SYSC 4001 – Operating Systems (Fall 2025)
Assignment 2 Report

Srivathsan Murali (101287534) – Student 1
Emeka Anonyei (101209704) – Student 2

Repository Links:
Part II (C): github.com/Watson752/SYSC4001_A2/tree/master/SYSC4001_A2_P2
Part III (C++): github.com/Watson752/SYSC4001_A2/tree/master/SYSC4001_A2_P3

Overview

This report summarizes the analysis and implementation for Assignment 2. **Part I** examines CPU scheduling and memory management concepts. **Part II** demonstrates process creation using Unix `fork()` and `exec()`. **Part III** implements a C++ simulator that models interrupts, process duplication, and memory allocation behavior. All code and logs are provided in the linked GitHub repositories.

Note: For grading convenience, output logs for Part III (`execution.txt`, `system_status.txt`) are stored in `input_files/`.

1 Part I – Concepts (1.5 marks)

(a) First-Come First-Served Scheduling

FCFS is a simple non-preemptive algorithm that executes processes in order of arrival (FIFO queue).

Process	P1	P2	P3	P4	P5
Arrival (ms)	0	5	8	15	20
Burst (ms)	12	8	3	6	5

Gantt Chart

```
|---P1---|---P2---|---P3---|---P4---|---P5---|
0        12       20       23       29       34
```

Proc	Arr	Burst	Start	Comp	TAT	Wait
P1	0	12	0	12	12	0
P2	5	8	12	20	15	7
P3	8	3	20	23	15	12
P4	15	6	23	29	14	8
P5	20	5	29	34	14	9

Mean TAT = 14 msMean Wait = 7.2 ms. Longer jobs delay shorter ones (convoy effect).

(b) FCFS with I/O every 2 ms (0.5 ms I/O)

Each process pauses for I/O every 2 ms, extending total run time.

```
|---P1---|---P2---|---P3---|---P4---|---P5---|
0        15       25       29       36       42
```

Proc	Arr	Adj Exec	Comp	TAT	Wait
P1	0	15	15	15	0
P2	5	10	25	20	10
P3	8	4	29	21	17
P4	15	7	36	21	14
P5	20	6	42	22	16

Mean TAT 19.8 msMean Wait 11.4 ms. Frequent I/O reduces CPU utilization and increases latency.

(c) Memory Management – Multiple Partitions

Initial holes (KB): {85, 340, 28, 195, 55, 160, 75, 280}. Jobs: J1 = 140, J2 = 82, J3 = 275, J4 = 65, J5 = 190.

Algorithm	Internal Frag (KB)	Jobs Alloc	Remarks
First Fit	338	4/5	Fast; moderate waste
Best Fit	43	5/5	Efficient for small jobs
Worst Fit	453	5/5	Keeps large holes; wasteful

Best Fit minimizes fragmentation; First Fit balances speed and simplicity.

2 Part II – Concurrent Processes in Unix (1 mark)

Student 2 – Emeka Anonyei – `fork()`

`fork()` duplicates the calling process, creating a parent–child pair. Both execute the same code; parent receives child PID, child returns 0. Each has its own address space, preventing interference. This system call forms the foundation of Unix process hierarchies.

Student 1 – Srivathsan Murali – `exec()`

`exec()` replaces the current process image with a new program while keeping the same PID. Code and data segments are overwritten; control transfers to the new program's entry point. Descriptors and environment variables can persist depending on variant. Combined with `fork()`, it enables a parent to spawn a child and overlay it with another program—core to Unix command execution.

3 Part III – fork/exec API Simulator (2.5 marks)

Objective

To design a C++ API-level simulator that models interrupt handling, process creation (`FORK`) and program replacement (`EXEC`), and memory partition management.

Design Overview

Inputs: `trace.txt`, `vector_table.txt`, `device_table.txt`, `external_files.txt`. Outputs: `execution.txt` (event log) and `system_status.txt` (PCB snapshots).

Vector Table	Maps interrupts (FORK → 0x0004, EXEC → 0x0006).
Device Table	Stores device latencies for `END_IO`.
External Files	Program catalog (name, size MB).
Memory Partitions	Six slots: 40, 25, 15, 10, 8, 2 MB (`init` in 6).
PCB Table	PID, program, partition, size, state (running/waiting).
Trace Parser	Reads `CPU`, `SYSCALL`, `FORK`, `EXEC`, `END_IO`, conditionals.

Interrupt Flow

Each event follows:

1. Switch to kernel mode; save context.
2. Lookup vector table; load handler address.
3. Execute ISR body (`FORK`/`EXEC`).
4. Scheduler update and PCB/partition edit.
5. `IRET` – restore context to user mode.

Algorithm Summary

FORK: Duplicate parent PCB, assign partition (best-fit), child runs first, parent waits. **EXEC:** Replace image and size; reuse or relocate partition as needed.

```
ISR_FORK(parent):
  child <- clone(parent)
  assign_partition(child.size)
```

```
  parent.state = waiting; child.state = running
  log_execution(); log_system_status()

ISR_EXEC(pid, program):
  size <- lookup(program)
  if !fits(partition(pid), size): reassign_partition(pid, size)
  pcb[pid].program = program; pcb[pid].size = size
  log_execution(); log_system_status()
```

Simulation Results

Trace sample:

```
FORK,10
IF_CHILD,0
EXEC program1,50
IF_PARENT,0
EXEC program2,25
ENDIF,0
```

Observed behavior: Child PID 1 runs first (child-first policy), executes `program1` (10 MB → partition 4). Parent then executes `program2` (15 MB → partition 3). ISR logs show context save, vector lookup, scheduler, and `IRET`.

Execution log excerpt:

```
0,1,switch to kernel mode
1,10,context saved
11,1,find vector 2 in memory 0x0004
13,10,cloning PCB
23,0,scheduler called
...
time:247; trace: EXEC program1,50
time:1168; trace: EXEC program2,25
```

System status snapshots:

```
time:24; FORK,10
|1|init|5|1|running|
|0|init|6|1|waiting|
time:247; EXEC program1,50
|1|program1|4|10|running|
|0|init|6|1|waiting|
time:1168; EXEC program2,25
|0|program2|3|15|running|
```

Interpretation. `FORK` duplicates `init`; child inherits a partition and runs immediately. `EXEC` replaces its image without changing PID and updates the PCB. The logs mirror real kernel interrupt sequences.

Discussion & Conclusion

The C++ simulator accurately models core kernel-level operations for `fork()` and `exec()`. Process states and partition allocations match expected behavior, and output files verify correct ISR ordering. While simplified (no real scheduling or concurrency), the implementation demonstrates clear understanding of context switching and memory reuse in OS design.

References

Silberschatz, Galvin & Gagne, *Operating System Concepts*.
Linux manual pages (`fork`, `exec`, `shmget`, `semget`).
Assignment 2 Handout (Fall 2025).
Course slides – Process Management and Interrupts.