CMPSCI 377 Operating Systems **Spring 2010**

Lecture 13: March 4

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

13.1 Memory Management

Memory management is the act of allocating, removing, and protecting computer memory for multiple processes. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed. The management of main memory is critical to the computer system. Recall that initially, every program executable is resident on disk. The OS loads the program from the disk into main memory; how memory is allocated and where that memory is reserved is determined by the memory manager. While executing the program, the CPU fetches instructions and data from memory, possibly requiring further interactions with the memory manager.

13.1.1 Terminology

- **Segment:** It is a contiguous chunk of memory assigned to a process.
- **Physical Address:** A physical address, also real address or binary address, is the actual physical memory address that is used to access a specific storage cell in main memory.
- **Virtual Address:** It is an address *relative* to the start of a process' address space.

13.1.2 Generation of addresses

There are several techniques that can be used to determine how addresses are generated for use by a program.

Compile time: The compiler generates the exact physical location in memory starting from some fixed starting position k. The OS is not involved here. This is very restrictive because the compiler must know ahead of time how all memory in the system is going to be allocated in order to prevent using an address that might be used by another application.

Load time: Compiler generates an address, but at load time the OS determines the process' starting position. Once the process loads, it does not move in memory. This allows for greater flexibility, but still restricts the process location once it has been started.

Execution time: Compiler generates an address, but the OS can place it anywhere in memory. This is the most flexible technique because the OS can remap how the compiled addresses relate to physical memory addresses on the fly.

13.2 Uniprogramming

Perhaps the simplest model for using memory is to provide uniprogramming without memory protection, where each application runs with a hardwired range of physical memory addresses. A uniprogramming environment allows only one application to run at a time, thus an application can use the same physical addresses every time, even across reboots. However, only supporting a single process at a time prevents concurrency, and can reduce performance since multiple processes cannot be used to overlap periods of computation and I/O. Typically, uniprogramming applications use the lower memory addresses (low memory), and an operating system uses the higher memory addresses (high memory). In the simplest case, an application can address any physical memory location. More advanced systems protect the OS by checking all user program memory accesses against the OS memory bounds. This was used in early computer operating systems, and still is used in some simple devices such as cell phones.

13.3 Multiprogramming

Multiprogramming operating systems support multiple applications at once. Ideally, the OS should do this *transparently*–applications should be unaware that memory is shared and they should not care where in physical memory they are allocated. Secondly, the OS must provide *safety*, to ensure that that processes cannot corrupt each other or the OS. Finally, the main goal of multiprogramming is to improve *efficiency*. Multiprogramming should not degrade performance due to the fact that more advanced memory management is required.

13.4 Relocation

The role of relocation, the ability to execute processes independently from their physical location in memory, is central for memory management: virtually all the techniques in this field rely on the ability to relocate processes efficiently. The need for relocation is immediately evident when one considers that in a general-purpose multiprogramming environment a program cannot know in advance (before execution, i.e. at compile time) what processes will be running in memory when it is executed, nor how much memory the system has available for it, nor where it will be located. Hence a program must be compiled and linked in such a way that it can later be loaded starting from an unpredictable address in memory, an address that can even change during the execution of the process itself, if any swapping occurs.

It's easy to identify the basic requirement for a (binary executable) program to be relocatable: all the references to memory it makes during its execution must not contain absolute (i.e. physical) addresses of memory cells, but must be generated relatively, i.e. as a distance measured in contiguous memory words from some known point such as the start of the program's memory region. To support this, the *base* and *limit* addresses are used. These addresses refer to the first and last address of physical memory that a program can access respectively. Thus to ensure safety, all addresses generated for a process must reside within the base and limit addresses. Accessing an address outside this range can lead to what is called a "segmentation fault".

Relocation can be done in one of two ways. With *static* relocation, all addresses are generated once at load time. Once the program is running, it cannot be moved because there is no mechanism for recomputing the addresses that were previously determined. A *dynamic relocation* system generates all addresses during execution. In this case, the assembly code is produced with relative (or "logical") addresses for all data and instructions. These relative addresses are then added to the base address described previously. Typically, this is done in hardware using a special base (or "relocation") register so addresses can be calculated very quickly. To ensure protection, the address is also compared against the limit register to ensure that it is within the program's address space.

The benefit of dynamic relocation is that processes can be easily moved or grown during execution. This can be necessary if a process continuously allocates memory, or if a process must be moved in order to prevent memory fragmentation. However, there is some extra overhead since an addition is required before every memory access. In addition, the protection provided by base and limit registers can be overly restrictive, preventing sharing of memory between processes. Finally, the approach as described thus far, requires all processors to fit in memory, and limits the total size of a process to the available memory in the system. The dynamic relocation approach provides transparency and safety, and is reasonably efficient, but it has some limitations such as requiring full processes to be moved if they grow too large, which can be slow.

13.5 Memory Allocation

Memory allocation is the decision of where to acquire memory when a process requests it. To do this, the OS must keep track of what memory regions are in use and which are free, and it needs a policy for determining how free memory regions (or "holes") are given to processes. These decisions are made by the memory manager component of the OS kernel. Since most programs require many memory allocation/deallocation calls, memory management must be very fast, have low fragmentation, make good use of locality, and be scalable to multiple processors.

13.5.1 First-, Best- and Worst-Fit allocation techniques

A program may request memory when it is first instantiated, or dynamically as it runs and expands its heap. The memory manager may use one of several policies to decide what are the best free spots from which to take memory. The *first-fit* method finds the first chunk of desired size and returns it; it is generally considered to be very fast. First-fit simply scans through the list of free holes until it finds the first one large enough to accommodate the process. The *best-fit* approach attempts to find the "best" match for the request by finding the chunk that wastes the least of space, e.g. the hole that is closest to the size of the requested memory chunk, while still being big enough. This requires the full list of holes to be scanned in order to find the best match, but can lead to better utilization since smaller holes are filled up first. However, the residual free space in the hole can be very small and might be wasted. The opposite approach, *worst-fit* can also be used. Worst-fit takes memory from the largest free region, giving the process the most space to grow. As a general rule, first-fit is fastest, but increases fragmentation. Best-fit can result in small unusable holes, but in general studies have shown that it can produce better utilization than worst-fit.

13.5.2 Fragmentation

We say that *fragmentation* occurs when the allocated chunks of memory are inefficiently distributed throughout memory. If this happens, the OS must keep track of many small holes. This increases the bookkeeping required by the OS, but also means that a new process may request an amount of memory for which there should be sufficient free space, but none of the available memory regions are large enough to accommodate it. *External* fragmentation happens when there is a waste of space outside (ie, in between) allocated objects. This can occur when processes are frequently being loaded and unloaded, causing the available memory regions to be broken up into small chunks. *Internal fragmentation* happens when there is a waste of space inside an allocated area. Memory is typically allocated in evenly sized blocks, but a process may not require a full block, resulting in internal fragmentation. Attempting to keep track of all free space within blocks can be too expensive to be useful.

Compaction, or defragmentation, is a technique that reduces the amount of fragmentation in memory. It does this by physically rearranging the processes in memory to store the allocated regions contiguously. A simple form of compaction simply scans all of memory and moves every allocated region to form a contiguous chunk at the start of RAM. This creates a large regions of free space to impede the return of fragmentation. However, compaction can be an expensive and slow process since large amounts of memory may need to be moved. To reduce the cost, incremental compaction can be used where only a portion of the memory is defragmented–typically just enough to create sufficient free space to meet a current demand.

Swapping can also be used to rearrange processes, particularly when a system is under memory pressure. In this case, the memory used by an inactive process is copied to disk; once the copy is complete the main memory used by the process can be released and given to other processes. However, once the process starts running again, the data moved to disk must be swapped back into main memory, which can be very slow. Swapping worst best with dynamic relocation systems since the process can be brought back to any region that has sufficient space. Ideally, only processes that will not run in the near future should be swapped out since accessing disk is orders of magnitude more expensive than memory.