Writing Assignment 4

CS 444 Spring 2017

Author:
Brandon Dring

Professor:
Dr. Kevin McGrath

June 6th, 2017

Abstract

Here we are going to look at how different operating systems implement memory management. Specifically how Windows and FreeBSD do it and how they compare to the almighty Linux.

1 Introduction

There has been a lot of things that has changed in the past 50 years of computing. But, besides The Internet there isnt a greater advancement in technology than memory. From everything to access speeds, storage medium, reliability, price, and most importantly capacity/density. For example, I remember when I was a child my MP3 player had a 128MB SD for music. My phone has a 64GB microSD that was cheaper than the smaller SD card. And this wasnt over a huge time span, it happened within my memorable lifetime. The sheer importance of how a computer leverages, accesses and stores its data is essentially all that a computer does. With memory and storage becoming cheaper by the day, it paves the way for computers to to manipulate in new and exciting ways we have never seen.

2 FreeBSD

2.1 Overview

In FreeBSD every process will have its own private address space, in which it is divided into 3 segments: text,data, and stack. With the text segment being read only, containing instructions of the program, the data segment has data portions of the program, and the stack holding the runtime stack of a program[1]. Within the FreeBSD kernel, memory is managed on a page by page basis, through the `vm_page_t` struct. As with most things inside the kernel all Virtual memory functionality is stored within structs with the 'vm_ name preceding it.FreeBSD attempts to force pages into the L1 or L2 caches on the CPU. And monitors how often pages are used and decides whether to keep them in these small caches or kick them out. Like most 32 bit Operating Systems FreeBSD is limited to handle a memory configuration of just 4gb, but on a 64 bit system can go all the way up to 8TB.

2.2 Virtual Memory

Virtual memory and FreeBSD did not go hand in hand for quite some time, time implications have hindered proper releases of 4.X FreeBSD for nearly ten years. Over 40 companies and research groups had contributed just to implement shared memory spaces between processes.[1] To create a shared memory space between processes in FreeBSD 4.2, the `mmap()` function was created that allows processes to share files between address spaces. With any process having unlimited access to the file, and any changes to the file are shown throughout other processes that are using it as well. But all this was eventually scrapped to build the networking system. To create the virtual memory system it took over 10 years[1]. And in the end they settled on the design decision of storing the lion's share of the virtual memory on the disk instead of any RAM the computer has. This obviously generates a lot of extra disk traffic that the hard drive must deal with, thus slowing down the system for the user. On top of all that, there was also computer architecture specific issues, and lack of support for multi-CPUs [1]. In the end, after 10 years of attempted development in 4.4 FreeBSD the entire virtual memory system. Remedying all the ailments that had plagued prior 4.4 FreeBSD releases

```
#include <sys/mman.h>
     void *mmap(void *addr, size_t lengthint " prot ", int " flags ,
```

```
int fd, off_t offset);int munmap(void *addr, size_t length);
```

The complexity of creating a shared memory space in FreeBSD / Linux

2.3 Paging

Paging in FreeBSD can happen in a multitude of different way depending on the system load. If there is a heavy system load and lack of memory space the entire contents of the page will be swapped to secondary storage (i.e Hard drive). While if there is a modest system load, the paging system will implement a LRU-2 (Last Recently Used 2) system. Where the pages are broken up into two tiers between use frequency. And when the bottom tier becomes crowded, the last recently used page is thrown out. For the longest time FreeBSD pages were all one size, 4KB, it wasnt until FreeBSD 7.2 that they had introduced Superpages which were 2MB in size[1].

2.4 System Calls

System calls are relatively simple in FreeBSD, when a user makes a system call on a file like `read()` or `write()`, the FreeBSD Kernel switches into kernel mode and will copy the data into a buffer inside the kernel. Thus transferring the data from the user space into kernel space, and then switch back into user mode. Alternatively if there is a large quantity to transfer on a `read()` or `write()` call the kernel will instead remap the address space into the kernel instead of copying all the data over.

2.5 Memory Management/Functionality

Inside each process address space it has the stack space, but the stack size is based off of the bit size architecture of the operating system. Since the stack size is limited to create dynamic memory various system calls are implemented. To allocate memory inside the kernel FreeBSD uses the C traditional `malloc()` and `free()`. To create very large persistent dynamic memory allocations `zalloc()` and `zfree()` system calls are made.

3 Windows

3.1 Overview

Virtual address space is generally limited to how big the bit size is of the architecture. On a 32 bit Windows system the size can only grow up to 3GB per process, while a 64 bit system can have over 7TB[2]. Like most systems inside Windows, every piece of the kernel generally has a manager that overwatches and controls the general going on of its department. Likewise, Windows has a memory manager that overlooks paging, swapping and translating of its Virtual memory system to and from the disk.

3.2 Virtual Memory

The Windows virtual memory is kept in check by the Memory manager inside the kernel. Upon boot the memory manager creates two pools divided into the Nonpaged pool, and the Paged pool[2]. The Nonpaged pools focus on reliability, and are accessibly at any time, and there are no page faults.

While the Paged pools specializes in that it can be paged into and out of the system[2]. These memory pools are located in the system address space, but are mapped into the virtual memory space of each process. Which can then access, allocate or deallocate their memory accordingly. To give the illusion of shared memory Windows implements section objects that live in the process address space and map to virtual memory/pages needed by the process[2]. Then if a process needs a page that is already open, it simply just maps to the open page instead of pulling up its own personal page. From there the processes all just share the page, and any data that is altered in it.

3.3 Paging

Windows has divided its pages into two distinct and classical computer science names, being large pages which are 4MB and small pages 4KB. The memory management unit deals at the smallest level with a small page. Thus, the smallest unit of protection by the manager is at the size of a small page[2]. Within a large page over 1000 small pages can be stored, with an advantage of large pages being that when a byte is accessed inside a large page generally the entire page is loaded into cache. Because when part of a large page is accessed, generally the whole page is going to be accessed, as it is probably a large single file. Windows therefore leverages the hardware translation lookaside buffer for an instant reference of the bytes currently inside the buffer. But when done to small pages, each small page gets an entry in the lookaside buffer, cluttering the limited space[2]. As large pages are made of a multitude of small pages and they are all contiguous, so a large page requests may fail due to fragmentation of the system.

3.4 System Calls

To further understand how a system call to the memory manager works inside the Windows kernel, one has to further break down the Windows paging scheme. Windows pages are divided into free, reserved, committed/private, or shareable. With each definition coming exactly from its name. To allocate private pages there are Windows system calls such as VirtualAlloc(), and VirtualAllocEx(). The neat thing about reserving private pages, is that they come zero-initialized. Windows also makes extensive use of the heap, and each processes spawned will have their own heap. But to allocate dynamic memory instead of the classic malloc() call, instead Windows uses HeapAlloc(). To write data using the Windows API they also have a special function call that takes in 5 arguments called WriteFile()[3], and conversely to read its ReadFile().

```
BOOL WINAPI ReadFile(
    _In_       HANDLE      hFile,
    _Out_      LPVOID      lpBuffer,
    _In_       DWORD       nNumberOfBytesToRead,
    _Out_opt_  LPDWORD     lpNumberOfBytesRead,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
  );
```

Example of a simple ReadFile call on Windows[3]

3.5 Memory Management/Functionality

Windows also has a feature that it has called SuperFetch in which it assigns priority levels to a page based off of how many times it has been accessed in a specific time frame[2]. If Windows has

learned that a specific page will be used frequently it will both load the page on boot to reduce loading time. And keep it around in RAM to ensure that access time to it is instantaneous and not be pushed off into secondary storage. Windows also keeps tabs on what pages are active and being used. And should a page fault occur, Windows will load the page that failed into memory, and the pages preceding and following it[2]. As seen above, Windows allocates memory differently from FreeBSD.

4 Comparison to Linux

4.1 Virtual Memory

Here is where it gets interesting, there are some notable differences between the operating systems in how they implement virtual memory. Firstly, Windows of course has a manager that overwatches its Virtual memory department. And Windows has section objects within the process address space that maps page requests to a singular page somewhere in memory. And Windows Virtual memory is divided at boot into paged and non-paged pools, in which the only difference is how stable and persistent that pages in each of the pools are. And FreeBSD had a rough rough rough time getting started with their virtual memory system in that it 10 years to develop, and was eventually scrapped. Although in the end they do have a system call to map memory to multiple processes to share data. Which is very similar to Linux.

4.2 Paging

Overall each of the operating systems use pages somewhat similar. But what is different is that while FreeBSD and Windows both have zones (Groupings of pages). Linux is unique in that it has 28 possible zones that a page can have[4]. Which is ginormous compared to Windows and FreeBSD. Pages across all OSs also implement 2 different page sizes that are named very oddly. And are roughly 1000x the size of their small page counterparts. And page swapping to secondary storage occurs roughly the same way as well depending on system load, and what page was recently used.

```
struct zone {
        spinlock_t              lock;
        unsigned long           free_pages;
        unsigned long           pages_min;
        unsigned long           pages_low;
        unsigned long           pages_high;
        unsigned long           protection[MAX_NR_ZONES];
        spinlock_t              lru_lock;
        struct list_head        active_list;
        struct list_head        inactive_list;
        unsigned long           nr_scan_active;
        unsigned long           nr_scan_inactive;
        unsigned long           nr_active;
        unsigned long           nr_inactive;
        int                     all_unreclaimable;
        unsigned long           pages_scanned;
        int                     temp_priority;
```

```
        int                     prev_priority;
        struct free_area        free_area[MAX_ORDER];
        wait_queue_head_t       *wait_table;
        unsigned long           wait_table_size;
        unsigned long           wait_table_bits;
        struct per_cpu_pageset pageset[NR_CPUS];
        struct pglist_data      *zone_pgdat;
        struct page             *zone_mem_map;
        unsigned long           zone_start_pfn;
        char                    *name;
        unsigned long           spanned_pages;
        unsigned long           present_pages;
};
```

A look at how extensive Linux page zones can be.

4.3 System Calls

As one could guess the system calls between FreeBSD and Linux are pretty similar. It is just a simple `read()` or `write()` system call to read or write data from kernel space to and from user space. And to allocate memory it is just a simple `malloc()` and `free()`. Windows is where it gets fancy and there are certain uniquely named system calls to read and write file data, along with special ways in allocating page types.

Windows

```
LPVOID WINAPI HeapAlloc(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _In_ SIZE_T dwBytes
  );
```

FreeBSD/ Linux

```
#include <stdlib.h>

void *malloc(size_t size);
```

Windows Memory allocation system call vs Linux & FreeBSD

4.4 Memory Management/Functionality

Windows was ahead of the curve when it comes to memory management functionality. Windows had its SuperFetch ability in that it predicted what pages were going to be learned, and it keeps it in main memory for faster access. Linux followed suit and implemented a `readahead()` function to their systems, and to my knowledge FreeBSD has no such system.

5 Conclusion

There are several pros and cons to each operating system as to the ways they implement memory management. But in the end they all serve one purpose, which is allocate memory to store from the hard drive, and vice versa. Create a system to coordinate multi process memory usage, and lower data access time due to caching of disk data. Despite rather drastic differences in implementation, all provide a seamless experience to the end user. At no point does paging and caching, interprocess memory every become even noticeable. And the memory that these OSs are dealing with just keeps growing and growing but their resilient implementation methods still keeps chugging along. Because manipulating, storing, and protecting data is about as fundamental of a thing to what a computer does.

6 Citation

[1] M. K. McKusick, G. V. Neville-Neil, R. N. M. Watson, and M. K. McKusick, The design and implementation of the FreeBSD operating system. Upper Saddle River, NJ: Addison Wesley, 2015.

[2] M. E. Russinovich, D. A. Solomon, and A. Ionescu, Windows Internals, Part 2. Microsoft Press, 2012.

[3] Microsoft. "ReadFile Function." ReadFile Function (Windows). Microsoft, n.d. Web. 07 June 2017.

[4] Love, Robert M. "Linux Kernel Development." Zones. Sams Publishing, 12 Jan. 2005. Web. 07 June 2017.