Systems Programming and Operating Systems	**Induprakas Kodukula**
CS414, Summer 1996	**Class#15, 7/30/96**

1 Class #15

This section discusses the issue of memory management. In particular, the basic rudiments of process execution and the management of memory will be discussed. Memory allocation policies, hardware support and code generation strategy will also be discussed.

1.1 Virtual memory and address binding

It has already been mentioned that every process has associated with an address space. In the early generation operating systems, these addresses corresponded to actual addresses in the physical memory. However, with the advent of multiporgramming, protection became a major concern. Also, very often an entire program could not fit in memory, but portions of it could. It was useful to be able to run such programs as well. For this reason, the notion of a *logical address* also known as *virtual address* was introduced. The address space associated with a process was then a *logical/virtual address space.*

1.2 Address binding

Address binding is the process where symbols in the user's code are replaced by their addresses. It is easy on the part of a programmer's part to use variable names and write the program using thse variables than program using address values. However, before the program executes its instructions, *all* variable references must be bound to physical memory location. This is done in one of three ways.

1.2.1 Compile time

A compiler can at compile time generate the absolute address for code as well as data for a program. This approach works when the address space of the process is known at compile time. In typical multi programming environments, the implication is that every time the program needs to be run at a different address, the program needs to be recompiled. So, this approach is rarely used, except in isolated systems such as embedded controllers.

1.2.2 Load time

The second approach is to use *relocatable* addresses. In this scheme, all addresses are specified in terms of an *offset* from a *base* address. The value of the *base* address is filled in at run time. All accesses to code and data are done by an offset from the base register, and never as an absolute value. Once the program is loaded, all the addresses in the program are bound to physical address. This approach is more general than the previous one in the sense that for different runs, the program does not need to be recompiled. It just needs to be reloaded.

1.2.3 Execute time

Finally, it is possible that addresses in a program need to rebound even at run time. A reason this may become necessary is when a process is migrated while executing. In this case, all the old address accesses need to be replaced by new addresses that will be in effect after the migration. This is quite a complex operation and needs hardware support in general.

1.3 Size of executable

Another major concern of memory management is to manage the size of the executable. It is desirable to reduce the size of the executable and to share code during program execution (this is especially true of large shared libraries, such as X, where it is desirable to have only one copy of the code segment in memory and share it). The following strategies are used for this purpose.

1.3.1 Dynamic Linking

A typical program written in some language usually links with a lot of system provided libraries to form the executable. For example, an application written in C links with the standard C library, a math library and so on. A library written using X links with the various X toolkit and widget libraries and possibly Motif libraries. When these libraries are linked *statically*, these libraries are treated as any other object files and they are linked in to form a part of the executable's image map. This leads to the situation where basically each executable has its own private copy of the libraries used commonly. For example, every C program would end up using the standard C library. When *code* segments can be shared, dynamic linking avoids this problem. Each call to a procedure defined in a dynamic library is coded as a stub which has the job of finding the address in memory where the corresponding library is loaded into memory. This way, all the processes end up sharing the code of the shared library. Finally, this *stub* also has the responsibility of *loading* a routine into memory if it is not already loaded.

1.3.2 Dynamic Loading

Dynamic loading is a mechanism to make the memory usage of a process small even though the size of the executable is large. The basic idea is as follows: A routine is loaded into memory only if it is called by some other routine. To start with, only the main routine is loaded into memory. When the main routine calls another routine, that is loaded into memory. So, routines are loaded into memory only if they are executed and not before. This can be quite significant, since very often a particular execution path in a program may not use all the routines. The overhead is that it takes time to load a program into memory if it is not already there. This adds to execution time overhead.

1.3.3 Overlays

The third option used to reduce the memory image of a program is to use *overlays*. The idea behind overlays is the following. A typical application program (for instance, a compiler) usually consists of several disjoint and non overlapping phases. For example, for a compiler, the input program is read first and parsed, then it is translated to some intermediate representation, then optimizations are performed, followed by an assembler invocation, followed by a linker invocation. Each of these phases is usually quite disjoint in terms of the functionality provided and the functions used. So, a useful strategy for running in a limited memory environment could be the following: A program writes an *overlay driver* in addition to its core functionality. The function of the overlay driver is to simply load the relevant code of each phase into memory before the phase starts. When an old phase of the compiler is completed, the new code dealing with the next phase is brought into memory and overwrites the memory locations where the old phase was loaded. So, rather than the whole compiler be memory resident at any time, only a single phase and the overlay driver need to be memory resident at any time.

1.4 Swapping

Any program needs to be in memory while executing. However, when a program is not executing, it can sometimes be removed out of memory into a *backing store* such as a disk. This enables memory to be freed up so that other processes can execute. This approach is known as *swapping*.

Swapping can be combined with process scheduling. For example, in a round-robin mode of scheduling processes, a process that has just finished its time slice is guarenteed not to run for a significant amount of time. So, it is useful to reclaim the memory used by this process to let other processes run. Ideally, the memory manager swaps processes fast enough that the scheduler always finds the next process it wants to run in memory. *Swapping* can also be used if a high priority process arrives and wants to run in the place of a currently running low-priority process.

Swapping complicates the issue of address binding. For example, if *load-time* address binding is used, then a swapped out process must be swapped back to the same memory locations that it occupied. However, if the operating system supports *execute time address binding*, then this requirement is obviated.

Swapping requires a secondary storage device such as a disk. Typically, every machine has a portion of the disk marked as the swap area. The operating system together maintains the mapping between the swapped locations and processes. At the same time, the *scheduler* also needs to be aware of swapped out processes, since it is inefficient to schedule a process which is not in memory. Finally, since I/O typically happens asynchronously, it is important that user memory locations involved in I/O **not** be removed from memory even if a process is swapped out. This is because otherwise, the I/O subsystem would simply attempt to store the data at some memory location which would correspond to data of some other process.

For a typical user job that has the size of 1M, swapping to a disk with a data transfer rate of 20MBytes/sec would require 50 ms for the data transfer alone along with all the seek and latency overheads, which would add up to about a 100 ms. To swap out a process and to swap in another one would take 200ms. In other words, only about 5 processes can be swapped in and out in a sec, i.e. swapping is quite expensive. So, while swapping is useful, it is not recommended for good performance.

1.5 Memory Allocation Strategies

Next, the issue of how free memory can be allocated to the demands of various processors is examined. Conceptually, a memory system can be thought of as consisting of holes and blocks. Holes represent empty spaces and blocks represent regions of memory that have been allocated to processes. There are four strategies that are commonly used for allocating holes to processes:

first-fit: Allocate first block in memory that is large enough

best-fit Allocate the block in memory that matches the size of the program most closely.

worst-fit Allocate the block in memory that is the largest, irrespective of the program size

rotating first-fit: Variant of first-fit, but start searching from where you left off the last time, this is to avoid build up of a lot of small holes at the start of the list.

Extensive simulations carried out suggest that *first* and *best* fit perform better with respect to storage utilization than worst fit. Also, first fit performs faster than best-fit.

It is easy to see that when a new program needs to start up, it is possible to have enough memory for the process in the entire system, but not to have a single hole to be large enough to enable the process to run. This problem is known as *external fragmentation.* The amount of external fragmentation and the number of holes in a system is characterized by Knuth's *50% rule*, which states that if n is the number of blocks in the system, and h is the number of holes, then $h \approx \frac{n}{2}$.

> **Proof:** Let p be the probability that a chosen block has a hole on its right. Given any block, in a steady state, roughly half the number of operations on its right will be deletions and an equal half will be insertions. From this, it can be seen that $p \approx \frac{1}{2}$. So, the number of blocks with holes as right neighbors is approximately $\frac{n}{2}$. Since every hole must be the right neighbor of some block, it must be the case that $n \approx \frac{h}{2}$. $\square$

Next, it is possible to derive approximations for the amount of space that is *wasted* by holes. Let $\mathfrak{F}$ be the amount of space *wasted* by holes. Let S_0 be the average size of a block and kS_0 be the average size of hole. (k better be greater than 1, because otherwise, on the average, no process can be loaded into the system).

It is easy to see that $\mathfrak{F} = \frac{\mathfrak{k}}{\mathfrak{k}+2}$, given Knuth's *50% rule*

Finally, it should be noted that since a hole allocated to a process need not be an exact fit, some space is wasted in the hole itself that has been allocated this way. This problem is known as *internal fragmentation.*