Practical No: 3

Aim:Threading and Single Thread Control Flow

3.1: Practice Thread Creation and basic thread lifecycle using standard libraries(e.g., pthreads and Java threads)

Input:

```
import threading
import time

def task(name, delay):
    print(f"Task {name} started")
    time.sleep(delay)
    print(f"Task {name} finished after {delay} seconds.")
def single_threaded():
    start_time = time.time()
    task("A", 2)
    task("B", 2)
    end_time = time.time()
    print(f"Single-threaded execution time: {end_time - start_time} seconds")
def multi_threaded():
    start_time = time.time()
    t1 = threading.Thread(target=task, args=("A", 2))
    t2 = threading.Thread(target=task, args=("B", 2))
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    end_time = time.time()
    print(f"Multi-threaded execution time: {end_time - start_time} seconds")
if __name__ == "__main__":
    print("Running single-threaded version")
    single_threaded()

    print("Running multi-threaded version")
    multi_threaded()
```

Output:

Running single-threaded version
Task A started
Task A finished after 2 seconds.
Task B started
Task B finished after 2 seconds.
Single-threaded execution time: 4.000314235687256 seconds
Running multi-threaded version
Task A started
Task B started
Task A finished after 2 seconds.
Task B finished after 2 seconds.
Multi-threaded execution time: 2.0007643699645996 seconds

3.2: Observe execution order, thread joining and delays

Input:

```
import threading
import time

def task(name,delay):
  print(f"[{time.strftime('%H:%M:%S')}]Thread {name} starting")
  print(f"[{time.strftime('%H:%M:%S')}]Thread {name}sleeping for {delay}second")
  time.sleep(delay)
  print(f"[{time.strftime('%H:%M:%S')}]Thread {name} finished")

def main():
  print(f"[{time.strftime('%H:%M:%S')}] Main thread: Creating threads")

  threads = [
    threading.Thread(target=task, args=("A",3),name="Thread-A"),
    threading.Thread(target=task, args=("B",2),name="Thread-B"),
    threading.Thread(target=task, args=("C",1),name="Thread-C"),
  ]
  for t in threads:
    print(f"[{time.strftime('%H:%M:%S')}]Main thread:Starting {t.name}")
    t.start()
  for t in threads:
    print(f"[{time.strftime('%H:%M:%S')}]Main thread:Waiting for {t.name} to finish")
    t.join()
    print(f"[{time.strftime('%H:%M:%S')}]Main thread:{t.name}finished")
  print(f"[{time.strftime('%H:%M:%S')}]Main thread:All threads completed")
```

```
if __name__=="__main__":
   main()
```

Output:

```
[03:05:59] Main thread: Creating threads
[03:05:59]Main thread:Starting Thread-A
[03:05:59]Thread A starting
[03:05:59]Main thread:Starting Thread-B
[03:05:59]Thread Asleeping for 3second
[03:05:59]Thread B starting
[03:05:59]Main thread:Starting Thread-C
[03:05:59]Thread Bsleeping for 2second
[03:05:59]Thread C starting
[03:05:59]Main thread:Waiting for Thread-A to finish
[03:05:59]Thread Csleeping for 1second
[03:06:00]Thread C finished
[03:06:01]Thread B finished
[03:06:02]Thread A finished
[03:06:02]Main thread:Thread-Afinished
[03:06:02]Main thread:Waiting for Thread-B to finish
[03:06:02]Main thread:Thread-Bfinished
[03:06:02]Main thread:Waiting for Thread-C to finish
[03:06:02]Main thread:Thread-Cfinished
[03:06:02]Main thread:All threads completed
```

3.3: Compare Execution time between:

1) Sequential (single-threaded) execution
2) Multi-threaded execution

Input:

```
import threading
import time

def task(name, delay):
   print(f"[{time.strftime('%H:%M:%S')}] Task {name} started")
   time.sleep(delay)
   print(f"[{time.strftime('%H:%M:%S')}] Task {name} finished")

def sequential_execution():
   print("\n=== Sequential Execution ===")
   start = time.time()
```

```
    task("A", 3)
    task("B", 2)
    task("C", 1)
    end = time.time()
    print(f"Total time (Sequential): {end - start:.2f} seconds")

def multithreaded_execution():
    print("\n=== Multithreaded Execution ===")
    start = time.time()
    threads = [
        threading.Thread(target=task, args=("A", 3)),
        threading.Thread(target=task, args=("B", 2)),
        threading.Thread(target=task, args=("C", 1)),
    ]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    end = time.time()
    print(f"Total time (Multithreaded): {end - start:.2f} seconds")

sequential_execution()
multithreaded_execution()
```

Output:

```
=== Sequential Execution ===
[03:45:20] Task A started
[03:45:23] Task A finished
[03:45:23] Task B started
[03:45:25] Task B finished
[03:45:25] Task C started
[03:45:26] Task C finished
Total time (Sequential): 6.00 seconds

=== Multithreaded Execution ===
[03:45:26] Task A started
[03:45:26] Task B started
[03:45:26] Task C started
[03:45:27] Task C finished
[03:45:28] Task B finished
[03:45:29] Task A finished
Total time (Multithreaded): 3.00 seconds
```

Practical No: 4

Aim:Multi-threading and Fibonacci Generation

4.1: Implement multi-threading to generate and print Fibonacci sequences

Input:

```
import threading
def fibonacci(n,name):
  a,b=0,1
  print(f"{name} generating{n}Fibonacci numbers:")
  for i in range(n):
    print(f"{name}:{a}")
    a,b=b,a+b

#Create thread for two fibonacci sequences
t1=threading.Thread(target=fibonacci,args=(5,"Thread-1"))
t2=threading.Thread(target=fibonacci,args=(7,"Thread-2"))
t1.start()
t2.start()
t1.join()
t1.join()
print("All Fibonacci threads finished:")
```

Output:

```
Thread-1 generating5Fibonacci numbers:
Thread-1:0
Thread-1:1
Thread-1:1
Thread-1:2
Thread-1:3
Thread-2 generating7Fibonacci numbers:
Thread-2:0
Thread-2:1
Thread-2:1
Thread-2:2
Thread-2:3
Thread-2:5
Thread-2:8
All Fibonacci threads finished:
```

4.2: Thread safety and synchronization when accessing shared variables

Input:

```
import threading
lock = threading.Lock()
shared_sum = 0
```

```
def add_fibonacci_sum(n):
    global shared_sum
    a,b = 0,1
    for i in range(n):
        with lock:  #ensure only one thread updates at a time
            shared_sum += a
        a, b = b, a + b
threads = []

for i in range(3):
    t = threading.Thread(target=add_fibonacci_sum, args=(5,))
    threads.append(t)
    t.start()
for t in threads:
    t.join()
print("Total sum of Fibonacci numbers(shared:)",shared_sum)
```

Output:

Total sum of Fibonacci numbers(shared:) 21

4.3: Thread pooling and task delegation using ThreadPool Executor

Input:

```
from concurrent.futures import ThreadPoolExecutor
def fibonacci_list(n):
    seq = []
    a,b = 0,1

    for i in range(n):
        seq.append(a)
        a,b = b,a+b
    return seq

# Thread pool with 3 Workers
with ThreadPoolExecutor(max_workers=3) as executor:
    results = list(executor.map(fibonacci_list, [5,7,10]))
for i, seq in enumerate(results,1):
    print(f"Task {i} is: {seq}")
```

Output:

Task 1 is: [0, 1, 1, 2, 3]
Task 2 is: [0, 1, 1, 2, 3, 5, 8]
Task 3 is: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Practical No: 5

<u>Aim : Process Synchronization and Bounded Buffer Problem.</u>

<u>**5.1**: Stimulate producer-consumer bounded buffer using mutex and semaphores.</u>

<u>**5.2:** Buffer control with synchronized access.</u>

<u>Input:</u>

```
import threading
import time
import random

BUFFER_SIZE = 5
buffer = []
mutex = threading.Lock()
empty = threading.Semaphore(BUFFER_SIZE)
full = threading.Semaphore(0)
running = True    #Stop Flag

def producer():
    global running
    while running:
        item = random.randint(1, 100)
        empty.acquire()
        with mutex:
            buffer.append(item)
            print(f"Produced: {item}, Buffer: {buffer}")
        full.release()
        time.sleep(random.random())

def consumer():
    global running
    while running:
        full.acquire()
        with mutex:
            if buffer:
```

```
            item = buffer.pop(0)
            print(f"Consumed: {item}, Buffer: {buffer}")
        empty.release()
    time.sleep(random.random())

if __name__ == "__main__":
    t1 = threading.Thread(target=producer)
    t2 = threading.Thread(target=consumer)
    t1.start()
    t2.start()
    time.sleep(10)
    running = False
    empty.release()
    full.release()
    t1.join()
t2.join()
print("Simulation Finished")
```

Output:

```
Produced: 90, Buffer: [90]
Consumed: 90, Buffer: []
Produced: 40, Buffer: [40]
Consumed: 40, Buffer: []
Produced: 100, Buffer: [100]
Consumed: 100, Buffer: []
Produced: 98, Buffer: [98]
Consumed: 98, Buffer: []
Produced: 83, Buffer: [83]
Consumed: 83, Buffer: []
Produced: 99, Buffer: [99]
Consumed: 99, Buffer: []
Produced: 11, Buffer: [11]
Produced: 70, Buffer: [11, 70]
Consumed: 11, Buffer: [70]
Produced: 19, Buffer: [70, 19]
```

Consumed: 70, Buffer: [19]
Produced: 61, Buffer: [19, 61]
Consumed: 19, Buffer: [61]
Consumed: 61, Buffer: []
Produced: 21, Buffer: [21]
Consumed: 21, Buffer: []
Produced: 11, Buffer: [11]
Consumed: 11, Buffer: []
Produced: 93, Buffer: [93]
Consumed: 93, Buffer: []
Produced: 25, Buffer: [25]
Consumed: 25, Buffer: []
Produced: 29, Buffer: [29]
Consumed: 29, Buffer: []
Produced: 58, Buffer: [58]
Produced: 87, Buffer: [58, 87]
Consumed: 58, Buffer: [87]
Produced: 22, Buffer: [87, 22]
Consumed: 87, Buffer: [22]
Produced: 56, Buffer: [22, 56]
Consumed: 22, Buffer: [56]
Produced: 99, Buffer: [56, 99]
Consumed: 56, Buffer: [99]
Produced: 71, Buffer: [99, 71]
Simulation Finished

5.3: Circular queue technique(bounded buffer)

Input:

```
import threading
import queue
import time
import random
BUFFER_SIZE=5
buffer=queue.Queue(BUFFER_SIZE)
running=True
SENTINEL=None
```

```
def producer():
    global running
    while running:
        item=random.randint(1,100)
        buffer.put(item)
        print(f"Produced: {item}")
        time.sleep(random.random())
    buffer.put(SENTINEL)

def consumer():
    while True:
        item=buffer.get()
        if item is SENTINEL:
            buffer.task_done()
            break
        print(f"Consumed: {item}")
        buffer.task_done()
        time.sleep(random.random())

if __name__=="__main__":
    t1=threading.Thread(target=producer)
    t2=threading.Thread(target=consumer)

    t1.start()
    t2.start()
    time.sleep(10)
    running=False

    t1.join()
    t2.join()
    print("Simulation Finished")
```

Output:

Produced: 49
Consumed: 49
Produced: 70
Consumed: 70

Produced: 15
Produced: 48
Consumed: 15
Produced: 88
Consumed: 48
Consumed: 88
Produced: 62
Consumed: 62
Produced: 99
Consumed: 99
Produced: 1
Consumed: 1
Produced: 64
Consumed: 64
Produced: 59
Consumed: 59
Produced: 27
Consumed: 27
Produced: 40
Produced: 46
Produced: 18
Produced: 60
Consumed: 40
Produced: 68
Consumed: 46
Produced: 73
Consumed: 18
Produced: 84
Consumed: 60
Consumed: 68
Produced: 91
Produced: 36
Consumed: 73
Consumed: 84
Consumed: 91
Consumed: 36
Simulation Finished

Practical No: 6

AIM: Readers-Writers Problem- Synchronization in Shared Access

6.1. Implement Access reader and writer prioritization.

6.2. Use semaphores to allow multiple readers or exclusive.

Input:

```
import threading
import time
import random

# Semaphores
mutex = threading.Semaphore(1)   # Protects rc
db = threading.Semaphore(1)      # Controls access to shared data

# Shared resource and reader count
shared_data = 0
rc = 0
running = True  # Stop flag
def reader(reader_id):
    global rc, shared_data, running
    while running:
        # Entry section
        mutex.acquire()
        rc += 1
        if rc == 1:
            db.acquire()   # First reader locks db
        mutex.release()

        # Critical section
        print(f"[Reader] {reader_id} reads {shared_data}")
        time.sleep(random.uniform(0.2, 0.5))

        # Exit section
        mutex.acquire()
        rc -= 1
        if rc == 0:
            db.release()   # Last reader unlocks db
        mutex.release()
        # Pause before next read
        time.sleep(random.uniform(0.5, 1.0))
def writer(writer_id):
    global shared_data, running
```

```
    while running:
   db.acquire()  # Exclusive access
        shared_data += 1
        print(f"[Writer] {writer_id} writes {shared_data}")
        time.sleep(random.uniform(0.3, 0.6))
        db.release()
        # Pause before next write
        time.sleep(random.uniform(0.8, 1.5))

if __name__ == "__main__":
    readers = [threading.Thread(target=reader, args=(i,)) for i in range(3)]
    writers = [threading.Thread(target=writer, args=(i,)) for i in range(2)]
    for t in readers + writers:
        t.start()
    time.sleep(10)   # Run simulation for 10 sec
    running = False  # Stop all threads

    for t in readers + writers:
        t.join()
    print("Simulation finished.")
```

Output:

```
[Reader] 0 reads 0
[Reader] 1 reads 0
[Reader] 2 reads 0
[Writer] 0 writes 1
[Writer] 1 writes 2
[Reader] 0 reads 2
[Reader] 1 reads 2
[Reader] 2 reads 2
[Writer] 0 writes 3
[Reader] 2 reads 3
[Reader] 0 reads 3
[Reader] 1 reads 3
[Writer] 1 writes 4
[Reader] 2 reads 4
[Reader] 1 reads 4
[Reader] 0 reads 4
[Writer] 0 writes 5
[Reader] 1 reads 5
```

```
[Reader] 0 reads 5
[Reader] 2 reads 5
[Writer] 1 writes 6
[Writer] 0 writes 7
[Reader] 1 reads 7
[Reader] 2 reads 7
[Reader] 0 reads 7
[Writer] 1 writes 8
[Reader] 1 reads 8
[Reader] 0 reads 8
[Reader] 2 reads 8
[Writer] 0 writes 9
[Reader] 0 reads 9
[Reader] 1 reads 9
[Reader] 2 reads 9
[Writer] 1 writes 10
[Writer] 0 writes 11
[Reader] 2 reads 11
[Reader] 1 reads 11
[Reader] 0 reads 11
Simulation finished.
```

6.3. Extend to fairness writer access. in access and deadlock prevention.

Input:

```
import threading
import time
import random

# Semaphores
mutex = threading.Semaphore(1)        # Protects rc (read count)
db = threading.Semaphore(1)           # Controls access to shared data
serviceQueue = threading.Semaphore(1)  # Fairness: queue for both readers/writers
# Shared resource and counters
shared_data = 0
rc = 0
running = True  # Stop flag

def reader(reader_id):
  global rc, shared_data, running
  while running:
    # Fairness: wait in queue
```

```
        serviceQueue.acquire()
        mutex.acquire()
        rc += 1
        if rc == 1:
            db.acquire()  # First reader locks db
        mutex.release()
        serviceQueue.release()

        # Critical section
        print(f"[Reader] {reader_id} reads {shared_data}")
        time.sleep(random.uniform(0.2, 0.5))
        # Exit section
        mutex.acquire()
        rc -= 1
        if rc == 0:
            db.release()  # Last reader unlocks db
        mutex.release()
        time.sleep(random.uniform(0.5, 1.0))

def writer(writer_id):
    global shared_data, running
    while running:
        # Fairness: wait in queue
        serviceQueue.acquire()
        db.acquire()   # Exclusive access
        serviceQueue.release()

        # Critical section
        shared_data += 1
        print(f"[Writer] {writer_id} writes {shared_data}")
        time.sleep(random.uniform(0.3, 0.6))
        db.release()
time.sleep(random.uniform(0.8, 1.5))

if __name__ == "__main__":
    readers = [threading.Thread(target=reader, args=(i,)) for i in range(3)]
    writers = [threading.Thread(target=writer, args=(i,)) for i in range(2)]

    for t in readers + writers:
        t.start()
    time.sleep(10)   # Run simulation for 10 sec
    running = False  # Stop all threads
```

```
    for t in readers + writers:
        t.join()
    print("Simulation finished.")
```

Output:

```
[Reader] 0 reads 0
[Reader] 1 reads 0
[Reader] 2 reads 0
[Writer] 0 writes 1
[Writer] 1 writes 2
[Reader] 0 reads 2
[Reader] 1 reads 2
[Reader] 2 reads 2
[Reader] 1 reads 2
[Writer] 0 writes 3
[Reader] 2 reads 3
[Reader] 0 reads 3
[Writer] 1 writes 4
[Reader] 1 reads 4
[Writer] 0 writes 5
[Reader] 2 reads 5
[Reader] 0 reads 5
[Reader] 1 reads 5
[Writer] 1 writes 6
[Reader] 1 reads 6
[Reader] 2 reads 6
[Reader] 0 reads 6
[Writer] 0 writes 7
[Reader] 1 reads 7
[Writer] 1 writes 8
[Reader] 2 reads 8
[Reader] 0 reads 8
[Writer] 0 writes 9
[Reader] 1 reads 9
[Writer] 1 writes 10
[Reader] 2 reads 10
[Reader] 0 reads 10
[Reader] 1 reads 10
[Writer] 0 writes 11
[Reader] 0 reads 11
Simulation finished.
```

Practical No: 7

Aim: CPU Scheduling Algorithm (Part 1)- FCFS and Non-Preemptive Scheduling

7.1 Simulate First-Come First-Serve Scheduling

Input:

```
def fcfs_scheduling(processes):
    """
    processes: list of tuples (pid, arrival_time, burst_time)
    """
    processes.sort(key=lambda x: x[1])  # sort by arrival time

    start_time = []
    completion_time = []
    waiting_time = []
    turnaround_time = []

    current_time = 0
    gantt_chart = []

    for pid, arrival, burst in processes:
        if current_time < arrival:
            current_time = arrival  # CPU idle until process arrives

        start_time.append(current_time)
        gantt_chart.append((pid, current_time, current_time + burst))
        current_time += burst
        completion_time.append(current_time)

        tat = completion_time[-1] - arrival
        wt = tat - burst
        turnaround_time.append(tat)
        waiting_time.append(wt)
```

```
    avg_wt = sum(waiting_time) / len(processes)
    avg_tat = sum(turnaround_time) / len(processes)

    print("\n--- FCFS Scheduling ---")
    print("PID\tAT\tBT\tST\tCT\tTAT\tWT")
    for i, p in enumerate(processes):

print(f"{p[0]}\t{p[1]}\t{p[2]}\t{start_time[i]}\t{completion_time[i]}\t{turna
round_time[i]}\t{waiting_time[i]}")

    print(f"\nAverage Waiting Time: {avg_wt:.2f}")
    print(f"Average Turnaround Time: {avg_tat:.2f}")

    print("\nGantt Chart:")
    for pid, start, end in gantt_chart:
        print(f"| P{pid} ({start}-{end}) ", end="")
    print("|")

# Example usage
processes = [
    (1, 0, 5),
    (2, 2, 3),
    (3, 4, 1)
]

fcfs_scheduling(processes)
```

Output:

```
--- FCFS Scheduling ---
PID   AT   BT   ST   CT   TAT   WT
1     0    5    0    5    5     0
2     2    3    5    8    6     3
3     4    1    8    9    5     4
```

Average Waiting Time: 2.33
Average Turnaround Time: 5.33

Gantt Chart:
| P1 (0-5) | P2 (5-8) | P3 (8-9) |

<u>**7.2** Extend implementation to general non-preemptive scheduling.</u>
<u>**7.3** Analyze waiting time, turnaround time, and Gantt chart generation.</u>

<u>Input:</u>

```
def non_preemptive_priority(processes):
    """
    processes: list of tuples (pid, arrival_time, burst_time, priority)
    Lower priority value means higher priority.
    """
    n = len(processes)

    # Sort by arrival time first, then priority
    processes.sort(key=lambda x: (x[1], x[3]))
    completed = 0
    current_time = 0
    start_time = {}
    completion_time = {}
    waiting_time = {}
    turnaround_time = {}
    gantt_chart = []
    ready_queue = []
    visited = [False] * n

    while completed < n:
        # Add processes that have arrived by current_time
        for i in range(n):
            if processes[i][1] <= current_time and not visited[i]:
                ready_queue.append(processes[i])
```

```
            visited[i] = True

    if ready_queue:
        # Pick highest priority (lowest priority number)
        ready_queue.sort(key=lambda x: x[3])  # Sort by priority
        pid, at, bt, pr = ready_queue.pop(0)

        if current_time < at:
            # If no process is available to execute, CPU idles
            current_time = at
            start_time[pid] = current_time
            gantt_chart.append((pid, current_time, current_time + bt))
            current_time += bt
        else:
            # Process execution starts immediately
            start_time[pid] = current_time
            gantt_chart.append((pid, current_time, current_time + bt))
            current_time += bt

        completion_time[pid] = current_time
        turnaround_time[pid] = completion_time[pid] - at
        waiting_time[pid] = turnaround_time[pid] - bt
        completed += 1
    else:
        # CPU is idle if no processes are ready to execute
        current_time += 1

avg_wt = sum(waiting_time.values()) / n
avg_tat = sum(turnaround_time.values()) / n

# Print Results
print("\n--- Non-preemptive Priority Scheduling ---")
print("PID\tAT\tBT\tPriority\tST\tCT\tTAT\tWT")
for pid, at, bt, pr in processes:
```

```
print(f"{pid}\t{at}\t{bt}\t{pr}\t\t{start_time[pid]}\t{completion_time[pid]}\
t{turnaround_time[pid]}\t{waiting_time[pid]}")

    print(f"\nAverage Waiting Time: {avg_wt:.2f}")
    print(f"Average Turnaround Time: {avg_tat:.2f}")

    print("\nGantt Chart:")
    for pid, start, end in gantt_chart:
        print(f"| P{pid} ({start}-{end}) ", end="")
    print("|")

# Example usage
processes_priority = [
    (1, 0, 5, 2),  # (PID, Arrival Time, Burst Time, Priority)
    (2, 1, 3, 1),
    (3, 2, 8, 3),
    (4, 3, 6, 2)
]
non_preemptive_priority(processes_priority)
```

Output:

--- Non-preemptive Priority Scheduling ---

PID	AT	BT	Priority	ST	CT	TAT	WT
1	0	5	2	0	5	5	0
2	1	3	1	5	8	7	4
3	2	8	3	14	22	20	12
4	3	6	2	8	14	11	5

Average Waiting Time: 5.25
Average Turnaround Time: 10.75

Gantt Chart:
| P1 (0-5) | P2 (5-8) | P4 (8-14) | P3 (14-22) |

Practical No: 8

Aim: CPU Scheduling Algorithm (Part 2)-Round Robin

8.1 Implement Round Robin scheduling with configurable time quantum.
8.2 Compare with FCFS: fairness, turnaround, response time.
8.3 rack context switches and improve queue management.

Input:

```
from collections import deque
def round_robin(processes, time_quantum):
    """
    processes: list of tuples (pid, arrival_time, burst_time)
    time_quantum: int
    """
    n = len(processes)
    processes.sort(key=lambda x: x[1])  # sort by arrival time
    remaining_bt = {pid: bt for pid, at, bt in processes}
    completion_time = {}
    turnaround_time = {}
    waiting_time = {}
    response_time = {}
    start_time = {}
    gantt_chart = []

    ready_queue = deque()
    current_time = 0
    visited = [False] * n
    completed = 0
    context_switches = 0
    prev_pid = None

    while completed < n:
        # Add processes that have arrived
        for i, (pid, at, bt) in enumerate(processes):
            if at <= current_time and not visited[i]:
                ready_queue.append(pid)
                visited[i] = True
```

```
        if ready_queue:
            pid = ready_queue.popleft()
            if pid != prev_pid and prev_pid is not None:
                context_switches += 1
            prev_pid = pid
            if pid not in start_time:
                start_time[pid] = current_time
                response_time[pid] = current_time - next(at for p, at, bt in processes if p
== pid)

            exec_time = min(time_quantum, remaining_bt[pid])
            gantt_chart.append((pid, current_time, current_time + exec_time))
            current_time += exec_time
            remaining_bt[pid] -= exec_time

            # Add new arrivals during execution
            for i, (p, at, bt) in enumerate(processes):
                if at <= current_time and not visited[i] and p not in ready_queue:
                    ready_queue.append(p)
                    visited[i] = True

            if remaining_bt[pid] > 0:
                ready_queue.append(pid)
            else:
                completion_time[pid] = current_time
                completed += 1
        else:
            current_time += 1  # CPU idle

    # Calculate TAT & WT
    for pid, at, bt in processes:
        turnaround_time[pid] = completion_time[pid] - at
        waiting_time[pid] = turnaround_time[pid] - bt
    avg_wt = sum(waiting_time.values()) / n
    avg_tat = sum(turnaround_time.values()) / n
    avg_rt = sum(response_time.values()) / n
```

```
    # Print results
    print("\n--- Round Robin Scheduling ---")
    print(f"Time Quantum: {time_quantum}")
    print("PID\tAT\tBT\tST\tCT\tTAT\tWT\tRT")
    for pid, at, bt in processes:
        print(f"{pid}\t{at}\t{bt}\t{start_time[pid]}\t{completion_time[pid]}\t"
              f"{turnaround_time[pid]}\t{waiting_time[pid]}\t{response_time[pid]}")
    print(f"\nAverage Waiting Time: {avg_wt:.2f}")
    print(f"Average Turnaround Time: {avg_tat:.2f}")
    print(f"Average Response Time: {avg_rt:.2f}")
    print(f"Context Switches: {context_switches}")

    # Print Gantt Chart
    print("\nGantt Chart:")
    for pid, start, end in gantt_chart:
        print(f"| P{pid} ({start}-{end}) ", end="")
    print("|")

def fcfs(processes):
    processes.sort(key=lambda x: x[1])  # sort by arrival time
    n = len(processes)
    current_time = 0
    start_time = {}
    completion_time = {}
    turnaround_time = {}
    waiting_time = {}
    response_time = {}
    gantt_chart = []

    for pid, at, bt in processes:
        if current_time < at:
            current_time = at
        start_time[pid] = current_time
        response_time[pid] = current_time - at
        gantt_chart.append((pid, current_time, current_time + bt))
        current_time += bt
```

```
        completion_time[pid] = current_time
        turnaround_time[pid] = completion_time[pid] - at
        waiting_time[pid] = turnaround_time[pid] - bt
    avg_wt = sum(waiting_time.values()) / n
    avg_tat = sum(turnaround_time.values()) / n
    avg_rt = sum(response_time.values()) / n

    # Print results
    print("\n--- First-Come First-Serve (FCFS) ---")
    print("PID\tAT\tBT\tST\tCT\tTAT\tWT\tRT")
    for pid, at, bt in processes:
        print(f"{pid}\t{at}\t{bt}\t{start_time[pid]}\t{completion_time[pid]}\t"
              f"{turnaround_time[pid]}\t{waiting_time[pid]}\t{response_time[pid]}")
    print(f"\nAverage Waiting Time: {avg_wt:.2f}")
    print(f"Average Turnaround Time: {avg_tat:.2f}")
    print(f"Average Response Time: {avg_rt:.2f}")
    print("Context Switches: N/A (no preemption)")

    # Print Gantt Chart
    print("\nGantt Chart:")
    for pid, start, end in gantt_chart:
        print(f"| P{pid} ({start}-{end}) ", end="")
    print("|")

# Example usage
process_list = [
    (1, 0, 5),
    (2, 1, 4),
    (3, 2, 2),
    (4, 4, 1)]
fcfs(process_list.copy())
round_robin(process_list.copy(), time_quantum=2)
```

Output:

--- First-Come First-Serve (FCFS) ---

PID	AT	BT	ST	CT	TAT	WT	RT
1	0	5	0	5	5	0	0
2	1	4	5	9	8	4	4
3	2	2	9	11	9	7	7
4	4	1	11	12	8	7	7

Average Waiting Time: 4.50
Average Turnaround Time: 7.50
Average Response Time: 4.50
Context Switches: N/A (no preemption)

Gantt Chart:
| P1 (0-5) | P2 (5-9) | P3 (9-11) | P4 (11-12) |

--- Round Robin Scheduling ---
Time Quantum: 2

PID	AT	BT	ST	CT	TAT	WT	RT
1	0	5	0	12	12	7	0
2	1	4	2	11	10	6	1
3	2	2	4	6	4	2	2
4	4	1	8	9	5	4	4

Average Waiting Time: 4.75
Average Turnaround Time: 7.75
Average Response Time: 1.75
Context Switches: 6

Gantt Chart:
| P1 (0-2) | P2 (2-4) | P3 (4-6) | P1 (6-8) | P4 (8-9) | P2 (9-11) | P1 (11-12) |

Practical No: 9

AIM: Memory Management Techniques

9.1.Simulate FIFO and LRU page replacement using page reference strings. Measure
9.2.Hit/miss ratios under different reference patterns.
9.3.Extend to include frames and memory constraints.

Input:

```
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
from collections import deque

def fifo_page_replacement(pages, frames):
    memory = deque()
    hits, misses = 0, 0
    print("\n--- FIFO Page Replacement ---")
    for page in pages:
        if page in memory:
            hits += 1
        else:
            misses += 1
            if len(memory) >= frames:
                memory.popleft()
            memory.append(page)
        print(f"Page: {page} -> Memory: {list(memory)}")
    print(f"\nFIFO Results: Hits = {hits}, Misses = {misses}, "
          f"Hit Ratio = {hits/len(pages):.2f}, Miss Ratio = {misses/len(pages):.2f}")

def lru_page_replacement(pages, frames):
    memory = deque()
    hits, misses = 0, 0
    print("\n--- LRU Page Replacement ---")
    for page in pages:
        if page in memory:
            hits += 1
            memory.remove(page)
            memory.append(page)  # Move to most recently used
        else:
            misses += 1
```

```
        if len(memory) >= frames:
            memory.popleft()  # Remove least recently used
        memory.append(page)
    print(f"Page: {page} -> Memory: {list(memory)}")

print(f"\nLRU Results: Hits = {hits}, Misses = {misses}, "
      f"Hit Ratio = {hits/len(pages):.2f}, Miss Ratio = {misses/len(pages):.2f}")

if __name__ == "__main__":
    # Example page reference string
    reference_string = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
    frames = 3
    print("Reference String:", reference_string)
    print("Frames:", frames)
    fifo_page_replacement(reference_string, frames)
    lru_page_replacement(reference_string, frames)
```

Output:

```
Reference String: [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
Frames: 3

--- FIFO Page Replacement ---
Page: 7 -> Memory: [7]
Page: 0 -> Memory: [7, 0]
Page: 1 -> Memory: [7, 0, 1]
Page: 2 -> Memory: [0, 1, 2]
Page: 0 -> Memory: [0, 1, 2]
Page: 3 -> Memory: [1, 2, 3]
Page: 0 -> Memory: [2, 3, 0]
Page: 4 -> Memory: [3, 0, 4]
Page: 2 -> Memory: [0, 4, 2]
Page: 3 -> Memory: [4, 2, 3]
Page: 0 -> Memory: [2, 3, 0]
Page: 3 -> Memory: [2, 3, 0]
Page: 2 -> Memory: [2, 3, 0]

FIFO Results: Hits = 3, Misses = 10, Hit Ratio = 0.23, Miss Ratio = 0.77

--- LRU Page Replacement ---
Page: 7 -> Memory: [7]
Page: 0 -> Memory: [7, 0]
```

Page: 1 -> Memory: [7, 0, 1]
Page: 2 -> Memory: [0, 1, 2]
Page: 0 -> Memory: [1, 2, 0]
Page: 3 -> Memory: [2, 0, 3]
Page: 0 -> Memory: [2, 3, 0]
Page: 4 -> Memory: [3, 0, 4]
Page: 2 -> Memory: [0, 4, 2]
Page: 3 -> Memory: [4, 2, 3]
Page: 0 -> Memory: [2, 3, 0]
Page: 3 -> Memory: [2, 0, 3]
Page: 2 -> Memory: [0, 3, 2]

LRU Results: Hits = 4, Misses = 9, Hit Ratio = 0.31, Miss Ratio = 0.69

Input:

```
from collections import deque
def fifo_page_replacement(pages, frames):
    memory = deque()
    hits, misses = 0, 0

    for page in pages:
        if page in memory:
            hits += 1
        else:
            misses += 1
            if len(memory) >= frames:
                memory.popleft()  # remove oldest
            memory.append(page)
    hit_ratio = hits / len(pages)
    miss_ratio = misses / len(pages)
    return hits, misses, hit_ratio, miss_ratio

def lru_page_replacement(pages, frames):
    memory = []
    hits, misses = 0, 0
    for page in pages:
        if page in memory:
            hits += 1
            # Move to most recently used position
```

```
            memory.remove(page)
            memory.append(page)
        else:
            misses += 1
            if len(memory) >= frames:
                memory.pop(0)  # remove least recently used
            memory.append(page)
    hit_ratio = hits / len(pages)
    miss_ratio = misses / len(pages)
    return hits, misses, hit_ratio, miss_ratio

# Example usage
if __name__ == "__main__":
    # Example reference string
    reference_string = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
    frames = 3

    print("Reference string:", reference_string)
    print("Frames:", frames)

    fifo_res = fifo_page_replacement(reference_string, frames)
    lru_res = lru_page_replacement(reference_string, frames)
    print("\nFIFO:")
    print(f"Hits: {fifo_res[0]}, Misses: {fifo_res[1]}, "
          f"Hit Ratio: {fifo_res[2]:.2f}, Miss Ratio: {fifo_res[3]:.2f}")
    print("\nLRU:")
    print(f"Hits: {lru_res[0]}, Misses: {lru_res[1]}, "
          f"Hit Ratio: {lru_res[2]:.2f}, Miss Ratio: {lru_res[3]:.2f}")
```

Output:

Reference string: [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]
Frames: 3

FIFO:
Hits: 3, Misses: 10, Hit Ratio: 0.23, Miss Ratio: 0.77

LRU:
Hits: 4, Misses: 9, Hit Ratio: 0.31, Miss Ratio: 0.69

Practical No: 10

AIM:Disk Scheduling and Simple File System Designs

10.1: Simulate FCFS,SSTF,C-SCAN,C-LOOK,RSS for disk head movement

Input:

```
import random
class DiskScheduling:
  def __init__(self, requests, head, disk_size=200):
    self.requests = requests[:]  # make a copy to preserve the original list
    self.head = head
    self.disk_size = disk_size

  def fcfs(self):
    distance = 0
    head = self.head
    order = []
    for req in self.requests:
      distance += abs(head - req)
      order.append(req)
      head = req
    return order, distance

  def sstf(self):
    distance = 0
    head = self.head
    requests = self.requests[:]
    order = []
    while requests:
      closest = min(requests, key=lambda x: abs(x - head))
      distance += abs(head - closest)
      order.append(closest)
      head = closest
      requests.remove(closest)
    return order, distance

  def cscan(self):
    distance = 0
    head = self.head
    requests = sorted(self.requests)
    order = []
    right = [r for r in requests if r >= head]
    left = [r for r in requests if r < head]

    # Go rightwards till the end
    for r in right:
      distance += abs(head - r)
```

```
            order.append(r)
            head = r

        # Jump to the beginning of the disk and service left
        if left:
            distance += abs(self.disk_size - 1 - head)  # Go to end
            distance += self.disk_size - 1  # Jump to beginning
            head = 0

            for r in left:
                distance += abs(head - r)
                order.append(r)
                head = r
        return order, distance
    def clook(self):
        distance = 0
        head = self.head
        requests = sorted(self.requests)
        order = []
        right = [r for r in requests if r >= head]
        left = [r for r in requests if r < head]

        # Service right side
        for r in right:
            distance += abs(head - r)
            order.append(r)
            head = r

        # Jump to the smallest request on the left side
        if left:
            distance += abs(head - left[0])
            head = left[0]

            # Service left side
            for r in left:
                distance += abs(head - r)
                order.append(r)
                head = r
        return order, distance
    def rss(self):
        distance = 0
        head = self.head
        requests = self.requests[:]
        order = []
        random.shuffle(requests)

        for r in requests:
            distance += abs(head - r)
```

```
            order.append(r)
            head = r
        return order, distance

# Example Usage
if __name__ == "__main__":
    requests = [82, 170, 43, 140, 24, 16, 190]
    head = 50
    disk_size = 200
    algo = DiskScheduling(requests, head, disk_size)

    fcfs_order, fcfs_distance = algo.fcfs()
    print(f"FCFS order: {fcfs_order}")
    print(f"FCFS total distance: {fcfs_distance}")

    sstf_order, sstf_distance = algo.sstf()
    print(f"SSTF order: {sstf_order}")
    print(f"SSTF total distance: {sstf_distance}")

    clook_order, clook_distance = algo.clook()
    print(f"C-LOOK order: {clook_order}")
    print(f"C-LOOK total distance: {clook_distance}")

    cscan_order, cscan_distance = algo.cscan()
    print(f"C-SCAN order: {cscan_order}")
    print(f"C-SCAN total distance: {cscan_distance}")

    rss_order, rss_distance = algo.rss()
    print(f"RSS order: {rss_order}")
    print(f"RSS total distance: {rss_distance}")
```

Output:

```
FCFS order: [82, 170, 43, 140, 24, 16, 190]
FCFS total distance: 642
SSTF order: [43, 24, 16, 82, 140, 170, 190]
SSTF total distance: 208
C-LOOK order: [82, 140, 170, 190, 16, 24, 43]
C-LOOK total distance: 341
C-SCAN order: [82, 140, 170, 190, 16, 24, 43]
C-SCAN total distance: 391
RSS order: [16, 170, 140, 82, 24, 43, 190]
RSS total distance: 500
```

10.2:Design a basic file system structure with block allocation, directory management, and file operations (create, read,delete)

Input:

```
class FileSystem:
  def __init__(self, total_blocks=20, block_size=1):
    self.total_blocks = total_blocks
    self.blocks = block_size
    self.free_blocks = [True] * total_blocks  # True = free, False = occupied
    self.directory = {}  # (filename: {"size": size, "blocks": [block indices]})

  def allocate_blockd(self, num_blocks):
    """Allocate free blocks for a file. """
    allocated = []
    for i in range(self.total_blocks):
      if self.free_blocks[i]:
        allocated.append(i)
        if len(allocated) == num_blocks:
          for blk in allocated:
            self.free_blocks[blk] = False
          return allocated
    return None

  def create(self, filename, size):
    """Create a file with given size(in blocks)."""
    if filename in self.directory:
      print(f"Error: File '{filename}' already exists.")
      return

    num_blocks = (size + self.blocks) // self.blocks
    allocated = self.allocate_blockd(num_blocks)

    if allocated is None:
      print("Error: Not enough space to allocate file.")
    else:
      self.directory[filename] = {"size": size, "blocks": allocated}

  def read(self, filename):
    """Read file info."""
    if filename not in self.directory:
      print(f"Error: File '{filename}' not found.")
      return
    file_info = self.directory[filename]
    print(f"Reading file '{filename}':")
```

```
        print(f" -> Size: {file_info['size']} units")
        print(f" -> Books: {file_info['blocks']}")

    def delete(self, filename):
        """Delete a file and free its blocks."""
        if filename not in self.directory:
            print(f"File '{filename}' not found.")
            return
        for blk in self.directory[filename]['blocks']:
            self.free_blocks[blk] = True
        del self.directory[filename]
        print(f"File '{filename}' deleted successfully.")

    def show_directory(self):
        """Show all files and their block allocations."""
        if not self.directory:
            print("Directory is empty.")
            return
        print("Directory contents.")
        for fname, info in self.directory.items():
            print(f" -> {fname}:size={info['size']}, blocks={info['blocks']}")

    def show_free_blocks(self):
        """Show free/Used block status."""
        print("Block allocation status.")
        print("".join(["F" if free else "U" for free in self.free_blocks]))
```

Output:

```
fs=FileSystem(total_blocks=20,block_size=1)
fs.create('new.txt',3)
fs.read('new.txt')
Reading file 'new.txt':
 -> Size: 3 units
 -> Books: [0, 1, 2, 3]
fs.free_blocks
[False, False, False, False, True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True]
```