CMPSCI 377 Operating Systems **Fall 2013**

Lecture 12: October 15

Lecturer: Prashant Shenoy *Scribe: Armand Halbert*

12.1 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.

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

A program requests memory when it is first instantiated. The memory manager may use one of several policies to decide what are the best free spots from which to allocate. 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.

12.1.2 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. Typically, one-third of memory is wasted due to external fragmentation. *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 works 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. Swapping was developed before virtual memory and actually led to the development of virtual memory.

12.2 Paging

The main motivation for *paging* is the 90/10 rule. The 90/10 rule states that processes typically spend 90% of their time accessing 10% of their space in memory. This means that we really only need to keep the parts of a process in memory that is actually being used (saving about 90% of memory). Paging also helps with the problem of fragmentation: rather than allocating large contigeous regions in memory, multiple regions of fixed size are allocated. The size of holes in memory will also be of some multiple of the size of a page. Logically, the process will still se a contiguous memory space, but in reality pages will be allocated in no certian order (or placement). A place in memory where a page is mapped to is called a *frame*. Paging effectively eliminates external fragmentation but not internal fragmentation. Processes cannot be allocated less than a single page even if they will not use it. A typical page size is 4 kilobytes.

12.2.1 Paging Hardware

A process will use *virtual addresses* to refer to memory locations. This is necessary because the process sees the memory space as one contiguous region starting with address 0. When a program issues a memory load or store operation, the virtual addresses (VAs) used in those operations have to be translated into "real" physical memory addresses (PAs). Since this translation could potentially be required for every assembly instruction, address translation must be done very quickly. As a result, modern computers include paging hardware that performs these actions. The paging hardware maintains a *page table* (a big hash table) that maps VAs into PAs. Notice, however, that we can't map every single byte of virtual memory to a physical address; that would require a huge page table. Instead, the paging hardware works at coarser granularity and maps virtual pages to physical pages. Also, since we want to isolate each program's address space from other application's address spaces, there must be aseparate page table for each process; this ensures that even if multiple different processes use the same virtual address to refer to some data, that would not be a problem since these addresses would be mapped into different physical addresses. All virtual pages that are not being mapped into a physical address are marked as being *invalid*; segfaults occur when a program tries to reference or access a virtual address that is not valid. Besides valid bits, entries in the page table can also store other information, such as "read" and "write" bits to indicate which pages can be read/written.

12.2.2 Virtual addresses

Virtual addresses are made up of two parts: the first one contains a page number (p), and the second one contains an offset inside that page (d). Suppose our pages are 4KB (4096 bytes = 2^{12} bytes) long, and that our machine uses 32 bit addresses. Then we can have at most 2^{32} addressable bytes of memory; therefore, we could fit at most $\frac{2^{32}}{2^{12}} = 2^{20}$ pages. This means that we need 20 bits to address any single page. Thus the first part of a virtual address will be formed by its 20 most significant bits, which will be used to address an entry in the page table (p); the $32 - 20 = 12$ least significant bits of the VA will be used as an offset inside the page (d). Of course, with 12 bits of offset we can address $2^{12} = 4096$ bytes, which is exactly what we need in order to address every byte inside our 4kb pages.

For a second example, consider a system with only 256 bytes of memory and a page size of 16 bytes; assume that within a page, memory can be addressed at word level (4 byte) granularity. Such a system can support $\frac{256}{16} = 16$ pages. In order to address the 16 pages, we need 4 bits ($2^4 = 16$). Since each page is divided into $\frac{16}{4} = 4$ addressable words, we need 2 bits to calculate the offset. Thus the 4 most significant bits in a virtual address will be used to select the page, p, and the 2 least significant bits in each address will be used for d, the offset within a page. Given a virtual address, the correct page and offset can be easily found by writing out the address in binary and separating the bits into the p and d chunks. Converting each of these pieces back into decimal will reveal the correct page table entry and the offset within that page. In order to find the true *physical* address, you must then use p as an index into the page table to determine the actual physical frame where the page is stored. The final physical address is then calculated by using the frame address plus the offset within the page. An offset can be computed by $offset = addressMODpagesize$.

12.2.3 Efficient Paging

The translation of virtual addresses into physical addresses needs to be very fast because of how frequently it needs to be done. Registers are very fast, but are of a limited quantity and so are not practical for storing the page table. Using kernel memory to store page tables will work, but each memory access will turn into two memory accesses. The first is to access the page table, then the second is to access the memory that the program wants. In order to speed up the translation of virtual to physical addresses, modern computers contain a *Translation Look-aside Buffer* (TLB) in hardware. The TLB acts as a cache (it is a fast, fully associative, memory) of page to frame mappings. When an address must be translated, if it is found in the TLB then it can be very quickly mapped to the correct physical address. Only part of the page table is stored in the TLB at a time (the TLB is of limited size, typically 8 to 2048 entries) and so the rest of the page table is stored in memory. On a TLB miss, then the address translation must be done using the full set of page tables stored in the system's main memory. Since most applications exhibit locality in how they access memory, most addresses can be found within the TLB without requiring an expensive lookup. This is another use of the 90/10 rule that was previously described.