**CMPSCI 377 Operating Systems** **Fall 2008**

# Lecture 14: October 21

*Lecturer: Prashant Shenoy* *Scribe: Shashi Singh*

## 14.1 Memory Management

Memory management is the act of managing computer memory. 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. Initially the program executable is resident on disk. The OS loads the program from the disk into main memory. While executing the program, the CPU fetches instructions and data from memory.

### 14.1.1 Terminology

- **Segment:** It is a chunk of memory assigned to a process.
- **Physical Address:** A physical address, also real address, or binary address, is the memory address that is electronically (in the form of binary number) presented on the computer address bus circuitry in order to enable the data bus to access a particular storage cell of main memory.
- **Virtual Address:** It is an address *relative* to the start of a process' address space.

### 14.1.2 Generation of addresses

**Compile time:** The compiler generates the exact physical location in memory starting from some fixed starting position k. The OS is not involved here.

**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.

**Execution time:** Compiler generates an address, and OS cana place it anywhere in memory.

## 14.2 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. Given that a uniprogramming environment allows only one application to run at a time, an application can use the same physical addresses every time, even across reboots. Typically, applications use the lower memory addresses (low memory), and an operating system uses the higher memory addresses (high memory). An application can address any physical memory location. OS is protected from process by checking addresses used by process.

## 14.3 Multiprogramming

One step beyond the uniprogramming model is to provide multiprogramming without memory protection. When a program is copied into memory, a linker-loader alters the code of the program (loads, stores, jumps) to use the address of where the program lands in memory. In this environment, bugs in any program can cause other programs to crash, even the operating system. The third model is to have a multiprogrammed operating system with memory protection. Memory protection keeps user programs from crashing one another and the operating system.

## 14.4 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 is 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 number of contiguous memory words, from some known point. The memory references a program can generate are of two kinds: references to instructions ad references to data. The former kind is implied in the execution of program branches or subroutine calls: a jump machine instruction always involves the loading of the CPU program counter register with the address of the memory word containing the instruction to jump to. The executable code of a relocatable program must then contain only relative branch machine instructions, in which the address to branch to is specified as an increment (or decrement) with respect to the address of the current instruction (or to the content of a register or memory word). The latter kind comes into play when whenever program variables (including program execution variables, like a subroutine call stack) are accessed. In this case relocation is made possible by the use of indexed or increment processor addressing modes, in which the address of a memory word is computed at reference time as the sum of the content of a register plus an increment or a decrement.

As we'll see later, the memory references of a process in a multitasking environment must somehow be bounded, so to protect from unwanted interferences memory areas like the unwritable parts of the process itself, or the memory areas containing the images of other processes, etc. This is usually accomplished in hardware by comparing the address of each memory reference produced by a process with the content of one or more bound registers or memory words, so that the processor traps an exception to block the process should an illegal address be generated.

## 14.5 Memory Allocation

Usually memory is allocated from a large pool of unused memory area called the heap. In C++, dynamic allocation/deallocation must be manually performed using commands like *malloc, free, new and delete*. *Malloc* allocates space of a given size and gives a pointer back to the programmer; the programmer then can do whatever he wants with it. The *new* command, on the other hand, allocates a specific object of a given

size. The general way in which dynamic allocation is done is that the program asks the memory manager to allocate or free objects (or multiple pages); then, the memory manager asks the OS to allocate/free pages (or multiple pages). Notice, however, that the allocator *does not* give the whole allocated page back to the program; it just gives what it asked for. The rest of the page (ie, the parts not given to the program) is saved for future memory requests.

## 14.6 Allocation techniques

Since most of the programs require a lot of memory allocation/deallocation, we expect the memory management to be fast, to have low fragmentation, make good use of locality, and be scalable to multiple processors.

We say that *fragmentation* occurs when the heap, due to characteristics of the allocation algorithm, gets "broken up" into several unusable spaces, or when the overall utilization of the memory is compromised. External fragmentation happens when there is waste of space outside (ie, in between) allocated objects; internal fragmentation happens when there is waste of space inside an allocated area.

Remember that the allocator might at any single time have several pages with unused space, from which it could draw pieces of memory to give to requesting programs. There are several ways to decide what are the good spots, among those with free memory, 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; *best-fit* finds the chunk that wastes the least of space; and *worst-fit* takes memory from the largest existent spot, and as a consequence maximizes the free space available. As a general rule, first-fit is faster, but increases fragmentation. One useful technique that can be applied with these methods is always to coalesce free adjacent objects into one big free object.

**Compaction:** It is a process that reduces the amount of fragmentation in file systems. It does this by physically organizing the contents of the disk to store the pieces of each file close together and contiguously. It also attempts to create larger regions of free space to impede the return of fragmentation. Some defragmenters also try to keep smaller files within a single directory together, as they are often accessed in sequence.

## 14.7 Keeping track of free memory

How do we keep track of where the free pieces of memory are? One idea is to maintain a set of linked-lists of free space; each linked-list will store free chunks of some given size (say, one list for chunks of 4 bytes, one for chunks of 8, 16, etc). This approach resembles the best-fit algorithm, since we can easily find the list with chunks of memory that are the closest to what we need; the difference, of course, is that usually the linked lists store not objects of *arbitrary* size, but only powers of two. Also, we can obviously never coalesce adjacent elements of the lists, since then the very idea of segregating by size classes wouldn't make sense anymore.

Another possible technique to keep track of free memory is called Big Bag of Pages (BiBOP). In this technique, a bunch of pages, usually segregated by size classes, is maintained such that there is a header on each page; on this header, among other things, we can find a pointer to the next page of objects of that given size, and also a pointer to the first free "slot" inside that page. Using the right optimizations we can make BiBOP find and manage free slots of memory in O(1).