main.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include <thread>
#include <unistd.h>
#include <algorithm>
#include <queue>

using namespace std;

const int THREADS_MAX = 5;  //num of thread is set to be 5

pthread_t FCFSthreads[THREADS_MAX]; //initalize the array for thread and its corresponding id
int threadsID[THREADS_MAX];

int priority[THREADS_MAX] ={2,4,3,3,1};         //initializing all the need for threads upon
int burst[THREADS_MAX] = {20,25,25,15,10};
int arrival[THREADS_MAX];
int turnaround[THREADS_MAX]; // turnaround time = time complete - arrival time
int wait[THREADS_MAX]; //wait time = turnaround time - burst time
int complete[THREADS_MAX]; //time complete
int prevTime[THREADS_MAX];   // prev and curr takes turn in looping over the thread
int currTime[THREADS_MAX];

void* FCFSFunction(void* arg)
{
    int threadID = *((int*) arg);
    if(threadID == 1){
        prevTime[threadID - 1] = 0;
        arrival[threadID - 1] = prevTime[threadID - 1]; //if it is the first thread executing
    }                                                    //set prev = 0 or starting time so as arrival
    else{
        prevTime[threadID - 1] = currTime[threadID - 1];        //if not then prev must be the time that 1st thread completed
        arrival[threadID - 1] = prevTime[threadID - 1]; //with is currTime, then arrive is curr also
    }
    // printf("Thread %d executing from %d\n", threadID, prevTime);
    this_thread::sleep_for(chrono::milliseconds(burst[threadID - 1]));  // put thread to sleep for burst time
    currTime[threadID - 1] += burst[threadID - 1];      //update curr for future use
    complete[threadID - 1] = currTime[threadID - 1];    // curr time is esstienally complete time but make more reasoning with a different name
    turnaround[threadID - 1] = complete[threadID - 1] - arrival[threadID - 1]; //formula for turnaround time
    // printf("Thread %d finished in %d ms\n", threadID, currTime);
    return NULL;//(void*) prevTime;
}
void runFCFS(){
    for (int i = 0; i < THREADS_MAX; i++)
    {
        threadsID[i] = i + 1;   //using i from 0-4 but threadid were 1-5 so i+1
        pthread_create(&FCFSthreads[i],NULL,FCFSFunction,&threadsID[i]); // create each thread executing FCFSFunction
    }//with the address of threadsID[i] being the arg for the function
```

```c
        //since all threads arrive at the same same then
        // it should not be join immediately but after all threads arrive.
    for (int i = 0; i < THREADS_MAX; i++)
    {
        pthread_join(FCFSthreads[i],NULL); //(void**) &prevTime
    }   //wait for each thread to finish

    int waitTime = 0;           // printing for output
    int turnaroundTime = 0;
    printf("FCFS function\n");
    for(int i = 0; i < THREADS_MAX; i++){
        waitTime += turnaround[i] - burst[i];
        turnaroundTime += turnaround[i];
        printf(" T%d [%d - %d],",i+1,arrival[i], complete[i]);
    }
    int averageWaitTime = waitTime/ THREADS_MAX;
    int averageTurnaroundTime = turnaroundTime/ THREADS_MAX;
    printf("\n");
    printf("Average wait time: %d\n", averageWaitTime);
    printf("Turnaround time: %d\n\n", turnaroundTime);

}
//----------------------------------------------------------------------------------------------------------------------------------------------
struct Thread{
    int id;
    int arrivalTime;    //initalize all need for datas
    int burstTime;
    int completeTime;
    int turnaroundTime;
    int waitTime;
    int prevBurstTime = 0;//intensionally set to be 0 because when we are at index 0 meaning thread 1, the start time shoudl be 0
};
void bubbleSort(Thread SJFthreadsData[], int n) { //
    for (int i = 0; i < n - 1; i++) {   //loop over from first the second last element
        for (int j = 0; j < n - i - 1; j++) {   //loop over the window of unsorted portion
            if (SJFthreadsData[j].burstTime > SJFthreadsData[j + 1].burstTime) {  //compare the adjcent element in unsorted portion
                // Swap [j] and [j+1] if necessary
                int temp = SJFthreadsData[j].burstTime;
                int tempID = SJFthreadsData[j].id;

                SJFthreadsData[j].burstTime = SJFthreadsData[j + 1].burstTime;      //perform basic swap
                SJFthreadsData[j].id = SJFthreadsData[j + 1].id;        //upon burstTime and id so that the information
                                                                    //is consistent after swapping
                SJFthreadsData[j + 1].burstTime = temp;
                SJFthreadsData[j + 1].id = tempID;
            }
        }       //overall keep the num from small to large
    }
}
int prevBurstTime = 0;
void* SJF_Function(void* arg){
    Thread* thread = (Thread*)arg;
    this_thread::sleep_for(chrono::milliseconds(thread->burstTime));  // put thread to sleep for burst time

    thread->arrivalTime = prevBurstTime; //was 0 then its the last finished thread' completion time
    prevBurstTime += thread->burstTime; // using formula and rule according to SJF to calculate
    thread->completeTime = prevBurstTime; // now that prevBurstTime is cumulative updated to time it is now the complete time of current.
    thread->turnaroundTime = thread->completeTime - thread->arrivalTime;
    thread->waitTime = prevBurstTime;// waitTime is essentially when the cumulative prevburst finished
    return NULL;    //because thats when the next thread starts
};
void run_SJF(){
    pthread_t SJFthreads[THREADS_MAX];      //initializing the pthread array
    Thread SJFthreadsData[THREADS_MAX];     //initializing the pthread's data array

    SJFthreadsData[0].id = 1;
    SJFthreadsData[0].burstTime = 20;           //information were given

    SJFthreadsData[1].id = 2;
    SJFthreadsData[1].burstTime = 25;

    SJFthreadsData[2].id = 3;
    SJFthreadsData[2].burstTime = 25;

    SJFthreadsData[3].id = 4;
    SJFthreadsData[3].burstTime = 15;

    SJFthreadsData[4].id = 5;
    SJFthreadsData[4].burstTime = 10;
    // 1  2  3  4  5
    //{20,25,25,15,10}      this was the case
    bubbleSort(SJFthreadsData, THREADS_MAX);
    // 10 15 20 25 25       but after bubble sort this is the case
    //  5  4  1  2  3
    for(int i = 0; i < THREADS_MAX; i++)
    {
        pthread_create(&SJFthreads[i],NULL,SJF_Function,(void*)&SJFthreadsData[i]); //create thread in thread array
        pthread_join(SJFthreads[i],NULL); //and parallel with threadData that does SJF_Function
    }   //wait for one to join before move to next thread
    int waitTime = 0;           // printing for output
    int turnaroundTime = 0;
    printf("SJF function\n");
    for(int i = 0; i < THREADS_MAX; i++){
        waitTime += SJFthreadsData[i].waitTime;
        turnaroundTime += SJFthreadsData[i].turnaroundTime;
        printf(" T%d [%d - %d],",SJFthreadsData[i].id,SJFthreadsData[i].arrivalTime, SJFthreadsData[i].completeTime);
    }
    int averageWaitTime = waitTime/ THREADS_MAX;
    int averageTurnaroundTime = turnaroundTime/ THREADS_MAX;
    printf("\n");
    printf("Average wait time: %d\n", averageWaitTime);
    printf("Turnaround time: %d\n\n", turnaroundTime);
}
//-----------------------------------------------------------------------------------------------------------------------------------------------
int RRprevBurstTime = 0;
const int QUANTUM = 10;
pthread_mutex_t readyQueueMutex = PTHREAD_MUTEX_INITIALIZER;
struct RRThread{
    int id;
    int arrivalTime;    //initalize all need for datas
    int burstTime;
    int completeTime;
    int turnaroundTime;// Same as Thread struct but to keep the data separated
    int waitTime;
    int prevBurstTime = 0;//intensionally set to be 0 because when we are at index 0 meaning thread 1, the start time shoudl be 0
    int remainingTime;
    int firstArrival = 0;
};
int resultIncrement = 0;
queue<int> readyQueue;
const int totalBurst = 20 + 25 + 25 + 15 + 10;
int result[totalBurst/QUANTUM]; // initializing the num of total threads will be executing
void* RR_Function(void* arg){
    RRThread* thread = (RRThread*)arg;  //thread is a type RRThread* that takes arg as parameter
    while(true){
        pthread_mutex_lock(&readyQueueMutex); //prevent other thread from executing
        if(!readyQueue.empty()){   //if there is still task to do
            int currentThread = readyQueue.front();//record the current thread for the use of future continuing execution
            int runningTime = min(thread->burstTime,QUANTUM);
            thread->arrivalTime = RRprevBurstTime;//was 0 then its the last finished thread' completion time
            this_thread::sleep_for(chrono::milliseconds(runningTime));  //in the case of Quantum < burst time
            thread->remainingTime -= runningTime;   //we += the data and let the thread execute again
            thread->completeTime = RRprevBurstTime + runningTime;
            RRprevBurstTime = thread->completeTime;
            thread->turnaroundTime += (thread->completeTime - thread->arrivalTime);
            thread->waitTime += (thread->turnaroundTime - thread->arrivalTime);
            if(thread->remainingTime <=0){  //when thread are finished with its burst time thing should be the same as before
                thread->firstArrival = thread->arrivalTime;
                this_thread::sleep_for(chrono::milliseconds(thread->burstTime));
                thread->arrivalTime = RRprevBurstTime;    //was 0 then its the last finished thread' completion time
                RRprevBurstTime += thread->burstTime; // using formula and rule to calculate

                thread->completeTime = RRprevBurstTime + runningTime; // now that prevBurstTime is cumulative updated to time it is now the complete time of current.
                thread->turnaroundTime += (thread->completeTime - thread->arrivalTime);
                thread->waitTime = thread->turnaroundTime - thread->arrivalTime;// waitTime is essentially when the cumulative prevburst finished
                result[resultIncrement] = currentThread; //put the finshed thread into result that holds int that was overloaded
                resultIncrement++; //increment
                readyQueue.pop(); //pop the finished thread
            }
            else{
                result[resultIncrement] = currentThread;//put the finshed thread into result that holds int that was overloaded
                resultIncrement++; //increment   //first part of a thread execution
                readyQueue.pop();
                pthread_mutex_lock(&readyQueueMutex); //to execute have to wait for a lock if necessary
                readyQueue.push(currentThread); //remaining part of thread execution
                pthread_mutex_unlock(&readyQueueMutex); //unlock for others when finished
            }
        }
        else{
            pthread_mutex_unlock(&readyQueueMutex);// marginall case where there might be no thread on readyQueue at this point
        }//but later on might add on in, so its is safe to let the lock be open for next operation, since everything before was
    }//finished also.
    return NULL;//default return
};
void run_RR(){
    pthread_t RRthreads[THREADS_MAX];   //initializing the pthread array
    RRThread RRthreadsData[THREADS_MAX];   //initializing the pthread's data array

    RRthreadsData[0].id = 1;
    RRthreadsData[0].burstTime = 20;     //information were given

    RRthreadsData[1].id = 2;
    RRthreadsData[1].burstTime = 25;

    RRthreadsData[2].id = 3;
    RRthreadsData[2].burstTime = 25;

    RRthreadsData[3].id = 4;
    RRthreadsData[3].burstTime = 15;

    RRthreadsData[4].id = 5;
    RRthreadsData[4].burstTime = 10;

    //setup result array
    //setup queue
    for(int i = 0; i < THREADS_MAX; i++)
    {
        readyQueue.push(pthread_create(&RRthreads[i],NULL,SJF_Function,(void*)&RRthreadsData[i])); //create thread in thread array
    } //but at the samething we are pushing it to the ready queue for it to execute in order to satisfy quantum circle
    for(int i = 0; i < THREADS_MAX; i++)
    {
        pthread_join(RRthreads[i],NULL);  //wait for all thread to finish
    }//because the fact that a thread may execute multiple times, so it would be reasonable to have others load up after previous thread.

    int waitTime = 0;     // printing for output
    int turnaroundTime = 0;
    printf("RR function\n");
    for(int i = 0; i < sizeof(result); i++){
        waitTime += RRthreadsData[result[i]].waitTime;
        turnaroundTime += RRthreadsData[result[i]].turnaroundTime;
        printf(" T%d [%d - %d],",RRthreadsData[result[i]].id,RRthreadsData[result[i]].arrivalTime, RRthreadsData[result[i]].completeTime);
    }
    int averageWaitTime = waitTime/ THREADS_MAX;
    int averageTurnaroundTime = turnaroundTime/ THREADS_MAX;
    printf("\n");
    printf("Average wait time: %d\n", averageWaitTime);
    printf("Turnaround time: %d\n\n", turnaroundTime);
}
int main(){
    //To run the program click run button
    //FCFS
    runFCFS();
    //SJF
    run_SJF();
    //RR
    run_RR();
    return 0;
}
```