Aufgabe 1

Erklären Sie kurz die Begriffe „Handle“ und „Prozess/Thread“ im Kontext von Windows 2000.

Im MSDN-Glossary findet sich zum Thema „Handle“ die Erklärung (englisch):

1. A pointer to a pointer; that is, a variable that contains the address of another variable, which in turn contains the address of the desired object. In certain operating systems, the handle points to a pointer stored in a fixed location in memory, whereas that pointer points to a movable block. If applications start from the handle whenever they access the block, the operating system can perform memory management tasks such as garbage collection without affecting the applications.
2. Any token that an application can use to identify and access an object such as a device, a file, a window, or a dialog box.
3. One of several small squares displayed around a graphical object in a drawing application. The user can move or reshape the object by clicking on a handle and dragging.

Jede einzelne Definition enthält ein paar wichtige Begriffe, aber eigentlich steht nur in 2. die wahre Kernbedeutung: ein Handle ist ein Verweis auf ein Betriebssystemobjekt. Dabei gibt es verschiedenste Kategorien, von I/O über Speichermanagement bis hin zur graphischen Oberfläche.

Bezüglich „Prozess“ äußert sich die gleiche Quelle:
A running application that consists of a private virtual address space, code, data, and other operating-system resources, such as files, pipes, and synchronization objects that are visible to the process. A process also contains one or more threads that run in the context of the process.

Und „Thread“:
A process that is part of a larger process or application. A thread can execute any part of an application's code, including code that is currently being executed by another thread. All threads share the virtual address space, global variables, and operating-system resources of their respective processes.

Diese beiden Definitionen sind treffend und kurz, daher will ich sie in ihrer ganzen Schönheit stehen lassen.

Aufgabe 2

Machen Sie sich mit dem Microsoft Developer Network (MSDN) vertraut. Lesen Sie die Online-Dokumentation der folgenden Funktionen der Win32-API:

- `CreateFile, ReadFile, WriteFile, CloseFile`
- `GetStdHandle, GetLastError, CloseHandle`

Die erste Gruppe von Befehlen erlaubt den Zugriff auf Dateien. Diese umfassen unter Windows 2000:

- Konsolen
- Kommunikationsmittel
- Verzeichnisse (nur lesend)
- Datenträger
- Dateien
- Mailslots
- Pipes

Die Identifizierung erfolgt stets über Handles. Ich konnte die Funktion `CloseFile` nicht ausfindig machen, die MSDN empfiehlt für das Schließen einer Datei den Befehl `CloseHandle`.

`GetStdHandle` liefert ein Handle für das Standardeingabe-, Standardausgabe- oder Standardfehler-Gerät, was in der Regel die Tastatur und der Monitor sind.
Über `GetLastError` kann der letzte Fehler des jeweiligen Threads erfragt werden. Dieser Befehl findet z.B. Anwendung in Aufgabe 4.

Aufgabe 3

`CreateProcess` erlaubt die Erzeugung eines neuen Prozesses in einem eigenem Prozessraum. Dabei sind extrem viele Parameter einstellbar, von der Kommandozeile über Sicherheitseinstellungen bis hin zu Prioritätsstufen. Der UNIX-nahe Befehl `execv` (mit seinen Derivaten) führt den neuen Prozess im Speicher des aufrufenden Prozesses auf. Bei erfolgreichem Aufruf wird deshalb in diesem kleinen Programm nichts auf dem Bildschirm ausgegeben:

```
_execv(„CoolProgram.exe“, „-?“);
cout << „Launched CoolProgram“ << endl;
```

Um dieses Verhalten zu umgehen und `CreateProcess` zu emulieren, wird in UNIX der Befehl `fork` eingesetzt. Er kopiert einen Prozess, der Aufrufer erhält dabei die PID des Kindes, das Kind hingegen `NULL` zurückgeliefert. Im Kind kann dann `exec` ausgeführt werden ohne dass der Vaterprozess davon berührt wird.

Aufgabe 4

Sowohl das Auslesen der Datei als auch das Abfangen eventuell auftretender Fehler ist nicht weiter aufwändig und kann sehr schnell implementiert werden. Etwas komplizierter ist die Bedeutung der Parameter von `CreateProcess`, da sich die MSDN sehr geschickt weigert, ein erklärendes Beispiel zu zeigen.

Die allgemeine Syntax von `CreateProcess` lautet:

```
BOOL CreateProcess(LPCTSTR lpApplicationName,
                   LPTSTR  lpCommandLine,
                   LPSECURITY_ATTRIBUTES lpProcessAttributes,
                   LPSECURITY_ATTRIBUTES lpThreadAttributes,
                   BOOL    bInheritHandles,
                   DWORD   dwCreationFlags,
                   LPVOID  lpEnvironment,
                   LPCTSTR lpCurrentDirectory,
                   LPSTARTUPINFO         lpStartupInfo,
                   LPPROCESS_INFORMATION lpProcessInformation);
```

wobei der Rückgabewert `FALSE` (= 0) einen Fehler symbolisiert.

Eine Eigenschaft dieses Befehls ist, dass man `lpApplicationName` auf `NULL` setzen kann, wenn der Name des zu startenden Programms zu Beginn von `lpCommandLine` steht. Somit kann ich mir das Parsen der Shell-Datei stark vereinfachen und jede Zeile direkt als `lpCommandLine` deuten.

Ich fordere keine besonderen Sicherheitseinstellungen für die aufgerufenen Programme, sie sollen die für die Shell geltenden Rechte und Einschränkungen haben. Für den Code bedeutet das, dass `lpProcessAttributes` und `lpThreadAttributes` auf `NULL` gesetzt werden.

Die Handles sollen nicht vererbt werden (`bInheritHandles` auf `FALSE`), sowohl die Priorität als auch die Umgebungsvariablen entsprechen den Standardwerten (`dwCreationFlags` auf 0, `lpEnvironment` auf `NULL`). Das Verzeichnis, in dem die Prozesse ausgeführt werden, entspricht dem aktuellen Verzeichnis (`lpCurrentDirectory` ist ebenfalls `NULL`).

Da ich keine weiteren speziellen Einstellungen fordere, ist `lpStartupInfo` mit Nullen gefüllt, lediglich das Feld `cb` enthält die Größe der `StartupInfo`-Struktur in Bytes. `lpProcessInformation` benötigt auch keine Einstellungen, ebenfalls sind alle Werte mit Null überschrieben worden.

Schlussendlich steht folgender Befehl in meinem Programm:

```
CreateProcess(NULL,     // application name is first token of the command line
              cmdLine,  // contains application name, too
              NULL,     // default process security settings
              NULL,     // default thread security settings
              FALSE,    // process DOES NOT inherit from minshell.exe
              NULL,     // child has no access to minshell.exe
              NULL,     // no special environment
              NULL,     // same directory
              &StartupInfo,
              &ProcessInfo);
```

Bei erfolgreicher Ausführung von `CreateProcess` könnte die Shell sofort weiterarbeiten. Laut Aufgabenstellung soll aber solange gewartet werden, bis der Kindprozess beendet wurde. Ich erreiche dies, indem ich den Handle des Kindprozesses überwache und solange warte, bis er freigegeben wurde:

```
// wait until process terminates and clean up
WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
```

Quelltext

```
//////////////////////////////////////////////////////////
// Betriebssysteme I - Windows 2000
// Lab 2: A simple command shell
//
// author:          Stephan Brumme
// last changes:    November 20, 2001


// Win32 API
#include <windows.h>

// console output (printf etc.)
#include <stdio.h>


int main(int argc, char* argv[])
{
    // file handle to access the shell commands
    FILE* fd;

    // at most 512 characters per line
    const int MAX_LINE_LEN = 512;
    char  cmdLine[MAX_LINE_LEN+1];


    // a shell command file must be supplied
    if (argc > 2)
    {
        printf("usage: \"MinShell ShellCommands.txt\"\n");
        printf("or just \"MinShell\" if you want to type in the commands.\n");

        // wrong parameter count
        return 1;
    }

    if (argc == 2)
    {
        // open file
        fd = fopen(argv[1], "r");
        if (fd == NULL)
        {
            printf("Cannot find %s.\n", argv[1]);
            // failed
            return 2;
        }
    }
    else
    {
        // read from console
        fd = stdin;
        printf("Press [Return] to quit.\n\n");
    }


    // read and execute shell commands
    for (;;)
    {
        // prompt
        printf("MinShell$ ");

        // read a single command, abort if [Return] was pressed
        if (fgets(cmdLine, MAX_LINE_LEN, fd) == NULL ||
            cmdLine[0] == 10)
        {
            printf("\n... done\n");
            return 0;
        }
```

```
        // strip CR-LF
        char* pCmdLine = cmdLine;
        while(*pCmdLine != 0x0A &&
              *pCmdLine != 0x0D &&
              *pCmdLine != 0x00)
            pCmdLine++;
        *pCmdLine = 0x00;

        // show shell command (if not typed in from console)
        if (fd != stdin)
            printf("%s\n", cmdLine);

        // fill in start-up info values
        STARTUPINFO StartupInfo;
        ZeroMemory(&StartupInfo, sizeof(StartupInfo));
        StartupInfo.cb = sizeof(StartupInfo);

        // CreateProcess returns some process information
        PROCESS_INFORMATION ProcessInfo;
        ZeroMemory(&ProcessInfo, sizeof(ProcessInfo));

        if (CreateProcess(NULL, // application name is first token of the command line
                    cmdLine,  // contains application name, too
                    NULL,     // default process security settings
                    NULL,     // default thread security settings
                    FALSE,    // process DOES NOT inherit from minshell.exe
                    0,        // child has no access to minshell.exe
                    NULL,     // no special environment
                    NULL,     // same directory
                    &StartupInfo,
                    &ProcessInfo) == 0)
        {
            // error handling
            const int MAX_ERROR_LEN = 255;
            char strError[MAX_ERROR_LEN+1];

            DWORD nError = GetLastError();
            FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
                          NULL,
                          nError,
                          0,
                          strError,
                          MAX_ERROR_LEN,
                          NULL);

            printf("Failed to create \"%s\": (%d) %s\n", cmdLine, nError, strError);

            // failed to execute a program
            // return 3;
        }


        // wait until process terminates and clean up
        WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
        CloseHandle(ProcessInfo.hProcess);
        CloseHandle(ProcessInfo.hThread);
    }

    // successful
    return 0;
}
```