**ECS150 - Homework 1**

**Short Answer Questions**

1. False.

Time-sharing was not widespread on second-generation computers because the hardware was not mature enough to implement it. These computers could not adequately protect the OS from unwanted users and programs. Additionally, batch processing was preferred during this generation of computing.

1. Shortest Job Next

Firstly, SJN has the smallest average turnaround time out of all the non-preemptive priority functions. FIFO is not ideal due to the “convoy effect”, which describes a situation wherein shorter processes could get stuck behind a time and resource-heavy process. One could also consider HRRN if jobs did not arrive at the same time. However, since arrival time will be 0 for all jobs, HRRN’s priority function will result in 0 for all jobs, thus defaulting to either FIFO or random scheduling by the arbitration rule. This is certainly worse than SJF which depends only on the required service time.

1. (c) is True. The rest are False.
2. MLFQ attempts to address the pitfall that you cannot predict a job’s completion time in advance, so this statement is False.
3. The MLFQ is adaptive, such that process priorities are changed according to how processes are behaving over time. The MLFQ observes this behavior and adjusts priorities over time, it is not an automatic process. So, this statement is False.
4. The MLFQ utilizes the preemptive decisions at quanta to approximate SJF, so this statement is True
5. Low priority processes can suffer from starvation, which causes delays, so this statement is False.
6. c) fork(), exec(), wait()

Commands are handled by process-oriented system calls. The shell is a user program, so it prompts the user for an input and waits for a command. The shell then navigates through the file system to find the inputted executable and calls fork() to create a new child process to run the specified command. The command then calls a variant of exec() and waits for the command to complete by calling wait(). Once the child completes, the shell returns from wait() and awaits the next command.

**Long Answer Questions**

1. The following protections are necessary:

b. Change to kernel mode. – A user could simply switch to kernel mode and execute privileged operations otherwise

c. Read from Kernel memory – This poses a security threat as the kernel memory could contain pieces of code from other processes

d. Write into Kernel memory – This poses a security threat as users can mess with other processes or the Operating System itself

e. Instruction fetch from kernel memory – although the instruction itself could be protected (i.e. it can be fetched but is protected from being executed), a security threat is posed when users are able to access kernel memory

f. Turn on timer interrupt – Turning on the timer takes away control from the OS for a period until the timer runs out. If processes can raise interrupts as they like, they could potentially repeatedly take away control from the OS and undertake malicious activity during that time.

g. Turn off timer interrupt – Turning the timer on ensures that control will eventually be given back to the OS once the timer runs out. If users are allowed to turn off the timer however, the OS will be unable to regain control, thus, posing a security threat. Additionally, time sharing cannot take place.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **time** | **run** | **q0 // 1** | **q1 // 3** | **order** |
| **0** | A | A(5,1) |  |  |
| **1** | B | B(2,1) | A(4,3) |  |
| **2** | A |  | A(4,3),B(1,3) |  |
| **3** | A | C(7,1) | A(3,2),B(1,3) |  |
| **4** | A | C(7,1) | A(2,1),B(1,3) |  |
| **5** | C | C(7,1),D(1,1) | B(1,3),A(1,0) |  |
| **6** | D | D(1,1) | B(1,3),A(1,0),C(6,3) | D completes at time 6-7 |
| **7** | E | E(4,1) | B(1,3),A(1,0),C(6,3) |  |
| **8** | B |  | B(1,3),A(1,0),C(6,3),E(3,3) | B completes at time 8-9 |
| **9** | A |  | A(1,0),C(6,3),E(3,3) | A completes at time 9-10 |
| **10** | C |  | C(6,3),E(3,3) |  |
| **11** | C |  | C(5,2),E(3,3) |  |
| **12** | C |  | C(4,1),E(3,3) |  |
| **13** | E |  | E(3,3),C(3,0) |  |
| **14** | E |  | E(2,2),C(3,0) |  |
| **15** | E |  | E(1,1),C(3,0) | E completes at time 15-16 |
| **16** | C |  | C(3,0) |  |
| **17** | C |  | C(2,0) |  |
| **18** | C |  | C(1,0) | C completes at time 18-19 |
| **19** |  |  |  |  |

1. Though there are multiple instances of the same program, each instance is a separate process.
2. One instance of the program can share instructions with another instance if instructions have their own protections (i.e., they can be fetched but are protected from being executed). Additionally, if instructions lie in critical sections, mutual exclusion, progress, and bounded waits must be ensured to maintain consistency and prevent race conditions.
3. Instances of the same program can share data in memory as long as the shared data is synchronized to maintain consistency. Two processes may read the same data, but if one process starts writing while the other is reading, the results can be inconsistent. Hence, precautions such as locks and atomic hardware instructions should be used to guarantee correctness of data in memory.