Ex.No:1	Implementation of Process and I/O System calls
Date: 08/02/2023	

Aim:

To develop a java program for implementation of process and I/O system calls.

Algorithm:

Step 1: Start

Step 2: Import the java package

Step 3: Create a class called experiment 1 and object for classes.

Step 4: Call the function

Step 5: In first child, check whether the number is odd or even by n % 2 == 0.

Step 6: If n % 2 == 0 then even or odd

Step 7: In second child, get the string and reverse it.

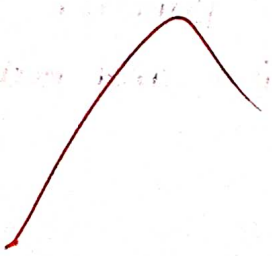
Step 8: Go to the main function

Step 9: Print the function and return the values

Step 10: Stop

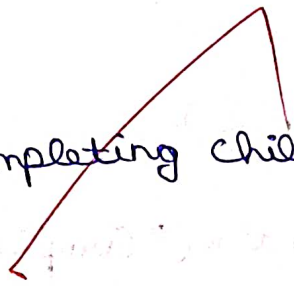
Program:

```
Package experiment1;
public class Experiment1 {
    public static void main (String[] args) throws Exception {
        Thread t = Thread.currentThread();
        System.out.println("Inside Main thread" + t.getId());
        Child1 c1 = new Child1();
        c1.join();
        Child2 c2 = new Child2();
        c2.join();
        System.out.println("Completing main thread");
       }
      }
package experiment1;
import java.util.Scanner;
public class Child1 extends Thread {
  Child1()
  {
   super("Child1");
   start();
  }
  public void run()
  {
   int n;
   try
   {
   System.out.println("Inside Child1" + this.getId());
   System.out.println("Enter the number:");
   Scanner sc = new Scanner(System.in);
```



```
        n = sc.nextInt();
        if (n%2==0)
        {
         System.out.println(n+" is an even number");
         }
        else
        {
         System.out.println(n+" is a odd number");
         }
         } catch (Exception e)
        { }
        System.out.println("Completing child1 Thread");
      } }
package experiment1;
import java.util.Scanner;
public class child2 extends Thread {
   child2()
   {
    super("child2");
    start();
   }
 public void run()
{
   try
   {
    int i;
    System.out.println("Inside Child2"+this.getId());
    String str;
    System.out.println("Enter the string:");
    Scanner sc = new Scanner(System.in);
    str = sc.nextLine();
```

```
System.out.println("Reversed string:");
for (i= (str.length()-1); i>=0; i--)
    {
     System.out.print (str.charAt(i));
    }
 System.out.println(" ");
}catch (Exception e)
{}
System.out.println("Completing Child1 Thread");
}}
```



Output:

```
Inside the main thread
Main thread ID: 1
Inside child 1
Enter the number: 5
The number is odd
Completing child 1
Inside child 2
Enter the string: Hello
The reversed string: olleH
Completing child 2
Completing main thread
```

Result:

Thus the java program for implementation of Process and I/O system calls has been implemented and verified successfully.

o/p

Criteria	Marks
Preparation	15 /20
Observation	20 /25
Interpretation of Result	20 /20
Viva	8 /10
Total	63 /75
Faculty Signature with Date	

Ex.No:2	Implementation of CPU Scheduling Algorithms
Date: 15/02/2023	

Aim:

To develop a java program for implementation of CPU scheduling algorithm.

Algorithm:

Step 1 : Start

Step 2: Get the number of process

Step 3 : Get the burst time and priority of each process

Step 4 : Sort the processes based on the priority

Step 5 : Calculation of Turn Around Time and waiting Time

a) tot_TAT = tot_TAT + pre_TAT

b) avg_TAT = tot_TAT / num_of_proc

c) tot_WT = tot_WT + pre_WT + PRE_BT

d) avg_WT = tot_WT / num_of_proc

Step 6: Display the result

Step 7: Stop

Program:

```
import java.util.*;
class process
{
 int pid, bt, priority;
 process(int pid, int bt, int priority)
 {
   this.pid = pid;
   this.bt = bt;
   this.priority = priority;
 }
public int prior()
{
 return priority;
 }
}
public class GFG
{ public 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];
 }
 public void findTurnAroundTime(process proc[], int n, int
                                wt[], int tat[])
 {
   for(int i=1; i<n; i++)
      tat[i] = proc[i].bt + wt[i];
 }
```

```
public void findavgTime (process proc [], int n)
{ int wt[] = new int [n], tat[] = new int [n], total_wt = 0,
total_tat = 0;
    findWaitingTime (proc, n, wt);
    findTurnAroundTime (proc, n, wt, tat);
    System.out.print ("\n Process burst time waiting time Time
around time \n");
    for (int i = 1; i < n ; i++)
    { total_wt = total_wt + wt[i];
      total_tat = total_tat + tat[i];
      System.out.println (" " + proc[i].pid + "\t\t" + proc[i].bt +
"\t" + wt[i] + "\t\t" + tat[i] + "\n");
    }
    System.out.print ("\n Average waiting time = " + (float) total_
wt / (float) n);
    System.out.print ("\n Average turnaround time = " + (float)
total_tat / (float) n);
}
public void priorityScheduling (process proc[], int n)
{ Arrays.sort(proc, new Comparator < process > ( )
  { @override
    public int compare (process a, process b)
    { return b.prior() - a.prior();
    } } );
  System.out.print ("Order in which processes gets executed
    \n");
```

```
for (int i=1; i<n; i++)
  System.out.println(proc[i].pid+" ");
findavgTime(proc, n); }
public static void main(String[]
  args) {
    GFG ob = new GFG();
    int n=5;
    process proc[] = new process[n];
    proc[0] = new process(4, 0, 2);
    proc[1] = new process(2, 1, 4);
    proc[2] = new process(3, 2, 6);
    proc[3] = new process(5, 3, 10);
    proc[4] = new process(1, 4, 8);
ob.priorityScheduling(proc, n);
  }
}
```

Output: process order in which gets executed 4,5,3,2,1

Processes	Burst time	Waiting time	Turn around time
1	4	11	15
2	2	9	11
3	3	6	9
4	5	0	5
5	1	4	5

Average waiting time = 6

Average turn around time = 9

Result:

Thus the java program for implementation of CPU scheduling algorithm has been implemented and verified successfully.

PS - o/p 4d

Criteria	Marks
Preparation	15 /20
Observation	20 /25
Interpretation of Result	10 /20
Viva	8 /10
Total	53 /75
Faculty Signature with Date	29/8

Ex.No:3	Implementation of Classical Synchronization problems using semaphores
Date: 15/03/2023	

Aim:

To develop a java program for implementation of classical synchronization problems using semaphores.

Algorithm:

Step 1: Start

Step 2: Each philosopher is represented as a seperate thread and each chopstick is represented as a seperate semaphore.

Step 3: Each philosopher initially attems to acquire the chopstick to their left & then to their right.

Step 4: If the chopstick to the ~~right~~ left is already in use the philosopher waits until it becomes available.

Step 5: If the chopstick to the right is already in use, the philosopher release the chopstick to the left & tries again.

Step 6: Once a philosopher has acquired both chopsticks, they eat for a fixed amount of time.

Step 7: After eating, the philosopher releases both Chopsticks.

Step 8: Repeat from step 2 until every philosopher eats.

Step 9: Stop.

Program:

```
import java.util.concurrent.Semaphore;
import java.util.concurrent.ThreadLocalRandom;
public class Main {
static int philosopher =5;
static philosopher philosophers[] = new philosopher
                                    [philosopher];
static chopstick chopsticks[] = new chopstick [philosopher];
static class chopstick {
public semaphore mutex = new Semaphore(1);
void grab() {
try{
mutex.acquire();
}
catch(Exception e) {
e.printStackTrace(System.out);
}
}
void release(){
mutex.release();
}
boolean is free(){
return mutex.availablePermits() >0;
}
}
Static class philosopher extends Thread {
public int number;
public chopstick leftchopstick;
```

```
public chopstick rightchopstick;
philosopher(int num, chopstick left, chopstick right){
number = num;
leftchopstick = left;
rightchopstick = right;
}
public void run(){
while(true){
leftchopstick.grab();
System.out.println("philosopher" + (number+1) + "grabs
    left chopstick.");
rightchopstick.grab();
System.out.println("philosopher" + (number+1) + "grabs
    right chopstick");
eat();
leftchopstick.release();
System.out.println("philosopher" + (number+1) + "
    release left chopstick");
rightchopstick.release();
System.out.println("philosopher" + (number+1) + "
    release right chopstick");
}}
void eat(){
try{
int sleepTime = ThreadLocalRandom.current().next
```

```
Int(0, 1000);
System.out.println("philosopher" + (number+1) + " eats
for" + sleepTime);
Thread.sleep(sleepTime);
}
catch (Exception e) {
e.printStackTrace(System.out);
}
}
}
public static void main (String argv[]) {
for (int i=0; i< philosopher; i++){
chopsticks[i] = new chopstick();
}
for (int i=0; i< philosopher ; i++){
philosophers[i] = new philosopher(i, chopsticks[i],
    chopsticks[(i+1)%philosopher]);
philosophers[i].start();

}
while (true) {
try {
Thread.sleep(1000);
boolean deadlock = true;
for (chopstick f : chopsticks){
if (f.isfree()){
deadlock = false;
break;
}
```

```
}
if (deadlock) {
Thread.sleep(1000);
System.out.println("Everyone Eats");
break;
}
}
catch(Exception e){
e.printStackTrace(System.out);
}
}
System.out.println("Exit the program!");
System.exit(0);
}
}
```

Output:

Philosopher 2 grabs left Chopstick
Philosopher 5 grabs left Chopstick
Philosopher 3 grabs left Chopstick
Philosopher 4 grabs left Chopstick
Philosopher 1 grabs left Chopstick
Everyone Eats
Exit the Program!

Result:

Thus the java program for implementation of Classical Synchronization problems using semaphores has been implemented and verified successfully.

Criteria	Marks
Preparation	/20
Observation	/25
Interpretation of Result	/20
Viva	/10
Total	/75
Faculty Signature with Date	

Ex.No:4	Implementation of Memory Allocation Strategies
Date: 29/3/2023	

Aim:

To develop a java program for the implementation of memory allocation strategies.

Algorithm:

Step 1: Start

Step 2: The first fit algorithm allocates the first available memory block that is enough to accomodate the request.

Step 3: Start with 1st block of memory and check if it's large enough to accomodates the request - if it is correct, then return address of allocated memory block

Step 4: If block is not found, return not allocated.

Step 5: In best fit, calculate the difference between the size of the block and size of request. If difference is smaller, then fit the best fit block to the current block.

Step 6: In worst fit, the difference is calculated and if it is larger then update the worst fit block to the current block.

Step 7: Stop

Program:

```
import java.util.*;
public class BtW{
  static void firstfit(int blocksize[], int m, int
{                                    processSize[], int n)
int allocation[] = new int[n];
for(int i=0; i<allocation.length; i++)
{allocation[i] = -1;
}
for(int i=0; i<n; i++)
 {for (int j=0; j<m; j++)
  {if (blockSize[j] >= processsize[i])
   {allocation[i] = i;
    blockSize[j] = processSize[i];
    break;
   }
  }
 }
 System.out.println("\n ProcessNo.\t Process size \t
 for (int i=0; i<n; i++)                    Block No");
{System.out.print(" "+(i+1)+ "\t\t" + processSize[i] +"\t\t");
 if (allocation[i] != -1)
 {System.out.print (allocation[i]+1);
 }
else
 {
  System.out.print ("Not allocated");
 }
 System.out.println();
```

```
}}
Static void bestfit (int blocksize[], int m, int processSize[],
                                                    int n)
{int allocation[] = new int[n];
 for (int i=0; i<allocation.length; i++)
 {allocation[i] = -1;
 }
 for (int i=0; i<n; i++)
 {int bestind = -1;
  for (int j=0; j<m; j++)
  {if (blockSize[j] >= processSize[i])
   {if (bestind == -1)
       bestind = j;
    else if (blockSize[bestind] > blockSize[j])
       bestind = j;
   }
  }
  if (bestind != -1)
  {allocation[i] = bestind;
   blocksize[bestind] = processSize[i];
  }
 }
System.out.println("\nProcessNo \t ProcessSize \t BlockNo");
 for (int i=0; i<n; i++)
  {System.out.println(" "+(i+1)+ "\t\t"+ processSize[i]+
 if(allocation[i]!=-1)                              "\t\t");
  {
   System.out.println(allocation[i]+1);
  }
 else
  {
```

```
            System.out.println("Not allocated");
        }
        System.out.println();
    }
}
Static void worstfit(int blockSize[], int m, int processSize[],
                                                    int n)
{
int allocation[] = new int[n];
for(int i=0; i< allocation.length; i++)
 {
 allocation[i] = -1;
 }
 for(int i=0; i<n; i++)
  {
   int worstind =-1;
   for(int j=0; j<m; j++)
   {if(blocksize[j] >= processSize[i])
     {if(worstind ==-1)
        worstind=j;
      else if(blockSize[worstind] < blocksize[j])
        worstind = j;
     }
   }
  if(worstind != -1)
   {
    allocation[i] = worstind;
    blocksize[worstint] = processSize[i];
   }
  }
System.out.println("\n ProcessNo \t ProcessSize \t BlockNo");
for(int i= 0; i<n; i++)
 {
  System.out.print(" "+(i+1)+"\t\t" + processSize[i] + "\t\t";
 if(allocation[i] != -1)
{
```

```
            System.out.println(allocation[i]+1);
            }
        else
            {
            System.out.println("Not allocated");
            }
        System.out.println();
        }
    }
public static void main(String[] args)
    {
    int blockSize[] = {350, 75, 150, 175, 300};
    int Pro
    int blockSize[] = {200, 400, 600, 500, 300, 250};
    int processSize[] = {357, 210, 468, 491};
    int m = blockSize.length;
    int n = processSize.length;
    System.out.println("First Fit");
     first.fit(blockSize, m, processSize, n);
    System.out.println("Best Fit");
     bestfit(blockSize, m, processSize, n);
    System.out.println("Worst fit");
     worstfit(blocksize, m, processSize, n);
}
```

Output:

First fit

processno	process size	blockNo
1	357	2
2	210	3
3	468	4
4	491	Not allocated

357 → 400
210 → 600
468 → 500
491 → NA

Best fit

processno	process size	blockNo
1	357	2
2	210	6
3	468	4
4	491	3

375 → 400 → b2
210 → 250 → b6
468 → 500 → b4
491 → 600 → b3

Worst fit

processno	process size	blockNo
1	357	3
2	210	4
3	468	Not allocated
4	491	Not allocated

357 → 600
210 → 500
468 → NA
491 → NA

Result:

Thus the java program for the implementation of memory allocation strategies is executed and it's is output is verified successfully.

Criteria	Marks
Preparation	20 /20
Observation	25 /25
Interpretation of Result	15 /20
Viva	8 /10
Total	68 /75
Faculty Signature with Date	12/24

Ex.No:5	Implementation of Page Replacement Algorithms
Date: 5/4/2023	

Aim:

To develop a java program for the implementation of page replacement algorithm (FIFO, LRU, optimal)

Algorithm:

Step 1: Start

Step 2: In FIFO, Create a queue data structure to hold the pages in memory. When a page is referenced check if it already in memory.

Step 3: If the pages in memory, do nothing. If the page is not in memory, add it to the end of the queue. If queue is full, remove page at front of queue and add new page to the end.

Step 4: In LRU, Create a hash table data structure to hold the pages in memory and their access time.

Step 5: Whenever a page is referenced, update its access time in the hash table. Whenever a page needs to be replaced, Find the page in the hash table with the smallest access time and replace it with new page.

Step 6: In optimal, look ahead in the page reference string and find the page that will not used for longest time in future and replace it.

Step 7: Stop

Program:

```
import java.util.*;
public class pagereplacement
{
public static void fifo(int[] pages, int frames)
{
Queue<Integer> queue = new LinkedList<>();
int PageFaults = 0;
For(int i=0; i<pages.length; i++)
{
if (queue.size() < Frames)
{if (!queue.contains (Pages[i]))
{
queue.add(pages[i]);
pagefaults ++;
}
}
else
{if(!queue.contains(pages[i]))
{queue.remove();
queue.add(pages[i]);
pageFaults ++;
}
}
System.out.println("FIFO Page replacement Algorithm:");
System.out.println("PageFaults: " + pagefaults); }
public static void Lru(int[]pages, int frames)
{ Set<Integer> set = new LinkedHashset<>();
Map<Integer, Integer>map = new Hashmap<>();
int Pagefaults = 0;
```

```
For(int i=0; i<pages.length; i++)
{
int page = pages[i];
if (Set size () < frames)
{ if (!set.contains (page))
 {set.add (page);
  pageFaults++;
 }
 map.put (page, i);
 }
else
 {if (!set.contains (page))
 {int Lru = Integer.Max_value, val = 0;
  For (int p ; oset)
  {if (map.get (p) < Lru)
   {Lru = map.get (p);
    val = p;
   }
  }
  Set.remove (val);
  Set.add (page);
  pageFaults++;
 }
 map.put (page, i);
 }
}
System.out.println ("LRU Page replacement
                                Algorithm");
System.out.println ("Page faults: " + pagefaults);
}
```

```
public static void optimal(int[] pages, int frames)
{ Set<Integer> set = new LinkedHashSet<>();
  Map<Integer, Integer> map = new HashMap<>();
  int pagefaults = 0;
  For (int i=0; i< pages.length; i++)
  {int page = pages[i];
   if (set.size() < frames)
   {if (!set.contains(page))
    {set.add(page);
     pagefaults ++;
    }
    map.put(page, i);
   }
   else
   {if (!set.contains(page))
    {int optimal = Integer.Min_value, val=0;
     For (int p : set)
     {if (!Arrays.asList(Arrays.copyOfRange(pages,
                                  i+1, pages.length)),
                                      contains(p))

      {set.remove(p);
       set.add(page);
       pagefaults ++;
       break;
      }
     else
     {if (map.get(p) > optimal)
      {optimal = map.get(p);
       val = p;   } } }
```

```
            if (!set.contains(page))
            {
             set.remove(val);
             set.add(page);
             pagefaults++;
            }
          }
          map.put(page, i);
        }
      }
      System.out.println("Optimal Page replacement
                                        Algorithm");
      System.out.println("Page Faults: " + pagefaults);
    }
    public static void main(String args[])
    {
      int[] pages = {0, 2, 4, 3, 5, 1, 6, 2, 3, 1, 4, 5, 1, 3, 1};
      Fifo(pages, 3);
      Lru(pages, 3);
      optimal(pages, 3);
    }
}
```

Output:

FIFO Page Replacement Algorithm

Page Faults : 14

LRU Page Replacement Algorithm

Page faults : 13

Optimal Page Replacement Algorithm

Page Faults : 13

Result:

Thus, a java program has developed for the implementation of Page Replacement Algorithm is executed and it's output has been verified successfully.

Criteria	Marks
Preparation	20 /20
Observation	20 /25
Interpretation of Result	20 /20
Viva	8 /10
Total	68 /75
Faculty Signature with Date	10/14

Ex.No:6	Implementation of Disk Scheduling Algorithms
Date: 12/4/2023	

Aim:

To develop a java program for the implementation of disk scheduling algorithms.

Algorithm:

Step 1 : Start

Step 2 : In FCFS, it handles the request in which they arrive. Initialise the current position of the disk head.

Step 3: Read the first request in the queue. Move the disk head to the requested position.

Step 4: Read the data at the requested position. Remove the request from the queue. Re

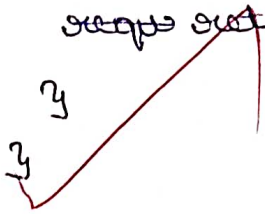
Step 5 : Repeat the above steps for all repeats in the queue.

Step 6 : Stop

Program:

```
import java.util.*;
public class FCFSDiskScheduling {
 public static void main (String[] args) {
  Scanner sc = new Scanner (System.in);
  System.out.print("Enter the no. of disk requests:");
int numRequests = scanner.nextInt();
  System.out.print("Enter the disk request queue:");
  int[] requests = new int [numRequests];
  for (int i=0; i<numRequests; i++) {
     requests[i] = sc.nextInt();
  }
  System.out.print("Enter the initial head position:");
  int initialHead = sc.nextInt();
  int totalHeadMovements = fcfs (requests, initialHead);
  System.out.println("Total head movements: "+totalHead
                                                Movements);
}
public static int fcfs (int[] requests, int initialHead) {
  int totalHeadMovements = 0;
  int currentHead = initialHead;
  for (int i=0; i<requests.length ; i++) {
    int currentRequest = requests[i];
    int headMovement = Math.abs (currentRequest -
                                        currentHead);
```

```
        totalHeadMovements += headMovement;
        currentHead = currentRequest;
    }
    return totalHeadMovements;
}
}
```



Output:

```
Enter the number of disk requests : 5
Enter the disk request queue : 23
89
132
42
187
Enter the initial head position : 100
Total head movements : 421
```

Result:

Thus, a java program has been developed for the implementation of Disk Scheduling Algorithms is executed and it's output has been verified successfully.

Criteria	Marks
Preparation	20 /20
Observation	25 /25
Interpretation of Result	20 /20
Viva	8 /10
Total	73 /75
Faculty Signature with Date	26/4