

Presented by:

Jayson Lana

Aerom Canimo

Marcial Zipagan Jr.

Nikki C. Gonzales

Name: Nikki C. Gonzales

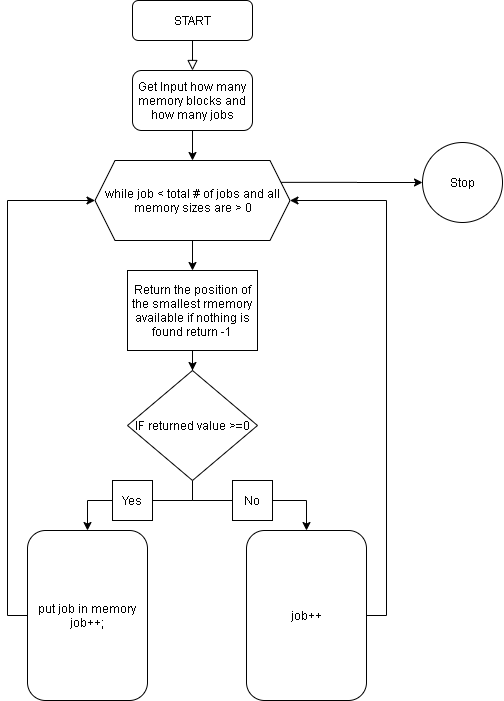
Course/Section: BSCS 2A (NS)

**Dynamic Memory Allocation:**

1. **Best Fit-**

Definition: Best Fit Algorithm allows process to be arrange in the memory where the memory size is closely fitted to the job size. It will look for the smallest possible free space to allocate the current job.

Diagram:



Code:

int jnth=0,nth=0,min=-1,i=0,k=0,count=0;

typedef struct memory{

int jobs[200];

int jp;

int size;

}memory;

memory mem[200];

typedef struct jobs{

int block;

int size;

}jobs;

jobs job[200];

int check(){

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

if(mem[i].size>0){

return 0;

}

}

return 1;

}

int best\_fit(int k){

int min=0,check=0;

//first check if job k is greater than all memory sizes

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

if(job[k].size<=mem[i].size){

check=1;

}

}

if(check==0){

return -1;

}

//first find the largest memory size available

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

if(mem[i].size>mem[min].size){

min=i;

}

}

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

// printf("mem %d",mem[i].size);

if(mem[i].size<mem[min].size && i!=min){

if(job[k].size<=mem[i].size){

min=i;

}

}

}

// printf("Min:%d k:%d\n",min,k+1);

return min;

}

void calc(void){

int min=0;

while(check()==0 && k<jnth){

min=best\_fit(k);

if(min>=0){

// place the job in the memory and minus the jobs size to the memory

mem[min].jobs[mem[min].jp]=k;

mem[min].jp++;

job[k].block=min+1;

mem[min].size-=job[k].size;

}

k++;

}

}

int main()

{

printf("Best Fit\n");

//inputs

//set memory

printf("How much Memory:");

scanf("%d",&nth);

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

printf("Bloc %d Size:",i+1);

scanf("%d",&mem[i].size);

}

//set jobs

printf("How many Jobs:");

scanf("%d",&jnth);

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

printf("Job %d Size:",i+1);

scanf("%d",&job[i].size);

}

printf("Batch Jobs:[JobName,SIZE]\n");

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

printf("Job%d: %d\n",i+1,job[i].size);

}

printf("\n\n");

calc();

printf("Memory:\n");

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

printf("Block %d INTERNAL FRAGMENTATION:%d\nJOBS:",i+1,mem[i].size);

if(mem[i].jp==0){

printf(" FREE SPACE");

}else{

for(int k=0;k<mem[i].jp;k++){

printf(" %d,",mem[i].jobs[k]+1);

}

}

printf("\n\n");

}

printf("JOB:\n");

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

if(job[i].block<=0){

printf("Job %d-> No Allocation\n",i+1);

}else{

printf("Job %d-> Block:%d\n",i+1,job[i].block);

}

}

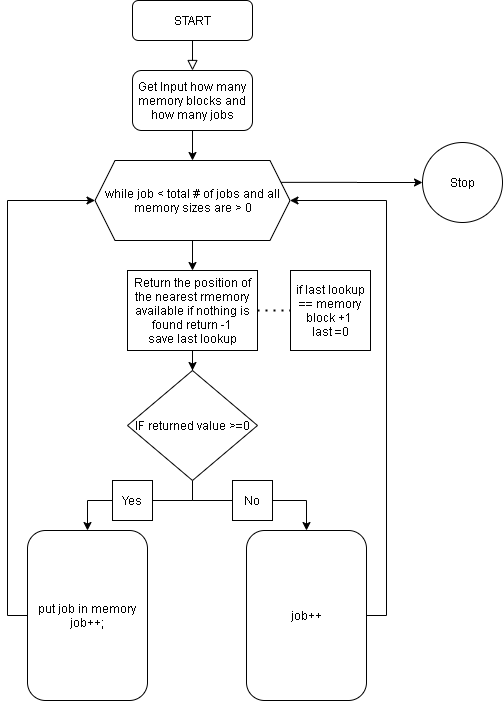
return 0;

}

1. **Next Fit-**

Definition: Next Fit is like First Fit where it search the nearest possible free space to allocate the job. But the difference is that it will start the lookup for free spaces for the next concurring jobs at the end where it last lookup ends. It manages to eliminate internal fragmentation.

Diagram:

  
Code:

int jnth=0,nth=0,min=-1,i=0,k=0,count=0;

typedef struct memory{

int jobs[200];

int jp;

int size;

int occupied;

}memory;

memory mem[200];

typedef struct jobs{

int block;

int size;

}jobs;

jobs job[200];

int check(){

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

if(mem[i].size>0){

return 0;

}

}

return 1;

}

int next\_fit(int k,int last){

int min=0,check=0;

//first check if job k is greater than all memory sizes

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

if(job[k].size<=mem[i].size){

check=1;

}

}

if(check==0){

return -1;

}

printf("hi");

if(last+1==nth){

last=0;

}

//first find the first memory size available

for(int i=last;i<nth;i++){

if(mem[i].size>=job[k].size && mem[i].occupied==0 && mem[i].size>0){

return i;

}

}

printf("Min:%d k:%d\n",min,k+1);

return -1;

}

void calc(void){

int min=0;

while(check()==0 && k<jnth){

min=next\_fit(k,min);

if(min>=0){

// place the job in the memory and minus the jobs size to the memory

mem[min].jobs[mem[min].jp]=k;

mem[min].jp++;

job[k].block=min+1;

mem[min].occupied=1;

mem[min].size-=job[k].size;

}

k++;

}

}

int main()

{

printf("Next Fit\n");

//inputs

//set memory

printf("How much Memory:");

scanf("%d",&nth);

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

printf("Bloc %d Size:",i+1);

scanf("%d",&mem[i].size);

}

//set jobs

printf("How many Jobs:");

scanf("%d",&jnth);

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

printf("Job %d Size:",i+1);

scanf("%d",&job[i].size);

}

printf("Batch Jobs:[JobName,SIZE]\n");

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

printf("Job%d: %d\n",i+1,job[i].size);

}

printf("\n\n");

calc();

printf("Memory:\n");

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

printf("Block %d INTERNAL FRAGMENTATION:%d\nJOBS:",i+1,mem[i].size);

if(mem[i].jp==0){

printf(" FREE SPACE");

}else{

for(int k=0;k<mem[i].jp;k++){

printf(" %d,",mem[i].jobs[k]+1);

}

}

printf("\n\n");

}

printf("JOB:\n");

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

if(job[i].block<=0){

printf("Job %d-> No Allocation\n",i+1);

}else{

printf("Job %d-> Block:%d\n",i+1,job[i].block);

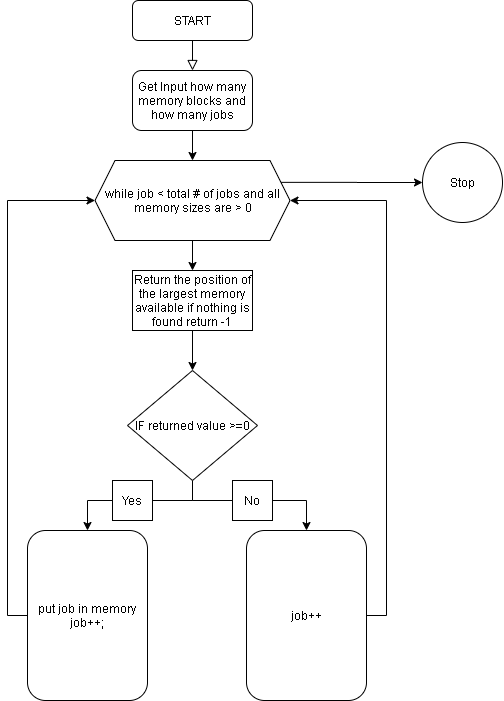
}

}

1. **Worst Fit-**

Definition: Worst Fit is the algorithm which allocates jobs to the highest memory free space available. It eliminates external fragmentation which allows more jobs to be allocated in the remaining free spaces allowing less spaces to be wasted.

Diagram:

  
Code:

int jnth=0,nth=0,min=-1,i=0,k=0,count=0;

typedef struct memory{

int jobs[200];

int jp;

int size;

int occupied;

}memory;

memory mem[200];

typedef struct jobs{

int block;

int size;

}jobs;

jobs job[200];

int check(){

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

if(mem[i].size>0){

return 0;

}

}

return 1;

}

int worst\_fit(int k){

int min=0,check=0;

//first check if job k is greater than all memory sizes

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

if(job[k].size<=mem[i].size){

check=1;

}

}

if(check==0){

return -1;

}

printf("hi");

//first find the largest memory size available

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

if(mem[i].size>mem[min].size && mem[i].occupied==0 && mem[i].size>0){

min=i;

}

}

printf("Min:%d k:%d\n",min,k+1);

return min;

}

void calc(void){

int min=0;

while(check()==0 && k<jnth){

min=worst\_fit(k);

if(min>=0){

// place the job in the memory and minus the jobs size to the memory

mem[min].jobs[mem[min].jp]=k;

mem[min].jp++;

job[k].block=min+1;

mem[min].occupied=1;

mem[min].size-=job[k].size;

}

k++;

}

}

int main()

{

printf("Worst Fit\n");

//inputs

//set memory

printf("How much Memory:");

scanf("%d",&nth);

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

printf("Bloc %d Size:",i+1);

scanf("%d",&mem[i].size);

}

//set jobs

printf("How many Jobs:");

scanf("%d",&jnth);

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

printf("Job %d Size:",i+1);

scanf("%d",&job[i].size);

}

printf("Batch Jobs:[JobName,SIZE]\n");

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

printf("Job%d: %d\n",i+1,job[i].size);

}

printf("\n\n");

calc();

printf("Memory:\n");

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

printf("Block %d INTERNAL FRAGMENTATION:%d\nJOBS:",i+1,mem[i].size);

if(mem[i].jp==0){

printf(" FREE SPACE");

}else{

for(int k=0;k<mem[i].jp;k++){

printf(" %d,",mem[i].jobs[k]+1);

}

}

printf("\n\n");

}

printf("JOB:\n");

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

if(job[i].block<=0){

printf("Job %d-> No Allocation\n",i+1);

}else{

printf("Job %d-> Block:%d\n",i+1,job[i].block);

}

}

return 0;

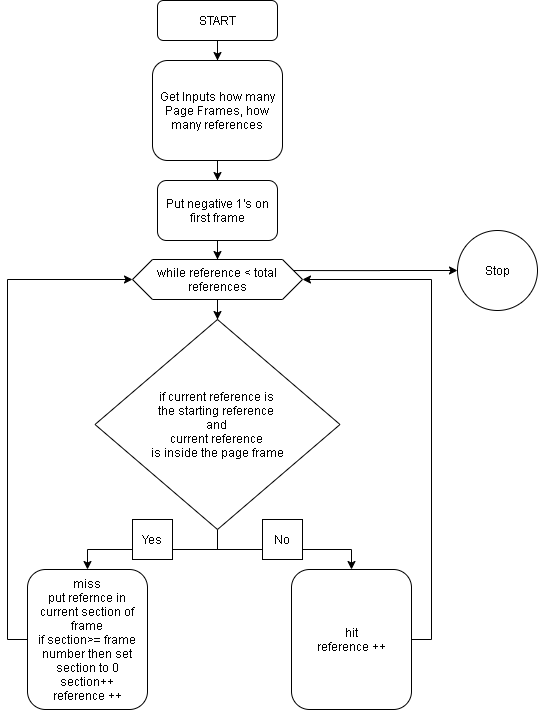
}

**Page Replacement Algorithms**

1. **FCFS-**

Definition: This algorithm simply lets the oldest job be replaced in page frame if all frames are occupied and lets new process take over it. Simply it works like a queue.

Diagram:



Code:

int pnth=0,frame\_nth=0,k=0,section=0;

typedef struct reference{

int name;

}reference;

reference ref[200];

typedef struct pages{

int references[200];

int rp;

//1=hit 0=miss

int status;

}pages;

pages page[200];

int inside\_frame(int current,int ref\_name){

for(int i=0;i<frame\_nth;i++){

if(current>0){

page[current].references[i]=page[current-1].references[i];

page[i].rp++;

}

}

for(int i=0;i<frame\_nth;i++){

if(page[current].references[i]== ref\_name){

// printf(" %d \n",page[current].references[i]);

return 1;

}

}

//printf("\n");

return 0;

}

void calc(void){

section=0;

//put negative ones in frame 1

for(int i=0;i<frame\_nth;i++){

page[0].references[i]=-1;

}

while(k<pnth){

if(k==0 || inside\_frame(k,ref[k].name)==0){

//miss

if(section>=frame\_nth){

section=0;

}

page[k].references[section]=ref[k].name;

page[k].status=0;

section++;

k++;

}else{

//hit

page[k].status=1;

k++;

}

}

}

int main()

{

printf("FCFS\n");

printf("How many Page Frames:");

scanf("%d",&frame\_nth);

printf("How many Reference:");

scanf("%d",&pnth);

printf("Referemce Name:\n");

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

printf("Reference %d:",i+1);

scanf("%d",&ref[i].name);

}

calc();

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

printf("\nFRAME %d: REFERENCE # %d\n",i+1,ref[i].name);

for(int k=0;k<frame\_nth;k++){

if(page[i].references[k]==-1){

printf("EMPTY");

}else{

printf("%d",page[i].references[k]);

}

printf("\n");

}

if(page[i].status==0){

printf("MISS");

}else{

printf("HIT");

}

}

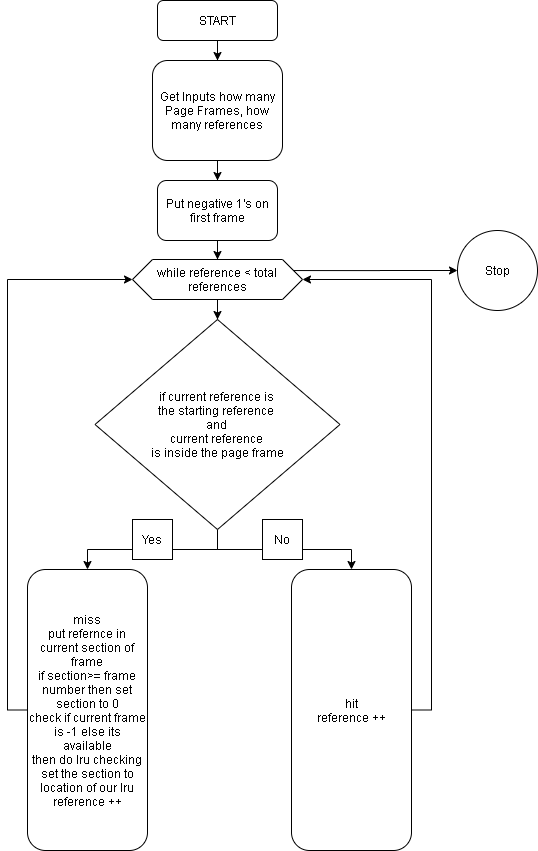
return 0;

}

1. **Least Recently Used-**

Definition: This algorithm changes its job in page frame by letting out the oldest job inside the page frame considering the time of the new incoming processes. It replaces the job that has ran longer inside the frame.

Diagram:

  
Code:

int pnth=0,frame\_nth=0,k=0,section=0;

typedef struct reference{

int name;

}reference;

reference ref[200];

typedef struct pages{

int references[200];

int rp;

//1=hit 0=miss

int status;

}pages;

pages page[200];

int inside\_frame(int current,int ref\_name){

for(int i=0;i<frame\_nth;i++){

if(current>0){

page[current].references[i]=page[current-1].references[i];

page[i].rp++;

}

}

for(int i=0;i<frame\_nth;i++){

if(page[current].references[i]== ref\_name){

// printf(" %d \n",page[current].references[i]);

return 1;

}

}

//printf("\n");

return 0;

}

int reverse\_longest(int current){

int loc=0,min=current-1,temp=0;

for(int i=0;i<frame\_nth;i++){

if(current>0){

page[current].references[i]=page[current-1].references[i];

page[i].rp++;

}

}

// for negative or empty frames

for(int i=0;i<frame\_nth;i++){

if(page[current].references[i]== -1){

loc=i;

return loc;

}

}

//lru

for(int i=0;i<frame\_nth;i++){

for(int x=current-1;x>=0;x--){

if(page[x].references[i]==ref[x].name){

temp=x;

break;

}

}

if(temp<min){

min=temp;

loc=i;

temp=0;

}

}

return loc;

}

void calc(void){

section=0;

//put negative ones in frame 1

for(int i=0;i<frame\_nth;i++){

page[0].references[i]=-1;

}

while(k<pnth){

if(k==0 || inside\_frame(k,ref[k].name)==0){

//miss

section=reverse\_longest(k);

page[k].references[section]=ref[k].name;

page[k].status=0;

k++;

}else{

//hit

page[k].status=1;

k++;

}

}

}

int main()

{

printf("LRU\n");

printf("How many Page Frames:");

scanf("%d",&frame\_nth);

printf("How many Reference:");

scanf("%d",&pnth);

printf("Referemce Name:\n");

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

printf("Reference %d:",i+1);

scanf("%d",&ref[i].name);

}

calc();

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

printf("\nFRAME %d: REFERENCE # %d\n",i+1,ref[i].name);

for(int k=0;k<frame\_nth;k++){

if(page[i].references[k]==-1){

printf("EMPTY");

}else{

printf("%d",page[i].references[k]);

}

printf("\n");

}

if(page[i].status==0){

printf("MISS");

}else{

printf("HIT");

}

}

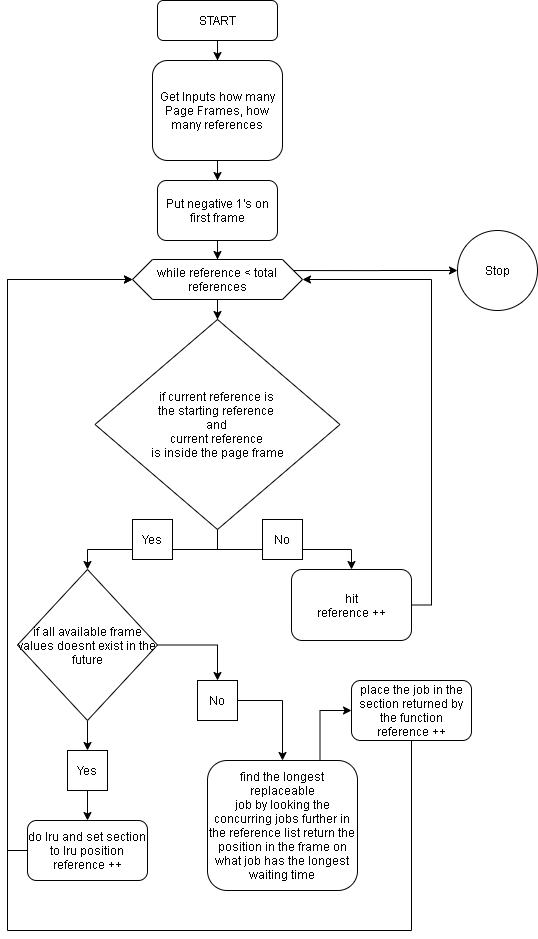
return 0;

}

1. **Optimal-**

Definition: Optimal Page Algorithm replaces the page which will not be used for the longest duration time. This algorithm needs to look at the next occurring jobs which job is not going to be process for the longest time and that page frame is going to be replaced.

Diagram:

  
Code:

int pnth=0,frame\_nth=0,k=0,section=0;

typedef struct reference{

int name;

}reference;

reference ref[200];

typedef struct pages{

int references[200];

int rp;

//1=hit 0=miss

int status;

}pages;

pages page[200];

int inside\_frame(int current,int ref\_name){

for(int i=0;i<frame\_nth;i++){

if(current>0){

page[current].references[i]=page[current-1].references[i];

page[i].rp++;

}

}

for(int i=0;i<frame\_nth;i++){

if(page[current].references[i]== ref\_name){

// printf(" %d \n",page[current].references[i]);

return 1;

}

}

//printf("\n");

return 0;

}

int exist(int start,int num){

for(int i=start+1;i<pnth;i++){

if(num==ref[i].name){

return 1;

}

}

return 0;

}

int find\_longest(int current){

int max=current,loc=0;

for(int i=0;i<frame\_nth;i++){

if(page[current].references[i]== -1){

loc=i;

break;

}

if(exist(current,page[current].references[i])==0){

loc=i;

return loc;

}

for(int x=current+1;x<pnth;x++){

if(page[current].references[i]==ref[x].name){

if(max<x){

max=x;

loc=i;

}break;

}

}

}

printf("current:%d max:%d\n",current,max);

//after find location of longest duration job return position of max

return loc;

}

int all\_exist(int current){

for(int i=0;i<frame\_nth;i++){

for(int x=current+1;x<pnth;x++){

if(page[current-1].references[i]==ref[x].name){

//printf("%d\n",page[current-1].references[0]);

return 1;

}

}

}

return 0;

}

int reverse\_longest(int current){

int loc=0,hops=0,temp=0;

for(int i=0;i<frame\_nth;i++){

if(current>0){

page[current].references[i]=page[current-1].references[i];

page[i].rp++;

}

}

for(int i=0;i<frame\_nth;i++){

for(int x=current-1;x>=0;x--){

if(page[x].references[i]==page[current].references[i]){

hops++;

}else{

break;

}

}

if(hops>temp){

loc=i;

temp=hops;

hops=0;

}

}

return loc;

}

void calc(void){

section=0;

//put negative ones in frame 1

for(int i=0;i<frame\_nth;i++){

page[0].references[i]=-1;

}

while(k<pnth){

if(k==0 || inside\_frame(k,ref[k].name)==0){

//miss

if(all\_exist(k)==0){

printf("k;%d all:%d\n",k,all\_exist(k));

section=reverse\_longest(k);

page[k].references[section]=ref[k].name;

page[k].status=0;

k++;

}else{

section=find\_longest(k);

page[k].references[section]=ref[k].name;

page[k].status=0;

k++;

}

}else{

//hit

page[k].status=1;

k++;

}

}

}

int main()

{

printf("Optimal\n");

printf("How many Page Frames:");

scanf("%d",&frame\_nth);

printf("How many Reference:");

scanf("%d",&pnth);

printf("Referemce Name:\n");

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

printf("Reference %d:",i+1);

scanf("%d",&ref[i].name);

}

calc();

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

printf("\nFRAME %d: REFERENCE # %d\n",i+1,ref[i].name);

for(int k=0;k<frame\_nth;k++){

if(page[i].references[k]==-1){

printf("EMPTY");

}else{

printf("%d",page[i].references[k]);

}

printf("\n");

}

if(page[i].status==0){

printf("MISS");

}else{

printf("HIT");

}

}

return 0; }

Name: Jayson P. Lana

Course/Section: BSCS 2A (NS)

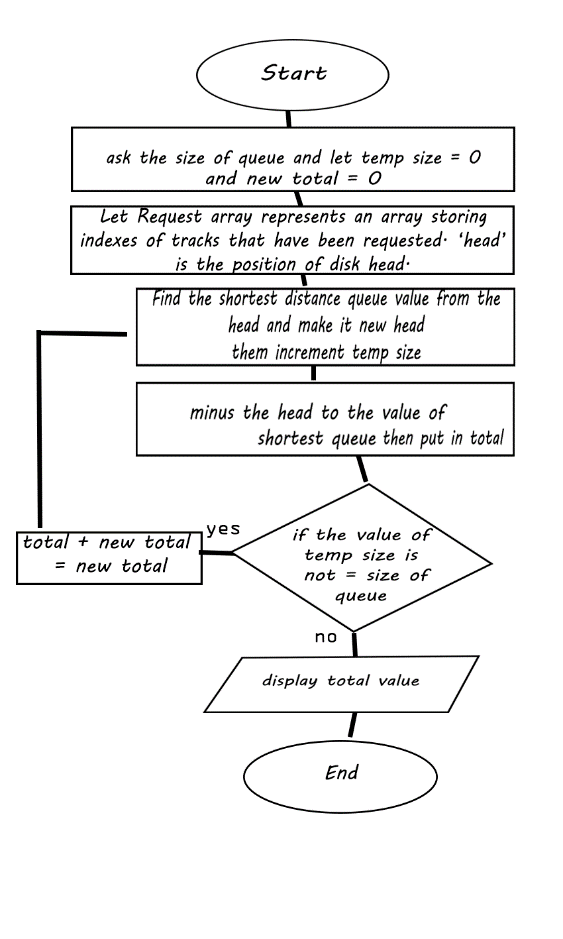
**Disk Scheduling Algorithms**

**Shortest Seek Time First (SSTF) –**  
Basic idea is the tracks which are closer to current disk head position should be serviced first in order to *minimise the seek operations*.

**Algorithm –**

1. Let Request array represents an array storing indexes of tracks that have been requested. ‘head’ is the position of disk head.
2. Find the positive distance of all tracks in the request array from head.
3. Find a track from requested array which has not been accessed/serviced yet and has minimum distance from head.
4. Increment the total seek count with this distance.
5. Currently serviced track position now becomes the new head position.
6. Go to step 2 until all tracks in request array have not been serviced.

**FLOWCHART**

****

**Code in C:**

#include<stdio.h>

#include<conio.h>

#include<math.h>

int main()

{

int queue[100],t[100],head,seek=0,n,i,j,temp;

float avg;

// clrscr();

printf("\*\*\* SSTF Disk Scheduling Algorithm \*\*\*\n");

printf("Enter the size of Queue\t");

scanf("%d",&n);

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

{

printf("Enter the Queue\t");

scanf("%d",&queue[i]);

}

printf("Enter the initial head position\t");

scanf("%d",&head);

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

t[i]=abs(head-queue[i]);

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

{

for(j=i+1;j<n;j++)

{

if(t[i]>t[j])

{

temp=t[i];

t[i]=t[j];

t[j]=temp;

temp=queue[i];

queue[i]=queue[j];

queue[j]=temp;

}

}

}

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

{

seek=seek+abs(head-queue[i]);

head=queue[i];

}

printf("\nTotal Seek Time is%d\t",seek);

avg=seek/(float)n;

return 0;

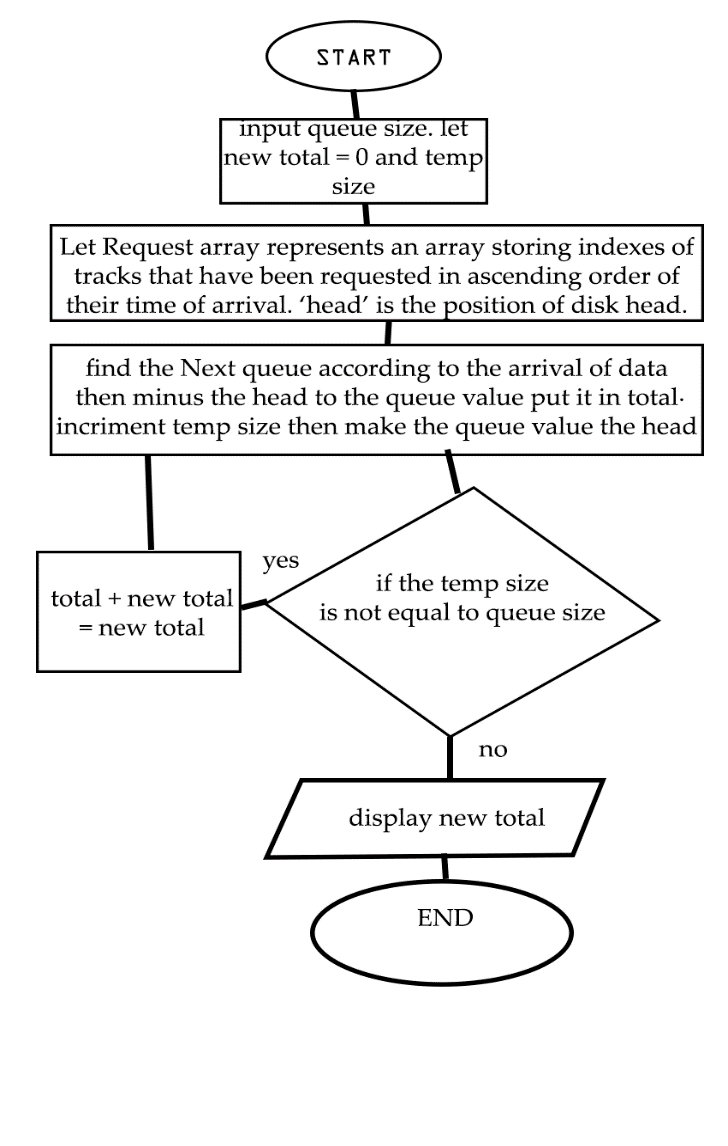
}

**First Come First Serve (FCFS)**  
FCFS is the simplest [disk scheduling algorithm](https://www.geeksforgeeks.org/disk-scheduling-algorithms/). As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue. The algorithm looks very fair and there is no starvation (all requests are serviced sequentially) but generally, it does not provide the fastest service.

**Algorithm:**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
3. Increment the total seek count with this distance.
4. Currently serviced track position now becomes the new head position.
5. Go to step 2 until all tracks in request array have not been serviced.

**FLOWCHART**



**CODE IN C:**

#include<stdio.h>

#include<math.h>

int main()

{

int queue[20],n,head,i,j,k,seek=0,diff;

float avg;

printf("\*\*\*FCFS Disk Scheduling\*\*\n");

printf("Enter the size of queue request\n");

scanf("%d",&n);

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

printf("Enter the queue value: ");

scanf("%d",&queue[i]);

}

printf("Enter the initial head position\n");

scanf("%d",&head);

queue[0]=head;

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

{

diff=abs(queue[j+1]-queue[j]);

seek+=diff;

printf("Disk head moves from %d to %d with seek %d\n",queue[j],queue[j+1],diff);

}

printf("Total seek time is %d\n",seek);

return 0;

}

**SCAN (Elevator) algorithm**  
In SCAN disk scheduling algorithm, head starts from one end of the disk and moves towards the other end, servicing requests in between one by one and reach the other end. Then the direction of the head is reversed and the process continues as head continuously scan back and forth to access the disk. So, this algorithm works as an elevator and hence also known as the **elevator algorithm**. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

**Algorithm-**

1. Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
2. Let direction represents whether the head is moving towards left or right.
3. In the direction in which head is moving service all tracks one by one.
4. Calculate the absolute distance of the track from the head.
5. Increment the total seek count with this distance.
6. Currently serviced track position now becomes the new head position.
7. Go to step 3 until we reach at one of the ends of the disk.
8. If we reach at the end of the disk reverse the direction and go to step 2 until all tracks in request array have not been serviced.

**FLOWCHART**

**Code in C++:**

#include<bits/stdc++.h>

using namespace std;

int main(){

int i,j,k,n,m,sum=0,x,y,h;

cout<<"Enter the size of disk\n";

cin>>m;

cout<<"Enter number of requests\n";

cin>>n;

cout<<"Enter the requests\n";

vector <int> a(n),b;

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

cin>>a[i];

}

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

if(a[i]>m){

cout<<"Error, Unknown position "<<a[i]<<"\n";

return 0;

}

}

cout<<"Enter the head position\n";

cin>>h;

int temp=h;

a.push\_back(h);

a.push\_back(m);

a.push\_back(0);

sort(a.begin(),a.end());

for(i=0;i<a.size();i++){

if(h==a[i])

break;

}

k=i;

if(k<n/2){

for(i=k;i<a.size();i++){

b.push\_back(a[i]);

}

for(i=k-1;i>=0;i--){

b.push\_back(a[i]);

}

}

else{

for(i=k;i>=0;i--){

b.push\_back(a[i]);

}

for(i=k+1;i<a.size();i++){

b.push\_back(a[i]);

}

}

temp=b[0];

cout<<temp;

for(i=1;i<b.size();i++){

cout<<" -> "<<b[i];

sum+=abs(b[i]-temp);

temp=b[i];

}

cout<<'\n';

cout<<"Total head movements = "<< sum<<'\n';

cout<<"Average head movement = "<<(float)sum/n<<'\n';

return 0;

}

Name: Aerom J. Canimo

Course/Section:BSCS-2A(NS)

**Banker’s Algorithm**

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

**Algorithm:**

**Banker's Algorithm Notations**

Here is an important notation used in Banker's algorithm:

* X: Indicates the total number of processes of the system.
* Y: Indicates the total number of resources present in the system.

**Available**

[I: Y] indicate which resource is available.

**Max**

[l:X,l: Y]: Expression of the maximum number of resources of type j or process i

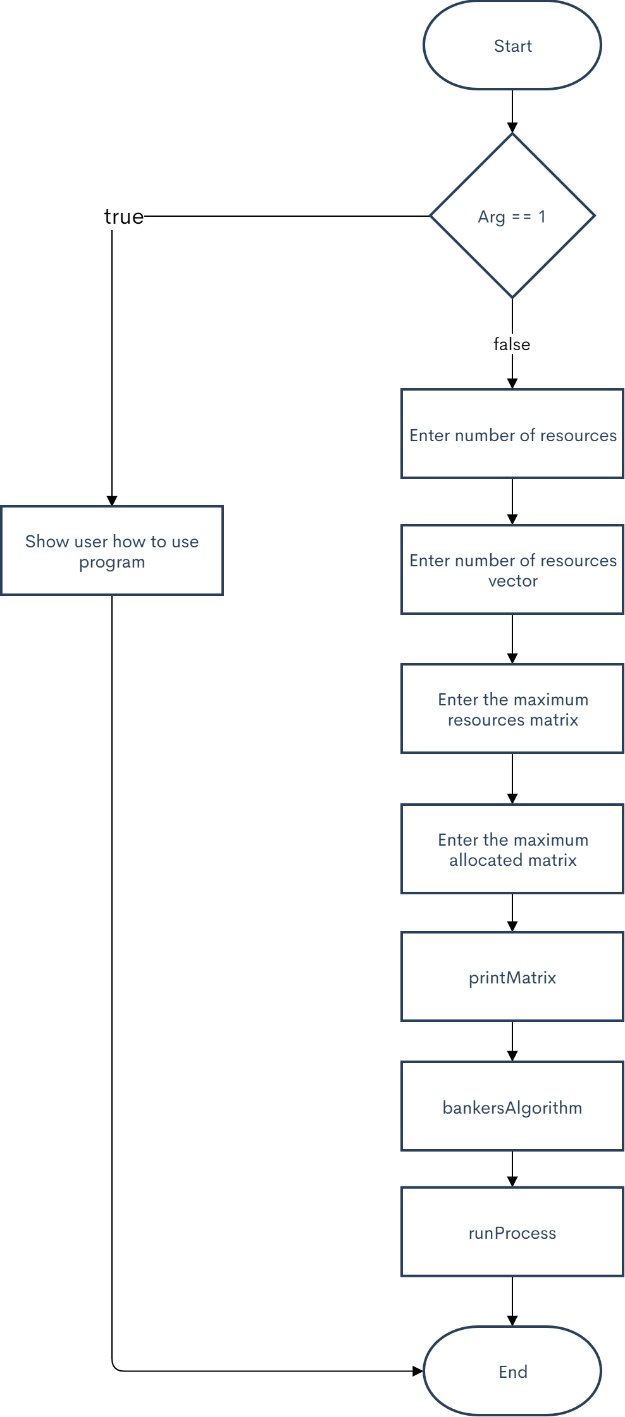
**Allocation**

[l:X,l:Y]. Indicate where process you have received a resource of type j

**Need**

Express how many more resources can be allocated in the future

**Flowchart:**



Name: Marcial Zipagan Jr.

Course/Section: BSCS 2A (NS)

1. **Belady’s Anomalys**

-In Operating System, process data is loaded in fixed sized chunks and each chunk is referred to as a page. The processor loads these pages in the fixed sized chunks of memory called frames. Typically, the size of each page is always equal to the frame size. A page fault occurs when a page is not found in the memory and needs to be loaded from the disk. If a page fault occurs and all memory frames have been already allocated, then replacement of a page in memory is required on the request of a new page. This is referred to as demand-paging. The choice of which page to replace is specified by a page replacement algorithms. The commonly used page replacement algorithms are FIFO, LRU, optimal page replacement algorithms etc.

Generally, on increasing the number of frames to a process’ virtual memory, its execution becomes faster as less number of page faults occur. Sometimes the reverse happens, i.e. more number of page faults occur when more frames are allocated to a process. This most unexpected result is termed as**Belady’s Anomaly**. Bélády’s anomaly is the name given to the phenomenon where increasing the number of page frames results in an increase in the number of page faults for a given memory access pattern.

This phenomenon is commonly experienced in following page replacement algorithms:

1. First in first out (FIFO)
2. Second chance algorithm
3. Random page replacement algorithm
4. **Thrashing**

- occurs when a system spends more time processing page faults than executing transactions. While processing page faults is necessary to in order to appreciate the benefits of virtual memory. Thrashing has a negative effect on the system. As the page fault rate increases, more transactions need processing from the paging device. The queue at the paging device increases, resulting in increased service time for a page fault . While the transactions in the system are waiting for the paging device, CPU utilization, system throughput and system response time decrease, resulting in below optimal performance of a system. Thrashing becomes a greater threat as the [degree of multiprogramming](http://denninginstitute.com/modules/vm/green/multip.html) of the system increases.

The graph shows that there is a degree of multiprogramming that is optimal for [system performance](http://denninginstitute.com/modules/vm/green/sysper.html) CPU utilization reaches a maximum before a swift decline as the degree of multiprogramming increases and thrashing occurs in the over-extended system. This indicates that controlling the load on the system is important to avoiding thrashing. In the system represented by the graph, it is important to maintain the multiprogramming degree that corresponds to the peak of the graph. The selection of [a replacement policy](http://denninginstitute.com/modules/vm/yellow/repol.html) to implement virtual memory plays an important role in reducing possibility of thrashing occurring. A policy based on [the local mode](http://denninginstitute.com/modules/vm/bluyel/mempar.html) will tend to limit the effect of thrashing. In local mode, a transaction will replace pages from its assigned [partition](http://denninginstitute.com/modules/vm/bluyel/mempar.html). Its need to access memory will not affect transactions using other partitions. If other transactions have enough page frames in the partitions they occupy, they will continue to be processed efficiently. A replacement policy based on [the global mode](http://denninginstitute.com/modules/vm/bluyel/mempar.html) is more likely to cause thrashing. Since all pages of memory are available to all transactions, a memory-intensive transaction may occupy a large portion of memory, making other transactions susceptible to page faults and resulting in a system that thrashes. Other ways of preventing thrashing include using the [working set](http://denninginstitute.com/modules/vm/orange/ws.html) strategy, preparing a transactions partition, and increasing page size.