 **THE UNIVERSITY OF AZAD JAMMU AND**  **Kashmir**

**DEPARTMENT OF SOFTWARE ENGINEERING**

**Sub to: Eng. Mam sidra Rafique**

**Sub to: Asif u llah**

**Roll no: 2023-SE-38**

**Subject: operating system**

**Secession: 2023-2027**

**Open ended lab (OEL)** Explanation for Memory Swapping Simulation

Memory swapping is a process where the operating system moves data between RAM (main memory) and disk (secondary storage) to free up space for active processes. It helps run more programs than available physical memory can hold.

**Example**

**Studying with Limited Desk Space**

Your desk (RAM) can hold only 3 books.

You need a 4th book, but no space is left.

You move one book to a shelf (hard drive) to make room.

When you need the old book again, you swap it back.

**Mobile Apps**

You open a game on your phone, but it takes too much space.

The phone moves an unused app to storage to free up RAM.

When you switch back to the old app, it reloads from storage.

This swapping helps computers run multiple programs efficiently.

**Core element:**

**Memory size:**

Memory size refers to the amount of data a computer can store in its memory (RAM or storage). It is measured in units like **KB (Kilobytes), MB (Megabytes), GB (Gigabytes), and TB (Terabytes).**

**Example:**

* A smartphone with **4GB RAM** can run apps smoothly.
* A laptop with **512GB storage** can store many files, videos, and software.
* A USB drive with **16GB** can hold thousands of photos.

Processes:

A process is a program that is running in memory. Each process requires a certain amount of memory (e.g., 200 MB, 400 MB, etc.). If memory is full, the system must decide which process to swap out to accommodate a new process.

Swapping:

Swapping is when a computer moves data from memory (RAM) to storage (hard disk) and brings it back when needed. This helps run more programs even if memory is full.

**Example:**  
If you open too many apps on your phone, some apps pause in the background. When you go back to them, they reload. This is like swapping

Steps for Simulating Memory Swapping

1. Define memory size:
2. In this example, we assume a memory size of 1 GB, which will be represented in the program as 1024 MB (since 1 GB = 1024 MB).

Define processes:

A process is a program that is running on a computer. It includes the program code and its current activity.

**Example:**

* When you open **Google Chrome**, it becomes a process.
* Playing a **video game** runs as a process.

Simulate memory allocation:

Simulating memory allocation means creating a model or program to show how a computer assigns memory to different tasks or programs.

**Example:**  
Imagine a hotel with **10 rooms** (memory). Guests (programs) come in, and the manager (operating system) assigns them rooms. If a guest leaves, the room becomes free for another guest. This is like memory allocation in a compute

* Memory full scenario: .

memory full scenario happens when a computer's RAM or storage has no space left to run new programs or save files.

**Example:**

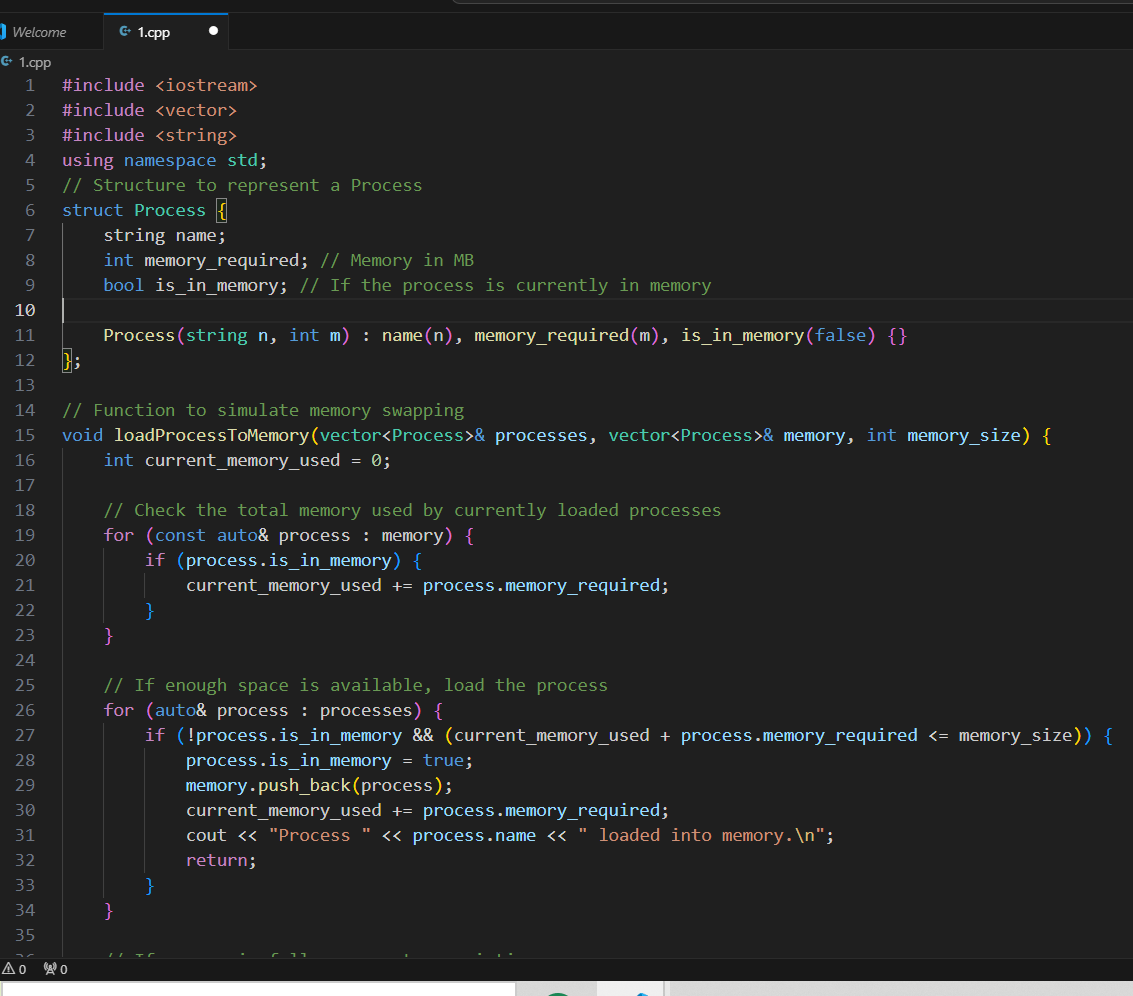
* If your **phone's storage is full**, you can't take new photos.
* If your **computer's RAM is full**, it may slow down or crash when opening more ap

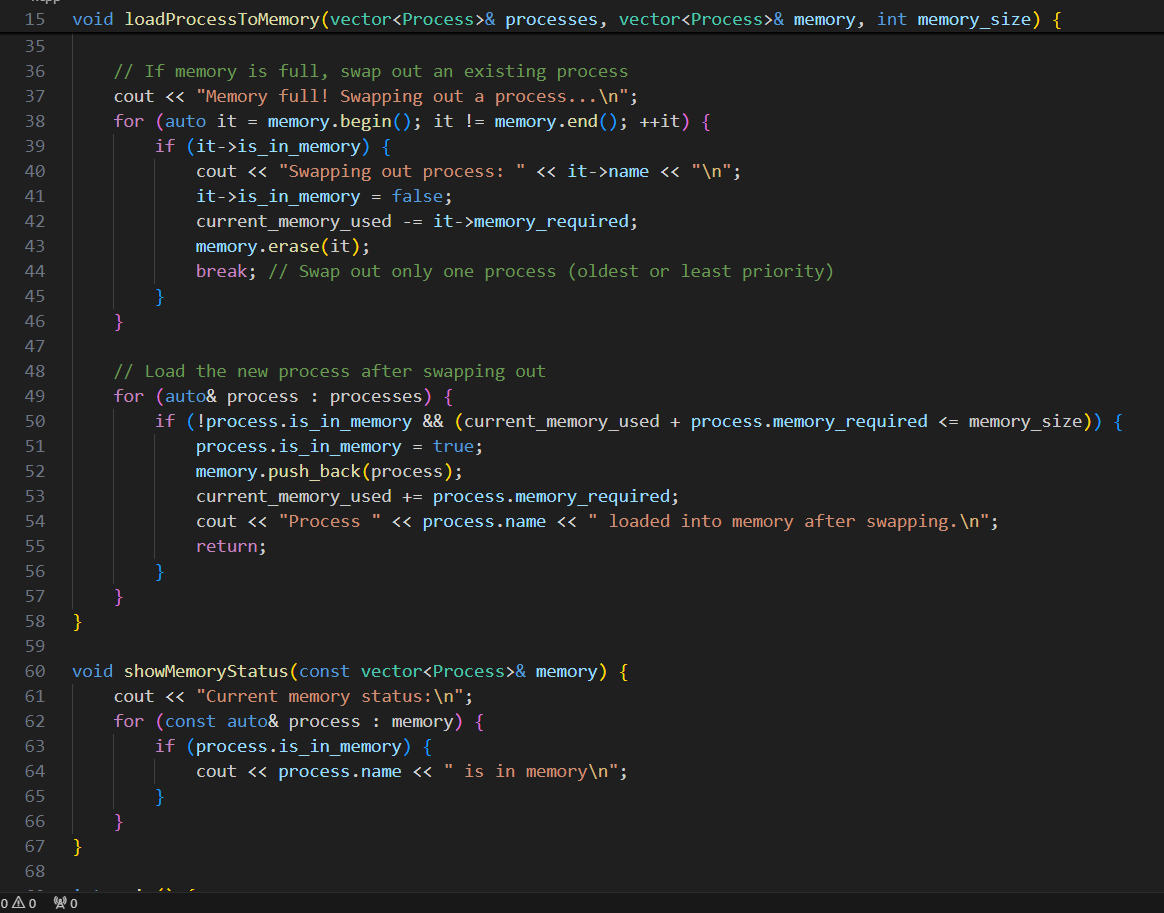
Swap out a process: Swapping out a process means moving a program from RAM (fast memory) to the hard disk (slow storage) to free up space for other active programs.

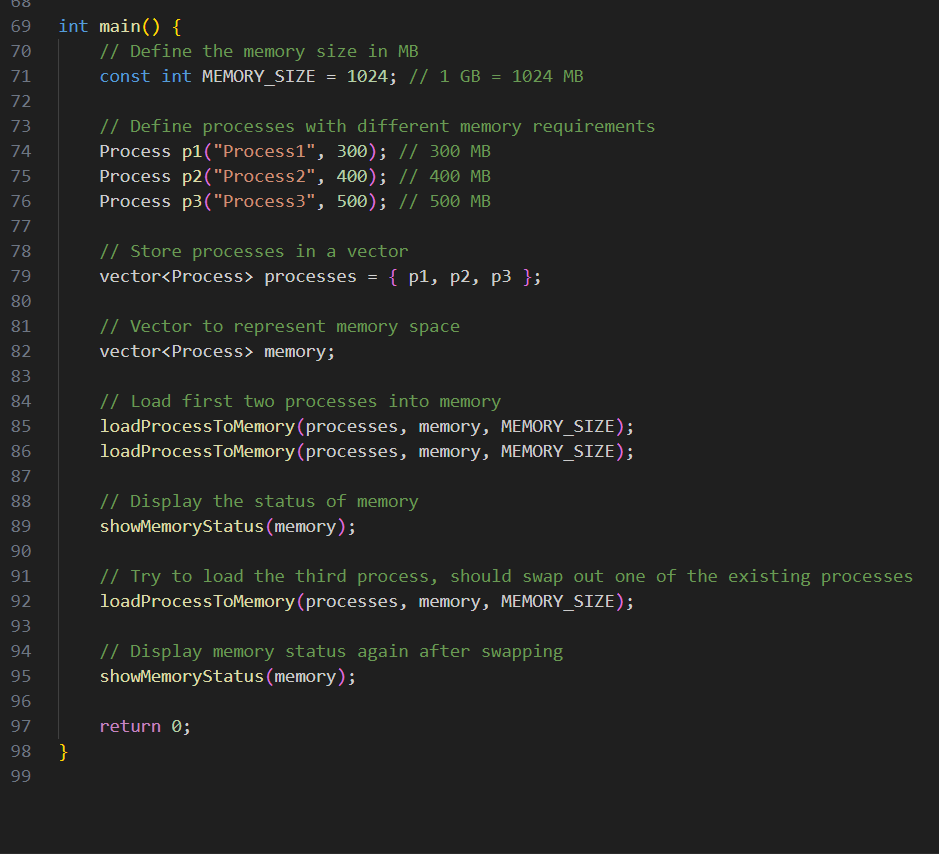
**Example:**

* If you open **too many apps** on your phone, some apps pause in the background.
* When playing a **game**, and you switch to another app, the game might reload when you return.
* On a **computer**, if RAM is full, an inactive program is moved to disk and brought back when needed.

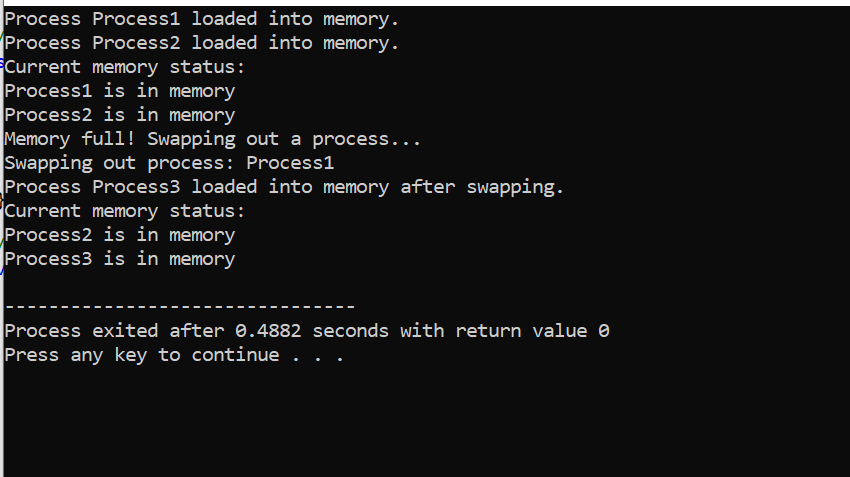
Code for memory swapping







Output:



**Segmentation**

1. Logical to Physical Address Translation

Objective:

The objective is to take a logical address consisting of a segment number and an offset, and convert it into a physical address using segment base addresses and segment sizes.

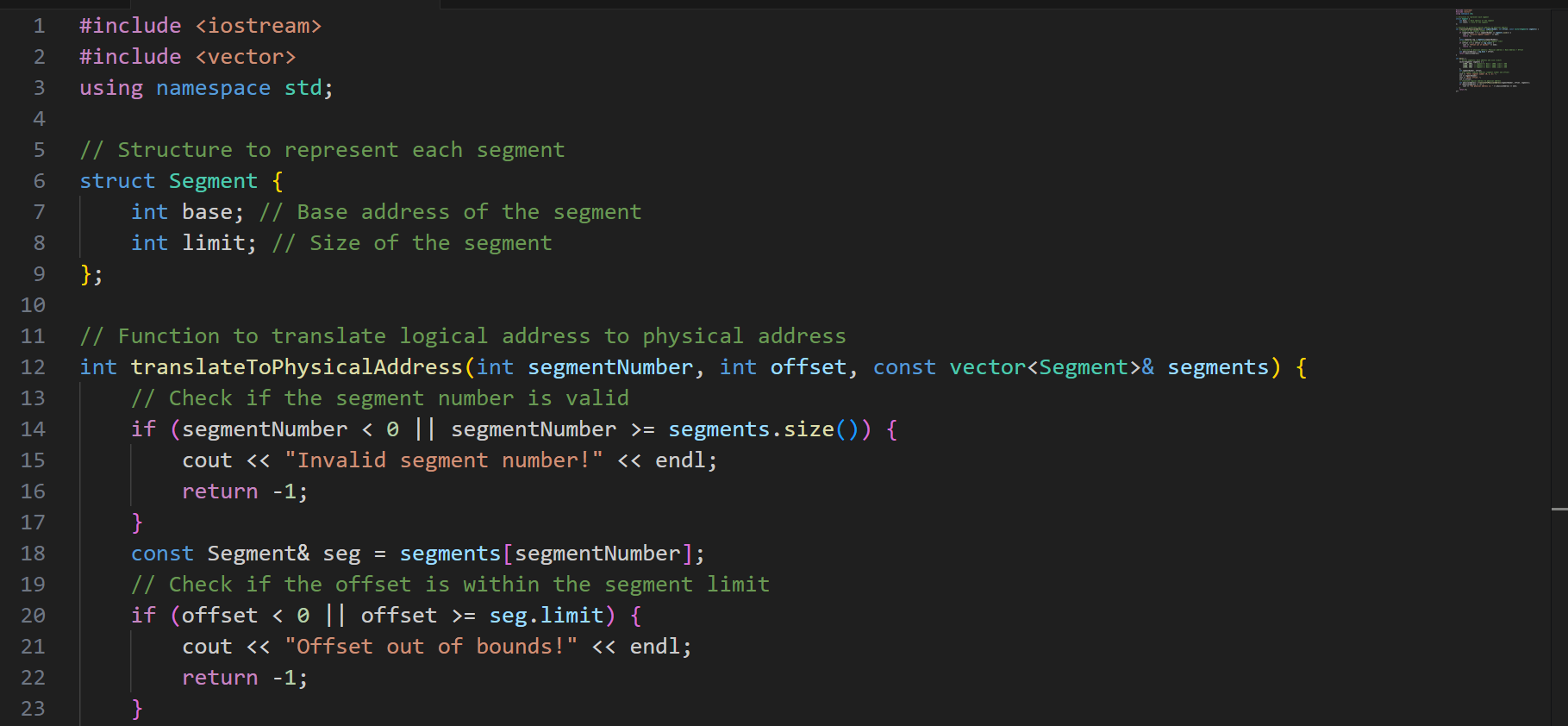
Steps for Logical to Physical Address Translation

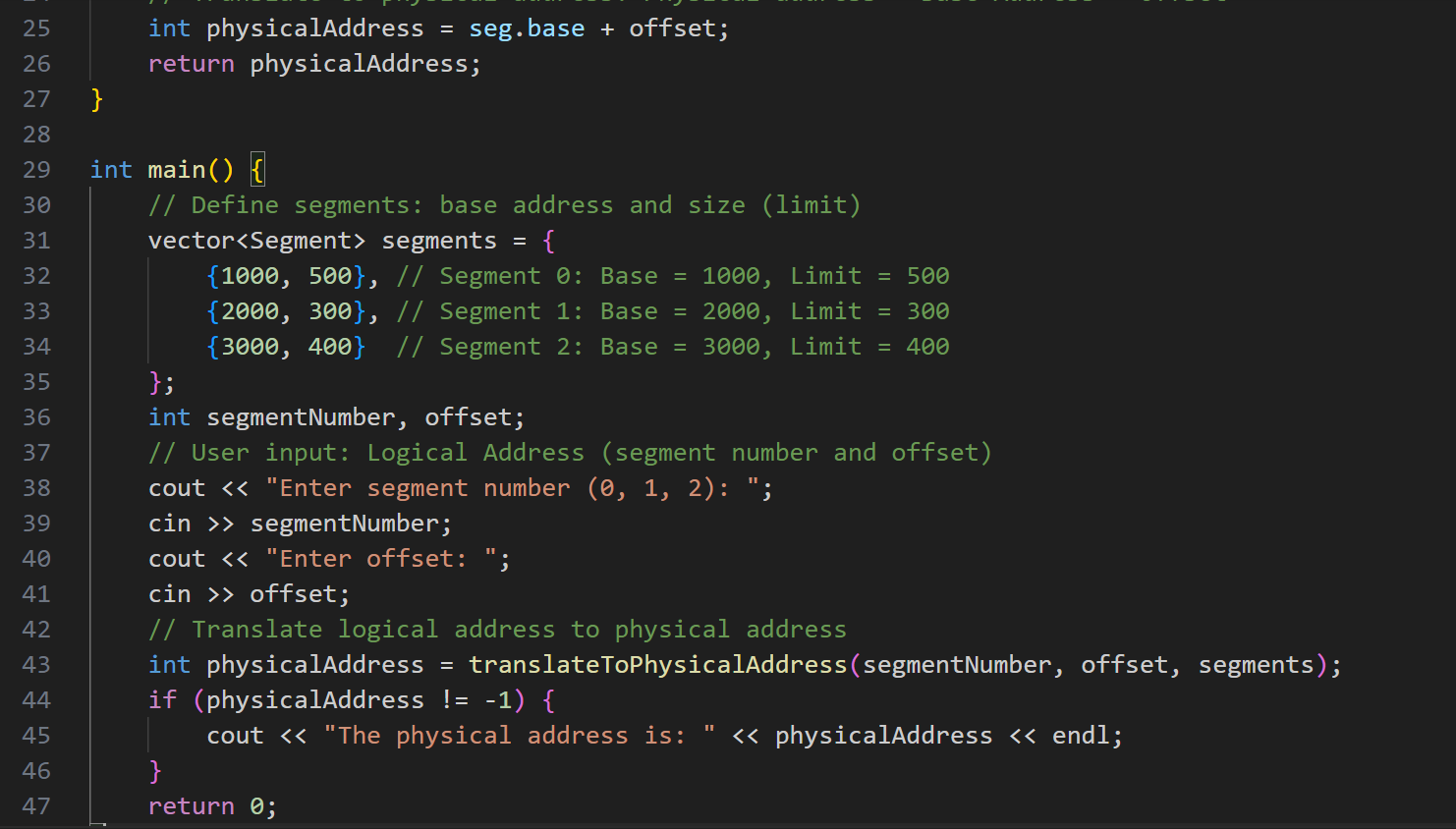
1. Segment Number and Offset: The logical address consists of a segment number and an offset within that segment.

2. Segment Base Address: Each segment has a base address that is the starting point of that segment in physical memory.

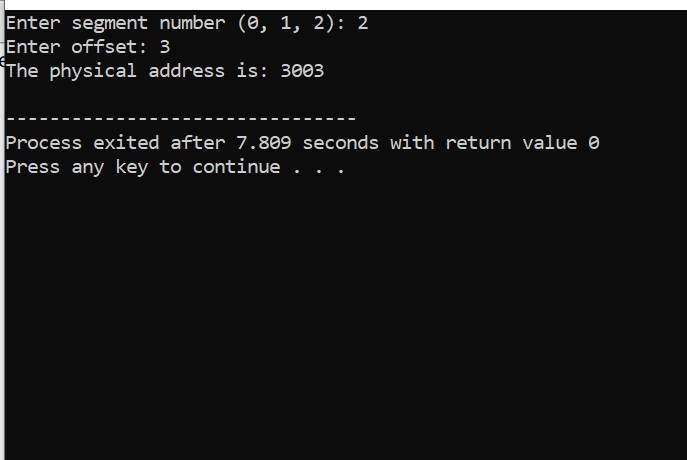
3. Translate: The physical address is calculated by adding the base address of the segment to the offset.

Code implemention:





Output:



Explanation of the Code

1. Segment Structure:

Base:

Represents the base address of the segment in physical memory.

limit:

Represents the size of the segment, i.e., how large the segment is.

2. Translate To Physical Address Function:

This function takes in a segment-number, an offset, and a vector of Segment objects.

It checks if the segment number is valid and if the offset is within the segment's limit.

If both conditions are met, it calculates the physical address by adding the segment's base address to the offset.

3. Input:

The program prompts the user to enter the segment number and offset. These are used to look up the corresponding segment and then calculate the physical address.

Output:

The program outputs the physical address, or an error message if the segment number is invalid or the offset is out of bounds.

Example of Running the Code

Input:

Enter segment number (0, 1, 2): 1

Enter offset: 15

Output:

The physical address is: 2150

In this example, Segment 1 has a base address of 2000, and an offset of 150 is provided. The physical address is calculated as 2000 + 150 = 2150.

Q3.Visualize Segmentation Fragmentation

Objective:

The goal of this task is to simulate memory allocation and deallocation in a segmented memory model, and visualize how fragmentation occurs over time. Fragmentation occurs when free memory blocks are scattered across the system after allocating and deallocating segments, leading to inefficient use of memory.

Concepts Involved:

Memory Segmentation: In a segmented memory system, memory is divided into segments, and each segment has its own base address and size. Each program will have several segments (code, data, stack, etc.), and these segments can be allocated and deallocated dynamically.

Fragmentation:

Fragmentation happens when there are gaps of unused memory after deallocation. These gaps may be too small to use for future allocations but still occupy space. There are two types:

External Fragmentation:

The free memory is fragmented into small blocks, making it difficult to allocate larger blocks of memory.

Internal Fragmentation:

This happens when a segment is allocated a block larger than its actual size, leading to unused space within the allocated segment.

Steps to Simulate Segmentation Fragmentation:

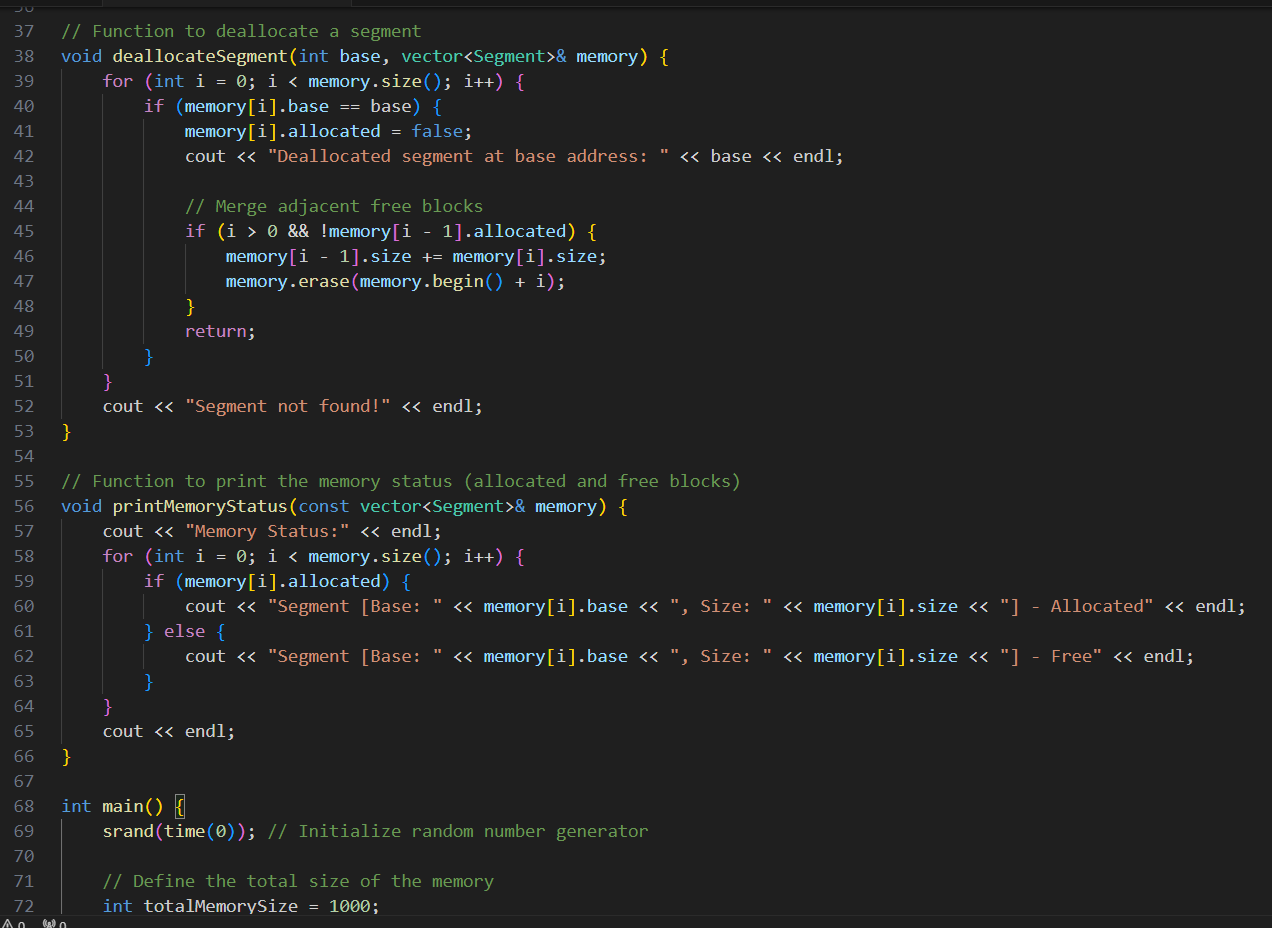
1. Allocate memory for segments by randomly choosing free spots in the memory.

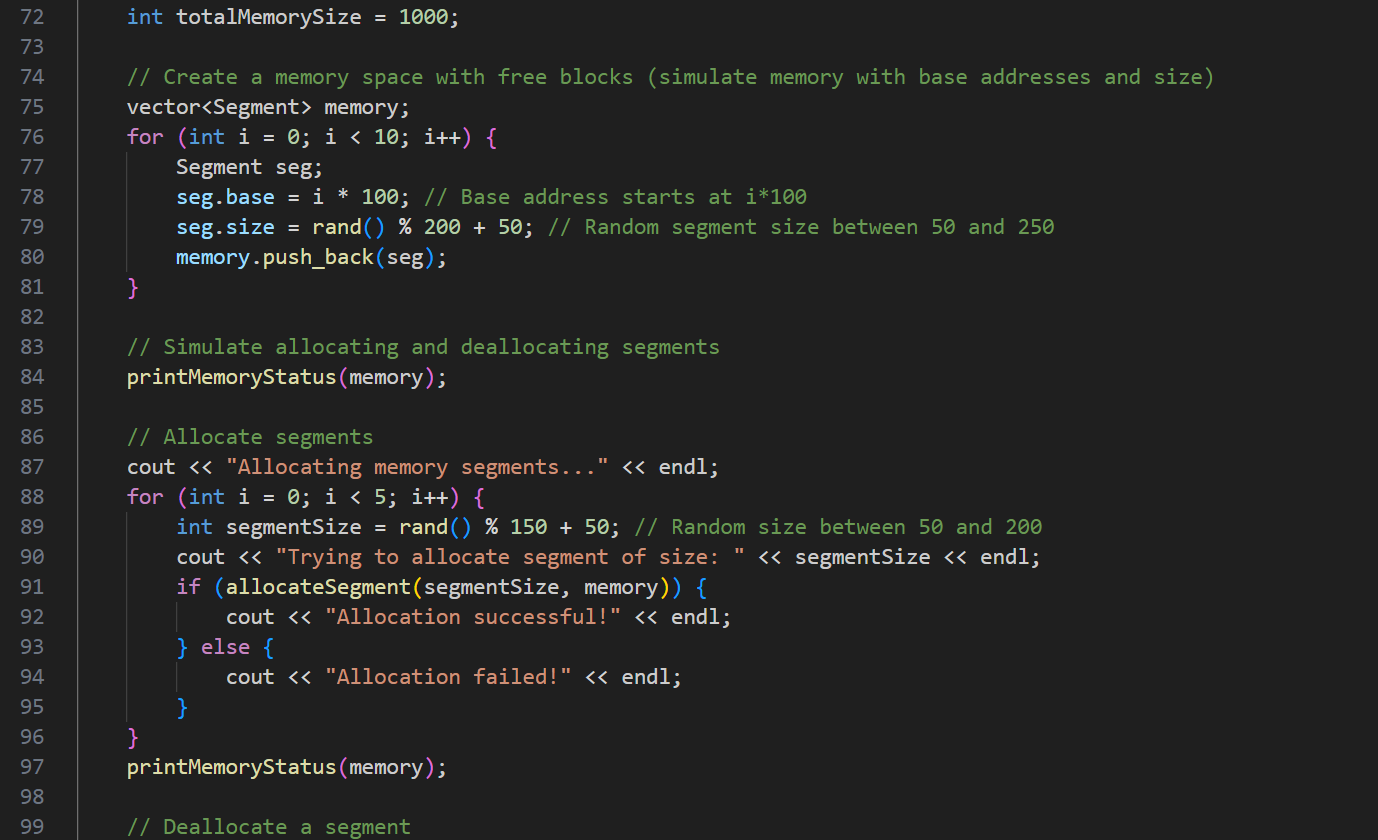
2. Deallocate memory and visualize how the segments are removed, creating holes or gaps.

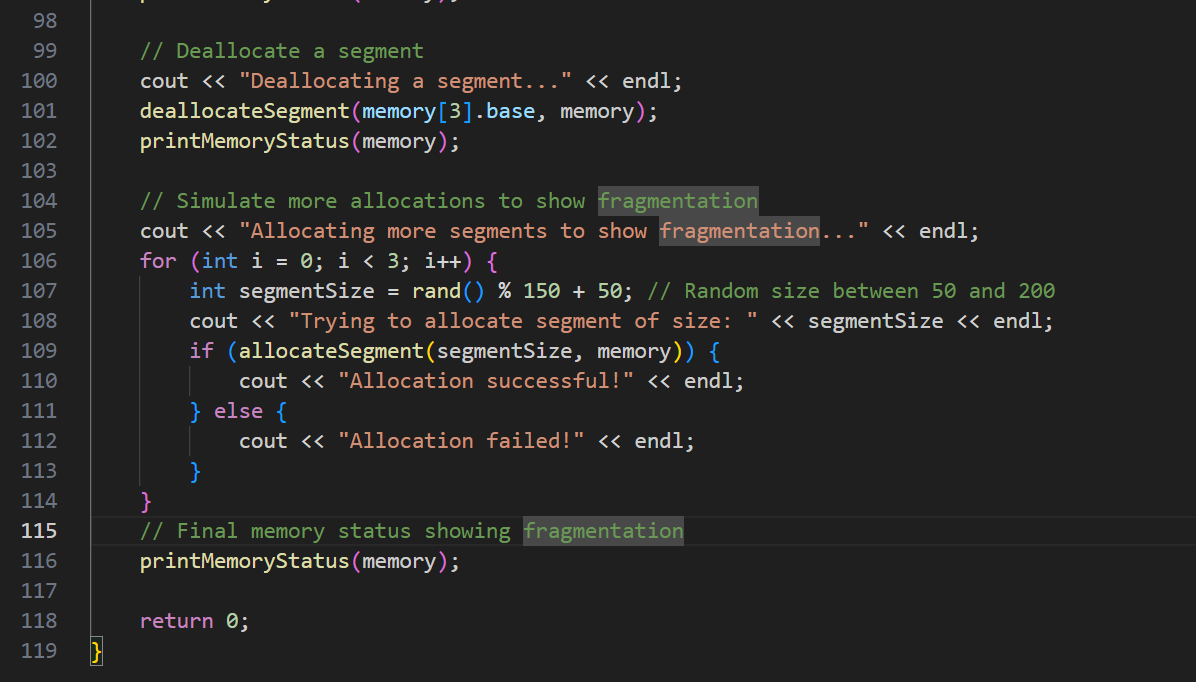
3. Keep track of free and allocated memory to display the fragmentation visually.

Code implementation:

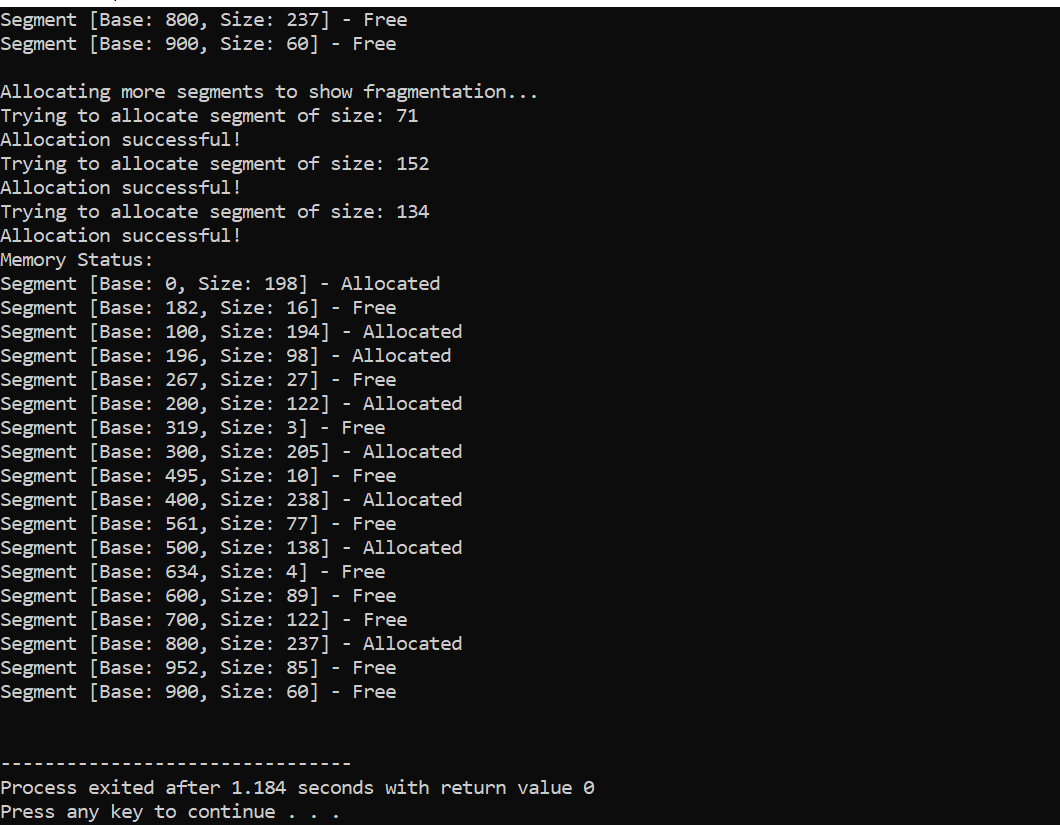








0utput:



Explanation of the Code:

1:Data Structure (Segment):

base: The starting address of the segment.

size: The size of the segment.

allocated: A boolean to check if the segment is allocated or free.

1. Allocate Function (allocated Segment):The function iterates over the memory blocks and tries to find a free block large enough to accommodate the segment size.

Once found, it marks the block as allocated and sets the size.

3.Deallocate Function (deallocate Segment):

This function marks a segment as deallocated based on its base address, freeing up space.

4. Print Memory Status (print Memory Status):

This function prints the current state of the memory, showing which segments are allocated and which are free. It helps to visualize how memory is fragmented over time.

Simulation:

The program first prints the initial state of memory, then randomly allocates segments.

After that, a segment is deallocated, and further segments are allocated to demonstrate fragmentation.

It outputs the state of memory after each operation.

Q:no:04:

**Introduction to Paging:-**

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory, thus avoiding fragmentation. In paging, the logical address space (program's address space) is divided into fixed-size blocks, called pages, and the physical memory is divided into blocks of the same size, called frames.

The operating system maintains a page table to keep track of the mapping between the logical address space (pages) and the physical memory (frames). The logical address generated by the CPU is divided into two parts: the page number and the offset within the page. The page table stores the base address of each page in the physical memory.

2. Define a Page Size and Logical Address Space:

Page Size:

The page size determines the size of each page in memory. A common page size is 4 KB (kilobytes). The page size influences the efficiency of memory management because smaller pages reduce internal fragmentation but may lead to larger page tables, while larger pages reduce the page table size but may lead to wasted space due to internal fragmentation.

For this simulation, let's assume the page size is 4 KB. This means that each page will be 4 KB in size.

Logical Address Space:

The logical address space represents the addresses that the program can generate, which is typically based on the CPU architecture. For example, a 32-bit address space can address up to \(2^{32}\) bytes, i.e., 4 GB of memory.

Condider: we have a 32-bit logical address space. This means the program can generate addresses from 0 to \(2^{32} - 1\) (i.e., from 0 to 4 GB).

Number of Pages:

The number of pages can be calculated using the formula:

Creating the Page Table:

What is a Page Table?

The page table is a data structure used by the operating system to manage the mapping between the logical pages (from the process's address space) and physical frames in memory. The page table contains an entry for each page, and each entry holds the physical address of the corresponding frame.

Page Table Structure:

For a 32-bit logical address space, with 4 KB pages, the number of frames in the physical memory would depend on the size of physical memory available. Let's assume physical memory of 8 GB (which is 2^33 bytes). The number of frames is calculated as:

\[

\text{Number of Frames} = \frac{\text{Physical Memory Size}}{\text{Page Size}} = \frac{8 \text{ GB}}{4 \text{ KB}} = 2^{21} \text{ frames}

\]

Each page will map to a frame, so the page table will have 2^20 entries, each pointing to one of the 2^21 frames in physical memory.

Each entry in the page table will store the frame number (which is a reference to the physical frame). For simplicity, we assume each frame number fits into a 21-bit address space (for 2^21 frames).

Example of a Page Table Entry:

For each logical page, we have a page table entry that stores the frame number (physical address). An entry might look like this:

|  |  |
| --- | --- |
| Logical Page Number | Frame Number |
| 0 | 1001 |
| 1 | 2003 |
| 2 | 1507 |

Translating Logical Addresses to Physical Addresses:

Structure of a Logical Address:

A logical address is divided into two parts:

Page Number: The higher-order bits represent the page number.

Offset: The lower-order bits represent the offset within the page.

For example: given a 32-bit logical address and 4 KB pages:

Page size = 4 KB = \(2^{12}\) bytes.

The lower 12 bits of the logical address are the offset within the page.

The upper 20 bits are the page number.

Step-by-Step Address Translation:

To translate a logical address to a physical address:

Extract the page number from the logical address. The number of bits for the page number is equal to the total number of bits in the address minus the number of bits for the offset. In this case, it’s 20 bits for the page number and 12 bits for the offset.

Look up the page number in the page table to find the corresponding frame number.

Combine the frame number with the offset to form the physical address. The offset remains the same.

For example, if we have a logical address of 0x00123456, we can break it down into:

Page Number: The upper 20 bits (from 0x00123).

Offset: The lower 12 bits (0x456).

We then look up the page table entry for page number 0x00123. Suppose the frame number for this page is 0x100. We combine the frame number 0x100 with the offset 0x456, so the physical address will be 0x100456.

Example Simulation of Paging

Sample Address Translation

consider:

Logical address = 0x00123456 (32-bit address)

Page size = 4 KB (12 bits for offset)

Page table entry for page 0x00123 = frame 0x100

We break down the address:

Page number = 0x00123 (the upper 20 bits)

Offset = 0x456 (the lower 12 bits)

After looking up the page table, the frame number for page 0x00123 is 0x100.

So, the physical address is:

\[

\text{Physical Address} = \text{Frame Number} \, \| \, \text{Offset} = 0x100 \, \| \, 0x456 = 0x100456

\]

Thus, the logical address 0x00123456 translates to the physical address 0x100456.

Conclusion:

Paging allows efficient use of memory by eliminating fragmentation and enabling non-contiguous allocation. The page table acts as a crucial element in translating logical addresses to physical addresses. This simulation demonstrates how a logical address is translated to a physical address using a page table.

Through paging, the system achieves a dynamic and flexible memory management scheme, optimizing memory usage and providing a secure and isolated address space for each running process.Q:05: Page Replacement Algorithms:-

FIFO (First In, First Out) Page Replacement Algorithm:-

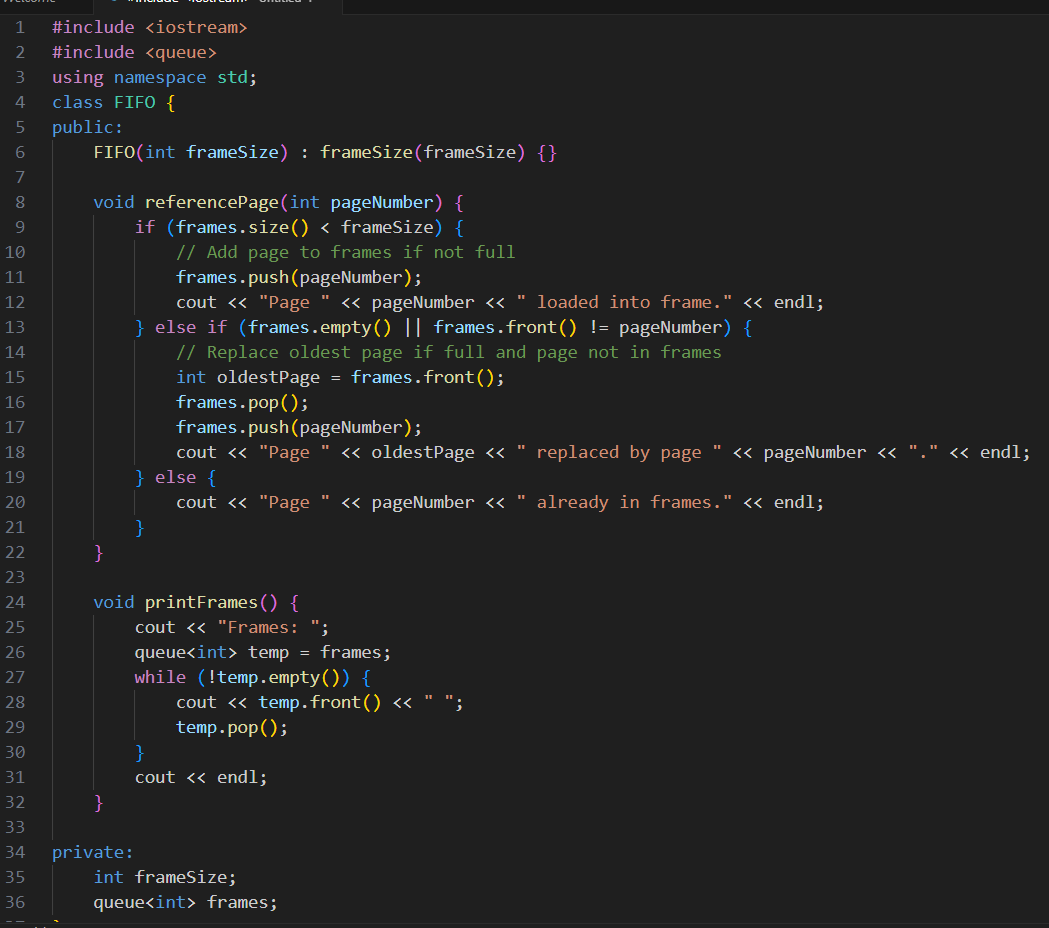
FIFO is one of the simplest page replacement algorithms, which follows the principle that the oldest page in memory is replaced when a new page needs to be loaded, and the memory is full. Here's how it works:

1. Pages are loaded into a set number of frames (memory slots).

2. When a new page needs to be loaded and all frames are occupied, the oldest page (the one that has been in memory the longest) is replaced with the new page.

3. If the page is already in memory (frame), there is no change.

Code or algorithm of FIFO:



|  |  |  |
| --- | --- | --- |
| Description: page 2 | Output:  Description: 1 | Output:  Description: 2 |

Explanation of FIFO Page Replacement:

The FIFO algorithm works as follows:

A set number of frames are available in memory.

Pages are loaded into memory, and when all frames are full, the oldest page (the one that was first loaded) is replaced by a new page.

In your code, pages are represented as integers, and the queue (FIFO) data structure is used to maintain the order of pages.

FIFO Behavior:

1. When a page reference occurs:

If the page is already in memory, no action is taken.

If the page isn't in memory and there is space, it is added.

If the page isn't in memory and there’s no space, the oldest page is removed and replaced with the new page.

1. The program allows the user to interactively simulate this behavior and observe how memory changes over time.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_