Process Management
Practice Exercises

COMP-310/ECSE-427 Winter 23

Question 1: Fork-Exec 1

How many times does the following program print Bonjour?
First, answer without running the code. Then, use the code to double-check your solution.

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

void fork_test(){
  if (fork() !=0){
    if (fork () !=0){
      fork();
    }
    else
      printf("hello 1\n");

  } else {

    if(fork() !=0) {
       printf("hello\n");
    } else {
       fork();
    }
  }

printf("Bonjour\n");

}

int main() {
  fork_test();
  return 0;
}
```

Question 2: Fork-Exec 2

When a process creates a new process using the fork() operation, which of the following states is shared between the parent process and the child process?
a. Stack
b. Heap
c. Shared memory segments

Question 3: Multi-process Communication

A browser is, basically, an infinite loop consisting of handling events and updating the screen. If you run on a machine with a single CPU core, does it make sense to use multiple processes in the browser? If yes, why? If not, why not?

Question 4: Remote Procedure Calls

We have a server that implements a single remotely callable procedure

```
int position(char c, char* s)
```

It takes as arguments a single character and a null-terminated character string, and returns the position of the first occurrence of the character in the string, or -1 if the character does not occur in the string.

Write pseudo-code for a client stub and a server stub for this remote procedure call. Describe clearly the format of the messages sent between client and server. You can assume the existence of helper functions to allocate and de-allocate memory for these messages.

Question 5: Scheduling 1

Two processes, A and B, have the following sequential execution patterns:

A: CPU (4 ms), I/O (2 ms), CPU (4 ms), I/O (2 ms), CPU (4 ms)
B: CPU (1 ms), I/O (2 ms), CPU (1 ms), I/O (2 ms), CPU (1 ms)

The I/O operations for the two processes do not interfere with each other and are blocking (i.e., the process cannot continue until the I/O completes). Both processes are ready to run at time 0.

1 - If the processes are run consecutively one after another, what is the elapsed time for both to complete?

2 - Sketch the execution pattern under non-preemptive scheduling and determine the total elapsed time for completion. You may assume that processes are scheduled in the order in which they become ready to run and that in the event of a tie A has priority over B. You may further assume that the scheduler and context switches take zero time.

3 – Repeat (2) but for a preemptive scheduler that operates on a time slice of 2 ms, that is, no process can run for more than 2 ms at a time (unless no other process is runnable).

4 – From your results of (2) and (3) what are the advantages and disadvantages of preemption for this scenario?

Question 6: Scheduling 2

The following processes arrive for execution at the times indicated. Each process runs for the amount of time listed. Assume the OS uses **non-preemptive scheduling**. Also, base all decisions on the information you have at the time the decision must be made.

Process	Arrival Time	Run Time
A	0.0	8
B	0.4	4
C	1.0	1

1 – What is the average turnaround time for these processes using FCFS scheduling?

2 – What is the average turnaround time for these processes with the SJF scheduling?

3 – Compute the average turnaround time if the CPU is left idle for the first 1 unit and then SJF scheduling is used. Processes A and B are waiting during this idle time.

Question 7: Semaphores

Consider a semaphore, as we saw in class. Show that, if the wait() and post() semaphore operations are not executed atomically, then mutual exclusion may be violated.

Question 8: Dining Philosophers Revisited

Remember the dining philosophers problem seen in class: 5 philosophers come to dinner and sit at a round table. In front of each philosopher, there is a plate of noodles. Between each plate, there is one chopstick. To be able to eat, philosophers need to hold the chopsticks on both sides of their plate. The dinner rules are as follows:

- There is an infinite food supply
- Philosophers can only be in one of two states: eating, or thinking.
- **Philosophers cannot talk to each other.**
- Philosophers cannot grab both chopsticks simultaneously (i.e., atomically). They can only grab one chopstick at a time.
- Assume that once a philosopher starts eating, they will only eat for a finite time (i.e., they cannot hold the chopsticks forever).

Our task is to find an algorithm where all the philosophers can forever alternate between eating and thinking. No philosopher should starve. In class, we have looked at several approaches to the dining philosophers problem. Now, your task is to formalize the algorithm.

1 – Using the synchronization primitives we have seen in class (i.e., locks, semaphores, condition variables), write the pseudo-code for a solution to this problem.

2 – Explain why this solution avoids deadlocks.