LAB 1

Objective: To Show The Implementation Of fork( )

Theory

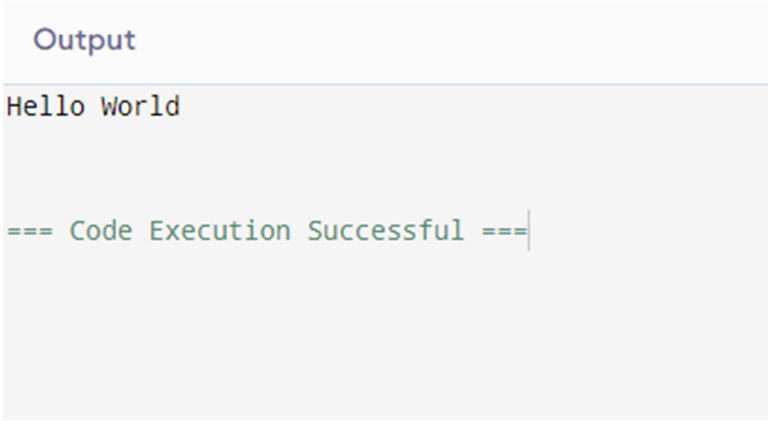
fork() is a system call in Unix/Linux used to create a new process by duplicating the current process. The new process is called the child process, while the original is called the parent process. After a successful fork(), two processes run concurrently: the parent and the child. The child process uses the same pc(program counter), same CPU registers, and same open files which use in the parent process. It takes no parameters and returns an integer value. Below are different values returned by **fork().**

- **Negative Value**: The creation of a child process was unsuccessful.
- **Zero**: Returned to the newly created child process.
- **Positive value**: Returned to parent or caller. The value contains the process ID of the newly created child process.

Program

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
fork();
printf("Hello World\n");
return 0;
 }
```

Output



Conclusion

The fork() system call successfully creates a new process, demonstrating the parent-child relationship in a Unix/Linux environment. The parent process can identify the child through its PID, while the child process can identify its parent through the parent PID. This practical implementation highlights the concurrent nature of process execution in operating systems

LAB 2

Objective: To show the implementation of first come first served (fcfs) algorithm

Theory

The **First Come, First Served (FCFS)** scheduling algorithm is one of the simplest CPU scheduling techniques. Processes are executed in the order they arrive in the ready queue. It is a non-preemptive algorithm, meaning once a process starts execution, it runs to completion without interruption.

Program

```
#include<stdio.h>

int main() {
  int n, bt[20], wt[20], tat[20], avwt = 0, avtat = 0, i, j;

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

  printf("\nEnter
  Process Burst Time\n");
  for (i = 0; i < n; i++)
  { printf("P[%d]: ", i
  + 1); scanf("%d",
  &bt[i]);
  }

  wt[0] = 0;
  for (i = 1;
  i < n; i++)
  { wt[i] = 0;
  for (j = 0;
  j < i; j++)
  wt[i] +=
  bt[j];
  } printf("\nProcess\t\tBurst Time\t
  Waiting Time\t Turnaround Time"); for (i =
  0; i < n; i++) { tat[i] = bt[i] + wt[i];
  avwt += wt[i]; avtat += tat[i];
```

```
  printf("\nP[%d]\t\t%d\t\t%d\t\t%d", i + 1,
  bt[i], wt[i], tat[i]);
  }
  avwt /= i; avtat /= i;
  printf("\n\nAverage Waiting
  Time: %d", avwt);
  printf("\nAverage Turnaround
  Time: %d", avtat);

return 0; }
```

Output

```
Enter total number of processes: 3

Enter Process Burst Time
P[1]: 1
P[2]: 3
P[3]: 4

Process     Burst Time   Waiting Time    Turnaround Time
P[1]        1       0       1
P[2]        3       1       4
P[3]        4       4       8

Average Waiting Time: 1
Average Turnaround Time: 4

=== Code Execution Successful ===
```

Conclusion

We implemented the First Come First Serve (FCFS) algorithm and observed the output. In conclusion, the provided C program implements the First Come First Served (FCFS) scheduling algorithm to simulate the execution of processes in an operating system. The program prompts the user to enter the total number of processes and their respective burst times. It then calculates the waiting time and turnaround time for each process based on the FCFS scheduling policy.

LAB 3

Objective: To Show The Implementation Of Shortest Job First (sjf) algorithm.

Theory

The **Shortest Job First (SJF)** scheduling algorithm selects the process with the shortest burst time from the ready queue for execution. This minimizes the average waiting time and turnaround time. SJF can be:

1. **Non-Preemptive**: Once a process starts execution, it runs to completion.
2. **Preemptive (Shortest Remaining Time First - SRTF)**: A running process can be preempted if a new process arrives with a shorter burst time.

Program

```
#include<stdio.h>

int main() { int bt[20], p[20], wt[20],
  tat[20], i, j, n, total = 0, pos, temp;
  float avg_wt, avg_tat;

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

  printf("\nEnter Burst Time
  for Processes:\n"); for (i =
  0; i < n; i++)
  { printf("p%d:", i + 1);
  scanf("%d", &bt[i]); p[i] = i
  + 1;
  }
    for (i = 0; i
    < n; i++)
    { pos = i; for
    (j = i + 1; j
    < n; j++) { if
    (bt[j] <
    bt[pos])
        pos
  = j; }
  temp =
  bt[i];
  bt[i] =
  bt[pos];
  bt[pos] =
  temp;
  temp =
```

```
  p[i]; p[i]
  = p[pos];
  p[pos] =
  temp; }

  wt[0] = 0;
  for (i = 1;
  i < n; i++)
  { wt[i] = 0;
  for (j = 0;
  j < i; j++)
      wt[i] += bt[j];
    total
  +=
  wt[i]; }

  avg_wt =
  (float)total / n;
  total = 0;

  printf("\nProcess\tBurst Time\tWaiting
  Time\tTurnaround Time\n"); for (i = 0; i < n;
  i++) { tat[i] = bt[i] + wt[i]; total +=
  tat[i]; printf("\np%d\t\t%d\t\t%d\t\t%d",
  p[i], bt[i], wt[i], tat[i]);
  }

  avg_tat = (float)total / n;
  printf("\n\nAverage Waiting Time
  = %f", avg_wt); printf("\nAverage
  Turnaround Time = %f\n", avg_tat);

  return 0;
}
```

Output

```
Enter number of processes:3

Enter Burst Time for Processes:
p1:2
p2:3
p3:2

Process Burst Time  Waiting Time    Turnaround Time

p1      2       0       2
p3      2       2       4
p2      3       4       7

Average Waiting Time = 2.000000
Average Turnaround Time = 4.333333


=== Code Execution Successful ===
```

Conclusion

The Shortest Job First (SJF) scheduling algorithm efficiently minimizes the average waiting time and turnaround time by prioritizing processes with shorter burst times. However, it suffers from the starvation problem, where long processes might be delayed indefinitely if shorter processes keep arriving. It is ideal in scenarios where burst times are predictable.

LAB 4

Objective: To Show The Implementation Of Round Robin (Rr) Algorithm.

Theory

The **Round Robin (RR)** scheduling algorithm is a preemptive CPU scheduling technique where each process is assigned a fixed time slice, called a **time quantum**. Processes are executed cyclically in the ready queue until they complete.

Program

```
#include<stdio.h>

int main() {
  int i, limit, total = 0, x, counter = 0, time_quantum;
  int wait_time = 0, turnaround_time = 0, arrival_time[10],
  burst_time[10], temp[10]; float average_wait_time,
  average_turnaround_time;

  printf("Enter Total Number
  of Processes:"); scanf("%d",
  &limit); x = limit;

  for(i = 0; i < limit; i++)
    { printf("\nEnter Details of
    Process[%d]\n", i + 1);
    printf("Arrival Time:\t");
    scanf("%d", &arrival_time[i]);
    printf("Burst Time:\t");
    scanf("%d", &burst_time[i]);
    temp[i] = burst_time[i];
  }

  printf("\nEnter Time Quantum:\t"); scanf("%d",

  &time_quantum); printf("\nProcess ID\tBurst

  Time\t Turnaround Time\tWaiting Time\n");

  for(total = 0, i = 0; x != 0;) {
    if(temp[i] <= time_quantum &&
      temp[i] > 0) { total +=
      temp[i]; temp[i] = 0;
      counter = 1;
    } else
    if(temp[i]
    > 0) {
```

```
      temp[i] -=
      time_quantum;
      total +=
      time_quantum;
    }

    if(temp[i] == 0 && counter == 1) { x--;
      printf("\nProcess[%d]\t%d\t\t%d\t\t\t%d", i + 1, burst_time[i],
      total - arrival_time[i], total -
arrival_time[i] - burst_time[i]);
      wait_time += total -
      arrival_time[i] - burst_time[i];
      turnaround_time += total -
      arrival_time[i]; counter = 0;
    }

    if(i == limit - 1)
      i = 0;
    else if(arrival_time[i +
      1] <= total) i++;
    else
      i
   =
  0; }

  average_wait_time = (float)wait_time / limit;
  average_turnaround_time =
  (float)turnaround_time / limit;

  printf("\n\nAverage Waiting Time:\t%f",
  average_wait_time); printf("\nAverage Turnaround
  Time:\t%f\n", average_turnaround_time);

  return 0;
}
```

Output

```
Enter Total Number of Processes:3

Enter Details of Process[1]
Arrival Time:   1
Burst Time: 3

Enter Details of Process[2]
Arrival Time:   2
Burst Time: 4

Enter Details of Process[3]
Arrival Time:   3
Burst Time: 2

Enter Time Quantum: 1

Process ID  Burst Time   Turnaround Time    Waiting Time

Process[1]  3       4           1
Process[3]  2       4           2
Process[2]  4       7           3

Average Waiting Time:   2.000000
Average Turnaround Time:    5.000000


=== Code Execution Successful ===
```

Conclusion

The Round Robin (RR) scheduling algorithm ensures fairness by allotting equal time slices (quantum) to all processes. It prevents long processes from monopolizing the CPU and is particularly suitable for time-sharing systems

LAB 5

Objective: To show the implementation of FIFO page replacement algorithm

Theory

The **FIFO Page Replacement Algorithm** is one of the simplest page replacement strategies used in operating systems. It replaces the oldest page in memory i.e the one that has been in memory the longest, when a new page needs to be loaded, and there is no available free space in the page frame.

Program

```
#include <stdio.h>

#include <stdlib.h>

int pagefault(int a[], int frame[], int n, int no) {

    int i, j, avail, count = 0, k;

    for (i = 0; i < no; i++) {

        frame[i] = -1;

    }

    j = 0;

    for (i = 0; i < n; i++) {

        avail = 0;

        for (k = 0; k < no; k++) {

            if (frame[k] == a[i]) {

                avail = 1;

                break;

            }

        }

        if (avail == 0) {

            frame[j] = a[i];

            j = (j + 1) % no; // Move to the next frame in a
circular manner
```

```
            count++;          // Increment the page fault counter
        }
    }
    return count;
}
int main() {
    int n, i, *a, *frame, no, fault;
    printf("\nENTER THE NUMBER OF PAGES:\n");
    scanf("%d", &n);
    a = (int *)malloc(n * sizeof(int));
    printf("ENTER THE PAGE NUMBERS:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    printf("ENTER THE NUMBER OF FRAMES: ");
    scanf("%d", &no);
    frame = (int *)malloc(no * sizeof(int));
    fault = pagefault(a, frame, n, no);
    printf("Page Fault Count: %d\n", fault);
    free(a);
    free(frame);
    return 0;
}
```

Output

```
ENTER THE NUMBER OF PAGES:
3
ENTER THE PAGE NUMBERS:
2
3
4
ENTER THE NUMBER OF FRAMES: 3
Page Fault Count: 3


=== Code Execution Successful ===
```

Conclusion

In conclusion, the FIFO (First-In-First-Out) page replacement algorithm is a straightforward approach for managing memory in operating systems. It operates on the principle of evicting the oldest page from memory when a new page needs to be loaded. Despite its simplicity, FIFO can suffer from the "Belady's Anomaly," where increasing the number of frames may actually lead to more page faults.

LAB 6

Objective: To show the implementation of LRU page replacement algorithm.

Theory

The **Least Recently Used (LRU)** page replacement algorithm is a commonly used strategy that replaces the page that has not been used for the longest time. It is based on the principle of temporal locality, assuming that recently used pages are more likely to be accessed again soon.

Program

```
#include <stdio.h>
int n, ref[100], fs, frame[100], count = 0;
void input();
void show();
void cal();
int main() {
    printf("**********     LRU     Page     Replacement     Algorithm
**************\n");
    input();
    cal();
    show();
    return 0;
}
void input() {
    int i;
    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &n);
      printf("Enter the reference string:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &ref[i]);
    }

    printf("Enter the frame size: ");
```

```
    scanf("%d", &fs);
}
void cal() {
    int i, j, k = 0, c1, c2, r, temp[100];
        for (i = 0; i < fs; i++) {
        frame[i] = -1;
    }
    for (i = 0; i < n; i++) {
        c1 = 0;
        for (j = 0; j < fs; j++) {
            if (ref[i] == frame[j]) {
                c1 = 1; // Page hit
                break;
            }
        }
        if (c1 == 0) {
            count++;
            if (k < fs) {
                frame[k] = ref[i];
                k++;
            } else {
                for (r = 0; r < fs; r++) {
                    c2 = 0;
                    for (j = i - 1; j >= 0; j--) {
                        if (frame[r] != ref[j]) {
                            c2++;
                        } else {
                            break;
                        }
                    }
                    temp[r] = c2;
                }
```

```
                int max_index = 0;
                for (r = 1; r < fs; r++) {
                    if (temp[r] > temp[max_index]) {
                        max_index = r;
                    }
                }
                frame[max_index] = ref[i];
            }
        }
        printf("Step %d: ", i + 1);
        for (j = 0; j < fs; j++) {
            if (frame[j] != -1) {
                printf("%d ", frame[j]);
            } else {
                printf("- ");
            }
        }
        printf("\n");
    }
}
void show() {
    printf("Total Page Faults: %d\n", count);
}
```

Output

```
********** LRU Page Replacement Algorithm **************
Enter the number of pages in the reference string: 3
Enter the reference string:
1
2
3
Enter the frame size: 4
Step 1: 1 - - -
Step 2: 1 2 - -
Step 3: 1 2 3 -
Total Page Faults: 3
```

Conclusion

The LRU Page Replacement Algorithm uses the principle of temporal locality to replace the page that has not been used for the longest time. It minimizes page faults compared to FIFO in most cases but requires additional bookkeeping to track usage history. This makes it more efficient than FIFO but also more complex in terms of implementation.

LAB 7

Objective: To show the implementation of FCFS disk scheduling algorithm.

Theory

Disk scheduling is a technique used by operating systems to manage the order in which disk I/O requests are processed. The First-Come, First-Served (FCFS) algorithm processes requests in the order they arrive in the queue. This algorithm is simple and fair but may not always be the most efficient.

Program

#include <stdio.h>

```
#include <stdlib.h>
int main() {
    int queue[100], n, head, i, j, seek = 0, diff;
    float avg;
    printf("*** FCFS Disk Scheduling Algorithm ***\n");
    printf("Enter the size of the Queue: ");
    scanf("%d", &n);
    if (n <= 0 || n > 100) {
        printf("Invalid queue size. Please enter a number between 1
and 100.\n");
        return 1;
    }
    printf("Enter the Queue: \n");
    for (i = 0; i < n; i++) {
        scanf("%d", &queue[i]);
    }
    printf("Enter the initial head position: ");
    scanf("%d", &head);
    printf("\nProcessing Disk Queue:\n");
    for (i = 0; i < n; i++) {
        diff = abs(queue[i] - head);
        seek += diff;
```

```
        printf("Move from %d to %d with Seek %d\n", head, queue[i],
diff);
        head = queue[i];                }
    avg = (float)seek / n;
    printf("\nTotal Seek Time: %d\n", seek);
    printf("Average Seek Time: %.2f\n", avg);


    return 0;
}
```

Output

```
*** FCFS Disk Scheduling Algorithm ***
Enter the size of the Queue: 3
Enter the Queue:
1
2
3
Enter the initial head position: 2

Processing Disk Queue:
Move from 2 to 1 with Seek 1
Move from 1 to 2 with Seek 1
Move from 2 to 3 with Seek 1

Total Seek Time: 3
Average Seek Time: 1.00


=== Code Execution Successful ===
```

Conclusion

The FCFS Disk Scheduling Algorithm is simple and fair, as it processes requests in the order they arrive. However, it may lead to inefficiencies (high seek time) if requests are not in an optimal order. Alternative algorithms like SCAN or LOOK can provide better performance in terms of seek time.

LAB 8

Objective: To show implementation of banker's algorithm

Theory

The Banker's Algorithm, proposed by Edsger Dijkstra, is used in operating systems to manage resource allocation and ensure system safety. It determines whether a system is in a safe state by simulating resource allocation for processes.

Program

```
#include <stdio.h>

void calculateNeed(int need[][10], int max[][10], int alloc[][10],
int np, int nr) {

    for (int i = 0; i < np; i++)

        for (int j = 0; j < nr; j++)

            need[i][j] = max[i][j] - alloc[i][j];

}

int isSafe(int processes[], int avail[], int max[][10], int
alloc[][10], int np, int nr) {

    int need[10][10];

    calculateNeed(need, max, alloc, np, nr);

    int finish[10], safeSeq[10], work[10];

    for (int i = 0; i < np; i++) finish[i] = 0;

    for (int i = 0; i < nr; i++) work[i] = avail[i];

    int count = 0;

    while (count < np) {

        int found = 0;

        for (int p = 0; p < np; p++) {

            if (finish[p] == 0) {

                int j;

                for (j = 0; j < nr; j++)

                    if (need[p][j] > work[j])

                        break;

                if (j == nr) {
```

```
                    for (int k = 0; k < nr; k++)
                        work[k] += alloc[p][k];
                    safeSeq[count++] = p;
                    finish[p] = 1;
                    found = 1;
                }
            }
        }
        if (found == 0) {
            printf("System is not in a safe state\n");
            return 0;
        }
    }
    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < np; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");
    return 1;
}
int main() {
    int np, nr;
    printf("Enter number of processes and number of resources: ");
    scanf("%d %d", &np, &nr);
    int processes[np], avail[nr], max[np][10], alloc[np][10];
    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < np; i++)
        for (int j = 0; j < nr; j++)
            scanf("%d", &alloc[i][j]);
    printf("Enter the maximum demand matrix:\n");
    for (int i = 0; i < np; i++)
        for (int j = 0; j < nr; j++)
            scanf("%d", &max[i][j]);
```

```
    printf("Enter the available resources:\n");
    for (int i = 0; i < nr; i++)
        scanf("%d", &avail[i]);
    isSafe(processes, avail, max, alloc, np, nr);
    return 0;
}
```

Output

```
Enter number of processes and number of resources: 2
2
Enter the allocation matrix:
2
2
2
2
Enter the maximum demand matrix:
2
2
2
22
Enter the available resources:

2
2
System is in a safe state.
Safe sequence is: 0 1


=== Code Execution Successful ===2
```

Conclusion

The Banker's Algorithm is an efficient way to avoid deadlocks by ensuring that the system remains in a safe state. This implementation demonstrates how the algorithm checks system safety by simulating resource allocation and release, preventing unsafe states.

LAB 9

Objective: To implement program for process creation

Theory

Process creation is fundamental in operating systems, allowing a parent process to create child processes. In UNIX-like systems, the fork() system call is used for this purpose.

- **Parent Process:** The original process that calls fork().
- **Child Process:** A duplicate of the parent process, created by fork().

Program

```
#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

    pid_t pid;

    pid = fork();

    if (pid < 0) {

        perror("Fork failed");

        return 1;

    } else if (pid == 0) {

        printf("Hello from the child process! PID: %d, Parent
PID: %d\n", getpid(), getppid());

    } else {

        printf("Hello from the parent process! PID: %d, Child
PID: %d\n", getpid(), pid);

    }

    return 0;

}
```

Output

```
Hello from the parent process! PID: 17142, Child PID: 17145
Hello from the child process! PID: 17145, Parent PID: 17142


=== Code Execution Successful ===
```

Conclusion

The program demonstrates the creation of a child process using the `fork()` system call. It highlights how the child process inherits the parent's execution context while being assigned a unique process ID.