David Merrick

CS 311

15 March, 2013

Writeup for Final

1. *File I/O*

In this course, one of the central themes was the universality of I/O in the POSIX API. This means that, in essence, "everything is a file." Files, pipes, buffers, devices, sockets, processes, shared memory, etc. are abstracted to files, sometimes appear in the file system (via virtual filesystems that contain data structures like /proc), and can be referred to by file descriptors. The same four system calls–open(), read(), write(), and close()–can all be used to perform I/O on all types of files, including devices such as terminals.

STDIN, STDOUT, STDERR

In POSIX, every process that runs from the shell opens three standard file descriptors. These are 0 for STDIN, 1 for STDOUT, and 2 for STDERR. The Windows API works similarly. When a program begins execution, it opens standard input, standard output, and standard error. Windows even uses the same file descriptors as POSIX does for these streams.

Open():

POSIX:

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

In the POSIX API, the open() system call returns a file descriptor to a file that can then be read from and/or written to, depending on the flags passed into the function. If the file doesn't exist and O_CREAT is specified, it creates the file and then returns the descriptor. The open() function accepts a const char *pathname, flags, and the access mode. The pathname is the file to be opened, the flags argument is a bitmask that specifies the access mode of the file (read, create, etc.), and the mode is the file permissions to open it with. On success, this function returns a file descriptor referring to the open file. On error, it returns -1 and sets the errno variable to indicate what went wrong.

Windows:

```
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
);
```

In Windows, the CreateFile() system call performs similar functionality to POSIX's open(). It accepts a filename, access mode (read, write, both, or neither), a sharing mode (the level of access that other processes have to the file–read, write, none), a pointer to a security attributes structure (that determines whether or not child processes can inherit the file descriptor), a creation disposition (what action to take on a file or device that exists or doesn't exist), flags and attributes (which specify whether the file is hidden, encrypted, readonly, etc.), and an optional file descriptor of a template file (which supplies file attributes for the file that is being created). On success, the function returns a file handle, which is analogous a file descriptor in POSIX.

Close():

POSIX:

int close(int fd);

In POSIX, the close() system call frees an open file descriptor. It accepts only an integer for the file descriptor. It returns 0 on success, or -1 on error, in which case errno is also set appropriately. When a process terminates, the POSIX kernel automatically closes all the associated file descriptors.

Windows:

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

The Windows kernel equivalent of close() is CloseHandle(). This function accepts a handle object. A handle can refer to a pipe, mutex, process, semaphore, thread, file, device, etc. This construct is similar to the way that POSIX refers to everything as a file. The function returns a nonzero value (TRUE) on success, and zero (FALSE) on failure.

Read():

POSIX:

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

In POSIX, the read() call reads data from an open file descriptor. It accepts an integer for the file descriptor, a character pointer for the buffer to read data into, and size_t type to specify how many characters to read. This function returns the number of bytes actually read, 0 on EOF, or -1 on error, in which case errno is set appropriately.

Windows:

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

In Windows, a similar function to POSIX's read() is ReadFile(). This function accepts a handle (The handle must have been created with read access to the resource), a pointer to a buffer to read the data into, a number of bytes to read, an optional pointer to a variable that receives the number of bytes read, and an optional pointer to an OVERLAPPED structure. This function returns a nonzero (TRUE) return value on success, otherwise it returns 0 (FALSE). The error can be obtained with the GetLastError() function.

Write():

POSIX:

ssize_t write(int fd, void *buffer, size_t count);

In POSIX, the write() system call prints data to an open file referred to by a file descriptor. This function is very similar to read(); it accepts an integer file descriptor, a pointer to a buffer containing the characters to be written, and a size_t variable with the number of bytes to be written. On success, it returns the number of bytes written (which may be less than the specified number of bytes to be written), and -1 on error, in which case errno is set appropriately.

Windows:

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

In Windows, a similar function to POSIX's write() is WriteFile(). This function accepts a handle (This handle must have been created with write access to the resource), a pointer to a buffer containing the data to be written, the number of bytes to write, an optional pointer to a variable that receives the number of bytes written, and an optional pointer to an OVERLAPPED structure.

Error checking:

In Windows, the last-error code value is maintained on a per-thread basis. The GetLastError() function returns the value of this error. Error codes are 32-bit values that are little-endian.

In POSIX, most system calls return -1 on error, and set the global variable errno according to what type of error occurred. This variable is an integer.

Summary:

In POSIX, file I/O can be performed by open(), close(), read(), and write(). Analogous functionality can be achieved in the Windows API with CreateFile(), CloseHandle(), ReadFile(), and WriteFile().

In terms of the universality of I/O, POSIX and Windows both refer to files, sockets, pipes, etc with file handles/descriptors. Where they differ is that POSIX actually has virtual filesystems mounted in the filesystem tree that give access to data structures. These virtual filesystems include /dev/shm to access shared memory objects and /proc. Windows, at least as far as I could find, does not have an analogous feature to virtual filesystems for performing I/O.

2. *Signals*

Handling Signals:

POSIX uses a wide variety of signals, which are software interrupts that indicate certain events. In contrast, Windows supports only a small set of signals. These include SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM. As a substitute for signals, Windows uses Messages or Event Objects to notify processes of the occurrence of events.

Windows and POSIX both handle SIGINT (the interrupt signal) very similarly. Windows even has a signal.h library with a signal() system call that is very similar to POSIX. Signal() is used to assign a handler to a specific signal. If the signal to handle is SIGINT, a POSIX program can be written in almost exactly the same syntax as a Windows program.

Any similarity, however, ends with the limited support Windows has for signals. If, for example, you wanted to write a program that would trigger on a timer elapsing, the way that you would go about this differs greatly between Windows and POSIX. In POSIX, you would define a signal handler function alarm_function(), use signal(SIGALRM, alarm_function) to assign this function to be the handler for SIGALRM, spawn a child process that sends the alarm signal to the parent, then the parent would execute alarm_function(). In Windows, there is no SIGALRM. The analogous program in the Windows API would first define a VOID CALLBACK function alarm_function(), would set a 5-second timer in the main() function with SetTimer(0, 0, 5000, (TIMERPROC)alarm_function), then would simply continue executing intil it was interrupted by the timer elapsing. In Windows, there is no need to create a separate process or thread for synchronization to invoke the callback function; that function is invoked in the same thread that calls SetTimer.

Sending Signals:

For the sake of argument, imagine that we wished to prematurely trigger the alarm_function() in both of the previously-discussed example programs. In POSIX, we would use the kill() function to send the SIGALRM signal to the process, which would then trigger the alarm_function. In Windows, the analogous call would be PostMessage(WM_TIMER). This uses a Windows Message to trigger the callback function (alarm_function in this case).

Summary:

POSIX uses signals extensively for software interrupts. Windows does not natively support most POSIX-style signals, but similar functionality can be achieved using Windows Messages and Event Objects.

3. *Processes*

Creating Processes:

POSIX:

```
pid_t fork(void);
int execl(const char path, const char arg, ...);
int execlp(const char file, const char arg, ...);
int execle(const char path, const char arg, ..., char const envp[]);
int execv(const char path, char const argv[]);
int execvp(const char file, char const argv[]);
```

In POSIX, the fork() system call is used to create a copy of a process. This copy can then be replaced by a new process using the exec() system call. Fork() returns 0 in the child, the process ID in the parent, and -1 on error, in which case errno is set appropriately. In POSIX, this child process is an exact copy of the parent process for all intents and purposes. After the child process is created, the parent has no access to it unless a pipe or other form of IPC has previously been established.

The exec() family of functions replaces the current process image with a new process image. Most of the variations of exec() are variadic functions (meaning that they accept a variable number of arguments) that accept a string referring to the path of the process to be executed and a variable number arguments to pass to those processes at runtime. The exec() functions don't return unless an error occurred, in which case they will return -1.

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
);
```

Windows is very different from POSIX with respect to creating processes. In Windows, processes are created in a single step with the CreateProcess() function. CreateProcess() creates an operating environment for the child process. This environment includes the working directory, environment variables, and command-line arguments. Unlike in POSIX, where the parent has no way of communicating with the child by default, the CreateProcess function returns a handle which allows the parent application to perform operations on the child process and its environment during execution. Also unlike in POSIX, the child process created is not an exact copy of the parent; it is a brand-new process. The process to be executed also has to be specified explicitly in the CreateProcess() function.

The CreateProcess() function accepts 10 different parameters. The first is optional and is the name of the process to be executed, the second is also optional and is the command line to be executed, the third and fourth parameters are also optional and are the security attributes that determine whether the returned handle to the child or thread can be inherited by that child or thread, the fifth is a boolean value that determines if the child process can inherit the parent process's file descriptors, the sixth is creation flags that control the priority and creation of the process, the seventh is an optional environment specifier if the caller wants the process to inherit a different environment from the calling process, the eighth is an optional argument that specifies the current directory a process will start with, the ninth is a

pointer to a STARTUPINFO structure, the tenth is a pointer to a PROCESS_INFORMATION structure that receives identification information about the new process.

Sharing Memory between Processes:

In POSIX, processes can share memory mapped under the /dev/shm virtual filesystem. To set up shared memory, the memory object must first be opened with shm_open(path, flags) function, then truncated, then mapped with mmap() and MAP_SHARED specified. Multiple processes can then write to and read from this shared memory file using a file descriptor. The parent can map this memory with a file descriptor to it, and when it forks, the children will all inherit this file descriptor.

In contrast, in Windows, all processes start execution by default without inheriting the file descriptors of the parent. This presents a problem when the caller wants the processes to share memory. Windows solves this with named shared memory objects. These can be instantiated with the CreateFileMapping() function. Similarly to POSIX, this function returns a handle to a shared memory object specified by a name. Other processes can share access to the same file by using a shared file mapping object or creating a separate object backed by the same file.

Managing Processes:

The Windows API allows processes to be grouped together using Jobs to simplify management. A similar construct in POSIX is process groups. Processes can be assigned to groups with the setpgid() function. Assigning a process to a group allows signals to be sent to the entire group simultaneously.

Zombie Processes:

In Windows, a process can be killed with the TerminateProcess() function. This immediately stops any user-mode parts of the program from executing. As long as any handles to the process or drivers used by the process are still open, however, the kernel-mode process object will remain open. If a thread of the process was in the middle of an I/O operation, for instance, the kernel signals to the driver responsible for the I/O to cancel the operation. By default, when a parent process dies in Windows, the child processes are totally separate and keep running. This is different from in POSIX where, if a process dies, its child processes are adopted by the init process, which then calls wait() on them. In Windows, if a Job object is created, and the child processes are assigned to this object along with the parent, the children will terminate when the parent dies. There is not really an analogous construct to zombie processes in Windows because processes in Windows do not inherit in the same way as they do in POSIX. Each child process, by default, operates independently of its parent unless otherwise specified, and even then the processes are not related in the same way as they are in POSIX.

Summary:

The POSIX and Windows API differ greatly with respect to process creation. In particular, spawning children is completely different. Windows creates children in a single step, and they do not by default inherit from their parent like they do in POSIX. However, by specifiying certain arguments in the CreateProcess() function to set up the same process environment and pass the file handles to the child, similar functionality to POSIX's fork() function can be achieved by CreateProcess(). Processes can share memory in Windows in a similar fashion to the way they do in POSIX. Process management is similar in the two kernels.

4. *Pipes*

Like POSIX, Windows supports several forms of interprocess communication, including pipes, sockets, and shared memory. Much of the functionality is similar in Windows and POSIX pipes.

POSIX:

int pipe(int pipefd[2]);

To open a pipe in POSIX, you must first create an integer array of size 2, then pass a pointer to this array into the pipe() function. Index 0 of this array will contain the file descriptor for the read end of the pipe, and index 1 will

contain the write end. The pipe's read and write ends can then be read from and written to using the read() and write() functions, respectively.

Windows:

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

In Windows, the CreatePipe() function accepts four parameters. The first is a pointer to a variable for the read handle of the pipe, the second is a pointer to a variable for the write handle of the pipe, the third is an optional specifier for pipe attributes (whether the handles can be inherited, etc), and the size of the pipe buffer, in bytes. The pipe's read and write ends can then be read from and written to using the ReadFile() and WriteFile() functions, respectively.

Summary:

Working with pipes in POSIX and Windows is nearly identical. In both kernels, pipes have different handles/file descriptors for their read and write ends, they are treated like files, and they can even be read from and written to analogously, using read() and write() in POSIX and ReadFile() and WriteFile() in Windows.