**SJF PREEMPTIVE:**

class Process:

    def \_\_init\_\_(self,pid,at,bt):

        self.pid = pid

        self.at = at

        self.bt = bt

        self.remaining\_time = bt

        self.start = None

        self.exit = None

        self.tat = None

        self.wt = None

    def turnaround(self,exit):

        self.exit = exit

        self.tat = self.exit - self.at

        return self.tat

    def wait(self):

        self.wt = self.tat - self.bt

        return self.wt

def srtf(processes):

    processes = sorted(processes, key=lambda p:p.at)

    queue = []

    completed = []

    current = 0

    table = []

    while processes or queue:

        if processes != [] and processes[0].at <= current:

            queue.append(processes.pop(0))

        if queue != []:

            queue = sorted(queue, key=lambda p: p.remaining\_time)

            process = queue.pop(0)

            process.start = current

            process.remaining\_time -= 1

            if process.remaining\_time == 0:

                process.exit = current + 1

                process.tat = process.turnaround(process.exit)

                process.wt = process.wait()

                completed.append((process.pid, process.start, process.exit, process.tat, process.wt))

                table.append((process.pid,process.at,process.bt,process.tat,process.wt))

            else:

                if completed == []:

                    completed.append((process.pid, process.start,current + 1, None, None))

                elif completed[-1][0] != process.pid:

                    completed.append((process.pid, process.start,current + 1, None, None))

                queue.append(process)

        current += 1

    return (completed, table)

if \_\_name\_\_ == "\_\_main\_\_":

    process\_list = [

            Process(1,0,2),

            Process(2,3,5),

            Process(3,4,7),

            Process(4,4,6)]

    completed\_seq, table\_seq = srtf(process\_list)

    wt = 0

    print("PID  AT  BT  TAT WT")

    for p in table\_seq:

        print(f"{p[0]}\t{p[1]}\t{p[2]}\t{p[3]}\t{p[4]}")

        wt += p[4]

    awt = wt/4

    print("Average: ", awt)

    for p in range(len(completed\_seq)):

        if p != 0:

            if completed\_seq[p][0] == completed\_seq[p-1][0]:

                print(f" {completed\_seq[p][1]} {completed\_seq[p][0]} {completed\_seq[p][2]} ",end=" ")

            else:

                print(f"| {completed\_seq[p][1]} {completed\_seq[p][0]} {completed\_seq[p][2]} |",end=" ")

        else:

            print(f" | {completed\_seq[p][1]} {completed\_seq[p][0]} {completed\_seq[p][2]} | ",end=" ")

**PRIORITY PREEMPTIVE:**

class Process:

    def \_\_init\_\_(self,pid,at,bt,priority):

        self.pid = pid

        self.at = at

        self.bt = bt

        self.priority = priority

        self.remaining\_time = bt

        self.start = None

        self.exit = None

        self.tat = None

        self.wt = None

    def turnaround(self,exit):

        self.exit = exit

        self.tat = self.exit - self.at

        return self.tat

    def wait(self):

        self.wt = self.tat - self.bt

        return self.wt

def pp(processes):

    processes = sorted(processes, key=lambda p: p.at)

    completed = []

    queue = []

    current = 0

    table = []

    while processes or queue:

        if processes != [] and processes[0].at <= current:

            temp = []

            for p in range(len(processes)):

                if processes[p].at <= current:

                    temp.append(processes[p])

            temp = sorted(temp,key=lambda p: p.priority,reverse=True)

            i = processes.index(temp[0])

            processes.pop(i)

            queue.append(temp.pop(0))

        if queue != []:

            queue = sorted(queue, key = lambda p: p.priority, reverse = True)

            process = queue.pop(0)

            process.start = current

            process.remaining\_time -= 1

            if process.remaining\_time == 0:

                process.exit = current + 1

                process.tat = process.turnaround(process.exit)

                process.wt = process.wait()

                completed.append((process.pid, process.start, process.exit, process.wt, process.tat))

                table.append((process.pid, process.at, process.bt, process.tat, process.wt))

            else:

                if completed == []:

                    completed.append((process.pid, process.start, current + 1, None, None))

                elif completed[-1][0] != process.pid:

                    completed.append((process.pid, process.start, current + 1, None, None))

                queue.append(process)

        current += 1

    return (completed,table)

if \_\_name\_\_ == "\_\_main\_\_":

    n = 4

    process\_list = [

            Process(1,0,2,3),

            Process(2,3,5,1),

            Process(3,4,7,4),

            Process(4,4,6,2)]

    process\_seq, table\_seq = pp(process\_list)

    wt = 0

    print("PID  AT  BT  TAT WT")

    for p in table\_seq:

        print(f"{p[0]}\t{p[1]}\t{p[2]}\t{p[3]}\t{p[4]}")

        wt += p[4]

    avg\_wt = wt/4

    print("Average: ", avg\_wt)

    gannt\_chart = ""

    for p in range(len(process\_seq)):

        if p != 0:

            if process\_seq[p][0] == process\_seq[p-1][0]:

                gannt\_chart += f" {process\_seq[p][1]} {process\_seq[p][0]} {process\_seq[p][2]} |"

            else:

                gannt\_chart += f" {process\_seq[p][1]} {process\_seq[p][0]} {process\_seq[p][2]} | "

        else:

            gannt\_chart += f"| {process\_seq[p][1]} {process\_seq[p][0]} {process\_seq[p][2]} |"

    print(gannt\_chart)

**ROUND ROBIN:**

class Process:

    def \_\_init\_\_(self,pid,at,bt):

        self.pid = pid

        self.at = at

        self.bt = bt

        self.arrival = at

        self.remaining\_time = bt

        self.start = None

        self.exit = None

        self.tat = None

        self.wt = None

    def turnaround(self,exit):

        self.exit = exit

        self.tat = self.exit - self.arrival

        return self.tat

    def wait(self):

        self.wt = self.tat - self.bt

        return self.wt

def srtf(processes,time\_quant):

    processes = sorted(processes, key=lambda p: p.at)

    completed = []

    queue = []

    current = 0

    table = []

    while processes != [] or queue != []:

        flag = False

        if processes != [] and processes[0].at <= current:

            queue.append(processes.pop(0))

        if queue != []:

            queue = sorted(queue, key = lambda p: p.at)

            process = queue.pop(0)

            process.start = current

            if process.remaining\_time <= time\_quant:

                flag = True

                current += process.remaining\_time

                process.remaining\_time = 0

                process.exit = current

                process.tat = process.turnaround(process.exit)

                process.wt = process.wait()

                completed.append((process.pid, process.start, process.exit, process.wt, process.tat))

                table.append((process.pid, process.at, process.bt, process.tat, process.wt))

            else:

                flag = True

                current += time\_quant

                process.remaining\_time -= time\_quant

                completed.append((process.pid, process.start, current, None, None))

                process.at = current

                queue.append(process)

        if not flag :

            current += 1

    return (completed,table)

if \_\_name\_\_ == "\_\_main\_\_":

    n = 4

    process\_list = [

            Process(1,0,2),

            Process(2,3,5),

            Process(3,4,7),

            Process(4,4,6)]

    process\_seq, table\_seq = srtf(process\_list,2)

    wt = 0

    print("PID  AT  BT  TAT WT")

    for p in table\_seq:

        print(f"{p[0]}\t{p[1]}\t{p[2]}\t{p[3]}\t{p[4]}")

        wt += p[4]

    avg\_wt = wt/4

    print("Average: ", avg\_wt)

    gannt\_chart = ""

    for p in range(len(process\_seq)):

        if p != 0:

            if process\_seq[p][0] == process\_seq[p-1][0]:

                gannt\_chart += f" {process\_seq[p][1]} {process\_seq[p][0]} {process\_seq[p][2]} |"

            else:

                gannt\_chart += f" {process\_seq[p][1]} {process\_seq[p][0]} {process\_seq[p][2]} |"

        else:

            gannt\_chart += f"| {process\_seq[p][1]} {process\_seq[p][0]} {process\_seq[p][2]} |"

    print(gannt\_chart)

Ex no. 13

Date :

**Implementing Readers - Writers Problem**

**Problem Statement:**

Implement Reader-Writer problem using multi threading concept in Python.

**Problem Description:**

The Reader-Writer problem involves multiple threads trying to access a shared resource concurrently. Readers can access the resource simultaneously without conflict, while only one writer at a time is allowed access. When a writer is modifying the resource, no other readers or writers can access it. This problem aims to maintain data consistency and ensure exclusive access for writers while allowing non-conflicting access for readers. The solution typically employs synchronization mechanisms to achieve this concurrency control.

**Algorithm:**

1. Import the required threading module and set up global variables for shared data (x), readers count, a write lock, a read lock, and a condition variable.
2. Define a Reader function that takes a reader\_num argument:

a. Acquire the condition variable.

b. Wait while there's an active writer (readers\_count == -1).

c. Increment readers\_count to indicate an active reader.

d. Print that the reader is reading.

e. Print the shared data (x).

f. Decrement readers\_count and notify waiting writers if no more readers.

g. Release the condition variable.

1. Define a Writer function that takes a writer\_num argument:

a. Acquire the condition variable.

b. Wait while there are active readers (readers\_count > 0).

c. Set readers\_count to -1 to indicate an active writer.

d. Print that the writer is writing and update the shared data (x).

e. Reset readers\_count and notify waiting readers.

f. Release the condition variable.

1. In the main block:

a. Create lists of reader threads and writer threads.

b. Shuffle the order of threads to randomize execution.

c. Start all threads.

d. Wait for all threads to complete using thread.join().

**Code:**

import threading

import random

global x  # Shared Data

x = 0

readers\_count = 0

write\_lock = threading.Lock()

read\_lock = threading.Lock()

printcondition = threading.Condition()

def Reader(reader\_num):

    global x, readers\_count

    with condition:

        while readers\_count == -1:

            condition.wait()

            print(f'Reader {reader\_num} is waiting')

        readers\_count += 1

        print(f"Reader {reader\_num} is Reading!")

    print('Shared Data:', x)

    print()

    with condition:

        readers\_count -= 1

        if readers\_count == 0:

            condition.notify()  # Notify waiting writers

            print("No readers are reading")

    print()

def Writer(writer\_num):

    global x

    global readers\_count

    with condition:

        while readers\_count > 0:

            condition.wait()

            print(f'Writer {writer\_num} is waiting')

        readers\_count = -1  # Mark that a writer is active

        print(f'Writer {writer\_num} is Writing!')

    x += 1  # Write on the shared memory

    with condition:

        readers\_count = 0

        condition.notify()  # Notify waiting readers

    print(f'Writer {writer\_num} is Releasing the lock!')

    print()

if \_\_name\_\_ == '\_\_main\_\_':

    reader\_threads = [threading.Thread(target=Reader, args=(i+1,)) for i in range(5)]

    writer\_threads = [threading.Thread(target=Writer, args=(i+1,)) for i in range(5)]

    all\_threads = reader\_threads + writer\_threads

    random.shuffle(all\_threads)

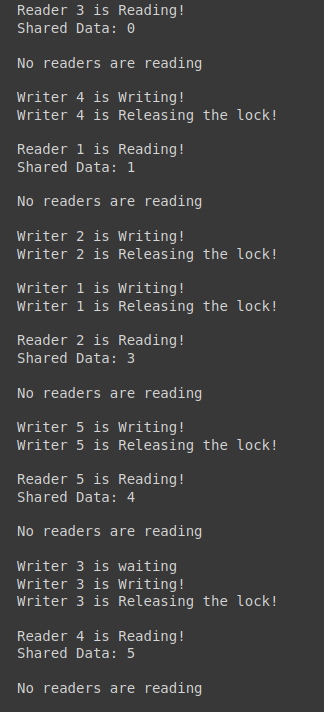
    for thread in all\_threads:

        thread.start()

    for thread in all\_threads:

        thread.join()

**Output:**



**Ex. No: 14**

**Date:**

**BANKER’S ALGORITHM**

**Problem Statement:**

To design and implement the Banker's Algorithm to ensure safe resource allocation in a multi-process environment in python.

**Problem Description:**

The Banker's Algorithm is a fundamental component of operating systems that helps prevent deadlock in multi-process systems. Deadlock occurs when processes are waiting for resources that will never be available, leading to system paralysis. The Banker's Algorithm works to avoid deadlock by ensuring that resource requests are made safely, and it monitors the system's state to grant resources in a manner that does not lead to deadlock.

**Algorithm:**

1. Initialize the maximum need matrix, allocation matrix, and available matrix.
2. When a process requests resources, check if the requested resources can be safely allocated. If the request can be granted without causing an unsafe state, allocate the resources; otherwise, make the process wait.
3. When a process releases resources, update the allocation matrix and the available matrix.
4. Periodically check the system's state to determine if it's in a safe or unsafe state. If the system is in a safe state, allocate resources to the waiting processes in a way that maintains safety. If the system is in an unsafe state, make the processes wait until the system becomes safe.
5. Continue the above steps until all processes have completed their execution.
6. Display the safe sequence.

**Code:**

m = int(input("Enter resource number : "))

n = int(input("Enter number of processes : "))

alloc = eval(input())

#input 1

#[[0,0,1,2],[1,0,0,0],[1,3,5,4],[0,6,3,2],[0,0,1,4]]

#[[0,0,1,2],[1,7,5,0],[2,3,5,6],[0,6,5,2],[0,6,5,6]]

#input 2

#[[2,3,1,4],[1,2,4,4],[3,2,4,7],[2,2,4,5],[6,4,4,5]]

#[[3,8,3,5],[6,10,7,8],[7,9,9,10],[5,12,8,10],[9,10,6,10]]

max = eval(input())

#[[7, 5, 3 ],[3, 2, 2 ],[ 9, 0, 2 ],[2, 2, 2],[4, 3, 3]]

avail = eval(input())

# [1,5,2,0]

# [2,6,3,2]

f = [0]\*n

ans = [0]\*n

ind = 0

for k in range(n):

f[k] = 0

need = [[ 0 for i in range(m)]for i in range(n)]

for i in range(n):

for j in range(m):

need[i][j] = max[i][j] - alloc[i][j]

y = 0

for k in range(5):

for i in range(n):

if (f[i] == 0):

flag = 0

for j in range(m):

if (need[i][j] > avail[j]):

flag = 1

break

if (flag == 0):

ans[ind] = i

ind += 1

for y in range(m):

avail[y] += alloc[i][y]

f[i] = 1

print("The following is the need matrix")

for i in need:

print(i)

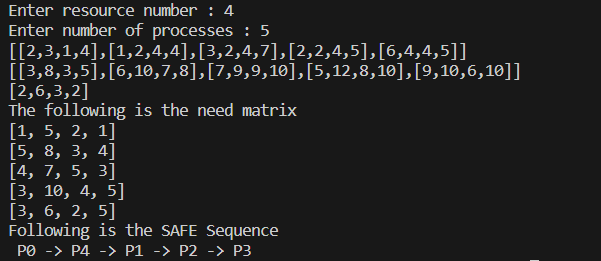
print("Following is the SAFE Sequence")

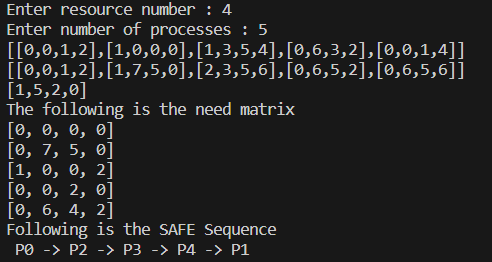
for i in range(n - 1):

print(" P", ans[i], " ->", sep="", end="")

print(" P", ans[n - 1], sep="")

Output :





**Result**

Banker’s Algorithm has been successfully implemented in python and the results have been verified.

**15. MEMORY ALLOCATION:**

def firstfit(wholes, w, process, p):

    allocation = [(-1,-1)] \* p

    for i in range(p):

        for j in range(w):

            if wholes[j] >= process[i]:

                allocation[i] = (j, wholes[j])

                wholes[j] -= process[i]

                break

    print("Process", "  ", "Process Size"," ", "Block","    ","Allocated in block with memory size")

    for i in range(p):

        print(" ",i+1,"      ",process[i],end="       ")

        if allocation[i][0] != -1:

            print(allocation[i][0] + 1, "       ",allocation[i][1])

        else:

            print("Not allocated")

def bestfit(wholes, w, process, p):

    allocation = [(-1,-1)] \* p

    for i in range(p):

        bestind = -1

        for j in range(w):

            if wholes[j] >= process[i]:

                if bestind == -1:

                    bestind = j

                elif wholes[bestind] > wholes[j]:

                    bestind = j

        if bestind != -1:

            allocation[i] = (bestind, wholes[bestind])

            wholes[bestind] -= process[i]

    print("Process", "  ", "Process Size"," ", "Block","    ","Allocated in block with memory size")

    for i in range(p):

        print(" ",i+1,"      ",process[i],end="       ")

        if allocation[i][0] != -1:

            print(allocation[i][0] + 1, "       ",allocation[i][1])

        else:

            print("Not allocated")

def worstfit(wholes, w, process, p):

    allocation = [(-1,-1)] \* p

    for i in range(p):

        worstind = -1

        for j in range(w):

            if wholes[j] >= process[i]:

                if worstind == -1:

                    worstind = j

                elif wholes[worstind] < wholes[j]:

                    worstind = j

        if worstind != -1:

            allocation[i] = (worstind, wholes[worstind])

            wholes[worstind] -= process[i]

    print("Process", "  ", "Process Size"," ", "Block","    ","Allocated in block with memory size")

    for i in range(p):

        print(" ",i+1,"      ",process[i],end="       ")

        if allocation[i][0] != -1:

            print(allocation[i][0] + 1, "       ",allocation[i][1])

        else:

            print("Not allocated")

if \_\_name\_\_ == "\_\_main\_\_":

    #w = int(input("Enter no.of wholes: "))

    #wholes = []

    #for i in range(w):

    #   wholes.append(int(input("Enter memory size: "))

    #p = int(input("Enter no.of processes: "))

    #process = []

    #for i in range(p):

    #   process.append(int(input("Enter process size: "))

    wholes = [100,500,200,300,600]

    process = [212,417,112,426]

    w = len(wholes)

    p = len(process)

    print("F - First fit\nB - Best fit\nW - Worst fit\n")

    c = input("Enter memory allocation type (F/B/W): ")

    if c == "F":

        firstfit(wholes, w, process, p)

    elif c == "B":

        bestfit(wholes, w, process, p)

    elif c == "W":

        worstfit(wholes, w, process, p)

**17. PAGE REPLACEMENT ALGORITHM**

**FIRST IN FIRST OUT:**

from queue import Queue

def pageFaults(pages, n, capacity):

    s = set()

    indexes = Queue()

    page\_faults = 0

    page\_hits = 0

    for i in range(n):

        if (len(s) < capacity):

            if (pages[i] not in s):

                s.add(pages[i])

                page\_faults += 1

                indexes.put(pages[i])

            else:

                page\_hits += 1

        else:

            if (pages[i] not in s):

                val = indexes.queue[0]

                indexes.get()

                s.remove(val)

                s.add(pages[i])

                indexes.put(pages[i])

                page\_faults += 1

            else:

                page\_hits += 1

    return [page\_faults , page\_hits]

if \_\_name\_\_ == '\_\_main\_\_':

    pages= []

    no\_of\_page = int(input("ENTER THE NUMBER OF PAGES : "))

    for i in range(no\_of\_page):

        a = int(input("ENTER THE REFERENCE STRING OF A PAGE : "))

        pages.append(a)

    print("REFRENCE STRING : ",pages)

    n = len(pages)

    capacity = int(input("ENTER THE NUMBER OF FRAMES : "))

    a = pageFaults(pages, n, capacity)

    print("PAGE HITS : ", a[1])

    print("PAGE FAULT : ", a[0])

    print(f"THE RATIO OF PAGE HIT TO THE PAGE FAULT IS {a[1]}:{a[0]} : ", a[1]/a[0])

    print("THE PAGE HIT PECENTAGE IS : ", a[0] \* 100 / n)

    print("THE PAGE FAULT PERCENTAGE IS : ", a[1] \* 100 / n)

**b) LRU (Least Recently Used Algorithm)**

**Problem Description:**

The problem is to develop a page replacement algorithm, specifically the Least Recently Used (LRU) algorithm, to efficiently manage a fixed-size memory (cache) in a computer system. The system operates with virtual memory, and when a page is needed that is not currently in the cache, it results in a page fault. The goal is to minimize the number of page faults by determining which page to evict from the cache when it is full.

**Code:**

capacity = int(input("Enter the capacity:"))

'''

n = int(input('Size:'))

processList = []

for i in range(n):

    val = int(input("Num:"))

    processList.append(val)

'''

processList = [7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1]

s = []

pageFaults = 0

for i in processList:

    if i not in s:

        if(len(s) == capacity):

            s.remove(s[0])

            s.append(i)

        else:

            s.append(i)

        pageFaults +=1

    else:

        s.remove(i)

        s.append(i)

    for j in s:

        print(j,end=" ")

    print()

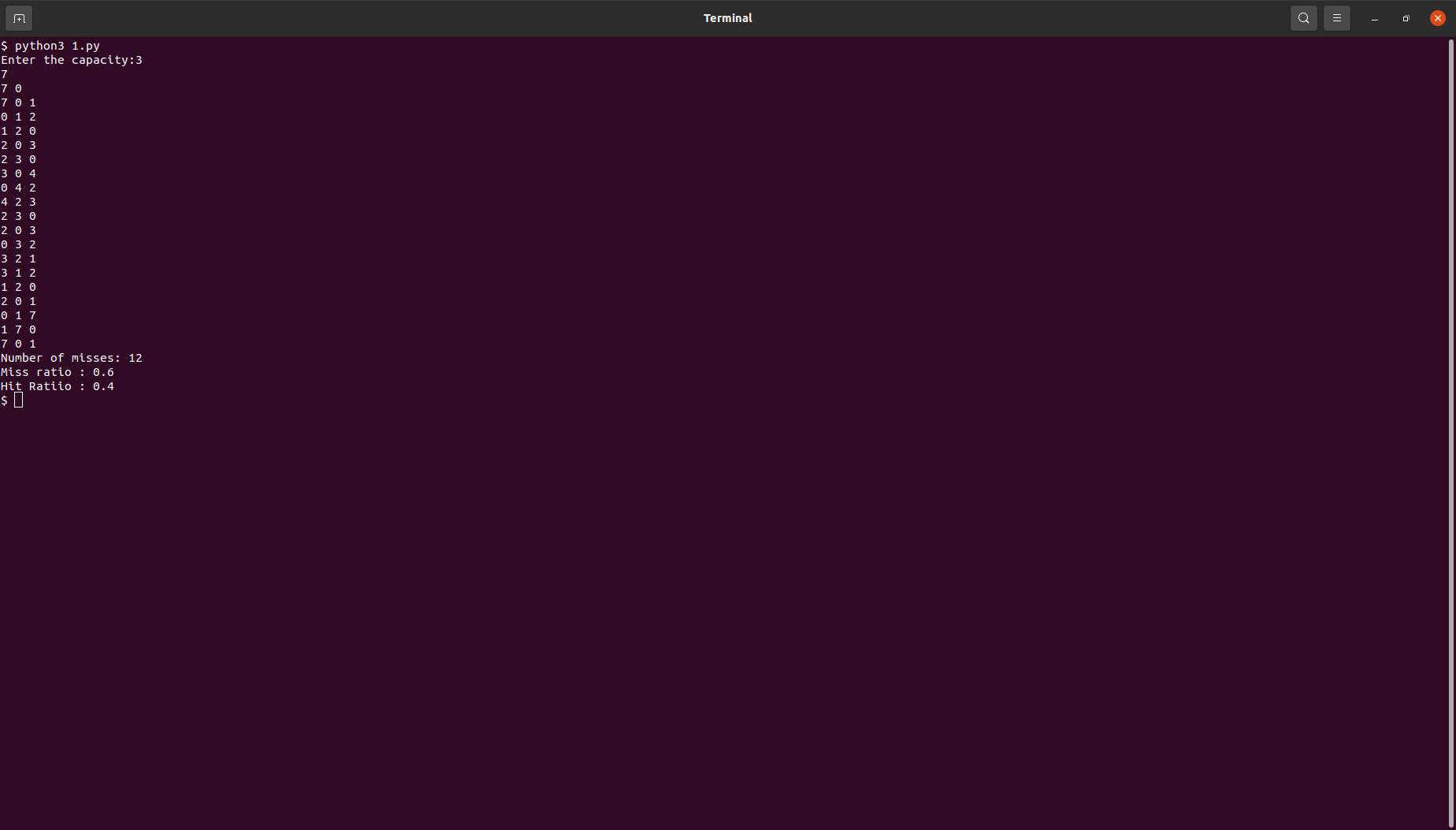
ratio = pageFaults/(len(processList))

print('Number of misses:', pageFaults)

print('Miss ratio :',ratio)

print('Hit Ratiio :', 1-ratio)

**Output:**

****

**(C) OPTIMAL PAGE REPLACEMENT**

**Problem description:**

Optimal Page Replacement aims to minimize page faults in a memory management system. Given a memory capacity and a sequence of page references, the algorithm maintains a set of pages currently in memory. When a page reference is encountered:

a. If the page is already in memory, no action is taken.

b. If the page is not in memory:

i. If the memory set is not full, the page is added, and a page fault is recorded.

ii. If the memory set is full, the algorithm replaces the page that will be accessed furthest in the future, minimizing page faults.

The algorithm requires knowledge of future page accesses, which is often not practical in real-time scenarios.

def exists\_in\_frame(key, frame):

    for i in range(len(frame)):

        if frame[i] == key:

            return True

    return False

def find\_least\_recently\_used(pg, frame, pn, index):

    result = -1

    farthest = index

    for i in range(len(frame)):

        j = 0

        for j in range(index, pn):

            if frame[i] == pg[j]:

                if j > farthest:

                    farthest = j

                    result = i

                break

        if j == pn - 1:

            return i

    return 0 if result == -1 else result

def optimal\_page(pg, pn, frame\_count):

    frames = []

    hits = 0

    for i in range(pn):

        if exists\_in\_frame(pg[i], frames):

            hits += 1

            continue

        if len(frames) < frame\_count:

            frames.append(pg[i])

        else:

            j = find\_least\_recently\_used(pg, frames, pn, i + 1)

            frames[j] = pg[i]

    misses = pn - hits

    hit\_ratio = (hits / pn) \* 100

    miss\_ratio = (misses / pn) \* 100

    print("No. of hits =", hits)

    print("No. of misses =", misses)

    print("Hit Ratio =", "{:.2f}%".format(hit\_ratio))

    print("Miss Ratio =", "{:.2f}%".format(miss\_ratio))

# Driver Code

page\_references = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3]

page\_count = len(page\_references)

frame\_number = 4

optimal\_page(page\_references, page\_count, frame\_number)

A screenshot of a computer program

Description automatically generated

**SSTF**

def calculateDifference(queue, head, diff):

    for i in range(len(diff)):

        diff[i][0] = abs(queue[i] - head)

def findMin(diff):

    index = -1

    minimum = 999999999

    for i in range(len(diff)):

        if (not diff[i][1] and

                minimum > diff[i][0]):

            minimum = diff[i][0]

            index = i

    return index

def shortestSeekTimeFirst(request, head):

        if (len(request) == 0):

            return

        l = len(request)

        diff = [0] \* l

        for i in range(l):

            diff[i] = [0, 0]

        seek\_count = 0

        seek\_sequence = [0] \* (l + 1)

        for i in range(l):

            seek\_sequence[i] = head

            calculateDifference(request, head, diff)

            index = findMin(diff)

            diff[index][1] = True

            seek\_count += diff[index][0]

            head = request[index]

        seek\_sequence[len(seek\_sequence) - 1] = head

        print("Total number of seek operations =", seek\_count)

                                                                                          print("Seek Sequence is")

        for i in range(l + 1):

            print(seek\_sequence[i])

if \_\_name\_\_ =="\_\_main\_\_":

    proc = [176, 79, 34, 60, 92, 11, 41, 114]

    shortestSeekTimeFirst(proc, 50)