Name: Pavin Disatapundhu
Email: disatapp@onid.orst.edu
Class: CS344-400
Assignment: FINAL

1 System API's Introduction

In this write up I will be comparing and contrasting Windows API with the Posix API in an easy to understand manor, while still preserving the depth the the individual topics (of course). This report will be focusing following topics: file I/O, pipes, mutual exclusions, and parallel process. These topics are highly important, and learning the proper way to utilize them essential for any developer. This report will assume that you are familiar with the POSIX systems calls.

2 File I/O

As we know, file I/O are a crucial part of system. It's the most basic and simple form of communication between programs. It provides a way files to transfer data and most of these functions are accessible within the C or C++ standard library. Although, these functions provide a very portable method of transferring data, their functionalities can be very limited. These limitations is why the POSIX API exist. When talking about POSIX I/O functions four common system calls come to mind, these functions are the open, close, read and write operation. This chapter will compare and contrast the similarities for the file I/O of the Window API and POSIX.

2.1 Open for business

The *open()* system call was used often in this course to create and/or open files. The call's parameter can be used to change the mode, and set flags. The problem with this function is that it's a Unix exclusive call, therefore, it cannot be used on a windows environment. Instead the *CreateFile* call is used. This function performs similarly to *open()* with a few addition.

The following code shows the syntax for *CreateFile*:

```
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
);
```

Upon a successful call *CreateFile* returns an object call an open handle to a specified file, device, or pipe, a return similar to that of the file descriptor return by the *open()* function. As seen on the table1 *CreateFile* has may parameters. These argument parameters share similarities to the parameter found in *open()*.

To specify a file name in *CreateFile* the user can be specified in the **IpFileName** field, similar to the pathname argument found in *open()* both of the parameters are treat. One of the biggest difference between the two function is that *CreateFile* can handle files, devices , and directories.

dwDesiredAccess parallel the access modes in *open()*, which set a specific flags to either read or write the files.

The **dwCreationDisposition** parameter is used to set a flag, this flag tells the function how duplicate files should be handled if the file has already be found. This can be done in a similar manor in *open()*.

Originally *CreateFile* was design to handle file communication, however, it's functionality have been greatly enhanced over the years. As a result *CreateFile* provides much more flexibility then *open()* .

The file types can be specified in the **dwFlagsAndAttributes**, additionally this can contain the Security Quality of Service flag, which provides a security mechanism for the file.

By modifying the **dwShareMode** users can request the share mode of the file/device. Syntax aside the majority of the windows I/O functions can used in a similar fashion to their POSIX counterpart.

2.2 Closing time

Handlers or file descriptors are similar to a refrigerator, once it has been open it's a good idea to close it. Closing handles make our code more readable and reliable. As mentioned before *CreateFile* creates a handle for the file, therefore, to close that handle we use the system call *CloseHandle*.

The following code shows the syntax for *CloseHandle*:

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

Similar to *close()* for POSIX *CloseHandle* takes only one argument, hObject takes the name of an open objects like files, I/O ports, token, threads, pipes and simply close it. Once called the function will return true or false depending if the Handle was successfully closed. However, unlike *close()*, *CloseHandle* return a non-zero value if the call is successful and a zero on failure, while *close()* will return 0 on success and -1 if not.

2.3 Learning to Read

The read function can be used once a desired file has been opened. Unlike the write function this function is used to this call is only used to retrieve information from the file. In a Unix environment the *read()* function is used to read a number of bytes from the associated open file descriptor. *ReadFile*() functions the same way with a few added functionalities.

The following code shows the syntax for *ReadFile*:

```
BOOL WINAPI ReadFile(
        _In_          HANDLE hFile,
        _Out_         LPVOID lpBuffer,
        _In_          DWORD nNumberOfBytesToRead,
        _Out_opt_     LPDWORD lpNumberOfBytesRead,
        _Inout_opt_   LPOVERLAPPED lpOverlapped
);
```

As usual the *ReadFile* takes the handled object as an argument and return a value upon completion. The values return values are similar to that of the CloseFile function. The function with either return a nonzero if success and a zero upon a failure. You may have have notice that the *ReadFile* parameters are similar to the *read()* function in Unix. If you didn't, here is a break quick brake down of these parameters.

hFile simply takes the handle for the file, this is equivalent to the file descriptor parameter in the *read()*.

IpBuffer resembles the buffer parameter in *read()*, which is a void pointer buffer that points to the address of the memory into which input data is placed. If the buffer is set to an invalid value the *ReadFile* may fail.

nNumberOFBytesToRead does exactly what the name suggest. It take nNumber of bytes and read it from the file. This resembles count parameter in *read()*.

The **IpNumberOfBytesRead** is a pointer that points to the number of bytes that were read, similar to the *read()* in Unix return value. But this parameter will only be used when passing a synchronous hFile parameter, synchronous meaning that the function will block the progress of a program until the read process is complete.

ReadFile also provides optional functionalities within the function, the **IpOverlapped** parameter to point to an overlapped structured if the hFile Object was open with the FILE_FLAG_OVERLAPPED flag set. Without diving into too much detail, this allows the *ReadFile* function to finish the read process without blocking the progress, also called Asynchronous file handling. Just as *lseek()* in Unix, it accomplishes this by changing the offset of the members in the OVERLAPPED structure.

We have just highlighted the main different between *read()* and *ReadFile*. As can be seen the two functions share a lot of similarities. But the read function for the Unix API provides less flexibility then windows.

2.4 Learning to Write

The write function is another call to use once a file has been opened. To write in a file on a Windows machine we use the *WriteFile* function, evidently in Unix the function is called *write()*. Since *WriteFile* is very similar to *ReadFile* we will only discuss the notable differences between *WriteFile* and *write()* which haven't been addressed.

The following code shows the syntax for *WriteFile*:

```
BOOL WINAPI \textit{WriteFile}(
        _In_          HANDLE hFile,
        _In_          LPCVOID lpBuffer,
        _In_          DWORD nNumberOfBytesToWrite,
        _Out_opt_     LPDWORD lpNumberOfBytesWritten,
        _Inout_opt_   LPOVERLAPPED lpOverlapped
);
```

Upon a successful call *WriteFile* will return a nonzero number, and upon a non-successful call it returns a zero. For *write()* there are also 2 possible return on success, it will return the number of bytes, else it will return -1. If nothing was written the number of bytes return will be 0.

Like parameters in *write()*, **hFile** handle is the handle name and **IpBuffer** is the pointer which points to the buffer that will contain the written data. **nNumberOfBytesToWrite** is the number of bytes to be written. The additional **IpNumberOfBytesWritten** is a pointer to the number of bytes written when handling synchronous objects and Overlapped function have been explained in the pervious section.

3 Pipes

Aside from being the spawn point for man-eating-flowers; pipes are important in interprocess communications (IPC). In Unix, they are one of the oldest methods of IPC, they provide an elegant solution to creating two simultaneous processes that can communicate with one another using system I/O. The 2 main Unix Pipes that were discussed in this class are the Anonymous pipes and FIFO. The first type of pipe is also called Unnamed pipes are local pipes that cannot be used for communication between local and network processes. The second type also known as named pipe, contains a 'name' and exist as special files in the system giving it the ability to communicate locally and non-locally. Analogously these 2 process can be found in Windows as well.

3.1 We are Anonymous

In Unix unnamed pipes can be created using the *pipe()* system call. To create an anonymous pipe in windows, the *CreatePipe* function can be used. Similar to Unnamed pipes in unix anonymous pipes are information flows one direction and is called *pipe()*. The *pipe()* call only takes up to 2 parameter

in which the first parameter is the file descriptor and the second is the a flag. Unlike *CreatePipe*, the process cannot be used alone to send data, it relay heavily on the use of *read()* and *write()* system call to read and write data.

The following code shows the syntax for *CreatePipe*:

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

Upon a successful the function return a non-zero number or a zero for an unsuccessful call. Contrary in POSIX the *pipe()* function return zero on success and -1 when unsuccessful. On a successful call *CreatePipe* returns 2 handles while *pipe()* returns 2 file descriptors.

hReadPipe and hWritePipe are pointers to the read and write handle of the pipe . The *CreatePipe* function creates a read and a write handle to a buffer using the ReadFile and *WriteFile* I/O functions. Like Unix, the way pipes are used in Windows is done through creating child processes. To do so a parent process can pass the pipe handles to another unrelated process using inheritance.

As usual the Window API has other parameter that aren't present in it's POSIX counterpart.

IpPipeAttribute, this parameter is a pointer to a SECURITY_ATTRIBUTES structure which determines if the handle can be have child process. It acts a extra security measure to check if the pipe can be used. It is an optional parameter, so if the parameter is blank the no security check will be done.

nSize is the parameter for the buffer size this holds an advantage over the *pipe()* function in that the system will us the passed value to calculate an appropriate buffering mechanism.

3.2 Named Pipes

Named pipes are arguably more useful then Unnamed pipes. These pipes makes it possible for 2 processes to communicate over a network. In POSIX each named pipe is defined as half-duplex pipes (unidirectional) because the nature of FIFO, which cannot communicate simultaneous. But simultaneous connection is possible in Window named pipe because they are byte/message-oriented and can handle both one-way and duplex pipes. It is possible due to each instance of the pipe having its own handles, and separate conduit for communication. The system call used in to create a pipe in this case is the *CreateNamedPipe* function. Named pipes in Unix and Window implementation is similar, first the connecting (client) process has connect to the current (server) process by using the "name" of the pipe. For the connection to form both client must be present. However, in Windows, the server will implement *CreateNamedPipe* until the client connect to the named pipe with the *CreateFile*.

The following code shows the syntax for *CreateNamedPipe*:

```
BOOL 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
);
```

As the following code shows handling Named Pipes for Unix and Windows is significant different. To create a named pipe in Unix we can use the mkfifo() function. It has 2 parameter, one for a specified

pathname and another for the mode. Although mkfifo share similar functionalities *CreateNamedPipe* provides the user will a bundle of additional functionality.

The **dwOpenMode** and **dwPipeMode** are parameters that are not present in mkfifo. The dwOpenMoade arguments is used to set the access mode of the pipes, changing the modes can change the pipe from a bidirectional to one-way. The dwPipeMode is used to determine the data that is being written into the pipe, the vald data types are bytes, and messages mode. The nMax-Instance is the number of instances that can be created in the pipe as mention before the instances make two-way communication possible.

4 Mutual Exclusion

In UNIX the*pthread_create()* system call allows the application to perform multiple tasks concurrently. When the function is called a main thread is created and the threads that are created share a global memory and heap, with private local variables. However the order of the threads to be used by the CPU is non-determinant. This isn't a problem in a single processor system, where threads can't be execute parallel. But while multithreading, it would be impossible to efficiently use treads without assistant from mutual exclusion.

4.1 Creating a thread in Windows

In windows, the equivalent call to create a thread is by using the *CreateThread* function.

The following code shows the syntax for *CreateThread*:

```
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
);
```

This function will return the handled value of the new thread on success. Additionally, there are option parameters that reused to creating a thread that are similar to*pthread_create*.

IpThreadAttribute is a flag parameter similar to the IpPipeAttribute in *CreatePipe*.

IpStartAddress, and **IpParamter** are similar parameters to the thread function and argument parameter in *pthread_create()*.

dwStackSize is number of bytes in the stack. Unlike *pthread_create* *CreateThread* allows us to specified the initial size of the thread stack inside the function, the golden rule to choosing the size is to choose the smallest size possible.

dwCreationFlag is another special parameter that allows more control over the function. When the flag is zero the thread will run immediately after it hase been create, but a CREATE_SUSPENDED flag maybe set to create a suspended that will resume when ResumeThread is called. This is flag is the unix equivalent of pause(), in which the way to resume the thread is kill()-ing the thread. STACK_SIZE_PARAM_IS_A_RESERVATION flag can be set to used the specific parameter in dw-StackSize as the initial stack.

IpThreadId is the pointer to the thread id that is produced, in POSIX this is the return value of *pthread_create()*.

4.2 Mutex

In computer sciences mutual exclusion is used to ensure that threads can use synchronized their actions. Mutual exclusion allows threads to synchronized their actions with the use of shared resources. The underlying process behind Mutual Exclusion(Mutex is that no two processes can access the same variable at the same time. Because threads share the same global memory, the memory should not be modify without the use of Mutex.

We can achieve this by protect the resources from each process by locking and unlocking it the resource. In POSIX this can be achieved with the *pthread_mutex_init()* followed by the *pthread_mutex_lock()* function in which the argument is the accepts a valuable from the pthread_mutex_t structure. Then the *pthread_mutex_unlock()* function can be used to unlock the mutex.

Similarly in windows, a mutex object has to be created with *CreateMutex*:

```
HANDLE WINAPI CreateMutex(
        _In_opt_  LPSECURITY_ATTRIBUTES lpMutexAttributes,
        _In_      BOOL bInitialOwner,
        _In_opt_  LPCTSTR lpName
);
```

Instead of having a lock style function, once the mutex object has been created it can be assigned to one thread at a time. Like the ball in a game of football. The thread with the Mutex object is the only thread that can write in the shared memory. This performs similarly to *pthread_mutex_init()* in combination with the *pthread_mutex_lock()* funciton, when this function is called the user may set the **bIntitialOwner** to determine which thread "own" this mutex object. If set to true, the calling thread will have ownership, else the ownership of the object can be assigned later.

A Mutex name can be assigned the object in the **IpName** parameter. In the case that a mutex object is not assigned to any thread the *OpenMutex* function can be used. If the Mutex object was created with a IpName assign, the *OpenMutex* function can be used to give the Mutex object to a valid thread.

Once the thread is done writing the object maybe unlocked", it release the Mutex to with the use of the *ReleaseMutex*. The function takes one parameter, which is, the mutex handles name. This is similar to *close()* function. If the thread that "owns" the mutex did not properly release it upon terminating it can cause serious error.

5 Parallel Process

The POSIX *fork* process can be use to create new processes. As we know *fork* in POSIX works by creating a child copy of the parent process, once at child process is successful created the program ends up with 2 process that execute in the same line of code. However, in windows there is no such function. The reason is because the nature of the Windows API makes it hard to manage the multiple process when there are multiples threads.

5.1 Creating New Process

To create additional process Windows uses the is *CreateProcess* system call. The *CreateProcess* doesn't create the same child-parent relationship as in Unix. Instead an entirely new process is created. Therefore, once the new process is created with *CreateProcess* it will not be terminated even when the original process ends. This function is analogous to the exec() and *fork* command combine in POSIX.

The following code shows the syntax for *CreateProcess*:

```
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
);
```

On a successful return the function will return a true or false depending on it's success. Unlike *fork* the process id isn't return, but *CreateProcess* creates a pointer (**IpProcessInformation**) that points to the process id. As seen unlike *fork* the *CreateProcess* has many more parameters.

The first argument **IpApplicationName** specifies the application name will be excruciated. This optional parameter used along with the **IpCommandline** parameter can be used to executed any program that will run on the local machines, very similar to the *execve()* in POSIX. When *CreateProcess* create a brand new process unless the **bInheritHandles** is set to true each inheritable handle in the process will not be inherit by the new process. The **IpProcessAttribute** and **IpThreadAttribute** is are security parameter, that will check if the new process can inherit the new process object. Using these parameter is similar to using other attribute parameter which points the SECURITY_ATTRIBUTES structure. e.g. **IpPipeAttribute** in *CreatePipe*.

As mentioned before after a successful *CreateProcess* call, the child process created is an independent program. Therefore, the wait() function isn't necessary when using the Windows API. Once a process is finished *ExitProcess()* maybe used to terminate a process, similarly in Unix *exit()* can be used.

6 Reference:

"Apache Hadoop Fundamentals – HDFS and MapReduce Explained with a Diagram." The Geek Stuff RSS. N.p., n.d. Web. 12 Dec. 2014.http:\\www.thegeekstuff.com/2012/03/linux-threads-intro/h.

"Asynchronous I/O." Wikipedia. Wikimedia Foundation, 12 Nov. 2014. Web. 12 Dec. 2014. http:\\en.wikipedia.org/wiki/Asynchronous_I/O.

"*CloseHandle* Function." (Windows). N.p., n.d. Web. 11 Dec. 2014. http:\\msdn.microsoft.com/en-us/library/windows/desktop/ms724211(v=vs.85).aspx.

"*CreateFile* Function." (Windows). N.p., n.d. Web. 10 Dec. 2014. http:\\msdn.microsoft.com/en-us/library/windows/desktop/aa363858%28v=vs.85%29.aspx.

"*CreateMutex* Function." (Windows). N.p., n.d. Web. 12 Dec. 2014. http:\\msdn.microsoft.com/en-us/library/windows/desktop/ms682411(v=vs.85).aspx.

"*CreatePipe* Function." MSDN. Microsoft, n.d. Web. 10 Dec. 14. http%3A%2F%2Fmsdn.microsoft.com%2Fen-us%2Flibrary%2Fwindows%2Fdesktop%2Faa365152(v%3Dvs.85).aspx.

"*CreateProcess* Function." (Windows). N.p., n.d. Web. 12 Dec. 2014. http:\\msdn.microsoft.com/en-us/library/windows/desktop/ms682425%28v=vs.85%29.aspx.

"Creating Threads." (Windows). N.p., n.d. Web. 11 Dec. 2014. http:\\msdn.microsoft.com/en-us/library/windows/desktop/ms682516%28v=vs.85%29.aspx.

Kerrisk, Michael. The Linux Programming Interface: A Linux and UNIX System Programming Handbook. San Francisco: No Starch, 2010. Print.

"Mutual Exclusion." Wikipedia. Wikimedia Foundation, 12 Nov. 2014. Web. 12 Dec. 2014. http:\\en.wikipedia.org/wiki/Mutual_exclusion.

"Named and Unnamed Pipes: Clearing the Confusion." Named and Unnamed Pipes: Clearing the Confusion. N.p., n.d. Web. 10 Dec. 2014. http:\\www.cs.fredonia.edu/zubairi/s2k2/csit431/pipes.html.

"SAS(R) 9.2 Companion for Windows, Second Edition." Overview of Pipes. SAS Institute Inc., n.d. Web. 12 Dec. 2014. http://support.sas.com/documentation/cdl/en/hostwin/63285/HTML/default/viewer.htm#pipeover.htm