CS330 - Quiz 3

Total: 15 marks Duration: 25 mins

open books and notes, no mobile phones, friends disconnected during exam

Be precise, no marks for vague answers

Roll No:

Q1. Select the correct answer(s) for the following questions. No partial marks. (2×4)

(i) In a uni-processor system, which of the following statement(s) are true regarding threads of a multi-threaded process?

(a) More than one thread can be inside system call handler only if they have separate OS stacks.

(b) More than one thread can be in RUNNING state if they have separate OS stacks.

(c) More than one thread can be inside system call handler if interrupts are disabled during system calls.

(d) More than one thread can be inside system call handler if context switching is disabled during system calls.

(e) None of the above.

Ans: a,c

(ii) Which of the following statement(s) are true?

(a) If a process is interrupted during system call handling, the OS will crash.

(b) If a process is de-scheduled during system call handling, it must resume its execution in OS mode.

(c) If a process is waiting for an I/O event, it must resume its execution in OS mode.

(d) A multitasking OS can disable interrupts whenever the CPU is executing in OS mode.

(e) None of the above.

Ans: c

(iii) Which of the following statement(s) are true?

(a) Multi-level feedback queue scheduling favors I/O bound processes.

(b) Starvation problem in multi-level feedback queue scheduling can be addressed by carefully tweaking the value of time quantum.

(c) Dynamically determining the value of time quantum can address the issue of a malicious user tricking the multi-level feedback queue scheduler.

(d) Standard deviation of waiting time can be used to measure the fairness of any scheduling algorithm.

(e) For a given scheduling algorithm, average waiting time monotonically increases with number of processes.

Ans: a, b, c, d

(iv) Which of the following statement(s) are true?

(a) There can be a common signal handler function for multiple signals.

(b) If two processes execute `kill(pid, SIGSEGV)` around the same time, one of the calls may fail.

(c) If two processes execute `kill(pid, SIGSEGV)` around the same time, the target process (with PID=pid) may execute the registered signal handler twice.

(d) If two processes execute `kill(pid, SIGSEGV)` around the same time, the target process (with PID=pid) may execute the registered signal handler only once.

(e) Within a signal handler function, a signal can be generated to another processes using the `kill` system call.
Ans: a, c, d, e

Q2. Consider the following C program. (4)

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<signal.h>
4 char *ptr1 = NULL;
5 char *ptr2 = NULL;
6 void sighandler(int signo)
7 {
8     printf("I am in sighandler\n");
9     ptr1 = ptr2;
```

```
10 }
11
12 main()
13 {
14    if(signal(SIGSEGV, &sighandler) < 0)
15         perror("signal");
16
17    ptr2 = (char *)malloc(100);
18    ptr2[0] = 'A';
19    printf("%c\n", *ptr1);
20 }
```

What will be the output of this program when executed in a UNIX/Linux OS? Explain.

Ans: The program will print `I am in sighandler` infinitely.
Reason: The program counter (RIP) will point to the instruction that dereferences memory location pointed to by `ptr1`. The memory address `ptr1` is stored in a register (e.g., RAX) before it can be dereferenced. Note that, you have seen this behavior in Assignment-II for page fault handler implementation where if the register state is not saved and restored, the accessed memory location changed when the instruction gets re-executed after exception handling. Therefore, after the page fault and `sighandler` execution, the value of the register involved remain unchanged and causes repeated SEGFAULTs.

Q3. Consider the following C program. (3)

```
 1 #include<stdio.h>
 2 #include<stdlib.h>
 3 #include<unistd.h>
 4 #include<fcntl.h>
 5 main()
 6 {
 7     char buf[8];
 8     int fd, dfd;
 9     fd = open("some.txt", O_RDWR);
10     if(fd < 0){
11          perror("open");
12          exit(-1);
13     }
14     dfd = dup(fd);
15     buf[7] = 0;
16     read(fd, buf, 7);
17     printf("%s\n", buf);
18     read(dfd, buf, 7);
19     printf("%s\n", buf);
20 }
~
```

Assume that `some.txt` is a valid file whose content is: `AAAAAAABBBBBBBCCCCCCC...`. What will be the output of the above program when executed in the UNIX OS? (no explanation needed)

Ans:
```
AAAAAAA
BBBBBBB
```
The `dup` system call simply create a duplicate file descriptor, the file object pointed to by the two file descriptors remains the same. (See slide-17, IPC.pdf)