Operating System QUIZ 1

Marks: 30 **Time: 40 mins**

---

1. (25 points) Answer the following questions with an explanation. No partial marking for 0.5/1 mark questions, which need only a single-word answer. If there is a different answer that you think is correct, please inform the group. Others can verify.

   (a) (0.5 points) The OS maintains the process-related information in Process Control Block (PCB).

   (b) (0.5 points) Which process becomes the parent of orphan processes?Init process

   (c) (1 point) The illusion of each process having a dedicated CPU is achieved through Multiprogramming or Multitasking or Virtualization of CPU

   (d) (1 point) The data structure where the OS saves the kernel context of a process during a context switch is called the Process Control Block (PCB) or Kernel Stack.

   (e) (1 point) In Round Robin scheduling, the CPU executes each process for a fixed duration referred to as quantum/time slice.

   (f) (1 point) POSIX API defines a standard set of system calls for portability across different OSes.

   (g) (1 point) Consider a process P that executes the fork system call twice. That is, it runs code like *int ret1 = fork(); int ret2 = fork();*. How many **direct children of P** and how many other descendants **(indirect children) of P** are created by the above lines of code? You may assume that all fork system calls succeed. Two direct children of P are created, and one other descendant of P is created.

   (h) (1 point) Given two different machines running different, but POSIX-compliant, operating systems, an application's source code written for one machine must always be rewritten to run on the other. Is this true or false? Explain your answer in one line.
   False. If the code is written using the POSIX API, it need not be rewritten for another POSIX-compliant system.

   (i) (2 points) In Round Robin with time slice = 2s, three processes A, B, C arrive at the same time and each runs for 6s. What are the average response time and the average turnaround time? Explain. partial marking only for explanation but answer needs to be correct.
   Response Time = First Run Time - Arrival Time; Process A: Starts at 0s. Response Time = 0 - 0 = 0s; Process B: Starts at 2s. Response Time = 2 - 0 = 2s; Process C: Starts at 4s. Response Time = 4 - 0 = 4s; Average Response Time = (0 + 2 + 4) / 3 = 6 / 3 = **2s**; Turnaround Time = Completion Time - Arrival Time; Process A: Completes its final slice at 14s. Turnaround Time = 14 - 0 = 14s; Process B: Completes its final slice at 16s. Turnaround Time = 16 - 0 = 16s; Process C: Completes its final slice at 18s. Turnaround Time = 18 - 0 = 18s; Average Turnaround Time = (14 + 16 + 18) / 3 = 48 / 3 = **16s**

(j) (2 points) Consider a process P1 that forks P2, P2 forks P3, and P3 forks P4. P1 and P2 continue to execute while P3 terminates. Now, when P4 terminates, which process must wait for and reap P4? init (orphan processes are reaped by init)

(k) (2 points) Four processes arrive at time 0 with the following burst times: Processes = [P1, P2, P3, P4]; Arrival Time = [0, 0, 0, 0]; Burst Time = [5, 2, 8, 3]; Compute the turnaround time of each process and the average turnaround time if scheduled using First-Come, First-Served (FCFS). partial marking only for explanation but answer needs to be correct.
Since all arrive at 0, they execute in the order P1 → P2 → P3 → P4, Completion Times (CT): P1 = 5; P2 = 7; P3 = 15; P4 = 18; Turnaround Times (TT = CT − AT): P1 = 5 − 0 = 5; P2 = 7 − 0 = 7; P3 = 15 − 0 = 15; P4 = 18 − 0 = 18
Average TT: (5+7+15+18)/4=45/4=11.25

(l) (2 points) Why is division by zero considered a trap and not an interrupt? Explain how the CPU and OS cooperate to handle it. partial marking allowed based on answers.
While executing the instruction the CPU **cannot produce a valid result, so it raises a trap** (not an interrupt, since it's caused by the running instruction itself)
Handling it : Trap mechanism, the CPU raises a trap, switches to kernel mode, and **invokes the OS trap handler**, which sends a signal to terminate or handle the process safely.

(m) (3 points) Consider the following set of processes: partial marking only for explanation but answer needs to be correct.

Process	**Arrival Time**	**Burst Time**
P1	0	8
P2	1	4
P3	2	9
P4	3	5

```
| P1 |    P2    |    P4     |       P1       |          P3          |
+----+--------+---------+--------------+------------------+
0    1        5         10             17                 26
```

1. Calculate the **Turnaround Time (TT)** for each process and the **average Turnaround Time**.
2. Calculate the **Response Time (RT)** for each process and the **average Response Time**.
3. Identify the **CPU scheduling policy** used and explain your reasoning.

1. TT=CompletionTime - ArrivalTime; TT(P1) = 17 - 0 = 17; TT(P2) = 5 - 1 = 4; TT(P3) = 26 - 2 = 24; TT(P4) = 10 - 3 = 7;
Average Turnaround Time = (17+4+24+7)÷4=52÷4=13.0

2. RT = FirstRunTime - ArrivalTime; RT(P1) = 0 - 0 = 0; RT(P2) = 1 - 1 = 0; RT(P3) = 17 - 2 = 15; RT(P4) = 5 - 3 = 2;
   Average Response Time = (0+0+15+2)÷4=17÷4=4.25
3. The scheduling policy is Shortest Remaining Time First (SRTF); It is preemptive: Process P1 starts running but is interrupted (preempted) at time 1 when P2 arrives. This is a clear sign of a preemptive algorithm.

(n) (3 points) The chart below illustrates the execution of three processes. partial marking only for explanation but answer needs to be correct.

Process	Arrival Time	Burst Time
P1	0	5
P2	1	4
P3	3	6

```
| P1 | P2 | P1 | P3 | P2 |P1|      P3      |
+----+----+----+----+----+--+--------------+
0    2    4    6    8    10 11             15
```

1. Calculate the **Turnaround Time (TT)** for each process and the **average Turnaround Time**.
2. Calculate the **Response Time (RT)** for each process and the **average Response Time**.
3. Identify the **CPU scheduling policy** used and justify your answer.

1. TT = CompletionTime - ArrivalTime; TT(P1) = 11 - 0 = 11; TT(P2) = 10 - 1 = 9; TT(P3) = 15 - 3 = 12
   Average Turnaround Time = (11+9+12)÷3=32÷3=10.67
2. RT=First Run Time - Arrival Time; RT(P1) = 0 - 0 = 0; RT(P2) = 2 - 1 = 1; RT(P3) = 6 - 3 = 3
   Average Response Time = (0+1+3)÷3=4÷3=1.33
3. The scheduling policy is Round Robin (RR). It uses time-slicing: Processes are not running to completion. Instead, they run for a short, fixed period before being preempted and placed at the back of the ready queue. For example, P1 runs, then P2 runs, then P1 gets another turn; It has a fixed time quantum

(o) (2 points) A process makes a system call to read from a file. Explain step-by-step what transitions occur between user mode and kernel mode until the data is returned to the process. partial marking only for explanation
A process in user mode makes a read() system call, causing a trap that switches the CPU to kernel mode. The OS validates the request, initiates the I/O with the disk controller, and may put the process to sleep. When the data is ready, the kernel copies it to the process's buffer and executes a return instruction, switching the CPU back to user mode to resume the process with the requested data.

(p) (1 point) A process waiting for data from disk is in the Blocked process state.

(q) (1 point) The unique identifier assigned to a process by the OS is called PID.

(r) (1 point) For what types of workloads does SJF deliver the same response times as RR? Jobs Arrive in Order of Increasing Length or All Jobs Have Equal Length

2. (5 points) Below are the C codes involving different scenarios. Assuming there are no syntax or semantic errors, answer the questions:

(a) (2 points) What values of x will be printed by the child and the parent? Explain. 1 mark for correct print and 1 for explanation. Partial allowed only for explanation

```
int main() {
    int x = 5;
    if (fork() == 0) {
        x += 5;
        printf("Child:x=%d\n", x);
    } else {
        x -= 5;
        printf("Parent:x=%d\n", x); }
    return 0; }
```

Output:
Parent: x = 0; Child: x = 10
Explanation: Each process has its own copy of x after the fork.

(b) (1 point) How many times is "Hello" printed in total? Explain your answer.0.5 mark for correct print and 0.5 for explanation. No partial allowed.

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

int main() {
    fork();
    fork();
    printf(''Hello\n");
    return 0; }
```

Output: Hello
Hello
Hello
Hello
Explanation: First fork() $\rightarrow$ 2 processes
Second fork() $\rightarrow$ each process forks again $\rightarrow$ 4 processes total
Each process prints "Hello" once.

(c) (2 points) What is the output of the code below? What is the mathematical relation between the number of print statements and the value of i? 1 mark for correct print and 1 for explanation. This output is not fixed but pattern is fixed and also the relation is fixed.

```
#include <stdio.h>
#include <unistd.h>
int main() {
for(int i = 0; i < 4; i++) {
int ret = fork();
if(ret == 0)
printf("child %d\n", i);
} }
```

Output:

```
child 0
child 1
child 1
child 2
child 2
child 2
child 2
child 3
child 3
child 3
child 3
child 3
child 3
child 3
child 3
```

At the end, 16 processes exist. Only the child process (where fork() returns 0) prints. Parent processes do not print. So at each fork, exactly the new child prints once.

1st iteration (i=0) → 1 child prints → "child 0"

2nd iteration (i=1) → 2 children print → "child 1" (twice)

3rd iteration (i=2) → 4 children print → "child 2" (four times)

4th iteration (i=3) → 8 children print → "child 3" (eight times)

$1 + 2 + 4 + 8 = 15$ lines total.

Relation: The statement "child i" is printed 2^i times for i=0 to 3. **This is the only acceptable response.**