PROBLEM STATEMENT 3:

Write a ‘C’ program using ‘gcc’ compiler to extract system information, Model name, Cache Size, Number of CPU cores, CPU clock speed, Total Memory, Free Memory, OS Name, OS Version.

OBJECTIVE:

To extract system information such as the model name, cache size, number of CPU cores, CPU clock speed, total memory, free memory, OS name, and OS version. This information can be useful for debugging, monitoring system performance, and identifying hardware and software configurations.

C SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  printf("Extracting system information using gcc compiler.\n\n");
  printf("Model Name\n");
  system("cat /proc/cpuinfo | grep -m 1 'model name'");
  printf("Cache Size\n");
  system("cat /proc/cpuinfo | grep -m 1 'cache size'");
  printf("Number of CPU CORES\n");
  system("cat /proc/cpuinfo | grep -m 1 'cpu cores'");
  printf("CPU Clock Speed\n");
  system("cat /proc/cpuinfo | grep -m 1 'cpu MHz'");
  printf("Total Memory\n");
  system("cat /proc/meminfo | grep -m 1 'MemTotal'");
  printf("Free Memory\n");
  system("cat /proc/meminfo | grep -m 1 'MemFree'");
  printf("OS Name\n");
  system("cat /etc/os-release | grep -m 1 'NAME'");
  printf("OS Version\n");
  system("cat /etc/os-release | grep -m 1 'VERSION'");
}
```

OUTPUT:

```
>>> ~ gcc 3_OS_INFO.c -o 3_OS_INFO && ./3_OS_INFO
Extracting system information using gcc compiler.

Model Name
model name      : AMD Ryzen 5 5600H with Radeon Graphics
Cache Size
cache size      : 512 KB
Number of CPU CORES
cpu cores       : 6
CPU Clock Speed
cpu MHz         : 3481.377
Total Memory
MemTotal:       15198792 kB
Free memory
MemFree:        13422764 kB
OS Name:
NAME="Archcraft"
OS Version
>>> ~
```

PROBLEM STATEMENT 4:

Write a ‘C’ program using ‘gcc’ compiler to extract PID of parent and child processes. Child process is to be created using fork() system call.

OBJECTIVE:

To extract the PIDs (process IDs) of the parent and child processes in a C program using the fork() system call.

C SOURCE CODE:

```c
#include <stdio.h>
#include <unistd.h>

int main() {
  int child_pid;
  child_pid = fork();

  if (child_pid > 0) {
          printf("I am the parent process (PID: %d)\n", getpid());
          printf("My child's PID is: %d\n", child_pid);
  } else if (child_pid == 0) {
          printf("I am the child process (PID: %d)\n", getpid());
  }
}
```

OUTPUT:

```
>>> ~ gcc 4_PPID_PID.c -o 4_PPID_PID && ./4_PPID_PID
I am the parent process (PID: 2751)
My child's PID is: 2752
I am the child process (PID: 2752)
>>> ~
```

PROBLEM STATEMENT 5:

Write and execute a C program to implement FCFS CPU Scheduling Algorithm.

OBJECTIVE:

To implement the First-Come First-Serve (FCFS) CPU scheduling algorithm in C.

THEORY:

The FCFS algorithm is a simple and straightforward scheduling algorithm that allocates the CPU to the process that requests it first, without taking into account the process's priority, CPU burst time, or other factors.

The FCFS algorithm is often used as a baseline or reference algorithm to compare the performance of other scheduling algorithms. It is simple to implement and does not require any additional data structures or overhead. However, it can lead to long waiting times and poor overall performance, especially in systems with multiple processes with different CPU burst times or priorities.

C SOURCE CODE:

```
#include <stdio.h>

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

int main() {
  int num_processes;

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

  struct Process processes[num_processes];

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

```
  for (int i = 0; i < num_processes - 1; i++) {
          int min_ind = i;
          for (int j = i + 1; j < num_processes; j++) {
          if (processes[j].arrival_time < processes[min_ind].arrival_time) min_ind = j;
          }
          struct Process temp = processes[i];
          processes[i] = processes[min_ind];
          processes[min_ind] = temp;
  }
  double avg_TAT = 0, avg_WT = 0;
  printf("\nProcess\tArrival Time\tBurst Time\tStart Time\tCompletion Time\tTurnaround
Time\tWaiting Time\n");
  int current_time = 0;
  for (int i = 0; i < num_processes; i++) {
          current_time = current_time > processes[i].arrival_time ? current_time :
processes[i].arrival_time;
          processes[i].start_time = current_time;
          processes[i].completion_time = current_time + processes[i].burst_time;
          processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
          processes[i].waiting_time = processes[i].start_time - processes[i].arrival_time;
          avg_TAT += processes[i].turnaround_time;
          avg_WT += processes[i].waiting_time;
          current_time += processes[i].burst_time;
          printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrival_time,
processes[i].burst_time, processes[i].start_time, processes[i].completion_time,
processes[i].turnaround_time, processes[i].waiting_time);
  }

  printf("\nAverage Turnaround Time is: %f\nAverage Waiting Time is: %f", avg_TAT /
num_processes, avg_WT / num_processes);
}
```

OUTPUT:

```
PS C:\Users\lucky> gcc .\5_FCFS.c -o .\5_FCFS && .\5_FCFS
Enter the number of processes: 5
Enter id, arrival time, and burst time for process 1: 1 5 3
Enter id, arrival time, and burst time for process 2: 2 3 5
Enter id, arrival time, and burst time for process 3: 3 0 5
Enter id, arrival time, and burst time for process 4: 4 5 1
Enter id, arrival time, and burst time for process 5: 5 8 4

Process Arrival Time    Burst Time      Start Time      Completion Time Turnaround Time Waiting Time
3       0               5               0               5               5               0
2       3               5               5               10              7               2
1       5               3               10              13              8               5
4       5               1               13              14              9               8
5       8               4               14              18              10              6

Average Turnaround Time is: 7.800000
Average Waiting Time is: 4.200000
PS C:\Users\lucky> |
```

PROBLEM STATEMENT 6:

Write and execute a C program to implement SJF CPU Scheduling Algorithm.

OBJECTIVE:

To implement the Shortest Job First (SJF) CPU scheduling algorithm in C.

THEORY:

The Shortest Job First (SJF) CPU scheduling algorithm is a scheduling algorithm that allocates the CPU to the process with the shortest burst time, without taking into account the process's arrival time or priority. The SJF algorithm aims to minimise the waiting time and turnaround time of the processes by allocating the CPU to the shortest processes first.

There are two variants of the SJF algorithm:

- Non-preemptive SJF: In this variant, once a process starts executing, it continues to execute until it completes or blocks on I/O. This can lead to long waiting times for processes with longer burst times.
- Preemptive SJF: In this variant, a process can be interrupted and moved to the back of the queue if a shorter process arrives. This can reduce the waiting time for shorter processes but can also increase the overhead of context switching and process management.

C SOURCE CODE:

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

void main() {
  printf("Enter the number of processes: ");
  int numProcesses;
  scanf("%d", &numProcesses);
  int burstTime[numProcesses];
  int arrivalTime[numProcesses];
  int temp;
  int processNumber[numProcesses];
  bool completed[numProcesses];
  for (int i = 0; i < numProcesses; ++i) {
        printf("Process %d \nEnter the Arrival time : ", i + 1);
        scanf("%d", &arrivalTime[i]);
        printf("Enter the CPU BURST time : ");
        scanf("%d", &burstTime[i]);
        processNumber[i] = i + 1;
        completed[i] = false;
```

```
        printf("\n");
    }
    for (int i = 0; i < numProcesses; ++i) {
        for (int k = i + 1; k < numProcesses; ++k) {
        if (burstTime[i] > burstTime[k]) {
        temp = burstTime[i];
        burstTime[i] = burstTime[k];
        burstTime[k] = temp;
        temp = arrivalTime[i];
        arrivalTime[i] = arrivalTime[k];
        arrivalTime[k] = temp;
        temp = processNumber[i];
        processNumber[i] = processNumber[k];
        processNumber[k] = temp;
        } else if (burstTime[i] == burstTime[k]) {
        if (arrivalTime[i] > arrivalTime[k]) {
        temp = burstTime[i];
        burstTime[i] = burstTime[k];
        burstTime[k] = temp;
        temp = arrivalTime[i];
        arrivalTime[i] = arrivalTime[k];
        arrivalTime[k] = temp;
        temp = processNumber[i];
        processNumber[i] = processNumber[k];
        processNumber[k] = temp;
        }
        }
        }
    }

    int earliestArrivalTime = arrivalTime[0];
    int earliestArrivalTimeIndex;
    for (int i = 0; i < numProcesses; ++i) {
        if (earliestArrivalTime > arrivalTime[i]) {
        earliestArrivalTime = arrivalTime[i];
        earliestArrivalTimeIndex = i;
        }
    }
    completed[earliestArrivalTimeIndex] = true;
    int currentTime = burstTime[earliestArrivalTimeIndex] +
                arrivalTime[earliestArrivalTimeIndex];
    for (int k = 0; k < numProcesses; ++k) {
        printf("Process %d  Arrival time %d : CPU BURST %d \n", processNumber[k],
        arrivalTime[k], burstTime[k]);
    }
    printf("\n");

    printf(
        "Process %d || Start Time %d  || END Time %d || Waiting Time %d || "
```

```
"TurnAround Time %d\n",
processNumber[earliestArrivalTimeIndex],
arrivalTime[earliestArrivalTimeIndex], currentTime, 0,
burstTime[earliestArrivalTimeIndex]);

 for (int i = 0; i < numProcesses; ++i) {
        for (int p = 0; p < numProcesses; ++p) {
        if (completed[p] == true) {
        continue;
        } else if (currentTime >= arrivalTime[p]) {
        completed[p] = true;
        printf(
        "Process %d || Start Time %d || END Time %d || Waiting Time %d || "
        "TurnAround time %d \n",
        processNumber[p], currentTime, currentTime + burstTime[p],
        currentTime - arrivalTime[p],
        currentTime - arrivalTime[p] + burstTime[p]);
        currentTime = currentTime + burstTime[p];
        break;
        }
        }
 }
}
```

OUTPUT:

```
PS C:\Users\lucky> gcc .\6_SJW.c -o .\6_SJW && .\6_SJW
Enter the number of processes: 5
Process 1
Enter the Arrival time : 5 2
Enter the CPU BURST time :
Process 2
Enter the Arrival time : 6 4
Enter the CPU BURST time :
Process 3
Enter the Arrival time : 2 7
Enter the CPU BURST time :
Process 4
Enter the Arrival time :  1 8
Enter the CPU BURST time :
Process 5
Enter the Arrival time : 0 7
Enter the CPU BURST time :
Process 1  Arrival time 5 : CPU BURST 2
Process 2  Arrival time 6 : CPU BURST 4
Process 5  Arrival time 0 : CPU BURST 7
Process 3  Arrival time 2 : CPU BURST 7
Process 4  Arrival time 1 : CPU BURST 8

Process 5 || Start Time 0  || END Time 7 || Waiting Time 0 || TurnAround Time 7
Process 1 || Start Time 7 || END Time 9 || Waiting Time 2 || TurnAround time 4
Process 2 || Start Time 9 || END Time 13 || Waiting Time 3 || TurnAround time 7
Process 3 || Start Time 13 || END Time 20 || Waiting Time 11 || TurnAround time 18
Process 4 || Start Time 20 || END Time 28 || Waiting Time 19 || TurnAround time 27
PS C:\Users\lucky> |
```

PROBLEM STATEMENT 7:

Write and execute a C program to implement FCFS Page replacement Algorithm.

OBJECTIVE:

To implement the First-Come First-Serve (FCFS) Page replacement algorithm in C.

THEORY:

The First-Come First-Serve (FCFS) page replacement algorithm is a simple page replacement algorithm that allocates memory to pages in the order they are requested. The FCFS algorithm is based on the principle of "first in, first out," and it does not take into account the access patterns or frequencies of the pages.

The FCFS algorithm is simple to implement but can lead to poor performance, especially in systems with multiple processes or pages with different access patterns or frequencies. The FCFS algorithm does not take into account the access patterns or frequencies of the pages, and it can lead to long waiting times and high page fault rates for pages that are accessed frequently or recently.

C SOURCE CODE:

```
#include <stdio.h>

int main() {
  int num_frames;
  int num_references;
  int page_faults;

  printf("Enter the number of frames: ");
  scanf("%d", &num_frames);
  printf("Enter the number of references: ");
  scanf("%d", &num_references);

  int frames[num_frames];
  int references[num_references];
  printf("Enter the reference string: ");
  for (int i = 0; i < num_references; i++) {
          scanf("%d", &references[i]);
  }
  for (int i = 0; i < num_frames; i++) {
          frames[i] = -1;
  }
```

```
 page_faults = 0;
 for (int i = 0; i < num_references; i++) {
         int found = 0;
         for (int j = 0; j < num_frames; j++) {
         if (frames[j] == references[i]) {
         found = 1;
         break;
         }
         }
         if (!found) {
         frames[page_faults % num_frames] = references[i];
         page_faults++;
         }
         printf("[");
         for (int i = 0; i < num_frames; i++) {
         printf("%d, ", frames[i]);
         }
         printf("]\n");
 }
 printf("Total page faults: %d\n", page_faults);
}
```

OUTPUT:

```
PS C:\Users\lucky> gcc .\7_FCFS.c -o .\7_FCFS && .\7_FCFS
Enter the number of frames: 3
Enter the number of references: 9
Enter the reference string: 1 7 4 8 4 1 8 9 1
[1, -1, -1, ]
[1, 7, -1, ]
[1, 7, 4, ]
[8, 7, 4, ]
[8, 7, 4, ]
[8, 1, 4, ]
[8, 1, 4, ]
[8, 1, 9, ]
[8, 1, 9, ]
Total page faults: 6
PS C:\Users\lucky> |
```

PROBLEM STATEMENT 8:

Write and execute a C program to implement LRU Page Replacement Algorithm.

OBJECTIVE:

To implement the Least Recently Used (LRU) Page replacement algorithm in C.

THEORY:

The Least Recently Used (LRU) page replacement algorithm is a page replacement algorithm that allocates memory to pages based on their access patterns or frequencies. The LRU algorithm aims to minimize the page fault rate and the memory overhead by replacing the least recently used pages first.

The LRU algorithm is known to have good average page fault rate performance, especially in systems with a large number of processes or pages with different access patterns or frequencies. However, it requires additional memory and processing overhead to maintain the queue or list and the access times or frequencies of the pages. It also does not take into account the priority or fairness of the processes or pages. As a result, the LRU algorithm is not always practical or suitable for all systems.

C SOURCE CODE:

```
#include <stdio.h>

int main() {
  int num_frames;
  int num_references;
  int page_faults;

  printf("Enter the number of frames: ");
  scanf("%d", &num_frames);
  printf("Enter the number of references: ");
  scanf("%d", &num_references);

  int frames[num_frames];
  int references[num_references];
  int last_used[num_frames];
  printf("Enter the reference string: ");
  for (int i = 0; i < num_references; i++) {
          scanf("%d", &references[i]);
  }
  for (int i = 0; i < num_frames; i++) {
          frames[i] = -1;
```

```
        last_used[i] = -1;
    }

   page_faults = 0;
    printf("Frames are as follows (-1 signifies empty frame spot):\n\n");

    for (int i = 0; i < num_references; i++) {
        int found = 0;
        for (int j = 0; j < num_frames; j++) {
        if (frames[j] == references[i]) {
        found = 1;
        last_used[j] = i;
        break;
        }
        }
        if (!found) {
        int oldest_index = 0;
        for (int j = 1; j < num_frames; j++) {
        if (last_used[j] < last_used[oldest_index]) {
        oldest_index = j;
        }
        }
        frames[oldest_index] = references[i];
        last_used[oldest_index] = i;
        page_faults++;
        }
        printf("[");
        for (int i = 0; i < num_frames; i++) {
        printf("%d, ", frames[i]);
        }
        printf("]\n");
    }
    printf("\nTotal page faults: %d\n", page_faults);
}
```

OUTPUT:

```
PS C:\Users\lucky> gcc .\8_LRU.c -o .\8_LRU && .\8_LRU
Enter the number of frames: 3
Enter the number of references: 11
Enter the reference string: 1 8 9 3 5 8 3 3 6 8 6
Frames are as follows (-1 signifies empty frame spot):

[1, -1, -1, ]
[1, 8, -1, ]
[1, 8, 9, ]
[3, 8, 9, ]
[3, 5, 9, ]
[3, 5, 8, ]
[3, 5, 8, ]
[3, 5, 8, ]
[3, 6, 8, ]
[3, 6, 8, ]
[3, 6, 8, ]

Total page faults: 7
PS C:\Users\lucky>
```

PROBLEM STATEMENT 9:

Write and execute a C program to implement SSTF Disk Scheduling Algorithm.

OBJECTIVE:

To implement the Shortest-Seek-Time-First (SSTF) Disk Scheduling algorithm in C.

THEORY:

SSTF (Shortest Seek Time First) is a disk scheduling algorithm used in operating systems to schedule requests for disk access. It works by selecting the request with the shortest seek time from the current head position and servicing it first. The head is then moved to the position of the serviced request, and the process is repeated until all requests have been serviced.

The SSTF algorithm is known to have good average seek time performance, especially in systems with a small number of disk access requests or stable access patterns. However, it requires additional memory and processing overhead to maintain the queue and compute the seek times of the requests. It also does not take into account the priority or fairness of the requests or the direction of the head movement. As a result, the SSTF algorithm is not always practical or suitable for all systems.

C SOURCE CODE:

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

int main() {
  int num_requests;
  int head_position;
  int total_seek_distance = 0;
  printf("Enter the number of disk requests: ");
  scanf("%d", &num_requests);
  int requests[num_requests];

  printf("Enter the disk requests: ");
  for (int i = 0; i < num_requests; i++) {
        scanf("%d", &requests[i]);
  }
  printf("Enter the current head position: ");
  scanf("%d", &head_position);

  printf("\nSeek Sequence is: ");
```

```
 while (num_requests > 0) {
        int min_seek_time = INT_MAX;
        int min_seek_time_index = -1;

        for (int i = 0; i < num_requests; i++) {
        int seek_time = abs(requests[i] - head_position);
        if (seek_time < min_seek_time) {
        min_seek_time = seek_time;
        min_seek_time_index = i;
        }
        }
        printf("%d ", requests[min_seek_time_index]);
        head_position = requests[min_seek_time_index];
        for (int i = min_seek_time_index; i < num_requests - 1; i++) {
        requests[i] = requests[i + 1];
        }
        num_requests--;
        total_seek_distance += min_seek_time;
 }
 printf("\nTotal Head Movements: %d", total_seek_distance);
}
```

OUTPUT:

```
PS C:\Users\lucky> gcc .\9_SSTF.c -o .\9_SSTF && .\9_SSTF
Enter the number of disk requests: 10
Enter the disk requests: 10 70 60 30 55 80 70 90 15 40
Enter the current head position: 53

Seek Sequence is: 55 60 70 70 80 90 40 30 15 10
Total Head Movements: 117
PS C:\Users\lucky> |
```

PROBLEM STATEMENT 10:

Write and execute a C program to implement SCAN Disk Scheduling Algorithm.

OBJECTIVE:

To implement the SCAN CPU scheduling algorithm in C.

THEORY:

SCAN (also known as the Elevator Algorithm) is a disk scheduling algorithm used in operating systems to schedule requests for disk access. It works by moving the disk head in one direction, servicing all requests in that direction, and then reversing direction and servicing all requests in the other direction.

The SCAN algorithm reduces the average seek time for disk accesses by moving the head back and forth across the entire disk, rather than just servicing requests in the order they are received. This can be especially beneficial in systems with many processes competing for disk access, as it allows the disk head to service requests from multiple processes in a single pass.

C SOURCE CODE:

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

int main() {
  int num_requests;
  int head_position;
  int direction;
  int total_seek_operations;
  int disk_size;

  printf("Enter the number of disk requests: ");
  scanf("%d", &num_requests);
  printf("Enter the disk size: ");
  scanf("%d", &disk_size);

  int requests[num_requests];
  printf("Enter the disk requests: ");
  for (int i = 0; i < num_requests; i++) {
          scanf("%d", &requests[i]);
  }
  printf("Enter the current head position: ");
  scanf("%d", &head_position);
```

```
int temp_head_position = head_position;


printf("Enter the direction (0 for left, 1 for right): ");
scanf("%d", &direction);

for (int i = 0; i < num_requests - 1; i++) {
        int min_ind = i;
        for (int j = i + 1; j < num_requests; j++) {
        if (requests[j] < requests[min_ind]) {
        min_ind = j;
        }
        }
        int temp = requests[i];
        requests[i] = requests[min_ind];
        requests[min_ind] = temp;
}
total_seek_operations = 0;
printf("\nScheduled requests: ");
if (direction == 0) {
        for (int i = num_requests - 1; i >= 0; i--) {
        if (requests[i] <= head_position) {
        printf("%d ", requests[i]);
        total_seek_operations += abs(requests[i] - temp_head_position);
        temp_head_position = requests[i];
        }
        }
        total_seek_operations += temp_head_position - 0;
        temp_head_position = 0;
        printf("%d ", temp_head_position);

        for (int i = 0; i < num_requests; i++) {
        if (requests[i] > head_position) {
        printf("%d ", requests[i]);
        total_seek_operations += abs(requests[i] - temp_head_position);
        temp_head_position = requests[i];
        }
        }
} else {
        for (int i = 0; i < num_requests; i++) {
        if (requests[i] >= head_position) {
        printf("%d ", requests[i]);
        total_seek_operations += abs(requests[i] - temp_head_position);
        temp_head_position = requests[i];
        }
        }
        total_seek_operations += disk_size - temp_head_position;
        temp_head_position = disk_size;
        printf("%d ", temp_head_position);
```

```
        for (int i = num_requests - 1; i >= 0; i--) {
        if (requests[i] < head_position) {

        printf("%d ", requests[i]);
        total_seek_operations += abs(requests[i] - temp_head_position);
        temp_head_position = requests[i];
        }
        }
  }
  printf("\n");
  printf("Total seek operations: %d\n", total_seek_operations);
}
```

OUTPUT:

```
PS C:\Users\lucky> gcc .\10_SCAN.c -o .\10_SCAN && .\10_SCAN
Enter the number of disk requests: 10
Enter the disk size: 200
Enter the disk requests: 10 60 170 80 90 145 65 180 110 50
Enter the current head position: 50
Enter the direction (0 for left, 1 for right): 0

Scheduled requests: 50 10 0 60 65 80 90 110 145 170 180
Total seek operations: 230
PS C:\Users\lucky>
```