OSSP Eval 2

Patil Amit Gurusidhappa

19104004

B11

Q1. A,B,C

Round robin with CPU and I/O Burst Time

#include <bits/stdc++.h>

using namespace std;

struct bursts

{

    string burstType;

    int burstVal;

    int remainingBurst;

};

struct process

{

    int p\_id;

    int arrival\_time;

    int priority;

    int share;

    queue<struct bursts> burst;

    int turnaround\_time;

    int waiting\_time;

    int response\_time;

    int start\_time = -1;

    int completion\_time;

    int burst\_time = 0;

};

int main()

{

    FILE \*fp = fopen("p.txt", "r");

    if (fp == NULL)

        printf("Error in opening file");

    else

    {

        char \*line = NULL;

        size\_t len = 0;

        ssize\_t read;

        int n;

        if ((read = getline(&line, &len, fp)) != -1)

            n = atoi(line);

        struct process p[n];

        int i = 0;

        while ((read = getline(&line, &len, fp)) != -1)

        {

            char \*token = strtok(line, " ");

            p[i].p\_id = atoi(token);

            token = strtok(NULL, " ");

            p[i].arrival\_time = atoi(token);

            token = strtok(NULL, " ");

            p[i].priority = atoi(token);

            token = strtok(NULL, " ");

            p[i].share = atoi(token);

            token = strtok(NULL, " ");

            while (token != NULL && atoi(token) != -1)

            {

                struct bursts b;

                b.burstType = token;

                token = strtok(NULL, " ");

                b.burstVal = atoi(token);

                token = strtok(NULL, " ");

                b.remainingBurst = b.burstVal;

                p[i].burst.push(b);

                p[i].burst\_time = p[i].burst\_time + b.burstVal;

            }

            i++;

        }

        fclose(fp);

        int current\_time = 0;

        int completed = 0;

        bool mark[n];

        for (int i = 0; i < n; i++)

            mark[i] = false;

        float total\_turnaround\_time = 0;

        float total\_waiting\_time = 0;

        float total\_response\_time = 0;

        queue<struct process> cpu;

        queue<struct process> io;

        queue<int> occur;

        queue<int> iooccur;

        for (int i = 0; i < n; i++)

        {

            if (p[i].arrival\_time == 0 && mark[i] == false)

            {

                cpu.push(p[i]);

                mark[i] = true;

            }

        }

        while (completed != n)

        {

            bool cpu\_occupied = false;

            bool io\_occupied = false;

            int cpid = -1;

            for (int i = 0; i < n; i++)

            {

                if (current\_time >= p[i].arrival\_time && mark[i] == false)

                {

                    cpu.push(p[i]);

                    mark[i] = true;

                }

            }

            current\_time++;

            if (!cpu.empty())

            {

                cpu\_occupied = true;

                struct process cp = cpu.front();

                int idx = cp.p\_id - 1;

                cpid = cp.p\_id;

                struct bursts b = cp.burst.front();

                if (p[idx].start\_time == -1)

                    p[idx].start\_time = current\_time - 1;

                occur.push(idx + 1);

                p[idx].burst.front().remainingBurst--;

                cpu.front().burst.front().remainingBurst--;

                if (p[idx].burst.front().remainingBurst == 0)

                {

                    p[idx].burst.pop();

                    cpu.pop();

                    if (p[idx].burst.empty())

                    {

                        p[idx].completion\_time = current\_time;

                        p[idx].turnaround\_time = p[idx].completion\_time - p[idx].arrival\_time;

                        p[idx].waiting\_time = p[idx].turnaround\_time - p[idx].burst\_time;

                        p[idx].response\_time = p[idx].start\_time - p[idx].arrival\_time;

                        total\_turnaround\_time += p[idx].turnaround\_time;

                        total\_waiting\_time += p[idx].waiting\_time;

                        total\_response\_time += p[idx].response\_time;

                        completed++;

                    }

                    else

                        io.push(p[idx]);

                }

                else

                {

                    cpu.push(cp);

                    cpu.pop();

                }

            }

            if (cpu\_occupied == false)

                occur.push(0);

            if (!io.empty() && cpid != io.front().p\_id)

            {

        printf("\n|");

        while (!occur.empty())

        {

            if (occur.front() == 0)

                printf("    |");

            else

                printf(" P%d |", occur.front());

            occur.pop();

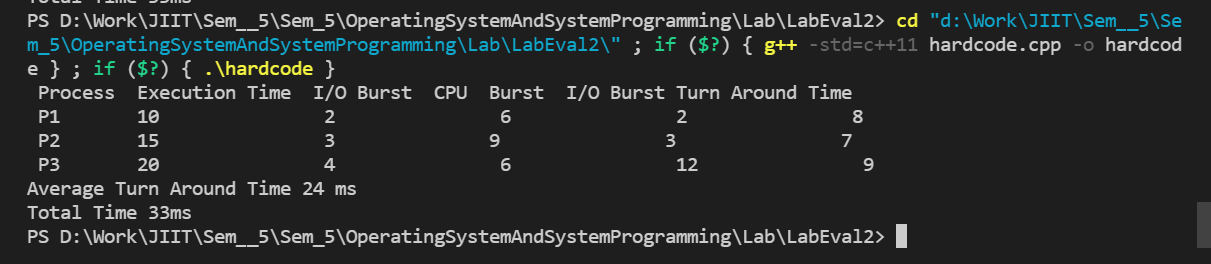
        }

    }

    return 0;

}

Output



Q2. A,B,C

#include <iostream>

using namespace std;

int main()

{

    int n, m, i, j, k;

    n = 5;

    m = 4;

    int alloc[5][4] = {{2, 0, 0, 1},

                       {3, 1, 2, 1},

                       {2, 1, 0, 3},

                       {1, 3, 1, 2},

                       {1, 4, 3, 2}};

    int max[5][4] = {{4, 2, 1, 2},

                     {5, 2, 5, 2},

                     {2, 3, 1, 6},

                     {1, 4, 2, 4},

                     {3, 6, 6, 5}};

    int avail[4] = {3, 3, 2, 1};

    int req[4] = {0, 0, 2, 0};

    int f[n], ans[n], ind = 0;

    for (k = 0; k < n; k++)

    {

        f[k] = 0;

    }

    int need[n][m];

    for (i = 0; i < n; i++)

    {

        for (j = 0; j < m; j++)

            need[i][j] = max[i][j] - alloc[i][j];

    }

    cout << "Need Matrix\n"

         << endl;

    cout << "  "

         << " A B C D \n";

    for (i = 0; i < 5; i++)

    {

        cout << "P" << i << " ";

        for (j = 0; j < 4; j++)

        {

            cout << need[i][j] << " ";

        }

        cout << '\n';

    }

    int y = 0;

    for (k = 0; k < 5; k++)

    {

        for (i = 0; i < n; i++)

        {

            if (f[i] == 0)

            {

                int flag = 0;

                for (j = 0; j < m; j++)

                {

                    if (need[i][j] > avail[j])

                    {

                        flag = 1;

                        break;

                    }

                }

                if (flag == 0)

                {

                    ans[ind++] = i;

                    for (y = 0; y < m; y++)

                        avail[y] += alloc[i][y];

                    f[i] = 1;

                }

            }

        }

    }

    cout << endl;

    cout << "Following is the SAFE Sequence" << endl;

    for (i = 0; i < n - 1; i++)

        cout << " P" << ans[i] << " ->";

    cout << " P" << ans[n - 1] << endl;

    int fl = 0;

    for (i = 0; i < 4; i++)

    {

        if (req[i] > avail[i])

        {

            fl = 1;

            break;

        }

    }

    cout << endl;

    if (fl == 0)

    {

        cout << "Yes the process P4 gets the request immediately." << endl;

    }

    else

    {

        cout << "No, the process P4  not gets the request." << endl;

    }

    return (0);

}

Output:

