Differences and similarities in Process and Thread Management and Scheduling between the Linux, Windows, and FreeBSD Operating Systems

Alex Hoffer

CS 444
April 2017
Spring 2017

Contents

1 Windows

1.1 Processes

1.1.1 Similarities to Linux

Process Creation When a new process is created in either Windows or Linux, the new process is also given a newly created thread. Processes in both Windows and Linux can have 1 or more threads assigned to them, though multithreaded programs in Windows are easier to program (the explanation for this can be found in the *Differences from Linux* section of this work). Additionally, newly created processes in Windows are allocated similar memory resources to that of Linux. Each process on both systems has specially allocated memory segments corresponding to the text segment, initialized data segment, uninitialized data segment, stack, and heap of the running program that are unique for that specific process.

Process Identification Both offer the ability to get the currently running process's ID, with *GetCurrentProcessId()* in Windows and *getpid()* in Linux. Windows and Linux are also both subject to *race conditions*, or the tendency for a program to rely on one process to complete a sequence of actions before another process completes its own sequence, since both of their processes are subject to nondeterministic scheduling (more on this in the *CPU Scheduling* section).

Process Termination/Exiting A process can terminate in Windows and Linux in either a normal or abnormal fashion. Termination in normal or abnormal fashion for both of these systems yields an exit value that other processes can access. When a process dies in either Windows or Linux, if it has a child that has not properly been waited for and no terminating signal has been sent by the user, then the child process will continue to run. This orphaned child process is called a *zombie*.

1.1.2 Differences from Linux

Process Creation To create a new process in Windows, we must execute a *CreateProcess* call. To do this same function in Linux, we traditionally use the *fork* system call. The differences between these calls is indicative of how the two operating systems handle the parent and child relationship inherent in a process hierarchy. Specifically, *CreateProcess* takes 10 arguments, including (but not limited to) desired security attributes, creation flags, and environment. The *CreateProcess* function signature is listed below:

```
BOOL WINAPI CreateProcess(
  _In_opt_      LPCTSTR               lpApplicationName,
  _Inout_opt_   LPTSTR                lpCommandLine,
  _In_opt_      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_          BOOL                  bInheritHandles,
  _In_          DWORD                 dwCreationFlags,
  _In_opt_      LPVOID                lpEnvironment,
  _In_opt_      LPCTSTR               lpCurrentDirectory,
  _In_          LPSTARTUPINFO         lpStartupInfo,
  _Out_         LPPROCESS_INFORMATION lpProcessInformation
);
```

A Linux developer, on the other hand, can spawn a child process by simply using *fork()*.
Another important difference in process creation between these two systems is that the *CreateProcess* call returns a *bool* that is set to true if the process and subsequent thread are successful in creation,

and false otherwise. Alternatively, *fork()* returns an integer value that, if the process is successfully created, corresponds to the process's ID. Thus, we can see that with Windows, the act of specifying a process's attributes are quite important, while with Linux, the programmer's awareness of the process hierarchy is emphasized, since they can easily store and access process IDs in order to identify which process is running at a given time. Meanwhile, Windows does not maintain the relationships between parent and child processes [2]. [2] goes on to claim that the reason behind this lack of interest in parent and child process relationships in Windows is because the act of forking in Linux is, in practice, dreadful when creating multithreaded programs. This is because the *fork* system call makes it so that the newly created child process is an exact replica of the parent process, retaining precise copies of the parent's threads and synchronization objects, making management between multiprocessors a nightmare.

Process Identification In Windows, processes can be identified by both handles AND process IDs, while in Linux, we can only identify a process by its ID, which we can access using the *getpid()* system call. With Windows, we can get a process handle by using the *OpenProcess* function, and get a process ID by using the *GetCurrentProcessId*. The *OpenProcess* function allows for you to enumerate that handle's access rights and contains the ability to set whether or not the process is inheritable. I think Windows includes this handle as well as the process ID in a strategy to make security a priority in the act of process management, since you can specify what a process can access through its handle. In this sense, the identity of a Windows process is a much more complex entity, since it can include a large quantity of information. In fact, both the ID and the handle of a process is stored in a *PROCESS_INFORMATION* structure. Access control information of processes in Linux can be found in a process's *process credentials*.
Of note is that Windows does not provide a function to discover a process's parent's ID, while Linux processes maintain records of their parents' process IDs [3]. Again, this is likely because of Windows' different methods of handling process hierarchies.

Process Termination/Exiting In Linux, normal process termination occurs with the *_exit()* system call [3], while in Windows the function responsible for normal process termination is *ExitProcess*. The act of process termination is reflected upon system resources differently across these two operating systems. When a process terminates either normally or abnormally in Linux, synchronization resources such as semaphores or memory locks are closed and unlocked, respectively. Like [2] claims, with Windows, it is essential that resources shared across processes like semaphores and memory locks are deliberately freed by the programmer in the act of termination. That is, Windows offers less garbage collection in process termination when it comes to synchronization resources than Linux does.

1.2 Threads

1.2.1 Similarities to Linux

Thread creation Creating a thread complete with stack size specification and security attributes can be done in one complex function call in Windows, but Linux can specify the exact same things for a thread as Windows can, just through the use of four methods, namely:

- pthread_create
- pthread_attr_init
- pthread_attr_setstacksize
- pthread_attr_destroy

[2] claims POSIX Pthreads, which are used prominently in the Linux operating system, provides similar features to Windows threads, but Windows allows for a "broader collection of functions". In fact, Windows threads are similar enough with POSIX Pthreads in how threads are managed that there are open source Pthreads libraries available for Windows system developers.

Thread synchronization Threads in both Windows and Linux share resources with other threads. Additionally, both Windows and Linux use semaphores and mutexes to restrict access to shared resources between threads.

Thread termination/exiting Normal thread exiting can be done in a single function in both Windows and Linux. The function for this in Windows is *ThreadExit*, and in Linux it is *pthread_exit*.

1.2.2 Differences from Linux

Thread creation In Linux, threads are not built-in and must be implemented using the POSIX Pthreads library. Windows, on the other hand, has built-in threads which serve as the kernel's basic unit of scheduling. Several different parameters present in thread creation function for Windows than Linux, just like how a Windows process requires more parameters in its creation. This is because Windows intends for you to specify more about process and thread functionality during the genesis of processes and threads than Linux does. In Windows, threads are crucially important. The thread creation function signature for Windows is as follows:

```
HANDLE WINAPI CreateThread(
  _In_opt_   LPSECURITY_ATTRIBUTES     lpThreadAttributes,
  _In_       SIZE_T                    dwStackSize,
  _In_       LPTHREAD_START_ROUTINE    lpStartAddress,
  _In_opt_   LPVOID                    lpParameter,
  _In_       DWORD                     dwCreationFlags,
  _Out_opt_  LPDWORD                   lpThreadId
);
```

Meanwhile, to create a thread in Linux, we simply use the *pthread_create* function, which takes only four parameters. Again, we can see through this example that Windows emphasizes the programmer's specification of attributes such as access control in the creation of threads and processes.

Thread synchronization An important distinction in Windows vs. Linux thread management is the inclusion of *events* in Windows as a method of thread synchronization. Events are objects that are used to notify waiting threads when an event, like I/O operations on files, are completed [6]. Windows also utilizes *critical sections* while Linux implements *conditional variables*, like *pthread_cond_wait*.

1.3 CPU Scheduling

1.3.1 Similarities to Linux

Timeslice properties Both Windows and Linux use timeslices, or quantums, of about 10 milliseconds to in the hundreds of milliseconds (though Windows has traditionally lower timeslices, likely in an effort to favor higher priority threads, which is discussed in a later section). Both operating systems also institute *reentrant* timeslices, which means that if a thread is interrupted during its execution, then it is returned with what remains of its timeslice after interruption, rather than a fresh timeslice.

Base priorities Both operating systems assign a *base priority* to a process, that is, a priority calculated according to a scheduling algorithm. Although Windows and Linux both favor different priorities and utilize different scheduling algorithms, the idea of attaching an initial priority subject to change to a process is universal across them both.

1.3.2 Differences from Linux

Scheduling Classes In Windows, there are two scheduling classes, *real time* and *dynamic*. Windows favors threads that have a higher priority value (e.g. 32 rather than 0) attached to it [5]. Linux, on the other hand, uses three scheduling classes, *normal*, *Fixed Round Robin*, and *Fixed FIFO*, though the default scheduling class is the *Fixed Round Robin*, or the *Completely Fair Scheduler (CFS)* [3]. Linux favors threads that have a lower priority value attached to it in an effort to institute a level of class equality in their scheduling methodology.

2 FreeBSD

2.1 Processes

2.1.1 Similarities to Linux

UNIX-like operating systems Since FreeBSD is a UNIX-like operating system, FreeBSD and Linux are very similar in how they handle processes and threads. First and foremost, like Linux and Windows, FreeBSD processes contain information that an operating system needs to have in order to handle program execution, such as process ID, registers, stack, heap, etc.

Preservation of the Parent-Child Relationship Like Linux, FreeBSD uses a *fork()* system call to create a new child process. This call returns the pid of the new process, and that new process can retain the ID of its parent using a function call [4]. In this way, FreeBSD and Linux share a common interest in preserving an easy and convenient way for a developer to understand the process hierarchy of a program's execution. Thus, FreeBSD does not contain the *handles* or complex forking procedure that Windows utilizes. Like both Windows and Linux, a programmer can get the running process's ID with a *getpid()* function call. Unlike Windows but like Linux, a child process is the exact duplicate of the context of its parent [4], which, as stated previously, is a concern in the context of multiprocessing.

2.1.2 Differences from Linux

No considerable differences There are no significant differences in the way processes themselves are created, monitored, or destroyed between FreeBSD and Linux. There are, however, differences in how processes are chosen for CPU time (outlined in the *CPU Scheduling* section of this paper).

2.2 Threads

2.2.1 Similarities to Linux

Like Linux, FreeBSD is POSIX-compliant, meaning threads can be managed using the Pthreads library. This means that the following sample of two fundamental thread functions are the same between FreeBSD and Linux:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void*(*start_routine)(void *), void *arg)

int pthread_join(pthread_t thread, void **value_ptr)
```

It follows that since FreeBSD utilizes the Pthreads library, nearly all of its thread management is identical to that of Linux: there are not many major differences between thread creation, destruction, and synchronization between the two operating systems.

2.2.2 Differences from Linux

No considerable differences Such as with processes, the emergence of both FreeBSD and Linux from the UNIX operating system led to few discernible differences between the two in managing threads.

2.3 CPU Scheduling

2.3.1 Similarities to Linux

The most profound differences between the two UNIX-like operating systems can be found in the way they allocate CPU time to different processes.

Nice values and similar priority model Like Linux, FreeBSD sets large *nice values* (numbers that correspond to job priority) such as the range between 0 to 20 as being of low priority, while small nice values such as the range between -20 - 0 are set for higher priority for execution [1].

Getting, setting priority Both Linux and FreeBSD programmers can include the *sys.resource.h* file in order to use functions that can get the nice value of a specified process or set the nice value of a specified process. These functions are included below, courtesy of [3].

```
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

2.3.2 Differences from Linux

Default schedulers The default scheduler for the FreeBSD is the ULE, while the default scheduler for Linux is the CFS. The ULE is built on low-level and high-level scheduling subsystems. The low-level one is frequently running and quickly grabs the next thread to run based off what the highest prioritied task is in the system's *run queues* [4]. Meanwhile, CFS does not use run queues to choose the next process to run, instead this scheduling strategy computes the next process based on round-robin time-sharing, as described in the section on Linux vs. Windows for scheduling.

References

[1] Nathan Boeger and Mana Tominaga. Freebsd system programming, 2001.

[2] Johnson M. Hart. *Windows System Programming (3rd Edition)*. Addison-Wesley Professional, 2004.

[3] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[4] Marshall Kirk. McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The design and implementation of the FreeBSD operating system*. Addison-Wesley/Pearson, 2015.

[5] Mark E. Russinovich, Alex Ionescu, and David A. Solomon. *Windows internals, Part 2*. Microsoft, 2012.

[6] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows internals*. Microsoft Press, 2012.