OPERATING SYSTEMS II
CS444 FALL 2017

OCTOBER 25, 2017

OPERATING SYSTEM FEATURE COMPARISON

PROCESSES, THREADS, AND CPU 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. Compare and Contrast OS Processes

What is a process? Although this might seem like a relatively trivial question to answer, its not really. Put simply, a process is a program in execution. However, this is not really a sufficient answer, and it requires a bit of technical knowledge to answer the question more completely. 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 CPU runtime values. In both Windows and FreeBSD, a process is identified via a PID, or a process identifier, which is just a unique integer value. 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. 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 a process is very similar across the board; it is an instance of a program being executed. Processes can communicate with other processes, create child processes, create threads, etc. There are, however, some very interesting differences between Windows and Linux processes. For example, in Windows, when you stop a parent process, all of the parents child processes will continue their execution to completion [**?**]. In Linux, if a parent process is stopped, all child processes are also stopped [**?**]. Why this is the case, I really have no idea. Ive searched and searched for an answer to no avail. Obviously, theres some benefit to either of these approaches, and in Unix-like systems, there are ways in which you can obtain behavior similar to that of Windows [**?**]. 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]. Another difference that stands out between Windows and Linux is 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].

II. 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 process can create multiple threads. A thread is a high-level abstraction that provides the ability to execute more than one sequence of code for a given executing program. Threads allow for parallel processing and concurrency. FreeBSD and Windows operating systems implement both user and kernel modes; where 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, and most Unix-like operating systems, threading follows the POSIX standard (pthreads), whereas Windows follows its own implementation standards (Win32 threads) [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);
}
```

Both operating systems allow for threads that execute exclusively in kernel space, inaccessible to users in user mode. 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 process can create multiple threads. A thread is a high-level abstraction that provides the ability to execute more than one sequence of code for a given executing program. Threads allow for parallel processing and concurrency. FreeBSD and Windows operating systems implement both user and kernel modes; where 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, and most Unix-like operating systems, threading follows the POSIX standard (pthreads), whereas Windows follows its own implementation standards (Win32 threads) [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 in FreeBSD and Linux are the same. When comparing Linux pthreads to Windows Win32 threads, there are some inherent differences, however, they are extremely similar in the way that programmers can use them; almost identical, in fact. In terms of the API and thread interface, pthreads and Win32 threads are almost exactly the same. There are some key differences in the way threads are implemented between the kernels, though. And, furthermore, what goes on under the hood, so to speak, is very different between these kernels. 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 process in Windows absolutely must have at least one thread in order to execute [1]. Why this is the case is hard to say. It is my opinion that, generally speaking, the Windows kernel is more strict in terms of what it allows a programmer to do with the systems resources. In other words, it seems that Linux provides programmers with greater power and greater responsibility. To me, this makes sense because Linux kernels are open-source, and Windows kernels are not. This seems like a silly reason, but Im not exactly able to go dig around in the Windows kernel and modify it, etc. In Linux, I am free to do so. I think a lot of the differences between Linux and Windows threading comes down to freedom and the ability to use a given systems resources freely.

III. 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]. Again, I find it difficult to speculate why this is the case. If time is the concern, then I believe that Windows would have the faster scheduling algorithm since it simply always grabs the highest priority task first. If fairness is the concern, then Linux CFS is the most fair in terms of sharing resources. In Windows, I cannot change the I/O scheduler to behave the way I want it to. In Linux, I can. Why this matters, I dont know. I think most of it, again, comes down to freedom, and the idea behind most concepts in Windows seems to be to limit that freedom. In this case, Windows decides what tasks are more important and prioritizes giving them CPU time. That way, a programmer cannot force his program to be the highest priority. In Linux, this is quite the opposite. Freedom.

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].