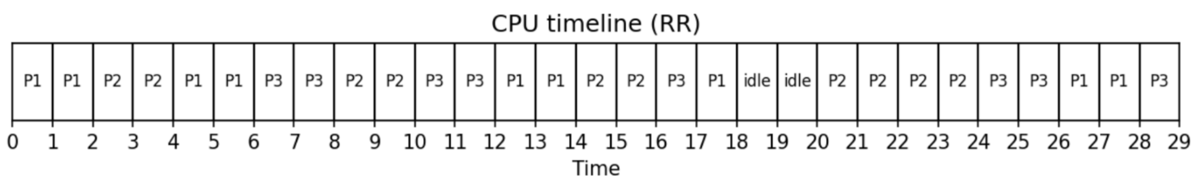
SRTF (preemptive shortest remaining time first)

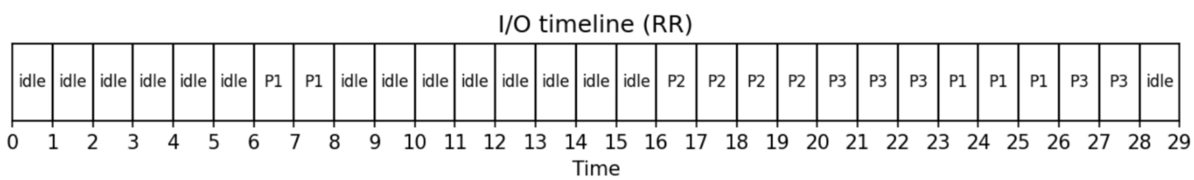




- CPU Utilization: **87.10%**
- Process metrics:
    - P1 — Turnaround **17**, Waiting **3**, Response **0**
    - P2 — Turnaround **30**, Waiting **16**, Response **13**
    - P3 — Turnaround **15**, Waiting **1**, Response **1**

RR (round-robin, quantum = 2)





- CPU Utilization: **93.10%**
- Process metrics:
    - P1 — Turnaround **28**, Waiting **9**, Response **0**
    - P2 — Turnaround **23**, Waiting **9**, Response **1**
    - P3 — Turnaround **26**, Waiting **10**, Response **3**

FIFO achieved the highest CPU utilization (100%).

Why: In this workload and with a single I/O device, FIFO scheduled processes in arrival order and never left the CPU idle between time units there were no scheduling-induced gaps or quantum-induced context-switching times modeled as idle. Preemptive policies (SRTF) and time-sliced policies (RR) cause more context switching and can lead to more times when the ready queue is empty while I/O is still happening (or the scheduler switches tasks), producing small idle windows in this discrete-time model. SJF also produced high utilization but slightly lower than FIFO because its ordering led to times when processes were in I/O and the CPU needed to wait for arrivals or IO-completions, increasing total simulation time slightly.

2 Process Management

Write a C program where a parent process creates two child processes using fork().

- The first child computes the sum of all even numbers from 1 to 50.
- The second child computes the sum of all odd numbers from 1 to 50.
- Each child prints its result and exits.
- The parent waits for both children to finish (wait()), then prints the combined total (sum of even + odd).

What we expect students to demonstrate

- Correct use of fork() twice (creating two children).
- Independent child computations; ordered parent wait() synchronization.
- Awareness that children do not share address space with parents (hence parent recomputes or uses IPC if they choose to extend

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

int main() {
    pid_t c1, c2;
    int status;

    c1 = fork();
    if (c1 == 0) {
        // First child: sum of even numbers 1 - 50
```

```c
        int sum_even = 0;
        for (int i = 2; i <= 50; i += 2)
            sum_even += i;
        printf("Child 1 (Even Sum): %d\n", sum_even);
        exit(sum_even % 255); // return small value to parent
    }
    c2 = fork();
    if (c2 == 0) {
        // Second child: sum of odd numbers 1 - 50
        int sum_odd = 0;
        for (int i = 1; i <= 50; i += 2)
            sum_odd += i;
        printf("Child 2 (Odd Sum): %d\n", sum_odd);
        exit(sum_odd % 255); // return small value to parent
    }
    // Parent waits for both children
    int even_sum = 0, odd_sum = 0;
    pid_t wpid;
    while ((wpid = wait(&status)) > 0) {
        if (WIFEXITED(status)) {
            int result = WEXITSTATUS(status);
            if (wpid == c1) even_sum = result;
            else if (wpid == c2) odd_sum = result;
        }
    }

    // Parent recomputes totals to avoid %255 limitation
    int total = 0;
    for (int i = 1; i <= 50; i++) total += i;

    printf("Parent: Combined Total = %d\n", total);
    return 0;
}
```

3 Build a System & File Utility

Objective: Write a collection of small command-line utilities that perform file operations, mathematical calculations, and report system information. Then, create a master program that runs all of them as child processes.

Create a folder that should contain the following files:

3.1 word_count.c

Contains a simple implementation of the wc (word count) command.

- The program should take a single filename as a command-line argument.
- It should read the specified file and print the total number of lines, words, and characters.
- If no filename is provided, it should print an error message.

3.2 factorial.c

Contains a program to calculate the factorial of a number. This serves as the "algorithmic" component.

- The program should take a single non-negative integer N as a command-line argument.
- It should calculate the factorial of N (N!) and print the result.
- Handle basic error cases, such as no argument being provided or the input not being a valid number. (You don't need to worry about the result overflowing for large numbers).

3.3 cal.c

Contains a simple implementation of the *cal* command using Zeller's congruence.

- The program should take month and year as arguments and calculate the first day of the month using the formula for the Gregorian calendar.
- If the month and year are not provided, it should print an error message.
- Use Zeller's congruence to determine the first day of the given month and construct the rest of the month.

3.4 run_all.c

This program will run all the executables of the above programs as child processes using the fork-wait-exec sequence.

- The program will expect two command-line arguments: a number (for factorial) and a filename (for word_count).
- Create 3 child processes using fork().
- Inside the child processes, use an exec() function to execute the other programs:
  - Child 1 will execute ./factorial with the number argument.

– Child 2 will execute ./word_count with the filename argument.

– Child 3 will execute ./cal.

The parent process will wait() for all child processes to finish before exiting.

3.5 Makefile

This Makefile should build the first 3 programs (word_count, factorial, cal) before building the main program (run_all).

- The output of compiling a C file should be an executable file of the same name (e.g., word_count.c compiles to word_count).
- Include a clean rule to remove all generated executables.

Factorial.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    // Convert argument to integer
    int num = atoi(argv[1]);

    // Calculate factorial
    long long factorial = 1;
    for (int i = 1; i <= num; ++i) {
        factorial *= i;
    }

    // Print result
    printf("Factorial of %d is %lld\n", num, factorial);

    return 0;
}
```

word-count.c

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    // Check if a filename is provided as a command-line argument.
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1;
    }
```

```c
    // Open the specified file for reading.
    FILE *file = fopen(argv[1], "r");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }
    int lines = 0, words = 0, chars = 0;
    int in_word = 0;
    char ch;
    // Read the file character by character until the end.
    while ((ch = fgetc(file)) != EOF) {
        chars++; // Increment character count.

        // Increment line count on newline character.
        if (ch == '\n') {
            lines++;
        }
        // Check for word boundaries (space, tab, newline).
        if (ch == ' ' || ch == '\t' || ch == '\n') {
            in_word = 0;
        } else if (in_word == 0) {
            in_word = 1;
            words++; }}
    // Close the file.
    fclose(file);
    // Print the counts.
    printf("Lines: %d, Words: %d, Characters: %d\n", lines, words, chars);
    return 0;
}
```

cal.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function to get the number of days in a given month of a year.
int get_days_in_month(int month, int year) {
    if (month == 2) {
        // Check for leap year.
        if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
            return 29;
        } else {
            return 28;
        }
```

```
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        return 30;
    } else {
        return 31;
    }
}

int zellers_congruence(int day, int month, int year) {
    if (month < 3) {
        month += 12;
        year--;
    }
    int k = year % 100;
    int j = year / 100;
    int day_of_week = (day + 13 * (month + 1) / 5 + k +
    k / 4 + j / 4 + 5 * j) % 7;
    return (day_of_week + 6) % 7;
}

int main(int argc, char *argv[]) {
    int month, year;

    // Use current month and year if not provided.
    if (argc < 3) {
        time_t t = time(NULL);
        struct tm tm = *localtime(&t);
        month = tm.tm_mon + 1;
        year = tm.tm_year + 1900;
    } else {
        month = atoi(argv[1]);
        year = atoi(argv[2]);

        if (month < 1 || month > 12 || year < 1) {
            fprintf(stderr, "Error: Invalid month or year provided.\n");
            return 1;
        }
    }

    char *months[] = {"January", "February", "March",
    "April", "May", "June", "July",
                      "August", "September", "October",
                      "November", "December"};

    printf("    %s %d\n", months[month - 1], year);
```

```c
    printf("Su Mo Tu We Th Fr Sa\n");

    int first_day = zellers_congruence(1, month, year);
    int days_in_month = get_days_in_month(month, year);

    for (int i = 0; i < first_day; i++) {
        printf("   ");
    }

    for (int day = 1; day <= days_in_month; day++) {
        printf("%2d ", day);
        if ((day + first_day) % 7 == 0) {
            printf("\n");
        }
    }
    printf("\n");

    return 0;
}
```

run-all.c

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

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

    // Use fork(), exec(), and wait() to run the three programs.
    for (int i = 0; i < 3; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            // Child process
            if (i == 0) {
                // Child 1: runs ./factorial with argv[1]
                execlp("./factorial", "factorial", argv[1], NULL);
            } else if (i == 1) {
                // Child 2: runs ./word-count with argv[2]
                execlp("./word-count", "word-count", argv[2], NULL);
            } else {
                // Child 3: runs ./cal with argv[2]
                execlp("./cal", "cal", argv[2], NULL);
            }
```

```c
            // If exec fails, the child will exit
            exit(1);
        }
    }
    // Parent waits for all 3 children to finish.
    for (int i = 0; i < 3; i++) {
        wait(NULL);}
    printf("\nAll child processes finished.\n");
    return 0;
}
```

makefile

```makefile
CC = gcc
TARGETS = word-count factorial cal run-all
all: $(TARGETS)
# Rule to build the main program, depends on the other utilities
run_all: run_all.c word_count factorial cal
	$(CC) -o run_all run_all.c

# Rule to build word_count
word_count: word_count.c
	$(CC) -o word_count word_count.c

# Rule to build factorial
factorial: factorial.c
	$(CC) -o factorial factorial.c

# Rule to build cal
cal: cal.c
	$(CC) -o cal cal.c

# Clean rule to remove all generated executables
clean:
	rm -f $(TARGETS)
```