CS 444 - Spring 2016 - Writing Assignment 4

Cody Malick

malickc@oregonstate.edu

Abstract

Understanding how an operating system handles memory is key to understanding the overall flow of data through an OS. In this report, we will cover how Linux, Windows, and FreeBSD handle this important job in their respective kernels, and contrast them.

CONTENTS

I. Introduction

Memory management is the bedrock of any operating system used in the real world. Memory allows processes near instant access to the data storage they need, and in large quantities! How operating systems handle memory is a simple concept, but are expanded on differently by different operating systems. In this report, we will look at how different operating systems handle the cornerstone that is memory management, and compare them to each other.

II. Linux

Linux handles memory in fairly straightforward, simplistic way. It starts out with pages. Pages are the simplest unit of memory in the OS. The kernel is actually unable to address any unit of measurement smaller than a page. The kernel also divides the sum total of pages into zones. Finally, we'll talk about the slab layer, allowing easy allocation of large amounts of data.

A. Pages

Pages are the simplest and smallest addressable unit of memory usable by the kernel. The size of the page is dependant on the architecture the OS is running on. If it is a thirty-two bit system, the size of the page is usually four kilobytes, while a sixty-four bit system usually has a size of eight kilobytes. [1]

It's important to point out that the kernel does not directly manage the physical pages on the hardware. This is left to the MMU (memory management unit). The kernel's job is to communicate with the MMU in sizes it understands (pages), and work together with the hardware to get the job done.

The page struct is very short, and simple. This is important because it is used constantly, and you don't want a large structure you have to pass around constantly. [1]

```
struct page {
    unsigned long flags;
    atomic_t _count;
    atomic_t _mapcount;
    unsigned long private;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
    void *virtual;
};
```

We will briefly go over what the important fields in the struct do. First is the `flags` field. This field simply describes the status of the page. This includes describing if the page is locked or dirty (memory that may need to written to disk). `_count` stores the number of references using a given page. When the count is greater than negative one, then the page is still being used, and should not be reused. The caching mechanism in Linux uses the `private`, `mapping` variables to use the page as a caching point. The `private` variable indicates that the memory pointed to is private, and the `mapping` variable holds an address space being used by the page cache. Lastly, `virtual` holds the virtual address that points at the physical page. This field is particularlly important to users, as it enables easy and quick use of the memory space.

Now that we know how Linux handles its basic memory unit, lets look at how it classifies these into zones.

B. Zones

Linux uses zones to divide memory pages into subsections in order to deal with specific hardware limitations. Specifically, some hardware devices can only directly access certain physical memory, and some operating systems can map more memory than they actually have. This is called virtual memory. There are four primary zones in Linux:

`ZONE_DMA`: Memory that is directly accessed by hardware, bypassing the MMU.

`ZONE_DMA32`: The same as `ZONE_DMA` but restricts direct memory access to 32-bit devices.

`ZONE_NORMAL`: This contains the standard size page, with standard mapping. Should not allow DMA. This is where general use memory lives.

`ZONE_HIGHMEM`: Contains memory wich is not permanently mapped to the kernel's address space.

[1] Next up is the slab layer, which allows the quick and simple allocation and deallocation of memory for data structures.

C. Slabs

Linux has a slab layer that facilitates a very important function: the quick and easy allocation and deallocation of memory for data structures. The slab layer maintains someting called a *free list*. The job of the free list is to hold memory that's already been allocated for the purpose of quickly allocate a data structure's memory. This is done for data structures that are frequently used, and provides a nice performance increase. Essentially, the slab layer caches data structures. [1]

Know all of this, next is examining how exactly all this wonderful infrastructure is used, with basic allocation and deallocation of memory.

D. Basic Allocation and Deallocation

1) Allocation: Allocation is the bedrock of using memory. If you couldn't put something there, it would be useless! Here are the basics of allocating memory in Linux. At the simplest level, Linux provides the `alloc_pages()` function. This function allows you to allocate two to the nth power pages, where `order` is a parameter of `alloc_pages`. Here is an the actual prototype of the `alloc_pages` function:

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

`gfp_t gfp_mask` is a flag that is needed to get pages from memory. `gfp` stands for `__get_free_pages()`, and the different flag types allow for allocation of memory in specific ways.[1] `order` is simply the number of pages we want allocated as a power of two.

Although it is a useful function, as users, we more often want to obtain memory in terms of bytes. We can do this in the kernel by using the `kmalloc()` function. This function is simliar to `malloc()`, but it has a flags parameter that is identical to the `gfp_t gfp` flag from `alloc_pages`. `kmalloc` is very useful if we know the exact size of the structure we are allocating for using the C function `sizeof()`.

2) Deallocation: Deallocation is equally as important as allocation. We need to be able to free the space we allocated in order to reuse it! Here are some basic deallocation functions that are used in Linux. To directly free a page, you can use any of the following functions:

```
void __free_pages(struct page *page, unsigned int order)
void free_pages(unsigned long addr, unsigned int order)
void free_page(unsigned long addr)
```

Any of these functions would free the pages allocated with `alloc_pages()`.

The counterpart to `kmalloc()` is `kfree()`. You can pass in the object that you have given memory to, and the `kfree()` function will return that piece of memory back to the pool of available memory.

E. Virtual Memory

Linux, as other major operating systems do, gives each of its processes a virtual memory space to operate out of. This simplifys things on many levels for the developer on that OS, but also in the management of what process is using what bit of memory at a time. Instead of having each developer and each program written worry about what memory they are directly accessing, they instead work with a set of virtual memory that they can play around with, and blow things up in.

In Linux, we can allocate virtual memory using `vmalloc()`. This function is the same as `kmalloc()` or even C's classic `malloc()` function, but the major difference is that it allocates a virtually continguous set of memory, but not gaurenteed contiguous virtual memory. This makes things much easier for the average developer, as they do not need in-depth knowledge of how the OS works with memory in order to write programs for that platform.

Linux keeps things fairly straightforward and easy to understand in its memory management layer. Next, we will examine how Windows and FreeBSD implement the same interfaces to handle memory.

III. Windows

Windows does things vastly differently than Linux or FreeBSD. Windows implements a large module called the Memory Manager to handle all things memory. The memory manager's job is, specifically, to manage allocation and deallocation of virtual memory. This layer handles dealing with each process in their virtual memory space and mapping each to their respective physical memory.

A. Memory Manager Components

Memory management at the physical level in Windows is simliar to Linux in that it works with a MMU to do the physical reading and writing from disk. The big differences that we start to see are when we look at the software implementation of Windows memory manager.

The memory manager has several important components that keep it functioning as a whole. Following are a brief description of what each of those pieces are, and what they do: [2]

A set of system services that allocate, deallocate, and manage virtual memory. These exist in the kernel.

A fault handler for memory management exceptions

Six top level routines running in six seperate threads that compose the active memory management protocols

`balance set manager`: Responsible for overall memory management policies

`process/stack swapper`: Performs process and kernel thread stack inswapping and outswapping.

`modified page writer`: Writes dirty pages to disk.

`mapped page writer`: writes dirty pages in mapped files to disk.

`segment dereference thread`: Responsible for cache reduction and page file growth and shrinkage.

`zero page thread`: Responsible for zeroing out pages in memory for reuse.

Each of these pieces of the module make up the Windows memory management system. These different memory management functions are similar to how Linux has different flusher threads to help manage what gets written to disk and when.

Although Linux does have flusher threads, it contrasts the design philosophies behind the two different OSes greatly when you look at how the modules were put together to manage the different systems.

B. Virtual Memory

As stated above, the virtual memory space in Windows is managed by the balance set manager. The balance set manager is responsible for driving all access to virtual memory. Each process gets its own chunk of two gigabyte memory on thirty-two bit systems, and four gigabytes on sixty-four bit systems. In virtual memory, a process can grow to roughly eight thousand gigabytes in virtual memory space on sixty four bit systems.[2]

As you can see, Windows does things fairly differently than Linux. It has a much more hands-on approach to managing memory, and actively has up to six different threads policing memory at a time. Following this, FreeBSD will be compared to Linux, and the differences will be much less vivid than Linux to Windows.

IV. FreeBSD

FreeBSD often shares very simliar implementations with Linux. In terms of memory management, it is still the case. Memory allocation and deallocation are comperable.

A. Pages

At the bottom of FreeBSD, we still have pages. Pages are the smallest addressable unit usable by the FreeBSD kernel. And again, FreeBSD interacts and works with the onbaord MMU to manage memory.

B. Memory Allocation and Deallocation

FreeBSD implements a general use memory manager interface simliar to the C functions `malloc()` and `free()` functions to help manage its memory. Specifically requesting a certain number of pages does not seem to be a feature of the FreeBSD kernel. Linux also uses simliar implementations that are almost identical to `malloc`. This is not very surprising as Linux and FreeBSD stick very closely to their C based roots. [3]

C. Virtual Memory

FreeBSD also provides a virtual address space for processes to execute in and provide the quality of life improvement to not need to directly manage memory. FreeBSD implements its virtual memory system very much like Linux. It gives each process a large section of memory entirely belonging to the process. Then it manages where the memory gets allocated and attached to physical memory.

As we can see, FreeBSD has an even more straightforward memory management system than Linux. As surprising as this is, it makes developing services and applications for this OS very simple as far as memory management goes.

V. Conclusion

Memory management is the bedrock for any operating system. Understanding how they work, and how they work with virtual memory to make developer's lives easier, is important to developing modules and programs for them.

VI. Appendix A - Linux Structs

All following structs have been pulled from the classroom text. [1]

```
for future code samples
```

VII. Appendix B - Windows Structs

```
for future code samples
```

VIII. Appendix C - FreeBSD Structs

```
for future code samples
```

IX. Bibliography

References

[1] R. Love, *Linux Kernel Development*, 3rd ed. Developer's Library, 2010.

[2] M. R. D. Solomon and A. Ionescu, *Windows Internals Part 2*, 6th ed. Microsoft Press, 2012.

[3] FreeBSD. (2016, jan) 2.6. io system. [Online]. Available: https://www.freebsd.org/doc/enUS.ISO8859-1/books/design-44bsd/overview-io-system.html