OPERATING SYSTEMS - UNIT 4 – CHAPTER 1

Topic 1: Levels of Memory Management

Main memory is generally the most critical resource in a computer system in terms of the speed at which programs run, and it is important to manage it as efficiently as possible. The operating system is responsible for allocating memory to process.

The Memory manager is the subsystem of an operating system in charge of allocating large blocks of memory to processes. Each process gets from the operating system, a block of memory to use, but the process itself handles the internal management of that memory. Each process has memory allocator for that process.

The Memory allocators do not call the operating system each time they get a memory request. Instead, they allocate space from a large block of free memory that is allocated to the process by the operating system. When the per-process memory allocator runs out of the memory to allocate, it will ask for another large chunk of memory from operating system.

Topic 2: Linking and loading of a Process

The initialization sets up the memory pool that is allocated from by the per-process memory manager. A program must go through two major steps before it can be loaded into memory for execution. First, the source code is converted into a load module. Second, when a process is started, the load module is loaded from disk into memory.

2.1 Creating a Load Module:

A **load module** is an executable program stored in a partitioned data set program library. Creating a load module to execute only, will require that you use a batch loader or program management loader. Creating a load module that can be stored in a program library requires that you use the binder or linkage editor. In all cases, the load module is relocatable, which means that it can be located at any address in virtual storage within the confines of the residency mode (**RMODE**).

Once a program is loaded, control is passed to it, with a value in the base register. This gives the program its starting address, where it was loaded, so that all addresses can be resolved as the sum of the base plus the offset. Relocatable programs allow an identical copy of a program to be loaded in many different address spaces, each being loaded at a different starting address.

- **Batch loader**
  The **batch loader** combines the basic editing and loading services (which can also be provided by the linkage editor and program manager) into one job step. The batch loader accepts object decks and load modules, and loads them into virtual storage for execution. Unlike the binder and linkage editor, the batch loader does not produce load modules that can be stored in program libraries. The batch loader prepares the executable program in storage and passes control to it directly.

- **Program management loader**
  The program management loader increases the services of the program manager component by adding support for loading program objects. The loader reads both program objects and load modules into virtual storage and prepares them for execution. It resolves any address constants in the program to point to the appropriate areas in virtual storage and supports the 24-bit, 31-bit and 64-bit addressing modes.
- **Load library**
  A **load library** contains programs ready to be executed.

2.2 Loading Load Module:

When a program is executed, the operating system allocates the memory to the process and then loads the load module into the memory allocated to the process. The executable code and initialized data are copied into the processor's memory from the load module. Additional chunks of memory are also allocated for both uninitialized data and for stack for use by the programming languages.

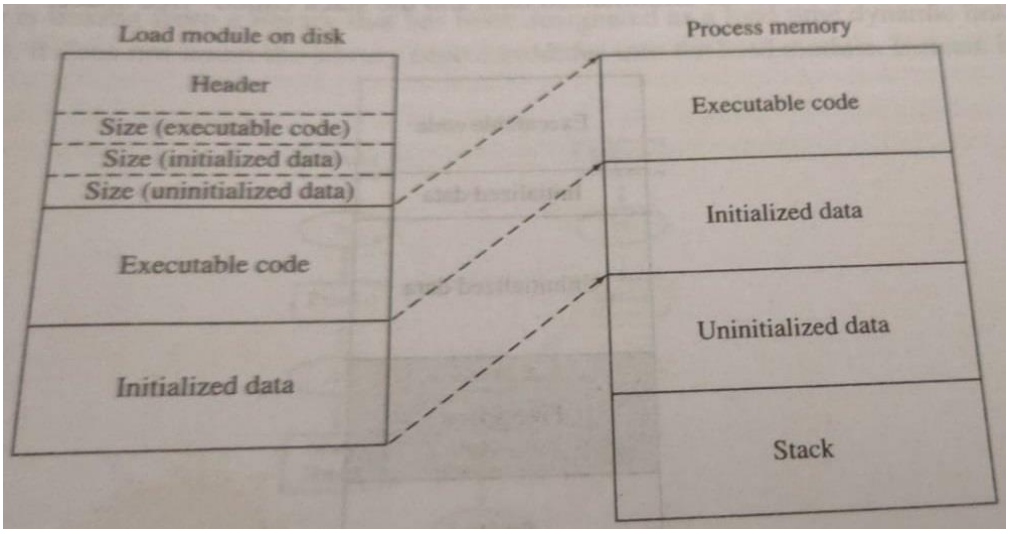


Fig. Loading a Program into a Process

Topic 3: Variations in Program Loading:

A whole program usually is not written in a single file. Apart from code and data definitions in multiple files, a user code often makes references to code and data defined in some "libraries". Linking is the process in which references to "externally" defined objects (code and data) are processed so as to make them operational. Traditionally linking used to be performed as a task after basic translation of the user program files, and the output of this stage is a single executable program file. This is known as static linking. A more versatile technique is more commonly used these days which is called - dynamic linking.

Static linking: - In static linking all the modules that are required to complete a program are physically placed together to generate a executable program file. The file can then be "loaded" at any subsequent time to run the program.

Dynamic linking:- In dynamic linking the actual task of linking is performed just prior to running the program and with the individual modules actually in the memory. This approach has several advantages over static linking as listed below:

- A single copy of an object module in the memory may form part of the execution image of several programs, thus reducing overall memory requirement in the system.
- Size of the executable program files remain small, since the component modules are not physically copied in that file.
- Actual requirement of having individual modules in a program may be determined at run time of a program and linking can be done accordingly. This happens when a program takes a particular course of execution involving certain object modules, depending on some run-time conditions, such as user options. Dynamic execution can allow a program to control the choice of modules to be linked in a particular run.

Dynamic linking based on the application type and requirement of memory size can be used in two variations:

1. **Runtime Dynamic Linking:** Run-time dynamic linking enables the process to continue running even if a DLL is not available. The process can then use an alternate method to accomplish its objective. For example, if a process is unable to locate one DLL, it can try to use another, or it can notify the user of an error. If the user can provide the full path of the missing DLL, the process can use this information to load the DLL even though it is not in the normal search path. This situation contrasts with load-time linking, in which the system simply terminates the process if it cannot find the DLL. Run-time dynamic linking can cause problems if the DLL uses the **DllMain** function to perform initialization for each thread of a process, because the entry-point is not called for threads that existed before **LoadLibrary** or **LoadLibraryEx** is called.
2. **Load time Dynamic Linking:** When the system starts a program that uses load-time dynamic linking, it uses the information the linker placed in the file to locate the names of the DLLs that are used by the process. The system then searches for the DLLs. For more information, see Dynamic-Link Library Search Order. If the system cannot locate a required DLL, it terminates the process and displays a dialog box that reports the error to the user. Otherwise, the system maps the DLL into the virtual address space of the process and increments the DLL reference count. The system calls the entry-point function. The function receives a code indicating that the process is loading the DLL. If the entry-point function does not return TRUE, the system terminates the process and reports the error.

OPERATING SYSTEMS - UNIT 4 – CHAPTER 2

Topic 1 : Fragmentation and Compaction

Fragmentation is an unwanted problem in the operating system in which the processes are loaded and unloaded from memory, and free memory space is fragmented. Processes can't be assigned to memory blocks due to their small size, and the memory blocks stay unused. It is also necessary to understand that as programs are loaded and deleted from memory, they generate free space or a hole in the memory. These small blocks cannot be allotted to new arriving processes, resulting in inefficient memory use.

User processes are loaded and unloaded from the main memory, and processes are kept in memory blocks in the main memory. Many spaces remain after process loading and swapping that another process cannot load due to their size. Main memory is available, but its space is insufficient to load another process because of the dynamical allocation of main memory processes.

There are mainly two types of fragmentation in the operating system. These are as follows:

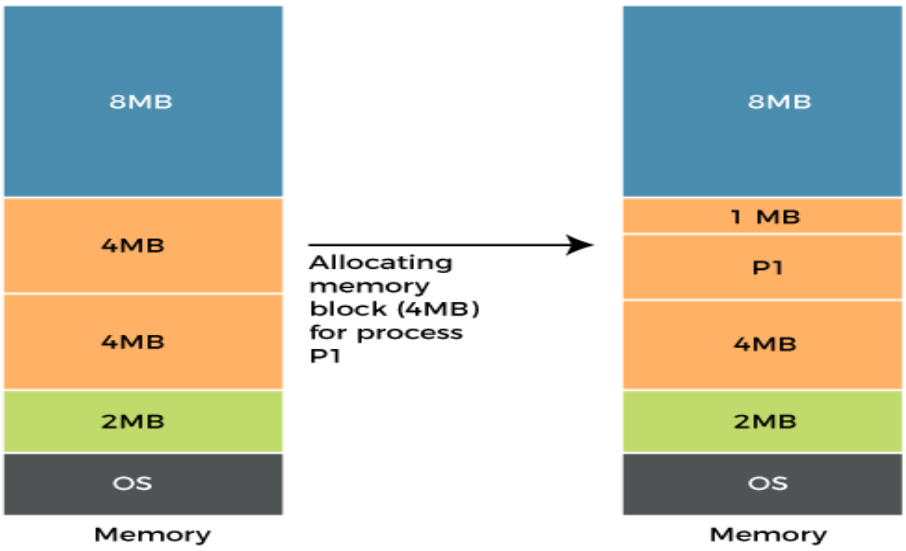
1. **Internal Fragmentation**
2. **External Fragmentation**

Internal Fragmentation

When a process is allocated to a memory block, and if the process is smaller than the amount of memory requested, a free space is created in the given memory block. Due to this, the free space of the memory block is unused, which causes **internal** fragmentation.

Example:

Assume that memory allocation in RAM is done using fixed partitioning (i.e., memory blocks of fixed sizes). **2MB, 4MB, 4MB**, and **8MB** are the available sizes. The Operating System uses a part of this RAM.



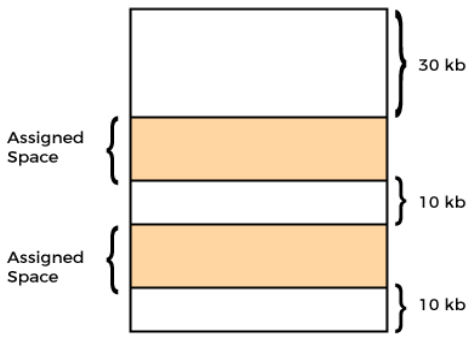
Let's suppose a process **P1** with a size of **3MB** arrives and is given a memory block of **4MB**. As a result, the **1MB** of free space in this block is unused and cannot be used to allocate memory to another process. It is known as **internal fragmentation**.

The problem of internal fragmentation may arise due to the fixed sizes of the memory blocks. It may be solved by assigning space to the process via dynamic partitioning. Dynamic partitioning allocates only the amount of space requested by the process. As a result, there is no internal fragmentation.

External fragmentation

External fragmentation happens when a dynamic memory allocation method allocates some memory but leaves a small amount of memory unusable. The quantity of available memory is substantially reduced if there is too much external fragmentation. There is enough memory space to complete a request, but it is not contiguous. It's known as **external** fragmentation.

For Example:



Process 05 needs 45kb memory space

Let's take the example of external fragmentation. In the above diagram, you can see that there is sufficient space **(50 KB)** to run a process **(05) (need 45KB)**, but the memory is not contiguous. You can use compaction, paging, and segmentation to use the free space to execute a process.

This problem occurs when you allocate RAM to processes continuously. It is done in paging and segmentation, where memory is allocated to processes non-contiguously. As a result, if you remove this condition, external fragmentation may be decreased.

Compaction : Compaction is another method for removing external fragmentation. External fragmentation may be decreased when dynamic partitioning is used for memory allocation by combining all free memory into a single large block. The larger memory block is used to allocate space based on the requirements of the new processes. This method is also known as defragmentation.

Topic 2: Dealing with Fragmentation

In dynamically allocated memory systems, the fragmentation is inevitable. There are several ways with which we can be able to control the loss due to fragmentation. Some of the techniques are discussed below:

1. Separate Code and Data Spaces

The processor knows when it is fetching an instruction and when it is fetching data. So, the processor can use two relocation registers, one for code and one for data addresses. This allows us to put the code and data parts of a program into two different blocks of physical memory. Hence, it will be easier to find blocks that are large enough.

2. Segments

Segmentation is **a memory management technique in which the memory is divided into the variable size parts**. Each part is known as a segment which can be allocated to a process. The details about each segment are stored in a table called a segment table. A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments. Segmentation gives user's view of the process which paging does not give. Here the user's view is mapped to physical memory.
There are types of segmentation:

Virtual memory segmentation –Each process is divided into a number of segments, not all of which are resident at any one point in time.

Simple segmentation –Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

Segment Table – It maps two-dimensional Logical address into one-dimensional Physical address. It's each table entry has:

- **Base Address:** It contains the starting physical address where the segments reside in memory.
- **Limit:** It specifies the length of the segment.

3. Page tables in hardware registers

A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the program executed by the accessing process, while physical addresses are used by the hardware, or more specifically, by the random-access memory (RAM) subsystem. The page table is a key component of virtual address translation that is necessary to access data in memory.

The memory management unit (MMU) inside the CPU stores a cache of recently used mappings from the operating system's page table. This is called the translation lookaside buffer (TLB), which is an associative cache.

When a virtual address needs to be translated into a physical address, the TLB is searched first. If a match is found, which is known as a *TLB hit*, the physical address is returned and memory access can continue. However, if there is no match, which is called a *TLB miss*, the MMU or the operating system's TLB miss handler will typically look up the address mapping in the page table to see whether a mapping exists, which is called a *page walk*. If one exists, it is written back to the TLB, which must be done because the hardware accesses memory through the TLB in a virtual memory system, and the faulting instruction is restarted, which may happen in parallel as well. The subsequent translation will result in a TLB hit, and the memory access will continue.

(Also read paging concept in 5th Unit for more Paging techniques)