SYSC 4001 - Operating Systems

Assignment 02

Brodie Macleod - 101302779 (Student 1)

& Sahil Todeti - 101259541 (Student 2)

https://github.com/auriza-jpg/SYSC4001_A2_P3.git

https://github.com/auriza-jpg/SYSC4001_A2_P2.git

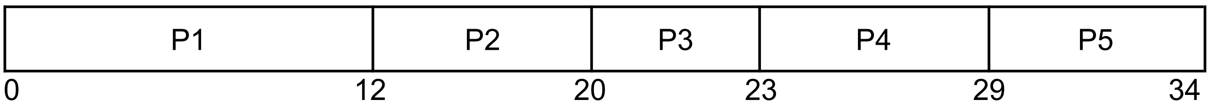
Part 1

a) [0.5 marks] With the help of Gantt charts, draw the execution timeline for the following scheduling algorithms:

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

i. **First-Come first First-Served:**

1) The First-Come First-Served (FCFS) scheduling algorithm schedules processes to the CPU in the order they arrive in the “ready” queue. When an admitted process arrives in the ready queue, its PCB is added to the tail of the FIFO queue, and the CPU scheduler selects the process at the head of the queue when called by the process termination ISR. Once a process is dispatched and begins execution, it will always run to completion. The drawback of this lack of preemption is the convoy effect. The convoy effect involves short processes or I/O-bound processes being delayed behind long CPU-bound processes, increasing waiting time and turnaround time, as well as delaying response time.



2) Completion Time of each process

P1: 12 ms

P2: 20 ms

P3: 23 ms

P4: 29 ms

P5: 34 ms

3) Turnaround Time = Completion Time (ms) - Arrival Time (ms)

P1: 12 - 0 = 12 ms

P2: 20 - 5 = 15 ms

P3: 23 - 8 = 15 ms

P4: 29 - 15 = 14 ms

P5: 34 - 20 = 14 ms

4) Mean Turnaround Time = $\frac{12+15+15+14+14}{5}$ = 14 ms

ii. Round Robin (time slice of 4 ms).

1) Round Robin (RR, is a pre-emptive CPU scheduling algorithm designed for time-sharing systems. In RR, each process is allocated a fixed time slice, during which it executes on the CPU.

P1	P1	P2	P3	P1	P2	P4	P5	P4	P5

0 4 8 12 15 19 23 27 31 33 34

2) Completion Time of each process

P1: 19 ms

P2: 27 ms

P3: 15 ms

P4: 33 ms

P5: 34 ms

3) Turnaround Time = Completion Time (ms) - Arrival Time (ms)

P1: 19 - 0 = 19 ms

P2: 27 - 5 = 22 ms

P3: 19 - 8 = 11 ms

P4: 33 - 15 = 18 ms

P5: 34 - 20 = 14 ms

4) Mean Turnaround Time = $\frac{19+22+11+18+14}{5}$ = 16.8 ms

iii. Shortest Job First with preemption.

1) Shortest Job First, SJF, is a kind of process which are always scheduled based on the shortest remaining time among all the arrived processes.

P1	P3	P1	P4	P5	P2

0 8 11 15 21 26 34

2) Completion Time of Each Process:

P1: 15 ms

P2: 34 ms

P3: 11 ms

P4: 21 ms

P5: 26 ms

3) Turnaround Time = Completion Time (ms) - Arrival Time (ms)

P1: 15 - 0 = 15 ms

P2: 34 - 5 = 29 ms

P3: 11 - 8 = 3 ms

P4: 21 - 15 = 6 ms

P5: 26 - 20 = 6 ms

4) Mean Turnaround Time = $\frac{15+29+3+6+6}{5}$ = 11.8 ms

iv. Multiple queues with feedback (high-priority queue: quantum = 2; mid-priority queue: quantum = 3; low-priority queue: FIFO)

1) Multiple Level Queues, uses multiple queues with varying priorities and quanta (high: 2 ms, mid: 3 ms, low: FCFS). Processes start at the highest priority and are demoted if they consume their quantum

P1	P1	P2	P2	P3	P2	P3	P1	P4	P4	P5	P5	P1	P2	P4

0 2 5 7 8 10 13 14 15 17 20 22 25 31 33 34

2) Completion Times of each Process.

P1: 31 ms

P2: 33 ms

P3: 14 ms

P4: 34 ms

P5: 25 ms

3) Turnaround Time = Completion Time (ms) - Arrival Time (ms)

P1: 31 - 0 = 31 ms

P2: 33 - 5 = 28 ms

P3: 14 - 8 = 6 ms

P4: 34 - 15 = 19 ms

P5: 25 - 20 = 5 ms

4) Mean Turnaround Time = $\frac{31+28+6+19+5}{5}$ = 17.8 ms

b) [0.5 mark] Now assume that each process in part a) requests to do an I/O every 2 ms, and the duration of each of these I/O is 0.5 ms. Create new Gantt diagrams considering the I/O operations and repeat all the parts done in part a) using this new input trace

1) FCFS, with I/O

| P1 | I/O | P1 | I/O | P2 | P1 | P3 | P2 | P1 | P3 | P4 | P2 | P1 | P5 | P4 |
0 2 2.5 4.5 5 7 9 11 13 15 16 18 20 22 24 26

| P2 | P1 | P5 | P4 | P5 | IDLE | P4 | P5 |
26 28 30 32 34 36 38 40 ...

P1	IDLE	P1	IDL E	P 2	P 1	P 3	P 2	P 1	P3	P4	P2	P1	P5	P4	P2	P1	P5	P4	P5

| P1 | IDLE | P1 | IDLE | P2 | P1 | P3 | P2 | P1 | P3 | P4 | P2 | P1 | P5 | P4 | P2 | P1 | P5 | P4 |
P5 |
0 2 2.5 4.5 5 7 9 11 13 15 16 18 20 22 24 26 28 30 32 34

2) Round Robin (time slice of 4 ms), with I/O

| P1 | IDLE | P1 | IDLE | P1 | P2 | P1 | P3 | P2 | P1 | P3 | P4 | P2 | P1 | P5 | P4 | P2 | P5 | P4 |
P5 |
0 2 2.5 4.5 5 7 9 11 13 15 17 18 20 22 24 26 28 30 32 34

3) Shortest Job First with preemption, with I/O

| P1 | IDLE | P1 | IDLE | P1 | P2 | P3 | P1 | P3 | P1 | P2 | P1 | P2 | P5 | P2 | P5 | P4 | P5 | P4 |
IDLE | P4 |
0 2 2.5 4.5 5 7 9 11 13 14 16 18 20 22 24 26 28 30 31 33
33.5 34

4) Multiple Queues, with I/O

| P1 | IDLE | P1 | IDLE | P2 | P1 | P3 | P2 | P1 | P3 | P4 | P2 | P1 | P5 | P4 | P2 | P1 | P5 | P4 |
P5 |
0 2 2.5 4.5 5 7 9 11 13 15 16 18 20 22 24 26 28 30 32 34

c) [0.5 mark] Consider a multiprogrammed system that uses multiple partitions (of variable size) for memory management. A linked list of holes (the “free” list) is maintained by the operating system to keep track of the available memory in the system.

Position	**Hole Size**	**Status**
1	85 KB	Free
2	340 KB	Free
3	28 KB	Free
4	195 KB	Free
5	55 KB	Free
6	160 KB	Free
7	75 KB	Free
8	280 KB	Free

Job No.	**Arrival Time**	**Memory Requirement**
J1	**t_1**	140 KB
J2	**t_2**	82 KB
J3	**t_3**	275 KB
J4	**t_4**	65 KB
J5	**t_5**	190 KB

[with $t_1 < t_2 < t_3 < t_4 < t_5$]

1) Determine which free partition will be allocated to each job for the following algorithms:
• (i) First Fit
• (ii) Best Fit
• (iii) Worst Fit
For each algorithm, show the allocation table (which job gets which partition), the remaining free memory after all allocations, the total internal fragmentation and the total external fragmentation.

2) [0.2 marks] Based on your calculations, analyze and compare the three algorithms according to memory utilization efficiency and fragmentation. Based on your analysis, justify which algorithm would be most appropriate for a system with frequent small allocations and a system with mixed workload sizes. Show all calculations step by step; also ,draw the memory state after each allocation. Calculate fragmentation metrics for comparison and provide a detailed justification for your analysis

Part 2

1) `fork()` system call is used to create new processes by duplicating the current process. After `fork()` is called, it inherits the child process, and two processes run independently: a parent process and a child process. `fork()` returns zero to the child process and returns the child's process ID (PID) to the parent process. If `fork()` fails, it returns a negative value.}

2) The exec system call replaces the memory image of the calling process with a new program. When called using the Syscall interface through the standard library, the operating system switches from user mode to kernel mode and loads the code, data, and stack, etc, of the provided executable into the process's memory, replacing the previous program entirely. The process retains its PID but begins execution at the `main()` of the program, and does not give control to the original program unless an exception occurs during its execution. In assignments, and mentioned in "Tutorial on the Unix/Linux operating system" found on Brightspace, mention for assignments we use.

```
execlp(char *filename, char *arg0, ...  ..  ........  ... , (char *)
```

Argument 1 is the name of the executable file, the remaining are the parameters passed to the new program's main() function, and will end with a null pointer to indicate its termination

3. Extend the processes above once more. Use the wait system call. Process 1 starts as in 2, and when
Process 2 starts, and it waits for it. Process 2 runs until it reaches a value lower than -500. When this happens, Process 1 should end too.

Part3: report

This assignment section involved extending the existing Interrupts-API simulator from the previous assignment to include FORK and EXEC system calls.

Analyze results (From Test8). This Test Case contains many of the features contained in smaller ones

Init (Trace.txt)	Program 2	Program 3	CPU Bursts Programs
FORK, 10 IF_CHILD, 0 EXEC program1, 20 CPU, 8 EXEC program2, 15 IF_PARENT, 0 EXEC program3, 25 CPU, 20 ENDIF, 0 CPU, 12	CPU, 10 FORK, 8 IF_CHILD, 0 EXEC program4, 18 IF_PARENT, 0 CPU, 6 ENDIF, 0 CPU, 5	CPU, 10 FORK, 9 IF_CHILD, 0 EXEC program5, 16 IF_PARENT, 0 CPU, 7 ENDIF, 0 CPU, 8	Program 1: CPU, 12 Program 4: CPU, 14 Program 5: CPU, 11

Notes and explanations reference the timestamp of a given line.

Execution.txt (For test_8) + More Notes	Notes, Explanations and System Status
0**,** 1, switch to kernel mode 1, 10, context saved 11, 1, find vector 2 in memory position 0x0004 12, 1, load address 0X0695 into the PC 13, 10, Cloning the PCB	**Calling Fork (init)** ” SYSCALL” (Save context, vector to address, run ISR) - ISR copies the information needed from the PCB of the parent process to the child process. In the simulation, a new PCB block initialized with fields from the parent block. As the Partitions are fixed, we need to place it in an empty partition. The new PID is acquired by incrementing the largest current PID
23,0, Scheduler called 23, 1, IRET ************************************	time: 24; current trace: FORK, init, 10 (**Result of calling fork)** +--+ | PID |program name |partition number | size | state | +--+ | 1 | init | 5 | 1 | running | | 0 | init | 6 | 1 | waiting | +--+ **Note:** Incremented PID and new partition. Same size and same name (copied from parent). Default interrupt return. Child init is running while the parent is waiting. *Here*, the child process is interacting with conditionals. To assign IF_CHILD and post-ENDIF instructions to the child program, the simulator uses tags to build a child_trace from the parents, ignoring IF_PARENT to ENDIF. The child's trace is built as [EXEC], skips adding [CPU8, EXEC] and breaks to begin executing the trace. This involves a recursive call to parse_trace () with [EXEC] and the wait queue + parent PCB

24, 1, switch to kernel mode 25, 10, context saved 35, 1, find vector 3 in memory position 0x0006 36, 1, load address 0X042B into the PC 37, 20, Program is 5 Mb large 57, 75, loading program into memory 132, 4, marking partition as occupied 136, 4, updating PCB 140, 0, scheduler called 140, 1, IRET	**Calling Exec program1, 20** Like fork, the system call interface loads the EXEC ISR, taking the program name/Address as a parameter. The simulator queries the external_files table to obtain the new program's size, and loads it into memory taking 15 * size ms The old memory partition is freed to represent the process of freeing heap, stack and pointers from the PCB A new partition is allocated based on the program size, representing assigning a new stack, heap etc to the new program. The PCB is updated to contain the new program's name ** Return to user mode as the new program is being executed. The simulator searches for the new program's trace (simulating adding the program to the process's memory). Another recursive call to parse_trace with the current wait queue and the program's loaded trace. time: 365; current trace: EXEC, program3, 25 \\---+ | PID |program name |partition number | size | state | +---+ | 0 | program3 | 4 | 9 | running | +---+
141, 12, CPU Burst *****************	Simulated a burst of 12ms of CPU while executing Program 1. When complete, control is returned to the parent (PID 0), as the child process's (PID 1) program has finished executing. Achieved with the **"important"** break line, which allows the first recursive parse_trace to return with the updated execution and time.

```
177, 1, switch to kernel mode
178, 10, context saved
188, 1, find vector 3 in memory position 0x0006
189, 1, load address 0X042B into the PC
190, 25, Program is 9 Mb large
215, 135, loading program into memory
350, 7, marking partition as occupied
357, 7, updating PCB
364, 0, scheduler called
364, 1, IRET
```

Calling EXEC for PID 0 (root parent), loading program 3 into the parent's PCB. It is visible that the further IF_CHILD CPU and EXEC instructions were not executed as the child process terminated. This is also visible in the system_status update when exec finishes, PID 0, now containing program 3, is the only active PCB in the table.

time: 365; current trace: EXEC, program3, 25 **(State after exec)**

```
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   0 |    program3 |               4 |    9 | running |
+------------------------------------------------------+
```

```
365, 10, CPU Burst
375, 1, switch to kernel mode
376, 10, context saved
386, 1, find vector 2 in memory position 0x0004
387, 1, load address 0X0695 into the PC
388, 9, Cloning the pcb
397,0, Scheduler called
397, 1, IRET
398, 1, switch to kernel mode
399, 10, context saved
409, 1, find vector 3 in memory position 0x0006
410, 1, load address 0X042B into the PC
411, 16, Program is 4 Mb large
427, 60, loading program into memory
487, 8, marking partition as occupied
495, 8, updating PCB
503, 0, scheduler called
503, 1, IRET
504, 11, CPU Burst
913, 7, CPU Burst
920, 8, CPU Burst
```

365 begins executing program 3's (PID 0) 10ms of CPU burst
375 Fork is then called, creating a child (PID 1)

time: 398; current trace: FORK, program3, 9 **(State after Fork)**

```
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |    program3 |               3 |    9 | running |
|   0 |    program3 |               4 |    9 | waiting |
+------------------------------------------------------+
```

398 EXEC on PID 1, conditional instruction from the program3's IF_CHILD trace

time: 504; current trace: EXEC, program5, 16

```
+------------------------------------------------------+
| PID |program name |partition number | size |   state |     (State after EXEC)
+------------------------------------------------------+
|   1 |    program5 |               3 |    4 | running |
|   0 |    program3 |               4 |    9 | waiting |
+------------------------------------------------------+
```

504 executing program 5's 11ms of CPU Burst, and returning control to parent (PID 0), within the IF_PARENT conditional (**913, 7, CPU 7),** and then exiting the IF_DEF to terminate after executing **920, 8, CPU burst.**

There were certain required test cases which must be run, along with questions in the form of comments. The first was given by **TestCase1**

```
Init PCB block
FORK, 10 //fork is called by init
IF_CHILD, 0 EXEC program1, 50 //child executes program1
IF_PARENT, 0
EXEC program2, 25 //parent executes program2
ENDIF, 0 //rest of the trace doesn't really matter (why?)
Contents of program1:
CPU, 100
Contents of program2:
SYSCALL, 4
OUTPUT FROM OutputFiles/_test_1_system_status.txt
time: 24; current trace: FORK, init, 10                    NOTES:
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+
|   1 |        init |               5 |    1 | running |   init 5 (child) and init 6
(parent)
|   0 |        init |               6 |    1 | waiting |
+------------------------------------------------------+
time: 246; current trace: EXEC, program1, 50               **if child block*
```

```
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+   child runs to completion
(program1)
|   1 |    **program1** |                 4 |   10 | running |
|   0 |        init |                 6 |    1 | waiting |
+------------------------------------------------------+
time: 648; current trace: EXEC, program2, 25                **if_parent block**
+------------------------------------------------------+
| PID |program name |partition number | size |   state |
+------------------------------------------------------+    parent is running to
completion
|   0 |    **program2** |                 3 |   15 | running |
+------------------------------------------------------+
```

The rest of the trace after ENDIF would not matter, as all PCBs that could potentially reach past the conditionals have replaced their memory with a new program. This is demonstrated in test_case9

```
… (same as test_case1)
ENDIF
CPU 55
```

With execution.txt output never reaching the CPU call:

```
…
661, 250, SYSCALL ISR_Activities
911, 1, IRET    //Note: IRET is associated with program2's SYSCALL.
```

Other individual testing, with various programs, to see how variables affected the simulation. This can be viewed in test cases 4 to 8. These were created to test performance variations given variable context switch overheads, as well as IO versus CPU-bound processes, and slower versus faster loading speeds. **Slower load-per-MB time** makes EXEC-heavy traces significantly longer, especially when large programs (20 MB) are repeatedly loaded. **IO-bound tests** are most affected by context switch overhead, increasing or decreasing significantly. The most drastic differences in performance time were seen in tests that involved forking and executing several large file-sized IO-heavy programs, and increasing or decreasing the loading speed, as well as the context switch overhead. Context switching also heavily affected exec or fork-heavy tasks, as many SYSCALLs are needed. Overall, across all test cases, modifying the load speed saw the most dramatic performance increase.