Netaji Subhas University of Technology

COCSC09

Operating System

Sandeep Chaudhary

2019UIC3574

S. No	Topic	Page No.

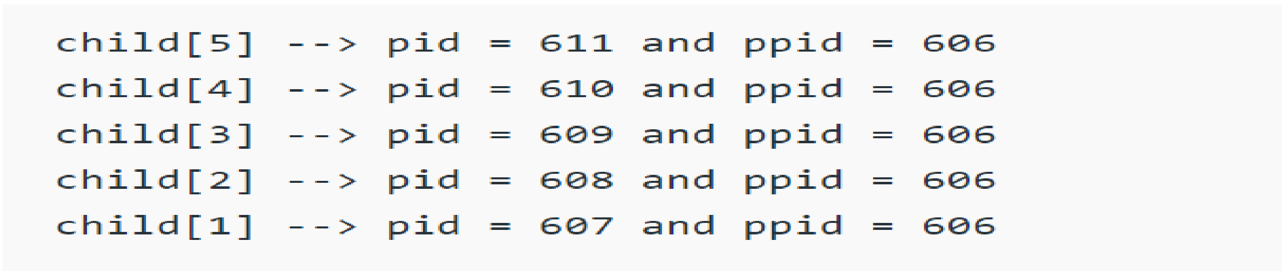
INDEX

Que 1) Write a program to create 5 child processes of a parent process using fork. Print the pids and ppids of all the processes**.**

Code

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>


int main(){

      for(int i=0;i<5;i++){

            if(fork()==0){

                     sleep(5-i);

                      printf("child[%d] --> pid = %d and ppid = %d\n",i+1,getpid(), getppid());

                       return 0;

            }

      }

    sleep(5);

    return 0;

}
```

Result



```
child[5] --> pid = 611 and ppid = 606
child[4] --> pid = 610 and ppid = 606
child[3] --> pid = 609 and ppid = 606
child[2] --> pid = 608 and ppid = 606
child[1] --> pid = 607 and ppid = 606
```

Que 2) Create 2 programs: a client program who sorts 10 numbers using any sort algorithm and a sever program who forks a child process and then executes the client using execvp command.

Code

//Server file

```
#include <stdio.h>

#include <unistd.h>
```

```
int main() {
  char* comd = "insertionSort";
  char* list[] = {" insertionSort ","client.c",NULL};

  int status = execvp(comd, list);

  if (status == -1) {
    printf("Failed\n");
    _exit(1);
  }

  return 0;
}
```

//client file named by **{ client.c }**

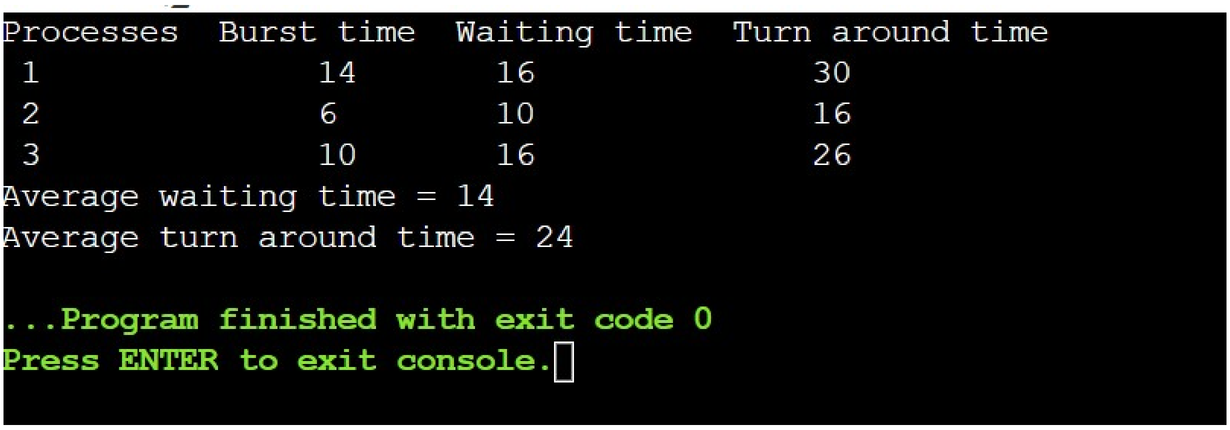
```
#include <math.h>
#include <stdio.h>

void insertionSort(int array[], int size) {
 for (int step = 1; step < size; step++) {
  int key = array[step];
  int j = step - 1;

  while (key < array[j] && j >= 0) {
   array[j + 1] = array[j];
   --j;
  }
  array[j + 1] = key;
 }
}


int main() {
 int data[] = {9, 5, 1, 4, 0, 3,13,17,15,10};
 int size = sizeof(data) / sizeof(data[0]);
 insertionSort(data, size);
 printf("Sorted array in ascending order:\n");
 printArray(data, size);
}
```

Que 3) Implement Round Robin Scheduling algorithm taking the time slice as 2 ms. Implement it using queue data structure: read the process number, its entering time and its CPU burst time and store them in a queue data structure called ready queue. Take at least 4 processes and find the average waiting time and average turnaround time for each process.

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



void findWaitingTime(int processes[], int n,
          int bt[], int wt[], int quantum)
{

   int rem_bt[n];
   for (int i = 0 ; i < n ; i++)
      rem_bt[i] = bt[i];

   int t = 0;
   while (1)
   {
      bool done = true;
      for (int i = 0 ; i < n; i++)
      {
            if (rem_bt[i] > 0)
            {
                done = false;

                if (rem_bt[i] > quantum)
                {

                    t += quantum;

                    rem_bt[i] -= quantum;
                }

                else
                {
                    t = t + rem_bt[i];

                    wt[i] = t - bt[i];

                    rem_bt[i] = 0;
                }
            }
        }

        if (done == true)
        break;
    }
}

void findTurnAroundTime(int processes[], int n,
                        int bt[], int wt[], int tat[])
{
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

void findavgTime(int processes[], int n, int bt[],
                                int quantum)
{
    int wt[n], tat[n], total_wt = 0, total_tat = 0;

    findWaitingTime(processes, n, bt, wt, quantum);

    findTurnAroundTime(processes, n, bt, wt, tat);

    cout << "Processes "<< " 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 << " " << i+1 << "\t\t" << bt[i] <<"\t "
            << wt[i] <<"\t\t " << tat[i] <<endl;
    }

    cout << "Average waiting time = "
        << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
        << (float)total_tat / (float)n;
}

// Driver code
int main(){

    int processes[] = {1, 2, 3};
    int n = sizeof processes / sizeof processes[0];

    int burst_time[] = {10, 5, 8};

    int quantum = 2;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
```

}

Result



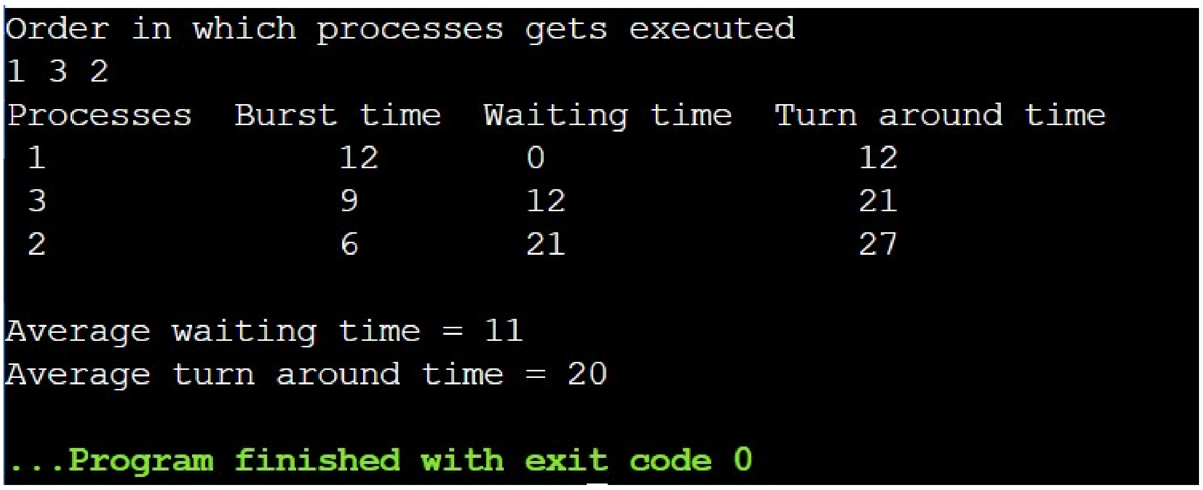
```
Processes  Burst time  Waiting time  Turn around time
 1              14        16              30
 2              6         10              16
 3              10        16              26
Average waiting time = 14
Average turn around time = 24

...Program finished with exit code 0
Press ENTER to exit console.
```

Que 4) Implement the Priority Scheduling algorithm with pre-emption. Implement it using priority queue data structure: read the process number, its entering time, its priority and its CPU burst time.Take at least 6 processes and find the average waiting time and average turnaround time for each process.

```
#include<bits/stdc++.h>

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 "<< " 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)
{
sort(proc, proc + n, comparison);

cout<< "Order in which processes gets executed \n";
for (int i = 0 ; i < n; i++)
cout << proc[i].pid <<" " ;

findavgTime(proc, n);
}

int main()
{
Process proc[] = {{1, 10, 2}, {2, 5, 0}, {3, 8, 1}};
int n = sizeof proc / sizeof proc[0];
priorityScheduling(proc, n);
return 0;
}
```

<u>Result</u>



```
Order in which processes gets executed
1 3 2
Processes  Burst time  Waiting time  Turn around time
 1              12        0                12
 3              9         12               21
 2              6         21               27

Average waiting time = 11
Average turn around time = 20

...Program finished with exit code 0
```

<u>Que 5)</u> Write a program using fork() and pipe() where one child process sends a message from one pipe to another child process and then returns a new message acknowledging the first message through another pipe to first child process.

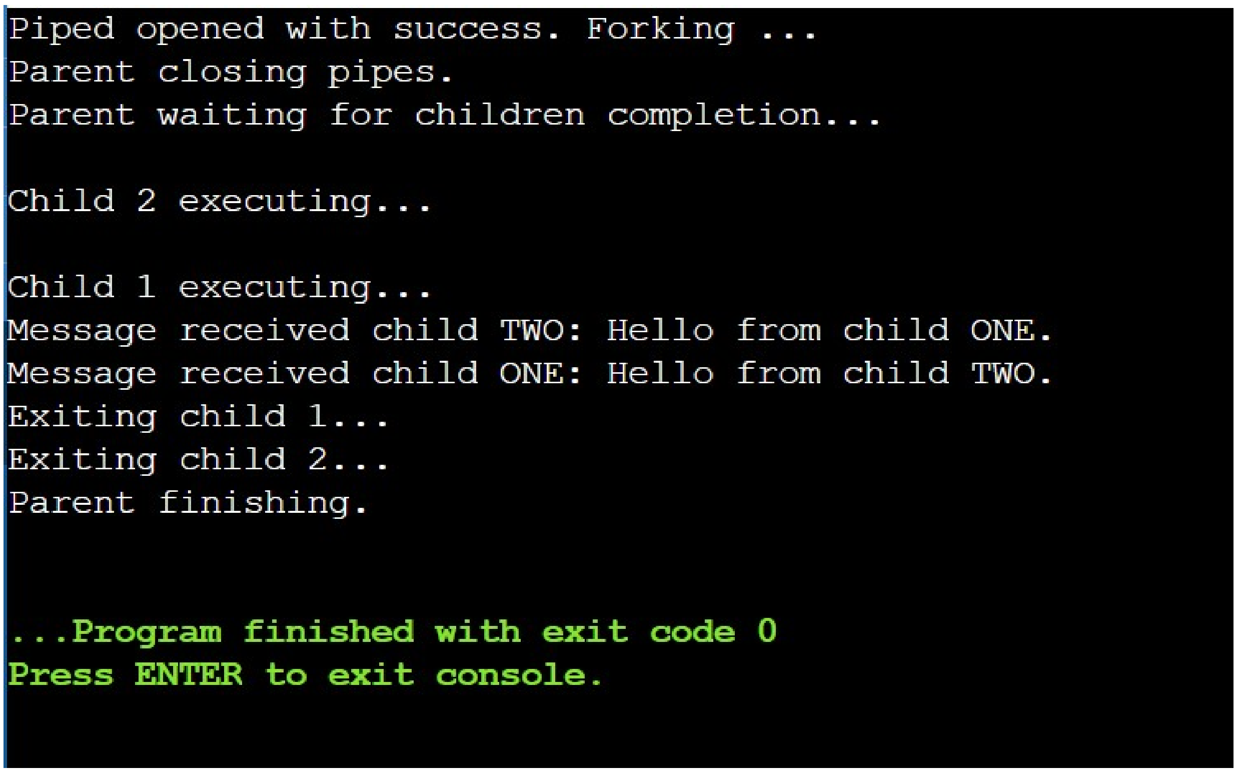
Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

#define BUF_SIZE 256

int main() {
int pfd1[2];
int pfd2[2];

   ssize_t numRead = -1;

   const char* messageOne = "Hello from child ONE.\n";
const char* messageTwo = "Hello from child TWO.\n";

   const unsigned int commLen = strlen(messageOne) + 1;

   char buf[BUF_SIZE];

   if (pipe(pfd1) == -1)
   {
      printf("Error opening pipe 1!\n");
      exit(1);
   }

   if (pipe(pfd2) == -1)
   {
        printf("Error opening pipe 2!\n");
        exit(1);
    }

    printf("Piped opened with success. Forking ...\n");

         //  child  1
switch   (fork())
{
        case -1:
       printf("Error forking child 1!\n");
       exit(1);

        case 0:
     printf("\nChild 1 executing...\n");

            if (close(pfd1[0]) == -1){
   printf("Error closing reading end of pipe 1.\n");
                _exit(1);
            }

            if (close(pfd2[1]) == -1){
          printf("Error closing writing end of pipe 2.\n");
                _exit(1);
            }


            if (write(pfd1[1], messageOne, commLen) != commLen){
                printf("Error writing to pipe 1.\n");
                _exit(1);
            }
            if (close(pfd1[1]) == -1){
             printf("Error closing writing end of pipe
1.\n");
                exit(1);
            }



            numRead = read(pfd2[0], buf, commLen);
            if (numRead == -1){
printf("Error reading from pipe 2.\n");
               _exit(1);
            }

            if (close(pfd2[0]) == -1){
printf("Error closing reding end of pipe 2.\n");
               _exit(1);
            }

            printf("Message received child ONE: %s", buf);
printf("Exiting child 1...\n");
            _exit(0);

        default:
break;
    }

    switch (fork()){
        case -1:            printf("Error
forking child 2!\n");
```

```
            exit(1);        case 0:
printf("\nChild 2 executing...\n");

            if (close(pfd2[0]) == -1){              printf("Error
closing reading end of pipe 2.\n");
                _exit(1);
            }

            if (close(pfd1[1]) == -1){
                printf("Error closing writing end of pipe 1.\n");
                _exit(1);
            }



            if (read(pfd1[0], buf, commLen) == -1){
printf("Error reading from pipe 1.\n");
                _exit(EXIT_FAILURE);
            }

            if (close(pfd1[0]) == -1){              printf("Error
closing reading end of pipe 1.\n");
                _exit(EXIT_FAILURE);
            }



            if (write(pfd2[1], messageTwo, commLen) != commLen){
                printf("Error writing to the pipe.");
                _exit(EXIT_FAILURE);
            }
            if (close(pfd2[1]) == -1){
printf("Error closing writing end of pipe 2.");
                _exit(EXIT_FAILURE);
            }


            printf("Message received child TWO: %s", buf);
            printf("Exiting child 2...\n");
            _exit(EXIT_SUCCESS);


        default:
            break;
    }

    printf("Parent closing pipes.\n");

    if (close(pfd1[0]) == -1){        printf("Error closing
reading end of the pipe.\n");
exit(EXIT_FAILURE);
    }

    if (close(pfd2[1]) == -1){        printf("Error
closing writing end of the pipe.\n");
exit(EXIT_FAILURE);
    }

    if (close(pfd2[0]) == -1){        printf("Error closing
reading end of the pipe.\n");
exit(EXIT_FAILURE);
    }
```

```
   if (close(pfd1[1]) == -1){        printf("Error
closing writing end of the pipe.\n");
exit(EXIT_FAILURE);
   }


   printf("Parent waiting for children
completion...\n");    if (wait(NULL) == -1){
printf("Error waiting.\n");        exit(EXIT_FAILURE);
   }

   if (wait(NULL) == -1){
printf("Error waiting.\n");
exit(EXIT_FAILURE);
   }


   printf("Parent finishing.\n");
exit(EXIT_SUCCESS);
}
```

Result



```
Piped opened with success. Forking ...
Parent closing pipes.
Parent waiting for children completion...

Child 2 executing...

Child 1 executing...
Message received child TWO: Hello from child ONE.
Message received child ONE: Hello from child TWO.
Exiting child 1...
Exiting child 2...
Parent finishing.


...Program finished with exit code 0
Press ENTER to exit console.
```

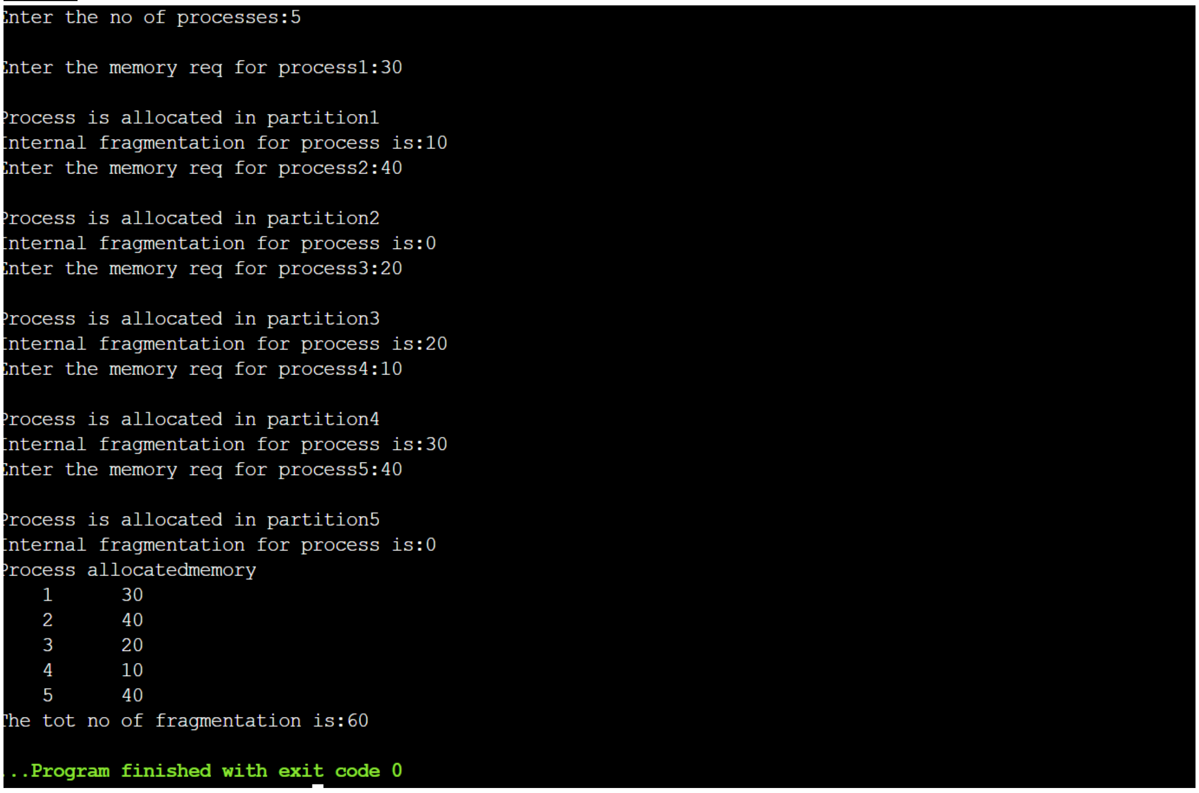
Que 9) Implement the memory management with fixed partitioning. You have to ask the user for total memory, size of OS, size of partitions, number of processes and memory requirement for each process. Allocate the memory accordingly.

<u>Code</u>

```
#include<stdio.h>

int main(){

int m,p,s,p1;

int m1[4],i,f,f1=0,f2=0,fra1,fra2,s1,pos;

printf("Enter the memory size:");

scanf("%d",&m);

printf("Enter the no of partitions:");

scanf("%d",&p);

s=m/p;

printf("Each partn size is:%d",s);

printf("\nEnter the no of processes:");

scanf("%d",&p1);

pos=m;

for(i=0;i<p1;i++)
```

```
{
if(pos<s)
{
printf("\nThere is no further memory for process%d",i+1);
m1[i]=0;
break;
}
else
{
printf("\nEnter the memory req for process%d:",i+1);
scanf("%d",&m1[i]);
if(m1[i]<=s)
{
printf("\nProcess is allocated in partition%d",i+1);
fra1=s-m1[i];
printf("\nInternal fragmentation for process is:%d",fra1);
f1=f1+fra1;
pos=pos-s;
}
else
{
printf("\nProcess not allocated in partition%d",i+1);
s1=m1[i];

while(s1>s)
{
s1=s1-s;
pos=pos-s;
}
pos=pos-s;
fra2=s-s1;
f2=f2+fra2;

printf("\nExternal Fragmentation for this process is:%d",fra2);

}}}

printf("\nProcess\tallocatedmemory");

for(i=0;i<p1;i++)

printf("\n%5d\t%5d",i+1,m1[i]);

f=f1+f2;

printf("\nThe tot no of fragmentation is:%d",f);

return 0;

}
```

Result



```
Enter the no of processes:5

Enter the memory req for process1:30

Process is allocated in partition1
Internal fragmentation for process is:10
Enter the memory req for process2:40

Process is allocated in partition2
Internal fragmentation for process is:0
Enter the memory req for process3:20

Process is allocated in partition3
Internal fragmentation for process is:20
Enter the memory req for process4:10

Process is allocated in partition4
Internal fragmentation for process is:30
Enter the memory req for process5:40

Process is allocated in partition5
Internal fragmentation for process is:0
Process allocatedmemory
    1      30
    2      40
    3      20
    4      10
    5      40
The tot no of fragmentation is:60

...Program finished with exit code 0
```

Que 10) Implement the variable partitioning memory management using worst fit, Best-fit and First fit contiguous memory allocation techniques. You have to ask the user for total memory, size of OS, number of processes and memory requirement for each process. Allocate the memory accordingly using each technique. Give the summary of allocation and remaining holes in the memory.

Code

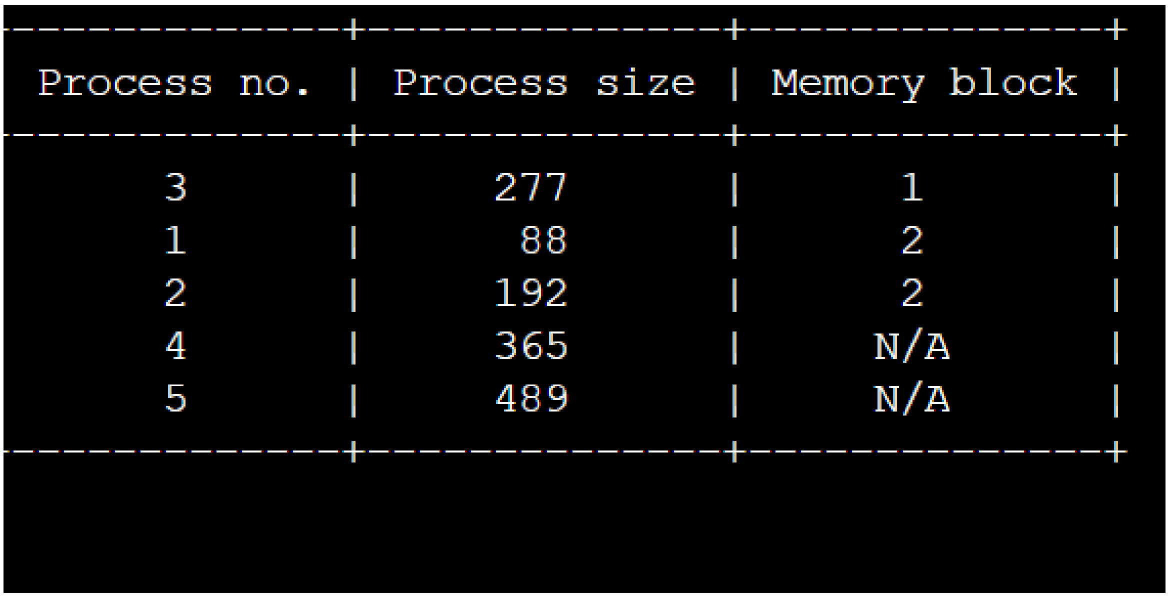
Worst Fit

```
#include <iostream>

#include <queue>

#include <vector>

using namespace std;


class process {

public:

   size_t size;

   pid_t no;

};


class memory {

public:

   size_t size;

   pid_t no;

   queue<process> space_occupied;

   void push(const process p)

   {

      if (p.size <= size) {

         space_occupied.push(p);

         size -= p.size;

      }

   }


   process pop()

   {

      process p;


      if (!space_occupied.empty()) {
            p = space_occupied.front();
            space_occupied.pop();
            size += p.size;
            return p;
        }
    }

    bool empty()
    {
        return space_occupied.empty();
    }
};

vector<memory> worst_fit(vector<memory> memory_blocks,
                         queue<process> processess)
{
    int i = 0, index = 0, max;
    memory na;
    na.no = -10;

    while (!processess.empty()) {
        max = 0;

        for (i = 0; i < memory_blocks.size(); i++) {
            if (memory_blocks.at(i).size >= processess.front().size
                && memory_blocks.at(i).size > max) {
                max = memory_blocks.at(i).size;
                index = i;
            }
        }
        if (max != 0) {
            memory_blocks.at(index).push(processess.front());
        }

        else {
            na.size += processess.front().size;
            na.push(processess.front());
        }

        processess.pop();
    }



    if (!na.space_occupied.empty()) {
        memory_blocks.push_back(na);
    }

    return memory_blocks;
}

void display(vector<memory> memory_blocks)
{
    int i = 0, temp = 0;
    process p;
    cout << "+-------------+--------------+--------------+"
         << endl;
    cout << "| Process no. | Process size | Memory block |"
         << endl;
    cout << "+-------------+--------------+--------------+"
         << endl;
```

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

    while (!memory_blocks.at(i).empty()) {
        p = memory_blocks.at(i).pop();
        temp = to_string(p.no).length();
        cout << "|" << string(7 - temp / 2 - temp % 2, ' ')
             << p.no << string(6 - temp / 2, ' ')
             << "|";

        temp = to_string(p.size).length();
        cout << string(7 - temp / 2 - temp % 2, ' ')
             << p.size
             << string(7 - temp / 2, ' ') << "|";

        temp = to_string(memory_blocks.at(i).no).length();
        cout << string(7 - temp / 2 - temp % 2, ' ');

        // If memory blocks is assigned
        if (memory_blocks.at(i).no != -10) {
            cout << memory_blocks.at(i).no;
        }

        else {
            cout << "N/A";
        }
        cout << string(7 - temp / 2, ' ')
             << "|" << endl;
    }
}
cout << "+-------------+--------------+--------------+"
     << endl;
```

```
}

// Driver Code
int main()
{

    vector<memory> memory_blocks(5);
    vector<memory> worst_fit_blocks;

    queue<process> processess;
    process temp;

    memory_blocks[0].no = 1;
    memory_blocks[0].size = 400;

    memory_blocks[1].no = 2;
    memory_blocks[1].size = 500;

    memory_blocks[2].no = 3;
    memory_blocks[2].size = 300;

    memory_blocks[3].no = 4;
    memory_blocks[3].size = 200;

    memory_blocks[4].no = 5;
    memory_blocks[4].size = 100;

    temp.no = 1;
    temp.size = 88;
    processess.push(temp);
    temp.no = 2;
    temp.size = 192;

    processess.push(temp);
    temp.no = 3;
    temp.size = 277;

    processess.push(temp);
    temp.no = 4;
    temp.size = 365;

    processess.push(temp);
    temp.no = 5;
    temp.size = 489;

    processess.push(temp);

    worst_fit_blocks = worst_fit(memory_blocks,
                          processess);

    display(worst_fit_blocks);
    memory_blocks.clear();
    memory_blocks.shrink_to_fit();
    worst_fit_blocks.clear();
    worst_fit_blocks.shrink_to_fit();
    return 0;
}
```

Result



```
-------------+--------------+--------------+
 Process no. | Process size | Memory block |
-------------+--------------+--------------+
     3       |     277      |      1       |
     1       |      88      |      2       |
     2       |     192      |      2       |
     4       |     365      |     N/A      |
     5       |     489      |     N/A      |
-------------+--------------+--------------+
```

Best Fit

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Process Class
class process {
public:
   // Size & number of process
   size_t size;
   pid_t no;
};

// Memory Class
class memory {
public:
   size_t size;
```

```
// Number of memory & queue of space
// occupied by process
pid_t no;
queue<process> space_occupied;

// Function to push process in a block
void push(const process p)
{
    if (p.size <= size) {
        space_occupied.push(p);
        size -= p.size;
    }
}

// Function to pop and return the
// process from the block
process pop()
{
    process p;

    // If space occupied is empty
    if (!space_occupied.empty()) {
        p = space_occupied.front();
        space_occupied.pop();
        size += p.size;
        return p;
    }
}

// Function to check if block is
```

```
    // completely empty
    bool empty()
    {
        return space_occupied.empty();
    }
};

// Function to get data of processess
// allocated using Best Fit
vector<memory> best_fit(vector<memory> memory_blocks,
                        queue<process> processess)
{
    int i = 0, min, index = 0;
    memory na;
    na.no = -10;

    // Loop till processe is not empty
    while (!processess.empty()) {
        min = 0;

        // Traverse the memory_blocks
        for (i = 0; i < memory_blocks.size(); i++) {
            if (memory_blocks.at(i).size >= processess.front().size && (min == 0 ||
memory_blocks.at(i).size < min)) {
                min = memory_blocks.at(i).size;
                index = i;
            }
        }

        if (min != 0) {
            memory_blocks.at(index).push(processess.front());
        }
        else {
            na.size += processess.front().size;
            na.push(processess.front());
        }

        // Pop the processe
        processess.pop();
    }

    // If space is no occupied then push
    // the current memory na
    if (!na.space_occupied.empty()) {
        memory_blocks.push_back(na);
    }

    // Return the memory_blocks
    return memory_blocks;
}

// Function to display the allocation
// of all processess
void display(vector<memory> memory_blocks)
{
    int i = 0, temp = 0;
    process p;
    cout << "+-------------+--------------+--------------+"
         << endl;
    cout << "| Process no. | Process size | Memory block |"
         << endl;
    cout << "+-------------+--------------+--------------+"
```

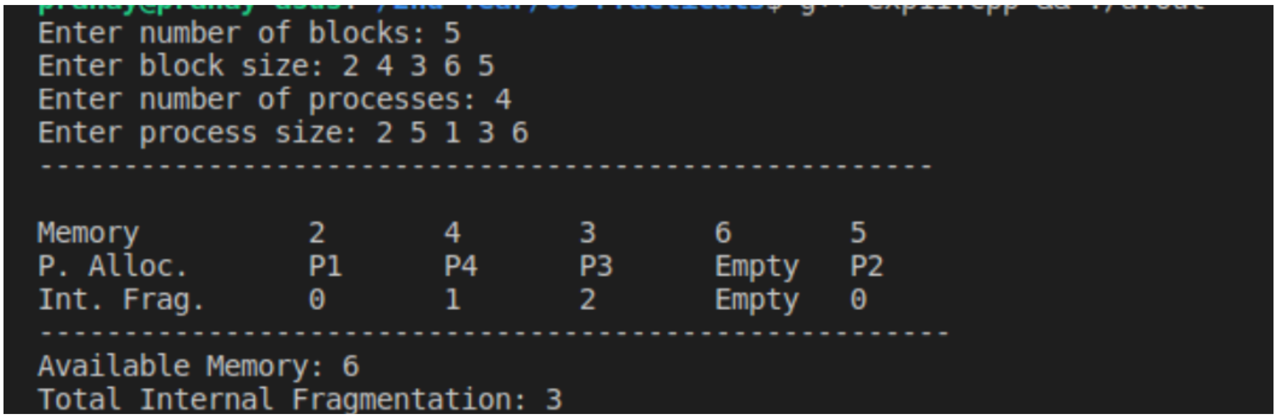
```
     << endl;

  // Traverse memory blocks size
  for (i = 0; i < memory_blocks.size(); i++) {

    // While memory block size is not empty
    while (!memory_blocks.at(i).empty()) {
      p = memory_blocks.at(i).pop();
      temp = to_string(p.no).length();
      cout << "|" << string(7 - temp / 2 - temp % 2, ' ')
         << p.no << string(6 - temp / 2, ' ')
         << "|";

      temp = to_string(p.size).length();
      cout << string(7 - temp / 2 - temp % 2, ' ')
         << p.size
         << string(7 - temp / 2, ' ') << "|";

      temp = to_string(memory_blocks.at(i).no).length();
      cout << string(7 - temp / 2 - temp % 2, ' ');

      // If memory blocks is assigned
      if (memory_blocks.at(i).no != -10) {
        cout << memory_blocks.at(i).no;
      }

      // Else memory blocks is assigned
      else {
        cout << "N/A";
      }
      cout << string(7 - temp / 2, ' ')
```

```
                << "|" << endl;
        }
    }
    cout << "+-------------+--------------+--------------+"
         << endl;
}

// Driver Code
int main()
{
    // Declare memory blocks
    vector<memory> memory_blocks(5);

    // Declare best fit blocks
    vector<memory> best_fit_blocks;

    // Declare queue of all processess
    queue<process> processess;
    process temp;

    // Set sample data
    memory_blocks[0].no = 1;
    memory_blocks[0].size = 400;

    memory_blocks[1].no = 2;
    memory_blocks[1].size = 500;

    memory_blocks[2].no = 3;
    memory_blocks[2].size = 300;

    memory_blocks[3].no = 4;
```

```
memory_blocks[3].size = 200;

memory_blocks[4].no = 5;
memory_blocks[4].size = 100;

temp.no = 1;
temp.size = 88;

// Push the processe to queue
processess.push(temp);
temp.no = 2;
temp.size = 192;

// Push the processe to queue
processess.push(temp);
temp.no = 3;
temp.size = 277;

// Push the processe to queue
processess.push(temp);
temp.no = 4;
temp.size = 365;

// Push the processe to queue
processess.push(temp);
temp.no = 5;
temp.size = 489;

// Push the processe to queue
processess.push(temp);
```

```
    // Get the data
    best_fit_blocks = best_fit(memory_blocks,
                              processess);

    // Display the data
    display(best_fit_blocks);
    memory_blocks.clear();
    memory_blocks.shrink_to_fit();
    best_fit_blocks.clear();
    best_fit_blocks.shrink_to_fit();
    return 0;
}
```

Result



```
Enter number of blocks: 5
Enter block size: 2 4 3 6 5
Enter number of processes: 4
Enter process size: 2 5 1 3 6
-----------------------------------------------------

Memory          2       4       3       6       5
P. Alloc.       P1      P4      P3      Empty   P2
Int. Frag.      0       1       2       Empty   0
------------------------------------------------------
Available Memory: 6
Total Internal Fragmentation: 3
```

First fit

```
// C++ program for the implementation
// of the First Fit algorithm
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Process Class
class process {
```

```
public:
   // Size & number of process
   size_t size;
   pid_t no;
};

// Memory Class
class memory {
public:
   size_t size;

   // Number of memory & queue of space
   // occupied by process
   pid_t no;
   queue<process> space_occupied;

   // Function to push process in a block
   void push(const process p)
   {
      if (p.size <= size) {
         space_occupied.push(p);
         size -= p.size;
      }
   }

   // Function to pop and return the
   // process from the block
   process pop()
   {
      process p;
```

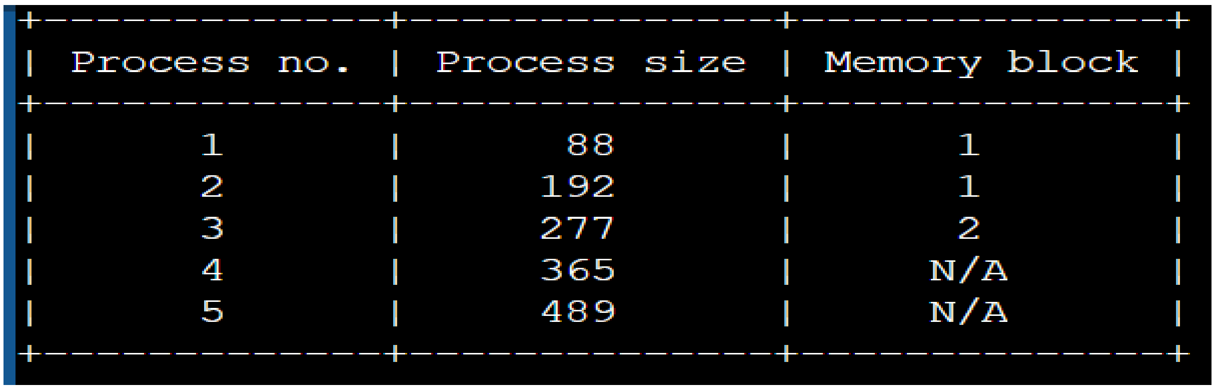
```
        // If space occupied is empty
        if (!space_occupied.empty()) {
            p = space_occupied.front();
            space_occupied.pop();
            size += p.size;
            return p;
        }
    }

    // Function to check if block is
    // completely empty
    bool empty()
    {
        return space_occupied.empty();
    }
};

// Function to get data of processess
// allocated using first fit
vector<memory> first_fit(vector<memory> memory_blocks,
                         queue<process> processess)
{
    int i = 0;
    bool done, done1;
    memory na;
    na.no = -10;
    while (!processess.empty()) {
        done = 0;
        for (i = 0; i < memory_blocks.size(); i++) {
            done1 = 0;
            if (memory_blocks.at(i).size
                >= processess.front().size) {
                memory_blocks.at(i).push(processess.front());
                done = 1;
                done1 = 1;
                break;
            }
        }

        // If process is done
        if (done == 0 && done1 == 0) {
            na.size += processess.front().size;
            na.push(processess.front());
        }

        // pop the process
        processess.pop();
    }
    if (!na.space_occupied.empty())
        memory_blocks.push_back(na);
    return memory_blocks;
}

// Function to display the allocation
// of all processess
void display(vector<memory> memory_blocks)
{
    int i = 0, temp = 0;
    process p;
    cout << "+-------------+--------------+--------------+"
         << endl;
    cout << "| Process no. | Process size | Memory block |"
```

```
         << endl;
    cout << "+-------------+--------------+--------------+"
         << endl;

    // Traverse memory blocks size
    for (i = 0; i < memory_blocks.size(); i++) {

        // While memory block size is not empty
        while (!memory_blocks.at(i).empty()) {
            p = memory_blocks.at(i).pop();
            temp = to_string(p.no).length();
            cout << "|" << string(7 - temp / 2 - temp % 2, ' ')
                 << p.no << string(6 - temp / 2, ' ')
                 << "|";

            temp = to_string(p.size).length();
            cout << string(7 - temp / 2 - temp % 2, ' ')
                 << p.size
                 << string(7 - temp / 2, ' ') << "|";

            temp = to_string(memory_blocks.at(i).no).length();
            cout << string(7 - temp / 2 - temp % 2, ' ');

            // If memory blocks is assigned
            if (memory_blocks.at(i).no != -10) {
                cout << memory_blocks.at(i).no;
            }

            // Else memory blocks is assigned
            else {
                cout << "N/A";
            }
            cout << string(7 - temp / 2, ' ')
                << "|" << endl;
        }
    }
    cout << "+-------------+--------------+--------------+"
        << endl;
}

// Driver Code
int main()
{
    // Declare memory blocks
    vector<memory> memory_blocks(5);

    // Declare first fit blocks
    vector<memory> first_fit_blocks;

    // Declare queue of all processess
    queue<process> processess;
    process temp;

    // Set sample data
    memory_blocks[0].no = 1;
    memory_blocks[0].size = 400;

    memory_blocks[1].no = 2;
    memory_blocks[1].size = 500;

    memory_blocks[2].no = 3;
    memory_blocks[2].size = 300;
```

```
memory_blocks[3].no = 4;

memory_blocks[3].size = 200;


memory_blocks[4].no = 5;

memory_blocks[4].size = 100;


temp.no = 1;

temp.size = 88;


// Push the process

processess.push(temp);


temp.no = 2;

temp.size = 192;


// Push the process

processess.push(temp);


temp.no = 3;

temp.size = 277;


// Push the process

processess.push(temp);


temp.no = 4;

temp.size = 365;


// Push the process

processess.push(temp);
```

```
    temp.no = 5;

    temp.size = 489;


    // Push the process

    processess.push(temp);


    // Get the data

    first_fit_blocks = first_fit(memory_blocks, processess);


    // Display the data

    display(first_fit_blocks);

    memory_blocks.clear();

    memory_blocks.shrink_to_fit();

    first_fit_blocks.clear();

    first_fit_blocks.shrink_to_fit();

    return 0;

}
```

Result



```
+-------------+---------------+---------------+
| Process no. | Process size  | Memory block  |
+-------------+---------------+---------------+
|      1      |      88       |       1       |
|      2      |     192       |       1       |
|      3      |     277       |       2       |
|      4      |     365       |      N/A      |
|      5      |     489       |      N/A      |
+-------------+---------------+---------------+
```

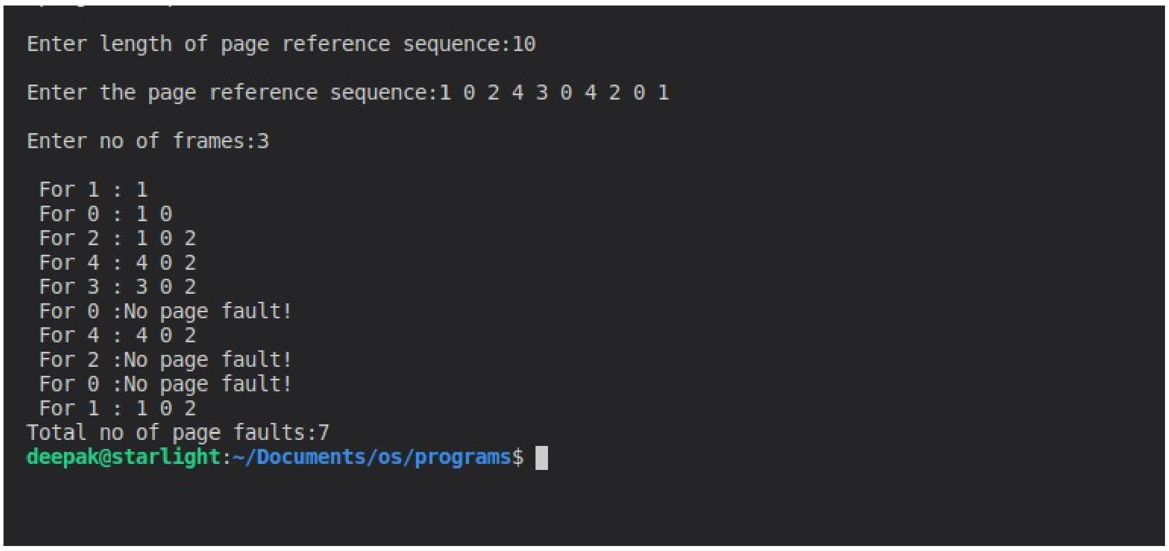
Que 11) Implement all the page replacement techniques (FIFO, OPT, LRU) and display the page faults. Ask the user for number of free frames, reference string and then output the no of page faults for every algorithm.

LRU

```
import java.util.ArrayList;

public class LRU {

    // Driver method
```

```
public static void main(String[] args) {
   int capacity = 4;
   int arr[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

   // To represent set of current pages.We use
   // an Arraylist
   ArrayList<Integer> s=new ArrayList<>(capacity);
   int count=0;
   int page_faults=0;
   for(int i:arr)
   {
      // Insert it into set if not present
      // already which represents page fault
      if(!s.contains(i))
      {

      // Check if the set can hold equal pages
      if(s.size()==capacity)
      {
         s.remove(0);
         s.add(capacity-1,i);
      }
      else
         s.add(count,i);
         // Increment page faults
         page_faults++;
         ++count;

      }
      else
      {
            // Remove the indexes page
            s.remove((Object)i);
            // insert the current page
            s.add(s.size(),i);
          }

        }
        System.out.println(page_faults);
    }
}
```

Result



```
Enter length of page reference sequence:10

Enter the page reference sequence:1 0 2 4 3 0 4 2 0 1

Enter no of frames:3

 For 1 : 1
 For 0 : 1 0
 For 2 : 1 0 2
 For 4 : 4 0 2
 For 3 : 3 0 2
 For 0 :No page fault!
 For 4 : 4 0 2
 For 2 :No page fault!
 For 0 :No page fault!
 For 1 : 1 0 2
Total no of page faults:7
deepak@starlight:~/Documents/os/programs$
```

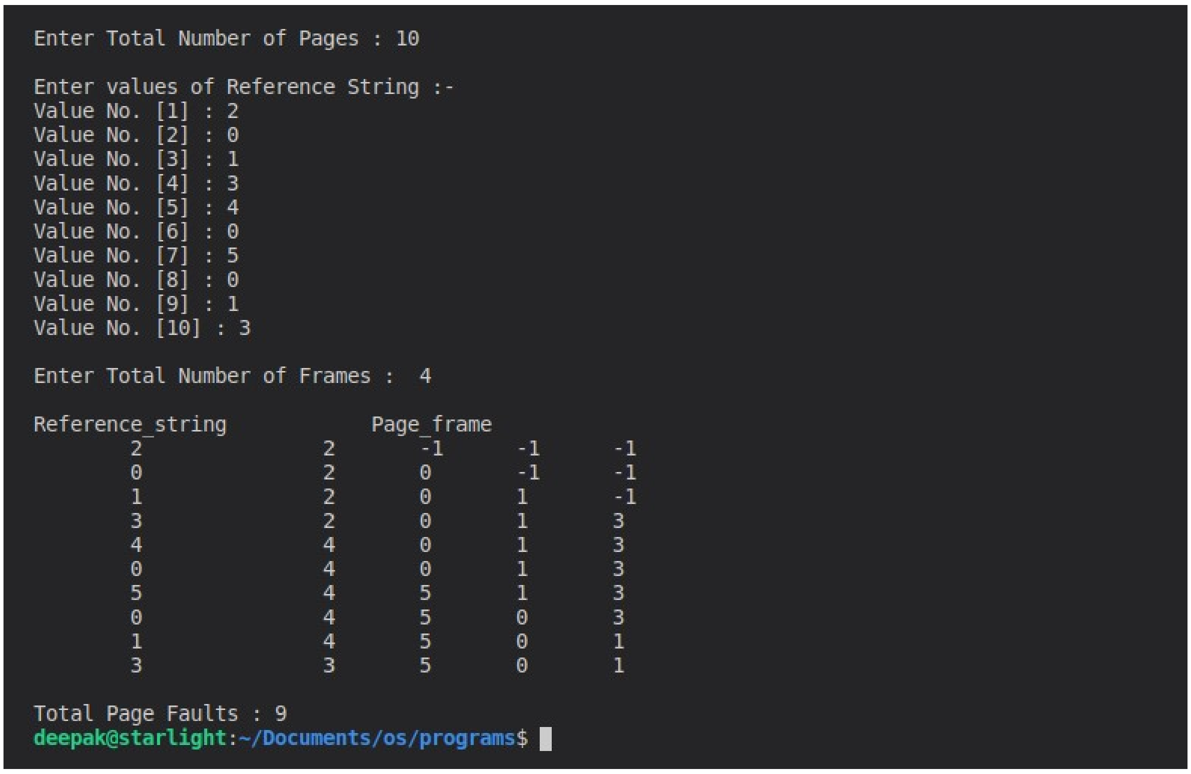
(FIFO)

```
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Queue;



class Test{
```

```
static int pageFaults(int pages[], int n, int capacity){

  HashSet<Integer> s = new HashSet<>(capacity);
  Queue<Integer> indexes = new LinkedList<>() ;
  int page_faults = 0;
  for (int i=0; i<n; i++)
  {

    if (s.size() < capacity)
    {

      if (!s.contains(pages[i]))
      {
        s.add(pages[i]);
        page_faults++;
        indexes.add(pages[i]);
      }
    }

    // If the set is full then need to perform FIFO
    // i.e. remove the first page of the queue from
    // set and queue both and insert the current page
    else
    {
      // Check if current page is not already
      // present in the set
      if (!s.contains(pages[i]))
      {
        //Pop the first page from the queue
        int val = indexes.peek();
```

```
                    indexes.poll();

                    // Remove the indexes page
                    s.remove(val);

                    // insert the current page
                    s.add(pages[i]);

                    // push the current page into
                    // the queue
                    indexes.add(pages[i]);

                    // Increment page faults
                    page_faults++;
                }
            }
        }

        return page_faults;
    }

    // Driver method
    public static void main(String args[])
    {
        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                        2, 3, 0, 3, 2};

        int capacity = 4;
        System.out.println(pageFaults(pages, pages.length, capacity));
    }
}
```

Result



```
Enter Total Number of Pages : 10

Enter values of Reference String :-
Value No. [1] : 2
Value No. [2] : 0
Value No. [3] : 1
Value No. [4] : 3
Value No. [5] : 4
Value No. [6] : 0
Value No. [7] : 5
Value No. [8] : 0
Value No. [9] : 1
Value No. [10] : 3

Enter Total Number of Frames :  4

Reference_string            Page_frame
        2                2       -1      -1      -1
        0                2       0       -1      -1
        1                2       0       1       -1
        3                2       0       1       3
        4                4       0       1       3
        0                4       0       1       3
        5                4       5       1       3
        0                4       5       0       3
        1                4       5       0       1
        3                3       5       0       1

Total Page Faults : 9
deepak@starlight:~/Documents/os/programs$ 
```

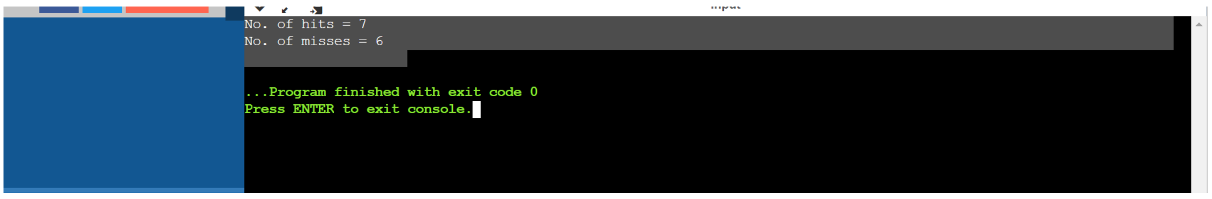
Optimal

```
#include <bits/stdc++.h>

using namespace std;


// Function to check whether a page exists

// in a frame or not

bool search(int key, vector<int>& fr)

{

  for (int i = 0; i < fr.size(); i++)

    if (fr[i] == key)

      return true;

  return false;

}


// Function to find the frame that will not be used

// recently in future after given index in pg[0..pn-1]
```

```
int predict(int pg[], vector<int>& fr, int pn, int index)
{
  // Store the index of pages which are going
  // to be used recently in future
  int res = -1, farthest = index;
  for (int i = 0; i < fr.size(); i++) {
    int j;
    for (j = index; j < pn; j++) {
      if (fr[i] == pg[j]) {
        if (j > farthest) {
          farthest = j;
          res = i;
        }
        break;
      }
    }

    // If a page is never referenced in future,
    // return it.
    if (j == pn)
      return i;
  }

  // If all of the frames were not in future,
  // return any of them, we return 0. Otherwise
  // we return res.
  return (res == -1) ? 0 : res;
}

void optimalPage(int pg[], int pn, int fn)
{
    // Create an array for given number of
    // frames and initialize it as empty.
    vector<int> fr;

    // Traverse through page reference array
    // and check for miss and hit.
    int hit = 0;
    for (int i = 0; i < pn; i++) {

        // Page found in a frame : HIT
        if (search(pg[i], fr)) {
            hit++;
            continue;
        }

        // Page not found in a frame : MISS

        // If there is space available in frames.
        if (fr.size() < fn)
            fr.push_back(pg[i]);

        // Find the page to be replaced.
        else {
            int j = predict(pg, fr, pn, i + 1);
            fr[j] = pg[i];
        }
    }
    cout << "No. of hits = " << hit << endl;
    cout << "No. of misses = " << pn - hit << endl;
}
```

```
// Driver Function

int main()

{

   int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };

   int pn = sizeof(pg) / sizeof(pg[0]);

   int fn = 4;

   optimalPage(pg, pn, fn);

   return 0;

}
```

<u>Result</u>



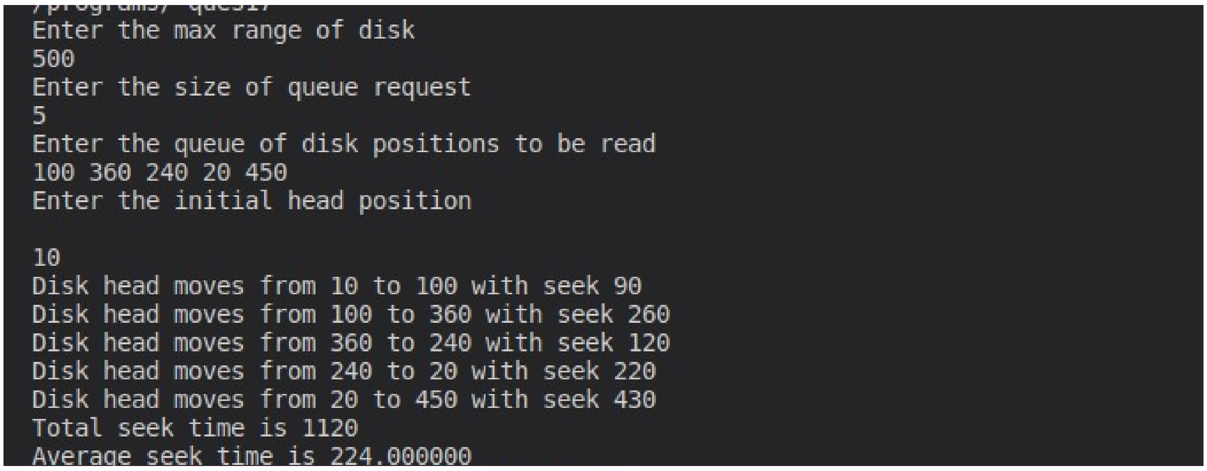
```
No. of hits = 7
No. of misses = 6


...Program finished with exit code 0
Press ENTER to exit console.
```

Que 12) Implement the disk scheduling algorithms: FCFS, SCAN, C-SCAN.

FCFS disk scheduling algorithm.

```
#include <stdio.h> #include <stdlib.h> int main()

{

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

float avg;

printf("Enter the max range of disk\n");

scanf("%d", &max);

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

 scanf("%d", &n);

   printf("Enter the queue of disk positions to be read\n");

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

   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);

avg = seek / (float)n;

printf("Average seek time is %f\n", avg);

return 0;

}
```

<u>Result</u>



```
Enter the max range of disk
500
Enter the size of queue request
5
Enter the queue of disk positions to be read
100 360 240 20 450
Enter the initial head position

10
Disk head moves from 10 to 100 with seek 90
Disk head moves from 100 to 360 with seek 260
Disk head moves from 360 to 240 with seek 120
Disk head moves from 240 to 20 with seek 220
Disk head moves from 20 to 450 with seek 430
Total seek time is 1120
Average seek time is 224.000000
```

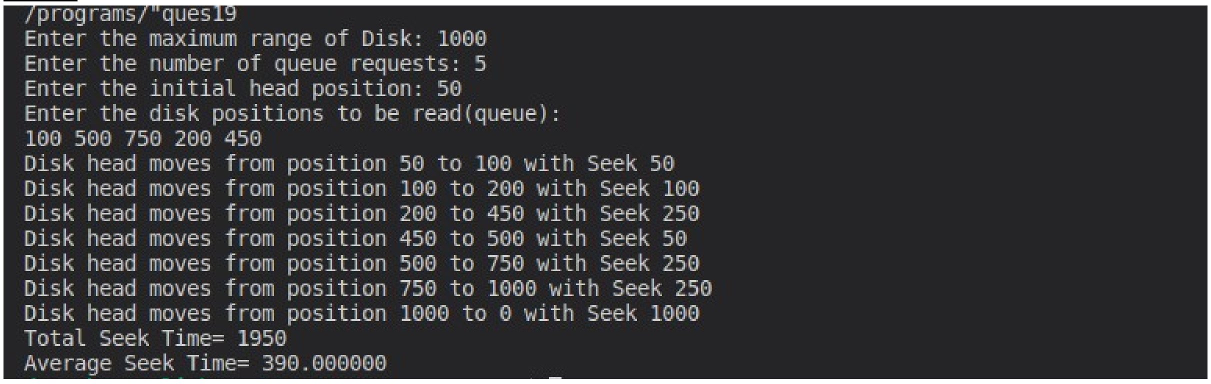
C Scan Disk Scheduling Algorithms

```
#include <stdio.h> int absoluteValue(int);


int main()

{     int queue[25], n, headposition, i, j, k, seek = 0, maxrange, difference, temp, queue1[20], que
ue2[20], temp1 = 0, temp2 = 0;

float averageSeekTime;

     printf("Enter the maximum range of Disk: ");     scanf("%d", &maxrange);

     printf("Enter the number of queue requests: ");     scanf("%d", &n);
```

```
 printf("Enter the initial head position: ");     scanf("%d", &headposition);
 printf("Enter the disk positions to be read(queue): \n");     for (i = 1; i <= n; i++)
{        scanf("%d", &temp);
    if (temp > headposition)
   {            queue1[temp1] = temp;            temp1++;
   }        else        {            queue2[temp2] = temp;            temp2++;
   }
}     for (i = 0; i < temp1 - 1; i++)
{        for (j = i + 1; j < temp1; j++)
   {
       if (queue1[i] > queue1[j])
       {
          temp = queue1[i];                queue1[i] = queue1[j];                queue1[j] = temp;
       }
   }
}
for (i = 0; i < temp2 - 1; i++)
{        for (j = i + 1; j < temp2; j++)
   {            if (queue2[i] > queue2[j])
       {                temp = queue2[i];
                              queue2[i] = queue2[j];                queue2[j] = temp;
       }
   }
}     for (i = 1, j = 0; j < temp1; i++, j++)
{        queue[i] = queue1[j];
}     queue[i] = maxrange;
 queue[i + 1] = 0;
 for (i = temp1 + 3, j = 0; j < temp2; i++, j++)
{        queue[i] = queue2[j];
}     queue[0] = headposition;
 for (j = 0; j <= n + 1; j++)
```

```
   {        difference = absoluteValue(queue[j + 1] - queue[j]);

        seek = seek + difference;

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

   }

   averageSeekTime = seek / (float)n;


   printf("Total Seek Time= %d\n", seek);     printf("Average Seek Time= %f\n", averageSeekTime); }
int absoluteValue(int x)

{

   if (x > 0)

   {        return x;

   }    else     {        return x * -1;

   }

}
```

Result



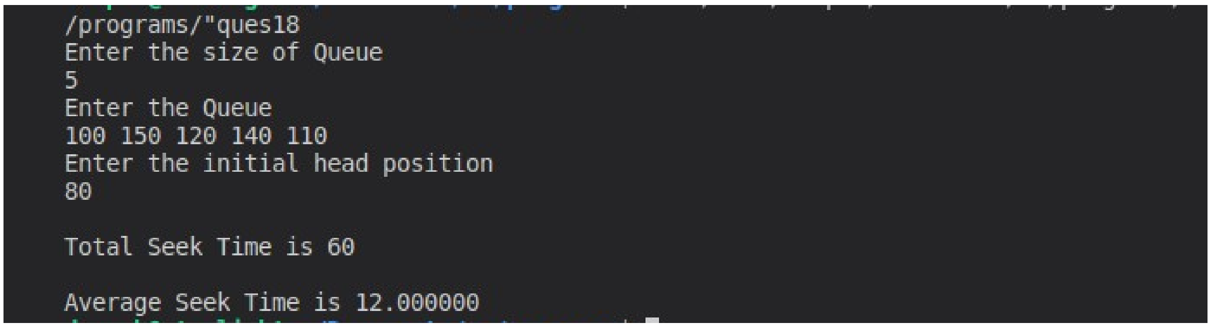
```
/programs/"ques19
Enter the maximum range of Disk: 1000
Enter the number of queue requests: 5
Enter the initial head position: 50
Enter the disk positions to be read(queue):
100 500 750 200 450
Disk head moves from position 50 to 100 with Seek 50
Disk head moves from position 100 to 200 with Seek 100
Disk head moves from position 200 to 450 with Seek 250
Disk head moves from position 450 to 500 with Seek 50
Disk head moves from position 500 to 750 with Seek 250
Disk head moves from position 750 to 1000 with Seek 250
Disk head moves from position 1000 to 0 with Seek 1000
Total Seek Time= 1950
Average Seek Time= 390.000000
```

--

Scan Disk Algorithms

```
#include <stdio.h>

#include <math.h>

#include <stdlib.h>


int main()

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

 printf("Enter the size of Queue\n");
 scanf("%d", &n);

 printf("Enter the Queue\n");


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

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

   }

   printf("Enter the initial head position\n");     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\n", seek);     avg = seek / (float)n;     printf("\nAverage Seek
Time is %f\n", avg);     return 0;

}
```

<u>Result</u>



```
/programs/"ques18
Enter the size of Queue
5
Enter the Queue
100 150 120 140 110
Enter the initial head position
80

Total Seek Time is 60

Average Seek Time is 12.000000
```