**Operating System Practicals**

**Name: Nishant Kumar Giri**

**Roll No.: AC-1254**

**Semester: III**

**Practical 1**

Write a program (using fork () and/or exec () commands) where parent and child execute: a) same program, same code. b) same program, different code. c) before terminating, the parent waits for the child to finish its task.

**CODE**

1. same program, same code.

#include <iostream>     *//input output stream*

#include <unistd.h>     *//it is used because fork() is defined in it*

#include <sys/types.h>      *//it is used for type pid\_t for process ID*

using *namespace* std;

*int* main()

{

*pid\_t* pid; *//creating process id using system id pid\_t*

    pid=fork(); *//calling fork command*

    cout<<"pid = "<<pid<<endl; *//printing process id, this statement will run for both child and process*

    return 0;

}

**OUTPUT**

**Text

Description automatically generated with medium confidence**

**CODE**

1. same program, different code.

#include <sys/wait.h>

#include <stdio.h>

#include <unistd.h>

*int* main()

{

*pid\_t* pid, pid1;

*/\* fork a child process \*/*

    pid = fork();

    if (pid < 0)

    { */\* error occurred \*/*

        fprintf(stderr, "Fork Failed!");

        return 1;

    }

    else if (pid == 0)

    { */\* child process \*/*

        pid1 = getpid();

        printf("\nchild: pid = %d \n",pid); */\* A \*/*

        printf("child: pid1 = %d \n",pid1); */\* B \*/*

    }

    else

    { */\* parent process \*/*

        pid1 = getpid();

        printf("\nparent: pid = %d \n",pid); */\* C \*/*

        printf("parent: pid1 = %d \n",pid1); */\* D \*/*

        wait(NULL);

    }

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**CODE**

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

#include <sys/wait.h>

#include <stdio.h>

#include <unistd.h>

*int* main()

{

*pid\_t* pid;

*/\* fork a child process \*/*

    pid = fork();

    if (pid < 0) { */\* error occurred \*/*

        fprintf(stderr, "Fork Failed");

        return 1;

    }

    else if (pid == 0) { */\* child process \*/*

        printf("Child Process");

        printf("\nChild Process Terminated");

    }

    else { */\* parent process \*/*

*/\* parent will wait for the child to complete \*/*

        wait(NULL);

        printf("\nChild Complete \n");

    }

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 2**

Write a program to report behaviour of Linux kernel including kernel version, CPU type and model. (CPU information)

**CODE**

#include<iostream>

using *namespace* std;

*int* main()

{

    cout<<"\n----------CPU Information----------\n";

    system("cat /proc/cpuinfo | grep 'cpu family'");

    system("cat /proc/cpuinfo | grep 'model'");

    system("cat /proc/cpuinfo | grep 'vendor'");

    cout<<"\n----------KERNEL Information----------\n";

    system("cat /proc/sys/kernel/osrelease");

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 3**

Write a program to report behaviour of Linux kernel including information on configured memory, amount of free and used memory. (Memory information)

**CODE**

#include <iostream>

using *namespace* std;

*int* main()

{

    cout << "----------MEMORY Information----------\n";

    system("cat /proc/meminfo | grep 'MemTotal'");

    system("cat /proc/meminfo | grep 'MemFree'");

    system("cat /proc/meminfo | grep 'MemAvailable'");

    cout << "\n\n";

    system("vmstat -s | grep 'total memory'");

    system("vmstat -s | grep 'used memory'");

    system("vmstat -s | grep 'free memory'");

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 4**

Write a program to print file details including owner access permissions, file access time, where file name is given as argument.

**CODE**

#include<string.h>

#include<iostream>

using *namespace* std;

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

{

*char* str[100]="";

    strcat(str , "ls -l ");

    strcat(str ,*argv*[1]);

    strcat(str ,"| awk '{print $1 , $6 , $7 , $8 }'");

    system(str);

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 5**

Write a program to copy files using system calls.

**CODE**

#include <unistd.h> *//read(),write(),open(),...*

#include <fcntl.h>  *//flags O\_RDONLY(),...*

#include <iostream> *//input output stream*

using *namespace* std;

*int* main(*int* *argc*, *char* *\*argv*[])

{

*char* buf[128];

*int* src, des, n; *//src is source file id, des is destinations and n for checking number of bits read and written*

    if (*argc* != 3)

    {

        std::cerr << "Error! Two files expected";

        exit(1);

    }

    else

    {

        src = open(*argv*[1], O\_RDONLY); *//open a file in readonly*

        if (src == -1)

        {

            perror("Error");

            exit(0);

        }

        else

        {

            des = open(*argv*[2], O\_WRONLY | O\_CREAT, 0640); *//set's the flag of opening file as writeonly*

            if (des == -1)

            {

                perror("Error");

                close(src); *//close() is used to close a file*

                exit(0);

            }

            else

            {

                while ((n = read(src, &buf, 128)) > 0) *//reading from src file and storing the characters in buf and their count in n. A max of 128 characters can be read at a time*

                {

                    if (write(des, &buf, n) != n) *//if the number of characters sent to be written is not same as number of written characters written in the file then it will throw an error*

                    {

                        perror("Error:");

                        close(src);

                        close(des);

                        exit(0);

                    }

                }

                write(STDOUT\_FILENO, "copy operation completed!\n",30); *//STDOUT\_FILENO is the macro storing value for console output*

                close(src);

                close(des);

            }

        }

    }

    return 0;

}

**OUTPUT**

**Text

Description automatically generated with medium confidence**

**Practical 6**

Write program to implement FCFS scheduling algorithm.

**CODE**

*//FCFS scheduling algorithm*

#include<iostream>

using *namespace* std;

*int* main()

{

*int* n;

    cout<<"Please enter the number of processes: ";

    cin>>n;

*int* burst\_time[n];

    for(*int* i=1; i<=n; i++)

    {

        cout<<"Please enter the Burst time for P"<<i<<": ";

        cin>>burst\_time[i];

    }

*int* wt\_time[n];

    wt\_time[1]=0;

    for(*int* i=2; i<=n; i++) *//calculating waiting time for each process*

    {

        wt\_time[i]=wt\_time[i-1]+burst\_time[i-1];

    }

*int* turnaround\_time[n];

    for(*int* i=1; i<=n; i++) *//calculating turnarond time for each process*

    {

        turnaround\_time[i]=wt\_time[i]+burst\_time[i];

    }

*float* avg\_wait\_time=0, avg\_turnaround\_time=0;

    for(*int* i=1;i<=n;i++)

    {

        avg\_wait\_time+= wt\_time[i]; *//calculating sum of waiting time of all process*

        avg\_turnaround\_time+= turnaround\_time[i]; *//calculating sum of trunaround time of all process*

    }

    cout<<"     Burst Time \tWaiting Time \tTurnaround Time"<<endl;

    for(*int* i=1; i<=n; i++)

    {

        cout<<"P"<<i+1<<"  \t"<<burst\_time[i]<<"\t\t"<<wt\_time[i]<<"\t\t"<<turnaround\_time[i]<<endl;

    }

    avg\_wait\_time= avg\_wait\_time/n;

    avg\_turnaround\_time= avg\_turnaround\_time/n;

    cout<<"\nAverage Waiting time = "<<avg\_wait\_time<<endl;

    cout<<"\nAverage Turnaround time = "<<avg\_turnaround\_time<<endl;

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 7**

Write program to implement Round Robin scheduling algorithm

**CODE**

#include <iostream>

using *namespace* std;

*//Ready Queue*

*void* queueUpdation(*int* *queue*[], *int* *timer*, *int* *arrival*[], *int* *n*, *int* *maxProccessIndex*)

{

*int* zeroIndex;

    for (*int* i = 0; i < *n*; i++)

    {

        if (*queue*[i] == 0)

        {

            zeroIndex = i;

            break;

        }

    }

*queue*[zeroIndex] = *maxProccessIndex* + 1;

}

*//queue maintained*

*void* queueMaintainence(*int* *queue*[], *int* *n*)

{

    for (*int* i = 0; (i < *n* - 1) && (*queue*[i + 1] != 0); i++)

    {

*int* temp = *queue*[i];

*queue*[i] = *queue*[i + 1];

*queue*[i + 1] = temp;

    }

}

*void* checkNewArrival(*int* *timer*, *int* *arrival*[], *int* *n*, *int* *maxProccessIndex*, *int* *queue*[])

{

    if (*timer* <= *arrival*[*n* - 1])

    {

*bool* newArrival = false;

        for (*int* j = (*maxProccessIndex* + 1); j < *n*; j++)

        {

            if (*arrival*[j] <= *timer*)

            {

                if (*maxProccessIndex* < j)

                {

*maxProccessIndex* = j;

                    newArrival = true;

                }

            }

        }

*//adds the incoming process to the ready queue*

        if (newArrival)

        {

            queueUpdation(*queue*, *timer*, *arrival*, *n*, *maxProccessIndex*);

        }

    }

}

*int* main()

{

*int* n, tq, timer = 0, maxProccessIndex = 0;

*float* avgWait = 0, avgTT = 0;

    cout << "\nPlease enter the Time Quantum : ";

    cin >> tq;

    cout << "\nPlease enter the number of processes : ";

    cin >> n;

*int* arrival[n], burst[n], wait[n], turn[n], queue[n], temp\_burst[n];

*bool* complete[n];

    cout << "\nPlease enter the Arrival Time (in ascending order) : ";

    for (*int* i = 0; i < n; i++)

    {

        cin >> arrival[i];

    }

    cout << "\nPlease enter the CPU Burst Time of the processes : ";

    for (*int* i = 0; i < n; i++)

    {

        cin >> burst[i];

        temp\_burst[i] = burst[i];

    }

    for (*int* i = 0; i < n; i++)

    { *//Initializing the queue and complete array*

        complete[i] = false;

        queue[i] = 0;

    }

    while (timer < arrival[0]) *//Incrementing Timer until the first process arrives*

    {

        timer++;

    }

    queue[0] = 1;

    while (true)

    {

*bool* flag = true;

        for (*int* i = 0; i < n; i++)

        {

            if (temp\_burst[i] != 0)

            {

                flag = false;

                break;

            }

        }

        if (flag)

        {

            break;

        }

        for (*int* i = 0; (i < n) && (queue[i] != 0); i++)

        {

*int* ctr = 0;

            while ((ctr < tq) && (temp\_burst[queue[0] - 1] > 0))

            {

                temp\_burst[queue[0] - 1] -= 1;

                timer += 1;

                ctr++;

*//Checking and Updating the ready queue until all the processes arrive*

                checkNewArrival(timer, arrival, n, maxProccessIndex, queue);

            }

            if ((temp\_burst[queue[0] - 1] == 0) && (complete[queue[0] - 1] == false))

            {

*//turn array currently stores the completion time*

                turn[queue[0] - 1] = timer;

                complete[queue[0] - 1] = true;

            }

*bool* idle = true;

            if (queue[n - 1] == 0)

            {

                for (*int* i = 0; i < n && queue[i] != 0; i++)

                {

                    if (complete[queue[i] - 1] == false)

                    {

                        idle = false;

                    }

                }

            }

            else

            {

                idle = false;

            }

            if (idle)

            {

                timer++;

                checkNewArrival(timer, arrival, n, maxProccessIndex, queue);

            }

*//Maintaining the entries of processes*

*//after each premption in the ready Queue*

            queueMaintainence(queue, n);

        }

    }

    for (*int* i = 0; i < n; i++)

    {

        turn[i] = turn[i] - arrival[i];

        wait[i] = turn[i] - burst[i];

    }

    cout << "\nProcesses\tArrival Time\tCPU Burst Time\tWaiting Time\tTurnaround Time" << endl;

    for (*int* i = 0; i < n; i++)

    {

        cout << i + 1 << "\t\t" << arrival[i] << "\t\t"

             << burst[i] << "\t\t" << wait[i] << "\t\t" << turn[i] << endl;

    }

    for (*int* i = 0; i < n; i++)

    {

        avgWait += wait[i];

        avgTT += turn[i];

    }

    cout << "\nAverage Waiting Time : " << (avgWait / n)

         << "\nAverage Turn Around Time : " << (avgTT / n);

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 8**

Write program to implement SJF scheduling algorithm

**CODE**

*//SJF scheduling algorithm*

#include<iostream>

using *namespace* std;

*int* main()

{

*int* n;

    cout<<"Please enter the number of processes: ";

    cin>>n;

*int* burst[n],process[n];

    for(*int* i=0; i<n; i++)

    {

        cout<<"Please enter the CPU Burst Time for process P"<<i+1<<" :";

        cin>>burst[i];

        process[i]=i+1;

    }

*int* j,k;

*int* temp1,temp2;

    for(j=1; j<n; j++) *//sorting burst time and swapping elements of process[] along with burst[]*

    {

        temp1=burst[j];

        temp2=process[j];

        for(k=j; k>0 && temp1<burst[k-1]; k--)

        {

            burst[k]=burst[k-1];

            process[k]=process[k-1];

        }

        burst[k]=temp1;

        process[k]=temp2;

    }

*int* wait\_time[n],turnaround\_time[n];

    wait\_time[0]=0;

    for(*int* i=1; i<n; i++)

        wait\_time[i]=wait\_time[i-1]+burst[i-1]; *//calculating wait time*

    for(*int* i=0; i<n; i++)

        turnaround\_time[i]=wait\_time[i]+burst[i]; *//calculating turnaround time*

*float* avg\_wt=0,avg\_tt=0;

    cout<<"Processes \t  Burst Time \t   Waiting Time  \t  Turnaround Time"<<endl; *//printing wait time & turnaround time*

    for(*int* i=0; i<n; i++)

    {

        cout<<"P"<<process[i]<<"  \t\t   "<<burst[i]<<"   \t\t      "<<wait\_time[i]<<"   \t\t\t   "<<turnaround\_time[i]<<endl;

        avg\_wt+=wait\_time[i]; *//calculating total sum of wait time*

        avg\_tt+=turnaround\_time[i]; *//calculating total sum of turnaround time*

    }

    avg\_wt=avg\_wt/n; *//calculating avg wait time*

    avg\_tt=avg\_tt/n; *//calculating avg turnaround time*

    cout<<"\nAverage Waiting Time = "<<avg\_wt<<endl;

    cout<<"\nAverage Turnaround Time = "<<avg\_tt<<endl;

    return 0;

}

**OUTPUT**

**A picture containing graphical user interface

Description automatically generated**

**Practical 9**

Write program to implement non-preemptive priority based scheduling algorithm

**CODE**

*//non-preemptive priority scheduling algorithm*

#include <bits/stdc++.h>

#include <iostream>

using *namespace* std;

*struct* Process

{

*int* pid;

*int* bt;

*int* priority;

};

*bool* comparison(Process *a*, Process *b*)

{

    return (*a*.priority > *b*.priority);

}

*void* findWaitingTime(Process *proc*[], *int* *n*, *int* *wt*[])

{

*wt*[0] = 0;

    for (*int* i = 1; i < *n*; i++)

*wt*[i] = *proc*[i - 1].bt + *wt*[i - 1];

}

*void* findTurnAroundTime(Process *proc*[], *int* *n*, *int* *wt*[], *int* *tat*[])

{

    for (*int* i = 0; i < *n*; i++)

*tat*[i] = *proc*[i].bt + *wt*[i];

}

*void* findavgTime(Process *proc*[], *int* *n*)

{

*int* wt[*n*], tat[*n*], total\_wt = 0, total\_tat = 0;

    findWaitingTime(*proc*, *n*, wt);

    findTurnAroundTime(*proc*, *n*, wt, tat);

    cout << "\nProcesses  "

         << " CPU Burst time  "

         << " Waiting time  "

         << " Turn around time\n";

    for (*int* i = 0; i < *n*; i++)

    {

        total\_wt = total\_wt + wt[i];

        total\_tat = total\_tat + tat[i];

        cout << "   " << *proc*[i].pid << "\t\t" << *proc*[i].bt << "\t    " << wt[i] << "\t\t  " << tat[i] << endl;

    }

    cout << "\nAverage Waiting Time = " << (*float*)total\_wt / (*float*)*n*;

    cout << "\nAverage Turn around Time = " << (*float*)total\_tat / (*float*)*n*;

}

*void* priorityScheduling(Process *proc*[], *int* *n*)

{

    std::sort(*proc*, *proc* + *n*, comparison);

    cout << "\nOrder of execution: ";

    for (*int* i = 0; i < *n*; i++)

    {

        cout << *proc*[i].pid << " ";

    }

    cout<< endl;

    findavgTime(*proc*, *n*);

}

*int* main()

{

*int* n;

    cout << "\nPriority Scheduling\nPlease enter the number of Processes = ";

    cin >> n;

    Process \*proc = new Process[n];

    for (*int* i = 0; i < n; i++)

    {

        cout << "\nPlease enter the CPU Burst Time for Process P" << i + 1 << "= ";

        cin >> proc[i].bt;

        cout << "Please enter the Priority of Process P" << i + 1 << "= ";

        cin >> proc[i].priority;

        proc[i].pid = i + 1;

    }

    priorityScheduling(proc, n);

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 10**

Write program to implement preemptive priority based scheduling algorithm

**CODE**

#include <iostream>

using *namespace* std;

*int* main()

{

*int* n;

    cout << "Please enter the number of processes: ";

    cin >> n;

*float* total, wait[n];

*float* p[n], twaiting = 0, waiting = 0;

*int* proc;

*int* stack[n];

*float* brust[n], arrival[n], sbrust, temp[n], top = n, prority[n];

*int* i;

    for (i = 0; i < n; i++)

    {

        p[i] = i;

        stack[i] = i;

        cout << "\nPlease enter the Arrival Time: ";

        cin >> arrival[i];

        cout << "Please enter the CPU Brust Time: ";

        cin >> brust[i];

        cout << "Please enter the Priority Time: ";

        cin >> prority[i];

        temp[i] = arrival[i];

        sbrust = brust[i] + sbrust;

    }

    for (i = 0; i < sbrust; i++)

    {

*//section 1*

        proc = stack[0];

        if (temp[proc] == i)

            twaiting = 0;

        else

            twaiting = i - (temp[proc]);

        temp[proc] = i + 1;

        wait[proc] = wait[proc] + twaiting;

        waiting = waiting + (twaiting);

        brust[proc] = brust[proc] - 1;

        if (brust[proc] == 0)

        {

            for (*int* x = 0; x < top - 1; x++)

                stack[x] = stack[x + 1];

            top = top - 1;

        }

        for (*int* z = 0; z < top - 1; z++)

        {

            if ((prority[stack[0]] > prority[stack[z + 1]]) && (arrival[stack[z + 1]] <= i + 1))

            {

*int* t = stack[0];

                stack[0] = stack[z + 1];

                stack[z + 1] = t;

            }

        }

    }

    cout << "\nAverage Waiting Time : " << waiting / n;

*float* tu = (sbrust + waiting) / n;

    cout << endl

         << "Average Turnaround Time : " << tu << endl;

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 11**

Write program to implement SRJF scheduling algorithm

**CODE**

#include <iostream>

using *namespace* std;

*void* waiting\_time(*struct* process *a*[], *int* *n*);

*struct* process

{

*int* process\_id;

*int* burst\_time;

*int* waiting\_time;

*int* arrival\_time;

*int* remain\_time;

} arr[100];

*int* process\_finish[100];

*int* main()

{

    arr[99].remain\_time = 9999;

*int* n; *//No of process in variable n*

    cout << "\nPlease enter the number of Processes : ";

    cin >> n;

    cout << endl;

    for (*int* i = 0; i < n; i++) *//Take the Burst time for each process by using loop*

    {

        arr[i].process\_id = i + 1; *//increment the process\_id by 1 after each burst\_time*

        cout << "Please enter the CPU Burst Time of P" << i + 1 << " : ";

        cin >> arr[i].burst\_time;

        arr[i].remain\_time = arr[i].burst\_time; *//copy each process burst\_time to another array remain\_time[]*

        cout << "Please enter the Arrival Time : ";

        cin >> arr[i].arrival\_time;

        cout << endl;

    }

    waiting\_time(arr, n);

    return 0;

}

*void* waiting\_time(*struct* process *a*[], *int* *n*)

{

*int* remain = 0, sum\_wait = 0, sum\_turnaround = 0, endTime, smallest;

    cout << "\n\nProcess  Turnaround Time  Waiting Time\n\n";

*int* process\_f = 0; *// handle the INDEX of array process\_finish.*

    for (*int* time = 0; remain != *n*; time++)

    {

        smallest = 99;

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

        {

            if ((*a*[i].arrival\_time <= time) && (*a*[i].remain\_time < *a*[smallest].remain\_time) && (*a*[i].remain\_time > 0))

            {

                smallest = i;

            }

        }

*a*[smallest].remain\_time--;

        if (*a*[smallest].remain\_time == 0)

        {

            process\_finish[process\_f] = smallest + 1; *//to assign a process # which finish the total job*

            process\_f++;

*a*[smallest].process\_id = smallest + 1; *//to ssign a process\_id*

*int* tt;

            remain++; *//One process complete the total job*

            endTime = time + 1; *//Total competional time of process*

            tt = endTime - *a*[smallest].arrival\_time; *//Calculate the TURNaround TIME (competionalTime - TT )*

*a*[smallest].waiting\_time = tt - *a*[smallest].burst\_time; *//Calculate the Waiting Time*

            cout << "\nP[" << smallest + 1 << "]\t\t" << tt << "\t\t" << *a*[smallest].waiting\_time;

            sum\_wait += tt - *a*[smallest].burst\_time; *//For find Average Waiting Time*

        }

    }

    cout << "\n\nAverage Waiting Time = " << sum\_wait \* 1.0 / *n*;

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 12**

Write program to calculate sum of n numbers using thread library.

**CODE**

#include<pthread.h>

#include<stdio.h>

#include<stdlib.h>

*int* sum;

*void* \*runner(*void* \**param*);  *// threads calls this function*

*int* main(*int* *argc* , *char* \**argv[]*)

{

*pthread\_t* tid;

*pthread\_attr\_t* attr;

    if(*argc*!=2)

    {

        fprintf(stderr,"Usage : a.out<integer value>\n");

        return -1;

    }

    if(atoi(*argv*[1])<0)

    {

        fprintf(stderr,"%d must be >=0\n",atoi(*argv*[1]));

        return -1;

    }

    pthread\_attr\_init(&attr);

    pthread\_create(&tid,&attr,runner,*argv*[1]);

    pthread\_join(tid,NULL);

    printf("Sum = %d\n",sum);

}

*void* \*runner(*void* \**param*)

{

*int* i , upper=atoi(*param*);

    sum=0;

    for(i=1 ;i<=upper ;i++)

    {

        sum += i;

    }

    pthread\_exit(0);

}

**OUTPUT**

**Text

Description automatically generated**

**Practical 13**

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

**CODE**

1. **first fit**

*/\* program to implement first-fit allocation strategies \*/*

#include <iostream>

using *namespace* std;

*int* main()

{ *// main function starts*

*int* MemoryBlock[10], Process[10], NumberOfBlock, NumberOfProcess, flags[10],

        allocation[10], i, j;

    for (i = 0; i < 10; i++)

    { *// updating initial allocation status*

        flags[i] = 0;

        allocation[i] = -1;

    }

    cout << "Please enter the number of Memory Blocks: ";

    cin >> NumberOfBlock; *// enter number of memory block*

    cout << "\nPlease enter the Size of each Memory Block: ";

    for (i = 0; i < NumberOfBlock; i++)

    {

        cin >> MemoryBlock[i];

    } *// enter size of each memory block*

    cout << "\nPlease enter the number of Processes: ";

    cin >> NumberOfProcess; *// enter number of processes*

    cout << "\nPlease enter each Process size: ";

    for (i = 0; i < NumberOfProcess; i++)

    {

        cin >> Process[i];

    } *// enter size of each process*

*/\* allocating according to first fit strategies \*/*

    for (i = 0; i < NumberOfProcess;

         i++)

    { *// comparing each process to each memory block*

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

        {

            if (flags[j] == 0 && MemoryBlock[j] >= Process[i])

            {

*/\* if the mem block is not allocated and size of process is less*

*than mem block it will be allocated \*/*

                allocation[j] = i; */\* updating status of memory block to*

*allocated and storing process number \*/*

                flags[j] = 1;

                break;

            }

        }

    }

*/\* displaying gannt chart table \*/*

    cout << "\nBlock no.\tSize\t\tProcess number.\t\t Process Size";

    for (i = 0; i < NumberOfBlock; i++)

    {

        cout << "\n"

             << i + 1 << "\t\t" << MemoryBlock[i] << "\t\t";

        if (flags[i] == 1)

        {

            cout << allocation[i] + 1 << "\t\t\t" << Process[allocation[i]];

        }

        else

        {

            cout << "Not allocated";

        }

    }

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**CODE**

1. **best fit**

*/\* program to implement best-fit allocation strategies \*/*

#include <iostream>  *//input output*

using *namespace* std; *//standard namespace*

*int* main()

{ *//main function*

*int* MemoryBlock[10], Processes[10], numberOfMemoryBlocks, numberOfProc,

        flags[10], allocation[10];

*int* i, j, smallest;

*//setting intial status of memory block to not allocated*

    for (i = 0; i < 10; i++)

    {

        flags[i] = 0;

        allocation[i] = -1;

    }

    cout << "Please enter the number of Memory Partitions: ";

    cin >> numberOfMemoryBlocks; *//enter number of mem block*

    cout << "\nPlease enter size of each partiton: ";

    for (i = 0; i < numberOfMemoryBlocks; i++)

    {

        cin >> MemoryBlock[i];

    } *//enter size of each memory block*

    cout << "\nPlease enter number of processes: ";

    cin >> numberOfProc; *//enter number of processess*

    cout << "\nPlease enter the size of each process: ";

    for (i = 0; i < numberOfProc; i++)

    {

        cin >> Processes[i];

    } *//enter size of each process*

*// allocation as per best fit*

    for (i = 0; i < numberOfProc; i++)

    { *//comparing each process to each mem block*

        smallest = -1; *//initiating smallest memory block*

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

            if (flags[j] == 0 && MemoryBlock[j] >= Processes[i])

            {

                smallest = j;

                break;

            }

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

        {

            if (flags[j] == 0 && MemoryBlock[j] >= Processes[i] &&

                MemoryBlock[j] < MemoryBlock[smallest])

                smallest = j;

        }

        if (smallest != -1)

        {

            allocation[smallest] = i;

            flags[smallest] = 1;

        }

    }

*/\* displaying details \*/*

    cout << "\nPartition\tSize\tProcess No.\tSize";

    for (i = 0; i < numberOfMemoryBlocks; i++)

    {

        cout << "\n"

             << i + 1 << "\t\t" << MemoryBlock[i] << "\t";

        if (flags[i] == 1)

            cout << allocation[i] + 1 << "\t\t" << Processes[allocation[i]];

        else

            cout << "Not allocated";

    }

    cout << endl;

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**

**CODE**

1. **worst fit**

*/\* program to implement worst-fit allocation strategies \*/*

#include <iostream>  *//input output stream*

using *namespace* std; *// standard namespace*

*int* main()

{ *// main function*

*int* NumberOfBlock, NumberOfProcess, MemoryBlock[20], Processes[20];

    cout << " Please enter the number of Memory Blocks: ";

    cin >> NumberOfBlock; *// enter number of blocks*

    cout << " Please enter the number of processes: ";

    cin >> NumberOfProcess; *// enter number of processes*

    cout << " Please enter the size of " << NumberOfBlock << " blocks: ";

    for (*int* i = 0; i < NumberOfBlock; i++)

    {

        cin >> MemoryBlock[i];

    } *// enter size of each mem block*

    cout << " Please enter the size of " << NumberOfProcess << " processes: ";

    for (*int* i = 0; i < NumberOfProcess; i++)

    {

        cin >> Processes[i];

    } *// enter size of each processes*

*// performing worst fit allocation strategies*

    for (*int* i = 0; i < NumberOfProcess; i++)

    {

*/\* comparing each process with each memory block \*/*

*int* max = MemoryBlock[0];

*int* pos = 0;

        for (*int* j = 0; j < NumberOfBlock; j++)

            if (max < MemoryBlock[j])

            {

                max = MemoryBlock[j];

                pos = j;

            }

*/\* displaying details \*/*

        if (max >= Processes[i])

        {

            cout << "\nProcess " << i + 1 << " is allocated to block "

                 << pos + 1;

            MemoryBlock[pos] = MemoryBlock[pos] - Processes[i];

        }

        else

        {

            cout << "\nProcess " << i + 1 << " can't be allocated!";

        }

    }

    cout << endl;

    return 0;

}

**OUTPUT**

**Text

Description automatically generated**