**Lab Manual**

**Program Name: CSE-AIML,CSE-AIDE, CSE-Cyber Security.**

**Course: Operating System**

**Semester: 2**

**SME: Dr.Gnanaprakasam Thangavel, Prof Naiwrita Borah, Prof Deepanshu S Satwaliya, Prof Sapna Gupta, Dr. Arjun Magotra**

|  |  |  |
| --- | --- | --- |
| **Program Name: To Implement Scheduling Algorithms FCFS** | | |
| 1 | **Background & Use-case** | Process is a program in Execution.  Process life cycle include two cycles: CPU Cycles and I/o Cycles.  Process to be executed with CPU waits in ready queue.  In order to improve CPU utilization, various schemes are rendered, in the order in which process are allotted to the CPU.  FCFS algorithm works on the principle of First Come First Serve. |
| 2 | **Problem Statement** | Implement FCFS the current state of the system as follows.  Process | Arrival | Burst  P2 | 0 | 4  P1 | 1 | 5  P4 | 2 | 5  P3 | 3 | 3 | |
| 3 | **Research Methodology** | Algorithmic Steps  1)Input the CPU burst time of processes bt[i]  2)Evaluate waiting time as wt[i]= wt[i-1] + bt[i-1]  3)Evaluate turnaround time as tt[i] =wt[i]+bt[i]  4)Evaluate avg waiting time =total waiting time/n  5)Evaluate avg turnaround time=total turnaround time/n  Code  print("FIRST COME FIRST SERVE SCHEDULLING")  n= int(input("Enter number of processes : "))  d = dict()  for i in range(n):  key = "P"+str(i+1)  a = int(input("Enter arrival time of process"+str(i+1)+": "))  b = int(input("Enter burst time of process"+str(i+1)+": "))  l = []  l.append(a)  l.append(b)  d[key] = l  d = sorted(d.items(), key=lambda item: item[1][0])  ET = []  for i in range(len(d)):  # first process  if(i==0):  ET.append(d[i][1][1])  # get prevET + newBT  else:  ET.append(ET[i-1] + d[i][1][1])  TAT = []  for i in range(len(d)):  TAT.append(ET[i] - d[i][1][0])  WT = []  for i in range(len(d)):  WT.append(TAT[i] - d[i][1][1])  avg\_WT = 0  for i in WT:  avg\_WT +=i  avg\_WT = (avg\_WT/n)  print("Process | Arrival | Burst | Exit | Turn Around | Wait |")  for i in range(n):  print(" ",d[i][0]," | ",d[i][1][0]," | ",d[i][1][1]," | ",ET[i]," | ",TAT[i]," | ",WT[i]," | ")  print("Average Waiting Time: ",avg\_WT) |
| 4 | **Skills Required** | Programming in Python and knowledge of IDE to execute the program |
| 5 | **Expected output** | Waiting Time of each process, Turnaround time, Average Waiting time, Average Turnaround time  Snapshot of the Result  FIRST COME FIRST SERVE SCHEDULLING    Enter number of processes : 4  Enter arrival time of process1: 1  Enter burst time of process1: 5  Enter arrival time of process2: 0  Enter burst time of process2: 4  Enter arrival time of process3: 3  Enter burst time of process3: 3  Enter arrival time of process4: 2  Enter burst time of process4: 5    Process | Arrival | Burst | Exit | Turn Around | Wait |  P2 | 0 | 4 | 4 | 4 | 0 |  P1 | 1 | 5 | 9 | 8 | 3 |  P4 | 2 | 5 | 14 | 12 | 7 |  P3 | 3 | 3 | 17 | 14 | 11 |  Average Waiting Time: 5.25 |

|  |  |  |
| --- | --- | --- |
| **Program Name: To Implement Scheduling Algorithms SJF** | | |
| 1 | **Background & Use-case** | Process is a program in Execution.  Process life cycle includes two cycles: CPU Cycles and I/o Cycles.  Process to be executed with CPU waits in the ready queue.  In order to improve CPU utilization, various schemes are rendered, in the order in which processes are allotted to the CPU.  SJF algorithm works on the principle of Shortest jobs will be executed First |
| 2 | **Problem Statement** | Implement SJF the current state of the system as follows.  Process Arrival Time Burst Time  P3 1 0  P2 2 1  P1 3 3 |
| 3 | **Research Methodology** | Algorithmic Steps  1)Input the CPU burst time of processes bt[i]  2)Sort the process in the ascending order of their burst time.  3)Evaluate waiting time as wt[i]= wt[i-1] + bt[i-1].  4)Evaluate turnaround time as tt[i] =wt[i]+bt[i]  5)Evaluate avg waiting time =total waiting time/n  6)Evaluate avg turnaround time=total turnaround time/n  Code    # Python 3 implementation of SJF  import heapq as hq  # number of process  SIZE=4  # This function schedules the  # process according to the SJF  # scheduling algorithm.  def sjfNonpremetive(arr):  # Used to sort the processes  # according to arrival time  index = 0  for i in range(SIZE - 1):  index = i  for j in range(i + 1, SIZE) :  if (arr[j][1] < arr[index][1]) :  index = j      arr[i], arr[index]=arr[index],arr[i]    # ctime stores the current run time  ctime = arr[0][1]  # priority queue, wait, is used  # to store all the processes that  # arrive <= ctime (current run time)  # this is a minimum priority queue  # that arranges values according to  # the burst time of the processes.  wait=[]  temp = arr[0][1]  # The first process is  # pushed in the wait queue.  hq.heappush(wait,arr[0].copy())  arr[0][1] = -1  print("Process id",end="\t")  print("Arrival time",end="\t")  print("Burst time",end="\t")  print()  while (wait) :  print(end="\t")  print(wait[0][2],end= "\t\t")  print(wait[0][1],end="\t\t")  print(wait[0][0],end="\t\t")  print()  # ctime is increased with  # the burst time of the  # currently executed process.  ctime += wait[0][0]  # The executed process is  # removed from the wait queue.  hq.heappop(wait)  for i in range(SIZE):  if (arr[i][1] <= ctime  and arr[i][1] != -1) :  hq.heappush(wait,arr[i].copy())  # When the process once  # enters the wait queue  # its arrival time is  # assigned to -1 so that  # it doesn't enter again  # int the wait queue.  arr[i][1] = -1      # Driver Code  if \_\_name\_\_ == '\_\_main\_\_':  # an array of process info structures.  arr=[None]\*SIZE  arr[0] =[3, 4, "p1"]  arr[1] = [8, 0, "p2"]  arr[2] = [4, 5, "p3"]  arr[3] = [2, 9, "p4"]  print("Process scheduling according to SJF is: \n")  sjfNonpremetive(arr) |
| 4 | **Skills Required** | Programming in Python and knowledge of IDE to execute the program |
| 5 | **Expected output** | Waiting Time of each process, Turnaround time, Average Waiting time, Average Turnaround time  Snapshot of the Result  Enter number of processes: 3  Enter Burst Time:  Process 1: 3  Process 2: 2  Process 3: 1  Process Arrival Time Burst Time Waiting Time  P3 1 0 1  P2 2 1 3  P1 3 3 6  Average Waiting Time = 1.33  Average Turnaround Time = 3.33 |

|  |  |  |
| --- | --- | --- |
| **Program Name: To Implement Scheduling Algorithms priority** | | |
| 1 | **Background & Use-case** | Process is a program in Execution.  Process life cycle includes two cycles: CPU Cycles and I/o Cycles.  Process to be executed with CPU waits in the ready queue.  In order to improve CPU utilization, various schemes are rendered, in the order in which processes are allotted to the CPU.  Priority algorithm works on the principle of highest priority jobs will be Executed First |
| 2 | **Problem Statement** | Implement Priority Scheduling the current state of the system as follows.  Processes Burst time  1 10  3 8  2 5 |
| 3 | **Research Methodology** | Algorithmic Steps  1)Input the CPU burst time of processes bt[i]  2)Sort the process in the ascending order of their priority.  3)Evaluate waiting time as wt[i]= wt[i-1] + bt[i-1].  4)Evaluate turnaround time as tt[i] =wt[i]+bt[i]  5)Evaluate avg waiting time =total waiting time/n  6)Evaluate avg turnaround time=total turnaround time/n  Code  # Python3 program for implementation of  # Priority Scheduling  # Function to find the waiting time  # for all processes  def findWaitingTime(processes, n, wt):  wt[0] = 0  # calculating waiting time  for i in range(1, n):  wt[i] = processes[i - 1][1] + wt[i - 1]  # Function to calculate turn around time  def findTurnAroundTime(processes, n, wt, tat):  # Calculating turnaround time by  # adding bt[i] + wt[i]  for i in range(n):  tat[i] = processes[i][1] + wt[i]  # Function to calculate average waiting  # and turn-around times.  def findavgTime(processes, n):  wt = [0] \* n  tat = [0] \* n  # Function to find waiting time  # of all processes  findWaitingTime(processes, n, wt)  # Function to find turn around time  # for all processes  findTurnAroundTime(processes, n, wt, tat)  # Display processes along with all details  print("\nProcesses Burst Time Waiting",  "Time Turn-Around Time")  total\_wt = 0  total\_tat = 0  for i in range(n):  total\_wt = total\_wt + wt[i]  total\_tat = total\_tat + tat[i]  print(" ", processes[i][0], "\t\t",  processes[i][1], "\t\t",  wt[i], "\t\t", tat[i])  print("\nAverage waiting time = %.5f " % (total\_wt / n))  print("Average turn around time = ", total\_tat / n)  def priorityScheduling(proc, n):  # Sort processes by priority  proc = sorted(proc, key=lambda proc: proc[2],  reverse=True)  print("Order in which processes gets executed")  for i in proc:  print(i[0], end=" ")  findavgTime(proc, n)  # Driver code  if \_\_name\_\_ == "\_\_main\_\_":  # Process id's  proc = [[1, 10, 1],  [2, 5, 0],  [3, 8, 1]]  n = 3  priorityScheduling(proc, n) |
| 4 | **Skills Required** | Programming in Python and knowledge of IDE to execute the program |
| 5 | **Expected output** | Waiting Time of each process, Turnaround time, Average Waiting time, Average Turnaround time  Snapshot of the Result  Order in which processes gets executed  1 3 2  Processes Burst time Waiting time Turn around time  1 10 0 10  3 8 10 18  2 5 18 23  Average waiting time = 9.33333  Average turn around time = 17 |

|  |  |  |
| --- | --- | --- |
| **Program Name: To Implement Scheduling Algorithms RR** | | |
| 1 | **Background & Use-case** | Process is a program in Execution.  Process life cycle includes two cycles: CPU Cycles and I/o Cycles.  Process to be executed with CPU waits in the ready queue.  In order to improve CPU utilization, various schemes are rendered, in the order in which processes are allotted to the CPU.  The RR algorithm works on the principle of distribution of fixed timeslice among the processes.. |
| 2 | **Problem Statement** | Quantum=1  Process CPU BURST  P1 1  P2 3  P3 4 |
| 3 | **Research Methodology** | Code  if \_\_name\_\_ == '\_\_main\_\_':  # Python program for implementation of RR Scheduling  print("Enter Total Process Number: ")  total\_p\_no = int(input())  total\_time = 0  total\_time\_counted = 0  # proc is process list  proc = []  wait\_time = 0  turnaround\_time = 0  for \_ in range(total\_p\_no):  # Getting the input for process  print("Enter process arrival time and burst time")  input\_info = list(map(int, input().split(" ")))  arrival, burst, remaining\_time = input\_info[0], input\_info[1], input\_info[1]  # processes are appended to the proc list in following format  proc.append([arrival, burst, remaining\_time, 0])  # total\_time gets incremented with burst time of each process  total\_time += burst  print("Enter time quantum")  time\_quantum = int(input())  # Keep traversing in round robin manner until the total\_time == 0  while total\_time != 0:  # traverse all the processes  for i in range(len(proc)):  # proc[i][2] here refers to remaining\_time for each process i.e "i"  if proc[i][2] <= time\_quantum and proc[i][2] >= 0:  total\_time\_counted += proc[i][2]  total\_time -= proc[i][2]  # the process has completely ended here thus setting it's remaining time to 0.  proc[i][2] = 0  elif proc[i][2] > 0:  # if process has not finished, decrementing it's remaining time by time\_quantum  proc[i][2] -= time\_quantum  total\_time -= time\_quantum  total\_time\_counted += time\_quantum  if proc[i][2] == 0 and proc[i][3] != 1:  # if remaining time of process is 0  # and  # individual waiting time of process has not been calculated i.e flag  wait\_time += total\_time\_counted - proc[i][0] - proc[i][1]  turnaround\_time += total\_time\_counted - proc[i][0]  # flag is set to 1 once wait time is calculated  proc[i][3] = 1  print("\nAvg Waiting Time is ", (wait\_time \* 1) / total\_p\_no)  print("Avg Turnaround Time is ", (turnaround\_time \* 1) / total\_p\_no) |
| 4 | **Skills Required** | Programming in Python and knowledge of IDE to execute the program |
| 5 | **Expected output** | Waiting Time of each process, Turnaround time, Average Waiting time, Average Turnaround time  Snapshot of the Result  Enter Total Process Number:  3  Enter process arrival time and burst time  0 1  Enter process arrival time and burst time  2 3  Enter process arrival time and burst time  3 4  Enter time quantum  1  Avg Waiting Time is 0.6666666666666666  Avg Turnaround Time is 3.3333333333333335 |

|  |  |  |
| --- | --- | --- |
| **Program Name: To Implement Solution for Classic Problems of Synchronization** | | |
| 1 | **Background & Use-case** | Process Synchronisation is used to Synchronize the execution of critical Section Statement. |
| 2 | **Problem Statement** | Implement the solution to the classic Problem of Synchronization Producer -Consumer Problem. |
| 3 | **Research Methodology** | // Python program for the above approach    import threading  import time    # Shared Memory variables  CAPACITY = 10  buffer = [-1 for i in range(CAPACITY)]  in\_index = 0  out\_index = 0    # Declaring Semaphores  mutex = threading.Semaphore()  empty = threading.Semaphore(CAPACITY)  full = threading.Semaphore(0)    # Producer Thread Class  class Producer(threading.Thread):  def run(self):    global CAPACITY, buffer, in\_index, out\_index  global mutex, empty, full    items\_produced = 0  counter = 0    while items\_produced < 20:  empty.acquire()  mutex.acquire()    counter += 1  buffer[in\_index] = counter  in\_index = (in\_index + 1)%CAPACITY  print("Producer produced : ", counter)    mutex.release()  full.release()    time.sleep(1)    items\_produced += 1    # Consumer Thread Class  class Consumer(threading.Thread):  def run(self):    global CAPACITY, buffer, in\_index, out\_index, counter  global mutex, empty, full    items\_consumed = 0    while items\_consumed < 20:  full.acquire()  mutex.acquire()    item = buffer[out\_index]  out\_index = (out\_index + 1)%CAPACITY  print("Consumer consumed item : ", item)    mutex.release()  empty.release()    time.sleep(2.5)    items\_consumed += 1    # Creating Threads  producer = Producer()  consumer = Consumer()    # Starting Threads  consumer.start()  producer.start()    # Waiting for threads to complete  producer.join()  consumer.join() |
| 4 | **Skills Required** | Programming in Python and knowledge of IDE to execute the program |
| 5 | **Expected output** | Producer produced : 1  Consumer consumed item : 1  Producer produced : 2  Producer produced : 3  Consumer consumed item : 2  Producer produced : 4  Producer produced : 5  Consumer consumed item : 3  Producer produced : 6  Producer produced : 7  Producer produced : 8  Consumer consumed item : 4  Producer produced : 9  Producer produced : 10  Consumer consumed item : 5  Producer produced : 11  Producer produced : 12  Producer produced : 13  Consumer consumed item : 6  Producer produced : 14  Producer produced : 15  Consumer consumed item : 7  Producer produced : 16  Producer produced : 17  Consumer consumed item : 8  Producer produced : 18  Consumer consumed item : 9  Producer produced : 19  Consumer consumed item : 10  Producer produced : 20  Consumer consumed item : 11  Consumer consumed item : 12  Consumer consumed item : 13  Consumer consumed item : 14  Consumer consumed item : 15  Consumer consumed item : 16  Consumer consumed item : 17  Consumer consumed item : 18  Consumer consumed item : 19  Consumer consumed item : 20 |

|  |  |  |
| --- | --- | --- |
| **Program Name: Bankers Algorithm** | | |
| 1 | **Background & Use-case** | Deadlock Avoidance Algorithm |
| 2 | **Problem Statement** | To run the safety algorithm to avoid Deadlock |
| 3 | **Research Methodology** | # Python3 program to illustrate  # Banker's Algorithm  # Number of processes  P = 5  # Number of resources  R = 3  # Function to find the need of each process  def calculateNeed(need, maxm, allot):  # Calculating Need of each P  for i in range(P):  for j in range(R):    # Need of instance = maxm instance -  # allocated instance  need[i][j] = maxm[i][j] - allot[i][j]  # Function to find the system is in  # safe state or not  def isSafe(processes, avail, maxm, allot):  need = []  for i in range(P):  l = []  for j in range(R):  l.append(0)  need.append(l)    # Function to calculate need matrix  calculateNeed(need, maxm, allot)  # Mark all processes as infinish  finish = [0] \* P    # To store safe sequence  safeSeq = [0] \* P  # Make a copy of available resources  work = [0] \* R  for i in range(R):  work[i] = avail[i]  # While all processes are not finished  # or system is not in safe state.  count = 0  while (count < P):    # Find a process which is not finish  # and whose needs can be satisfied  # with current work[] resources.  found = False  for p in range(P):    # First check if a process is finished,  # if no, go for next condition  if (finish[p] == 0):    # Check if for all resources  # of current P need is less  # than work  for j in range(R):  if (need[p][j] > work[j]):  break    # If all needs of p were satisfied.  if (j == R - 1):    # Add the allocated resources of  # current P to the available/work  # resources i.e.free the resources  for k in range(R):  work[k] += allot[p][k]  # Add this process to safe sequence.  safeSeq[count] = p  count += 1  # Mark this p as finished  finish[p] = 1  found = True    # If we could not find a next process  # in safe sequence.  if (found == False):  print("System is not in safe state")  return False    # If system is in safe state then  # safe sequence will be as below  print("System is in safe state.",  "\nSafe sequence is: ", end = " ")  print(\*safeSeq)  return True  # Driver code  if \_\_name\_\_ =="\_\_main\_\_":    processes = [0, 1, 2, 3, 4]  # Available instances of resources  avail = [3, 3, 2]  # Maximum R that can be allocated  # to processes  maxm = [[7, 5, 3], [3, 2, 2],  [9, 0, 2], [2, 2, 2],  [4, 3, 3]]  # Resources allocated to processes  allot = [[0, 1, 0], [2, 0, 0],  [3, 0, 2], [2, 1, 1],  [0, 0, 2]]  # Check system is in safe state or not  isSafe(processes, avail, maxm, allot) |
| 4 | **Skills Required** | Programming in Python and knowledge of IDE to execute the program |
| 5 | **Expected output** | System is in safe state.  Safe sequence is: 1 3 4 0 2 |

|  |  |  |
| --- | --- | --- |
| **Program Name: FIFO Page Replacement Algorithm** | | |
| 1 | **Background & Use-case** | Operating systems use FIFO as a simple page replacement algorithm to manage memory. When a page fault occurs, it replaces the oldest page in the page frames, assuming it is the least likely to be needed in the near future. Because of its simplicity and speed, it is widely used in embedded and real-time systems. Increasing the number of page frames, on the other hand, can sometimes result in more page faults, a phenomenon known as Belady's anomaly. Despite this, FIFO remains popular and useful for comparing more complex page replacement algorithms. |
| 2 | **Problem Statement** | To implement FIFO page replacement algorithm |
| 3 | **Research Methodology** | The problem that FIFO page replacement algorithm aims to solve is how to manage the allocation of physical memory to processes in a way that minimizes the number of page faults. When a process tries to access a page that is not currently in memory, a page fault occurs, and the operating system must fetch the missing page from disk, which is a slow and expensive operation.  The FIFO algorithm works by maintaining a queue of page frames that are currently in memory. When a new page needs to be loaded into memory, the oldest page (i.e., the one at the front of the queue) is evicted to make room for the new page. This ensures that the page frames are used in a first-in, first-out order.    Algorithm :    The FIFO (First-In, First-Out) page replacement algorithm is a simple and straightforward algorithm that operates based on a queue of page frames that are currently in memory. When a new page needs to be loaded into memory, the oldest page (i.e., the one at the front of the queue) is evicted to make room for the new page. The steps involved in the FIFO algorithm can be summarized as follows:   1. Initialize an empty queue of page frames. 2. For each page reference in the reference string: a. Check if the page is already in memory. b. If the page is already in memory, do nothing and continue to the next page reference. c. If the page is not in memory, check if there is an empty page frame available. i. If there is an empty page frame, load the page into the empty frame. ii. If there is no empty page frame, evict the oldest page (i.e., the page at the front of the queue) and load the new page into that page frame. d. Increment the page fault counter. 3. Output the total number of page faults.     Code:  # Python3 implementation of FIFO page  # replacement in Operating Systems.  from queue import Queue  # Function to find page faults using FIFO  def pageFaults(incomingStream, n, frames):  print("Incoming \t pages")  # Using Hashset to quickly check if a given  # incoming stream item in set or not  s = set()  # Queue created to store pages in FIFO manner  # since set will not store order or entry  # we will use queue to note order of entry of incoming page  queue = Queue()  page\_faults = 0  for i in range(n):  # if set has lesser item than frames  # i.e. set can hold more items  if len(s) < frames:  # If incoming item is not present, add to set  if incomingStream[i] not in s:  s.add(incomingStream[i])  # increment page fault  page\_faults += 1  # Push the incoming page into the queue  queue.put(incomingStream[i])  # If the set is full then we need to do page replacement  # in FIFO manner that is remove first item from both  # set and queue then insert incoming page  else:  # If incoming item is not present  if incomingStream[i] not in s:  # remove the first page from the queue  val = queue.queue[0]  queue.get()  # Remove from set  s.remove(val)  # insert incoming page to set  s.add(incomingStream[i])  # push incoming page to queue  queue.put(incomingStream[i])  # Increment page faults  page\_faults += 1  print(incomingStream[i], end="\t\t")  for q\_item in queue.queue:  print(q\_item, end="\t")  print()  return page\_faults  # Driver code  incomingStream = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1]  n = len(incomingStream)  frames = 3  page\_faults = pageFaults(incomingStream, n, frames)  hits = n - page\_faults  print("\nPage Faults: " + str(page\_faults))  print("Hit: " + str(hits)) |
| 4 | **Skills Required** | To implement page replacement algorithms in Python, a student should follow the following steps:   1. Have a good understanding of Python programming language syntax and structure. 2. Understand operating systems concepts such as memory management, page replacement, and paging. 3. Get familiar with algorithms and data structures such as linked lists, queues, and stacks, which are commonly used in page replacement algorithms. 4. Develop proficiency in debugging and testing code to ensure that it works correctly. 5. Gain knowledge of optimization techniques, such as loop unrolling and caching, to help improve code performance. 6. Be aware of version control systems such as Git, which can help keep track of changes to code and collaborate with others. |
| 5 | **Expected output** | Incoming pages  7 7  0 7 0  1 7 0 1  2 0 1 2  0 0 1 2  3 1 2 3  0 2 3 0  4 3 0 4  2 0 4 2  3 4 2 3  0 2 3 0  3 2 3 0  2 2 3 0  1 3 0 1  Page Faults: 11  Hit: 3 |

|  |  |  |
| --- | --- | --- |
| **Program Name: Optimal Page Replacement Algorithm** | | |
| 1 | **Background & Use-case** | OPT is an algorithm for optimal page replacement used in computer operating systems to reduce page faults when managing virtual memory. It is useful when physical memory is constrained and virtual memory contains more pages than can fit in physical memory. It chooses the page that will be inactive for the longest amount of time in the future. It offers the best performance possible in terms of page faults, but it is difficult to implement because it requires knowledge of future page access patterns. Consequently, it is typically employed for benchmarking and performance evaluation rather than in operational systems. |
| 2 | **Problem Statement** | To implement Optimal page replacement algorithm |
| 3 | **Research Methodology** | The Optimal page replacement algorithm chooses the page that will be inactive for the longest amount of time in the future. It is difficult to implement because the future page access pattern must be known. Here are the instructions for the OPT algorithm in sequential order:    1. When a page error occurs, check if a free page frame is available. If there is a free page frame, load the page into it.  2. Select the page that will not be used for the longest period of time in the future if there is no available page frame. This requires knowledge of the page access pattern in the future, which is generally impossible to predict.  3. If the selected page is dirty, it should be written back to disk before being replaced.  4. Load the requested page into the page frame of choice.  5. Repeat the preceding steps for each page fault.    Notably, it is challenging to implement the OPT algorithm in practice because it requires knowledge of future page access patterns. Consequently, it is commonly used for benchmarking and performance evaluation.    Source Code :      #programming code for Optimal Page Replacement Algorithm  #entering the number of frames  print(“Enter the number of frames: ”, end = “”)  storage = int(input())  f, fault, pf = [], 0, ‘No’  #entering the referencing string  print(“Fill the reference string”, end =””)  string = list(map(int, input().strip().split()))  print(“\nString|Frame -> t”, end =’’)  #calculating the number of requests  for i in range (storage)  print(I, end= ‘ ‘)  print(“Fault\n ↓\n”)  occurrence = [None for i in range(storage)]  #calculating the number of faults  for i in range (len(s));  if s[i] not in f;  if len(f)< storage  f.append(s[i])  else:  for x in range(len(f));  if f[x] not in s[i+1]:  f[x] = s[i]  break  else:  occurrence[x] = s[i+1:].index(f[x])  else:    #calculating the fault rate  if[occurance.index(max(occurence))] = s[i]  fault += 1  pf= ‘Yes’  else:  pf = ‘No’  print(“ %d\t\t” %s[i], end= ‘’)  for x in f:  print (x, end = ‘ ’)  for x in range (storage – len(f)):  print(‘ ’, end = ‘ ’)  for x in range (storage – len(f)):  print (‘ ’, end = ‘ ’)  print (“%s” %pf )  print(“\n Total number of requests: %d\n Total number of page faults: %d, Fault Rate: %0.2f%%"%(len(s),fault,(fault/len(s))\*100)”))  } |
| 4 | **Skills Required** | To implement page replacement algorithms in Python, a student should follow the following steps:   1. Have a good understanding of Python programming language syntax and structure. 2. Understand operating systems concepts such as memory management, page replacement, and paging. 3. Get familiar with algorithms and data structures such as linked lists, queues, and stacks, which are commonly used in page replacement algorithms. 4. Develop proficiency in debugging and testing code to ensure that it works correctly. 5. Gain knowledge of optimization techniques, such as loop unrolling and caching, to help improve code performance. 6. Be aware of version control systems such as Git, which can help keep track of changes to code and collaborate with others. |
| 5 | **Expected output** | Enter the number of frames:  3  Fill the reference string:  3 2 1 5 3 9  Total number of requests:  6  Total number of page faults:  2  Fault rate:  2 |

|  |  |  |
| --- | --- | --- |
| **Program Name: LRU Page Replacement Algorithm** | | |
| 1 | **Background & Use-case** | LRU is a popular page replacement algorithm that attempts to keep the most frequently used pages in memory. When a page fault occurs, it replaces the most recently used page, utilising the principle of locality to reduce the number of page faults and improve performance. LRU is widely used in desktop and server operating systems, but it is more difficult to implement than other algorithms and requires the maintenance of data structures, which can be computationally expensive. |
| 2 | **Problem Statement** | To implement LRU page replacement algorithm |
| 3 | **Research Methodology** | The problem that LRU (Least Recently Used) page replacement algorithm aims to solve is how to manage memory effectively in an operating system. As programs run, they access different pages of memory, and the operating system needs to keep track of which pages are currently in use and which can be removed from memory to make space for new pages. The goal is to minimize the number of page faults, which occur when a program requests a page that is not currently in memory, by keeping the most frequently used pages in memory.  The LRU algorithm solves this problem by replacing the page that has been used least recently when a page fault occurs. This allows the most frequently used pages to remain in memory and reduces the likelihood of page faults, improving overall system performance. The challenge is to maintain the data structures required to keep track of the most recently used pages efficiently, which can be computationally expensive.  Here are the stepwise algorithm for LRU (Least Recently Used) page replacement algorithm:   1. Initialize a page frame of fixed size. 2. Initialize a counter for each page frame to keep track of when it was last accessed. 3. When a new page needs to be added to the page frame, check if it is already present in the page frame. If it is, update its counter to the current time and continue. If it is not, find the page with the lowest counter value, which indicates it was accessed least recently. 4. Replace the page with the lowest counter value with the new page and update its counter to the current time. 5. Repeat steps 3 and 4 for every new page that needs to be added to the page frame.   The basic idea behind LRU is to keep track of when pages were last accessed and replace the least recently used page when a new page needs to be added to the page frame. This allows the most frequently used pages to remain in memory, reducing the number of page faults and improving system performance.    **Source Code :**  # Python implementation of above algorithm  def pageFaults(pages, n, capacity):  # To represent set of current pages. We use  # an unordered\_set so that we quickly check  # if a page is present in set or not  s = set()  # To store least recently used indexes  # of pages.  indexes = {}  # Start from initial page  page\_faults = 0  for i in range(n):  # Check if the set can hold more pages  if len(s) < capacity:  # Insert it into set if not present  # already which represents page fault  if pages[i] not in s:  s.add(pages[i])  # increment page fault  page\_faults += 1  # Store the recently used index of  # each page  indexes[pages[i]] = i  # If the set is full then need to perform lru  # i.e. remove the least recently used page  # and insert the current page  else:  # Check if current page is not already  # present in the set  if pages[i] not in s:  # Find the least recently used pages  # that is present in the set  lru = float('inf')  for page in s:  if indexes[page] < lru:  lru = indexes[page]  val = page  # Remove the indexes page  s.remove(val)  # insert the current page  s.add(pages[i])  # increment page fault  page\_faults += 1  # Update the current page index  indexes[pages[i]] = i  return page\_faults  # Driver code  pages = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]  n = len(pages)  capacity = 4  print(pageFaults(pages, n, capacity)) |
| 4 | **Skills Required** | To implement page replacement algorithms in C, a student should follow the following steps:   1. Have a good understanding of C programming language syntax and structure. 2. Understand operating systems concepts such as memory management, page replacement, and paging. 3. Get familiar with algorithms and data structures such as linked lists, queues, and stacks, which are commonly used in page replacement algorithms. 4. Develop proficiency in debugging and testing code to ensure that it works correctly. 5. Gain knowledge of optimization techniques, such as loop unrolling and caching, to help improve code performance. 6. Be aware of version control systems such as Git, which can help keep track of changes to code and collaborate with others. |
| 5 | **Expected output** | Total number of page faults:  6 |

|  |  |  |
| --- | --- | --- |
| **Program Name: Memory Allocation Algorithm (Best Fit,First Fit ,Worst Fit)** | | |
| 1 | **Background & Use-case** | First-Fit Allocation is a memory allocation technique used in operating systems to allocate memory to a process. In First-Fit, the operating system searches through the list of free blocks of memory, starting from the beginning of the list, until it finds a block that is large enough to accommodate the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.  **Advantages** of First-Fit Allocation include its simplicity and efficiency, as the search for a suitable block of memory can be performed quickly and easily. Additionally, First-Fit can also help to minimize memory fragmentation, as it tends to allocate memory in larger blocks.  **Disadvantages** of First-Fit Allocation include poor performance in situations where the memory is highly fragmented, as the search for a suitable block of memory can become time-consuming and inefficient. Additionally, First-Fit can also lead to poor memory utilization, as it may allocate larger blocks of memory than are actually needed by a process.  Overall, First-Fit Allocation is a widely used memory allocation technique in operating systems, but its effectiveness may vary depending on the specifics of the system and the workload being executed.  For both [fixed and dynamic memory allocation schemes](https://www.geeksforgeeks.org/partition-allocation-methods-in-memory-management/), the operating system must keep list of each memory location noting which are free and which are busy. Then as new jobs come into the system, the free partitions must be allocated. These partitions may be allocated by 3 ways:  1. First-Fit Memory Allocation  2. Best-Fit Memory Allocation  3. Next-Fit Memory Allocation |
| 2 | **Problem Statement** | * Write a Program Code for First Fit Algorithm. * Write a Program Code for Best Fit Algorithm. * Write a Program Code for Worst Fit Algorithm. |
| 3 | **Research Methodology** | **Write a Program Code for First Fit Algorithm.**  def FirstFit(block\_Size, blocks, process\_Size, proccesses):  # code to store the block id of the block that needs to be allocated to a process  allocate = [-1] \* proccesses  occupied = [False] \* blocks  # Any process is assigned with the memory at the initial stage  # find a suitable block for each process  # the blocks are allocated as per their size  for i in range(proccesses):  for j in range(blocks):  if not occupied[j] and (block\_Size[j] >= process\_Size[i]):  # assign the block j to p[i] process  allocate[i] = j  occupied[j] = True  break  print("Process No. Process Size Block No.")  for i in range(proccesses):  print(str(i + 1) + "\t\t\t" + str(process\_Size[i]) + "\t\t\t", end=" ")  if allocate[i] != -1:  print(allocate[i] + 1)  else:  print("Not Allocated")  # Driver code  block\_Size = [100, 50, 30, 120, 35]  process\_Size = [20, 60, 70, 40]  m = len(block\_Size)  n = len(process\_Size)  FirstFit(block\_Size, m, process\_Size, n)    **Write a Program Code for Best Fit Algorithm.**  # Python3 implementation of the First  # sit memory management algorithm  # using linked list  # Two global counters  g = 0; k = 0  # Structure for free list  class free:  def \_\_init\_\_(self):  self.tag=-1  self.size=0  self.next=None  free\_head = None; prev\_free = None  # Structure for allocated list  class alloc:  def \_\_init\_\_(self):  self.block\_id=-1  self.tag=-1  self.size=0  self.next=None  alloc\_head = None;prev\_alloc = None  # Function to create free  # list with given sizes  def create\_free(c):  global g,prev\_free,free\_head  p = free()  p.size = c  p.tag = g  p.next = None  if free\_head is None:  free\_head = p  else:  prev\_free.next = p  prev\_free = p  g+=1  # Function to print free list which  # prints free blocks of given sizes  def print\_free():  p = free\_head  print("Tag\tSize")  while (p != None) :  print("{}\t{}".format(p.tag,p.size))  p = p.next    # Function to print allocated list which  # prints allocated blocks and their block ids  def print\_alloc():  p = alloc\_head  print("Tag\tBlock ID\tSize")  while (p is not None) :  print("{}\t{}\t{}\t".format(p.tag,p.block\_id,p.size))  p = p.next    # Function to allocate memory to  # blocks as per First fit algorithm  def create\_alloc(c):  global k,alloc\_head  # create node for process of given size  q = alloc()  q.size = c  q.tag = k  q.next = None  p = free\_head  # Iterate to find first memory  # block with appropriate size  while (p != None) :  if (q.size <= p.size):  break  p = p.next    # Node found to allocate  if (p != None) :  # Adding node to allocated list  q.block\_id = p.tag  p.size -= q.size  if (alloc\_head == None):  alloc\_head = q  else :  prev\_alloc = alloc\_head  while (prev\_alloc.next != None):  prev\_alloc = prev\_alloc.next  prev\_alloc.next = q    k+=1    else: # Node found to allocate space from  print("Block of size {} can't be allocated".format(c))  # Function to delete node from  # allocated list to free some space  def delete\_alloc(t):  global alloc\_head  # Standard delete function  # of a linked list node  p = alloc\_head; q = None  # First, find the node according  # to given tag id  while (p != None) :  if (p.tag == t):  break  q = p  p = p.next    if (p == None):  print("Tag ID doesn't exist")  elif (p == alloc\_head):  alloc\_head = alloc\_head.next  else:  q.next = p.next  temp = free\_head  while (temp != None) :  if (temp.tag == p.block\_id) :  temp.size += p.size  break    temp = temp.next    # Driver Code  if \_\_name\_\_ == '\_\_main\_\_':  blockSize = [100, 500, 200]  processSize = [417, 112, 426, 95]  m = len(blockSize)  n = len(processSize)  for i in range(m):  create\_free(blockSize[i])  for i in range(n):  create\_alloc(processSize[i])  print\_alloc()  # Block of tag id 0 deleted  # to free space for block of size 426  delete\_alloc(0)  create\_alloc(426)  print("After deleting block with tag id 0.")  print\_alloc()    **Write a Program Code for Worst Fit Algorithm.**  # Python3 implementation of worst - Fit algorithm  # Function to allocate memory to blocks as  # per worst fit algorithm  def worstFit(blockSize, m, processSize, n):    # Stores block id of the block  # allocated to a process    # Initially no block is assigned  # to any process  allocation = [-1] \* n    # pick each process and find suitable blocks  # according to its size ad assign to it  for i in range(n):    # Find the best fit block for  # current process  wstIdx = -1  for j in range(m):  if blockSize[j] >= processSize[i]:  if wstIdx == -1:  wstIdx = j  elif blockSize[wstIdx] < blockSize[j]:  wstIdx = j  # If we could find a block for  # current process  if wstIdx != -1:    # allocate block j to p[i] process  allocation[i] = wstIdx  # Reduce available memory in this block.  blockSize[wstIdx] -= processSize[i]  print("Process No. Process Size Block no.")  for i in range(n):  print(i + 1, " ",  processSize[i], end = " ")  if allocation[i] != -1:  print(allocation[i] + 1)  else:  print("Not Allocated")  # Driver code  if \_\_name\_\_ == '\_\_main\_\_':  blockSize = [100, 500, 200, 300, 600]  processSize = [212, 417, 112, 426]  m = len(blockSize)  n = len(processSize)  worstFit(blockSize, m, processSize, n) |
| 4 | **Skills Required** | Programming in Python and knowledge of IDE to execute the program |
| 5 | **Expected output** | Show the First Fit, Best Fit & Worst Fit Allocation.  **Result of First Fit:- *(as given below)***  Process No. Process Size Block No.  1 20 1  2 60 4  3 70 Not Allocated  4 40 2  **Result of Best Fit:- *(as given below)***  Block with size 426 can't be allocated  Tag Block ID Size  0 0 95  1 1 417  2 2 112  After deleting block with tag id 1.  Tag Block ID Size  0 0 95  2 2 112  3 1 426  **Result of Worst Fit:- *(as given below)***  Process No. Process Size Block no.  1 212 5  2 417 2  3 112 5  4 426 Not Allocated |

|  |  |  |
| --- | --- | --- |
| **Program Name: Implementation of File and I/O** | | |
| 1 | **Background & Use-case** | File Handling is the storing of data in a file using a program. In C programming language, the programs store results, and other data of the program to a file using *file handling* in C. Also, we can extract/fetch data from a file to work with it in the program.  The operations that you can perform on a File in C are −   * Creating a new file * Opening an existing file * Reading data from an existing file * Writing data to a file * Moving data to a specific location on the file * Closing the file * Mode = “r” − open for reading, this mode will open the file for reading purpose only, i.e. the contents can only be viewed, nothing else like edits can be done to it. * This mode cannot create a new file and open() returns NULL, if we try to create a new file using this mode. |
| 2 | **Problem Statement** | 1.Write program to copy contents of one file to another file. |
| 3 | **Research Methodology** | # open both files  with open('first.txt','r') as firstfile, open('second.txt','a') as secondfile:    # read content from first file  for line in firstfile:    # append content to second file  secondfile.write(line)   1. Program Count the Numbers of Words in a File   file = open("C:\data.txt", "rt")  data = file.read()  words = data.split()  print('Number of words in text file :', len(words))    3.Program to Write a Sentence to a File    # Open the file for writing  with open('file.txt', 'w') as f:  # Define the data to be written  data = ['This is the first line', 'This is the second line', 'This is the third line']  # Use a for loop to write each line of data to the file  for line in data:  f.write(line + '\n')  # Optionally, print the data as it is written to the file  print(line)  # The file is automatically closed when the 'with' block ends   |  | | --- | |  | |
| 4 | **Skills Required** | To implement page replacement algorithms in Python, a student should follow the following steps:   1. Have a good understanding of Python programming language syntax and structure. 2. Understand operating systems concepts such as memory management, page replacement, and paging. 3. Get familiar with algorithms and data structures such as linked lists, queues, and stacks, which are commonly used in page replacement algorithms. 4. Develop proficiency in debugging and testing code to ensure that it works correctly. 5. Gain knowledge of optimization techniques, such as loop unrolling and caching, to help improve code performance.   Be aware of version control systems such as Git, which can help keep track of changes to code and collaborate with others. |
| 5 | **Expected output** | Python 9  2.Hello World  Number of words in text file : 21  3.Enter sentence that you would like to write:  This is the first line  This is the second line  This is the third line |

|  |  |  |
| --- | --- | --- |
| **Program Name: Linux Shell Scripting** | | |
| 1 | Background & Use-case | Linux is an open source operating system (OS). An operating system is the software that directly manages a system’s hardware and resources, like CPU, memory, and storage. The OS sits between applications and hardware and makes the connections between all of your software and the physical resources that do the work.  The Linux command is a utility of the Linux operating system. All basic and advanced tasks can be done by executing commands. The commands are executed on the Linux terminal. The terminal is a command-line interface to interact with the system, which is similar to the command prompt in the Windows OS. Commands in Linux are case-sensitive. **What is Linux Good For?** Linux appears almost everywhere. You can find it on the desktop, server, cloud, mobile devices, supercomputers, and as part of the Internet of Things (IoT), among other platforms. Linux is important because it is customizable, you don’t have to jump through someone else’s hoop to create an application, it’s fast, and it works on older hardware. The Linux learning curve can cause woe initially, but the experience gained in working with Linux translates to all of the platforms it supports, which are many. Linux is actually the basis for other operating systems, like Android, because it does provide so much flexibility. **Why is Linux Important?** The most critical reason to use Linux is that it provides a means of increasing return on investment (ROI) in every environment because you don’t necessarily need to pay anything for the operating system and it’s potentially possible to use hardware that you already have. Linux is important because it can make the difference between your organization seeing a profit or experiencing a loss.  Linux is also one of the most successful examples of an open source project. Its code is publicly accessible, anyone can modify and distribute its code, and it’s developed collaboratively with the Linux community. These facets directly contribute to its key benefits, like strong security. **Benefits of Linux** There’s a strong case to make for using Linux in your business, research project, web application, specialized environment, or to address some other practical need. The sections below outline ten benefits of using Linux in your organization. **Strong Security** Linux is more secure than most other operating systems today because:   * Linux has numerous distributions and more appear every day. Some well-known distributions include Ubuntu, Debian, and CentOS. All of them are updated regularly, so creating a virus that targets Linux as a whole is nearly impossible because it’s a moving target. * Linux assumes that everyone only has privileges to their own applications and data. When an application is installed and configured by the administrator, the user can’t do anything with it other than use it. * Linux also isn’t the most popular operating system out there when you consider its use in a standalone venue. Virus writers tend to go for the lower hanging fruit: Operating systems that affect a lot of people.  **Low Cost** One of the pros of Linux is that the Linux kernel is free and it comes under the open source [GNU GPL (General Public License)](https://www.gnu.org/licenses/gpl-3.0.en.html) so you can add whatever you want to it to create a custom configuration. It’s possible to download just about every Linux distribution. For example you can download a form of the Linux kernel with a few add-ons, that is fully functional, out of the box, without cost. Add-ons such as the paid services for [Ubuntu](https://ubuntu.com/pricing), and [Red Hat Enterprise](https://www.redhat.com/en/store/linux-platforms), may cost you. **Terminal Support** You don’t need to install special software to contact the backend servers for your project using add-on software with Linux. All you need is the Secure Shell (SSH) utility to access the server securely. In addition, you have access to editors like [Emacs](https://www.linode.com/docs/guides/emacs-evil-mode/), [Nano](https://www.linode.com/docs/guides/use-nano-text-editor-commands/), and [Vim](https://www.linode.com/docs/guides/introduction-to-vim-customization/) that allow you to update config files or hosted Python scripts on the fly. These advantages of using Linux mean that developers spend more time writing and testing code than figuring out some arcane process to complete tasks. **Amazing Driver Support** Linux comes with the drivers you need right out of the box so you won’t waste time searching for the driver disk to use with your device. One of the pros of Linux is that the device support doesn’t end with today’s devices. It’s quite possible to revitalize an older machine to use as a firewall, router, or backup server. **Great Scalability** The ability to scale is essential for any practical software need. Linux provides scalability in several ways:   * It runs on so many different platforms that it is hard to find a device that you can’t use Linux on. It scales to the device you need to use, including the ubiquitous robots and industrial computers. * It has a minimal footprint so you can use it on devices of nearly any capability. * It doesn’t bog down under load. * It only uses resources when the application isn’t using them.  **Strong Developer Support** Flexibility doesn’t sell a user on an operating system, applications do. Linux supports a plethora of [programming languages](https://www.slant.co/topics/635/~best-languages-to-write-a-desktop-linux-application-in) that developers use in various disciplines. For example, if you’re a data scientist and rely on Python as your programming language, then you run your applications on Linux because they run faster and with fewer resources. Linux supports all of the major languages including C, C++, Java, and JavaScript. If you have any sort of business, research, or other practical software need, Linux runs the code faster with fewer resources. **Available Source Code** Source code for the Linux kernel is readily available, so it’s possible to view whenever needed. This availability has a number of advantages over other operating systems:   * There are many people looking for potential flaws in Linux and many coming up with solutions. * It’s possible to track flaws down or modify Linux to meet specific needs. * It’s possible to discover how the operating system works by observing the source code in a debugger.  **Compatible Applications** Because of the way the operating system is designed and the fact that the kernel is open source, software developers have better knowledge of how to create compatible, reliable, applications. There are also fewer layers to deal with when working with Linux. **Incredible Server Support** Most developers view Linux as a server operating system and are careful to create applications that run in such an environment. This means that Linux applications often provide robust security and reliability, and still offer an application that runs quickly. Many professionals are attracted to Linux because they don’t have time to keep fixing the operating system and want it to be as invisible as possible. Consequently, you often find Linux used as a workstation operating system for professionals of all stripes. **Flexible UI Customization** When working with most operating systems, the user only has access to one user interface (UI). If the user doesn’t like how that UI works, that’s too bad. With Linux, it’s possible to [install any of a number of user interfaces](https://www.cbtnuggets.com/blog/certifications/open-source/5-linux-graphical-user-interfaces-compared) and customize those interfaces as needed. This means extra work at the outset and a higher learning curve, but the user eventually ends up with the right UI to meet specific needs. In short, the UI becomes invisible.  [Linux](https://www.javatpoint.com/linux-tutorial) provides a powerful command-line interface compared to other operating systems such as [Windows](https://www.javatpoint.com/windows) and MacOS. We can do basic work and advanced work through its terminal. We can do some basic tasks such as creating a file, deleting a file, moving a file, and more. In addition, we can also perform advanced tasks such as administrative tasks (including package installation, user management), networking tasks (ssh connection), security tasks, and many more. |
| 2 | Problem Statement | **Linux Commands**pwd Command The pwd command stands for “print working directory,” and it outputs the absolute path of the directory you’re in. For example, if your username is “Aurobindo” and you’re in your Documents directory, its absolute path would be: /home/Aurobindo/Documents.  To use it, simply type pwd in the terminal:  pwd  # My result: /home/kinsta/Documents/linux-commands cd Command The cd command refers to “change directory” and, as its name suggests, switches you to the directory you’re trying to access.  For instance, if you’re inside your Documents directory and you’re trying to access one of its subfolders called CSE, you can enter it by typing:  cd CSE  You can also supply the absolute path of the folder:  cd /home/Deepanshu/CSE  some tricks with the cd command Go to the home folder: cdMove a level up: cd ..Return to the previous directory: cd -  1. **mkdir Command:** The [mkdir](https://www.javatpoint.com/linux-mkdir) command is used to create a new directory under any directory.   **Syntax: mkdir <directory name>**   1. **rmdir Command:** The [rmdir](https://www.javatpoint.com/linux-rmdir) command is used to delete a directory.   Syntax: **rmdir <directory name>**   1. **ls Command:** The [ls](https://www.javatpoint.com/linux-ls) command is used to display a list of content of a directory.   Syntax: ls   1. **touch Command:** The [touch](https://www.javatpoint.com/linux-touch) command is used to create empty files. We can create multiple empty files by executing it once.   Syntax:   1. touch <file name> 2. touch <file1> <file2> .... 3. **cat Command:** The [cat](https://www.javatpoint.com/linux-cat) command is a multi-purpose utility in the Linux system. It can be used to create a file, display content of the file, copy the content of one file to another file, and more.   Syntax: cat [OPTION]... [FILE]..  To create a file, execute it as follows:   1. cat > <file name> 2. // Enter file content   Press "CTRL+ D" keys to save the file. To display the content of the file, execute it as follows:   1. cat <file name> 2. rm Command: The [rm](https://www.javatpoint.com/linux-rm) command is used to remove a file.   Syntax:  rm <file name>   1. cp Command: The [cp](https://www.javatpoint.com/linux-cp) command is used to copy a file or directory.   Syntax:To copy in the same directory:   1. cp <existing file name> <new file name> 2. cp file\_to\_copy.txt new\_file.txt 3. You can also copy entire directories by using the recursive flag: 4. cp -r dir\_to\_copy/ new\_copy\_dir/ 5. Remember that in Linux, folders end with a forward slash (/).  rm Command: rm command to remove files and directories.rm file\_to\_copy.txt rm -r dir\_to\_remove/  to remove a directory with content inside of it, you need to use the force (-f) and recursive flags:  rm -rf dir\_with\_content\_to\_remove/  mv Command: mv command to move (or rename) files and directories through your file system. mv source\_file destination\_folder/  mv command\_list.txt commands/ mkdir Command: To create folders in the shell, you use the mkdir command. Just specify the new folder’s name, ensure it doesn’t exist, and you’re ready to go. mkdir images/  To create subdirectories with a simple command, use the parent (-p) flag:  mkdir -p movies/2004/ man Command It displays the manual page of any other command (as long as it has one).  To see the manual page of the mkdir command, type:  man mkdir  You could even refer to the man manual page:  man man   1. sdf |
| 3 | **Research Methodology** | Experiment all commands in linux terminal |
| 4 | **Skills Required** | Shell Scripting |
| 5 | **Expected output** | linux commands implementation |

|  |  |  |
| --- | --- | --- |
| **Program Name: Any Subdivisions for Shell Scripting** | | |
| 1 | **Background & Use-case** | Shell scripting |
| 2 | **Problem Statement** | To find a number is prime or not |
| 3 | **Research Methodology** | #!/bin/bash  echo -e "Enter Number : \c"  read n  for((i=2; i<=$n/2; i++))  do  ans=$(( n%i ))  if [ $ ans -eq 0 ]  then  echo "$n is not a prime number."  exit 0  fi  done  echo "$n is a prime number." |
| 4 | **Skills Required** | scripting in linux |
| 5 | **Expected output** | Enter the number 3  3 is a prime number |