Memory Management

The memory of a computer system is shared by a large number of processes, so memory management has traditionally been a very important task of an operating system. Memories keep becoming cheaper and larger every year; however, the pressure on memory as an OS resource persists because both the size of processes and the number of processes that an operating system has to service at any time also keep growing. The basic issues in memory management are efficient use of memory, protection of memory allocated to a process against illegal accesses by other processes, performance of individual processes, and performance of the system.

❖ Managing the Memory Hierarchy

A memory hierarchy comprises cache memories like the L1 and L3 caches, the memory management unit (MMU), memory, and a disk. Its purpose is to create an illusion of a fast and large memory at a low cost.

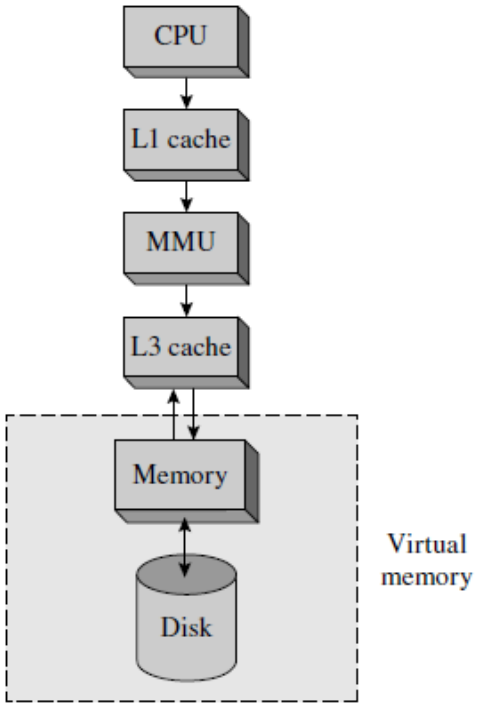


Figure 11.1 Managing the memory hierarchy.

The upper half of Figure 11.1 illustrates the memory hierarchy. The CPU refers to the fastest memory, the cache, when it needs to access an instruction or data. If the required instruction or data is not available in the cache, it is fetched from the next lower level in the memory hierarchy, which could be a slower cache or the random access memory (RAM), simply called memory in this book. If the required instruction or data is

also not available in the next lower level memory, it is fetched there from a still lower level, and so on. Performance of a process depends on the hit ratios in various levels of the memory hierarchy, where the hit ratio in a level indicates what fraction of instructions or data bytes that were looked for in that level were actually present in it.

The caches are managed entirely in the hardware. The kernel employs special techniques to provide high cache hit ratios for a process. Memory is managed jointly by the kernel and the run-time library of the programming language in which the code of the process is written. The kernel allocates memory to user processes. The primary performance concern in this function is accommodating more user processes in memory, so that both system performance and user service would improve. The kernel meets this concern through efficient reuse of memory when a process completes.

The disk in the virtual memory is managed entirely by the kernel; the kernel stores different parts of each process's address space on the disk in such a manner that they can be accessed efficiently. It contributes to good execution performance of processes in a virtual memory.

❖ Static And Dynamic Memory Allocation

Memory allocation is an aspect of a more general action in software operation known as binding. Two other actions related to a program—it's linking and loading—are also aspects of binding.

A binding for an attribute of an entity such as a function or a variable can be performed any time before the attribute is used. Different binding methods perform the binding at different times. The exact time at which binding is performed may determine the efficiency and flexibility with which the entity can be used. Broadly speaking, we can differentiate between early binding and late binding. Late binding is useful in cases where the OS or run-time library may have more information about an entity at a later time, using which it may be able to perform a better quality binding. For example, it may be able to achieve more efficient use of resources such as memory. Early and late binding are represented by the two fundamental binding methods of static and dynamic binding, respectively.

Definition: Static Binding-A binding performed before the execution of a program (or operation of a software system) is set in motion.

Definition: Dynamic Binding-A binding performed during the execution of a program (or operation of a software system).

Static memory allocation can be performed by a compiler, linker, or loader while a program is being readied for execution. Dynamic memory allocation is performed in a

"lazy" manner during the execution of a program; memory is allocated to a function or a variable just before it is used for the first time.

Static memory allocation to a process is possible only if sizes of its data structures are known before its execution begins. If sizes are not known, they have to be guessed; wrong estimates can lead to wastage of memory and lack of flexibility. For example, consider an array whose size is not known during compilation. Memory is wasted if we overestimate the array's size, whereas the process may not be able to operate correctly if we underestimate its size. Dynamic memory allocation can avoid both these problems by allocating a memory area whose size matches the actual size of the array, which would be known by the time the allocation is performed. It can even permit the array size to vary during operation of the process. However, dynamic memory allocation incurs the overhead of memory allocation actions performed during operation of a process.

Operating systems choose static and dynamic memory allocation under different circumstances to obtain the best combination of execution efficiency and memory efficiency. When sufficient information about memory requirements is available a priori, the kernel or the run-time library makes memory allocation decisions statically, which provides execution efficiency. When little information is available a priori, the memory allocation decisions are made dynamically, which incurs higher overhead but ensures efficient use of memory. In other situations, the available information is used to make some decisions concerning memory allocation statically, so that the overhead of dynamic memory allocation can be reduced.

❖ Memory Allocation To A Process

➢ *Stacks and Heaps*

The compiler of a programming language generates code for a program and allocates its static data. It creates an object module for the program. The linker links the program with library functions and the run-time support of the programming language, prepares a ready-to-execute form of the program, and stores it in a file. The program size information is recorded in the directory entry of the file. The run-time support allocates two kinds of data during execution of the program. The first kind of data includes variables whose scope is associated with functions, procedures, or blocks, in a program and parameters of function or procedure calls. This data is allocated when a function, procedure or block is entered and is deallocated when it is exited. Because of the last-in, first-out nature of the allocation/deallocation, the data is allocated on the stack. The second kind of data is dynamically created by a program through language features like the new statement of Pascal, C++, or Java, or the malloc, calloc statements of C. We refer to such data as program-controlled dynamic data (PCD data). The PCD data is allocated by using a data structure called a heap.

Stack: In a stack, allocations and deallocations are performed in a last-in, first out (LIFO) manner in response to push and pop operations, respectively. We assume each entry in the stack to be of some standard size, say, l bytes. Only the last entry of the stack is accessible at any time. A contiguous area of memory is reserved for the stack. A pointer called the stack base (SB) points to the first entry of the stack, while a pointer called the top of stack (TOS) points to the last entry allocated in the stack. We will use the convention that a stack grows toward the lower end of memory.

During execution of a program, a stack is used to support function calls. The group of stack entries that pertain to one function call is called a stack frame; it is also called an activation record in compiler terminology. A stack frame is pushed on the stack when a function is called. To start with, the stack frame contains either addresses or values of the function's parameters, and the return address, i.e., the address of the instruction to which control should be returned after completing the function's execution. During execution of the function, the run-time support of the programming language in which the program is coded creates local data of the function within the stack frame. At the end of the function's execution, the entire stack frame is popped off the stack and the return address contained in it is used to pass control back to the calling program.

Two provisions are made to facilitate use of stack frames: The first entry in a stack frame is a pointer to the previous stack frame on the stack. This entry facilitates popping off of a stack frame. A pointer called the frame base (FB) is used to point to the start of the topmost stack frame in the stack. It helps in accessing various stack entries in the stack frame.

Heap: A heap permits allocation and deallocation of memory in a random order. An allocation request by a process returns with a pointer to the allocated memory area in the heap, and the process accesses the allocated memory area through this pointer. A deallocation request must present a pointer to the memory area to be deallocated. The use of a heap to manage the PCD data of a process. As "holes" develop in the memory allocation as data structures are created and freed. The heap allocator has to reuse such free memory areas while meeting future demands for memory.

- ***Memory Fragmentation***

Memory Fragmentation - The existence of unusable areas in the memory of a computer system.

Table 11.3 describes two forms of memory fragmentation. External fragmentation occurs when a memory area remains unused because it is too small to be allocated. Internal fragmentation occurs when some of the memory allocated to a process remains unused, which happens if a process is allocated more memory than it needs. We would have

internal fragmentation if an allocator were to allocate, say, 100 bytes of memory when a process requests 50 bytes; this would happen if an allocator dealt exclusively with memory blocks of a few standard sizes to limit its overhead.

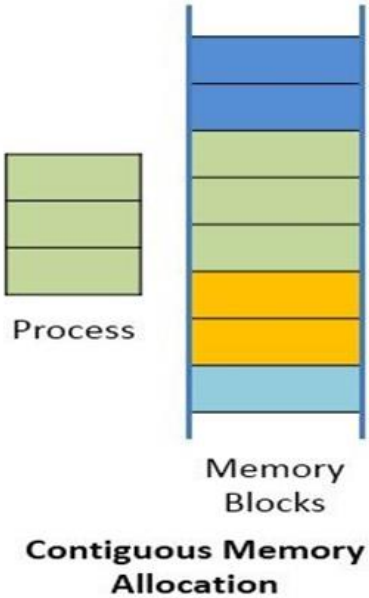
Table 11.3 Forms of Memory Fragmentation

Form of fragmentation	Description
External fragmentation	Some area of memory is too small to be allocated.
Internal fragmentation	More memory is allocated than requested by a process, hence some of the allocated memory remains unused.

❖ Contiguous Memory Allocation

In contiguous memory allocation, all the available memory space remains together in one place. It means freely available memory partitions are not scattered here and there across the whole memory space. In the contiguous memory allocation, both the operating system and the user must reside in the main memory. The main memory is divided into two portions one portion is for the operating and other is for the user program.

In the contiguous memory allocation when any user process request for the memory a single section of the contiguous memory block is given to that process according to its need. We can achieve contiguous memory allocation by dividing memory into the fixed-sized partition. A single process is allocated in that fixed sized single partition. But this will increase the degree of multiprogramming means more than one process in the main memory that bounds the number of fixed partition done in memory. Internal fragmentation increases because of the contiguous memory allocation.



Contiguous Memory Allocation

The memory can be divided either in the fixed-sized partition or in the variable-sized partition in order to allocate contiguous space to user processes.

- ***Fixed sized partition***

In the fixed sized partition the system divides memory into fixed size partition (may or may not be of the same size) here entire partition is allowed to a process and if there is some wastage inside the partition is allocated to a process and if there is some wastage inside the partition then it is called internal fragmentation.

Advantage: Management or book keeping is easy.

Disadvantage: Internal fragmentation

- ***Variable size partition***

In the variable size partition, the memory is treated as one unit and space allocated to a process is exactly the same as required and the leftover space can be reused again.

Advantage: There is no internal fragmentation.

Disadvantage: Management is very difficult as memory is becoming purely fragmented after some time.