Operating System Feature Comperison: I/O

by Sinan Topkaya

CS 444 - Spring 2016

Abstract: This paper will examine I/O and functionalities such as data structions and algorithms. I will be examining how Windows implements them and also how FreeBSD implements them then compare it with how Linux implements them. This paper will mainly focus on different types of devices, I/O scheduling and the like.

WRITING ASSIGNMENT - I/O COMPARISON

WINDOWS

The Windows I/O System consists of many executive components that together manage devices.

I/O System Components

1) Uniform security and naming across devices to protect shareable resources.
2) High-performance asynchronous packet-based I/O to allow for the implementation of scalable applications.
3) Services that allow drivers to be written in a high-level language and eaasily ported between different machine architectures.
4) Layering and extensibility to allow for the addition of drivers that tranparently modify the bhavior of other drivers or devices, withour requiring any changes to the driver whole behavior or device is modified.
5) Dynamic loading and unloading of device drivers so that drivers can be loaded on demand and not consume system resources when unneeded
6) Support for Plug and Play, where the system locates and installs drivers for newly detected hardware, assigns them hardware resources they require, and also allows applications to dis-cover and activate device interfaces
7) Support for power management so that the system or individual devices can enter low power states.
8) Support for multiple installable file systems, including FAT, the CD-ROM file sustem(CDFS), the Universal Disk Format (UDF) file system, and the Windows file system (NTFS)
9) Windows Management Instrumentation (WMI) support and diagnosability so that drivers can be managed and monitored through WMI applications and scripts.[1]

To implement the above features Windows I/O system consists executive components as well as device drivers.

The I/O system includes the hardware abstraction layer(HAL) at the bottom, this insulates drivers from the specifics of the processor and interrupt controller by providing APIs that hide differentces between platforms. We can think of HAL as the bus driver for all the devices. The next layer includes the different Drivers, these drivers are then accessed by the I/O system layer which include, WDM WMI routines, PnP manager, Power manager and I/O manager.

The I/O system layer work closely with each other.

Windows Management Instrumentation support routines, called the Windows Driver Model(WDM) WMI provider, allow device drivers to indirectly act as providers. I/O manager is the heart of the I/O system. It connects applications and system components to virtual, logical, and physical devices, and it defines the infrastructure that support device drivers. The Pnp manager works closely with the I/O manager and a type of device driver called a bus driver to guide the allocation of hardware resources as well as to detect and respond to the arrival and removal of hardware devices. The power manager also works closely with the I/O manager and the PnP manager to guide the system, as well as individual device drivers, through power-state transitions.

Windows supports a wide range of device driver types and programming environment. These drivers could be user-mode or kernel-mode drivers.

Windows Drivers

User mode drivers include windows subsystem drivers, that translate device-indipendent graphics requests to printer-specific commands. These commands are then typically forwarded to a kernel-mode port driver such as the universal serial bus(USB) printer port driver. User-Mode Driver Framework(UMDF) is also another user-mode driver that windows supports. These drivers are hardware device drivers that run in user mode. They also communicate to the kernel-mode.

Kernel-mode drivers include WDM driver, Layered Driver.

WDM Drivers are device drivers. WDM includes support for Windows power management, Plug and Play, and WMI, and most Plug and Play drivers adhere to WDM. There are 3 types of WDM drivers:[1]

1) Bus drivers manage a logical or physical bus. Examples of buses include PCMCIA, PCI, USB, and IEEE 1394. A bus driver is responsible for detecting and informing the PnP manager of devices attached to the bus it controls as well as managing the power setting of the bus
2) Function drivers manage a particular type of device. Bus drivers present devices to function drivers via the PnP manager. The function driver is the driver that exports the operational interface of the device to the operating system. In general, it is the driver with the most knowledge about the operation of the device.

3) Filter drivers logically layer either above or below function driver, augmenting or changing the behavior of a device or another driver. For example, a keyboard capture utility could be implemented with a keyboard filter driver that layers above the keyboard function driver.

Support for an individual piece of hardware is often divided among several drivers, each providing a part of the functionality required to make the device work properly. In addition to WDM bus driver, function driver, and filter driver, hardware support might be split between the layered driver and they are as follows: [1]

1) Class drivers implement the I/O processing for a particular class of devices, such as disk, keyboard or CD-ROM, where the hardware interfaces have been standarlized, so one driver can serve devices from a wide variety of mufacturers
2) Miniclass drivers implement I/O processing that is vendor defined for a particular class of devices
3) Port devices implement the processing of an I/O request specific to a type of I/O port, such as SATA, and are implemented as kernel-mode libraries of functions rather than actual device drivers.
4) Miniport drivers map a generic I/O request to a type of port into an adapter type, such as a specific netwrok adapter. Miniport drivers are actual device drivers that import the functions supplied by a port driver.

I/O Processing depends of the type of I/O in Windows. For example asynchronous I/O allows an application to issue multiple I/O requests and continue executing while the device performs the I/O operation. Synchronous I/O is when the application thread waits while the device performs the data operation and returns a status code when the I/O is complete. When used in their simplest form, Windows ReadFile and WriteFile functions are executed synchronously. Regardless of the type of I/O request, internally I/O operations issued to a driver on behalf of the application are performed asynchronously, that is, once an I/O request has ben initiated, the device driver returns to the I/O system.

Another usefull I/O is called the fast I/O. This is a special mechanism that allows I/O system to bypass generating an IRP and isntead go directly to the driver stack to complete an I/O request. Mapped File I/O is an important feature of the I/O system, one that the I/O system and the memory manager produce jointly. Mapped file I/O refers to ability to view a file residing on disk as part of a process vitual memoery. A program therefore can access the file as a large array without buffering data. Scattered/Gather I/O is also supported by windows. This is a high performance I/O and avalaible through ReadFileScatter and Write FileGather functions.

```
// tHIS IS THE readfilescatter function THAT IS USED BY THE SCATHER/GATHER I/O

BOOL WINAPI ReadFileScatter(
   _In_         HANDLE                 hFile,
   _In_         FILE_SEGMENT_ELEMENT   aSegmentArray[],
   _In_         DWORD                  nNumberOfBytesToRead,
   _Reserved_   LPDWORD                lpReserved,
   _Inout_      LPOVERLAPPED           lpOverlapped
);
```

FreeBSD

Similar to Windows, the threads of a process operate in either user mode or kernel mode. The kernel state includes parameters to the current system call, and have 2 primary strctures; process structure, thread structure. FreeBSD supposers transparent multiprogramming, it does so by context switching, by swithcing between the execution context of the threads within the same or different process. A mechanism is also provided for scheduling the execution of threads, that is for deciding which one to execute next.

Now I will first write about how FreeBSD implements Process then I will talk about how it implements threads and how it deal with scheduling.

Processes: First I will write about the Process strcuture then I will briefly explain what they are, later I will write more about the state of a process. Every running command starts at least one new process and there are a number of system processes that are run by FreeBSD. Each process is uniquely identifies by a number called PID. Similar to files each process has one owner and group, and the owner and group permissions are used to determine which files and devices the process can open. The command `%ps` allows the user to display a list of the currently running processes, their PIDs, and how much memory they are using, this is the structure of a process. The processes can be scheduled, to specify which process to run first or next.[2]

Process Structure:

1) Process identification: the PID and the parent PID
2) Signal Slate: Signals pending delivery and summary of signal actions

3) Tracing: Process tracing information
4) Timers: Real-Time timer and CPU-utilization counters

Process states inlude NEW, NORMAL and ZOMBIE states. NEW specifies that there is a undergoing process creation, NORMAL specifies that threads will be runnable, sleeping or stopped, and the ZOMBIE state is when the process in undergoing termination. A process may create a new process that is a copy of the original by using the fork system call. The fork call return twice: once in the parent process, where the return value is the process identifier of the child, and once in the child process, where the return value is 0.

The kernel begins by allocating memory for the new process and thread entries. These thread and process entries are initialized in the steps: One part is copied from the parent's corresponding structure, another part is zeroed, and the rest is explicitly initilized. The zeroed fields include recent CPU utilization, wait channel, swap and sleep time, timers, tracing, and pending-signal information. The copied portions include all the privilages and limitation inherited from the parent.

Threads: Thread Structure

1) Scheduling: The thread priority, use-mode scheduling priority, recent CPU utilization and amount of time spent sleeping; the run state of a thread; additional status flags; if the thread is sleeping, the wait channel, the identify of the event for which thread is waiting and a pointer to a string describing the event
2) TSB: the user- and kernel-mode execution states
3) Kernel Stack: the per-thread execution stack for the kernel
4) Machine state: The machine-dependent thread information[2]

The `kthread_add()` is used to create a kernel thread. And the new thread rund in kernle mode only. It is alter then added to the process specified by procp argument. if this thread wanted to be started then `kthread_start()` is used. The structures used by this function includes name of the thread, the function for this thread to run. Later the caller must `sched_add()` to start the kernel. POSIX threads can also be created using FreeBSD.

CPU Scheduling: Kernel switches among threads in an effort to share CPU effectively; this activity is called context switching. Swithing between thread can occir synchronously or asynchronously. Mutex

Synchronization can be used for short-term thread sychronization.i

There are number of ways the scheduling can be done, like timeshare thread scheduling, multiprocessor scheduling, adaptive idle and Traditional Timeshare Thread Scheduling.

For example multiprocessor scheduling is used when a runnable as a result of wakeup, unlock, thread creation, or other event is called by `sched_pickcpu()` The best CPU to run is chosen(if there are multiple processors.) The whole system is searched for a least loaded CPU, and also checks to see the priority of the threads its running. Using several algorithms it decides the best fit CPU for the work.[2]

Answer to the questions

Windows

How is it different compared to Linux? and How is is similar to Linux? and Why?: Process and Threads: In Windows a process hold the address space, handle table, statistics and at least one thread and it does not inherent parent/child relationship. While Linux process is called a task and although it also hold the informations basic address space, handle table and statistics it does not have to have a thread. Linux does inherent parent.child relationship and uses basic scheduling unit to schedule tasks. Inlinux there are no threads per-se, tasks can act like Windows threads by sharing handle table, PID and address space. Windows uses basic scheduling unit to schedule threads and Fibers are used for cooperative user-mode threads while Pthreats are used for Linux for cooperative user-mode threads

CPU Scheduling: In Windows there are 2 scheduling classes: Real-time and Dynamic, while Linux can have 3 scheduling classes, normal, fixed round robin and fixed fifo. Windows favors higher priorities while threads favor lower priorities. In linux priorities of normal threads decay as they use CPU, and priorities of interactive threads boost. In Windows prioritie of dynamic threads get boosted on wakeups and thread prioroties are never lowered. In both operating systems the newly created thread starts with a base priority.

Although they have differences between each other, a user can ultimately accomplish the same things. It is a matter of what you need and if possible to know, want to use. We must understand that although the functions work differently they ultimately try to accomplish the same end result. For example `pthread_create` and `CreateThread` are really similar however they have different parameters.

FreeBSD

How is it didderent compared to Linux? and How is it similar to Linux? and Why?: Freebsd and Linux have different CPU scheduler, this mainly impacts the process management, specifically with regards to multicore scnarious. As mentioned above FreeBSD uses alot of different kinds of algorithms for CPU scheduling. The similarity between each other is that theya r eboth POSIX systems. If we are to compare default scheduler for both systems Linux uses CFS while FreeBSD uses ULE. Although ULE is similar to CFS it can be instructed to favor interactive processes.

References

[1] D. S. Mark Russinovich and A. Ionescu, *Windows Internals, Part 1 6/e*. Microsoft Press, 2012.

[2] M. K. McCusick and G. V. Neville-Neil, *Design and Implementation of the FreeBSD Operating System 2/e*. Addison-Wesley, 2015.