OS/Practical Part

Operating system algorithms are the methods or rules that an operating system uses to manage the processes on the CPU and other resources. They are designed to achieve various goals such as maximizing CPU utilization, fair allocation of CPU, minimizing turnaround time, waiting time and overhead.

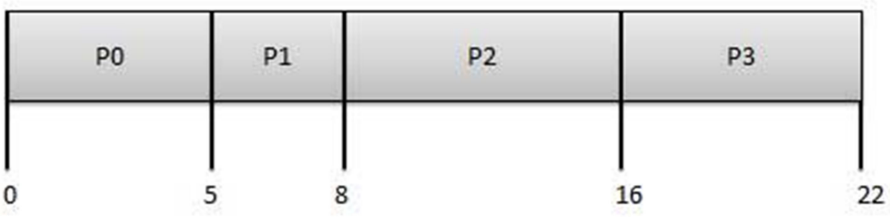
There are many types of operating system algorithms, but some of the most common ones are:

- **First-Come, First-Served (FCFS) Scheduling**: This algorithm assigns the CPU to the first process that arrives in the ready queue. It is simple and easy to implement, but it has a high average waiting time and poor performance in batch systems where processes have fixed arrival times.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

```cpp
// C++ program to Calculate Waiting
// Time for given Processes
#include <iostream>
using namespace std;

// Function to Calculate waiting time
// and average waiting time
void CalculateWaitingTime(int at[],
                          int bt[], int N)
{

    // Declare the array for waiting
    // time
    int wt[N];

    // Waiting time for first process
    // is 0
    wt[0] = 0;

    // Print waiting time process 1
    cout << "PN\t\tAT\t\t"
        << "BT\t\tWT\n\n";
    cout << "1"
        << "\t\t" << at[0] << "\t\t"
        << bt[0] << "\t\t" << wt[0] << endl;

    // Calculating waiting time for
    // each process from the given
    // formula
    for (int i = 1; i < 5; i++) {
        wt[i] = (at[i - 1] + bt[i - 1]
                + wt[i - 1]) - at[i];

        // Print the waiting time for
        // each process
        cout << i + 1 << "\t\t" << at[i]
            << "\t\t" << bt[i] << "\t\t"
            << wt[i] << endl;
    }

    // Declare variable to calculate
    // average
    float average;
    float sum = 0;

    // Loop to calculate sum of all
    // waiting time
    for (int i = 0; i < 5; i++) {
        sum = sum + wt[i];
    }

    // Find average waiting time
    // by dividing it by no. of process
    average = sum / 5;

    // Print Average Waiting Time
    cout << "\nAverage waiting time = "
        << average;
}
```

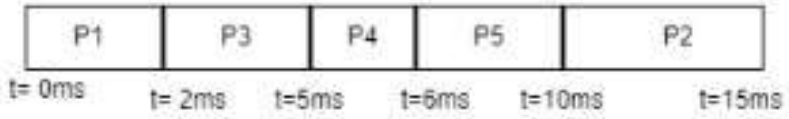
```
// Driver code
int main()
{
    // Number of process
    int N = 5;

    // Array for Arrival time
    int at[] = { 0, 1, 2, 3, 4 };

    // Array for Burst Time
    int bt[] = { 4, 3, 1, 2, 5 };

    // Function call to find
    // waiting time
    CalculateWaitingTime(at, bt, N);
    return 0;
}
//this code is contributed by snehalsalokhe
```

Shortest Job First Algorithm



Completion time: It is the time at which the process finishes its execution. From the Gantt chart we can see that the completion time for process P1 is 2ms, P3 is 5ms, P4 is 6ms, P5 is 10ms and P2 is 15ms. For every process, we just have to look at the immediate right bar. The time denoted below it represents the completion time for the process.

Turn around time(T.A.T): It is the difference between completion time(C.T.) and arrival time(A.T.). The Turnaround time for all the five processes is shown in the below table.

Waiting time(W.T.): It is the difference between turnaround time and burst time. The waiting time actually represents the time that a particular process has spent in the ready queue waiting for CPU allocation. The waiting time(W.T.) for all the five processes is shown in the table below.

The average waiting time can easily be found by adding all the waiting times of the process available and dividing it by the number of processes.

The average waiting time for the below table is = (0+9+0+2+1)/5 = 12/5= 2.4

Process id/ no.	Arrival Time(A.T.)	Burst Time(B.T.)	Completion time(C.T)	Turn around time(T.A.T) (C.T. - A.T.)	Waiting time(W.T.) (T.A.T. - B.T.)
P1	0 ms	2 ms	2ms	2-0= 2ms	2-2=0ms
P2	1 ms	5 ms	15ms	15-1= 14ms	14-5= 9ms
P3	2 ms	3 ms	5ms	5-2= 3ms	3-3 = 0ms
P4	3 ms	1 ms	6ms	6-3= 3ms	3-1= 2ms
P5	5 ms	4 ms	10ms	10-5= 5ms	5-4= 1ms

```
#include<iostream>
#include<algorithm>
using namespace std;

struct node{
    char pname[50];
    int btime;
    int atime;
}a[50];

void insert(int n){
    int i;
    for(i=0;i<n;i++){
        cout<<"Enter name of process "<<i+1<<"\n";
        cin>>a[i].pname;
        cout<<"Enter arrival time of process "<<i+1<<"\n";
        cin>>a[i].atime;
        cout<<"Enter burst time of process "<<i+1<<"\n";
        cin>>a[i].btime;
    }
}

bool btimeSort(node a,node b){
    return a.btime < b.btime;
}

void disp(int n){
    int ttime=0,i;
    int j,tArray[n];
    for(i=0;i<n;i++){
        j=i;
        while(a[j].atime<=ttime&&j!=n){
            j++;
        }
        sort(a+i,a+j,btimeSort);
        tArray[i]=ttime;
        ttime+=a[i].btime;
    }
    tArray[i] = ttime;

    float averageWaitingTime=0;
    float averageTAT=0;
    cout<<"\n";
    cout<<"P.Name  AT\tBT\tWT\tTAT\n";
    for (i=0; i<n; i++){
        cout << a[i].pname << "\t";
        cout << a[i].atime << "\t";
        cout << a[i].btime << "\t";
        cout << tArray[i]-a[i].atime << "\t";
        averageWaitingTime+=tArray[i]-a[i].atime;
        cout << tArray[i]-a[i].atime+a[i].btime << "\t";
        averageTAT+=tArray[i]-a[i].atime+a[i].btime;
        cout <<"\n";
    }
    cout<<"\n";
    cout<<"\nGantt Chart\n";
    for (i=0; i<n; i++){
        cout <<"|   "<< a[i].pname << "   ";
    }
    cout<<"\n";
```

```
    for (i=0; i<n+1; i++){
        cout << tArray[i] << "\t";
    }
    cout<<"\n";
    cout<<"Average Waiting time: "<<(float)averageWaitingTime/(float)n<<endl;
    cout<<"Average turn around time: "<<(float)averageTAT/(float)n<<endl;

}

int main(){
    int nop,choice,i;
    cout<<"Enter number of processes\n";
    cin>>nop;
    insert(nop);
    disp(nop);
//    system("pause");
    return 0;
}
```

First Fit Algorithm for Memory Management

In the first fit, partition is allocated which is first sufficient from the top of Main Memory.

Problem Statement

Given block sizes and process size we need to find which block is assigned to which process under first fit algorithm.

Example:

Input : block_Size[] = {100, 500, 200, 300, 600};

Process_Size[] = {212, 417, 112, 426};

Output:

Process No.	Process Size	Block no.
1	212	2
2	417	5
3	112	2
4	426	Not allocated

Implementation

- Input memory blocks and processes with sizes.

-Initialize all memory blocks as free.

Start by picking each process and check if it can be assigned to current block. 'If size-of-process <= size-of-block if yes then assign and check for next process.

- If not then keep checking the further blocks.

Example

Given:

block_Size[] = {100, 500, 200, 300, 600};

process_Size[] = {212, 417, 112, 426};

n =number of processes = 4,

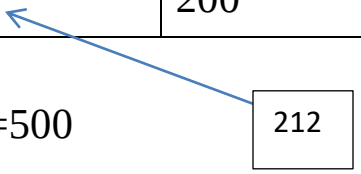
m =number of available blocks = 5

let allocation be an array of size n=4 and initialize it with -1

-1	-1	-1	-1

i=0 process_size[0]=212

100	500	200	300	600



First fit block_size=500

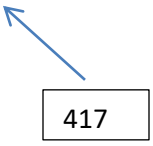
allocation[0]=1

block_size[1]-=212=500-212=288

---

i=1 process_size[1]=417

100	288	200	300	600



First fit block_size=600

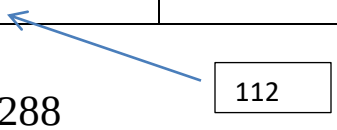
allocation[1]=4

block_size[1]-=417=600-417=183

100	288	300	183

i=2 process_size[2]=112

100	288	200	300	183



first fit block_size=288

allocation[2]=1

block_size[2]-=112=288-112=176

100	176	200	300	183

---

i=3 process_size[3]=426

first fit block _ size= not available

allocation[3]=-1 (remains)

Allocation

1	4	1	-1

Process No.	Process size	Block no.
1	212	2
2	417	5
3	112	2
4	426	Not allocated

Advantages

Its advantage is that it is the fastest search as it searches only the first block i.e. enough to assign a process.

Disadvantages

it may create problems of not allowing processes to take space even if it was possible to allocate. Consider the above example, process number 4 (of size 426) does not get memory. However it was possible to allocate memory if we had allocated using best fit algorithm

---

```cpp
#include <iostream>
#include <cstring>
using namespace std;

void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++) {
        cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main() {
    int blockSize[] = {155, 280, 230, 145, 400, 330};
    int processSize[] = {312, 215, 265, 140, 290};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0;
}
```

Best Fit Algorithm for Memory Management

In the best fit allocates, the process to a partition which is the smallest sufficient partition among the free available partition.

Problem Statement

Given block sizes and process size we need to find which block is assigned to which process under best fit algorithm.

Example:

Input : block_Size[] = {100, 500, 200, 300, 600};

Process_Size[] = {212, 417, 112, 426};

Output:

Process No.	Process Size	Block no.
1	212	4
2	417	3
3	112	2
4	426	5

Implementation

- Input memory blocks and processes with sizes.

-Initialize all memory blocks as free.

Start by picking each process and find the minimum block size that can be assigned to current process i.e find min (block_Size[1],block_size[2],….block_size[n]> process_Size[current],if found then assigned it to the current process

- If not then leave that process and keep checking the further processes.

Example

Given:

block_Size[] = {100, 500, 200, 300, 600};

process_Size[] = {212, 417, 112, 426};

n =number of processes = 4,

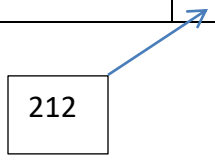
m =number of available blocks = 5

let allocation be an array of size n=4 and initialize it with -1

-1	-1	-1	-1

i=0 process_size[0]=212

100	500	200	300	600



best fit block_size=300

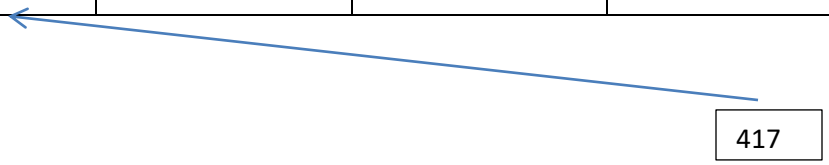
allocation[0]=3

block_size[3]-=212=300-212=88

---

i=1 process_size[1]=417

100	500	200	88	600



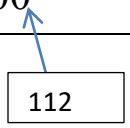
best fit block_size=500

allocation[1]=1

block_size[1]-=500-417=83

i=2 process_size[2]=112

100	83	200	88	600



best fit block_size=200

allocation[2]=2

block_size[2]-=112=200-112=88

100	83	88	88	600

---

i=3 process_size[3]=426

best fit block _ size= 600

allocation[3]=4

block_size[3]-=600-426-174

100	83	88	88	174

Allocation

3	1	2	4

Process No.	Process size	Block no.
1	212	4
2	417	2
3	112	3
4	426	5

Although, best fit minimizes the wastage space ,it consumes a lot of processor time for searching the block which is close to required size. Also, best fit may perform poorer than other algorithms in some cases.

```cpp
#include <iostream>
#include <cstring>
using namespace std;

void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));

    for (int i = 0; i < n; i++) {
        int bestIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {
                    bestIdx = j;
                }
            }
        }

        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] -= processSize[i];
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++) {
        cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main() {
    int blockSize[] = {155, 280, 230, 145, 400, 330};
    int processSize[] = {312, 215, 265, 140, 290};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    bestFit(blockSize, m, processSize, n);

    return 0;
}
```

Worst Fit Algorithm

Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

Example:

```
Input : blockSize[]   = {100, 500, 200, 300, 600};
        processSize[] = {212, 417, 112, 426};
Output:
Process No.    Process Size    Block no.
   1        212         5
   2        417         2
   3        112         5
   4        426         Not Allocated
```

Worst fit algorithm Implementation:

1- Input memory blocks and processes with sizes.
2- Initialize all memory blocks as free.
3- Start by picking each process and find the maximum block size that can be assigned to current process i.e., find max(bockSize[1], blockSize[2],.....blockSize[n]) > processSize[current], if found then assign it to the current process.
5- If not then leave that process and keep checking the further processes.

```
-----------------------------------------------------------
```

Output

```
Process No.    Process Size    Block no.
   1           212             5
   2           417             2
   3           112             5
   4           426             Not Allocated
```

```cpp
#include <iostream>
#include <cstring>
using namespace std;

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));

    for (int i = 0; i < n; i++) {
        int wstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (wstIdx == -1 || blockSize[wstIdx] < blockSize[j]) {
                    wstIdx = j;
                }
            }
        }

        if (wstIdx != -1) {
            allocation[i] = wstIdx;
            blockSize[wstIdx] -= processSize[i];
        }
    }

    cout << "\nProcess No.\t Process Size\t Block no.\n";
    for (int i = 0; i < n; i++) {
        cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main() {
    int blockSize[] = {150, 420, 280, 310, 420};
    int processSize[] = {305, 220, 270, 140, 390};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

Among the CPU scheduling strategies, Round Robin Scheduling is one of the most efficient and the most widely used scheduling algorithm which finds its employability not only in process scheduling in operating systems but also in network scheduling.

Round Robin (RR) Scheduling

This scheduling strategy derives its name from an age old round-robin principle which advocated that all participants are entitled to equal share of assets or opportunities in a turn wise manner. In RR scheduling, each process gets equal time slices (or time quanta) for which it executes in the CPU in turn wise manner. When a process gets its turn, it executes for the assigned time slice and then relinquishes the CPU for the next process in queue. If the process has burst time left, then it is sent to the end of the queue. Processes enter the queue on first come first serve basis.

Round Robin scheduling is preemptive, which means that a running process can be interrupted by another process and sent to the ready queue even when it has not completed its entire execution in CPU. It is a preemptive version of First Come First Serve (FCFS) scheduling algorithm.

Working Principle of Round Robin Scheduling

- Any new process that arrives the system is inserted at the end of the ready queue in FCFS manner.
- The first process in the queue is removed and assigned to the CPU.
- If the required burst time is less than or equal to the time quantum, the process runs to completion. The scheduler is invoked when the process completes executing to let in the next process in the ready queue to the CPU.
- If the required burst time is more than the time quantum, the process executes up to the allotted time quantum. Then its PCB (process control block) status is updated and it is added to the end of the queue. Context switch occurs and the next process in the ready queue is assigned to the CPU.
- The above steps are repeated until there are no more processes in the ready queue.

We can understand the workings RR scheduling algorithm through the aid of the following example.

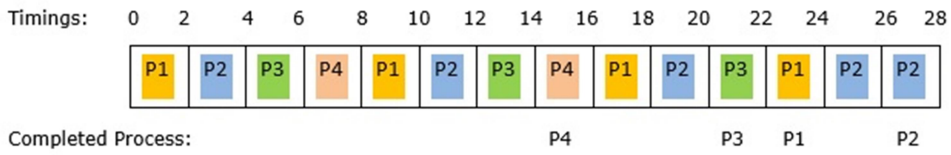
Example of Round Robin Scheduling

Let us consider a system that has four processes which have arrived at the same time in the order P1, P2, P3 and P4. The burst time in milliseconds of each process is given by the following table−

Process	CPU Burst Times in ms
P1	8
P2	10
P3	6
P4	4

Let us consider time quantum of 2ms and perform RR scheduling on this. We will draw GANTT chart and find the average turnaround time and average waiting time.

GANTT Chart with time quantum of 2ms



Average Turnaround Time

Average TAT = Sum of Turnaround Time of each Process / Number of Processes

=(TATP1+TATP2+TATP3+TATP4)/4

(24 + 28 + 22 + 16) / 4 = 22.5 ms

In order to calculate the waiting time of each process, we multiply the time quantum with the number of time slices the process was waiting in the ready queue.

Average Waiting Time

Average WT = Sum of Waiting Time of Each Process / Number of processes

=(WTP1+WTP2+WTP3+WTP4)/4

= (8*2 + 9*2+ 8*2+ 6*2) / 4 = 15.5 ms

```cpp
#include <iostream>

/*at = Arrival time,
bt = Burst time,
time_quantum= Quantum time
tat = Turn around time,
wt = Waiting time*/

using namespace std;

int main(){
    int i,n,time,remain,temps=0,time_quantum;

    int wt=0,tat=0;

    cout<<"Enter the total number of process="<<endl;
    cin>>n;

    remain=n;

    int at[n];
    int bt[n];
    int rt[n];


    cout<<"Enter the Arrival time, Burst time for All the processes"<<endl;
    for(i=0;i<n;i++)
    {
      cout<<"Arrival time for process "<<i+1<<endl;
        cin>>at[i];
        cout<<"Burst time for process "<<i+1<<endl;
        cin>>bt[i];
        rt[i]=bt[i];
    }

    cout<<"Enter the value of time QUANTUM:"<<endl;
    cin>>time_quantum;

    cout<<"\n\nProcess\t\t:Turnaround Time:Waiting Time\n\n";
    for(time=0,i=0;remain!=0;)
    {
        if(rt[i]<=time_quantum && rt[i]>0)
        {
            time += rt[i];

            rt[i]=0;
            temps=1;
        }

        else if(rt[i]>0)
        {
            rt[i] -= time_quantum;

            time += time_quantum;

        }

        if(rt[i]==0 && temps==1)
        {
            remain--;
```

```
            printf("Process{%d}\t:\t%d\t:\t%d\n",i+1,time-at[i],time-at[i]-bt[i]);
            cout<<endl;

            wt += time-at[i]-bt[i];
            tat += time-at[i];
            temps=0;
        }

        if(i == n-1)
            i=0;
        else if(at[i+1] <= time)
            i++;
        else
            i=0;
    }

    cout<<"Average waiting time "<<wt*1.0/n<<endl;
    cout<<"Average turn around time "<<tat*1.0/n<<endl;;

    return 0;
}
```

Deadlock Avoidance Algorithm (Banker's Algorithm)

The algorithm makes use of numerous data structures that change over time:

Available

A vector of length m represents the number of accessible resources of each category.

Allocation

An n*m matrix indicates the number of resources of each kind currently assigned to a process. The column represents the resource, while the rows represent the process.

Request

A n*m matrix represents each process's current request. Process P_i is requesting k additional instances of resource type R_i if request[i][j] = k.

The Banker's Algorithm is a deadlock avoidance and resource allocation algorithm.

It determines if allocating a resource will cause a deadlock or whether allocating a resource to a process is safe, and if not, the resource is not assigned to that process.

Determining a safe sequence (even if it is only one) ensures that the system does not enter a state of deadlock.

Example:

```
********** Deadlock Detection Algorithm ************
Enter the no of Processes       3
Enter the no of resource instances      3
Enter the Max Matrix
3 6 8
4 3 3
3 4 4
Enter the Allocation Matrix
3 3 3
2 0 3
1 2 4
Enter the available Resources
1 2 0
Process  Allocation      Max     Available
P1          3 3 3       3 6 8     1 2 0
P2          2 0 3       4 3 3
P3          1 2 4       3 4 4

system is in Deadlock and the Deadlock process are
P0      P1      P2
```

```cpp
// C++ program to illustrate Banker's Algorithm
#include<iostream>
using namespace std;

// Number of processes
const int P = 5;

// Number of resources
const int R = 3;

// Function to find the need of each process
void calculateNeed(int need[P][R], int maxm[P][R],
                int allot[P][R])
{
    // Calculating Need of each P
    for (int i = 0 ; i < P ; i++)
        for (int j = 0 ; j < R ; j++)

            // Need of instance = maxm instance -
            //                allocated instance
            need[i][j] = maxm[i][j] - allot[i][j];
}

// Function to find the system is in safe state or not
bool isSafe(int processes[], int avail[], int maxm[][R],
            int allot[][R])
{
    int need[P][R];

    // Function to calculate need matrix
    calculateNeed(need, maxm, allot);

    // Mark all processes as infinish
    bool finish[P] = {0};

    // To store safe sequence
    int safeSeq[P];

    // Make a copy of available resources
    int work[R];
    for (int i = 0; i < R ; i++)
        work[i] = avail[i];

    // While all processes are not finished
    // or system is not in safe state.
    int count = 0;
    while (count < P)
    {
        // Find a process which is not finish and
        // whose needs can be satisfied with current
        // work[] resources.
        bool found = false;
        for (int p = 0; p < P; p++)
        {
            // First check if a process is finished,
            // if no, go for next condition
            if (finish[p] == 0)
            {
                // Check if for all resources of
                // current P need is less
```

```
                // than work
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                // If all needs of p were satisfied.
                if (j == R)
                {
                    // Add the allocated resources of
                    // current P to the available/work
                    // resources i.e.free the resources
 namespace std;
                        work[k] += allot[p][k];

                    // Add this process to safe sequence.
                    safeSeq[count++] = p;

                    // Mark this p as finished
                    finish[p] = 1;

                    found = true;
                }
            }
        }

        // If we could not find a next process in safe
        // sequence.
        if (found == false)
        {
            cout << "System is not in safe state";
            return false;
        }
    }

    // If system is in safe state then
    // safe sequence will be as below
    cout << "System is in safe state.\nSafe"
        " sequence is: ";
    for (int i = 0; i < P ; i++)
        cout << safeSeq[i] << " ";

    return true;
}

// Driver code
int main()
{
    int processes[] = {0, 1, 2, 3, 4};

    // Available instances of resources
    int avail[] = {0, 0, 0};

    // Maximum R that can be allocated
    // to processes
    int maxm[][R] = {{0, 0, 0},
                    {2, 0, 2},
                    {0, 0, 0},
                    {1, 0, 0},
                    {0, 0, 2}};
```

```
    // Resources allocated to processes
    int allot[][R] = {{0, 1, 0},
                      {2, 0, 0},
                      {3, 0, 3},
                      {2, 1, 1},
                      {0, 0, 2}};

    // Check system is in safe state or not
    isSafe(processes, avail, maxm, allot);

    return 0;
}
```