Processes, Threading and CPU Scheduling

George Crary

Operating Systems II

Spring 2017

Abstract

Between the Windows, Linux and FreeBSD operating systems many similarities exist as they fundamentally try to achieve the same things however differences do still exist. This analysis inspects these details and the origins that led to them. While both FreeBSD and Linux share lineage from Unix (Linux via Minix), Windows does not obviously. The result of these differences can be interpreted by the organizational style and control that they come from. For example the Linux development team is purely concerned with the Linux kernel itself while the FreeBSD organization involves itself with both kernel development and core system utilities. FreeBSD in comparison to Linux is subject to a more structured design as a result of this. Microsoft on other hand has development interests from the Windows kernel all the way out to userland and their design commitments reflect that. Additionally the historical environment that these operating systems were developed in also have an influence worth noting. FreeBSD‘s Unix ancestors hail from the days of the 16-bit PDP11 while FreeBSD and Linux began their lives on the x86 platform. Windows began targeting x86 with the release of Windows NT as well.

CONTENTS

I. PROCESSES

Processes throughout the operatings systems contain a large amount of similarities. Throughout the implementations of such concept all of them share such things as program code, open file descriptors, signaling and threading facilities. In addition other living aspects of the program are stored in a process such as run time, threads that are being executed, a data section for global variables and a private virtual address space for its given threads. Information for the CPU scheduler is also maintained within the process data structure as well. For FreeBSD and Linux these similarities are contained in their respective task_struct data structure and are very similar as result of being Unix derivatives. In Windows these details are contained in the Windows Executive Process structure, also known as the EPROCESS structure. On top of that some process related data in Windows such as the Process Environment Block live in userland address space for accessibility reasons which is another difference [1].

Within the Unix realms FreeBSD and Linux both begin new processes with a fork system call. The fork calling process is turned into a parent process while the newly forked process is called a child process. In FreeBSD this can involve the rfork system call, which can handle controlling which resources get shared between the parent and child, while on Linux this is similarly handled with the clone system call [2]. Once the child process is up and running an exec system call can be made to run an entirely different program. This exec call overlays the program into the programs address space and begins executing it. The separation of these two system calls allows for any necessary preparation and cleanup to take place before the new program runs. Windows handles process creation in a similar fashion however its much more streamlined. Within Windows CreateProcess gets called and handles identifying the application type. As Windows supports executing Win16, Posix, MS-DOS executables these must be handled additionally with the standard exe executables. In those respective situations the appropriate Windows support image is executed and handled from there [1]. These differences between process creation really speak to the philosophical differences of the operating systems. The fork and exec system calls used in Linux and FreeBSD highlight the mindset of ”Do one thing right and do it well” and showcases the composition of powerful and concise utilities. The Microsoft way of doing things showcases a commitment to backwards compatibility and total control by consolidating process creation into a single interface.

II. THREADING

Threading is a common concept to all three operating systems. Significant differences begin to show however between Windows and the Unix related operating systems. Even within the Unix family threading is different. Threads are a facility for multiple paths of execution for a given process to occur at a single time while allowing for process resources to be shared. On a multicore machine this means threads can run in parallel potentially which enables a concurrent model of programming for a process. Within the Linux world threads are just seen as special processes and arent scheduled in a particularly different way. Within FreeBSD threads are looked upon as lightweight processes as context switching between threads is considerably faster than a context switch between processes [3]. Windows has a similar notion of threads as lightweight processes but also provides a userland scheduled threading facility called Fibers and another concept known as jobs. Jobs simply represent a collection of processes that are to be managed together.
Across the Unix systems the threading support is largely standardized thanks to POSIXs pthreads library.

One of the largest similarities between the three operating systems when it comes to threading is the User mode and Kernel mode distinction. Threads can operate in either User mode and Kernel mode and have their own respective access levels to address spaces. The user mode limitations to address space can help protect the operating system from exploitation in the event a server thread executing on the behalf of a client goes rogue. Separate stacks for these modes are given to threads as well in order for kernel specific operations to be cleanly supported from user mode operations.

III. CPU Scheduling

CPU Scheduling is a common source of difference between the three operating systems. As the operating system must schedule the order of processes and thread operation across limited computing resources different approaches arise. Each operating system provides facilities for giving processes and threads different levels of priorities. In Linux for example this consists of a niceness value ranging from -20 to +19 with 0 as the default value. In FreeBSD this value ranges from 0 to 255 with different priority classes existing to serve the needs of user mode and kernel mode threads. Windows implements priority across the range 0 to 32 with the greater half representing real time priority levels and the lower end representing variable levels with 0 reserved for the zero page thread [1].

The schedulers behind each of these operating systems are drastically different. After Linux kernel version 2.5 was released a constant time scheduler, also known as the O(1) scheduler, was used. This scheduler could determine the next process to be ran in constant time. This behavior benefited high process count workloads seen on big iron workstations for example as the previous algorithm wasted a lot of time running in linear time. By version 2.6 this was changed to a new scheduler named CFS, the completely fair scheduler [2]. To handle scheduling fairness better CFS divides the processor time uniformly across N processes and then weighs given times appropriately with respect to the relative nice values.

FreeBSD uses the ULE scheduler which is named after a filename pun. This scheduler is divided into two schedulers. The low level scheduler and the high level scheduler. The former handles selecting a new thread to run from a non-empty run queue in a round robin fashion whenever a thread blocks [3]. The high level scheduler handles run queues for each CPU and maintaining the thread priorities. The Windows scheduler is similar to the ULE scheduler in terms of its high level CPU concerns and low level task picking. In Window‘s case however processor affinity is used to handle the high level case of where a thread may be executed.

IV. IO Introduction

The input output (IO) subsystem to any operating system is fundamental to achieving any meaningful real world computation. In the before times programs were simply loaded, ran and the result was dumped out at the end for inspection. Of course overtime more interactive facilities for providing input state to programs became developed.

To support different IO devices of differing standards, specifications and vendors the concept of device drivers eventually became a thing. These drivers would handle controlling the underlying hardware while providing a software api for the operating system to interact with. These drivers support all sorts of IO devices from spinning hard drive disks to keyboard devices.

Within the Unix realm theres a common philosophy [2] that everything is a file. While this isnt exactly true, sockets for example arent necessarily files, but most things can act like files thanks to file descriptors which unites the common file-esque access patterns across all devices. This approach provides a consistent and reusable interface and mentality to developing within Unix that has contributed to its longevity as an operating system. With everything as a file as the developing mentality utilities from decades ago can remain largely stable, maintainable and interoperable with other programs and especially other devices of any kind. Once a driver for a device has been written any program can act with it in that common way with a relative degree of smoothness.

Windows on the other hand doesnt subscribe to this philosophy at all and its really a matter of Unixs origin as a OS made by developers for developers. From an operating system designers standpoint its a lot easier to get developers to use your operating system if you simply tell every new user (for Unixs case these were other developers within Bell Labs) to just treat everything as a file. Obviously this approach is simplistic despite it mulling over internal aspects, it worked tremendously as far as growing a community of hardcore enthusiasts willing to hack on Unix and derivatives to what it is today. As for Windows, this kind of user base growth wasnt necessary nor wanted as internal APIs and order could be created for their own developers and its target customer base had no need to interact with everything as a file.

In Linux this philosophy can be applied to IO devices, as devices files, but there is an underlying distinction between block devices that spit back chunks of data in blocks and character devices that return streams of characters. In FreeBSD however this isnt the case. [4] Support for block devices were dropped since caching for disk block devices reorders writes making it hard to know what has been physically committed to disk in the event of a crash recovery. This is a strong example of FreeBSDs design philosophy.

V. Driver Models

Across both Unix and Windows a device interaction interface is provided. Within the Windows realm of things this is called the Windows Driver Model known as the WDM. The WDM was a kernel change introduced in Windows 2000. Within Linux this is similarly called the Linux Kernel Device Model. For Windows however drivers are written using the Windows Driver Kit towards the WDM standard that all hardware devices and their accompanying software drivers have to comply with. The Windows Driver Model at a high level also includes many stipulations for power management and the Plug and Play model of device usage. [1] The primary goal of this is to increase usability from an average users standpoint (everyone knows how to plug a device in and play) while reducing complexities overall for software developer as far as supporting devices on a driver by driver basis.

VI. Async IO

Asynchronous IO is also a facility provided by the Windows and the Nixs on top of regular IO. Also known as Async IO, it allows for a program to dispatch an IO request and continue execution. FreeBSD and Linux are extremely similar in this aspect as they both implement the POSIX api for async io. Windows on the other hand provides support for this using IRP, known as I/O request packets. Attached is the facilities provided within the Posix for Aysnc IO.

VII. Dynamically Loadable Modules

Between both Windows and the Unixs facilities for dynamically loaded modules are available. In Windows these are called dynamically linked libraries, aka DLLs, and within Linux these are are called shared object files (with the .so extension). The advantage of dynamically loaded modules is that they can expose and provide system functionality between multiple programs without requiring the duplication of static code. Without DLLs or shared objects each program binary would involve lugging around its own library code which would increase the overall size of all binaries throughout the system. Additionally when multiple drivers mutually use common functions this is called stacked device drivers. This concept is common to Windows, Linux and FreeBSD.

VIII. IO Scheduling

When it comes to scheduling IO operations stark differences between the operating systems begin to appear. IO scheduling is an important issue when it comes to handling physical devices as many IO devices such as platter spinning hard drives need to actuate a read head across the disks cylinders. As the the read head is actuated it seeks to whatever sector its trying to read. Naive IO schedulers can cause the read head to waste time seeking between sectors when this activity can be optimized to reduce the total range of motion for the seek head and increase the lifespan of the device. Optimizations as such can increase the throughput of IO reads and writes overall however fairness is a consideration that comes into play. As in most cases its not just a single process making tons of different requests, its multiple processes doing their requests. In the event that requests get served unfairly a process might become IO starved and wait too long for it. Therefore in Linux the default scheduler is the CFQ, Completely Fair Queuing scheduler, that tries to completely be fair by dividing up IO time in nanoseconds based on the requesting processes [2]. FreeBSD on the other hand uses a CLook elevator scheduler [3]. Lastly Windows has a priority based IO scheduler similar to its process scheduler [1].

IX. Disk Hibernation

In terms of physical device features Disk Hibernation is a common feature to all three operating systems. Disk Hibernation consists of the saving the operating systems to disk and powering it off during periods of long inactivity to conserve power. This can be useful for spare drives on a system that arent in use as theres no reason to keep spinning platters.

In Windows however this a much more prominent feature as Windows in general had an early lifecycle targeted more towards your average consumer, laptops including, so power consumption is a larger focus. This is in contrast to the server mainframe customer base of Unixs early history where time.

X. Memory Management

Across all three operating systems memory management is very critical subsystem important to all aspects of program life. Fundamentally all of these operating harness the concept of virtual memory spaces as an abstraction on top of physical memory. This abstraction is for the purpose of security and consistency. In terms of security it isolates processes memory spaces from one another. In a scenario where there isnt virtual memory spaces for processes a bugged program could overwrite into another process's memory space and stomp all over the data. In an adversarial situation this could be an attacker trying to overwrite program instructions for a privileged process to wreak havoc. So this ultimately serves as a first line of defence against that. In terms of consistency this concept of virtual memory allows for a more machine independent view of process memory as a whole.

XI. Pages

In addition to virtual address spaces, paging is a common scheme between these operating systems. Paging provides the operating system the ability to give processes more address space then there is available system memory by managing in memory pages along with pages on disk. The operating system handles this with swapping pages in and out of memory and this is usually handled by a page table. This is possible because in most cases the entire program does not have to reside in memory at a given time. A program might be handling hot parts of code for given durations and colder parts of memory can be swapped out. This feature can be extremely critical for multiprocessing systems as many processes can be running at a given time but not all parts of these processes have reside in memory at the same time either. This can lead to faster context switches throughout the stack.

Page tables across the operating systems are a bit different. In Windows there are two levels of indirections to how page tables are stored. There are page directories and then page tables after that layer. In Unix this distinction for page table storage is split across 4 separate layers (In some cases it's just 3 if the address spaces size doesnt demand it.) There's the page general directory, page upper directory, page middle directory and then finally the page table entry level. Within Windows there are two page sizes available. Large pages being of 2MB in size and small pages that are just 4KB in size. In Linux however there is a similar small page size of 4KB just like Windows. Along with that however there are hugepages in Linux and in FreeBSD these are called superpages. [3] Windows pages can exist in 4 different states. Free, renowned, committed and shared.

Copy on write is a common feature worth noting. Whenever a process is forked the memory address space of the parent process gets copied over to the new process. In many cases the child process will use the data copied over however rarely update any actual variables. So in this case its needless to copy all of it over until its actually necessary. This resolve this it became a common idiom to complete the actually copying once something is actually written in the forks child process. This is done at the paging level and is handled by the page table whenever theres an update to the childs private copy of a page. [2]

Listing 1. Brief Linux Shared Memory example.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("hello.txt", ’R’)) == -1) /*Here the file must exist */
{
        perror("ftok");
        exit(1);
    }

    /*  create the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }

    /* attach to the segment to get a pointer to it: */
    data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }

    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    return 0;
}
```

Within Windows many facilities are provided in terms of memory management. Address mapping ,paging, memory mapped files, copy on write memory are a few worth naming. When it comes to processes 32 bit processes are given a 2GB address space by default (this can be upped to 3 or 4 GBs on 32 bit and 64 bit Windows respectively). 64 Bit processes within Windows however can have a virtual address space up to 8192 GB in size. In the real world this is rarely seen as general purpose computing doesnt really need to access that much at a given time when solutions generally scale across machines/cores. The improvement past 2 GBs however is nice nonetheless.

In Windows there’s the concept of the working set which represents the set of pages physically present. This falls into 5 key parts that are unique to Windows memory management. A working set manager is provided. A stack swapper is available for rapid switching of stacks for a given process. Modified page and memory mapped pages have their own writers. On top of that a zero page thread exists as a special thread within the Windows operating system solely for zeroing out memory pages. In Linux a unique thread for returning zerod out pages does not exist but a function called ”get free page” does exist within the Kernel however. [2] These utilities can be very important as sensitive data may be freed up and returned to the kernel and then subsequently handed off to another process when it requests a page. This can be an attack channel for adversaries if not handled correctly. In FreeBSD this is handled by the default page handler ”pgo getpage”. [3] A lot of these subtle differences at least within Unix and Linux can be traced back to the hardware these operating systems were developed for. As memory became less of a luxury and systems became less dependant on disks for paging differences began to arise.

Windows also differentiates itself by being fully reentrant along with finely grained locking. With that it helps manage allocation and freeing of virtual memory. It also helps with shared mappings, mapped files and flushing of pages. In Linux there’s a legacy locking system called the Big Kernel Lock that was implemented for SMP systems originally. This has fallen out of use and is becoming practically deprecated as drivers are using it less and less. [2]

Processes in Windows can simply be seen as a container for threads. This is similar to the Unix concept of process groups. Strangely however Windows allows for multiple stacks and heaps to exist for a process. This is a big difference and can allow for switching between user mode and kernel mode very rapidly.

References

[1] Mark Russinovich, David Solomon, and Alex Ionescu, “Windows internals, part 1 6/e,” 2012.

[2] R. Love, “Linux kernel development 3/e,” 2010.

[3] M. K. McCusick and G. V. Neville-Neil, “Design and implementation of the freebsd operating system 2/e,” 2015.

[4] F. Community, “Freebsd architecture handbook,” 2016.