A Brief Comparison of the POSIX and Windows Based API

Austin Dubina

December 6, 2012

Contents

Chapter 1

Introduction

Most of the time in Operating Systems I was spent exploring the POSIX API (application programming interface) in C as an introduction to operating systems programming. However POSIX is not the only API that is used for the concepts learned throughout this course. These concepts include but are not limited to file system interaction, file I/O, signals, process creation and control, pipes, threads, synchronization, IPC, shared memory, sockets, Multiplexing, and much more! Windows being one of the most dominate and prevalent operating systems that exists today, makes learning its proprietary API a must for any serious software developer or computer scientist.

In this paper I will exam four concepts covered in class; the file system interaction, file I/O, process creation, and pipes. The following concepts learned from a POSIX based API will be compared, contrasted, and extended to the Windows API.

This report will detail the differences and similarities of few fundamental concepts of a system programing interface such as file system interaction, file I/O, processes, and pipes. The similarities and differences between the POSIX based API and Windows Based API will be compared and contrasted against one another using examples learned from lecture, the Linux Programming Interface and Windows System Programming.

It is true, while the end result any program performing interoperable tasks from a standard library such as C / C++ may be agnostic over which programming interface was used complete those tasks. The system calls, libraries, and functions used to perform more complex system wide tasks such as file I/O, multi-threaded execution and memory allocation is inherently different between a POSIX based system and a Windows based system.

Chapter 2

The File System Interface

As an operating system as a whole, it is important to note that unlike UNIX, or Linux, Windows does not conform to the X/Open standard or any other open industry standards formulated by standards bodies or industry consortia (Hart, 2010). This deviation makes the Windows system API, significantly more different from any POSIX based API. While many will debate whether this deviation is good for the computer industry as a whole, the proprietary Windows system API is still widely used in industry today (along with POSIX based APIs), and cannot be simply ignored.

Therefore, before we start diving into details about the underlying differences between the two programming interfaces, The Windows System Programming book outlines a few basic Windows principles that I think are important to understanding the two APIs, especially for developers (like myself) who coming from a POSIX based background.

As I read through Windows System Programming, it became apparent that Windows has some fundamental characteristics that separates itself from POSIX, this became very apparent over time. The first and most apparent was the Windows definition of an object handle (more accurately file handle), which is rather analogous to UNIX's file descriptor.

According to the MSDN an object is a data structure that represents a system resources, such as a file, thread, or graphic image. An application cannot directly access object data or the system resource that an object represents. Instead, an application must obtain an object handle, which it can use to examine or modify the system resource (About Handles and Objects, 2012). Similarly, a file descriptor in POSIX is an index for an entry in a kernel-resident array data structure. This data structure is called the file descriptor table, and each process in UNIX has its own table. When a process passes its file descriptor to the kernel, typically through a system call, the kernel will access the file on behalf of the process. Each process will inherit three open file descriptors which corresponds to the I/O streams standard in, out, and error. In fact, Microsofts C libraries also provide compatibility functions which wrap these native handles to support the POSIX-like convention of integer file descriptors (Jerry, 2012). In addition to files, objects in Windows include processes, threads, pipes, memory mapping, events, and much more. Each object in Windows also has an associated security attribute.

Chapter 3

File I/O

Probably the most simplistic and universal form of communication between programs is file reading and writing. While standard libraries from C / C++ for example provide portable communication between programs on both platforms, this communication is very limited. MSDN discusses an extensive list of useful functions for handling file I/O, however for intents and purposes of this report we will only discuss functions that relate to the so-called universal I/O model described in The Linux Programming Interface. These are system calls that open and close a file, and read and write data (Kerrisk, 2010).

```
HANDLE WINAPI CreateFile(
  _In_      LPCTSTR lpFileName,
  _In_      DWORD dwDesiredAccess,
  _In_      DWORD dwShareMode,
  _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  _In_      DWORD dwCreationDisposition,
  _In_      DWORD dwFlagsAndAttributes,
  _In_opt_  HANDLE hTemplateFile
);
```

The *CreateFile()* function in windows creates or opens an I/O device which is defined by MSDN as a file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resources, mailslot (discussed in section IV), and pipes. This function returns an object handle that can be used to access the file or device for various types of I/O. This is somewhat similar to the *open()* system call in UNIX which either opens an existing file or creates and opens a new file. This function returns an integer known as the previously mentioned file descriptor. File descriptors are used to refer to all types of open files including pipes, Sockets, FIFOs, terminals, devices, and regular files.

For a file to be opened in either Windows or UNIX, the user must provide some basic parameters. In both POSIX and Windows based APIs the developer must provide at least a pathname to target to the desired file, access mode flags, and permissions (if the file is being created for the first time), these parameters are described in detail below for both POSIX and Windows.

The *IpFileNAme* parameter is a pointer to a null-terminated string that specifies the name of the object, such as a file or the name of a device to create/open, for example a COM port. This is analogous to the *pathname* parameter in UNIX which is also a pointer to a null-terminated string that specifies the location of the desired file.

The *dwDesiredAccess* parameter is the Windows equivalent of the UNIX *flags* parameter which can be combined with a bit-wise 'or' to requests either read, write, and many more privileges from the OS.

The mode parameter in UNIX is only used when creating new file and is responsible for setting the permissions. This is somewhat similar to the security attribute, which is also used to describe the objects owner and determine whether users are allowed or denied various rights to the particular device.

While, many more attributes exist for both APIs to further control how a file is read, written, and accessed. The above examples provided a basic insight to the similarities and differences between POSIX and Windows when creating a file.

Closing a File

It is generally good practice to close a given handle or file descriptor when it is no longer needed by the program. This typically makes code much cleaner, easier to read, and helps avoid issues down the road especially when dealing with pipes and forked related processes in a POSIX based environment. Conveniently it just so happens that both UNIX and Windows have system calls to free consumable resources such as handles and file descriptors.

Windows has a single all-purpose *CloseHandle()* function to close and invalidate kernel handles and to release system resources. This function takes a handle object as an argument and returns either true or false if the handle was successfully closed.

```
BOOL WINAPI CloseHandle(
  _In_  HANDLE hObject
);
```

Likewise, the *close()* function in Unix takes a given file descriptor as its argument and returns -1 on error.

Reading and Writing to a File

The read and write functions in Windows are not only similar to one another but also complement POSIXs implementation of file reading and writing as well. Whether reading/writing in Windows or UNIX, the user must specify a handle/file descriptor, buffer, and number of bytes to read or write. The only apparent difference between the two implementations is the return value. Windows *WriteFile()* and *ReadFile()* function is of type boolean which either returns true or false. Unlike UNIX which returns the number of bytes read or written and negative one on error, Windows represents this as parameters *lpNumberOfBytesWritten* and *lpNumberOfBytesRead*. Also, Windows has an additional parameter *lpOverlapped* which points to a structure if the *hFile* parameter was opened with FILE_FLAG_OVERLAPPED, otherwise this can be NULL. This parameter is basically equivalent to UNIX's *lseek()*, where you can specify a byte offset at which to start writing to the file or device.

Chapter 4

Creating Processes and Managing Them

In UNIX the *fork()* system call allows one process, the parent, to create a new process, the child. This is done by making the new child process and (almost) exact duplicate of the parent: the child obtains copies of the parents stack, data, heap, and text segments (Kerrisk, 2010). The key to understanding a fork is to realize that as soon as the function returns, two processes now exist and start executing from same line of code. Since the *fork()* system call takes no arguments, it is probably one of the simplest ways of dividing up a task for any given interface.

Windows has no equivalent to the UNIX *fork()* function because fork is difficult to emulate exactly in Windows. The Windows System Programming states that while this may seem like a limitation, forking by itself is not really appropriate in any multithreaded application because of the numerous issues that arise when creating an exact replica of a multi threaded program with exact copies of all threads and synchronization objects, especially on a multiprocessor computer.

Instead, Windows can emulate a *fork()* by combining the functionality of *fork()* and *exec()* into a single operation, or as The Linux Programming Interface refers to it as a spawn. This fundamental method for creating and managing a process in Windows is done by using the *CreateProcess()* function, which (as you might have guessed) creates a process with a single thread. As mentioned earlier Windows does not maintain a parent-child relationship among processes. Therefore children will continue to run even after the parent process has been terminated. Also, *CreateProcess* function has 10 different parameters (compared to *fork()* which has 0) that can be used to utilize the full potential of the function. However for the intents and purposes of this paper we are only going discuss a few the required parameters and functions required to successfully make a basic process in Windows.

```
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
);
```

Parameters

In Windows, if a child process requires access to an object referenced by a handle in the parent; it must be inheritable. The *bInheritHandles* flag on the *CreateProcess()* call determines whether the child process will inherit copies of the inheritable handles of open files, processes, and so forth. This is frequently done in Windows to share the standard input and output handles to children. It is also worth noting, that because shared handles between children are not unique, Windows adds an ID to each handle so that they can be individually identified. While Windows processes are identified by both handles and process IDs, UNIX has no process handles and instead uses the ID return by fork to manage each process.

The *dwCreationFlags* is another required parameter in the *CreateProcess()* function, which combines several flags to control the priority of the new processs threads. The NORMAL_PRIORITY_CLASS can be used as a default priority execution.

The final two required parameters are *lpStartupInfo* and *lpProcessInformation*. *lpStartupInfo* is a pointer to a STARTUPINFO or STARTUPINFOEX structure. The *lpStartupInfo* specifies the main window appearance and standard device handles for the new process. *lpProcessInformation* specifies the structure for containing the returned process, thread, handles, and identification.

Exiting a Process

Just like in UNIX after a process has finished its work, the process (for windows its a thread running in the process) must exist. In Windows, this is done using the *ExitProcess()* which returns nothing. Rather, the calling process and all its threads terminate (Hart, 2010). if the developer wants to check the exit status of the process he/she can use the *GetExitCodeProcess()* to determine the exit code. It is important to note that before a process exits a check should be made to make sure all the resources that were shared with that process have been freed.

UNIX processes have a process ID, or pid, comparable to the Windows process ID. *getpid()* is similar to the *GetCurrentProcessID()* function, but there are no Windows equivalents to *getppid()* (get parent or calling process identification) because Windows does not have process parents or UNIX-like groups. Conversely, UNIX does not have process handles, so it has no functions comparable to *GetCurrentProcess()* or *OpenProcess()* functions (Hart, 2010).

Chapter 5

Interprocess Communication Though Pipes

Pipes are one of the oldest methods of interprocess communication (IPC) on the UNIX system, pipes fulfill a recurring need to have two processes talk to one another by having the output of one program be the direct input of another program. Windows is no different, the two primary mechanisms for interprocess communication in windows is the anonymous pipe and the named pipe both of which are analogous to UNIXs unnamed and named pipe (FIFO) which will be further discussed during this section.

Anonymouse/Unamed Pipes

Like UNIX Windows anonymous pipes are half-duplexed, meaning they allow communication only one way, and are byte streamed. In Windows, each pipe has two handles: a read and a write handle and is called by using the *CreatePipe()* call. This is analogous to using the *pipe()* call in UNIX which returns a read and write file descriptor 0 and 1.

```
BOOL WINAPI CreatePipe(
  _Out_     PHANDLE hReadPipe,
  _Out_     PHANDLE hWritePipe,
  _In_opt_  LPSECURITY_ATTRIBUTES lpPipeAttributes,
  _In_      DWORD nSize
);
```

In order to use the pipe, there obviously has to be another process to read and write from. The trick to using pipes in Windows is to pass the desired handle presumably created by the parent to an associated child process. This is accomplished by setting the child procedures input handle in the start-up structure to *phRead as previously mentioned. Reading a pipe read handle will block if the pipe is empty. Otherwise, the read will accept as many bytes as are in the pipe, up to the number specified. Write operations to a full pipe will also block. This also true for UNIX.

Named Pipes

Both UNIX and Windows support named pipes. Both of which are much more powerful than unnamed pipe as this allows for communication between non-related processes such as client-server communication over networks. However, unlike UNIX named pipes (FIFOs) which

are still unidirectional and byte streamed (SUSv3 explicitly notes that opening a FIFO with the O_RDWR flag is unspecified and should be avoided for portability reasons, although it can be done), Windows named pipes are duplex and message-oriented and can have more than one handle on any given pipe. This makes communication over networks very easy when using named pipes on Windows. The *CreateNamedPipe()* system call creates the first instance of a named pipe and returns a handle. According to the Windows System Programming, the creating process is regarded as the server. Client processes possibly on another system, open the pipe with CreateFile as mentioned in section I. Below is the syntax for creating a named pipe on Windows

```
HANDLE WINAPI CreateNamedPipe(
  _In_      LPCTSTR lpName,
  _In_      DWORD dwOpenMode,
  _In_      DWORD dwPipeMode,
  _In_      DWORD nMaxInstances,
  _In_      DWORD nOutBufferSize,
  _In_      DWORD nInBufferSize,
  _In_      DWORD nDefaultTimeOut,
  _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

Due to the shear number of required parameters used to create a named pipe on Windows I will have to revert the reader to the MSDN for an in depth description of each argument. It is suffice to say however when using UNIX named pipes, only the pathname and access mode is required to successfully create a named pipe. Below is the syntax for creating a FIFO on UNIX.

```
int mkfifo(const char *pathname, mode_t mode);
```

Mailslots

Another interesting IPC that is not directly related to pipes and does not exclusively exist in UNIX is the Windows Mailslot. A mailslot is a lot like a named pipe in that it also has a name that unrelated processes can use for communication. However, unlike most other IPC mechanisms, mailslots are neither connection-oriented nor bidirectional; clients simply send messages to a server process. Mailslot clients never read from a mailslot: only servers can (Dowd, 2007).

Chapter 6

Conclusion

This paper examined some fundamental differences between a POSIX and Windows based API in regards to file system interaction, file I/O, process creation, and pipes. The first and most prominent distinction between the two operating systems was the Windows object handle vs the UNIX file descriptor. Both the file descriptor and the object handle, point or index a systems resource data structure table; such as a file, thread, or graphic image. Each system resource has its own unique table and because a program cannot directly access these resources directly they must ask the kernel to access the given system resource on behalf of the process. While the file descriptor is an integer index reference rather than a pointer to a memory location; file descriptors is less opaque and making the Windows handle (at least to me) a more abstract concept. And because of UNIXs universal I/O model all system calls for performing I/O refer to open files using a file descriptor. This makes learning new system calls very easy and intuitive in a POSIX based system like UNIX.

Windows seemed to offer as many if not more file I/O system calls as UNIX that are equivalent in functionality and purpose. However, the parameters used in opening, closing, reading, and writing to and from files, although generally verbose, can be very complex and confusing to understand in Windows, at least at first. Probably one of the biggest file I/O differences between UNIX and Windows are parameters required to successfully make a system call; specifically the security attribute. While the security attribute can do many things in Windows, it seemed mostly used in setting the UNIX equivalent of file permissions. Windows does not have the same type of user group setup as UNIX therefor must be treated differently.

Windows simply does not have a system call for forking a process. Instead Windows can emulate a fork by combining the functionality of *fork()* and *exec()* into a single operation, or as The Linux Programming Interface refers to it as a spawn. This fundamental method for creating and managing a process in Window is done by using the *CreateProcess()* function which executes a program with specified parameters pointed to by *lpApplicationName*. In windows, if a child process (which does not have the same parent-child relationship as UNIX) requires access to an object referenced by a handle in a the parent; it must be inheritable. the *bInheritHandles* flag on the *CreateProcess()* call determines whether the child process will inherit copies of the inheritable handles of open files, processes, and so forth. This is frequently done in Windows to share the standard input and output handles to children.

Passing a handles to other processes is the key to success when using pipes in Windows. Unlike

UNIX which simply passes a copy of its file descriptors (via fork) to related processes Windows sets up the childs procedures input handle in the start-up structure. However, because forking will duplicate unnecessary and even unwanted duplicates of file descriptors, it is up to the developer to manage these descriptors and close any unused ones.

Having only read about the Windows API and not actually done any coding in it, I would say the two APIs are inherently different in terms of the systems calls and parameters but are very similar in purpose and functionality. While some things like Mailslots do not natively exist in UNIX, or forking vice versa. Each API has methods for accomplishing the end goal. This is unfortunate for a developers aiming to create truly portable and universal applications. However, having multiple tools, each of which have their strengths and weaknesses, give us as developers the freedom and power to choose platforms and standards which suit our specific needs.

Final Thoughts

Reflecting back on this course as a whole and recalling all the concepts that have been covered and absorbed over the last few months was no trifling thing; in fact this has been one of the most conceptually challenging classes I have taken. However, I am a competitively more competent programmer / student because of this class. This has been one of the best courses I have ever taken as a professional student here at Oregon State University. I hope you enjoyed reading this paper, now go and get some rest.

Because remember

> "If you haven't got your health, then you haven't got anything." - The Princess Bride

Bibliography

[1] Dowd, M., McDonald, J., & Schuh, J. (2007). *The art of software security assessment: identifying and preventing software vulnerabilities*. Indianapolis, Ind.: Addison-Wesley.

[2] Hart, J (2010). *Windows System Programming (4th ed.)*. Boston, MA: Pearson Education.

[3] About Handles and Objects (Windows). (2012). Retrieved December 3, 2012, from http://msdn.microsoft.com/en-us/library/windows/desktop/ms724176(v=vs.85).aspx

[4] Jerry. (2012, September 29). File descriptor vs. file handle in Unix/Linux. *UNIX/LINUX System and Database Administration Tips*. Retrieved Dec 1, 2012, from http://satips.blogspot.com/2012/09/file-descriptior-vs-file-handle-in.html

[5] Kerrisk, M (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA: No Starch Press.