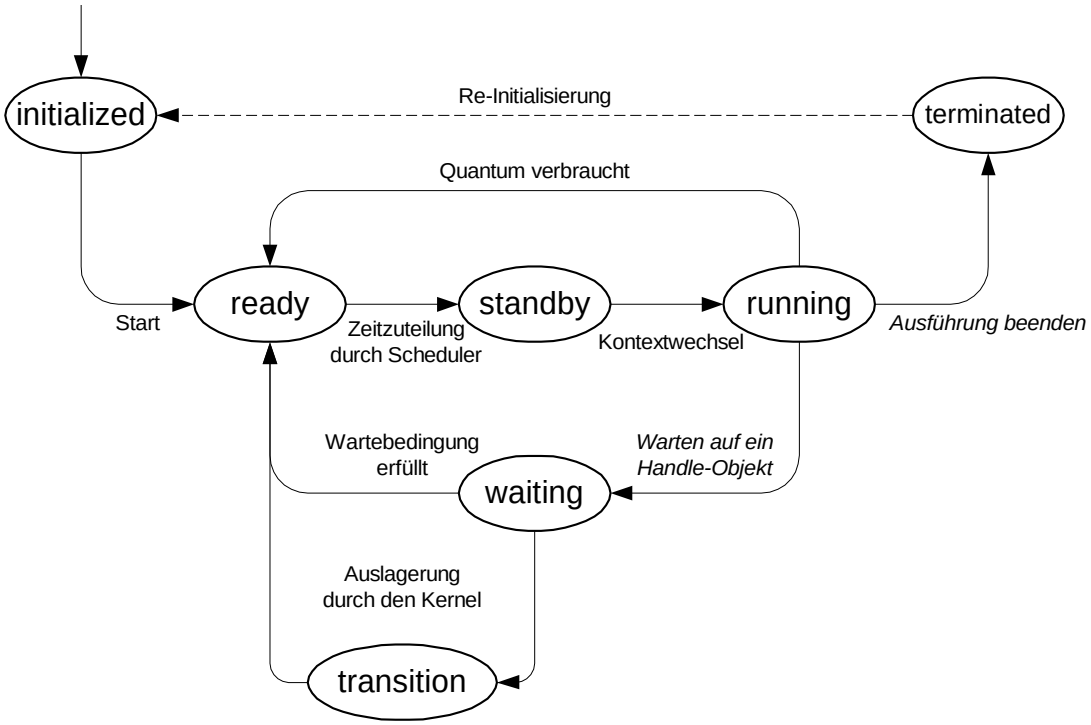
Aufgabe 1: Thread Scheduling unter Windows 2000

Erklären Sie den Ansatz zum Thread-Scheduling von Windows 2000 ! Was ist die kleinste vom Betriebssystem verwaltete Scheduling-Einheit ? In welchen Scheduling-Zuständen kann sich ein Thread befinden ? Durch welche Entscheidungen und Ereignisse geschehen Zustandsübergänge ?

Unter Windows 2000 wird ein Prozess nur als „Hülle“ für einen oder mehrere Threads betrachtet (siehe Übungsblatt 2). Die nächstkleinere Einheit, Fibers genannt, müssen vom erzeugenden Programm selbst verwaltet werden. Das Betriebssystem koordiniert die Rechenzeitzuteilung auf Basis der einzelnen Thread-Prioritäten.

Die einzelnen Zustände und die entsprechenden Zustandsübergänge eines Threads habe ich in der nachfolgenden Grafik dargestellt (kursive Übergänge werden durch den Thread hervorgerufen):



Erklären Sie die Begriffe Quantum, Prioritätsschub *und* Prioritätsumkehr *!*

Ein *Quantum* stellt die Dauer einer Zeitscheibe dar, unter Windows 2000 Professional sind dies 2 Ticks, was etwa 20 ms entspricht. Nach Ablauf dieser Zeit wird das Scheduling wieder aktiv. Das Betriebssystem kann, ja nach Bedarf, die Dauer auch ändern.
Um auf einem System zu gewährleisten, dass niederpriore Threads die Chance haben, trotz laufender hochpriorer Threads auch ausgeführt werden zu können (Verhinderung von Starvation), wird ihre Priorität kurzfristig angehoben, was man als *Prioritätsschub* bezeichnet.
Die *Prioritätsumkehr* bezeichnet den Effekt, dass ein hochpriorer Threads auf ein Betriebsmittel wartet, das ein niederpriorer Thread besitzt. Da letzterer selten oder gar nicht zur Ausführung kommt (der hochpriore Thread belegt den Prozessor), kann er das Betriebsmittel nicht freigeben. Als Folge dieser Situation verlangsamt sich das System und kommt schließlich zu einem Stillstand.

Aufgabe 2: Thread-Erzeugung und –Synchronisation

Machen Sie sich mit den Begriffen und Konzepten der Thread-Erzeugung und –Synchronisation der Win32-API vertraut !

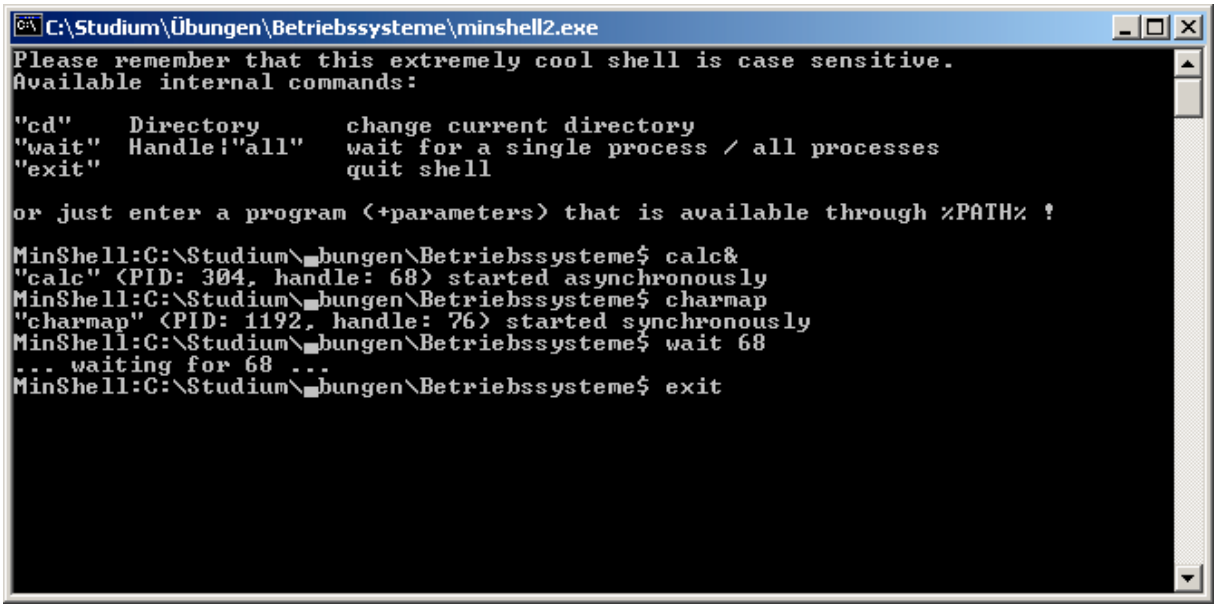
(siehe Aufgabe 3)

Aufgabe 3: Erweiterung der Kommandoshell

Modifizieren Sie ihren Kommandointerpreter `min_shell` *aus Aufgabe 4, Übungsblatt 2, derart, dass es die asynchrone Ausführung erlaubt. Ein Kommando soll dann asynchron ablaufen, wenn es mit einem & abgeschlossen wird.*
Erweitern Sie ihre Shell um die internen Befehle

- `cd` *Wechseln des aktuellen Verzeichnisses*
- `exit` *Beenden der Shell*
- `wait` *Warten auf das Ende eines oder aller asynchroner Prozesse*

Zwar wurde im letzten Übungsblatt bereits eine Shell programmiert, ich habe jedoch so gut wie alles neu geschrieben. Der Grund liegt darin, dass ich bisher recht viele Standard-C-Routinen (z.B. `fopen`) benutzte und nun explizit die Win32-API (z.B. `CreateFile`) verwende. Der Lohn dafür ist eine nur 5120 Bytes grosse Exe-Datei, d.h. eine Reduzierung der Programmgröße auf weniger als ein Drittel bei gleichzeitig verdoppelter Funktionalität. Ich kenne kein Unix-System, dass derartige Leistungen aus einem C++ -Compiler holt ☺



Die Abstraktion der Eingabe als Datei funktioniert wieder erstaunlich gut, die Win32-Befehle sind:

```
// file handle to access the shell commands
 HANDLE hFile;

 if (argc == 2)
 {
     // open file (read shared !)
     hFile = CreateFile(argv[1], GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL, NULL);
     if (hFile == INVALID_HANDLE_VALUE)
     {
         printf("Cannot find %s.\n", argv[1]);
         // failed
         return 2;
     }
 }
 else
     // read from console
     hFile = GetStdHandle(STD_INPUT_HANDLE);
```

Alle nachfolgenden Zugriffe auf `hFile` brauchen keine Unterscheidung mehr zwischen Tastatur oder tatsächlicher Datei zu treffen.

Ich verwalte die Handles der asynchronen Prozesse in einem Feld namens `arHandles`. Wenn der Benutzer das Kommando `wait` verwendet, dann wird mit Hilfe von

```
// wait for process
DWORD dwResult = WaitForSingleObject(hProcess, INFINITE);
```

solange gewartet, bis das Betriebssystem den jeweiligen Handle freigibt, d.h. der Prozess beendet wurde. Es besteht die Möglichkeit, `wait` mit einem Handle als Dezimalzahl aufzurufen und so auf einen Prozess oder mit `all` auf alle asynchronen Prozesse zu warten. Das Feld `arHandles` wird dann über die selbstgeschriebenen Funktionen `AddHandle` bzw. `RemoveHandle` aktualisiert.

Einfach und unkompliziert gestaltet sich der Verzeichniswechsel, hierzu reicht:

```
BOOL ChangeDirectory(const char* dir)
{
    return SetCurrentDirectory(dir);
}
```

Wie man im Screenshot auf der vorherigen Seite erkennen kann, gibt die Shell am Prompt auch immer den aktuellen Pfad aus, so dass man jederzeit die aktuelle Position im Verzeichnisbaum erkennen kann.

Die Erzeugung eines neuen Prozesses übernimmt `NewProcess`, die dabei verwendeten Parameter von `CreateProcess` habe ich bereits im letzten Übungsblatt erklärt.

Der komplette Quelltext des Programms befindet sich im Anhang.

Aufgabe 4: Konzepte der Speicherverwaltung

Machen Sie sich mit den Konzepten der Speicherverwaltung unter Windows 2000 vertraut. Beantworten Sie dazu die folgenden Fragen:
Was ist der working set *eines Prozesses ?*
Was ist ein page fault *(Seitenfehler) ? Wo liegt der Unterschied zwischen* soft *und* hard page faults *?*
Beschreiben Sie den Lebenszyklus einer (virtuellen) Speicherseite unter Windows 2000. Erklären Sie die Dynamik der Seitenlisten.

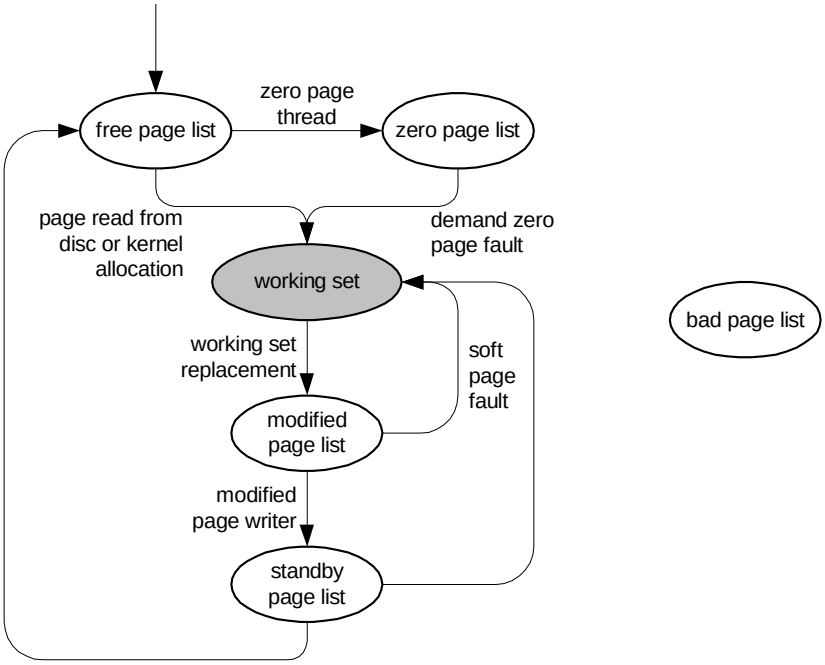
Der *working set* eines Prozesses umfasst die Seiten seines virtuellen Speichers, auf die ohne Seitenfehler zugegriffen werden kann. In der Praxis heißt das, dass der working set alle nicht-ausgelagerten Seiten enthält, was meist den Seiten entspricht, auf die er in letzter Zeit zugriff.

Befindet sich eine Speicherseite, auf die zugegriffen wird, nicht im working set des Prozesses, so tritt ein Seitenfehler (*page fault*) auf. Bei *hard page fault* bedeutet dies, dass von einem externen Medium nachgeladen werden muss, während bei *soft page faults* die betreffende Seite noch im Hauptspeicher befindlich ist. Meist tritt dieser Fall beim Zugriff mehrerer Prozesse auf die gleiche Seite auf.

Zu Beginn werden die Speicherseiten in der Free Page List verwaltet. Periodisch oder auf Anfrage überschreibt der Zero Page Thread den Speicher mit Nullen, um zu verhindern, dass Daten ungewollt für andere Prozesse sichtbar sind. Er legt diese bereinigten Seiten in der Zero Page List ab.

Fordert ein Prozess neuen Speicher an, so werden diese der zero page list oder der free page list entnommen. Wenn der working set verringert werden muss, so gehen benutzte Seiten in die modified page list über. Sobald ihr Inhalt in die Auslagerungsdatei geschrieben werden konnte, werden sie der standby page list zugeordnet (sie bleiben aber noch als Kopie im Speicher). Sowohl aus der modified als auch der standby page list können über ein soft page fault die Seiten wieder dem working set zugeordnet werden. Erst wenn zusätzlicher Speicher angefordert werden soll und keine freien Seiten mehr existieren, werden aus standby page list Seiten in die Free Page List verschoben.

Neben all diesen Listen existiert noch eine bad page list für fehlerhafte Seiten.



Quelltext

Die Shell aus Aufgabe 3 besteht aus nur einer Datei:

MinShell2.cpp:

```
//////////////////////////////////////////////////////////
// Betriebssysteme I - Windows 2000
// Assignment 3.3: An extended command shell
//
// author:          Stephan Brumme
// last changes:    November 27, 2001

#include <windows.h>
#include <stdio.h>


//////////////////////////////////////////////////////////////////
// read a single command
BOOL ReadLine(const HANDLE hFile, char* buffer)
{
    // FALSE, if error
    BOOL bRead;
    // determines end of line
    BOOL bCRLF = FALSE;
    // number of read bytes per read access (only 0 or 1 !)
    unsigned long nRead  = 0;
    // total amount of read bytes
    unsigned long nIndex = 0;

    // read in command byte by byte
    do
    {
        // read a single byte
        bRead = ReadFile(hFile, &buffer[nIndex], 1, &nRead, NULL);
        // strip EOL
        if (buffer[nIndex] == 0x0D)
            nIndex--;
        if (buffer[nIndex] == 0x0A)
            bCRLF = TRUE;

        // next byte
        nIndex++;
    }
    while (nRead == 1 && bCRLF == FALSE);

    // terminate string, we moved one too far
    buffer[nIndex-1] = 0;

    // return last status
    return bRead;
}


//////////////////////////////////////////////////////////////////
// parse a string and looks for a given command
// return a pointer to the parameter list
// or NULL if command not found
char* SearchCommand(const char* command, const char* text)
{
    // look for command
    char* pReturn = strstr(text, command);
    // must be at the beginning of the line
    if (pReturn != text)
        return NULL;

    // jump to parameter list
    pReturn += strlen(command);

    // remove whitespaces
    while (*pReturn == ' ')
        pReturn++;
```

```
    return pReturn;
}


//////////////////////////////////////////////////////////////////////
const MAX_HANDLES = 255;
HANDLE arHandles[MAX_HANDLES];

//////////////////////////////////////////////////////////////////////
// add handle to internal list
// no checking !
void AddHandle(HANDLE handle)
{
    for (int i=0; i<MAX_HANDLES; i++)
        if (arHandles[i] == 0)
        {
            arHandles[i] = handle;
            break;
        }
}


//////////////////////////////////////////////////////////////////////
// remove handle from internal list
// no checking !
void RemoveHandle(HANDLE handle)
{
    for (int i=0; i<MAX_HANDLES; i++)
        if (arHandles[i] == handle)
        {
            arHandles[i] = 0;
            break;
        }
}


//////////////////////////////////////////////////////////////////////
// change directory
// return TRUE if successful
BOOL ChangeDirectory(const char* dir)
{
    return SetCurrentDirectory(dir);
}


//////////////////////////////////////////////////////////////////////
// wait for a single process
// return TRUE if successful
BOOL Wait(HANDLE hProcess)
{
    printf("... waiting for %d ...\n", hProcess);

    // wait for Godot
    DWORD dwResult = WaitForSingleObject(hProcess, INFINITE);
    CloseHandle(hProcess);

    // remove handle from list
    RemoveHandle(hProcess);

    return (dwResult != WAIT_FAILED);
}


//////////////////////////////////////////////////////////////////////
// create a new process
BOOL NewProcess(char* name, BOOL bWait = TRUE)
{
    // 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
                      name,     // 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", name, nError, strError);

        return FALSE;
    }

    printf("\"%s\" (PID: %d, handle: %d) started ", name, ProcessInfo.dwProcessId,
                                                         ProcessInfo.hProcess);

    // wait until process terminates and clean up
    if (bWait == TRUE)
    {
        printf("synchronously\n");

        // wait
        WaitForSingleObject(ProcessInfo.hProcess, INFINITE);

        CloseHandle(ProcessInfo.hProcess);
        CloseHandle(ProcessInfo.hThread);
    }
    else
    {
        printf("asynchronously\n");
        CloseHandle(ProcessInfo.hThread);

        // store handle
        AddHandle(ProcessInfo.hProcess);
    }

    return TRUE;
}


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// here we go !

int main(int argc, char* argv[])
{
    // 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;
    }
    printf("Please remember that this extremely cool shell is case sensitive.\n"    \
           "Available internal commands:\n\n"                                       \
           "\"cd\"    Directory      change current directory\n"                    \
```

```
            "\"wait\"  Handle|\"all\"   wait for a single process / all processes\n" \
            "\"exit\"                   quit shell\n\n"                               \
            "or just enter a program (+parameters) that is available through %%PATH%% !\n\n");

    // file handle to access the shell commands
    HANDLE hFile;

    if (argc == 2)
    {
        // open file (read shared !)
        hFile = CreateFile(argv[1], GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
        if (hFile == INVALID_HANDLE_VALUE)
        {
            printf("Cannot find %s.\n", argv[1]);
            // failed
            return 2;
        }
    }
    else
        // read from console
        hFile = GetStdHandle(STD_INPUT_HANDLE);

    // clear HANDLE list
    ZeroMemory(&arHandles, sizeof arHandles);


    //
    // DA MAIN LOOP
    //

    // read and execute shell commands
    for (;;)
    {
        // at most 512 characters per line
        const int MAX_LINE_LENGTH = 512;
        char  cmdLine[MAX_LINE_LENGTH+1];

        // prompt
        printf("MinShell:");
        // show current directoy
        static char strDirectory[MAX_LINE_LENGTH];
        if (GetCurrentDirectory(sizeof strDirectory, strDirectory))
            printf(strDirectory);
        printf("$ ");


        // read the command
        BOOL bDone = ReadLine(hFile, cmdLine);

        // error or exit ?
        if (cmdLine[0] == 0 || SearchCommand("exit", cmdLine))
        {
            printf("\n... winke, winke, Lala !\n");
            return 0;
        }

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

        //
        // now the internal commands
        // "exit" was already handled
        //

        // "cd [directory]"
        if (char* directory = SearchCommand("cd", cmdLine))
        {
            if (!ChangeDirectory(directory))
                printf("Unable to change directory to \"%s\".\n", directory);
            // read next command
            continue;
        }

        // "wait"
```

```
        if (char* strHandle = SearchCommand("wait", cmdLine))
        {
            // wait for all ?
            if (strstr(strHandle, "all"))
            {
                // take a look at each handle
                for (int i=0; i<MAX_HANDLES; i++)
                    if (arHandles[i] != 0)
                        // wait and remove from list
                        Wait(arHandles[i]);
            }
            else
            {
                // get numerical value
                HANDLE dwHandle = (HANDLE) atoi(strHandle);
                // wait and wait and wait ...
                if (!Wait(dwHandle))
                    printf("Unable to use handle %d (converted from %s).\n", dwHandle,
strHandle);
            }

            // read next command
            continue;
        }


        //
        // no internal command found ? let‘s try to launch new process !
        //
        if (cmdLine[strlen(cmdLine)-1] == ‘&‘)
        {
            // remove ampersand
            cmdLine[strlen(cmdLine)-1] = 0;
            // launch synchronously
            NewProcess(cmdLine, FALSE);
        }
        else
            // launch asynchronously
            NewProcess(cmdLine, TRUE);
    }

    CloseHandle(hFile);

    // successful
    return 0;
}
```