ESTD. 1917

UNIVERSITY COLLEGE OF ENGINEERING

(AUTONOMOUS)

OSMANIA UNIVERSITY, HYDERABAD, TELANGANA STATE, INDIA

A University Accredited by NAAC with A+ Grade
Category -I Graded Autonomy by UGC

ESTD. 1929

Operating Systems Record

Name	:	
Roll No.	:	**1005227330**
Class	:	**BE(CSE)**
Semester	:	**IInd Year, IV Sem**
Academic Year	:	**2023-24**
Department	:	**Computer Science**

CERTIFICATE

This is to certify that the programming assignments and projects submitted by ____________________ with Roll No. 1005227330__, pursuing B.E. in Computer Science and Engineering, during IInd Year, IV Sem for the subject “Operating Systems”, are genuine and represent their individual work.

Submitted on ______________________

Faculty Member

INDEX

1) Write a program for FCFS scheduling algorithm

```
#include <stdio.h>

struct Process {
  int id;
  int arrival_time;
  int burst_time;
  int waiting_time;
  int turnaround_time;
};

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

  struct Process proc[n];
  for (int i = 0; i < n; i++) {
    proc[i].id = i + 1;
    printf("Enter arrival time and burst time for process %d: ", i + 1);
    scanf("%d %d", &proc[i].arrival_time, &proc[i].burst_time);
  }

  proc[0].waiting_time = 0;
  proc[0].turnaround_time = proc[0].burst_time;

  for (int i = 1; i < n; i++) {
    proc[i].waiting_time = proc[i - 1].waiting_time + proc[i - 1].burst_time;
    proc[i].turnaround_time = proc[i].waiting_time + proc[i].burst_time;
  }

  printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\n", proc[i].id, proc[i].arrival_time, proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
  }

  float avg_waiting_time = 0, avg_turnaround_time = 0;
  for (int i = 0; i < n; i++) {
    avg_waiting_time += proc[i].waiting_time;
    avg_turnaround_time += proc[i].turnaround_time;
  }
  avg_waiting_time /= n;
  avg_turnaround_time /= n;

  printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
  printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);
```

```
    return 0;
}
```

Output:

```
Enter number of processes: 3
Enter arrival time and burst time for process 1: 0
24
Enter arrival time and burst time for process 2: 0
3
Enter arrival time and burst time for process 3: 0
3

Process Arrival Burst   Waiting Turnaround
1       0       24      0       24
2       0       3       24      27
3       0       3       27      30

Average Waiting Time: 17.00
Average Turnaround Time: 27.00
```

2) Write a program for SJF-non preemptive algorithm

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

struct Process {
   int id;
   int arrival_time;
   int burst_time;
   int waiting_time;
   int turnaround_time;
};

void sortProcesses(struct Process proc[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (proc[i].arrival_time > proc[j].arrival_time ||
            (proc[i].arrival_time == proc[j].arrival_time && proc[i].burst_time > proc[j].burst_time)) {
            struct Process temp = proc[i];
            proc[i] = proc[j];
            proc[j] = temp;
         }
      }
   }
}

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

   struct Process proc[n];
   for (int i = 0; i < n; i++) {
      proc[i].id = i + 1;
      printf("Enter arrival time and burst time for process %d: ", i + 1);
      scanf("%d %d", &proc[i].arrival_time, &proc[i].burst_time);
   }

   sortProcesses(proc, n);

   int time = 0;
   for (int i = 0; i < n; i++) {
      if (time < proc[i].arrival_time) {
         time = proc[i].arrival_time;
      }
      proc[i].waiting_time = time - proc[i].arrival_time;
      time += proc[i].burst_time;
      proc[i].turnaround_time = proc[i].waiting_time + proc[i].burst_time;
```

```
    }

    printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\n", proc[i].id, proc[i].arrival_time, proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
    }

    float avg_waiting_time = 0, avg_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += proc[i].waiting_time;
        avg_turnaround_time += proc[i].turnaround_time;
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
}
```

Output:

```
Enter number of processes: 3
Enter arrival time and burst time for process 1: 0
20
Enter arrival time and burst time for process 2: 0
30
Enter arrival time and burst time for process 3: 0
10

Process Arrival Burst   Waiting Turnaround
3       0       10      0       10
1       0       20      10      30
2       0       30      30      60

Average Waiting Time: 13.33
Average Turnaround Time: 33.33
```

3) Write a program for SJF-preemptive algorithm

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

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
};

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

    struct Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].id = i + 1;
        printf("Enter arrival time and burst time for process %d: ", i + 1);
        scanf("%d %d", &proc[i].arrival_time, &proc[i].burst_time);
        proc[i].remaining_time = proc[i].burst_time;
    }

    int time = 0, completed = 0;
    while (completed < n) {
        int idx = -1;
        int min_remaining_time = INT_MAX;

        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0 && proc[i].arrival_time <= time) {
                if (proc[i].remaining_time < min_remaining_time) {
                    min_remaining_time = proc[i].remaining_time;
                    idx = i;
                }
            }
        }

        if (idx != -1) {
            proc[idx].remaining_time--;
            if (proc[idx].remaining_time == 0) {
                completed++;
                proc[idx].turnaround_time = time + 1 - proc[idx].arrival_time;
                proc[idx].waiting_time = proc[idx].turnaround_time - proc[idx].burst_time;
            }
        }
        time++;
    }

    printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t%d\n", proc[i].id, proc[i].arrival_time, proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
    }

    float avg_waiting_time = 0, avg_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += proc[i].waiting_time;
        avg_turnaround_time += proc[i].turnaround_time;
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
}
```

Output:

```
Enter number of processes: 5
Enter arrival time and burst time for process 1: 0
10
Enter arrival time and burst time for process 2: 1
3
Enter arrival time and burst time for process 3: 2
2
Enter arrival time and burst time for process 4: 3
4
Enter arrival time and burst time for process 5: 4
5

Process Arrival Burst   Waiting Turnaround
1       0       10      14      24
2       1       3       0       3
3       2       2       2       4
4       3       4       3       7
5       4       5       6       11

Average Waiting Time: 5.00
Average Turnaround Time: 9.80
```

4) Write a program for priority scheduling-non preemptive algorithm

```
#include <stdio.h>

struct Process {
  int id;
  int arrival_time;
  int burst_time;
  int priority;
  int waiting_time;
  int turnaround_time;
};

void sortProcesses(struct Process proc[], int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
      if (proc[i].arrival_time > proc[j].arrival_time ||
        (proc[i].arrival_time == proc[j].arrival_time && proc[i].priority > proc[j].priority)) {
        struct Process temp = proc[i];
        proc[i] = proc[j];
        proc[j] = temp;
      }
    }
  }
}

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

  struct Process proc[n];
  for (int i = 0; i < n; i++) {
    proc[i].id = i + 1;
    printf("Enter arrival time, burst time and priority for process %d: ", i + 1);
    scanf("%d %d %d", &proc[i].arrival_time, &proc[i].burst_time, &proc[i].priority);
  }

  sortProcesses(proc, n);

  int time = 0;
  for (int i = 0; i < n; i++) {
    if (time < proc[i].arrival_time) {
      time = proc[i].arrival_time;
    }
    proc[i].waiting_time = time - proc[i].arrival_time;
    time += proc[i].burst_time;
    proc[i].turnaround_time = proc[i].waiting_time + proc[i].burst_time;
  }

  printf("\nProcess\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t\t%d\t%d\n", proc[i].id, proc[i].arrival_time, proc[i].burst_time,
proc[i].priority, proc[i].waiting_time, proc[i].turnaround_time);
  }

  float avg_waiting_time = 0, avg_turnaround_time = 0;
  for (int i = 0; i < n; i++) {
    avg_waiting_time += proc[i].waiting_time;
    avg_turnaround_time += proc[i].turnaround_time;
  }
  avg_waiting_time /= n;
  avg_turnaround_time /= n;

  printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
  printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);

  return 0;
}
```

Output:

```
Enter number of processes: 5
Enter arrival time, burst time and priority for process 1: 0
10
3
Enter arrival time, burst time and priority for process 2: 0
1
1
Enter arrival time, burst time and priority for process 3: 0
2
4
Enter arrival time, burst time and priority for process 4: 0
1
5
Enter arrival time, burst time and priority for process 5: 0
5
2
Process Arrival Burst   Priority        Waiting Turnaround
2       0       1       1               0       1
5       0       5       2               1       6
1       0       10      3               6       16
3       0       2       4               16      18
4       0       1       5               18      19
Average Waiting Time: 8.20
Average Turnaround Time: 12.00
```

5) Write a program for priority scheduling-preemptive algorithm

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

struct Process {
   int id;
   int arrival_time;
   int burst_time;
   int remaining_time;
   int priority;
   int waiting_time;
   int turnaround_time;
};
int main() {
   int n;
   printf("Enter number of processes: ");
   scanf("%d", &n);

   struct Process proc[n];
   for (int i = 0; i < n; i++) {
      proc[i].id = i + 1;
      printf("Enter arrival time, burst time and priority for process %d: ", i + 1);
      scanf("%d %d %d", &proc[i].arrival_time, &proc[i].burst_time, &proc[i].priority);
      proc[i].remaining_time = proc[i].burst_time;
   }
   int time = 0, completed = 0;
   while (completed < n) {
      int idx = -1;
      int highest_priority = INT_MAX;

      for (int i = 0; i < n; i++) {
         if (proc[i].remaining_time > 0 && proc[i].arrival_time <= time) {
            if (proc[i].priority < highest_priority) {
               highest_priority = proc[i].priority;
               idx = i;
            }
         }
      }
      if (idx != -1) {
         proc[idx].remaining_time--;
         if (proc[idx].remaining_time == 0) {
            completed++;
            proc[idx].turnaround_time = time + 1 - proc[idx].arrival_time;
            proc[idx].waiting_time = proc[idx].turnaround_time - proc[idx].burst_time;
         }
      }
      time++;
```

```
    }
    printf("\nProcess\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\n", proc[i].id, proc[i].arrival_time, proc[i].burst_time,
proc[i].priority, proc[i].waiting_time, proc[i].turnaround_time);
    }

    float avg_waiting_time = 0, avg_turnaround_time = 0;
    for (int i = 0; i < n; i++) {
        avg_waiting_time += proc[i].waiting_time;
        avg_turnaround_time += proc[i].turnaround_time;
    }
    avg_waiting_time /= n;
    avg_turnaround_time /= n;

    printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
    printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);

    return 0;
}
```

Output:

```
Enter number of processes: 5
Enter arrival time, burst time and priority for process 1: 0
10
3
Enter arrival time, burst time and priority for process 2: 1
1
1
Enter arrival time, burst time and priority for process 3: 2
2
4
Enter arrival time, burst time and priority for process 4: 3
1
5
Enter arrival time, burst time and priority for process 5: 4
5
2
Process Arrival Burst   Priority        Waiting Turnaround
1       0       10      3               6       16
2       1       1       1               0       1
3       2       2       4               14      16
4       3       1       5               15      16
5       4       5       2               0       5

Average Waiting Time: 7.00
Average Turnaround Time: 10.80
```

6) Write a program for round robin scheduling algorithm

```
#include <stdio.h>

struct Process {
  int id;
  int arrival_time;
  int burst_time;
  int remaining_time;
  int waiting_time;
  int turnaround_time;
};

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

  struct Process proc[n];
  for (int i = 0; i < n; i++) {
    proc[i].id = i + 1;
    printf("Enter arrival time and burst time for process %d: ", i + 1);
    scanf("%d %d", &proc[i].arrival_time, &proc[i].burst_time);
    proc[i].remaining_time = proc[i].burst_time;
  }

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

  int time = 0, completed = 0;
  while (completed < n) {
    for (int i = 0; i < n; i++) {
      if (proc[i].remaining_time > 0 && proc[i].arrival_time <= time) {
        if (proc[i].remaining_time > quantum) {
          time += quantum;
          proc[i].remaining_time -= quantum;
        } else {
          time += proc[i].remaining_time;
          proc[i].waiting_time = time - proc[i].arrival_time - proc[i].burst_time;
          proc[i].turnaround_time = time - proc[i].arrival_time;
          proc[i].remaining_time = 0;
          completed++;
        }
      }
    }
  }

  printf("\nProcess\tArrival\tBurst\tWaiting\tTurnaround\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\n", proc[i].id, proc[i].arrival_time, proc[i].burst_time,
proc[i].waiting_time, proc[i].turnaround_time);
  }

  float avg_waiting_time = 0, avg_turnaround_time = 0;
  for (int i = 0; i < n; i++) {
    avg_waiting_time += proc[i].waiting_time;
    avg_turnaround_time += proc[i].turnaround_time;
  }
  avg_waiting_time /= n;
  avg_turnaround_time /= n;

  printf("\nAverage Waiting Time: %.2f", avg_waiting_time);
  printf("\nAverage Turnaround Time: %.2f\n", avg_turnaround_time);

  return 0;
}
```

Output:

```
Enter number of processes: 4
Enter arrival time and burst time for process 1: 0
6
Enter arrival time and burst time for process 2: 0
3
Enter arrival time and burst time for process 3: 0
1
Enter arrival time and burst time for process 4: 0
7
Enter time quantum: 1

Process Arrival Burst   Waiting Turnaround
1       0       6       9       15
2       0       3       6       9
3       0       1       2       3
4       0       7       10      17

Average Waiting Time: 6.75
Average Turnaround Time: 11.00
```

7) Write a program to demonstrate first fit, best fit, worst fit

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

void firstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                allocation[i] = j;
                blockSize[j] = -1;
                break;
            }
        }
    }

    printf("\nFirst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf(" %d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d", allocation[i] + 1);
        } else {
            printf("Not Allocated");
        }
        printf("\n");
    }

    printf("Remaining Memory of Each Block:\n");
    for (int i = 0; i < m; i++) {
        if (blockSize[i] == -1) {
            printf("Block %d: Allocated\n", i + 1);
        } else {
            printf("Block %d: %d\n", i + 1, blockSize[i]);
        }
    }
}

void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        int bestIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (bestIdx == -1 || blockSize[bestIdx] > blockSize[j]) {
                    bestIdx = j;
                }
            }
        }
        if (bestIdx != -1) {
            allocation[i] = bestIdx;
            blockSize[bestIdx] = -1;
        }
    }

    printf("\nBest Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf(" %d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d", allocation[i] + 1);
        } else {
            printf("Not Allocated");
        }
        printf("\n");
    }

    printf("Remaining Memory of Each Block:\n");
    for (int i = 0; i < m; i++) {
        if (blockSize[i] == -1) {
            printf("Block %d: Allocated\n", i + 1);
        } else {
            printf("Block %d: %d\n", i + 1, blockSize[i]);
        }
    }
}

void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[n];
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[worstIdx] < blockSize[j]) {
                    worstIdx = j;
                }
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] = -1;
        }
    }

    printf("\nWorst Fit Allocation:\n");
    printf("Process No.\tProcess Size\tBlock No.\n");
    for (int i = 0; i < n; i++) {
        printf(" %d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d", allocation[i] + 1);
        } else {
            printf("Not Allocated");
        }
        printf("\n");
    }

    printf("Remaining Memory of Each Block:\n");
    for (int i = 0; i < m; i++) {
        if (blockSize[i] == -1) {
            printf("Block %d: Allocated\n", i + 1);
        } else {
            printf("Block %d: %d\n", i + 1, blockSize[i]);
        }
    }
}

int main() {
    int m, n;

    printf("Enter number of memory blocks: ");
    scanf("%d", &m);
    int blockSize[m];

    printf("Enter sizes of memory blocks:\n");
    for (int i = 0; i < m; i++) {
        printf("Block %d: ", i + 1);
        scanf("%d", &blockSize[i]);
    }

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

    printf("Enter sizes of processes:\n");
```

```
  for (int i = 0; i < n; i++) {
    printf("Process %d: ", i + 1);
    scanf("%d", &processSize[i]);
  }

  int originalBlockSize[m];
  for (int i = 0; i < m; i++) {
    originalBlockSize[i] = blockSize[i];
  }

  firstFit(originalBlockSize, m, processSize, n);
  for (int i = 0; i < m; i++) {
    originalBlockSize[i] = blockSize[i];
  }
  bestFit(originalBlockSize, m, processSize, n);
  for (int i = 0; i < m; i++) {
    originalBlockSize[i] = blockSize[i];
  }
  worstFit(originalBlockSize, m, processSize, n);

  return 0;
}
```

Output:

```
Enter number of memory blocks: 5
Enter sizes of memory blocks:
Block 1: 100
Block 2: 500
Block 3: 200
Block 4: 300
Block 5: 600
Enter number of processes: 4
Enter sizes of processes:
Process 1: 212
Process 2: 417
Process 3: 112
Process 4: 426

First Fit Allocation:
Process No.    Process Size    Block No.
 1          212          2
 2          417          5
 3          112          3
 4          426          Not Allocated
Remaining Memory of Each Block:
Block 1: 100
Block 2: Allocated
Block 3: Allocated
Block 4: 300
Block 5: Allocated

Best Fit Allocation:
Process No.   Process Size   Block No.
 1          212          4
 2          417          2
 3          112          3
 4          426          5
Remaining Memory of Each Block:
Block 1: 100
Block 2: Allocated
Block 3: Allocated
Block 4: Allocated
Block 5: Allocated

Worst Fit Allocation:
Process No.   Process Size   Block No.
 1          212          5
 2          417          2
 3          112          4
 4          426          Not Allocated
Remaining Memory of Each Block:
Block 1: 100
Block 2: Allocated
Block 3: 200
Block 4: Allocated
Block 5: Allocated
```

8) Write a program for FIFO page replacement algorithm

```
#include <stdio.h>

void fifoPageReplacement(int pages[], int numPages, int frameSize) {
  int frame[frameSize];
  int pageFaults = 0, k = 0, i, j;
  int isPresent;

  for (i = 0; i < frameSize; i++) {
    frame[i] = -1;
  }

  for (i = 0; i < numPages; i++) {
    isPresent = 0;

    for (j = 0; j < frameSize; j++) {
      if (frame[j] == pages[i]) {
        isPresent = 1;
        break;
      }
    }

    if (!isPresent) {
      frame[k] = pages[i];
      k = (k + 1) % frameSize;
      pageFaults++;
      printf("Page %d caused a page fault. Frame: ", pages[i]);
      for (j = 0; j < frameSize; j++) {
        printf("%d ", frame[j]);
      }
      printf("\n");
    } else {
      printf("Page %d is already in the frame. No page fault.\n", pages[i]);
    }
  }

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

int main() {
  int numPages, frameSize;

  printf("Enter number of pages: ");
  scanf("%d", &numPages);
  int pages[numPages];

  printf("Enter page numbers:\n");
```

```
    for (int i = 0; i < numPages; i++) {
        printf("Page %d: ", i + 1);
        scanf("%d", &pages[i]);
    }

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

    fifoPageReplacement(pages, numPages, frameSize);

    return 0;
}
```

Output:

```
Enter number of pages: 20
Enter page numbers:
Page 1: 7
Page 2: 0
Page 3: 1
Page 4: 2
Page 5: 0
Page 6: 3
Page 7: 0
Page 8: 4
Page 9: 2
Page 10: 3
Page 11: 0
Page 12: 3
Page 13: 2
Page 14: 1
Page 15: 2
Page 16: 0
Page 17: 1
Page 18: 7
Page 19: 0
Page 20: 1
Enter number of frames: 3
Page 7 caused a page fault. Frame: 7 -1 -1
Page 0 caused a page fault. Frame: 7 0 -1
Page 1 caused a page fault. Frame: 7 0 1
Page 2 caused a page fault. Frame: 2 0 1
Page 0 is already in the frame. No page fault.
Page 3 caused a page fault. Frame: 2 3 1
Page 0 caused a page fault. Frame: 2 3 0
Page 4 caused a page fault. Frame: 4 3 0
Page 2 caused a page fault. Frame: 4 2 0
Page 3 caused a page fault. Frame: 4 2 3
Page 0 caused a page fault. Frame: 0 2 3
Page 3 is already in the frame. No page fault.
Page 2 is already in the frame. No page fault.
Page 1 caused a page fault. Frame: 0 1 3
Page 2 caused a page fault. Frame: 0 1 2
Page 0 is already in the frame. No page fault.
Page 1 is already in the frame. No page fault.
Page 7 caused a page fault. Frame: 7 1 2
Page 0 caused a page fault. Frame: 7 0 2
Page 1 caused a page fault. Frame: 7 0 1
Total Page Faults: 15
```

9) Write a program for Optimal page replacement algorithm

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

void optimalPageReplacement(int pages[], int n, int frameCount) {
  int frame[frameCount];
  int pageFaults = 0;

  for (int i = 0; i < frameCount; i++) {
    frame[i] = -1;
  }

  printf("\nPage Reference\tFrames\n");

  for (int i = 0; i < n; i++) {
    int page = pages[i];
    int found = 0;

    for (int j = 0; j < frameCount; j++) {
      if (frame[j] == page) {
        found = 1;
        break;
      }
    }

    if (!found) {
      int replaceIdx = -1, farthest = -1;

      for (int j = 0; j < frameCount; j++) {
        int k;
        for (k = i + 1; k < n; k++) {
          if (frame[j] == pages[k]) {
            if (k > farthest) {
              farthest = k;
              replaceIdx = j;
            }
            break;
          }
        }
        if (k == n) {
          replaceIdx = j;
          break;
        }
      }

      if (replaceIdx == -1) {
        replaceIdx = 0;
```

```
            }

            frame[replaceIdx] = page;
            pageFaults++;
        }

        printf("%d\t\t\t", page);
        for (int j = 0; j < frameCount; j++) {
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }

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

int main() {
    int n, frameCount;

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

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

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

    optimalPageReplacement(pages, n, frameCount);

    return 0;
}
```

Output:

```
Enter number of pages: 20
Enter page reference string:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter number of frames: 3

Page Reference  Frames
```

```
7          7 - -
0          7 0 -
1          7 0 1
2          2 0 1
0          2 0 1
3          2 0 3
0          2 0 3
4          2 4 3
2          2 4 3
3          2 4 3
0          2 0 3
3          2 0 3
2          2 0 3
1          2 0 1
2          2 0 1
0          2 0 1
1          2 0 1
7          7 0 1
0          7 0 1
1          7 0 1
Total Page Faults: 9
```

10) Write a program for LRU page replacement algorithm

```
#include<stdio.h>
int main()
{
int fr[10],p[50],i,j,n;
int max,found=0,lg[10],index,k,l,flag1=0,flag2=0,pf=0,frsize;
printf("total no of pages in reference string: ");
scanf("%d",&n);
printf("enter reference string:\n");
for(i=0;i<n;i++)
 scanf("%d",&p[i]);
printf("no of frames: ");
scanf("%d",&frsize);
for(i=0;i<frsize;i++)
{
fr[i]=-1;
}
for(j=0;j<n;j++)
{
 flag1=0;flag2=0;
 for(i=0;i<frsize;i++)
 {
 if(fr[i]==p[j])
 {
 flag1=1;flag2=1;break;
  }
 }
 if(flag1==0)
 {
 for(i=0;i<frsize;i++)
 {
 if(fr[i]==-1)
 {
 fr[i]=p[j];
 flag2=1;
 pf++;
 break;
 }
 }
 }
 if(flag2==0)
 {
 for(i=0;i<frsize;i++)
 {lg[i]=0;}
 for(i=0;i<frsize;i++)
 {
 for(k=j-1;k>=0;k--)
 {
 if(fr[i]==p[k])
 {
 lg[i]=k-j;
 break;
 }
 }
 }
 found=0;
 for(i=0;i<frsize;i++)
 {
 if(lg[i]==0)
 {
 index=i;found=1;break;
 }
 }
 if(found==0)
 {
 max=lg[0];
 index=0;
 for(i=1;i<frsize;i++)
 {
 if(max>lg[i])
 {
 max=lg[i];index=i;
 }
 }
 }
 fr[index]=p[j];
 pf++;
 }
 printf("\n");
  for(i=0;i<frsize;i++)
 printf("\t%d",fr[i]);
}
 printf("\nno of page faults:%d\n",pf);
return 0;
}
```

<u>Output:</u>

```
total no of pages in reference string: 20
enter reference string:
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
no of frames: 3


     7      -1     -1
     7       0      -1
```

```
    7    0    1
    2    0    1
    2    0    1
    2    0    3
    2    0    3
    4    0    3
    4    0    2
    4    3    2
    0    3    2
    0    3    2
    0    3    2
    1    3    2
    1    3    2
    1    0    2
    1    0    2
    1    0    7
    1    0    7
    1    0    7
no of page faults:12
```

11) Write a program to demonstrate producer-consumer problem

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

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t empty;
sem_t full;
pthread_mutex_t mutex;

void* producer(void* param);
void* consumer(void* param);

int main() {
   pthread_t producer_thread, consumer_thread;

   sem_init(&empty, 0, BUFFER_SIZE);
   sem_init(&full, 0, 0);
   pthread_mutex_init(&mutex, NULL);
   pthread_create(&producer_thread, NULL, producer, NULL);
   pthread_create(&consumer_thread, NULL, consumer, NULL);
   pthread_join(producer_thread, NULL);
   pthread_join(consumer_thread, NULL);
   sem_destroy(&empty);
   sem_destroy(&full);
   pthread_mutex_destroy(&mutex);

   return 0;
}
void* producer(void* param) {
   int item;
   for (int i = 0; i < 10; ++i) {
      sleep(rand() % 3);

      item = rand() % 100;
      sem_wait(&empty);
      pthread_mutex_lock(&mutex);

      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
```

```
        printf("Producer produced %d\n", item);

        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    pthread_exit(0);
}

void* consumer(void* param) {
    int item;
    for (int i = 0; i < 10; ++i) {
        sleep(rand() % 3);

        sem_wait(&full);
        pthread_mutex_lock(&mutex);

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

        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    pthread_exit(0);
}
```

Output:

```
Producer produced 77
Consumer consumed 77
Producer produced 35
Consumer consumed 35
Producer produced 49
Consumer consumed 49
Producer produced 27
Consumer consumed 27
Producer produced 63
Consumer consumed 63
Producer produced 26
Consumer consumed 26
Producer produced 11
Consumer consumed 11
Producer produced 29
Consumer consumed 29
Producer produced 62
Consumer consumed 62
Producer produced 35
Consumer consumed 35
```

12) Write a program to demonstrate dining-philosophers problem

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

#define NUM_PHILOSOPHERS 5

sem_t forks[NUM_PHILOSOPHERS];
pthread_t philosophers[NUM_PHILOSOPHERS];

void* philosopher(void* num);
void think(int philosopher_number);
void eat(int philosopher_number);
void pick_up_forks(int philosopher_number);
void put_down_forks(int philosopher_number);

int main() {
   int i;
   int philosopher_numbers[NUM_PHILOSOPHERS];

   for (i = 0; i < NUM_PHILOSOPHERS; i++) {
      sem_init(&forks[i], 0, 1);
   }

   for (i = 0; i < NUM_PHILOSOPHERS; i++) {
      philosopher_numbers[i] = i;
      pthread_create(&philosophers[i], NULL, philosopher, &philosopher_numbers[i]);
   }

   for (i = 0; i < NUM_PHILOSOPHERS; i++) {
      pthread_join(philosophers[i], NULL);
   }

   for (i = 0; i < NUM_PHILOSOPHERS; i++) {
      sem_destroy(&forks[i]);
   }

   return 0;
}

void* philosopher(void* num) {
   int philosopher_number = *(int*)num;

   while (1) {
      think(philosopher_number);
```

```
        pick_up_forks(philosopher_number);
        eat(philosopher_number);
        put_down_forks(philosopher_number);
    }
}

void think(int philosopher_number) {
    printf("Philosopher %d is thinking.\n", philosopher_number);
    sleep(rand() % 3);
}

void eat(int philosopher_number) {
    printf("Philosopher %d is eating.\n", philosopher_number);
    sleep(rand() % 3);
}

void pick_up_forks(int philosopher_number) {
    int left_fork = philosopher_number;
    int right_fork = (philosopher_number + 1) % NUM_PHILOSOPHERS;

    if (philosopher_number % 2 == 0) {
        sem_wait(&forks[left_fork]);
        printf("Philosopher %d picks up left fork %d.\n", philosopher_number, left_fork);
        sem_wait(&forks[right_fork]);
        printf("Philosopher %d picks up right fork %d.\n", philosopher_number, right_fork);
    } else {
        sem_wait(&forks[right_fork]);
        printf("Philosopher %d picks up right fork %d.\n", philosopher_number, right_fork);
        sem_wait(&forks[left_fork]);
        printf("Philosopher %d picks up left fork %d.\n", philosopher_number, left_fork);
    }
}

void put_down_forks(int philosopher_number) {
    int left_fork = philosopher_number;
    int right_fork = (philosopher_number + 1) % NUM_PHILOSOPHERS;

    sem_post(&forks[left_fork]);
    printf("Philosopher %d puts down left fork %d.\n", philosopher_number, left_fork);
    sem_post(&forks[right_fork]);
    printf("Philosopher %d puts down right fork %d.\n", philosopher_number, right_fork);
}
```

Output:

```
Philosopher 0 is thinking.
Philosopher 1 is thinking.
Philosopher 2 is thinking.
Philosopher 3 is thinking.
Philosopher 4 is thinking.
Philosopher 0 picks up left fork 0.
Philosopher 0 picks up right fork 1.
Philosopher 0 is eating.
Philosopher 2 picks up left fork 2.
Philosopher 2 picks up right fork 3.
Philosopher 2 is eating.
Philosopher 4 picks up left fork 4.
Philosopher 4 picks up right fork 0.
Philosopher 0 puts down left fork 0.
Philosopher 0 puts down right fork 1.
Philosopher 0 is thinking.
Philosopher 4 is eating.
Philosopher 3 picks up left fork 3.
Philosopher 3 picks up right fork 4.
Philosopher 2 puts down left fork 2.
Philosopher 2 puts down right fork 3.
Philosopher 2 is thinking.
Philosopher 1 picks up right fork 1.
Philosopher 1 picks up left fork 2.
Philosopher 1 is eating.
```

13) Write a program to demonstrate Readers Writers problem

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

sem_t rw_mutex;
sem_t mutex;
int read_count = 0;
int data = 0;

void* reader(void* param);
void* writer(void* param);

int main() {
  pthread_t readers[5], writers[2];
  int reader_ids[5], writer_ids[2];

  sem_init(&rw_mutex, 0, 1);
  sem_init(&mutex, 0, 1);

  for (int i = 0; i < 5; i++) {
    reader_ids[i] = i;
    pthread_create(&readers[i], NULL, reader, &reader_ids[i]);
  }
  for (int i = 0; i < 2; i++) {
    writer_ids[i] = i;
    pthread_create(&writers[i], NULL, writer, &writer_ids[i]);
  }

  for (int i = 0; i < 5; i++) {
    pthread_join(readers[i], NULL);
  }

  for (int i = 0; i < 2; i++) {
    pthread_join(writers[i], NULL);
  }

  sem_destroy(&rw_mutex);
  sem_destroy(&mutex);

  return 0;
}
void* reader(void* param) {
  int reader_id = *(int*)param;
  while (1) {
    sem_wait(&mutex);
    read_count++;
    if (read_count == 1) {
        sem_wait(&rw_mutex);
    }
    sem_post(&mutex);

    printf("Reader %d is reading data: %d\n", reader_id, data);
    sleep(rand() % 3);

    sem_wait(&mutex);
    read_count--;
    if (read_count == 0) {
        sem_post(&rw_mutex);
    }
    sem_post(&mutex);

    sleep(rand() % 3);
  }
}
void* writer(void* param) {
  int writer_id = *(int*)param;

  while (1) {
    sem_wait(&rw_mutex);

    data = rand() % 100;
    printf("Writer %d is writing data: %d\n", writer_id, data);
    sleep(rand() % 3);

    sem_post(&rw_mutex);

    sleep(rand() % 3);
  }
}
```

<u>Output:</u>

```
Reader 0 is reading data: 0
Reader 1 is reading data: 0
Writer 0 is writing data: 75
Reader 2 is reading data: 75
Writer 1 is writing data: 42
Reader 3 is reading data: 42
Reader 4 is reading data: 42
Writer 2 is writing data: 96
Reader 0 is reading data: 96
Reader 1 is reading data: 96
```

14) Write a Program to demonstrate all Arithmetic Operations in Shell Scripting

```
#!/bin/bash
# Arithmetic Operations in Shell Scripting
# Input variables
echo "Enter two numbers:"
read num1
read num2
# Addition
sum=$((num1 + num2))
echo "Sum: $sum"
# Subtraction
diff=$((num1 - num2))
echo "Difference: $diff"
# Multiplication
product=$((num1 * num2))
echo "Product: $product"
# Division
if [ $num2 -ne 0 ]; then
quotient=$((num1 / num2))
echo "Quotient: $quotient"
else
echo "Cannot divide by zero."
fi
# Modulus
if [ $num2 -ne 0 ]; then
remainder=$((num1 % num2))
echo "Remainder: $remainder"
else
echo "Cannot calculate remainder when dividing by zero."
fi
```

Output:

```
Enter two numbers:
10
0
Sum: 10
Difference: 10
Product: 0
Cannot divide by zero.
Cannot calculate remainder when dividing by zero.
```

15) Write a Program to demonstrate do-while loop in Shell Scripting

```
#!/bin/bash
# Initialize the variable
number=0

# Start of do-while loop
while true; do
   # Display prompt
   echo "Enter a positive number:"
    # Read user input
   read number

   # Check if the input is a positive number
   if [ $number -gt 0 ]; then
      break  # Exit the loop if a positive number is entered
   else
      echo "Invalid input. Please enter a positive number."
   fi
done
# End of do-while loop

echo "You entered a positive number: $number"
```

Output:

```
Enter a positive number: -5
Invalid input. Please enter a positive number.

Enter a positive number:
0
Invalid input. Please enter a positive number.

Enter a positive number:
7
You entered a positive number: 7
```

16) Write a Program to demonstrate if condition in Shell Scripting

```
#!/bin/bash
# Program to demonstrate if condition in Shell Scripting
# Input variables
echo "Enter a number:"
read number
# Check if the number is positive, negative, or zero
if [ $number -gt 0 ]; then
echo "The number is positive."
elif [ $number -lt 0 ]; then
echo "The number is negative."
else
echo "The number is zero."
fi
```

Output:

```
Enter a number: -5
The number is negative.
```

17) Write a Program to demonstrate CASE condition in Shell Scripting

```
#!/bin/bash
# Input variable
echo "Enter a number (1-3):"
read choice

# Switch-like behavior using case statement
case $choice in
  1)
    echo "You chose Option 1."
    ;;
  2)
    echo "You chose Option 2."
    ;;
  3)
    echo "You chose Option 3."
    ;;
  *)
    echo "Invalid choice. Please enter a number between 1 and 3."
    ;;
esac
```

<u>Output:</u>

```
Enter a number (1-3):
1
You chose Option 1.

Enter a number (1-3):
2
You chose Option 2.

Enter a number (1-3):
3
You chose Option 3.

Enter a number (1-3):
5
Invalid choice. Please enter a number between 1 and 3.
```

18) Write a Program to demonstrate logical operators in Shell Scripting

```
#!/bin/bash
# Program to demonstrate Logical Operators in Shell Scripting
# Input variables
echo "Enter two numbers:"
read num1
read num2
# Logical AND
if [ $num1 -gt 0 ] && [ $num2 -gt 0 ]; then
echo "Both numbers are positive."
else
echo "At least one of the numbers is not positive."
fi
# Logical OR
if [ $num1 -eq 0 ] || [ $num2 -eq 0 ]; then
echo "At least one of the numbers is zero."
else
echo "Neither of the numbers is zero."
fi
# Logical NOT
if [ ! $num1 -eq $num2 ]; then
echo "The two numbers are not equal."
else
echo "The two numbers are equal."
fi
```

Output:

```
Enter two numbers:
5
-3

At least one of the numbers is not positive.
Neither of the numbers is zero.
The two numbers are not equal.
```