```
// a1809860, Duc Gia Huy, Le
// a1805637, Yuanxi, Wang
// a1805276, Millicent Jesslyn, Danli
// Charity Case
/*
created by Andrey Kan
andrey.kan@adelaide.edu.au
2021
*/
#include <iostream>
#include <fstream>
#include <deque>
#include <vector>
#include <bits/stdc++.h>

// std is a namespace: https://www.cplusplus.com/doc/oldtutorial/namespaces/
static int TIME_ALLOWANCE = 8;  // allow to use up to this number of time slots at
once
const int PRINT_LOG = 0; // print detailed execution trace

class Customer
{
public:
    std::string name;
    int priority;
    int arrival_time;
    int slots_remaining; // how many time slots are still needed
    int playing_since;
    int timesplayed;

    Customer(std::string par_name, int par_priority, int par_arrival_time, int
par_slots_remaining)
    {
        name = par_name;
        priority = par_priority;
        arrival_time = par_arrival_time;
        slots_remaining = par_slots_remaining;
        playing_since = -1;
        timesplayed = 0;
    }
};

class Event
{
public:
    int event_time;
    int customer_id;  // each event involes exactly one customer

    Event(int par_event_time, int par_customer_id)
    {
        event_time = par_event_time;
        customer_id = par_customer_id;
    }
};

void initialize_system(
    std::ifstream &in_file,
    std::deque<Event> &arrival_events,
    std::vector<Customer> &customers,
```

```
    std::vector<int> &burstTimes)
{
    std::string name;
    int priority, arrival_time, slots_requested;

    // read input file line by line
    // https://stackoverflow.com/questions/7868936/read-file-line-by-line-using-
ifstream-in-c
    int customer_id = 0;
    while (in_file >> name >> priority >> arrival_time >> slots_requested)
    {
        Customer customer_from_file(name, priority, arrival_time, slots_requested);
        customers.push_back(customer_from_file);

        // new customer arrival event
        Event arrival_event(arrival_time, customer_id);
        arrival_events.push_back(arrival_event);

        burstTimes.push_back(slots_requested);

        customer_id++;
    }
}

void print_state(
    std::ofstream &out_file,
    int current_time,
    int current_id,
    const std::deque<Event> &arrival_events,
    const std::deque<int> &customer_queue_high,
    const std::deque<int> &customer_queue_low)
{
    out_file << current_time << " " << current_id << '\n';
    if (PRINT_LOG == 0)
    {
        return;
    }
    std::cout << current_time << ", " << current_id << '\n';
    for (int i = 0; i < arrival_events.size(); i++)
    {
        std::cout << "\t" << arrival_events[i].event_time << ", " <<
arrival_events[i].customer_id << ", ";
    }
    std::cout << '\n';
    for (int i = 0; i < customer_queue_high.size(); i++)
    {
        std::cout << "\t cus_hi: " << customer_queue_high[i] << ", ";
    }
    for (int i = 0; i < customer_queue_low.size(); i++)
    {
        std::cout << "\t cus_lo: " << customer_queue_low[i] << ", ";
    }
    std::cout << '\n';
}

// Function for calculating mean
double findMean(std::vector<int> a, int n)
{
      int sum = 0;
```

```
      for (int i = 0; i < n; i++)
            sum += a[i];

      return (double)sum / (double)n;
}

// Function for calculating median, https://www.geeksforgeeks.org/finding-median-
of-unsorted-array-in-linear-time-using-c-stl/
double findMedian(std::vector<int> a, int n)
{

    // If size of the arr[] is even
    if (n % 2 == 0) {

        // Applying nth_element
        // on n/2th index
        nth_element(a.begin(),
                    a.begin() + n / 2,
                    a.end());

        // Applying nth_element
        // on (n-1)/2 th index
        nth_element(a.begin(),
                    a.begin() + (n - 1) / 2,
                    a.end());

        // Find the average of value at
        // index N/2 and (N-1)/2
        return (double)(a[(n - 1) / 2]
                        + a[n / 2])
               / 2.0;
    }

    // If size of the arr[] is odd
    else {

        // Applying nth_element
        // on n/2
        nth_element(a.begin(),
                    a.begin() + n / 2,
                    a.end());

        // Value at index (N/2)th
        // is the median
        return (double)a[n / 2];
    }
}

bool has0timesplayed( std::deque< std::pair<int,int> >queue, std::vector<Customer>
cus){
    for (int i = 0; i < queue.size(); i++){
        if (cus[queue[i].second].timesplayed == 0){
            return true;
        }
    }
    return false;
}

// process command line arguments
```

```
// https://www.geeksforgeeks.org/command-line-arguments-in-c-cpp/
int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        std::cerr << "Provide input and output file names." << std::endl;
        return -1;
    }
    std::ifstream in_file(argv[1]);
    std::ofstream out_file(argv[2]);
    if ((!in_file) || (!out_file))
    {
        std::cerr << "Cannot open one of the files." << std::endl;
        return -1;
    }


    // deque: https://www.geeksforgeeks.org/deque-cpp-stl/
    // vector: https://www.geeksforgeeks.org/vector-in-cpp-stl/
    std::deque<Event> arrival_events; // new customer arrivals
    std::vector<Customer> customers; // information about each customer
    std::vector<int> burstTimes;

    // read information from file, initialize events queue
    initialize_system(in_file, arrival_events, customers, burstTimes);

    int mean = findMean(burstTimes,burstTimes.size());
    int med = findMedian(burstTimes,burstTimes.size());
    // std::cout << "median: " << med << std::endl;
    // std::cout << "mean: " << mean << std::endl;
    // TIME_ALLOWANCE = (mean+med+med)/3;
    // std::cout << "Time quantum: " << TIME_ALLOWANCE << std::endl;

    int current_id = -1; // who is using the machine now, -1 means nobody
    int time_out = -1; // time when current customer will be preempted
    // std::deque< std::pair<int,int> > initialqueue;
    std::deque< std::pair<int,int> > queue_high_0;
    std::deque< std::pair<int,int> > queue_low_1;

    // step by step simulation of each time slot
    bool all_done = false;
    int turn = 0;
    for (int current_time = 0; !all_done; current_time++)
    {
        // welcome newly arrived customers
        while (!arrival_events.empty() && (current_time ==
arrival_events[0].event_time))
        {
            //
initialqueue.push_back( std::make_pair(customers[arrival_events[0].customer_id].slo
ts_remaining,arrival_events[0].customer_id) );
            if (customers[arrival_events[0].customer_id].priority == 0){

queue_high_0.push_back( std::make_pair(customers[arrival_events[0].customer_id].slo
ts_remaining,arrival_events[0].customer_id) );
            }
            else if (customers[arrival_events[0].customer_id].priority == 1){

queue_low_1.push_back( std::make_pair(customers[arrival_events[0].customer_id].slot
```

```
s_remaining,arrival_events[0].customer_id) );
            }
            arrival_events.pop_front();
        }
        // check if we need to take a customer off the machine
        if (current_id >= 0)
        {
            if (current_time == time_out)
            {
                int last_run = current_time - customers[current_id].playing_since;
                customers[current_id].slots_remaining -= last_run;
                if (customers[current_id].slots_remaining > 0)
                {
                    // customer is not done yet, waiting for the next chance to
play
                    if (customers[current_id].priority == 0){

queue_high_0.push_back(std::make_pair(customers[current_id].slots_remaining,current
_id));
                    }
                    else{

queue_low_1.push_back(std::make_pair(customers[current_id].slots_remaining,current_
id));
                    }
                }
                current_id = -1; // the machine is free now
            }
        }
        // if machine is empty, schedule a new customer
        if (current_id == -1)
        {
            TIME_ALLOWANCE = 4;
            bool timesplayed0 = has0timesplayed(queue_high_0,customers);
            bool timesplayed1 = has0timesplayed(queue_low_1,customers);
            if (timesplayed0 || timesplayed1){
                if (timesplayed0){
                    current_id = queue_high_0[0].second;
                    queue_high_0.pop_front();
                    if (TIME_ALLOWANCE > customers[current_id].slots_remaining)
                    {
                        time_out = current_time +
customers[current_id].slots_remaining;
                    }
                    else
                    {
                        time_out = current_time + TIME_ALLOWANCE;
                    }
                    customers[current_id].playing_since = current_time;
                    customers[current_id].timesplayed++;
                }
                else if (timesplayed1){
                    current_id = queue_low_1[0].second;
                    queue_low_1.pop_front();
                    if (TIME_ALLOWANCE > customers[current_id].slots_remaining)
                    {
                        time_out = current_time +
customers[current_id].slots_remaining;
                    }
```

```
                    else
                    {
                        time_out = current_time + TIME_ALLOWANCE;
                    }
                    customers[current_id].playing_since = current_time;
                    customers[current_id].timesplayed++;
                }
            }
            else {
                TIME_ALLOWANCE = (med+mean)/2;
                // high priority customer will have 3 turn to access the game
immediately
                if (turn < 3) {
                    if (!queue_high_0.empty()) // is anyone waiting?
                    {
                        std::sort(queue_high_0.begin(), queue_high_0.end());
                        current_id = queue_high_0[0].second;
                        // int slotsremaining = queue_high_0[0].first;
                        queue_high_0.pop_front();
                        if (TIME_ALLOWANCE > customers[current_id].slots_remaining)
                        {
                            time_out = current_time +
customers[current_id].slots_remaining;
                        }
                        else
                        {
                            time_out = current_time + TIME_ALLOWANCE;
                        }
                        customers[current_id].playing_since = current_time;
                    } else if (!queue_low_1.empty())
                    {
                        std::sort(queue_low_1.begin(), queue_low_1.end());
                        current_id = queue_low_1[0].second;
                        queue_low_1.pop_front();
                        if (TIME_ALLOWANCE > customers[current_id].slots_remaining)
                        {
                            time_out = current_time +
customers[current_id].slots_remaining;
                        }
                        else
                        {
                            time_out = current_time + TIME_ALLOWANCE;
                        }
                        customers[current_id].playing_since = current_time;
                    }
                }
                else if (turn > 2) {
                    if (!queue_low_1.empty()) // is anyone waiting?
                    {
                        std::sort(queue_low_1.begin(), queue_low_1.end());
                        current_id = queue_low_1[0].second;
                        queue_low_1.pop_front();
                        if (TIME_ALLOWANCE > customers[current_id].slots_remaining)
                        {
                            time_out = current_time +
customers[current_id].slots_remaining;
                        }
                        else
                        {
```

```
                            time_out = current_time + TIME_ALLOWANCE;
                        }
                        customers[current_id].playing_since = current_time;
                    } else if (!queue_high_0.empty())
                    {
                        std::sort(queue_high_0.begin(), queue_high_0.end());
                        current_id = queue_high_0[0].second;
                        queue_high_0.pop_front();
                        if (TIME_ALLOWANCE > customers[current_id].slots_remaining)
                        {
                            time_out = current_time +
customers[current_id].slots_remaining;
                        }
                        else
                        {
                            time_out = current_time + TIME_ALLOWANCE;
                        }
                        customers[current_id].playing_since = current_time;
                    }
                }
                turn++;
                turn = turn % 5;
            }
        }
        std::deque<int> queue_high_0_2;
        for (int i = 0; i < queue_high_0.size(); i++){
            queue_high_0_2.push_back(queue_high_0[i].second);
        }
        std::deque<int> queue_low_1_2;
        for (int i = 0; i < queue_low_1.size(); i++){
            queue_low_1_2.push_back(queue_low_1[i].second);
        }
        print_state(out_file, current_time, current_id, arrival_events,
queue_high_0_2, queue_low_1_2);
        // exit loop when there are no new arrivals, no waiting and no playing
customers
        all_done = (arrival_events.empty() && queue_high_0.empty() &&
queue_low_1.empty() && (current_id == -1));
    }
    return 0;
}
```