CS444: OPERATING SYSTEMS II

Memory Management

Author:
Mark BEREZA

Instructor:
D. Kevin MCGRATH

November 19, 2017
Fall Term

Contents

1 Introduction 2

2 Virtual Memory and Pages 2

3 Page Fetching 3

4 Page Replacement 5

5 Conclusion 5

References 7

1 Introduction

Although caching can help reduce the number of times processes need to access memory, the fact remains that primary memory, as the name would suggest, is the memory used most by processes and the kernel itself. As a result efficient use of primary memory, or RAM, is vital to any modern operating system. This paper will analyze the approaches to memory management taken by FreeBSD and Windows and compare these strategies to Linux's version of memory management.

2 Virtual Memory and Pages

Two fundamental concepts in memory management, virtual memory and paging, are common to all three operating systems being discussed. In other words, Windows, Linux, and FreeBSD are similar in that they require that the hardware provide mapping between physical addresses and a set of virtual address that are actually used by the system. Additionally, all three operating systems are similar in the sense that the they segment the virtual address space into fixed-size blocks known as pages that map to pages in physical memory.

Increases in the size of memory available to modern systems have pushed the addressing capacity of 32-bit systems to their limits, since such virtual address spaces can only uniquely address up to 4GB of physical memory. For 32-bit systems, the 4GB of virtual address space needs to be shared between the kernel and the user processes and this is where one may see the first of many differences between these three operating systems when it comes to memory management. While Linux and FreeBSD both reserve (by default) 1GB of virtual memory to the kernel and the rest to processes, Windows splits virtual memory down the middle (by default) between the system and the user processes. This can change, however, for a process that is marked as large address aware (3GB) or if PAE is enabled (1.5GB) [1]. The difference in default behavior is likely a result of the Windows system because far less lightweight than Linux or FreeBSD and thus it would make sense for more memory to be held in reserve as a result.

Another complication resulting from advancements in memory hardware is the size of pages themselves. While 4KB has become the de-facto standard page size for all three operating systems, modern processors now support pages of size 2MB and 4MB. While all three operating systems has some level of support for these non-standard page sizes (called Huge pages in Linux, Super Pages in FreeBSD, and Large Pages on Windows [2], Windows seems to be the only one of the three systems to embrace the benefits of large pages, namely the reduction in TLB overhead associated with addressing large sections of memory, by integrating their existence into standard OS memory management. Linux and FreeBSD, on the other hand, like to maintain that page size is synonymous with 4KB. This difference is likely a result of the diverging end goals of these systems. FreeBSD is based largely on BSD, which itself is a product of academia and thus aims to be elegant and simple in design. Linux, in a similar vein, embraces singular solutions capable of addressing a wide array of use cases due to its need to run on a huge variety of hardware. Windows does not share these end goals and instead aims to simply maximize performance and usability for the most common use case in order to enhance the user experience and thus produce a better product. The added complexity needed to leverage the hardware and address all kinds corner cases is ultimately hidden from the end users due to the closed-source nature of Windows, so this is likely not a major concern for its developers.

Another example of this difference in complexity is how each of these operating systems handle page faults, or when a process tries to access a page that has not been mapped to physical memory yet. Linux handles page faults by causing the program in question to sleep until page replacement occurs or pages are swapped out. FreeBSD simply evicts entire processes based on a Least Recently Used (LRU) page replacement algorithm. The page fault behavior for Windows, on the other hand, varies widely depending on the context of the fault. For example, if the request page is not in memory but is on disk in the page file, the requested page is simply brought into memory. On the other hand, if the requested page is accessed from user space but is already allocated for the kernel, an access violation is raised [1].

3 Page Fetching

Two vital factors that any memory management strategy must address are when pages are fetched and when and how pages are replaced.

When it come to page fetching, all three operating systems use some form of the demand paging strategy. Pure demand paging means that new pages are only fetched when a page fault occurs and only the requested page is fetched. The difference arises in the fact that only Linux implements a pure demand paging strategy, whereas Windows and FreeBSD implement an approximation of demand paging with some additional anticipatory paging policies /citeComparison1.

upon a page fault, Windows will usually load the faulted page into memory, as would be expected from any demand paging strategy. However, Windows uses a version of demand paging known as cluster demand paging. As the name would imply, every time a faulted page would be loaded into memory, instead 1-8 pages (known as a cluster) are loaded in all at once. The additional page are often just before or after the faulted page, assuming that locality will generally mean that these pages will also be requested in the near future. Additionally, Windows implements an anticipatory fetching strategy with its logical prefetcher [1]. The logical prefetcher monitors the startup of the system or an application and records all page faults that occur. It then uses this information to prefetch pages it expects will be needed the next time the system or said application boots. Moreover, Windows also supports an optional feature known as SuperFetch [1], a form of proactive memory management. This in particular sets it apart from both Linux and FreeBSD because this system uses historical page usage and file access information along with some heuristics to prefetch pages the system expects the user will need in an intelligent manner. The sample code provided below demonstrates how a process' memory usage may be obtained in Windows [3].

```
void PrintMemoryInfo( DWORD processID ) {
    HANDLE hProcess;
    PROCESS_MEMORY_COUNTERS pmc;
    // Print the process identifier.
    printf( "\nProcess ID: %u\n", processID );
    // Print information about the memory usage of the process.
    hProcess = OpenProcess(  PROCESS_QUERY_INFORMATION |
                                    PROCESS_VM_READ,
                                    FALSE, processID );
    if (NULL == hProcess)
        return;
    if ( GetProcessMemoryInfo( hProcess, &pmc, sizeof(pmc)) ) {
        printf( "\tPageFaultCount: 0x%08X\n", pmc.PageFaultCount );
```

```
        printf( "\tPeakWorkingSetSize: 0x%08X\n",
                  pmc.PeakWorkingSetSize );
        printf( "\tWorkingSetSize: 0x%08X\n", pmc.WorkingSetSize );
        printf( "\tQuotaPeakPagedPoolUsage: 0x%08X\n",
                  pmc.QuotaPeakPagedPoolUsage );
        printf( "\tQuotaPagedPoolUsage: 0x%08X\n",
                  pmc.QuotaPagedPoolUsage );
        printf( "\tQuotaPeakNonPagedPoolUsage: 0x%08X\n",
                  pmc.QuotaPeakNonPagedPoolUsage );
        printf( "\tQuotaNonPagedPoolUsage: 0x%08X\n",
                  pmc.QuotaNonPagedPoolUsage );
        printf( "\tPagefileUsage: 0x%08X\n", pmc.PagefileUsage );
        printf( "\tPeakPagefileUsage: 0x%08X\n",
                  pmc.PeakPagefileUsage );
    }
    CloseHandle( hProcess );
}

int main( void ) {
    // Get the list of process identifiers.
    DWORD aProcesses[1024], cbNeeded, cProcesses;
    unsigned int i;
    if ( !EnumProcesses( aProcesses, sizeof(aProcesses), &cbNeeded ) ) {
        return 1;
    }
    // Calculate how many process identifiers were returned.
    cProcesses = cbNeeded / sizeof(DWORD);
    // Print the memory usage for each process
    for ( i = 0; i < cProcesses; i++ ) {
        PrintMemoryInfo( aProcesses[i] );
    }
    return 0;
}
```

Overall, the strategy used by Windows is far more complicated than the one used by Linux, which essentially performs pure demand paging with no prefetching. This should be unsurprising since it is common for Windows, which prioritizes performance for common use cases for the average consumer over code simplicity, to use complex strategies involving data access history and multiple methods of anticipatory fetching to gain a performance edge, perhaps to alleviate some of the sluggishness due to heavy-weight nature of the OS as a whole. Linux, which could accurately be described as a kernel that is catered towards developers, seems to prioritize the simplicity of a general strategy that can handle the majority or cases adequately. These priorities make sense for an open-source kernel that is maintained by a wide array of volunteer developers, and Window's strategy makes sense since its likely complicated source code is not the product they ship to customers.

FreeBSD's page fetching strategy could perhaps be described as a sort of middle ground between the Windows and Linux approaches. Although it does not utilize the complex historical usage heuristics of Windows, it does perform some level of anticipatory paging. FreeBSD, like Windows and Linux, uses the number and memory locations of page faults caused by a process over time to approximate its working set, or the subsection of the virtual address space that is relevant to the process at a give time period. By presuming locality of reference, the "phenomenon whereby memory references of a running program are localized within the virtual address space over short periods" [4], the system can prefetch pages adjacent to the faulted page within the current process' working set whenever a page fault occurs. This is

done to help minimize page faults and thus improve performance. Although this strategy is not much more complicated than Linux's pure demand paging, the choice to utilize some form of prefetching is likely due to FreeBSD advertising itself as being performance-conscious. While whether or not FreeBSD's claim can be substantiated is beyond the scope of this paper, it is useful to help understand its developers' decision to trade-off some simplicity in favor of better performance. The system doesn't take this trade-off nearly as far as Windows, and this likely because FreeBSD still ships its source code as its product, to an extent, since it is open source.

4 Page Replacement

The reason why page replacement is a very important aspect of any memory management strategy can be explained thusly:
"Obviously, many computers do not have enough main memory to retain all pages in memory. Thus, it eventually becomes necessary to move some pages to secondary storage (back to the filesystem or to the swap space). Bringing in a page is demand driven. For paging it out, however, there is no immediate indication when a page is no longer needed by a process. The kernel must implement some strategy for deciding which pages to move out of memory so that it can replace these pages with the ones that are currently needed in memory. Ideally, the strategy will choose pages for replacement that will not be needed soon." [4].

FreeBSD's implementation of page replacement is an approximation of the global least recently used algorithm, or global LRU. FreeBSD keep track of when pages are used and uses this data to decide which page(s) should be swapped out when available memory becomes low. The global qualifier applied to this algorithm for FreeBSD simply means that any page in use by any process is a candidate for replacement.

Similarly, Linux also implements a global LRU algorithm to determine which pages should be swapped out, but it has a caveat. Although all pages used by all processes are up for grabs, the Linux kernel separates pages into two queues: the active list and the inactive list. Pages stored in the active list are the kernel's best guess of the current working set and the inactive list contains all the other pages, which are available to be swapped.

Windows somewhat breaks away from this pattern by having implementations of both LRU and first in, first out (FIFO) algorithms [1]. The FIFO algorithm simply selects the page that has been in memory the longest to be swapped out. Moreover, Windows uses a combination of local and global strategies, where local simply means that the page to be swapped out is chosen only from the list of pages in use by the process that caused the page fault.

Once again, it appears that Windows is sacrificing simplicity of form for diversity of strategies that can handle different use cases efficiently. Linux and FreeBSD, which both have their roots in academia, seem to follow algorithms defined through research like LRU and demand paging more faithfully.

5 Conclusion

In conclusion, Linux and FreeBSD's approaches to memory management are quite similar, whereas Windows implements a lot of the same concepts (LRU, demand paging, virual memory), but in more nuanced and complex ways. As mentioned earlier, Windows prioritizes the performance of the end product over the organization or simplicity of the code base because of its closer-source nature. Thus, Windows developers are more than willing to put more burden

on the code base in order to improve performance in a variety of use cases. Linux and FreeBSD, on the other hand, utilize simple but effictive algorithms that work in the general case, thus keeping their open source code base simpler. Additionally, FreeBSD is willing to add some use-case specific improvements (like page prefetching) that Linux is not. This can likely be attributed to the fact that Linux aims to runs a larger range of hardware environments than FreeBSD and FreeBSD advertising itself as a performance-consious operating system.

REFERENCES

[1] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals Part 2*, 6th ed. Redmond, Washington: Microsoft Press, 2012.
[2] R. Coker, "Hugepages - debian wiki," https://wiki.debian.org/Hugepages, (Accessed on 11/18/2017).
[3] "Collecting memory usage information for a process (windows)," https://msdn.microsoft.com/en-us/library/windows/desktop/ms682050(v=vs.85).aspx, (Accessed on 11/19/2017).
[4] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Addison-Wesley Professional, 2014.