**Hand in this answer sheet only!**

**（The table for Q6 is attached at the end.）**

|  |
| --- |
| 1. Periodic timer interrupts can be employed to track the execution status of instructions or specific code segments. By observing which code segments are active during these interrupts, one can establish a statistical profile aligning with the program's time allocation across its various sections. Once this statistical profile is available, the programmer can optimize those code sections that are utilizing a significant portion of CPU resources. |
| 1. To guarantee the effective operation of an operating system, most computer systems incorporate hardware mechanisms for distinguishing between user mode and kernel mode. This distinction is established through a dedicated mode bit integrated into the computer's hardware, representing either kernel mode (0) or user mode (1) at any given time. When the computer executes a user application, it operates in user mode. Nevertheless, when a user application seeks a service from the operating system, typically through a system call, it necessitates a transition from user mode to kernel mode to fulfill the request. |
| 1. The system-call interface within a programming language acts as a bridge to access the system calls provided by the underlying operating system. This interface intervenes when function calls are made to the API and triggers the corresponding system calls within the operating system. Consequently, the intricate aspects of the operating-system interface are shielded from the programmer by the API, and they are handled by the runtime support library. System calls, being intricately detailed, can be more challenging to work with directly. Utilizing APIs enables programs to compile and run on different systems, enhancing portability. |
| 1. User-level threads are highly cost-effective in terms of resource usage. Creating, destroying, and switching among them does not impose a significant burden on system resources and doesn't interrupt the CPU. In contrast, kernel-supported threads are more resource-intensive. They require system calls for creation and destruction, and the kernel must manage their scheduling to share CPU access. However, they possess greater capabilities as they are independently scheduled and can block individually.  * Many-to-One: This model involves mapping many user-level threads to a single kernel thread. * One-to-One: In this model, each user-level thread is directly mapped to a kernel thread. * Many-to-Many: This model allows multiple user-level threads to be mapped to multiple kernel threads. * Two-level Model: Like Many-to-Many, this model permits a user thread to be bound to a specific kernel thread, offering more control over thread management. |
| a. CPU Utilization:   * The algorithm(s) with the maximum CPU utilization is/are Round Robin (RR). * RR ensures that all processes get a fair share of the CPU time, resulting in high utilization.   b. Average Waiting Time:   * The algorithm(s) with the minimum average waiting time is/are Shortest Job First (SJF), both non-preemptive and preemptive versions. * SJF minimizes waiting time by selecting the shortest job, which reduces the overall waiting time for processes.   c. Fairness:   * The algorithm(s) that is/are the fairest is/are Round Robin (RR). * RR provides fairness by giving each process an equal time quantum, ensuring that no process is starved for CPU time. |
| 1. (a) 13   (b) If interrupts occur in each instruction, it is not going to get the same value at the shared memory 2000 due to the non-atomic nature of memory access and the lack of synchronization between the threads. The critical section that needs to be protected is the part of code where memory 2000 is accessed and modified such as:  For thread\_1:  mov 2000, %dx  add $1, %dx  mov %dx, 2000  For thread\_2:  mov 2000, %dx  add $2, %dx  mov %dx, 2000 |
| 1. In the initial step, thread 0 checks if the %ax register is equal to 1. This condition holds true for the specific value of ax in thread 0, leading it to proceed to the signaling section, where it stores the value 1 in memory location 2000. Once thread 0 completes its task, thread 1 takes over. In this case, the condition for jumping to the signaling section is not met, so thread 1 continues to execute the waiter section. Here, it retrieves the value from memory location 2000 and verifies that it is not equal to 1. Since thread 0 has already set the memory address 2000 to 1, thread 1 does not jump and continues its execution. The value at location 2000 serves as a flag to determine whether the signaling or waiting operation should be executed, and its final value is 1. |
|  |
| 1. Initially, thread 0 begins by checking if the %ax register equals 1, which is not the case for the initial %ax value. Consequently, thread 0 proceeds to execute the waiter section. In this section, it retrieves the value from memory location 2000 (which is initially 0) and checks if it's not equal to 1. If the condition is met, thread 0 jumps back to the beginning of the waiter section, creating a loop that continues until it is interrupted. Therefore, thread 0 remains in this loop. Thread 1 enters the picture and changes the condition by executing the signaler section, setting the value at memory location 2000 to 1. Now, when thread 0 reaches the last "test" in the waiter section, it finds that %cx and 1 are equal, preventing the subsequent jump. Consequently, the program finally halts. Therefore, increasing the interrupt interval to a larger number will extend the number of loop iterations for thread 0 and increase the waiting time for thread 1. However, this program is not making efficient use of the CPU, as thread 0 continuously spins in a loop, wasting CPU time. |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Question 6:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | A | B | C | D |
| FCFS | Finish | 7 | 11 | 15 | 18 |
| Response | 0 | 9 | 11 | 9 |
| Turnaround | 7 | 9 | 11 | 9 |
| SJF | Finish | 7 | 11 | 15 | 18 |
| Response | 0 | 5 | 7 | 6 |
| Turnaround | 7 | 9 | 11 | 9 |
| RR | Finish | 13 | 18 | 23 | 25 |
| Response | 0 | 1 | 5 | 6 |
| Turnaround | 7 | 9 | 11 | 9 |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| FCFS | A | A | A | A | A | A | A | B | B | B | B | C | C | C | D | D | D | D |  |
| SJF | A | A | A | B | B | B | B | C | C | C | C | D | D | D | C | C | D | D |  |
| RR | A | B | C | D | A | B | C | D | C | D | A | B | C | D | C | D | D | D |  |