VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELIGENT SYSTEMS

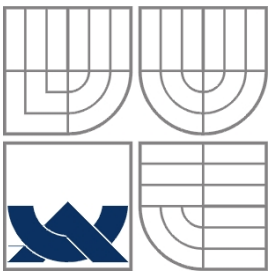
APLIKACE K MONITOROVÁNÍ UDÁLOSTÍ OS WINDOWS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

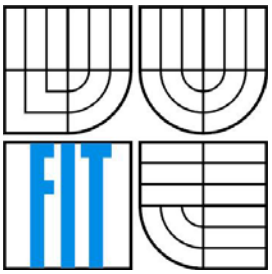
AUTOR PRÁCE
AUTHOR

MICHAL KOBLIHA

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE K MONITOROVÁNÍ UDÁLOSTÍ OS WINDOWS

APPLICATION FOR MONITORING EVENTS OF OS WINDOWS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — MICHAL KOBLIHA
AUTHOR

VEDOUCÍ PRÁCE — ING. LUKÁŠ GRULICH
SUPERVISOR

BRNO 2007

Zadání bakalářské práce/5172/2006/xkobli04

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů

Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Kobliha Michal**
Obor: Informační technologie
Téma: **Aplikace k monitorování událostí OS Windows**
Kategorie: Operační systémy

Pokyny:

1. Seznamte se s existujícími aplikacemi pro monitorování událostí OS Windows.
2. Analyzujte požadavky na takový systém (jde zejména o monitorování operací s myší a klávesnicí, důležité je následné zpracování dat)
3. Implementujte a testujte navržený systém.
4. Diskutujte možnosti dalšího vývoje.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Grulich Lukáš, Ing.**, UITS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Licenční smlouva

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Abstrakt

Práce zmiňuje jednotlivé generace operačního systému Windows a použití vybraných funkcí jeho aplikačního programového rozhraní a navazuje uvedením jazyka C jakožto nástroje vhodného pro realizaci procesu monitorování systémových událostí. Podrobně rozebírá jednotlivé fáze vývoje aplikace pro zachytávání událostí systému Windows a stručně prezentuje dosažené výsledky.

Klíčová slova

Microsoft Windows, WinAPI, Hooking, C99

Abstract

Thesis mentions individual generations of operating system Windows and usage of its chosen application environment functions, followed by presentation of C language as a tool suitable for realization of a process monitoring system events. It analyses individual phases of development of an application for capturing system events and briefly represents achieved outcomes.

Keywords

Microsoft Windows, WinAPI, Hooking, C99

Citace

Michal Kobliha: Aplikace k monitorování událostí OS Windows, bakalářská práce, Brno, FIT VUT v Brně, 2007

Aplikace k monitorování událostí OS Windows

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Lukáše Grulicha
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Michal Kobliha
14.5.2007

Poděkování

Děkuji Ing. Lukáši Grulichovi za hodnotné rady a odborné vedení během mé práce.

Obsah

Úvod

V bakalářské práci se zabývám vývojem aplikace pro monitorování událostí systému Microsoft Windows a popisem nástrojů spojených s jejím vývojem.

Ve zkratce nastiňuji vývoj produktu Microsoft Windows a principy tvorby Win32 aplikací s využitím WinAPI (Windows Application Programming Interface) v programovacím jazyce C. V dalších kapitolách se snažím srozumitelným způsobem ozřejmit problematiku hookingu a jednotlivé nástroje a metody, které se při zachytávání událostí používají. Rovněž stručně shrnuji vhodnost a použitelnost těchto metod v praxi.

Druhá polovina mé práce se zaměřuje na problematiku návrhu a vývoje aplikace k monitorování událostí EventMon. Podrobněji se zabývám rozložením celkového návrhu na jednotlivé podproblémy, jejich analýzu a implementaci vlastního řešení. Závěrem práce shrnuji dosažené výsledky a prezentuji navrženou aplikaci.

1 Microsoft Windows

1.1 Vývoj systému Microsoft Windows

Windows byl světu poprvé představen světu v roce 1983 jakožto *rozšiřující grafická nadstavba operačního systému MS-DOS* (Microsoft Disk Operating System) s uživatelsky velmi intuitivním ovládáním pomocí polohovacího zařízení zvaného myš. Na maloobchodní trh se Windows dostaly poprvé až v roce 1985.

O dva roky později byla na trh uvedena nová verze *Windows 2.0*, která běží ve 32 bitovém režimu a v rozšířeném (enhanced) módu, dokáže využívat více než 640 kB paměti a umožnuje spouštění aplikací od externích vývojářů. Dalším významným krokem ve vývoji se staly Windows 3.1 a jejich vylepšená verze *Windows for Workgroups 3.11* obohacené o podporu sítě a obsahující řadu dalších vylepšení částečně převzatých z vývoje *Windows NT*.

V roce 1995 došlo k zlomu ve vývoji a na trh byl uveden *Windows 95* jako samostatný systém s novým vzhledem pracovní plochy s nabídkou Start, lištou pro minimalizované aplikace, vylepšeným ovládáním. Windows 95 se vyznačoval oproti předchozím verzím zlepšením podpory 32 bitových aplikací, vylepšenou správou paměti umožňující úplné využití dostupné paměti, rozšířenými možnostmi v oblasti multimédií, her a internetových aplikací. Pro spouštění starších 16 bitových aplikací však stále využívají upravenou kopii systému MS-DOS, stejně tak jako o tři roky později představený *Windows 98*, který byl navíc obohacen o podporu USB a opravdu fungující Plug-and-Play.

Revolucí ve vývoji operačního systému Windows se stal *Windows 2000*, které převzal vzhled Windows 9x a upravené jádro systému Windows NT, ale bohužel se na trhu se softwarem příliš neujal.

Mnohem úspěšnějším se stal později vydaný *Windows XP* určený jak pro domácí použití, tak i pro nasazení ve firmách. Tento operační systém dodávaný jak pro 32bitové tak pro 64bitové architektury se vyznačuje velmi dobrou podporou hardwarových součástí a technologie *Plug-and-Play*, díky níž systém nevyžaduje restartování při každé změně hardwarového nastavení, systém dále používá plně chráněný model správy paměti schopný využívat až 4 GB operační paměti. Klíčové struktury jádra a kód ovladačů zařízení jsou v tomto systému určeny jen pro čtení a ostatní aplikace je nemohou běžně ovlivňovat.

Poslední systém *Windows Vista* je založen na stejném jádře, je však mnohem hardwarově náročnější. Oproti systému Windows XP se v této nové verzi objevilo nové grafické rozhraní s názvem *Aero*, vylepšení správy souborů s indexováním a filtry, podporu virtuálních adresářů. Dále pak stojí za zmínku *WinFX* které, nahrazuje původní WinAPI a rozšiřuje jej o zpřístupnění nových komponent systému.

1.2 Programy v prostředí Windows

1.2.1 Aplikace versus software

Aplikace neboli aplikační programové vybavení počítače určené pro přímou interakci s uživatelem na rozdíl od programového vybavení (software), který nemusí být v interakci s uživatelem by se měla vyznačovat jednoduchými a pro nezasvěceného uživatele intuitivními ovládacími prvky, zapouzdřenými v celistvé grafické nebo textové rozhraní pro interakci s uživatelem, přičemž je zřejmé, že obraz je z hlediska informace daleko výkonnější než text. Textová komunikace prostřednictvím příkazů je historicky nejstarším způsobem komunikace s počítačem, dodnes je hojně využívána, bohužel s sebou přináší spoustu omezení jako je například nutnost uživatele seznámit s jednotlivými příkazy, zdlouhavé vypisování náročnějších požadavků a jejich parametrů a omezené možnosti zobrazování výsledků.

Aplikace by měla uživatele sama vést k řešení a zpracování konkrétního problému přehledným uspořádáním a popisem jednotlivých ovládacích prvků, jakožto i zobrazením potřebných výsledků. Takto navržená aplikace povede k efektivnějšímu využití schopností a potenciálu stroje, na kterém je spuštěna.

1.2.2 Tvorba aplikací v prostředí Windows

Operační systém Windows je víceúlohový, víceuživatelský operační systém s grafickým rozhraním, který je zařazen mezi událostní, zprávami řízené operační systémy. Každá činnost v tomto systému je buď důsledkem zpracování zprávy, nebo příčinou odeslání zprávy nové, přičemž zpráva je obecně datová struktura vytvářená operačním systémem v případě, že nastane libovolná událost. Událostí se rozumí jak operace provedená operačním systémem (zpráva od časovače, zpráva ukončení aplikace), tak i operace provedená samotným uživatelem (změny na jednotlivých prvcích ovládacího rozhraní, stisky tlačítek myši nebo kláves, atd.).

Jednotlivé zprávy obsahují detailní informace o druhu a původu události. Operační systém určuje, komu budou tyto informace zasílány a jak budou dále zpracovávány. Primárně všechny zprávy odesílá ke zpracování aplikaci, ve které ke zdrojové události došlo, některé druhy zpráv jsou pak distribuovány také ostatním spuštěným aplikacím a jádru systému.

1.2.2.1 Smyčka zpráv a fronta zpráv

Mechanismus, který se v aplikaci stará o zpracování příchozích zpráv se nazývá *smyčka zpráv* a *fronta zpráv*. Zjistí-li operační systém Windows uživatelem způsobenou událost, vygeneruje

odpovídající zprávu a vloží tuto odpověď do fronty zpráv, náležející spuštěné aplikaci. Každý program ve Windows tedy obsahuje vlastní hlavní smyčku zpráv, která nepřetržitě a opakovaně ověřuje status fronty zpráv, to znamená, že zjišťuje, zda fronta neobsahuje nezpracované zprávy. Tato činnost je realizována pomocí speciálních funkcí WinAPI a aplikace tento postup opakuje až do svého ukončení.

Smyčka zpráv je obvykle cyklus, který obsahuje nejprve volání funkce, která vrátí prostřednictvím datové struktury zprávy informace o každé zprávě čekající ve frontě zpráv dané aplikace. Poté budou volány další funkce, které řídí tok zpráv aplikace. Dále dochází k převedení zpráv virtuálních kláves na znakové zprávy a nakonec dochází k volání funkce známé jako *procedura okna*, která má za úkol zpracování zpráv příslušejících danému oknu či jednotlivým prvkům uživatelského rozhraní. Cyklus bude probíhat tak dlouho, dokud neobdrží zprávu, která informuje o tom, že má být ukončena činnost programu.

1.2.2.2 Procedura okna

Procedura okna není funkcí nebo procedurou WinAPI, nýbrž programátorem aplikace vytvořená procedura. Úkolem této procedury je obsloužit všechny zprávy generované uživatelem. Každé okno či dialog aplikace ve Windows obsahuje svou vlastní *proceduru okna*. Tato funkce na vstupu přebírá ukazatel na zdrojové okno, kde k události došlo, a také typ a parametry zprávy. Tělem této funkce je rozsáhlá větvící se struktura podmíněných příkazů, přičemž jednotlivé větve provádějí specifické funkce pro obsloužení určitého typu zprávy. Na konci této funkce zpravidla bývá volání funkce zabezpečující zpracování systémových zpráv operačního systému, neobsloužených vlastní procedurou okna.

1.2.2.3 Třída okna a vytvoření okna aplikace

Při spuštění aplikace ve Windows, je první věcí, která se vykoná, *registrace třídy okna*. Každé okno aplikace musí mít svou vlastní třídu okna. Je důležité pochopit, že třída okna není totéž co třída v objektových programovacích jazycích. V terminologii Windows je to určitý typ okna, který je registrován k použití danou aplikací, fyzicky se jedná o určitou strukturu. Jednotlivé prvky této struktury slouží k nastavení jeho vlastností, jež určují jak jeho vzhled, tak i další vlastnosti využívané systémem.

Při registraci třídy okna je vráceno aplikaci unikátní identifikační číslo registrované třídy, podle kterého je možné běžící aplikaci jednoznačně identifikovat. Poté nejčastěji dochází k *vytvoření hlavního okna aplikace*.

1.2.2.4 Vstupní bod aplikace

Nyní již víme téměř vše důležité o jednotlivých částech kostry aplikace ve Windows a zbývá pouze zmínit, že při spouštění aplikace je volána funkce `WinMain`, která zapouzdřuje všechny předchozí zmiňované prvky do jednoho celku. Deklarace funkce `WinMain` typicky vypadá takto:

```
int WinMain(  HINSTANCE hInstance, HINSTANCE hPrevInstance,
              LPSTR lpCmdLine, int nCmdShow);
```

Při svém volání funkce `WinMain` přebírá několik parametrů, prostřednictvím kterých předává operační systém aplikaci informace potřebné pro její spuštění a běh jakožto handle (rukojeť) předchozí a současné instance aplikace, parametry předávané aplikaci z příkazové řádky a rozšiřující parametry, které upravují, jak bude zobrazeno hlavní okno aplikace. Při spouštění aplikace v 16bitovém prostředí vždy nejprve WinMain zkontroluje, zda již není spuštěna jiná instance téže aplikace. V 32bitovém prostředí dochází rovnou k registraci nové třídy okna, která slouží k nastavení jeho specifických vlastností. Jakmile je třída úspěšně zaregistrována, provedou se operace pro vytvoření a zobrazení okna aplikace a dojde ke spuštění smyčky zpráv. V tomto okamžiku se započne s obsluhou smyčky zpráv až do ukončení aplikace.

1.3 Rozhraní WinAPI

Jak již bylo zmíněno výše, moderní grafické aplikace komunikují s uživatelem pomocí dialogů, formulářů, tlačítek a různých dalších ovládacích prvků. Všechny tyto komponenty se v operačním systému Windows dají realizovat pomocí volání funkcí WinAPI, což je rozhraní, navržené především pro použití v programovacím jazyce C nebo objektovém C++, zpřístupňující širokou škálu jak grafických, tak systémových komponent pro aplikační využití. Toto rozhraní představuje pro aplikaci nejpřímější cestu jak pracovat v systému Windows.

1.3.1 Systémové služby WinAPI

Systémové služby rozhraní WinAPI umožňují aplikacím využívat nezbytné zdroje poskytované systémem Windows, jako například funkce souborového systému, přidělování paměti, přístupu ke vstupním a výstupním zařízením a jejich správě a v neposlední řadě také funkce správy procesů, jejich synchronizace a koordinace jednotlivých vláken spuštěných v rámci jedné aplikace.

1.3.1.1 Funkce souborového systému

Funkce souborového systému nabízí přístup nejen k souborům v adresářových strukturách na pevných discích a jiných úložných zařízeních počítače, ale i k různým vstupním a výstupním zařízením. Tyto funkce podporují širokou škálu souborových systémů, zahrnujíce FAT12, FAT16 a

FAT32 souborové systémy, CDFS systém souborů disku CD-ROM, přístup ke sdíleným prostředkům v počítačových sítích a na operačních systémech Windows s jádrem Windows NT navíc souborový systém NTFS (New Technology File System).

1.3.1.2 Funkce správy procesů a vláken

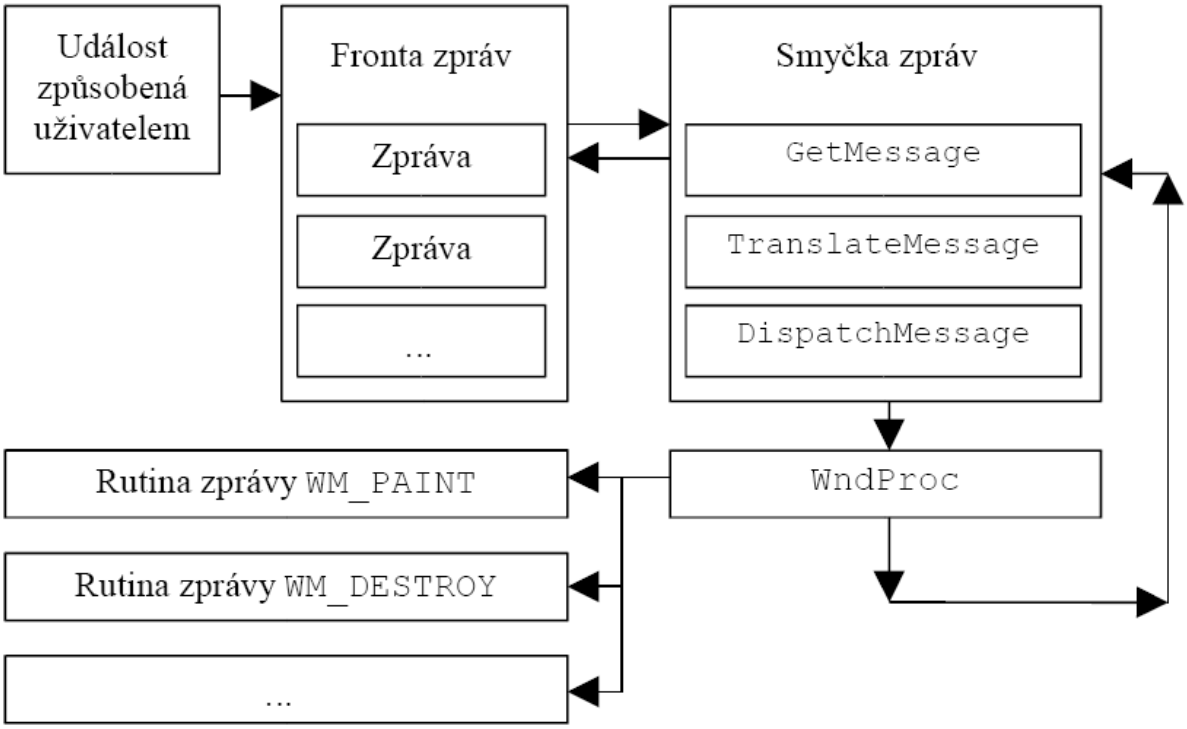
Aplikace ve Windows mohou sdílet části svého kódu nebo posílat informace jiným aplikacím.

Nejčastěji se sdílené funkce a procedury umísťují do dynamicky připojitelných knihoven DLL, které mohou být libovolnými aplikacemi nahrány za běhu do paměti a zavolána požadovaná funkce umístěná v dané knihovně.

Komunikace mezi procesy je zajišťována pomocí tzv. *pipes* (rour), jejichž jeden konec je přidružen ke standardnímu vstupu a výstupu spouštěného příkazu či procesu a druhý je předán v podobě *handle* (rukojeti) aplikaci. Existuje několik typů rour umožňujících jak jednosměrnou komunikaci, kdy je roura otevřena pomocí `popen`, tak obousměrnou komunikaci pomocí roury vytvořené funkcí `pipe`.

Mezi nejdůležitější a nejčastěji používané jsou funkce potřebné při vytváření kostry aplikace. Patří sem funkce pro vytvoření třídy okna `RegisterClass` a `RegisterClassEx`, kterým se jako parametr předává ukazatel na strukturu obsahující informace o ikoně aplikace, barvách výplně a typu kurzoru a některých dalších vlastnostech a stylech okna. Tyto vlastnosti jsou následně využívány při vytváření jednotlivých oken aplikace.

Při vytváření smyčky zpracovavání zpráv WndProc je obvykle využíváno volání funkce `GetMessage`, vracející prostřednictvím datové struktury `MSG` informace o každé zprávě čekající ve frontě zpráv dané aplikace. Po překladu zpráv virtuálních kláves na znakové zprávy pomocí `TranslateMessage` je nejčastěji volána funkce `DispatchMessage` zprostředkující předání parametrů zprávy proceduře okna, kde dochází k vlastnímu vyhodnocování zprávy. V proceduře okna nemusí být deklarovány a ošetřovány všechny typy zpráv – nejčastěji jsou zde ošetřovány zprávy WM_PAINT (zajišťující překreslování okna) WM_MOUSEMOVE, WM_xBUTTONDOWN, WM_xBUTTONUP, WMxBUTTONDBLCLK (určené ke zjišťování stavu myši či jiného polohovacího zařízení emulujícího myš), WM_KEYDOWN, WM_KEYUP, WMCHAR (ošetřující zprávy vyvolané událostmi klávesnice). Bývá rovněž dobrým zvykem ošetřit uzavření okna křížkem zpracováním zprávy WM_DESTROY. K tomuto se používá odeslání zprávy `PostQuitMessage`. Ostatní zprávy nutné ke správnému fungování aplikace lze vhodně ošetřit například voláním `DefWindowProc` na konci funkce okna.



Obrázek 1-1:Schéma zpracovávání zpráv aplikací

Pro zasílání zprávy aplikaci jsou určeny dvě funkce, PostMessage a SendMessage. Obě dvě vyžadují jako parametr okno, kam se má zpráva odeslat. Je možné zaslat zprávu všem oknům zadáním hodnoty HWND_BROADCAST místo parametru okna. Následují tři parametry identifikující typ zprávy a předávají specifické informace a hodnoty pro různé typy zpráv:

```
BOOL PostMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
BOOL SendMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Zprávy lze posílat i vláknům aplikace použitím funkcí `PostThreadMessage` a `SendThreadMessage`, s tím rozdílem, že parametrem příjemce zprávy je identifikátor vlákna `idThread`. Funkce `SendMessage` a `SendThreadMessage` lze s výhodou používat pro synchronizaci, dochází k čekání na doručení zprávy, zatímco `PostMessage` a `PostThreadMessage` nečekají a pokračuje se v provádění následujících instrukcí.

1.3.2 Funkce grafického uživatelského rozhraní

WinAPI poskytuje také mnoho funkcí pro výstup grafického obsahu na monitory, tiskárny a jiná výstupní zařízení. K tomuto účelu slouží ve Windows datová struktura *device context*. Do device contextu je možné vykreslovat jednoduché obrazce pomocí funkcí `MoveTo`, `MoveToEx`, `LineTo`, `Ellipse` a několika dalších funkcí, jako například funkce `TextOut`, díky níž je možné tyto obrazce obohatit o textový výstup.

Abychom mohli kreslit, musíme si nejprve vytvořit okno aplikace a jemu přiřadit device context. To se dá provést voláním `CreateWindow` nebo `CreateWindowEx`. Jako parametry jsou předávány požadované vlastnosti okna, třída, titulek okna, rozměry a pozice. S vytvořeným oknem se dá později různě manipulovat. Například funkcí `ShowWindow` se dá okno skrýt, zobrazit, minimalizovat a maximalizovat, pomocí funkce `MoveWindow` lze okno přemístit a roztáhnout na požadovanou velikost, funkce `SetWindowText` umožňuje změnit titulek okna a nesmíme zapomenout na `DestroyWindow`, která zničí vytvořené okno. Všechny tyto funkce vyžadují jakožto první parametr ukazatel na strukturu okna HWND.

1.3.2.1 Základní typy oken

Při tvorbě oken se můžeme setkat s několika základními typy oken. V deklaraci funkce CreateWindow k tomuto účelu slouží parametr dwStyle:

```
HWND CreateWindow( LPCTSTR lpClassName, LPCTSTR lpWindowName, DWORD
                   dwStyle, int x, int y, int nWidth, int nHeight, HWND
                   hWndParent, HMENU hMenu, HINSTANCE hInstance, LPVOID
                   lpParam);
```

Parametr dwStyle může obsahovat některý z následujících nejčastěji používaných stylů nebo jejich kombinací, pokud se jednotlivé styly navzájem nevylučují:

- **WS_OVERLAPPEDWINDOW** **vytvoří klasické okno, stejné jako při zadání kombinace WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX**
- **WS_CHILDWINDOW** **styl samostatného okna bez menubaru**
- **WS_DISABLED** **vytvoří okno bez možnosti interakce s uživatelem**
- **WS_DLGFRAME** **styl dialogového boxu**
- **WS_HSCROLL** **okno s horizontálním a vertikálním scrollbarem**
- **WS_VSCROLL**
- **WS_MAXIMIZE** **vytvoří maximalizované a minimalizované okno**
- **WS_MAXIMIZE**
- **WS_SYSMENU** **styl okna se systémovým menu**
- **WS_VISIBLE** **zviditelní okno hned po jeho vytvoření**

Pro hlavní okno aplikace je vhodný styl WS_OVERLAPPEDWINDOW. Pro jednotlivá vnořená ale i samostatná okna bez nástrojových lišt je určen styl WS_CHILDWINDOW nebo

zkráceně WS_CHILD. Pro okno které nebude přijímat žádné zprávy od uživatele (nebude reagovat na stisky kláves a myši) vyhovuje WS_DISABLED – okno se dá později „oživit“ pomocí funkce `EnableWindow`. Styl WS_VISIBLE se používá, potřebujeme-li zobrazit okno ihned po jeho vytvoření. Bez jeho zadání je možné okno později zobrazit již výše zmiňovanou funkcí `ShowWindow`.

1.3.2.2 Okno s posuvníky pomocí WinAPI

Pokud bychom chtěli zobrazit v okně více informací, než rozměry okna dovolují, je možné vyřešit takovýto problém oknem s posuvníky. Prakticky k tomu budeme potřebovat inicializovat dvě okna, jedno okno (hWnd) jako hlavní okno s posuvníky, a do něj zobrazíme část dalšího okna, které bude jeho potomkem a v němž se budou zobrazovat informace:

```
HWND hWnd = CreateWindow("Třída","Hlavní okno", WS_OVERLAPPEDWINDOW |
            WS_VSCROLL | WS_HSCROLL, 0, 0, CW_USEDEFAULT,
            CW_USEDEFAULT, (HWND) NULL, (HMENU) NULL, hInstance,
            (LPVOID) NULL);

HWND sWnd = CreateWindow("Třída","Scrollovací okno", WS_CHILD, 0, 0,
            CW_USEDEFAULT, CW_USEDEFAULT, (HWND) NULL, (HMENU) NULL,
            hInstance, (LPVOID) NULL);
```

Nyní stačí nastavit rozsahy posuvníků funkcí `SetScrollRange`, ve smyčce zpráv obsluhovat zprávy informující o posouvání horizontálního (WM_HSCROLL) a vertikálního (WM_VSCROLL) posuvníku a vhodně ošetřit posuvy o řádek (SB_LINEUP, SB_LINEDOWN, SB_LINELEFT, SB_LINERGHT) nebo stránku (SB_PAGEUP, …) voláním funkce `SetWindowPos` pro posunutí vnitřního okna (sWnd). Posouvání okna je možné realizovat také pomocí zachytávání zprávy kolečka myši WM_MOUSEWHEEL, kdy je v horním slově parametru WPARAM smyčky zachytávání událostí předávána informace, o kolik se kolečko myši otočilo. V tomto případě je potřeba ošetřit pozici posuvníku pomocí funkce `SetScrollPos`.

1.3.2.3 Minimalizace aplikace do systémové lišty

Při používání operačního systému Windows se často setkáváme s možností minimalizovat aplikace, které nevyžadují neustálou pozornost uživatele do systémové lišty. Takto „schovaná“ aplikace nezabírá zbytečný prostor a usnadňuje uživateli přepínání mezi ostatními programy. Realizovat takovéto chování aplikace je možné naplněním struktury NOTIFYICONDATA a následným zavoláním funkce ShellNotifyIcon:

```
NOTIFYICONDATA niDATA;
               niData.cbSize = sizeof(NOTIFYICONDATA);
```

```
                niData.uID = jedninečné_id;
                niData.uFlags = NIF_ICON | NIF_MESSAGE | NIF_TIP;
                niData.hIcon = LoadIcon(NULL, ikona_aplikace);
                strcpy(niData.szTip, název_aplikace);
                niData.hWnd =hlavní_okno;
                niData.uCallbackMessage =zpráva_ze_system_tray;
Shell_NotifyIcon(NIM_ADD, &niData);
```

Následnou minimalizaci aplikace je možné ošetřit zachycením zprávy WM_SYSCOMAND s parametrem SC_MINIMIZE v dolním slově WPARAM a následným schováním okna aplikace `ShowWindow(hWnd, SW_HIDE)`.

Při poklikání na ikonu aplikace v systémové liště dojde k vytvoření zprávy `zpráva_ze_system_tray`, na kterou je možné reagovat například zobrazením hlavního okna aplikace na popředí. Odstranění této ikony se provádí příkazem `Shell_NotifyIcon (NIM_DELETE, &niData)`, kde jako druhý parametr je stejně jako při vytváření ikony předáván ukazatel na její strukturu.

1.3.2.4 Dialogové boxy

Celkem častým způsobem komunikace uživatele s aplikací jsou dialogové boxy. Tato kategorie zahrnuje nejen běžné dialogové boxy obsahující text a tlačítka, ale také dialogové boxy pro otevření nebo uložení souboru, výběr fontu či barvy.

Nejčastěji se používá `MessageBox`, který se hojně využívá kvůli vysoké variabilitě a snadnosti implementace. U okna dialogového boxu je možné nastavit záhlaví (lpCaption), text uvnitř okna (lpText) a rodiče okna, v němž se okno zobrazí. Je však možno předat i hodnotu NULL a okno se zobrazí nezávisle na ostatních oknech. Syntaxe volání funkce vypadá asi následovně:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);
```

Parametr uType specifikuje, zda součástí boxu bude i ikona (MB_ICONEXCLAMATION, MB_ICONWARNING, MB_ICONINFORMATION, MB_ICONSTOP, MB_ICONASTERISK, MB_ICONQUESTION, MB_ICONHAND či MB_ICONERROR) a jaká tlačítka se v dialogovém boxu použijí (např. MB_OK, MB_OKCANCEL, MB_YESNO, MB_ABORTRETRYIGNORE, atd.). Funkce dialogového boxu vrací vždy hodnotu podle tlačítka, které uživatel stisknul (IDOK, IDCONTINUE, IDYES, IDNO, atd.). Dojde-li k uzavření okna křížkem, vrací stejnou hodnotu, jako by bylo stisknuto tlačítko „Storno“, tedy ID_CANCEL. Tento dialogový box je vskutku možné použít téměř na cokoli od potvrzování provedení určitého procesu až po jednoduché informativní zprávy.

Dalším velmi potřebným dialogovým boxem je dialogový box pro otevření souboru. Pro jeho použití je však potřeba naplnit strukturu OPENFILENAME. V praxi stačí nastavit položky

`lpstrFilter`, `nFilterIndex` a `Flags` pro nastavení filtrování přípony a vlastností souboru. Položka `lpstrFile` slouží k předání názvu souboru, měla by tedy ukazovat na řetězcovou proměnnou a `nMaxFile` určuje její maximální délku. Nastavení okna rodiče je možné pomocí `lpstrTitle`. Velmi podobný je i dialogový box pro uložení souboru, který využívá jako parametr stejnou strukturu. Prototypy deklarací funkcí vypadají následovně:

```
BOOL GetOpenFileName (LPOPENFILENAME lpofn);
BOOL GetSaveFileName (LPOPENFILENAME lpofn);
```

V případě, že je dialogové okno uzavřeno křížkem nebo tlačítkem storno, vrací funkce hodnotu FALSE. Rozšířený kód určující chybu nebo jiné specifikum uzavření dialogu je možné získat funkcí `CommDlgExtendedError`. V opačném případě je navráceno TRUE a v řetězci, na který ukazovala položka `lpstrFile` se objeví název souboru zvolený v dialogovém boxu.

Dialog pro výběr fontu opět vyžaduje předání struktury jakožto parametru. K tomuto Účelu slouží struktura CHOOSEFONT, tentokráte však není nutné nastavovat mnoho položek, stačí nastavit `nFontType` určující zda se jedná o výběr tiskárnového (PRINTER_FONTTYPE) nebo klasického fontu (SCREEN_FONTTYPE). Někdy bývá vhodné omezit minimální a maximální velikost `nSizeMin` a `nSizeMax`. Tyto úpravy však vyžadují i nastavení položky `Flags` na CF_LIMITSIZE. Deklarace funkce `ChooseFont` je obdobná jako u dialogových boxů pro výběr souboru, zvolený font je předáván v položce `lpLogFont` typu `LOGFONT`.

Posledním užitečným dialogovým boxem s podobným chováním je ChooseColor, dialog pro výběr barvy. V datové struktuře CHOOSECOLOR vrací zvolenou barvu v položce `rgbResult`.

1.3.2.5 Tlačítka a ostatní ovládací prvky

V grafických prostředích se velmi často používají rozmanité ovládací prvky v podobě tlačítek, přepínačů a jiných elementů. Většina těchto komponent se realizuje voláním WinAPI funkce CreateWindow, pouze se jako identifikátor třídy předává speciální identifikátor třídy dané komponenty. V praxi to znamená, že volání této funkce pro vytvoření tlačítka bude vypadat následovně:

```
HWND CreateWindow ("BUTTON", LPCTSTR Name, DWORD dwStyle, int x, int y,
                   int w, int h, HWND hWndParent, HMENU hMenu, HINSTANCE
                   hInst, LPVOID lpParam);
```

Při vytváření ovládacích prvků zastávají jednotlivé parametry stejnou funkci jako při vytváření oken, dokonce můžeme použít také styly WS_DISABLED a WS_VISIBLE deklarované původně pro

využití při vytváření oken pomocí stejné funkce. Další volby vlastností komponenty lze dosáhnout doplněním některého z následujících stylů do parametru dwStyle:

- **BS_AUTO3STATE** -automatický 3stavový checkbox
- **BS_AUTOCHECKBOX** -automatický checkbox
- **BS_AUTORADIOBUTTON** -automatické přepínací tlačítko
- **BS_CHECKBOX** -obyčejný checkbox
- **BS_PUSHBUTTON** -klasické tlačítko
- **BS_DEFPUSHBUTTON** -tlačítko s hrubším okrajem
- **BS_FLAT** -ploché tlačítko bez 3D efektu
- **BS_GROUPBOX** -specifikuje obdélník spojených ovládacích prvků
- **BS_MULTILINE** -určuje, že text v tlačítku může mít více řádek
- **BS_LEFTTEXT** -zarovná text k levému okraji

Možností, určujících styl tlačítek je více, uvádím zde pouze ty, které jsou nejčastěji používané.

1.4 Programovací jazyk C

Jazyk C se začal vyvíjet v Bellových laboratořích AT&T v roce 1969. Největšího pokroku dosáhl Dennis MacAlistair Ritchie v roce 1972. Své pojmenování C získal, protože většinu svých vlastností zdědil ze staršího jazyka B. O rok později se stal jazyk C dostatečně stabilním a většina zdrojového kódu jádra UNIXU, původně napsaného v assembleru pro počítače z rodiny PDP-11, a zařadil se tak mezi prvními operačními systémy napsanými v jiném než strojovém jazyce.

Jazyk C má spoustu programátorsky příjemných vlastností, je nízkoúrovňový, kompilovatelný, relativně minimalistický a dostatečně mocný na většinu systémového programování. Platformě závislé části kódu lze dořešit zápisem inline assembleru přímo do kódu. Zdrojový kód je mnohem čitelnější než assembler, je jednoduché jej zapsat a je snáze přenositelný na jiné architektury. Velmi často jsou operační systémy, knihovny, ovladače, aplikace, překladače a interprety ostatních programovacích jazyků implementovány v jazyce C. Od prvotní specifikace původního jazyka C uběhla již dlouhá doba, během které se objevilo velké množství druhů tohoto programovacího jazyka a tak zde zmíním pouze významné varianty, které jsou dnes považovány za standard.

1.4.1 K&R C

Roku 1978 vyšlo první vydání neformální specifikace jazyka v podobě knihy The C Programming Language autorů Dennise MacAlistair Ritchieho a Briana Kernighana. Tato kniha mezi programátory vstoupila ve známost jako K&R C. Z této knihy pochází také legendární program „Hello World!“ který se objevil namísto dlouhého teoretického výkladu syntaxe a stal se tak v informatice tradicí.

Verze jazyka C, kterou zmínění autoři takto popsali, zavádí některé základní vlastnosti jazyka (jako například datové `typy struct`, `long int`, `unsigned int`) a změny oproti původní koncepci (původní operátory `=-` a `=+` byly změněny za `-=` a `+=`; mátly totiž lexikální analyzér překladače jazyka C). K&R C je považován za základní normu, kterou musejí obsahovat všechny verze překladače jazyka C.

1.4.2 ANSI C a ISO C

V druhé polovině sedmdesátých let začal být jazyk C vytlačován jednoduššími programovacími jazyky pro mikropočítače (varianty jazyků BASIC, PASCAL, atd.). V osmdesátých letech byl přejat i na platformu PC a v roce 1983 se ANSI (American National Standards Institute) dohodla na sestavení komise, aby vytvořila standardní specifikaci jazyka C.

V roce 1990 byl standard ANSI C adaptován institucí ISO (International Organization for Standardization) jako norma „ISO 9899|ISO/EIC 9899:1990“. Jedním z cílů standardizačního procesu ANSI C bylo vytvořit nový standard zahrnující mnoho nestandardních vlastností jazyka C, podporovaných jen některými překladači. Navíc byl jazyk obohacen o funkční prototypy a schopnější preprocesor.

1.4.3 ISO C99

Po standardizaci ISO C se vývoj soustředil převážně na jazyk C++, ale i přesto došlo k dalšímu pokroku a po vydání dokumentu „ISO 9899:1999“ obvykle nazývaném zkráceně C99, bylo C99 uznáno v roce 2000 jako ANSI standard.

Standard C99 obsahuje několik nových vlastností, které byly mnohdy v překladačích již implementovány jako rozšíření. Nejčastěji využívaná rozšíření této normy jsou například inline funkce, možnost deklarace proměnných kdekoli v programu, nové datové typy long long int, complex, bool, pole s nekonstantní velikostí. Některé funkce náchylné na buffer owerflow byly doplněny novými, bezpečnějšími. Tento standard je v některých ohledech přísnější než předchozí ANSI C, například je zakázáno se odkazovat na stejnou paměť ukazateli různých typů – umožňuje to sice vylepšenou optimalizaci, ale může to způsobovat problémy s překladem starších programů.

Kompilátory firmy Microsoft a Borland nepodporují všechna rozšíření ISO C99. Borland X Builder nepodporuje některé datové typy, deklaraci proměnných jinde než na začátku bloku programu a produkt firmy Microsoft má navíc potíže s dynamickými strukturami během ladění aplikace. Standardu ISO C99 se zatím nejvíce blíží GCC (The GNU Compiler Collection).

Vývoj překladače GCC začal v roce 1999 jako GNU projekt, jehož úsilím je vývoj open source překladače a prostředí s podporou několika platforem, různých počítačových architektur, prostředí a specifických rozšíření jazyka C. Tento kompiler je spolu s dalšími balíčky distribuován jako součást integrovaného vývojového prostředí Bloodshed Dev-CPP.

1.5 Hooking

Hooking je programovací technika, při které dochází k nahrazení (nebo jen k monitorování) původního volání WinAPI funkce novou funkcí, která nahradí původní kód funkce kódem vlastním. Někdy je možné toto nahrazení řešit ještě před spuštěním vlastního procesu. V takovém případě se často jedná o uživatelské aplikace, které sami spouštíme a cílem hookingu bývá často vyvolání nějaké změny v chování programu. Příkladem takovéto modifikace může být oprava chyby (patch) aplikace, která je umístěna na disku CD-ROM. U systémových procesů a služeb a u některých programů není možné použití techniky hookingu před spuštěním, v takovýchto případech použijeme wide-hookingu, neboli zavěšení za běhu. Tato technika je běžně využívána rootkity a antiviry.

Ne všechny metody hookingu jsou použitelné na všech Windows. Většina metod je zcela funkčních na verzích s jádrem Windows NT, na Windows 9x a starších se však při použití těchto metod můžeme setkat s jistými obtížemi.

1.5.1 Hooking modifikací souboru

Při použití této metody se jedná o fyzickou záměnu původní knihovny nebo spustitelného souboru obsahujícího funkci, kterou chceme nahradit. Je několik možností, jak dosáhnout úspěchu.

První možností je nalezení vstupního bodu dané funkce a následný přepis kódu. Pokud je kód funkce příliš krátký, můžeme jej nahradit kódem načtení dalšího modulu pomocí funkce `LoadLibrary`, a takto zavolat upravenou funkci z další knihovny.

Jinou variantou této možnosti je využití funkcí jádra, pokud víme, na jakém systému Windows program a změnený modul poběží. V takovém případě je možné díky skutečnosti, že každý proces má svoji kopii v paměti ihned po svém spuštění a jádro systému se v rámci verze operačního systému Windows načítá vždy na stejné místo do paměti, použít přímou adresaci do kernelu a upravit funkci načítání knihovny. Další variantou by bylo vyčkávat na upravovaný program, až nahraje dynamicky připojitelnou knihovnu a upravit jeho hlavní inicializační část.

Ještě jinou snadno proveditelnou a poměrně častou metodou je úplné nahrazení modulu. Při použití této techniky je vytvořen vlastní modul, který úplně nahrazuje požadované funkce, zatímco pro všechny ostatní funkce načítá původní modul a přesměrovává zpracování na funkce původní. Tato metoda má praktickou nevýhodu u rozsáhlých knihoven, kde je potřeba přesměrovávat velké množství funkcí. I přes tyto nevýhody se tato technika často používá například ve spojení s knihovnami funkcí OpenGL pro monitorování chování aplikace, anebo pro modifikaci jejího prostředí.

Hooking aplikace před spuštěním je velice jednoúčelový a zaměřený pouze na jednu aplikaci nebo modul. Velkým nedostatkem této techniky je úprava jen těch procesů, které se spustí později než

hook, vysoké nároky na bezchybnost nových funkcí a také nepřístupnost některých knihoven a souborů, které jsou využívány systémem.

1.5.2 Wide-hooking

Úprava za běhu vyžaduje sice hlubší znalosti systému, ale zajišťuje mnohem lepší výsledky. Touto metodou je možné upravovat funkce ve všech procesech, ke kterým získáme právo na zápis do jejich paměti. Pro samotný přepis se používá funkce WinAPI WriteProcessMemory.

Jako u předchozí metody je několik možností, jak dosáhnout přepisu požadované funkce. První z metod jak wide-hooku docílit, je přepsání IAT (Import Address Table). V této tabulce jsou obsaženy popisy importu a adresy importovaných funkcí. Pokud je libovolná WinAPI funkce volána nepřímo, pak je voláno právě místo v IAT, kde je uložena adresa skoku na místo, které se zaplní až po zavedení procesu do paměti. Zavaděč procesu pak doplní patřičné adresy do IAT, kam se odkazují WinAPI volání programu. Pokud potřebujeme některou z odkazovaných funkcí změnit, jednoduše přepíšeme v této tabulce pouze adresu na vlastní funkci.

Výhodou této metody je její bezchybnost, při provádění větších změn však musíme měnit více funkcí a pro ANSI a WIDE funkce je tato metoda zdlouhavější. K vyhledávání v IAT můžeme použít funkci ze systémové knihovny imagehlp.dll:

```
LPVOID ImageDirectoryEntryToData( LPVOID hInst, BOOL MappedAsImage, short
DirectoryEntry,   ULONG Size);
```

Jako parametr hInst předáváme funkci číslo instance aplikace, které je možné zjistit například pomocí volání funkce `GetModuleHandle` s parametrem NULL. Místo parametru `DirectoryEntry` uvedeme konstantu 1 a dostaneme tak ukazatel na první položku v IAT tabulce. Strukturu záznamů v této tabulce popisuje struktura IMAGE_IMPORT_DESCRIPTOR deklarovaná následovně:

```
typedef struct _IMAGE_THUNK_DATA {
       union {
             PBYTE ForwarderString;
             PDWORD Function;
             DWORD Ordinal;
             PIMAGE_IMPORT_BY_NAME AddressOfData;
       } ;
 } IMAGE_THUNK_DATA,*PIMAGE_THUNK_DATA;

 typedef struct _IMAGE_IMPORT_DESCRIPTOR {
       union {
             DWORD Characteristics;
```

```
            PIMAGE_THUNK_DATA OriginalFirstThunk;
        } ;
        DWORD TimeDateStamp;
        DWORD ForwarderChain;
        DWORD Name;
        PIMAGE_THUNK_DATA FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```

Pokud chceme upravit funkci v nějakém konkrétním modulu, prohledáváme postupně položky IAT tabulky, dokud nenarazíme na položku s odkazem `Name` na jméno upravované knihovny. Nyní musíme postupně prohledat seznam importovaných funkcí a zjistit, jestli se mezi nimi nenachází ta, kterou chceme změnit. Pokud zde najdeme požadovanou funkci, zjistíme si potřebné informace o stránce pomocí `VirtualQuery`, poté zavoláme `VirtualProtect`, abychom umožnili zápis na stránku paměti, kam jej nemáme běžně povolen. Poté přepíšeme starou adresu funkce a opětovným voláním `VirtualProtect` obnovíme ochranu.

Technika hookování za běhu se nejčastěji používá v různých rootkitech, což jsou programy skrývající určité funkce, soubory, adresáře či aplikace před uživatelem, ostatními aplikacemi nebo i některými součástmi systému. Důvodem bývá často jejich ochrana před modifikací, popřípadě zabránění jejich šíření. Nicméně tuto techniku používají také některé počítačové viry a spyware ke svému zneviditelnění.

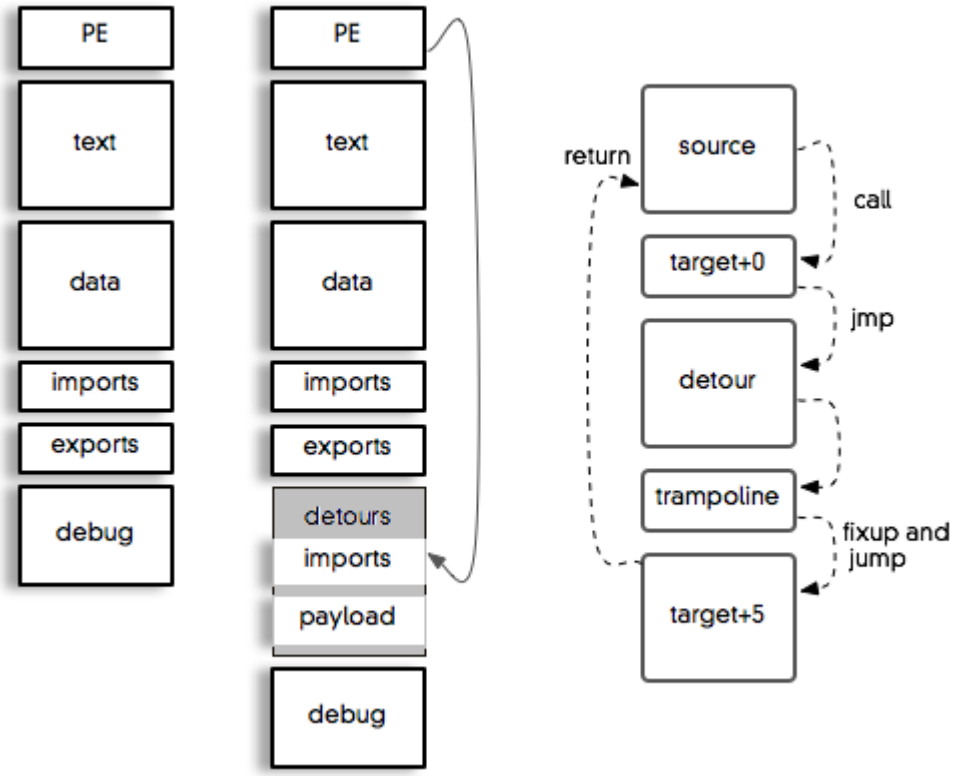
1.5.2.1 Wide-hook se zachováním původní funkce

Vetšinou přepsaní původní funkce nestačí. Např. v případě, že danou funkci nechceme přímo nahradit, ale pouze kontrolovat její výstup, nebo ji nahradit jen v určitých případech. Dobrým příkladem může být skrytí určitých souborů přepsáním vyhledávacích funkcí. Abychom skutečně skryli jen požadované soubory a nic nebylo nápadné, musíme pro ostatní případy zachovat funkci původní. V případě přepsání adresy v IAT je to poměrně jednoduché. Pro volání originální funkce si stačí opět zjistit její začátek pomocí `GetProcAddress` a z nové funkce ji zavolat. Problém nastane při přepsání vstupního bodu. Tímto přepsáním byla vlastně nenávratně zničena původní funkce. Je potřeba uložit originální instrukci.

Nejčastěji se používá postup, kdy víme, že potřebujeme přepisovat prvních 5 bajtů, ale nevíme, kolik je to instrukcí, ani jak jsou dlouhé. V paměti si vyhradíme 16 bajtů místa, na které zapíšeme instrukci NOP (0x90h). Za těchto 16 bajtů umístíme ještě instrukci relativního skoku JMP kterou později doplníme adresou. Takto vytvořenému místu se také říká trampoline, protože zde bude docházet k přesměrování, obrazně k odražení se zpátky do těla původní funkce. K určení počtu bajtů prvních dvou instrukcí můžeme použít některý z volně dostupných algoritmů, popřípadě využít některého z debuggerů či integrovaných prostředí, pokud budeme upravovat jen malý počet konkrétních funkcí. Po zjištění, kolik bajtů tvoří první instrukce (počet musí přesahovat 5bajtů kvůli

přepsání instrukcí skoku), zkopírujeme tyto instrukce na začátek bloku instrukcí NOP a doplníme vypočítanou adresu skoku. Této metodě se říká také *detour* (objížďka), protože dochází k přesměrování původní funkce na nový kód a po jeho vykonání nejčastěji dojde k provedení původního kódu funkce.

Problémem při použití wide-hooku je instrukce relativního skoku na začátku funkce, naštěstí se však takovéto konstrukce nevyskytují často. Dalším problémem jsou příliš krátké funkce, které končí při méně než pěti bajtech. Nicméně s takovýmito problémy se příliš nesetkáváme a wide-hook je nejčastější metoda úpravy požadovaných funkcí nejen v komerčních, ale i volně šířitelných programech.



Obrázek 1-2: Wide hook pomocí detours

Wide-hook se zachováním původní funkce je však možné za běhu také odstranit – pomocí funkce, která provede v případě přepisu IAT vrácení původní adresy, v případě přepsání entry pointu se pak jedná o zkopírování původních bajtů na počátek funkce.

1.5.3 DLL injection

Tato metoda wide-hookingu využívá funkci `CreateRemoteThread`, která vytváří u cizího procesu nové vlákno a spouští ho. Jako parametr však potřebujeme rukojeť požadovaného procesu. Můžeme ji snadno získat funkcí `OpenProces`, pouze potřebujeme získat přístupová práva k paměti daného procesu. Touto technikou je možné spustit nové vlákno z libovolného místa v paměti cizího

procesu. Nyní již stačí pomocí funkce `GetProcAddress` zjistit, kde je v paměti nahraná WinAPI funkce `LoadLibrary`. S úspěchem můžeme využít toho, že funkce `LoadLibrary` má pouze jeden parametr, který dokážeme pomocí funkce `CreateRemoteThread` předat nově vzniklému vláknu. Díky této skutečnosti nasměrujeme funkci pro vytvoření nového vlákna na `LoadLibrary` a jako parametr předáme jméno nové knihovny. Po načtení nového dynamického modulu se přejde k vykonání jeoh inicializační části, která se vykoná. Do této části umístíme kód, který zajistí hook potřebných funkcí. Jakmile se dokončí inicializační část knihovny, vlákno už nemá co vykonávat a ukončí se, ale upravený modul zůstane v paměti dál. Toto řešení je z pohledu programátora velmi jednoduše implementovatelné, bohužel je k němu potřeba vytvářet vlastní knihovnu.

1.5.3.1 Hook přímou úpravou kódu

Tato metoda využívá obdobný postup jako metoda předchozí, nevyžaduje však samostatnou knihovnu. Pomocí funkce `OpenProces` a `VirtualAllocEx` si naalokujeme další pamět v cizím procesu, kam pomocí `WriteProcesMemory` zapíšeme vlastní kód. Tento kód bude při každém spuštění upravované aplikace umístěn v paměti jinde, proto použijeme jen blízké skoky (NEAR). Pak stačí jenom upravený kód spustit pomocí CreateRemoteThread.

V operačním systému Windows 9x není dokonce potřeba vytvářet vlastní kód, ale je možné přepsat přímo kód existující. Opět zjistíme pomocí funkce `OpenProcess` rukojeť procesu, pomocí `VirtualProtectEx` si povolíme zápis do stránky paměti s upravovaným procesem a upravíme její kód funkcí `WriteProcessMemory`.

2 Vlastní realizace aplikace

2.1 Volba technologií pro implementaci

Před zahájením vlastní realizace aplikace jsem se dlouho rozmýšlel, který programovací jazyk použiji. Vizuálně a objektově orientované jazyky jako Java, Visual Basic a Visual C++ nepřicházely v úvahu. Psaní relativně malé aplikace v objektovém nástroji, který přednostně nabízí editaci grafického uživatelského prostředí a kód se v nich dopisuje z velké části pouze ve formě reakcí na události vyvolané uživatelem nebo systémem, mi připadalo nevyhovující. Proto jsem vizuálně orientované nástroje předem zavrhl a začal pátrat po vhodnějším nástroji. Protože jsem zažil éru slavných 16 bitových mikropočítačů a jsem dodnes zastáncem „čistého“ programování, chtěl jsem pro realizaci této aplikace zvolit nástroj na nižší úrovni. Objektové jazyky jako C++ a Java jsou příliš mocné a složité a těžko se v nich realizují programy závislé na systému. Nejprve jsem uvažoval, že bych mohl použít některou z variant programovacího jazyka BASIC, bohužel však asi neexistuje žádná varianta tohoto jazyka, ve které by bylo možné realizovat aplikaci pro systém Windows.

Prakticky jedinou variantou, schopnou vytvářet 32 bitové aplikace s grafickým uživatelským rozhraním, je pomalý a nevhodný Visual Basic. Kdysi dávno jsem realizoval velké množství aplikací v programovacím jazyce PASCAL. Tento procedurální programovací jazyk se vyznačuje velmi jednoduchou syntaxí a velmi snadno se v něm dá naprogramovat i rozsáhlejší aplikace. Bohužel jediný dostupný volně šířitelný kompilátor a IDE pro programovací jazyk PASCAL pro platformu PC je Turbo PASCAL od firmy Borland. Obsahuje sice IDE prostředí určené k vývoji aplikací pro operační systémy Windows, bohužel je tento nástroj na úrovni zastaralých Windows 3.11, popřípadě Windows 95 a bylo by potřeba jeho prostředí obohatit a rozšířit o knihovní funkce nových Windows. Existují sice knihovny rozšiřující tento nástroj i o podporu některých funkcí novějších Windows, bohužel však autoři těchto knihoven se rozhodli ponechat si autorská práva a rozšíření prezentují pouze ve formě demonstračních programů.

Při dalším pátrání jsem narazil na produkt Borland DELPHI. Jedná se v podstatě o upravený a zdokonalený PASCAL pro Windows. Prostředí DELPHI by bylo dostatečně mocné pro vytvoření aplikace pro monitorování událostí. Bohužel mi však nevyhovovalo z obdobných důvodů jako u vizuálních jazyků. Na stránkách firmy Borland jsem objevil také produkt *Borland Builder X*. Toto prostředí je velmi vhodné pro jakýkoliv vývoj nejen šestnáctibitových aplikací pro MS-DOS, ale i plně třicetidvoubitových aplikací vytvářených v jazyce C pro systémy Windows. Rozhodl jsem se tedy realizovat svou aplikaci v jazyce C, protože je velmi dobře podporován i dalšími nástroji. Vývojové prostředí umožňuje zvolit pro kompilaci vyvíjené aplikace některý z velkého množství kompilátorů jako je mimo jiné Mingw, a pokud žádný z obsažených kompilátorů nevyhovuje, je možné přiřadit k prostředí externí kompilátor. Pokusil jsem se tedy nastavit jako implicitní kompilační nástroj sadu kompilátoru GCC, ale kvůli složitému nastavování linkeru a kvůli poznatku, že prostředí nepovolí kompilaci programu, který splňuje normu ISO C99 jsem toto prostředí zavrhl. Největší nepříjemnosti způsoboval fakt, že tento nástroj nepovoloval deklaraci proměnných uprostřed bloku kódu a značně tak brzdil vývoj programu.

Rozhodl jsem se vyzkoušet Microsoft Visual Studio a nastavit jeho prostředí pro vývoj aplikace v jazyce C podle poslední normy, ale při ladění prvních verzí aplikačního rozhraní jsem byl zklamán debuggerem, kterým je tento balík vybaven. Sice je tento nástroj velmi výkonný, ale při korektním uvolňování dynamické struktury zastaví aplikaci a ohlásí, že došlo k chybě. Tento jev byl způsoben tím, že se dealokovávalo dynamické pole uvnitř funkce, jíž bylo předáváno jako korektní ukazatel na začátek struktury a přímé mazání a posouvání ukazatele na další položku struktury tento debugger zmátlo.

Pro konečný vývoj aplikace jsem tedy zvolil jiný nástroj. Protože vývojové prostředí Bloodsheed Dev-CPP používá přímo sadu kompilátorů GCC a podporuje většinu mnou požadovaných vlastností, rozhodl jsem se dokončit aplikaci v tomto nástroji. Kvůli větší kompatibilitě jsem však průběžně zkoušel aplikaci zkompilovat i v prostředí Microsoft Visual Studia a snažil se tak dosáhnout toho, aby byl kód aplikace co nejvíce přenositelný. Vývojové prostředí Dev-CPP obsahuje

jen jednoduchý a pro složitější programy prakticky nepoužitelný debugger, takže ladení aplikace probíhalo pomocí doplnění kontrolních funkcí, které vypisovaly sledované události do souboru, nebo okna aplikace. Tato metoda ladění sice není příliš účinná vzhledem k času, potřebnému na objevení chyby, ale zato vede ke zdokonalování programátora v algoritmickém myšlení, pochopení podstaty vzniklého problému a pozdějšímu vyvarování se podobné chybě. Takto získané poznatky považuji za adekvátní náhradu vůdči bezmyšlenkovitému ladění programu v debuggeru.

Pro vývoj aplikace jsem použil jen volání běžných dokumentovaných funkcí rozhraní *Windows Application Programming Interface*, abych dosáhl co nejvyšší stability na hostitelském systému a dostatečné rychlosti programu. Kód programu splňuje normu jazyka ISO C99, je však upraven tak, aby jej bylo možné přeložit ve dvou prostředích, a to jak v prostředí Dev-CPP, tak i v Microsoft Visual Studiu.

2.2 Shrnutí požadavků na vyvíjenou aplikaci

Při návrhu aplikace jsem vzal v úvahu již existující programy s širokou škálou využití. Byly to jak programy, určené k monitorování stisknutých kláves většinou za účelem nechtěného shromažďování informací, tak i neškodné aplikace počítající počet pixelů, kolik myš od prvního spuštění aplikace urazila. Některé profesionální komerční monitorovací produkty umožňovaly odesílání logu událostí do emailových schránek nebo například opětovné přehrávání a úpravu zalogovaných událostí v podobě maker.

2.2.1 Základní požadavky

Základním požadavkem na program se stalo zachytávání a analýza zpráv vyvolaných událostmi myši a klávesnice. O příjmu těchto zpráv by měl program uživatele informovat okamžitě od zahájení zachytávání v informačním okně. Navíc by aplikace měla umožňovat ukládání logování do souboru, přičemž je kladen důraz na to, aby bylo možné nastavit ukádání výpisu událostí způsobených v jednotlivých oknech aplikací *separátně*. To znamená, že pro uživatelem zvolené aplikace bude vytvářen samostatný logovací soubor. Navíc by mělo být umožněno kontrolní logování do globálního souboru přijatých zpráv. Rovněž by mělo být umožněno nastavení, zda se budou monitorovat pouze klávesy, nebo zda uživatel požaduje monitorovat v dané aplikaci pouze události myši, popřípadě oba dva typy těchto událostí. Program by si dále měl dokázat zapamatovat jednotlivá nastavení tak, aby vyžadoval po úvodním nastavení pokud možno co nejméně uživatelských zásahů, s vyjímkou prohlížení logovacích souborů a případnou změnou nastavení pro již přednastavenou aplikaci. Mělo by být umožněno buď automatické, nebo alespoň ruční přidání nastavení pro další, nový, dříve nemonitorovaný program.

2.2.2 Požadavky na uživatelské rozhraní

Z uživatelského hlediska by prostředí aplikace mělo působit jednoduchým dojmem a snadným ovládáním. Bylo by vhodné, aby uživateli daná aplikace nepřekážela při práci a v případě potřeby umožňovala dočasné přerušení monitorování například z důvodu zadávání hesel, či jiných závažných událostí. Po většinu času tato aplikace poběží bez získání větší pozornosti uživatele, ale v případě potřeby musí umožňovat snadný přístup k zalogovaným událostem. Tohoto chování je možné využít například při pádu jiné aplikace, do které uživatel zadal velké množství textu, který se neuložil. Příkladem může být libovolný textový editor nebo databázový systém. Tímto by se dal zformulovat další požadavek na aplikaci, čímž je okamžité ukládání výsledků, a jejich snadná dostupnost.

2.2.3 Nároky aplikace na prostředí

Při akceptování všech těchto požadavků jsem se rozhodl vyvinout aplikaci, která bude umožňovat většinu těchto vlastností a s ohledem na běžné uživatele Windows nebude vyžadovat žádné zásahy do programového vybavení počítače či jeho konfigurace. Pro svou inicializaci, instalaci ani běžné používání nebude aplikace vyžadovat žádné zvláštní nástroje. Vzhledem k druhu vyvíjené aplikace a nejčastěji používanou verzi operačního systému Windows je tento program odladěn pro použití na operačním systému Windows XP a měl by správně fungovat také na systémech Windows 2000, Windows Vista a také na Windows NT minimální verze 4.0 s nainstalovanými opravami Service Pack. Realizace všech částí kódu programu pomocí WinAPI a funkcí základních knihoven jazyka C by měla zajišťovat dostatečnou stabilitu aplikace.

2.2.4 Návrh aplikace

Před vlastní realizací aplikace jsem se rozhodl zamyslet se nad vyvíjeným programem a rozčlenit si realizaci programu na jednotlivé části. Program musí pracovat s databází pro uchovávání nastavených informací pro jednotlivá monitorovaná okna, další částí je návrh vhodného uživatelského rozhraní aplikace a obsluha překreslování. S tím souvisí otázka řešení smyčky zachytávání zpráv pro jednotlivá okna vyvíjeného programu. Dále bude potřeba vyřešit problematiku logování událostí do souboru, která se dá rozčlenit na vyhodnocování zprávy a vlastní proces logování. Samostatnou a nejdůležitější částí bude funkce pro zachytávání zpráv ostatních procesů a jejich přeposílání do naší aplikace.

2.3 Implementace programové části

Při vlastní realizací aplikace jsem se nejdříve zaměřil na vývoj kostry programu. Vytvořil jsem nejprve jednoduchou funkci `WinMain`, v níž jsem zaregistroval novou třídu aplikace vyplněním jednotlivých položek struktury WNDCLASS a zavoláním funkce `RegisterClass`. Poté jsem

vytvořil zkušební okno aplikace a nastavil mu jen defaultní atributy. Dále jsem implementoval cyklus klasické smyčky zpráv, kde je nejprve zpráva zachycena funkcí `GetMessage`, následně dojde k překladu kódů virtuálních kláves pomocí `TranslateMessage` a rozeslání těchto zpráv příslušným ovládacím prvkům aplikace funkcí `DispatchMessage`. Pro první spuštění kostry programu bylo ještě nutné vytvořit *proceduru okna*, kterou jsem pojmenoval `MainWndProc` a doplnit odkaz na ni do struktury pro vytvoření třídy. Do těla procedury okna jsem umístil konstrukci pro větvení programu podle typu přijaté zprávy v parametru `uMsg`. Ošetřil jsem zpracování zprávy `WM_CREATE`; v takovémto případě se neprovede žádná akce, protože k inicializaci okna aplikace dojde již ve `WinMain`. Aby bylo možné aplikaci ukončit i křížkem v rohu jejího hlavního okna, pro zprávy `WM_CLOSE` a `WM_DESTROY` dojde k odeslání zprávy `PostQuitMessage`, která způsobí korektní ukončení aplikace. Všechny ostatní zprávy prozatím zpracuji zavoláním defaultní funkce `DefWindowProc`.

Obrázek 2-1: Kostra aplikace

2.3.1 Funkce Paint

Při požadavku na překreslení okna aplikace, tedy při přijetí zprávy WM_PAINT, by mělo docházet k aktualizaci obsahu okna. Tato zpráva dorazí, jestliže je okno překryto jiným oknem a je následně přeneseno do popředí, nebo při jeho minimalizaci a opětovné maximalizaci. Upravil jsem tedy proceduru okna a vytvořil novou funkci Paint. V této funkci jsem si připravil device context okna pro kreslení do okna aplikace a vytvořil nový font založený na fontu operačního systému, jehož výšku jsem pro jistotu nastavil napevno na 12 pixelů.

2.3.2 Výpis procesů

V této fázi vývoje aplikace jsem přemýšlel, jak identifikovat jednotlivé okolní aplikace. Jsou v podstatě dvě možnosti. První možností je zjistit k jakému procesu okno patří, což se dá téměř jednoznačně určit pouze podle názvu spustitelného souboru, kterým byla aplikace spuštěna. Tento způsob jsem zavrhl, protože uživateli aplikace by název spustitelného souboru nic neříkal a znesnadňovalo by to ovládání aplikace. Zvolil jsem raději druhou možnost a to identifikaci oken

podle textu jejich titulku. Tato volba se později ukázala s ohledem na některé ostatní aplikace poněkud nešťastná, a to z toho důvodu, že aplikace obsahuje několik vnořených oken, ale toto dění se dá později v aplikaci ošetřit nastavením stejných logovacích pravidel pro obě vnořená okna.

Do funkce `Paint` jsem doplnil úvodní nastavení počítadla řádků výpisu aplikací `rowcnt` hodnotou `GUI_LINE`, což je konstanta, kterou jsem deklaroval kvůli stanovení výšky jednotlivých položek řádku výpisu programů. Dále pomocí funkce `EnumWindows` prohledávám všechna okna aplikací spuštěných v systému a jako parametr předávám další funkci `EnumWindowsProc`. V této funkci jsem realizoval vypisování seznamu aplikací do hlavního okna pomocí funkcí `GetWindowText` a `TextOut` s využitím počítadla rowcnt. Při spustění laděné aplikace jsem si uvědomil, že pro dlouhý výpis procesů bude potřeba přidat do okna aplikace posuvníky.

2.3.3 Okno se scrollbary

Vytvoření posuvného okna poněkud zbrzdilo vývoj aplikace. Po několika pokusech jsem přišel na to, že nejjednodušší metodou jak vytvořit posuvné okno je inicializovat hlavní okno aplikace nastavením vlastností `WS_VSCROLL|WS_HSCROLL|ES_AUTOHSCROLL|WS_AUTOVSCROLL`, čímž dojde k doplnění okna o scrollbary, a následně inicializovat další okno jakožto jeho potomka, které se v něm bude pohybovat. Tomuto oknu jsem přiřadil globální proměnnou v podobě rukojeti msWnd. Pro určení potřebných rozměrů vnitřního okna jsem vytvořil proměnné HMAXSCROLL a VMAXSCROLL, které se aktualizují při každém překreslení okna vzhledem k počtu rádků výpisu oken aplikací.

Při oživování funkcí scrollování je důležité ošetřit zprávy způsobené událostmi kliknutí na jednotlivé části scrollbarů. Pokud dojde k události na vertikálním posuvníku, je nutné obsloužit zprávu `WM_VSCROLL`. Tato zpráva obsahuje v dolním slově parametru wParam identifikátor součásti posuvníku. V případě že dojde k tažení za posuvník, tak tento identifikátor je `SB_THUMBPOSITION` a horní slovo parametru wParam obsahuje aktuální pozici posuvníku. V našem případě bychom chtěli vidět vždy celý řádek výpisu, takže pozici posuvníku přeupravím tak, aby byla dělitelná konstantou výšky řádku, a tím docílím požadovaného efektu. Pro události způsobené kliknutím na dolní šipku je posláno `SB_LINEDOWN`, což v našem případě způsobí posun o jeden grafický řádek dolů - musíme však ošetřit, aby nedošlo k většímu posuvu než je rozdíl velikosti oken. Z tohoto důvodu jsem potřeboval funkci, která by dokázala vracet výšku okna, a uvědomil jsem si, že u horizontálního posuvníku budu potřebovat obdobnou funkci pro šířku, a tak jsem nadeklaroval jednoduché funkce `GetClientWidth` a `GetClientHeight` kterým se jako parametr předává handle příslušného okna a vracejí typ integer.

Při posuvu o řádek nahoru v případě, že jako parametr zprávy je `SB_LINEUP`, dochází k obdobnému porovnání - tentokráte z toho důvodu, aby okno neutíkalo dolů. Pro posuvy o stránku

nahoru a dolů při obdržení `SB_PAGEUP` a `SB_PAGEDOWN` jsem nastavil posun o deset grafických řádků a následné ošetření nadměrného posunu okna zůstává stejné jako u posunu o řádek.

Po vyhodnocení události vertikálního posuvníku jsem usoudil, že je vhodné přenastavit jeho rozsah pomocí funkce `SetScrollRange`, protože mohlo dojít ke změně počtu běžících aplikací. Následně upravím jeho pozici voláním funkce `SetScrollPos` a přemístím vnitřní okno `msWnd` uvnitř hlavního okna o zápornou hodnotu posuvníků provedením funkce `SetWindowPos`. Jak jsem si později uvědomil, je ještě nutné ošetřit stav, kdy je šířka hlavního okna větší než vnitřního a ošetřit v takovém případě ještě nechtěné posouvání vnitřního okna do stran.

Pro ošetření událostí horizontálního scrollbaru při obdržení `WM_HSCROLL` jsem postupoval obdobně. V tomto případě se v dolním slově parametru wParam objeví některá z konstatnt `SB_THUMBPOSITION`, `SB_LINERIGHT`, `SB_LINELEFT`, `SB_PAGERIGHT` nebo `SB_PAGELEFT`. Pro všechny tyto možnosti jsem postupoval obdobně jako u vertikálního posuvníku, opět jsem nejdříve upravil hodnotu posuvníku tak aby nepřetekla přes jeho rozsah, dále aktualizoval pozici a rozsah horizontálního posuvníku a závěrem přenastavil pozici vnitřního okna.

Vertikální posun okna je ještě vhodné realizovat pomocí kolečka myši. V takovém případě musíme vhodně obsloužit zprávu WM_MOUSEWHEEL, která v horním slově parametru wParam vrací velikost posunu kolečka. Tato velikost je dělitelná konstantou WHEEL_DELTA. Bohužel Microsoft Visual Studio tyto konstanty nemá deklarované, a tak je nutné je nadefinovat například následujícím způsobem:

```
#ifndef WM_MOUSEWHEEL
#define WM_MOUSEWHEEL 0x020A
#endif
#ifndef WHEEL_DELTA
#define WHEEL_DELTA 120
#endif
```

Po vyhodnocení velikosti otočení kolečka myši provádím stejné operace s posuvníky a vnitřním oknem jako u obsluhy vertikálního posuvníku.

2.3.4 Minimalizace do systémové lišty a menu aplikace

Pro skrytí aplikace do systémové lišty nestačí pouze naplnit strukturu `NOTIFYICONDATA` a vytvořit ikonu v systémové liště voláním funkce `Shell_NotifyIcon(NIM_ADD, &niData)`, ale je potřeba ještě zajistit schování okna příkazem `ShowWindow(hWnd, SW_HIDE)`. Všechny tyto části kódu umístěné ve `WinMain` způsobí okamžitou minimalizaci aplikace ihned po jejím spuštění.

Pro opětovnou maximalizaci aplikace při poklepání myši na ikonu aplikace dojde k předání zprávy WM_APP, kterou je potřeba obsloužit. Protože jsem se rozhodl implementovat menu při

poklepání pravým tlačítkem myši, nadefinoval jsem několik vlastních konstant podle počtu položek v menu a jejich hodnota je inkrementována od WM_APP vždy o jednotku.

Pro vlastní menu jsem implementoval novou funkci ShowContextMenu, vytvářející menu aplikace o třech položkách. Za pomocí funkce CreatePopupMenu a vložení jednotlivých položek InsertMenu dochází k vytvoření klasického menu minimalizované aplikace v systémové liště. Na závěr této funkce provádím ještě volání funkce TrackPopupMenu, která vytvořené menu zobrazí na momentálních souřadnicích myši, jež získávám pomocí funkce GetCursorPos.

Nyní stačí jen ošetřit jednotlivé zprávy vyvolané po obdržení zprávy WM_APP, kdy v lParam obdržíme identifikátor akce myši. V případě, že dojde k poklikání WM_LBUTTONDBLCLK, zobrazíme okno aplikace příkazem ShowWindow, v případě, že klikneme pravým tlačítkem WM_RBUTTONDOWN, provedeme zobrazení kontextového menu.

Obsluhu jednotlivých událostí kliknutí na položky kontextového menu je možné zjistit po přijetí zprávy WM_COMMAND, kdy je v horní polovině wParam předáván identifikátor položky menu.

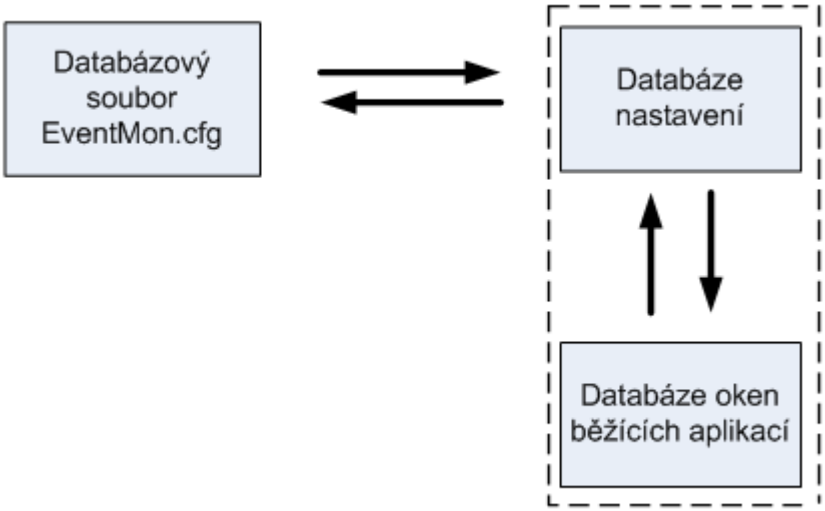
V kontextovém menu jsem realizoval celkem tři položky, jednou se ovládá hook a unhook aplikace, další slouží k rychlému přístupu k souboru s globálním logováním a poslední slouží k ukončení aplikace.

2.3.5 Dynamická struktura pro práci s nastavením

V další fázi vývoje jsem se pozastavil nad skutečností, že mám téměř vytvořenou grafickou podobu aplikace, ale stále nemám navrženou strukturu pro ukládání informací o jednotlivých komponentách nastavujících logování pro jednotlivé aplikace. Pro každou aplikaci v seznamu bude potřeba několik ovládacích prvků, navíc do struktury musí být uloženo jméno aplikace a cesta k logovacímu souboru. Deklarace výsledné dynamické struktury vypadá následovně:

```
typedef struct db_buf buf_db;
struct db_buf{char *appname;   //application name
              BOOL Mmov;       //global logging
              BOOL Mclck;      //mouse
              BOOL Mkbd;       //keyboard
              HWND HMmov; //button global logging
              HWND HMclck;     //button mouse logging
              HWND HMkbd; //button keyboard logging
              HWND Hmlog;      //select log file button
              HWND HMshw; //open log button
              char *logfile;          //log file name
              buf_db *next;           //next};
```

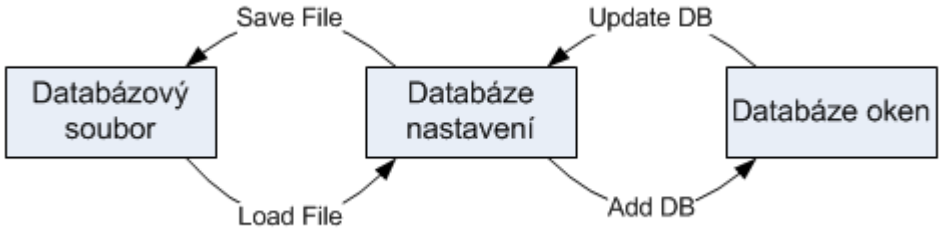
Nyní bylo nutné uvědomit si, jaké operace se s danou strukturou budou provádět. Program by měl být schopen zapamatovat si své nastavení a někam ho ukládat tak, aby při dalším spuštění aplikace došlo k jeho opětovnému obnovení. Dále bude potřeba stejné struktury pro generování seznamu aplikací a jejich ovládacích prvků. Tyto skutečnosti znázorňuje následující obrázek, ze kterého vyplývá, že budeme pro aplikaci potřebovat dvě nezávislé dynamické struktury:



Obrázek 2-2: Schéma propojení dynamických struktur

2.3.5.1 Podpůrné funkce pro dynamické struktury

Jak jsem se zmínil již v předchozí kapitole, pro uložení nastavení aplikace a pro databázi oken běžících aplikací bude potřeba celkem dvou struktur, mezi kterými bude docházet k vzájemným aktualizacím. To bude vyžadovat vytvoření vhodných podpůrných funkcí. Úplný diagram vztahů mezi souborem s databází a dvěmi dynamickými strukturami vypadá takto:



Obrázek 2-3: Podpůrné funkce dynamických struktur

Do hlavního programu jsem doplnil dvě globální proměnné typu ukazatel na strukturu buf_db, přičemž proměnná `database` bude určena pro databázi otevřených oken, a `filedata` bude sloužit

jako databáze nastavení. Podpůrné funkce pro práci s těmito dynamickými strukturami jsem se rozhodl umístit do samostatného souboru `dbtool.c`, do kterého jsem později z hlavního souboru programu přemístil i některé další funkce.

2.3.5.2 Funkce pro úpravy dynamických struktur

První potřebnou funkcí, kterou jsem napsal, byla funkce pro přidání další položky do databáze. Tuto funkci jsem pojmenoval `adddb` a jako parametr je potřeba jí předat ukazatel na databázi, do níž se bude položka přidávat, dále řetězec s názvem položky, který nesmí být prázdný, název logovacího souboru a nastavení přepínačů pro globální logování, myš a klávesnici. V těle této funkce je nejprve zkontrolováno, zda není ukazatel na databázi prázdný. Pokud ano, dojde k alokaci paměti potřebné pro první záznam a řetězce s názvem a cestou k logovacímu souboru, následnému překopírování těchto řetězců do alokovaného místa a nastavení přepínačů struktury podle předávaných parametrů. V případě, že databáze již obsahuje nějaké záznamy, funkce nejprve prochází dynamickou strukturu až na poslední záznam a na jeho položku next nastaví ukazatel nově vytvořené položky.

Další funkce sloužící pro aktualizaci databáze nastavení `upddb` je již složitější. Této funkci se jako parametry předávají ukazatele na obě dvě databáze, dále parametr id specifikující identifikátor, s jehož pomocí je možné rozlišovat, který záznam a položka v databázích se bude vyhledávat, aktualizovat a synchronizovat. Funkce je určena pro použití v proceduře okna, kde při svém volání obdrží jedinečný identifikátor položky, která vyvolala událost. V těle funkce se tento identifikátor rozloží na identifikátor určující pořadí záznamu v databázi `loop` a identifikátor položky `box`. Následně dochází k prohledávání databáze oken a nalezení záznamu se stejným pořadovým číslem. Dále dochází k vyhodnocení položky, která se má aktualizovat a její vlastní aktualizaci. Následně je vyhledáván záznam v databázi s nastavením, která má stejné jméno jako aktualizovaný záznam v databázi oken a jeho jednotlivé položky jsou rovněž přepsány aktualizovanými hodnotami.

Pro zjištění, zda již v databázi existuje záznam, jsem vyvinul funkci `isindb`, které se předá pouze ukazatel na databázi a název hledané položky. Funkce postupně prohledává všechny záznamy v databázi, a pokud má některá z nich stejný název, vrací hodnotu `TRUE`. V případě, že databáze takovou položku neobsahuje, vrátí `FALSE`.

Obdobou předchozí funkce je funkce `finddb`, která rovněž prochází jednotlivé záznamy v databázi a hledá záznam stejného jména, ale na rozdíl od předchozí funkce vrací ukazatel na nalezený záznam. Když se hledání nezdaří, vrátí `NULL`.

Poslední ze základních funkcí pro práci s databází je funkce `cleandatabase`, která slouží k vyprázdnění dynamické struktury. Postupně prochází záznamy databáze, uvolňuje všechny dynamicky alokované řetězce a ničí vytvořená okna, pokud některé rukojeti neobsahují NULL. V této funkci dochází k úplnému zrušení databáze.

2.3.5.3 Databázové funkce pracující se soubory

Pro vývoj funkcí pracujících se souborem bylo nutné vymyslet způsob, jak ukládat potřebné informace o jednotlivých nastaveních. Prakticky je potřeba uložit všechny položky struktury `buf_db` s vyjímkou rukojetí oken. Jedná se celkem o tři přepínače a dva dynamické řetězce. Vzhledem k tomu, že by bylo vhodné, aby byl i soubor s nastavením snadno čitelný, zvolil jsem formát, ve kterém je každý záznam z dynamické struktury reprezentován třemi řádky v souboru. Na prvním řádku jsou jednotlivé přepínače v pořadí Mmov, Mclck a Mkbd reprezentované znaky 0 pro hodnotu FALSE a 1 pro hodnotu TRUE. Na následující řádek je uložena hodnota položky appname, tedy název položky, jejíž nastavení se právě ukládá a na poslední třetí řádek je uložena cesta a název logovacího souboru dané položky. Tento formát uložených dat považuji za vyhovující vzhledem k charakteru ukládaných položek struktury.

Funkce `loadfile` realizuje načtení databázového souboru. Jako parametry je nutné jí předat ukazatel na databázi, kam se mají data ze souboru načíst a řetězec určující název souboru. Při svém zavolání se funkce pokusí otevřít zadaný soubor pro čtení. Pokud neuspěje, dojde k jejímu ukončení a databáze zůstane prázdná. V opačném případě dochází k načítání ze souboru po jednotlivých znacích. Pomocí proměnné row, počítající čísla řádků záznamu a proměnné line, pro počítání znaků na řádku, jsou postupně analyzovány hodnoty uložených přepínačů, poté řetězec názvu záznamu a jména logovacího souboru. Pro tyto řetezce jsem dopsal funkce pro realokaci a přidání znaku k řetězci `addctos` a spojení dvou řetězců včetně realokace `addstos`. Informace jsou přidány do požadované databáze voláním funkce `adddb`.

O něco jednodušší byla realizace funkce pro uložení databázového souboru. Tato funkce nejprve přepíše soubor s nastavením a otevře jej pro zápis. Poté postupně prochází dynamickou strukturu databáze a ukládá jednotlivé záznamy do souboru pomocí funkce fprintf. Uložení položky trvá o něco kratší dobu, než její načtení, protože se řetězce zapisují najednou, kdežto při načítání ze souboru je řetězec rekonstruován po znacích, dokud nedojde k nalezení znaku konce řádku.

Během ladění funkce ukládání databáze jsem narazil na problém. Při ukládání databáze před ukončením ladícího programu nedošlo k jejímu uložení a soubor zůstal prázdný s nulovou velikostí. Tento problém byl způsoben bufferem, který je při práci se souborem vytvářen. Příkazem `setbuff` jsem tedy provedl nastavení jeho velikosti na NULL a tím jsem docílil okamžitého zápisu do souboru.

2.3.6 Začlenění databázových struktur do aplikace

Po dopsání podpůrných funkcí pro práci s databázemi bylo nutné tyto databáze vhodně začlenít do vyvíjené aplikace a najít vhodná místa pro přeaktualizování záznamů. Rozhodl jsem se nejprve začlenit databázovou strukturu pro otevřená okna aplikací. Do těla funkce `Paint` jsem dopsal pomocí funkce `TextOut` záhlaví jednotlivých položek, která se objeví v posuvném okně `msWnd`, a na její

začátek hned za inicializaci proměnné čítače grafických řádků rowcnt jsem přidal volání `cleandatabase(&database)`, které zajistí vyprázdnění databáze oken při každém překreslení.

2.3.6.1 Úprava funkce EnumWindowsProc

Další kód bylo nutné dopsat do těla funkce EnumWindowsProc, ve které dochází ke generování seznamu oken. Původní vypisování jména aplikace jsem vymazal a nahradil konstrukcí filtrující názvy oken „EventMon“, což je název vyvíjené aplikace. Toto filtrování jsem zavedl proto, aby později nedocházelo ke zbytečnému logování dění i při nastavovaní aplikace samotné. Podmínku jsem doplnil o filtrování prázdného řetězce a do těla podmínky jsem dopsal konstrukci pro přidání nové položky do databáze s úvodním nastavením aplikace. Toto jsem realizoval pomocí volání funkce adddb. Tato funkce předává ukazatel na nově vytvořenou položku, který si uložím do dočasné proměnné buffer.

Dalším krokem je doplnění prvků grafického uživatelského rozhraní. Pro záznam o každém okně, které je v předchozím kroku přidáno do databáze, je nutné vytvořit ovládací prvky pro nastavení jednotlivých možností logování. K vytvoření a uschování rukojetí na tyto komponenty slouží ve struktuře `buf_db` právě položky typu `HWND HMmov`, `HMclck`, `HMkbd`, `HMlog a HMshw`. Při vytváření těchto komponent předávám jako identifikátor komponenty při volání funkce `CreateWindow` klíč vypočítaný číslem řádku komponenty vynásobeným 100, ke kterému přičítám konstantu 800, protože identifikátory komponent s nižší hodnotou jsem již pravděpodobně využil k něčemu jinému. Jednotlivé ovládací prvky na řádku příslušící jedné aplikaci rozlišuji přičtením další konstanty 1 až 5, podle požadované akce. Handle na jednotlivé vytvářené komponenty přiřazuji do dočasné proměnné, která ukazuje na poslední položku přidanou do databáze.

Na grafický řádek nejprve přidávám checkbox, kterým bude možné nastavování globálního logování (s identifikátorem končícím číslem 1), dále pak tlačítko pro výběr samostatného logovacího souboru (4), pak následují další dva checkboxy pro nastavení logování myši (2) a klávesnice (3). Na konci grafického řádku vytvářím dlouhé tlačítko, jehož název je totožný s názvem aplikace, pro jejíž nastavování řádek slouží. Rukojeti na takto vytvořené komponenty uložím do patřičných položek dočasné proměnné buffer, která je stále nastavena na poslední přidanou položku do databáze oken. Dojde tak k uložení rukojetí jednotlivých ovládacích prvků do databáze.

Pro dodatečná nastavení ovládacích prvků zasílám funkcí `SendMessage` zprávy a určuji tak některé jejich další vlastnosti, které je možné specifikovat až po jejich vytvoření. Závěrem funkce provádím zobrazení a aktualizaci komponent voláním funkcí `ShowWindow` a `UpdateWindow`.

Nyní je po překompilování a spuštění aplikace možné nastavovat jednotlivé položky. Tímto jsem dokončil začlenění databáze oken do programu. Začlenění databáze nastavení `filedata` je o něco těžší. Vyžaduje přidání další vnořené podmínky namísto příkazu `adddb`, který doteď vytvářel v podstatě jen položky nové. Touto podmínkou je vrácení nenulového ukazatele na strukturu, při vyhledávání v databázi nastavení. V takovém případě dojde k provedení funkce `addb` pro databázi

`database` s parametry vrácenými tímto ukazatelem. V opačném případě dojde k vytvoření nové položky v databázi oken s implicitním nastavením.

2.3.6.2 Úprava spouštění aplikace

Při spuštění aplikace by mělo docházet k automatickému načítání nastavení. K tomuto účelu slouží funkce `loadfile`, které je potřeba předat jako parametr databázi, kam se budou položky ze souboru načítat. Volání této funkce je vhodné umístit do funkce WinMain někam před volání funkcí pro zobrazení hlavního okna aplikace. Další nezbytnou úpravou je automatické uložení databáze s nastavením při ukončování programu. Protože funkce savefile bude volána pouze z jednoho místa, doplnil jsem ji ještě o příkaz cleandatabase, pro závěrečné vyprázdnění databáze s nastavením.

2.3.6.3 Oživení ovládacích komponent nastavení

Pro uložení nastavení před ukončením programu jsem přidal do procedury okna do větve, která je prováděna při obdržení zprávy `WM_DESTROY` při ukončování aplikace volání následujících dvou funkcí:

```
savefile(&filedata,"EventMon.cfg");
cleandatabase(&database);
```

Tato dvě volání způsobí nejen automatické uložení, ale také vyprázdnění dynamických struktur před ukončemím programu.

Nyní, když máme hotové načítání a ukládání databáze nastavení a kopírování nastavení pro již známá okna do databáze, je vhodné dopsat funkci provádějící synchronizaci upravených nastavení provedených v konfiguračním okně aplikace v obou databázích a provést oživení jednotlivých komponent. Do funkce MainWndProc v části podmíněné přijetím zprávy `WM_COMMAND` dopíšeme do části default kód podmíněný přijetím zprávy s dolním slovem parametru wParam dekrementovaným o 800, jehož zbytek po dělení stem je v rozmezí 1 až 3, víme s jistotou, že přišla událost od checkboxu. Provedeme tedy volání funkce upddb s předáním parametrů obou databází, získaným identifikátorem komponenty a stavem checkboxu, který je možné získat použitím funkce SendDlgMessage, které se předávají jako parametry rukojeť okna, získaný identifikátor komponenty a jako druh zprávy `BM_GETCHECK`. Zbývající parametry jsou nulové.

2.3.6.4 Prvek nastavení samostatného logovacího souboru

Pro událost způsobenou komponentou, která po vyhodnocení předchozího výrazu má zbytek 4, je nutné nejprve vytvořit dialogový box uložení souboru. Pro tento účel jsem do souboru dbtool umístil funkci `SaveFileDialog` s jediným parametrem, kterým je výstupní řetězec. V těle této funkce dochází k naplnění datové struktury `OPENFILENAME` nezbytnými položkami, dále je nastaven filtr

souborů na přípony log a txt. Poté je volána WinAPI funkce `GetSaveFileName` a dojde ke vytvoření dialogového okna. V případě, že je dialog uzavřen křížkem nebo tlačítkem storno, vrací funkce název implicitního logovacího souboru „EventMon.log“.

2.3.6.5 Oživení komponent pro otevření logovacích souborů

Při poklikání na název aplikace v konfiguračním okně aplikace dojde rovněž k vytvoření zprávy `WM_COMMAND` a při vyhodnocení výrazu obdržíme zbytek 5. V takovémto případě bude volána funkce openlog umožňující otevření souboru s logovanými událostmi dané aplikace.

Na začátku těla funkce openlog dochází opět k vyhledání záznamu podle identifikátoru komponenty a položka logfile tohoto záznamu je testována na prázdný řetězec. Pokud je u dané aplikace nastaven logovací soubor, dojde k jeho otevření v textovém editoru notepad.

Nyní při spuštění výsledné aplikace máme program, který se dokáže minimalizovat do systémové lišty, reaguje na změny v konfiguračním okně a tyto změny promítne v databázi s nastavením, kterou již dokáže automaticky ukládat a opětovně načítat při svém dalším spuštění.

2.3.7 System wide hook

Pro realizaci zachytávání událostí bylo nutné zvolit vhodnou metodu. První metoda hookingu pomocí modifikace souboru či přepsání jeho části je nevhodná. Aplikace by měla umožňovat korektní odstranění hooku při svém ukončení, či jen dočasné přerušení monitorování, čehož bychom při použití zmíněné první metody nemohli dosáhnout. K tomuto účelu se výborně hodí *wide hook*. Tento druh hooku umožňuje monitorování funkce, ale nenahrazuje její kód. Jedná se pouze o úpravu, kdy je v položkách tabulky importů IAT nalezen odkaz na danou funkci a její kód je analyzován kvůli zjištění délky jejích počátečních instrukcí. Poté dojde k přepsání prvních instrukcí skoku do těla nové obslužné funkce, ale původní instrukce se překopírují do předem připravené oblasti vyplněné bezvýznamnými instrukcemi NOP. Na konci toho bloku instrukcí je instrukce skoku, která umožní přeskok na další instrukci v původní obslužné funkci. Tuto techniku hookování jsem chtěl původně realizovat.

Pro pokusné účely jsem vytvořil nový program využívající funkci pro tento účel vytvořené knihovny. Dalším ladícím programem jsem chtěl hooking realizovat. Deklaroval jsem si strukturu popisující jednotlivé položky v IAT a začal jsem je prohledávat. Podařilo se mi nalézt položku, odkazující se na mojí knihovnu, ale nezdařilo se mi nalézt import funkce. Po opravě chyby v programu jsem konečně nalezl adresu, kde je funkce v paměti umístěna. Při pokusu o přístup k instrukcím na této adrese docházelo často k pádu ladící aplikace kvůli nedostatečnému oprávnění. Pro korektní chování aplikace by musela aplikace získat práva přístupu na úrovni systému. Vzhledem k tomuto problému a složitosti algoritmu analyzujícího instrukce, jsem implementaci zmíněné techniky tímto způsobem vzdal. Je sice možné nastavit breakpoint některému debuggeru dovnitř modifikované funkce a zjistit, jaké instrukce se tam nalézají, ale takovýto jednoúčelový hook není

použitelný pro zachytávání událostí klávesnice a myši, které jsou generovány pomocí systémových funkcí uložených v chráněných knihovnách. Kód těchto funkcí by se mohl s každou verzí takovéto knihovny lišit a proto je jednoúčelový hook prakticky nepoužitelný.

2.3.7.1 Hook pomocí WinAPI

Z tohoto důvodu jsem se rozhodl realizovat hook pomocí funkcí rozhraní WinAPI `SetWindowsHookEx` a `UnhookWindowsHookEx`. Tato technika hookování je pro monitorování systémových událostí dostačující a taktéž se řadí mezi system wide hooky, jenž je možné za běhu odstranit.

Pro realizaci vlastní hookovací funkce touto technikou je nutné umístit hook funkce provádějící zachytávání zpráv do dynamicky připojitelné knihovny. Toto je způsobeno jevem, že operační systém neumožňuje nahrání kódu spustitelného souboru na stejná místa jako kód knihovních funkcí. Díky tomuto poznatku jsem v programovacím nástroji vytvořil další samostatný projekt pro realizaci hookovací knihovny.

2.3.7.2 Tvorba hookovací knihovny

S ohledem na to, že hook bude umístěn v dynamicky připojitelné knihovně, jsem začal zvažovat, jak bude probíhat komunikace mezi aplikací a hookovací knihovnou. První možností je nadeklarovat v knihovně sdílený segment a aplikací si data z tohoto segmentu postupně vyzvedávat. Takovéto provedení by bylo poněkud neohrabané a těžkopádné, protože aplikace by musela být doplněna například o časovač, při jehož události by docházelo k vyzvednutí výsledků z knihovny. Navíc by zpracovávání výsledků v knihovně mohlo mít negativní vliv na rychlost odezvy systému a spuštěných aplikací. Dalším problémem by bylo zachycení většího počtu zpráv, než je možné uložit, a pak by mohlo docházet k jejich zahazování. Tyto teoretické úvahy mě navedly na myšlenku realizovat komunikaci mezi knihovnou hook.dll a aplikací rovněž pomocí zasílání zpráv. Fronta zpráv je průběžně aplikací vyprazdňována a aplikace se o jejím přijetí dozvídá automaticky prováděním hlavní smyčky zpráv.

Pro zaslání zprávy aplikaci bude nutné specifikovat nový typ zprávy z důvodu odlišení běžných zpráv od zpráv hookovací knihovny. Implementace vytvoření nového typu zprávy je proveditelná voláním WinAPI funkce `RegisterWindowMessage`.

Pro realizaci knihovny jsem do nového projektu vytvořil nový soubor hook.c. Pokračoval jsem vytvořením inicializační funkce knihovny, nesoucí název DllMain. Syntaxe prototypu inicializační funkce vypadá následovně:

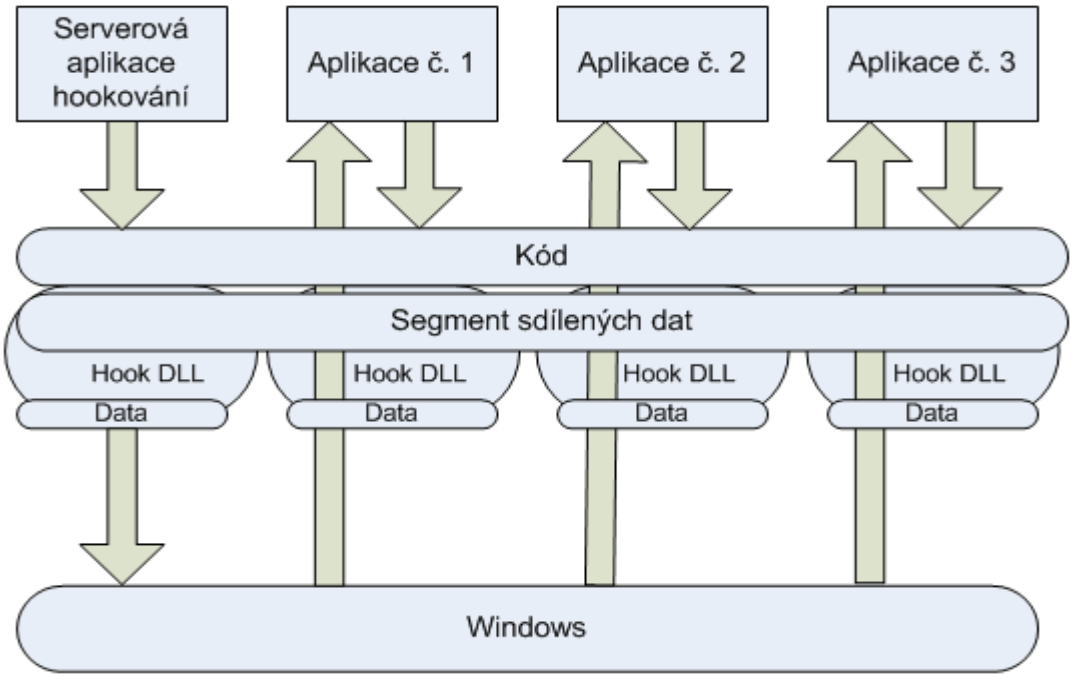
```
BOOL APIENTRY DllMain(HINSTANCE hinstDLL, DWORD call, LPVOID lpReserved);
```

Dále jsem pokračoval vytvořením globální proměnné hInst typu HINSTANCE, do které jsem přiřadil v inicializační části rukojeť instance. Pro účely pozdějšího odesílání vlastních zpráv do aplikace jsem předchystal globální proměnné a zaregistroval dva nové druhy zpráv. Zprávu HOOKKBDMSG, která bude sloužit pro odesílání informací o událostech klávesnice a HOOKMSEMSG, sloužící k informování zachytávací aplikace o událostech myši.

Pokračoval jsem vytvořením funkcí MyMouseHook a MyKbdHook, jejichž prototyp obsahuje identifikátor kódu volání a dva parametry, které jsou součástí zachycené zprávy. Na úvod těchto funkcí jsem pro korektní fungování hooku vytvořil konstrukci, která zkouší, zda se tento kód rovná HC_ACTION, což je jediný případ, kdy má význam vyhodnocovat přijaté parametry. V případě, že předaný kód je jiný, ukončím funkci návratovou hodnotou získanou voláním `CallNextHookEx`, která předává hookovací informace další hookovací proceduře v řetezci hooků. Tělem těchto funkcí se stal pouze příkaz SendMessage, který bude odesílat zprávu do okna přes rukojeť předanou při volání procedury zavěšení hooku. Jako typ zprávy jsem nastavil nově zaregistrované typy zpráv. Zbývají dva parametry zprávy, které je možné zaslat aplikaci, a v obou případech potřebuji předat oba dva parametry a navíc identifikátor okna, pro které byla původní zpráva určena. Protože události klávesnice a myši jsou implicitně odesílány oknu na popředí, je možno handle tohoto okna získat funkcí `GetForegroundWindow`. Tuto rukojeť přidávám do zprávy jako parametr `wParam`. Pro úpravu ostatních parametrů jsem vytvořil funkci `compparam`, která zhustí předávané informace do parametru `lParam`.

Další důležitou součástí bude funkce pro nastavení hooku, která provede pouze zavolání funkce SetWindowsHookEx pro oba dva hooky, tedy WH_KEYBOARD a WH_MOUSE. Dalšími parametry se předává zachytávací funkce a rukojeť na instanci knihovny. Poslední parametr určuje identifikátor vlákna - v našem případě bude nulový. Této funkci jsem původně předával jako parametr rukojeť okna aplikace EventMon pro příkazy SendMessage, bohužel jsem však zjistil, že takovýto hook odesílal zprávy pro aplikaci pouze, když byla na popředí. Výše popsaný problém byl způsoben tím, že handle předaná této funkci byla nulová a odeslání zprávy s nulovou rukojetí okna způsobí doručení této zprávy do fronty zpráv stejného procesu, kterým byla odeslána. Tuto chybu jsem opravil získáním korektní rukojeti pomocí volání `FindWindow("EventMonClass","EventMon")`.

Knihovnu jsem vybavil funkcí UnHook(), která v případě, že není některý z ukazatelů na hookovací proceduru roven prázdné hodnotě NULL, provede odinstalování zachytávání pomocí volání funkce `UnhookWindowsHookEx`. Pro případ pádu aplikace jsem inicializační funkci knihovny obohatil o volání funkce pro odinstalování, v případě, že je volána z důvodu uvolnění knihovny z paměťového prostoru (DLL_PROCESS_DETACH). Do hlavičkového souboru hookovací knihovny jsem doplnil pouze importy na funkce `Hook()` a `UnHook()`.



Obrázek 2-4:Schéma wide hooku

2.3.7.3 Přidání kódu pro vyhodnocování zachycených událostí do aplikace

Dalším krokem vývoje aplikace je načtení knihovny hook.dll a propojení na její dvě funkce Hook a UnHook. Tato část kódu je realizována ve funkci `WinMain` v podobě volání WinAPI funkce `LoadLibrary` s odkazem na hookovací knihovnu jakožto parametrem. V případě chyby vypíšu chybové hlášení v podobě MessageBoxu a ukončím aplikaci. Dojde-li ke korektnímu načtení modulu, propojím funkce nabízené modulem na předem připravené ukazatele `pfAttach` a `pfDetach` pro pozdější použití. Pro vyhodnocování zpráv, které bude odesílat hook aplikaci, je nutné zaregistrovat stejné zprávy i v aplikaci, a proto jsem ještě doregistroval nové typy zprávy HOOKKBDMSG a HOOKMSEMSG.

V proceduře zpracování zpráv u položky menu zastupující Enable a Disable jsem doplnil volání právě těchto importovaných funkcí. Tímto umožníme zavěšení a odstranění monitorování za běhu pomocí položky menu. Zbývá doplnit kód pro vyhodnocování zpráv zasílaných hookem. Za tímto účelem na začátek procedury umístíme podmínky pro jednotlivé dvě zprávy a jako první krok provedeme rozbalení parametru lParam voláním funkce decompparam. Následně parametry vyhodnotíme a pomocí funkce finddb najdeme v databázi nastavení odpovídající záznam se stejným názvem, jaký má okno, jehož rukojeť byla odeslána jako wParam. Podle nastavení položek záznamu z databáze provedeme zapsání události do zvoleného logovacího souboru nebo do hlavního logu zavoláním funkce `appendmessage`. Jako parametry jsou předávány pouze název souboru a titulek aplikace, pokud se jedná o logování do hlavního souboru a text zprávy.

V hlavním logovacím souboru jsou okna aplikací uzavřena v hranatých závorkách a umístěna vždy na novém řádku. Speciální klávesy jsou uzavřeny mezi znaky < a >, a události myši jsou ukládány ve složených závorkách. V samostatných logovacích souborech je syntaxe ukládaných událostí stejná, pouze se do souboru neukládá titulek okna.

2.3.7.4 Ladění hookování

V této fázi byla aplikace testována, zda splňuje všechny požadované vlastnosti. Při testech se ukázalo, že v knihovně funkce je nutné nahradit volání funkce SendMessage za volání PostMessage, aby se zabránilo nežádoucímu tuhnutí procesů.

Další úprava spočívala v ošetření programu proti nechtěnému spuštění ve více instancích. Instalace několika hooků do systému jendnak nemá smysl a navíc systém značně zatěžuje. Ošetření aplikace spočívá v deklaraci nové globální proměnné eventmonwindow, která je uvnitř funkce EnumWindowsProc inkrementována při každém nalezení okna aplikace. Když při spuštění aplikace zjistí, že běží více těchto oken, automaticky se ukončí.

2.3.7.5 Informační okno

V této fázi byla aplikace doplněna o informační okno, které zobrazuje všechny zachycené události, hned po přijetí zprávy a jejím následném dekódování. Jedná se o další okno aplikace inicializované ve funkci WinMain, do jehož komponent jsou předávány výsledky událostí přijímaných od hookovací funkce.

2.4 Popis obsluhy aplikace

V této kapitole jsem se rozhodl uvést základní návod, jak s aplikací pro monitorování událostí zacházet. I když jsem se při vývoji snažil o vytvoření uživatelsky přívětivého, jednoduchého a intuitivního rozhraní a o řádný popis a ošetření jednotlivých ovládacích prvků, nemusí být každému uživateli jasné, jak správně zacházet s tímto programem.

2.4.1 Instalace a údržba aplikace

Aplikace EventMon nevyžaduje žádné importy do registrů, ani jiné specifické postupy pro instalaci, avšak pro jeho spuštění je nutné, aby dynamická knihovna hook.dll, jež slouží, jak už je patrné z jejího názvu, k navázání hooku na systém, byla ve stejném adresáři jako program. Pokud tato knihovna není pro program z nějakého důvodu přístupná, zobrazí se chybová hláška „Could not load HOOK.DLL“ a program se ukončí.

Konfigurace aplikace se ukládá do souboru EventMon.cfg. Pokud tento soubor neexistuje, aplikace si jej dokáže sama vytvořit a při své inicializaci použije implicitní nastavení.

Z důvodu údržby a úspory místa na pevném disku je vhodné čas od času narůstající soubory s logy promazat.

2.4.2 Ovládání grafického rozhraní aplikace

Aplikace EventMon se skládá ze dvou oken, první okno zobrazuje list spuštěných procesů a k nim přiřazené ovládací prvky, druhé okno slouží k zobrazování pohybů a stisků tlačítek myši a zároveň informace o kombinacích stlačených kláves.

Po spuštění aplikace se hlavní okno, které slouží k ovládání aplikace, zobrazí jako ikona na systémové liště, do níž se pokaždé minimalizuje, aby nezabírala místo na liště pro přepínání spuštěných aplikací. Klepnutím pravým tlačítkem myši na tuto ikonu se zobrazí nabídka o několika položkách, obsahující položku „Enable“. Kliknutím na tuto položku se dává programu podnět k započetí monitoringu činností uživatele na klávesnici a myši, což se zobrazuje v informačním okně aplikace.

Obrázek 2-5: Ikona aplikace v systémové liště

2.4.2.1 Informační okno

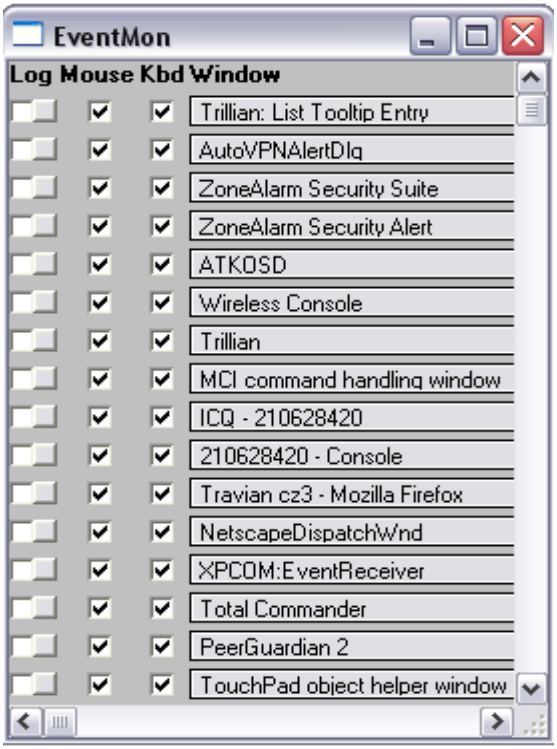
Informační okno je zobrazováno „always on top“, tedy „vždy nahoře“, a je možné jej umístit libovolně na obrazovku klasickým tahem za horní lištu okna. Pokud je monitoring vstupů z klávesnice a myši momentálně zapnut, pak je v nabídce místo položky „Enable“ na témž místě položka „Disable“, sloužící k zastavení monitorování těchto vstupů. Druhé tlačítko této nabídky je „Open LOG“ což je tlačítko sloužící k otevření hlavního log souboru v aplikaci notepad. Poslední tlačítko v této nabídce je tlačítko „Exit“ sloužící k ukončení celého programu. Před ukončením není potřeba monitoring manuálně vypínat, dojde k tomu automaticky.

Obrázek 2-6: Informační okno

Jak můžeme vidět na obrázku, okno se skládá ze dvou částí, přičemž v horní části je zobrazeno pět tlačítek. Tři velká tlačítka reprezentují levé, prostřední a pravé tlačítko myši, zbývající dvě malá tlačítka slouží k indikaci točení kolečkem myši nahoru a dolů. Stisknutí tlačítka myši je v informačním okně ohraničeno černým okrajem, nestlačené pak prostorově vystupuje. Na ilustračním obrázku vidíme, že okno indikuje stlačení levého tlačítka myši. Dolní část okna obsahuje panel, který informuje o momentální poloze myši a také stlačené kombinaci kláves.

2.4.2.2 Nastavování logování událostí

Hlavní ovládací okno aplikace zobrazující seznam oken běžících aplikací obsahuje velké množství ovládacích prvků. Při kliknutí myší na název aplikace dojde k otevření samostatného logovacího souboru aplikace. Pokud soubor neexistuje nebo nebyl zatím vytvořen, neprovede se žádná akce. Prvním checkboxem se nastavuje logování aplikace do společného souboru EventMon.log. Vedle něj je umístěno tlačítko pro zvolení samostatného logovacího souboru. Pokud je zadán samostatný logovací soubor a zároveň zatržen checkbox pro logování do společného souboru, provede se logování do obou souborů. Další dva checkboxy slouží k nastavení událostí, které se budou ukládat do souboru, první je pro zachytávání událostí myši a druhý slouží k nastavení zachytávání klavesnice. Tato nastavení jsou společná pro logování jak do globálního souboru, tak i pro samostatný logovací soubor.



Obrázek 2-7: Okno s konfigurací

2.4.3 Možnosti rozšíření aplikace

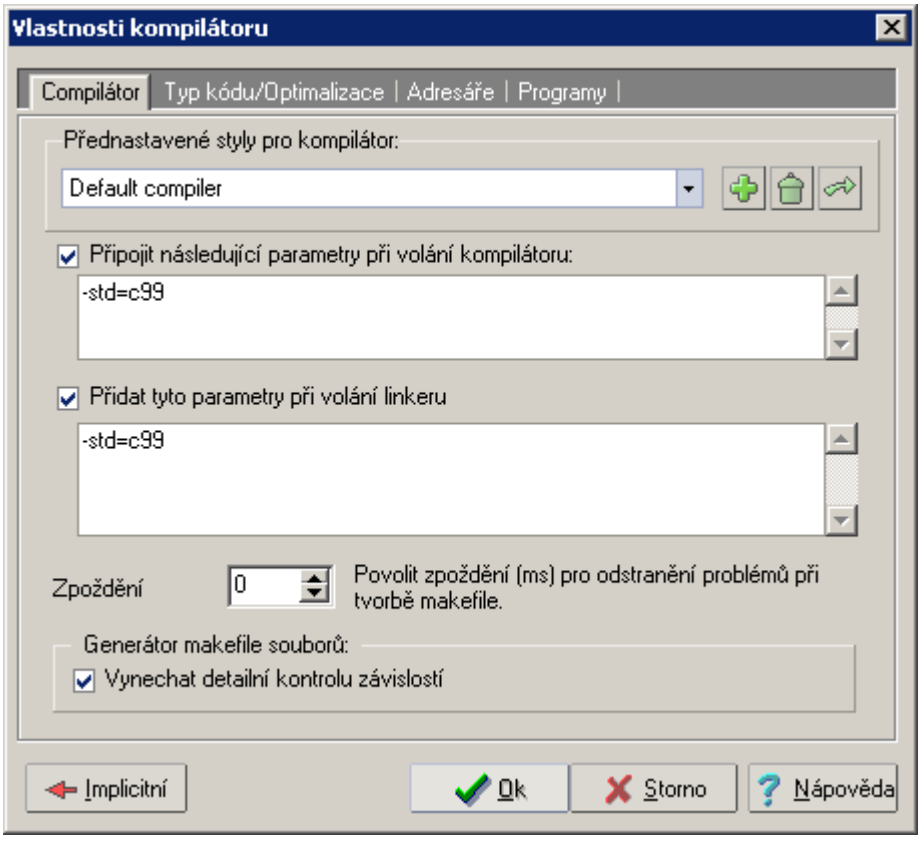
Aplikaci by bylo vhodné rozšířit o možnost nastavení zachytávání jednotlivých podkategorií zpráv pro každou aplikaci. Dále by do souborů logu mohla být přidána časová razítka, která by identifikovala, kdy došlo k zalogování dané události. Jinou variantou rozšíření aplikace by mohlo být její využití jakožto programu automatizujícího procesy klikání a vypisování textu v podobě možnosti definovat makra a na jejich základě pak generovat sled zvolených událostí.

2.4.4 Překlad aplikace

Aplikace EventMon je napsaná tak, že ji je možno bez větších problémů přeložit ve většině kompilátorů jazyka C, které splňují alespoň základní požadavky normy ISO C99.

2.4.4.1 Překlad pomocí Dev-CPP

V prostředí Dev-CPP je nutné nejdříve nastavit kompilátor prostředí tak, aby implicitně kompiloval podle normy ISO C99. Toto je možné provést v menu nástroje, nastavení kompilátoru. Do pole „Připojit parametry při volání kompilátoru“ vepíšeme -std=c99. Stejně tak nastavíme linker a zaškrtneme oba dva checkboxy. Nyní je vhodné zkontrolovat, zda máme prostředí Dev-CPP minimálně ve verzi 4.9.9.0 a Mingw 3.3.1.



Obrázek 2-8: Nastavení IDE Dev-CPP

Nyní otevřeme nejprve projekt knihovny. Můžeme tak učinit kliknutím na soubor Hook.dev. Pro kompilaci projektu stiskneme klávesu <F9>. Dojde k překladu a vytvoření dynamické knihovny hook.dll. Dalším krokem je nakopírování této knihovny do složky projektu aplikace. Opět otevřeme vývojové prostředí, tentokráte však kliknutím na EventMon.dev. Provedeme kompilaci, a pokud je vše správně nastaveno, program by se měl vzápětí bez potíží rozeběhnout.

2.4.4.2 Překlad v Microsoft VS nebo Microsoft VC

Překlad programu v těchto prostředích je o trochu složitější. V případě, že nebude možné projekt knihovny a programu otevřít pomocí přiložených souborů Hook.sln a EventMon.sln, bude nutné pro kompilaci založit nové projekty, u nichž nesmíme zapomenout vypnout UNICODE. Nyní by mělo být možné program bez problému překompilovat.

3 Závěr

Při tvorbě této práce jsem se naučil využívat komponent rozhraní WinAPI pro tvorbu aplikací operačního systému Windows a zdokonalil jsem se ve znalostech programování v jazyce C. Počáteční problémy s implementací jednotlivých částí programu, jehož první testovací verze byly značně nestabilní, se mi podařilo v pozdější fázi ladění odstranit.

Vrámci projektu se mi podařilo realizovat filtry zachytávání zpráv na úrovni jednoduchého filtrování, v případném pokračování projektu bych však chtěl implementovat rozšířený filtr. V další verzi programu bych chtěl rovněž realizovat možnost tvorby makroinstrukcí a automatického provádění sledu nastavených událostí. Současná verze aplikace nemůže v žádném případě konkurovat komerčním programům, může se však stát užitečnou utilitou.

Upravenou knihovnu pro zachytávání bych chtěl použít pro svou další práci na projektu monitorování zamrznutí programů.

Literatura

[1] Milníky v historii společnosti Microsoft. Dokument dostupný na URL
http://www.microsoft.com/cze/presspass/billgates/milniky.asp (duben 2007)

[2] Msdn Library. Dokument dostupný na URL
http://msdn2.microsoft.com/en-us/library/ (duben 2007)

[3] The C Programming Language. Dokument dostupný na URL
http://cs.wikipedia.org/wiki/The_C_Programming_Language (květen 2007)

[4] Windows API. Dokument dostupný na URL
http://cs.wikipedia.org/wiki/Windows_API (květen 2007)

[5] Programování ve Win32 API. Dokument dostupný na URL
http://www.win32api.euweb.cz/main.php?type=clanek4 (květen 2007)

[6] System services. Dokument dostupný na URL
http://msdn2.microsoft.com/en-us/library/aa969179.aspx (květen 2007)

[7] Meziprocesová komunikace. Dokument dostupný na URL
http://cs.wikipedia.org/wiki/Meziprocesov%C3%A1_komunikace (květen 2007)

[8] C++ Q&A. Dokument dostupný na URL
http://www.microsoft.com/msj/0200/c/c0200.aspx (květen 2007)

[9] Hand detouring Windows Funciton Calls With HT. Dokument dostupný na URL
http://www.matasano.com/log/620/ (květen 2007)

[10] Hooking Windows API. Dokument dostupný na URL
http://www.l0t3k.net/biblio/windows/cz/ (duben 2007)

[11] Systémové programování. Dokument dostupný na URL
http://vyuka.ft.utb.cz/file/17/Lectures/SYP_Lecture3.pdf (duben 2007)

[12] C Programovací jazyk. Dokument dostupný na URL
http://cs.wikipedia.org/wiki/C_%28programovac%C3%AD_jazyk%29 (duben 2007)

[13] Event Logging. Dokument dostupný na URL
http://www.codeproject.com/system/Event+Logging (duben 2007)

[14] BB2TD Windows Programming. Dokument dostupný na URL
http://www.ssv-embedded.de/ssv/pc104/p83.htm (duben 2007)

[15] Vitrual Key Strokes. Dokument dostupný na URL
http://delphi.about.com/od/objectpascalide/l/blvkc.htm (duben 2007)

[16] API hooking revealed. Dokument dostupný na URL
http://www.codeguru.com/cpp/w-p/system/misc/article.php/c5667/ (květen 2007)

Seznam použitých zkratek a symbolů

ANSI	American National Standards Institute
CDFS	CD-ROM File System
FAT	File Allocation Table
IAT	Import Application Table
ISO	International Organization for Standarditation
MS-DOS	Microsoft Disk Operating System
NTFS	New Technology File System
WinAPI	Windows Application Programming Interface

Seznam příloh