Final

Sam Quinn

December 9, 2013

1 Signals

Signals are a from of notification that are sent between processes. Many times these signals are sent to let a processes know that an event has occurred. Signals are sent to and from software but have many similar features of hardware interrupts. Signals can be triggered by the kernel and the user. The kernel will send a signal if the hardware has notified the kernel of an invalid operation. A user my terminate a processes with the *Control-C* which sends a signal to the running process. Lastly if implemented signals can be used as a form of IPC where processes send signals between each other that a specific event has occurred.

1.1 Signals

POSIX
POSIX has many signals > 25, each of these signals are used for simple IPC and abnormal scenarios that result in an error (e.g. divide by 0).

Common POSIX Signals

- SIGHUP
- SIGCHLD
- SIGINT
- SIGKILL
- SIGQUIT

Windows
Windows doesn't have as many signals as POSIX uses but supports some. Others can be implemented with the Windows API and messages.

All of Windows Signals

- SIGABRT

- SIGFPE
- SIGILL
- SIGINT
- SIGSEGV
- SIGTERM

1.2 Sending Signals

Windows Syntax
In a Windows program signals are sent either with the *signal()* system call or the *raise()* system call. Each of these calls perform differently. The *signal()* call sets up an association for the execution of any signal received while the *raise()* function programmatically generates a signal within an application. Many times Windows programs will make use of Windows Messages to send information to another process rather than sending signals. The way Windows sends messages is with the *PostMessage()* call or the *SendMessage()* call. The *PostMessage()* call adds the message to the message queue that is associated with the threads or processes. The *SendMessage()* call will send the message and will not return until the thread or process has dealt with the message.

POSIX Syntax
In POSIX signals are sent with the *kill()* system call similar to the *raise()* system call in Windows. The *kill()* system call in POSIX will send a designated signal to a specific process. Conversely the Windows *raise()* system call will only send the designated signal to the current process.

1.3 Handling Signals

In both POSIX and Windows you must include *signals.h* library to be able to use signals with in a program. Also in both you may use a signal handler that you create or use a predefined signal handler. If you wish to use the predefined handler the constants are *SIG_DFL*-to use the default action for each signal or *SIG_IGN*-Ignore signals, these constants exist in both POSIX and Windows. There a very few discrepancies between the way POSIX and Windows handle signals and can almost be programmed with the same syntax.

2 File I/O

While file I/O is often misconceived as only the process of reading and writing to a file, file input output covers a far broader spectrum. File I/O can not only be use to read and write to files but the techniques used with in file I/O are universal to many other programming needs. File input output can be use to read and write to and from pipes, sockets, files, FIFO's, terminals, devices and more. With out a doubt file I/O is one of the most important concepts to understand with in programming. Both in POSIX and in the Windows API file descriptors are used to pin point an opened file. As a standard in both POSIX and in Windows programs the file descriptors for STDIN, STDOUT, and STDERR are the same. STDIN = 0, STDOUT = 1, and STDERR = 2.

2.1 Open

POSIX Syntax
Within POSIX you can open or create a file with the *open()* system call. This call returns a file descriptor of the file it just opened or created. The *open()* call has a wide variety of flags you may use wile opening or creating a file to suit your needs.

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

A break down of what all the variables mean.

- pathname - The name of a file or device in the local directory or a path to a file or device in another directory.
- flags - The requested access to the file or device, which can be read, write, both or neither.
- mode - Permissions to set if creating file. Sets the permissions to Group, User, and Other with read, write, both, or neither.

Windows Syntax
The way Windows opens or creates a file is with the *CreateFile()* system call. If this call executes successfully it will return a handle that for the file it has opened or created. The *CreateFile()* call seems to me to have many more features that the POSIX *open()*. Some of these features that exist solely in the Windows API are included like SecurityAttributes and Templates. This may be beneficial for many reasons but require the programmer to fill in more variables than in the POSIX *open()* function which could be annoying.

```
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
);
```

A break down of what all of the variables mean.

- lpFileName - The name of the file or device to be created or opened.
- dwDesiredAccess - The requested access to the file or device, which can be read, write, both or neither.
- dwShareMode - The requested sharing mode of the file or device, which can be read, write, both, delete, all of these, or none.
- lpSecurityAttributes - A pointer to a SECURITY_ATTRIBUTES struct. *Note this is optional.
- dwCreationDisposition - An action to take on a file or device that exists or does not exist.
- dwFlagsAndAttributes - Extra flags and attributes.
- hTemplateFile - A handle to a template file. *Note this is optional.

2.2 Read

POSIX Syntax
In POSIX the *read()* system call is use to read data from anything that can be represented with a file descriptor. When *read()* reads from the descriptor successfully it will return the number of bytes read. On an error *read()* will return -1.

```
ssize_t read(int fd, void *buf, size_t count);
```

A break down of what all of the variables mean.

- fd - A valid file descriptor in which you wish to read from.
- buf - A pointer to a buffer to store the read characters in.
- count - The number of bytes to read from the file descriptor.

Windows Syntax
In Windows for reading from a file or device the *ReadFile()* system call is used. The *ReadFile()* system call reads from both synchronous and asynchronous files or devices. If *ReadFile()* system call runs successfully it will return TRUE(non-zero) and FALSE(zero) if failed. Like in previous comparisons of Windows and POSIX system calls the Windows one has more variables which I will explain below.

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

A break down of what all of the variables mean.

- hFile - A handle to either a file or a device.
- lpBuffer - A pointer to a buffer to store the read characters.
- nNumberOfBytesToRead - The maximum bytes to read.
- lpNumberOfBytesRead - A pointer to the number of bytes already read. *Note this is optional.
- lpOverlapped - If the falg FILE_FLAG_OVERLAPPED is used when opening the file or device this is a pointer to an OVERLAPPED struct. *Note this is optional.

2.3 Write

POSIX Syntax
In POSIX the *write()* system call is used to write to a specific file or device with a valid file descriptor. If *write()* writes to the file or device successfully it will return the number of bytes written, if *write()* fails it will return -1.

```
ssize_t write(int fd, const void *buf, size_t count);
```

A break down of what all of the variables mean.

- fd - A valid file descriptor in which you wish to write to.
- buf - A pointer to a buffer in which the characters to write are stored.
- count - The number of bytes to write to the file descriptor.

Windows Syntax
In Windows the *WriteFile()* system call is used to write to a file or device. If *WriteFile()* system call runs successfully it will return TRUE(non-zero) and FALSE(zero) if failed. Similarly to the *ReadFile()* System call used by Windows programs *WriteFile()* system call also is designed for both synchronous and asynchronous operation.

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

A break down of what all of the variables mean.

- hFile - A handle to either a file or a device.
- lpBuffer - A pointer to a buffer in which the characters to write are stored.
- nNumberOfBytesToWrite - The number of bytes to be written to the file or device.
- lpNumberOfBytesWritten - A pointer to the number of bytes already written.
- lpOverlapped - If the falg FILE_FLAG_OVERLAPPED is used when opening the file or device this is a pointer to an OVERLAPPED struct. *Note this is optional.

2.4 Close

POSIX Syntax
To close a file descriptor in POSIX the *close)* system Call is used. If a file is no longer needed to be accessed through a file descriptor the *close()* system call may be used to disassociate the assigned file descriptor from that file. After a file descriptor is closed that file descriptor may be used again for another file. The *close()* system call will return 0 on a successful close and -1 on a failed close.

```
int close(int fd);
```

- fd - A vaild file descriptor in which you wish to close.

Windows Syntax
To close a handle in Windows the *CloseHandle()* system call is used. The *CloseHandle()* call will close the specified handle and invalidate all pending operations that use that handle. If *CloseHandle()* system call runs successfully it will return TRUE(non-zero) and FALSE(zero) if failed.

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

- hObject - A valid handle to an open object.

3 IPC - Synchronization

Programs that take advantage of multiple threads or processes need some way of communication between them. While IPC is a broad topic which covers all inter process communication I am only comparing synchronization facilities that is encompassed with in the broader IPC term. Synchronization between processes is needed to coordinate their actions to avoid multiple process updating a region of shared memory at the same time. When two or more processes try to update the same file it is known as a race condition. The two synchronization facilities that I am going to cover to avoid race cases between processes are semaphores and mutexes.

3.1 Mutexes

Since threaded programs share global variables mutexes are often the go to synchronization facility for threaded program. Mutexes allow serialization of specific parts of code in a parallel program. The Serialized section of code is referred to as the critical section. Mutexes have only two states locked and unlocked. If a mutex is locked and another process tries to access the critical section it will block and wait for the process in the critical section to finish. Once the mutex unlocks the mutex will now allow the waiting process to enter the critical section.

3.1.1 Creating a Mutex

POSIX Syntax
In POSIX to create a mutex you must define a global variable with the type *pthread_mutex_t* followed by the name of your mutex. In POSIX if you are using threads then you must include the *pthread.h* library to use both the thread functionality and mutexes. Each mutex defined must be initialized for statically allocated mutexes the *PTHREAD_MUTEX_INITALIZER* constant is used.

Windows Syntax
In Windows programs they handle creating mutexes a little differently. To create a mutex in Windows you must use the *CreateMutex()* function. The *CreateMutex()* function will return the handle to the mutex on a successful creation or return NULL on a failed creation.

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

An overview of what each variable means.

- lpMutexAttributes - A pointer to a SECURITY_ATTRIBUTES struct. *Note this is optional.
- bInitialOwner - Determines weather or not to give ownership to the calling thread.
- lpName - The name of the mutex object.

3.1.2 Using a Mutex

POSIX Syntax
To lock and unlock a mutex in POSIX the functions *pthread_mutex_lock()* and *pthread_mutex_unlock()* are used. Each of these functions are required to be passed a valid pointer to a mutex that has been initialized. When either of these functions executes successfully they return zero and if they fail errno is set accordingly.

Windows Syntax
In a Windows program that uses mutexes the functions *WaitForSingleObject()* and *ReleaseMutex()* functions are used. The *WaitForSingleObject()* is analogous to the POSIX *pthread_mutex_lock()* function but gives an option to define a time out interval. If the time out interval is reached then the process waiting on the mutex stops waiting and continues without ever accessing the critical section. The Windows *ReleaseMutex()* function has the exact same functionality as the POSIX counterpart.

3.2 Semaphores

Semaphores work are used as a synchronization facility for both threaded programs as well as multiple processed programs. Semaphores have the same general concept as mutexes in which they protect access to a critical section of code. The main difference between semaphores and mutexes are that semaphores are a non negative integer that can be incremented or decremented. When a semaphore is zero the next process who requests access to the critical section is granted access. Each request for access there after increments the semaphore. Unlike mutexes semaphores keep the order of who requested access next while in mutexes it is a free for all. There are two types of semaphores named and unnamed I will be talking about named semaphores only.

3.2.1 Creating a Semaphore

POSIX Syntax
In POSIX to create a semaphore the *sem_open()* function is used. This function either creates the semaphore with the provided details or opens an existing one. This is useful because in multiprocess programs semaphores are stored in shared memory so each process doesnt know if the semaphore has been created or not. On a successful execution of *sem_open()* it returns the address of the either new semaphore or the existing semaphore. If *sem_open()* fails to open or create a new semaphore it returns *SEM_FAILED* and errno is set accordingly. When the program is done with the semaphore the *sem_unlink* function is used which waits for all the processes that are using the semaphore to finish then deletes the semaphore.

Windows Syntax
To create a semaphore in Windows you must use the *CreateSemaphore()* function. This functions is very similar to the POSIX version but adds an extra variable that allows he programmer to set a max value for the semaphore to reach. Similar the POSIX version the *CreateSemaphore()* function will either create a semaphore or open an existing one and return the handle to it.

3.2.2 Using a Semaphore

POSIX Syntax
POSIX uses two functions to manipulate semaphores the *sem_wait()* function and the *sem_post()* function. *sem_wait()* increments the semaphores value while *sem_post()* decrements the value.

Windows Syntax
Windows has multiple functions that can perform the same task as the POSIX *sem_wait()* function. To increment a semaphore in a Windows program both of the following functions will work *SignalObjectAndWait()* and *WaitForSingleObject()*. Decrementing a semaphore in Widows the *ReleaseSemaphore()* function is used.

4 Pipes

Pipes are one of the oldest form of IPC and are still used in abundance in today programming. Pipes allow direct connection to other process to send and receive information. Pipe can be used to send the output data from one process to the input of an other process. Pipes are unidirectional meaning that data can only flow through them one direction requiring 2 pipes for read and write support. Pipes are byte steams which relate to plumbing pipes in the real world, there are no boundary between the data it all flows through the pipe and the reader on the other end chooses how much to read at a time. Pipes do have a limited capacity and will block the write process until there is room to store more data in the pipe. One of pipes great features is that the data written to a pipe that is within the pipes buffer is guaranteed to be atomic. Atomic data ensures that the data has not been jumbled up or separated in any way so the reader can receive the data with confidence.

POSIX Syntax
One of POSIX's advantages is that it thinks of everything as a file, this includes files, sockets, devices, STDIN, STDOUT, and pipes. This gives uniform implementation to the programmers when writing POSIX programs. To create a pipe in POSIX the *pipe()* function is used. The *pipe()* call in POSIX Requires an two dimensional array of integers to be passed in where both the file descriptors for the read and write ends of the pipe created will be stored. The [0]th element of the integer array is the read file descriptor and the [1]st element is the write file descriptor. This is analogous to the STDIN and STDOUT file descriptors which make it easier to remember which file descriptor does what.

Windows Syntax
Windows has two different types of pipes named pipes and anonymous pipes. Anonymous pipes are easier to implement because they require less overhead so I will be talking solely about anonymous pipes in this comparison. Windows anonymous pipes only work locally and cannot be used for communication over a network. Creating a pipe in Windows is very similar to POSIX, Windows uses the *CreatePipe()* function which also returns two handles like the POSIX *pipe()* does. Since the Windows API uses pipes as files similar to POSIX does, the same file I/O functions I described in section 2 work for reading and writing to and from pipes in Windows programs.

Sources

Signals

The Linux Programming Interface - Chapter 20(Signals)
Unix to Windows Porting Dictionary for HPC
The GNU C Library
MSDN - Signal, Message Functions

File I/O

The Linux Programming Interface - Chapter 4(File I/O: The Universal I/O Model)
Linux Programmer's Manual
MSDN - CreateFile function, ReadFile function, WriteFile function, CloseHandle function

IPC

Linux Programmer's Manual
The Linux Programming Interface

- Chapter 30(Threads: Thread Synchronization)
- Chapter 53(POSIX Semaphores)

Pipes

The Linux Programming Interface
MSDN - Pipes