CS 444 GROUP 15

Final paper

Writer:

Changxu YAN

931-927-069

December 7, 2016

1 File Systems

File systems are an integral part of any operating system. They allow users to upload and store files, provide access to data, and make hard drives useful. Different operating systems differ in their native file system. Modern Linux uses ext4, Windows uses NTFS, last but not least, FreeBSD uses ZFS. File systems are the fundamental principle behind how data is transferred in I/O situations. Every file store in a computer is stored and retrieved via a file system. Without a file system all of the data on the disk would be a pile of bytes unknowing where one file ends and another begins. For each file system there needs to be a way for the data structures to be written and read from the physical volume. This is where the file system drivers come in. Microsoft Windows has a very different way of approaching file systems than the Linux implementation. FreeBSD uses *ZFS* as their native file system which have been consider ”next-gen file system” by folks. Linux, Windows and FreeBSD have the similar architecture but the way they are implemented with in the operating systems are very different.

Linux

Linux supports dozens of different file systems with the most common being Minux, Ext[2,3,4] and JFS. Linux has one of the most versatile file system support all because of the *VFS* (Virtual File System). The *VFS* creates the abstraction from real underlying file I/O operations. With all of the I/O operations flowing through the *VFS* for Linux to support different file system all that is needed is a driver to translates standardized *VFS* data operations to the current file system operations. This allows the Linux system to perform I/O operations without even needing to know what the actual file system of the disk is. [1].

The *VFS* describes the system’s files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the *VFS* inodes describe files and directories within the system; the contents and topology of the Virtual File System. A *VFS* specifies an interface between the kernel and a concrete file system. Therefore, it is easy to add support for new file system types to the kernel simply by fulfilling the contract.

Windows

Windows on the other hand only supports a handful of file systems. Since Microsoft does not take advantage of a standardize virtual file system like Linux the operating system must know each file system and how to specifically talk to the media. Windows achieves this with an advanced *I/O Manager* that in turn uses drivers just like the Linux kernel to perform the actually operations on to the file system [2].

Windows either at boot or when a new volume is mounted first tries to identify what type of filesytem the volume contains. Every Widows supported file system contains the information need for the *FSD* (File system Driver) in the volumes boot sector. If the file system format is unrecognized or boot sector has been corrupt rendering the information Windows uses to identify the volumes file system unreadable Windows will default to the *RAW FSD*. When Windows declares a file system as *RAW* the user is prompted to see if they would like to format the volume. If the boot sector is intact and Windows identifies the file system as a supported type Windows creates a *Device Object* that the operating system will use to map the requested I/O operations to the physical media. After a *FSD* claims a volume all of the I/O operations to that volume is passed trough the *FSD*. All of the I/O operations are mapped from the *Device Object* by the *VPB* (Volume Parameter Block) to the volumes responsible *FSD*. When Windows mounts a volume and assigns a drive letter to it, it is really just a symbolic link to the *Device Object* [2].

FreeBSD

Traditionally, the native FreeBSD file system has been the Unix File System UFS which has been modernized as UFS2. It supported Snapshots and journaled soft updates. Since FreeBSD 7.0, the *Z File System (ZFS)* is also available as a native file system. FreeBSD also supports a multitude of other file systems so that data from other operating systems can be accessed locally, such as data stored on locally attached USB storage devices, flash drives, and hard disks. This includes support for the *Linux Extended File System (EXT)* and the Reiser file system.

ZFS, initially develped by Sun Microsystem, is significantly different from any previous file system because it is more than just a file system. Combining the traditionally separate roles of volume manager and file system provides *ZFS* with unique advantages. Such as volume management, snapshots, checksumming, RAID, compression and deduplication, replication and more. The file system is now aware of the underlying structure of the disks. Traditional file systems could only be created on a single disk at a time. If there were two disks then two separate file systems would have to be created. In a traditional hardware RAID configuration, this problem was avoided by presenting the operating system with a single logical disk made up of the space provided by a number of physical disks, on top of which the operating system placed a file system. Even in the case of software RAID solutions like those provided by GEOM, the UFS file system living on top of the RAID transform believed that it was dealing with a single device. *ZFS*'s combination of the volume manager and the file system solves this and allows the creation of many file systems all sharing a pool of available storage. One of the biggest advantages to *ZFS*'s awareness of the physical layout of the disks is that existing file systems can be grown automatically when additional disks are added to the pool. This new space is then made available to all of the file systems. *ZFS* also has a number of different properties that can be applied to each file system, giving many advantages to creating a number of different file systems and data sets rather than a single monolithic file system. [3]

COMPARISON

Similarities

- Read information from the MBR(master Boot Record).
- Each file system gets a file descriptor mapped to unique location.
- Need a file system driver to write to actual volume.

Differences

- Windows kernel needs to be aware of what the underlying file system is.
- Volumes in Windows get assigned letters, Volumes are assigned devices with numbers in Linux.
- FreeBSD ZFS utilizing virtual devices to manage one ore more disks like a RAID hardware controller do.
- FreeBSD ZFS using 256-bit checksum to ensure data integrity and reliability.

2 I/O AND FUNCTIONALITY

To compare I/O and its functionality, we need to understand what they are. Linux and FreeBSD belongs to POSIX API. For POSIX, there are a bunch of standards for ”UnixLike” APIs, all called POSIX followed by some obscure serial number. Windows API is Microsoft’s core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems. POSIX support more ”UnixLike” Operating systems, such as linux, FreeBSD and MacOS, it has better portablity compare to WIN32 API. These two interfaces has lots of differences and similarities. In here, I will talk about File I/O, signal handling and processes.[1]

File I/O

Both POSIX and Windows API are using the universal I/O model. All system calls for performing I/O refer to open files using a file descriptor, a (usually small) nonnegative integer. You can find file descriptor for all types of files: pipe, FIFOs, sockets, terminals, devices, and regular files.

In POSIX API, we open and create a file descriptor using

```
file_descriptor = open(path,flags,mode);
```

returning a -1 means an error occurred and errno is set appropriately. The most common flags I use are O_RDONLY, O_WRONLY, O_RDWR, and O_CREAT, O_APPEND. Using O_CREAT flags will give you the option that if there is not a file exist in that name, it will create new one with the mode permission, such as S_IRWXU for user has read write and execute permission. Using O_APPEND is equivalent to use

```
lseek(int fd, off_t offset, int SEEK_END)
```

to position offset at the end of the file.

```
num_read = read(fd, buffer, count)
```

This function reads from the open file referred to by fd then put the content in the buffer. it returns the number of how many 8 bits it reads.

```
num_written = write(fd, buffer, count)
```

is kind of similar to read, but writes from buffer to the open file referred to by fd. It returns the number of how many 8bits it written. The way to close a file is fairly straight forward, just

```
close(file_descriptor);
```

In WIN API, the way to copy a file has lots of simalarity as POSIX API.

The way to create a file is very different. A struct like HANDLE must be used to create a file.

2.1 Block I/O

Overview: Block I/O devices are used by random access with same size blocks of data. Each small unit on a block device is called sector. All device I/O will be done in units of sectors, which are then allocate to blocks in memory [1](pg. 289-290).

The sub-system performes these I/O operations is the block I/O scheduler. Block devices use request queues for keep their pending I/O requests [1](pg. 297). The I/O scheduler functions by managing this request queue, deciding the order of requests in the queue and when to dispatch each request to the block device. Seek time is the amount of time required to position the hard disk's head at the location of a specific block. One of the primary optimizations that can be made to block I/O performance is minimizing seek time. I/O schedulers sometimes do this by merging and sorting the requests in the queue. The entire request queue is kept sorted sectorwise so that all seeking along the queue moves sequentially [1](pg. 298-299).

Algorithms/Scheduler

There are several types of I/O schedulers built into the Linux kernel we worked with already and these schedulers are optimized for different purposes. The NOOP elevator is simple. It does not perform any data ordering. NOOP merges requests into queue. The Deadline I/O Scheduler is designed to prevent starvation of certain requests (such as write operations starving reads) [1](pg. 300). The Shortest Seek Time First sheduler is pretty self explainning. Lower sector numbers mean that the cylinder is closer to the spindle, while higher numbers indicate the cylinder is further away. The sstf algorithm check which request is closest to the current request position of the head, and then maniplate that request next[4].

In the version of the Linux kernel we were working with, the default scheduler was the Completely Fair Scheduler (CFS). The Completely Fair Scheduler (CFS) is a process scheduler. It was introduced into the 2.6.23 release of the Linux kernel. It handles CPU resource allocation for all running processes. It goal at maximizing overall CPU utilization and maximizing user interactive response performance. The idea behind CFS is to model scheduling as if the system had an ideal processor that could multitask perfectly. In this kind of system, each runnable process n would receive 1/n of the CPU time. CFS run processes round-robin style for very small tasks (in quantities called slices of time), so that in a given period of time it will appear as though many processes are running in parallel [1](pg. 48-50).

CFS is not the only type of scheduling available in the Linux kernel. The Linux scheduler provides soft real-time behavior, it tries to schedule processes within timing deadlines. Linux provides two real-time scheduling policies, FIFO and round-robin [1](pg. 64). sentially queues containing process descriptors for runnable processes. Round-robin scheduling cycles through processes, running each for a pre-specified amount of time known as a timeslice. On a more technical level, this means placing the process descriptor at the end of the runqueue after its timeslice, then running all subsequent tasks until it reaches the end of the queue, at which time it will start the cycle again. FIFO (First In, First Out) scheduling runs each FIFO process of the same priority until it is finished, then runs the next process in the runqueue. It does this in order of when processes were placed in the runqueue. On a technical level, FIFO scheduling behaves almost identically to round-robin scheduling but with infinite timeslices.

2.2 Character I/O

Overview: A key requirement of any realtime operating system is high-performance character I/O. Character devices can be described as devices to which I/O consists of a sequence of bytes transferred serially, as opposed to block-oriented devices (e.g. disk drives). Character devices are accessed one byte at a time.

Linux

As in the POSIX and UNIX tradition, these character devices are located in the OS pathname space under the /dev directory. For example, a serial port to which a modem or terminal could be connected might appear in the system as:

/dev/ser1

Typical character devices found on PC hardware include:

serial ports

parallel ports

text-mode consoles

pseudo terminals (ptys)

Windows

The Windows operating system has it's own system for I/O and utilizes it's own I/O manager. The I/O manager is the core of the I/O system on Windows because it defines the orderly framework within which I/O requests are delivered to device drivers. [2] The I/O system on Windows is packet driven and is represented by an I/O request packet called an IRP. These IRPs are designed to travel between one I/O system to another, allowing an individual application thread to manage multiple I/O request concurrently. An IRP is created in memory by the I/O manager representing an I/O operation. A pointer to the IRP is then passed to the correct driver until the completion of the I/O operation. Once the I/O operation has completed or needs to be passed on to another driver, the IRP is sent back to the I/O manager to either dispose of, or pass on to another driver. In addition to handling IRPs, the I/O manager provides flexible I/O services that allow environment subsystems, such as Windows and Posix to implement their respective I/O functions. The I/O manager has no knowledge of anything except for files, and passes the responsibility to translate file-oriented comments such as open, close and read to the driver. A typical I/O request starts with an application executing an I/O related function that is processed by the I/O manager, one or more device drivers, and the hardware abstraction layer (HAL). In Windows Operating System, threads perform I/O on virtual files which refer to any source or destination for I/O that is treated as if it were a file. This can be anything from files, directories pipes and mailslots.

FreeBSD

FreeBSD is a Unix-like operating system that descended from Research Unix via the Berkeley Software Distribution. Since FreeBSD is an operating system that is Unix-like, it contains a lot of features that are derived from UNIX.

The basic model of the Unix I/O system is a sequence of bytes that can be accessed either randomly or sequentially, and there are no access methods and no control blocks in a typical Unix user process. FreeBSD categorizes hardware devices by being either structured or unstructured. Structured devices, also known as block devices, are typically disks, magnetic tapes and include most random access devices. On block oriented devices, the FreeBSD kernel supports read-modify-write-type buffering actions to allow reading and writting in a totally random byte-addressed fashion. On FreeBSD filesystems are created on block devices. Unstructured devices, also known as character devices, are devices which do not support a block structure. These can include communication lines, raster plotters and unbuffered magnetic tapes and disks.

Character devices do typically support large block I/O transfers. The BSD kernel introduced an IPC mechanism which is more flexible than pipes and is based on sockets called the Socket IPC. A socket is an endpoint of communication referred to by a descriptor. Two processes can each create a socket, and then connect those two endpoints to produce a reliable byte stream. This new Socket IPC mechanism does essentially what was available previously with the use of pipes without the need for a common parent process to setup the communication channel. The transparency of sockets allows the kernel to redirect the output of one process to the input of another regardless if they have a common parent process. The socket IPC mechanism can also create a connection between sockets on two unrelated process and it can even connect them on different machines. This mechanism widely opens up the functionality of communication between processes however does require extensions to the traditional Unix I/O system calls.

Previously, the read and write system calls were used for byte-stream type connections, but six new calls were added to allow sending and receiving addressed messages. The system calls for writing include send, sendto and sendmsg, and the system calls for reading messages include recv, recvfrom, and recvmesg. The FreeBSD kernel used to rely on the traditional read and write system calls, however 4.2BSD introduced the ability to do scatter/gather I/O. [5] Scatter/gather I/O is where a single procedure call sequentially reads data from multiple buffers and writes it to a single data stream, or reads data from a data stream and writes it to multiple buffers. Scatter input uses the readv system call to allow a single read to be placed in several different buffers while the writev system call allows several buffers to be written in a single atomic write. Instead of the traditional read and write which passes a single buffer and length parameter, the scatter/gather process passes in a pointer to an array of buffers and lengths, along with a count describing the size of the array.

3 PROCESSES AND THREADS

3.1 Processes

Process is an instance of a computer program that is being executed. It contains the program code and its current activities. Each process has a unique id call *pid*, when you want to do something with the particular process, you can always refer to the *pid*.

Linux

To create a process in Linux, the *fork()* system call is used. it creates a child process from the currently executing process which is referred to as the parent. It is common to initiate a new program after the *fork()* command. *fork()* returns interesting things, it will return 0 for child case, -1 for error as usual, child process ID to parent process. Thus, we can use a switch to forking different processes. Child has a unique process ID and its own copy of the parent's descriptors. If the parent process relies on the child process it can inquire about the child's status by using the *wait()* system call. When we using forking, we have to make sure the parent process get killed after child process, otherwise, the child process will be in the middle of nowhere then become a zombie process.

Child processes when completed exit using the *exit()* system call, however, these processes are not completely destroyed from the system quite yet. When a child process calls *exit()*, it is put into a "Zombie" state until the parent calls *wait()* [1].

Windows

In Windows operating system, the main principles of processes are preserved with process having their own address spaces. However, Microsoft Windows does it slightly different. Windows uses the CreateProcess() system call which functions similar to the Linux fork() implementation. All of the information needed to execute a program is passed in to the CreateProcess() function. Since Windows does not clone the parent process like Linux does the CreateProcess() function requires many more parameters to achieve the same effect but potentially has more control of the child process [2].

Creating child processes is a foreign concept to Windows API. However, there are several Windows functions that emulate the forking process in POSIX API. This is not meaning that Windows API does not support multiple processes; it just works in a different way. The CreateProcess function in Windows is the equivalent of using fork(), execl() and system() respectively in UNIX. There are Windows equivalent for exit(), getpid(), wait() and kill() in UNIX.

The way creating child processes in Windows is different from POSIX API. Windows treats things more as objects rather than files and is evident in some of the Windows functions for process management. The process handlers for Windows are WaitForMultipleObjects and WaitForSingleObject, which is like calling waitpid() or wait() in UNIX, but notice the names of these functions. They are focused around objects and not ”children” like in UNIX. The biggest difference between multiple processes in Windows and UNIX has to be the concept of the parent and child. Because of this, there are no Windows equivalents for the exec() functions and its variants.

In WINAPI, when you creating a new process, the new process runs independently from the creating process, usually refer to parent- child relationship which is same as POSIX processing.

If CreateProcess succeeds, it returns a PROCESS_INFORMATION structure containing handles and identifiers for the new process and its primary thread. You also need a wait function to wait on child process to stop before parent process. The thread and process handles are created with full access rights, although access can be restricted if you specify security descriptors. When you no longer need these handles, close them by using the CloseHandle function.

3.2 CPU scheduling

Overview: A scheduler is the mechanism for sharing the CPU between multiple processes in an OS. The scheduler makes it possible to execute multiple programs at the same time, thus sharing the CPU with users of varying needs. The goal of a scheduler is to allocate CPU resource efficiently.At the meantime it has to provide a responsive user experience. scheduler runs determined on policies. A policy is the behavior of the scheduler that determines what process runs when. Changes in policy can result in operating systems that are optimized for specific tasks (such as user responsiveness). Therefore, this is a critical aspect of an operating system [1](pg. 43).

Linux

In the version of the Linux kernel we were working with, the default scheduler was the Completely Fair Scheduler (CFS). The Completely Fair Scheduler (CFS) is a process scheduler. It was introduced into the 2.6.23 release of the Linux kernel. It handles CPU resource allocation for all running processes.

It goal at maximizing overall CPU utilization and maximizing user interactive response performance. The idea behind CFS is to model scheduling as if the system had an ideal processor that could multitask perfectly. In this kind of system, each runnable process n would receive 1/n of the CPU time. CFS run processes round-robin style for very small tasks (in quantities called slices of time), so that in a given period of time it will appear as though many processes are running in parallel [1](pg. 48-50).

CFS is not the only type of scheduling available in the Linux kernel. The Linux scheduler provides soft real-time behavior, it tries to schedule processes within timing deadlines. Linux provides two real-time scheduling policies, FIFO and round-robin [1](pg. 64). sentially queues containing process descriptors for runnable processes. Round-robin scheduling cycles through processes, running each for a pre-specified amount of time known as a timeslice. On a more technical level, this means placing the process descriptor at the end of the runqueue after its timeslice, then running all subsequent tasks until it reaches the end of the queue, at which time it will start the cycle again. FIFO (First In, First Out) scheduling runs each FIFO process of the same priority until it is finished, then runs the next process in the runqueue. It does this in order of when processes were placed in the runqueue. On a technical level, FIFO scheduling behaves almost identically to round-robin scheduling but with infinite timeslices.

FreeBSD

ULE is the successor to the traditional BSD scheduler. FreeBSD start to use ULE from 7.0. It offers much improved performance on SMP systems as well as uniprocessor systems. It follows a more traditional design with run queues and time slices. It strives to be fair, but can be instructed to favor interactive processes. ULE should improve performance in both uniprocessor and multiprocessor environments,[6] as well as interactive response under heavy load.[5]

Windows

Windows NT processor scheduling detemines which job is going to load onto CPU at the given time. Because first in first out queues is not optimal for Windows usually, it classified different processes with priority. Windows NT treats same prioty as equal, it assigns time slices to them equally in a round-robin fashion. If a higher priority thread becomes avilable to run, Windows NT suspend lower priority tasks and start excuting the higherpriority one[6].

There are six named priority levels:

- Realtime
- High
- Above Normal

- Normal
- Below Normal
- Low

Applications start at a base priority level of eight. The system dynamically adjusts the priority level to give all applications access to the processor.

3.3 Threads

Threads function slimilarly to that of processes in regards to performance but with the added benefit of being able to share the same namespace as the parent, that is to say, a thread is a process with a share flag eabled. Threads are essentially ”light weight” process since they do not require their own address space and namespace it is more memory efficient, however, threads can not have threads of their own.

Linux

The Linux kernel views each thread as a unique process that merely shares resources of another process. Every data structure that is used for a process is also used for a thread within a Linux system. While Microsoft Windows has specific kernel support for threads to allow threads to be simple forms of processes, Linux on the other hand does not implement any thread improvements since Linux processes themselves are already considered ”Light Weight”. The Linux implementation of a thread is the same as that of a process and are created the same way. Linux creates a thread with the same **clone()** system call as with a process, with the exception of a few flags. The flags passed to the **clone()** system call instruct the kernel to share the address space, filesytem resources, file descriptors, and signal handlers to the newly created thread [1].

Windows

Like mentioned above Windows has specific implementations to create a thread unlike the universal **clone()** system call that Linux uses. To create a thread in Windows the **CreateThread()** is used. Each thread created in Windows shares the thread context of the main thread (parent process). The thread context includes the thread’s set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space. Windows threads also acquire the same security context as that of their parent unless specifically setup to receive otherwise [2].

FreeBSD

Kernel threading was not with early version of free BSD. It was until the release of FreeBSD 5.0. However, it worked but did not perform well, so from version 7.0 onward, FreeBSD started using a 1:1 threading model, called libthr libthr which is 1:1 threading [5](2.3.1.5). It is also used for 1:1 threading library as pthread's thread ID but handling of this is internal to the library and cannot be relied on.

References

[1] R. Love, *Linux Kernel Development*, 3rd ed. Pearson Education, Inc., 2010.

[2] M. Russinovich, *Windows Internals*, 6th ed. Microsoft Press, 2012.

[3] freebsd.org, "The z file system," https://www.freebsd.org/doc/handbook/zfs.html.

[4] I. I. of Technology, "Disk scheduling algorithms," http://www.cs.iit.edu/~cs561/cs450/disksched/disksched.html.

[5] freebsd.org, "A look inside..." https://www.freebsd.org/doc/en_US.ISO8859-1/articles/linux-emulation/inside.html.

[6] msdn.microsoft.com, "Scheduling prioities," https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx.