Operating System Feature Comperison: Memory Management

by Sinan Topkaya

CS 444 - Spring 2016

Abstract: This paper will examine how Windows and FreeBSD implements memeory management, later on there will be information about the comparison of these operating systems to Linux.

Writing Assignment - Memory Manager Comparison

Intro

Memory management is the act of managing computer memoery at the system level. The essential requirement of memory management is to provide way to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This paper will examine how Windows and FreeBSD achieves this.

Windows

Memory manager is part of the Windows executive and therefore exists in the file Ntoskrnl.exe. In Windows memeoery manager consists of various components such as, set of executive systems, trap handlers, balance set manager, process/stack swapper, modified, mapped page writer, segment and the zero page threads.[1]

Memory Usage

The memory and Process performance counter objects provide access to most of the dtails about system and process memory utilization. This can be viwed through Windows Task Manager. This shows the information on pysical memory usable by Windows, cached memory, memory that is freem paged memory, nonpages memoery.

Like other Windows executive services, the memory management services allow their caller to supply a process handle indicating the particular process whose virtual memory is to be manipulated. For example, if a process creates a child process, by default it has the right to manipulate the child process's virtual memory. Thereafter, the parent process can allocate, deallocate, read, and write memory on behalf of the child process by calling virtual memory services and passing a handle to the child process as an argument. This feature is used by subsystems to manage the memory of their client processes. It is also essential for implementing debuggers because debuggers must be able to read and write to the memory of the process being debugged.

Most of these services are exposed through the Windows API, it consists of three groups of functionsm heap function which is used for allocations smalled than a page, virtual memory functionsm which

operate page granularity and memory mapped file function, like createfilemapping, createafilemappingnuma, mapviewoffile, etc.. functions.

Large and Small Pages

The virtual address space is divided into units called pages. That is because the hardware memory management unit translates virtual to physical addresses at the granularity of a page. Hence a page is the smallest unit of protection at the hardware level. The processors on which Windows runs support two page sizes, called small and large.

Pages in a process virtual address space are free, reserved, commited, or shareable. Commited and shareable pages are pages that, when accessed, ultimately translate to valid pages in physical memory. Commited cannot be shared woth other processes, while shareable pages can be. In general, it's better to let the memory manager decide which pages remain in physical memory. However, there might be special circumstances where it might be necessary for an application or device driver to lock pages in physical memory. Pages can be locked in memory in the way, either with windows applications or device drivers.

Allocation Granularity

Windows aligns each region of reserved process address space to begin on an integral boundary defined by the value of the system allocation granularity, which can be retrieved from the Windows GetSystemInfo or GetNativeSystemInfo function. This value is 64 KB, a granularity that is used by the memory manager to efficiently allocate metadata (for example, VADs, bitmaps, and so on) to support various process operations. In addition, if support were added for future processors with larger page sizes (for example, up to 64 KB) or virtually indexed caches that require systemwide physical-to-virtual page alignment, the risk of requiring changes to applications that made assumptions about allocation alignment would be reduced.[1]

Finally once a region of address space is reserved, the operating system ensures that the suze of the region is a multiple of the system page size.

Memory Pools

Windows may also create memoery pools. The memory manager creates the following memory pools that the system uses to allocate memory: nonpaged pool and paged pool. Both memory pools are located in

the region of the address space that is reserved for the system and mapped into the virtual address space of each process. The nonpaged pool consists of virtual memory addresses that are guaranteed to reside in physical memory as long as the corresponding kernel objects are allocated. The paged pool consists of virtual memory that can be paged in and out of the system. To improve performance, systems with a single processor have three paged pools, and multiprocessor systems have five paged pools.

FREEBSD

In FreeBSD each process has its own private adddress space. This space is initally divided into three segments: text, date and stack. The text segment is read-only and contains the machine instructions of a program. The data and stack segments are both readable and writable. The data segment contains the initialized and uninitialized data portions of a program, whereas the stack segment holds the application's run-time stack. On most machines, the stack segment is extended automatically by the kernel as the process executes. A process can expand or contract its data segment by making a system call, whereas a process can change the size of its text segment only when the segment's contents are overlaid with data from the filesystem, or when debugging takes place. The initial contents of the segments of a child process are duplicates of the segments of a parent process.

To support large sparce address spaces, mapped files and shared memoery freeBsd calls mmap, that allows unrealted processes to request a shared mapping of a file into their address spaces. If multiple processes are mapped to the same file, it changes the protion of the adress space.

The kernel often does allocations of memory that are needed for only the duration of a single system call. In a user process, such short-term memory would be allocated on the run-time stack. Because the kernel has a limited run-time stack, it is not feasible to allocate even moderate-sized blocks of memory on it. Consequently, such memory must be allocated through a more dynamic mechanism. For example, when the system must translate a pathname, it must allocate a 1-Kbyte buffer to hold the name. Other blocks of memory must be more persistent than a single system call, and thus could not be allocated on the stack even if there was space. An example is protocol-control blocks that remain throughout the duration of a network connection.[2]

Conclusion

Both Linux and Windows have common features in between them, for example both operating systems use hardware abstraction later that is not system dependents so therefore the kernel can be coded independently, also if pages share same copy of the page is used between the processes. In both operating systems a file can be mapped onto memory, whcih then can be used for read/write instructions. For the above reasons both operating systems are actually really similar however they do use different functions and libraries and sometime different algorithms but to do the exactly the same thing. According to a forum from freebsd the memory allocation performance for FreeBSD is alittle bit better. Even if they both use the POSIX standards BSD uses almost half the memory because of this. This is usually caused by locks.

References

[1] D. S. Mark Russinovich and A. Ionescu, *Windows Internals, Part 1 6/e*. Microsoft Press, 2012.

[2] M. K. McCusick and G. V. Neville-Neil, *Design and Implementation of the FreeBSD Operating System 2/e*. Addison-Wesley, 2015.