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

i. same program, same code.

ii. same program, different code.

iii. before terminating, the parent waits for the child to finish its task

program-

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

// Function executed by both parent and child
void same_code() {
    printf("This is the same code executed by both parent and child.\n");
}

// Function executed only by the child process
void different_code() {
    printf("This is the child process executing different code.\n");
}

int main() {
    pid_t pid; // Process ID

    // Create a child process
    pid = fork();
    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        return 1;
    }

    if (pid == 0) {
        // Child process
        printf("Child process starts.\n");

        // Case II: Same program, different code
        different_code();
```

```
        // Uncomment the line below to execute a completely different program:
        // execlp("/bin/echo", "echo", "Hello from child process!", NULL);

        exit(0); // End child process
    } else {
        // Parent process
        printf("Parent process starts.\n");

        // Case I: Same program, same code
        same_code();

        // Case III: Parent waits for child to finish
        wait(NULL); // Parent waits for the child to complete
        printf("Parent process ends after child completes.\n");
    }

    return 0;
}
```

output-

```
root@RIYA:/mnt/c/Users/844ri# cd Documents
root@RIYA:/mnt/c/Users/844ri/Documents# ls
'My Music'  'My Pictures'  'My Videos'   a.out   file1   file2   output   riya   riya.cpp
root@RIYA:/mnt/c/Users/844ri/Documents# vim file.cpp
root@RIYA:/mnt/c/Users/844ri/Documents# g++ file.cpp -o program
root@RIYA:/mnt/c/Users/844ri/Documents# ./program
Parent process starts.
This is the same code executed by both parent and child.
Child process starts.
This is the child process executing different code.
Parent process ends after child completes.
```

Write a program to report behaviour of Linux kernel including kernel version, CPU type and CPU information.

Program-

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

int main() {
    // 1. Report Kernel Version
    printf("Kernel Version:\n");
    system("uname -r"); // 'uname -r' shows the kernel version

    // 2. Report CPU Type
    printf("\nCPU Type:\n");
    system("uname -m"); // 'uname -m' to get the machine arcitecture ingormation

    // 3. Report CPU Information (detailed)
    printf("\nDetailed CPU Information:\n");
    system("lscpu"); // 'lscpu' provides detailed CPU architecture information

    return 0;
}
```

output-

```
root@RIYA:/mnt/c/Users/844ri/Documents# vim file.cpp
root@RIYA:/mnt/c/Users/844ri/Documents# g++ file.cpp -o program
root@RIYA:/mnt/c/Users/844ri/Documents# ./program
Kernel Version:
5.15.167.4-microsoft-standard-WSL2

CPU Type:
x86_64

Detailed CPU Information:
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         48 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen 5 5600H with Radeon Graphics
    CPU family:          25
    Model:               80
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            0
    BogoMIPS:            6587.45
```

Write a program to report behaviour of Linux kernel including information on configured memory, amount of free and used memory. (Memory information)

program-

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

// Function to get and display memory information
void get_memory_info() {
    ifstream meminfo("/proc/meminfo");
    string line;

    if (!meminfo.is_open()) {
        cerr << "Failed to open /proc/meminfo" << endl;
        exit(1);
    }

    cout << "\nMemory Information:\n";

    // Read the /proc/meminfo file line by line and extract memory details
    while (getline(meminfo, line)) {
        if (line.find("MemTotal") != string::npos) {
            cout << line << endl; // Total memory
        }
        if (line.find("MemFree") != string::npos) {
            cout << line << endl; // Free memory
        }
        if (line.find("MemAvailable") != string::npos) {
            cout << line << endl; // Available memory
        }
        if (line.find("Buffers") != string::npos) {
            cout << line << endl; // Buffered memory
        }
        if (line.find("Cached") != string::npos) {
            cout << line << endl; // Cached memory
        }
        if (line.find("SwapTotal") != string::npos) {
            cout << line << endl; // Total swap memory
        }
```

```
        if (line.find("SwapFree") != string::npos) {
            cout << line << endl; // Free swap memory
        }
    }

    meminfo.close();
}

int main() {
    cout << "Linux Kernel Memory Information:\n";

    // Get and display memory information
    get_memory_info();

    return 0;
}
```

output-

```
root@RIYA:/mnt/c/Users/844ri/Documents# vim file1.cpp
root@RIYA:/mnt/c/Users/844ri/Documents# g++ file1.cpp -o program
root@RIYA:/mnt/c/Users/844ri/Documents# ./program
Linux Kernel Memory Information:

Memory Information:
MemTotal:        7802036 kB
MemFree:         7103832 kB
MemAvailable:    7111300 kB
Buffers:            1864 kB
Cached:           209384 kB
SwapCached:            0 kB
SwapTotal:       2097152 kB
SwapFree:        2097152 kB
```

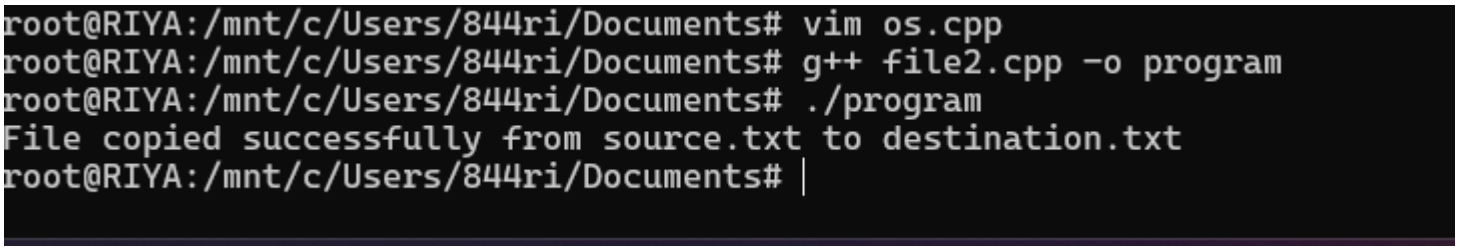
Write a program to copy files using system calls.

Program-

```
#include <iostream>
#include <fcntl.h>   // For open()
#include <unistd.h>  // For read(), write(), close()
#include <sys/types.h>  // For types used in system calls
#include <sys/stat.h>   // For file permissions

using namespace std;

void copy_file(const char* source, const char* destination) {
    // Open the source file in read-only mode
    int source_fd = open("file.cpp", O_RDONLY);
    if (source_fd == -1) {
        perror("Failed to open source file");
        exit(1);
    }

    // Open the destination file in write-only mode (create it if it doesn't exist)
    int dest_fd = open("os.cpp", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    if (dest_fd == -1) {
        perror("Failed to open destination file");
        close(source_fd);
        exit(1);
    }

    char buffer[1024];
    ssize_t bytes_read, bytes_written;

    // Read from the source file and write to the destination file
    while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0) {
        bytes_written = write(dest_fd, buffer, bytes_read);
        if (bytes_written != bytes_read) {
            perror("Error writing to destination file");
            close(source_fd);
            close(dest_fd);
            exit(1);
        }
    }

    if (bytes_read == -1) {
        perror("Error reading from source file");
    }
```

```
    // Close both source and destination files
    close(source_fd);
    close(dest_fd);

    cout << "File copied successfully from " << source << " to " << destination << endl;
}

int main() {
    const char* source = "source.txt";         // Specify source file name
    const char* destination = "destination.txt"; // Specify destination file name

    // Copy the file
    copy_file(source, destination);

    return 0;
}
```

output-



Write a program to implement FCFS scheduling algorithm.

Program-

```
#include <iostream>
using namespace std;

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

   cout << "Enter the number of processes (maximum 20): ";
   cin >> n;
   cout << "\nEnter the burst time for each process:\n";
   for (i = 0; i < n; i++) {
      cout << "P[" << i+1 << "]: ";
      cin >> bt[i];
   }
   wt[0] = 0;
   for (i = 1; i < n; i++) {
      wt[i] = 0;
      for (j = 0; j < i; j++) {
         wt[i] += bt[j];
      }
   }
   cout << "\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time\n";
   for (i = 0; i < n; i++) {
      tat[i] = wt[i] + bt[i];
      avWt += wt[i];
      avTat += tat[i];
      cout << "P[" << i+1 << "]\t\t" << bt[i] << "\t\t" << wt[i] << "\t\t" << tat[i] << "\n";
   }
   avWt /= n;
   avTat /= n;
   cout << "\nAverage Waiting Time: " << avWt;
   cout << "\nAverage Turnaround Time: " << avTat;
   return 0;
}
```

output-

```
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> g++ fcfs.cpp -o program
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> ./program
Enter the number of processes (maximum 20): 3

Enter the burst time for each process:
P[1]: 24
P[2]: 3
P[3]: 3

Process         Burst Time      Waiting Time    Turnaround Time
P[1]            24              0               24
P[2]            3               24              27
P[3]            3               27              30

Average Waiting Time: 17
Average Turnaround Time: 27
```

Write a program to implement SJF scheduling algorithm.

Program-

```
#include <iostream>
using namespace std;
int main() {
   int n, bt[20], wt[20], tat[20];
   int avWt = 0, avTat = 0, i, j;
   cout << "Enter the number of processes (maximum 20): ";
   cin >> n;
   cout << "\nEnter the burst time for each process:\n";
   for (i = 0; i < n; i++) {
      cout << "P[" << i+1 << "]: ";
      cin >> bt[i];
   }
   for (i = 0; i < n; i++) {
      for (j = i+1; j < n; j++) {
         if(bt[i] > bt[j]) {
            int temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;
         }
      }
   }
   wt[0] = 0;
   for (i = 1; i < n; i++) {
      wt[i] = 0;
      for (j = 0; j < i; j++) {
         wt[i] += bt[j];
      }
   }
   cout << "\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time\n";
   for (i = 0; i < n; i++) {
      tat[i] = wt[i] + bt[i];
      avWt += wt[i];
      avTat += tat[i];
      cout << "P[" << i+1 << "]\t\t" << bt[i] << "\t\t" << wt[i] << "\t\t" << tat[i] << "\n";
   }
   avWt /= n;
   avTat /= n;
   cout << "\nAverage Waiting Time: " << avWt;
   cout << "\nAverage Turnaround Time: " << avTat;
   return 0;
}
```

output-

```
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> g++ sjf.cpp -o program
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> ./program
Enter the number of processes (maximum 20): 4

Enter the burst time for each process:
P[1]: 6
P[2]: 8
P[3]: 7
P[4]: 3

Process         Burst Time      Waiting Time    Turnaround Time
P[1]            3               0               3
P[2]            6               3               9
P[3]            7               9               16
P[4]            8               16              24

Average Waiting Time: 7
Average Turnaround Time: 13
```

Write a program to implement non-preemptive priority-based scheduling algorithm

program-

```
#include <iostream>
using namespace std;

int main()
{
   int n;
   cout << "Enter the number of processes: ";
   cin >> n;
   int CPU = 0;     // CPU Current time
   int allTime = 0; // Time needed to finish all processes
   int arrivaltime[n], bursttime[n], priority[n];
   int ATt[n];
   int NoP = n; // number of Processes
   int PPt[n];
   int waitingTime[n];
   int turnaroundTime[n];
   int i = 0;
   cout << "Enter the arrival time, burst time, and priority for each process:\n";
   for (i = 0; i < n; i++)
   {
      cout << "Process " << i + 1 << ":\n";
      cout << "Arrival time: ";
      cin >> arrivaltime[i];
      cout << "Burst time: ";
      cin >> bursttime[i];
      cout << "Priority (lower value means higher priority): ";
      cin >> priority[i];
      PPt[i] = priority[i];
      ATt[i] = arrivaltime[i];
   }
   int LAT = 0; // Last Arrival Time
   for (i = 0; i < n; i++)
      if (arrivaltime[i] > LAT)
         LAT = arrivaltime[i];
   int MAX_P = 0; // Max Priority
   for (i = 0; i < n; i++)
      if (PPt[i] > MAX_P)
         MAX_P = PPt[i];
   int ATi = 0;     // Pointing to Arrival Time index
   int P1 = PPt[0]; // Pointing to 1st priority Value
   int P2 = PPt[0]; // Pointing to 2nd priority Value
```

```
// Finding the First Arrival Time and Highest priority Process
int j = -1;
while (NoP > 0 && CPU <= 1000)
{
   for (i = 0; i < n; i++)
   {
      if ((ATt[i] <= CPU) && (ATt[i] != (LAT + 10)))
      {
         if (PPt[i] != (MAX_P + 1))
         {
            P2 = PPt[i];
            j = 1;

            if (P2 < P1)
            {
               j = 1;
               ATi = i;
               P1 = PPt[i];
               P2 = PPt[i];
            }
         }
      }
   }

   if (j == -1)
   {
      CPU = CPU + 1;
      continue;
   }
   else
   {
      waitingTime[ATi] = CPU - ATt[ATi];
      CPU = CPU + bursttime[ATi];
      turnaroundTime[ATi] = CPU - ATt[ATi];
      ATt[ATi] = LAT + 10;
      j = -1;
      PPt[ATi] = MAX_P + 1;
      ATi = 0;       // Pointing to Arrival Time index
      P1 = MAX_P + 1; // Pointing to 1st priority Value
      P2 = MAX_P + 1; // Pointing to 2nd priority Value
      NoP = NoP - 1;
   }
}
```

```
    cout << "\nProcess_Number\tBurst_Time\tPriority\tArrival_Time\tWaiting_Time\tTurnaround_Time\n\n";
    for (i = 0; i < n; i++)
    {
        cout << "P" << i + 1 << "\t\t" << bursttime[i] << "\t\t" << priority[i] << "\t\t" << arrivaltime[i] << "\t\t" <<
waitingTime[i] << "\t\t" << turnaroundTime[i] << endl;
    }

    float AvgWT = 0;  // Average waiting time
    float AVGTaT = 0; // Average Turnaround time
    for (i = 0; i < n; i++)
    {
        AvgWT = waitingTime[i] + AvgWT;
        AVGTaT = turnaroundTime[i] + AVGTaT;
    }

    cout << "Average waiting time = " << AvgWT / n << endl;
    cout << "Average turnaround time = " << AVGTaT / n << endl;

    return 0;
}
```

output-

```
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> g++ priority.cpp -o program
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> ./program
Enter the number of processes: 5
Enter the arrival time, burst time, and priority for each process:
Process 1:
Arrival time: 0
Burst time: 10
Priority (lower value means higher priority): 3
Process 2:
Arrival time: 0
Burst time: 1
Priority (lower value means higher priority): 1
Process 3:
Arrival time: 0
Burst time: 2
Priority (lower value means higher priority): 4
Process 4:
Arrival time: 0
Burst time: 1
Priority (lower value means higher priority): 5
Process 5:
Arrival time: 0
Burst time: 5
Priority (lower value means higher priority): 2

Process_Number  Burst_Time      Priority        Arrival_Time    Waiting_Time    Turnaround_Time

P1              10              3               0               6               16
P2              1               1               0               0               1
P3              2               4               0               16              18
P4              1               5               0               18              19
P5              5               2               0               1               6
Average waiting time = 8.2
Average turnaround time = 12
```

Write a program to implement SRTF scheduling algorithm

program-

```
#include <iostream>
using namespace std;

int main() {
   int n, bt[20], at[20], wt[20], tat[20], rt[20];
   int time = 0, smallest, remain = 0, end, avWt = 0, avTat = 0;
   cout << "Enter the number of processes (maximum 20): ";
   cin >> n;
   cout << "\nEnter the arrival time and burst time for each process:\n";
   for (int i = 0; i < n; i++) {
      cout << "P[" << i+1 << "] Arrival Time: ";
      cin >> at[i];
      cout << "P[" << i+1 << "] Burst Time: ";
      cin >> bt[i];
      rt[i] = bt[i];
   }
  cout << "\nProcess\t\tWaiting Time\tTurnaround Time\n";
   rt[19] = 9999;
   while (remain != n) {
      smallest = 19;
      for (int i = 0; i < n; i++) {
         if (at[i] <= time && rt[i] > 0 && rt[i] < rt[smallest]) {
            smallest = i;
         }
      }
     rt[smallest]--;
      if (rt[smallest] == 0) {
         remain++;
         end = time + 1;
         wt[smallest] = end - at[smallest] - bt[smallest];
         tat[smallest] = end - at[smallest];
         avWt += wt[smallest];
         avTat += tat[smallest];
         cout << "P[" << smallest+1 << "]\t\t" << wt[smallest] << "\t\t" << tat[smallest] << endl;
      }
      time++;
   }
   cout << "\nAverage Waiting Time: " << (float(avWt) / n);
   cout << "\nAverage Turnaround Time: " << (float(avTat) / n);
   return 0;
```

}

output-

```
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> g++ srtf.cpp -o program
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> ./program
Enter the number of processes (maximum 20): 4

Enter the arrival time and burst time for each process:
P[1] Arrival Time: 0
P[1] Burst Time: 8
P[2] Arrival Time: 1
P[2] Burst Time: 4
P[3] Arrival Time: 2
P[3] Burst Time: 9
P[4] Arrival Time: 3
P[4] Burst Time: 5

Process         Waiting Time    Turnaround Time
P[2]            0               4
P[4]            2               7
P[1]            9               17
P[3]            15              24

Average Waiting Time: 6.5
Average Turnaround Time: 13
```

Write a program to calculate sum of n numbers using Pthreads. A list of n numbers is divided into two smaller list of equal size, two separate threads are used to sum the sub lists.

```
#include <iostream>
#include <pthread.h>

using namespace std;

// Global variables to hold sum results from threads
int sum1 = 0;
int sum2 = 0;

// Structure to hold arguments for each thread
struct SumArgs {
    int* numbers; // Pointer to the array of numbers
    int start;    // Starting index for the sublist
    int end;      // Ending index for the sublist
};

// Function to sum a sublist of numbers
void* sum_list(void* arg) {
    SumArgs* args = (SumArgs*)arg;
    int sum = 0;

    // Sum the numbers in the sublist
    for (int i = args->start; i < args->end; ++i) {
        sum += args->numbers[i];
    }

    // Store result in global variable
    if (args->start == 0) sum1 = sum;
    else sum2 = sum;

    return nullptr;
}

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

    // Dynamically allocate memory for the array
    int* numbers = new int[n];

    cout << "Enter the numbers: ";
    for (int i = 0; i < n; ++i) {
```

```
        cin >> numbers[i];
    }

    // Calculate the middle point to split the list into two halves
    int mid = n / 2;

    // Create two threads
    pthread_t thread1, thread2;

    SumArgs args1 = {numbers, 0, mid};    // First half of the list
    SumArgs args2 = {numbers, mid, n};    // Second half of the list

    // Create the threads
    pthread_create(&thread1, nullptr, sum_list, &args1);
    pthread_create(&thread2, nullptr, sum_list, &args2);

    // Wait for both threads to finish
    pthread_join(thread1, nullptr);
    pthread_join(thread2, nullptr);

    // Calculate the total sum
    int total_sum = sum1 + sum2;

    cout << "The sum of the numbers is: " << total_sum << endl;

    // Free the allocated memory
    delete[] numbers;

    return 0;
}
```

output-

```
root@RIYA:/mnt/c/Users/844ri/Documents# vim file3.cpp
root@RIYA:/mnt/c/Users/844ri/Documents# g++ file3.cpp -o program
root@RIYA:/mnt/c/Users/844ri/Documents# ./program
Enter the number of elements: 5
Enter the numbers: 3 6 8 2 6
The sum of the numbers is: 25
```

Write a program to implement first-fit, best-fit and worst-fit allocation strategies

program-

```
#include <iostream>

using namespace std;

void firstFit(int blockSize[], int blocks, int processSize[], int processes) {
   int allocation[processes];
   for (int i = 0; i < processes; i++)
      allocation[i] = -1;
   for (int i = 0; i < processes; i++) {
      for (int j = 0; j < blocks; j++) {
         if (blockSize[j] >= processSize[i]) {
            allocation[i] = j;
            blockSize[j] -= processSize[i];
            break;
         }
      }
   }
   cout << "\nFirst Fit Allocation:\n";
   cout << "Process No.\tProcess Size\tBlock No.\n";
   for (int i = 0; i < processes; i++) {
      cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";
      if (allocation[i] != -1)
         cout << allocation[i] + 1;
      else
         cout << "Not Allocated";
      cout << endl;
   }
}
void bestFit(int blockSize[], int blocks, int processSize[], int processes) {
   int allocation[processes];
   for (int i = 0; i < processes; i++)
      allocation[i] = -1;
   for (int i = 0; i < processes; i++) {
      int bestIdx = -1;
      for (int j = 0; j < blocks; j++) {
         if (blockSize[j] >= processSize[i]) {
            if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx])
               bestIdx = j;
         }
      }
      if (bestIdx != -1) {
         allocation[i] = bestIdx;
         blockSize[bestIdx] -= processSize[i];
      }
   }

   cout << "\nBest Fit Allocation:\n";
   cout << "Process No.\tProcess Size\tBlock No.\n";
   for (int i = 0; i < processes; i++) {
      cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";
      if (allocation[i] != -1)
         cout << allocation[i] + 1;
      else
```

```
            cout << "Not Allocated";
        cout << endl;
    }
}
void worstFit(int blockSize[], int blocks, int processSize[], int processes) {
    int allocation[processes];
    for (int i = 0; i < processes; i++)
        allocation[i] = -1;
    for (int i = 0; i < processes; i++) {
        int worstIdx = -1;
        for (int j = 0; j < blocks; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx])
                    worstIdx = j;
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }

    cout << "\nWorst Fit Allocation:\n";
    cout << "Process No.\tProcess Size\tBlock No.\n";
    for (int i = 0; i < processes; i++) {
        cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}
int main() {
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int blocks = sizeof(blockSize) / sizeof(blockSize[0]);
    int processes = sizeof(processSize) / sizeof(processSize[0]);
    cout << "Choose Allocation Strategy:\n";
    cout << "1. First Fit\n";
    cout << "2. Best Fit\n";
    cout << "3. Worst Fit\n";
    int choice;
    cin >> choice;

    switch (choice) {
        case 1:
            firstFit(blockSize, blocks, processSize, processes);
            break;
        case 2:
            bestFit(blockSize, blocks, processSize, processes);
            break;
        case 3:
            worstFit(blockSize, blocks, processSize, processes);
            break;
        default:
            cout << "Invalid choice!";
    }
    return 0;
```

}

output-

```
collect2.exe: error: ld returned 1 exit status
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> g++ dynamic_allocation.cpp -o program
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> ./program
Choose Allocation Strategy:
1. First Fit
2. Best Fit
3. Worst Fit
1

First Fit Allocation:
Process No.     Process Size    Block No.
 1              212             2
 2              417             5
 3              112             2
 4              426             Not Allocated
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> ./program
Choose Allocation Strategy:
1. First Fit
2. Best Fit
3. Worst Fit
2

Best Fit Allocation:
Process No.     Process Size    Block No.
 1              212             4
 2              417             2
 3              112             3
 4              426             5
PS C:\Users\844ri\OneDrive\Desktop\New folder (2)> ./program
Choose Allocation Strategy:
1. First Fit
2. Best Fit
3. Worst Fit
3

Worst Fit Allocation:
Process No.     Process Size    Block No.
 1              212             5
 2              417             2
 3              112             5
 4              426             Not Allocated
```