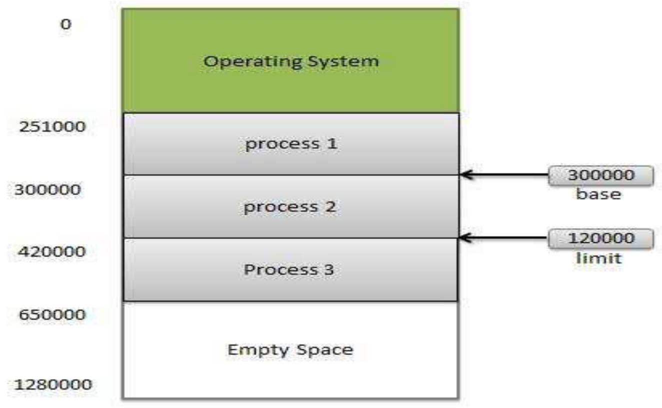
MEMORY MANAGEMENT

Memory management is the functionality of the operating system which handles or manages primary memory. Memory management involves keeping track of each and every memory location either allocated to a process or it's free, checking how much memory is to be allocated to a process, deciding which process get memory at what time, and tracking whenever memory gets unallocated and correspondingly updating the status.

It is important to note that memory is the central to the operation of a modern computer system and it consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of program counter. Main memory and registers built into the processor itself are the only storage that the CPU can access directly. Therefore, any instruction in execution and any data being used by the instructions must be in one of these direct-access storage locations. If the data is not in memory it must be moved there before the CPU can operate on them.

The operating system makes sure that each process has a separate memory space. It does so by determining the range of legal addresses that a process can access and ensuring that, a process can access only those legal addresses. This protection is provided through the use of two registers, **a base and a limit register**. The base register holds the smallest legal physical memory address while the limit register specifies the size of the range. For instance, if the base register holds 300000 and limit register holds 120000, then the program can legally access all addresses from 300000 through 419999 (inclusive) as shown in the diagram below.



Usually, to accomplish memory space protection, the CPU compares address generated in user mode with the registers. Any attempt by an executing program in user mode to access operating system memory or other process's memory results in a trap to the operating system, which treats the attempt as a fatal error. This prevents a user program from either accidentally or deliberately modifying the code or data structures of either the operating system or other user programs.

Memory Mapping

An address generated by the CPU is commonly referred to as **logical/virtual address** whereas an address seen by the memory unit-that is, the one loaded into memory address register of the memory is referred to as **physical address.** During run-time, mapping from virtual to physical address is done by hardware called **memory management unit.** The value of relocation (base) register is added to every address generated by a user process at the time it is sent to memory. User programs deals with logical addresses but they never see the real physical addresses. For example, if the base register is at 14000, then an attempt to address location 0 is dynamically relocated to location 14000. Similarly, an access to location 346 is mapped to location 14346. The program is creating a pointer to location 346, stores it in memory, manipulates it, and compares it with other addresses – all as number 346 but it never sees the real physical address 14346.

Memory Allocation Approaches

As processes enter the system, they are put into an input queue. The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory. When a process is allocated space, it is loaded into memory, and it can then compete for the CPU. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue depending on the memory allocation approach used.

a. **Fixed partition approach**

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. In this scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

It comes with the advantage that the scheme is simple and requires minimal operating system processing overhead. However, main memory utilization is extremely inefficient because programs are not of same size and therefore, loading programs of smaller capacity leads to internal fragmentation. In addition, the number of partitions specified at system generation time limit the number of active processes that can be in the main memory at any particular time. This approach was primarily used in batch environments and it is no longer used.

b. Variable partition approach

Initially, all memory is available for user processes and is considered one large block of available memory, called a **hole.** When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

In general, at any given time we have a *set* of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

The allocation problem with this scheme is how to satisfy a request of size ***n*** from a list of free holes. Various solutions to this problem exist where a set of holes is searched to determine which hole is best to allocate. The common strategies include;

- **First-fit:** allocate the first hole that is big enough. Searching can either start at the beginning of the set of holes or where previous first-fit search end and searching an end once a free large enough hole is located.
- **Best-fit:** allocate the smallest hole that is big enough. Entire list of holes must be searched unless the list is ordered by size. The smallest leftover hole is produced.
- **Worst-fit:** allocate the largest hole. Entire list is searched unless sorted by size. The largest leftover hole is produced.

Memory management policies

Various memory management policies differ in many aspects and they are compared using hardware support and performance in the context of mapping.

1. **Swapping**

A process can be swapped temporarily out of memory to a backing store and then brought back into the memory for continued execution. Assume a higher priority process arrives and wants service. The memory manager can swap out the lower priority process so that it can load and execute the higher priority process. When the higher priority process finishes, the lower priority process can be swapped back in and execution continued. This variant is sometimes referred to as roll out, roll in.

A swapped out process is swapped back into the same memory space that it was previously occupying if binding is done at assembly or load time. However, if binding is done during execution time, then it is possible to swap a process into a different memory space. Swapping requires a backing store. The backing store is commonly a fast disk which must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

2. **Relocation**

As processes are loaded and removed from memory, the free memory space is broken into little pieces. A time comes when there is enough total memory space to satisfy a request but the available spaces are not contiguous, i.e. storage is fragmented into a large number of small holes. This fragmentation can be severe. If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

One solution to this problem is compaction whose goal is to shuffle the memory contents so as to place all free memory together in one large block. If relocation is dynamic and done at execution time, then compaction would be possible. Relocation requires only moving the program and data and then changing the base register to reflect the new base address. The simplest compaction algorithm is to move all processes towards one end of memory and all holes move in the other direction, producing one large hole of available memory.

3. Paging

In this scheme, the physical memory is divided into fixed-size blocks called **frames** and the logical memory into blocks of the same size called **pages**. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is similarly divided into fixed-size blocks that are of the same size as memory frames. Every address generated by the CPU is divided into two parts; **a page number and a page offset.** The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with page offset to define the physical memory address that is sent into memory unit.

When this scheme is used, external fragmentation is avoided since any free frame can be allocated to a process that needs it. However, there can be internal fragmentation if memory requirements of a process do not coincide with page boundaries. Paging is similar to fixed partitions but the only difference is that partitions are rather small and programs occupy more than one partition and they need not be contiguous.

4. Segmentation

A user program can be subdivided using segmentation, in which the program and its associated data are divided into a number of **segments.** This scheme divides a program into a number of small blocks called segments. It is not required that all segments of all programs be of the same length, although there is a maximum segment length. Similar to paging, a logical address using segmentation consists of two parts, in this case a segment number and an offset.

Because of the use of unequal-size segments, segmentation is similar to dynamic (variable) partitioning. In segmentation, a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less.

Segmentation scheme makes use of a segment table for each process and a list of free blocks of main memory. Each segment table entry gives the starting address in main memory of the corresponding segment. The entry should also provide the length of the segment, to assure that invalid addresses are not used.