Differences and similarities in memory management between the Linux, Windows, and FreeBSD Operating Systems

Alex Hoffer

CS 444

June 2017

Spring 2017

CONTENTS

I. Introduction

Memory management is a crucial aspect of any operating system. Memory management's primary functions are to provide virtual memory spaces to different processes, properly map these virtual memory spaces to their corresponding physical addresses, and allocate and de-allocate pages of virtual memory and assign such pages safely and responsibly to processes as they request them.

II. Windows

A. Similarities to Linux

1) Virtual memory addresses: Both systems support the use of 32-bit and 64-bit virtual memory addresses. [**?**] [**?**]

2) Hardware Abstraction Layer: Of course, a common similiarity that we encounter in the comparison of any two operating systems is their use of the *Hardware Abstraction Layer* (HAL). The HAL is beneficial to programmers because it handles system-specific operations and everything else in the kernel is built on top of it, which means kernel developers can implement things without being concerned about what system it's running on. Since we're discussing two widely used operating systems in Windows and Linux, the fact that both use a HAL is unsurprising, after all, they wouldn't be that popular if their kernel operations weren't portable to a myriad of different computers. In the context of memory management, the HAL lets the programmer in both operating systems to write methods that can do things like memory paging in a universal manner.

3) Data Structures: Naturally, the way in which kernels generally keep track of what segments of memory are occupied are through the use of data structures. There are some similarities in data structure usage to do this between these two systems. In Windows, the tree data structure is exclusively used, and each node of the memory tree consists of variables that are called *Virtual Address Descriptors* (VADs) [**?**]. These descriptors hold whether a given node is free to write to, occupied with data, or reserved to be filled with data. In Linux, a different data structure IS used by default (noted below in the "Differences" section, along with implications), but when this data structure reaches 32 items, it is converted into a tree [**?**]. This means that a programmer in both systems can access free, taken, and about-to-be-taken memory addresses using tree traversal methods. Here is the function signature for allocating memory to be placed in a memory tree in the Windows kernel: [?]

```
void __RPC_FAR * __RPC_USER midl_user_allocate (size_t cBytes);
```

Meanwhile, here is the function signature for allocating memory to be placed in either a memory tree or the data structure used when there are less than 32 items needed:

```
void * kmalloc(size_t size, int flags);
```

B. Differences from Linux

1) User space vs Kernel space memory allowance: In Linux, 3 GB of memory is allocated for the user space and 1 GB of memory is allocated for the kernel space, while in Windows, only 2 GB of memory is allocated for the user space and 2 GB is allocated for the kernel space [?] [?]. This is an intriguing difference, since it means that the Linux memory management system favors the user space, while Windows places equal emphasis on both. Perhaps this is because Windows expects its kernel to handle larger, more robust computations where additional memory space is necessary. Another difference is that while both systems support 32-bit or 64-bit virtual memory addressses, only Linux has both built-in. For Windows, using 64-bit virtual memory addresses is only possible through the use of the *Windows 64-bit Edition*.

2) Data Structures: As mentioned earlier, in Linux, if there are less than 32 items in a memory data structure, it is by default not a tree. It is instead a linked list. In this sense, memory management in Linux could be construed as being less rigid for the programmer than in Windows. Since linked lists are by reputation less efficient than trees, Linux likely switches to trees once the data structure hits the 32 item limit because if there are less than 32 items in a linked list, its difference in performance when compared to the tree data structure is negligible. But, since there are both trees and linked lists within the Linux memory management system, this means there are provided functionalities within the Linux kernel that aren't present in the Windows kernel related to linked list operations. Let's consider one that I encountered when Linux kernel hacking for Project 4 of Kevin McGrath's Operating Systems 2 course:

```
list_for_each_entry(sp, slob_list, list){}
```

Found in the slob.c file of the memory management folder in Linux, this method iterates through a list of pages as defined by the argument *slob_list* and places each passing page into the variable *sp* so the programmer can see what's going on a given slab of memory.

III. FreeBSD

A. Similarities to Linux

1) Paging system: Like in the section on Windows' memory management similarity to Linux, it's important to note that a major similarity between FreeBSD and Linux is their emphasis on virtualized memory. Within both kernels' paging systems, they use what is called the ”buddy system” when allocating memory at the page level [**?**] [?]. The ”buddy system” in this context is when a block of memory is divided, or *partitioned*, into smaller chunks to better fit the size requirements of memory requests. This means that our favorite Best-Fit algorithm from Project 4 of Operating Systems 2 could just have easily been written into FreeBSD. Additionally, both kernels' paging systems use *on-demand paging* and *swapping* [**?**] [?]. On-demand paging refers to a construct of virtualization where ”pages of data are not copied from disk

to RAM until they are needed". It's a kind of swapping, which is when you copy everything related to a process from main memory to an auxiliary type of storage. Of course, these are necessary methods from a memory management perspective since occupying a massive amount of main memory would greatly slow down computing speed.

B. Differences from Linux

1) Portability: Since FreeBSD runs only on x86 architectures, all computers running it must conform to its *Pmap module* [?]. The Pmap module is basically an abstracted layer that handles the physical hardware manipulations necessary to manage memory programmatically, although only with computers of the x86 architecture type. This means that FreeBSD's memory management system doesn't have a HAL of the breadth like we've seen with Linux (or even Windows!). Linux transforms architecture specific data structures for use with memory management into kernel page tables when code is compiled, giving its memory management system portability.

2) Allocating memory: We've seen what allocating memory in the kernel looks like in Linux and Windows, so let's take a look at how to allocate kernel memory in FreeBSD: [?]

```
void* malloc(unsigned long size, struct malloc_type *type, int flags);
```

Notice the additional argument separating it from its Linux equivalent. Interestingly, in FreeBSD, the call to allocate memory in Userland is basically the same as it is to allocate memory in the kernel.

3) Different memory allocator algorithms: One key difference programmers may notice between memory management in Linux and FreeBSD is the memory allocators they use. FreeBSD utilizes what is called the *zone* allocator, which is considered a *region-based* kind of memory management [**?**]. This means that modules in the kernel will tell the zone allocator the sizes of memory zones (think of these zones as chunks in memory) before running, and the zone allocator will split up memory to correspond to these zones. Then, when running, processes in need of memory are given these pre-sliced zones of memory. Linux, on the other hand, typically uses the *slab* allocator [?]. The slab allocator, which is like a version of the zone allocator that has been adapted to use methods that function essentially as constructors and destructors in object-oriented programming paradigms. Imagine that, object-oriented programming in the C language!