

Operating Systems

DIGITAL ASSIGNMENT-1

Name: **VIBHU KUMAR SINGH**

Reg. No: **19BCE0215**

Teacher: **Manikandan K.**

**Q1. Implement a program to allocate memory by applying the following strategies.**

**a. FIRST FIT**

**b. BEST FIT**

**c. WORST FIT**

**A1.**

a)First Fit-

**ALGORITHM-**

1. Input the no. of Processes and no. of blocks.

2. After that get the size of each block and process requests.

3. Now allocate processes

if(block size >= process size)

//allocate the process

else

//move on to next block

4. Display the processes with the blocks that are allocated to a respective process.

5. Stop.

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

struct process

{

int id;

int memory\_required;

int allocated\_partition;

int external\_fragment;

bool allocated;

};

struct partition

{

int id;

int size;

int internal\_fragment;

bool alloted;

};

int main()

{

int memory,no\_of\_partitions,no\_of\_processes,i,j,sum=0;

printf("Enter total memory: ");

scanf("%d",&memory);

printf("Enter number of partitions: ");

scanf("%d",&no\_of\_partitions);

struct partition parti[no\_of\_partitions];

for(i=0;i<no\_of\_partitions;i++)

{

printf("Enter memory for partition%d: ",i+1);

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

parti[i].id = i+1;

parti[i].alloted=false;

sum+=parti[i].size;

}

if(sum < memory)

{

no\_of\_partitions++;

parti[i].size = memory - sum;

parti[i].id = i+1;

parti[i].alloted=false;

printf("Size of partition%d: %d\n",i,parti[i].size );

}

int total\_internal\_fragment=0, total\_external\_fragment=0;

printf("Enter number of processes: ");

scanf("%d",&no\_of\_processes);

struct process proc[no\_of\_processes];

for (i = 0; i < no\_of\_processes; ++i)

{

printf("Enter memory required for process%d: ",i+1 );

scanf("%d",&proc[i].memory\_required);

proc[i].id = i+1;

proc[i].external\_fragment=0;

proc[i].allocated = false;

for(j=0;j<no\_of\_partitions;j++)

{

if(proc[i].memory\_required<=parti[j].size && !parti[j].alloted)

{

proc[i].allocated = true;

proc[i].allocated\_partition = parti[j].id;

parti[j].internal\_fragment = parti[j].size - proc[i].memory\_required;

total\_internal\_fragment+=parti[j].internal\_fragment;

parti[j].alloted=true;

break;

}

}

for(j=0;j<no\_of\_partitions;j++)

{

if(parti[j].alloted==false)

{

proc[i].external\_fragment+=parti[j].size;

}

}

}

for(j=0;j<no\_of\_partitions;j++)

{

if(!parti[j].alloted)

{

total\_external\_fragment+=parti[j].size;

parti[j].internal\_fragment = -1;

}

}

printf("ProcessID\tMemory required\tAllocated\tAllocated Partition\n");

for(i=0;i<no\_of\_processes;i++)

{

if(proc[i].allocated)

{

printf("%d\t\t%d\t\t\tYES\t%d\t\n",proc[i].id, proc[i].memory\_required, proc[i].allocated\_partition);

}

else

{

printf("%d\t\t%d\t\t\tNO\t\t%d\n",proc[i].id, proc[i].memory\_required, proc[i].external\_fragment);

}

}

printf("\n");

printf("Internal Fragmentation\n");

for(i=0;i<no\_of\_partitions;i++)

{

if(parti[i].internal\_fragment==-1)

{

printf("%d ---\n",parti[i].id);

}

else

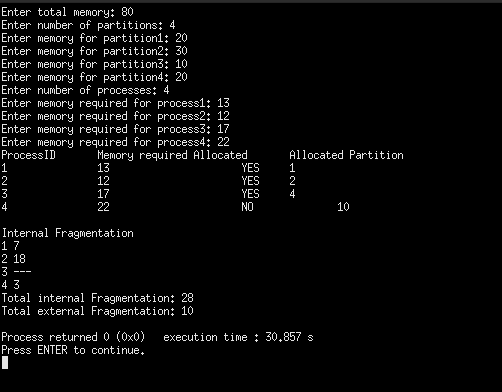
printf("%d %d\n",parti[i].id,parti[i].internal\_fragment );

}

printf("Total internal Fragmentation: %d\nTotal external Fragmentation: %d\n",total\_internal\_fragment, total\_external\_fragment);

}

**OUTPUT-**



b)BEST FIT-

**ALGORITHM-**

1)Enter the memory blocks with size.

2)Enter the process blocks with size.

3)Set all the memory blocks as free.

4)Start by picking up each process

5)Find the minimum block size that is best to assign to the current process.

6)If the best fit memory size is found, it is allocated to the process.

7)If the memory block and memory demand do not match, leave the process and search for another process.

8)End

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

struct process

{

int id;

int memory\_required;

int allocated\_partition;

int external\_fragment;

bool allocated;

};

struct partition

{

int id;

int size;

int internal\_fragment;

bool alloted;

};

int main()

{

int memory,no\_of\_partitions,no\_of\_processes,i,j,sum=0;

printf("Enter total memory: ");

scanf("%d",&memory);

printf("Enter number of partitions: ");

scanf("%d",&no\_of\_partitions);

struct partition parti[no\_of\_partitions];

for(i=0;i<no\_of\_partitions;i++)

{

printf("Enter memory for partition%d: ",i+1);

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

parti[i].id = i+1;

parti[i].alloted=false;

sum+=parti[i].size;

}

if(sum < memory)

{

no\_of\_partitions++;

parti[i].size = memory - sum;

printf("Size of partition%d: %d\n",i+1,parti[i].size );

parti[i].id = i+1;

parti[i].alloted = false;

}

for(i=0;i<no\_of\_partitions-1;i++)

{

for(j=0;j<no\_of\_partitions-1-i;j++)

{

if(parti[j].size>parti[j+1].size)

{

struct partition temp = parti[j];

parti[j] = parti[j+1];

parti[j+1] = temp;

}

}

}

int total\_internal\_fragment=0, total\_external\_fragment=0;

printf("Enter number of processes: ");

scanf("%d",&no\_of\_processes);

struct process proc[no\_of\_processes];

for (i = 0; i < no\_of\_processes; ++i)

{

printf("Enter memory required for process%d: ",i+1 );

scanf("%d",&proc[i].memory\_required);

proc[i].id = i+1;

proc[i].external\_fragment=0;

proc[i].allocated = false;

for(j=0;j<no\_of\_partitions;j++)

{

if(proc[i].memory\_required<=parti[j].size && !parti[j].alloted)

{

proc[i].allocated = true;

proc[i].allocated\_partition = parti[j].id;

parti[j].internal\_fragment = parti[j].size - proc[i].memory\_required;

total\_internal\_fragment+=parti[j].internal\_fragment;

parti[j].alloted=true;

break;

}

}

for(j=0;j<no\_of\_partitions;j++)

{

if(parti[j].alloted==false)

{

proc[i].external\_fragment+=parti[j].size;

}

}

}

for(j=0;j<no\_of\_partitions;j++)

{

if(!parti[j].alloted)

{

total\_external\_fragment+=parti[j].size;

parti[j].internal\_fragment = -1;

}

}

printf("ProcessID\tMemory required\tAllocated\tAllocated Partition\n");

for(i=0;i<no\_of\_processes;i++)

{

if(proc[i].allocated)

{

printf("%d\t\t%d\t\t\tYES\t%d\t\n",proc[i].id, proc[i].memory\_required, proc[i].allocated\_partition);

}

else

{

printf("%d\t\t%d\t\t\tNO\t%d\n",proc[i].id, proc[i].memory\_required, proc[i].external\_fragment);

}

}

printf("\n");

printf("Internal Fragmentation\n");

for(i=0;i<no\_of\_partitions;i++)

{

if(parti[i].internal\_fragment==-1)

{

printf("%d ---\n",parti[i].id);

}

else

printf("%d %d\n",parti[i].id,parti[i].internal\_fragment );

}

printf("Total internal Fragmentation: %d\nTotal external Fragmentation: %d\n",total\_internal\_fragment, total\_external\_fragment);

while(1)

{

char choice,dummy;

printf("Do you want to continue? (Y/n) : ");

getchar();

scanf("%c",&choice);

if(choice=='Y')

{

int id,new\_memory\_required,allocated\_partition;

bool allocated;

printf("Enter id of the process leaving: ");

scanf("%d",&id);

printf("Enter memory required for the new processs: ");

scanf("%d",&new\_memory\_required);

int partition\_to\_remove = proc[id-1].allocated\_partition;

int memory\_to\_remove = proc[id-1].memory\_required;

printf("Partition to remove: %d\n", partition\_to\_remove);

for(i=0;i<no\_of\_partitions;i++)

{

if(parti[i].id == partition\_to\_remove)

{

parti[i].alloted = false;

}

}

allocated = false;

for(j=0;j<no\_of\_partitions;j++)

{

if(new\_memory\_required<=parti[j].size && !parti[j].alloted)

{

allocated = true;

allocated\_partition = parti[j].id;

parti[j].internal\_fragment = parti[j].size - new\_memory\_required;

parti[j].alloted=true;

break;

}

}

if(allocated)

{

printf("Process allocated in partition: %d\n",allocated\_partition );

printf("New internal Fragmentation for the partition = %d\n",parti[j].internal\_fragment );

}

else

{

printf("Process cannot be allocated\n");

}

}

else{

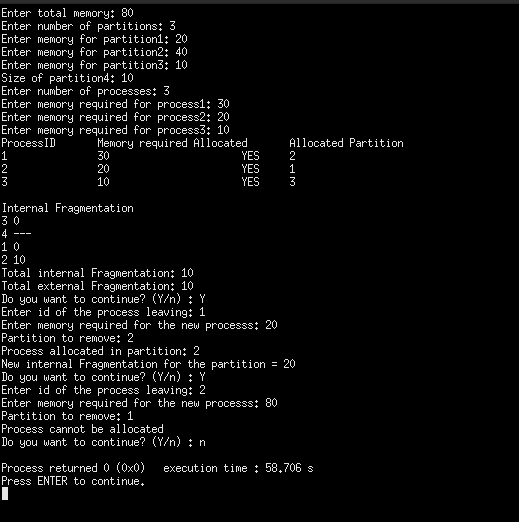
return 0;

}

}

}

**OUTPUT-**



C)WORST FIT-

**ALGORITHM-**

1)Input memory block with a size.

2)Input process with size.

3)Initialize by selecting each process to find the maximum block size that can be assigned to the current process.

4)If the condition does not fulfill, they leave the process.

5)If the condition is not fulfilled, then leave the process and check for the next process.

6)Stop.

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

struct process

{

int id;

int memory\_required;

int allocated\_partition;

int external\_fragment;

bool allocated;

};

struct partition

{

int id;

int size;

int internal\_fragment;

bool alloted;

};

int main()

{

int memory,no\_of\_partitions,no\_of\_processes,i,j,sum=0;

printf("Enter total memory: ");

scanf("%d",&memory);

printf("Enter number of partitions: ");

scanf("%d",&no\_of\_partitions);

struct partition parti[no\_of\_partitions];

for(i=0;i<no\_of\_partitions;i++)

{

printf("Enter memory for partition%d: ",i+1);

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

parti[i].id = i+1;

parti[i].alloted=false;

sum+=parti[i].size;

}

if(sum < memory)

{

no\_of\_partitions++;

parti[i].size = memory - sum;

parti[i].id = i+1;

parti[i].alloted=false;

printf("Size of partition%d: %d\n",i+1,parti[i].size );

}

for(i=0;i<no\_of\_partitions-1;i++)

{

for(j=0;j<no\_of\_partitions-1-i;j++)

{

if(parti[j].size<parti[j+1].size)

{

struct partition temp = parti[j];

parti[j] = parti[j+1];

parti[j+1] = temp;

}

}

}

int total\_internal\_fragment=0, total\_external\_fragment=0;

printf("Enter number of processes: ");

scanf("%d",&no\_of\_processes);

struct process proc[no\_of\_processes];

for (i = 0; i < no\_of\_processes; ++i)

{

printf("Enter memory required for process%d: ",i+1 );

scanf("%d",&proc[i].memory\_required);

proc[i].id = i+1;

proc[i].external\_fragment=0;

proc[i].allocated = false;

for(j=0;j<no\_of\_partitions;j++)

{

if(proc[i].memory\_required<=parti[j].size && !parti[j].alloted)

{

proc[i].allocated = true;

proc[i].allocated\_partition = parti[j].id;

parti[j].internal\_fragment = parti[j].size - proc[i].memory\_required;

total\_internal\_fragment+=parti[j].internal\_fragment;

parti[j].alloted=true;

break;

}

}

for(j=0;j<no\_of\_partitions;j++)

{

if(parti[j].alloted==false)

{

proc[i].external\_fragment+=parti[j].size;

}

}

}

for(j=0;j<no\_of\_partitions;j++)

{

if(!parti[j].alloted)

{

total\_external\_fragment+=parti[j].size;

parti[j].internal\_fragment = -1;

}

}

printf("ProcessID\tMemory required\tAllocated\tAllocated Partition\n");

for(i=0;i<no\_of\_processes;i++)

{

if(proc[i].allocated)

{

printf("%d\t\t%d\t\t\tYES\t%d\t\n",proc[i].id, proc[i].memory\_required, proc[i].allocated\_partition);

}

else

{

printf("%d\t\t%d\t\t\tNO\t%d\n",proc[i].id, proc[i].memory\_required, proc[i].external\_fragment);

}

}

printf("\n");

printf("Internal Fragmentation\n");

for(i=0;i<no\_of\_partitions;i++)

{

if(parti[i].internal\_fragment==-1)

{

printf("%d ---\n",parti[i].id);

}

else

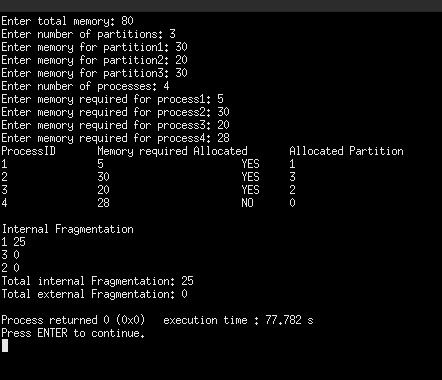
printf("%d %d\n",parti[i].id,parti[i].internal\_fragment );

}

printf("Total internal Fragmentation: %d\nTotal external Fragmentation: %d\n",total\_internal\_fragment, total\_external\_fragment);

}

**OUTPUT-**



**Q2. Implement a program for page replacement using the following:**

**a. FIFO**

**b. LRU**

**c. OPTIMAL**

**A2.**

**a)FIFO**

**ALGORITHM-**

Step 1. Start to traverse the pages.

Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.

Step 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.

Step 4. If the current page is present in the memory, do nothing.

Step 5. Else, pop the topmost page from the queue as it was inserted first.

Step 6. Replace the topmost page with the current page from the string.

Step 7. Increment the page faults.

Step 8. Stop

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

int pointer;

int faults ,hits;

void print(int frame\_size,int frame[])

{

int i;

for(i=0;i<frame\_size;i++)

{

if(frame[i]==-1)

printf("- ");

else

printf("%d ",frame[i]);

}

printf("\n");

}

void add\_reference(int frame\_size,int frame[], int reference)

{

int i;

bool alloted = false;

for(i=0;i<frame\_size;i++)

{

if(frame[i]==reference)

{

alloted = true;

printf(" Hit for %d | ", reference);

hits++;

break;

}

else if(frame[i]==-1)

{

alloted = true;

frame[i] = reference;

printf("Fault for %d | ", reference);

faults++;

break;

}

}

if(alloted == false)

{

faults++;

printf("Fault for %d | ", reference);

frame[pointer] = reference;

pointer = (pointer+1)%frame\_size;

}

print(frame\_size, frame);

}

int main()

{

int frame\_size,i,number\_of\_references;

printf("Enter frame size: ");

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

int frame[frame\_size];

for(i=0;i<frame\_size;i++)

{

frame[i] = -1;

}

print(frame\_size,frame);

printf("Enter the number of references: ");

scanf("%d",&number\_of\_references);

int reference[number\_of\_references];

for(i=0;i<number\_of\_references;i++)

{

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

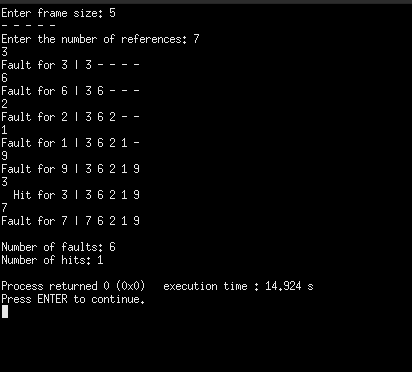
add\_reference(frame\_size,frame,reference[i]);

}

printf("\nNumber of faults: %d \nNumber of hits: %d\n",faults,hits );

}

**OUTPUT-**

**

**b)LRU**

**ALGORITHM-**

Step 1. Start the process

Step 2. Declare the page size

Step 3. Determine the number of pages to be inserted.

Step 4. Get the value.

Step 5. Declare the counter and stack value.

Step 6. Choose the least recently used page by the counter value.

Step 7. Stack them as per the selection.

Step 8. Display the values.

Step 9. Terminate the process.

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

int pointer;

int faults ,hits;

void print(int frame\_size,int frame[])

{

int i;

//printf("Printing the Frames: ");

for(i=0;i<frame\_size;i++)

{

if(frame[i]==-1)

printf("- ");

else

printf("%d ",frame[i]);

}

printf("\n");

}

int predict(int reference\_length, int references[], int page\_no ,int frame\_size,int frame[], int start)

{

int pos = -1, farthest = start, i;

for(i=0;i<frame\_size;i++)

{

int j;

for(j=start-1;j>=0;j--)

{

if(frame[i]==references[j])

{

if(j<farthest)

{

farthest=j;

pos=i;

}

break;

}

}

if(j==page\_no)

return i;

}

if(pos == -1)

return 0;

else

return pos;

}

void add\_reference(int frame\_size,int frame[], int reference, int current\_position,int reference\_length, int references[])

{

int i;

bool allocated=false;

for(i=0;i<frame\_size;i++)

{

if(frame[i]==reference)

{

printf(" Hit for %d | ", reference);

hits++;

allocated = true;

break;

}

else if(frame[i]==-1)

{

frame[i] = reference;

printf("Fault for %d | ", reference);

faults++;

allocated = true;

break;

}

}

if(allocated==false)

{

int j = predict(reference\_length,references,current\_position,frame\_size,frame,current\_position+1);

frame[j] = reference;

printf("Fault for %d | ", reference);

faults++;

}

print(frame\_size, frame);

}

int main()

{

int frame\_size,i,number\_of\_references;

printf("Enter frame size: ");

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

int frame[frame\_size];

for(i=0;i<frame\_size;i++)

{

frame[i] = -1;

}

print(frame\_size,frame);

printf("Enter the number of references: ");

scanf("%d",&number\_of\_references);

int reference[number\_of\_references];

for(i=0;i<number\_of\_references;i++)

{

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

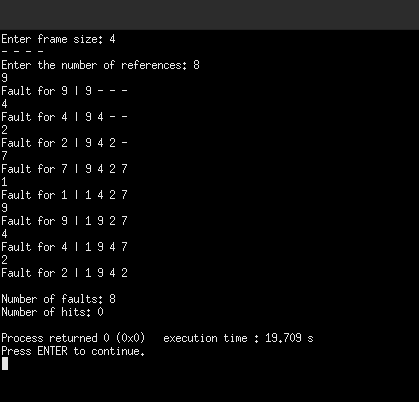
add\_reference(frame\_size,frame,reference[i],i,number\_of\_references,reference);

}

printf("\nNumber of faults: %d \nNumber of hits: %d\n",faults,hits );

}

**OUTPUT-**



**c)OPTIMAL**

**ALGORITHM-**

Step 1: Push the first page in the stack as per the memory demand.

Step 2:Push the second page as per the memory demand.

Step 3:Push the third page until the memory is full.

Step 4:As the queue is full, the page which is least recently used is popped.

Step 5:repeat step 4 until the page demand continues and until the processing is over.

Step 6:Terminate the program.

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

int pointer;

int faults ,hits;

void print(int frame\_size,int frame[])

{

int i;

for(i=0;i<frame\_size;i++)

{

if(frame[i]==-1)

printf("- ");

else

printf("%d ",frame[i]);

}

printf("\n");

}

int predict(int reference\_length, int references[], int page\_no ,int frame\_size,int frame[], int start)

{

int pos = -1, farthest = start, i;

for(i=0;i<frame\_size;i++)

{

int j;

for(j=start;j<reference\_length;j++)

{

if(frame[i]==references[j])

{

if(j>farthest)

{

farthest=j;

pos=i;

}

break;

}

}

if(j==page\_no)

return i;

}

if(pos == -1)

return 0;

else

return pos;

}

void add\_reference(int frame\_size,int frame[], int reference, int current\_position,int reference\_length, int references[])

{

int i;

bool allocated=false;

for(i=0;i<frame\_size;i++)

{

if(frame[i]==reference)

{

printf(" Hit for %d | ", reference);

hits++;

allocated = true;

break;

}

else if(frame[i]==-1)

{

frame[i] = reference;

printf("Fault for %d | ", reference);

faults++;

allocated = true;

break;

}

}

if(allocated==false)

{

int j = predict(reference\_length,references,current\_position,frame\_size,frame,current\_position+1);

frame[j] = reference;

printf("Fault for %d | ", reference);

faults++;

}

print(frame\_size, frame);

}

int main()

{

int frame\_size,i,number\_of\_references;

printf("Enter frame size: ");

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

int frame[frame\_size];

for(i=0;i<frame\_size;i++)

{

frame[i] = -1;

}

print(frame\_size,frame);

printf("Enter the number of references: ");

scanf("%d",&number\_of\_references);

int reference[number\_of\_references];

for(i=0;i<number\_of\_references;i++)

{

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

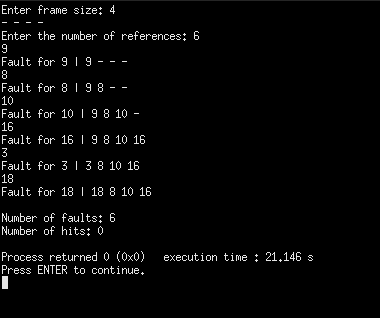
add\_reference(frame\_size,frame,reference[i],i,number\_of\_references,reference);

}

printf("\nNumber of faults: %d \nNumber of hits: %d\n",faults,hits );

}

**OUTPUT-**



**Q3.Simulate with a program to schedule disk in seek optimization.**

**A3.**

**FCFS-**

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

int main()

{

int i,no\_of\_requests,initial\_head;

printf("Enter the number of requests: ");

scanf("%d",&no\_of\_requests);

int request[no\_of\_requests];

printf("Enter the requests: ");

for (i = 0; i < no\_of\_requests; ++i)

{

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

}

printf("Enter initial position of R/W head: ");

scanf("%d",&initial\_head);

int seek\_time=0;

printf("%d -> ",initial\_head );

for(i=0;i<no\_of\_requests;i++)

{

if(i == no\_of\_requests-1)

printf("%d\n", request[i] );

else

printf("%d -> ", request[i] );

seek\_time += abs(request[i] - initial\_head);

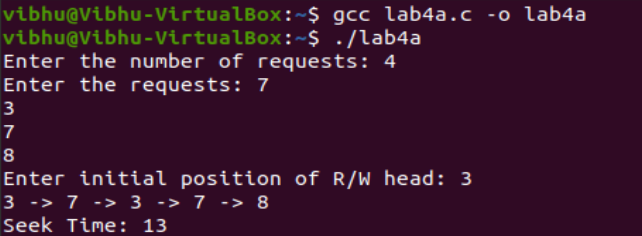
initial\_head = request[i];

}

printf("Seek Time: %d\n", seek\_time);

}

**OUTPUT-**



**SSTF-**

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

struct request

{

int request\_track\_number;

bool visited;

};

int main()

{

int i,no\_of\_requests,initial\_head,limit,j,choice,previous\_head;

printf("Enter the number of requests: ");

scanf("%d",&no\_of\_requests);

struct request req[no\_of\_requests];

printf("Enter the requests: ");

for (i = 0; i < no\_of\_requests; ++i)

{

scanf("%d",&req[i].request\_track\_number);

req[i].visited = false;

}

printf("Enter initial position of R/W head: ");

scanf("%d",&initial\_head);

int seek\_time=0;

printf("%d -> ",initial\_head );

int n = no\_of\_requests;

while(n)

{

int min = 1e9;

int min\_track\_number, position;

for(i=0;i<no\_of\_requests;i++)

{

if(abs(initial\_head - req[i].request\_track\_number) < min && req[i].visited == false)

{

min = abs(initial\_head - req[i].request\_track\_number);

min\_track\_number = req[i].request\_track\_number;

position = i;

}

}

initial\_head = req[position].request\_track\_number;

req[position].visited = true;

printf("%d ->",min\_track\_number);

seek\_time += min;

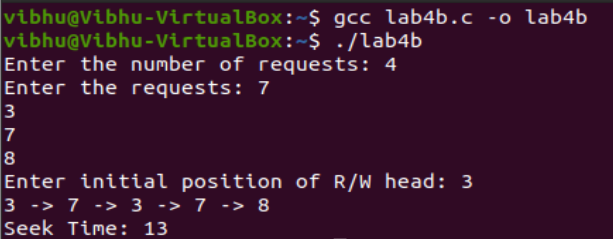
n--;

}

printf("\nSeek Time: %d\n", seek\_time);

}

**OUTPUT-**



**SCAN-**

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

struct request

{

int request\_track\_number;

bool visited;

};

int main()

{

int i,no\_of\_requests,initial\_head,limit,j,choice,previous\_head;

printf("Enter the number of requests: ");

scanf("%d",&no\_of\_requests);

struct request req[no\_of\_requests];

printf("Enter the requests: ");

for (i = 0; i < no\_of\_requests; ++i)

{

scanf("%d",&req[i].request\_track\_number);

req[i].visited = false;

}

printf("Enter initial position of R/W head: ");

scanf("%d",&initial\_head);

printf("Enter the previous position of R/W head: ");

scanf("%d",&previous\_head);

printf("Enter the cylinder size: ");

scanf("%d",&limit);

if(previous\_head - initial\_head > 0 )

{

choice = 2;

}

else

choice = 1;

//scanf("%d",&choice);

int seek\_time=0;

printf("%d -> ",initial\_head );

if(choice == 1)

{

for(i=initial\_head;i<limit;i++)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

printf("%d -> ", limit-1);

seek\_time += abs(limit-1 - initial\_head);

initial\_head = limit-1;

for(i=initial\_head;i>=0;i--)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

seek\_time += abs(initial\_head - 0);

printf("0 \n");

}

else if(choice == 2)

{

for(i=initial\_head;i>=0;i--)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

printf("%d -> ", 0);

seek\_time += abs(0 - initial\_head);

initial\_head = 0;

for(i=initial\_head;i<limit;i++)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

seek\_time += abs(limit-1 - initial\_head );

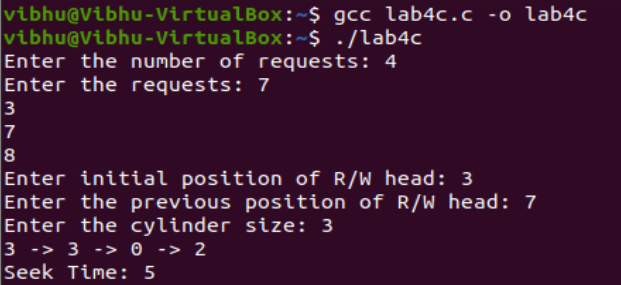
printf("%d \n", limit-1);

}

printf("Seek Time: %d\n", seek\_time);

}

**OUTPUT-**

**

**C-SCAN-**

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

struct request

{

int request\_track\_number;

bool visited;

};

int main()

{

int i,no\_of\_requests,initial\_head,limit,j,choice,previous\_head;

printf("Enter the number of requests: ");

scanf("%d",&no\_of\_requests);

struct request req[no\_of\_requests];

printf("Enter the requests: ");

for (i = 0; i < no\_of\_requests; ++i)

{

scanf("%d",&req[i].request\_track\_number);

req[i].visited = false;

}

printf("Enter initial position of R/W head: ");

scanf("%d",&initial\_head);

printf("Enter the previous position of R/W head: ");

scanf("%d",&previous\_head);

printf("Enter the cylinder size: ");

scanf("%d",&limit);

if(previous\_head - initial\_head > 0 )

{

choice = 2;

}

else

choice = 1;

//scanf("%d",&choice);

int seek\_time=0;

printf("%d -> ",initial\_head );

int cp\_initial\_head = initial\_head;

if(choice == 1)

{

for(i=initial\_head;i<limit;i++)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

printf("%d -> \n", limit-1);

seek\_time += abs(limit-1 - initial\_head);

initial\_head = 0;

for(i=0;i<cp\_initial\_head;i++)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

printf("\n");

}

else if(choice == 2)

{

for(i=initial\_head;i>=0;i--)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

printf("%d -> ", 0 );

seek\_time += abs(initial\_head - 0);

initial\_head = limit-1;

for(i=limit;i>cp\_initial\_head;i--)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

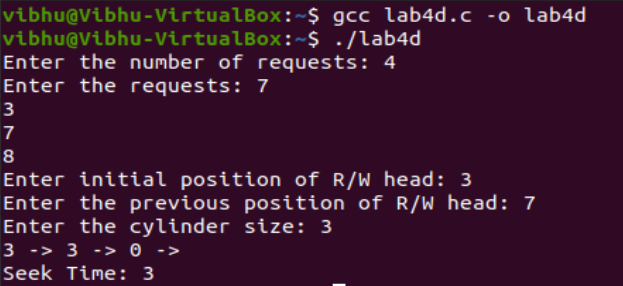
printf("\n");

}

printf("Seek Time: %d\n", seek\_time);

}

**OUTPUT-**

**

**C-LOOK-**

**SOURCE CODE-**

#include "stdio.h"

#include "stdlib.h"

#include "stdbool.h"

struct request

{

int request\_track\_number;

bool visited;

};

int main()

{

int i,no\_of\_requests,initial\_head,limit,j,choice,previous\_head;

printf("Enter the number of requests: ");

scanf("%d",&no\_of\_requests);

struct request req[no\_of\_requests];

printf("Enter the requests: ");

for (i = 0; i < no\_of\_requests; ++i)

{

scanf("%d",&req[i].request\_track\_number);

req[i].visited = false;

}

printf("Enter initial position of R/W head: ");

scanf("%d",&initial\_head);

printf("Enter the previous position of R/W head: ");

scanf("%d",&previous\_head);

printf("Enter the cylinder size: ");

scanf("%d",&limit);

if(previous\_head - initial\_head > 0 )

{

choice = 2;

}

else

choice = 1;

int seek\_time=0;

printf("%d -> ",initial\_head );

int cp\_initial\_head = initial\_head;

if(choice == 1)

{

for(i=initial\_head;i<limit;i++)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

initial\_head = 0;

for(i=0;i<cp\_initial\_head;i++)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

printf("\n");

}

else if(choice == 2)

{

for(i=initial\_head;i>=0;i--)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

initial\_head = limit-1;

for(i=limit;i>cp\_initial\_head;i--)

{

for(j=0;j<no\_of\_requests;j++)

{

if(req[j].request\_track\_number == i && req[j].visited == false)

{

printf("%d -> ", req[j].request\_track\_number);

req[j].visited = true;

seek\_time += abs(req[j].request\_track\_number - initial\_head);

initial\_head = req[j].request\_track\_number;

}

}

}

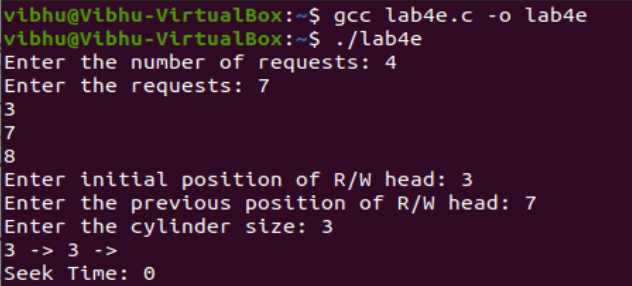
printf("\n");

}

printf("Seek Time: %d\n", seek\_time);

}

**OUTPUT-**

**