RUSTAMJI INSTITUTE OF TECHNOLOGY

BSF ACADEMY, TEKANPUR

Practical File for

CS405 (OPERATING SYSTEM)

(Session: Jan-Jun 2025)

Submitted by

Shani Bharadwaj (0902cs231105)

B.Tech. 4th Semester (2023-2027 batch)

Department of Computer Science & Engineering

Subject Teacher

Ms. Aishwarya Sharma

TABLE OF CONTENTS

Sr.No	List of Programs
1.	Write a program to implement FCFS CPU Scheduling
2.	Write a program to implement SJF CPU Scheduling
3.	Write a program to implement Priority CPU Scheduling
4.	Write a program to implement Round Robin CPU Scheduling
5.	Write a program to compare various CPU scheduling algorithms over different scheduling criteria
6.	Write a program to implement Producer consumer problem
7.	Write a program to implement Reader writer problem
8.	Write a program to implement Dining Philosophers problem
9.	Write a program to implementation and compare various page replacement algorithms
10.	Write a program to implementation and compare various disk and drum scheduling algorithm

EXPERIMENT 1

Program Description : Write a program to implement FCFS CPU Scheduling

Solution:

ALGORITHM

1. Start
2. Input the number of processes n.
3. For each process i = 0 to n-1:
    - Input burst time BT[i].
4. Set WT[0] = 0 (first process has no waiting time).
5. For i = 1 to n-1:
    - WT[i] = WT[i-1] + BT[i-1]
6. For each process i = 0 to n-1:
    - TAT[i] = BT[i] + WT[i]
7. Calculate total waiting time and total turnaround time.
8. Calculate average waiting time and average turnaround time.
9. Display process number, burst time, waiting time, and turnaround time.
10. End

SOURCE CODE

```
#include <stdio.h>

int main() {
   int n, i;
   int burstTime[100], waitingTime[100], turnAroundTime[100];
   float totalWT = 0, totalTAT = 0;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   // Input burst times
   for (i = 0; i < n; i++) {
      printf("Enter burst time for Process %d: ", i + 1);
      scanf("%d", &burstTime[i]);
   }
```

```
    // First process has 0 waiting time
    waitingTime[0] = 0;

    // Calculate waiting time
    for (i = 1; i < n; i++) {
        waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
    }

    // Calculate turnaround time
    for (i = 0; i < n; i++) {
        turnAroundTime[i] = burstTime[i] + waitingTime[i];
        totalWT += waitingTime[i];
        totalTAT += turnAroundTime[i];
    }



// Output
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\n", i + 1, burstTime[i], waitingTime[i],
turnAroundTime[i]);
    }

    printf("\nAverage Waiting Time = %.2f", totalWT / n);
    printf("\nAverage Turnaround Time = %.2f\n", totalTAT / n);

    return 0;
}
```

OUTPUT

```
Enter the number of processes: 3
Enter burst time for Process 1: 5
Enter burst time for Process 2: 3
Enter burst time for Process 3: 8

Process Burst Time      Waiting Time    Turnaround Time
P1      5               0               5
P2      3               5               8
P3      8               8               16

Average Waiting Time = 4.33
Average Turnaround Time = 9.67
```

EXPERIMENT 2

Program Description : Write a program to implement SJF CPU Scheduling

Solution:

ALGORITHM

1. Start
2. Input number of processes n.
3. For each process i = 0 to n-1, input burst time BT[i] and assign process ID.
4. Sort the processes according to burst time in ascending order.
5. Set WT[0] = 0
6. For i = 1 to n-1:
   → WT[i] = WT[i-1] + BT[i-1]
7. For each process i = 0 to n-1:
   → TAT[i] = BT[i] + WT[i]
8. Calculate average WT and TAT.
9. Display process ID, BT, WT, and TAT.
10. End

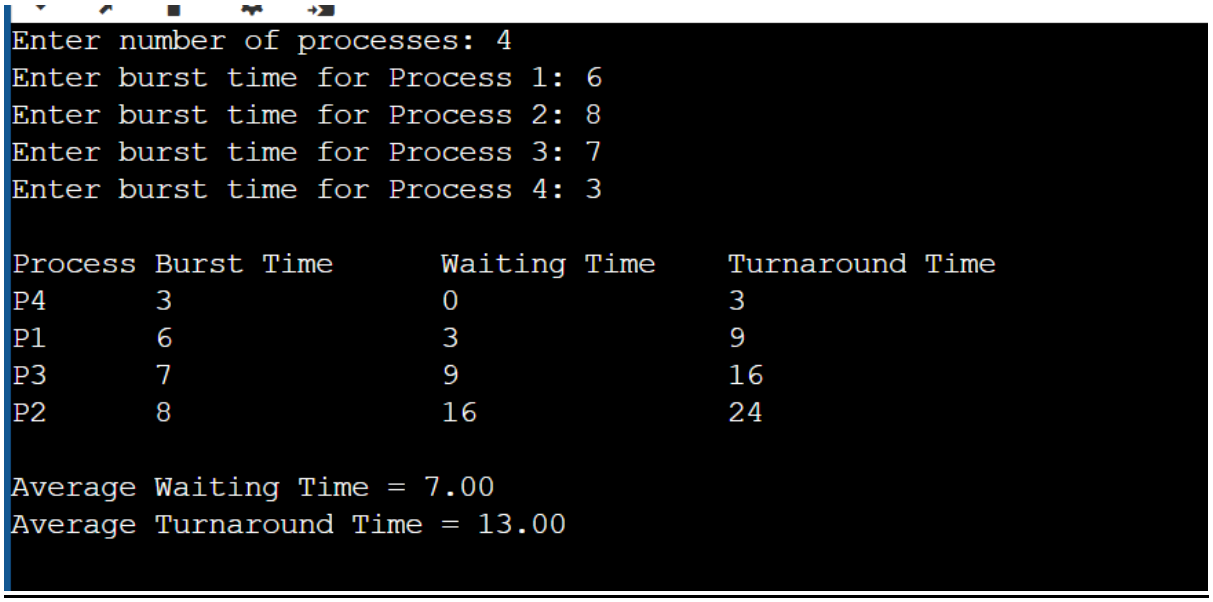
SOURCE CODE

```
#include <stdio.h>

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int n, i, j;
   int process[20], burstTime[20], waitingTime[20], turnAroundTime[20];
   float totalWT = 0, totalTAT = 0;

   printf("Enter number of processes: ");
   scanf("%d", &n);

   // Input burst times and assign process IDs
   for (i = 0; i < n; i++) {
```

```
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &burstTime[i]);
        process[i] = i + 1;
    }

    // Sort processes by burst time (Selection Sort)
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (burstTime[i] > burstTime[j]) {
                swap(&burstTime[i], &burstTime[j]);
                swap(&process[i], &process[j]);
            }
        }
    }

    // First process has 0 waiting time
    waitingTime[0] = 0;

    // Calculate waiting time
    for (i = 1; i < n; i++) {
        waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
    }

    // Calculate turnaround time
    for (i = 0; i < n; i++) {
        turnAroundTime[i] = burstTime[i] + waitingTime[i];
        totalWT += waitingTime[i];
        totalTAT += turnAroundTime[i];
    }

    // Output
    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\t\t%d\n", process[i], burstTime[i], waitingTime[i],
turnAroundTime[i]);
    }

    printf("\nAverage Waiting Time = %.2f", totalWT / n);
    printf("\nAverage Turnaround Time = %.2f\n", totalTAT / n);
```

```
    return 0;
}
```

OUTPUT



```
Enter number of processes: 4
Enter burst time for Process 1: 6
Enter burst time for Process 2: 8
Enter burst time for Process 3: 7
Enter burst time for Process 4: 3

Process Burst Time      Waiting Time    Turnaround Time
P4      3               0               3
P1      6               3               9
P3      7               9               16
P2      8               16              24

Average Waiting Time = 7.00
Average Turnaround Time = 13.00
```

EXPERIMENT 3

Program Description : Write a program to implement Priority CPU Scheduling

Solution:

ALGORITHM

1. Start
2. Input number of processes n
3. For each process i = 0 to n-1:
    - Input burst time BT[i]
    - Input priority PR[i]
    - Assign process ID
4. Sort processes based on priority (ascending)
5. Set WT[0] = 0
6. For i = 1 to n-1:
   → WT[i] = WT[i-1] + BT[i-1]
7. For each process i = 0 to n-1:
   → TAT[i] = BT[i] + WT[i]
8. Calculate and print average WT and TAT
9. Display all process details
10. End

Source Code

```
#include <stdio.h>

void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int n, i, j;
   int process[20], burstTime[20], priority[20];
   int waitingTime[20], turnAroundTime[20];
   float totalWT = 0, totalTAT = 0;

   printf("Enter number of processes: ");
```

```
    scanf("%d", &n);

    // Input burst time and priority
    for (i = 0; i < n; i++) {
        process[i] = i + 1;
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &burstTime[i]);
        printf("Enter priority for Process %d (lower number = higher priority): ", i +
1);
        scanf("%d", &priority[i]);
    }

    // Sort by priority (Selection Sort)
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (priority[i] > priority[j]) {
                swap(&priority[i], &priority[j]);
                swap(&burstTime[i], &burstTime[j]);
                swap(&process[i], &process[j]);
            }
        }
    }

    // First process has 0 waiting time
    waitingTime[0] = 0;

    // Calculate waiting time
    for (i = 1; i < n; i++) {
        waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];
    }

    // Calculate turnaround time
    for (i = 0; i < n; i++) {
        turnAroundTime[i] = burstTime[i] + waitingTime[i];
        totalWT += waitingTime[i];
        totalTAT += turnAroundTime[i];
    }

    // Output
    printf("\nProcess\tBT\tPriority\tWT\tTAT\n");
```

```
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t\t%d\t%d\n", process[i], burstTime[i], priority[i],
waitingTime[i], turnAroundTime[i]);
    }

    printf("\nAverage Waiting Time = %.2f", totalWT / n);
    printf("\nAverage Turnaround Time = %.2f\n", totalTAT / n);

    return 0;
}
```

OUTPUT

```
Enter number of processes: 3
Enter burst time for Process 1: 10
Enter priority for Process 1 (lower number = higher priority): 2
Enter burst time for Process 2: 5
Enter priority for Process 2 (lower number = higher priority): 1
Enter burst time for Process 3: 8
Enter priority for Process 3 (lower number = higher priority): 3

Process BT      Priority        WT      TAT
P2      5       1               0       5
P1      10      2               5       15
P3      8       3               15      23

Average Waiting Time = 6.67
Average Turnaround Time = 14.33
```

EXPERIMENT 4

Program Description : Write a program to implement Round Robin CPU Scheduling

Solution:

ALGORITHM

1. Start
2. Input number of processes n and time quantum q
3. For each process i = 0 to n-1, input burst time BT[i]
4. Initialize RT[i] = BT[i] (Remaining Time)
5. Initialize time = 0
6. Repeat while not all processes are complete:
    - For each process i:
        - If RT[i] > 0:
            - If RT[i] > q:
                - time += q, RT[i] -= q
            - Else:
                - time += RT[i], WT[i] = time - BT[i], RT[i] = 0
7. For each process i:
   → TAT[i] = BT[i] + WT[i]
8. Calculate average WT and TAT
9. End

SOURCE CODE

```
#include <stdio.h>

int main() {
   int n, i, time = 0, tq, remaining;
   int BT[100], RT[100], WT[100], TAT[100];
   float totalWT = 0, totalTAT = 0;

   printf("Enter the number of processes: ");
   scanf("%d", &n);

   // Input burst times
   for (i = 0; i < n; i++) {
      printf("Enter burst time for Process %d: ", i + 1);
```

```
    scanf("%d", &BT[i]);
    RT[i] = BT[i]; // Initialize remaining time
}

printf("Enter time quantum: ");
scanf("%d", &tq);

remaining = n;

// Round Robin logic
while (remaining != 0) {
    for (i = 0; i < n; i++) {
        if (RT[i] > 0) {
            if (RT[i] > tq) {
                time += tq;
                RT[i] -= tq;
            } else {
                time += RT[i];
                WT[i] = time - BT[i];
                RT[i] = 0;
                remaining--;
            }
        }
    }
}

// Calculate Turnaround Time
for (i = 0; i < n; i++) {
    TAT[i] = BT[i] + WT[i];
    totalWT += WT[i];
    totalTAT += TAT[i];
}

// Output
printf("\nProcess\tBT\tWT\tTAT\n");
for (i = 0; i < n; i++) {
    printf("P%d\t%d\t%d\t%d\n", i + 1, BT[i], WT[i], TAT[i]);
}

printf("\nAverage Waiting Time = %.2f", totalWT / n);
```

```
    printf("\nAverage Turnaround Time = %.2f\n", totalTAT / n);

    return 0;
}
```

OUTPUT

```
Enter the number of processes: 3
Enter burst time for Process 1: 10
Enter burst time for Process 2: 5
Enter burst time for Process 3: 8
Enter time quantum: 4

Process BT      WT      TAT
P1      10      13      23
P2      5       12      17
P3      8       13      21

Average Waiting Time = 12.67
Average Turnaround Time = 20.33
```

EXPERIMENT 5

Program Description : Write a program to compare various CPU scheduling algorithms over different scheduling criteria

Solution:

ALGORITHM

Input:

- Number of processes n
- For each process:
  - Burst Time BT[i]
  - Priority P[i]
- Time Quantum TQ (for Round Robin)

Step-by-Step Algorithm:

1. Start

2. Input Section

- Input total number of processes n
- For each process:
  - Assign Process ID PID = i + 1
  - Input Burst Time BT[i]
  - Input Priority P[i]
- Input Time Quantum TQ for Round Robin

3. Copy Input Data

- Copy original process data into separate arrays/lists for each algorithm (FCFS, SJF, Priority, RR) to avoid modifying shared data.

4. FCFS Scheduling

1. Set WT[0] = 0
2. For i = 1 to n-1:
   - WT[i] = WT[i-1] + BT[i-1]
3. For each process i:
   - TAT[i] = BT[i] + WT[i]
4. Compute and print Average WT and Average TAT

5. SJF Scheduling (Non-Preemptive)

1. Sort all processes in increasing order of BT[i]

2. Set WT[0] = 0
3. For i = 1 to n-1:
    - WT[i] = WT[i-1] + BT[i-1]
4. For each process i:
    - TAT[i] = BT[i] + WT[i]
5. Compute and print Average WT and Average TAT

6. Priority Scheduling (Non-Preemptive)

1. Sort all processes in increasing order of Priority (lower value = higher priority)
2. Set WT[0] = 0
3. For i = 1 to n-1:
    - WT[i] = WT[i-1] + BT[i-1]
4. For each process i:
    - TAT[i] = BT[i] + WT[i]
5. Compute and print Average WT and Average TAT

7. Round Robin Scheduling

1. Set RemainingTime[i] = BT[i] for all i
2. Initialize time = 0 and remaining = n
3. Repeat while remaining ≠ 0:
    - For i = 0 to n-1:
        - If RemainingTime[i] > 0:
            - If RemainingTime[i] > TQ:
                - time += TQ
                - RemainingTime[i] -= TQ
            - Else:
                - time += RemainingTime[i]
                - WT[i] = time - BT[i]
                - RemainingTime[i] = 0
                - remaining--
4. For each process i:
    - TAT[i] = BT[i] + WT[i]
5. Compute and print Average WT and Average TAT

8. End

SOURCE CODE

```
#include <stdio.h>

#define MAX 10

typedef struct {
   int pid, bt, priority;
} Process;

void calculateAverageTimes(int n, int wt[], int tat[], int bt[], float* avgWT, float*
avgTAT) {
   int i;
   *avgWT = *avgTAT = 0;
   for (i = 0; i < n; i++) {
      tat[i] = bt[i] + wt[i];
      *avgWT += wt[i];
      *avgTAT += tat[i];
   }
   *avgWT /= n;
   *avgTAT /= n;
}

// FCFS Scheduling
void FCFS(Process p[], int n) {
   int wt[MAX] = {0}, tat[MAX];
   for (int i = 1; i < n; i++) {
      wt[i] = wt[i - 1] + p[i - 1].bt;
   }
   float avgWT, avgTAT;
   calculateAverageTimes(n, wt, tat, (int[]){p[0].bt, p[1].bt, p[2].bt, p[3].bt},
&avgWT, &avgTAT);
   printf("\nFCFS:\nAverage WT: %.2f, Average TAT: %.2f\n", avgWT, avgTAT);
}

// SJF Scheduling (Non-Preemptive)
void SJF(Process p[], int n) {
   Process temp;
   int wt[MAX] = {0}, tat[MAX], bt[MAX];
```

```
    // Sort by burst time
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (p[i].bt > p[j].bt) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
    for (int i = 0; i < n; i++) bt[i] = p[i].bt;
    for (int i = 1; i < n; i++)
        wt[i] = wt[i - 1] + bt[i - 1];
    float avgWT, avgTAT;
    calculateAverageTimes(n, wt, tat, bt, &avgWT, &avgTAT);
    printf("\nSJF:\nAverage WT: %.2f, Average TAT: %.2f\n", avgWT, avgTAT);
}

// Priority Scheduling (Non-Preemptive)
void PriorityScheduling(Process p[], int n) {
    Process temp;
    int wt[MAX] = {0}, tat[MAX], bt[MAX];
    // Sort by priority
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (p[i].priority > p[j].priority) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
    for (int i = 0; i < n; i++) bt[i] = p[i].bt;
    for (int i = 1; i < n; i++)
        wt[i] = wt[i - 1] + bt[i - 1];
    float avgWT, avgTAT;
    calculateAverageTimes(n, wt, tat, bt, &avgWT, &avgTAT);
    printf("\nPriority Scheduling:\nAverage WT: %.2f, Average TAT: %.2f\n",
avgWT, avgTAT);
}

// Round Robin Scheduling
void RoundRobin(Process p[], int n, int tq) {
    int rt[MAX], wt[MAX] = {0}, tat[MAX];
```

```
    int time = 0, done;
    for (int i = 0; i < n; i++) rt[i] = p[i].bt;
    while (1) {
        done = 1;
        for (int i = 0; i < n; i++) {
            if (rt[i] > 0) {
                done = 0;
                if (rt[i] > tq) {
                    time += tq;
                    rt[i] -= tq;
                } else {
                    time += rt[i];
                    wt[i] = time - p[i].bt;
                    rt[i] = 0;
                }
            }
        }
        if (done) break;
    }
    float avgWT, avgTAT;
    int bt[MAX];
    for (int i = 0; i < n; i++) bt[i] = p[i].bt;
    calculateAverageTimes(n, wt, tat, bt, &avgWT, &avgTAT);
    printf("\nRound Robin:\nAverage WT: %.2f, Average TAT: %.2f\n", avgWT,
avgTAT);
}

int main() {
    int n, tq;
    Process p[MAX];

    printf("Enter number of processes: ");
    scanf("%d", &n);

    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("Enter Burst Time for Process %d: ", p[i].pid);
        scanf("%d", &p[i].bt);
        printf("Enter Priority for Process %d: ", p[i].pid);
        scanf("%d", &p[i].priority);
```

```
    }

    printf("Enter Time Quantum for Round Robin: ");
    scanf("%d", &tq);

    // Copy original process list for each algorithm
    Process fcfsList[MAX], sjfList[MAX], prioList[MAX], rrList[MAX];
    for (int i = 0; i < n; i++) {
        fcfsList[i] = p[i];
        sjfList[i] = p[i];
        prioList[i] = p[i];
        rrList[i] = p[i];
    }

    FCFS(fcfsList, n);
    SJF(sjfList, n);
    PriorityScheduling(prioList, n);
    RoundRobin(rrList, n, tq);

    return 0;
}
```

OUTPUT

```
Enter number of processes: 3
Enter Burst Time for Process 1: 10
Enter Priority for Process 1: 2
Enter Burst Time for Process 2: 5
Enter Priority for Process 2: 1
Enter Burst Time for Process 3: 8
Enter Priority for Process 3: 3
Enter Time Quantum for Round Robin: 4

FCFS:
Average WT: 8.33, Average TAT: 16.00

SJF:
Average WT: 6.00, Average TAT: 13.67

Priority Scheduling:
Average WT: 6.67, Average TAT: 14.33

Round Robin:
Average WT: 12.67, Average TAT: 20.33
```

EXPERIMENT 6

Program Description : Write a program to implement Producer consumer problem

Solution:

ALGORITHM

Producer Algorithm:

1. Repeat until done
2. Generate an item (e.g., random number)
3. Wait (decrement) empty (check if space is available)
4. Lock mutex (enter critical section)
5. Add item to buffer[]
6. Print item produced
7. Unlock mutex
8. Signal (increment) full (item added)
9. Sleep or delay (simulate time to produce)

Consumer Algorithm:

1. Repeat until done
2. Wait (decrement) full (check if item available)
3. Lock mutex (enter critical section)
4. Remove item from buffer[]
5. Print item consumed
6. Unlock mutex
7. Signal (increment) empty (slot freed)
8. Sleep or delay (simulate time to consume)

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define SIZE 5 // size of buffer

int buffer[SIZE];
int in = 0, out = 0;
```

```
sem_t empty;  // counts empty slots
sem_t full;   // counts full slots
pthread_mutex_t mutex; // mutual exclusion

void* producer(void* arg) {
   int item, i;
   for (i = 0; i < 10; i++) {
      item = rand() % 100; // generate random item

      sem_wait(&empty);            // wait if buffer full
      pthread_mutex_lock(&mutex);    // enter critical section

      buffer[in] = item;
      printf("Producer produced: %d\n", item);
      in = (in + 1) % SIZE;

      pthread_mutex_unlock(&mutex);  // exit critical section
      sem_post(&full);             // signal item available

      sleep(1); // simulate production time
   }
   return NULL;
}

void* consumer(void* arg) {
   int item, i;
   for (i = 0; i < 10; i++) {
      sem_wait(&full);             // wait if buffer empty
      pthread_mutex_lock(&mutex);    // enter critical section

      item = buffer[out];
      printf("Consumer consumed: %d\n", item);
      out = (out + 1) % SIZE;

      pthread_mutex_unlock(&mutex);  // exit critical section
      sem_post(&empty);            // signal space available

      sleep(2); // simulate consumption time
   }
```

```
    return NULL;
}

int main() {
    pthread_t prod, cons;

    // Initialize mutex and semaphores
    sem_init(&empty, 0, SIZE);
    sem_init(&full, 0, 0);
    pthread_mutex_init(&mutex, NULL);

    // Create producer and consumer threads
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    // Destroy mutex and semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

OUTPUT

```
Producer produced: 83
Consumer consumed: 83
Producer produced: 86
Producer produced: 77
Consumer consumed: 86
Producer produced: 15
Producer produced: 93
Consumer consumed: 77
Producer produced: 35
Producer produced: 86
Consumer consumed: 15
Producer produced: 92
Producer produced: 49
Consumer consumed: 93
Producer produced: 21
Consumer consumed: 35
Consumer consumed: 86
Consumer consumed: 92
Consumer consumed: 49
Consumer consumed: 21
```

EXPERIMENT 7

Program Description : Write a program to implement Reader writer problem

Solution:

ALGORITHM

Reader Algorithm:

1. Wait (lock) mutex
2. Increment readcount
3. If readcount == 1, wait on wrt (first reader locks writers)
4. Signal (unlock) mutex
5. Read the shared resource
6. Wait (lock) mutex
7. Decrement readcount
8. If readcount == 0, signal wrt (last reader unlocks writers)
9. Signal (unlock) mutex

Writer Algorithm:

1. Wait on wrt (wait until no readers/writers)
2. Write to shared resource
3. Signal wrt

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t wrt;
pthread_mutex_t mutex;
int readcount = 0;
int data = 0; // shared data

void* reader(void* arg) {
   int f = *((int*)arg);
   while (1) {
      pthread_mutex_lock(&mutex);
      readcount++;
      if (readcount == 1)
```

```
        sem_wait(&wrt); // first reader locks writer
    pthread_mutex_unlock(&mutex);

    // Reading section
    printf("Reader %d: read data = %d\n", f, data);
    sleep(1);

    pthread_mutex_lock(&mutex);
    readcount--;
    if (readcount == 0)
        sem_post(&wrt); // last reader unlocks writer
    pthread_mutex_unlock(&mutex);

    sleep(1); // simulate delay
  }
  return NULL;
}

void* writer(void* arg) {
  int f = *((int*)arg);
  while (1) {
    sem_wait(&wrt);

    // Writing section
    data++;
    printf("Writer %d: wrote data = %d\n", f, data);
    sleep(2);

    sem_post(&wrt);
    sleep(2); // simulate delay
  }
  return NULL;
}

int main() {
  pthread_t rtid[5], wtid[5];
  int i, id[5];

  sem_init(&wrt, 0, 1);
  pthread_mutex_init(&mutex, NULL);
```

```
    // Create 3 readers and 2 writers
    for (i = 0; i < 3; i++) {
        id[i] = i + 1;
        pthread_create(&rtid[i], NULL, reader, &id[i]);
    }
    for (i = 0; i < 2; i++) {
        id[i] = i + 1;
        pthread_create(&wtid[i], NULL, writer, &id[i]);
    }

    // Join threads (never returns in infinite loop, used in simulation)
    for (i = 0; i < 3; i++)
        pthread_join(rtid[i], NULL);
    for (i = 0; i < 2; i++)
        pthread_join(wtid[i], NULL);

    pthread_mutex_destroy(&mutex);
    sem_destroy(&wrt);

    return 0;
}
```

OUTPUT

```
Reader 1: read data = 0
Reader 2: read data = 0
Reader 3: read data = 0
Writer 1: wrote data = 1
Writer 2: wrote data = 2
Reader 1: read data = 2
Reader 3: read data = 2
Reader 2: read data = 2
Writer 1: wrote data = 3
Writer 2: wrote data = 4
Reader 3: read data = 4
Reader 2: read data = 4
Reader 1: read data = 4
Writer 1: wrote data = 5
Writer 2: wrote data = 6
Reader 2: read data = 6
Reader 3: read data = 6
Reader 1: read data = 6
Writer 1: wrote data = 7
Writer 2: wrote data = 8
```

EXPERIMENT 8

Program Description : Write a program to implement Dining Philosophers problem

Solution:

ALGORITHM

1. Repeat forever:
2. Think
3. Wait(mutex) // Try to enter eating phase
4. Wait(fork[i]) // Pick up left fork
5. Wait(fork[(i+1)%5]) // Pick up right fork
6. Eat
7. Signal(fork[i]) // Put down left fork
8. Signal(fork[(i+1)%5]) // Put down right fork
9. Signal(mutex) // Leave eating phase

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5  // Number of philosophers

sem_t forks[N];       // One semaphore per fork
sem_t room;           // Semaphore to limit max philosophers eating at once

void* philosopher(void* num) {
   int id = *(int*)num;
   int left = id;
   int right = (id + 1) % N;

   while (1) {
      printf("Philosopher %d is thinking.\n", id);
      sleep(1); // thinking
```

```
        sem_wait(&room);          // enter room
        sem_wait(&forks[left]);    // pick left fork
        sem_wait(&forks[right]);   // pick right fork

        printf("Philosopher %d is eating using forks %d and %d.\n", id, left, right);
        sleep(2); // eating

        sem_post(&forks[left]);    // put down left fork
        sem_post(&forks[right]);   // put down right fork
        sem_post(&room);          // leave room

        printf("Philosopher %d finished eating and is thinking again.\n", id);
        sleep(1); // thinking again
    }
    return NULL;
}

int main() {
    pthread_t tid[N];
    int i, ids[N];

    sem_init(&room, 0, N - 1); // allow only 4 philosophers to try eating at once

    for (i = 0; i < N; i++)
        sem_init(&forks[i], 0, 1); // each fork available

    for (i = 0; i < N; i++) {
        ids[i] = i;
        pthread_create(&tid[i], NULL, philosopher, &ids[i]);
    }

    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL); // wait for threads (runs infinitely)

    for (i = 0; i < N; i++)
        sem_destroy(&forks[i]);
    sem_destroy(&room);

    return 0;
}
```

OUTPUT

```
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 0 is eating using forks 0 and 1.
Philosopher 2 is eating using forks 2 and 3.
Philosopher 0 finished eating and is thinking again.
Philosopher 4 is eating using forks 4 and 0.
Philosopher 2 finished eating and is thinking again.
Philosopher 1 is eating using forks 1 and 2.
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 4 finished eating and is thinking again.
Philosopher 3 is eating using forks 3 and 4.
Philosopher 1 finished eating and is thinking again.
Philosopher 0 is eating using forks 0 and 1.
Philosopher 4 is thinking.
Philosopher 1 is thinking.
Philosopher 3 finished eating and is thinking again.
Philosopher 2 is eating using forks 2 and 3.
Philosopher 0 finished eating and is thinking again.
Philosopher 4 is eating using forks 4 and 0.
```

EXPERIMENT 9

Program Description : Write a program to implementation and compare various page replacement algorithms

Solution:

ALGORITHM

1.FIFO (First-In-First-Out)

Algorithm:

1. Start
2. Initialize an empty queue of size = number of frames
3. For each page in the reference string:
    - If page is in memory → Page Hit
    - Else → Page Fault
        - If queue is not full → Add page
        - Else → Remove oldest page, add new page
4. Count total page faults
5. End

2. LRU (Least Recently Used)

Algorithm:

1. Start
2. Maintain a list of pages in memory, most recently used at front
3. For each page in reference string:
    - If page is in list → Move it to front (Page Hit)
    - Else → Page Fault:
        - If list not full → Insert at front
        - Else → Remove least recently used (end), insert at front
4. Count page faults
5. End

3. Optimal Page Replacement
Algorithm:

1. Start
2. For each page:
    - If in memory → Page Hit
    - Else → Page Fault
        - If space available → Add page
        - Else → Replace page that is **not needed for longest time in future**
3. Count page faults
4. End

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>

int isInMemory(int page, int frame[], int f) {
   for (int i = 0; i < f; i++) {
      if (frame[i] == page)
         return 1;
   }
   return 0;
}

int findLRU(int time[], int f) {
   int min = time[0], pos = 0;
   for (int i = 1; i < f; i++) {
      if (time[i] < min) {
         min = time[i];
         pos = i;
      }
   }
   return pos;
}

int findOptimal(int pages[], int frame[], int current, int n, int f) {
   int farthest = -1, index = -1;
   for (int i = 0; i < f; i++) {
      int j;
      for (j = current + 1; j < n; j++) {
```

```
            if (frame[i] == pages[j])
                break;
        }
        if (j == n)
            return i; // not used again
        if (j > farthest) {
            farthest = j;
            index = i;
        }
    }
    return index;
}

void FIFO(int pages[], int n, int f) {
    int frame[10], front = 0, pageFaults = 0;
    for (int i = 0; i < f; i++) frame[i] = -1;

    for (int i = 0; i < n; i++) {
        if (!isInMemory(pages[i], frame, f)) {
            frame[front] = pages[i];
            front = (front + 1) % f;
            pageFaults++;
        }
    }

    printf("FIFO: Page Faults = %d\n", pageFaults);
}

void LRU(int pages[], int n, int f) {
    int frame[10], time[10], counter = 0, pageFaults = 0;
    for (int i = 0; i < f; i++) frame[i] = -1;

    for (int i = 0; i < n; i++) {
        if (isInMemory(pages[i], frame, f)) {
            for (int j = 0; j < f; j++) {
                if (frame[j] == pages[i]) {
                    time[j] = counter++;
                    break;
                }
            }
```

```
        } else {
            int pos;
            for (pos = 0; pos < f; pos++) {
                if (frame[pos] == -1) break;
            }

            if (pos != f) {
                frame[pos] = pages[i];
                time[pos] = counter++;
            } else {
                int lru = findLRU(time, f);
                frame[lru] = pages[i];
                time[lru] = counter++;
            }
            pageFaults++;
        }
    }

    printf("LRU : Page Faults = %d\n", pageFaults);
}

void Optimal(int pages[], int n, int f) {
    int frame[10], pageFaults = 0;
    for (int i = 0; i < f; i++) frame[i] = -1;

    for (int i = 0; i < n; i++) {
        if (!isInMemory(pages[i], frame, f)) {
            int pos;
            for (pos = 0; pos < f; pos++) {
                if (frame[pos] == -1) break;
            }

            if (pos != f)
                frame[pos] = pages[i];
            else {
                int replaceIndex = findOptimal(pages, frame, i, n, f);
                frame[replaceIndex] = pages[i];
            }

            pageFaults++;
```

```
        }
    }

    printf("Optimal: Page Faults = %d\n", pageFaults);
}

int main() {
    int n, f, i;
    int pages[50];

    printf("Enter number of pages: ");
    scanf("%d", &n);

    printf("Enter reference string: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }

    printf("Enter number of frames: ");
    scanf("%d", &f);

    printf("\n--- Page Replacement Comparison ---\n");
    FIFO(pages, n, f);
    LRU(pages, n, f);
    Optimal(pages, n, f);

    return 0;
}
```

OUTPUT

```
Enter number of pages: 12
Enter reference string: 7 0 1 2 0 3 0 4 2 3 0 3
Enter number of frames: 3

--- Page Replacement Comparison ---
FIFO: Page Faults = 10
LRU : Page Faults = 9
Optimal: Page Faults = 7
```

EXPERIMENT 10

Program Description : Write a program to implementation and compare various disk and drum scheduling algorithm

Solution:

ALGORITHM

1. FCFS (First-Come-First-Serve)

Algorithm:

1. Start from the initial head position.
2. Visit all requests in the order they appear.
3. Calculate absolute difference in head position for each request.
4. Sum all movements.

2. SSTF (Shortest Seek Time First)

Algorithm:

1. Start from the initial head position.
2. At each step, choose the closest track to the current head.
3. Move to that track.
4. Repeat until all requests are serviced.

3. SCAN (Elevator Algorithm)

Algorithm:

1. Move in one direction (e.g., towards 0).
2. Service all requests in that direction.
3. On reaching the end, reverse and continue.

4. C-SCAN (Circular SCAN)

Algorithm:

1. Move in one direction (e.g., towards 0).
2. On reaching the end, jump to the other end without servicing.
3. Continue servicing in same direction.

5. LOOK

Algorithm:

1. Move in one direction, only as far as the last request in that direction.
2. Then reverse and go back only as far as needed.

6. C-LOOK

Algorithm:

1. Move in one direction to the farthest request.
2. Jump to the lowest request.
3. Continue in same direction.

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

void sort(int arr[], int n) {
   for(int i=0;i<n-1;i++)
      for(int j=i+1;j<n;j++)
         if(arr[i]>arr[j]){
            int temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
         }
}

int FCFS(int requests[], int n, int head) {
   int total = 0;
   for (int i = 0; i < n; i++) {
      total += abs(head - requests[i]);
      head = requests[i];
   }
   return total;
}

int SSTF(int requests[], int n, int head) {
   int total = 0, done[n];
   for (int i = 0; i < n; i++) done[i] = 0;

   for (int count = 0; count < n; count++) {
      int min = INT_MAX, index = -1;
      for (int i = 0; i < n; i++) {
         if (!done[i] && abs(head - requests[i]) < min) {
            min = abs(head - requests[i]);
```

```
            index = i;
          }
        }
        total += abs(head - requests[index]);
        head = requests[index];
        done[index] = 1;
    }
    return total;
}

int SCAN(int requests[], int n, int head, int disk_size) {
    int total = 0, i;
    sort(requests, n);

    for (i = 0; i < n && requests[i] < head; i++);
    for (int j = i; j < n; j++) {
        total += abs(head - requests[j]);
        head = requests[j];
    }

    total += abs(head - (disk_size - 1));
    head = disk_size - 1;

    for (int j = i - 1; j >= 0; j--) {
        total += abs(head - requests[j]);
        head = requests[j];
    }

    return total;
}

int CSCAN(int requests[], int n, int head, int disk_size) {
    int total = 0, i;
    sort(requests, n);

    for (i = 0; i < n && requests[i] < head; i++);
    for (int j = i; j < n; j++) {
        total += abs(head - requests[j]);
        head = requests[j];
    }
```

```
    total += abs(head - (disk_size - 1)) + (disk_size - 1);
    head = 0;

    for (int j = 0; j < i; j++) {
        total += abs(head - requests[j]);
        head = requests[j];
    }

    return total;
}

int LOOK(int requests[], int n, int head) {
    int total = 0, i;
    sort(requests, n);

    for (i = 0; i < n && requests[i] < head; i++);
    for (int j = i; j < n; j++) {
        total += abs(head - requests[j]);
        head = requests[j];
    }

    for (int j = i - 1; j >= 0; j--) {
        total += abs(head - requests[j]);
        head = requests[j];
    }

    return total;
}

int CLOOK(int requests[], int n, int head) {
    int total = 0, i;
    sort(requests, n);

    for (i = 0; i < n && requests[i] < head; i++);
    for (int j = i; j < n; j++) {
        total += abs(head - requests[j]);
        head = requests[j];
    }
```

```
    if (i > 0) {
        total += abs(head - requests[0]);
        head = requests[0];

        for (int j = 1; j < i; j++) {
            total += abs(head - requests[j]);
            head = requests[j];
        }
    }

    return total;
}

int main() {
    int requests[100], n, head, disk_size;

    printf("Enter number of disk requests: ");
    scanf("%d", &n);

    printf("Enter disk requests: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &requests[i]);

    printf("Enter initial head position: ");
    scanf("%d", &head);

    printf("Enter disk size (max track number): ");
    scanf("%d", &disk_size);

    printf("\n--- Disk Scheduling Results ---\n");
    printf("FCFS   : %d\n", FCFS(requests, n, head));
    printf("SSTF   : %d\n", SSTF(requests, n, head));
    printf("SCAN   : %d\n", SCAN(requests, n, head, disk_size));
    printf("C-SCAN : %d\n", CSCAN(requests, n, head, disk_size));
    printf("LOOK   : %d\n", LOOK(requests, n, head));
    printf("C-LOOK : %d\n", CLOOK(requests, n, head));

    return 0;
}
```

OUTPUT

```
Enter number of disk requests: 8
Enter disk requests: 98 183 37 122 14 124 65 67
Enter initial head position: 53
Enter disk size (max track number): 200

--- Disk Scheduling Results ---
FCFS   : 640
SSTF   : 236
SCAN   : 331
C-SCAN : 382
LOOK   : 299
C-LOOK : 322
```