Lab no: 4 **Date: 2082/05/15**

Title: Write a program to calculate the average turnaround time and waiting time for user input process parameters using FCFS process scheduling algorithm.

Introduction

The First Come First Serve (FCFS) scheduling algorithm is one of the simplest and most fundamental CPU scheduling techniques used in operating systems. In this method, processes are executed in the order in which they arrive in the ready queue, much like customers waiting in line at a ticket counter. The process that arrives first is executed first, and once a process starts execution, it runs until completion without being interrupted, making FCFS a non-preemptive scheduling algorithm.

In FCFS, the CPU acts like a queue where processes wait for their turn. Each process has certain parameters: Arrival Time (AT), which is when the process enters the ready queue; Burst Time (BT**)**, which is the amount of CPU time required by the process; Completion Time (CT), when the process finishes execution; Turnaround Time (TAT), which is the total time taken from arrival to completion; and Waiting Time (WT), which is the time a process spends waiting in the ready queue before execution. These values help in evaluating the performance of the scheduling algorithm.

The Turnaround Time (TAT) is calculated using the formula:

TAT=CT−ATTAT = CT - ATTAT=CT−AT

The Waiting Time (WT**)** is calculated using the formula:

WT=TAT−BTWT = TAT - BTWT=TAT−BT

FCFS scheduling is fair because every process gets CPU time in the order of its arrival, and it is also very easy to implement. However, one major drawback of FCFS is the convoy effect, where shorter processes may have to wait for a long time if a longer process arrives first.

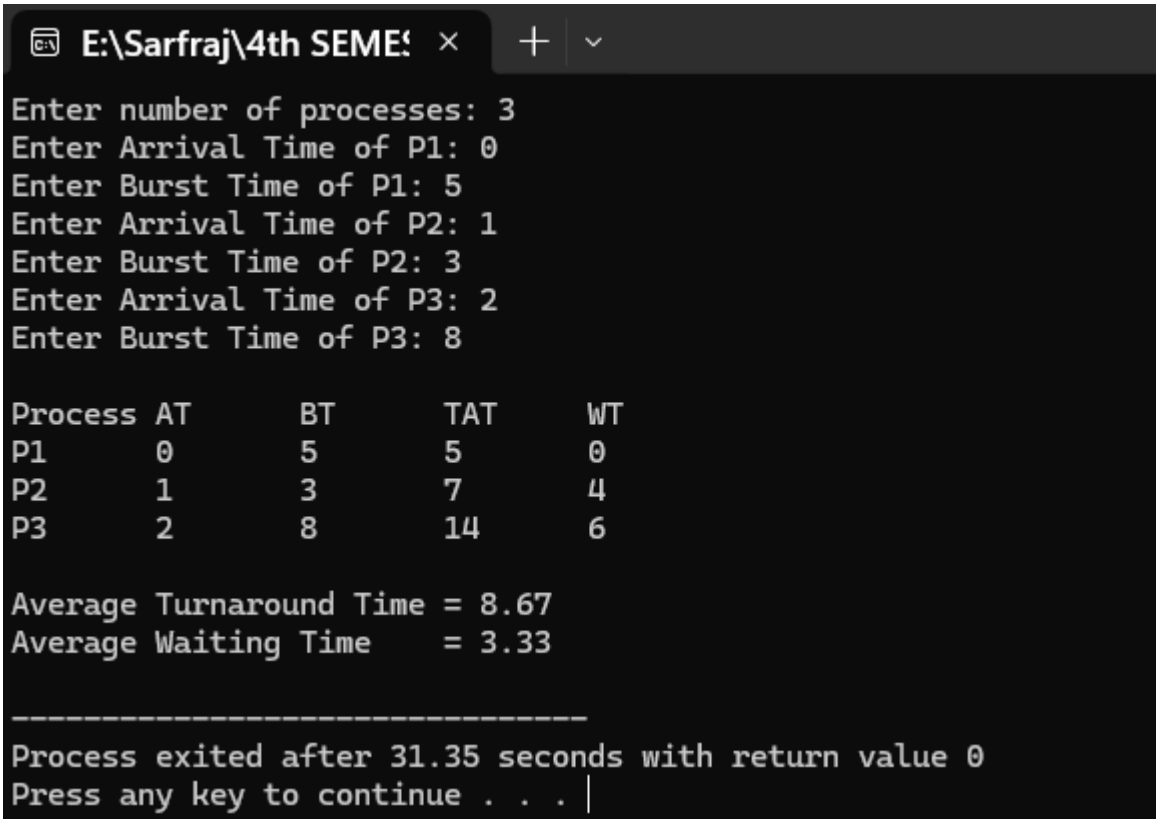
IDE: DEV C++

Source code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct Process {
   int pid;   // Process ID
   int at;    // Arrival Time
   int bt;    // Burst Time
   int ct;    // Completion Time
   int tat;   // Turnaround Time
   int wt;    // Waiting Time
};

// Comparator for stable_sort (keeps input order when AT ties)
bool cmpByAT(const Process& a, const Process& b) {
   return a.at < b.at;
}

int main() {
   int n;
   cout << "Enter number of processes: ";
   if (!(cin >> n) || n <= 0) {
      cout << "Invalid number of processes.\n";
      return 0;
   }

   vector<Process> p(n);
   for (int i = 0; i < n; i++) {
      p[i].pid = i + 1;
      cout << "Enter Arrival Time of P" << i + 1 << ": ";
      cin >> p[i].at;
      cout << "Enter Burst Time of P" << i + 1 << ": ";
      cin >> p[i].bt;
      p[i].ct = p[i].tat = p[i].wt = 0;
   }

   // Sort by arrival time; stable_sort preserves input order on ties
   stable_sort(p.begin(), p.end(), cmpByAT);

   int time = 0;
   for (int i = 0; i < n; i++) {
      if (time < p[i].at) {
```

```
            time = p[i].at; // CPU idle until next process arrives
        }
        time += p[i].bt;
        p[i].ct  = time;
        p[i].tat = p[i].ct - p[i].at;
        p[i].wt  = p[i].tat - p[i].bt;
    }

    double avg_tat = 0.0, avg_wt = 0.0;

    cout << "\nProcess\tAT\tBT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        cout << "P" << p[i].pid << "\t"
             << p[i].at << "\t"
             << p[i].bt << "\t"
             << p[i].tat << "\t"
             << p[i].wt  << "\n";
        avg_tat += p[i].tat;
        avg_wt  += p[i].wt;
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Turnaround Time = " << (avg_tat / n);
    cout << "\nAverage Waiting Time    = " << (avg_wt  / n) << endl;

    return 0;
}
```

Output:


E:\Sarfraj\4th SEME
Enter number of processes: 3
Enter Arrival Time of P1: 0
Enter Burst Time of P1: 5
Enter Arrival Time of P2: 1
Enter Burst Time of P2: 3
Enter Arrival Time of P3: 2
Enter Burst Time of P3: 8
Process AT BT TAT WT
P1 0 5 5 0
P2 1 3 7 4
P3 2 8 14 6
Average Turnaround Time = 8.67
Average Waiting Time = 3.33
Process exited after 31.35 seconds with return value 0
Press any key to continue . . .

Lab no: 5 **Date: 2082/05/14**

Title:Write a program to calculate the average turnaround time and waiting time for user input process parameters using SJF process scheduling algorithm.

Introduction

The Shortest Job First (SJF) scheduling algorithm is one of the most efficient CPU scheduling techniques used in operating systems. In this method, the process with the **smallest burst time** is executed first. If two processes have the same burst time, they are scheduled according to their order of arrival. SJF can be of two types: Non-Preemptive SJF, where once a process starts execution it runs until completion, and Preemptive SJF **(Shortest** Remaining Time First - SRTF), where a newly arrived process with a shorter burst time can preempt the currently running process.

In SJF, the CPU gives priority to processes that require less execution time, which minimizes the average waiting time and average turnaround time compared to FCFS. Each process has the same set of parameters as in other scheduling algorithms: Arrival Time (AT), Burst Time (BT), Completion Time (CT), Turnaround Time (TAT), and Waiting Time (WT).

The Turnaround Time (TAT) is calculated as:

TAT=CT−ATTAT = CT - ATTAT=CT−AT
The Waiting Time (WT) is calculated as:
WT=TAT−BTWT = TAT - BTWT=TAT−BT

SJF is considered the optimal scheduling algorithm for minimizing average waiting time. However, its major drawback is that it requires knowledge of the CPU burst time in advance, which is not always possible. Moreover, in the non-preemptive version, a longer process may block shorter ones that arrive later, and in the preemptive version, frequent context switching may occur.

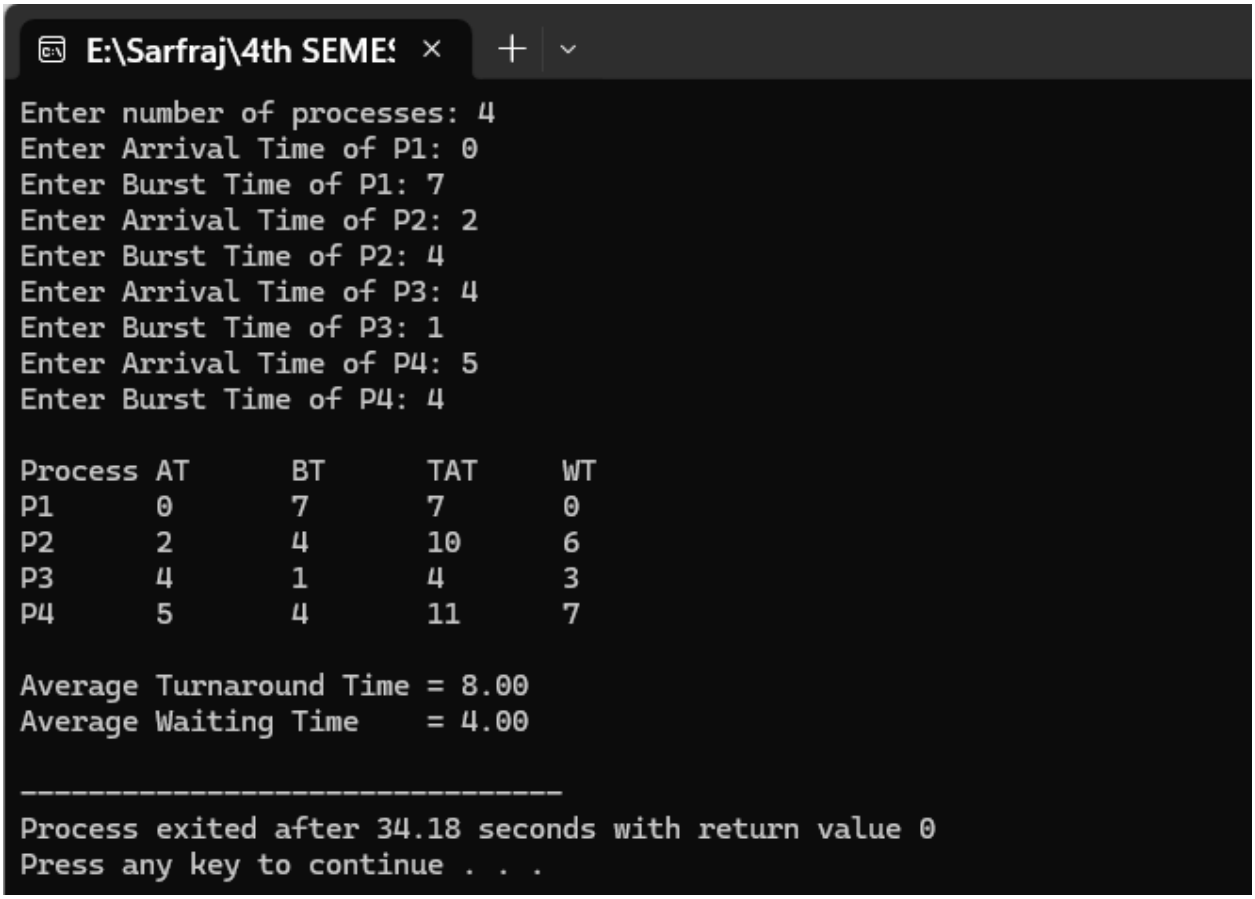
IDE: DEV C++

Source code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

struct Process {
    int pid;   // Process ID
    int at;    // Arrival Time
    int bt;    // Burst Time
    int ct;    // Completion Time
    int tat;   // Turnaround Time
    int wt;    // Waiting Time
    bool done; // Flag to mark completion
};

int main() {
    int n;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> p(n);
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        cout << "Enter Arrival Time of P" << i + 1 << ": ";
        cin >> p[i].at;
        cout << "Enter Burst Time of P" << i + 1 << ": ";
        cin >> p[i].bt;
        p[i].ct = p[i].tat = p[i].wt = 0;
        p[i].done = false;
    }

    int completed = 0, time = 0;
    while (completed < n) {
        int idx = -1;
        int minBT = 1e9;

        // Select process with shortest burst time among arrived & not done
        for (int i = 0; i < n; i++) {
            if (!p[i].done && p[i].at <= time) {
                if (p[i].bt < minBT) {
                    minBT = p[i].bt;
                    idx = i;
                } else if (p[i].bt == minBT) {
```

```
                // Tie-breaking by arrival time
                if (p[i].at < p[idx].at) idx = i;
            }
        }
    }

    if (idx == -1) {
        time++; // CPU idle
    } else {
        time += p[idx].bt;
        p[idx].ct = time;
        p[idx].tat = p[idx].ct - p[idx].at;
        p[idx].wt = p[idx].tat - p[idx].bt;
        p[idx].done = true;
        completed++;
    }
  }

  double avg_tat = 0.0, avg_wt = 0.0;

  cout << "\nProcess\tAT\tBT\tTAT\tWT\n";
  for (int i = 0; i < n; i++) {
    cout << "P" << p[i].pid << "\t"
         << p[i].at << "\t"
         << p[i].bt << "\t"
         << p[i].tat << "\t"
         << p[i].wt << "\n";
    avg_tat += p[i].tat;
    avg_wt += p[i].wt;
  }

  cout << fixed << setprecision(2);
  cout << "\nAverage Turnaround Time = " << (avg_tat / n);
  cout << "\nAverage Waiting Time    = " << (avg_wt / n) << endl;

  return 0;
}
```

Output:


E:\Sarfraj\4th SEME
Enter number of processes: 4
Enter Arrival Time of P1: 0
Enter Burst Time of P1: 7
Enter Arrival Time of P2: 2
Enter Burst Time of P2: 4
Enter Arrival Time of P3: 4
Enter Burst Time of P3: 1
Enter Arrival Time of P4: 5
Enter Burst Time of P4: 4
Process AT BT TAT WT
P1 0 7 7 0
P2 2 4 10 6
P3 4 1 4 3
P4 5 4 11 7
Average Turnaround Time = 8.00
Average Waiting Time = 4.00
Process exited after 34.18 seconds with return value 0
Press any key to continue . . .

Lab no: 6 **Date: 2082/05/14**

Title:Write a program to calculate the average turnaround time and waiting time for user input process parameters using RR process scheduling algorithm.

Introduction

The Round Robin (RR) scheduling algorithm is one of the most widely used preemptive CPU scheduling techniques in operating systems. In this method, each process is assigned a fixed time slice or time quantum, and the CPU executes each process for that duration in a cyclic order. If a process does not finish within its time quantum, it is preempted and placed at the back of the ready queue, and the CPU is given to the next process. This continues until all processes are completed.

Round Robin scheduling is designed especially for time-sharing systems, as it ensures that no single process monopolizes the CPU and that all processes get an equal share of CPU time. Each process has the same set of parameters as in other scheduling algorithms: Arrival Time (AT), Burst Time (BT), Completion Time (CT), Turnaround Time (TAT), and Waiting Time (WT).

The Turnaround Time (TAT) is calculated as:

TAT=CT−ATTAT = CT - ATTAT=CT−AT

The Waiting Time (WT) is calculated as:

WT=TAT−BTWT = TAT - BTWT=TAT−BT

Round Robin is considered a fair scheduling algorithm because each process gets an equal opportunity to run. Its performance heavily depends on the choice of the time quantum. If the time quantum is too large, RR behaves like FCFS. If the time quantum is too small, too much context switching occurs, which reduces CPU efficiency.

IDE: DEV C++

Source code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <iomanip>
using namespace std;

struct Process {
    int pid;   // Process ID
    int at;    // Arrival Time
    int bt;    // Burst Time
    int rt;    // Remaining Time
    int ct;    // Completion Time
    int tat;   // Turnaround Time
    int wt;    // Waiting Time
};

int main() {
    int n, tq;
    cout << "Enter number of processes: ";
    cin >> n;

    vector<Process> p(n);
```

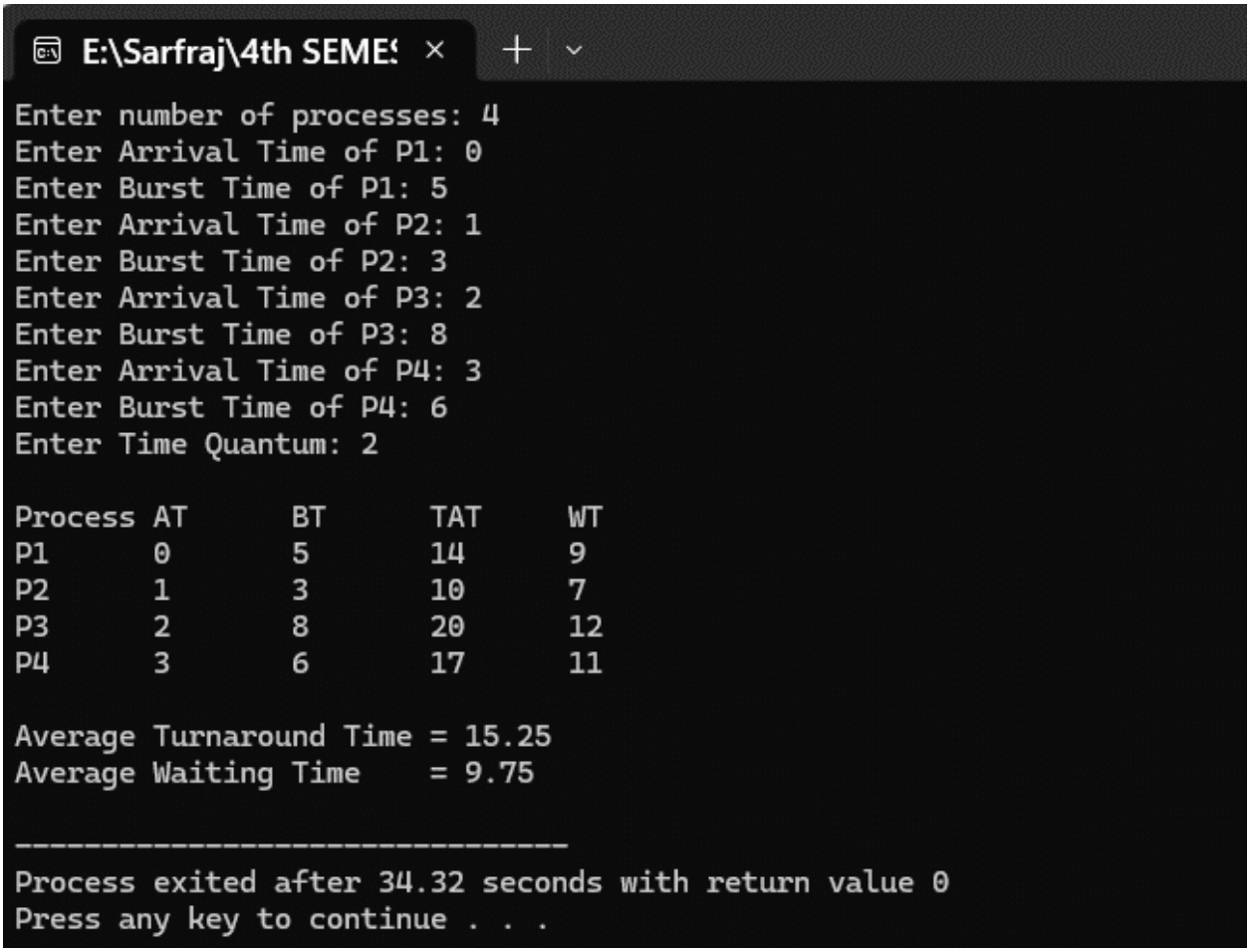
```
for (int i = 0; i < n; i++) {
    p[i].pid = i + 1;
    cout << "Enter Arrival Time of P" << i + 1 << ": ";
    cin >> p[i].at;
    cout << "Enter Burst Time of P" << i + 1 << ": ";
    cin >> p[i].bt;
    p[i].rt = p[i].bt; // initially remaining time = burst time
    p[i].ct = p[i].tat = p[i].wt = 0;
}

cout << "Enter Time Quantum: ";
cin >> tq;

queue<int> q;  // stores process indices
int time = 0, completed = 0;
vector<bool> inQueue(n, false);

// Initially push processes that arrive at time 0
for (int i = 0; i < n; i++) {
    if (p[i].at == 0) {
        q.push(i);
        inQueue[i] = true;
    }
```

```
}

while (completed < n) {
    if (q.empty()) {
        time++; // CPU idle
        for (int i = 0; i < n; i++) {
            if (!inQueue[i] && p[i].at <= time) {
                q.push(i);
                inQueue[i] = true;
            }
        }
        continue;
    }

    int idx = q.front();
    q.pop();

    if (p[idx].rt > tq) {
        time += tq;
        p[idx].rt -= tq;
    } else {
        time += p[idx].rt;
        p[idx].rt = 0;
```

```
            p[idx].ct = time;
            p[idx].tat = p[idx].ct - p[idx].at;
            p[idx].wt = p[idx].tat - p[idx].bt;
            completed++;
        }

        // Push newly arrived processes into queue
        for (int i = 0; i < n; i++) {
            if (!inQueue[i] && p[i].at <= time && p[i].rt > 0) {
                q.push(i);
                inQueue[i] = true;
            }
        }

        // If current process not finished, push it back
        if (p[idx].rt > 0) {
            q.push(idx);
        }
    }

    double avg_tat = 0.0, avg_wt = 0.0;

    cout << "\nProcess\tAT\tBT\tTAT\tWT\n";
```

```
    for (int i = 0; i < n; i++) {
        cout << "P" << p[i].pid << "\t"
             << p[i].at << "\t"
             << p[i].bt << "\t"
             << p[i].tat << "\t"
             << p[i].wt << "\n";
        avg_tat += p[i].tat;
        avg_wt += p[i].wt;
    }

    cout << fixed << setprecision(2);
    cout << "\nAverage Turnaround Time = " << (avg_tat / n);
    cout << "\nAverage Waiting Time    = " << (avg_wt / n) << endl;

    return 0;
}
```

Output:


E:\Sarfraj\4th SEME
Enter number of processes: 4
Enter Arrival Time of P1: 0
Enter Burst Time of P1: 5
Enter Arrival Time of P2: 1
Enter Burst Time of P2: 3
Enter Arrival Time of P3: 2
Enter Burst Time of P3: 8
Enter Arrival Time of P4: 3
Enter Burst Time of P4: 6
Enter Time Quantum: 2
Process AT BT TAT WT
P1 0 5 14 9
P2 1 3 10 7
P3 2 8 20 12
P4 3 6 17 11
Average Turnaround Time = 15.25
Average Waiting Time = 9.75
Process exited after 34.32 seconds with return value 0
Press any key to continue . . .

Lab no: 7 **Date: 2082/05/14**

Title:Write a program to display the process allocation for user input free blocks and incoming process using Best fit allocation.

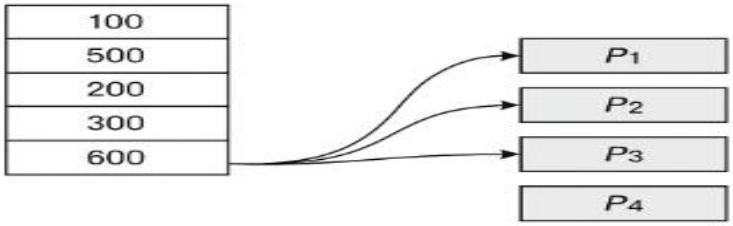
Introduction

The Best Fit Allocation algorithm is a memory management strategy used in operating systems to allocate processes into available memory blocks in the most space-efficient way possible. When a process requests memory, the operating system searches through the list of free blocks and assigns it to the smallest block that is large enough to hold the process. This approach aims to minimize unused space inside the block, reducing internal fragmentation**.**

Compared to First Fit (which allocates the first suitable block) and Worst Fit (which allocates the largest available block), Best Fit is more selective, as it tries to utilize memory blocks more efficiently by leaving behind the least amount of leftover space.

However, the Best Fit method has some drawbacks:

- It may lead to external fragmentation, since it leaves behind many very small unusable gaps in memory.
- It requires scanning the entire list of memory blocks to find the "best" match, making it slower than First Fit in practice.



Process	Size	Block Allocated
P1	212	Block 4
P2	417	Block 2
P3	112	Block 3
P4	426	Block 5

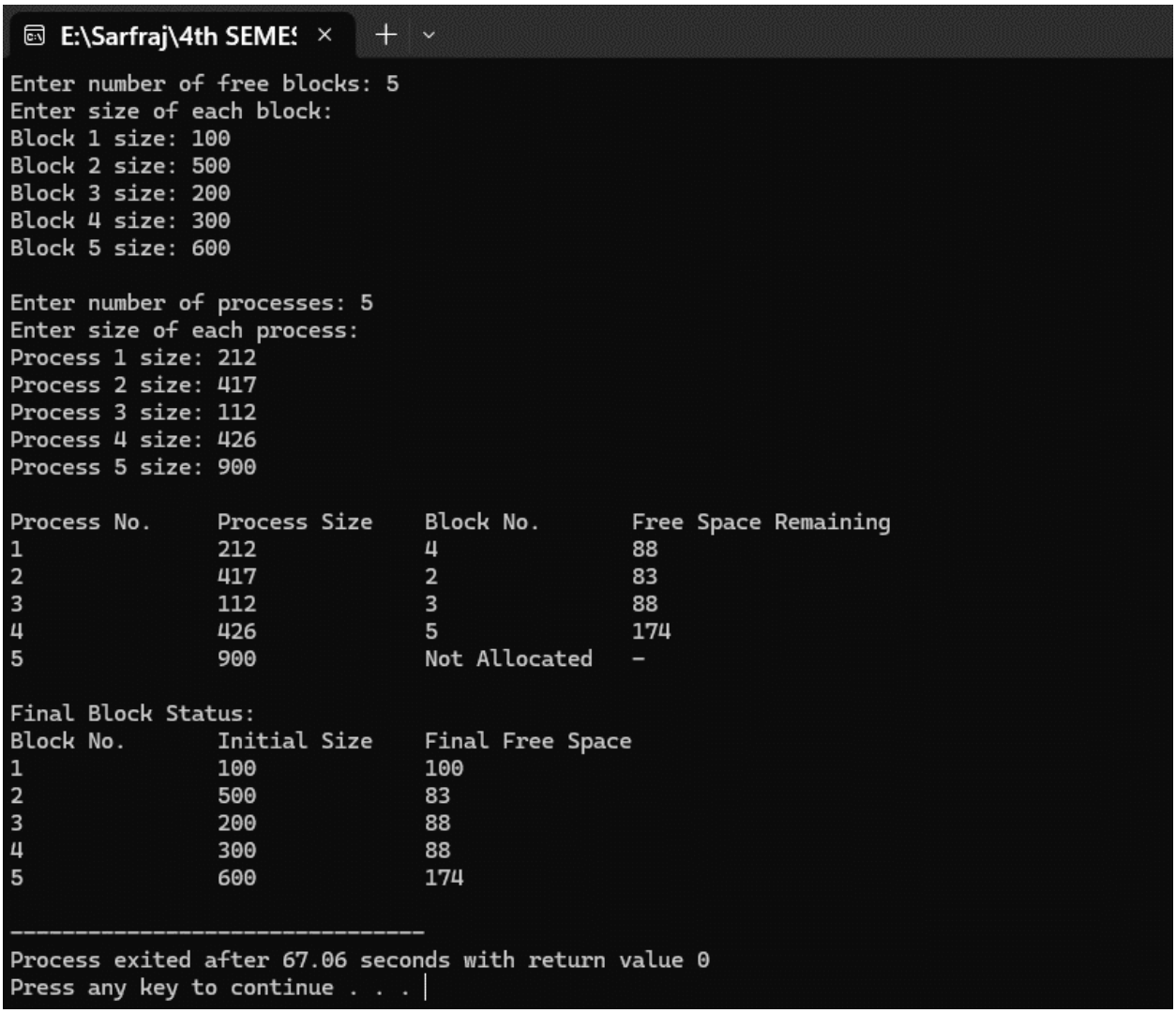
IDE: DEV C++

Source code:

```
#include <stdio.h>
int main() {
    int nb, np;
    int blockSize[20], initialBlockSize[20], processSize[20], allocation[20], freeSpace[20];

    // Input number of blocks
    printf("Enter number of free blocks: ");
    scanf("%d", &nb);

    printf("Enter size of each block:\n");
    for (int i = 0; i < nb; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &blockSize[i]);
        initialBlockSize[i] = blockSize[i]; // Save original size for later
        freeSpace[i] = blockSize[i];        // Initially all space is free
    }

    // Input number of processes
    printf("\nEnter number of processes: ");
    scanf("%d", &np);

    printf("Enter size of each process:\n");
    for (int i = 0; i < np; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &processSize[i]);
        allocation[i] = -1; // Initially not allocated
    }

    // Best Fit Allocation
    for (int i = 0; i < np; i++) {
        int bestIndex = -1;
        for (int j = 0; j < nb; j++) {
            if (freeSpace[j] >= processSize[i]) {
                if (bestIndex == -1 || freeSpace[j] < freeSpace[bestIndex]) {
                    bestIndex = j;
                }
            }
        }
```

```
        if (bestIndex != -1) {
            allocation[i] = bestIndex;
            freeSpace[bestIndex] -= processSize[i];
        }
    }

    // Process Allocation Table
    printf("\nProcess No.\tProcess Size\tBlock No.\tFree Space Remaining\n");
    for (int i = 0; i < np; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d\t\t%d", allocation[i] + 1, freeSpace[allocation[i]]);
        } else {
            printf("Not Allocated\t-");
        }
        printf("\n");
    }

    // Final Block Status Table
    printf("\nFinal Block Status:\n");
    printf("Block No.\tInitial Size\tFinal Free Space\n");
    for (int i = 0; i < nb; i++) {
        printf("%d\t\t%d\t\t%d\n", i + 1, initialBlockSize[i], freeSpace[i]);
    }

    return 0;
}
```

Output:


E:\Sarfraj\4th SEME
Enter number of free blocks: 5
Enter size of each block:
Block 1 size: 100
Block 2 size: 500
Block 3 size: 200
Block 4 size: 300
Block 5 size: 600
Enter number of processes: 5
Enter size of each process:
Process 1 size: 212
Process 2 size: 417
Process 3 size: 112
Process 4 size: 426
Process 5 size: 900
Process No. Process Size Block No. Free Space Remaining
1 212 4 88
2 417 2 83
3 112 3 88
4 426 5 174
5 900 Not Allocated -
Final Block Status:
Block No. Initial Size Final Free Space
1 100 100
2 500 83
3 200 88
4 300 88
5 600 174
Process exited after 67.06 seconds with return value 0
Press any key to continue . . .

Lab no: 8 **Date: 2082/05/14**

Title:Write a program to display the process allocation for user input free blocks and incoming process using Worst fit allocation.

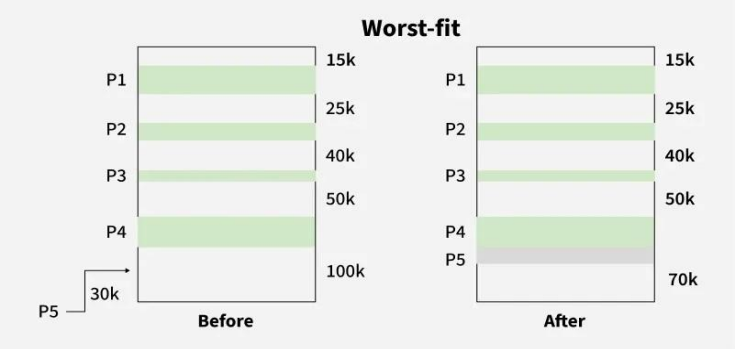
Introduction

The Worst Fit Allocation algorithm is a memory management strategy used in operating systems to allocate processes into available memory blocks. Unlike Best Fit, where the process is assigned to the smallest suitable block, in Worst Fit the process is allocated to the largest available free block that can accommodate it. The idea behind this method is to leave behind the largest possible remaining free space, thereby reducing the chances of creating very small unusable fragments.

Compared to First Fit (which allocates the first suitable block) and Best Fit (which allocates the smallest suitable block), Worst Fit tends to spread processes more evenly across memory and avoids filling small blocks quickly. This can sometimes reduce the occurrence of small leftover gaps that cannot be used later.

However, the Worst Fit method also has drawbacks:

- It may lead to poor utilization of memory, since large blocks get split frequently, leaving fragmented free spaces.
- Like Best Fit, it requires scanning the entire list of free blocks to find the “worst” match, which increases allocation time.



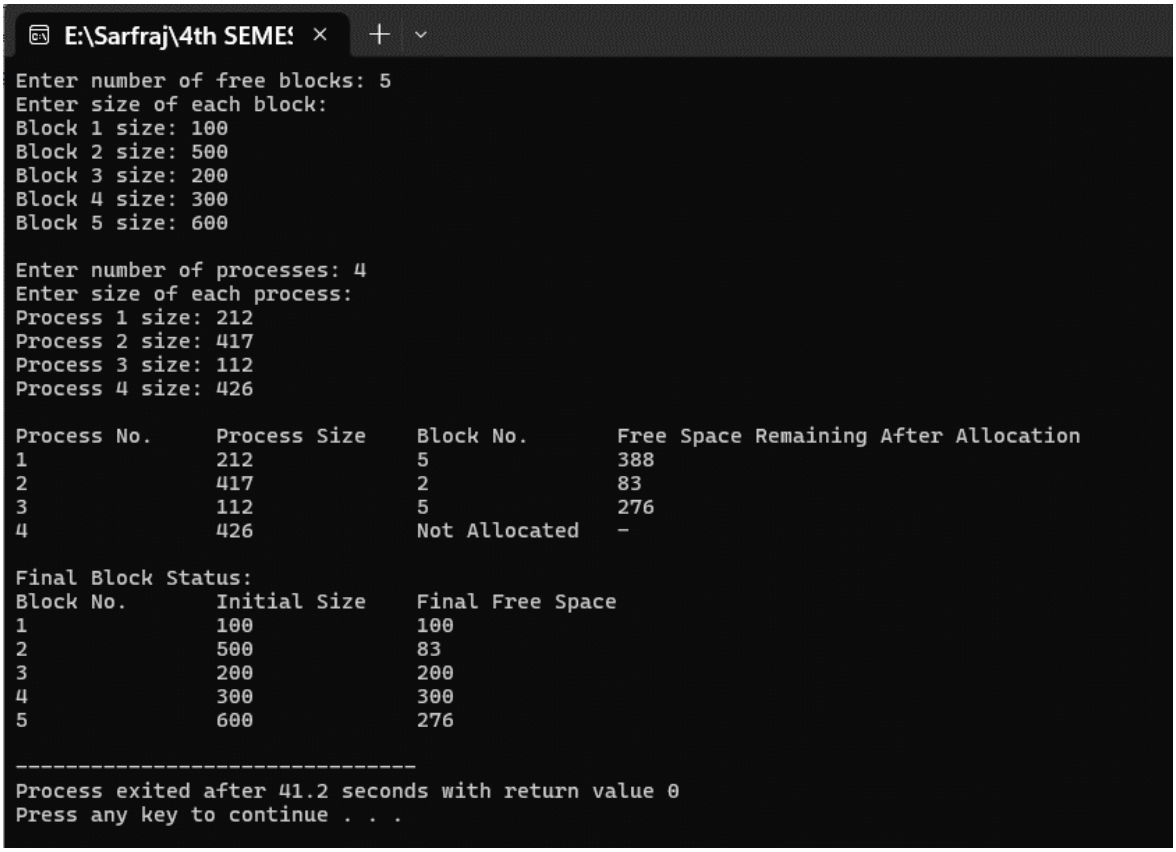
IDE: DEV C++

Source code:

```
#include <stdio.h>

int main() {
   int nb, np,i,j;
   int blockSize[20], initialBlockSize[20], processSize[20], allocation[20], freeSpace[20], usedSpaceAfterAlloc[20];

   // Input number of blocks
   printf("Enter number of free blocks: ");
   scanf("%d", &nb);

   printf("Enter size of each block:\n");
   for ( i = 0; i < nb; i++) {
      printf("Block %d size: ", i + 1);
      scanf("%d", &blockSize[i]);
      initialBlockSize[i] = blockSize[i]; // Store initial size
      freeSpace[i] = blockSize[i];        // Initially all space is free
   }

   // Input number of processes
   printf("\nEnter number of processes: ");
   scanf("%d", &np);

   printf("Enter size of each process:\n");
   for (i = 0; i < np; i++) {
      printf("Process %d size: ", i + 1);
      scanf("%d", &processSize[i]);
      allocation[i] = -1; // Not allocated initially
      usedSpaceAfterAlloc[i] = -1; // Default for not allocated
   }

   // Worst Fit Allocation
   for (i = 0; i < np; i++) {
      int worstIndex = -1;
      for (j = 0; j < nb; j++) {
         if (freeSpace[j] >= processSize[i]) {
            if (worstIndex == -1 || freeSpace[j] > freeSpace[worstIndex]) {
               worstIndex = j;
            }
         }
      }

      if (worstIndex != -1) {
         allocation[i] = worstIndex;
```

```
            freeSpace[worstIndex] -= processSize[i]; // Update free space
            usedSpaceAfterAlloc[i] = freeSpace[worstIndex]; // Record remaining space at this moment
        }
    }

    // Process Allocation Table
    printf("\nProcess No.\tProcess Size\tBlock No.\tFree Space Remaining After Allocation\n");
    for ( i = 0; i < np; i++) {
        printf("%d\t\t%d\t\t", i + 1, processSize[i]);
        if (allocation[i] != -1) {
            printf("%d\t\t%d", allocation[i] + 1, usedSpaceAfterAlloc[i]);
        } else {
            printf("Not Allocated\t-");
        }
        printf("\n");
    }

    // Final Block Status Table
    printf("\nFinal Block Status:\n");
    printf("Block No.\tInitial Size\tFinal Free Space\n");
    for (i = 0; i < nb; i++) {
        printf("%d\t\t%d\t\t%d\n", i + 1, initialBlockSize[i], freeSpace[i]);
    }

    return 0;
}
```

Output:


E:\Sarfraj\4th SEME:
Enter number of free blocks: 5
Enter size of each block:
Block 1 size: 100
Block 2 size: 500
Block 3 size: 200
Block 4 size: 300
Block 5 size: 600
Enter number of processes: 4
Enter size of each process:
Process 1 size: 212
Process 2 size: 417
Process 3 size: 112
Process 4 size: 426
Process No. Process Size Block No. Free Space Remaining After Allocation
1 212 5 388
2 417 2 83
3 112 5 276
4 426 Not Allocated -
Final Block Status:
Block No. Initial Size Final Free Space
1 100 100
2 500 83
3 200 200
4 300 300
5 600 276
Process exited after 41.2 seconds with return value 0
Press any key to continue . . .

Lab no: 9 **Date: 2082/05/14**

Title:Write a program to display the process allocation for user input free blocks and incoming process using First fit allocation.

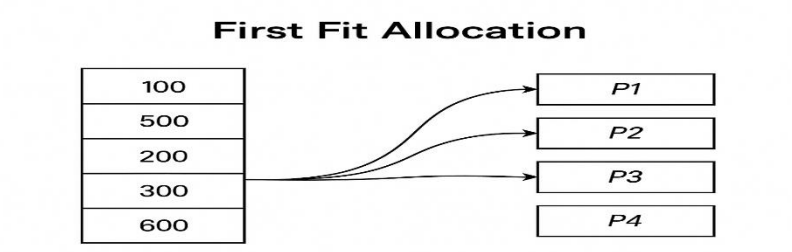
Introduction

The First Fit Allocation algorithm is a memory management strategy used in operating systems to allocate processes into available memory blocks. In this method, the operating system scans the list of free blocks sequentially and assigns the process to the first free block that is large enough to hold it. This makes First Fit one of the simplest and fastest allocation techniques.

Compared to Best Fit (which assigns the smallest suitable block) and Worst Fit (which assigns the largest suitable block), First Fit does not aim for optimal space utilization but instead prioritizes speed, since it stops searching as soon as a suitable block is found. This often results in faster allocation compared to the other methods.

However, the First Fit method also has drawbacks:

- It can lead to external fragmentation, as small gaps may be left unutilized between allocated blocks.
- Over time, the lower part of memory tends to fill up quickly, and larger processes may have to wait even though sufficient free memory exists in higher blocks.



Process	Size	Block Allocated
P1	212	Block 2
P2	417	Block 1
P3	112	Block 3
P4	426	Block 4

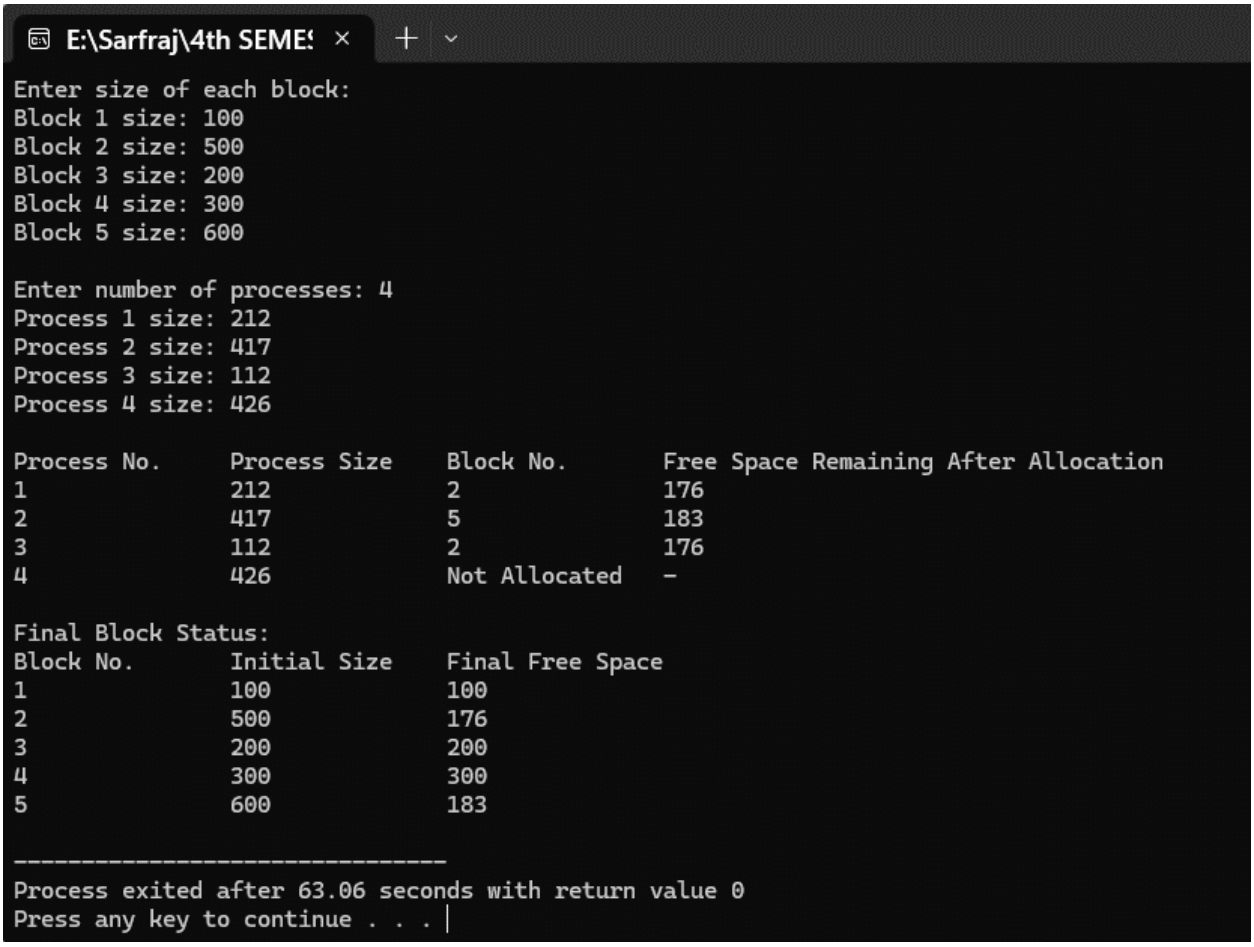
IDE: DEV C++

Source code:

```
#include <stdio.h>

int main() {
    int nb, np;

   // Step 1: Input number of free memory blocks
    printf("Enter number of free blocks: ");
    scanf("%d", &nb);

    // Arrays to store block information
    int blockSize[nb], blockFree[nb], initialBlockSize[nb];

    // Step 2: Input size of each memory block
    printf("Enter size of each block:\n");
    for (int i = 0; i < nb; i++) {
        printf("Block %d size: ", i + 1);
        scanf("%d", &blockSize[i]);

        blockFree[i] = blockSize[i];         // Initially, all space is free
        initialBlockSize[i] = blockSize[i];  // Keep original size for final report
    }

    // Step 3: Input number of processes
    printf("\nEnter number of processes: ");
    scanf("%d", &np);

    int processSize[np], allocation[np];

    // Step 4: Input process sizes
    for (int i = 0; i < np; i++) {
        printf("Process %d size: ", i + 1);
        scanf("%d", &processSize[i]);
        allocation[i] = -1; // Initially, no process is allocated
    }

    // Step 5: First Fit Allocation Algorithm
    for (int i = 0; i < np; i++) {       // For each process
        for (int j = 0; j < nb; j++) {   // Find first block that fits
            if (blockFree[j] >= processSize[i]) {
```

```
            allocation[i] = j;          // Allocate block j to process i
            blockFree[j] -= processSize[i]; // Reduce free space in the block
            break; // Stop searching once allocated (First Fit)
        }
    }
  }

  // Step 6: Display Process Allocation Table
  printf("\nProcess No.\tProcess Size\tBlock No.\tFree Space Remaining After Allocation\n");
  for (int i = 0; i < np; i++) {
    printf("%d\t\t%d\t\t", i + 1, processSize[i]);
    if (allocation[i] != -1)
      printf("%d\t\t%d", allocation[i] + 1, blockFree[allocation[i]]);
    else
      printf("Not Allocated\t-");
    printf("\n");
  }

  // Step 7: Display Final Block Status
  printf("\nFinal Block Status:\n");
  printf("Block No.\tInitial Size\tFinal Free Space\n");
  for (int i = 0; i < nb; i++) {
    printf("%d\t\t%d\t\t%d\n", i + 1, initialBlockSize[i], blockFree[i]);
  }

  return 0;
}
```

Output:


E:\Sarfraj\4th SEME
Enter size of each block:
Block 1 size: 100
Block 2 size: 500
Block 3 size: 200
Block 4 size: 300
Block 5 size: 600
Enter number of processes: 4
Process 1 size: 212
Process 2 size: 417
Process 3 size: 112
Process 4 size: 426
Process No. Process Size Block No. Free Space Remaining After Allocation
1 212 2 176
2 417 5 183
3 112 2 176
4 426 Not Allocated -
Final Block Status:
Block No. Initial Size Final Free Space
1 100 100
2 500 176
3 200 200
4 300 300
5 600 183
Process exited after 63.06 seconds with return value 0
Press any key to continue . . .

Lab no: 10 **Date: 2082/07/12**

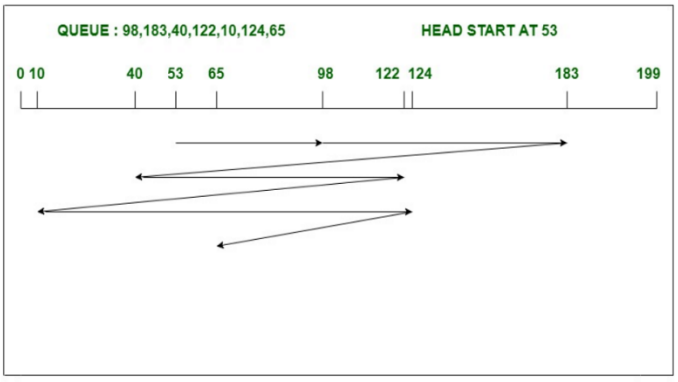
Title:Write a program to calculate the seek time for user input pending request, total number of cylinder and current position of I/O read/write head using FCFS disk scheduling algorithm.

Introduction

The First-Come-First-Served (FCFS) disk scheduling algorithm is the simplest disk scheduling technique that processes disk I/O requests in the order they arrive in the request queue. In this algorithm, the disk arm moves to satisfy each request in the sequence they were received, without any optimization for minimizing seek time.

How FCFS Works:

1. Requests are processed in the order they arrive (First-In-First-Out)
2. The disk head moves from its current position to the first request in the queue
3. After servicing the first request, it moves to the second request, and so on
4. The process continues until all requests are satisfied
5. Total seek time is calculated as the sum of distances moved between consecutive requests



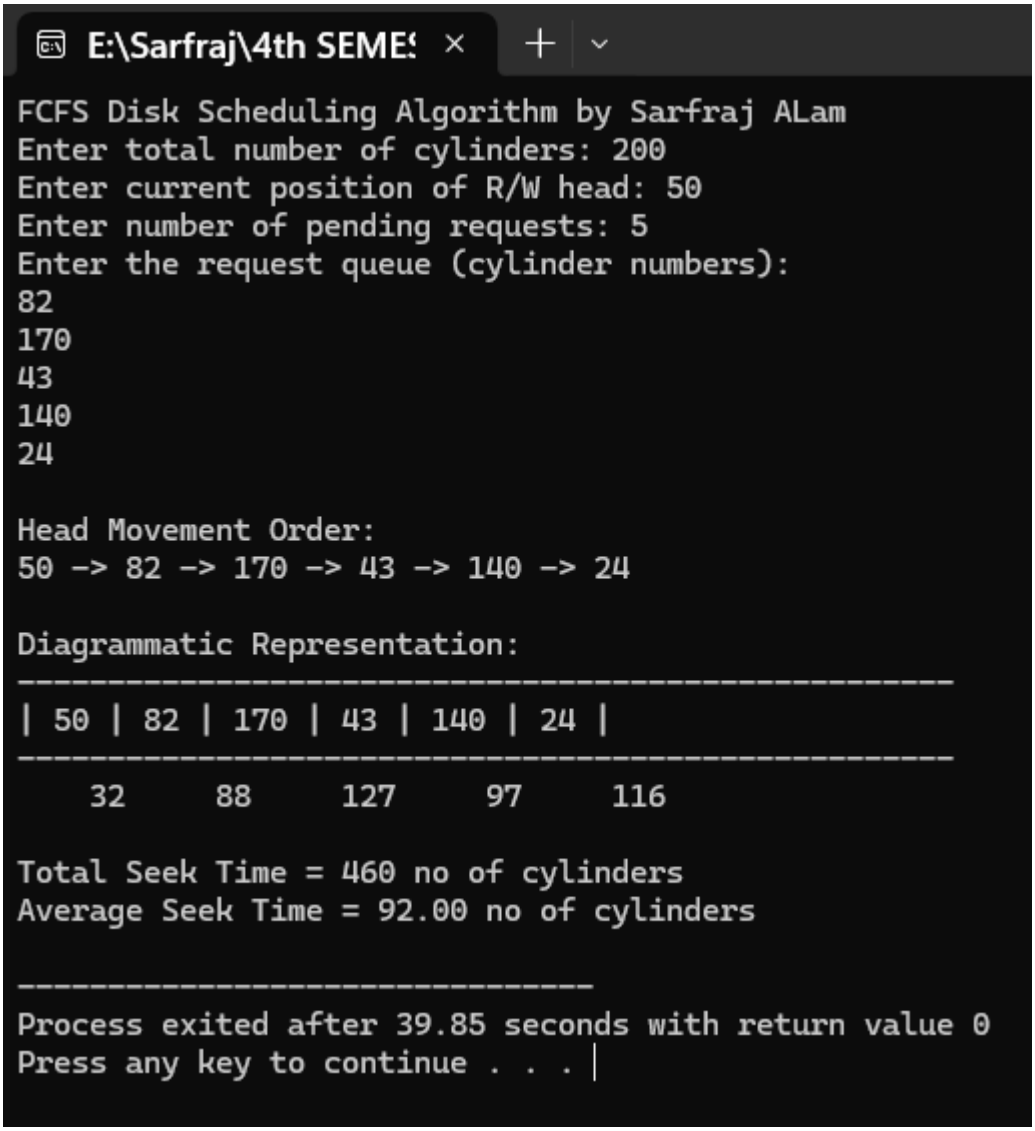
IDE: DEV C++
Language: C

Source code:

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

int main() {
    int n, head, totalCylinders;
    int seekTime = 0;

    printf("FCFS Disk Scheduling Algorithm by Sarfraj Alam\n");

    // Input total cylinders
    printf("Enter total number of cylinders: ");
    scanf("%d", &totalCylinders);

    // Input head position
    printf("Enter current position of R/W head: ");
    scanf("%d", &head);

    // Input request queue size
    printf("Enter number of pending requests: ");
    scanf("%d", &n);

    int requests[n];
    printf("Enter the request queue (cylinder numbers):\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
        if (requests[i] < 0 || requests[i] >= totalCylinders) {
            printf("Invalid request: %d (out of range)\n", requests[i]);
            return 1;
        }
    }

    // FCFS scheduling
    int current = head;
    printf("\nHead Movement Order:\n");
    printf("%d", current);
    for (int i = 0; i < n; i++) {
        seekTime += abs(current - requests[i]);
        current = requests[i];
        printf(" -> %d", current);
    }
```

```
    // Gantt-style diagram
    printf("\n\nDiagrammatic Representation:\n");
    printf("-----------------------------------------------------\n");
    current = head;
    printf("| %d ", current);
    for (int i = 0; i < n; i++) {
        printf("| %d ", requests[i]);
    }
    printf("|\n");
    printf("-----------------------------------------------------\n");

    // Distances (seek)
    printf("   ");
    current = head;
    for (int i = 0; i < n; i++) {
        int dist = abs(current - requests[i]);
        printf("  %d   ", dist);
        current = requests[i];
    }
    printf("\n");

    // Results
    printf("\nTotal Seek Time = %d no of cylinders\n", seekTime);
    printf("Average Seek Time = %.2f no of cylinders\n", (float)seekTime / n);

    return 0;
}
```

Output:


E:\Sarfraj\4th SEME
FCFS Disk Scheduling Algorithm by Sarfraj ALam
Enter total number of cylinders: 200
Enter current position of R/W head: 50
Enter number of pending requests: 5
Enter the request queue (cylinder numbers):
82
170
43
140
24
Head Movement Order:
50 -> 82 -> 170 -> 43 -> 140 -> 24
Diagrammatic Representation:
| 50 | 82 | 170 | 43 | 140 | 24 |
32 88 127 97 116
Total Seek Time = 460 no of cylinders
Average Seek Time = 92.00 no of cylinders
Process exited after 39.85 seconds with return value 0
Press any key to continue . . .

Lab no: 11 **Date: 2082/07/12**

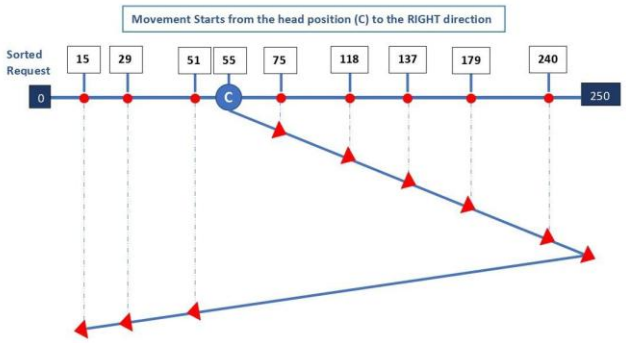
Title:Write a program to calculate the seek time for user input pending request, total number of cylinder and current position of I/O read/write head using SCAN disk scheduling algorithm.

Introduction

The SCAN disk scheduling algorithm, also called the “Elevator Algorithm”, is one of the most commonly used disk scheduling techniques. The algorithm moves the read/write head in one fixed direction (either left or right), servicing all pending requests on its path. Once it reaches the end of the disk, the head reverses direction and continues servicing requests in the opposite direction, similar to how an elevator moves up and down in a building.

.How SCAN Works:

1. The disk arm begins moving in one direction (either toward cylinder 0 or toward the last cylinder).
2. All pending requests in that direction are serviced as the head moves.
3. When the head reaches the last cylinder in the current direction, it reverses direction.
4. The disk arm then services all remaining requests while moving back in the opposite direction.
5. This process continues until all requests in the queue are processed**.**



IDE: DEV C++
Language: C

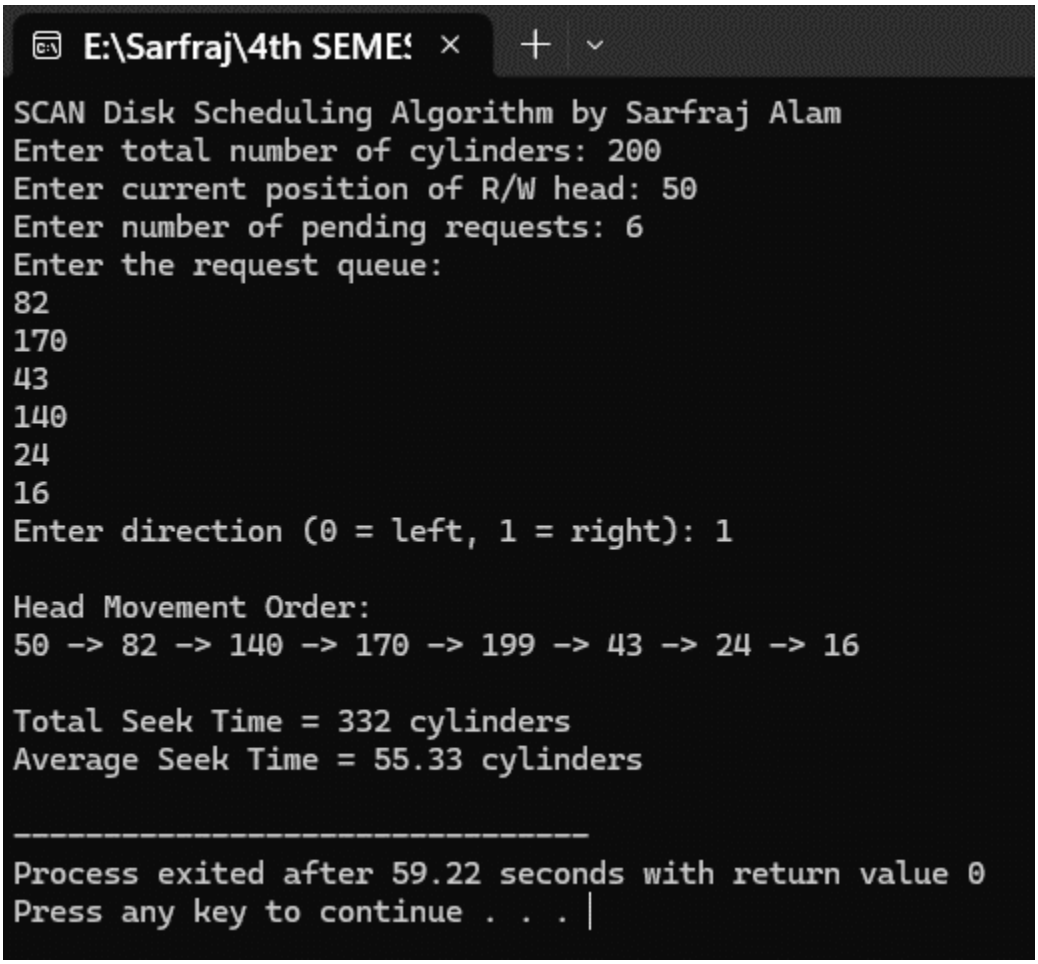
Source code:

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

int main() {
    int totalCylinders, head, n, direction;
    int seekTime = 0;

    printf("SCAN Disk Scheduling Algorithm by Sarfraj Alam\n");

    // Input total cylinders
    printf("Enter total number of cylinders: ");
    scanf("%d", &totalCylinders);

    // Input head position
    printf("Enter current position of R/W head: ");
    scanf("%d", &head);

    // Input size of request queue
    printf("Enter number of pending requests: ");
    scanf("%d", &n);

    int requests[n];
    printf("Enter the request queue:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
        if (requests[i] < 0 || requests[i] >= totalCylinders) {
            printf("Invalid request %d (out of range)\n", requests[i]);
            return 1;
        }
    }

    // Input scan direction
    printf("Enter direction (0 = left, 1 = right): ");
    scanf("%d", &direction);

    // Add head into list for easier processing
    int allRequests[n + 1];
    for (int i = 0; i < n; i++) {
        allRequests[i] = requests[i];
    }
```

```
allRequests[n] = head;

// Sort all requests (including head)
for (int i = 0; i < n + 1; i++) {
    for (int j = i + 1; j < n + 1; j++) {
        if (allRequests[i] > allRequests[j]) {
            int temp = allRequests[i];
            allRequests[i] = allRequests[j];
            allRequests[j] = temp;
        }
    }
}

// Find head index in sorted array
int headIndex = 0;
for (int i = 0; i < n + 1; i++) {
    if (allRequests[i] == head) {
        headIndex = i;
        break;
    }
}

printf("\nHead Movement Order:\n%d", head);

// SCAN to the right
if (direction == 1) {
    // Move from head to right end
    for (int i = headIndex + 1; i < n + 1; i++) {
        seekTime += abs(head - allRequests[i]);
        head = allRequests[i];
        printf(" -> %d", head);
    }

    // Go to last cylinder
    if (head != totalCylinders - 1) {
        seekTime += abs(head - (totalCylinders - 1));
        head = totalCylinders - 1;
        printf(" -> %d", head);
    }
```

```
        // Then move left servicing remaining requests
        for (int i = headIndex - 1; i >= 0; i--) {
            seekTime += abs(head - allRequests[i]);
            head = allRequests[i];
            printf(" -> %d", head);
        }
    }

    // SCAN to the left
    else {
        // Move from head left
        for (int i = headIndex - 1; i >= 0; i--) {
            seekTime += abs(head - allRequests[i]);
            head = allRequests[i];
            printf(" -> %d", head);
        }

        // Go to cylinder 0
        if (head != 0) {
            seekTime += abs(head - 0);
            head = 0;
            printf(" -> %d", head);
        }

        // Then move right servicing remaining requests
        for (int i = headIndex + 1; i < n + 1; i++) {
            seekTime += abs(head - allRequests[i]);
            head = allRequests[i];
            printf(" -> %d", head);
        }
    }

    printf("\n\nTotal Seek Time = %d cylinders\n", seekTime);
    printf("Average Seek Time = %.2f cylinders\n", (float)seekTime / n);

    return 0;
}
```

Outputs:


E:\Sarfraj\4th SEME
SCAN Disk Scheduling Algorithm by Sarfraj Alam
Enter total number of cylinders: 200
Enter current position of R/W head: 50
Enter number of pending requests: 6
Enter the request queue:
82
170
43
140
24
16
Enter direction (0 = left, 1 = right): 1
Head Movement Order:
50 -> 82 -> 140 -> 170 -> 199 -> 43 -> 24 -> 16
Total Seek Time = 332 cylinders
Average Seek Time = 55.33 cylinders
Process exited after 59.22 seconds with return value 0
Press any key to continue . . .

Lab no: 12 **Date: 2082/07/12**

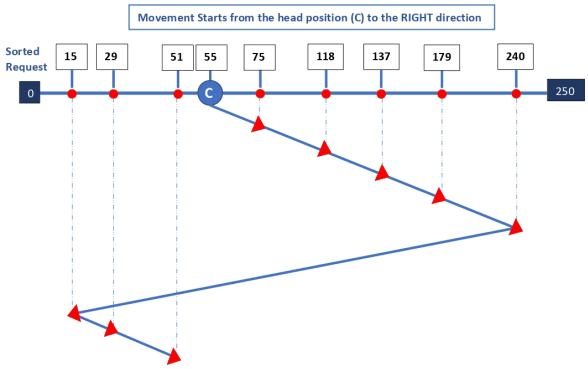
Title:Write a program to calculate the seek time for user input pending request, total number of cylinder and current position of I/O read/write head using LOOK disk scheduling algorithm.

Introduction

The LOOK disk scheduling algorithm is an efficient disk scheduling technique that improves on SCAN. Unlike SCAN, which always moves the head to the ends of the disk, LOOK only moves the head as far as the farthest request in the current direction. Once it reaches the farthest request, the head reverses direction and continues servicing remaining requests, similar to an elevator that only moves as far as the top or bottom passenger. This reduces unnecessary movement and decreases overall seek time.

How LOOK Works:

1. The disk arm begins moving in a chosen direction (left or right).
2. All pending requests in that direction are serviced as the head moves.
3. When the head reaches the farthest request in the current direction, it reverses direction.
4. The disk arm then services all remaining requests while moving in the opposite direction.
5. This process continues until all requests are serviced.



IDE: DEV C++
Language: C

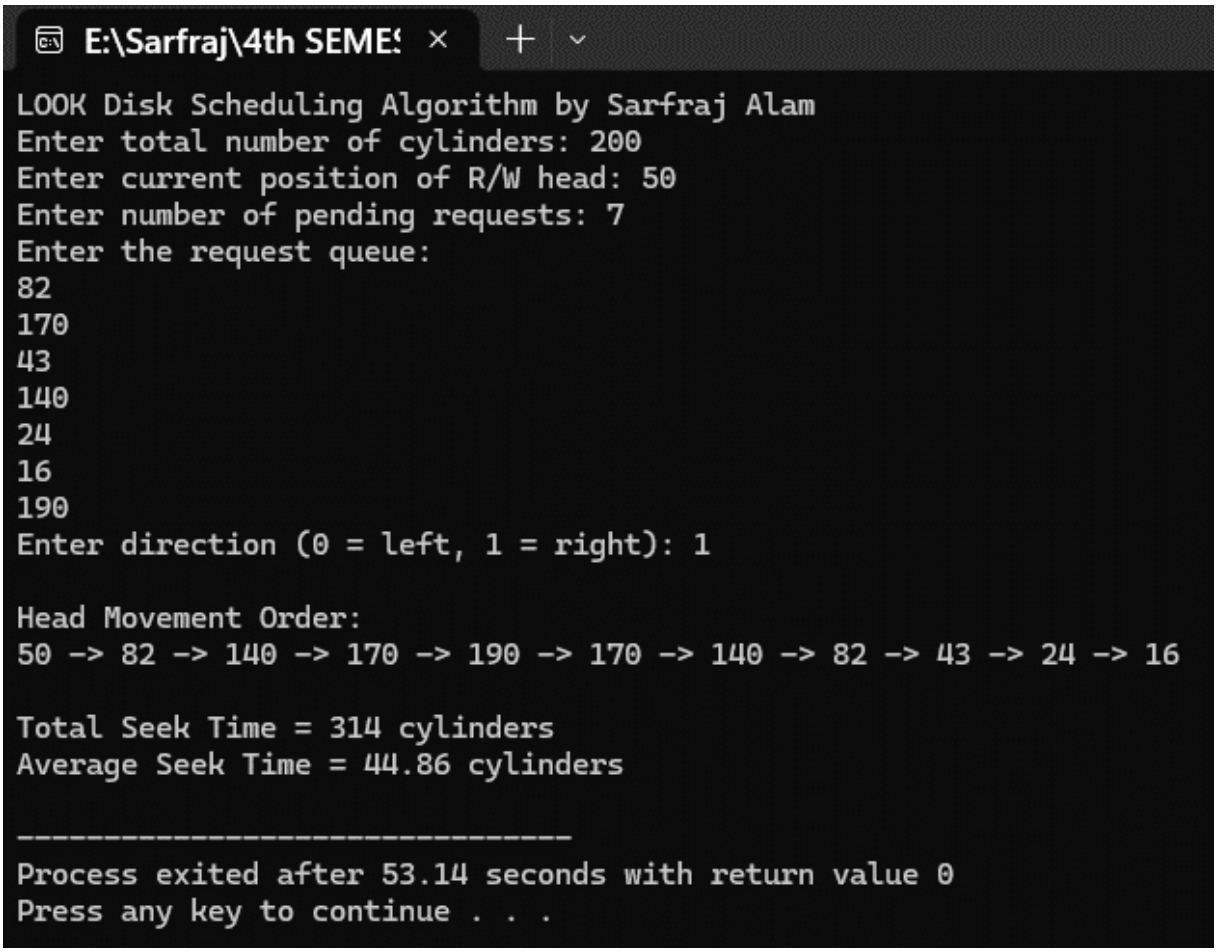
Source code:

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

int main() {
    int totalCylinders, head, n, direction;
    int seekTime = 0;

    printf("LOOK Disk Scheduling Algorithm by Sarfraj Alam\n");

    // Input total cylinders
    printf("Enter total number of cylinders: ");
    scanf("%d", &totalCylinders);

    // Input head position
    printf("Enter current position of R/W head: ");
    scanf("%d", &head);

    // Input size of request queue
    printf("Enter number of pending requests: ");
    scanf("%d", &n);

    int requests[n];
    printf("Enter the request queue:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
        if (requests[i] < 0 || requests[i] >= totalCylinders) {
            printf("Invalid request %d (out of range)\n", requests[i]);
            return 1;
        }
    }

    // Input LOOK direction
    printf("Enter direction (0 = left, 1 = right): ");
    scanf("%d", &direction);

    // Sort the request queue
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (requests[i] > requests[j]) {
                int temp = requests[i];
```

```
            requests[i] = requests[j];
            requests[j] = temp;
        }
    }
}

printf("\nHead Movement Order:\n%d", head);

// LOOK to the right first
if (direction == 1) {
    // Move from head to the farthest right request
    for (int i = 0; i < n; i++) {
        if (requests[i] >= head) {
            seekTime += abs(head - requests[i]);
            head = requests[i];
            printf(" -> %d", head);
        }
    }
    // Then move left to remaining requests
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] < head) {
            seekTime += abs(head - requests[i]);
            head = requests[i];
            printf(" -> %d", head);
        }
    }
}

// LOOK to the left first
else {
    // Move from head to the farthest left request
    for (int i = n - 1; i >= 0; i--) {
        if (requests[i] <= head) {
            seekTime += abs(head - requests[i]);
            head = requests[i];
            printf(" -> %d", head);
        }
    }
    // Then move right to remaining requests
    for (int i = 0; i < n; i++) {
```

```
            if (requests[i] > head) {
                seekTime += abs(head - requests[i]);
                head = requests[i];
                printf(" -> %d", head);
            }
        }
    }

    printf("\n\nTotal Seek Time = %d cylinders\n", seekTime);
    printf("Average Seek Time = %.2f cylinders\n", (float)seekTime / n);

    return 0;
}
```

Outputs:


E:\Sarfraj\4th SEME
LOOK Disk Scheduling Algorithm by Sarfraj Alam
Enter total number of cylinders: 200
Enter current position of R/W head: 50
Enter number of pending requests: 7
Enter the request queue:
82
170
43
140
24
16
190
Enter direction (0 = left, 1 = right): 1
Head Movement Order:
50 -> 82 -> 140 -> 170 -> 190 -> 170 -> 140 -> 82 -> 43 -> 24 -> 16
Total Seek Time = 314 cylinders
Average Seek Time = 44.86 cylinders
Process exited after 53.14 seconds with return value 0
Press any key to continue . . .

Lab no: 13 **Date: 2082/07/12**

Title:Write a program to test if the system is free from deadlock or not for the user input allocation, max and available matrix.

Introduction

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It ensures that a system can allocate resources to processes safely without leading to deadlock. The algorithm works like a bank that only lends money if it can guarantee that all clients can eventually be served.

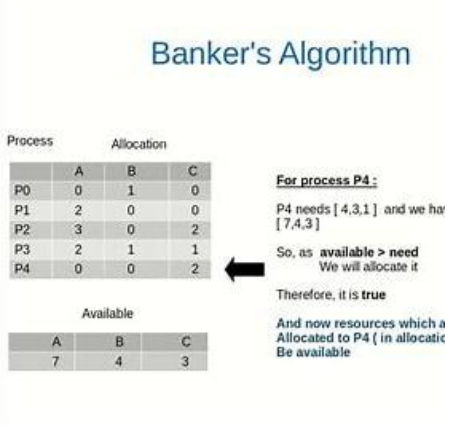
Key Terms

1. Available**:** The number of available instances of each resource type.
2. Max**:** The maximum demand of each process for each resource type.
3. Allocation: The number of resources currently allocated to each process.
4. Need: The remaining resources a process may still request to complete its task.

$$\textbf{Need} = \textbf{Max} - \textbf{Allocation}$$

How Banker's Algorithm Works

1. When a process requests resources, the system checks if the request can **be** safely granted**.**
2. The system pretends to allocate the requested resources.
3. It then checks if there is a safe sequence of process execution**:**
4. A sequence in which all processes can finish without waiting indefinitely.
5. If a safe sequence exists, the request is granted; otherwise, the process must wait.



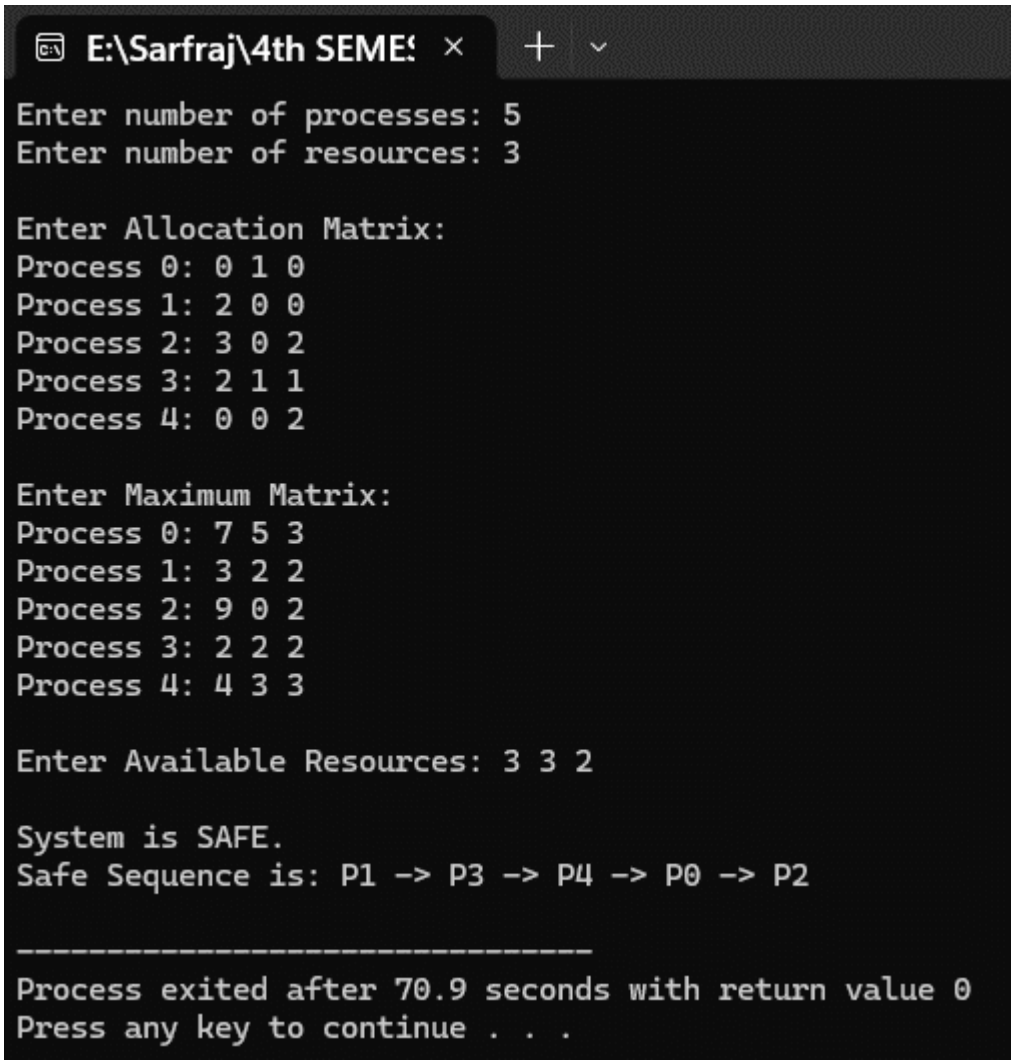
IDE: DEV C++
Language: C

Sourse code:

```
#include <stdio.h>

int main() {
   int n, m; // n = number of processes, m = number of resources

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

   printf("Enter number of resources: ");
   scanf("%d", &m);

   int allocation[n][m];
   int max[n][m];
   int available[m];
   int need[n][m];

   // Input Allocation Matrix
   printf("\nEnter Allocation Matrix:\n");
   for (int i = 0; i < n; i++) {
      printf("Process %d: ", i);
      for (int j = 0; j < m; j++)
         scanf("%d", &allocation[i][j]);
   }

   // Input Max Matrix
   printf("\nEnter Maximum Matrix:\n");
   for (int i = 0; i < n; i++) {
      printf("Process %d: ", i);
      for (int j = 0; j < m; j++)
         scanf("%d", &max[i][j]);
   }

   // Input Available Matrix
   printf("\nEnter Available Resources: ");
   for (int j = 0; j < m; j++)
      scanf("%d", &available[j]);

   // Calculate Need Matrix = Max - Allocation
   for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++)
         need[i][j] = max[i][j] - allocation[i][j];

   int finish[n];
   for (int i = 0; i < n; i++)
      finish[i] = 0; // initialize all processes as unfinished
```

```
    int safeSeq[n];
    int count = 0;

    // Banker's Algorithm
    while (count < n) {
        int found = 0;

        for (int p = 0; p < n; p++) {
            if (!finish[p]) {
                int canAllocate = 1;
                for (int r = 0; r < m; r++) {
                    if (need[p][r] > available[r]) {
                        canAllocate = 0;
                        break;
                    }
                }

                if (canAllocate) {
                    // Allocate resources
                    for (int r = 0; r < m; r++)
                        available[r] += allocation[p][r];

                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }

        if (!found) {
            printf("\nSystem is NOT SAFE! Deadlock may occur.\n");
            return 0;
        }
    }

    // Print safe sequence
    printf("\nSystem is SAFE.\nSafe Sequence is: ");
    for (int i = 0; i < n; i++) {
        printf("P%d", safeSeq[i]);
        if (i != n - 1) printf(" -> ");
    }
    printf("\n");

    return 0;
}
```

Output:


E:\Sarfraj\4th SEME
Enter number of processes: 5
Enter number of resources: 3
Enter Allocation Matrix:
Process 0: 0 1 0
Process 1: 2 0 0
Process 2: 3 0 2
Process 3: 2 1 1
Process 4: 0 0 2
Enter Maximum Matrix:
Process 0: 7 5 3
Process 1: 3 2 2
Process 2: 9 0 2
Process 3: 2 2 2
Process 4: 4 3 3
Enter Available Resources: 3 3 2
System is SAFE.
Safe Sequence is: P1 -> P3 -> P4 -> P0 -> P2

Process exited after 70.9 seconds with return value 0
Press any key to continue . . .

Lab no: 14 **Date: 2082/08/15**

Title:Prepare a lab report for basic Linux command.

Introduction

Linux is a powerful operating system that provides a command-line interface (CLI) for efficient system administration and file management. Basic Linux commands are essential tools that allow users to navigate the file system, manage files and directories, monitor system processes, and perform various administrative tasks. Understanding these fundamental commands is crucial for working effectively in a Linux environment.

Command Line Interface (CLI): The CLI provides direct access to the operating system through text-based commands. It offers more precision and power compared to graphical interfaces, making it preferred by system administrators and developers.

Basic Command Structure:

command [options] [arguments]

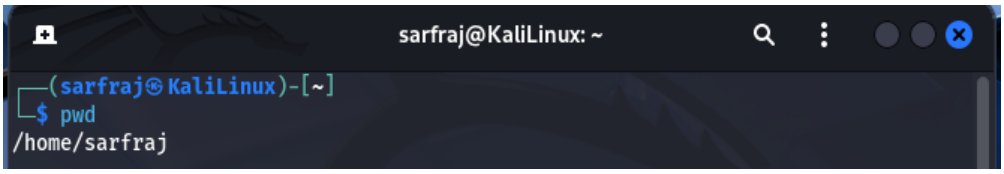
Terminal: Kali Linux Terminal

Operating System: Kali Linux

Some Basic Commands in Kali Linux

1. pwd (Print Working Directory)

Purpose: Displays the current directory path



2. cd (Change Directory)

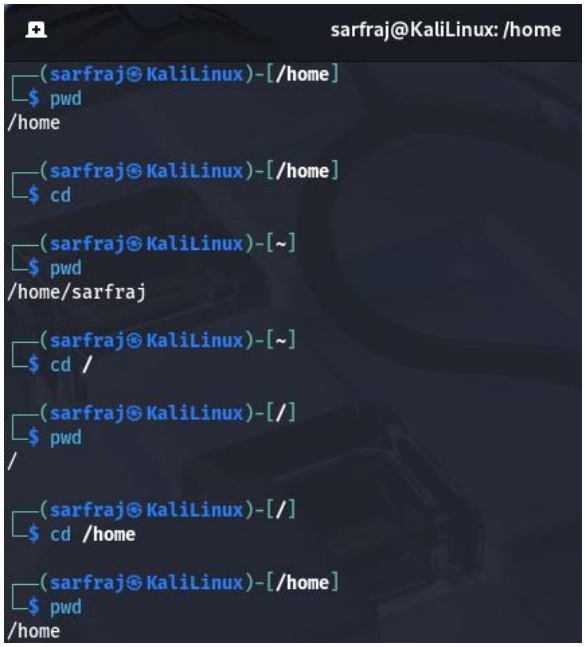
Purpose: Navigate between directories

cd /path/to/directory# Change to specific directory

cd .. # Move to parent directory

cd ~ # Move to home directory

cd - # Move to previous directory



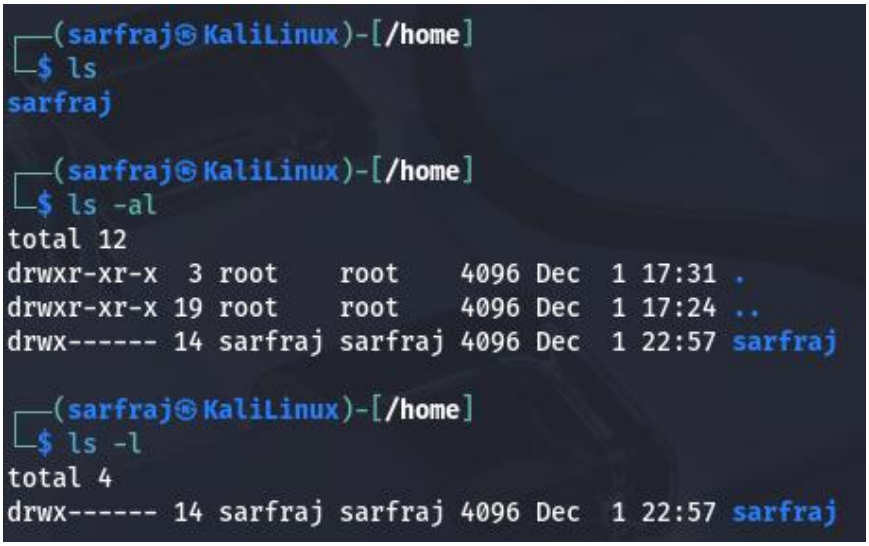
3. ls (List Directory Contents)

Purpose: Display files and directories

ls *# List current directory*

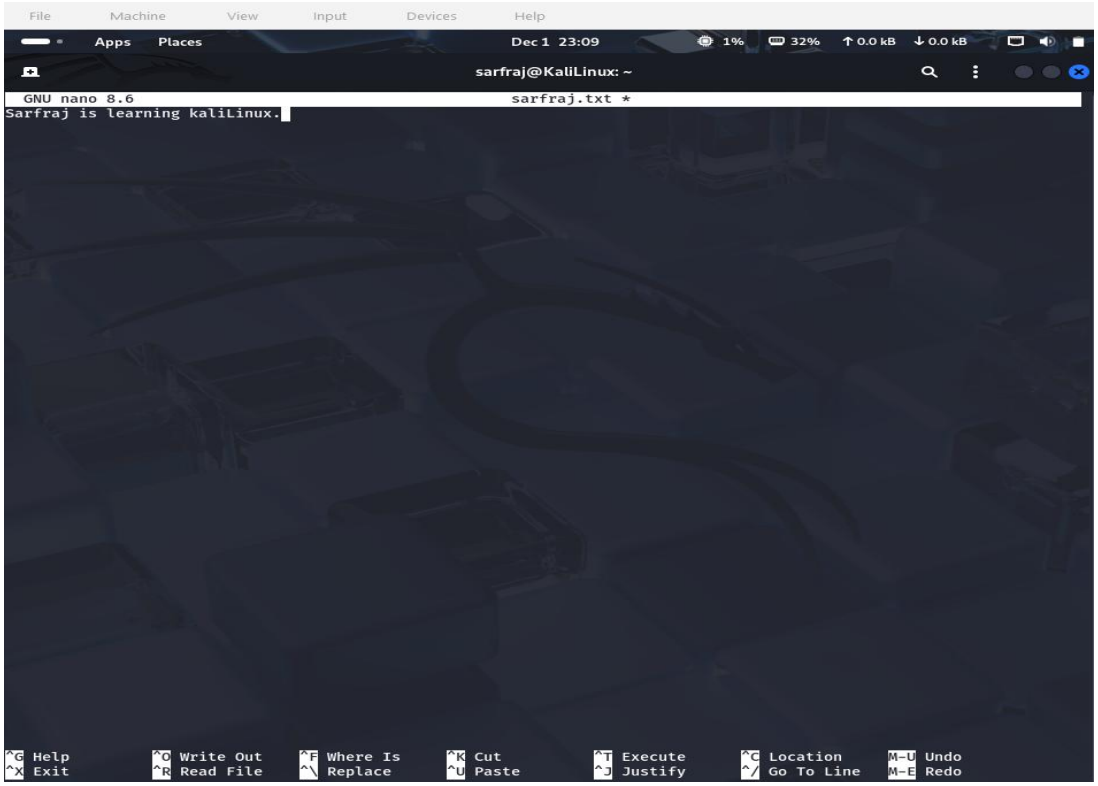
ls -l *# Long format listing*

ls -al *# Include hidden files*



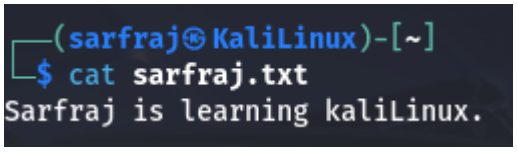
4. nano (Text Editor)

Purpose: Simple command-line text editor



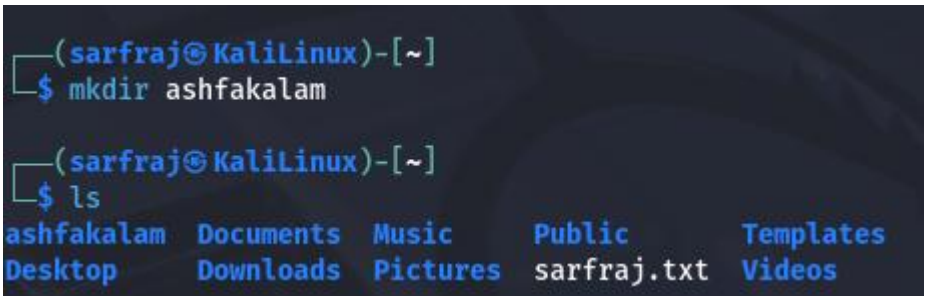
5. cat (Concatenate and Display)

Purpose: Display file contents or combine files



6.Mkdir (Make Directory)

Purpose: Create new directories

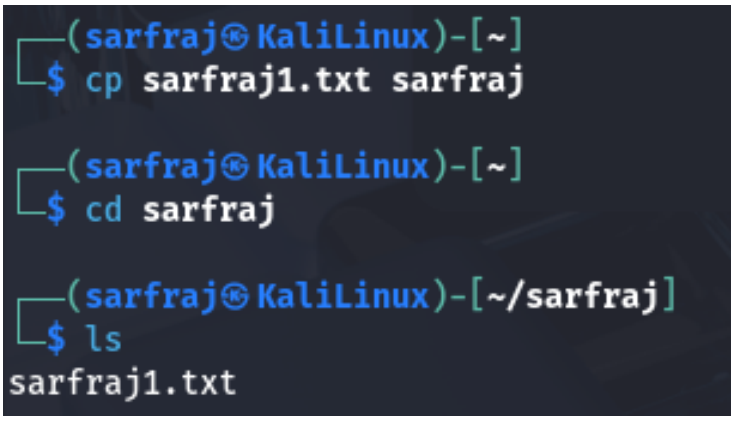


7. shred (Secure File Deletion)

Purpose: Securely delete files by overwriting data

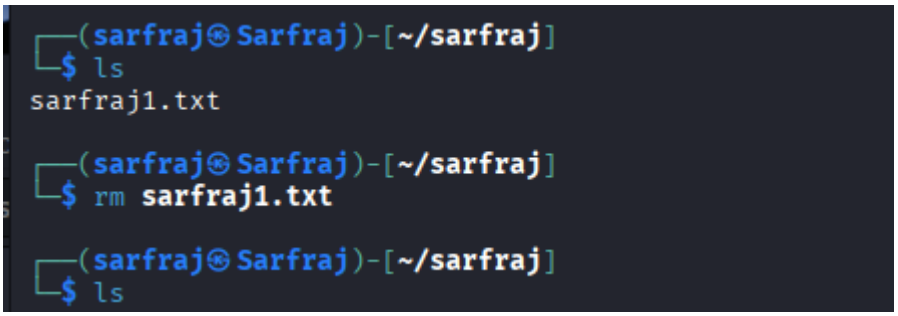
8. cp (Copy Files/Directories)

Purpose: Copy files and directories



9. rm (Remove Files/Directories)

Purpose: Delete files and directories



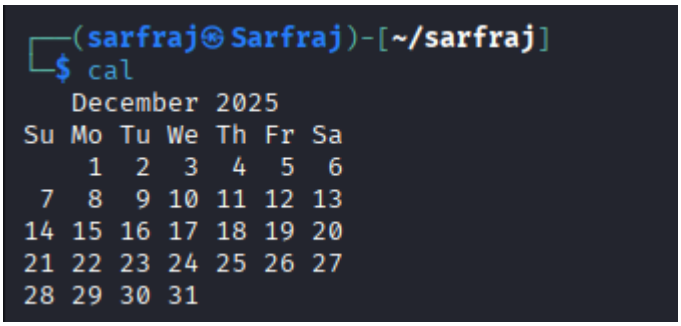
10. echo (Display Text)

Purpose: Display text or create simple files

11. cal (Calendar)

Purpose: Display calendar

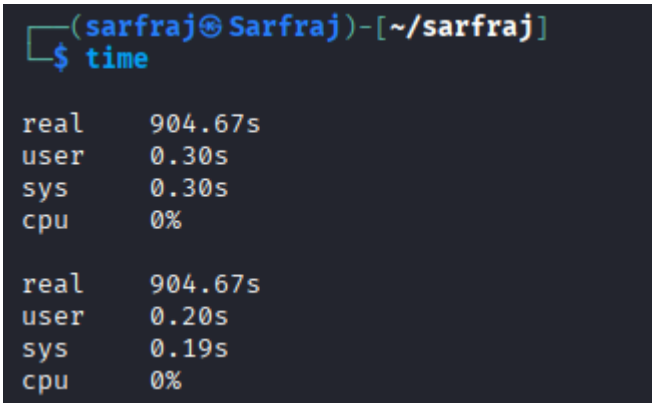
Syntax: cal [OPTION] [MONTH] [YEAR]



12. time (Command Execution Time)

Purpose: Measure command execution time

Syntax: time [OPTION] COMMAND



13. ifconfig (Network Interface Configuration)

Purpose: Configure and display network interfaces

Syntax: ifconfig [INTERFACE] [OPTIONS]

```
┌──(sarfraj㉿Sarfraj)-[~/sarfraj]
└─$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fd17:625c:f037:2:57dc:59eb:a6bd:cf1a  prefixlen 64  scopeid 0×0
<global>
        inet6 fd17:625c:f037:2:a00:27ff:fe62:af5c  prefixlen 64  scopeid 0×0<
global>
        inet6 fe80::a00:27ff:fe62:af5c  prefixlen 64  scopeid 0×20<link>
        ether 08:00:27:62:af:5c  txqueuelen 1000  (Ethernet)
        RX packets 208  bytes 232145 (226.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 252  bytes 20488 (20.0 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0×10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 8  bytes 480 (480.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 480 (480.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

14. ip address (Modern Network Configuration)

Purpose: Modern replacement for ifconfig

Syntax: ip [OPTIONS] OBJECT COMMAND

Options:

- addr : Address management
- link : Network device management
- route : Routing table management
- -4 : IPv4 only
- -6 : IPv6 only

```
┌──(sarfraj㉿Sarfraj)-[~/sarfraj]
└─$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000
    link/ether 08:00:27:62:af:5c brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute eth0
       valid_lft 83430sec preferred_lft 83430sec
    inet6 fd17:625c:f037:2:57dc:59eb:a6bd:cf1a/64 scope global temporary dyna
mic
       valid_lft 86370sec preferred_lft 14370sec
    inet6 fd17:625c:f037:2:a00:27ff:fe62:af5c/64 scope global dynamic mngtmpa
ddr noprefixroute
       valid_lft 86370sec preferred_lft 14370sec
    inet6 fe80::a00:27ff:fe62:af5c/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

15. cmp (Compare Files)

Purpose: Compare two files byte by byte

```
┌──(sarfraj㉿Sarfraj)-[~]
└─$ cmp sarfraj1.txt b.txt
sarfraj1.txt b.txt differ: byte 6, line 1
```

16. sort (Sort File Contents)

Purpose: Sort lines in text files

```
┌──(sarfraj㉿Sarfraj)-[~]
└─$ sort sarfraj.txt

this is test.
```

17. ping (Network Connectivity Test)

Purpose: Test network connectivity to remote hosts

```
┌──(sarfraj㉿Sarfraj)-[~]
└─$ ping -c4 google.com
PING google.com (142.250.182.238) 56(84) bytes of data.
64 bytes from tzdelb-bd-in-f14.1e100.net (142.250.182.238): icmp_seq=1 ttl=25
5 time=20.3 ms
64 bytes from tzdelb-bd-in-f14.1e100.net (142.250.182.238): icmp_seq=2 ttl=25
5 time=21.2 ms
64 bytes from tzdelb-bd-in-f14.1e100.net (142.250.182.238): icmp_seq=3 ttl=25
5 time=26.7 ms
64 bytes from tzdelb-bd-in-f14.1e100.net (142.250.182.238): icmp_seq=4 ttl=25
5 time=25.2 ms

── google.com ping statistics ──
4 packets transmitted, 4 received, 0% packet loss, time 7045ms
rtt min/avg/max/mdev = 20.306/23.345/26.720/2.675 ms
```

18. traceroute (Network Route Tracing)

Purpose: Trace network route to destination

```
┌──(sarfraj㉿Sarfraj)-[~]
└─$ traceroute google.com
traceroute to google.com (142.250.193.14), 30 hops max, 60 byte packets
 1  10.0.2.2 (10.0.2.2)  16.617 ms  16.072 ms  15.647 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
```

19. sudo(superuserdo) and apt (System Administration)

Purpose: Execute commands as root and manage packages

```
┌──(sarfraj㉿Sarfraj)-[~]
└─$ sudo apt update
[sudo] password for sarfraj:
Get:1 http://mirror.kku.ac.th/kali kali-rolling InRelease [34.0 kB]
Get:2 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Packages [21.0 MB]
Ign:2 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Packages
Ign:3 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Contents (deb)
Ign:2 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Packages
Ign:2 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Packages
Ign:3 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Contents (deb)
Ign:2 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Packages
Ign:3 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Contents (deb)
Get:2 http://kali.download/kali kali-rolling/main amd64 Packages [21.0 MB]
Ign:3 http://mirror.kku.ac.th/kali kali-rolling/main amd64 Contents (deb)
Get:3 http://kali.download/kali kali-rolling/main amd64 Contents (deb) [52.6
MB]
Fetched 73.1 MB in 1min 26s (848 kB/s)
1306 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

20. grep (Text Search)

Purpose: Search for patterns in files

```
┌──(sarfraj㉿Sarfraj)-[~]
└─$ ip address | grep eth0:
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000

┌──(sarfraj㉿Sarfraj)-[~]
└─$ ip addres
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group def
ault qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP g
roup default qlen 1000
    link/ether 08:00:27:62:af:5c brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute eth0
       valid_lft 82314sec preferred_lft 82314sec
    inet6 fd17:625c:f037:2:57dc:59eb:a6bd:cf1a/64 scope global temporary dyna
mic
       valid_lft 86325sec preferred_lft 14325sec
    inet6 fd17:625c:f037:2:a00:27ff:fe62:af5c/64 scope global dynamic mngtmpa
ddr noprefixroute
       valid_lft 86325sec preferred_lft 14325sec
    inet6 fe80::a00:27ff:fe62:af5c/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

Conclusion:

These 20 basic Linux commands form the foundation of command-line operations in Kali Linux. They cover:

- **File Management**: pwd, cd, ls, mkdir, cp, rm
- **Text Operations**: nano, cat, echo, grep, sort, cmp
- **System Information**: cal, time, ifconfig, ip address
- **Network Operations**: ping, traceroute
- **Security**: shred
- **System Administration**: sudo, apt

Lab no: 15 **Date: 2082/08/15**

Title:Prepare a lab report to create and terminate process.

Introduction

Process management is a fundamental concept in operating systems. A process is a running instance of a program that contains the program code and its current activity. Understanding how to create and terminate processes is crucial for system administration, programming, and managing system resources effectively.

Process Concepts:

- **Process:** An executing program with its own memory space
- **Process ID (PID):** Unique identifier assigned to each process
- **Parent Process:** Process that creates another process
- **Child Process:** Process created by another process
- **Process State:** Current status of a process (running, sleeping, stopped, zombie)

Process Termination Methods:

1. **Natural Termination:** Process completes execution
2. **Signal Termination:** Using kill signals
3. **Force Termination:** Forceful process killing
4. **Parent Termination:** When parent process ends

Terminal: Kali Linux Terminal

Operating System: Kali Linux

Source code:

1. **Process creation**

sleep 300 & # Create a sleep process for 300 seconds

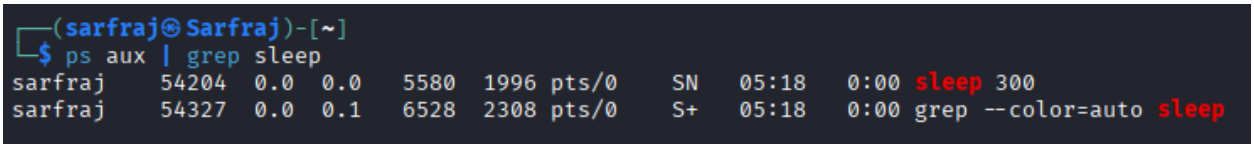
[1] 54204 # Process id

2. **Viewing Processes**

ps aux | grep sleep # Find specific process

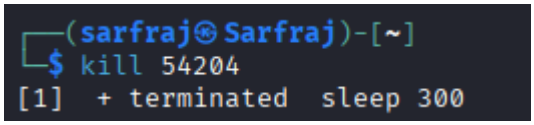
sarfraj 54204 0.0 0.0 5580 1996 pts/0 SN 05:18 0.00 sleep 300

sarfraj 54327 0.0 0.1 6528 2308 pts/0 S+ 05:18 0.00 grep --color=auto sleep

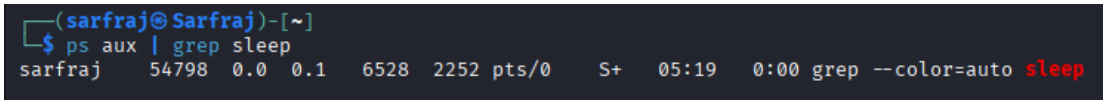


3. **Terminating the process**

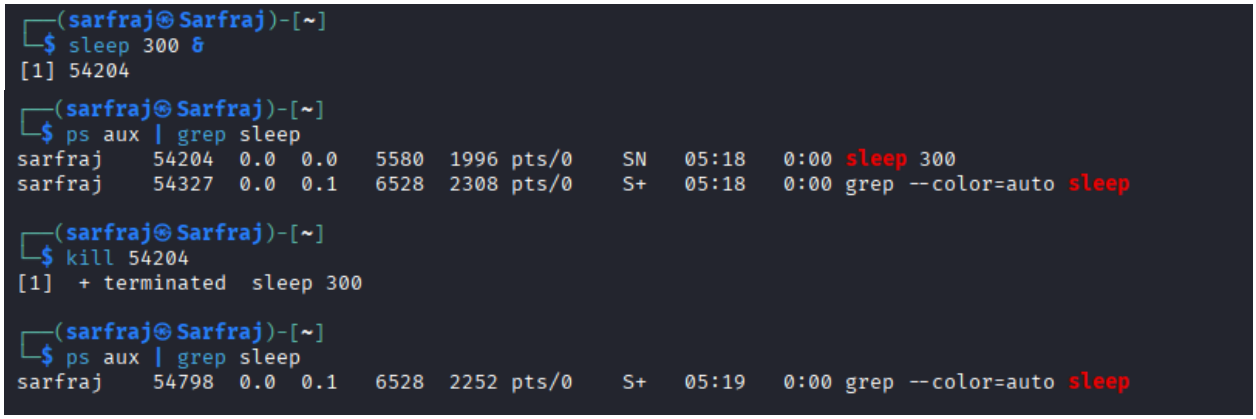
kill 54204 # Stop process by ID number



4 **Checking if the process is terminated or not**



Output:



Process Information:

- **PID**: Process ID number (unique for each process)
- **Job**: Background job number [1], [2], etc.
- **Status**: Running, Terminated, Stopped

Conclusion:

Successfully created and terminated processes in Kali Linux:

- Created sleep process with sleep 300 &
- Found process using ps aux | grep sleep
- Terminated process with kill 30410
- Verified termination with ps aux | grep sleep