|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1.Show that the following algorithm satisfies all the requirements for the critical section.(10)    **public** **class** Algorithm **extends** MutualExclusion {  **public** Algorithm() {  flag[0] = **false**;  flag[1] = **false**;  turn = TURN\_0;  }    **public** **void** *enteringCriticalSection*(**int** t) {  **int** other;  other = 1 - t;  flag[t] = **true**;  turn = other;  **while** ( (flag[other] == **true**) && (turn == other) )  Thread.yield();  }    **public** **void** *leavingCriticalSection*(**int** t) {  flag[t] = **false**;}  ANSWER:  Mutual Exclusion: The while loop in the critial function preserves mutual exclusion. It will keep the process who's turn it's not stuck in the while loop until the flags allow it through.  Progress: If the process doesn't get stuck in the loop than it can proceed with the critical section. This ensures the process requirement.  Bounded waiting: The processes will take turns runing the critical section as the flags are changing.  2 .Servers can be designed to limit the number of open connections. For example, a server may wish to have only *N* socket connections at any point in time. As soon as *N* connections are made, the server will not accept another incoming connection until an existing connection is released. Explain and write  how semaphores can be used by a server to limit the number of concurrent connections (15)  ANSWER:  A semaphore is initialized to the number of allowable open socket connections. When a connection is accepted, the acquire() method is called; when a connection is released, the release() method is called. If the system reaches the number of allowable socket connections, subsequent calls to acquire() will block until an existing connection is terminated and the release method is invoked.   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | 3. Consider the following set of processes, with the length of the CPU burst given in milliseconds (20)  Process                Burst Time          Priority   P1                               5                        2  P2                                3                        1   P3                              11                      4   P4                               7                        2  P5                                8                        3   The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.   a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).   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 these scheduling algorithms?  d. Which of the algorithms results in the minimum average waiting time (over all processes)?   |  |  | | --- | --- | |  |  |   Answer: a. The four Gantt charts are  P1, p2,p3,p4,p5 FCFS waitingtime = (0 + 5 +8+19+26)/5=11.6 =(   |  |  |  |  |  | | --- | --- | --- | --- | --- | | P1 2 5 | P2 1 3 | P3 4 11 | P4 2 7 | P5 3 8 |   5 8 19 26 34   |  |  |  |  |  | | --- | --- | --- | --- | --- | | P2 | P1 | P4 | P5 | P3 |   3 8 15 23 34  SJF waiting time = (0+ 3+ 23+ 8 +15)/5 = 9.8  RR turnaround time p1 =20 p2 = 11 p3 =34 p4 =29 p5 =31  p1, p2, p3, p4, p5 p1, p2, p3, p4, p5 p1, p3, p4, p5 p3, p4, p5 p3, p3  RR waiting time p1 = 15, p2 = 8, p3= 23, p4 = 21, p5=24  Prioriy p2 , p1, p4, p5 p3 turnaround time =(3+ 0+8+23+15)/5=49/5=9.8  Turn around time p1= 11 p2= 3 p3=34 p4= 15 p5=23  Waiting time p1= 3 p2=0 p3=23 p4=8 p5= 15  d. Shortest Job First SJF   |  |  |  |  | | --- | --- | --- | --- | |  | 4. Consider the exponential average formula used to predict the length of  the next CPU burst. What are the implications of assigning the following  values to the parameters used by the algorithm?  a. a = 0.02 and t0 = 100 milliseconds  b. a = 0.89 and t0 = 10 milliseconds  ANSWER:  Answer: When a = 0.02 and t0 = 100 milliseconds, the formula almost makes a prediction of 100 milliseconds for the next CPU burst. When a = 0.89 and t0 = 10 milliseconds, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memoryless, and simply predicts the length of the previous burst for the next quantum of CPU execution. | | | |  |  | | | Ungraded | 5. In the following situation deadlock is prevented by ensuring locks are all acquired in a certain order. However, the **TTTT?** function illustrates a scenario where deadluck might happen. Fix the **TTTT?**  function to prevent deadlocks. The **TTTT?** function is as follows:    **Void TTTT?  (Account  from, Account to, double amount)**  **{S**emaphore lock1, lock2, lock3;  wait(lock3);  **Lock1  = getLock(from);**  **Lock2  = getLock(to);**  **Wait(Lock1);**  **Wait(Lock2);**  **Take (from,  amount);**  **PUT (to ,  amount);**  ignal(lock3);  **Signal(Lock2);**  **Signal(Lock2);**  **}** | | | | |  | |  | |   6. A ROOM is to be shared among different People, each of which has a unique number. The ROOM can be accessed simultaneously by several People, subject to the following constraint: The sum of all unique numbers associated with all the people currently accessing the Room must be less than P. Write a monitor to coordinate access to the Room.  Answer: The pseudocode is as follows:  monitor People\_ access {  int curr sum = 0;  int n;  condition c;  void access People(int my num) {  while (curr sum + my num >= n)  c.wait();  curr sum += my num;  }  void Leave (int my num) {  curr sum -= my num;  c.broadcast();  }  }   |  |  |  | | --- | --- | --- | | Ungraded | 7. The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:    int available\_resources = 10;  When a process wishes to obtain a number of resources, it invokes the decrease  function  int decrease (int N) {      if (available\_resources < N) return -1;   else { available\_resources -= N; return 0;      }  }  When a process wants to return a number of resources, it calls the increase  function:     int increase (int N) {     available resources += N;     return 0; }   The preceding program segment produces a race condition. Do the following:   a. Identify the data involved in the race condition.   b. Identify the location (or locations) in the code where the race condition occurs.   c. Using a semaphore or mutex lock, fix the race condition. It is permissible to modify the decrease  function so that the calling process is blocked until sufficient resources are available.  ANSWER  . Identify the data involved in the race condition: The variable available resources.  • Identify the location (or locations) in the code where the race condition occurs: The code that decrements available resources and the code that increments available resources are the statements that could be involved in race conditions  • Using a semaphore, fix the race condition: Use a semaphore to represent the available resources variable and replace increment and decrement operations by semaphore increment and semaphore decrement operations. | | |  |  |   8. Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every 2 milliseconds of CPU computing and that each I/O operation takes 20 milliseconds to complete. Also assume that the context switching overhead is 0.2 millisecond and that all processes are long-running tasks. What is the CPU utilization for a round-robin scheduler when:   a. The time quantum is 2 millisecond   b. The time quantum is 20 milliseconds  AnSWER   1. The time quantum is 2 millisecond: Irrespective of which process is scheduled, the scheduler incurs a 0.2 millisecond context-switching cost for every context-switch. This results in a CPU utilization of 2/2.2 \* 100 = 91%.   b. The time quantum is 20 milliseconds: The I/O-bound tasks incur a context switch after using up only 2 millisecond of the time quantum. The time required to cycle through all the processes is therefore 10\*2.2 + 20.2 (as each I/O-bound task executes for 2 millisecond and then incur the context switch task, whereas the CPU-bound task executes for 20 milliseconds before incurring a context switch). The CPU utilization is therefore 40/42.2 \* 100 = 94.78% | | |  |  | | |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | 4. Consider the exponential average formula used to predict the length of  the next CPU burst. What are the implications of assigning the following  values to the parameters used by the algorithm?  a. a = 0.02 and t0 = 100 milliseconds  b. a = 0.89 and t0 = 10 milliseconds  ANSWER:  Answer: When a = 0.02 and t0 = 100 milliseconds, the formula almost makes a prediction of 100 milliseconds for the next CPU burst. When a = 0.89 and t0 = 10 milliseconds, the most recent behavior of the process is given much higher weight than the past history associated with the process. Consequently, the scheduling algorithm is almost memoryless, and simply predicts the length of the previous burst for the next quantum of CPU execution. | | |
|  |  | |
| Ungraded | 5. In the following situation deadlock is prevented by ensuring locks are all acquired in a certain order. However, the **TTTT?** function illustrates a scenario where deadluck might happen. Fix the **TTTT?**  function to prevent deadlocks. The **TTTT?** function is as follows:    **Void TTTT?  (Account  from, Account to, double amount)**  **{S**emaphore lock1, lock2, lock3;  wait(lock3);  **Lock1  = getLock(from);**  **Lock2  = getLock(to);**  **Wait(Lock1);**  **Wait(Lock2);**  **Take (from,  amount);**  **PUT (to ,  amount);**  ignal(lock3);  **Signal(Lock2);**  **Signal(Lock2);**  **}** | | | |
|  | |  | |