Operating System
LAB RECORD

Submitted By :- Ayushman Mishra

Roll No :- S3322BCA001

Year :- BCA 3rd Semester

Experiment no.	Name Of Experiment
1	Write a program (using fork() and/or exec() commands) where parent and child execute: **a)** same program, same code. **b)**same program, different code. **c)**before terminating, the parent waits for the child to finish its task.
2	Write a program to report behavior of Linux kernel including kernel version, CPU type andmodel. (CPU information)
3	Write a program to report behavior of Linux kernel including information on configuredmemory, amount of free and used memory. (memory information)
4	Write a program to print file details including owner access permissions, file access time,where file name is given as argument.
5	Write a program to copy files using system calls.
6	Write a program using C to implement FCFS scheduling algorithm.
7	Write a program using C to implement Round Robin scheduling algorithm.
8	Write a program using C to implement SJF scheduling algorithm.

9	Write a program using C to implement non-preemptive priority based scheduling algorithm.
10	Write a program using C to implement preemptive priority based scheduling algorithm.
11	Write a program using C to implement SRTF scheduling algorithm.
12	Write a program using C to implement first-fit, best-fit and worst-fit allocation strategies.

Experiment-1 :- Write a program (using fork() and/or exec() commands) where parent and child execute:

a) Same program, same code.

Code

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

int main() {

  int pid;
  pid = fork();

  if (pid == 0) {
    printf("I am the child process.\n");
  } else {
    printf("I am the parent process.\n");
  }

  return 0;
}
```

Output

```
I am the child process.

I am the parent process.
```

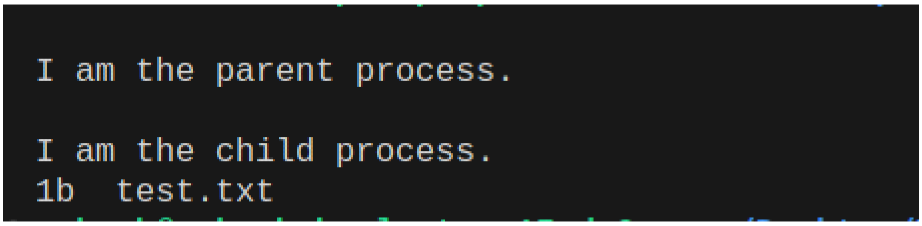
b) Same program, different code.

Code

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

int main() {
  int pid;
```

```
  pid = fork();

  if (pid == 0) {
    printf("I am the child process.\n");
    execlp("ls", "ls", NULL);
  } else {
    printf("I am the parent process.\n");
  }

  return 0;
}
```

Output



c) Before terminating, the parent waits for the child to finish its task.

Code

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

int main()
{
  int pid, status;
  pid = fork();

  if (pid == 0)
  {
    printf("I am the child process.\n");
    execlp("ls", "ls", NULL);
  }
  else
  {
    printf("I am the parent process.\n");
    wait(&status);
    printf("The child process has finished.\n");
  }

  return 0;
```

```
}
```

Output

```
I am the parent process.

I am the child process.
1b   1c   test.txt

The child process has finished.
```

Experiment-2 :- Write a program to report behavior of Linux kernel including kernel version, CPU type and model. (CPU information)

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/utsname.h>

int main()
{
    struct utsname uts;
    uname(&uts);

    printf("Kernel version: %s\n", uts.release);
    printf("CPU type: %s\n", uts.machine);
    printf("Name of OS: %s\n", uts.sysname);
    printf("Name of Machine: %s\n", uts.nodename);
    printf("Release number of Kernel: %s\n", uts.version);

    return 0;
}
```

Output

```
Kernel version: 5.19.0-38-generic
CPU type: x86_64
Name of OS: Linux
Name of Machine: rakesh-hp-laptop-15-da0xxx
Release number of Kernel: #39~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Fri Mar 17 21:16:15 UTC 2
```

Experiment-3 :- Write a program to report behavior of Linux kernel including information on configured memory, amount of free and used memory. (memory information)

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/sysinfo.h>

int main()
{
```

```
    struct sysinfo si;
    sysinfo(&si);

    printf("Configured memory: %lu\n", si.totalram);
    printf("Free memory: %lu\n", si.freeram);
    printf("Used memory: %lu\n", si.totalram - si.freeram);

    return 0;
}
```

Output

```
Configured memory: 12461580288
Free memory: 2807148544
Used memory: 9654431744
```

Experiment-4 :- Write a program to print file details including owner access permissions, file access time, where file name is given as argument.

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>

int main(int argc, char **argv)
{
    struct stat file_stat;
    char *file_name;

    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s <file_name>\n", argv[0]);
        exit(1);
    }

    file_name = argv[1];

    if (stat(file_name, &file_stat) < 0)
    {
        perror("stat");
        exit(1);
    }

    printf("File name: %s\n", file_name);
    printf("Owner permissions: %o\n", file_stat.st_mode & 0777);
    printf("File access time: %s\n", ctime(&file_stat.st_atime));
```

```
    return 0;
}
```

Code Snippet

```
gcc -o 4 4.c
./"4" test.txt
```

Output

```
File name: test.txt
Owner permissions: 664
File access time: Tue May 16 08:02:37 2023
```

Experiment-5 :- Write a program to copy files using system calls.

Code

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

int main(int argc, char **argv)
{
    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s <source_file> <destination_file>\n",
argv[0]);
        exit(1);
    }

    char *source_file = argv[1];
    char *destination_file = argv[2];

    int source_fd = open(source_file, O_RDONLY);
    if (source_fd < 0)
    {
        perror("open");
        exit(1);
    }

    int destination_fd = open(destination_file, O_WRONLY | O_CREAT, 0666);
    if (destination_fd < 0)
    {
        perror("open");
        exit(1);
    }
```

```
    char buffer[1024];
    ssize_t bytes_read;
    while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0)
    {
        write(destination_fd, buffer, bytes_read);
    }

    close(source_fd);
    close(destination_fd);

    return 0;
}
```

Code Snippet

```
gcc -o 5 5.c
./"5" test.txt test2.txt
```

Output

```
test.txt                              test2.txt
1   This is Manish Kumar Ray          1   This is Manish Kumar Ray
```

Experiment-6 :- Write a program using C to implement FCFS scheduling algorithm.

Code

```
#include <stdio.h>

int main()
{
    // Declare variables
    int n, i, j, bt[10], wt[10], tat[10], total_wt = 0, total_tat = 0;

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

    for (i = 0; i < n; i++)                     // Get the burst time of each process
    {
        printf("Enter the burst time of process %d: ", i + 1);
        scanf("%d", &bt[i]);
    }

    wt[0] = 0;                           // Calculate the waiting time of each process
    for (i = 1; i < n; i++)
    {
        wt[i] = wt[i - 1] + bt[i - 1];
```

```
    }

    tat[0] = bt[0];                // Calculate the turnaround time of each process
    for (i = 1; i < n; i++)
    {
        tat[i] = tat[i - 1] + bt[i];
    }

    for (i = 0; i < n; i++)    // Calculate the average waiting time and turnaround
time
    {
        total_wt += wt[i];
        total_tat += tat[i];
    }
    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;

    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++)
    {
        printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
    }

    printf("Average waiting time: %f\n", avg_wt);
    printf("Average turnaround time: %f\n", avg_tat);

    return 0;
}
```

Output

```
Enter the number of processes: 4

Enter the burst time of process 1: 2
Enter the burst time of process 2: 2
Enter the burst time of process 3: 3
Enter the burst time of process 4: 4

Process    Burst Time   Waiting Time    Turnaround Time
1               2               0               2
2               2               2               4
3               3               4               7
4               4               7               11

Average waiting time: 3.250000
Average turnaround time: 6.000000
```

Experiment-7 :- Write a program using C to implement Round Robin scheduling algorithm.

Code

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

```
int main()
{
    int n, i, j, bt[10], wt[10], tat[10], total_wt = 0, total_tat = 0, quantum;

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

    // Get the burst time of each process
    for (i = 0; i < n; i++)
    {
        printf("Enter the burst time of process %d: ", i + 1);
        scanf("%d", &bt[i]);
    }

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

    // Initialize the waiting time and turnaround time of each process
    for (i = 0; i < n; i++)
    {
        wt[i] = 0;
        tat[i] = 0;
    }

    // Calculate the waiting time and turnaround time of each process
    int current_process = 0;
    int remaining_bt = bt[current_process];
    while (1)
    {
        if (remaining_bt <= 0)
        {
            // The current process has finished executing
            tat[current_process] = wt[current_process] + bt[current_process];
            current_process = (current_process + 1) % n;
            remaining_bt = bt[current_process];
            wt[current_process] = 0;
        }
        else
        {
            // The current process is still executing
            remaining_bt -= quantum;
            wt[current_process] += quantum;
            current_process = (current_process + 1) % n;
        }

        if (current_process == 0)
        {
            // All processes have been executed
            break;
```

```
        }
    }

    // Calculate the average waiting time and turnaround time
    for (i = 0; i < n; i++)
    {
        total_wt += wt[i];
        total_tat += tat[i];
    }
    float avg_wt = (float)total_wt / n;
    float avg_tat = (float)total_tat / n;

    // Print the waiting time and turnaround time of each process
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++)
    {
        printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
    }

    // Print the average waiting time and turnaround time
    printf("Average waiting time: %f\n", avg_wt);
    printf("Average turnaround time: %f\n", avg_tat);

    return 0;
}
```

Output

```
Enter the number of processes: 4

Enter the burst time of process 1: 2
Enter the burst time of process 2: 2
Enter the burst time of process 3: 3
Enter the burst time of process 4: 4

Enter the time quantum: 2

Process Burst Time      Waiting Time    Turnaround Time
1               2               2               0
2               2               0               2
3               3               2               0
4               4               2               0

Average waiting time: 1.500000
Average turnaround time: 0.500000
```

Experiment-8 :- Write a program using C to implement SJF

scheduling algorithm.

Code

```
#include <stdio.h>

// Function to sort the processes according to their burst time
void sort(int p[], int b[], int n)
{
    int i, j, temp, min;

    for (i = 0; i < n - 1; i++)
    {
        min = i;

        for (j = i + 1; j < n; j++)
        {
            if (b[min] > b[j])
            {
                min = j;
            }
        }

        temp = p[i];
        p[i] = p[min];
        p[min] = temp;

        temp = b[i];
        b[i] = b[min];
        b[min] = temp;
    }
}

// Function to calculate the average waiting time and turnaround time
void calculate(int p[], int b[], int n, int w[], int t[])
{
    int i, sum_w = 0, sum_t = 0;

    for (i = 0; i < n; i++)
    {
        w[i] = 0;
        t[i] = 0;
    }

    for (i = 0; i < n; i++)
    {
        if (i == 0)
        {
            w[i] = 0;
        }
        else
```

```
        {
            w[i] = w[i - 1] + b[i - 1];
        }

        t[i] = w[i] + b[i];

        sum_w += w[i];
        sum_t += t[i];
    }

    printf("\nProcess\t   Burst Time\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++)
    {
        printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, b[i], w[i], t[i]);
    }

    printf("Average waiting time = %.2f\n", (float)sum_w / n);
    printf("Average turnaround time = %.2f\n", (float)sum_t / n);
}

int main()
{
    int n, i, p[10], b[10], w[10], t[10];

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

    for (i = 0; i < n; i++)
    {
        printf("Enter the burst time of process %d: ", i + 1);
        scanf("%d", &b[i]);
    }

    sort(p, b, n);

    calculate(p, b, n, w, t);

    return 0;
}
```

Output

```
Enter the number of processes: 4

Enter the burst time of process 1: 5
Enter the burst time of process 2: 3
Enter the burst time of process 3: 4
Enter the burst time of process 4: 2

Process    Burst Time   Waiting Time    Turnaround Time
1               2               0               2
2               3               2               5
3               4               5               9
4               5               9               14

Average waiting time = 4.00
Average turnaround time = 7.50
```

Experiment-9 :- Write a program using C to implement non-preemptive priority based scheduling algorithm.

Code

```
#include <stdio.h>

// Function to swap two variables
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
int main()
{
    int n;
    printf("\nEnter Number of Processes: ");
    scanf("%d", &n);

    // b is array for burst time, p for priority and index for process id
    int b[n], p[n], index[n];
    for (int i = 0; i < n; i++)
    {
        printf("Enter Burst Time and Priority Value for Process %d: ", i + 1);
        scanf("%d %d", &b[i], &p[i]);
        index[i] = i + 1;
    }
    for (int i = 0; i < n; i++)
    {
        int a = p[i], m = i;
```

```
    // Finding out highest priority element and placing it at its desired
position
        for (int j = i; j < n; j++)
        {
            if (p[j] > a)
            {
                a = p[j];
                m = j;
            }
        }

        // Swapping processes
        swap(&p[i], &p[m]);
        swap(&b[i], &b[m]);
        swap(&index[i], &index[m]);
    }

    // T stores the starting time of process
    int t = 0;

    // Printing scheduled process
    printf("\nOrder of process Execution is\n");
    for (int i = 0; i < n; i++)
    {
        printf("P%d is executed from %d to %d\n", index[i], t, t + b[i]);
        t += b[i];
    }
    printf("\n");
    printf("Process Id     Burst Time   Wait Time    TurnAround Time\n");
    int wait_time = 0;
    for (int i = 0; i < n; i++)
    {
        printf("  P%d\t\t%d\t\t%d\t\t%d\n", index[i], b[i], wait_time, wait_time
+ b[i]);
        wait_time += b[i];
    }
    return 0;
}
```

Output

```
Enter Number of Processes: 4

Enter Burst Time and Priority Value for Process 1: 5 20
Enter Burst Time and Priority Value for Process 2: 2 30
Enter Burst Time and Priority Value for Process 3: 6 40
Enter Burst Time and Priority Value for Process 4: 4 10

Order of process Execution is
P3 is executed from 0 to 6
P2 is executed from 6 to 8
P1 is executed from 8 to 13
P4 is executed from 13 to 17

Process Id     Burst Time   Wait Time    TurnAround Time
  P3            6              0               6
  P2            2              6               8
  P1            5              8               13
  P4            4              13              17
```

Experiment-10 :- Write a program using C to implement preemptive priority based scheduling algorithm.

Code

```
#include <stdio.h>
#define MAX 20

typedef struct
{
    int pid;
    int burst;
    int burst_left;
    int priority;
    int arrival;
    int waiting;
    int turnaround;
} Process;

typedef struct
{
    int pid;
    int start;
} Gantt;

void main()
{
    Process p[MAX];
    Gantt gnt[MAX];
    Process tmp;
    int total_waiting = 0;
    int total_turnaround = 0;
```

```
int gnti = 0;
int n, step, next;
int total_burst = 0;

printf("Enter number of processes:\n");
scanf("%d", &n);
printf("Enter Burst times for the processes:\n");
for (int i = 0; i < n; i++)
{
    scanf("%d", &p[i].burst);
    p[i].burst_left = p[i].burst;
    p[i].pid = i;
    total_burst += p[i].burst;
}
printf("Enter Priorities for the processes:\n");
for (int i = 0; i < n; i++)
{
    scanf("%d", &p[i].priority);
}
printf("Enter Arrival time for the processes:\n");
for (int i = 0; i < n; i++)
{
    scanf("%d", &p[i].arrival);
}
for (int t = 0; t < total_burst; t++)
{
    next = -1;
    for (int k = 0; k < n; k++)
    {
        if (p[k].arrival <= t && p[k].burst_left > 0)
        {
            if ((next != -1) && (p[k].priority < p[next].priority))
                next = k;
            else if (next == -1)
                next = k;
        }
    }
    if (next != -1)
    {
        p[next].burst_left--;
        if (gnti != 0)
        {
            if (gnt[gnti - 1].pid != p[next].pid)
            {
                gnt[gnti].start = t;
                gnt[gnti].pid = p[next].pid;
                gnti++;
            }
        }
```

```
            else
            {
                gnt[gnti].pid = p[next].pid;
                gnt[gnti].start = t;
                gnti++;
            }
        }
    }
    for (int i = 0; i < n; i++)
    {
        for (int g = gnti - 1; g >= 0; g--)
        {
            if (gnt[g].pid == p[i].pid)
            {
                if (g != gnti - 1)
                    p[i].waiting = gnt[g + 1].start - p[i].burst;
                else
                    p[i].waiting = total_burst - p[i].burst;
                break;
            }
        }
        p[i].turnaround = p[i].waiting + p[i].burst;
        p[i].waiting -= p[i].arrival;
    }

    printf("\n");
    puts("+-----+------------+----------+-------------+-------------+-----------------+");
    puts("| PID | Burst Time | Priority | Arrival time | Waiting Time |
Turnaround Time |");
    puts("+-----+------------+----------+-------------+-------------+-----------------+");
    for (int i = 0; i < n; i++)
    {
        printf("| %2d  |     %2d     |    %2d    |      %2d      |      %2d      |
%2d       |\n", p[i].pid, p[i].burst, p[i].priority, p[i].arrival, p[i].waiting,
p[i].turnaround);
        puts("+-----+------------+----------+-------------+-------------+-----------------
+");
    }

    for (int i = 0; i < n; i++)
    {
        total_waiting += p[i].waiting;
        total_turnaround += p[i].turnaround;
    }
    printf("Total waiting time: %d\n", total_waiting);
    printf("Average waiting time: %.2f\n", (float)total_waiting / n);
    printf("Total turnaround time: %d\n", total_turnaround);
    printf("Average turnaround time: %.2f\n\n", (float)total_turnaround / n);
```

```
    printf("Gantt Chart:\n\n");
    gnt[gnti].start = total_burst;
    for (int i = 0; i < gnti; i++)
    {
        printf("%d", gnt[i].start);
        step = (gnt[i + 1].start - gnt[i].start) / 2;
        for (int k = 0; k <= step; k++)
            printf("  ");
        printf("P%d", gnt[i].pid);
        for (int k = 0; k <= step; k++)
            printf("  ");
    }

    printf("%d\n\n", total_burst);
}
```

Output

```
Enter number of processes:
4
Enter Burst times for the processes:
5 5 5 5
Enter Priorities for the processes:
1 2 3 4
Enter Arrival time for the processes:
3 2 1 0

+-----+------------+----------+--------------+--------------+-----------------+
| PID | Burst Time | Priority | Arrival time | Waiting Time | Turnaround Time |
+-----+------------+----------+--------------+--------------+-----------------+
|  0  |      5     |     1    |       3      |       0      |        8        |
+-----+------------+----------+--------------+--------------+-----------------+
|  1  |      5     |     2    |       2      |       5      |       12        |
+-----+------------+----------+--------------+--------------+-----------------+
|  2  |      5     |     3    |       1      |      10      |       16        |
+-----+------------+----------+--------------+--------------+-----------------+
|  3  |      5     |     4    |       0      |      15      |       20        |
+-----+------------+----------+--------------+--------------+-----------------+
Total waiting time: 30
Average waiting time: 7.50
Total turnaround time: 56
Average turnaround time: 14.00

Gantt Chart:

0  P3  1  P2  2  P1  3      P0      8      P1      12      P2      16      P3      20
```

Experiment-11 :- Write a program using C to implement SRTF scheduling algorithm.

Code

```
#include <stdio.h>

void main()
{
    int a[10], b[10], x[10];
```

```
    int waiting[10], turnaround[10], completion[10];
    int i, j, smallest, count = 0, time, n;
    double avg = 0, tt = 0, end;
    printf("\n");

    printf("\nEnter the number of Processes: ");
    scanf("%d", &n);
    printf("\n");

    for (i = 0; i < n; i++)
    {
        printf("Enter arrival time of process %d : ", i + 1);
        scanf("%d", &a[i]);
    }
    printf("\n");

    for (i = 0; i < n; i++)
    {
        printf("Enter burst time of process %d : ", i + 1);
        scanf("%d", &b[i]);
    }
    for (i = 0; i < n; i++)
        x[i] = b[i];

    b[9] = 9999;

    for (time = 0; count != n; time++)
    {
        smallest = 9;
        for (i = 0; i < n; i++)
        {
            if (a[i] <= time && b[i] < b[smallest] && b[i] > 0)
                smallest = i;
        }
        b[smallest]--;

        if (b[smallest] == 0)
        {
            count++;
            end = time + 1;
            completion[smallest] = end;
            waiting[smallest] = end - a[smallest] - x[smallest];
            turnaround[smallest] = end - a[smallest];
        }
    }
    printf("\nPID \t Burst Time \t Arrival Time \tWaiting Time \tTurnaround
Time   Completion Time");
    for (i = 0; i < n; i++)
    {
        printf("\n %d \t    %d \t\t    %d\t\t\t%d   \t\t%d\t\t%d", i + 1, x[i], a[i],
```

```
waiting[i], turnaround[i], completion[i]);
        avg = avg + waiting[i];
        tt = tt + turnaround[i];
    }

    printf("\n\nAverage waiting time = %lf\n", avg / n);
    printf("Average Turnaround time = %lf", tt / n);
    printf("\n");
}
```

Output

```
Enter the number of Processes: 4

Enter arrival time of process 1 : 0
Enter arrival time of process 2 : 1
Enter arrival time of process 3 : 2
Enter arrival time of process 4 : 4

Enter burst time of process 1 : 5
Enter burst time of process 2 : 4
Enter burst time of process 3 : 2
Enter burst time of process 4 : 1

PID      Burst Time      Arrival Time   Waiting Time    Turnaround Time   Completion Time
 1          5               0               3               8               8
 2          4               1               7               11              12
 3          2               2               0               2               4
 4          1               4               0               1               5

Average waiting time = 2.500000
Average Turnaround time = 5.500000
```

Experiment-12 :- Write a program using C to implement first-fit, best-fit and worst-fit allocation strategies.

Code

```
#include <stdio.h>
#include <limits.h>
#define num_blocks 5

// Define the structure for a memory block.
struct memory_block
{
    int size;
    int allocated;
};

// Initialize the memory blocks.
struct memory_block blocks[num_blocks] = {
    {100, 0},
    {200, 0},
    {300, 0},
    {400, 0},
    {500, 0}};
```

```
// Print the memory blocks.
void print_blocks()
{
    for (int i = 0; i < num_blocks; i++)
    {
        printf("Block %d: size=%d, allocated=%d\n", i, blocks[i].size,
blocks[i].allocated);
    }
}

// First-fit allocation algorithm.
int first_fit(int size)
{
    for (int i = 0; i < num_blocks; i++)
    {
        if (blocks[i].size >= size && !blocks[i].allocated)
        {
            blocks[i].allocated = 1;
            return i;
        }
    }
    return -1;
}

// Best-fit allocation algorithm.
int best_fit(int size)
{
    int best_block = -1;
    int best_size = INT_MAX;
    for (int i = 0; i < num_blocks; i++)
    {
        if (blocks[i].size >= size && !blocks[i].allocated && blocks[i].size <
best_size)
        {
            best_block = i;
            best_size = blocks[i].size;
        }
    }
    return best_block;
}

// Worst-fit allocation algorithm.
int worst_fit(int size)
{
    int worst_block = -1;
    int worst_size = 0;
    for (int i = 0; i < num_blocks; i++)
    {
        if (blocks[i].size >= size && !blocks[i].allocated && blocks[i].size >
```

```
worst_size)
        {
            worst_block = i;
            worst_size = blocks[i].size;
        }
    }
    return worst_block;
}

int main()
{
    printf("\nInitial memory blocks :- \n");
    print_blocks();

    // Allocate memory using first-fit, best-fit, and worst-fit algorithms.
    int first_block = first_fit(100);
    int best_block = best_fit(100);
    int worst_block = worst_fit(100);

    printf("\nMemory blocks after allocation :-\n");
    print_blocks();

    // Free the memory allocated by first-fit.
    blocks[first_block].allocated = 0;

    printf("\nMemory blocks after freeing :-\n");
    print_blocks();

    return 0;
}
```

Output

```
Initial memory blocks :-
Block 0: size=100, allocated=0
Block 1: size=200, allocated=0
Block 2: size=300, allocated=0
Block 3: size=400, allocated=0
Block 4: size=500, allocated=0

Memory blocks after allocation :-
Block 0: size=100, allocated=1
Block 1: size=200, allocated=0
Block 2: size=300, allocated=0
Block 3: size=400, allocated=0
Block 4: size=500, allocated=0

Memory blocks after freeing :-
Block 0: size=100, allocated=0
Block 1: size=200, allocated=0
Block 2: size=300, allocated=0
Block 3: size=400, allocated=0
Block 4: size=500, allocated=0
```