COMP 304 Operating Systems: Assignment 2

Due Lecture Time (2.30 pm), April 18, 2022

Notes: This is an individual assignment. All forms of cheating will be harshly punished! You may discuss the problems with your peers but the submitted work must be your own work. No late assignment will be accepted. Submit your answers through blackboard. This assignment is worth 3% of your total grade. Max possible grade: 100 points.

Contact TA: Ilyas Turimbetov (ENG 230)
iturimbetov18@ku.edu.tr

Problem 1

(20 points) Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst time	Priority
P1	7	3
P2	10	2
P3	3	3
P4	2	2
P5	5	1

Table 1: Table to test captions and labels.

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

(a) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 4 ms) scheduling. Assume the context switch overhead is **2 ms**.

(b) Calculate the waiting time of each process for each of the scheduling algorithms in part (a). Which of the schedules results in the minimal average waiting time?

(c) Calculate average turnaround time for each of the scheduling algorithms in part (a).

Problem 2

(10 points) Now assume there is a different CPU executing the same tasks as in Problem 1. This CPU takes **2ms more** time to execute each of the tasks, but the context switch overhead is **0.5 ms**.

(a) Calculate the CPU utilisation for all four scheduling algorithms in Problem 1.

(b) Does this CPU have a higher average waiting time than the one you calculated in Problem 1 part (b)? Provide an answer for each of the scheduling algorithms

Problem 3

(30 pts) For this question, use the code skeleton that is included in the assignment folder. In this code, you should complete the parts with a `TODO` comment and explain why you added/altered the code the way you did in the report. You should not change/add any code outside the explicitly mentioned `TODO` parts of the skeleton code both for Part A and B.
The provided skeleton code is a sample e-commerce application. The app has N items in `stock`. Every time the `sell()` function is called, one item is removed from stock and added to `sold` (Lines 35 to 50).

Part A: Racy Code

In this part, your goal is to fill in the `trigger_race_condition()` function so that when you run the code, more items are sold than available. We have added a `SLOWDOWN` macro in the middle of the critical section to increase the chances of context-switch happening at those points, and consequently, triggering a race-condition. The program will output a message if more stocks than available are sold.
The program creates new threads with the `create_new_thread()` function which executes the `sell()` method in a new thread, so that multiple instances of `sell()` run in parallel. Note that you may need to run the program several times for the race-condition to be triggered. Once you observe a race condition, include a screenshot as well as the explanations for the added code in the report.
To compile the code, use `gcc code.c -lpthread` which links the needed `pthread` library to the generated binary. You can then run the code via `./a.out`.

Part B: Fixing the race

Continuing from the code of Part A, you are required to ensure mutual exclusion to eliminate the race condition by using semaphores. APIs that create and use the semaphores are already provided in the skeleton code. Explain the changes in the code in your report.

Part C: Measuring time

Measure the time required to run the code for Part A and Part B separately, by adding a `time` command before program execution, e.g., `time ./a.out`. Report and explain the time differences.

Problem 4

(40 points) You run a pizza restaurant that has a policy not to charge for the orders that are not delivered in an hour. To be able to deliver on time, the orders with the furthest address are prioritized. There are 5 delivery guys working at the restaurant. Each of them can only deliver 1 order in a single run. Several orders can come at the exact same moment.

(a) Use the starter code provided for you below to write a monitor (in pseudocode) that allocates orders in a way described above, using the distance numbers to decide the delivery order. Once an order arrives, a function `request_delivery(int order_id, int distance)` is called. If all delivery cars are busy, the order is made to wait. When a delivery car comes back to the restaurant `release_car()` function is called. You can assume there is a function `sort()` that sorts an array of integers in descending order.

(b) Every time you fail to deliver on time, you lose money. Is the prioritization by distance policy putting you at the risk of bankruptcy? Justify your answer.

(c) Assume sometimes an order might be so big that you need several delivery guys at the same moment to be able to deliver it (for example, there is a party and the host serves pizza to the guests). This is reflected by a change to the `request_delivery(int order_id, int distance, int size)` function, which now takes in parameter `size` that shows the number of delivery cars needed to satisfy the order. Add this functionality to the monitor you wrote in part (a).

(d) Does the functionality you have added in part (c) affect the risk of bankruptcy? Justify your answer.

```
monitor deliveryMonitor {
    struct order {
       int id;
       int dist;
       int size;
    };

    int available_cars;  //number of available delivery guys
    struct order waiting_orders[M];
    int num_waiting; //number of orders currently waiting
    // !you can introduce shared variables

    // !implement this function
    void request_delivery(int order_id, int distance, int size)
    {
        waiting_orders[num_waiting].id = order_id;
        waiting_orders[num_waiting].dist = distance;
        waiting_orders[num_waiting].size = size;
    }

    void release_car() {
        // !implement this function
    }

    void initialize()
    {
        available_cars = 5;
        num_waiting = 0;
    }
}
```