Intro. to Computer Security Project 1

Windows Socket Programming

Source Code Explanation

```
// Version of window, 0x501 means WINXP
#define _WIN32_WINNT 0x501
#include <winsock2.h>
#include <ws2tcpip.h>
#include <cstdio>
using namespace std;
```

In this section, we include the necessary header files for socket programming.
winsock2.h is for socket utilities.
ws2tcpi.h is for tcp and ip related utilities, like **getaddrinfo**, **freeaddrinfo**.
cstdio is for standard I/O

_WIN32_WINNT controls the version of header to use

```
const int DEFAULT_BUFLEN = 512; // read buffer size
const int SEC_MSG_LEN = 7; // length of message to send
const char *SEC_IP = "140.113.216.151";  // server IP
const char *SEC_PORT = "2000";  // server port
const char SEC_MSG[SEC_MSG_LEN+1] = "0000000";  // message to send
```

Here we define the constants that would be used later.

```
#pragma comment(lib,"Ws2_32.lib")
```

This makes the linker to link this program with Ws2_32.lib

```
// the socket to connect with server
SOCKET clientSocket;
```

Since the socket must be used across threads, we put it in global scope.

The following two functions will be executed by two threads.

The return value is a **DWORD** indicating whether the function succeeds. It can be obtained by **GetExitCodeThread** from the HANDLE. **lpParam** is a pointer that carries the data passed to the thread, but we don't use it in this program.

sec_send is used to send data to the server.

```
DWORD WINAPI sec_send(LPVOID lpParam)
{
    printf("Sending data to server...\n");
    int iResult;
    iResult = send(clientSocket, SEC_MSG, SEC_MSG_LEN, 0);
    if (iResult == SOCKET_ERROR) {
        printf("send failed: %d\n", WSAGetLastError());
        closesocket(clientSocket);
        WSACleanup();
        return 1;
    }
```

Here, we send the data using the socket opened earlier, and check whether it succeeded or not. If it failed, we close the socket and return with 1, which indicates error.

```
    printf("Shuting down sending connection...\n");
    iResult = shutdown(clientSocket, SD_SEND);
    if (iResult == SOCKET_ERROR) {
        printf("shutdown sending failed: %d\n", WSAGetLastError());
        closesocket(clientSocket);
        return 1;
    }
```

We then close the sending channel since we no longer need to send any data.

```
    return 0;
}
```

sec_send is used to receive data to the server.

```
DWORD WINAPI sec_recv(LPVOID lpParam)
{
    char recvbuf[DEFAULT_BUFLEN+1];

    int iResult;
    do {
        iResult = recv(clientSocket, recvbuf, DEFAULT_BUFLEN, 0);
        if (iResult > 0) {
            recvbuf[iResult] = '\0';
            printf("%d bytes received: %s\n", iResult, recvbuf);
        } else if (iResult == 0 || WSAGetLastError() == WSAECONNRESET)
            printf("Connection closed\n");
        else
            printf("recv failed: %d\n", WSAGetLastError());
    } while (iResult > 0);
```

Here, we keeping looping until the connection is closed or an error has happened.

The client would print the data it receives, since the data may not be terminated by a '\0', we add one to ensure correct output. Since the input may fill the buffer, we declare recvbuf to be of length DEFAULT_BUFLEN+1.

iResult indicates the number of bytes received, when the server closes the connection, the value would be 0. If the server dose not shutdown the socket properly before closing the connection, we may get **WSAECONNRESET** error. We also handle this case here.

```
    return 0;
}

int main()
{
    printf("Initializing Winsock...\n");
    WSADATA wsdata;
    int iResult = WSAStartup(MAKEWORD(2,2), &wsdata);
```

Here we initialize Winsock using **WSAStartup**. We request the verison 2.2, and the function would store the implementation information in wsdata. **MAKEWORD** is used to concatenate two bytes into a word.

```
if (iResult != 0) {
    printf("WSAStartup failed: %d\n", iResult);
    return 1;
}

struct addrinfo *result = NULL, *ptr = NULL, hints;
ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_INET;  // Use IPv4
hints.ai_socktype = SOCK_STREAM;  // Sequenced, reliable, two-way connection
hints.ai_protocol = IPPROTO_TCP; // Use TCP protocol
```

Here we initialize the hints structure with zero and set the appropriate values. hints would provide hints about the type of socket the caller supports in the call to **gettaddrinfo**.

```
printf("Obtaining the server address...\n");
iResult = getaddrinfo(SEC_IP, SEC_PORT, &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo failed: %d\n", iResult);
    WSACleanup();
    return 1;
}
```

Here we use **gettaddrinfo** to get the address information of the server. This function can be used to translate a host name into IP address, but since we already have the IP address, we simply pass the address to the function.

```
printf("Creating a SOCKET for connection with server...\n");
clientSocket = INVALID_SOCKET;
ptr = result;
clientSocket = socket(ptr->ai_family, ptr->ai_socktype,
        ptr->ai_protocol);
```

Here we use **socket** to ask the OS to create the socket for us. The parameters stand for address family (IPv4), socket type, and protocol being used, respectively.

```
if (clientSocket == INVALID_SOCKET) {
    printf("cannot open socket: %ld\n", WSAGetLastError());
    freeaddrinfo(result);
    WSACleanup();
```

```
        return 1;
    }

    printf("Connecting to the server...\n");
    iResult = connect(clientSocket, ptr->ai_addr,
            (int)ptr->ai_addrlen);
```

Here we ask the OS to actually connect to the server with the socket.

```
    if (iResult == SOCKET_ERROR) {
        closesocket(clientSocket);
        clientSocket = INVALID_SOCKET;
    }

    freeaddrinfo(result);
    ptr = NULL;
```

Here we free the **addrinfo** object created by **getaddrinfo** since it is no longer used.

```
    if (clientSocket == INVALID_SOCKET) {
        printf("Unable to connect to the server!\n");
        WSACleanup();
        return 1;
    }

    printf("Creating threads...\n");
    HANDLE threads[2];
    threads[0] = CreateThread(NULL, 0, sec_send, NULL, 0, NULL);
    threads[1] = CreateThread(NULL, 0, sec_recv, NULL, 0, NULL);
```

Here we create two threads to execute **sec_send** and **sec_recv** function.

```
    WaitForMultipleObjects(2, threads, true, INFINITE);
```

Let this thread wait infinitely long for the two threads to complete execution.

```
    printf("Closing...\n");
    closesocket(clientSocket);
    WSACleanup();
    return 0;
}
```

Introduction to Winsock

"A socket is an endpoint of an inter-process communication flow." -- Wikipedia

Processes can use sockets to communicate with each other. A socket can be used to send and receive messages. Basically, programs request the OS to create sockets for them, and the socket is controlled by the OS. The socket between two processes may be used to communicate across the network, or it may be used under the same host.

In this project, we use Winsock to create TCP socket with the server. Winsock is an API that defines how a program under Windows can access network service.

A TCP socket is identified by four-tuple: source IP, source port, destination IP and destination port. When a client tries to connect with a server, the OS would assign a port for the source socket. We use **getaddrinfo()** to get the destination IP and port information.

In this project, several functions are used:

int **WSAStartup**(WORD wVersionRequested, LPWSADATA lpWSAData); int **WSACleanup**(void);	**WSAStartup** is used to initialize the use of the Winsock DLL by a process, while **WSACleanup** terminates the use of it. Notice that **WSACleanup** ends the Winsock service of all threads in the same process.
SOCKET **socket**(int af, int type, int protocol); int **closesocket**(SOCKET s);	**socket** is used to create a socket while **closesocket** is used to close it. A process should create a socket before it can connect with others.
int **connect**(SOCKET s, const struct sockaddr *name, int namelen); int **shutdown**(SOCKET s, int how);	**connect** is used to actually connect the socket with remote host. **shutdown** is used to disable send or receive functionality so as to ensure all data are received before disconnection.
int **send**(SOCKET s, const char *buf, int len, int flags); int **recv**(SOCKET s, char *buf, int len, int flags);	
int **WSAGetLastError**(void);	**WSAGetLastError** is used to get error status of the last failed operation.

Reference

Winsock Reference
http://msdn.microsoft.com/en-us/library/windows/desktop/ms741416%28v=vs.85%29.aspx

Introduction to Thread

"Multiple threads in a process are like multiple cooks reading off the same cook book and following its instructions, not necessarily from the same page." -- Wikipedia

When a program is about to execute, the OS loads the program binary into the memory. This instance of executing program is called a process. A process can have different threads of execution. Threads within the same process can share some resource like program code and memory, but each can be executing different instructions concurrently and be scheduled by OS independently. Different threads may be executing on different processes at the same time, or they may be switched back and forth by the OS, with only one thread actually executing at a given time. This is called context-switching.

In this project, we use Windows API to create threads. In particular, we have used two functions:

```
HANDLE CreateThread(_ATTRIBUTES lpThreadAttributes,
SIZE_T dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId
);
```

CreateThread can be used to create a thread, the thread will start executing at **lpStartAddress**, which is a pointer to a function. Parameter can be passed as a pointer to the value. The function returns the handle to the thread.

```
DWORD WaitForMultipleObjects(
DWORD nCount,
const HANDLE *lpHandles,
BOOL bWaitAll,
DWORD dwMilliseconds
);
```

WaitForMultipleObjects can be used to block the current thread until one or all of the specified objects are in the signaled state. In this project, it is used to wait for send/recv threads to terminate.

Reference

Process and Thread Functions
http://msdn.microsoft.com/en-us/library/windows/desktop/ms684847%28v=vs.85%29.aspx