Lab No: 1
Practical Slip No: 1

Q1) Write a Python program to Implement CPU Scheduling using FCFS algorithm.

```
# First-Come, First-Served (FCFS) Scheduling Algorithm

def findWaitingTime(processes, n, bt, wt):
    wt[0] = 0
    for i in range(1, n):
        wt[i] = bt[i - 1] + wt[i - 1]

def findTurnAroundTime(processes, n, bt, wt, tat):
    for i in range(n):
        tat[i] = bt[i] + wt[i]

def findAvgTime(processes, n, bt):
    wt = [0] * n
    tat = [0] * n
    findWaitingTime(processes, n, bt, wt)
    findTurnAroundTime(processes, n, bt, wt, tat)

    print("Processes  Burst time  Waiting time  Turnaround time")
    total_wt = 0
    total_tat = 0
    for i in range(n):
        total_wt += wt[i]
        total_tat += tat[i]
        print(f" {i+1}\t\t{bt[i]}\t\t{wt[i]}\t\t{tat[i]}")

    print(f"\nAverage waiting time = {total_wt / n}")
    print(f"Average turnaround time = {total_tat / n}")

processes = [1, 2, 3]
n = len(processes)
burst_time = [10, 5, 8]

findAvgTime(processes, n, burst_time)
```

Q2) Write a Python program to demonstrate working of FIFO using Queue interface in Python.

```
from queue import Queue

q = Queue(maxsize=3)
```

```
q.put(10)
q.put(20)
q.put(30)

print("Queue Elements:")
while not q.empty():
    print(q.get())
```

Lab No: 2
Practical Slip No: 2

Q1) Write a Python program to implement the below commands in Linux using command line arguments: wc, ls, cat.

```
import sys
import os

if len(sys.argv) < 2:
    print("Usage: python script.py <command> <file_name>")
    sys.exit(1)

command = sys.argv[1]

if command == "ls":
    os.system("ls")
elif command == "wc":
    filename = sys.argv[2]
    os.system(f"wc {filename}")
elif command == "cat":
    filename = sys.argv[2]
    os.system(f"cat {filename}")
else:
    print("Invalid command!")
```

Q2) Write a Python program to Implement the second largest (Second Chance) number in a list.

```
def second_largest(numbers):
    unique_numbers = list(set(numbers))
    unique_numbers.sort(reverse=True)
    return unique_numbers[1] if len(unique_numbers) > 1 else None

numbers = [10, 20, 4, 45, 99, 99]
```

```
print("Second Largest Number:", second_largest(numbers))
```

Lab No: 3
Practical Slip No: 3

Q1) Write a Python program to implement Round Robin CPU Scheduling Algorithm.

```
def roundRobin(processes, n, bt, quantum):
    rem_bt = bt[:]
    t = 0

    while True:
        done = True
        for i in range(n):
            if rem_bt[i] > 0:
                done = False
                if rem_bt[i] > quantum:
                    t += quantum
                    rem_bt[i] -= quantum
                else:
                    t += rem_bt[i]
                    rem_bt[i] = 0
                    print(f"Process {i+1} finished at time {t}")
        if done:
            break

processes = [1, 2, 3, 4]
n = len(processes)
burst_time = [24, 3, 3, 6]
quantum = 4

roundRobin(processes, n, burst_time, quantum)
```

Q2) Write a Python program to simulate Page Replacement Algorithm using FIFO.

```
def fifoPageReplacement(pages, capacity):
    page_set = set()
    page_queue = []
    page_faults = 0

    for page in pages:
        if page not in page_set:
            if len(page_set) >= capacity:
                removed_page = page_queue.pop(0)
```

```
            page_set.remove(removed_page)
        page_set.add(page)
        page_queue.append(page)
        page_faults += 1

    return page_faults

pages = [7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2]
capacity = 3
print("Page Faults:", fifoPageReplacement(pages, capacity))
```

(More practical slips will continue in this format below.)

Slip No: 4

Q1) Write a Python program to implement SJF (Shortest Job First) CPU scheduling algorithm. [15M]

python
CopyEdit

```
def sjf_scheduling(processes, n, bt):
    processes.sort(key=lambda x: x[1])  # Sort by burst time
    wt = [0] * n
    tat = [0] * n

    for i in range(1, n):
        wt[i] = wt[i - 1] + processes[i - 1][1]

    for i in range(n):
        tat[i] = wt[i] + processes[i][1]

    total_wt = sum(wt)
    total_tat = sum(tat)

    print("Processes Burst time Waiting time Turnaround time")
    for i in range(n):
        print(f" {processes[i][0]}\t\t {processes[i][1]}\t\t
{wt[i]}\t\t {tat[i]}")

    print(f"\nAverage waiting time = {total_wt / n:.2f}")
    print(f"Average turnaround time = {total_tat / n:.2f}")

if __name__ == "__main__":
```

```
    processes = [(1, 6), (2, 8), (3, 7), (4, 3)]
    n = len(processes)
    sjf_scheduling(processes, n, processes)
```

Q2) Write a Python program to implement LRU (Least Recently Used) page replacement algorithm. [15M]

python
CopyEdit

```
def lru_page_replacement(pages, n, capacity):
    memory = []
    page_faults = 0

    for page in pages:
        if page not in memory:
            if len(memory) < capacity:
                memory.append(page)
            else:
                memory.pop(0)  # Remove least recently used page
                memory.append(page)
            page_faults += 1
        else:
            memory.remove(page)  # Move accessed page to the end
            memory.append(page)
        print(memory)

    print("Total Page Faults:", page_faults)

pages = [2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5]
capacity = 3
lru_page_replacement(pages, len(pages), capacity)
```

Q3) Viva [5M]

(Prepare to answer questions on SJF scheduling and LRU page replacement.)

Slip No: 5

Q1) Write a Python program to implement Round Robin CPU scheduling algorithm. [15M]

python
CopyEdit

```
def round_robin(processes, bt, quantum):
    n = len(processes)
    rem_bt = bt[:]
    t = 0  # Current time

    while True:
        done = True
        for i in range(n):
            if rem_bt[i] > 0:
                done = False
                if rem_bt[i] > quantum:
                    t += quantum
                    rem_bt[i] -= quantum
                else:
                    t += rem_bt[i]
                    rem_bt[i] = 0
                    print(f"Process {processes[i]} finished at time
{t}")

        if done:
            break

processes = [1, 2, 3, 4]
burst_time = [5, 10, 15, 20]
quantum = 5
round_robin(processes, burst_time, quantum)
```

Q2) Write a Python program to implement Optimal page replacement algorithm. [15M]

python
CopyEdit

```
def optimal_page_replacement(pages, capacity):
    memory = []
    page_faults = 0

    for i in range(len(pages)):
        if pages[i] not in memory:
            if len(memory) < capacity:
                memory.append(pages[i])
            else:
```

```
                future_uses = {page: pages[i:].index(page) if page
in pages[i:] else float('inf') for page in memory}
                least_needed = max(future_uses, key=future_uses.get)
                memory.remove(least_needed)
                memory.append(pages[i])
            page_faults += 1
        print(memory)

    print("Total Page Faults:", page_faults)

pages = [7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1]
capacity = 3
optimal_page_replacement(pages, capacity)
```

Q3) Viva [5M]

(Prepare to answer questions on Round Robin scheduling and Optimal page replacement.)

Slip No: 6

Q1) Write a Python program to implement Priority CPU scheduling algorithm (Preemptive). [15M]

python
CopyEdit

```
import heapq

def priority_scheduling(processes):
    heapq.heapify(processes)
    t = 0

    while processes:
        priority, pid, bt = heapq.heappop(processes)
        t += bt
        print(f"Process {pid} with priority {priority} completed at
time {t}")

processes = [(3, 1, 10), (1, 2, 5), (4, 3, 7), (2, 4, 3)]
priority_scheduling(processes)
```

Q2) Write a Python program to implement Banker’s Algorithm for Deadlock Avoidance. [15M]

python
CopyEdit

```
def is_safe(processes, avail, maxm, alloc):
    n = len(processes)
    m = len(avail)
    need = [[maxm[i][j] - alloc[i][j] for j in range(m)] for i in
range(n)]
    safe_seq = []
    finish = [False] * n
    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)):
                work = [work[j] + alloc[i][j] for j in range(m)]
                safe_seq.append(processes[i])
                finish[i] = True
                break
        else:
            print("System is in deadlock!")
            return False

    print("Safe sequence:", safe_seq)
    return True

processes = [0, 1, 2, 3, 4]
avail = [3, 3, 2]
maxm = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
alloc = [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]
is_safe(processes, avail, maxm, alloc)
```

Q3) Viva [5M]

(Prepare to answer questions on Priority Scheduling and Banker’s Algorithm.)

Slip No. 7

Q1) Write a program to implement the Banker's Algorithm for Deadlock Avoidance.

[15M]

python
CopyEdit

```
def is_safe(processes, available, max_need, allocation):
    n = len(processes)
    m = len(available)
    need = [[max_need[i][j] - allocation[i][j] for j in range(m)]
for i in range(n)]
    work = available[:]
    finish = [False] * n
    safe_sequence = []

    while len(safe_sequence) < n:
        found = False
        for i in range(n):
            if not finish[i] and all(need[i][j] <= work[j] for j in
range(m)):
                work = [work[j] + allocation[i][j] for j in
range(m)]
                safe_sequence.append(processes[i])
                finish[i] = True
                found = True
                break
        if not found:
            print("System is in an unsafe state!")
            return False

    print(f"System is in a safe state! Safe Sequence:
{safe_sequence}")
    return True

processes = [0, 1, 2, 3, 4]
available = [3, 3, 2]
max_need = [[7, 5, 3], [3, 2, 2], [9, 0, 2], [2, 2, 2], [4, 3, 3]]
allocation = [[0, 1, 0], [2, 0, 0], [3, 0, 2], [2, 1, 1], [0, 0, 2]]

is_safe(processes, available, max_need, allocation)
```

Q2) Write a program to implement the Least Recently Used (LRU) Page Replacement Algorithm.

[15M]

python
CopyEdit

```
def lru_page_replacement(pages, capacity):
    memory = []
    page_faults = 0

    print("\nPage Replacement Process:")
    for page in pages:
        if page not in memory:
            if len(memory) == capacity:
                memory.pop(0)
            memory.append(page)
            page_faults += 1
            print(f"Page {page} added. Memory: {memory}")
        else:
            memory.remove(page)
            memory.append(page)
            print(f"Page {page} accessed. Memory: {memory}")

    print(f"\nTotal Page Faults: {page_faults}")

pages = [7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1]
capacity = 3
lru_page_replacement(pages, capacity)
```

Q3) Viva

[5M]

Slip No. 8

Q1) Write a program to implement Shortest Job First (SJF) Scheduling (Non-Preemptive).

[15M]

python
CopyEdit

```
def sjf_scheduling(processes):
```

```
    processes.sort(key=lambda x: x[1])
    n = len(processes)
    wt = [0] * n
    tat = [0] * n
    total_wt = 0
    total_tat = 0
    ct = 0

    print("\nGantt Chart:")
    for i in range(n):
        ct += processes[i][1]
        tat[i] = ct - processes[i][0]
        wt[i] = tat[i] - processes[i][1]
        total_wt += wt[i]
        total_tat += tat[i]
        print(f"| P{processes[i][2]} ", end="")
    print("|")

    avg_wt = total_wt / n
    avg_tat = total_tat / n

    print("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time")
    for i in range(n):

print(f"P{processes[i][2]}\t{processes[i][1]}\t\t{wt[i]}\t\t{tat[i]}
")

    print(f"\nAverage Waiting Time: {avg_wt}")
    print(f"Average Turnaround Time: {avg_tat}")

processes = [(0, 6, 1), (1, 8, 2), (2, 7, 3), (3, 3, 4)]
sjf_scheduling(processes)
```

Q2) Write a program to implement the Optimal Page Replacement Algorithm.

[15M]

python
CopyEdit

```
def optimal_page_replacement(pages, capacity):
    memory = []
```

```
    page_faults = 0

    print("\nPage Replacement Process:")
    for i in range(len(pages)):
        if pages[i] not in memory:
            if len(memory) < capacity:
                memory.append(pages[i])
            else:
                future_use = []
                for page in memory:
                    if page in pages[i+1:]:
                        future_use.append(pages[i+1:].index(page))
                    else:
                        future_use.append(float('inf'))
                memory[future_use.index(max(future_use))] = pages[i]
            page_faults += 1
            print(f"Page {pages[i]} added. Memory: {memory}")
        else:
            print(f"Page {pages[i]} already in memory: {memory}")

    print(f"\nTotal Page Faults: {page_faults}")

pages = [7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1]
capacity = 3
optimal_page_replacement(pages, capacity)
```

Q3) Viva

[5M]

Slip No. 9

Q1) Write a program to implement the Round Robin (RR) Scheduling Algorithm.

[15M]

python
CopyEdit

```
def round_robin(processes, burst_time, quantum):
    n = len(processes)
    remaining_bt = burst_time[:]
    wt = [0] * n
```

```
    tat = [0] * n
    time = 0

    print("\nGantt Chart:")
    while True:
        done = True
        for i in range(n):
            if remaining_bt[i] > 0:
                done = False
                if remaining_bt[i] > quantum:
                    time += quantum
                    remaining_bt[i] -= quantum
                    print(f"| P{processes[i]} ", end="")
                else:
                    time += remaining_bt[i]
                    wt[i] = time - burst_time[i]
                    remaining_bt[i] = 0
                    print(f"| P{processes[i]} ", end="")
        if done:
            break
    print("|")

    for i in range(n):
        tat[i] = burst_time[i] + wt[i]

    print("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time")
    for i in range(n):

print(f"P{processes[i]}\t{burst_time[i]}\t\t{wt[i]}\t\t{tat[i]}")

    print(f"\nAverage Waiting Time: {sum(wt)/n}")
    print(f"Average Turnaround Time: {sum(tat)/n}")

processes = [1, 2, 3, 4]
burst_time = [5, 4, 2, 1]
quantum = 2
round_robin(processes, burst_time, quantum)
```

Q2) Write a program to implement First-In-First-Out (FIFO) Page Replacement Algorithm.

[15M]

python
CopyEdit

```
from collections import deque

def fifo_page_replacement(pages, capacity):
    memory = deque()
    page_faults = 0

    print("\nPage Replacement Process:")
    for page in pages:
        if page not in memory:
            if len(memory) == capacity:
                memory.popleft()
            memory.append(page)
            page_faults += 1
            print(f"Page {page} added. Memory: {list(memory)}")
        else:
            print(f"Page {page} already in memory: {list(memory)}")

    print(f"\nTotal Page Faults: {page_faults}")

pages = [7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1]
capacity = 3
fifo_page_replacement(pages, capacity)
```

Q3) Viva

[5M]

Slip No. 10

Q1) Write a program to implement Priority Scheduling (Non-Preemptive).

[15M]

python
CopyEdit

```
def priority_scheduling(processes):
    processes.sort(key=lambda x: x[2])
```

```
    n = len(processes)
    wt = [0] * n
    tat = [0] * n
    total_wt = 0
    total_tat = 0
    ct = 0

    print("\nGantt Chart:")
    for i in range(n):
        ct += processes[i][1]
        tat[i] = ct - processes[i][0]
        wt[i] = tat[i] - processes[i][1]
        total_wt += wt[i]
        total_tat += tat[i]
        print(f"| P{processes[i][3]} ", end="")
    print("|")

    print("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround
Time")
    for i in range(n):

print(f"P{processes[i][3]}\t{processes[i][1]}\t\t{processes[i][2]}\t
\t{wt[i]}\t\t{tat[i]}")

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

processes = [(0, 10, 1, 1), (1, 1, 3, 2), (2, 2, 2, 3), (3, 1, 4,
4)]
priority_scheduling(processes)
```

Q2) Write a program to implement Least Frequently Used (LFU) Page Replacement Algorithm.

[15M]

python
CopyEdit

```
from collections import Counter

def lfu_page_replacement(pages, capacity):
```

```
    memory = []
    frequency = Counter()
    page_faults = 0

    print("\nPage Replacement Process:")
    for page in pages:
        if page not in memory:
            if len(memory) == capacity:
                lfu_page = min(memory, key=lambda p: frequency[p])
                memory.remove(lfu_page)
                del frequency[lfu_page]
            memory.append(page)
            page_faults += 1
        frequency[page] += 1
        print(f"Page {page} added. Memory: {memory}, Frequency:
{dict(frequency)}")

    print(f"\nTotal Page Faults: {page_faults}")

pages = [7, 0, 1, 2, 0, 3, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1]
capacity = 3
lfu_page_replacement(pages, capacity)
```

Q3) Viva

[5M]