**Roll No ______________**

**Sonopant Dandekar Shikshan Mandali"s**

**Sonopant DandekarArts, V.S.Apte Commerce,**

**M.H. Mehta Science Collage**

**DEPARTMENT OF COMPUTER SCIENCE**

# CERTIFICATE

**Certified That Mr./Miss ______________________________________________________________**

**Of______________________________________________has Satisfactorily completed a course of**

**Necessary experiment in _______________________________________________________under**

**My supervision in the SY BSC Computer Science in the Year 2025 – 2026**

**Head of Department** **Subject Teacher**

**Date: / /2025**

sdsm 2025-26

INDEX

| SR. NO | AIM | PRATICAL DATE | SUBMIT DATE | SIGN |
|---|---|---|---|---|
| 1 | Process Communication using Shared Memory | | | |
| 2 | Process Communication using Message Passing | | | |
| 3 | Threading and Single Thread Control Flow | | | |
| 4 | Multi-threading and Fibonacci Generation | | | |
| 5 | Process Synchronization and Bounded Buffer Problem | | | |
| 6 | Readers-Writers Problem – Synchronization in Shared Access | | | |
| 7 | CPU Scheduling Algorithms (Part 1) – FCFS and Non-premptive Scheduling | | | |
| 8 | CPU Scheduling Algorithms (Part 2) – Round Robin | | | |

sdsm 2025-26

**Date:- / /**

# Pratical No 01

**Aim: Process Communication using Shared Memory**

A. Understand shared memory concepts in inter-process communication.
B. Implement producer-consumer synchronization using shared memory and semaphores.
C. Explore issues of race conditions and how to avoid them.

---

**A**

**Input:**

```
import multiprocessing
def writer(shared_value):
    print("Writer: Writing value 100 to shared memory.")
    shared_value.value=100
def reader(shared_value):
    print(f"Reader: Read value {shared_value.value}from shared memory.")
if __name__ =="__main__":
    shared_value=multiprocessing.Value('i',0)
    p1=multiprocessing.Process(target=writer, args=(shared_value,))
    p2=multiprocessing.Process(target=reader, args=(shared_value,))
    p1.start()
    p1.join()
    p2.start()
    p2.join()
```

**Output:**

```
C:\Users\student>cd desktop

C:\Users\student\Desktop>cd 65015OS

C:\Users\student\Desktop\65015OS>python practical1a.py
Writer: Writing value 100 to shared memory.
Reader: Read value 100from shared memory.
```

dsm 2025-26

**B]**

**Input:**

```
import multiprocessing
import time
import random
BUFFER_SIZE=5
def producer(buffer,empty,full,mutex):
    for i in range(10):
        item=random.randint(1,100)
        empty.acquire()
        mutex.acquire()
        buffer.append(item)
        print(f"[Producer] Produced:{item}|Buffer:{list(buffer)}")
        mutex.release()
        full.release()
        time.sleep(random.uniform(0.5,1.5))
def consumer(buffer,empty,full,mutex):
    for i in range(10):
        full.acquire()
        mutex.acquire()
        item=buffer.pop(0)
        print(f"[Consumer]Consumer:{item}|Buffer:{list(buffer)}")
        mutex.release()
        empty.release()
        time.sleep(random.uniform(0.5,2.0))
if __name__=="__main__":
    manager=multiprocessing.Manager()
    buffer=manager.list()
    empty=multiprocessing.Semaphore(BUFFER_SIZE)
    full=multiprocessing.Semaphore(0)
    mutex=multiprocessing.Lock()
```

sdsm 2025-26

```
producer_process=multiprocessing.Process(target=producer,args=(buffer,empty,full,mutex))
consumer_process=multiprocessing.Process(target=consumer,args=(buffer,empty,full,mutex))
    producer_process.start()
    consumer_process.start()
    producer_process.join()
    consumer_process.join()
    print("Processing complete.")
```

**Output:**

```
C:\Users\student\Desktop\65015OS>python practical1b.py
[Producer] Produced:3|Buffer:[3]
[Consumer]Consumer:3|Buffer:[]
[Producer] Produced:7|Buffer:[7]
[Consumer]Consumer:7|Buffer:[]
[Producer] Produced:87|Buffer:[87]
[Producer] Produced:98|Buffer:[87, 98]
[Consumer]Consumer:87|Buffer:[98]
[Producer] Produced:10|Buffer:[98, 10]
[Consumer]Consumer:98|Buffer:[10]
[Producer] Produced:11|Buffer:[10, 11]
[Consumer]Consumer:10|Buffer:[11]
[Producer] Produced:96|Buffer:[11, 96]
[Producer] Produced:71|Buffer:[11, 96, 71]
[Consumer]Consumer:11|Buffer:[96, 71]
[Producer] Produced:22|Buffer:[96, 71, 22]
[Consumer]Consumer:96|Buffer:[71, 22]
[Producer] Produced:77|Buffer:[71, 22, 77]
[Consumer]Consumer:71|Buffer:[22, 77]
[Consumer]Consumer:22|Buffer:[77]
[Consumer]Consumer:77|Buffer:[]
Processing complete.
```

**C]**

**Input:**

```
import multiprocessing
def increment(shared_counter):
    for _ in range(1000):
        shared_counter.value += 1
if __name__ == "__main__":
    counter = multiprocessing.Value('i', 0)
    p1 = multiprocessing.Process(target=increment, args=(counter,))
    p2 = multiprocessing.Process(target=increment, args=(counter,))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print(counter.value)
```

**Output:**

```
C:\Users\student\Desktop\65015OS>python practical1c.py
2000
```

sdsm 2025-26

**Date:-   /   /**

# Pratical No 02

## Aim: Process Communication using Message Passing

A. Use message queues/pipes to solve the producer-consumer problem.
B. Compare and contrast shared memory vs. message-passing approaches.
C. Analyze blocking vs. non-blocking communication.

---

## A.

**Input:**

```
import threading
import queue
import time
q = queue.Queue()
def producer():
    for i in range(5):
        q.put(i)
        print(f"Produced: {i}")
        time.sleep(0.5)
def consumer():
    while True:
        item = q.get()
        print(f"Consumed: {item}")
        q.task_done()
        if item == 4:
            break
threading.Thread(target=producer).start()
threading.Thread(target=consumer).start()
```

**OUTPUT:**

```
C:\65009>python p4.py
Produced: 0
Consumed: 0
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
```

sdsm 2025-26

**B**

**INPUT:**

```
from multiprocessing import Process, Queue, Value

# Shared memory using Value
def add_shared(val):
    for _ in range(5):
        val.value += 1

# Message passing using Queue
def add_queue(q):
    for i in range(5):
        q.put(i)

if __name__ == "__main__":
    from time import sleep
    v = Value('i', 0)
    q = Queue()

    p1 = Process(target=add_shared, args=(v,))
    p2 = Process(target=add_queue, args=(q,))

    p1.start(); p2.start()
    p1.join(); p2.join()

    print("Shared Memory Result:", v.value)
    print("Message Passing Result:", [q.get() for _ in range(q.qsize())])
```

**OUTPUT:**

```
C:\65009>python p5.py
Shared Memory Result: 5
Message Passing Result: [0, 1, 2, 3, 4]
```

sdsm 2025-26

**C**

**INPUT:**

```
from multiprocessing import Process, Queue
import time
def producer(q):
    q.put("Message")  # blocks if queue is full
def consumer(q):
    msg = q.get() # blocks until an item is available
    print("Blocking:", msg)
if __name__ == "__main__":
    q = Queue()
    Process(target=producer, args=(q,)).start()
    Process(target=consumer, args=(q,)).start()
```

**OUTPUT:**

```
C:\65009>python p7.py
Blocking: Message
```

sdsm 2025-26

**Date:-   /   /**

# Pratical No 03

## Aim: Threading and Single Thread Control Flow

A. Practice thread creation and basic thread lifecycle using standard libraries
B. Observe execution order, thread joining, and delays.
C. Measure execution time for sequential vs threaded execution

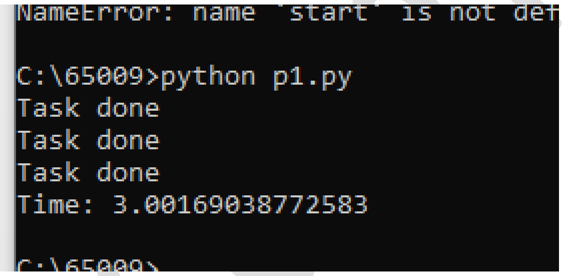
**A**

**Input:**

```
import threading
import time
def task():
    time.sleep(1)
    print("Task done")

start = time.time()
for _ in range(3):
    task()
print("Time:", time.time() - start)
```

**Output:**



```
NameError: name 'start' is not def

C:\65009>python p1.py
Task done
Task done
Task done
Time: 3.00169038772583

C:\65009>
```

Sem 2025-26

**B**

**Input:**

```
import threading
import time
def task():
    time.sleep(1)
    print("Thread task done")
start = time.time()
threads = [threading.Thread(target=task) for _ in range(3)]
for t in threads: t.start()
for t in threads: t.join()
print("Time:", time.time() - start)
```

**Output:**

```
C:\65009>python p1.py
Thread task done
Thread task done
Thread task done
Time: 1.0027854442596436
```

sdsm 2025-26

**C**

**Input:**

```
import threading
import time

def write_file(name):
    with open(name, "w") as f:
        for _ in range(50000):
            f.write("line\n")

start = time.time()
t1 = threading.Thread(target=write_file, args=("a.txt",))
t2 = threading.Thread(target=write_file, args=("b.txt",))
t1.start(); t2.start()
t1.join(); t2.join()
print("Time:", time.time() - start)
```

**Output:**

```
C:\65009>python p1.py
Time: 0.030933141708374023

C:\65009>
```

**Date:- / /**

# Pratical No 04

## Aim: Multi-threading and Fibonacci Generation

A. Implement multi-threading to generate and print Fibonacci sequences.
B. Explore thread safety, synchronization when accessing shared variables.
C. Introduce concepts of thread pooling and task delegation

A

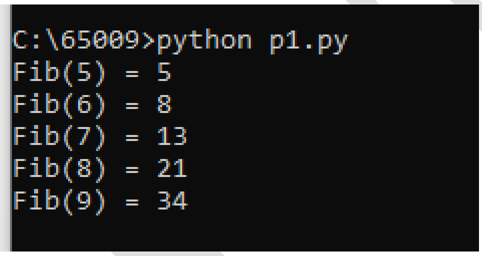
**Input:**

```
import threading
from concurrent.futures import ThreadPoolExecutor


def fib(n):
    a, b = 0, 1
    for _ in range(n): a, b = b, a + b
    print(f"Fib({n}) = {a}")


threads = [threading.Thread(target=fib, args=(i,)) for i in range(5, 10)]
for t in threads: t.start()
for t in threads: t.join()
```

**Output:**



```
C:\65009>python p1.py
Fib(5) = 5
Fib(6) = 8
Fib(7) = 13
Fib(8) = 21
Fib(9) = 34
```

Sam 2025-26

**B**

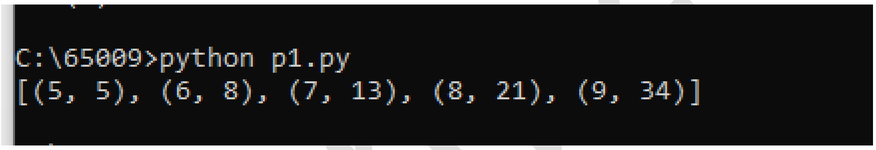
**Input:**

```
import threading
from concurrent.futures import ThreadPoolExecutor
lock = threading.Lock()
results = []

def fib_safe(n):
    a, b = 0, 1
    for _ in range(n): a, b = b, a + b
    with lock:
        results.append((n, a))

threads = [threading.Thread(target=fib_safe, args=(i,)) for i in range(5, 10)]
for t in threads: t.start()
for t in threads: t.join()
print(sorted(results))
```

**Output:**



```
C:\65009>python p1.py
[(5, 5), (6, 8), (7, 13), (8, 21), (9, 34)]
```

sdsm 2025-26

**C**

**Input:**

```
import threading
from concurrent.futures import ThreadPoolExecutor

def fib(n):
    a, b = 0, 1
    for _ in range(n): a, b = b, a + b
    return f"Fib({n}) = {a}"

with ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(fib, i) for i in range(5, 10)]
    for f in futures:
        print(f.result())
```

**Output:**

```
C:\65009>python p1.py
[(5, 5), (6, 8), (7, 13), (8, 21), (9, 34)]
```

sdsm 2025-26

Date:- / /

# Pratical No 05

## Aim: Process Synchronization and Bounded Buffer Problem

A. Simulate producer-consumer bounded buffer using mutex and semaphores.
B. Implement buffer control with synchronized access.
C. Introduce circular queue techniques for managing shared buffers.

A

**Input**

```
import threading
import time
import random

BUFFER_SIZE = 5
buffer = [None] * BUFFER_SIZE
in_index = out_index = 0
buffer_lock = threading.Condition()

NUM_ITEMS = 10  # Total number of items to produce/consume
stop_event = threading.Event()

def producer():
    global in_index
    for _ in range(NUM_ITEMS):
        item = random.randint(1, 100)
        with buffer_lock:
            while (in_index + 1) % BUFFER_SIZE == out_index:  # Buffer full
                buffer_lock.wait()
            buffer[in_index] = item
            print(f"Produced: {item} at position {in_index}")
            in_index = (in_index + 1) % BUFFER_SIZE
            buffer_lock.notify_all()
        time.sleep(random.uniform(0.1, 0.5))

    # Signal that producer is done
    with buffer_lock:
        stop_event.set()
        buffer_lock.notify_all()

def consumer():
    global out_index
    count = 0
    while count < NUM_ITEMS:
        with buffer_lock:
            while in_index == out_index:
                if stop_event.is_set():
                    return  # Exit if producer is done and buffer is empty
                buffer_lock.wait()
```

sdsm 2025-26

```
            item = buffer[out_index]
            buffer[out_index] = None
            print(f"Consumed: {item} from position {out_index}")
            out_index = (out_index + 1) % BUFFER_SIZE
            count += 1
            buffer_lock.notify_all()
        time.sleep(random.uniform(0.1, 0.5))

prod_thread = threading.Thread(target=producer)
cons_thread = threading.Thread(target=consumer)

prod_thread.start()
cons_thread.start()

prod_thread.join()
cons_thread.join()

print("Processing complete.")
```

**Output:**

```
(c) Microsoft Corporation. All rights rese

C:\65009>python p1.py
Produced: 90 at position 0
Consumed: 90 from position 0
Produced: 74 at position 1
Consumed: 74 from position 1
Produced: 31 at position 2
Consumed: 31 from position 2
Produced: 31 at position 3
Consumed: 31 from position 3
Produced: 68 at position 4
Consumed: 68 from position 4
Produced: 16 at position 0
Consumed: 16 from position 0
Produced: 6 at position 1
Consumed: 6 from position 1
Produced: 37 at position 2
Consumed: 37 from position 2
Produced: 84 at position 3
Produced: 82 at position 4
Consumed: 84 from position 3
Consumed: 82 from position 4
Processing complete.
```

2025-26

**B**

**Input**

```
import threading
import time
import random

BUFFER_SIZE = 5
NUM_ITEMS = 10  # Total items to produce and consume

buffer = [None] * BUFFER_SIZE
in_index = out_index = 0
mutex = threading.Lock()
empty_slots = threading.Semaphore(BUFFER_SIZE)
filled_slots = threading.Semaphore(0)

produced_count = 0
consumed_count = 0
done_event = threading.Event()

def producer():
    global in_index, produced_count
    for _ in range(NUM_ITEMS):
        item = random.randint(1, 100)
        empty_slots.acquire()
        with mutex:
            buffer[in_index] = item
            print(f"Produced: {item} at position {in_index}")
            in_index = (in_index + 1) % BUFFER_SIZE
            produced_count += 1
        filled_slots.release()
        time.sleep(random.uniform(0.1, 1))
    done_event.set()  # Signal that production is done

def consumer():
    global out_index, consumed_count
    while True:
        # Exit if everything is consumed and producer is done
        if done_event.is_set() and filled_slots._value == 0:
            break

        filled_slots.acquire()
        with mutex:
            item = buffer[out_index]
            print(f"Consumed: {item} from position {out_index}")
            buffer[out_index] = None
            out_index = (out_index + 1) % BUFFER_SIZE
            consumed_count += 1
        empty_slots.release()
        time.sleep(random.uniform(0.1, 1))

        if consumed_count >= NUM_ITEMS:
```

sdsm 2025-26

```
        break

# Creating threads
prod_thread = threading.Thread(target=producer)
cons_thread = threading.Thread(target=consumer)

prod_thread.start()
cons_thread.start()

prod_thread.join()
cons_thread.join()

print("Processing complete.")
```

**Output**

```
C:\65009>
C:\65009>python p2.py
Produced: 87 at position 0
Consumed: 87 from position 0
Produced: 29 at position 1
Consumed: 29 from position 1
Produced: 68 at position 2
Consumed: 68 from position 2
Produced: 14 at position 3
Consumed: 14 from position 3
Produced: 39 at position 4
Consumed: 39 from position 4
Produced: 83 at position 0
Consumed: 83 from position 0
Produced: 89 at position 1
Consumed: 89 from position 1
Produced: 16 at position 2
Produced: 69 at position 3
Consumed: 16 from position 2
Produced: 91 at position 4
Consumed: 69 from position 3
Consumed: 91 from position 4
Processing complete.
```

**C**

**Input**

```
import threading
import time
import random

BUFFER_SIZE = 5
NUM_ITEMS = 10  # Total items to produce and consume
buffer = [None] * BUFFER_SIZE
in_index = out_index = 0
mutex = threading.Lock()
empty_slots = threading.Semaphore(BUFFER_SIZE)
filled_slots = threading.Semaphore(0)
produced_count = 0
consumed_count = 0
done_event = threading.Event()

def producer():
    global in_index, produced_count
    for _ in range(NUM_ITEMS):
        item = random.randint(1, 100)
        empty_slots.acquire()
        with mutex:
            buffer[in_index] = item
            print(f"Produced: {item} at position {in_index}")
            in_index = (in_index + 1) % BUFFER_SIZE
            produced_count += 1
        filled_slots.release()
        time.sleep(random.uniform(0.1, 1))
    done_event.set()  # Signal that production is done

def consumer():
    global out_index, consumed_count
    while True:

        # Exit if everything is consumed and producer is done
        if done_event.is_set() and filled_slots._value == 0:
            break
        filled_slots.acquire()
        with mutex:
            item = buffer[out_index]
            print(f"Consumed: {item} from position {out_index}")
            buffer[out_index] = None
            out_index = (out_index + 1) % BUFFER_SIZE
            consumed_count += 1
        empty_slots.release()
        time.sleep(random.uniform(0.1, 1))
        if consumed_count >= NUM_ITEMS:
            break
```

sdsm 2025-26

```python
# Creating threads
prod_thread = threading.Thread(target=producer)
cons_thread = threading.Thread(target=consumer)
prod_thread.start()
cons_thread.start()
prod_thread.join()
cons_thread.join()

print("Processing complete.")
```

**Output**

```
C:\65009>
C:\65009>python p3.py
Produced: 13 at position 0
Consumed: 13 from position 0
Produced: 47 at position 1
Consumed: 47 from position 1
Produced: 12 at position 2
Consumed: 12 from position 2
Produced: 49 at position 3
Consumed: 49 from position 3
Produced: 45 at position 4
Consumed: 45 from position 4
Produced: 25 at position 0
Consumed: 25 from position 0
Produced: 7 at position 1
Produced: 31 at position 2
Consumed: 7 from position 1
Produced: 4 at position 3
Produced: 57 at position 4
Consumed: 31 from position 2
Consumed: 4 from position 3
Consumed: 57 from position 4
Processing complete.

C:\65009>
```

**Date:- / /**

# Pratical No 06

**Aim: Readers-Writers Problem – Synchronization in Shared Access**

A. Implement reader and writer prioritization.
B. Use semaphores to allow multiple readers or exclusive writer access.
C. Extend to fairness in access and deadlock prevention.

**A**

**Input**

```
import threading
import time

mutex = threading.Semaphore(1)
rw_mutex = threading.Semaphore(1)
read_count = 0

def reader(id):
    global read_count
    time.sleep(0.1 * id)
    mutex.acquire()
    read_count += 1
    if read_count == 1:
        rw_mutex.acquire()

    mutex.release()
    print(f"Reader {id} is reading")
    time.sleep(1)  # Simulate reading
    mutex.acquire()
    read_count -= 1
    if read_count == 0:
        rw_mutex.release()
    mutex.release()
def writer(id):
    time.sleep(0.2 * id)
    rw_mutex.acquire()
    print(f"Writer {id} is writing")
    time.sleep(1.5)  # Simulate writing
    rw_mutex.release()

threads = []
for i in range(3):
    t = threading.Thread(target=reader, args=(i+1,))
    threads.append(t)
for i in range(2):
    t = threading.Thread(target=writer, args=(i+1,))
    threads.append(t)
for t in threads:
    t.start()
```

sksm 2025-26

```
for t in threads:
    t.join()

print("All operations complete.")
```

**Output**

```
C:\65009>python py6a.py
Reader 1 is reading
Reader 2 is reading
Reader 3 is reading
Writer 1 is writing
Writer 2 is writing
All operations complete.
```

sdsm 2025-26

**B**

**Input**

```
import threading
import time
mutex = threading.Semaphore(1)
rw_mutex = threading.Semaphore(1)
wrt_mutex = threading.Semaphore(1)
read_count = 0
def reader(id):
    global read_count
    time.sleep(0.1 * id)  # Staggered start
    wrt_mutex.acquire()
    mutex.acquire()
    read_count += 1
    if read_count == 1:
        rw_mutex.acquire()
    mutex.release()
    wrt_mutex.release()
    print(f"Reader {id} is reading")
    time.sleep(1)
    mutex.acquire()
    read_count -= 1
    if read_count == 0:
        rw_mutex.release()
    mutex.release()
def writer(id):
    time.sleep(0.2 * id)
    wrt_mutex.acquire()
    rw_mutex.acquire()
    print(f"Writer {id} is writing")
    time.sleep(1.5)
    rw_mutex.release()
    wrt_mutex.release()
threads = []
for i in range(3):
    t = threading.Thread(target=reader, args=(i+1,))
    threads.append(t)
for i in range(2):
    t = threading.Thread(target=writer, args=(i+1,))
    threads.append(t)
for t in threads:
    t.start()
for t in threads:
    t.join()
print("All operations complete.")
```

**Output**

```
C:\65009>python p6b.py
Reader 1 is reading
Reader 2 is reading
Writer 1 is writing
Reader 3 is reading
Writer 2 is writing
All operations complete.
```

sdsm 2025-26

**C**

**Input**

```
import threading
import time
mutex = threading.Lock()
queue = threading.Semaphore(1)
rw_mutex = threading.Semaphore(1)
read_count = 0
def reader(id):
    global read_count
    time.sleep(0.1 * id)  # Staggered start
    queue.acquire()
    mutex.acquire()
    read_count += 1
    if read_count == 1:
        rw_mutex.acquire()
    mutex.release()
    queue.release()
    print(f"Reader {id} is reading")
    time.sleep(1)
    mutex.acquire()
    read_count -= 1
    if read_count == 0:
        rw_mutex.release()
    mutex.release()
def writer(id):
    time.sleep(0.2 * id)  # Staggered start
    queue.acquire()
    rw_mutex.acquire()
    print(f"Writer {id} is writing")
    time.sleep(1.5)
    rw_mutex.release()
    queue.release()
threads = []
for i in range(3):
    t = threading.Thread(target=reader, args=(i+1,))
    threads.append(t)
for i in range(2):
    t = threading.Thread(target=writer, args=(i+1,))
    threads.append(t)
for t in threads:
    t.start()
for t in threads:
    t.join()
print("All operations complete.")
```

**Output**

```
C:\65009>python py6.py
Reader 1 is reading
Reader 2 is reading
Writer 1 is writing
Reader 3 is reading
Writer 2 is writing
All operations complete.
```

sdsm 2025-26

Date:-   /   /

# Pratical No 07

**Aim: CPU Scheduling Algorithms (Part 1) – FCFS and Non-preemptive Scheduling**

A. Simulate First-Come First-Serve scheduling.
B. Extend implementation to general non-preemptive scheduling.
C. Analyze waiting time, turnaround time, and Gantt chart generation

**A**

**Input**

```
processes = [(1, 0, 5), (2, 2, 3), (3, 4, 1)]  # (PID, Arrival, Burst)
processes.sort(key=lambda x: x[1])  # Sort by Arrival Time

time = 0
gantt = []
for pid, at, bt in processes:
    if time < at:
        time = at
    start = time
    time += bt
    end = time
    gantt.append((pid, start, end))
    print(f"P{pid}: Waiting={start - at}, Turnaround={end - at}")

print("Gantt:", gantt)
```

**Output**

```
C:\65009>python p1.py
P1: Waiting=0, Turnaround=5
P2: Waiting=3, Turnaround=6
P3: Waiting=4, Turnaround=5
Gantt: [(1, 0, 5), (2, 5, 8), (3, 8, 9)]

C:\65009>
```

**B**

**Input**

```
processes = [(1, 0, 5), (2, 2, 3), (3, 4, 1)]
done = []
time = 0
gantt = []

while len(done) < len(processes):
    ready = [p for p in processes if p[1] <= time and p not in done]
    if not ready:
        time += 1
        continue
    p = min(ready, key=lambda x: x[2])  # Pick shortest burst
    start = time
    time += p[2]
    end = time
    done.append(p)
    gantt.append((p[0], start, end))
    print(f"P{p[0]}: Waiting={start - p[1]}, Turnaround={end - p[1]}")

print("Gantt:", gantt)
```

**Output**

```
C:\65009>python p1.py
P1: Waiting=0, Turnaround=5
P3: Waiting=1, Turnaround=2
P2: Waiting=4, Turnaround=7
Gantt: [(1, 0, 5), (3, 5, 6), (2, 6, 9)]

C:\65009>_
```

sdsm 2025-26

**C**

**Input**

```
processes = [(1, 0, 4, 2), (2, 1, 3, 1), (3, 2, 1, 3)]  # (PID, AT, BT, Priority)
done = []
time = 0
gantt = []

while len(done) < len(processes):
    ready = [p for p in processes if p[1] <= time and p not in done]
    if not ready:
        time += 1
        continue
    p = min(ready, key=lambda x: x[3])  # Lower number = higher priority
    start = time
    time += p[2]
    end = time
    done.append(p)
    gantt.append((p[0], start, end))
    print(f"P{p[0]}: Waiting={start - p[1]}, Turnaround={end - p[1]}")

print("Gantt:", gantt)
```

**Output**

```
C:\65009>python p1.py
P1: Waiting=0, Turnaround=4
P2: Waiting=3, Turnaround=6
P3: Waiting=5, Turnaround=6
Gantt: [(1, 0, 4), (2, 4, 7), (3, 7, 8)]
```

sdsm 2025-26

Date:- / /

# Pratical No 08

## Aim: CPU Scheduling Algorithms (Part 2) – Round Robin

A. Implement Round Robin scheduling with configurable time quantum.
B. Compare with FCFS: fairness, turnaround, response time.
C. Track context switches and improve queue management.

**A**

**Input**

```
from collections import deque
processes = [(1, 0, 5), (2, 1, 3), (3, 2, 8)]
quantum = 2
queue = deque()
time = 0
remaining = {pid: bt for pid, at, bt in processes}
arrival = {pid: at for pid, at, bt in processes}
done = set()
gantt = []
response = {}
while len(done) < len(processes):
    for pid, at, bt in processes:
        if at <= time and pid not in queue and pid not in done and remaining[pid] > 0:
            queue.append(pid)
    if queue:
        pid = queue.popleft()
        if pid not in response:
            response[pid] = time - arrival[pid]
        exec_time = min(quantum, remaining[pid])
        gantt.append((pid, time, time + exec_time))
        time += exec_time
        remaining[pid] -= exec_time
        if remaining[pid] > 0:
            queue.append(pid)
        else:
            done.add(pid)
    else:
        time += 1
for pid, at, bt in processes:
    ct = next(end for p, _, end in reversed(gantt) if p == pid)
    tat = ct - at
    wt = tat - bt
    print(f"P{pid}: Waiting={wt}, Turnaround={tat}, Response={response[pid]}")
print("Gantt:", gantt)
```

**Output**

```
C:\65009>python p1.py
P1: Waiting=4, Turnaround=9, Response=0
P2: Waiting=6, Turnaround=9, Response=3
P3: Waiting=6, Turnaround=14, Response=4
Gantt: [(1, 0, 2), (1, 2, 4), (2, 4, 6), (3, 6, 8), (1, 8, 9), (2, 9, 10), (3, 10, 12), (3, 12, 14), (3, 14, 16)]
```

sdsm 2025-26

**B**
**Input**

```
processes = [(1, 0, 5), (2, 1, 3), (3, 2, 8)]
quantum = 2

def fcfs():
    time = 0
    gantt = []
    for pid, at, bt in sorted(processes, key=lambda x: x[1]):
        if time < at:
            time = at
        start = time
        time += bt
        gantt.append((pid, start, time))
    return gantt

def rr():
    from collections import deque
    time = 0
    queue = deque()
    remaining = {pid: bt for pid, at, bt in processes}
    arrival = {pid: at for pid, at, bt in processes}
    done, gantt = set(), []
    while len(done) < len(processes):
        for pid, at, _ in processes:
            if at <= time and pid not in queue and pid not in done and remaining[pid] > 0:
                queue.append(pid)
        if queue:
            pid = queue.popleft()
            exec_time = min(quantum, remaining[pid])
            gantt.append((pid, time, time + exec_time))
            time += exec_time
            remaining[pid] -= exec_time
            if remaining[pid] > 0:
                queue.append(pid)
            else:
                done.add(pid)
        else:
            time += 1
    return gantt
print("FCFS Gantt:", fcfs())
print("RR Gantt:", rr())
```

**Output**

```
C:\65009>python p1.py
FCFS Gantt: [(1, 0, 5), (2, 5, 8), (3, 8, 16)]
RR Gantt: [(1, 0, 2), (1, 2, 4), (2, 4, 6), (3, 6, 8), (1, 8, 9), (2, 9, 10), (3, 10, 12), (3, 12, 14), (3, 14, 16)]

C:\65009>
```

sdsm 2025-26

**C**
**Input**

```
from collections import deque

processes = [(1, 0, 4), (2, 1, 3), (3, 2, 5)]
quantum = 2
time = 0
queue = deque()
remaining = {pid: bt for pid, at, bt in processes}
arrival = {pid: at for pid, at, bt in processes}
done = set()
visited = set()
gantt = []
prev_pid = None
context_switches = 0

while len(done) < len(processes):
    for pid, at, bt in processes:
        if at <= time and pid not in visited and pid not in done:
            queue.append(pid)
            visited.add(pid)

    if queue:
        pid = queue.popleft()
        if pid != prev_pid:
            context_switches += 1
        prev_pid = pid
        exec_time = min(quantum, remaining[pid])
        gantt.append((pid, time, time + exec_time))
        time += exec_time
        remaining[pid] -= exec_time
        for pid2, at2, _ in processes:
            if at2 <= time and pid2 not in visited and pid2 not in done:
                queue.append(pid2)
                visited.add(pid2)
        if remaining[pid] > 0:
            queue.append(pid)
        else:
            done.add(pid)
    else:
        time += 1

print("Gantt Chart:", gantt)
print("Context Switches:", context_switches - 1)  # First one doesn't count
```

**Output**

```
C:\65009>python p1.py
Gantt Chart: [(1, 0, 2), (2, 2, 4), (3, 4, 6), (1, 6, 8), (2, 8, 9), (3, 9, 11), (3, 11, 12)]
Context Switches: 5
```