CS 537: Intro to Operating Systems (Fall 2017)

Worksheet 3 - Scheduling & Process API

Due: Sep 27th 2017 (Wed) in-class OR email **Simmi** before 5:30 pm

1. **Basic Scheduling**
   Assume a workload with the following characteristics.

Process	Arrival Time (in sec)	Service Time (in sec)
A	0	8
B	2	4
C	5	7

For SJF, STCF, if there is a tie, schedule the process that arrived the earliest.
For RR, assume a time-slice of 1 sec. When a **new process** enters a particular queue, it is placed at the **end** of the round robin queue.

(a) Can you write the **schedule** (e.g., AABBCABC) in which these three processes may be scheduled for the following scheduling policies?

i. FIFO:

ii. SJF:

iii. STCF:

iv. RR:

(b) Fill in the **average turnaround time** (in sec) and **average response time** (in sec) of the three processes for the different schedulers given below.

	FIFO	SJF	STCF	RR
Average Turnaround Time				
Average Response Time				

(c) Assume an STCF scheduler for the original workload with processes A, B, and C. Assume a new process D requiring 5 seconds of CPU time arrives at some time T. What is/are the value(s) of T (assuming integer) such that process D **preempts** the process running at that time? Preemption means descheduling a process that is currently running (and not yet complete) and scheduling a different process.

(d) Assume an STCF scheduler for the original workload with processes A, B, and C. Assume a new process E arrives at time 8 seconds. What could be the **maximum** CPU burst time or service time (assuming integer) for process E such that it preempts the process that was running at time 8?

(e) With the round robin (RR) scheduling policy, a question arises when a new job arrives in the system: should we put the job at the front of the RR queue, or the back? Does this decision make a difference, or does RR behave pretty much the same way either way? (Explain)

(f) Why is STCF considered one of the best scheduling policies? What does it do? If it is so great, why do we rarely see it implemented in real world schedulers?

2. **Multi-Level Feedback Queue**
Assuming processes A, B, and C are **CPU-bound** (no I/O). There is **no overhead** for context switching. When a **new process** enters a particular queue, it is placed at the **end** of the round robin queue.
Consider a **Multi-Level Feedback Queue (MLFQ)** Scheduler with **3 queues** as shown below. After a time interval of **10 seconds**, all processes are **boosted up** to the **top-most queue** with the highest priority.

queue #	**priority**	**Round Robin time slice**
2	2 (highest)	1
1	1	2
0	0 (lowest)	3

The table below gives the details of the **workload**.

Process	**Arrival Time**	**Service Time**
A	0	10
B	3	3
C	5	4

(a) Fill the table below with the details of which process will be scheduled for each time unit and at which priority level. In this table, for each time unit n, write which process will be scheduled from time n to $n + 1$.

	Time Unit																
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Priority 2																	
Priority 1																	
Priority 0																	

(b) Fill the table below with the **turnaround time** and **response time** of each process.

Process	**Turnaround Time**	**Response Time**
A		
B		
C		

3. **Process API**
You may assume that the following two code snippets compile successfully and APIs like **fork**(), **exec()**, and **wait()** never fails. Try NOT to run the code without thinking about it. You may run them to verify your answers, though. Remember, you won't have your best friend, the C compiler, with you during your exam! :)

Program 1

```
int counter = 0;
int main() {
    int i;
    for (i = 0; i < 3; i++) {
        if (fork() == 0) {
            printf("Hello\n");
            counter++;
        }
    }
    printf("%d\n", counter);
}
```

Program 2

```
int counter = 0;
int main() {
    int i;
    for (i = 0; i < 3; i++) {
        if (fork() == 0) {
            char *myargs[3];
            myargs[0] = "echo";
            myargs[1] = "Hello";
            myargs[2] = NULL;
            execvp(myargs[0], myargs);
            counter++;
        }
    }
    for (i = 0; i < 3; i++) {
        wait(NULL);
    }
    printf("%d\n", counter);
}
```

(a) After program 1 is executed, how many processes are created?

(b) After program 2 is executed, how many processes are created?

(c) Draw a process tree diagram, with the `counter` value printed out by the process in the parenthesis beside that process, after **program 1** is executed. If a process does NOT print the `counter` value, write `NULL`.

For example, the following diagram represents that process A creates process B and process C, and that process C creates process D; the `counter` value printed out by process A is 0, B is 1, D is 3 and C does NOT print the `counter` value. In your process tree diagram, there is no specific requirement for labeling a process as long as different processes are labeled with distinct characters.

```
   A(0)
  /    \
B(1)  C(NULL)
         \
         D(3)
```

(d) Can you draw a similar process tree diagram for **program 2**?