OPERATING SYSTEMS II
CS444 FALL 2017

OCTOBER 25, 2017

OPERATING SYSTEM FEATURE COMPARISON

PROCESSES AND SCHEDULING

BY

COREY HEMPHILL

Abstract

This document examines, compares, and contrasts low level operating system kernel operations and implementations such as processes, threads, and CPU I/O scheduling for Windows, FreeBSD, and Linux.

I. Introduction

It easy to pretend that a given operating system kernel performs all of its basic essential operations via some kind of black magic. The reality is that, to the layman, what goes on inside of an operating system really is magic. To an engineer, however, there is no such thing as magic. One of the most fundamental operations an operating system performs is I/O scheduling, for which kernels have entire block I/O subsystems to handle this functionality. It is essential for software engineers to have a fundamental understanding of the block I/O implementations of a number of different operating systems, as well as threading process implementations, in order to make technical decisions as a professional in the field. Here, we will examine both Windows and FreeBSD kernel block I/O implementations, processes, and threads, and we will compare them to the block I/O implementation, processes, and threads of the Linux kernel.

II. Compare and Contrast OS Processes

What is a process? Although this might seem like a relatively trivial question to answer, its not really. In terms of computing, its a question that requires a bit of technical knowledge to answer sufficiently. In both Windows and FreeBSD kernels, a process is a single instance of a program being executed by the CPU at a given time. A running process includes all of the necessary dependencies and resources such as included source code files, running threads, virtual address space, and system signal calls [1] [2]. A process maintains file descriptors which manage a number of vital items such as what CPU registers are being used, data stacks, tables for storing other file descriptors, and overall CPU runtime values. A process is identified via a PID. For the most part, Windows and FreeBSD processes are extremely similar with a few key differences. FreeBSD represents processes by a task struct whereas Windows represents a process via something they refer to as an executive process [1]. Furthermore, both operating systems utilize background processes. In FreeBSD, background processes are referred to as daemons [2], whereas in Windows they are referred to as Windows Services [1].

When comparing Windows and FreeBSD with the Linux kernel, one can see that inherently, a process is the same across the board; it is an instance of a program executed via the CPU. Processes, daemons, and Windows services all maintain similar functionalities and uses. In both Linux and FreeBSD, daemons are background processes which execute imperative system features such as logging [2] [3]. The main difference that stands out is between Windows and Linux in that background processes are named differently, and Windows processes also have the ability to manage files that currently live in memory, as well as print operations [1].

III. Compare and Contrast OS Threads

For all intents and purposes, a thread is the most basic unit of an operating system, and they are the unit for which CPU time is allocated during a processs execution. A thread is a high-level abstraction that provides the ability to execute more than one sequence for a given executing program. Threads allow for parallel processing and concurrency. FreeBSD and Windows operating systems implement both user and kernel modes; every single thread maintains its own stack and counter when running in kernel mode. Switching to user mode resets all of these values that a thread maintains [1] [2]. The following code excerpt is from the Windows threading implementation: [4] [5]

```
typedef struct _DEVICE_EXTENSION {
KEVENT evKill;
PKTHREAD thread;
};
NTSTATUS StartThread(PDEVICE_EXTENSION pdx)
{
NTSTATUS status
HANDLE hthread;
KeInitializeEvent(&pdx-¿evKill, NotificationEvent, FALSE);
status = PsCreateSystemThread(&hthread, THREAD_ALL_ACCESS,
NULL, NULL, NULL, (PKSTART_ROUTINE) ThreadProc, pdx);
if (NT_SUCCESS(status))
return status;
ObReferenceObjectByHandle(hthread, THREAD_ALL_ACCESS, NULL,
KernelMode, (PVOID*) &pdx-¿thread, NULL);
ZwClose(hthread);
return STATUS_SUCCESS;
}
VOID StopThread(PDEVICE_EXTENSION pdx) {     KeSetEvent(&pdx-¿evKill, 0, FALSE);
KeWaitForSingleObject(pdx-¿thread, Executive, KernelMode, FALSE, NULL);
ObDereferenceObject(pdx-¿thread);
}
VOID ThreadProc(PDEVICE_EXTENSION pdx)
{
KeWaitForXxx(¡at least pdx-¿evKill¿);
PsTerminateSystemThread(STATUS_SUCCESS);
}
```

For all intents and purposes, a thread is the most basic unit of an operating system, and they are the unit for which CPU time is allocated during a processs execution. A thread is a high-level abstraction that provides the ability to execute more than one sequence for a given executing program. Threads allow for parallel processing and concurrency. FreeBSD and Windows operating systems implement both user and kernel modes; every single thread maintains its own stack and counter when running in kernel mode. Switching to user mode resets all of these values that a thread maintains. For FreeBSD, threading follows the POSIX standard whereas Windows creates its own implementation standards. [1] [2]. Both operating systems allow for threads that execute exclusively in kernel space, inaccessible to users in user mode. In comparison to Linux, threading between FreeBSD and Linux are extremely similar; almost identical, in fact. Windows, however, has some key differences in the way threads are implemented. FreeBSD and Linux threads are capable of sharing resources between other executing processes and threads. Windows systems are required to have at least one thread in execution at a given

time to be processed by the I/O scheduler. A program in Windows absolutely must have at least one thread to execute [1].

IV. Compare and Contrast OS CPU Scheduling

The I/O scheduler is an extremely vital subsystem for any operating system. The primary function of an I/O scheduler is to manage the amount of time a given thread or resource is allowed to execute on the system CPU at a given time. It should be noted that there are a number of different I/O scheduling algorithm implementations that exist. One scheduler that FreeBSD utilizes is the ULE scheduler which retrieves threads and processes by their priority and schedules their execution on the CPU by that given priority [6]. Windows, on the other hand, implements its own I/O scheduler which determines a given tasks priority, ranks it against other tasks in the queue, and reassigns priority based on other tasks rankings. In comparison to Linux, there are a large number of differences between the I/O schedulers of Windows and FreeBSD (probably too many to cover here). The primary difference is that Linuxs default I/O scheduler, CFS (completely fair scheduler), focuses on a fair sharing of resources between threads and processes [3]. FreeBSD schedules by an assigned priority, and Windows schedules based on arbitrary rankings of priority between processes and threads [1] [6].

V. Conclusion

Although Windows, FreeBSD, and Linux share a lot in common in terms of their implementation of key functionalities, there are also a lot of differences in how these functionalities operate, as well as the nomenclature that describes how these functionalities are expected and/or designed to behave. The small, nuanced differences may seem insignificant, but often they have large implications when considering which to use in terms of solving a given problem. In terms of processes, all three operating systems maintain more similarities than differences. Same with threads. However, CPU scheduling proved to be where these three operating systems differ most.

REFERENCES

[1] D. Solomon and M. Russinovich, *Microsoft Windows Internals, Part 1 6th Edition*. Microsoft Press, 2012.

[2] "Processes and daemons," https://www.freebsd.org/doc/handbook/basics-processes.html, freebsd.org, [Online; accessed 22-October-2017].

[3] R. Love, *Linux Kernel Development*. Pearson Education International, 2010.

[4] "System threads," https://www-user.tu-chemnitz.de/~heha/oney_wdm/ch09e.htm, Chemnitz University of Technology, [Online; accessed 22-October-2017].

[5] "Pscreatesystemthread routine," https://msdn.microsoft.com/en-us/library/windows/hardware/ff559932(v=vs.85).aspx, Microsoft, [Online; accessed 22-October-2017].

[6] "I/o scheduling in freebsds cam subsystem," https://people.freebsd.org/~imp/bsdcan2015/iosched-v3.pdf, Netflix.com, [Online; accessed 22-October-2017].