Operating systems
Midterm examination
2009/2010

Student Name:

NIA:

1. (3pt) Given the following set of processes:

Process	Burst Time (ms)	Priority
*P*1	10	3
*P*2	1	1
*P*3	2	3
*P*4	1	4
*P*5	5	2

The processes are assumed to have arrived in the order *P*1, *P*2, *P*3, *P*4, *P*5, 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 = 1) scheduling.

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

c. What is the waiting time of each process for each of the scheduling algorithms in part a?

d. Which of the schedules in part a results in the minimal average waiting time (over all processes)?

Answer:

a. The four Gantt charts are

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
FCFS	1	1	1	1	1	1	1	1	1	1	2	3	3	4	5	5	5	5	5
RR	1	2	3	4	5	1	3	5	1	5	1	5	1	5	1	1	1	1	1
SJF	2	4	3	3	5	5	5	5	5	1	1	1	1	1	1	1	1	1	1
Priority	2	5	5	5	5	5	1	1	1	1	1	1	1	1	1	1	3	3	4

b. Turnaround time

	FCFS	RR	SJF	Priority
*P*1	10	19	19	16
*P*2	11	2	1	1
*P*3	13	7	4	18
*P*4	14	4	2	19
*P*5	19	14	9	6

c. Waiting time (turnaround time minus burst time)

	FCFS	RR	SJF	Priority
*P*1	0	9	9	6
*P*2	10	1	0	0
*P*3	11	5	2	16
*P*4	13	3	1	18
*P*5	14	9	4	1

d. Shortest Job First

2. (1pt) Which of the following scheduling algorithms could result in starvation? Why?

a. First-come, first-served
b. Shortest job first
c. Round robin
d. Priority

Answer: Shortest job first and priority-based scheduling algorithms could result in starvation.

3. (1pt) Draw a state transition diagram of process execution states. Can a process transition from waiting for an I/O operation to the terminated state? Why or why not?

Answer: No. A process waiting for I/O must first enter the ready queue and then to the running state before it may terminate.

4. (0.5pt) What is the essential cause of the difference in cost between a context switch for

kernel-level threads and a switch that occurs between user-level threads?

Answer: Kernel-level threads require a system call for the switch to occur; user-level threads do not.

5. (0.5 pt) Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values
b. Heap memory
c. Global variables
d. Stack memory

Answer: The threads of a multithreaded process share heap memory and global variables. Each thread has its separate set of register values and a separate stack.

6. (0.5pt) The threading model supported by the typical Linux kernel is:

(a) one-to-one
(b) one-to-many
(c) many-to-one
(d) many-to-many
(e) two-level
(f) all of the above

Answer: (a)

7. (0.5pt) Explain the difference between “user mode” and “kernel mode” execution.

Answer: These are types of execution modes in an operating system.
User mode conventional operating mode for user apps, which have restricted privileges for what they can do on the system
Kernel mode: provides raw access to physical hardware for the OS; process has full privileges for what it can do on the system
System calls allow a user process to transition from user mode to kernel mode for use of specific OS services.

8. (3pt) Given the following code:

```
main() {
        printf(“Hello ”);
        fork();
        printf(“world”);
        fork();
        printf(“!”);
}
```

(a) Draw the process tree.
(b) What will the program print? Why?
(c) Is the order of printing the same for all executions? Justify your answer.
(d) Modify the program in such a way that it **creates exactly 3 processes** (including father) the father prints “Hello”, a child prints “world” and a child of a child prints “!” **exactly in this order**.

Answer:

(a) The father creates 2 children. One child creates another child.
(b) Hello world world !!!!
(c) The order may be different depending on the scheduler.
(d)

```
main() {
        printf(“Hello ”);
        int p=fork();
        if (p==0) {
                printf(“world”);
                exit(0);
        }
        else {
                wait(NULL);
                p=fork();
                if (p==0) {
                        printf(“!”);
                        exit(0);
                }
                else
                        wait(NULL);
        }
}
```