**COMP-304 OPERATING SYSTEMS ASSIGNMENT-2**

**1-)**

**a-)**

**For the First-Come-First-Serve Scheduling Algorithm:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P0** |  | **P1** |  | **P2** |  | **P3** |  | **P4** |

**0 14 15 23 24 44 45 49 50 56**

**For the Shortest Job First (SJF) Scheduling Algorithm**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P1** |  | **P3** |  | **P4** |  | **P0** |  | **P2** |

**0 8 9 13 14 20 21 35 36 56**

**For the Non-Preemptive Priority Scheduling Algorithm**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P1** |  | **P0** |  | **P4** |  | **P3** |  | **P2** |

**0 8 9 23 24 30 31 35 36 56**

**For the Round-Robin Scheduling Algorithm**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **P0** |  | **P1** |  | **P2** |  | **P3** |  | **P4** |  | **P0** |  | **P1** |  | **P2** |  | **P0** |  | **P2** |  | **P2** |

**0 6 7 13 14 20 21 25 26 32 33 39 40 42 43 49 50 52 53 59 60 62**

**b-)**

**FOR FCFS Scheduling ALGORITHM**

**Waiting time of the process= Start time of process – Arrival time of the process**

**Waiting time of P0= (0-0) unit= 0 unit**

**Waiting time of P1= (15-0) units= 15 units**

**Waiting time of P2= (24-2) units= 22 units**

**Waiting time of P3= (45-4) units= 41 units**

**Waiting time of P4= (50-6) units= 44 units**

**Average waiting time for FCFS Scheduling algorithm: (0+15+22+41+44)/5= 122/5= 24.4**

**For the Shortest Job First (SJF) Scheduling Algorithm**

**Waiting time of the process= Start time of the process – Arrival time of the process**

**Waiting time of P1: (0-0) unit= 0 unit**

**Waiting time of P3: (9-4) units= 5 units**

**Waiting time of P4: (14-6) units= 8 units**

**Waiting time of P0: (21-0) units= 21 units**

**Waiting time of P2: (36-2) units= 34 units**

**Average waiting time for SJF Scheduling algorithm: (0+5+8+21+34)/5= 68/5= 13.6**

**For the Non-Preemptive Priority Scheduling Algorithm**

**Waiting time of the process= Start time of the process – Arrival time of the process**

**Waiting time of the P1: (0-0) unit= 0 unit**

**Waiting time of the P0: (9-0) units= 9 units**

**Waiting time of the P4: (24-6) units= 18 units**

**Waiting time of the P3: (31-4) units= 27 units**

**Waiting time of the P2: (36-2) units= 34 units**

**Average waiting time for Priority Scheduling: (0+9+18+27+34)/5= 88/5= 17.6 units**

**For the Round-Robin Scheduling Algorithm**

**Waiting time of P0: (33-6)+(50-39)= (27+11)= 38 units**

**Waiting time of P1: (7-0)+(40-13)= 7+27=34 units**

**Waiting time of P2: (14-2)+(43-20)+(53-49)=12+23+4= 35+4= 39 units**

**Waiting time of P3: (21-4) = 17 units**

**Waiting time of P4: (26-6) = 20 units**

**Average waiting time For the Round-Robin Algorithm: (38+34+39+17+20)/5= 29.6 units**

**Shortest Job First(SJF) Scheduling Algorithm results in the minimal average waiting time.**

**c-)**

**FOR FCFS Scheduling ALGORITHM**

**Waiting time+CPU burst time-Arrival time= Turn Around Time**

**For P0: 0+14-0= 14 units**

**For P1: 15+8-0= 23 units**

**For P2: (24+20-2)= 42 units**

**For P3: (45+4-4)= 45 units**

**For P4: (50+6-6)= 50 units**

**Average Turn Around Time for FCFS= (14+23+42+45+50)/5= 174/5= 34.8**

**For Shortest Job First(SJF) Algorithm**

**Waiting time+CPU burst time-Arrival time= Turn Around Time**

**For P1: (0+8-0) units= 8 units**

**For P3: (9+4-4) units= 9 units**

**For P4: (14+6-6) units= 14 units**

**For P0: (21+14-0) units= 35 units**

**For P2: (36+20-2) units= 54 units**

**Average Turn Around Time for SJF= (8+9+14+35+54)/5= 120/5= 24 units**

**For the Non-Preemptive Priority Scheduling Algorithm**

**Waiting time+CPU burst time-Arrival time= Turn Around Time**

**For P1: (0+8-0) units= 8 units**

**For P0: (9+14-0) units= 23 units**

**For P4: (24+6-6) units= 24 units**

**For P3: (31+4-4) units= 31 units**

**For P2: (36+20-2) units= 54 units**

**Average Turn Around Time = (8+23+24+31+54)/5= 140/5= 28 units**

**For the Round-Robin Scheduling Algorithm**

**Waiting time+CPU burst time-Arrival time= Turn Around Time**

**I have considered all of the possible starts and ends of the processes while calculating the turn around times of P0, P1, P2, P3, and P4.**

**For P0: (0+6-0)+((33-6)+(39-33))+((50-39)+(52-50))= 6+(27+6)+(11+2)= 6+33+13= 52 units**

**For P1: ((7-0)+(13-7))+((40-13)+(42-40))= (7+6)+(27+2)= 13+29= 42 units**

**For P2: ((14-2)+(20-14))+((43-20)+(49-43))+((53-49)+(59-53))+((60-59)+(62-60))**

**: (12+6)+(23+6)+(4+6)+(1+2)= 18+29+10+3= 60 units (for P2)**

**For P3: ((21-4)+(25-21))= 17+4= 21 units**

**For P4: ((26-6)+(32-26))= 20+6= 26 units**

**Average Turn-Around Time: (52+42+60+21+26)/5= 201/5= 40.2**

**Shortest Job First(SJF) scheduling algorithm has the lowest average turn around time.**

**d-) For the non-preemptive scheduling algorithms, the average response time is equal to the average waiting time. In the question, non-preemptive version of the SJF algorithm is considered.**

**So, the average response time is 13.6 (found in part B) for SJF scheduling algorithm.**

**So, the average response time is 17.6 (found in part B) for non-preemptive priority scheduling algorithm.**

**So, the average response time for FCFS algorithm:**

**(0+15+(24-2)+(45-4)+(50-6))/5= (0+15+22+41+44)/5= 122/5= 24.4(It is equal to the average waiting time found in part B.)**

**The average response time for RR algorithm: (RR is preemptive)**

**(0+7+(14-2)+(21-4)+(26-6))/5= (0+7+12+17+20)/5= 56/5= 11.2**

**The Round Robin (RR) scheduling algorithm is the best in terms of the response time.**

**2-) The explanations of part-A, part-B, and part-C are under the screenshots.**

**A close-up of a graph

Description automatically generated with low confidenceTHE SCREENSHOT WITH THE RACE CONDITION**

**A picture containing text, appliance

Description automatically generatedTHE SCREENSHOT WITHOUT THE RACE CONDITION(THE FIXED VERSION)**

**THE EXPLANATIONS OF PART-A&PART-B&PART-C**

**In the init( ) function, I have written the sem\_init(&smp,0,1) function. The first parameter of this function is the address of the global variable called ‘smp’. The second parameter is an integer variable. If this parameter is equal to 0, it shows that the threads of a process share the semaphore called ‘smp’. If this parameter is not equal to 0, it indicates that multiple processes share the semaphore called ‘smp’. The third parameter is the initial value of the semaphore called ‘smp’.**

**//START OF THE PART-A**

**In the trigger\_race\_condition() function, I have created multiple threads. For creating a thread, I have used create\_new\_thread() function. As a parameter, I have passed the address of the sell() function(&sell). When I do not put lock() and unlock() function calls to the necessary places in the \*sell() function, the race condition happens. When there is a race condition, the threads share, and change the data. For this question, the shared datas are the integer variable which has a name ‘stock’ and the integer variable which has a name ‘sold’. Moreover, for the race condition case, the manipulations of these shared datas are done at the same time in multiple threads. For creating multiple threads, I have used the create\_new\_thread(&sell) function in a for loop. I have selected the number of iterations of the statement in the for loop from the suitable range, and I have selected a number which is big enough to obtain the concurrency among the threads , and to trigger the race condition. In the race condition, each run of the source code (a2.c) may result in different outcomes for a specifically given input. For the identical input, if there is no race condition, each run of the source code (a2.c) will always give identical outcomes. In this question, if there is a race condition, the sum of the number of products which are sold and the number of products which are in the stock may be different from 1000 (the value of total\_stock is defined as 1000 in the question). When there is a race condition, the outcome is based on which instance of sell() function executes when. If there is no race condition the sum of the number of products which are sold and the number of products which are in the stock will always result in total\_stock. The total\_stock is equal to 1000 in the question.**

**//END OF THE PART-A**

**//START OF THE PART B**

**To fix the race condition, I have called lock( ) function at the beginning of the \*sell( ) function, and I have called unlock() function before the “return NULL” statement in the \*sell( ) function. When I call these unlock( ) and lock( ) functions, the concurrency among multiple threads in the race condition is disrupted. So, the output of source code(a2.c) becomes deterministic, meaning that each time we execute the source code (a2.c) , we get the same output. For the case of no race condition, one thread waits the finish of another thread.**

**In the lock( ) function, I have written sem\_wait(&smp) function. It takes just 1 parameter. This parameter is the address of the global variable called ‘smp’. As an alternative, psem\_wait(smp) function can be used in the lock( ) function.**

**In the unlock( ) function, I have written sem\_post(&smp) function. It takes 1 parameter. This parameter is the address of the global variable called ‘smp’. As an alternative, psem\_post(smp) function can be used in the unlock( ) function.**

**//END OF THE PART B**

**//START OF THE PART C**

**The value reported by ‘real’ for the case without the race condition is greater than the value reported by ‘real’ for the case with the race condition. The value reported by ‘user’ for the case without the race condition is greater than the value reported by ‘user’ for the case with the race condition. The value reported by ‘sys’ for the case without race condition is greater than the value reported by ‘sys’ for the case with race condition. Because there is an additional time spent on doing locking and unlocking in the case without the race condition. Here, ‘real’ refers to the elapsed time in the clock from the start time of the time command call to the end of the execution of a2.c. The other processes running at the same time with a specific process are also considered in the calculation of the passed time in the clock from the start time of the time command call to the end of the execution of a2.c . The ‘user’ refers to the cpu time which is spent in the user mode for the execution of a specific process. The ‘sys’ indicates the cpu time which is spent in the kernel mode for the execution of a specific process. You can see the time statistics in the screenshots included in the report.**

**//END OF THE PART C**

**////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////**

**3-)**

**int allowableChair=N; //The global integer variable called “allowableChair” is used to represent the total number of allowable chairs.**

**sem\_t smph=1; //initializing the sem\_t variable to 1 (semaphore initialization)**

**// Variables called ‘allowableChair’ and ‘smph’ are used to check the availability of the chairs.**

**Process patient; // process called ‘patient’ is declared**

**Process dentist; // process called ‘dentist’ is declared**

**int get\_next\_patient(int comingPatients){**

**wait(smph); //decrementation of the value of semaphore**

**if(comingPatients+allowableChair<=N){//check the availability**

**//comingPatients will sit in the free chairs.**

**get\_dental\_treatment(comingPatients);**

**allowableChair= allowableChair-1 //if the sum of allowableChair and comingPatients is equal to N or is smaller than N, then it means that comingPatients are placed in the allowable chairs. So, we should decrease the number of allowableChair by 1.**

**} else if(comingPatients>0 and allowableChair==0){//checking whether the total number of allowableChair is 0 and the total number of comingPatients is greater than 0.**

**while(comingPatients>=0){//checking the number of comingPatients**

**comingPatients= comingPatients-1;//decrease the number of comingPatients**

**if(comingPatients==0){//if comingPatients is equal to 0**

**break; //stop**

**}**

**}**

**}**

**signal(smph); //incrementation the value of semaphore**

**}**

**int get\_dental\_treatment(){**

**wait(smph);//decrementation of the value of semaphore**

**if(allowableChair==N and smph.value<0){//checking whether the total number of available chairs is N**

**//put the process called ‘dentist’ to the list of waiting processes**

**//make the process called ‘dentist’ sleep (sleep( ) )**

**}**

**if(smph.value<=0){//checking whether the value of the semaphore is smaller than or equal to 0**

**//take the process called ‘dentist’ out from the list of waiting processes**

**wakeup(dentist) //to wake up the process called ‘dentist’**

**}**

**signal(smph); //incremention of the value of semaphore**

**}**

**int finish\_treatment(int treatedPatientChair){**

**wait(smph); //decremention of the value of semaphore**

**allowableChair= allowableChair+treatedPatientChair //when the treatment is done, increase the number of allowableChair by treatedPatientChair.**

**signal(smph); // incremention of the value of semaphore**

**}**

**////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////**

**4-) In this question, there are four resources of identical type and three processes in the system. If each process needs one resource, by assigning one resource to each process, we can see that the processes are successfully executed. This means that when each process needs one resource, there is no deadlock. If each process needs two resources, we assign two resources to one process, and one resource to each of the remaining processes. Here, the process which has two resources successfully gets executed. Moreover, the resources of this process are freed after the successful execution of the process. Then, we assign the freed resources to the other processes one by one. Finally, the remaining processes each of which have two resources get successfully executed. So, when each process needs two resources, there is no deadlock. In conclusion, the system which is explained in the question is deadlock-free.**

**////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////**

**NAME-SURNAME: BARIŞ KAPLAN**

**ID NUMBER: 69054**