CS311 - FA13: Final

Trevor Bramwell

December 11, 2013

Overview

This paper compares and contrasts Stream Sockets, Anonymous Pipes, and Multiprocessing, between the Windows and POSIX APIs. For each section I will provide a sample piece of code for each interface, using as many API functions as possible. I will first give an overview of what the example does, provide the example, then explain how the APIs differ within each example. When these interfaces are placed side by side, this should allow the reader to easily see the similarities and differences between them.

All POSIX examples come from the Linux Man Pages v3.54, all Windows examples come from the Microsoft Developer Network (MSDN) website. References to example are provided at the end of this document.

1 Stream Sockets

The first API I will be comparing is Stream Sockets. In Windows these are referred to as 'WinSock'. 'WinSock' has the same commands for creating and accepting connections as POSIX sockets, with the addition of *closesocket*. The difference is in Window's use of macros over *file descriptors* (fds).

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
3  #include <arpa/inet.h>
4  #include <netdb.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <string.h>
9
10 #define BUF_SIZE 500
11
12 int
13 main(int argc, char *argv[])
14 {
15     struct addrinfo hints;
16     struct addrinfo *result, *rp;
17     int sfd, s, j;
18     size_t len;
19     ssize_t nread;
20     char buf[BUF_SIZE];
21
22     if (argc < 3) {
23         fprintf(stderr, "Usage: %s host port msg...\n", argv[0]);
24         exit(EXIT_FAILURE);
25     }
26
```

```
27      /* Obtain address(es) matching host/port */
28
29      memset(&hints, 0, sizeof(struct addrinfo));
30      hints.ai_family = AF_UNSPEC;     /* Allow IPv4 or IPv6 */
31      hints.ai_socktype = SOCK_DGRAM; /* Datagram socket */
32      hints.ai_flags = 0;
33      hints.ai_protocol = 0;           /* Any protocol */
34
35      s = getaddrinfo(argv[1], argv[2], &hints, &result);
36      if (s != 0) {
37          fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
38          exit(EXIT_FAILURE);
39      }
40
41      /* getaddrinfo() returns a list of address structures.
42         Try each address until we successfully connect(2).
43         If socket(2) (or connect(2)) fails, we (close the socket
44         and) try the next address. */
45
46      for (rp = result; rp != NULL; rp = rp->ai_next) {
47          sfd = socket(rp->ai_family, rp->ai_socktype,
48                       rp->ai_protocol);
49          if (sfd == -1)
50              continue;
51
52          if (connect(sfd, rp->ai_addr, rp->ai_addrlen) != -1)
53              break;                   /* Success */
54
55          close(sfd);
56      }
57
58      if (rp == NULL) {                /* No address succeeded */
59          fprintf(stderr, "Could not connect\n");
60          exit(EXIT_FAILURE);
61      }
62
63      freeaddrinfo(result);            /* No longer needed */
64
65      /* Send remaining command-line arguments as separate
66         datagrams, and read responses from server */
67
68      for (j = 3; j < argc; j++) {
69          len = strlen(argv[j]) + 1;
70                  /* +1 for terminating null byte */
71
72          if (len + 1 > BUF_SIZE) {
73              fprintf(stderr,
74                      "Ignoring long message in argument %d\n", j);
75              continue;
76          }
77
78          if (write(sfd, argv[j], len) != len) {
79              fprintf(stderr, "partial/failed write\n");
80              exit(EXIT_FAILURE);
81          }
82
83          nread = read(sfd, buf, BUF_SIZE);
84          if (nread == -1) {
85              perror("read");
86              exit(EXIT_FAILURE);
87          }
88
89          printf("Received %ld bytes: %s\n", (long) nread, buf);
90      }
91
```

```
92      exit(EXIT_SUCCESS);
93  }
```

posix_sockets_client.c

```
1   #define WIN32_LEAN_AND_MEAN
2
3   #include <windows.h>
4   #include <winsock2.h>
5   #include <ws2tcpip.h>
6   #include <stdlib.h>
7   #include <stdio.h>
8
9
10  // Need to link with Ws2_32.lib, Mswsock.lib, and Advapi32.lib
11  #pragma comment (lib, "Ws2_32.lib")
12  #pragma comment (lib, "Mswsock.lib")
13  #pragma comment (lib, "AdvApi32.lib")
14
15
16  #define DEFAULT_BUFLEN 512
17  #define DEFAULT_PORT "27015"
18
19  int __cdecl main(int argc, char **argv)
20  {
21      WSADATA wsaData;
22      SOCKET ConnectSocket = INVALID_SOCKET;
23      struct addrinfo *result = NULL,
24                      *ptr = NULL,
25                      hints;
26      char *sendbuf = "this is a test";
27      char recvbuf[DEFAULT_BUFLEN];
28      int iResult;
29      int recvbuflen = DEFAULT_BUFLEN;
30
31      // Validate the parameters
32      if (argc != 2) {
33          printf("usage: %s server-name\n", argv[0]);
34          return 1;
35      }
36
37      // Initialize Winsock
38      iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
39      if (iResult != 0) {
40          printf("WSAStartup failed with error: %d\n", iResult);
41          return 1;
42      }
43
44      ZeroMemory( &hints, sizeof(hints) );
45      hints.ai_family = AF_UNSPEC;
46      hints.ai_socktype = SOCK_STREAM;
47      hints.ai_protocol = IPPROTO_TCP;
48
49      // Resolve the server address and port
50      iResult = getaddrinfo(argv[1], DEFAULT_PORT, &hints, &result);
51      if ( iResult != 0 ) {
52          printf("getaddrinfo failed with error: %d\n", iResult);
53          WSACleanup();
54          return 1;
55      }
56
57      // Attempt to connect to an address until one succeeds
58      for(ptr=result; ptr != NULL ;ptr=ptr->ai_next) {
59
60          // Create a SOCKET for connecting to server
61          ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
```

```
62              ptr->ai_protocol);
63          if (ConnectSocket == INVALID_SOCKET) {
64              printf("socket failed with error: %ld\n", WSAGetLastError());
65              WSACleanup();
66              return 1;
67          }
68
69          // Connect to server.
70          iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
71          if (iResult == SOCKET_ERROR) {
72              closesocket(ConnectSocket);
73              ConnectSocket = INVALID_SOCKET;
74              continue;
75          }
76          break;
77      }
78
79      freeaddrinfo(result);
80
81      if (ConnectSocket == INVALID_SOCKET) {
82          printf("Unable to connect to server!\n");
83          WSACleanup();
84          return 1;
85      }
86
87      // Send an initial buffer
88      iResult = send( ConnectSocket, sendbuf, (int)strlen(sendbuf), 0 );
89      if (iResult == SOCKET_ERROR) {
90          printf("send failed with error: %d\n", WSAGetLastError());
91          closesocket(ConnectSocket);
92          WSACleanup();
93          return 1;
94      }
95
96      printf("Bytes Sent: %ld\n", iResult);
97
98      // shutdown the connection since no more data will be sent
99      iResult = shutdown(ConnectSocket, SD_SEND);
100     if (iResult == SOCKET_ERROR) {
101         printf("shutdown failed with error: %d\n", WSAGetLastError());
102         closesocket(ConnectSocket);
103         WSACleanup();
104         return 1;
105     }
106
107     // Receive until the peer closes the connection
108     do {
109
110         iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
111         if ( iResult > 0 )
112             printf("Bytes received: %d\n", iResult);
113         else if ( iResult == 0 )
114             printf("Connection closed\n");
115         else
116             printf("recv failed with error: %d\n", WSAGetLastError());
117
118     } while( iResult > 0 );
119
120     // cleanup
121     closesocket(ConnectSocket);
122     WSACleanup();
123
124     return 0;
125 }
```

win32_sockets_client.c

From the examples you can see that both use a *struct addrinfo*, make the same calls to *getaddrinfo*, and free the struct with *freeaddrinfo*.

Both calls to sockets take the same arguments, with the difference that Windows returns a SOCKET type instead of an file descriptor. This is due to Windows using file handles, instead of descriptors.

Windows defines some extra symbolic constants for sockets, like SOCKET_ERROR, which is similar to the return value of '-1' in POSIX.

Instead of using *recv* and *send* the POSIX example uses *read* and *write* because they are the same call when no flags are passed. In Windows, because it uses file handles, the *send* and *recv* commands take SOCKET as their first argument instead of an integer.

2 Anonymous Pipes

The second API I will be comparing is Pipes. Pipes work similarly in POSIX in Windows. They have a designated read and write end, and errors occur when reading or writing to the wrong end. The ends are treated as files, so the file read and write commands work on them.The major difference is with their initialization.

Here is a program using POSIX pipes that creates a pipe and a child process, and passes command line arguments through the pipe, which the child in turn write to standard out.

The Windows program is different in which it have the child open a file, and write the file pipe, which the parent reads and then writes to standard out.

```
1  #include <sys/types.h>
2  #include <sys/wait.h>
3  #include <sys/types.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <unistd.h>
7  #include <string.h>
8
9  int
10 main(int argc, char *argv[])
11 {
12     int pipefd[2];
13     pid_t cpid;
14     char buf;
15
16     if (argc != 2) {
17      fprintf(stderr, "Usage: %s <string>\n", argv[0]);
18      exit(EXIT_FAILURE);
19     }
20
21     if (pipe(pipefd) == -1) {
22         perror("pipe");
23         exit(EXIT_FAILURE);
24     }
25
26     cpid = fork();
27     if (cpid == -1) {
28         perror("fork");
29         exit(EXIT_FAILURE);
30     }
31
32     if (cpid == 0) {    /* Child reads from pipe */
33         close(pipefd[1]);          /* Close unused write end */
34
```

```
35          while (read(pipefd[0], &buf, 1) > 0)
36              write(STDOUT_FILENO, &buf, 1);
37
38          write(STDOUT_FILENO, "\n", 1);
39          close(pipefd[0]);
40          _exit(EXIT_SUCCESS);
41
42      } else {            /* Parent writes argv[1] to pipe */
43          close(pipefd[0]);          /* Close unused read end */
44          write(pipefd[1], argv[1], strlen(argv[1]));
45          close(pipefd[1]);          /* Reader will see EOF */
46          wait(NULL);                /* Wait for child */
47          exit(EXIT_SUCCESS);
48      }
49  }
```

posix_pipes.c

```
1   #include <windows.h>
2   #include <tchar.h>
3   #include <stdio.h>
4   #include <strsafe.h>
5
6   #define BUFSIZE 4096
7
8   HANDLE g_hChildStd_IN_Rd = NULL;
9   HANDLE g_hChildStd_IN_Wr = NULL;
10  HANDLE g_hChildStd_OUT_Rd = NULL;
11  HANDLE g_hChildStd_OUT_Wr = NULL;
12
13  HANDLE g_hInputFile = NULL;
14
15  void CreateChildProcess(void);
16  void WriteToPipe(void);
17  void ReadFromPipe(void);
18  void ErrorExit(PTSTR);
19
20  int _tmain(int argc, TCHAR *argv[])
21  {
22     SECURITY_ATTRIBUTES saAttr;
23
24     printf("\n->Start of parent execution.\n");
25
26  // Set the bInheritHandle flag so pipe handles are inherited.
27
28     saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
29     saAttr.bInheritHandle = TRUE;
30     saAttr.lpSecurityDescriptor = NULL;
31
32  // Create a pipe for the child process's STDOUT.
33
34     if ( ! CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0) )
35        ErrorExit(TEXT("StdoutRd CreatePipe"));
36
37  // Ensure the read handle to the pipe for STDOUT is not inherited.
38
39     if ( ! SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0) )
40        ErrorExit(TEXT("Stdout SetHandleInformation"));
41
42  // Create a pipe for the child process's STDIN.
43
44     if (! CreatePipe(&g_hChildStd_IN_Rd, &g_hChildStd_IN_Wr, &saAttr, 0))
45        ErrorExit(TEXT("Stdin CreatePipe"));
46
47  // Ensure the write handle to the pipe for STDIN is not inherited.
48
```

```
49      if ( ! SetHandleInformation(g_hChildStd_IN_Wr, HANDLE_FLAG_INHERIT, 0) )
50         ErrorExit(TEXT("Stdin SetHandleInformation"));
51
52  // Create the child process.
53
54      CreateChildProcess();
55
56  // Get a handle to an input file for the parent.
57  // This example assumes a plain text file and uses string output to verify data flow.
58
59      if (argc == 1)
60         ErrorExit(TEXT("Please specify an input file.\n"));
61
62      g_hInputFile = CreateFile(
63          argv[1],
64          GENERIC_READ,
65          0,
66          NULL,
67          OPEN_EXISTING,
68          FILE_ATTRIBUTE_READONLY,
69          NULL);
70
71      if ( g_hInputFile == INVALID_HANDLE_VALUE )
72         ErrorExit(TEXT("CreateFile"));
73
74  // Write to the pipe that is the standard input for a child process.
75  // Data is written to the pipe's buffers, so it is not necessary to wait
76  // until the child process is running before writing data.
77
78      WriteToPipe();
79      printf( "\n->Contents of %s written to child STDIN pipe.\n", argv[1]);
80
81  // Read from pipe that is the standard output for child process.
82
83      printf( "\n->Contents of child process STDOUT:\n\n", argv[1]);
84      ReadFromPipe();
85
86      printf("\n->End of parent execution.\n");
87
88  // The remaining open handles are cleaned up when this process terminates.
89  // To avoid resource leaks in a larger application, close handles explicitly.
90
91      return 0;
92  }
93
94  void CreateChildProcess()
95  // Create a child process that uses the previously created pipes for STDIN and STDOUT.
96  {
97     TCHAR szCmdline[]=TEXT("child");
98     PROCESS_INFORMATION piProcInfo;
99     STARTUPINFO siStartInfo;
100    BOOL bSuccess = FALSE;
101
102 // Set up members of the PROCESS_INFORMATION structure.
103
104    ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION) );
105
106 // Set up members of the STARTUPINFO structure.
107 // This structure specifies the STDIN and STDOUT handles for redirection.
108
109    ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
110    siStartInfo.cb = sizeof(STARTUPINFO);
111    siStartInfo.hStdError = g_hChildStd_OUT_Wr;
112    siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
113    siStartInfo.hStdInput = g_hChildStd_IN_Rd;
```

```
114     siStartInfo.dwFlags |= STARTF_USESTDHANDLES;
115
116 // Create the child process.
117
118     bSuccess = CreateProcess(NULL,
119        szCmdline,     // command line
120        NULL,          // process security attributes
121        NULL,          // primary thread security attributes
122        TRUE,          // handles are inherited
123        0,             // creation flags
124        NULL,          // use parent's environment
125        NULL,          // use parent's current directory
126        &siStartInfo,  // STARTUPINFO pointer
127        &piProcInfo);  // receives PROCESS_INFORMATION
128
129     // If an error occurs, exit the application.
130     if ( ! bSuccess )
131        ErrorExit(TEXT("CreateProcess"));
132     else
133     {
134        // Close handles to the child process and its primary thread.
135        // Some applications might keep these handles to monitor the status
136        // of the child process, for example.
137
138        CloseHandle(piProcInfo.hProcess);
139        CloseHandle(piProcInfo.hThread);
140     }
141 }
142
143 void WriteToPipe(void)
144
145 // Read from a file and write its contents to the pipe for the child's STDIN.
146 // Stop when there is no more data.
147 {
148    DWORD dwRead, dwWritten;
149    CHAR chBuf[BUFSIZE];
150    BOOL bSuccess = FALSE;
151
152     for (;;)
153     {
154        bSuccess = ReadFile(g_hInputFile, chBuf, BUFSIZE, &dwRead, NULL);
155        if ( ! bSuccess || dwRead == 0 ) break;
156
157        bSuccess = WriteFile(g_hChildStd_IN_Wr, chBuf, dwRead, &dwWritten, NULL);
158        if ( ! bSuccess ) break;
159     }
160
161 // Close the pipe handle so the child process stops reading.
162
163     if ( ! CloseHandle(g_hChildStd_IN_Wr) )
164        ErrorExit(TEXT("StdInWr CloseHandle"));
165 }
166
167 void ReadFromPipe(void)
168
169 // Read output from the child process's pipe for STDOUT
170 // and write to the parent process's pipe for STDOUT.
171 // Stop when there is no more data.
172 {
173    DWORD dwRead, dwWritten;
174    CHAR chBuf[BUFSIZE];
175    BOOL bSuccess = FALSE;
176    HANDLE hParentStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
177
178     for (;;)
```

```
179     {
180         bSuccess = ReadFile( g_hChildStd_OUT_Rd, chBuf, BUFSIZE, &dwRead, NULL);
181         if( ! bSuccess || dwRead == 0 ) break;
182 
183         bSuccess = WriteFile(hParentStdOut, chBuf,
184                              dwRead, &dwWritten, NULL);
185         if (! bSuccess ) break;
186     }
187 }
188 
189 void ErrorExit(PTSTR lpszFunction)
190 
191 // Format a readable error message, display a message box,
192 // and exit from the application.
193 {
194     LPVOID lpMsgBuf;
195     LPVOID lpDisplayBuf;
196     DWORD dw = GetLastError();
197 
198     FormatMessage(
199         FORMAT_MESSAGE_ALLOCATE_BUFFER |
200         FORMAT_MESSAGE_FROM_SYSTEM |
201         FORMAT_MESSAGE_IGNORE_INSERTS,
202         NULL,
203         dw,
204         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
205         (LPTSTR) &lpMsgBuf,
206         0, NULL );
207 
208     lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
209         (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(TCHAR));
210     StringCchPrintf((LPTSTR)lpDisplayBuf,
211         LocalSize(lpDisplayBuf) / sizeof(TCHAR),
212         TEXT("%s failed with error %d: %s"),
213         lpszFunction, dw, lpMsgBuf);
214     MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);
215 
216     LocalFree(lpMsgBuf);
217     LocalFree(lpDisplayBuf);
218     ExitProcess(1);
219 }
```

win32_pipes.c

On Windows pipes are created using two designated file HANDLEs and a SECURITY_ATTRIBUTES, which determines whether or not children inherit file HANDLES. Whereas in POSIX a pipe is created using an integer array of size 2.

3 Multiprocessing

The third and file API I will be comparing is Multiprocessing. In POSIX multiprocessing is accomplished through *fork*, whereas Windows uses *CreateProcess*. The Windows multiprocessing interface works more like *exec* in POSIX. When *CreateProcess* is called it is pass a long list of commands that manage things like the processes file handle and environment inheritance, and the processes ability to create new threads or processes.

Both of the example follow a similar procedure: create a process, and wait for it to have a signal passed. In the POSIX example, an extra argument of the child error code can be set, and in the Windows example the name of the child program needs to be passed into the application.

```
1 #include <sys/wait.h>
2 #include <sys/types.h>
3 #include <stdlib.h>
```

```c
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    pid_t cpid, w;
    int status;

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {            /* Code executed by child */
        printf("Child PID is %ld\n", (long) getpid());
        if (argc == 1)
            pause();                    /* Wait for signals */
        _exit(atoi(argv[1]));

    } else {                    /* Code executed by parent */
        do {
            w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
            if (w == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }

            if (WIFEXITED(status)) {
                printf("exited, status=%d\n", WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf("killed by signal %d\n", WTERMSIG(status));
            } else if (WIFSTOPPED(status)) {
                printf("stopped by signal %d\n", WSTOPSIG(status));
            } else if (WIFCONTINUED(status)) {
                printf("continued\n");
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        exit(EXIT_SUCCESS);
    }
}
```

posix_procs.c

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

void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 )
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
```

```
22          argv[1],          // Command line
23          NULL,             // Process handle not inheritable
24          NULL,             // Thread handle not inheritable
25          FALSE,            // Set handle inheritance to FALSE
26          0,                // No creation flags
27          NULL,             // Use parent's environment block
28          NULL,             // Use parent's starting directory
29          &si,              // Pointer to STARTUPINFO structure
30          &pi )             // Pointer to PROCESS_INFORMATION structure
31      )
32      {
33          printf( "CreateProcess failed (%d).\n", GetLastError() );
34          return;
35      }
36
37      // Wait until child process exits.
38      WaitForSingleObject( pi.hProcess, INFINITE );
39
40      // Close process and thread handles.
41      CloseHandle( pi.hProcess );
42      CloseHandle( pi.hThread );
43  }
```

win32_procs.c

The major difference here is that *CreateProcess* is the closest thing Windows has to *fork*, but in reality it is more similar to *exec*. This is because the preferred way to create a 'child process' in Windows is through threads.

References

POSIX Examples all come from their respective man-pages.

3.1 Sockets

Winsock API Reference: http://msdn.microsoft.com/en-us/library/windows/desktop/ms741394(v=vs.85).aspx

Windows Client Socket Example: http://msdn.microsoft.com/en-us/library/windows/desktop/ms737591(v=vs.85).asp

3.2 Pipes

Windows Pipe Refrence: http://msdn.microsoft.com/en-us/library/windows/desktop/aa365139(v=vs.85).aspx

Windows Pipe Example: http://msdn.microsoft.com/en-us/library/windows/desktop/ms682499(v=vs.85).aspx

3.3 Multiprocessing

Window Multiprocessing API: http://msdn.microsoft.com/en-us/library/windows/desktop/ms684847(v=vs.85).aspx

Windows Multiprocessing Example: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682512(v=vs.85).