# CS 444 - Spring 2016 - Final Paper

Cody Malick

malickc@oregonstate.edu

**Abstract**

Understanding how an operating system functions, from handling user input, to managing memory, is important for any aspiring kernel developer to understand. In this report, a basic overview of many topics that are fundamental to operating system understanding are covered. Each topic is covered for Linux, Windows, and FreeBSD, then contrasted.

## Contents

## I. Introduction

Operating systems are complex beasts. Often they scare away developers that might be interested in working with them simply from intimidation or lack of fondness for C based languages. I have discovered this term that, although operating systems have many facets and complexities, many of the topics are interesting and easy to study and tweak. This paper covers many of the operating system subsystems studied this term. This includes processes, process scheduling, interrupts and interrupt handlers, memory management, and file systems.

## II. Processes and Scheduling

### A. Introduction

The first topic is that of processes and process scheduling. This section contains descriptions on how each of the operating systems, Linux, Windows, and FreeBSD handle processes, threads, and CPU scheduling. A lot can be learned about the design decisions of the minds behind the operating system by how they handle these core constructs. We will start by describing how Linux handles these, as Linux is the focus of this class.

### B. Linux

*1) Processes:* The first thing that should be defined, is simply what a process is: ”A *process* is a program (object code stored on some media) in the midst of execution.”[**?**] Although the Linux kernel refers to processes as tasks, in this section (for uniformity) we will only refer to them as processes.

A process in Linux is ultimately born, somewhere up the line, from the `init` process. The `init` is the absolute root of every process tree on any Linux machine. It has a process ID of 1. All processes are spawned from either the `fork()` or `exec()` function calls. `Fork()` creates an exact copy of the calling process as a child of the process, while `exec()` loads a new executable into the process space and executes it. These are the only ways new processes are created in Linux.

Each process has a process descriptor that describes every aspect of the process, the `task_struct`. The `task_struct` is contained in `<linux/sched.h>`. `Task_struct` describes the current state of the process, the parent process ID, the process ID, exit code, priority, etc. Everything you could possibly want to know about a process in Linux is in that struct. I’ve included the `Task_struct` as Appendix A as it is a very large structure.

Processes, of course, must have states of execution. On Linux, processes have five states, each represented by flags in the `task_struct`:

`TASK_RUNNING`: The process is either running or in a run-queue ready to be run (run-queues will be discussed in the CPU scheduling section).

`TASK_INTERRUPTIBLE`: the process is sleeping (blocked) waiting for a condition to un-block. Once the condition is met, it transitions to `TASK_RUNNING`.

`TASK_UNINTERRRUPTIBLE`: The same state as `TASK_INTERRUPTIBLE` with the difference being that it does not wake once an interrupt is received.

`_TASK_TRACED`: This flag shows that the process is being traced by another process. An example is GDB.

`_TASK_STOPPED`: Process execution has halted and it is not possible to start running. This state is a result of a halting interrupt from the CPU, or if a debugger sends it a signal.

*2) Threads:* As with processes, we will define what a thread is: “Threads of execution, often shortened to *threads*, are the objects of activity within the process.” [?] In Linux, threads are treated fundamentally the same as processes. The only difference is that they have a shared address space, filesystem resources, file descriptors, and signal handlers. Here is the `thread_info` struct:[?]

Listing 1. The Linux thread_info structure is how Linux tracks essential information about each thread and process

```
struct thread_info {
    struct task_struct      *task;
    struct exec_domain      *exec_domain;
    unsigned long           flags;
    unsigned long           status;
    __u32                   cpu;
    __s32                   preempt_count;
    mm_segment_t            addr_limit;
    struct restart_block    restart_block;
    unsigned long           previous_esp;
    __u8                    supervisor_stack[0];
};
```

Compared to the `task_struct`, this is a relatively simple data structure. It has a pointer to the `task_struct` that is unique to that thread, along with a few extra flags and sets of information that separate threads from processes.

*3) CPU Scheduling:* Linux ships with a few different schedulers by default, including a real time scheduler.[?] The default scheduler for Linux is the CFS, or the Completely Fair Scheduler. The CFS is only fair in so far as it tries its best to split the processors time into equal bits, 1/n, where n is the number of processes currently running. The CFS weights this with process priority, or nice value, and the higher the weight the larger percentage of processor time that process gets. So it's not completely fair, but that is the idea behind it.

One of the major problems with this system is that a very large chunk of processes creates incredibly small slices of CPU time. For example, if there were six-thousand processes and/or threads that all had the same weight, then the cpu slice for each process would be 1/6000. There is a point where that slice is not large enough to even complete the context switch required to start executing the process again. The CFS handles this by implementing a floor on slice size, known as *minimum granularity*.[?] With this system, every process will always have a minimum amount of run time, but the system can become very slow once this size is reached.

## *C. Windows*

Windows is a very different beast. The process and thread structure are quite a bit different from Linux. The CPU scheduler Windows uses has the same basic idea as the CFS, but varies greatly when it comes to how it handles priority and division of CPU time.

*1) Processes:* Processes in Windows are similar to Linux in only one way: the both have a process ID. Otherwise, processes in Windows handle very differently than Linux. Here is the basic process struct in Windows, `_PROCESS_INFORMATION`: [?]

Listing 2. The Windows _PROCESS_INFORMATION Structure is how Windows tracks some important information about each thread and process

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

The `HANDLE` data type is a 32-bit unique identifier stored in the Windows kernel to identify individual processes.[**?**] The `hThread` points to the primary thread a process is executing on. These first two of these identifying fields are used internally by the kernel to specify what functions are performing operations on the process and thread objects. The second set of identifiers, `dwProcessId` and `dwThreadId` are used as unique identifiers for the process and thread themselves. These are the equivalent of the PID used by the the Linux `tast_struct`.

Process creation is also different in Windows. Instead of the constantly traceable tree of child-parent relationships, a function called `CreateProcess()` is invoked, resulting in a new process completely independent of the parent process. [**?**]

Process state in Windows is determined by thread state. Here are the possible thread states in Windows:[**?**]

`Initialized`: A state that indicates the thread has been initialized, but has not yet started.

`Ready`: A state that indicates the thread is waiting to use a processor because no processor is free. The thread is prepared to run on the next available processor.

`Running`: A state that indicates the thread is currently using a processor.

`Standby`: A state that indicates the thread is about to use a processor. Only one thread can be in this state at a time.

`Terminated`: A state that indicates the thread has finished executing and has exited.

`Transition`: A state that indicates the thread is waiting for a resource, other than the processor, before it can execute. For example, it might be waiting for its execution stack to be paged in from disk.

`Unknown`: The state of the thread is unknown

`Wait`: A state that indicates the thread is not ready to use the processor because it is waiting for a peripheral operation to complete or a resource to become free. When the thread is ready, it will be rescheduled.

This is interesting for a few reasons. First, there is an 'unknown' thread state. In Linux, it is not possible for a thread or process to be in an unknown state, only the ones listed. Next, it's interesting to see how Windows has so many more states than Linux. One of the differences that sticks out most is that Windows has a state when a process is queued to enter the processor in the `Standby` state. By having all these different states available, you can really see the real time state of the process, as apposed to Linux where you have fewer states that cover less situations. Then again, this could just be a result of Linux having better processor and thread management, requiring less states.

*2) Threads:* Threads, in Windows, are handled as subcomponents of processes. They are grouped into thread pools, where multiple threads execute under one process. MSDN explains the idea behind this structure is to reduce the number of application threads, and provide management of worker threads. Threads are are spawned through the `CreateThread()` function call.[**?**]

*3) CPU Scheduling:* The Windows CPU scheduler, by default, schedules processes in much the same way that the CFS schedules processes. A process is given the default weight, and the scheduler tries to divide the CPU time evenly among the running processes. Once we introduce weights or priority, the scheduler's behavior is very different.

While the system is assigning CPU time slices to each process, if a higher priority thread becomes available, the CPU stops the current running process, mid-time slice, and starts running the higher priority threads.

This is vastly different from the CFS in that regard. The CFS will allow a current running thread to complete it's time slice before moving it out of the processor. By introducing this kind of behavior, the power of priority states becomes glaringly clear.

Windows has thirty-two levels of priority, each divided into subdivisions:

```
IDLE_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
REALTIME_PRIORITY_CLASS
```

Each of these classes has subdivisions inside of them, seven in each, that has steadily increasing higher priority, starting from one, rising to thirty-two. As a side note, the zero priority class is reserved for a thread that goes through and zeros out memory when a process terminates.[**?**] Besides this large difference in priority, the two CPU schedulers are very similar.

## *D. FreeBSD*

FreeBSD is an open source OS based off Unix, specifically the Berkeley variant, BSD. It looks and feels similar to Linux in a few ways, but has some subtle differences in process and thread structure. The scheduler is quite a bit different from the CFS.

*1) Processes:* Processes in FreeBSD are very similar to Linux's. Similar to Linux's process struct, `task_struct`, the FreeBSD `proc` struct is large and contains a lot of information. See Appendix B if you are interested in examining the structure. Processes are created using the same forking and exec process that Linux uses. After a process is forked, a new process that is identical, except for the PID, is created. The process has a copy of the parent's resources and addresses space.

Process states are handled in a much more simplified manner than either Windows or Linux in FreeBSD. Instead of having eight or five different states for a process, it only has three:

`NEW`: Process is undergoing creation

`NORMAL`: Thread or threads will be runnable, sleeping, or stopped

`ZOMBIE`: The process is undergoing process termination

As per above, `NORMAL` embodies three subdivisions, runnable, sleeping, or stopped, but overall the classification is simplified. [**?**]

*2) Threads:* A notable difference between Linux and FreeBSD is that it does not treat threads and processes the same way. Threads are handle in a more similar fashion to Windows. Each thread is attached to a parent process. If multiple threads are requested for one process, new processes are spawned, but they have the same PID. When a user looks for a process in the OS, they will only see the single entry representing the parent process.

Thread states are dictated by the list in the processes section, but have a caveat. When a thread is not running on the processor, which there can only be one of, all threads are in one of three queues: the run queue, the sleep queue, or the turnstile queue. As the names dictate, the run queue contains threads that are in a runnable state. Threads that are blocked and are awaiting events are in the sleep queue or the turnstile queue.

The turnstile queue is exclusively for short blocks on threads. In FreeBSD, short blocks are exclusively the result of read/write blocks. Because these kinds of blocks are common, each thread creates a turnstile queue when it is initialized, and creates a list of all threads that are blocked because of a read/write block on a certain piece of memory. When that block is lifted, the turnstile queue manages the order in which the blocked threads are resolved. Allocating turnstile queues on each thread as apposed to each lock results in lower memory usage by the kernel.

Sleep queues are similar to turnstiles in that they contain lists of threads where blocking is occurring, but the sleep queue is exclusively for medium to long term locks that can send interrupts if the lock is held for too long. [**?**]

*3) CPU Scheduling:* The default CPU scheduler for FreeBSD is the ULE scheduler. ULE is not an acronym. [**?**] The scheduler is contained in the namespace `syskernsched_ule.c`, and if you remove the underscore in the namespace, you will see the reasoning behind the name.

The ULE scheduler is split into two parts in FreeBSD: a low-level scheduler, and a high-level scheduler. The low-level scheduler runs often, and the high-level scheduler runs a few times a second.

The low-level scheduler runs extremely often, whenever a thread is blocked. When a block occurs, the low-level scheduler pulls the highest priority process from a set of run queues. These queues are organized from low to high priority. The high-level scheduler decides what the priority of each thread is, and sorts it into the appropriate run queue. Each core has it's own set of these run queues to prevent two cores from trying to access the same queue at the same time.

The different cores run a round robin system on the threads in the queues. Each thread gets equal runtime in the form of time quantums. These time quantums are the same as time slices. If a process uses an entire time quantum, it is moved to the back of the the queue that it came from, and a context switch occurs to the next highest priority thread.[**?**]

### *E. Summary*

Linux, Windows, and FreeBSD all have their own ways of handling processes, threads, and scheduling. Linux and FreeBSD share some of the same basic ideas with process and thread structure, while Windows takes an entirely different approach with thread pools and its process priority structure. Threads, processes, and process scheduling are all critical portions of an operating system. They lay the foundation for the usability we enjoy in our computers today. Next to be dissected is the interrupt structure of each operating system.

## III. Interrupts

### *A. Introduction*

As we saw in the last section, threads and processes are the mainstay of an operating systems ability to operate and complete tasks. At a more fundamental level, we have interrupts. Interrupts are how hardware communicates with the CPU, letting it know it has information of some kind to share with it. Once the interrupt is received, the CPU interrupts the kernel, handing off the information it has. Interrupts are taken care of by interrupt handlers. This section will examine what the structure of interrupts are in Linux, Windows, and FreeBSD, the general procedure of how they are handled, and where the interrupt handlers live in their respective OS.

## B. *Linux*

Linux has a fairly straightforward way of handling interrupts. It has the interrupt structure of top and bottom halves, and each type of interrupt is registered to a unique handler in the kernel. The handler is defined by the driver, and is called when its unique handler ID appears.

*1) Interrupts:* Interrupts are how a piece of hardware communicates with the CPU, letting it know that it has some data ready for it. An interrupt must have an interrupt handler associated with it in order to have the communication handled properly. These interrupt handlers are contained in a devices driver. This is pretty universal across operating systems. What we are going to look at specifically is the structure of interrupts that Linux handles. [**?**]

*2) Interrupt Structure:* Interrupts are split into two parts in Linux: top halves, and bottom halves. These names are quite misleading as top halves and bottom halves are not halves at all. The top half is usually closer to a fifth or even an eighth. The reason for this is that interrupt handlers have to be very fast! Interrupts, as they are aptly named, stop everything the CPU is doing in order to communicate with hardware. When this interrupt is called, all the code in the top half of the interrupt has to be handled extremely quickly. Because of this, only time sensitive information can be stored in the top half, and all information in the top half is handled by the interrupt handler.

The bottom half, on the other hand, can contain any extra information needed to process the request from the hardware. Because this information is not deemed time critical, it is queued up with the other requests that the CPU needs to handle and is put off until its time to be processed has come. [**?**]

*3) Drivers:* Drivers are where interrupt handlers exist in Linux. A handler ID is reserved by the system when a driver is installed. When an interrupt is called, the kernel goes and finds the appropriate handler, and calls the function to resolve the interrupt.

Here is an example of how an interrupt handler is requested by a driver:

Listing 3. An example of Linux' interrupt handling process

```
if ( request_irq ( irqn ,  my_interrupt ,  IRQF_SHARED, "my_device", my_dev)) {
    printk (KERN_ERR "my_device:␣cannot␣register␣IRQ␣%d\n", irqn);
return  -EIO;
}
```

In the above example, the requested interrupt line is `irqn`, the interrupt handler `my_interrupt`, and the device `my_device`. If the request is handled properly, it returns zero, otherwise it returns the code prints an error and returns an error code, `-EIO`, an IO error. [**?**]

Next we'll look at how Windows handles these same ideas, and explore what interesting differences there are.

## C. *Windows*

Windows has a similar setup to Linux only in that drivers are where interrupt handlers are defined. There are some interesting differences in what device drivers are responsible for, and how fundamental system IO is handled in Windows as compared to Linux.

*1) Interrupts:* Windows has two different types of interrupts: general IO interrupts, and exceptions. Interrupts are asynchronous while exceptions are synchronous. Windows refers to catching these interrupts as traps, as they cause the kernel to differ from normal paths of execution. A trap is the equivalent of an interrupt handler. [**?**]

*2) Drivers:* Drivers are important in every OS, but in Windows they have a very specific structure. Drivers have a few responsibilities besides handling interrupts as they do in Linux. They have the responsibility to start IO requests to the hardware, handle plug-and-play behavior, cancel IO routines, unload routines, and more. Drivers are much more of entire modules instead of specifically interrupt handlers as they are in Linux. [**?**]

This view of drivers makes the handling of IO much more of the drivers responsibility than it does in Linux. In Linux, the driver simply handles the interrupt and hands other information off to the OS. In Windows, the driver is more robust and has more responsibility on how things are communicated.

Now that we've explored the basics of interrupts in Windows, we can look at FreeBSD, and see what similarities and differences there are in its core structure.

### *D. FreeBSD*

FreeBSD shares a very similar interrupt structure and handling pattern with Linux. There are a few minute differences that should be pointed out.

*1) Interrupts:* Interrupts are handled almost identically to Linux's interrupt setup. Each interrupt must be registered via a device driver. FreeBSD has two types of interrupts: hardware interrupts and software interrupts. These are the equivalent of Linux's top and bottom halves. If a piece of information is time critical, it is put in a hardware interrupt, otherwise they are handed to a software interrupt. [**?**]

*2) Interrupt Handler:* Interestingly enough, interrupt handlers are also called traps in FreeBSD, the same as it is in Windows. Although they're name is the same, they handle fundamentally different structures. The FreeBSD interrupt handling model is almost identical to the Linux model. Interrupts are handled asynchronously, and have to be registered with a unique ID so that the kernel can find which driver needs to handle the interrupt. [**?**]

### *E. Conclusion*

Interrupts are a key part of any operating system. It is a fundamental in handling IO from external or internal devices. As apposed to other topics we've covered, it's interesting to see that the three operating systems have a lot in common in this area, as opposed to their individual unique approaches to other parts of the kernel. Next up is memory management and how that challenge is solved.

## IV. Memory Management

### *A. 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 section, we will look at how different operating systems handle the cornerstone that is memory management, and compare them to each other.

### *B. 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.

*1) Pages:* Pages are the simplest and smallest addressable unit of memory usable by the kernel. The size of the page is dependent 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. [**?**]

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. [**?**]

Listing 4. The Linux page struct - the primary descriptor and basic unit of memory

```
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 particularly 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.

*2) 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.

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

*3) 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 something 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. [**?**]

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

*4) Basic Allocation and Deallocation:*

*5) Allocation:* Allocation is the bedrock of using memory. If you could not 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:

Listing 5. The alloc_pages function allows manual allocation of requested pages

```
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.[**?**] `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 similar 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()`.

*6) 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:

Listing 6. Void pointers to the free pages 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.

*7) Virtual Memory:* Linux, as other major operating systems do, gives each of its processes a virtual memory space to operate out of. This simplifies 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 contiguous set of memory, but not guaranteed 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.

## C. 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.

*1) Memory Manager Components:* Memory management at the physical level in Windows is similar 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: [**?**]

- 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 separate 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 operating systems greatly when you look at how the modules were put together to manage the different systems.

*2) 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.[**?**]

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.

### D. FreeBSD

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

*1) 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 on-board MMU to manage memory.

*2) Memory Allocation and Deallocation:* FreeBSD implements a general use memory manager interface similar 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 similar 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) 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.

### E. Summary

Memory management is the bedrock for any operating system. Understanding how it works, and how it uses physical memory and virtual memory together to make developer's lives easier is important to developing modules and programs for them. Next is the last section on file systems.

## V. File Systems

### A. Introduction

As a general user, you don't want to have to deal directly with a computers input and output systems. It would be tiresome and tedious. To solve this problem, the file system and virtual file systems are wrappers around these arduous tasks that make it simple, and user friendly. The VFS is the unification of two ideas: making the user's life easy, while providing a generic interface that allows different types of file systems the same interfaces. Following is an examination of the FS and VFS implementations in Linux, Windows, and FreeBSD.

### B. Linux

*1) File Systems:* A file system, simply, is an abstraction of the IO layer that allows ease of use for the user in tracking, reading, and writing to disk. Instead of forcing the user to remember where a file is, what its size is, what type of file it is, etc., the file system provides interfaces that make this a, relatively, easy task. There are many different types of file systems. They are differentiated by minimum and maximum file size, journaling capability, and maximum partition size.

*2) Linux Default File Systems:* There are a few default file systems available in Linux. A few of them are all from the `EXT` family, like `EXT2`, `EXT3`, and `EXT4`. `EXT3` is considered the standard FS for stable Linux builds, but `EXT4` is the latest and greatest release from that family. The biggest difference between `EXT4` and its previous version is the maximum file and partition size.[**?**] One the great things about Linux is the number of options for file systems. If there are features you're looking for, but don't have, then you can pick one that works better for your needs. If you want a bleeding edge system, you can also grab your favorite choice from the Internet.

*3) Virtual Filesystem:* The VFS lies at the top layer of the OS input-output system. The virtual filesystem allows user-space programs to use standard UNIX systems calls such `read()` and `write()` regardless of storage medium or underlying file system. This is setup is only possible because Linux adds a layer of abstraction around the low-level filesystem. Although we take such functionality for granted, it is a very important system to have. The VFS is divide into two overall pieces: structures, and their associated operations. [**?**]

*4) VFS Structures:* The VFS provides an abstraction of the filesystem, or multiple filesystems through a few specific interfaces. Specifically, the VFS provides the following abstractions:

Files: an ordered string of bytes.

Dentries: Files that contain directory information. For example, '\this\is\a\path' has four dentries on the path to that directory. This is how the VFS contains directories as files.

Inodes: Otherwise known as an 'index node,' these files contain metadata about other files such as permissions, size, owner, etc.

Superblock: A file containing all the relevant information about a filesystem as a whole. This is a metadata file for an entire FS.

Each of these interfaces have sizable data structures, along with a corresponding operations struct that complements them, containing function pointers to give the VFS the great functionality it has.[**?**]

These interfaces are critical to how we use Linux every day. Without them, we would have to make manual calls to the different filesystems that managed different devices, in the protocols they require. Sounds like a mess!

## *C. Windows*

Windows does things quite a bit different than Linux as far as file systems go. This comes as no surprise as Windows has a trend of doing its own thing. Because it is not originally a Unix based or Unix inspired operating system, we get to see a very different approach to solving similar problems. In the following section, we will examine the Windows file system, `NTFS`, and its virtual file system layer.

*1) NTFS:* Windows has a few different file systems available by default. The native file system, however, is called `NTFS`, or New Technology File System. `NTFS` is fairly comparable to `EXT4` in maximum partition and file size. `NTFS` theoretically supports up to exabyte volume sizes, but Windows currently limits support to 256 terrabyte volume sizes with 64 kilobyte cluster sizes. `NTFS` has the normal array of features that standard file systems ship with, along with file and directory security, alternate data streams, file compression, symbolic and hard links, encryption, and transactional semantics. [**?**]

*2) NTFS Driver:* Windows implements the virtual file system layer using the file system module itself. In this, Windows does not have a distinct VFS layer. It implements the file system and the VFS in one module, the `NTFS` module. This is a hard contract between Linux and Windows. Linux separates the VFS layer entirely, and then has the FS plug into the VFS and integrate. If you wanted to plug another file system in to Windows, it would use a distinctly different set of interfaces from another. So if you wanted to use a `FAT32` file system type, it would not easily integrate with a `NTFS` file system. [**?**]

Now that we've examined how Windows handles its filesystem layer, and saw how different its monolithic approach is Linux' abstracted layers approach, we will examine how FreeBSD handles these same concepts.

### D. FreeBSD

FreeBSD is very similar to Linux in how it handles many things. The file system is no exception to this rule. FreeBSD is very similar to Linux in this department, and we examine the subtle differences it has.

*1) The Fast File System:* The Fast File System is FreeBSD's default file system. It is fairly standard as File systems come, and is well tested and stable. The FS has features like directory structure, file names, journaling, etc. The standard set of features. It does not do file and directory security, like Windows. Permissions are managed, but not access to specific files and directories.

*2) Virtual File System:* The VFS setup in FreeBSD has a few structs that are similar to the Linux structs. Something interesting about the FreeBSD versus the Linux is that FreeBSD makes explicit statements about how a file's existence is defined. For example, a file in FreeBSD exists until no references or descriptors are open in the OS. Linux may have similar functions, but the rules for these are not explicitly defined. Here are the basic objects that make up the FreeBSD basic IO system: [**?**]

> Files: A Linear array of bytes with at least one name. A file exists until all its names are explicitly deleted explicitly and no process holds a descriptor of that file. All IO devices are treated as files.
>
> Directory Entry: A file that contains information about itself and other files and directory entries.
>
> Pipes: Pipes are also a linear array of bytes, but are used exclusively for one direction data transfer through the use of an IO stream. If you need to read and write from a file at the same time, two pipes will need to be open.
>
> Socket: An object that is used for interprocess communication, it exists only as long as some process holds the descriptor for it.

This setup is almost identical to how Linux handles things, but with one big difference: other filesytems mounted inside of the virtual file system are treated simply as directories, not as whole units like Linux's superblocks. This simplifies the handling of the entire filesystem structure. A root directory of a filesystem is set so that the OS knows where everything starts.

### E. Conclusion

Linux, Windows, and FreeBSD all have their own way of doing things. Although Linux and FreeBSD share a lot of the same ideas, they are unique. Windows is another beast entirely with its modules and driver structures. Understanding how all these systems work are important as a developer working in the each of their respective kernels.

## VI. Conclusion

All of the covered topics are vital to any modern operating system. Understanding them can be tedious and require some time to study. This is often true, however, of many things worth learning! Understanding these basic topics will not only empower a developer to work and develop within a given operating system, but the developer could use some of the ideas learned here to solve other problems. If there's anything I've learned this term, it's that a lot of work has gone into developing these complex systems and they're worth learning about. Although I've only grasped the tip of the iceberg, I will continue to learn more about the inner workings of operating systems in the future.

## VII. Appendix A - Linux Structures

All following structure code has been pulled from the classroom text. [?]

Listing 7. The Linux task_struct is responsible for keeping track of the state of each process.

```
struct  task_struct  {
/* these are hardcoded - don't touch */
volatile long          state ;           /* -1 unrunnable, 0 runnable , >0 stopped */
long                   counter;
long                   priority ;
unsigned               long  signal ;
unsigned               long blocked;     /* bitmap of masked signals */
unsigned               long  flags ;     /* per process flags , defined below */
int  errno ;
long                   debugreg [8];     /* Hardware debugging registers */
struct exec_domain     *exec_domain;
/* various  fields  */
struct linux_binfmt    *binfmt;
struct  task_struct    *next_task , *prev_task ;
struct  task_struct    *next_run,   *prev_run;
unsigned long          saved_kernel_stack ;
unsigned long          kernel_stack_page ;
int                    exit_code ,  exit_signal ;
/* ??? */
unsigned long          personality ;
int                    dumpable:1;
int                    did_exec :1;
int                    pid ;
int                    pgrp;
int                    tty_old_pgrp ;
int                    session ;
/* boolean value for session group leader */
int                    leader ;
int                    groups[NGROUPS];
/*
* pointers to ( original ) parent process , youngest child , younger sibling ,
* older  sibling ,  respectively .  (p->father can be replaced with
* p->p_pptr->pid)
*/
struct  task_struct    *p_opptr , *p_pptr , *p_cptr ,
*p_ysptr, *p_osptr;
struct wait_queue      * wait_chldexit ;
unsigned short         uid, euid, suid, fsuid ;
unsigned short         gid, egid, sgid, fsgid ;
unsigned long          timeout,  policy,   rt_priority  ;
unsigned long           it_real_value  ,  it_prof_value ,   it_virt_value  ;
unsigned long           it_real_incr  ,  it_prof_incr ,   it_virt_incr  ;
struct  timer_list      real_timer ;
long                   utime,  stime,  cutime,  cstime,   start_time ;
/* mm fault and swap info:  this  can  arguably  be  seen  as  either
mm-specific or thread-specific  */
unsigned long           min_flt ,   maj_flt ,  nswap,  cmin_flt ,   cmaj_flt ,  cnswap;
int  swappable:1;
unsigned long          swap_address;
unsigned long           old_maj_flt ;      /*  old  value  of  maj_flt  */
unsigned long           dec_flt ;          /*  page  fault  count  of  the  last  time  */
unsigned long          swap_cnt;          /*  number of pages to  swap on next  pass  */
/*  limits  */
struct  rlimit          rlim [RLIM_NLIMITS];
unsigned short         used_math;
char                   comm[16];
/*  file  system  info  */
int                     link_count ;
struct  tty_struct      * tty ;             /*  NULL  if no  tty  */
/*  ipc  stuff  */
struct  sem_undo       *semundo;
struct  sem_queue      *semsleeping;
/*  ldt  for  this  task  - used by Wine.  If  NULL, default_ldt  is  used  */
struct  desc_struct  * ldt ;
/*  tss  for  this  task  */
struct  thread_struct  tss ;
/*  filesystem  information  */
struct  fs_struct       * fs ;
/*  open  file  information  */
struct  files_struct    * files ;
/*  memory management info */
struct  mm_struct      *mm;
/*  signal  handlers  */
```

```
struct   signal_struct   *sig;
#ifdef __SMP__
int                          processor;
int                          last_processor;
int                          lock_depth;      /* Lock depth.
We can context switch in and out
of holding a syscall kernel lock ... */
#endif
};
```

## VIII. Appendix B - FreeBSD Structures

FreeBSD `proc` struct: [?]

Listing 8. The FreeBSD proc structure is responsible for tracking the state of processes.

```
/*
* Description of a process.
*
* This structure contains the information needed to manage a thread of
* control, known in UN*X as a process; it has references to substructures
* containing descriptions of things that the process uses, but may share
* with related processes. The process structure and the substructures
* are always addressable except for those marked "(PROC ONLY)" below,
* which might be addressable only on a processor on which the process
* is running.
*/
struct   proc {
struct   proc *p_forw;          /* Doubly-linked run/sleep queue. */
struct   proc *p_back;
struct   proc *p_next;          /* Linked list of active procs */
struct   proc **p_prev;         /*    and zombies. */

/* substructures: */
struct   pcred *p_cred;         /* Process owner's identity. */
struct   filedesc *p_fd;        /* Ptr to open files structure. */
struct   pstats *p_stats;       /* Accounting/ statistics (PROC ONLY). */       struct   plimit *p_limit;
       /* Process limits. */
struct   vmspace *p_vmspace;    /* Address space. */
struct   sigacts *p_sigacts;    /* Signal actions, state (PROC ONLY). */
```

```
#define p_ucred          p_cred->pc_ucred
#define p_rlimit         p_limit->pl_rlimit

int      p_flag;                  /* P_* flags. */
char     p_stat;                  /* S* process status. */
char     p_pad1[3];

pid_t    p_pid;                   /* Process identifier. */
struct   proc *p_hash;      /* Hashed based on p_pid for kill+exit+... */
struct   proc *p_pgrpnxt;   /* Pointer to next process in process group. */
struct   proc *p_pptr;      /* Pointer to process structure of parent. */
struct   proc *p_osptr;     /* Pointer to older sibling processes. */

/* The following fields are all zeroed upon creation in fork. */
#define p_startzero      p_ysptr
struct   proc *p_ysptr;     /* Pointer to younger siblings. */
struct   proc *p_cptr;      /* Pointer to youngest living child. */
pid_t    p_oppid;           /* Save parent pid during ptrace. XXX */
int      p_dupfd;           /* Sideways return value from fdopen. XXX */

/* scheduling */
u_int    p_estcpu;          /* Time averaged value of p_cpticks. */
int      p_cpticks;         /* Ticks of cpu time. */
fixpt_t  p_pctcpu;          /* %cpu for this process during p_swtime */
void     *p_wchan;          /* Sleep address. */
char     *p_wmesg;          /* Reason for sleep. */
u_int    p_swtime;          /* Time swapped in or out. */
u_int    p_slptime;         /* Time since last blocked. */

struct   itimerval p_realtimer;   /* Alarm timer. */
struct   timeval p_rtime;         /* Real time. */
u_quad_t p_uticks;                /* Statclock hits in user mode. */
u_quad_t p_sticks;                /* Statclock hits in system mode. */
u_quad_t p_iticks;                /* Statclock hits processing intr. */

int      p_traceflag;             /* Kernel trace points. */
struct   vnode *p_tracep;         /* Trace to vnode. */
```

```
int        p_siglist ;                  /* Signals  arrived  but  not  delivered .  */

struct    vnode *p_textvp;              /* Vnode of executable.  */

char      p_lock;                       /* Process  lock  (prevent  swap) count.  */
char      p_pad2[3];                    /* alignment  */

/* End area  that  is  zeroed  on  creation .  */
#define  p_endzero           p_startcopy

/* The following  fields  are  all  copied upon  creation  in  fork .  */
#define  p_startcopy        p_sigmask

sigset_t  p_sigmask;       /*  Current  signal  mask.  */
sigset_t  p_sigignore ;    /*  Signals  being  ignored .  */
sigset_t  p_sigcatch ;     /*  Signals  being  caught  by  user .  */

u_char    p_priority ;     /*  Process   priority  .  */
u_char    p_usrpri ;       /*  User−priority  based  on p_cpu and p_nice.  */
char      p_nice;          /*  Process  "nice"  value .  */
char      p_comm[MAXCOMLEN+1];

struct    pgrp *p_pgrp;    /*  Pointer  to  process  group.  */

struct    sysentvec  *p_sysent; /*  System  call  dispatch  information .  */

struct    rtprio   p_rtprio ;           /*  Realtime  priority  .  */
/* End area  that  is  copied  on  creation .  */
#define  p_endcopy          p_addr
struct    user  *p_addr;   /*  Kernel  virtual  addr  of  u−area (PROC ONLY).  */
struct    mdproc p_md;     /*  Any machine−dependent fields.  */

u_short  p_xstat ;         /*  Exit  status  for  wait;  also  stop  signal .  */
u_short  p_acflag ;        /*  Accounting  flags .  */
struct    rusage  *p_ru;   /*  Exit  information .  XXX  */
};
```

## IX. Bibliography

### References

## X. Drafts

# CS 444 - Spring 2016 - Writing Assignment 1

Cody Malick

malickc@oregonstate.edu

**Abstract**

Understanding how an operating system handles processes, threads, and CPU scheduling are important topics for an aspiring kernel developer to understand. In this report, we will cover how Linux, Windows, and FreeBSD handle these important tasks in their respective kernels, and contrast them.

## CONTENTS

## I. Introduction

This report contains descriptions on how each of the operating systems, Linux, Windows, and FreeBSD handle processes, threads, and CPU scheduling. A lot can be learned about the design decisions of the minds behind the operating system by how they handle these core constructs. We will start by describing how Linux handles these, as Linux is the focus of this class.

## II. Linux

### A. Processes

The first thing that should be defined, is simply what a process is: "A *process* is a program (object code stored on some media) in the midst of execution."[1] Although the Linux kernel refers to processes as tasks, in this report (for uniformity) we will only refer to them as processes.

A process in Linux is ultimately born, somewhere up the line, from the `init` process. The `init` is the absolute root of every process tree on any Linux machine. It has a process ID of 1. All processes are spawned from either the `fork()` or `exec()` function calls. `Fork()` creates an exact copy of the calling process as a child of the process, while `exec()` loads a new executable into the process space and executes it. These are the only ways new processes are created in Linux.

Each process has a process descriptor that describes every aspect of the process, the `task_struct`. The `task_struct` is contained in `<linux/sched.h>`. `Task_struct` describes the current state of the process, the parent process ID, the process ID, exit code, priority, etc. Everything you could possibly want to know about a process in Linux is in that struct. I've included the `Task_struct` as Appendix A as it is a very large structure.

Processes, of course, must have states of execution. On Linux, processes have five states, each represented by flags in the `task_struct`:

`TASK_RUNNING`: The process is either running or in a run-queue ready to be run (run-queues will be discussed in the CPU scheduling section).

`TASK_INTERRUPTIBLE`: the process is sleeping (blocked) waiting for a condition to un-block. Once the condition is met, it transitions to `TASK_RUNNING`.

`TASK_UNINTERRRUPTIBLE`: The same state as `TASK_INTERRUPTIBLE` with the difference being that it does not wake once an interrupt is received.

`_TASK_TRACED`: This flag shows that the process is being traced by another process. An example is GDB.

`_TASK_STOPPED`: Process execution has halted and it is not possible to start running. This state is a result of a halting interrupt from the CPU, or if a debugger sends it a signal.

### B. Threads

As with processes, we will define what a thread is: "Threads of execution, often shortened to *threads*, are the objects of activity within the process." [1] In Linux, threads are treated fundamentally the same as processes. The only difference is that they have a shared address space, filesystem resources, file descriptors, and signal handlers. Here is the `thread_info` struct:[1]

```
struct thread_info {
        struct task_struct      *task;
```

```
    struct exec_domain    *exec_domain;
    unsigned long          flags;
    unsigned long          status;
    __u32                  cpu;
    __s32                  preempt_count;
    mm_segment_t           addr_limit;
    struct restart_block   restart_block;
    unsigned long          previous_esp;
    __u8                   supervisor_stack[0];
};
```

Compared to the `task_struct`, this is a realitively simple data structure. It has a pointer to the `task_struct` that is unique to that thread, along with a few extra flags and sets of information that seperate threads from processes.

### C. CPU Scheduling

Linux ships with a few different schedulers by default, including a real time scheduler.[2] The default scheduler for Linux is the CFS, or the Completely Fair Scheduler. The CFS is only fair in so far as it tries its best to split the processors time into equal bits, 1/n, where n is the number of processes currently running. The CFS weights this with process priority, or nice value, and the higher the weight the larger percentage of processor time that process gets. So it's not completely fair, but that is the idea behind it.

One of the major problems with this system is that a very large chunk of processes creates incredibly small slices of CPU time. For example, if there were six-thousand processes and/or threads that all had the same weight, then the cpu slice for each process would be 1/6000. There is a point where that slice is not large enough to even complete the context switch required to start executing the process again. The CFS handles this by implementing a floor on slice size, known as *minimum granularity*.[1] With this system, every process will always have a minimum amount of run time, but the system can become very slow once this size is reached.

## III. Windows

Windows is a very different beast. The process and thread structure are quite a bit different from Linux. The CPU scheduler Windows uses has the same basic idea as the CFS, but varies greatly when it comes to how it handles priority and division of CPU time.

### A. Processes

Processes in Windows are similiar to Linux in only one way: the both have a process ID. Otherwise, processes in Windows handle very differently than Linux. Here is the basic process struct in Windows, `_PROCESS_INFORMATION`: [3]

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

The `HANDLE` data type is a 32-bit unique identifier stored in the Windows kernel to identify individual processes.[4] The `hThread` points to the primary thread a process is executing on. These first two of these identifying fields are used internally by the kernel to specify what functions are performing operations on the process and thread objects. The second set of identifiers, `dwProcessId` and `dwThreadId` are used as unique identifiers for the process and thread themselves. These are the equivalent of the PID used by the the Linux `tast_struct`.

Process creation is also differnt in Windows. Instead of the constantly traceable tree of child-parent relationships, a function called `CreateProcess()` is invoked, resulting in a new process completely independent of the parent process. [5]

Process state in Windows is determined by thread state. Here are the possible thread states in Windows:[6]

> `Initialized`: A state that indicates the thread has been initialized, but has not yet started.
>
> `Ready`: A state that indicates the thread is waiting to use a processor because no processor is free. The thread is prepared to run on the next available processor.
>
> `Running`: A state that indicates the thread is currently using a processor.
>
> `Standby`: A state that indicates the thread is about to use a processor. Only one thread can be in this state at a time.
>
> `Terminated`: A state that indicates the thread has finished executing and has exited.
>
> `Transition`: A state that indicates the thread is waiting for a resource, other than the processor, before it can execute. For example, it might be waiting for its execution stack to be paged in from disk.
>
> `Unknown`: The state of the thread is unknown
>
> `Wait`: A state that indicates the thread is not ready to use the processor because it is waiting for a peripheral operation to complete or a resource to become free. When the thread is ready, it will be rescheduled.

This is interesting for a few reasons. First, there is an 'unknown' thread state. In Linux, it is isn't possible for a thread or process to be in an unknown state, only the ones listed. Next, it's interesting to see how Windows has so many more states than Linux. One of the differences that sticks out most is that Windows has a state when a process is queued to enter the processor in the `Standby` state. By having all these different states available, you can really see the real time state of the process, as apposed to Linux where you have fewer states that cover less situations. Then again, this could just be a result of Linux having better processor and thread management, requiring less states.

### *B. Threads*

Threads, in Windows, are handled as subcomponents of processes. They are grouped into thread pools, where multiple threads execute under one process. MSDN explains the idea behind this structure is to reduce the number of application threads, and provide management of worker threads. Threads are are spawned through the `CreateThread()` function call.[7]

### *C. CPU Scheduling*

The Windows CPU scheduler, by default, schedules processes in much the same way that the CFS schedules processes. A process is given the default weight, and the scheduler tries to divide the CPU time evenly among the running processes.

Once we introduce weights or priority, the scheduler's behavior is very different.

While the system is assigning CPU time slices to each process, if a higher priority thread becomes available, the CPU stops the current running process, mid-time slice, and starts running the higher priority threads.

This is vastly different from the CFS in that regard. The CFS will allow a current running thread to complete it's time slice before moving it out of the processor. By introducing this kind of behavior, the power of priority states becomes glaringly clear.

Windows has thirty-two levels of priority, each divided into subdivisions:

`IDLE_PRIORITY_CLASS`

`BELOW_NORMAL_PRIORITY_CLASS`

`NORMAL_PRIORITY_CLASS`

`ABOVE_NORMAL_PRIORITY_CLASS`

`HIGH_PRIORITY_CLASS`

`REALTIME_PRIORITY_CLASS`

Each of these classes has subdivisions inside of them, seven in each, that has steadily increasing higher priority, starting from one, rising to thirty-two. As a side note, the zero priority class is reserved for a thread that goes through and zeroes out memory when a process terminates.[8] Besides this large difference in priority, the two CPU schedulers are very similar.

## IV. FreeBSD

FreeBSD is an open source OS based off Unix, specifically the Berkeley variant, BSD. It looks and feels similar to Linux in a few ways, but has some subtle differences in process and thread structure. The scheduler is quite a bit different from the CFS.

### A. Processes

Processes in FreeBSD are very similar to Linux's. Similiar to Linux's process struct, `task_struct`, the FreeBSD `proc` struct is large and contains a lot of information. Processes are created using the same forking and exec process that Linux uses. After a process is forked, a new process that is identical, except for the PID, is created. The process has a copy of the parent's resources and addresses space.

Process states are handled in a much more simplified manner than either Windows or Linux in FreeBSD. Instead of having eight or five different states for a process, it only has three:

`NEW`: Process is undergoing creation

`NORMAL`: Thread or threads will be runnable, sleeping, or stopped

`ZOMBIE`: The process is undergoing process termination

As per above, `NORMAL` embodies three subdivisions, runnable, sleeping, or stopped, but overall the classification is simplified. [9]

### B. Threads

A notable difference between Linux and FreeBSD is that it does not treat threads and processes the same way. Threads are handle in a more simliar fashion to Windows. Each thread is attached to a parent process. If multiple threads are requested for one process, new processes are spawned, but they have the same PID. When a user looks for a process in the OS, they will only see the single entry representing the parent process.

Thread states are dictated by the list in the processes section, but have a caviat. When a thread is not running on the processor, which there can only be one of, all threads are in one of three queues: the run queue, the sleep queue, or the turnstile queue. As the names dictate, the run queue contains threads that are in a runnable state. Threads that are blocked and are awaiting events are in the sleep queue or the turnstile queue.

The turnstile queue is exclusively for short blocks on threads. In FreeBSD, short blocks are exclusively the result of read/write blocks. Because these kinds of blocks are common, each thread creates a turnstile queue when it is initialized, and creates a list of all threads that are blocked because of a read/write block on a certain piece of memory. When that block is lifted, the turnstile queue manages the order in which the blocked threads are resolved. Allocating turnstile queues on each thread as apposed to each lock results in lower memory usage by the kernel.

Sleep queues are similar to turnstiles in that they contain lists of threads where blocking is occuring, but the sleep queue is exclusively for medium to long term locks that can send interrupts if the lock is held for too long. [9]

## *C. CPU Scheduling*

The default CPU scheduler for FreeBSD is the ULE scheduler. ULE is not an acronym. [9] The scheduler is contained in the namespace `syskernsched_ule.c`, and if you remove the underscore in the namespace, you will see the reasoning behind the name.

The ULE scheduler is split into two parts in FreeBS: a low-level scheduler, and a high-level scheduler. The low-level scheduler runs often, and the high-level scheduler runs a few times a second.

The low-level scheduler runs extremely often, whenever a thread is blocked. When a block occurs, the low-level scheduler pulls the highest priority process from a set of run queues. These queues are organized from low to high priority. The high-level scheduler decides what the priority of each thread is, and sorts it into the appropriate run queue. Each core has it's own set of these run queues to prevent two cores from trying to access the same queue at the same time.

The different cores run a round robin system on the threads in the queues. Each thread gets equal runtime in the form of time quantums. These time quantums are the same as time slices. If a process uses an entire time quantum, it is moved to the back of the the queue that it came from, and a context switch occurs to the next highest priority thread.[9]

## V. Appendix A

The `task_struct`:[10]

```
  struct  task_struct  {
/*  these  are  hardcoded – don't touch  */
  volatile  long          state ;          /*  −1 unrunnable, 0 runnable ,  >0 stopped */
  long                    counter ;
  long                    priority  ;
  unsigned                long  signal ;
  unsigned                long  blocked;   /*  bitmap  of  masked  signals  */
  unsigned                long  flags ;    /*  per  process  flags ,  defined  below */
  int  errno ;
  long                    debugreg [8];    /*  Hardware debugging  registers  */
  struct  exec_domain     *exec_domain;
/*  various   fields   */
  struct  linux_binfmt    *binfmt;
  struct  task_struct     *next_task ,  *prev_task ;
  struct  task_struct     *next_run ,   *prev_run;
  unsigned  long          saved_kernel_stack ;
  unsigned  long          kernel_stack_page ;
  int                     exit_code ,   exit_signal  ;
  /*  ???  */
  unsigned  long          personality  ;
  int                     dumpable:1;
  int                     did_exec :1;
  int                     pid ;
  int                     pgrp;
  int                     tty_old_pgrp ;
  int                     session ;
  /*  boolean  value  for  session  group  leader  */
  int                     leader ;
  int                     groups[NGROUPS];
  /*
   *  pointers   to  ( original )  parent  process ,  youngest  child ,  younger  sibling ,
   *  older   sibling ,   respectively .   (p−>father can be  replaced  with
   *  p−>p_pptr−>pid)
   */
  struct  task_struct     *p_opptr ,  *p_pptr ,  *p_cptr ,
                          *p_ysptr ,  *p_osptr ;
  struct  wait_queue      * wait_chldexit ;
  unsigned  short         uid , euid , suid , fsuid ;
  unsigned  short         gid , egid , sgid , fsgid ;
  unsigned  long          timeout ,  policy ,  rt_priority  ;
  unsigned  long           it_real_value  ,  it_prof_value ,  it_virt_value  ;
  unsigned  long           it_real_incr  ,  it_prof_incr ,  it_virt_incr  ;
  struct   timer_list      real_timer ;
  long                    utime,  stime,  cutime,  cstime,  start_time ;
/* mm fault and swap info:  this  can  arguably  be  seen  as  either
   mm-specific or thread-specific  */
  unsigned  long           min_flt ,  maj_flt ,  nswap,  cmin_flt ,  cmaj_flt ,  cnswap;
  int  swappable:1;
  unsigned  long          swap_address;
  unsigned  long           old_maj_flt ;     /*  old  value  of  maj_flt  */
  unsigned  long           dec_flt ;         /*  page  fault  count  of  the  last  time  */
  unsigned  long          swap_cnt;          /*  number of pages to  swap on next pass  */
/*  limits  */
  struct   rlimit         rlim [RLIM_NLIMITS];
  unsigned  short         used_math;
  char                    comm[16];
/*  file  system  info  */
  int                      link_count ;
  struct   tty_struct      * tty ;           /*  NULL  if  no  tty  */
/*  ipc  stuff  */
  struct  sem_undo        *semundo;
  struct  sem_queue       *semsleeping;
/*  ldt  for  this  task  - used by Wine.  If  NULL, default_ldt  is  used  */
  struct   desc_struct  * ldt ;
/*  tss  for  this  task  */
  struct   thread_struct   tss ;
/*  filesystem  information  */
  struct   fs_struct       * fs ;
/*  open  file  information  */
  struct    files_struct    * files  ;
/*  memory  management  info  */
  struct  mm_struct       *mm;
/*  signal  handlers  */
  struct   signal_struct   * sig ;
```

```
#ifdef __SMP__
    int             processor;
    int             last_processor;
    int             lock_depth;     /* Lock depth.
                                       We can context switch in and out
                                       of holding a syscall kernel lock ... */
#endif
};
```

## VI. Appendix B

FreeBSD `proc` struct: [11]

```
 /*
 * Description of a process.
 *
 * This structure contains the information needed to manage a thread of
 * control, known in UN*X as a process; it has references to substructures
 * containing descriptions of things that the process uses, but may share
 * with related processes. The process structure and the substructures
 * are always addressable except for those marked "(PROC ONLY)" below,
 * which might be addressable only on a processor on which the process
 * is running.
 */
struct proc {
        struct  proc *p_forw;           /* Doubly-linked run/sleep queue. */
        struct  proc *p_back;
        struct  proc *p_next;           /* Linked list of active procs */
        struct  proc **p_prev;          /*     and zombies. */

        /* substructures: */
        struct  pcred *p_cred;          /* Process owner's identity. */
        struct  filedesc *p_fd;         /* Ptr to open files structure. */
        struct  pstats *p_stats;        /* Accounting/ statistics (PROC ONLY). */        struct  plimit
            *p_limit;           /* Process limits. */
        struct  vmspace *p_vmspace;     /* Address space. */
        struct  sigacts *p_sigacts;     /* Signal actions, state (PROC ONLY). */

#define p_ucred         p_cred->pc_ucred
#define p_rlimit        p_limit->pl_rlimit

        int     p_flag;                 /* P_* flags. */
        char    p_stat;                 /* S* process status. */
        char    p_pad1[3];

        pid_t   p_pid;                  /* Process identifier. */
        struct  proc *p_hash;   /* Hashed based on p_pid for kill+exit+... */
        struct  proc *p_pgrpnxt; /* Pointer to next process in process group. */
        struct  proc *p_pptr;   /* Pointer to process structure of parent. */
        struct   proc *p_osptr;    /* Pointer to older sibling processes. */

/* The following fields are all zeroed upon creation in fork. */
#define p_startzero     p_ysptr
        struct   proc *p_ysptr;    /* Pointer to younger siblings. */
        struct   proc *p_cptr;     /* Pointer to youngest living child. */
        pid_t    p_oppid;          /* Save parent pid during ptrace. XXX */
        int      p_dupfd;          /* Sideways return value from fdopen. XXX */

        /* scheduling */
        u_int    p_estcpu;         /* Time averaged value of p_cpticks. */
        int      p_cpticks;        /* Ticks of cpu time. */
        fixpt_t  p_pctcpu;         /* %cpu for this process during p_swtime */
        void     *p_wchan;         /* Sleep address. */
        char     *p_wmesg;         /* Reason for sleep. */
        u_int    p_swtime;         /* Time swapped in or out. */
        u_int    p_slptime;        /* Time since last blocked. */

        struct   itimerval  p_realtimer;  /* Alarm timer. */
        struct   timeval p_rtime;         /* Real time. */
        u_quad_t p_uticks;                /* Statclock hits in user mode. */
        u_quad_t p_sticks;                /* Statclock hits in system mode. */
        u_quad_t p_iticks;                /* Statclock hits processing intr. */

        int      p_traceflag;             /* Kernel trace points. */
        struct   vnode *p_tracep;         /* Trace to vnode. */

        int      p_siglist;               /* Signals arrived but not delivered. */

        struct   vnode *p_textvp;         /* Vnode of executable. */

        char     p_lock;                  /* Process lock (prevent swap) count. */
        char     p_pad2[3];               /* alignment */

/* End area that is zeroed on creation. */
#define p_endzero       p_startcopy

/* The following fields are all copied upon creation in fork. */
```

```
#define  p_startcopy       p_sigmask

         sigset_t  p_sigmask;      /* Current  signal  mask. */
         sigset_t  p_sigignore;    /* Signals  being  ignored. */
         sigset_t  p_sigcatch;     /* Signals  being  caught  by  user. */

         u_char   p_priority;      /* Process  priority. */
         u_char   p_usrpri;        /* User-priority  based on p_cpu and p_nice. */
         char     p_nice;          /* Process  "nice"  value. */
         char     p_comm[MAXCOMLEN+1];

         struct   pgrp *p_pgrp;    /* Pointer  to  process  group. */

         struct   sysentvec *p_sysent; /* System call  dispatch  information. */

         struct   rtprio  p_rtprio;     /* Realtime  priority. */
/* End area that is copied on creation. */
#define  p_endcopy         p_addr
         struct   user *p_addr;    /* Kernel  virtual  addr  of  u-area (PROC ONLY). */
         struct   mdproc p_md;     /* Any machine-dependent fields. */

         u_short  p_xstat;         /* Exit  status  for  wait;  also  stop  signal. */
         u_short  p_acflag;        /* Accounting  flags. */
         struct   rusage *p_ru;    /* Exit  information.  XXX */
};
```

## VII. Bibliography

### References

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

[2] R. Hat. (2016, jan) 4.2 cpu scheduling. [Online]. Available: https://access.redhat.com/documentation/en-US/RedHatEnterpriseLinux/6/html/PerformanceTuningGuide/s-cpu-scheduler.html

[3] MSDN. (2016, jan) Process_information structure. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684873

[4] ——. (2016, jan) Handles and objects. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms724457

[5] ——. (2016, jan) Creating processes. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682512

[6] ——. (2016, jan) Threadstate enumeration. [Online]. Available: https://msdn.microsoft.com/en-us/library/system.diagnostics.threadstate(v=vs.110).aspx

[7] ——. (2016, jan) Creating threads. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682516

[8] ——. (2016, jan) Scheduling priorities. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100

[9] K. McCusick and G. V. Neville-Neil, *Design and Implementation of the FreeBSD Operating System*, 2nd ed. Pearson Education, 2015.

[10] L. Torvalds. (2016, mar) sched.h. [Online]. Available: https://github.com/torvalds/linux/blob/master/include/linux/sched.h

[11] FreeBSD. (2016, jan) struct proc structure. [Online]. Available: http://people.freebsd.org/ meganm/data/tutorials/ddwg/ddwg63.html

# CS 444 - Spring 2016 - Writing Assignment 2

Cody Malick

malickc@oregonstate.edu

**Abstract**

Understanding how an operating system handles input and output to devices and how they are scheduled are important topics for an aspiring kernel developer. In this report, we will cover how Linux, Windows, and FreeBSD handle these important tasks in their respective kernels, and contrast them.

## Contents

## I. Introduction

Every computer, whether it's a mobile phone or a digital watch, needs to be able to read and write to storage. For general desktop computers, they have to be able to communicate with a vast variety of devices from hard drives to MicroSD cards. In this paper, we will discuss how Linux, Windows, and FreeBSD handle input and output, how they make it efficient, and contrast them to each other.

## II. Linux

### A. Virtual Filesystem

At the top layer of our input-output system, we have the virtual filesystem, otherwise known as the VFS. The virtual filesystem allows user-space programs to use standard unix systems calls such `read()` and `write()` regardless of storage medium. This is setup is only possible because Linux adds a layer of abstraction around the low-level filesystem. Although we take such functionality for granted, it is a very important system to have. [1]

The VFS provides an abstraction of the filesystem, or multiple filesystems through a few specific interfaces. Specifically, the VFS provides the following abstractions:

Files: an ordered string of bytes.

Dentries: Files that contain directory information. For example, '\this\is\a\path' has four dentries on the path to that directory. This is how the VFS contains directories as files.

Inodes: Otherwise known as an 'index node,' these files contain metadata about other files such as permissions, size, owner, etc.

Superblock: A file containing all the relevant information about a filesystem as a whole. This is a metadata file for an entire FS.

Each of these interfaces have sizable data structures, along with a corresponding operations struct that complements them, containing function pointers to give the VFS the great functionality it has.[1]

These interfaces are critical to how we use Linux every day. Without them, we would have to make manual calls to the different filesystems that managed different devices, in the protocols they require. Sounds like a mess!

### B. Block IO Layer

The Block IO layer is where the dirty work actually gets done. This layer manages how data is actual written to and retreived from the storage device. Any device that contains storage is considerd a block device to the kernel. Usually, these block devices contain segments called *sectors*, which are the smallest size that data can be stored in. The standard size is usually 512 bytes, but it's possible for these sizes to vary.

After a sector, we have the *block*. The block is the smallest addressable unit by the kernel. This is an abstraction imposed by the filesystem. The block can be the same size as a single sector, but not smaller, and a block must also contain a multiple of the sector size. If the sector size is 512 bytes, then the block cannot be 800 bytes, it must be 1024, or 2048 bytes. It cannot contain anything but whole sectors.

The actual reading and writing is handled by buffers and buffer heads. These buffers contain blocks that they are mapped to while they are pending to read or write from any given block. Buffer heads are the kernels way of having a descriptor pointing at these buffers. A buffer head contains all the information needed for the kernel to control and manipulate these buffers. [1]

```
struct buffer_head {
        unsigned long b_state; /* buffer state flags */
        struct buffer_head *b_this_page; /* list of pages buffers */
        struct page *b_page; /* associated page */
        sector_t b_blocknr; /* starting block number */
        size_t b_size; /* size of mapping */
        char *b_data; /* pointer to data within the page */
        struct block_device *b_bdev; /* associated block device */
        bh_end_io_t *b_end_io; /* I/O completion */
        void *b_private; /* reserved for b_end_io */
        struct list_head b_assoc_buffers; /* associated mappings */
        struct address_space *b_assoc_map; /* associated address space */
        atomic_t b_count; /* use count */
};
```

As you can see from the above code, the `buffer_head` struct contains all the information the kernel needs. It has `b_state` to track the state of the buffer, `buffer_head` to track the page's buffers, a pointer the the page associated with this buffer, and more. With this struct, the kernel can have direct control over what information is handled in the Block IO layer.

## *C. Scheduling*

Along with actually getting IO done, the kernel must decide what order these operations need to be done in. The kernel also needs to reduce the total number of requests by efficiently merging requests that can be merged. This is handled by the IO scheduler. There are many different approaches to different schedulers. It is a very fascinating area for that reason.

For example the noop (no operations) scheduler is the most basic one out there. All it does is maintain a request queue that is a first-in first-out data structure. It's only other responsibility is merging requests that are adjacent to other requests. This scheduler is the default scheduler for solid state drives and any sort of flash memory. The reason for this being that solid state devices have no moving parts, no head constantly seeking for different sectors. All read and write locations are self-sorting, and instant to read and write from. This scheduler performs absolutely poorly, however, when confronted with a spinning drive.

A spinning drive requires a scheduler to group requests by location. This is needed because if we simply serviced requests in a FIFO manner, then the head would spend enormous amounts of time seeking for sectors, and slow down [2]the OS tremendously. For this reason, the default scheduler for disk devices on Linux is the CFQ scheduler.

The Completely Fair Queuing IO Scheduler has the following functionality: it maintains a seperate qeuue for each process, it services these requests in a round robin fashion, and performs merging of adjacent requests. This works very well for a number of reasons. The first of which, is each process gets the IO it needs done in a fairly prompt manner, but also prevents any one process from being starved out of IO by another read or write heavy operation. Another is that if the system is a multiple user sytem, which almost every computer is these days, then each user will get their IO serviced in a prompt manner. [1]

### *D. Interrupts*

Interrupts are how a piece of hardware communicates with the cpu, letting it know that it has some data ready for it. An interrupt must have an interrupt handler associated with it in order to have the communication handled properly. These interrupt handlers are contained in a devices driver. This is pretty universal across operating systems. What we are going to look at specifically is the structure of interrupts that Linux handles. [1]

### *E. Interrupt Structure*

Interrupts are split into two parts in Linux: top halves, and bottom halves. These names are quite misleading as top halves and bottom halves are not halves at all. The top half is usually closer to a fifth or even an eighth. The reason for this is that interrupt handlers have to be very fast! Interrupts, as they are aptly named, stop everything the cpu is doing in order to communicate with hardware. When this interrupt is called, all the code in the top half of the interrupt has to be handled extremely quickly. Because of this, only time sensitive information can be stored in the top half, and all information in the top half is handled by the interrupt handler.

The bottom half, on the other hand, can contain any extra information needed to process the request from the hardware. Because this information is not deemed time critical, it is queued up with the other requests that the CPU needs to handle and is put off until its time to be processed has come. [1]

This is a basic overview of how Linux handles IO. Next, we'll examine how Windows and FreeBSD handle these same tasks, and compare them to how Linux operates.

## III. Windows

Windows does things quite a bit different from Linux, as we would expect. Because it is not originally a Unix based or Unix inspired operating system, we get to see a very different approach to solving similiar problems.

### *A. IO Manager*

Windows has a module called the IO Manager that is the core of all input and output requests. This system is packet based, meaning ever request comes through in the form of an IRP (IO request packet). The IO manager handles IO by taking a given IRP in memory, and handing it off to the appropriate driver, then disposing of the packet. Because of this, a lot of the IO done in windows is actually handled directly by the device drivers. In addition to handling IRPs, the IO manager provides an interface that allows simplification of driver tasks. For example, common tasks such as one driver calling functions in another driver. It also maintains IO buffers for the drivers. The IO manager gives a simple interfaces for these, reducing the complexity of implementing drivers in Windows.

In general execution of IO in Windows is done in the following pattern. First a request is made by an application, read for example, and is handed to the IO manager. The IO manager creates an IRQ which it then hands off to a driver for processing. [2]

### *B. Types of IO*

Windows has three classifications of IO: buffered IO, direct IO, and Neither IO. Buffered IO is IO dumped into a buffer for a specific device for write operations, and pulled from that devices buffer for read requests. Direct IO is when the IO manager takes a devices buffer and locks it into memory. It then handles the IO request, and releases control of that memory to the driver. The driver then pulls the data from the memory buffer. Doing this, the driver can have direct access to memory when it needs it. Lastly is Neither IO. As the name implies, this type of IO is when the method of IO is left entirely to the driver to handle. The IO manager simply hands off a request to the driver to handle.[2]

This is a very interesting classification of IO that neither FreeBSD nor Linux have. FreeBSD and Windows simply treat all devices as files, and the drivers simply handle the input from the user as if it was writing to a file, and the device communicates via interrupts to the CPU.

### *C. Drivers*

Drivers are important in every OS, but in Windows they have a very specific structure. Drivers have a few responsibilities besides handling interrupts as they do in Linux. They have the responsibility to start IO requests to the hardware, handle plug-and-play behavior, cancel IO routines, unload routines, and more. Drivers are much more of entire modules instead of specifically interrupt handlers as they are in Linux. [2]

This view of drivers makes the handling of IO much more of the drivers responsibility than it does in Linux. In Linux, the driver simply handles the interrupt and hands other information off to the OS. In Windows, the driver is more robust and has more responsibility on how things are communicated.

### *D. Scheduling*

The Windows IO scheduler handles IO requeusts by priority. It is a fairly straightfoward system. There are five levels of priority, critical, high, normal, low, and very low. Very low is usually restricted to background processes and content indexing. Doing things this way allows for a very interactive environment and high priority to any user initiated actions. Per the `Windows Internals Part 2` book I'm reading, low and high priority are not used. Normal is reserved for regular application IO, while cricial is reserved for the memory manager exclusively. These different priorities are sorted into individual queues, and are executed in decending order from high priority to low priority.

This scheduler shares the idea of multiple queues with the Linux scheduler, but otherwise it's a completely different beast. The scheduler isn't fair at all, and priority is king.[2]

Now that's we've seen the basic structure of Windows IO, we'll take a look at how FreeBSD handles these same ideas.

## IV. FreeBSD

FreeBSD is very similar to Linux in how it handles many of these processes. As both borrowed ideas from a purely unix operating system, we will see that things such as the VFS handle things similiarly.

### *A. Basic IO*

The IO in FreeBSD has a few structs that are similiar to the Linux IO structs. Something interesting about the FreeBSD versus the Linux is that FreeBSD makes explicit statements about how a file's existence is defined. For example, a file in

FreeBSD exists until no references or descriptors are open in the OS. Linux may have similiar functions, but the rules for these are not explicitly defined. Here are the basic objects that make up the FreeBSD basic IO system: [3]

> Files: A Linear array of bytes with at least one name. A file exists until all its names are explicitly deleted explicitly and no process holds a descriptor of that file. All IO devices are treated as files.
>
> Directory Entry: A file that contains information about itself and other files and directory entries.
>
> Pipes: Pipes are also a linear array of bytes, but are used exclusively for one direction data transfer through the use of an IO stream. If you need to read and write from a file at the same time, two pipes will need to be open.
>
> Socket: An object that is used for interprocess communication, it exists only as long as some process holds the descriptor for it.

This setup is almost identical to how Linux handles things, but with one big difference: other files sytems mounted inside of the virtual file system are treated simply as directories, not as whole units like Linux's superblocks. This simplifies the handling of the entire filesystem structure. A root directory of a filesystem is set so that the OS knows where everything starts.

## B. *Scheduling*

The FreeBSD scheduler uses a simplistic algorithm called `disksort()`. Disksort is almost incredibly straight forward: [4]

```
void disksort(drive qeuue *dq, buffer *bp)
{
  if(active list is empty) {
    place the buffer at the front of the active list;
    return;
  }
  if(request lies before the first active request) {
    locate the beginning of the next-pass list;
    sort bp into the next-pass list;
  } else {
    sort bp into the active list;
  }
}
```

The above pseudo code explains the basic idea. It is essentially a basic elevator that does merging and sorting as needed. It maintains two lists, current pass and next pass, and if a request requires a change in direction, add it to the next pass list. It is much more straightforward than the linux CFQ scheduler, though arguably less efficient. [3]

## C. *Interrupts*

Interrupts are handled almost identically to Linux's interrupt setup. Each interrupt must be registered via a device driver. FreeBSD has two types of interrupts: hardware interrupts and software interrupts. These are the equivalent of Linux's top and bottom halves. If a piece of information is time critical, it is put in a hardware interrupt, otherwise they are handed to a software interrupt. [3]

## V. Conclusion

Linux, Windows, and FreeBSD all have their own way of doing things. Although Linux and FreeBSD share a lot of the same ideas, they are unique. Windows is another beast entirely with its modules and driver structures. Understanding how all these systems work are important as a developer working in the each of their respective kernels. I am looking forward to diving into this topic more.

## 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
```

sectionBibliography

## 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

[4] K. McCusick and G. V. N. Neil, *Design and Implementation of the FreeBSD Operating System*, 2nd ed. Pearson Education, 2015.

# CS 444 - Spring 2016 - Writing Assignment 3

Cody Malick

malickc@oregonstate.edu

**Abstract**

Understanding how an operating system handles interrupts 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

Interrupts are how hardware communicates with the CPU, letting it know it has information of some kind to share with it. Once the interrupt is received, the CPU interrupts the kernel, handing off the information it has. Interrupts are taken care of by interrupt handlers. This report will examine what the structure of interrupts are in Linux, Windows, and FreeBSD, the general procedure of how they are handled, and where these interrupt handlers live in their respective OS.

## II. Linux

Linux has a fairly straightfoward way of handling interrupts. It has the interrupt structure of top and bottom halves, and each type of interrupt is registered to a unique handler in the kernel. The handler is defined by the driver, and is called when its unique handler ID appears.

### A. Interrupts

Interrupts are how a piece of hardware communicates with the CPU, letting it know that it has some data ready for it. An interrupt must have an interrupt handler associated with it in order to have the communication handled properly. These interrupt handlers are contained in a devices driver. This is pretty universal across operating systems. What we are going to look at specifically is the structure of interrupts that Linux handles. [1]

### B. Interrupt Structure

Interrupts are split into two parts in Linux: top halves, and bottom halves. These names are quite misleading as top halves and bottom halves are not halves at all. The top half is usually closer to a fifth or even an eighth. The reason for this is that interrupt handlers have to be very fast! Interrupts, as they are aptly named, stop everything the cpu is doing in order to communicate with hardware. When this interrupt is called, all the code in the top half of the interrupt has to be handled extremely quickly. Because of this, only time sensitive information can be stored in the top half, and all information in the top half is handled by the interrupt handler.

The bottom half, on the other hand, can contain any extra information needed to process the request from the hardware. Because this information is not deemed time critical, it is queued up with the other requests that the CPU needs to handle and is put off until its time to be processed has come. [1]

### C. Drivers

Drivers are where interrupt handlers exist in Linux. A handler ID is reserved by the system when a driver is installed. When an interrupt is called, the kernel goes and finds the appropriate handler, and calls the function to resolve the interrupt.

Here is an example of how an interrupt handler is requested by a driver:

```
if (request_irq(irqn, my_interrupt, IRQF_SHARED, "my_device", my_dev)) {
    printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
    return -EIO;
}
```

In the above example, the requested interrupt line is `irqn`, the interrupt handler `my_interrupt`, and the device `my_device`. If the request is handled properly, it returns zero, otherwise it returns the code prints an error and returns an error code, `-EIO`, an IO error. [1]

Next we'll look at how Windows handles these same ideas, and explore what interesting differences there are.

## III. Windows

Windows has a similiar setup to Linux only in that drivers are where interrupt handlers are defined. There are some interesting differences in what device drivers are responsible for, and how fundamental system IO is handled in Windows as compared to Linux.

### *A. Interrupts*

Windows has two different types of interrupts: general IO interrupts, and exceptions. Interrupts are asynchronus while exceptions are synchronus. Windows refers to catching these interrupts as traps, as they cause the kernel to differ from normal paths of execution. A trap is the equivilent of an interrupt handler. [2]

### *B. Drivers*

Drivers are important in every OS, but in Windows they have a very specific structure. Drivers have a few responsibilities besides handling interrupts as they do in Linux. They have the responsibility to start IO requests to the hardware, handle plug-and-play behavior, cancel IO routines, unload routines, and more. Drivers are much more of entire modules instead of specifically interrupt handlers as they are in Linux. [3]

This view of drivers makes the handling of IO much more of the drivers responsibility than it does in Linux. In Linux, the driver simply handles the interrupt and hands other information off to the OS. In Windows, the driver is more robust and has more responsibility on how things are communicated.

Now that we've explored the basics of interrupts in Windows, we can look at FreeBSD, and see what similiarities and differences there are in its core structure.

## IV. FreeBSD

FreeBSD shares a very similiar interrupt structure and handling pattern with Linux. There are a few minute differences that should be pointed out.

### *A. Interrupts*

Interrupts are handled almost identically to Linux's interrupt setup. Each interrupt must be registered via a device driver. FreeBSD has two types of interrupts: hardware interrupts and software interrupts. These are the equivalent of Linux's top and bottom halves. If a piece of information is time critical, it is put in a hardware interrupt, otherwise they are handed to a software interrupt. [4]

### *B. Interrupt Handler*

Interestingly enough, interrupt handlers are also called traps in FreeBSD, the same as it is in Windows. Although they're name is the same, they handle fundamentally different structures. The FreeBSD interrupt handling model is almost identical to the Linux model. Interrupts are handled asynchronously, and have to be registered with a unique ID so that the kernel can find which driver needs to handle the interrupt. [4]

## V. Conclusion

Interrupts are a key part of any OS. It is a fundamental in handling IO from external or internal devices. As apposed to other topics we've covered, it's interesting to see that the three operating systems have a lot in common in this area, as opposed to their individual unique approaches to other parts of the kernel.

## 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 1*, 6th ed. Microsoft Press, 2012.

[3] ——, *Windows Internals Part 2*, 6th ed. Microsoft Press, 2012.

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

# 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

# CS 444 - Spring 2016 - Writing Assignment 5

Cody Malick

malickc@oregonstate.edu

**Abstract**

Understanding how an operating system handles its file systems and virtual file systems, and how that helps make the user's life easier by abstracting the physical system is an important topic for an aspiring kernel developer. In this report, we will cover how Linux, Windows, and FreeBSD handle the VFS in their respective kernels.

## CONTENTS

## I. Introduction

As a general user, you don't want to have to deal directly with a computers input and output systems. It would be tiresome and tedious. To solve this problem, the file system and virtual file systems are wrappers around these arduous tasks that make it simple, and user friendly. The VFS is the unification of two ideas: making the user's life easy, while providing a generic interface that allows different types of file systems the same interfaces. Following is an examination of the FS and VFS implementations in Linux, Windows, and FreeBSD.

## II. Linux

### A. File Systems

A file system, simply, is an abstraction of the IO layer that allows ease of use for the user in tracking, reading, and writing to disk. Instead of forcing the user to remember where a file is, what its size is, what type of file it is, etc., the file system provides interfaces that make this a, relatively, easy task. There are many different types of file systems. They are differentiated by minimum and maximum file size, journaling capability, and maximum partition size.

*1) Linux Default File Systems:* There are a few default file systems available in Linux. A few of them are all from the `EXT` family, like `EXT2`, `EXT3`, and `EXT4`. `EXT3` is considered the standard FS for stable Linux builds, but `EXT4` is the latest and greatest release from that family. The biggest difference between `EXT4` and its previous version is the maximum file and partition size.[1] One the great things about Linux is the number of options for file systems. If there are features you're looking for, but don't have, then you can pick one that works better for your needs. If you want a bleeding edge system, you can also grab your favorite choice from the internet.

### B. Virtual Filesystem

The VFS lies at the top layer of the OS input-output system. The virtual filesystem allows user-space programs to use standard unix systems calls such `read()` and `write()` regardless of storage medium or underlying file system. This is setup is only possible because Linux adds a layer of abstraction around the low-level filesystem. Although we take such functionality for granted, it is a very important system to have. The VFS is divide into two overall pieces: structures, and their associated operations. [2]

### C. VFS Structures

The VFS provides an abstraction of the filesystem, or multiple filesystems through a few specific interfaces. Specifically, the VFS provides the following abstractions:

Files: an ordered string of bytes.

Dentries: Files that contain directory information. For example, '\this\is\a\path' has four dentries on the path to that directory. This is how the VFS contains directories as files.

Inodes: Otherwise known as an 'index node,' these files contain metadata about other files such as permissions, size, owner, etc.

Superblock: A file containing all the relevant information about a filesystem as a whole. This is a metadata file for an entire FS.

Each of these interfaces have sizable data structures, along with a corresponding operations struct that complements them, containing function pointers to give the VFS the great functionality it has.[2]

These interfaces are critical to how we use Linux every day. Without them, we would have to make manual calls to the different filesystems that managed different devices, in the protocols they require. Sounds like a mess!

## III. Windows

Windows does things quite a bit different than Linux as far as file systems go. This comes as no surprise as Windows has a trend of doing its own thing. Because it is not originally a Unix based or Unix inspired operating system, we get to see a very different approach to solving similiar problems. In the following section, we will examine the Windows file system, `NTFS`, and its virtual file system layer.

### *A. NTFS*

Windows has a few different file systems available by default. The native file system, however, is called `NTFS`, or New Technology File System. `NTFS` is fairly comperable to `EXT4` in maximum partition and file size. `NTFS` theoretically supports up to exabyte volume sizes, but Windows currently limits support to 256 terrabyte volume sizes with 64 kilobyte cluster sizes. `NTFS` has the normal array of features that standard file systems ship with, along with file and directory security, alternate data streams, file compression, symbolic and hard links, encryption, and transactional semantics. [3]

### *B. NTFS Driver*

Windows implements the virtual file system layer using the file system module itself. In this, Windows doesn't have a distinct VFS layer. It implements the file system and the VFS in one module, the `NTFS` module. This is a hard contract between Linux and Windows. Linux seperates the VFS layer entirely, and then has the FS plug into the VFS and integrate. If you wanted to plug another file system in to Windows, it would use a distinctly different set of interfaces from another. So if you wanted to use a `FAT32` file system type, it would not easily integrate with a `NTFS` file system. [3]

Now that we've examined how Windows handles its filesystem layer, and saw how different its monolithic approach is Linux' abstracted layers approach, we will examine how FreeBSD handles these same concepts.

## IV. FreeBSD

FreeBSD is very similar to Linux in how it handles many things. The file system is no exception to this rule. FreeBSD is very similiar to Linux in this department, and we examine the subtle differences it has.

### *A. The Fast File System*

The Fast File System is FreeBSD's default file system. It is fairly standard as File systems come, and is well tested and stable. The FS has features like directory structure, file names, journaling, etc. The standard set of features. It does not do file and directory security, like Windows. Permissions are managed, but not access to specific files and directories.

### *B. Virtual File System*

The VFS setup in FreeBSD has a few structs that are similiar to the Linux structs. Something interesting about the FreeBSD versus the Linux is that FreeBSD makes explicit statements about how a file's existence is defined. For example, a file in FreeBSD exists until no references or descriptors are open in the OS. Linux may have similiar functions, but the rules for these are not explicitly defined. Here are the basic objects that make up the FreeBSD basic IO system: [4]

> Files: A Linear array of bytes with at least one name. A file exists until all its names are explicitly deleted explicitly and no process holds a descriptor of that file. All IO devices are treated as files.
>
> Directory Entry: A file that contains information about itself and other files and directory entries.
>
> Pipes: Pipes are also a linear array of bytes, but are used exclusively for one direction data transfer through the use of an IO stream. If you need to read and write from a file at the same time, two pipes will need to be open.
>
> Socket: An object that is used for interprocess communication, it exists only as long as some process holds the descriptor for it.

This setup is almost identical to how Linux handles things, but with one big difference: other files sytems mounted inside of the virtual file system are treated simply as directories, not as whole units like Linux's superblocks. This simplifies the handling of the entire filesystem structure. A root directory of a filesystem is set so that the OS knows where everything starts.

## V. Conclusion

Linux, Windows, and FreeBSD all have their own way of doing things. Although Linux and FreeBSD share a lot of the same ideas, they are unique. Windows is another beast entirely with its modules and driver structures. Understanding how all these systems work are important as a developer working in the each of their respective kernels.

## VI. Appendix A - Linux Structs

All following structs have been pulled from the classroom text. [2]

```
for future code samples
```

## VII. Appendix B - Windows Structs

```
for future code samples
```

## VIII. Appendix C - FreeBSD Structs

```
for future code samples
```

sectionBibliography

## References

[1] A. J. Simon. (2015, nov) Linux file systems explained. [Online]. Available: https://help.ubuntu.com/community/LinuxFilesystemsExplained

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

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

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