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.

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.

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) × 100
   (b) Timeline showing when:
      i. CPU is running, which process
      ii. I/O devices are busy
      iii. CPU is idle

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]

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).

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.