Writing Topic #1: Processes

Alex Hoffer

CS 444
April 2017
Spring 2017

Contents

1 Windows

[4] [5] [3] [2] [1] The Windows operating system has many similarities and differences to Linux, and since it is proprietary software, it is not prone to the robust open source development that Linux is famous for. Nevertheless, Windows systems programming is well documented, and with the help of several resources we can get a good picture of how Windows manages processes, threads, and CPU scheduling time.

1.1 Processes

Processes are the fundamental building blocks of any operating system, and Windows' processes contain a lot of nuances that make process management seem, to me, more complex than in Linux. The following subsections denote where in Windows processes I see similarities to Linux and where I think there are differences between the two. Through this course of explanation, we will get a good idea of how Windows manages processes and reasons why Windows manages processes like they do.

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.

Process Identification Both offer the ability to get the currently running process' 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 a level of nondeterminism (more on this in the *CPU Scheduling* section.

Process Termination 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, now known as 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 a *fork*. 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
);
```

The *fork* system call, on the other hand, does not take any arguments. 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. *Fork()*, on the other hand, 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. On the other hand, Windows does not maintain the relationships between parent and child processes [1]. [1] 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 processese in Linux can be found in a process's *process credentials*. Additionally, Windows does not provide a function to discover a process's parent's ID, while Linux processes maintain records of their parents' process IDs [2]. Again, this is likely because of Windows' different methods of handling process hierarchies.

Process Termination In Linux, normal process termination occurs with the *_exit()* system call [2], 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 *har04* 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

(1)Similarities in what resources each thread in a process shares, such as: (2) [1] 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". SUCH AS: (3) Similar enough with POSIX Pthreads in how threads are managed that there are open source Pthreads libraries available for Windows system developers.

1.2.2 Differences from Linux

1.3 CPU Scheduling

1.3.1 Similarities to Linux

1.3.2 Differences from Linux

2 FreeBSD

2.1 Processes

2.1.1 Similarities to Linux

2.1.2 Differences from Linux

2.2 Threads

2.2.1 Similarities to Linux

2.2.2 Differences from Linux

2.3 CPU Scheduling

2.3.1 Similarities to Linux

2.3.2 Differences from Linux

References

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

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

[3] 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.

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

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