Memory Management Concepts

In the previous article- Memory Management Basics I discussed about different binding schemes. In this article I will discuss about the concepts dynamic loading, dynamic linking and shared libraries and different memory mapping schemes and protection.

Dynamic Loading

In our discussion so far, the entire program and all data of a process must be in physical memory for the process to execute. The size of a process is thus limited to the size of physical memory. To obtain better memory-space utilization, we use dynamic loading. With dynamic loading a routine is not loaded into memory until it is called.All routines are kept on disk in relocatable format. The main program is loaded into memory and is executed. When a routine needs to call another routine the calling routine first checks to see whether the other routine has been loaded. If not the relocatable linker is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

Advantages

- The advantage of dynamic loading is that an unused routine is never loaded. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In this case, although the total program size ,may be large the portion that is used and loaded may be much smaller.
- It does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage.Operating systems my help the programmer by providing library routines to implement dynamic loading.

Dynamic Linking and Shared Libraries

Some Operating systems support only **Static Linking** , in which the system language libraries are treated like any other object module and are combined by the loader into the binary program image.

The concept of dynamic linking is similar to that of dynamic loading. Here linking is postponed until execution time. This feature is usually used with system libraries such as language subroutine libraries. Without this facility, each program on a system must include a copy of its language library in the executable image. This requirement wastes both disk space and main memory.

With dynamic linking a **stub** is included in the image for each library routine reference. The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to locate the appropriate memory resident library routine or how to load the library if the routine is not already present. When the stub is executed , it checks to see whether the needed routine is already in memory. If not the program loads the routine into memory. Either way the stub replaces itself with the address of the routine executes the routine. The next time that particular code segment is reached, the library routine is executed directly , incurring no cost for dynamic linking. All processes that use a language library execute only one copy of the library code.

This feature can be extended to library updates such as bug fixes. A library may be replaced by a new

version and all programs that reference the library will automatically use the new version. Without dynamic linking all such programs would need to be relinked to gain access to the new library.

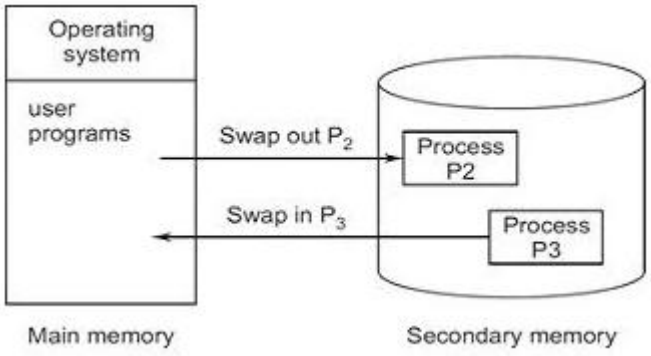
Comparison of Dynamic Linking and Dynamic Loading

Dynamic linking requires help from the Operating System. Dynamic loading require special support from the Operating System.

If the processes in memory are protected from one another , then the Operating System is the only entity that can check to see whether the needed routine is in another processes memory space or that can allow multiple processes to access the same memory addresses.

Swapping

Any process should be in main memory during its execution. During its execution if the process needs any Input/Output device or a higher priority process wants to enter into main memory. But the size of the main memory is limited. Thus for accommodating new process or for using I/O device, present running process needs to move into secondary memory. The process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. This is called "swap out". New process needs to move into main memory is called "swap in". For example, assume a multi-programming environment with a round robin scheduling algorithm. When a quantum expires the memory manager ill start to swap out the process that has just finished and to swap another process into the memory space that has been freed. In the meantime the CPU scheduler will allocate a time slice to some other process in memory. When each process finishes its quantum it will be wrapped with another process. Ideally the memory manager can swap processes fast enough that some processes will be in memory ready to execute when the CPU scheduler wants to reschedule the CPU. In addition the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.



Normally a process that is swapped out will be swapped back into the same memory space it occupied previously.This restriction is dictated by the method of address binding. If binding is done at assembly time or load time then the process cannot be easily moved to a different location. If execution time binding is being used however then a process can be swapped into a different memory space because the physical addresses are computed during execution time.

Swapping requires a backing store which commonly is a fast disk. It must be large enough to

accommodate copies of all memory images for all users and it must provide direct access to these 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 which checks to see whether the next process in the queue is in the 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.

The context switch time in such a swapping system is fairly high. To get an idea of it , let us assume that the user processes 10MB in size and the backing store is a standard hard disk with a transfer rate of 40MB per second. The actual transfer of the 10MB process to or from main memory takes:
10000KB/40000KB per second=1/4 second
=250 milliseconds

For efficient CPU utilization we want the execution time for each process to be long relative to the swap time.

Swapping is constrained by other factors as well.If we want to swap a process,we must be sure that it is completely idle.Of particular concern is an pending I/O.A process may be waiting for an I/O operation when we want to swap that process to free up memory. However,if the I/O is asynchronously accessing the user memory for I/O buffers,then the process cannot be swapped.Assume that the I/O operation is queued because the device is busy.If we want to swap out process P1 and swap in process P2, the I/O might then attempt to use memory that now belongs to process P2.

There are two main solutions to this problem:

Never swap a process with pending I/O or execute I/O operations into operating system buffers. Transfers between Operating system buffers and process memory then occur only when the process is wrapped in.

Contiguous Memory Allocation

The main memory must accommodate both the Operating System and the various user processes. We therefore need to allocate the parts of the main memory in the most efficient possible way.

The memory is usually divided into 2 partitions:-one for the resident operating system and one for the user processes. We can place the Operating System in either the low memory or high memory. The major factor affecting this decision is the location of the **interrupt vector**. Since the interrupt vector is often in the low memory, programmers usually place the Operating system in the low memory as well.

We usually want several user processes to reside in memory at the same time.We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.In this Contiguous Memory Allocation each process is contained in a single Contiguous section of memory.

Source: http://www.go4expert.com/articles/memory-management-concepts-t22381/