#include<stdio.h>

/\* Since a process/query can have multiple attributes like:

QueryID, ArrivalTime, BurstTime, WaitingTime, TurnAroundTime and CompletionTime,

we need to define "structure" (in C language) for that and have to create variables/objects of structure \*/

struct Query {

char QueryID[3]; int ArrivalTime; int BurstTime;

int CompletionTime; int TotalTime;

}Faculty[120], Student[120], Mix[120];

// Initializing required variables (globally):

int TimeQuantum=0, FacultyCount=0, StudentCount=0, MixCount=0, TotalQueries=0, Burst=120;

int TQ=0, WaitTime=0, TATime=0, counter=0, total, CTarr[120], maximumCT=0;

// Function to take Required inputs for a query:

// Time complexity = O(TotalQueries), TotalQueries is a limited int. void InputsForProcess() {

int QueryType, AT=1000, BT=0; ValidQuery:

printf("\nEnter total number of Queries: "); scanf("%d", &TotalQueries);

// Check whether entered query number is <0 or >120 if(TotalQueries<=0 || TotalQueries>120) {

printf("\nQueries cannot be <0 or >120!\n"); goto ValidQuery;

}

else {

TQ = TotalQueries; // for RoundRobin() function printf("\nEnter Time Quantum for each query: "); scanf("%d", &TimeQuantum);

// Taking inputs for all the queries

for(int i=0; i<TotalQueries; i++) { //Time complexity = O(TotalQueries) TryQuery:

printf("\nType of Query (1 for Faculty, 2 for Student): "); scanf("%d", &QueryType);

// Query Processing For Faculty if(QueryType == 1) {

printf("\nEnter Query ID: ");

scanf("%s", &Faculty[FacultyCount].QueryID[0]); FTime:

printf("Enter Query Arrival Time: "); scanf("%d", &AT);

// Check Time constraint

if(AT<1000 || AT>1200 || (AT<1100 && AT>1059) || (AT<1200 && AT>1159)) {

printf("\nEnter Correct Time!\n"); goto FTime;

}

else { // Simplifying ArrivalTime for further calculations

if (AT>=1000 && AT<1100) {

Faculty[FacultyCount].ArrivalTime = AT-1000;

}

else {

Faculty[FacultyCount].ArrivalTime = AT-1040;

}

}

FBTime:

printf("Enter Burst Time: "); scanf("%d", &BT);

if(Burst - BT < 0 || BT <= 0 || Faculty[FacultyCount].ArrivalTime +

BT >= 120) { // initially Burst=120 if(BT<=0) {

printf("\nBurst Time cannot be less than 0\n"); } else {

if (Burst-BT<=0) { int choice;

printf("\nSudesh Sharma will not have enough time to handle this Query because of high BurstTime."

"\nWant to change BurstTime? (1 : Yes; Else : No) "); scanf("%d", &choice);

if(choice==1) { goto FBTime; } else {

printf("\nOK. This query's all data will be lost\n"); goto TryQuery;

}

}

Time\n");

else {

printf("\nInvalid Burst time for corresponding Arrival

}

}

printf("Please enter valid Burst Time\n"); goto FBTime;

}

else {

Faculty[FacultyCount].BurstTime = BT;

}

Burst -= BT; // Updates Total Remaining Burst time Faculty[FacultyCount].TotalTime =

Faculty[FacultyCount].BurstTime; FacultyCount++;

}

// Query Processing For Student else if(QueryType == 2) {

printf("\nEnter Query ID: ");

scanf("%s", &Student[StudentCount].QueryID[0]); STime:

printf("Enter Query Arrival Time: "); scanf("%d", &AT);

// Check Time constraint

if(AT<1000 || AT>1200 || (AT<1100 && AT>1060) || (AT<1200 && AT>1160)) {

printf("\nEnter valid Time!\n"); goto STime;

}

else {

if (AT>=1000 && AT<1100) {

Student[StudentCount].ArrivalTime = AT-1000;

}

else {

Student[StudentCount].ArrivalTime = AT-1040;

}

}

SBTime:

printf("Enter Burst Time: "); scanf("%d", &BT);

if(Burst - BT < 0 || BT <= 0 || Student[StudentCount].ArrivalTime +

BT >= 120) { // initially Burst=120 if(BT<=0) {

printf("\nBurst Time cannot be less than 0\n"); } else {

if (Burst-BT<=0) { int choice;

printf("\nSudesh Sharma won't have enough time to handle this Query because of high BurstTime."

"\nWant to change BurstTime? (1 : Yes; Else : No) "); scanf("%d", &choice);

if(choice==1) { goto FBTime;

}

else {

printf("\nOK. This query's all data will be lost\n"); goto TryQuery;

}

}

else {

Time\n");

printf("\nInvalid Burst time for corresponding Arrival

}

}

printf("Please enter valid Burst Time\n"); goto SBTime;

}

else {

Student[StudentCount].BurstTime = BT; // Updates Total Remaining Burst time

}

Burst -= BT; Student[StudentCount].TotalTime =

Student[StudentCount].BurstTime; StudentCount++;

}

else { // In case any other wrong input printf("\nInvalid Input. Please try again.\n"); goto TryQuery;

}

}

}

}

// Sorting Faculties and Students Queries according to Arrival Time using QuickSort algorithm:

// Time complexity of Faculty QuickSort = O(nlog(n)), n=no. of Faculty queries to sort (limited)

int Fpartition(int low, int high) {

int pivot = Faculty[high].ArrivalTime; int i = (low - 1);

for (int j=low; j<=high; j++) {

if (Faculty[j].ArrivalTime < pivot) { i++;

Faculty[FacultyCount] = Faculty[i]; Faculty[i] = Faculty[j];

Faculty[j] = Faculty[FacultyCount];

}

}

Faculty[FacultyCount] = Faculty[i+1]; Faculty[i+1] = Faculty[high]; Faculty[high] = Faculty[FacultyCount]; return(i+1);

}

void FacultySort(int low, int high) {

if(low < high) {

int pi = Fpartition(low, high); FacultySort(low, pi-1); FacultySort(pi+1, high);

}

}

// Time complexity of Student QuickSort = O(mlog(m)), m=no. of Student queries to sort (limited)

int Spartition(int low, int high) {

int pivot = Student[high].ArrivalTime; int i = (low - 1);

for (int j=low; j<=high; j++) {

if (Student[j].ArrivalTime < pivot) { i++;

Student[StudentCount] = Student[i]; Student[i] = Student[j];

Student[j] = Student[StudentCount];

}

}

Student[StudentCount] = Student[i+1]; Student[i+1] = Student[high]; Student[high] = Student[StudentCount]; return(i+1);

}

void StudentSort(int low, int high) { if(low < high) {

int pi = Spartition(low, high); StudentSort(low, pi-1); StudentSort(pi+1, high);

}

}

// function to merge Faculty and Student's queries into one variable of structure (Mix):

// Time complexity = O(FacultyCount + StudentCount) void MergeQueries() {

int iSC=0, iFC=0; // Counting variables to keep count of added queries into Mix variable

if(FacultyCount !=0 && StudentCount !=0) { // got entries for both while(iSC < StudentCount && iFC < FacultyCount) {

if(Faculty[iFC].ArrivalTime == Student[iSC].ArrivalTime) { // both entries arrives at same time

Mix[MixCount] = Faculty[iFC]; //

priority to faculty

MixCount++;

then student

iFC++;

Mix[MixCount] = Student[iSC]; // and

MixCount++; iSC++;

}

else if(Faculty[iFC].ArrivalTime < Student[iSC].ArrivalTime) { // faculty entry came before

Mix[MixCount] = Faculty[iFC]; MixCount++;

iFC++;

}

else if(Faculty[iFC].ArrivalTime > Student[iSC].ArrivalTime) { // student entry came first

Mix[MixCount] = Student[iSC]; MixCount++;

iSC++;

}

}

if(MixCount != (FacultyCount + StudentCount)) { // in case there's any unadded query (which most probably will occur)

if(FacultyCount != iFC) { // Adding remained

Faculty Queries

while(iFC != FacultyCount) { Mix[MixCount] = Faculty[iFC]; MixCount++;

iFC++;

}

}

Student Queries

else if(StudentCount != iSC) { // Adding remained

while(iSC != StudentCount) { Mix[MixCount] = Student[iSC]; MixCount++;

iSC++;

}

}

}

}

only

else if(FacultyCount == 0) { //got entries for student

while(iSC != StudentCount) { Mix[MixCount] = Student[iSC]; MixCount++;

iSC++;

}

}

only

else if(StudentCount == 0) { //got entries for faculty

while(iFC != FacultyCount) { Mix[MixCount] = Faculty[iFC]; MixCount++;

iFC++;

}

}

}

// Function to apply RoundRobin operation on Mix variable's queries:

// Time complexity of Round Robin = O(1) void RoundRobin() {

total = Mix[0].ArrivalTime;

printf("\n==> Time is in minutes for all calculations\n"); printf("\nQuery

ID\tArrivalTime\tBurstTime\tWaitingTime\tTurnAroundTime\tCompletionTim e\n");

for(int i = 0; TQ != 0;) {

if(Mix[i].TotalTime <= TimeQuantum && Mix[i].TotalTime > 0) { // (First if) Process will complete without any preemption

total = total + Mix[i].TotalTime; Mix[i].TotalTime = 0;

counter = 1;

}

else if(Mix[i].TotalTime > 0) { // Process will preempt according to TimeQuantum

Mix[i].TotalTime -= TimeQuantum; total = total + TimeQuantum;

}

if(Mix[i].TotalTime == 0 && counter == 1) { // continue after first if TQ--;

int ATCalc = Mix[i].ArrivalTime+1000; int CTCalc = total+1000;

CTarr[i] = CTCalc; if(ATCalc>1059) { ATCalc += 40;

}

if(CTCalc>1059) { CTCalc += 40;

}

printf("\n%s\t\t%d hh:mm\t%d minutes\t%d minutes\t%d minutes\t%d

hh:mm",

CTCalc);

Mix[i].QueryID, ATCalc, Mix[i].BurstTime,

total-Mix[i].ArrivalTime-Mix[i].BurstTime, total-Mix[i].ArrivalTime,

WaitTime += total - Mix[i].ArrivalTime - Mix[i].BurstTime; TATime += total - Mix[i].ArrivalTime;

counter = 0;

}

if(i == TotalQueries - 1) { i = 0;

}

else if(Mix[i+1].ArrivalTime <= total) { i++;

}

else {

i = 0;

}

}

}

// Function to find maximum Completion Time:

// Time complexity = O(1) bcoz MixCount is limited int value void MaxCT() {

maximumCT = CTarr[0]; for(int i=1; i<MixCount; i++) {

if(maximumCT < CTarr[i]) { maximumCT = CTarr[i];

}

}

}

// Function to print Final Result of program:

// Time complexity = O(1) void PrintResult() {

MaxCT(); total = Mix[0].ArrivalTime; printf("\n\nSummary of Execution: \n\n");

printf("Total Time Spent on handling Queries: %d minutes\n", maximumCT- total-1000);

float avgWaitTime = WaitTime \* 1.0 / TotalQueries; float avgTATime = TATime \* 1.0 / TotalQueries;

printf("Average TurnAround Time : %.2f minutes\n", avgTATime); printf("Average Waiting Time : %.2f minutes", avgWaitTime); printf("\n\nProgram Execution Completed!\n\n");

}

// Main function:

// Overall Time Complexity = 2\*O(n + m) + O(nlog(n)) + O(mlog(m)) +

2\*O(1) = O(nlog(n)) + O(mlog(m)) int main() {

/\* Program execution sequence:

1. Taking inputs of queries from user
2. Sorting all queries according to ArrivalTime
3. Merging all queries (initial priority to Faculty's query)
4. Applying RoundRobin algorithm on merged queries
5. Print the results \*/

printf("\nWelcome to the OS Project made by TEAM-1.\n\n" "Please follow these instructions to execute the program:\n" "1. Enter number of queries between 0 & 120\n"

"2. Make sure to keep value of TimeQuantum minimum for convinience\n" "3. Enter Query Arrival Time in the format of HHMM\n"

" Example: 10:25 should be entered as 1025\n"

"4. Next Query's ArrivalTime must be less than previous Query's CompletionTime (ArrivalTime + BurstTime)\n"

"5. BurstTime must be entered such that (ArrivalTime + BurstTime) < 120\n");

InputsForProcess(); //Time Complexity = O(TotalQueries) FacultySort(0, FacultyCount-1); // Time Complexity = O(nlog(n));

n=FacultyCount

StudentSort(0, StudentCount-1); // Time Complexity = O(mlog(m)); m=StudentCount

MergeQueries(); // Time Complexity = O(TotalQueries) RoundRobin(); // Time Complexity = O(1) PrintResult(); // Time Complexity = O(1)

}