CSS344-400 Final Exam: Comparison of POSIX and Microsoft Windows API Features

Adam Cankaya, cankayaa@onid.oregonstate.edu

August 31, 2014

Extra Credit: A Fourth Topic!
Extra Extra Credit: In LaTeX!

Contents

1 Introduction

This paper was written for the user who is already familiar with the POSIX programming environment and wishes to see how already known concepts are implemented under the Windows API. It is assumed that the reader is not only already familiar with the concepts discussed, but is also comfortable reading the man pages for Unix-like operating systems and the Microsoft Developer Network (MSDN) reference pages for Windows. As such, specific details and examples on function and object implementation that are not covered here can be found in these free online resources.

2 Comparison of Threads

2.1 Data Types

POSIX pthread objects each have their own functions and data types: pthread_t, pthread_mutex_t, etc. The user must be familiar with both the functions and the data types involved in order to use them. In comparison, the Windows API thread objects return only a HANDLE, which references another resource, such as a file stream or pipe.[2]

2.2 Creating Threads

Thread creation and initialization in POSIX involves multiple commands including pthread_create(), pthread_attr_init(), pthread_attr_setstacksize(), and pthread_attr_destroy(). Each of these functions take in and return specialized variable types, such as pthread_t() and pthread_attr_t().

The Windows API simplifies thread creation with CreateThread() that simply returns a handle to the new thread. Here we can set whether the thread can be inherited by a child process, the stack size, the function to be executed by the thread, along with the variables to be passed in, the variable that receives the thread identifier, and flags to determine whether the thread should be run immediately or held until ResumeThread() is called.[3]

2.3 Exiting Threads

There are multiple ways to exit a thread in POSIX: the thread can return a value, the thread call pthread_exit() or pthread_cancel(), or a thread can call exit() which subsequently causes all other threads in the process to terminate. [1, p. 623]

Exiting a thread on Windows is relatively similar. Just as pthread_exit() requires a return value variable be passed into it, ExitThread() requires a DWORD exit code. When ExitThread() is called, the thread's stack is immediately deallocated and all pending I/O is canceled. If the thread is the last one within a process then the process itself also terminates. However, the thread object itself is not necessarily removed from the system, as this is only done once all handles to the thread are also closed. [3]

2.4 Thread Synchronization

POSIX pthreads can use signals to synchronize access to shared resources with methods such as semaphores, mutexes, and conditional variables. Each of these methods have their own functions and data types in the POSIX environment and all signals must be either caught by waiting threads or discarded - otherwise the user runs the risk of losing a signal. Threads in the Windows API can also be synched by use of semaphores and mutexes, and in addition they can utilize events and critical sections. There is also shared functionality across these synchronization types, with each using functions such as WaitForSingleObject() and CloseHandle().[2]

2.4.1 Semaphores

Windows and POSIX semaphore functions are relatively similar. Each system has a function for creating, using, and destroying semaphores. POSIX allows both named and unnamed semaphores. Functions include sem_init() (for unnamed) and sem_open() (for named) to create a semaphore, sem_wait() and sem_post() to decrement/increment the semaphore, and sem_destroy() (for unnamed) and sem_close() (for named) to terminate the association between a process and semaphore. [1, Ch. 52]

Windows has similar functions to create, decrement/increment and close semaphores: CreateSemaphore(), OpenSemaphore(), WaitForSingleObject(), ReleaseSemaphore() and CloseHandle(). [4] The main difference between the two API's implementation of semaphores is in their breadth - Windows semaphores are created within a thread and and propagated across the whole system, while POSIX semaphores are used for communication between threads within a single process or communication between multiple processes themselves. [2]

2.4.2 Mutexes

The functions provided for mutexes are also similar in POSIX and Windows. The reader is already familiar with POSIX functions such as pthread_mutex_int(), pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock(), and pthread_mutex_destroy. [1, Ch. 30]

The Windows API provides similar functionality with CreateMutex(), OpenMutex(), WaitForSingleObject(), ReleaseMutex(), and CloseHandle(). The only real differences are with the data types used as the parameters for the functions - both the

POSIX and Windows environments require certain specialized data structures. For example, CreateMutex() takes in a LPSECURITY_ATTRIBUTES parameter, which specifies a security file descriptor. As with thread implementation in general, Windows uses handles to interact with the mutex.

The general strategy is to use CreateMutex() to generate a mutex object and use CreateThread() to create threads to do work. When a thread wants to do work, it will first request ownership of the mutex by using WaitForSingleObject(). Once it obtains owernship of the mutex, the thread will do it's work, release ownership using ReleaseMutex(), and then terminate or exit the thread.[4]

2.4.3 Events

The Windows API implements a feature called events, which can be used for signaling threads that a particular event has occurred. Events are either set as signaled or unset as nonsignaled and can either be manually reset by the user or automatically reset when a waiting thread is released. [4] The Windows functions for interacting with events include CreateEvent(), OpenEvent(), SetEvent(), ResetEvent(), and once again, WaitForSingleObject() and CloseHandle().

Similar features of events can be used in POSIX by utilizing semaphores with count set to 1, however no timeout can be specified, risking a program running forever, and communication can only be between threads within a single process. Condition pthread functions include pthread_cond_init(), pthread_cond_signal(), pthread_cond_wait(), pthread_cond_timedwait(), and pthread_cond_destroy(). [2]

2.4.4 Critical Sections

In addition to events, the Windows API also provides another synchronization tool called critical sections which provide synchronization similar to POSIX mutexes except critical sections provide communication only between threads within a single process. In a single-process program, critical sections provide a more efficient method for mutual-exclusion synchronization, however unlike mutexes, there is no way to determine whether a critical section has completed. [4] The Windows functions for interacting with critical sections include InitializeCriticalSection(), InitializeCriticalSectionAndSpinCount(), EnterCriticalSection(), TryEnterCriticalSection(), LeaveCriticalSection(), and DeleteCriticalSection(). [2]

2.5 Additional Differences

- Under Windows, events are by default set to signaled, while in POSIX, pthreads do not provide an initial state.
- Windows event objects are asynchronous (non-blocking), while POSIX conditional variables are synchronous (blocking). This means that under Windows, applications implementing events will not wait for I/O to complete

before continuing with processing that does not require the completion of the I/O.

- Windows thread scheduling is implemented in the kernel only. POSIX implements threads both at the user and the kernel levels - generally user created threads are in the user space, but also implemented as a kernel level thread. Exactly how this is implemented depends upon the specific OS.[2]

3 Comparison of Time

3.1 Calendar Time

As we assume the user is already familiar with, POSIX time is represented internally as the number of seconds since what is called the Unix Epoch; the start of the year 1970, or more specifically, midnight UTC on the first day of January 1970. Like with other areas of POSIX, the user must be familiar with both the time related functions and their counterpart data types. For example, gettimeofday() can be used to retrieve the current time, but it takes in as parameters a struct timeval pointer and a struct timezone pointer. Similarly, time() returns the number of seconds since the Epoch in the form of a const time_t pointer, the evaluated value of which then must be further manipulated by the user to display a result in human readable form.[1, Ch. 10]

Similarly Windows also has a function to return time from a specific point in the past. Called FileTime, and returned using the function GetFileTime(), it is defined as the number of 100-nanosecond intervals since January 1, 1601. FileTime can be converted to system time and back again by use of the FileTimeToSystemTime() and SystemTimeToFileTime() functions.

3.2 Conversion of Time

POSIX does provide several functions to help make easier the use of the time_t data type. The simplest being ctime() which takes in a pointer to a time_t variable and then prints out a 26 byte string in the format "Wed Apr 20 18:34:30 2014". The functions gmtime() and localtime() both return a struct tm variable. Known as "broken-down time", the variable can then be further broken down into a series of int's representing the individual components of time (seconds, minutes, hours, etc). We can also use strftime() to convert broken-down time into a more printable form.[1, Ch. 10]

Like POSIX, the Windows API system time is given in coordinated universal time (UTC) zone, and the LocalTime is adjusted based on the local time zone. Time is often displayed in the form of struct _SYSTEMTIME which contains numerical values for the day, month, year, etc. The difference between current system and local time is displayed in the following example[5]:

Example 1 : Use of the Windows GetSystemTime and GetLocalTime functions.

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME st, lt;

    GetSystemTime(&st);
    GetLocalTime(&lt);

    printf("The system time is: %02d:%02d\n", st.wHour, st.wMinute);
    printf(" The local time is: %02d:%02d\n", lt.wHour, lt.wMinute);
}
```

```
// Sample output
The system time is: 19:34
The local time is: 12:34
```

3.3 Time Zones

POSIX systems store time zone information in /usr/share/zoneinfo. Within this directory are files that contain information about different regions and their time zones. The "local time" for a system is defined by a file in /etc/locatime which is usually a link to a file in the /usr/share/zoneinfo folder. Within programs, the "local time" can be defined by setting the TZ environmental variable. This variable subsequently influences the results of time functions such as localtime().[1, Ch. 10]

In the Windows environment, time zone information for global regions are stored in the time_zone_name registry. Each entry is of the form struct _TIME_ZONE_INFORMATION, which contains information such as the difference in minutes between UTC & local time (the bias) and dates for changing to and from daylight savings time. The system's local time zone can be displayed by use of the GetTimeZoneInformation() function and adjusted using the SetTimeZoneInformation() function.[5]

Example 2 : Retrieving the current system time zone information in the Windows environment.

```
TIME_ZONE_INFORMATION tzi;
DWORD dwRet;
dwRet = GetTimeZoneInformation(\&tzi);
if(dwRet == TIME_ZONE_ID_STANDARD || dwRet == TIME_ZONE_ID_UNKNOWN)
       wprintf(L"%s\n", tzi.StandardName);
else if( dwRet == TIME_ZONE_ID_DAYLIGHT )
       wprintf(L"%s\n", tzi.DaylightName);
```

```
else {
       printf("GTZI failed (%d)\n", GetLastError());
}
```

3.4 Process Time

Process time is the amount of time used by a CPU since the process was created. A POSIX process can call the times() function, which returns a clock_t variable, representing the number of "clock ticks" from some arbitrary undefined time in the past. Because this starting point in the past is unknown, the times() function is mostly used for counting elapsed time between two calls. In general, we can divide the number of clock ticks by 100 to obtain the number of seconds (this value of 100 represents the number of ticks per second and can be different depending upon the system - we can use sysconf(_SC_CLK_TCK) to obtain the exact conversion value). Similarly we can use clock() to measure the total CPU time used by a calling process. The value returned by clock() is measured in units of clocks per second, which is fixed at one million clocks per second, and so we simply divide the returned value by one million to obtain the total CPU time in seconds. [1, Ch. 10]

Similarly in Windows, the GetProcessTimes() function is used to measure the actual CPU time used by a process. The results are expressed in the FileTime format and include the process creation and exit time, along with the time the process spent in kernel and user modes. [5]

4 Comparison of Sockets

4.1 Creating a Socket

In POSIX systems, creating a socket begins with a call to socket(), to which three integers are passed as parameters. The first parameter represents the domain, which is traditionally expressed by the constants AF_UNIX, AF_INET or AF_INET6. The second parameter is the type, which is usually either the constant SOCK_STREAM for a streaming socket or SOCK_DGRAM for a datagram socket. The third parameter is the protocol, who's value is usually set to 0 for normal socket data types. If successful, the socket() call will return a socket file descriptor. [1, Ch. 56]

The Windows socket (WinSock) environment is invoked with a call to #include <winsock.h> and is very similar to the POSIX socket system. Like POSIX, WinSock uses the sockaddr struct as well as the socket() and connect() functions, which take in the same parameters previously discussed. [6] There is however one key additional requirement in the WinSock systems - the first WinSock function called in an application must be to WSAStartup(). This function is responsible for communicating information between Windows and the socket program. It takes in two parameters and then returns a zero on success or non-zero on failure. The first parameter is the version of WinSock the program

wishes to implement. Current options are 1.0, 1.1, 2.0, 2.1, & 2.2, and the parameter should be passed within a call to MAKEWORD() to ensure that it is in the format expected by WSAStartup(). The second parameter is a pointer to a WSADATA structure that will receive information from Windows about the socket implementation. [7]

Example 3 : WSAStartup() to initiate usage of WinSock.

```
void main(int argc, char *argv[]) {
        int sock;
        struct sockaddr_in echoServAddr;
        unsigned short echoServPort;
        char *servIP;
        char *echoString;
        char echoBuffer[RCVBUFSIZE];
        int echoStringLen;
        int bytesRcvd, totalBytesRcvd;
        WSADATA wsaData;

        if ((argc < 3) || (argc > 4)) {
                fprintf(stderr, "Usage: %s <Server IP> <Echo Word>
                [<Echo Port>]\n",argv[0]);
                exit(1);
        }
        servIP = argv[1];
        echoString = argv[2];
        if (argc == 4)
                echoServPort = atoi(argv[3]);
        else
                echoServPort = 7;
        if (WSAStartup(MAKEWORD(2, 0), &wsaData) != 0) {
                fprintf(stderr,"WSAStartup() failed");
                exit(1);
        }
}
```

4.2 Binding a Socket

Once a socket file descriptor is generated in POSIX, a call to bind() can be made on the server side to bind the socket to a fixed address. The bind() function takes in the socket file descriptor, along with a struct sockaddr pointer specifying the address the socket is to be bound, as well as a socklen_t parameter for the address length. Similar to other POSIX functions, the user must be familiar with both the functions as well as the specific data structure types required. [1, Ch. 56]

The bind() function under WinSock is similar as POSIX, taking in a socket file descriptor and a sockaddr pointer, but the name length variable is a regular int compared to the socklen_t struct required in POSIX. [7]

4.3 Stream Sockets

When creating a streaming type socket in POSIX, once the server address has been bound to the socket, a call to listen() is made on the server, indicating that it is ready to accept any incoming connections. The server can then call accept() which will block until a client connection is established. The client simply needs to make a call to connect() and pass in the address of the server socket to which it wishes to connect. Once the two sockets are connected, bidirectional data transmission can be accomplished using read() and write() until the client or server calls close(). Generally the server acts as the passive socket, listening for connections, while the clients are active sockets, looking to connect with a passive socket on the server. [1, Ch. 56]

The data transmission functions are the same for WinSock stream sockets as they are in POSIX, however once a socket connection is no longer needed, the close() function is slightly different. Instead of using close(), WinSock requires closesocket() - both require the same parameter, specifically the socket file descriptor that should be closed. In addition, once a socket has been closed, WinSock also requires a call to WSACleanup(), which will deallocate and release any resources reserved by the previous WSAStartup() call. Socket related errors can also be printed using a call to WSAGetLastError() immediately after the socket function call.[6]

4.4 Datagram Sockets

POSIX and Windows datagram sockets appear to be nearly identical in their implementation. Both environments involve a call to socket(), followed by using bind() on the server's address, but data transmission involves specific datagram compatable functions, specifically sendto() and recvfrom(). The parameters for these functions differ slightly between the two environments - calls under WinSock do not require specialized data types (except sockaddr), instead allowing simple int and char types, but otherwise operate the same as under POSIX. Of course, as is the case with stream sockets, socket related errors are reported under Windows using the WSAGetLastError() function. [1, Ch. 56][7]

5 Comparison of Pipes

POSIX pipes can be either unnamed or named (FIFO) - with unnamed pipes not having persistence beyond their process. Windows NT also supports both named and unnamed ("anonymous") pipes, however there are some key differences when compared to the POSIX pipe. Windows named pipes can act similar to sockets - specifically allowing communication between unrelated processes across different networks. This is accomplished by using multiple instances of the same named pipe - with each instance sharing the pipe name, but having their own buffers and handles, providing an isolated form of communication.[8]

5.1 Creating Pipes

As we assume the reader is already familiar with, in the POSIX environment, the pipe() function is used to create a new pipe. A successful call to pipe() returns an array of two file descriptors - one for the read end, one for the write end. Once a POSIX pipe has been established, we can then use read(), write(), fdopen(), printf() and scanf() on the pipe just like any other file descriptor. The pipe can be used for both intra-process and inter-process communication. [1, Ch. 44]

A named pipe can be created in Windows by use of the appropriately named CreateNamedPipe() function, which returns a handle for either the first instance of a named pipe or a new instance of an existing named pipe. An anonymous pipe can be created using CreatePipe(), which returns two handles - a read handle and a write handle. These handles can then be passed through inheritance to a child process or can be duplicated by using DuplicateHandle() and sending the duplicate to an unrelated process (see Interprocess section below). [8]

5.2 Using Pipes - Intraprocess

With a single process, a pipe can be used to execute UNIX shell commands with input sent and output received. This can be accomplished by use of the popen() and pclose() functions. The popen() function takes in two strings representing the command to be run and the mode (read or write). The function then automatically creates a pipe, forks a child process which in turns open a shell, which then forks another child process, which then executes the command and uses the pipe to read the output. If the popen() call is successful, it will return a file stream pointer for usage of stdin and stdout.[1, Ch. 44]

Intraprocess communication in Windows can be accomplished using anonymous (unnamed) pipes (it can also be done using named pipes, but these require additional overhead and also provide a potential security leak as they can be accessed from outside the network). [8]

5.3 Using Pipes - Interprocess

For connecting two or more POSIX processes, a call to fork() is made after the pipe has been established. This will create a child process which will inherit a copy of the parent's file descriptors. Once a parent and a child process is established, generally one will use close() to close its read end and the other will close its write end. Not doing this can cause race conditions with multiple processes trying to access the same end of the pipe at once. If bi-directional communication is desired, creating a second pipe is usually the easiest method. [1, Ch. 44]

Once a named pipe has been created in Windows by use of the CreateNamedPipe() function, communication can be established between two or more processes. The process that will be accepting a connection (server) will utilize the ConnectNamedPipe()

function, while the process attempting to establish a connection (client) will use either CallNamedPipe() for message-type communication or CreateFile() for data I/O. Asynchronous read and write can be accomplished using a named pipe and the ReadFileEx() and WriteFileEx() functions.

By using DuplicateHandle() and passing the handle via shared memory, one could in theory use anonymous pipes to accomplish a form of interprocess communication. A process can close an unnamed pipe by using the CloseHandle() function or simply terminating the process. [8]

References

[1] Kerrisk, Michael.
The Linux Programming Interface: A Linux and UNIX System Programming Handbook.
1st ed. San Francisco: No Starch, 2010.

[2] Sayegh, Robert. "PThreads Vs Win32 Threads", 2008.
Accessed 20 Aug. 2014.
<http://www.slideshare.net/abufayez/pthreads-vs-win32-threads>.

[3] "Process and Thread Reference".
Accessed 23 Aug. 2014.
<http://msdn.microsoft.com/en-us/library/windows/desktop/ms684852(v=vs.85).aspx>.

[4] "Synchronization Reference".
Accessed 24 Aug. 2014
<http://msdn.microsoft.com/en-us/library/windows/desktop/ms686679(v=vs.85).aspx>.

[5] "Time Reference".
Accessed 26 Aug. 2014.
<http://msdn.microsoft.com/en-us/library/windows/desktop/ms724962(v=vs.85).aspx>.

[6] "Transitioning From UNIX to Windows Socket Programming".
Accessed 29 Aug. 2014.
<http://cs.baylor.edu/ donahoo/practical/CSockets/WindowsSockets.pdf>.

[7] "Winsock Reference".
Accessed 29 Aug. 2014.
<http://msdn.microsoft.com/en-us/library/windows/desktop/ms741416(v=vs.85).aspx>.

[8] "Pipes Reference".
Accessed 31 Aug. 2014.
<http://msdn.microsoft.com/en-us/library/windows/desktop/aa365784(v=vs.85).aspx>.