a) same program, same code.

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <iostream>

using namespace std;

int main()

{

cout<<"SAME PROGRAM SAME CODE : "<<endl;

pid\_t pid;

pid=fork();

if(pid<0)

{

cout<<"Error , Child Not Created"<<endl;

}

else

{

system("ls");

}

return 0;

b) same program, different code.

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <iostream>

using namespace std;

int main()

{

cout<<"SAME PROGRAM D/F CODE : "<<endl;

pid\_t pid;

pid=fork();

if(pid==0)

{

system("ls");

}

else if(pid<0)

{

cout<<"Error , Child Not Created"<<endl;

}

else

{

system("ls -l");

}

return 0;

}

c) before terminating, the parent waits for the child to finish its task.

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <iostream>

#include<sys/wait.h>

using namespace std;

int main()

{

pid\_t pid;

pid=fork();

if(pid==0)

{

cout<<endl<<"This is a child process : "<<endl;

cout<<"Working directory is : "<<endl;

execlp("pwd" , "pwd" , NULL);

cout<<endl;

}

else

{

wait(NULL);

cout<<endl<<"Parent Process : "<<endl;

cout<<"Files in working directory are : "<<endl;

cout<<("ls" , "ls" , NULL);

cout<<endl;

}

return 0;

}

program to implement FCFS scheduling algorithm

#include <iostream>

#include <stdlib.h>

using namespace std;

struct Process{

int pID;

float at;

float bt;

float ct;

float wt;

float tat;

};

void calCompTime(struct Process \*p, int n){

p[0].ct = p[0].bt;

for(int i=1; i<n; i++){

p[i].ct = p[i-1].ct + p[i].bt;

}

}

void calTurnAdTime(struct Process \*p, int n){

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

p[i].tat = p[i].ct - p[i].at;

}

}

void calWaitTime(struct Process \*p, int n){

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

p[i].wt = p[i].tat - p[i].bt;

}

}

void printAvgTime(struct Process \*p, int n){

calCompTime(p, n);

calTurnAdTime(p, n);

calWaitTime(p, n);

cout << "\n process ArrivalTime BurstTime CompletionTime TurnAroundTime WaitingTime \n";

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

cout << " " << p[i].pID << " \t\t" << p[i].at << "\t" << p[i].bt << " \t\t "

<< p[i].ct << "\t\t" << p[i].tat << "\t\t" << p[i].wt << " \n";

}

// Calculating sum of wt and tat

float sumW = 0.0;

float sumT = 0.0;

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

sumW += p[i].wt;

sumT += p[i].tat;

}

// Printing average wt and tat

cout << "\n Average Waiting Time: " << sumW/n;

cout << "\n Average Turn Around Time: " << sumT/n << endl;

}

int main(){

int n;

cout << "\n Enter number of Processes: ";

cin >> n;

cout << endl;

Process p[n];

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

p[i].pID = i+1;

cout << " Enter Arrival Time of Process " << i+1 << ": ";

cin >> p[i].at;

cout << " Enter Burst Time of Process " << i+1 << ": ";

cin >> p[i].bt;

cout << endl;

}

printAvgTime(p, n);

cout << endl;

return 0;

program to implement SJF scheduling algorithm.

#include <iostream>

using namespace std;

struct Proc {

int no, at, bt, it, ct, tat, wt, rt;

};

class SJF {

public:

int n;

Proc p[10];

void readProc() {

cout << "<--SJF Scheduling Algorithm (Non-Preemptive)-->\n";

cout << "Enter Number of Processes: ";

cin >> n;

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

p[i] = read(i + 1);

}

Proc read(int i) {

Proc process;

cout << "\nProcess No: " << i << endl;

process.no = i;

cout << "Enter Arrival Time: ";

cin >> process.at;

cout << "Enter Burst Time: ";

cin >> process.bt;

return process;

}

void sortProc() {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (p[j].at > p[j + 1].at) {

Proc temp = p[j];

p[j] = p[j + 1];

p[j + 1] = temp;

}

}

}

}

void calCompTime() {

int j, min = 0;

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

for (j = i + 1, min = i; j < n && p[j].at <= p[i - 1].ct; j++) {

if (p[j].bt < p[min].bt)

min = j;

}

Proc temp = p[i];

p[i] = p[min];

p[min] = temp;

if (i == 0) {

p[i].it = p[i].at;

}

else

{

if (p[i].at <= p[i - 1].ct)

p[i].it = p[i - 1].ct;

else

p[i].it = p[i].at;

}

p[i].ct = p[i].it + p[i].bt;

}

}

void calTatWT() {

float avgtat = 0, avgwt = 0;

cout << "\nProcess\tAT\tBT\tCT\tTAT\tWT\n";

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

p[i].tat = p[i].ct - p[i].at;

avgtat += p[i].tat;

p[i].wt = p[i].tat - p[i].bt;

avgwt += p[i].wt;

cout << "P" << p[i].no << "\t" << p[i].at << "\t" << p[i].bt << "\t" << p[i].ct << "\t" << p[i].tat << "\t" << p[i].wt << "\t" << endl;

}

avgtat /= n;

avgwt /= n;

cout << "\nAverage TurnAroundTime=" << avgtat << endl;

cout << "Average WaitingTime=" << avgwt << endl;

}

};

int main() {

SJF scheduler;

scheduler.readProc();

scheduler.sortProc();

scheduler.calCompTime();

scheduler.calTatWT();

return 0;

}

program to implement non-preemptive priority-based scheduling algorithm.

#include <iostream>

#define MIN -9999

using namespace std;

struct Process{

int no, at, bt, ct, wt, tat, pri, status;

};

Process read(int i){

struct Process p;

cout << "\nProcess No: " << i << endl;

p.no = i;

cout << "Enter Priority: ";

cin >> p.pri;

cout << "Enter Arrival Time: ";

cin >> p.at;

cout << "Enter Burst Time: ";

cin >> p.bt;

p.status = 0;

return p;

}

int main(){

int n, l, ct = 0, remaining;

struct Process p[10], temp;

float avgtat = 0, avgwt = 0;

cout << " Non-Preemptive First Scheduling Algorithm (Higher No.-->High Priority)\n\n";

cout << "Enter Number of Processes: ";

cin >> n;

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

p[i] = read(i + 1);

for (int i = 0; i < n - 1; i++)

for (int j = 0; j < n - i - 1; j++)

if (p[j].at > p[j + 1].at) {

temp = p[j];

p[j] = p[j + 1];

p[j + 1] = temp;

}

p[9].pri = MIN;

remaining = n;

cout << "\nProcessNo\tAT\tBT\tPri\tCT\tTAT\tWT\n";

ct = p[0].at;

while(remaining != 0){

l = 9;

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

if (p[i].at <= ct && p[i].status != 1 && p[i].pri > p[l].pri) {

l = i;

}

}

p[l].ct = ct = ct + p[l].bt;

p[l].tat = p[l].ct - p[l].at;

p[l].wt = p[l].tat - p[l].bt;

p[l].status = 1;

avgtat += p[l].tat;

avgwt += p[l].wt;

remaining--;

cout << "P" << p[l].no << "\t\t" << p[l].at << "\t" << p[l].bt << "\t"

<< p[l].pri << "\t" << p[l].ct << "\t" << p[l].tat << "\t" << p[l].wt << "\t" << endl;

}

avgtat /= n;

avgwt /= n;

cout << "\nAverage TurnAroundTime=" << avgtat << "\nAverage WaitingTime=" << avgwt;

return 0;

}

program to implement first-fit, best-fit and worst-fit allocation strategies

#include<iostream>

using namespace std;

class MemoryManagementAlgo{

private:

int \*block\_size;

int total\_blocks;

int \*process\_size;

int total\_process;

int \*allocation;

public:

void input();

void First\_Fit();

void Best\_Fit();

void Worst\_Fit() ;

};

void MemoryManagementAlgo :: input(){

cout << "Enter the number of blocks available : ";

cin >> total\_blocks;

block\_size = new int[total\_blocks];

cout << "Enter block sizes : " << endl;

for (int i = 0; i < total\_blocks; i++){

cout << i + 1 << " - ";

cin >> block\_size[i];

}

cout << "Enter the number of processes available : ";

cin >> total\_process;

process\_size = new int[total\_process];

cout << "Enter process sizes : " << endl;

for (int i = 0; i < total\_process; i++){

cout << i + 1 << " - ";

cin >> process\_size[i];

}

}

void MemoryManagementAlgo :: First\_Fit(){

// intialization

allocation = new int[total\_process];

for (int i = 0; i < total\_process; i++){

allocation[i] = -1;

}

//--->loop of process and block

for (int i = 0; i < total\_process; i++){

for (int j = 0; j < total\_blocks; j++){

if (block\_size[j] >= process\_size[i]){

allocation[i] = j;

block\_size[j] -= process\_size[i];

break;

}

}

}

cout << "Process No.\t\tProcess Size\t\tBlock no." << endl;

for (int i = 0; i < total\_process; i++){

cout << " " << i + 1 << " \t\t\t" << process\_size[i] << " \t\t\t";

if (allocation[i] != -1){

cout << allocation[i] + 1;

}

else{

cout << "Not Allocated";

}

cout << endl;

}

// Reset block sizes to their original values

for (int i = 0; i < total\_process; i++) {

if (allocation[i] != -1) {

block\_size[allocation[i]] += process\_size[i];

}

}

}

void MemoryManagementAlgo :: Best\_Fit(){

// intialization

allocation = new int[total\_process];

for (int i = 0; i < total\_process; i++){

allocation[i] = -1;

}

//process

for (int i = 0; i < total\_process; i++){

// Find the best fit block for current process

int bestIdx = -1;

for (int j = 0; j < total\_blocks; j++){

if (block\_size[j] >= process\_size[i]){

if (bestIdx == -1){

bestIdx = j;

}

else if (block\_size[bestIdx] > block\_size[j]){

bestIdx = j;

}

}

}

if (bestIdx != -1){

// allocate block j to p[i] process

allocation[i] = bestIdx;

// Reduce available memory in this block.

block\_size[bestIdx] -= process\_size[i];

}

}

cout << "Process No.\t\tProcess Size\t\tBlock no." << endl;

for (int i = 0; i < total\_process; i++){

cout << " " << i + 1 << " \t\t\t" << process\_size[i] << " \t\t\t";

if (allocation[i] != -1){

cout << allocation[i] + 1;

}

else{

cout << "Not Allocated";

}

cout << endl;

}

// Reset block sizes to their original values

for (int i = 0; i < total\_process; i++) {

if (allocation[i] != -1) {

block\_size[allocation[i]] += process\_size[i];

}

}

}

void MemoryManagementAlgo :: Worst\_Fit(){

// intialization

allocation = new int[total\_process];

for (int i = 0; i < total\_process; i++){

allocation[i] = -1;

}

for (int i = 0; i < total\_process; i++){

// Find the best fit block for current process

int worstIdx = -1;

for (int j = 0; j < total\_blocks; j++){

if (block\_size[j] >= process\_size[i]){

if (worstIdx == -1){

worstIdx = j;

}

else if (block\_size[worstIdx] < block\_size[j]){

worstIdx = j;

}

}

}

if (worstIdx != -1){

// allocate block j to p[i] process

allocation[i] = worstIdx;

// Reduce available memory in this block.

block\_size[worstIdx] -= process\_size[i];

}

}

cout << "Process No.\t\tProcess Size\t\tBlock no." << endl;

for (int i = 0; i < total\_process; i++){

cout << " " << i + 1 << " \t\t\t" << process\_size[i] << " \t\t\t";

if (allocation[i] != -1){

cout << allocation[i] + 1;

}

else{

cout << "Not Allocated";

}

cout << endl;

}

// Reset block sizes to their original values

for (int i = 0; i < total\_process; i++) {

if (allocation[i] != -1) {

block\_size[allocation[i]] += process\_size[i];

}

}

}

int main(){

MemoryManagementAlgo ob ;

int choice;

ob.input();

do{

cout << "\nEnter choice : \n1 - First Fit \n2 - Best Fit \n3 - Worst Fit\n";

cin >> choice;

switch (choice){

case 1:

{

cout << "Your choice : First Fit" << endl;

ob.First\_Fit();

break;

}

case 2:

{

cout << "Your choice : Best Fit" << endl;

ob.Best\_Fit();

break;

}

case 3:

{

cout << "Your choice : Worst Fit" << endl;

ob.Worst\_Fit();

break;

}

default:

{

cout << "Invalid choice" << endl;

break;

}

}

}while(true);

return 0;

}

program to calculate sum of n numbers using Pthreads

#include<pthread.h>

#include<stdio.h>

#include<thread>

#include<iostream>

#include<cstdlib>

using namespace std;

void \*print(void \*p);

int main(int argc, char \*\*argv){

pthread\_t tid;

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

if(argc!=2){

cout<<"Error you have give wrong argument \n";

return -1;

}

if(atoi(argv[1])< 0){

cout<<"\n Integer value must be greater than 0.\n";

return -1;

}

int a = atoi(argv[1]);

pthread\_create(&tid , &attr , print , &a);

pthread\_join(tid , NULL);

return 0;

}

void \*print(void \*p){

int i ;

int num = \*(int\*)(p);

int sum = 0;

cout<<"\nI am inside a thread function \n"<<endl;

for(i=1;i<=num;i++)

sum += i;

cout<<"Sum is : "<<sum<<endl<<endl;

pthread\_exit(0);

}