UNIT III

Memory management

* Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.

* Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.

Basic Memory Concepts

- This is a very brief summary of the memory basics. **Virtual memory** is Physical memory (RAM) + Disk Page file (also called the swap file). It is shared by all processes running on the machine including the operating system.
- The *user address space* (UAS) is independent of virtual memory and is of finite size. It is measured per process on the machine. By definition, in a 32-bit operating system, virtual bytes is limited to 4GB (2^{32}). The 32-bit Windows operating system divides this into two parts UAS and System Address Space (SAS). The UAS is, in this case, for Intelligence Server to store data and code, and the SAS is for the operating system's use.

Private bytes reflect virtual memory usage, and they are a subset of allocated virtual bytes.

To help determine what is causing memory depletion, answer these questions:

- How is the project being used? Are there very complex reports? Reporting on very large data sets?
- Is the Scheduler being used heavily? Are many reports running on a single schedule? Is there a schedule that runs during the peak usage time?
- What is the prompted/non-prompted report mix?
- How is the History List used? Are there many messages?
- Is there high user concurrency?
- Is there high job concurrency (either jobs or large reports)?
- Are the governor settings too high for working set, XML cells, result set, and so on?

To answer these questions, you must be familiar with the system and how it is being used. But perhaps most useful is to know what the system was doing when the memory depletion occurred. To answer this question, use:

- The Windows Performance Monitor to characterize memory use over time. Examine it in relation to job and user concurrency. Typically, log these counters:
    - Process / Virtual Bytes (MSTRSVR process)
    - Process / Private Bytes (MSTRSVR process)
    - Process / Thread Count (MSTRSVR process)
    - MicroStrategy Server Jobs / Executing Reports
    - MicroStrategy Server Users / Open Sessions
- Are there spikes in memory? Large reports or exports being executed? Is there an upward trend over time?

Address Binding

- Computer memory uses both logical addresses and physical addresses. Address binding allocates a physical memory location to a logical pointer by associating a physical address to a logical address, which is also known as a virtual address. Address binding is part of computer memory management and it is performed by the operating system on behalf of the applications that need access to memory.
- Address binding relates to how the code of a program is stored in memory. Programs are written in human-readable text, following a series of rules set up by the structural requirements of the programming language, and using keywords that are interpreted into actions by the computer's Central Processing Unit. The point at which the executable version of a program is created dictates when address binding occurs. Some program languages, such as "C" and COBOL need to be compiled, while others, mainly scripts, run from the original program text rather than a machine code compiled binary version.

Compile Time

- The first type of address binding is compile time address binding. This allocates a space in

memory to the machine code of a computer when the program is compiled to an executable binary file. The address binding allocates a logical address to the starting point of the segment in memory where the object code is stored. The memory allocation is long term and can be altered only by recompiling the program.

Load Time

- If memory allocation is designated at the time the program is allocated, then no program can ever transfer from one computer to another in its compiled state. This is because the executable code will contain memory allocations that may already be in use by other programs on the new computer. In this instance, the program's logical addresses are not bound to physical addresses until the program is invoked and loaded into memory.

Execution Time

- Execution time address binding usually applies only to variables in programs and is the most common form of binding for scripts, which don't get compiled. In this scenario, the program requests memory space for a variable in a program the first time that variable is encountered during the processing of instructions in the script. The memory will allocate space to that variable until the program sequence ends, or unless a specific instruction within the script releases the memory address bound to a variable.

Logical and Physical Address in Operating System

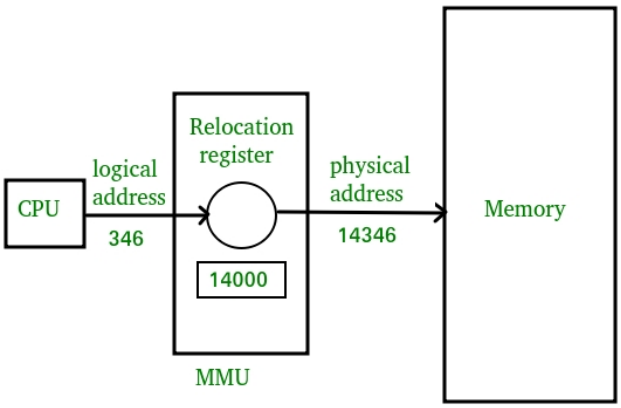
Logical Address

- **Logical Address** is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU. The term Logical Address Space is used for the set of all logical addresses generated by a program’s perspective.The hardware device called Memory-Management Unit is used for mapping logical address to its corresponding physical address.

Physical Address

- **Physical Address** identifies a physical location of required data in a memory. The user never directly deals with the physical address but can access by its corresponding logical address. The user program generates the logical address and thinks that the program is running in this logical address but the program needs physical memory for its execution, therefore, the logical address must be mapped to the physical address by MMU before they are used. The term Physical Address Space is used for all physical addresses corresponding to the logical

addresses in a Logical address space.



Differences Between Logical and Physical Address in Operating System

1. The basic difference between Logical and physical address is that Logical address is generated by CPU in perspective of a program where as the physical address is a location that exists in the memory unit.
2. Logical Address Space is the set of all logical addresses generated by CPU for a program where as the set of all physical address mapped to corresponding logical addresses is called Physical Address Space.
3. The logical address does not exist physically in the memory where as physical address is a location in the memory that can be accessed physically.
4. Identical logical addresses are generated by Compile-time and Load time address binding methods whereas they differs from each other in run-time address binding method.
5. The logical address is generated by the CPU while the program is running whereas the physical address is computed by the Memory Management Unit (MMU).

6. **Comparison Chart:**

PARAMENTER	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address	User can never view physical

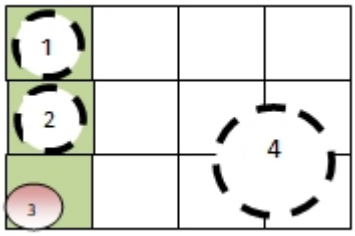
	of a program.	address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.

Memory Partitioning

- **Memory partitioning** is the system by which the memory of a computer system is divided into sections for use by the resident programs. These memory divisions are known as partitions. There are different ways in which memory can be partitioned: fixed, variable, and dynamic partitioning.

Fixed Partitioning

- Let us think about the jewelry box that stores rings. Different pieces of jewelry represent different computer processes. Say you have a jewelry box that displays rings. This box has been partitioned into equal parts.



- Fixed partitioning is ideal for our rings collection provided we always acquire rings with sizes less than these fixed partitions. However, we never have identical jewelry of the same size. Item 1 & 2 fit almost perfectly with little or no wasted space (partition almost the same size as the process). Item 3 however, leaves a lot more space unoccupied (small process in large partition), while item 4 has no partition large enough to fit (large process with unavailable sized partition).
- **Fixed partitioning** is therefore defined as the system of dividing memory into non-overlapping sizes that are fixed, unmoveable, static. A process may be loaded into a partition of equal or greater size and is confined to its allocated partition.
- If we have comparatively small processes with respect to the fixed partition sizes, this poses a big problem. This results in occupying all partitions with lots of unoccupied space left. This unoccupied space is known as fragmentation. Within the fixed partition context, this is known as internal fragmentation (IF). This is because of unused space created by a process within its allocated partition (internal).

Variable Partitioning

- An alternate solution to address these problems is variable partitioning. In the case of the jewelry box, we anticipate that we possess different jewelry items with different sizes. So, we divide the area of our jewelry box into partitions of different sizes. This way, smaller items of jewelry (small processes) are assigned small partitions while the bigger partitions are saved for the bigger items (large processes).

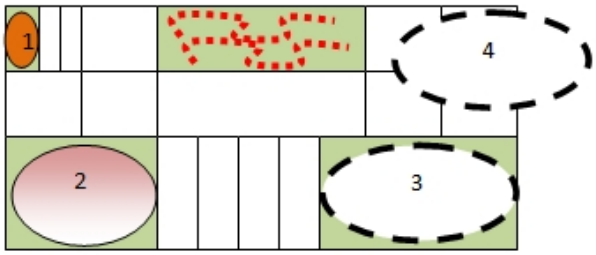


Figure 2: Illustration - Jewelry Box

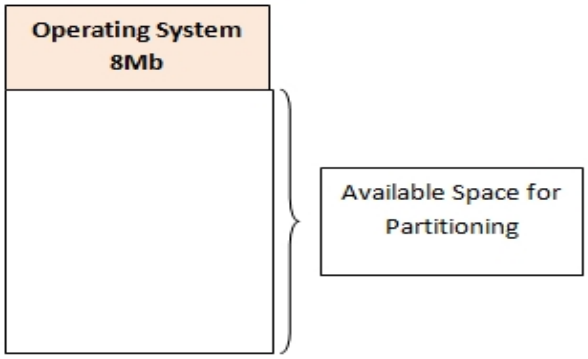
- **Variable partitioning** is therefore the system of dividing memory into non-overlapping but variable sizes. This system of partitioning is more flexible than the fixed partitioning configuration, but it's still not the most ideal solution. Small processes are fit into small partitions (item 1) and large processes fit into larger partitions (items 2 and 3). These processes do not necessarily fit exactly, even though there are other unoccupied partitions. Items 3 and 4 are larger processes of the same size, but memory has only one available partition that can fit either of them.

The flexibility offered in variable partitioning still does not completely solve our problems.

Dynamic Partitioning

Let us examine a typical 64MB space of memory and let's assume that the first 8MB of memory is reserved for the operating system.

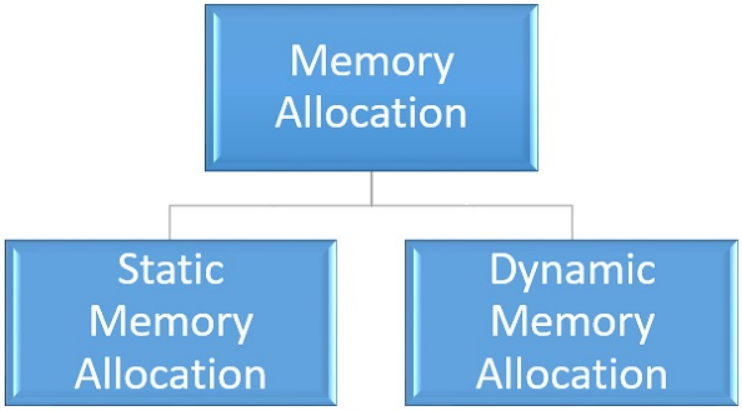
Figure 3: Memory Space



64Mb memory space

Memory Allocation

- **Memory allocation** is the process of setting aside sections of memory in a program to be used to store variables, and instances of structures and classes. There are two basic types of memory allocation.
- **Memory allocation** is an action of assigning the physical or the virtual memory address space to a process (its instructions and data). The two fundamental methods of memory allocation are static and dynamic memory allocation.
- Static memory allocation method assigns the memory to a process, **before its execution**. On the other hand, the dynamic memory allocation method assigns the memory to a process, **during its execution**.
- To get a process executed it must be first placed in the memory. Assigning space to a process in memory is called memory allocation. Memory allocation is a general aspect of the term **binding.**



Types of Memory Allocation

1. Static Memory Allocation

* Static memory allocation is performed when the compiler compiles the program and generate object files, linker merges all these object files and creates a single executable file, and loader loads this single executable file in main memory, for execution. In static memory allocation, the size of the data required by the process must be known **before** the execution of the process initiates.

* If the data sizes are not known before the execution of the process, then they have to be guessed. If the data size guessed is larger than the required, then it leads to **wastage** of memory. If the guessed size is smaller, then it leads to inappropriate execution of the process.

* Static memory allocation method does not need any memory allocation operation during the execution of the process. As all the memory allocation operation required for the process is done before the execution of the process has started. So, it leads to **faster** execution of a process.

* Static memory allocation provides more **efficiency** when compared by the dynamic memory allocation.

2. Dynamic Memory Allocation

* Dynamic memory allocation is performed while the program is in execution. Here, the memory is allocated to the entities of the program when they are to be used for the **first time** while the program is running.

* The actual size, of the data required, is known at the run time so, it allocates the **exact** memory space to the program thereby, reducing the memory wastage.

* Dynamic memory allocation provides **flexibility** to the execution of the program. As it can decide what amount of memory space will be required by the program. If the program is large enough then a dynamic memory allocation is performed on the different parts of the program, which is to be used currently. This reduces memory wastage and improves the performance of the system.

* Allocating memory dynamically creates an overhead over the system. Some allocation operations are performed repeatedly during the program execution creating more overheads, leading in **slow** execution of the program.

* Dynamic memory allocation does not require special support from the operating system. It is the responsibility of the programmer to design the program in a way to take advantage of dynamic memory allocation method.

* Thus the dynamic memory allocation is flexible but slower than static memory allocation.

Advantages of static and dynamic memory allocation

Static Memory Allocation

1. Static memory allocation provides an **efficient** way of assigning the memory to a process.

2. All the memory assigning operations are done before the execution starts. So, there are *no* **overheads** of memory allocation operations at the time of execution of the program.
3. Static memory allocation provides **faster** execution, as at the time of execution it doesn't have to waste time in allocation memory to the program.

Dynamic Memory Allocation

1. Dynamic memory allocation provides a **flexible** way of assigning the memory to a process.
2. Dynamic memory allocation **reduces** the memory **wastage** as it assigns memory to a process during the execution of that program. So, it is aware of the exact memory size required by the program.
3. If the program is large then the dynamic memory allocation is performed on the different parts of the program. Memory is assigned to the part of a program that is currently in use. This also reduces memory wastage and indeed improves **system performance**.

Disadvantages of static and dynamic memory allocation

Static Memory Allocation

1. In static memory allocation, the system is **unaware** of the memory requirement of the program. So, it has to guess the memory required for the program.
2. Static memory allocation leads to memory **wastage**. As it estimates the size of memory required by the program. So, if the estimated size is larger, it will lead to memory **wastage** else if the estimated size is smaller, then the program will execute **inappropriately**.

Dynamic Memory allocation

1. Dynamic memory allocation method has an **overhead** of assigning the memory to a process during the time of its execution.
2. Sometimes the memory allocation actions are repeated several times during the execution of the program which leads to more **overheads**.
3. The overheads of memory allocation at the time of its execution **slowdowns** the execution to some extent.

Memory protection

- **Memory protection** is a way to control memory access rights on a computer, and is a part of most modern instruction set architectures and operating systems. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug or malware within a process from affecting other processes, or the operating system itself. Protection may encompass all accesses to a specified area of memory, write accesses, or attempts to execute the contents of the area. An attempt to access unauthorized memory results in a hardware fault, e.g., a segmentation fault, storage violation exception, generally causing abnormal termination of the offending process. Memory protection for computer security includes additional techniques such as address space layout randomization and executable space protection.

Memory protection in different operating systems

Different operating systems use different forms of memory protection or separation. Although

memory protection was common on most mainframes and many minicomputer systems from the 1960s, true memory separation was not used in home computer operating systems until OS/2 (and in RISC OS) was released in 1987. On prior systems, such lack of protection was even used as a form of interprocess communication, by sending a pointer between processes. It is possible for processes to access System Memory in the Windows 9x family of Operating Systems.

Some operating systems that do implement memory protection include:

- Unix-like systems (since the late 1970s), including Solaris, Linux, BSD, macOS, iOS and GNU Hurd
- Plan9 and Inferno, created at Bell Labs as Unix successors (1992, 1995)
- OS/2 (1987)
- RISC OS (1987) (The OS memory protection is not comprehensive.)
- Microware OS-9, as an optional module (since 1992)
- Microsoft Windows family from Windows NT 3.1 (1993)

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types −

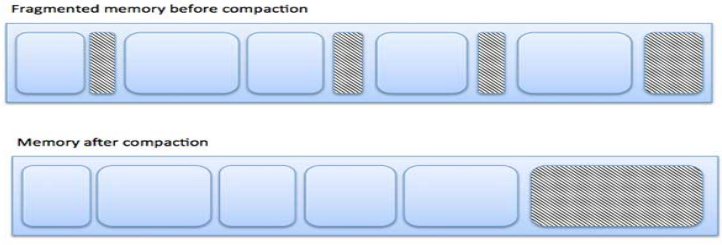
External fragmentation

Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.

Internal fragmentation

Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory −



External fragmentation can be reduced by compaction or shuffle memory contents to place all

free memory together in one large block. To make compaction feasible, relocation should be dynamic.

The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

COMPACTION

External Fragmentation

External fragmentation exists when enough total memory space exists to satisfy a request ,but it is not contiguous.

Internal Fragmentation

The allocation memory may be slightly larger than the requested memory. The difference between these two number is internal fragmentation.

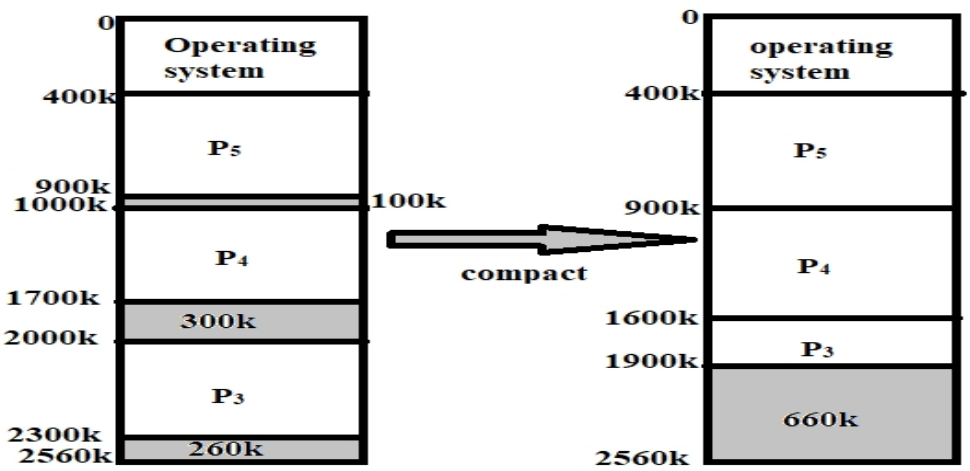
COMPACTION

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory content to place all free memory together in one large block.(or)the process of collecting all the wasted spaces is called as compaction.

For example, the three holes of sizes 100k,300k and 260k can be compacted into one hole of sizes 660k.

The process of collecting all the waste memory spaces is called as compaction. compaction is possible only if relocation is dynamic, and is done at execution time.

The simplest compaction algorithm is simply to move all processes toward one end of memory; all holes move in the other direction ,producing one large hole of available memory. This scheme can be quite expensive.



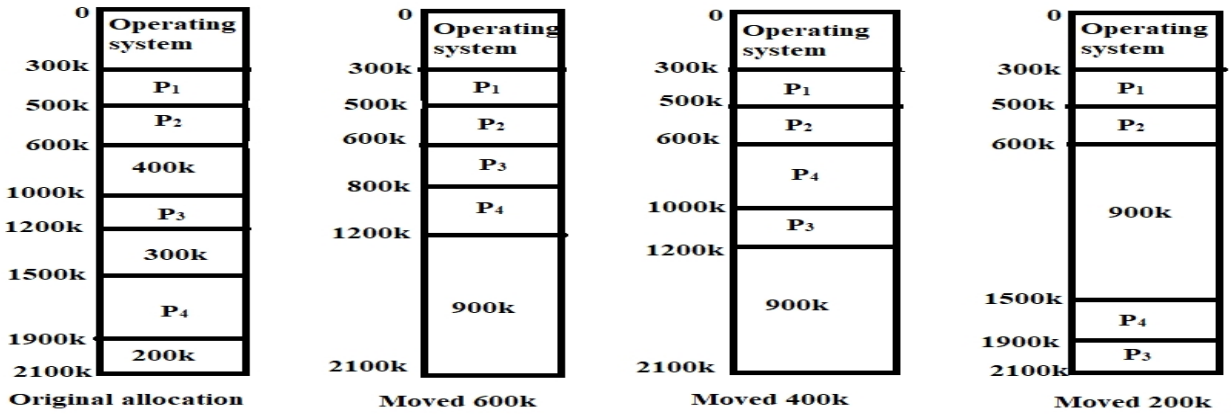
COMPACTION

The process of collecting all the waste memory spaces is called as compaction. compaction is possible only if relocation is dynamic, and is done at execution time.

The simplest compaction algorithm is simply to move all processes toward one end of memory; all holes move in the other direction ,producing one large hole of available memory. This scheme can be quite expensive.

consider the memory allocation shown in figure. If we this simple algorithm, we must move processes p3 and p4 ,for a total of 600k moved. In this situation, we could simply move process p4 above process p3, moving only 400k ,or move process p3 below process p4,moving only 200k.

SOME DIFFERENT WAYSTO COMPACT MEMORY



Swapping can also be combined with compaction. A process can be rolled out of main memory to a backing store and rolled in again later.

****