Windows Internals Course – University of Tokyo – July 2003

Virtual Memory Exercises

Arun Kishan – 2003/07/18a

Virtual Memory

The Windows NT operating system features a large and highly complex virtual memory system. Like most traditional systems, NT features a demand paging virtual memory system that allows applications to use more memory than the machine may physically possess. NT also provides support for more advanced virtual memory management features such as large page mappings (contiguous blocks of physical memory mapped to contiguous blocks of virtual memory) and address space windowing (mapping physical memory directly to views in the user address space). This latter ability allows machines to access physical memory beyond the range directly addressable by the logical address bits.

Question: *What is the advantage of large page mappings? On systems for which a large page is equal to four large pages, when is it possible to satisfy a large page allocation request? What is a possible disadvantage of using large pages?*

For all user-mode processes on the system, NT maps itself into the upper portion of the address space. The exact demarcation between the user-mode and kernel-mode portions of the address space depends on the image options (whether or not it requests a larger user address space) and whether or not the operating system is operating in native 64-bit or 32-bit mode. For example, on a standard 32-bit installation of Windows NT, the user-mode component may consume the lower 2 gigabytes of the address space, and the kernel may consume the rest.

Question: *What is the advantage of mapping the kernel into the address space of every user process?*

NT uses a multi-level page table scheme to perform the virtual-physical address translation, with most all processors that host NT providing a hardware translation lookaside buffer (TLB) that caches the most recently used translations. This approach allows the page tables themselves to be paged, allowing for conservation of physical memory when an address space is particular sparse. Further, the page tables describing system space may be shared by the page directories for all the processes in the system. An interesting feature of the NT system is that it performs a recursive self-map of a process' page tables into a contiguous portion of the kernel-mode part of the process' address space (e.g., the table mapping the lowest four megabytes of the address space at address *z*, the table mapping the next four megabytes at *z* + *page size*, and so on), allowing the virtual address of a PTE providing the translation for any particular virtual address to be determined with ease.

Question: *On standard x86 systems, a two-level page table scheme is used such that each process has a page table directory (the size of a page), containing pointers to 1024*

page tables, each of which contains 1024 PTEs, each of which maps 4 kilobytes. Note that mapping all page tables into the address space of a process thus consumes 4 megabytes of address space. Suppose that the CR3 processor register contains the physical address of the page directory for a process, which is swapped during a context swap between threads from different processes. How can the page directory for a process be setup, using the value in CR3, such that all page tables for the process are automatically mapped to virtual address 0xC0000000, and will automatically be re-mapped to another process' page tables after a context switch? What might have been a possible motivation for developing the fast virtual address – PTE address translation afforded by this page table mapping scheme?

Page Fault Handling

As described above, NT provides a demand-fault mechanism for loading the pages into the physical memory of an executing process. Until the page-in request is satisfied, the faulting thread remains suspended, after which point it is resumed by the operating system. Any user-accessible address may be paged (i.e., may instantaneously have an invalid PTE in the relevant page table), though no page fault may be incurred at IRQL greater than or equal to DISPATCH_LEVEL, as a page fault operation may cause a deadlock in the paging path. As such, kernel-mode code must take care when accessing particular virtual addresses at elevated IRQL. User-mode code can freely access any address in user space, as all user code runs at PASSIVE_LEVEL.

Experiment 1: Find and execute the *tokyo_vm.exe* process. You will find that when executed on its own, it will appear to exit without generating any input. This is because the process is incurring a series of page access violations that are unhandled by the NT operating system. However, if executed via *TokOSLaunchFaultingProcess*, the caller may supply a page fault handler that is granted an opportunity to process each page fault as it occurs. You may assume that the target process will incur only two types of faults: demand-zero stack faults and demand-page code faults. By inspecting the CPU context of the faulting thread, the identity of the fault may be determined and handled accordingly. In either case, the first step is to build a mapping for the faulted page to physical memory; in this exercise this is equivalent to *committing* the relevant virtual address region using NT APIs. You may assume that this operation also zeros the corresponding physical page automatically. When copying a segment of code to the fault address, be sure to flush the instruction cache before exiting the handler (and thus resuming the faulting thread). If all page faults are handled successfully, *tokyo_vm.exe* will produce an output based upon the passed parameters. Observe the output of the program for various inputs. Do you recognize the function?

Question: *Immediately after writing dynamically generated code to memory or modifying already present code, one typically flushes the data cache and invalidates the instruction cache. Why is this?*

Shared Memory

In NT, *sections* or *file mappings* are kernel objects used to describe a shareable region of virtual memory. Sections and file mappings are used to describe both private shared memory and mapped files. Mapping a view of a section allocates a window of virtual address space in the target process and computes the appropriate virtual to physical mappings using the *prototype PTEs* associated with the section object itself. Shared

memory apertures allocated in this way can be used to map shared DLLs into the address space of multiple running processes (while consuming the requisite set of physical pages only once) or as a means of enabling efficient IPC. In the latter case, though the actual data transfer is relatively efficient, some sort of kernel synchronization mechanism is required to serialize access to the shared memory.

Question: *Suppose some user-mode lock is dependent only on interlocked operations to a single memory location and falls back to suspending threads using event objects in the event of contention, such that the thread that releases the lock may signal an event to indicate lock availability. Would allocating such a lock in a shared memory region automatically provide synchronization between competing threads from different processes? If so, explain how it would work. If not, explain some of the issues that could arise.*

Experiment 2: Develop a solution to the classic consumer-producer problem using a bounded buffer allocated in a region of memory shared between two processes (with separate console windows). Make sure that the shared buffer is large enough to accommodate at least 128 outstanding data entries before requiring the data producer to be obstructed. Since the problem is restricted to a single producer and a single consumer, note that no more than two semaphore objects are required for correct synchronization. For the data producer process, simply read each character as it is entered into the standard input console (without awaiting the carriage return that signals end of line) and places it into the circular communication buffer. To ensure key sequences such as Ctrl-C continue to operate as expected, defer the processing of more complex input forms to the OS when adjusting the input console mode for this portion of the experiment. Once the consumer process receives notice of generated data, it should remove available characters from the shared buffer and display them to the screen.