LAB 7 : Memory Management Scheme in OS

Nirajan Bekoju
PUL076BCT039
076bct039.nirajan@pcampus.edu.np

Nishant Luitel
PUL076BCT041
076bct041.nishant@pcampus.edu.np

Nabin Da Shrestha
PUL076BCT037
076bct037.nabin@pcampus.edu.np

Prakash Chaulagain
PUL076BCT045
076bct045.prakash@pcampus.edu.np

I. Introduction

A. Physical and Logical Memory

Physical memory, also known as primary memory or RAM (Random Access Memory), is the actual hardware component of a computer that stores data and instructions that the processor is actively using or processing. It is a volatile memory, meaning that its contents are lost when the computer is shut down or loses power. Physical memory is typically measured in gigabytes (GB) or terabytes (TB) and is installed on the computer's motherboard.
On the other hand, logical memory is a virtual memory space created by the operating system to manage the data and programs that the processor uses. Logical memory is not limited to the amount of physical memory installed on the computer, and it can be much larger than the physical memory available. The logical memory is divided into pages, which are mapped to physical memory frames using the paging technique.
When a program is executed, its instructions and data are loaded into logical memory, which the processor accesses to perform the required operations. The operating system manages the mapping of logical memory pages to physical memory frames, ensuring that the required pages are loaded into physical memory when needed and swapping out pages that are no longer needed.
In summary, physical memory refers to the actual memory chips installed in the computer, while logical memory is a virtual memory space created by the operating system to manage the data and programs that the processor uses.

B. Paging

Paging is a memory management scheme used by operating systems to manage the memory used by processes. In this scheme, the physical memory is divided into fixed-size blocks called frames, and the logical memory used by a process is divided into fixed-size blocks called pages.
Each page in the logical memory is mapped to a frame in the physical memory. The mapping is maintained by a page table, which is a data structure that stores the mapping information for each page. The page table is maintained by the operating system and is used to translate virtual addresses used by the process into physical addresses used by the memory hardware.
When a process needs to access a page that is not currently in physical memory, a page fault occurs. The operating system then retrieves the page from the secondary storage, such as a hard disk, and stores it in an available frame in physical memory. The page table is updated to reflect the new mapping, and the process can continue executing.
Paging allows for efficient use of physical memory by only loading the necessary pages into memory when they are needed. It also allows for easy sharing of memory between processes, as multiple processes can map to the same physical frame. However, it can also introduce overhead due to the need for frequent page table updates and the potential for page faults.
Overall, paging is a widely used memory management scheme that provides a flexible and efficient way to manage memory in modern operating systems.

C. Segmentation

Segmentation is a memory management scheme that divides the physical memory into logical segments, each with its own address space. A segment is a contiguous block of memory that stores a particular type of data, such as a program code segment or a data segment. Each segment is assigned a segment number, which serves as an index to the segment table.
The segment table is a data structure that maps each segment number to a base address and a limit. The base address is the starting address of the segment in physical memory, while the limit is the size of the segment. When a program references a memory location, the memory management unit (MMU) uses the segment number and the offset within the segment to calculate the physical address of the memory location.
Here's an example of how segmentation works:

Suppose a program consists of three logical segments: code, data, and stack. The code segment contains the program's executable code, the data segment contains the program's variables, and the stack segment contains the program's runtime stack.
The operating system assigns each segment a segment number, a base address, and a limit. Let's say the code segment has segment number 0, base address 0x00000000, and limit 0x00001000 (4KB); the data segment has segment number 1, base address 0x00100000, and limit 0x00002000 (8KB); and the stack segment has segment number 2, base address 0x00200000, and limit 0x00001000 (4KB).
When the program references a memory location, the MMU first checks the segment number of the memory location. If the segment number is valid (i.e., it corresponds to a segment in the segment table), the MMU adds the offset within the segment to the base address of the segment to calculate the physical address of the memory location. For example, if the program wants to access the memory location at offset 0x00000300 within the data segment, the MMU would calculate the physical address as follows:

```
physical address = base address + offset
                 = 0x00100000 + 0x00000300
                 = 0x00100300
```

The MMU checks that the resulting physical address is within the limit of the segment before allowing the program to access the memory location.
Segmentation allows for more flexible memory allocation than paging, as each segment can be assigned a size that corresponds to the amount of memory needed for a particular type of data. However, segmentation can lead to fragmentation and requires more complex hardware support than paging.

II. Memory Management Scheme - Paging

Aim : To write a C program to implement memory management using paging technique.
Algorithm :

```
Step 1 : Start the program.
Step 2 : Read the base address, page size, number of pages and memory unit.
Step 3 : If the memory limit is less than the base address display the memory limit is less than limit.
Step 4 : Create the page table with the number of pages and page address.
Step 5 : Read the page number and displacement value.
Step 6 : If the page number and displacement value is valid, add the displacement value with the address corresponding to the page number and display the result.
Step 7 : Display the page is not found or displacement should be less than page size.
Step 8 : Stop the program.
```

Program :

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

int main()
{
    int base_address, page_size, num_pages, memory_limit;

    // Step 2: Read inputs
    cout << "Enter base address: ";
    cin >> base_address;

    cout << "Enter page size: ";
    cin >> page_size;

    cout << "Enter number of pages: ";
    cin >> num_pages;

    cout << "Enter memory limit: ";
    cin >> memory_limit;

    // Step 3: Check memory limit
    if (memory_limit < base_address)
    {
        cout << "Memory limit is less than base address.\n";
        return 0;
    }

    // Step 4: Create page table
    int *page_table = new int[num_pages];
    for (int i = 0; i < num_pages; i++)
    {
        page_table[i] = base_address + i * page_size;
    }

    // Step 5-7: Read and validate page number and displacement
    int page_num, displacement;
    while (true)
    {
        cout << "Enter page number (-1 to exit): ";
        cin >> page_num;
        if (page_num == -1)
        {
            break;
        }
        cout << "Enter displacement: ";
        cin >> displacement;
        if (page_num < 0 || page_num >= num_pages)
        {
            cout << "Error :: Page number is not valid.
                    Page number ranges from 0 to number of pages - 1\n";
        }
        else if (displacement < 0 || displacement >= page_size)
        {
            cout << "Displacement should be less than page size.\n";
        }
        else
        {
            int physical_address = page_table[page_num] + displacement;
            if (physical_address < memory_limit)
            {
                cout << "Physical address: " << physical_address << endl;
            }
            else
            {
                cout << "Error :: Physical address > memory limit.\n";
            }
        }
    }

    // Step 8: Clean up and exit
    delete[] page_table;
    return 0;
}
```

Ouput :

```
    $ g++ -o paging paging.cpp
    $ ./paging
    Enter base address: 1000
    Enter page size: 256
    Enter number of pages: 256
    Enter memory limit: 65536
```

```
Enter page number (-1 to exit): 3
Enter displacement: 200
Physical address: 1968
Enter page number (-1 to exit): 2
Enter displacement: 200
Physical address: 1712
Enter page number (-1 to exit): -1
```

III. Memory Management Scheme - Segmentation

Aim : To write a C program to implement memory management using segmentation.

Algorithm :

Step 1 : Start the program.
Step 2 : Read the base address, number of segments, size of each segment, memory limit.
Step 3 : If memory address is less than the base address display “invalid memory limit”.
Step 4 : Create the segment table with the segment number and segment address and display it.
Step 5 : Read the segment number and displacement.
Step 6 : If the segment number and displacement is valid compute the real address and display the same.
Step 7 : Stop the program.

Program :

```
#include <stdio.h>

int main() {
    int base_addr, num_segments, segment_size, memory_limit;
    int i, seg_num, displacement, real_addr;

    // Step 2: read inputs
    printf("Enter base address: ");
    scanf("%d", &base_addr);
    printf("Enter number of segments: ");
    scanf("%d", &num_segments);
    printf("Enter size of each segment: ");
    scanf("%d", &segment_size);
    printf("Enter memory limit: ");
    scanf("%d", &memory_limit);

    // Step 3: check memory limit
    if (memory_limit < base_addr) {
        printf("Invalid memory limit.\n");
        return 0;
    }

    // Step 4: create segment table
    printf("Segment Table:\n");
    printf("--------------\n");
    for (i = 0; i < num_segments; i++) {
        printf("Segment %d: Address %d\n", i, base_addr + i * segment_size);
    }

    // Step 5: read segment number and displacement
    printf("Enter segment number: ");
    scanf("%d", &seg_num);
    printf("Enter displacement: ");
    scanf("%d", &displacement);

    // Step 6: compute real address
    if (seg_num >= num_segments || displacement >= segment_size) {
        printf("Invalid segment number or displacement.\n");
    } else {
        real_addr = base_addr + seg_num * segment_size + displacement;
        printf("Real address = %d\n", real_addr);
    }

    // Step 7: end program
    return 0;
}
```

Ouput :

```
$ g++ -o segmentation segmentation.cpp
$ ./segmentation
Enter base address: 1000
Enter number of segments: 3
Enter size of each segment: 2048
Enter memory limit: 10000
Segment Table:
--------------
Segment 0: Address 1000
Segment 1: Address 3048
Segment 2: Address 5096
Enter segment number: 2
Enter displacement: 500
Real address = 5596
```

IV. Conclusion

In this lab, we have implemented two important memory management schemes, namely paging and segmentation, using C programming language.

In the paging technique, the main memory is divided into equal-sized pages and each page is allocated to the corresponding process. This allows for efficient use of memory and reduces external fragmentation. We have created a page table to map logical addresses to physical addresses and have demonstrated the process of retrieving the physical address using the page number and displacement value.

In the segmentation technique, the main memory is divided into variable-sized segments, each representing a logical unit of the program, such as a function, a procedure or a data structure. We have created a segment table to map logical addresses to physical addresses and have demonstrated the process of computing the real address using the segment number and displacement value.

Overall, these memory management techniques are crucial for efficient use of the limited physical memory available to the operating system. By implementing them using C programming, we have gained a better understanding of how they work and how they can be applied in real-world scenarios.