Experiment No: 1

Experiment Name: Basics of UNIX commands and Implementation of Shell Programming.

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

A process is a program under execution, while a thread is the lightweight unit of a process. Each process has states: New, Ready, Running, Waiting, and Terminated. Threads share the same memory space of a process but can execute independently, improving efficiency. Process creation in UNIX/Linux is typically done with `fork()` and terminated with `exit()`. Thread creation is done using `pthread_create()` and terminated using `pthread_exit()`. Understanding process and thread lifecycles is crucial for multitasking and concurrent execution.

Code:

```
import os

while True:

    # Take input from user

    command = input("my_shell> ")

    # Exit condition

    if command.lower() == "exit":

        print("Exiting shell...")

        break

    # Execute the command

    os.system(command)
```

Sample UNIX Commands to Try:

```
ls
pwd
mkdir test
cd test
rmdir test
cat filename.txt
```

Output:

```
my_shell>
```

Discussion:
This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 2

Experiment Name: Implementation of Process and thread (Life cycle of process): (i) Process creation and Termination; (ii) Thread creation and Termination

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

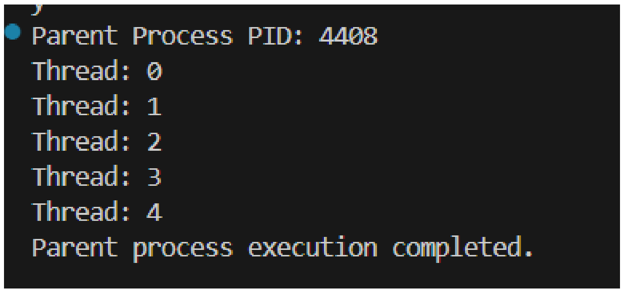
A process is a program under execution, while a thread is the lightweight unit of a process. Each process has states: New, Ready, Running, Waiting, and Terminated. Threads share the same memory space of a process but can execute independently, improving efficiency. Process creation in UNIX/Linux is typically done with `fork()` and terminated with `exit()`. Thread creation is done using `pthread_create()` and terminated using `pthread_exit()`. Understanding process and thread lifecycles is crucial for multitasking and concurrent execution.

Code:

```
import os

import threading


# ---------------- Thread Function ----------------

def print_numbers():

    for i in range(5):

        print("Thread:", i)


# ---------------- Process Creation ----------------

print("Parent Process PID:", os.getpid())


# On Unix/Linux, fork() creates a child process

pid = os.fork() if hasattr(os, "fork") else None
```

```
if pid == 0:
    # This block runs in the child process




print("Child Process PID:", os.getpid())
    print("Child process executing task...")
    os._exit(0)  # Child process terminates


# ---------------- Thread Creation ----------------
t = threading.Thread(target=print_numbers)
t.start()  # Start the thread
t.join()   # Wait for thread to finish


print("Parent process execution completed.")
```

Output:



```
Parent Process PID: 4408
Thread: 0
Thread: 1
Thread: 2
Thread: 3
Thread: 4
Parent process execution completed.
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 3

Experiment Name: Implementation of CPU Scheduling Algorithms. (i) FCFS, (ii) SJF

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

CPU Scheduling is the process of determining which process will use the CPU when multiple processes are ready.

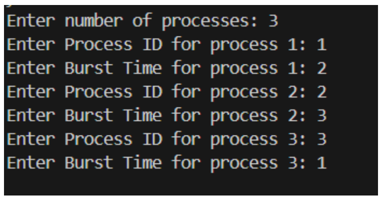
FCFS (First Come First Serve): Non-preemptive, processes are scheduled in the order they arrive in the ready queue. Simple but may cause the convoy effect**.**

SJF (Shortest Job First): Selects the process with the shortest burst time. Optimal for minimizing average waiting time but requires knowledge of burst times in advance.

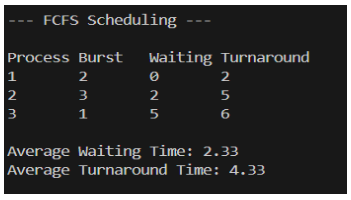
Code:

```
# CPU Scheduling: FCFS and SJF with user input



def fcfs(processes):

  n = len(processes)

  wt = [0] * n

  tat = [0] * n

  for i in range(1, n):

    wt[i] = processes[i-1][1] + wt[i-1]

  for i in range(n):

    tat[i] = processes[i][1] + wt[i]

  print("\nProcess\tBurst\tWaiting\tTurnaround")

  for i in range(n):

    print(f"{processes[i][0]}\t{processes[i][1]}\t{wt[i]}\t{tat[i]}")
    print(f"\nAverage Waiting Time: {sum(wt)/n:.2f}")
    print(f"Average Turnaround Time: {sum(tat)/n:.2f}")

def sjf(processes):
    processes.sort(key=lambda x: x[1])  # Sort by burst time
    print("\nSJF Scheduling Order:")
    fcfs(processes)

# User input
n = int(input("Enter number of processes: "))
processes = []

for i in range(n):
    pid = int(input(f"Enter Process ID for process {i+1}: "))
    bt = int(input(f"Enter Burst Time for process {pid}: "))
    processes.append((pid, bt))

print("\n--- FCFS Scheduling ---")
fcfs(processes)
print("\n--- SJF Scheduling ---")
sjf(processes)
```

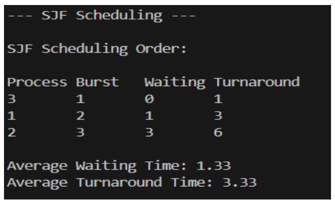
Output:



```
Enter number of processes: 3
Enter Process ID for process 1: 1
Enter Burst Time for process 1: 2
Enter Process ID for process 2: 2
Enter Burst Time for process 2: 3
Enter Process ID for process 3: 3
Enter Burst Time for process 3: 1
```



```
--- FCFS Scheduling ---

Process Burst   Waiting Turnaround
1       2       0       2
2       3       2       5
3       1       5       6

Average Waiting Time: 2.33
Average Turnaround Time: 4.33
```



```
--- SJF Scheduling ---

SJF Scheduling Order:

Process Burst   Waiting Turnaround
3       1       0       1
1       2       1       3
2       3       3       6

Average Waiting Time: 1.33
Average Turnaround Time: 3.33
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 4

Experiment Name: Implementation of CPU Scheduling Algorithms. (i) Shortest Remaining Time First and (ii) Priority based

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

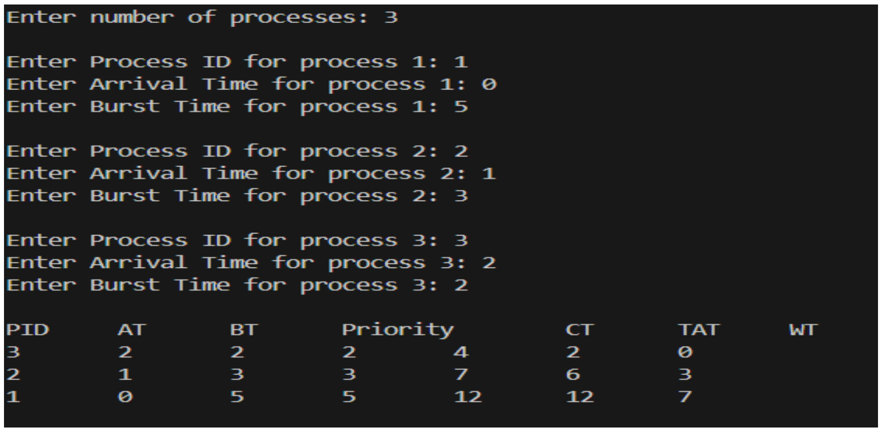
SRTF (Shortest Remaining Time First): Preemptive version of SJF where the CPU is allocated to the process with the shortest remaining burst time. Provides better average turnaround but may cause starvation.

Priority Scheduling: Each process is assigned a priority, and the CPU is allocated to the highest-priority process. Can be preemptive or non-preemptive. Risk of starvation, which can be solved using aging.

Code:

```
# ---------- Priority Scheduling (Auto Priority: lower BT = higher priority) ----------

def priority_scheduling(processes):

  # Priority auto assign: Priority = Burst Time

  processes = [(pid, at, bt, bt) for pid, at, bt in processes]


  # Sort by priority (lower burst → higher priority), then arrival time

  processes.sort(key=lambda x: (x[3], x[1]))


  time = 0

  results = []


  for pid, at, bt, pr in processes:

    if time < at:
            time = at
        ct = time + bt
        tat = ct - at
        wt = tat - bt
        results.append((pid, at, bt, pr, ct, tat, wt))
        time = ct

    # Print Table
    print("\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT")
    for r in results:
        print("\t".join(str(x) for x in r))



# ---------------- User Input ----------------
n = int(input("Enter number of processes: "))
processes = []

for i in range(n):
    pid = int(input(f"\nEnter Process ID for process {i+1}: "))
    at = int(input(f"Enter Arrival Time for process {pid}: "))
    bt = int(input(f"Enter Burst Time for process {pid}: "))
    processes.append((pid, at, bt))

priority_scheduling(processes)
```

Output:


```
Enter number of processes: 3

Enter Process ID for process 1: 1
Enter Arrival Time for process 1: 0
Enter Burst Time for process 1: 5

Enter Process ID for process 2: 2
Enter Arrival Time for process 2: 1
Enter Burst Time for process 2: 3

Enter Process ID for process 3: 3
Enter Arrival Time for process 3: 2
Enter Burst Time for process 3: 2

PID     AT      BT      Priority        CT      TAT     WT
3       2       2       2       4       2       0
2       1       3       3       7       6       3
1       0       5       5       12      12      7
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 5

Experiment Name: Implementation of Round Robin CPU Scheduling Algorithm.

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

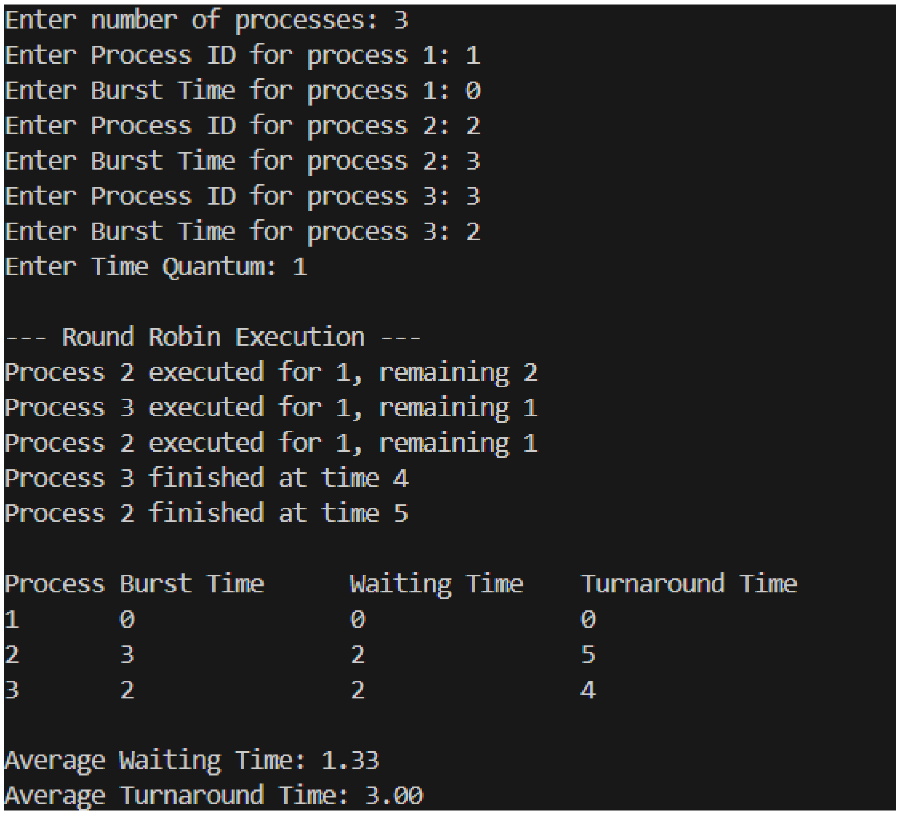
Round Robin (RR) is a preemptive scheduling algorithm designed for time-sharing systems. Each process gets a fixed time quantum. If a process does not finish within that time, it is preempted and placed at the end of the ready queue. This ensures fairness but performance depends on the time quantum.

Code:

```
# Round Robin Scheduling with WT and TAT

def round_robin(processes, quantum):

    n = len(processes)

    rem_bt = [bt for _, bt in processes]  # Remaining burst times

    t = 0  # Current time

    wt = [0] * n  # Waiting time

    tat = [0] * n  # Turnaround time


    print("\n--- Round Robin Execution ---")

    while True:

        done = True

        for i, (pid, bt) in enumerate(processes):

            if rem_bt[i] > 0:

                done = False

                if rem_bt[i] > quantum:

                    t += quantum

                    rem_bt[i] -= quantum
                print(f"Process {pid} executed for {quantum}, remaining {rem_bt[i]}")
            else:
                t += rem_bt[i]
                wt[i] = t - bt  # Waiting time = finish time - burst time
                tat[i] = t      # Turnaround time = finish time
                print(f"Process {pid} finished at time {t}")
                rem_bt[i] = 0
        if done:
            break

    # Print the table of WT and TAT
    print("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time")
    total_wt = total_tat = 0
    for i, (pid, bt) in enumerate(processes):
        print(f"{pid}\t{bt}\t\t{wt[i]}\t\t{tat[i]}")
        total_wt += wt[i]
        total_tat += tat[i]

    print(f"\nAverage Waiting Time: {total_wt/n:.2f}")
    print(f"Average Turnaround Time: {total_tat/n:.2f}")



# User input
n = int(input("Enter number of processes: "))
processes = []
```

```
for i in range(n):
    pid = int(input(f"Enter Process ID for process {i+1}: "))
    bt = int(input(f"Enter Burst Time for process {pid}: "))
    processes.append((pid, bt))


quantum = int(input("Enter Time Quantum: "))


round_robin(processes, quantum)
```

Output:



```
Enter number of processes: 3
Enter Process ID for process 1: 1
Enter Burst Time for process 1: 0
Enter Process ID for process 2: 2
Enter Burst Time for process 2: 3
Enter Process ID for process 3: 3
Enter Burst Time for process 3: 2
Enter Time Quantum: 1

--- Round Robin Execution ---
Process 2 executed for 1, remaining 2
Process 3 executed for 1, remaining 1
Process 2 executed for 1, remaining 1
Process 3 finished at time 4
Process 2 finished at time 5

Process Burst Time      Waiting Time    Turnaround Time
1       0               0               0
2       3               2               5
3       2               2               4

Average Waiting Time: 1.33
Average Turnaround Time: 3.00
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 6

Experiment Name: Producer-Consumer Problem using Semaphores and Reader Writer Problem

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

Producer-Consumer Problem: A classic synchronization problem where a producer generates data and a consumer uses it. The buffer is shared, so synchronization is needed to avoid overflow (producer produces too much) and underflow (consumer consumes empty buffer). Semaphores solve this problem by controlling access to the buffer.

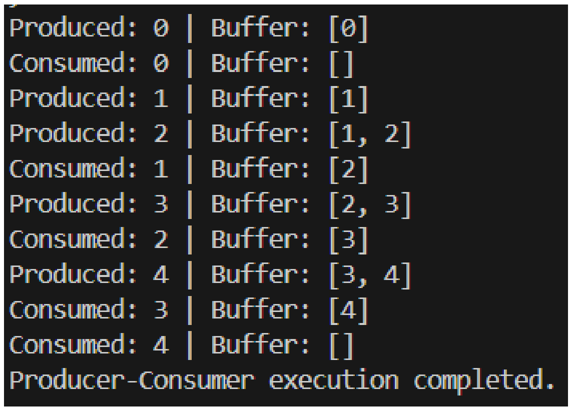
Reader-Writer Problem: Multiple readers can read simultaneously, but when a writer writes, no other process should access the shared data. Semaphores ensure mutual exclusion and synchronization.

Code:

```
import threading

import time

from threading import Semaphore, Lock


buffer = []

BUFFER_SIZE = 5


empty = Semaphore(BUFFER_SIZE)

full = Semaphore(0)

mutex = Lock()


def producer():
    for i in range(5):
        empty.acquire()
        mutex.acquire()
        buffer.append(i)
        print(f"Produced: {i} | Buffer: {buffer}")
        mutex.release()
        full.release()
        time.sleep(1)

def consumer():
    for i in range(5):
        full.acquire()
        mutex.acquire()
        item = buffer.pop(0)
        print(f"Consumed: {item} | Buffer: {buffer}")
        mutex.release()
        empty.release()
        time.sleep(2)

t1 = threading.Thread(target=producer)
t2 = threading.Thread(target=consumer)

t1.start()
t2.start()
t1.join()
t2.join()
```

print("Producer-Consumer execution completed.")

Output:



```
Produced: 0 | Buffer: [0]
Consumed: 0 | Buffer: []
Produced: 1 | Buffer: [1]
Produced: 2 | Buffer: [1, 2]
Consumed: 1 | Buffer: [2]
Produced: 3 | Buffer: [2, 3]
Consumed: 2 | Buffer: [3]
Produced: 4 | Buffer: [3, 4]
Consumed: 3 | Buffer: [4]
Consumed: 4 | Buffer: []
Producer-Consumer execution completed.
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 7

Experiment Name: Simulate algorithm for deadlock prevention and detection

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

Deadlock is a state where processes wait indefinitely for resources held by each other.

Deadlock Prevention: Ensures at least one of the necessary conditions for deadlock (Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait) does not occur.

Deadlock Detection: Allows deadlock to occur but provides algorithms to detect it (e.g., resource allocation graphs, wait-for graphs). After detection, recovery techniques like process termination or resource preemption are used.

Code:

```
# Deadlock Prevention Example in Python

import threading



# Create two locks

lock1 = threading.Lock()

lock2 = threading.Lock()



# Task 1 acquires locks in a consistent order

def task1():

    with lock1:

        print("Task1 acquired Lock1")

        with lock2:

            print("Task1 acquired Lock2")
        print("Task1 is executing critical section")
    print("Task1 released Lock1 and Lock2")


# Task 2 acquires locks in the same order to prevent deadlock
def task2():
    with lock1:  # Notice same order as Task1
        print("Task2 acquired Lock1")
        with lock2:
            print("Task2 acquired Lock2")
            print("Task2 is executing critical section")
    print("Task2 released Lock1 and Lock2")


# Create threads for tasks
t1 = threading.Thread(target=task1)
t2 = threading.Thread(target=task2)
# Start threads
t1.start()
t2.start()
# Wait for threads to complete
t1.join()
t2.join()
print("Deadlock prevention achieved by consistent lock ordering")
```

Output:

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 8

Experiment Name: Simulate the algorithm for deadlock avoidance and study about deadlock recovery

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

Deadlock Avoidance: Ensures that a system never enters a deadlock state by making safe resource allocation decisions. The most common method is Banker's Algorithm**,** which checks whether granting a resource request leaves the system in a safe state.

Deadlock Recovery: If deadlock occurs, recovery can be done by process termination (kill one or more processes) or resource preemption (take resources away from processes and reassign).

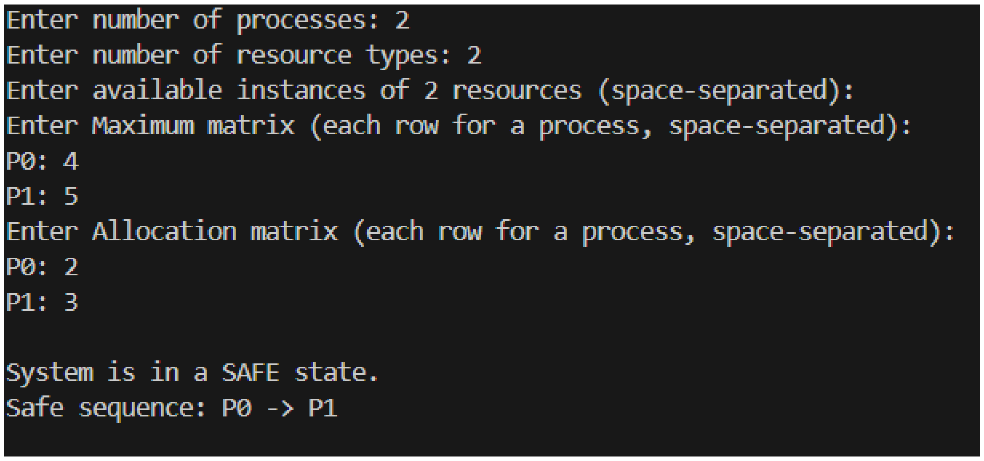
Code:

```
# Deadlock Avoidance using Banker's Algorithm



def is_safe(processes, avail, maxm, allot):

  n = len(processes)

  m = len(avail)

  # Calculate Need matrix

  need = [[maxm[i][j] - allot[i][j] for j in range(m)] for i in range(n)]

  finish = [0] * n

  safe_seq = []

  work = avail[:]


  while len(safe_seq) < n:

    for i in range(n):

      if not finish[i] and all(need[i][j] <= work[j] for j in range(m)):
```

```
                for k in range(m):
                    work[k] += allot[i][k]
                safe_seq.append(processes[i])
                finish[i] = 1
                break
        else:
            return False, []
    return True, safe_seq


# ---------------- User Input ----------------
n = int(input("Enter number of processes: "))
m = int(input("Enter number of resource types: "))


processes = [i for i in range(n)]
avail = list(map(int, input(f"Enter available instances of {m} resources (space-separated): 
").split()))


print("Enter Maximum matrix (each row for a process, space-separated):")
maxm = [list(map(int, input(f"P{i}: ").split())) for i in range(n)]


print("Enter Allocation matrix (each row for a process, space-separated):")
allot = [list(map(int, input(f"P{i}: ").split())) for i in range(n)]


# ---------------- Safety Check ----------------
safe, seq = is_safe(processes, avail, maxm, allot)
```

```
if safe:
    print("\nSystem is in a SAFE state.")
    print("Safe sequence:", " -> ".join(f"P{p}" for p in seq))
else:
    print("\nSystem is NOT in a safe state. Deadlock may occur.")
```

Output:



```
Enter number of processes: 2
Enter number of resource types: 2
Enter available instances of 2 resources (space-separated):
Enter Maximum matrix (each row for a process, space-separated):
P0: 4
P1: 5
Enter Allocation matrix (each row for a process, space-separated):
P0: 2
P1: 3

System is in a SAFE state.
Safe sequence: P0 -> P1
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 9

Experiment Name: Implementation Page replacement: (i) FIFO (ii) LRU (iii) LFU

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

Virtual memory uses pages, and when memory is full, a page replacement algorithm decides which page to remove.

FIFO (First-In-First-Out): Removes the oldest loaded page. Simple but may cause Belady's anomaly.

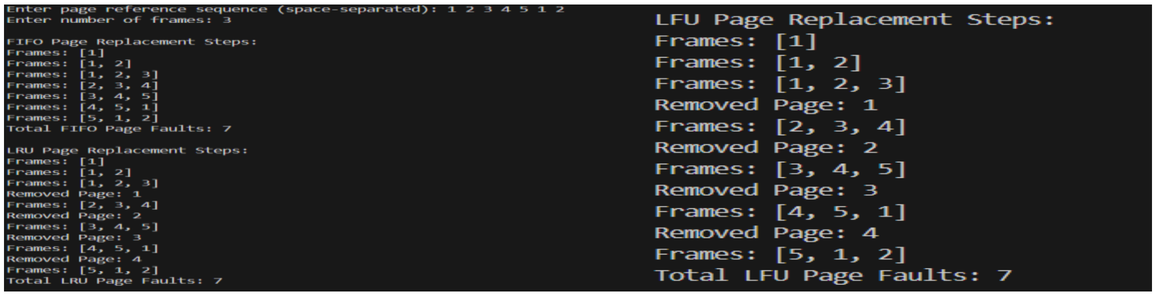
LRU (Least Recently Used): Removes the page that hasn't been used for the longest time. Good performance but requires tracking usage history.

LFU (Least Frequently Used): Removes the page with the least number of accesses. Useful when frequently accessed pages should remain in memory.

Code:

```
from collections import deque, Counter



# ---------------- FIFO Page Replacement ----------------

def fifo(pages, capacity):

   s, q = set(), deque()

   faults = 0

   print("\nFIFO Page Replacement Steps:")

   for p in pages:

      if p not in s:

         if len(s) < capacity:

            s.add(p)
            q.append(p)
        else:
            val = q.popleft()
            s.remove(val)
            s.add(p)
            q.append(p)
        faults += 1
    else:
        print(f"Page {p} hit")
    print("Frames:", list(q))
  print("Total FIFO Page Faults:", faults)


# ---------------- LRU Page Replacement ----------------
def lru(pages, capacity):
  cache, faults = [], 0
  print("\nLRU Page Replacement Steps:")
  for p in pages:
    if p not in cache:
        if len(cache) < capacity:
            cache.append(p)
        else:
            removed = cache.pop(0)
            cache.append(p)
            print(f"Removed Page: {removed}")
        faults += 1
    else:
            cache.remove(p)
            cache.append(p)
            print(f"Page {p} hit")
        print("Frames:", cache)
    print("Total LRU Page Faults:", faults)


# ---------------- LFU Page Replacement ----------------
def lfu(pages, capacity):
    cache = []
    freq = Counter()
    faults = 0
    print("\nLFU Page Replacement Steps:")
    for p in pages:
        if p in cache:
            freq[p] += 1
            print(f"Page {p} hit")
        else:
            if len(cache) < capacity:
                cache.append(p)
                freq[p] += 1
            else:
                # Find least frequently used page(s)
                min_freq = min(freq[x] for x in cache)
                candidates = [x for x in cache if freq[x] == min_freq]
                # Remove the oldest among least frequently used
                to_remove = candidates[0]
                cache.remove(to_remove)

                del freq[to_remove]

                cache.append(p)

                freq[p] += 1

                print(f"Removed Page: {to_remove}")

            faults += 1

        print("Frames:", cache)

    print("Total LFU Page Faults:", faults)


# ---------------- User Input ----------------

pages = list(map(int, input("Enter page reference sequence (space-separated): ").split()))

capacity = int(input("Enter number of frames: "))


# Run algorithms

fifo(pages, capacity)

lru(pages, capacity)

lfu(pages, capacity)
```

Output:



```
Enter page reference sequence (space-separated): 1 2 3 4 5 1 2
Enter number of frames: 3

FIFO Page Replacement Steps:
Frames: [1]
Frames: [1, 2]
Frames: [1, 2, 3]
Frames: [2, 3, 4]
Frames: [3, 4, 5]
Frames: [4, 5, 1]
Frames: [5, 1, 2]
Total FIFO Page Faults: 7

LRU Page Replacement Steps:
Frames: [1]
Frames: [1, 2]
Frames: [1, 2, 3]
Removed Page: 1
Frames: [2, 3, 4]
Removed Page: 2
Frames: [3, 4, 5]
Removed Page: 3
Frames: [4, 5, 1]
Removed Page: 4
Frames: [5, 1, 2]
Total LRU Page Faults: 7
```

```
LFU Page Replacement Steps:
Frames: [1]
Frames: [1, 2]
Frames: [1, 2, 3]
Removed Page: 1
Frames: [2, 3, 4]
Removed Page: 2
Frames: [3, 4, 5]
Removed Page: 3
Frames: [4, 5, 1]
Removed Page: 4
Frames: [5, 1, 2]
Total LFU Page Faults: 7
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness

Experiment No: 10

Experiment Name: Implementation of Disk Scheduling using FCFS, SCAN and C-SCAN algorithm

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

Disk scheduling determines the order of servicing I/O requests to reduce seek time.

FCFS (First Come First Serve): Processes requests in order of arrival. Simple but may lead to long waits.

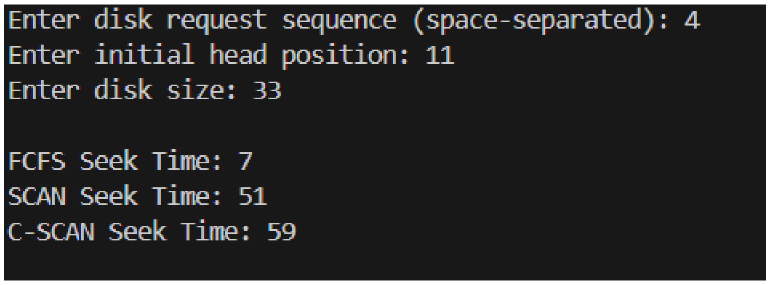
SCAN (Elevator Algorithm): The head moves in one direction servicing requests until the end, then reverses. Provides better performance than FCFS.

C-SCAN (Circular SCAN): Similar to SCAN but only services in one direction, then returns to the beginning. Provides uniform wait time.

Code:

```
# ---------------- Disk Scheduling Algorithms: FCFS, SCAN, C-SCAN ----------------


def fcfs(requests, head):

    seek = 0

    for r in requests:

        seek += abs(r - head)

        head = r

    print("\nFCFS Seek Time:", seek)


def scan(requests, head, disk_size):

    requests.sort()

    left = [r for r in requests if r < head]
    right = [r for r in requests if r >= head]
    seek = 0
    for r in right:
        seek += abs(r - head)
        head = r
    seek += abs(disk_size - head)  # Move to end
    head = disk_size
    for r in reversed(left):
        seek += abs(r - head)
        head = r
    print("SCAN Seek Time:", seek)

def cscan(requests, head, disk_size):
    requests.sort()
    right = [r for r in requests if r >= head]
    left = [r for r in requests if r < head]
    seek = 0
    for r in right:
        seek += abs(r - head)
        head = r
    seek += abs(disk_size - head)  # Move to end
    head = 0  # Jump to beginning
    seek += disk_size
    for r in left:
        seek += abs(r - head)
        head = r
    print("C-SCAN Seek Time:", seek)


# ---------------- User Input ----------------

requests = list(map(int, input("Enter disk request sequence (space-separated): ").split()))

head = int(input("Enter initial head position: "))

disk_size = int(input("Enter disk size: "))


# Run algorithms

fcfs(requests, head)

scan(requests, head, disk_size)

cscan(requests, head, disk_size)
```

Output:



```
Enter disk request sequence (space-separated): 4
Enter initial head position: 11
Enter disk size: 33

FCFS Seek Time: 7
SCAN Seek Time: 51
C-SCAN Seek Time: 59
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.

Experiment No: 11

Experiment Name: Simulate algorithm for deadlock prevention and detection.

Objective:

The objective of this experiment is to understand and implement the given operating system concept using Python.

Theory:

Deadlock is a state where processes wait indefinitely for resources held by each other.

Deadlock Prevention: Ensures at least one of the necessary conditions for deadlock (Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait) does not occur.

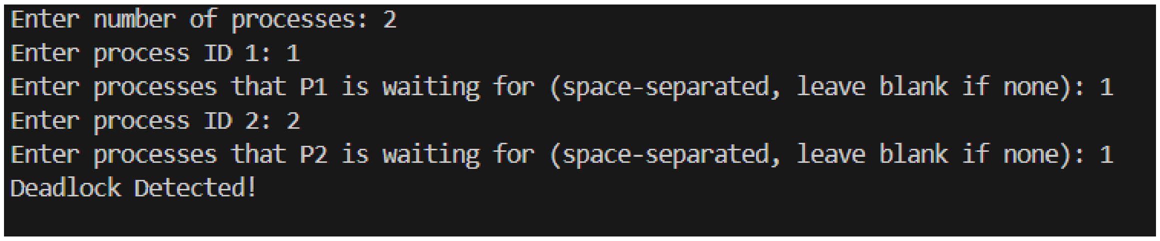
Deadlock Detection: Allows deadlock to occur but provides algorithms to detect it (e.g., resource allocation graphs, wait-for graphs). After detection, recovery techniques like process termination or resource preemption are used.

Code:

```
# ---------------- Deadlock Detection using Wait-For Graph ----------------


def detect_deadlock(graph):

    visited, rec_stack = set(), set()


    def dfs(v):

        visited.add(v)

        rec_stack.add(v)

        for nei in graph.get(v, []):

            if nei not in visited and dfs(nei):

                return True

            elif nei in rec_stack:
                return True
        rec_stack.remove(v)
        return False

    for node in graph:
        if node not in visited:
            if dfs(node):
                return True
    return False

# ---------------- User Input ----------------
n = int(input("Enter number of processes: "))
graph = {}

for i in range(n):
    proc = int(input(f"Enter process ID {i+1}: "))
    edges = input(f"Enter processes that P{proc} is waiting for (space-separated, leave blank
if none): ")
    if edges.strip():
        graph[proc] = list(map(int, edges.split()))
    else:
        graph[proc] = []

# ---------------- Deadlock Detection ----------------
if detect_deadlock(graph):
    print("Deadlock Detected!")
```

```
else:
    print("No Deadlock Detected.")
```

Output:



```
Enter number of processes: 2
Enter process ID 1: 1
Enter processes that P1 is waiting for (space-separated, leave blank if none): 1
Enter process ID 2: 2
Enter processes that P2 is waiting for (space-separated, leave blank if none): 1
Deadlock Detected!
```

Discussion:

This experiment demonstrates the respective operating system concept using Python implementation. The program was tested with sample inputs and verified for correctness.