#include <stdio.h>

#include <malloc.h>

typedef struct node Node;

typedef struct poly Poly;

struct node {

    int coeff;

    int exp;

    Node \*next;

};

struct poly {

    Node \*firstNode;

    Node \*lastNode;

};

Poly \*resultPoly;

void addPolyNode(Poly \*\*poly, int coeff, int exp){

    Poly \*p = \*poly;

    Node \*newNode = (Node\*) malloc(sizeof(Node));

    newNode->coeff = coeff;

    newNode->exp = exp;

    if(p->firstNode == NULL){

        newNode->next = NULL;

        p->firstNode = newNode;

        p->lastNode = newNode;

    } else {

        p->lastNode->next = newNode;

        newNode->next = NULL;

        p->lastNode = newNode;

    }

}

void traverse(Poly \*\*poly){

    Poly \*p = \*poly;

    Node \*temp = p->firstNode;

    while(temp != NULL){

        if(temp->next == NULL){

            printf(" %dx^%d ", temp->coeff, temp->exp);

            break;

        }

        printf(" %dx^%d + ", temp->coeff, temp->exp);

        temp = temp->next;

    }

    printf("\n");

}

void addPoly(Poly \*\*poly1, Poly \*\*poly2){

    Poly \*p1 = \*poly1;

    Poly \*p2 = \*poly2;

    Node \*ip1 = p1->firstNode;

    Node \*ip2 = p2->firstNode;

    resultPoly = (Poly\*)malloc(sizeof(Poly));

    while(ip1 != NULL && ip2 != NULL){

        if(ip1->exp == ip2->exp){

            addPolyNode(&resultPoly, ip1->coeff+ip2->coeff, ip1->exp);

            ip1 = ip1->next;

            ip2 = ip2->next;

        }else if(ip1->exp > ip2->exp){

            addPolyNode(&resultPoly, ip1->coeff, ip1->exp);

            ip1 = ip1->next;

        } else {

            addPolyNode(&resultPoly, ip2->coeff, ip2->exp);

            ip2 = ip2->next;

        }

    }

    traverse(&resultPoly);

}

int main() {

    Poly \*poly1 = (Poly\*)malloc(sizeof(Poly));

    poly1->firstNode =NULL;

    addPolyNode(&poly1, 3, 2);

    addPolyNode(&poly1, 2, 1);

    addPolyNode(&poly1, 1, 0);

    traverse(&poly1);

    Poly \*poly2 = (Poly\*)malloc(sizeof(Poly));

    poly2->firstNode =NULL;

    addPolyNode(&poly2, 4, 2);

    addPolyNode(&poly2, 6, 1);

    addPolyNode(&poly2, 5, 0);

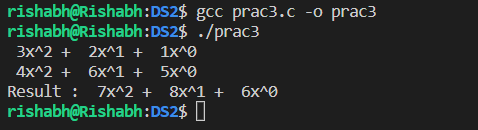
    traverse(&poly2);

    printf("Result : ");

    addPoly(&poly1, &poly2);

    return 0;

}



#include <stdio.h>

#include <pthread.h>

int m3[3][3];

int m1[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};

int m2[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};

int row = 0, col = 0;

void\* matMultiplyByRow(){

    int i = row++;

    for(int j =0; j < 3; j++){

        for (int k = 0; k < 3; k++){

            m3[i][j] += m1[i][k] \* m2[k][j];

        }

    }

}

void fillmat(int m[3][3], int val){

    for(int i = 0; i < 3; i++){

        for(int j = 0; j < 3; j++){

            m[i][j] = val;

        }

    }

}

void printMatrix(int m[3][3]){

    for(int i = 0; i < 3; i++){

        for(int j = 0; j < 3; j++){

            printf("%d  ", m[i][j]);

        }

        printf("\n");

    }

    printf("\n");

}

int main() {

    fillmat(m3, 0);

    pthread\_t th1, th2, th3;

    printf("thread 1\n");

    pthread\_create(&th1, NULL, matMultiplyByRow, NULL);

    pthread\_join(th1, NULL);

    printMatrix(m3);

    printf("thread 2\n");

    pthread\_create(&th2, NULL, matMultiplyByRow, NULL);

    pthread\_join(th2, NULL);

    printMatrix(m3);

    printf("thread 3\n");

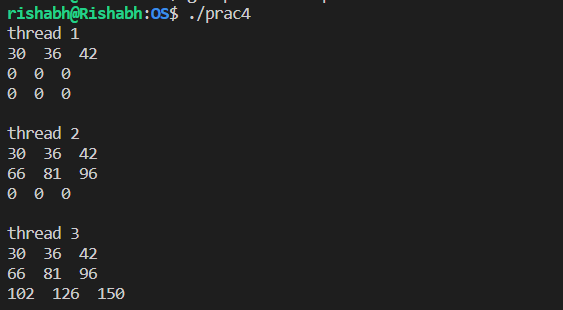
    pthread\_create(&th3, NULL, matMultiplyByRow, NULL);

    pthread\_join(th3, NULL);

    printMatrix(m3);

    return 0;

}



//FCFS algorithm

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

typedef struct processblock process\_block;

struct processblock{

    int arrival\_time;

    int burst\_time;

    int completion\_time;

    int turn\_around\_time;

    int waiting\_time;

    process\_block \*next;

};

process\_block \*head = NULL, \*last;

int size(){

    process\_block \*i = head;

    int len = 0;

    while (i != NULL)

    {

        len++;

        i = i->next;

    }

    return len;

}

void add\_process\_block\_at\_end(int at, int bt){

    process\_block \*newprocess\_block = (process\_block \*)malloc(sizeof(process\_block));

    newprocess\_block->arrival\_time = at;

    newprocess\_block->burst\_time = bt;

    if (head == NULL)

    {

        head = newprocess\_block;

        head->next = NULL;

        last = head;

    }

    else

    {

        last->next = newprocess\_block;

        last = newprocess\_block;

        newprocess\_block->next = NULL;

    }

}

void traverse(){

    process\_block \*i = head;

    printf("Process\t\tArrival Time\tBurst Time\tTurnaround Time\tCompletion Time\tWaiting Time\n");

    int p = 0;

    while (i != NULL)

    {

        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p, i->arrival\_time, i->burst\_time, i->turn\_around\_time, i->completion\_time, i->waiting\_time);

        i = i->next;

        p++;

    }

    printf("\n");

}

void first\_come\_first\_serve(process\_block \*\*pb){

    process\_block \*p = \*pb;

    process\_block \*process = p;

    double average\_waiting\_time = 0, average\_turn\_around\_time = 0;

    int current\_time = 0, ct = 0;

    while(process != NULL){

        current\_time = process->completion\_time = current\_time + process->burst\_time;

        process->turn\_around\_time = process->completion\_time - process->arrival\_time;

        process->waiting\_time = process->turn\_around\_time - process->burst\_time;

        average\_waiting\_time = average\_waiting\_time + process->waiting\_time;

        average\_turn\_around\_time = average\_turn\_around\_time + process->turn\_around\_time;

        process = process->next;

    }

    traverse();

    int num\_processes = size();

    average\_turn\_around\_time = average\_turn\_around\_time/ num\_processes;

    average\_waiting\_time = average\_waiting\_time / num\_processes;

    printf("Average Waiting Time : %.3f\n", average\_waiting\_time);

    printf("Average Turnaround Time : %.3f\n", average\_turn\_around\_time);

}

int main(){

    printf("Algorithm : First Come First Serve\n");

    add\_process\_block\_at\_end(0, 24);

    add\_process\_block\_at\_end(0, 3);

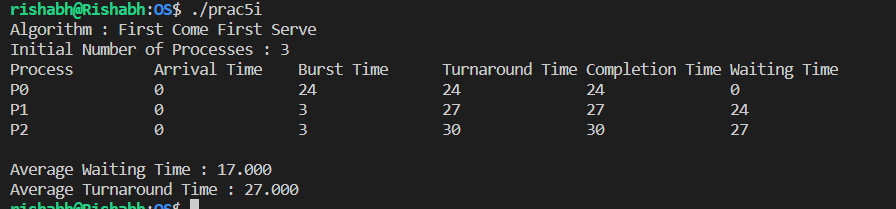
    add\_process\_block\_at\_end(0, 3);

    printf("Initial Number of Processes : %d\n", size());

    first\_come\_first\_serve(&head);

    return 0;

}



//Round Robin algorithm

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

typedef struct processblock process\_block;

struct processblock{

    int process\_id;

    int arrival\_time;

    int burst\_time;

    int completion\_time;

    int turn\_around\_time;

    int waiting\_time;

    process\_block \*next;

};

process\_block \*head = NULL, \*last;

int ini\_num\_processes = 0;

int size(){

    process\_block \*i = head;

    int len = 0;

    while (i != NULL)

    {

        len++;

        i = i->next;

    }

    return len;

}

process\_block\* process\_block\_exist(process\_block \*\*pb){

    process\_block \*p = \*pb;

    process\_block \*process = p;

    process\_block \*i = head;

    while(i != NULL) {

        if (process->arrival\_time == i->arrival\_time && process->burst\_time == i->burst\_time){

            return i;

            break;

        }

        i = i->next;

    }

    return NULL;

}

int process\_count = 0;

void add\_process\_block\_at\_end(int at, int bt){

    process\_block \*newprocess\_block = (process\_block \*)malloc(sizeof(process\_block));

    newprocess\_block->arrival\_time = at;

    newprocess\_block->burst\_time = bt;

    if (head == NULL)

    {

        head = newprocess\_block;

        head->next = NULL;

        last = head;

        head->process\_id = process\_count++;;

    }

    else

    {

        last->next = newprocess\_block;

        last = newprocess\_block;

        newprocess\_block->next = NULL;

        newprocess\_block->process\_id = process\_count++;

    }

}

void add\_updated\_process\_block\_at\_end(process\_block \*\*pb, int burst\_time){

    process\_block \*p = \*pb;

    process\_block \*process = p;

    process\_block \*newprocess\_block = (process\_block \*)malloc(sizeof(process\_block));

    newprocess\_block->arrival\_time = process->arrival\_time;

    newprocess\_block->burst\_time = burst\_time;

    if (head == NULL)

    {

        head = newprocess\_block;

        head->next = NULL;

        last = head;

        head->process\_id = process->process\_id;

    }

    else

    {

        last->next = newprocess\_block;

        last = newprocess\_block;

        newprocess\_block->next = NULL;

        newprocess\_block->process\_id = process->process\_id;

    }

}

void traverse(){

    int p = 0;

    process\_block \*i = head;

    printf("Initial Number of Processes : %d\n", ini\_num\_processes);

    printf("Process\t\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");

    while (i != NULL)

    {

        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i->process\_id, i->arrival\_time, i->burst\_time, i->completion\_time, i->turn\_around\_time, i->waiting\_time);

        i = i->next;

    }

    printf("\n");

}

void round\_robin(process\_block \*\*pb, int time\_quantum){

    process\_block \*p = \*pb;

    process\_block \*process = p;

    double average\_waiting\_time = 0, average\_turn\_around\_time = 0;

    int current\_time = 0;

    int num\_processes = size();

    while (process != NULL) {

        if (process->burst\_time <= time\_quantum){

            current\_time = process->completion\_time = current\_time + process->burst\_time;

        } else if (process->burst\_time > time\_quantum){

            int complement = process->burst\_time - time\_quantum;

            current\_time = process->completion\_time = current\_time + time\_quantum;

            add\_updated\_process\_block\_at\_end(&process, complement);

        }

        if(process->burst\_time <= time\_quantum){

            process->turn\_around\_time = process->completion\_time - process->arrival\_time;

            process->waiting\_time = process->turn\_around\_time - process->burst\_time;

            average\_waiting\_time = average\_waiting\_time + process->waiting\_time;

            average\_turn\_around\_time = average\_turn\_around\_time + process->turn\_around\_time;

        }

        process = process->next;

    }

    traverse();

    average\_turn\_around\_time = average\_turn\_around\_time/ num\_processes;

    average\_waiting\_time = average\_waiting\_time / num\_processes;

    printf("Average Waiting Time : %.3f\n", average\_waiting\_time);

    printf("Average Turnaround Time : %.3f\n", average\_turn\_around\_time);

}

int main(){

    printf("Algorithm : Round Robin\n");

    add\_process\_block\_at\_end(0, 24);

    add\_process\_block\_at\_end(0, 3);

    add\_process\_block\_at\_end(0, 3);

    ini\_num\_processes = size();

    round\_robin(&head, 4);

    return 0;

}

