Student Name : Vivek Kumar Chaudhary

Student ID : 11615987

Email Address : **chaudharyv101@gmail.com**

GitHub Link : **https://github.com/The-Code-Killer/OS-Report.git**

Code : Q.No. 14, Q.No. 19

Q.No. 14 :

```
#include <stdio.h>
#include <stdlib.h>

//Node containing cylinder number and pointer to next node
struct Node
{
	int cylinder;
	struct Node *next;
} *front, *rear, *ptr; //*front pointing to 1st node in the queue and rear pointing to last node in the queue

//function to add node in the queue i.e., cylinders in the ready queue
int queueIt(int cylinderArray[], int size) {
	for(int i = 0; i < size; i++) {
		ptr = (struct Node*) malloc(sizeof(struct Node));
		ptr -> cylinder = cylinderArray[i];
		ptr -> next = NULL;

		if(rear==NULL) {
			front = rear = ptr;
		}
		else {
			rear -> next = ptr;
			rear = ptr;
		}
	}
	return 0;
}

//function to remove cylinders from the ready queue i.e., deleting nodes from the queue
int dequeueIt() {
	ptr = front;
	front = front -> next;
	free(ptr);
	return 0;
}

//function to count number of disk-arm moves
int fcfs(int head) {
```

```
        int sumOfArmMoves = 143 - 125;

        while(front != NULL) {
                sumOfArmMoves += abs(head - front -> cylinder);
                head = front -> cylinder;
                dequeueIt();
        }
        return sumOfArmMoves;
}

int main() {
        front = NULL; rear = NULL;
        int numberOfProcess = 9, head = 143, totalDistance = 0;
        int cylinderArray[] = {86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130};

        printf("Position of head : %d\n", head );
        printf("Queue containing cylinder: ");
        for(int i = 0; i < numberOfProcess; i++)
                printf("%d ", cylinderArray[i]);

        queueIt(cylinderArray, numberOfProcess);

        totalDistance = fcfs(head);

        printf("\nThe total distance that the disk arm moves is : %d\n", totalDistance);

        return 0;
}
```

Description : There are cylinders numbered form 0 to 4999. There are some processes in the ready queue with cylinder number which is required to complete the process. I have to count the number of moves the disk arm will perform to complete all the processes according to FCFS scheduling algorithm with its current head position given at 143 and previous head was at 125. The number of moves between two cylinders will be equal to the difference between the two.

Algorithm :

```
sumOfMoves = 143 - 125, head = givenPosition
while(front != NULL) {
        sumOfMoves = sumOfMoves + abs(head - front -> cylinder)
        head = front -> cylinder
        dequeueIt()
}
return sumOfMoves
```

Time complexity : **Θ(n),** where ’n’ is number of processes in the ready queue.

Constraints : Number of processes are 9 with given cylinders and head is pointing at 143 and previously was at 125.

Code Snippet :

```
int numberOfProcess = 9, head = 143, totalDistance = 0;
int cylinderArray[] = {86, 1470, 913, 1774, 948, 1509, 1022, 1750,
                       130};
```

Test case : Initial queue : 86, 1470, 913, 1774, 948, 1509, 1022, 1750

head = 143, sumOfArmMoves = 143 - 125

sumOfArmMoves = sumOfArmMoves + abs(head - 86) = 75

head = 86, dequeue 86

queue : 1470, 913, 1774, 948, 1509, 1022, 1750

sumOfArmMoves = sumOfArmMoves + abs(head - 1470) = 1459

head = 1470, dequeue 1470

queue : 913, 1774, 948, 1509, 1022, 1750

sumOfArmMoves = sumOfArmMoves + abs(head - 913) = 2016

head = 913, dequeue 913

queue : 1774, 948, 1509, 1022, 1750

sumOfArmMoves = sumOfArmMoves + abs(head - 1774) = 2877

head = 1774, dequeue 1774

queue : 948, 1509, 1022, 1750

sumOfArmMoves = sumOfArmMoves + abs(head - 948) = 3703

head = 948, dequeue 948

queue : 1509, 1022, 1750

sumOfArmMoves = sumOfArmMoves + abs(head - 1509) = 4246

head = 1509, dequeue 1509

queue : 1022, 1750

sumOfArmMoves = sumOfArmMoves + abs(head - 1022) = 4751

head = 1022, dequeue 1022

queue : 1750

sumOfArmMoves = sumOfArmMoves + abs(head - 1750) = 5479

head = 1750, dequeue 1750, queue : NULL

sum of total moves by disk arm = sumOfArmMoves = 7099

Q.No. 19

```
#include <stdio.h>

//mergeUser will merge the arrays according to the increasing order of arrival time of respective processes
int mergeUser(int arrivalTime[], int serviceTime[], int remainingServiceTime[], int beg, int mid, int end) {
	int i = beg, j = mid + 1, k = 0;
	int tempArrival[end - beg + 1], tempService[end - beg + 1], tempRemaining[end - beg + 1];

	while(i <= mid && j <= end) {
		if(arrivalTime[i] < arrivalTime[j]) {
			tempArrival[k] = arrivalTime[i];
			tempService[k] = serviceTime[i];
			tempRemaining[k] = remainingServiceTime[i];
			i++;
		}
		else {
			tempArrival[k] = arrivalTime[j];
			tempService[k] = serviceTime[j];
			tempRemaining[k] = remainingServiceTime[j];
			j++;
		}
		k++;
	}

	if(i > mid) {
		while(j <= end){
			tempArrival[k] = arrivalTime[j];
			tempService[k] = serviceTime[j];
			tempRemaining[k] = remainingServiceTime[j];
			j++; k++;
		}
	}
	else {
			tempArrival[k] = arrivalTime[i];
			tempService[k] = serviceTime[i];
			tempRemaining[k] = remainingServiceTime[i];
			i++; k++;
	}

	for(i = beg, k = 0; i <= end; i++, k++) {
		arrivalTime[i] = tempArrival[k];
		serviceTime[i] = tempService[k];
		remainingServiceTime[i] = tempRemaining[k];
	}

	return 0;
```

```
}

//mergesortUser will sort the arrays recursively
int mergesortUser(int arrivalTime[], int serviceTime[], int remainingServiceTime[], int beg, int end)
{
	if(beg == end)
		return 0;
	int mid = (beg + end) / 2;
	mergesortUser(arrivalTime, serviceTime, remainingServiceTime, beg, mid);
	mergesortUser(arrivalTime, serviceTime, remainingServiceTime, mid + 1, end);
	mergeUser(arrivalTime, serviceTime, remainingServiceTime, beg, mid, end);

	return 0;
}

//getHighestPriority give the index of highest value in priority array
int getHighestPriority(int priority[], int serviceTime[], int remainingServiceTime[], int limit) {
	int highestPriority = 0, iter;
	for(iter = 1; iter < limit; iter++) {

		if(priority[highestPriority] == priority[iter]) {
			if((serviceTime[iter] - remainingServiceTime[iter]) <
(serviceTime[highestPriority] -remainingServiceTime[highestPriority])) {
				highestPriority = iter;
			}
		}
		else if(priority[highestPriority] < priority[iter]) {
				highestPriority = iter;
		    }
	}
	return highestPriority;
}

int main() {
	int numberOfProcess, sumArrival = 0, sumRemaining = 0, sum = 0, timeCounter = 0;
	printf("Enter number of processes : ");
	scanf("%d", &numberOfProcess);

	int arrivalTime[numberOfProcess], priority[numberOfProcess];
	int serviceTime[numberOfProcess], remainingServiceTime[numberOfProcess], waitingTime
= 0;

	int i = 0;

	printf("\n");
	for( ; i < numberOfProcess; i++) {
		printf("\nEnter arrival time of process P%d : ", i + 1);
		scanf("%d", &arrivalTime[i]);
```

```
            printf("\nEnter service time of process P%d : ", i + 1);
            scanf("%d", &serviceTime[i]);

            priority[i] = 0;
            remainingServiceTime[i] = serviceTime[i];

            sumArrival += arrivalTime[i];
            sumRemaining += remainingServiceTime[i];

            sum = (sumArrival > sumRemaining) ? sumArrival : sumRemaining;
        }

        mergesortUser(arrivalTime, serviceTime, remainingServiceTime, 0, numberOfProcess - 1);

        int timeLimit = 1;
        timeCounter = arrivalTime[0];
        while(timeCounter <= sum) {
            int highestPriority;
                while(timeCounter != arrivalTime[timeLimit]) {
                    highestPriority = getHighestPriority(priority, serviceTime,
remainingServiceTime, timeLimit);
                    remainingServiceTime[highestPriority] -= 1;
                    if(remainingServiceTime[highestPriority] == 0) {
                        priority[highestPriority] = 0;
                        waitingTime += (timeCounter + 1) -
arrivalTime[highestPriority] - serviceTime[highestPriority];
                    }
                    else
                        priority[highestPriority] += 1;

                    int i = 0;
                    while(i < timeLimit) {
                        if(i == highestPriority || remainingServiceTime[i] == 0) {
                            i++;
                            continue;
                        }
                        priority[i] += 2;
                        i++;
                    }

                    timeCounter++;
                    if(timeCounter > sum)
                        break;

                    if(remainingServiceTime[numberOfProcess - 1] == 0)
                        goto average;
                }
```

```
            if(timeCounter > sum)
                break;

            if(timeLimit == numberOfProcess) {
                highestPriority = getHighestPriority(priority, serviceTime,
remainingServiceTime, timeLimit);
                remainingServiceTime[highestPriority] -= 1;
                if(remainingServiceTime[highestPriority] == 0) {
                    priority[highestPriority] = 0;
                    waitingTime += (timeCounter + 1) - arrivalTime[highestPriority] -
serviceTime[highestPriority];
                }
                else
                    priority[highestPriority] += 1;

                int i = 0;
                while(i < timeLimit) {
                    if(i == highestPriority || remainingServiceTime[i] == 0) {
                        i++;
                        continue;
                    }
                    priority[i] += 2;
                    i++;
                }

                timeCounter++;
                if(timeCounter > sum)
                    break;
                continue;
            }

            timeLimit++;
        }

        average : printf("\nThe average waiting time for each process is : %f\n", (float)waitingTime/
numberOfProcess);

        return 0;
}
```

Description **:** The problem is based on preemptive dynamic priority scheduling algorithm. All the processes comes in the ready queue with priority value 0. Higher the priority value, higher will be the priority. That process will be executed which will have the highest priority in the ready queue. If two or more processes have same highest priority then

the one which was executed for the least time will be executed. Time slice for the execution of the process is equal to one i.e., processes will be preempted after executing for time equal to 1 unit. Priority of the executing process increase with the rate equals to one and that of the waiting processes increases with the rate equal to 2. Program should be generic i.e., user should give the arrival and burst time or service time for a process.

Algorithm **:** **TimeComplexity for taking input in array is* **O(n)***, n = number of processes*/*

```
mergesortUser(arrivalTime, serviceTime, remainingServiceTime, 0, numberOfProcess - 1);
                    //TimeComplexity for standard merge sort is O(nlogn)
                    //n = number of processes
int timeLimit = 1;
timeCounter = arrivalTime[0];
/*TimeComplexity for below loop depends up the sum of service time and complexity of
  function getHighestPriority()*/
while(timeCounter <= sum) {
        int highestPriority;
        while(timeCounter != arrivalTime[timeLimit]) {
        highestPriority=getHighestPriority(priority, serviceTime, remainingServiceTime,
timeLimit);
                /*TimeComplexity for above function is O(n), where n = number of
                  processes*/
                remainingServiceTime[highestPriority] -= 1;
                if(remainingServiceTime[highestPriority] == 0) {
                        priority[highestPriority] = 0;
                        waitingTime += (timeCounter + 1) - arrivalTime[highestPriority] -
serviceTime[highestPriority];
                }
                else
                        priority[highestPriority] += 1;

                int i = 0;
                while(i < timeLimit) {
                        if(i == highestPriority || remainingServiceTime[i] == 0) {
                                i++;
                                continue;
                        }
                                priority[i] += 2;
```

```
                        i++;
                    }

                    timeCounter++;
                    if(timeCounter > sum)
                        break;

                    if(remainingServiceTime[numberOfProcess - 1] == 0)
                        goto average;
                }

            if(timeCounter > sum)
                break;

            if(timeLimit == numberOfProcess) {
            highestPriority=getHighestPriority(priority, serviceTime, remainingServiceTime,
timeLimit);
                remainingServiceTime[highestPriority] -= 1;
                if(remainingServiceTime[highestPriority] == 0) {
                    priority[highestPriority] = 0;
                    waitingTime += (timeCounter + 1) - arrivalTime[highestPriority] -
serviceTime[highestPriority];
                }
                else
                    priority[highestPriority] += 1;

                int i = 0;
                while(i < timeLimit) {
                    if(i == highestPriority || remainingServiceTime[i] == 0) {
                        i++;
                        continue;
                    }
                    priority[i] += 2;
                    i++;
                }

                timeCounter++;
                if(timeCounter > sum)
                    break;
                continue;
            }

            timeLimit++;
        }
```

*/*Let sum of service time and arrival time be* **'s'** *and* ***'a'*** *respectively, then time complexity of this loop is* **O(n(s + a))***/

Time complexity : O(n(s + a)), where 'n' is the number of processes, 's' is the sum of

the service time of all the processes and ‘a’ is the sum of the arrival time of all the processes.

Constraints : Number of processes, n >= 1

Service Time, s >= 1

Arrival Time, a >= 0

Test case : Let ‘NumberOfProcesses’, ’ArrivalTime’ and ’ServiceTime’ be the input given by the user for the total number of processes, arrival time of each processes, service time i.e., burst time of each processes respectively. Let ‘HighestPriority’ be the highest priority among the priorities of all the processes.

Processes	Arrival Time	Service Time	Remaining time	Priority
P1	1	3	3	0
P2	2	2	2	0

By default, priority (‘Priority’) of each process will be 0 and remaining time (‘RemainingTime’) of process is equal to the service time (‘ServiceTime’) for each process.

RemainingTime of the process getting executed will decrease by 1 and if RemainingTime of the process becomes 0 then the priority will be 0 otherwise and ‘WaitingTime’ will store total waiting time of the process initialised with 0, priority will increase by 1.

Priority of all the processes arrived other than executing process will increase by 2.

Let ‘Time’ be the time. ‘Time’ will be equal to the least arrival time among the processes and will increment by 1 until remaining time of all the process becomes 0. Therefore,

At Time = 1

Processes arrived : P1

HighestPriority : 0

Process Executing : P1

WaitingTime : 0

Processes	Arrival Time	Service Time	Remaining time	Priority
P1	1	3	2	1
P2	2	2	2	0

At Time = 2

Processes arrived : P1, P2

HighestPriority : 1

Process Executing : P1

WaitingTime : 0

Processes	Arrival Time	Service Time	Remaining time	Priority
P1	1	3	1	2
P2	2	2	2	2

At Time = 3

Processes arrived : P1, P2
HighestPriority : 2
Process Executing : P2
WaitingTime : 0

Processes	Arrival Time	Service Time	Remaining time	Priority
P1	1	3	1	4
P2	2	2	1	3

At Time = 4

Processes arrived : P1, P2
HighestPriority : 4
Process Executing : P1
WaitingTime : 1

Processes	Arrival Time	Service Time	Remaining time	Priority
P1	1	3	0	0
P2	2	2	1	5

At Time = 5

Processes arrived : P1, P2
HighestPriority : 5
Process Executing : P2
WaitingTime : 3

Processes	Arrival Time	Service Time	Remaining time	Priority
P1	1	3	0	0
P2	2	2	0	0

Now,
Average waiting time for each process = WaitingTime / NumberOfProcesses = 1.50