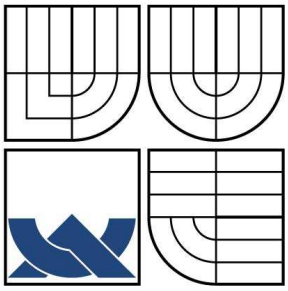
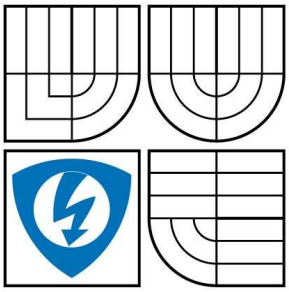


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKACNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

PROGRAMOVÁNÍ PROCESORŮ V 32- NEBO 64-BITOVÝCH OPERAČNÍCH SYSTÉMECH

PROGRAMMING OF PROCESSORS IN 32- OR 64-BIT OPERATION SYSTEMS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. RÁŠO ONDŘEJ
AUTHOR

VEDOUCÍ PRÁCE Ing. MIROSLAV BALÍK, PH.D.
SUPERVISOR

Anotace

Tato diplomová práce se zabývá aplikačně programovým rozhraním Win32 a programovým jazykem symbolických instrukcí. Práci lze tématicky rozdělit do čtyř částí. První část se zabývá základy 32bitového programování v jazyce symbolických instrukcí ve Windows.

V druhé části se výklad zaměřuje na popis prostředku Win32 sloužící ke komunikaci mezi počítači.

Ve třetí části jsou přehledně zpracovány vybrané úlohy v jazyce symbolických instrukcí s odpovídajícím popisem a komentářem.

Poslední část této diplomové práce se zabývá vytvořením vzorové aplikace. Tato aplikace a její jednotlivé části budou tvořit počítačová cvičení předmětu „Počítače a jejich periferie“ vyučovaný na fakultě Elektrotechniky a komunikačních technologií VUT v Brně.

Klíčová slova: Win32 API, DDL, TCP/IP, Peer to peer, Client-server, Socket, Multitasking, NetBIOS, Jazyk symbolickych instrukci.

Abstract

The thesis deals with the Win32 application programming interface and symbolic instructions programming language. It is divided into four parts. First part covers the basics of 32-bit programming in symbolic instructions language in Windows.

Second part describes the Win32 resources for computer intercommunication.

Third part presents, describes and comments on selected problems in symbolic instructions language.

Last part focuses on the creation of an example application. This application and its parts will used as the content of computer subject *Počítače a jejich periferie* at VUT Brno FEEC.

Keywords: Win32 API, DDL, TCP/IP, Peer to peer, Client-server, Socket, Multitasking, NetBIOS, Assembly language.

.

Prohlášení

Prohlašuji, že svoji diplomovou práci na téma Programování procesorů v 32- nebo 64- bitových operačních systémech jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....

SEZNAM ZKRATEK

ANSI	American National Standards Institute.
API	Application programming interface (programové rozhraní pro vytváření aplikací).
ARP	Address Resolution Protocol.
ASCII	American Standard Code for Information Interchange.
ASCIIZ	ASCII zero.
BSS	Block Started by Symbol.
CryptoAPI	Cryptography API.
DLL	Dynamic-link library (dynamicky linkovaná knihovna).
FIFO	First In First Out.
FTP	File Transfer Protocol.
GCC	GNU compiler collection.
GDI	Graphics device interface, rozhraní pro grafické prostředky.
GNU	Rekurzivní akronym,GNU is Not Unix.
GUI	Graphical User Interface, grafické uživatelské rozhraní.
HTTP	Hypertext Transfer Protocol.
IP	Internet Protocol.
IPv4	Internet Protocol verze 4.
IPv6	Internet Protocol verze 6.
ISO/OSI	International Standard Organization's Open System Interconnect
JSI	Jazyk symbolických instrukcí.
LAN	Local area network
LGPL	GNU Lesser General Public License.
NASM	The Netwide Assembler.
MDI	Multiple Document Interface.
MNAV	Počítače a jejich periferie.
MS	Microsoft.
MUTEX	Mutual exclusion object.
NCB	Network Control Block.
NetBIOS	Network Basic Input/Output System.
OS	Operační systém.
P2P	Peer to peer.
PC	Personal computer (osobní počítač).
PE	Portable Executable.

PDU	Protocol Data Unit.
SNMP	Simple Network Management Protocol.
STDCALL	Standardní volání funkcí.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol
UNICODE	Unicode Consortium.
Win32	Windows API pro 32 bitovou PC architekturu.
WndProc	Window’s procedure (procedura okna).
WWW	World Wide Web.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

Úvod

Jazyk symbolických instrukcí je jazyk nejnižší úrovně a je proto závislý na strojovém kódu procesoru. I když v průběhu posledních let bylo vytvořeno velké množství vyšších programovacích jazyků, jako je například Java, tak studium jazyka symbolických instrukcí má pořád svůj význam. Jedná se například o programování ovladačů [15], debugging a následnou exploitací kódu [3], nebo snaha o napsání maximálně efektivního kódu.

Cílem diplomové práce je prostudovat způsoby programování pod 32- bitovým aplikačně programovým rozhraním systému Windows v jazyce symbolických instrukcí; popsanou látku demonstrovat na příkladech a zaměřit se na možnosti komunikace. Posledním bodem je příprava části úloh pro počítačová cvičení.

Text diplomové práce se zabývá 32 bitovou verzí Windows API nazvanou Win32 API a komunikačním protokolem TCP/IP.

Velký důraz se přitom klade právě na příklady, které prakticky vysvětlují probíranou látku. To proto, aby text diplomové práce mohl sloužit jako studijní opora pro počítačové cvičení předmětu MNAV (Počítače a jejich periferie). Tento předmět, vyučovaný na fakultě Elektrotechniky a komunikačních technologií VUT v Brně, si klade za cíl podrobně seznámit studenty s architekturou mikroprocesorů a mikroprocesorových systému založených na platformě PC. Počítačová cvičení tohoto předmětu se právě zabývají řešením programátorských úloh v jazyce symbolických instrukcí na platformě PC s využitím rozhraní Win32. Z tohoto důvodu byla zvolena studijní forma výkladu od seznámení se s prostředím Win32 přes programy potřebné pro kompilaci nebo linkování. Následuje nastudování programátorských technik, jejíž zvládnutí je potřebné pro programování v jazyce symbolických instrukcí ve Win32 API. Poslední část textu se pak zabývá pokročilejšími technikami programovaní, jaké jsou například práce s vlákny nebo mutexy. Příklady z jednotlivých kapitol jsou začleněny v jedné vzorové aplikaci. Vytvoření této vzorové aplikace studenty předmětu MNAV pak může být podmínkou udělení zápočtu. Tato vzorová aplikace postihuje všechny probírané kapitoly a je jakýmsi logickým vyústěním práce studenta v počítačových cvičeních předmětu MNAV.

Vzorovou aplikaci, zdrojové soubory, včetně detailních komentářů, lze nalézt na přiloženém CD.

1 Úvod do programování Win32 API

1.1 Všeobecný úvod

Windows byl původně jen grafickou nástavbou operačního systému MS DOS. Postupem času se z něj však stal plnohodnotný operační systém. Tento operační systém používá chráněný režim (protected mode) procesoru a ve svém jádře poskytuje služby grafického rozhraní.

Aplikace, které komunikují s jádrem operačního systému se nazývají nativní aplikace. Tyto aplikace komunikují s jádrem OS pomocí aplikačně programového rozhraní (API) operačního systému. Toto rozhraní pro OS Windows se nazývá Win32. Přípona 32 značí, že se jedna o API pro 32 bitovou architekturu. Dále existují Win16,Win64 pro 16 bitovou respektive 64 bitovou architekturu. Rozhraní Win32 tedy podporuje všechny služby operačního systému i služby pro ovládání GUI.

Charakteristickým rysem programovaní pod Win32 je volaní funkcí Win32 rozhraní. Argumenty těchto funkcí jsou pak většinou složité struktury. Dalším rysem je pak objektová architektura grafických prvků.

Nativní aplikace pod OS DOS volali služby jádra přes vektor přerušení 0x21, pod OS Windows pak nativní aplikace volají funkce systémových knihoven DLL.

1.2 Windows API

Windows API lze podle funkčně rozdělit do 7 kategorií [9].

1) Administrace a managament
   Služby administrace a managamentu umožňují instalaci, konfiguraci a servis aplikací nebo systémů.

2) Diagnostické služby
   Jedná se o služby které umožňují odstraňovat chyby aplikací nebo systému a dále umožňují monitorovat výkon aplikací nebo systémů. Do této kategorie spadá například debugging.

3) Grafické a multimediální služby
   Tato služba umožňuje aplikacím pracovat s formátovaným textem, grafikou, zvukem nebo videem. Aplikace k těmto službám přistupují skrz rozhraní pro grafické prostředky (GDI). OpenGL nebo DirectX je představitelem Grafických a multimediálních služeb.

4) Síťově služby
   Síťové služby umožňují komunikaci mezi aplikacemi na rozdílných počítačích v síti, nebo umožňují sdílení zdrojů v síti, například adresářů nebo tiskáren. API Windows poskytuje různorodé standardy, jako například Sockety a Simple Network Management Protocol (SNMP). Dále poskytuje podporu celou řadou protokolů HTTP, FTP nebo třeba DNS.

5) Ochranné služby
   Ochranné služby umožňují autentifikaci heslem, ochrana všech sdílených systémových objektů, privilegované řízení přístupu, management práv.

Například možnosti kryptografie umožňuje aplikačně programové rozhraní CryptoAPI.

6) Systémové služby
Funkce systémových služeb umožňují aplikacím přístup ke zdrojům počítače a k základním vlastnostem operačního systému, jako například paměť, soubor systému, různá zařízení a vlákna.

7) Služby uživatelského rozhraní
Služby uživatelského rozhraní poskytují funkce pro tvorbu a řízení oken a dalších základních prvků (tlačítka, posuvníky, atd.).

Je důležité říci, že všechny funkce poskytované Win32, které mají jako parametr řetězec znaků, mají dvě varianty a to v závislosti na použitém druhu znakového řetězce. Buď funkce pracují s ANSI řetězcem nebo UNICODE. ANSI řetězce jsou klasické 8 bitové posloupností znaků z rozšířené ASCII tabulky. UNICODE jsou posloupností 16 bitových znaků všech existujících abeced.

Tyto funkce jsou pak rozlišený příponou A nebo W, v závislosti zda se jedná o funkce, které používají ANSI nebo UNICODE znakové řetězce. V následujícím textu se budou výhradně používat ANSI varianty funkcí.

Naprostá většina znakových řetězců je pak typu ASCIIZ, tedy nulou ukončené řetězce.

Příklad, definice znakového řetězce:

```
HELLO: DB "HELLO WORLD!",0
```

V tomto příkladu byl definován ukazatel HELLO na nulou ukončený řetězec. Jedná se o řetězec typu ASCII. K Vytvoření řetězce byla použita direktiva DB. Řetězec obsahuje text:“HELLO WORLD!“ a číslici nula.

1.3 Netwide Assembler

Netwide assembler je překladač jazyka symbolických adres. Tento překladač je distribuován pod licencí LGPL, což znamená, že je volně ke stažení na Internetu.

Mezi přednosti toho překladače patří velká podpora maker. Použití maker pak celý zdrojový kód zpřehlední a zjednoduší. Dokonce některé konstrukce maker se dají považovat za samostatný programovací jazyk, samozřejmě zde neexistuje možnost typové kontroly.

Překladač NASM se spouští v příkazové řádce následovně:

```
NASM soubor [možnosti]
```

V následujícím textu budou použity tyto možnosti překladače NASM:

`-o, output`	Jméno souboru jenž bude výstupem překladu.
`-e, preprocess only`	Překladač nebude překládat, ale pouze zpracuje zdrojový kód Preprocesorem (vloží soubory, rozvine makra, vloží konstanty,…).
`-F, format`	Specifikuje formát překladu.

-g, debugging	Do výsledného souboru budou přibaleny ladící informace.

Příklad, překlad do obj formátu:

```
nasm okno.asm –fobj
```

Přeloží zdrojový kód v souboru okno.asm do formátu OBJ a uloží jej do souboru okno.obj. Jelikož nebyl specifikován výstupní soubor, tak jej NASM automaticky pojmenoval na nasm.obj.

Příklad, zpracovaní preprocesorem:

```
nasm -o okno_preproc.txt -e okno.asm
```

Zpracuje zdrojový kód v souboru okno.asm preprocesorem a výstup programu bude uložen do souboru okno_preproc.txt. Pokud by nebylo zadáno jméno výstupního souboru, tak by byl výstup zobrazen na standardním výstupu (v okně terminálu).

1.4 Zdroje

Zdroje (resources) je komplexní skriptovací systém pro Microsoft Windows platformu. Zdroji se bude zabývat kapitola 8. Proto zde bude uveden jen příkaz pro jejich kompilaci.

Příklad, kompilování zdrojů:

```
rc dial.rc
```

Zkompiluje soubor dial.rc (Resource Script) a výstupem příkazu bude soubor dial.res (Compiled Resource Script).

1.5 Alink

Alink je program, který slouží linkování objektových souborů vytvořených například v NASM. Stejně jako NASM se Alink spouští v příkazové řádce. Program se linkuje takto:

```
ALINK soubor [volby] [další soubory] [volby] ...
```

Alink obsahuje mnoho voleb, jejich výpis je možno získat spuštěním linkeru s parametrem –h. Jediným povinným parametrem je jméno souboru jenž se má linkovat. Mezi důležité volby linkeru které budou dále v textu použity patří:

`-o, output`	Jméno souboru jenž bude výstupem linkeru.
`-oPE`	Výstup bude ve formátu Win32 PE (tedy soubor .EXE).
`-dll`	Výstup bude ve formátu DLL.
`-subsys`	Nastaví jaký subsystém bude přeložena aplikace používat. Pod volba GUI pak znamená grafickou aplikaci.

Příklad, linkování do formátu Win32 PE:

```
alink okno.obj dial.res -oPE -subsys gui
```

Linkuje soubor okno.obj a dial.res do formátu Win32 PE a ve výsledném programu budou použity prvky z grafického uživatelského rozhraní. Výstupní soubor se bude jmenovat okno.exe.

Příklad, linkování do formátu DLL:

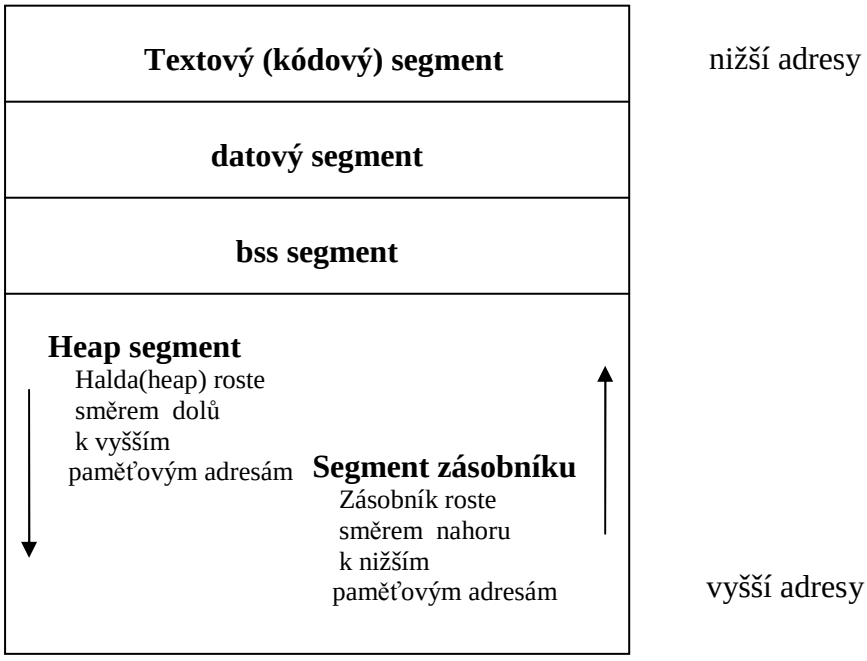
```
alink okno.obj -oPE –dll –o knihovna.dll
```

Linkuje soubor okno.obj do formátu dynamicky linkované knihovny. Výstupní soubor se bude jmenovat okno.dll.

1.6 Programování funkcí

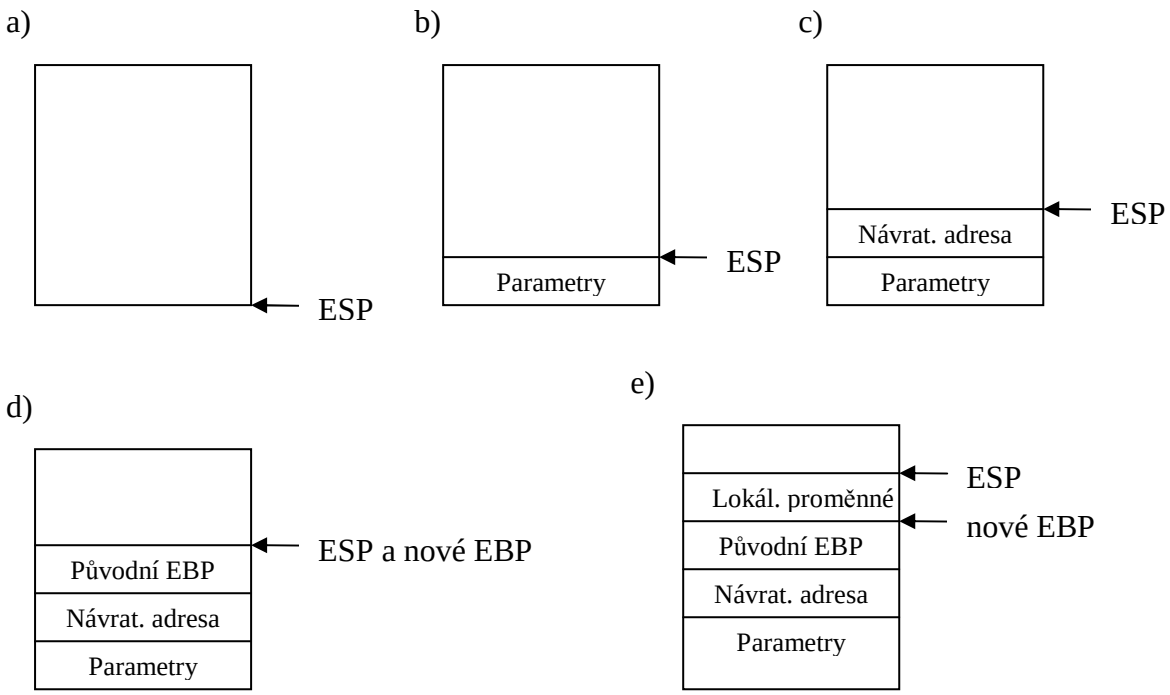
V následující části jsou použity pojmy volající a volaný. Volající je označení pro funkci, jež provádí volání funkce a Volaný je volaná funkce.

Základními operacemi při programování funkcí jsou operace prováděné nad zásobníkem. Zásobník je část paměti typu FILO v operační paměti. Klasické rozdělení paměti podle Von Neumannovi architektury je pak zobrazeno na obrázku 1.1.



Obr. 1.1: Rozdělení paměti.

Lze vidět že zásobník roste směrem nahoru k nižším paměťovým adresám. Zásobník se používá jako dočasné úložiště pro kontext během volání funkcí. Při volání funkce je potřeba alokovat místo v paměti pro parametry funkce, návratovou adresu, zálohovat registr EBP a alokovat místo pro lokální proměnné funkce. Vše je detailněji zobrazeno na obrázku 1.2.



Obr. 1.2: Obsah zásobníku při volání podprogramu. a) před voláním, b) po uložení parametrů, c) po uložení návratové adresy, d) záloha ESP, e) po uložení lokální proměnné

Na obr. 1.2 a) registr ESP ukazuje na vrchol zásobníku, jelikož je zásobník prázdný tak ukazuje na konec paměti. V hlavním programu pak dojde k přípravě na volání funkce. Instrukcemi PUSH se do zásobníku uloží, podle dohodnuté konvence, parametry volené funkce, obr 1.2 b). Následuje, obr. 1.2 c), instrukce CALL která podobně jako JMP skočí na návěští uvedené v operandů instrukce. Rozdíl je však v tom, že instrukce CALL uloží hodnotu adresu následují instrukce na zásobník, tedy zvýší hodnotu registru EIP o délku instrukce CALL a uloží jí na zásobník. Této hodnotě EIP se říká návratová adresa.

Dalším krokem podle obr. 1.2 d) je uložení registru EBP a zkopírování ESP do EBP. Registr EBP slouží, stejně jako ESP, k adresaci paměti v zásobníku. Adresace pomocí ESP je však značně komplikovaná, protože se hodnota ESP mění s každým přístupem do zásobníku. Tedy offset potřebný k adresaci se také neustálé mění. Proto se uvnitř podprogramu adresuje pomocí fixního registru EBP. Tento registr vždy ukazuje na začátek paměti v zásobníku určeného pro podprogram. Potřebný offset k adresaci bude tedy vždy stejný. Obsah původního registru EBP se ukládá proto, aby mohlo docházet k rekurzivnímu volání funkcí. Části paměti v zásobníkovém okně, která obsahuje parametry funkce, návratovou adresu funkce a původní obsah registru EBP, se říká okno zásobníku, anglicky termín je stack frame.

Po registru EBP se na zásobník uloží lokální proměnné funkce. K tomu lze použít instrukce SUB ESP,M nebo ENTER M,0. Obě sníží hodnotu registru ESP o M a tedy alokují počet bytů určený hodnotou M. Počet bytů je dán počtem proměnných vynásobených hodnotou 4. Všechny datové typy na rozhraní Win32 mají velikost 32 bitů, tedy 4 byte.

Po ukončení podprogramu se instrukcí LEAVE do ESP uloží EBP, tedy ESP bude ukazovat na začátek prostoru v paměti jenž byla vyhrazena pro lokální proměnné, obr. 1.2.d). Následně se obnoví obsah EPB původním EPB, obr. 1.2 c). Posledním krokem je pak skok do volajícího části programu a úklid zásobníku po parametrech podprogramu.

Návratová hodnota funkce je pak nejčastěji uložena v registru EAX:EDX, EAX, AX nebo AL v závislosti na návratovém typu.

Popis a význam ostatních segmentu je mimo rámec tohoto textu, jejich popis lze nalézt [10].

Dvě věci, které dosud nebyly záměrně zmíněny, jsou: v jakém pořadí se ukládají parametry a kdo uklízí obsah paměti určené pro parametry volaného podprogramu. Rozhoduje o tom použitá konvence volání funkce. Existuje několik konvencí volání. Tři nejběžnější popisuje tabulka 1.1.

Tab. 1.1: Nejběžnější konvence volání.

Název konvence volání	Ukládání parametrů funkce	Úklid zásobníku.	Použití
STDCALL	zprava do leva	Volaný uklízí zásobník.	Stdcall je standardní konvence volání ve Win32 API.
FASTCALL (Microsoft fastcall)	První dva argumenty funkcí jsou v ECX a EDX, ostatní se ukládají zprava do leva.	Volaný uklízí zásobník.	Toto systémové volání není standardizováno. Microsoft fastcall se také používá v kompilátorech GCC.
CDECL	zprava do leva	Volající uklízí zásobník.	Používá se v systémech naprogramovaných v jazyce C pro x86 architekturu.

Způsob ukládání parametru pak může být zprava do leva nebo zleva doprava a úklid zásobníku může provádět volaný nebo volající. V následujícím výkladu bude použita konvence StdCall, tedy konvence, kterou používá Win32 API. StdCall ukládá parametry podprogramu na zásobník zprava doleva. Zásobník pak uklízí volaný podprogram, slouží k tomu instrukce RET. Operandem této instrukce je počet bytů v paměti, jenž má být uklizen. Počet bytů je rovno počtu parametrů funkce násobených čtyřmi.

Příklad, funkce suma:

Je potřeba naprogramovat v jazyce symbolických instrukcí funkci, která sečte tři dvojslova. V jazyce C by pak funkční prototyp takové funkce vypadal následovně:

```
int suma (int param1,int param2,int param3);
```

Je předpoklad že int má rozměr 32 bitového čísla. Možná implementace volání takové funkce by mohla vypadat takto:

```
push dword [param3]
push dword [param2]
push dword [param1]
call suma
```

A implementace samotné funkce v jazyce symbolických instrukcí by vypadala následovně:

```
suma:
      push ebp
      mov ebp,esp
      sub esp,4              ;jedna lokální proměnná

      mov eax,[EBP + 8]
      add eax,[EBP + 12]
      add eax,[EBP + 16]
      mov [EBP – 4],eax
```

```
      mov esp, ebp
      pop ebp
ret 12                        ;tři parametry funkce
```

Trojici instrukcí na začátku funkce lze nahradit jedinou instrukcí ENTER 4,0. První parametr funkce je na adrese [EBP + 8], druhý na [EBP + 12] a poslední na [EBP + 16]. Po sečtení je výsledek uložen do lokální proměnné. Současně je výsledek v registru EAX, kde jej bude i číst volající jako návratovou hodnotu. Dvojicí instrukci MOV a POP se obnoví obsah registrů EBP a ESP na původní stav před voláním funkce. Opět je možno tyto instrukce nahradit jedinou instrukcí LEAVE. Následuje úklid zásobníku, to má na starosti podle konvence stdcall volaná funkce. Pokud by úklid měl na starosti volající, tak by po instrukci CALL musela následovat instrukcí ADD ESP,12 která provede úklid.

1.7 Knihovny DLL

Dynamicky linkovaná knihovna je druh knihovny, kdy se při linkování do výsledného souboru ukládají pouze tabulky odkazů na symboly definované v dynamické knihovně. Oproti staticky linkované knihovně je pak rozdíl v tom, že dll knihovna musí být k dispozici během spuštění programu. Výhoda je pak v menší velikosti linkovaného programu a jednoduší aktualizaci knihovny. Knihovna pak totiž muže být aktualizována nezávisle na programu. Mezi další výhodu pak patří skutečnost, že více programů může přistupovat k jediné instanci dll knihovny v paměti počítače. Nevýhoda dll je pak v rychlosti spuštění programů s dynamicky linkovanou knihovnou, čas potřebný ke spuštění je trochu delší.

Psaní dll knihoven pak spočívá v tom, že námi napsané funkce se vyexportují do formátu dll. Funkce z těchto knihoven je pak možno použít v hlavním modulu, ale také v úplně jiném programu který navíc může být napsán v libovolném programovacím jazyce (který samozřejmě umí pracovat s DLL). Kromě funkcí mohou tyto knihovny také obsahovat data, například proměnné, struktury, atd.

Při psaní DLL knihoven v jazyce symbolických adres je potřeba dodržovat konvenci volání funkcí a dále pak velikosti proměnných funkce a velikost návratové hodnoty funkce. Převážná většina funkcí v DLL knihovnách rozhraní Win32 dodržuje konvenci STDCALL a velikost všech proměnných je vždy 32 bitů.

Překladač NASM obsahuje dvě direktivy pro tvorbu DLL knihoven, jedná se o direktivy global a export. Direktiva global dává na vědomí, že funkce (nebo obecně nějaký symbol) bude přístupná ostatním modulům při linkovaní programu. Druhá direktiva export pak učiní funkci součást dll knihovny. Direktiva global má jeden povinný parametr a to jméno funkce. Direktiva export má pak parametry dva, první je jméno funkce a druhý pak jméno, pod kterým bude funkce volána z knihovny dll.

Příklad, vytvoření DLL knihovny:

```
      global get_mac
      export get_mac GetMac
```

Direktivou global je řečeno, že funkce get_mac bude přístupná ostatním modulům při linkovaní. Direktiva export pak vyexportuje funkci do dll formátu pod jménem GetMac. Funkce GetMac pak bude přístupná v jiných programech.

Závěrem kapitoly je potřeba říci, že Win32 API obsahuje velké množství typů proměnných a struktur. I když všechny proměnné typy ve Win32 API jsou dvojslova (32 bitové čísla), tak jejich datové typy jsou zcela nelogicky rozdílné. Při programování ve vyšším programovacím jazyku pak neustálé dochází k přetypování. V programování jazyce

symbolických instrukcí se pak stačí držet faktu, že všechny použité typy proměnných mají vždy rozměr 32 bitového čísla a tedy zcela ignorovat datový typ.

Pro struktury však toto neplatí, jednotlivé položky struktury mohou mít rozdílnou velikost datového typu. Kompletní seznam struktur a konstant obsahuje soubor win32n.inc. Standardně je tento soubor distribuován s překladačem NASM.

Příklad, importování souboru win32n.inc:

```
%include 'N:\work\work\win32n.inc'
```

Direktivou %include preprocesu NASM jsou deklarovány struktury a konstanty umístěné v souboru win32n.inc.

2 Základní práce s okny

2.1 Importování externích funkcí

Prvním krokem při programování win32 aplikací je importování funkcí win32 rozhraní. Většina funkcí, která bude použita, se nacházejí v dynamicky linkovaných knihovnách (DLL). Konkrétně v kernel32.dll a user32.dll.

Importovaní funkcí z těchto knihoven se provádí direktivou IMPORT. Tato direktiva má tři parametry. První dva jsou povinné a třetí nepovinný. První parametr určuje jméno funkce (obecně pak jméno symbolu), která má být vložena. Druhý pak jméno knihovny ve které se nachází. Třetí, nepovinný pak udává jméno, pod kterým je funkce známa v knihovně, ze které je funkce importována. Pokud není uveden třetí parametr, tak jsou shodná obě jména funkcí.

Dále je potřeba překladač NASM informovat, že importovaný symbol se nachází v jiném modulu. K tomu slouží direktiva extern.

Příklad, importováni funkci 1:

```
Extern WSAStartup
import WSAStartup Ws2_32.dll
```

Direktiva import importuje funkci WSAStartup z knihovny wsock32.dll. Pod stejným jménem pak bude funkce volána v programu.

Příklad, importováni funkcí 2:

```
Extern GetMessage
import GetMessage user32.dll GetMessageA
```

Direktiva import importuje funkci GetMessageA z knihovny user32.dll. V programu však tato funkce bude volána pod jménem GetMessage.

2.2 Definice třídy okna

Z pohledu filozofie programovaní ve Win32 je každý samostatný prvek oknem. Tímto samostatným prvkem může být například hlavní okno, podokna, tlačítka, list boxy, atd. Tedy instance nějaké třídy okna. Třída okna tedy určuje základní vlastnosti okna. Před samotným vytvářením instancí třídy okna je potřeba tuto třídu definovat a následně pak registrovat.

Definicí třídy je myšleno vyplnění struktury WNDCLASSEX. Struktura je definována následovně:

Tab. 2.1: Struktura WNDCLASSEX.

Syntaxe:	Popis poměných:
STRUC WNDCLASSEX	
.cbSize RESD 1	Velikost struktury v bytech.
.style RESD 1	Styl třídy okna (viz lit. [10]).
.lpfnWndProc RESD 1	Ukazatel na WndProc(viz kapitola 3.3).
.cbClsExtra RESD 1	Počet alokovaných extra bytů struktury.
.cbWndExtra RESD 1	Počet alokovaných extra bytů instance okna.
.hInstance RESD 1	Handle instance programu.
.hIcon RESD 1	Handle ikony okna.
.hCursor RESD 1	Handle kurzoru okna.
.hbrBackground RESD 1	Handle pozadí okna (viz lit. [14]).
.lpszMenuName RESD 1	Ukazatel na nulou ukončený řetězec který obsahuje název menu.
.lpszClassName RESD 1	Ukazatel na nulou ukončený řetězec který obsahuje název třídy.
.hIconSm RESD 1	Handle malé ikony okna.
ENDSTRUC	

Některé handle jsou známy až při běhu programu, konkrétně se jedna o hInstance, hIcon, hCursor a hIconSm. Proto se hodnoty těchto proměnných nastavují při definici na hodnotu NULL.

Příklad, definice WNDCLASSEX struktury:

```
WndClass:
      istruc WNDCLASSEX
            at WNDCLASSEX.cbSize, dd WNDCLASSEX_size
            at WNDCLASSEX.style, dd CS_VREDRAW+ CS_HREDRAW + CS_GLOBALCLASS
            at WNDCLASSEX.lpfnWndProc, dd WndProc
            at WNDCLASSEX.cbClsExtra, dd 0
            at WNDCLASSEX.cbWndExtra, dd 0
            at WNDCLASSEX.hInstance, dd NULL
            at WNDCLASSEX.hIcon, dd NULL
            at WNDCLASSEX.hCursor, dd NULL
            at WNDCLASSEX.hbrBackground, dd COLOR_BACKGROUND
            at WNDCLASSEX.lpszMenuName, dd NULL
            at WNDCLASSEX.lpszClassName, dd szWndClassName
            at WNDCLASSEX.hIconSm, dd NULL
      iend
```

Konstanta WNDCLASSEX_size obsahuje velikost struktury v bytech, tuto konstantu vytvořil automaticky NASM z deklarace struktury WNDCLASSEX. Hodnotou této konstanty je pak inicializována proměnná cbSize.

Proměnná style je inicializována součtem bitového pole. V souboru Win32.inc jsou CS_VREDRAW,CS_HREDRAW a CS_GLOBALCLASS deklarovány následovně:

```
                                    ; binární vyjádření:
CS_VREDRAW equ 1h                   ; 00000000 00000000 00000000 00000001
CS_HREDRAW equ 2h                   ; 00000000 00000000 00000000 00000010
CS_PUBLICCLASS equ 4000h            ; 00000100 00000000 00000000 00000000
CS_GLOBALCLASS equ CS_PUBLICCLASS   ; 00000100 00000000 00000000 00000000
                                    ; součet:
                                    ; 00000100 00000000 00000000 00000011
```

Tedy proměnná style pak obsahuje 4003h. Jsou tedy nastaveny bity, jimž odpovídají konstanty CS_HREDRAW, CS_VREDRAW a CS_GLOBALCLASS. CS_HREDRAW a CS_VREDRAW informuji systém, že má překreslit vnitřní okno pří horizontální nebo vertikální změně velikosti okna. CS_GLOBALCLASS informuje systém, že se jedna o aplikační globální třídu.

Ukazatel na proceduru okna (WndProc) je uložen do proměnné lpfnWndProc. Touto problematikou se zabývá kapitola 3.

Handle pozadí okna je inicializován konstantou COLOR_BACKGROUND, tedy barvou pozadí okna. Tato barva je určena systémem a uživatel ji může změnit v Ovládací panely, Zobrazení.

Název třídy je inicializován hodnotou szWndClassName, která je v datovém segmentu definována následovně:

```
szWndClassName: DB "WIN CLASS32",0
```

Ostatní hodnoty jsou inicializovány hodnotou NULL nebo 0.

Handl na instanci programu, ikony a kurzoru okna se získá až za běhu programu. Funkce GetModuleHandle slouží k získání handle instance programu. Její jediným parametrem je název modulu. Pro zjištění handle právě běžícího programu (modulu) je tato funkce volaná s parametrem NULL.

Tab. 2.2: Funkce GetModuleHandle.

Syntaxe:	Popis poměných:
HMODULE WINAPI GetModuleHandle(__in_opt LPCTSTR lpModuleName);	Jméno zavedeného modulu. Pro získání handle právě běžící instance programu má proměnná hodnotu NULL.
Návratová hodnota:	Návratovou hodnotou je handle modulu.
Implementováno v ANSI a Unicode verzi. Funkce se nachází v Kernel32.dll.	

Příklad, získání handle programu:

```
push dword NULL
call [GetModuleHandle]
mov [hInstance],eax
mov [WndClass + WNDCLASSEX.hInstance],eax
```

Instrukcemi push a call zavoláme funkci GetModuleHandle s argumentem NULL. Návratová hodnota je uložena v registru EAX a obsahuje handle instance programu. Handle instance je pak nakopírován do proměnné [hInstance] a dále pak do struktury WndClass.

Funkce LoadIcon a LoadCursor popsané v tabulkách 2.3 a 2.5 slouží k získání handle na ikonu okna respektive na kurzor okna. Pokud bude použit některý z předdefinovaných stylů, některé ze stylů popisují tabulky 2.4 a 2.6, bude obsahovat první argument obou funkcí hodnotu NULL.

Tab. 2.3: Funkce LoadIcon.

Syntaxe:	Popis poměných:
HICON LoadIcon(HINSTANCE hInstance, LPCTSTR lpIconName);	Handle na instance programu který obsahuje ikonu k nahrání. Pro použití předdefinovaných ikon má proměnná hodnotu NULL. Ukazatel na nulou ukončený řetězec, který obsahuje jméno ikony, jenž má být nahrána.
Návratová hodnota:	V případě úspěchu vrací funkce handle na nově nahranou ikonu. V případě selhání funkce vrací NULL (je možno volat GetLastError pro dodatečné informace).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Tab. 2.4: Předdefinované ikony.

Jméno předdefinované ikony.	Stručný popis:
IDI_APPLICATION	Defaultní ikona aplikace.
IDI_ASTERISK	Ikona s hvězdičkou (stejná ikona jako IDI_INFORMATION).
IDI_ERROR	Ikona s tvarem ruky.
IDI_EXCLAMATION	Ikona s vykřičníkem (stejná ikona jako IDI_WARNING).
IDI_HAND	Ikona s tvarem ruky (stejná ikona jako IDI_ERROR).
IDI_INFORMATION	Ikona s hvězdičkou.
IDI_QUESTION	Ikona s otazníkem.
IDI_WARNING	Ikona s vykřičníkem.
IDI_WINLOGO	Ikona s logem Windows: defaultní ikona aplikace.
IDI_SHIELD	Ikona se štítem.

Tab. 2.5: Funkce LoadCursor.

Syntaxe:	Popis poměných:
HCURSOR LoadCursor(HINSTANCE hInstance, LPCTSTR lpCursorName);	Handle na instance programu, který obsahuje kurzor k nahrání. Pro použití předdefinovaných ikon má proměnná hodnotu NULL. Ukazatel na nulou ukončený řetězec, který obsahuje jméno ikony, jenž má být nahrána.
Návratová hodnota:	V případě úspěchu vrací funkce handle na nově nahraný kurzor. V případě selhání funkce vrací NULL (je možno volat GetLastError pro dodatečné informace).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Tab. 2.6: Předdefinované kursory.

Jméno předdefinovaného kurzoru.	Stručný popis:
IDC_APPSTARTING	Standardní šipka se symbolem přesýpacích hodin.
IDC_ARROW	Standardní šipka.
IDC_CROSS	Kurzor ve tvaru kříže.
IDC_HAND	Kurzor ve tvaru ruky.
IDC_HELP	Standardní šipka s otazníkem.
IDC_UPARROW	Vertikální šipka.
IDC_WAIT	Přesýpací hodiny.

Příklad, získání handle ikony okna a kurzoru:

```
push dword IDI_QUESTION
push dword NULL
call [LoadIcon]
mov [WndClass + WNDCLASSEX.hIcon],eax

push dword IDC_CROSS
push dword NULL
call [LoadCursor]
mov [WndClass + WNDCLASSEX.hCursor],eax
```

Opět instrukcemi push a call byly zavolány, při dodržení konvence volání stdcall, funkce LoadIcon a LoadCursor. Oba dva handly (tedy návratové hodnoty) byli zkopírovány do struktury WndClass.

2.3 Registrace třídy okna

Po definici třídy je potřeba třídu zaregistrovat, k tomu slouží funkce RegisterClassEx a RegisterClass. Funkce RegisterClassEx je novější a proto bude použita v následujícím textu namísto RegisterClass. Tabulka 2.7 popisuje chování funkce RegisterClassEx.

Tab. 2.7: Funkce RegisterClassEx.

Syntaxe:	Popis poměných:
ATOM RegisterClassEx(CONST WNDCLASSEX *lpwcx);	Ukazatel na WNDCLASSEX strukturu (tato struktura musí být vyplněna před voláním).
Návratová hodnota:	V případě neúspěchu vrací funkce nulu (k získáni dodatečných informací slouží funkce GetLastError). V případě úspěchu je návratovou hodnotou třída atom která jednoznačně identifikuje registrované okno.
Funkce se nachází v user32.dll.	

Jediným argumentem třídy je ukazatel na vyplněnou strukturu typu WNDCLASSEX.

Příklad, registrace třídy okna:

```
      push dword WndClass
      call [RegisterClassEx]
      test eax,eax
      jz near .Finish
```

Je zavolána funkce RegisterClassEx, instrukcí TEST je pak testována návratová hodnota funkce uložena v registru EAX. Pokud EAX obsahuje nulu tak se instrukcí JZ skočí na návěští .Finish.

2.4 Vytvoření okna

Vytvořením okna je myšleno přiřazení pamětí pro definovanou třídu okna. Ve Win32 jsou k dispozici funkce CreateWindow a CreateWindowEx, kterými lze vytvořit okno. CreateWindowEx je shodná s CreateWindow, ale navíc obsahuje rozšířené styly pro vytvoření okna. Funkce CreateWindowEx je detailněji popsána v tabulce 2.8.

Tab. 2.8: Funkce CreateWindowEx.

Syntaxe:	Popis poměných:
HWND CreateWindowEx(	
DWORD dwExStyle,	Definuje rozšířený styl okna.
LPCTSTR lpClassName,	Ukazatel na nulou ukončený řetězec, který obsahuje název třídy.
LPCTSTR lpWindowName,	Ukazatel na nulou ukončený řetězec, který obsahuje název okna.
DWORD dwStyle,	Definuje styl okna.
int x,	Horizontální pozice levého horního rohu okna, který se má vytvořit.
int y,	Vertikální pozice levého horního rohu okna, který se má vytvořit.
int nWidth,	Horizontální šířka okna.
int nHeight,	Vertikální šířka okna.
HWND hWndParent,	Handle na rodiče nebo vlastníka vytvářeného okna. Pokud žádné mateřské okno neexistuje, tak je předána hodnota NULL.
HMENU hMenu,	Handle na nabídku okna
HINSTANCE hInstance,	Handle instance (programu) který vytvoří okno.
LPVOID lpParam	Ukazatel na data, která budou předána jako LPARAM zprávy WM_CREATE.
);	

Parametry x a y jsou měřeny v bodech, takže displej s rozlišením 1280×800 by měl 1280 horizontálních a 800 vertikálních bodů. Výchozí pozice závisí na tom, zda se jedná o okno mateřské, nebo o okno dceřiné. V případě okna mateřského je výchozím bodem levý horní roh obrazovky. V případě okna dceřiného (a také všech ovládacích prvků) je výchozím bodem levý horní roh mateřského okna.

Příklad, vytvoření mateřského okna:

```
[section .data class=DATA use32 align=16]
…
      szWndClassName: DB "WIN CLASS32",0
      szWndCaption: DB "okno",0
…
```

```
[section .code use32 class=CODE]
…
      push dword NULL
      push dword [hInstance]
      push dword NULL
      push dword NULL
      push dword 275
      push dword 595
      push dword 100
      push dword 100
      push dword WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX
      push dword szWndCaption
      push dword szWndClassName
      push dword 0
      call [CreateWindowEx]

      test eax,eax
      jz near .Finish
      mov [hWnd],eax
…
```

V datovém segmentu jsou nejprve definovány dva nulou ukončené datové řetězce. Ukazatel szWndClassName ukazuje na název třídy, registrovaný funkcí RegisterClassEx a ukazatel szWndCaption ukazuje na název okna.

V kódovém segmentu pak následují instrukce PUSH, kterými se plní argumenty funkce CreateWindowEx. Význam jednotlivých argumentů je uveden v tabulce 2.8.

Použitý styl charakterizuje okno, které lze překrýt jiným oknem (WS_OVERLAPPED), má svůj titulek (WS_CAPTION), systémové menu (WS_SYSMENU). V tomto menu jsou ale aktivní jen tlačítka pro minimalizaci a zavření okna. Okno nejde maximalizovat, to proto, že nebyl použit styl WS_MAXIMIZEBOX. Minimalizace okna funguje, styl WS_MINIMIZEBOX použit byl. Výpis všech možných stylů je možno nalézt například v lit. [1], [11]. Je nutno poznamenat, že styly oken jsou závislé na třídě okna, tedy zda se jedná například o tlačítko nebo seznam. V tomto příkladu byla použita skupina stylu pro okna, proto prefix WS (windows style). Například pro tlačítka je použit prefix BS (button style).

Nakonec je testována návratová hodnota funkce, pokud funkce vrátí nulu tak se skočí na návěští .Finish, v opačném případě se uloží handle právě vytvořeného okna do proměnné [hWnd].

Název třídy lpClassName je buď vytvořen pomocí funkce RegisterClassEx nebo je jedním z předdefinovaných názvu tříd podle tabulky 2.9.

Tab. 2.9: Předdefinované třídy oken.

Název třídy:	Popis:
Button	Třída pro tlačítko.
ComboBox	Třída pro combo box, kombinace seznamu a editačního tlačítka.
Edit	Třída pro edit, ovládací prvek pro editaci textu.
ListBox	Třída pro posuvný seznam.
MDIClient	Třída pro MDI (Multiple Document Interface) klientského okna.
ScrollBar	Třída pro scrollbar, posuvník.
Static	Třída pro statický text.

Příklad, vytvoření tlačítka:

```
[section .data class=DATA use32 align=16]

      szClsButn: DB "BUTTON",0
      szCButn: DB "Odesli!",0

[section .code use32 class=CODE]

      push dword NULL
      push dword [hInstance]
      push dword NULL
      push dword [hWnd]
      push dword 20
      push dword 70
      push dword 220
      push dword 510
      push dword WS_CHILD | BS_DEFPUSHBUTTON | WS_VISIBLE | WS_TABSTOP
      push dword szCButn
      push dword szClsButn
      push dword 0
      call [CreateWindowEx]
      mov [hButton],EAX
```

Vytvoření tlačítka se příliš neliší od vytvoření okna. Rozdíl je v handlu na rodiče okna. V případě mateřského okna byla hodnota NULL, nyní má však hodnotu [hwnd] (handle okna).

Pro název třídy szClsButn byl použita jedna z předdefinovaných tříd a proto nemusela být volána funkce RegisterClassEx pro registraci. Posledním krokem je pak uložení handle na právě vytvořené tlačítko.

Na tlačítku bude zobrazen text “Odesli!“.

Použitý styl WS_CHILD říká, že se jedna o dceřiné okno, styl BS_DEFPUSHBUTTON pak definuje standardní vlastnosti tlačítka. WS_VISIBLE říká, že tlačítko bude viditelné v okně rodiče. WS_TABSTOP definuje vlastnost tlačítka, kdy po stisku klávesy TAB může být prvek zaměřen. Tedy dochází ke změně zaměření dceřiného okna mezi všemi dceřinými okny které mají definovaný styl WS_TABSTOP. Je důležité poznamenat, že se jedna o styl dialogového okna a ne standardního okna. Proto samotné definovaní tohoto stylu u jednotlivých dceřiných prvků okna nestačí pro změnu zaměření prvku pomocí klávesy TAB. Dialogové okna jsou detailněji prostudovány v kapitole 8.

Příklad, vytvoření textového pole:

```
[section .data class=DATA use32 align=16]

      szClsEdit:  DB "EDIT",0
      szCEdit:    DB 0

[section .code use32 class=CODE]

      push dword NULL
      push dword [hInstance]
      push dword NULL
      push dword [hWnd]
      push dword 20
      push dword 330
      push dword 220
      push dword 170
      push dword WS_CHILD | WS_BORDER | WS_VISIBLE | WS_TABSTOP
```

```
      push dword szCEdit
      push dword szClsEdit
      push dword WS_EX_CLIENTEDGE
      call [CreateWindowEx]
      mov [hEdit],EAX
```

Zdrojový kód v příkladě vytvoří editační pole pro vkládání textu. Pole bude dlouhé 330 pixelů a vysoké 20 pixelů. Levý horní roh tohoto pole bude ležet na souřadnicích 170:220 (x:y v pixelech).

Styl WS_BORDER říká, že editační pole bude mít okraje a rozšířený styl WS_EX_CLIENTEDGE pak tvar těchto okrajů dále modifikuje, po použití toho stylu jsou okraje editačního pole propadlé.

Nakonec je handle vytvořeného okna nahrán do proměnné [hEdit].

Příklad, vytvoření posuvného seznamu:

```
[section .data class=DATA use32 align=16]

      szClsListbox: DB "LISTBOX",0

[section .code use32 class=CODE]

      push dword NULL
      push dword [hInstance]
      push dword NULL
      push dword [hWnd]
      push dword 205
      push dword 570
      push dword 10
      push dword 10
      push dword LBS_DISABLENOSCROLL | WS_VISIBLE | WS_CHILD | WS_BORDER |
WS_VSCROLL
      push dword NULL
      push dword szClsListbox
      push dword WS_EX_CLIENTEDGE
      call [CreateWindowEx]
      mov [hListbox],EAX
```

Tento zdrojový kód na vytvoření listoboxu se od předchozích liší jen minimálně. Použitý styl WS_VSCROLL zviditelní posuvník a styl LBS_DISABLENOSCROLL pak zobrazí posuvník i když nevzniká potřeba rolování textu v seznamu.

Příklad, vytvoření statického textu:

```
[section .data class=DATA use32 align=16]

      szClsStatic: DB "STATIC",0
      szCStatic: DB "Zprava k odeslani:",0

[section .code use32 class=CODE]

      push dword NULL
      push dword [hInstance]
      push dword NULL
      push dword [hWnd]
      push dword 15
      push dword 150
      push dword 223
```

```
        push dword 10
        push dword WS_CHILD | WS_VISIBLE | SS_CENTER
        push dword szCStatic
        push dword szClsStatic
        push dword 0
        call [CreateWindowEx]
```

Třída STATIC slouží k vytvoření statického textu. V příkladě je použita tato třída k zobrazení textu "Zprava k odeslani:" na souřadnice 10:223 (x:y v pixelech) mateřského okna.

Styl SS_CENTER centruje text doprostřed bloku učeného pro zobrazení textu.

Vytvořené okno, včetně všech ovládacích prvků (dceřiných oken), je zobrazeno na obrázku 2.1.

Obr. 2.1: Vytvořené hlavní okno

2.5 Zobrazení okna

Jakmile je okno vytvořeno je vhodné jej zobrazit. K tomuto účelu poskytuje rozhraní Win32 funkci ShowWindow. Jejím prvním parametrem je handle okna, které se má zobrazit a druhým parametrem je pak styl zobrazení. Zobrazením je myšleno nastavení příznaku stylu WS_VISIBLE příslušného okna. Například lze touto funkcí v průběhu běhu programu okno minimalizovat.

Tab. 2.10: Funkce ShowWindow.

Syntaxe:	Popis poměných:
BOOL ShowWindow(HWND hWnd, int nCmdShow);	 Handle okna které se má zobrazit. Specifikuje jak má být okno zobrazeno.
Návratová hodnota:	Pokud okno bylo již zobrazeno, tak vrací nenulovou hodnotu. Pokud okno bylo skryté, tak vrací nulu.
Funkce se nachází v user32.dll.	

Tab. 2.11: Styly zobrazení funkce ShowWindow.

Styl zobrazení:	Stručný popis:
SW_FORCEMINIMIZE	Minimalizuje okno, i když okno neodpovídá (pouze ve Windows 2000/XP).
SW_HIDE	Schová okno a aktivuje další okno.
SW_MAXIMIZE	Maximalizuje okno.
SW_MINIMIZE	Minimalizuje okno.
SW_RESTORE	Aktivuje a zobrazí okno. Při minimalizaci/maximalizaci si okno uchovává aktuální pozici a velikost.
SW_SHOW	Aktivuje a zobrazí v jeho současné velikosti a pozici.
SW_SHOWDEFAULT	Zobrazí okno podle původního stylu.
SW_SHOWMAXIMIZED	Maximalizuje a zobrazí okno.
SW_SHOWMINIMIZED	Minimalizuje a zobrazí okno.
SW_SHOWMINNOACTIVE	Stejně jako SW_SHOWMINIMIZED jen okno není aktivováno.
SW_SHOWNA	Stejně jako SW_SHOW jen okno není aktivováno.
SW_SHOWNOACTIVATE	Stejně jako SW_SHOWNORMAL jen okno není aktivováno.
SW_SHOWNORMAL	Aktivuje a zobrazí okno. Při minimalizaci/maximalizaci si okno uchovává aktuální pozici a velikost. Používá se při prvním zobrazení.

Po zobrazení okna vzniká někdy potřeba plochu klientského okna aktualizovat. K tomu slouží funkce UpdateWindow. Tato funkce zašle zprávu WM_PAINT přímo proceduře okna (WndProc) cílového okna a obchází při tom Frontu zpráv aplikace. Samozřejmě má volání této funkce smysl pouze tehdy, když je tato zpráva v proceduře okna explicitně obsloužena.

Tab. 2.12: Funkce UpdateWindow

Syntaxe:	Popis poměných:
BOOL UpdateWindow(HWND hWnd);	Handle okna, které se má být aktualizováno.
Návratová hodnota:	V případě úspěchu vrací funkce nenulovou hodnotu. V případě selhání funkce vrací nulu (je možno volat GetLastError pro dodatečné informace).
Funkce se nachází v user32.dll.	

Příklad, zobrazení a aktualizování okna:

```
push dword SW_SHOWNORMAL
push dword [hWnd]
call [ShowWindow]
push dword [hWnd]
call [UpdateWindow]
```

Kód v příkladu zobrazí a aktualizuje okno, jenž je specifikováno handlem v proměnné [hWnd].

2.6 Vybrané vlastnosti oken

2.6.1 Nastavení fontu

Stock object jsou objekty předdefinovaných fontů, per, štětců a palet. Funkcí GetStockObject pak lze získat handle takového objektu. Funkce má jediný parametr a to konstantu definující typ objektu, pro písmo pevné délky je definována konstanta OEM_FIXED_FONT.

Tab. 2.13: Funkce GetStockObject.

Syntaxe:	Popis poměných:
HGDIOBJ GetStockObject(int fnObject);	Identifikátor typu objektu (viz lit. [16]).
Návratová hodnota:	V případě úspěchu vrací funkce handle objektu. V případě selhání funkce vrací NULL (je možno volat GetLastError pro dodatečné informace).
Funkce se nachází v Gdi32.lib.	

Příklad, získání handle fontu:

```
push dword OEM_FIXED_FONT
call [GetStockObject]
mov [hFont],eax
```

Je získán handle na font pevné délky, o konkrétním typu fontu rozhoduje nastavení OS. Handle je následně uložen do proměnné [hFont].

K nastavení fontu v jednotlivých oknech slouží funkce SendMessage. Tato funkce je popsána tabulkou 2.14 a neslouží jen pro nastavení fontu, ale obecně ke změně nějaké vlastnosti okna. Funkce zašle zprávu oknu identifikovaným svým handlem, typ zprávy je pak určen druhým parametrem této funkce. Funkce volá proceduru okna a čeká na zpracování zprávy vláknem. Opakem je pak funkce PostMessage, tato funkce nečeká na zpracování zprávy vláknem, jinak je shodná s funkcí SendMessage.

Tab. 2.14: Funkce SendMessage.

Syntaxe:	Popis poměných:
LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);	 Handle okna kterému je zaslána zpráva. Identifikátor zprávy (druh zprávy). Dodatečné informace o zprávě. Dodatečné informace o zprávě.
Návratová hodnota:	Obsahuje informace o výsledku předání zprávy (závisí na druhu poslané zprávy).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Tab. 2.15: Funkce PostMessage.

Syntaxe:	Popis poměných:
BOOL PostMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);	 Handle okna kterému je zaslána zpráva. Identifikátor zprávy (druh zprávy). Dodatečné informace o zprávě. Dodatečné informace o zprávě.
Návratová hodnota:	Vrací nenulovou hodnotu pokud funkce proběhne úspěšně. Vrací nulu pokud nastane chyba (k získáni dodatečných informací slouží funkce GetLastError).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Příklad, nastavení fontu:

```
      push dword 1
      push dword [hFont]
      push dword WM_SETFONT
      push dword [hListbox]
      call [SendMessage]
```

V příkladu je zaslána zpráva posuvnému seznamu. Typ zprávy je WM_SETFONT, nastav font. Parametr wParam obsahuje handle fontu a lParam signalizuje zda se má font má ihned překreslit, hodnota 1 (tedy TRUE) font ihned po obdržení zprávy překreslí.

2.6.2 Plnění posuvného seznamu

Plnění posuvného seznamu se provádí zaslání zprávy LB_ADDSTRING.

Příklad, přidání textu do posuvného seznamu:

```
[section .data class=DATA use32 align=16]

      INIC_LIST:        DB    "Prijate data:",0
      INIC_LIST_size    dd    14

[section .code use32 class=CODE]

      push dword INIC_LIST
      push dword [INIC_LIST_size]
      push dword LB_ADDSTRING
      push dword [hListbox]
      call [SendMessage]
```

V příkladu je zaslána zpráva posuvnému seznamu. Typ zprávy je LB_ADDSTRING, přidej řetězec. Parametr wParam obsahuje proměnou s délkou řetězce a lParam je ukazatel na nulou ukončený řetězec.

2.6.3 Práce s posuvníkem v posuvném seznamu

Cílem následujícího příkladu je, aby se posuvník automaticky roloval směrem dolu spolu s přidávaným textem do seznamu. Tohoto efektu lze dosáhnout zasláním zpráv LB_GETCOUNT a LB_SETTOPINDEX. První zpráva zjistí počet položek v seznamu a zpráva LB_SETTOPINDEX pak zajistí, aby zvolená položka byla viditelná, pokud je to potřeba tak dojde k rolování posuvníku.

Příklad, práce s posuvníkem v posuvném seznamu:

```
push dword 0
push dword 0
push dword LB_GETCOUNT
push dword [hListbox]
call [SendMessage]
dec eax

push dword 0
push dword eax
push dword LB_SETTOPINDEX
push dword [hListbox]
call [SendMessage]
```

Po zaslání zprávy LB_GETCOUNT je do registru EAX uloženo počet položek v seznamu. Jelikož počet položek je číslován od jedné a index položky v posuvném seznamu od nuly, tak je nutné snížit hodnotu registru EAX o jedna. Zpráva LB_SETTOPINDEX pak zajisti, aby daný index položky byl viditelný. Parametr wParam tedy obsahuje index položky.

2.6.4 Limitace textu v editačním poli

Příklad, limitace textu v editačním poli:

```
push dword 0
push dword [TEXT_length]
push dword EM_SETLIMITTEXT
push dword [hEdit]
call [SendMessage]
```

Zpráva EM_SETLIMITTEXT nastaví maximální počet znaků které lze napsat do editačního pole. Délka, maximální počet znaků, je pak určen parametrem wParam funkce SendMessage.

2.6.5 Čtení a zápis v editačním poli

V příkladu této podkapitoly bude ukázán způsob, jak lze text z editačního pole zkopírovat do vlastní proměnné a následně pak způsob vymazáni editačního pole.

Příklad, čtení a zápis v editačním poli:

```
[section .data class=DATA use32 align=16]

      SEND_TEXT:              resb  40
      SEND_TEXT_size          dd    40
```

```
      szCEdit:                DB    0

[section .code use32 class=CODE]

      push dword SEND_TEXT
      push dword [SEND_TEXT_size]
      push dword WM_GETTEXT
      push dword [hEdit]
      call [SendMessage]

      push dword szCEdit
      push dword 0
      push dword WM_SETTEXT
      push dword [hEdit]
      call [SendMessage]
```

Nejprve je zaslána zpráva editačnímu poli WM_GETTEXT. Tato zpráva slouží k získání textu z editačního pole. Parametr wParam určuje maximální počet znaků co bude zkopírován, včetně znaku 0. lParam pak obsahuje ukazatel na alokované místo v paměti, kam bude text uložen. Proměnná [SEND_TEXT_size] obsahuje hodnotu 40. Do editačního pole lze tedy napsat maximálně 39 znaků.

Zpráva WM_SETTEXT slouží k nastavení textu v editačním poli. V příkladu je použit trik, kdy je zaslán prázdný řetězec, obsahuje jen 0. Dojde tedy k přepsaní textu prázdným řetězcem, tedy k vymazání textu v editačním poli.

2.6.6 Nastavení zaměření

Zaměření, anglický termín Focus, je stav okna kdy přímá znaky ze standardního vstupu, tedy z klávesnice. K nastavení zaměření slouží funkce SetFocus.

Tab. 2.16: Funkce SetFocus.

Syntaxe:	Popis poměných:
HWND SetFocus(HWND hWnd);	Handle okna jenž obdrží vstup z klávesnice.
Návratová hodnota:	V případě úspěchu vrací funkce handle okna, které mělo zaměření před volání této funkce. V případě selhání funkce vrací NULL (je možno volat GetLastError pro dodatečné informace).
Funkce se nachází v User32.lib.	

Příklad, nastavení zaměření:

```
      push dword [hEdit]
      call [SetFocus]
```

Funkce SetFocus nastaví zaměření na editační pole.

3 Smyčka zpracování zpráv

3.1 Definice zprávy

Zpráva je datová struktura, která je deklarována následovně:

Tab. 3.1: Struktura MSG.

Syntaxe:	Popis poměných:
STRUC MSG	
.hwnd RESD 1	Handle okna, kterému je zaslána zpráva.
.message RESD 1	Identifikátor zprávy (druh zprávy).
.wParam RESD 1	Dodatečné informace o zprávě.
.lParam RESD 1	Dodatečné informace o zprávě.
.time RESD 1	Čas odeslání zprávy.
.pt RESB POINT_size	Pozice kurzoru v době odesláni zprávy.
ENDSTRUC	

Zpráva tedy v sobě nese adresu příjemce (tedy handle okna) a další informace podle tabulky 3.1. Dodatečné informace obsahují wParam a lParam. Tyto informace závisí na typu poslané zprávy.

Příklad, struktura MSG:

Mějme zprávu typu WM_LBUTTONDOWN (uživatel stisknul levé tlačítko na myši) a je potřeba zjistit, které okno tuto zprávu poslalo. To proto, aby bylo možno zjistit kam uživatel kliknul. Tuto informaci obsahuje proměnná lparam.

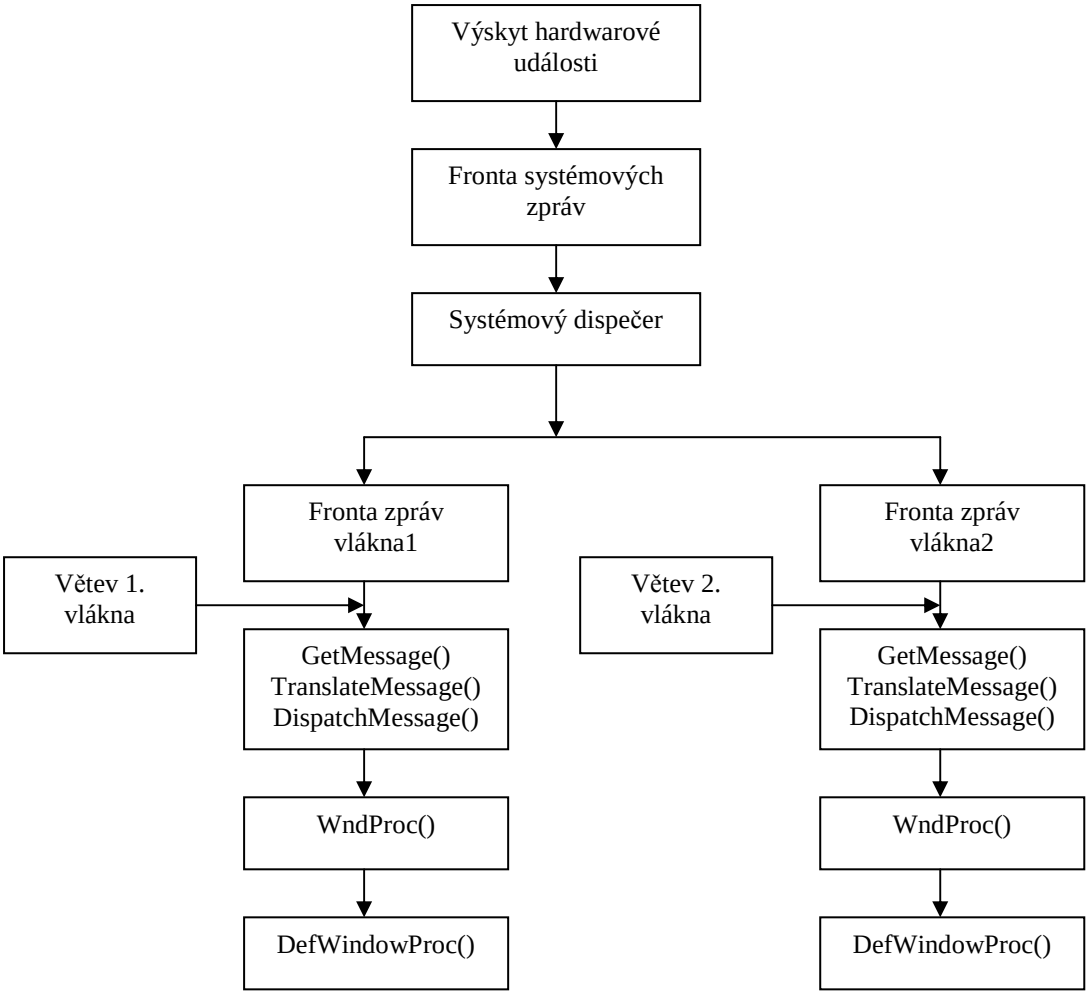
V následují kód zjišťuje, zda uživatel kliknul na tlačítko jehož handle obsahuje proměnná [hButton].

```
mov eax,[hButton]
cmp [lParam],eax
jne .Finish
```

První instrukcí je nahrán handle tlačítka z proměnné [hButton] do registru EAX. Dále se pak instrukcí CMP porovná proměnná [lParam] a registr EAX. Pokud jejich hodnoty nejsou shodné, tak se instrukcí JNE provede skok na návěští .Fininsh.

3.2 Smyčka zpráv

Tok zpráv v OS Windows je zobrazen na obrázku 3.1. Windows generuje zprávy pro každou hardwarovou událost (stisk klávesy, posun myší). Tato zpráva je pak předána do Fronty systémových zpráv. Systémový dispečer (scheduler) má pak na starostí jednotlivé zprávy přiřazovat příslušným vláknům. Jakmile zpráva dorazí do vlákna příslušné aplikace tak je nejprve uložena ve Frontě zpráv vlákna. Tam čeká do doby než je vyzvednuta funkcí GetMessage (nebo PeekMessage). Následně je pak zpráva zpracována funkcemi TranslateMessage a DispatchMessage, tyto funkce budou prostudovány dále v textu. V posledních dvou blocích je pak zpráva zpracována procedurou okna a defaultní procedurou okna. Opět tyto funkce budou podrobně prostudovány v následujícím textu.



Obr. 3.1: Vývojový diagram zpráv ve Windows

Vlákno nebo proces čtou zprávy z Fronty v tzv. Smyčce zpracování zpráv (message loop). Jedná se o nekonečnou smyčku, kde na začátku smyčky je načtena zpráva z Fronty zpráv pro vlákno a v těle smyčky je pak zpracována.

V jazyce C by pak smyčka zpracování zpráv vypadala následovně:

```
While( GetMessage(&msg, NULL, 0, 0)) {
      TranslateMessage();
      DispatchMessage();
}
```

Funkce GetMessage slouží k načtení zprávy a tělo cyklu While pak provede zpracování zprávy. Základní vlastností této funkce je, že uspí proces do příchodu zprávy.

Tab. 3.2: Funkce GetMessage.

Syntaxe:	Popis poměných:
BOOL GetMessage(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax);	 Ukazatel na MSG strukturu do které je uložena zpráva z fronty zpráv. Handle okna jehož zprávy budou zachyceny (pokud je hodnota NULL, tak budou zachytávány zprávy pro všechny okna aktuálního vlákna) . Minimální číselná hodnota zprávy, která bude přijata (proměnná slouží k filtrování zpráv) . Maximální číselná hodnota zprávy, která bude přijata (proměnná slouží k filtrování zpráv) .
Návratová hodnota:	Funkce vrací nulu v případě ukončení aplikace (zpráva WM_QUIT). V případě vzniku chyby vrací hodnotu -1 (k získání dodatečných informací je volána funkce GetLastError). V ostatních případech vrací hodnotu mimo 0 a -1.
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Funkce TranslateMessage překládá virtuální klávesy na klávesy znakové. Tedy přeloží se kód stisknuté klávesy klávesnice do znakové reprezentace (formát UNICODE) Funkce generuje zprávy WM_CHAR, WM_SYSCHAR, WM_DEADCHAR a WM_DEADCHAR.

Tab. 3.3: Funkce TranslateMessage.

Syntaxe:	Popis poměných:
BOOL TranslateMessage(const MSG *lpMsg);	Ukazatel na MSG strukturu.
Návratová hodnota:	Pokud je zpráva přeložena, tak funkce vrací nenulovou hodnotu, v opačném případě vrací nulu.
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Funkce DispatchMessage pak posílá data zprávy do procedury okna (WndProc). DispatchMessage zná adresu funkce WndProc, protože WndProc je asociována z daty okna (viz kapitola 2.2). Tedy při definici třídy okna je uvedeno která funkce bude procedurou okna.

Tab. 3.4: Funkce DispatchMessage.

Syntaxe:	Popis poměných:
LRESULT DispatchMessage(const MSG *lpmsg)	Ukazatel na MSG strukturu.
Návratová hodnota:	Vrací hodnotu, kterou vrací funkce zpracování zprávy.
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

K načtení zpráv z Fronty zpráv pro vlákna slouží také funkce PeekMessage. Funkci PeekMessage popisuje tabulka 3.5. Rozdíl oproti funkci GetMessage je ten, že PeekMessage nečeká na výskyt zprávy jako GetMessage, ale funkce vrací nulu, pokud pro daný proces není ve Frontě zpráv vlákna žádná zpráva. Vlákno se pak může zabývat jinou činností (například vykreslování scény pozadí). PeekMessage se upřednostňuje u aplikací, které velmi citlivě reagují na velké množství instrukcí (například počítačové hry).

Tab. 3.5: Funkce PeekMessage.

Syntaxe:	Popis poměných:
BOOL PeekMessage(	
LPMSG lpMsg,	Ukazatel na MSG strukturu, do které je uložena zpráva z fronty zpráv.
HWND hWnd,	Handle okna, jehož zprávy chceme zachytit (pokud je hodnota NULL, tak zachytáváme zprávy pro všechny okna aktuelního vlákna).
UINT wMsgFilterMin,	Minimální číselná hodnota zprávy, která bude přijata (proměnná slouží k filtrování zpráv).
UINT wMsgFilterMax,	Maximální číselná hodnota zprávy, která bude přijata (proměnná slouží k filtrování zpráv).
UINT wRemoveMsg	Určuje zda má být zpráva z fronty odebrána po provedení funkce PeekMessage.
);	
Návratová hodnota:	Pokud je ve smyčce zprava, tak funkce vrátí nenulovou hodnotu. Pokud není ve smyčce zprava, tak funkce vrátí nulu.
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Někdy je potřeba ošetřit zprávy určené dialogovému oknu standardním způsobem. Tím je myšleno například to, aby po stisku klávesy TAB se přepínalo zaměření mezi okny které mají definovaný styl WS_TABSTOP. Dialogové okna budou prostudovány dále v kapitole 8. Funkce IsDialogMessage pak rozhoduje, zda je zpráva určena pro dialogové okno a pokud ano, pak ji zpracuje a vrátí 0.

Tab. 3.6: Funkce IsDialogMessage.

Syntaxe:	Popis poměných:
BOOL IsDialogMessage(	
HWND hDlg,	Handle dialogového okna.
LPMSG lpMsg	Ukazatel na MSG strukturu.
);	
Návratová hodnota:	Pokud funkce zpracuje zprávu tak je návratovou hodnotou nula. Jinak funkce vrátí nenulovou hodnotu.
Funkce se nachází v user32.dll.	

Příklad, smyčka zpráv:

```
.MessageLoop:
      push dword 0
      push dword 0
      push dword NULL
      push dword Message
      call [GetMessage]
```

```
            test eax,eax
            jz near .Finish
            cmp eax,-1
            jz near .Finish

            push dword Message
            push dword [hWnd]
            call [IsDialogMessage]
            test eax,eax
            jnz near .MessageLoop


            push dword Message
            call [TranslateMessage]
            push dword Message
            call [DispatchMessage]
            jmp .MessageLoop

      .Finish:
            push dword 0
            call [ExitProcess]
```

Příklad demonstruje smyčku zpráv napsanou v jazyce symbolických instrukcí. Jak již bylo řečeno jedná se o nekonečnou smyčku jenž začíná návěštím .MessageLoop. Poté následuje volání funkce GetMessage, která naplní strukturu Message. Je zde kontrolována návratová hodnota, a pokud je rovna 0 nebo -1, tak je smyčka zpráv ukončena skokem na návěští .Finish.

Následně je testováno zda je zpráva určena pro dialogové okno, pokud ano tak je návratovou hodnotou této funkce 0 a instrukcí jnz je proveden skok zpět na začátek smyčky zpracování zpráv.

Dále následuje volání funkcí TranslateMessage a DispatchMessage jenž mají jako argument strukturu Message. Návratovou hodnotu těchto funkcí není potřeba kontrolovat. Nakonec funkcí DispatchMessage je zpráva (tedy struktura Message) předána proceduře okna.

Na návěští .Finish je volána funkce ExitProcess pro ukončení procesu, tedy aplikace. Funkce ExitProcess je lépe popsána tabulkou 3.7. Návratový kód 0 informuje OS, že aplikace skončila korektně.

Tab. 3.7: Funkce ExitProcess.

Syntaxe:	Popis poměných:
VOID WINAPI ExitProcess(__in UINT uExitCode);	Ukončující kód procesu.
Návratová hodnota:	Funkce nemá návratovou hodnotu.
Funkce se nachází v kernel32.dll.	

3.3 Procedura okna

Procedura okna je nejdůležitější části programu pracujícího s okny. Probíhá zde interakce uživatele s programem. Stiskne-li uživatel klávesu nebo například pohne myší je tato informace v podobě struktury WNDCLASSEX předána z Fronty zpráv vlákna skrz smyčku zpráv do procedury okna, kde dojde k jejímu programovému zpracování.

Procedura okna bývá někdy pojmenována jako Funkce obsluhy zpráv, nebo anglicky WndProc (Window's procedure).

O tom, která funkce bude procedurou okna se určuje při definici třídy okna, kdy hodnota WNDCLASSEX.lpfnWndProc je inicializována ukazatelem na proceduru okna.

Příklad, procedura okna:

Pro obsluhu zpráv je potřeba vytvořit funkci jejíž funkční prototyp by v jazyce C vypadal následovně:

```
LRESULT WndProc(HWND hWnd,UINT wMsg,WPARAM wParam,LPARAM lParam);
```

Argumenty takové funkce jsou některé z parametrů struktury WNDCLASSEX. hWnd je handle okna jemuž je zpráva zaslána, wMsg je identifikátor zprávy a wparam/lparam jsou pak dodatečné informace zprávy.

Funkční prototyp nemůže být libovolný. To proto, že je dán způsobem volání této funkce funkcí DispatchMessage.

Dále je potřeba pří definici třídy okna správně inicializovat ukazatel na proceduru okna.

```
WndClass:
istruc WNDCLASSEX
      at WNDCLASSEX.cbSize, dd WNDCLASSEX_size
      at WNDCLASSEX.style, dd CS_VREDRAW +CS_HREDRAW + CS_GLOBALCLASS
      at WNDCLASSEX.lpfnWndProc, dd WndProc
      at WNDCLASSEX.cbClsExtra, dd 0
      at WNDCLASSEX.cbWndExtra, dd 0
      at WNDCLASSEX.hInstance, dd NULL
      at WNDCLASSEX.hIcon, dd NULL
      at WNDCLASSEX.hCursor, dd NULL
      at WNDCLASSEX.hbrBackground, dd COLOR_BACKGROUND
      at WNDCLASSEX.lpszMenuName, dd NULL
      at WNDCLASSEX.lpszClassName, dd szWndClassName
      at WNDCLASSEX.hIconSm, dd NULL
iend
```

Funkce pro obsluhu zpráv pak může v jazyce symbolických instrukcí vypadat takto:

```
WndProc:
      WndProc_PARAMBYTES EQU 16
      enter 0,0

      mov eax,dword [ebp+12]

      cmp eax,WM_DESTROY
            je near .Destroy
      cmp eax,WM_CREATE
            je near .Create
      cmp eax,WM_CLOSE
            je near .Destroy
      cmp eax,WM_LBUTTONDOWN
            je near .Lbutton
      cmp eax,WM_COMMAND
            je near .Command

      push dword [ebp+20]
      push dword [ebp+16]
      push dword [ebp+12]
```

```
      push dword [ebp+8]
      call [DefWindowProc]

      leave
      ret WndProc_PARAMBYTES


      .Create:
            ;obslužná rutina pro návěstí .Create
      .Destroy:
            ;obslužná rutina pro návěstí . Destroy
      .Lbutton:
            ;obslužná rutina pro návěstí . Lbutton
      .Command:
            ;obslužná rutina pro návěstí . Command
      .Finish:
            xor eax,eax
            leave
            ret WndProc_PARAMBYTES
```

Způsobem popsaným v kapitole 1 je v příkladu vytvořena funkce WndProc. První instrukcí v těle této funkce je zkopírovaní druhého argumentu ([ebp+12], proměnná [wMsg]) do registru EAX. To proto, aby byly urychleny následující instrukce. Poté následuje blok testovaní druhu zprávy(typu události). Bývá dobrým zvykem ošetřit minimálně události WM_DESTROY a WM_CLOSE. Další události, jenž jsou v příkladu uvedeny, jsou WM_CREATE, WM_LBUTTONDOWN a WM_COMMAND.

Události WM_CREATE a WM_DESTROY mohou sloužit jako konstruktor a destruktor. Událost WM_CREATE je volána při vytvoření okna a WM_DESTROY naopak při jeho zničení. WM_CLOSE je volána při zavření okna.

WM_LBUTTONDOWN je volána při stisku levého tlačítka na myši.

Zpráva WM_COMMAND je volána když uživatel vyvolá nějakou interakci s ovládacím prvkem hlavního okna. Stisk tlačítka například vyvolá zprávu WM_COMMAND.

Pro každou z výše popsaných události je pak vytvořeno návěští, které obsahuje obslužnou rutinu. Například zprávy WM_DESTROY a WM_CLOSE jsou obslouženy na jediném návěští .Destroy. Zdrojový kód takové obslužné rutiny může vypadat následovně:

```
      .Destroy:
            push dword 0
            call [PostQuitMessage]
```

Procedurou PostQuitMessage je ukončeno aktuální vlákno. Podrobnější popis této procedury je v tabulce 3.8.

Pokud nastane situace, která není ošetřena v proceduře okna, tak je volána funkce DefWindowProc, jenž se postará o obsluhu této události.

Tab. 3.8: Procedura PostQuitMessage.

Syntaxe:	Popis proměnných:
void PostQuitMessage(int nExitCode);	Specifikátor výstupního kódu aplikace.
Procedura se nachází v user32.dll.	

Tab. 3.9: Funkce DefWindowProc.

Syntaxe:	Popis proměnných:
LRESULT DefWindowProc(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam);	 Handle na okno, jehož procedura zpracovává zprávu. Identifikátor zprávy (druh zprávy). Dodatečné informace o zprávě. Dodatečné informace o zprávě.
Návratová hodnota:	Návratová hodnota není jednoznačně dána, ale závisí na typu zprávy (proměnná Msg).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

4 Úvod do síťového programovaní

4.1 Rodina protokolů TCP/IP

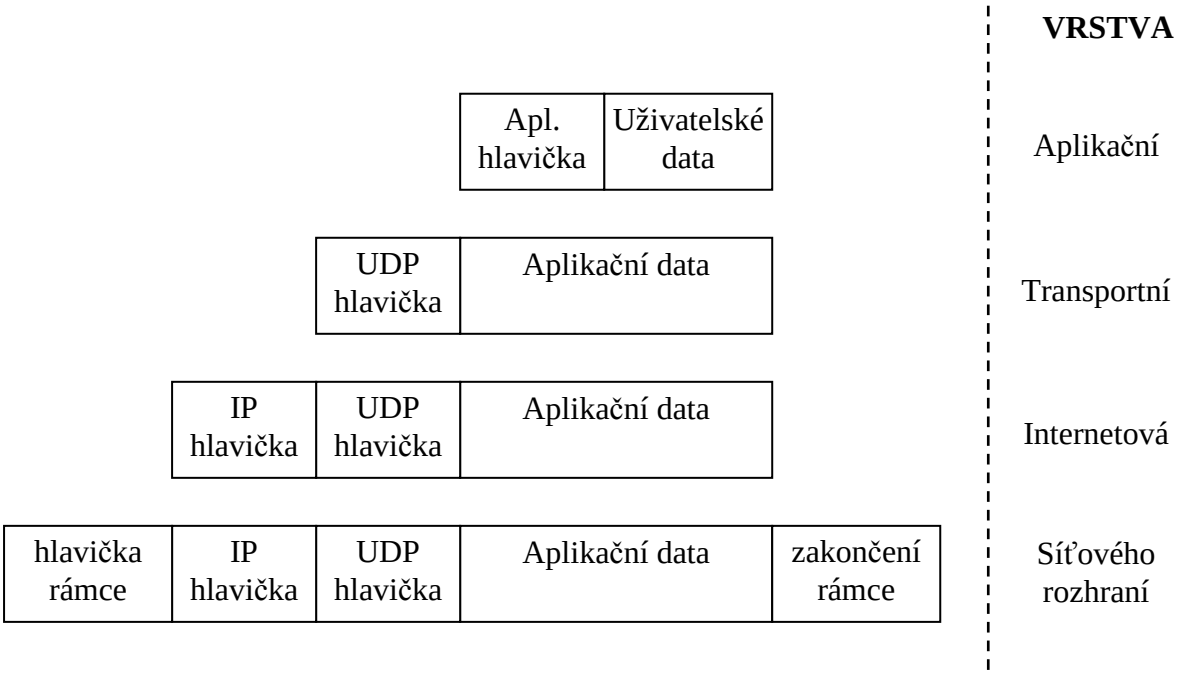
Protokoly této rodiny jsou v dnešní době nejpoužívanější protokoly k realizaci síťové komunikace, tvoří základ Internetu. Model TCP/IP pracuje pouze se čtyřmi vrstvami. Což je rozdíl oproti modelu ISO/OSI, ten má vrstev 7. Jednotlivé vrstvy, spolu s průběhem obalovaní uživatelských dat, lze vidět na obrázku 4.1.

Protokol na aplikační vrstvě může být například http. Uživatelské data pak obsahují text a formátovací značky webové stránky. Aplikační hlavička obsahuje titulek okna, meta informace a například použité kódování znaků.

Protokol transportní vrstvy na obrázku 4.1 je UDP, pro spojově orientovaný přenos lze místo něj použít TCP. V následujících podkapitolách budou oba prostudovány. Aplikační hlavička a uživatelské data jsou na transportní vrstvě označována jako aplikační data. K těmto datům je vytvořena hlavička.

Stejným způsobem jsou data obalena na internetové vrstvě. Jednotlivé položky hlavičky IP paketu popisuje tabulka 4.1. Význam s popisem lze najít [20]. Pomocí IP adresy pak probíhá adresace v Internetu. IP adresa má 4B (IPv4), nebo 6B (IPv6).

Poslední vrstva, vrstva síťového rozhraní, má na starosti vše, co je spojeno s ovládáním konkrétní přenosové cesty resp. sítě, a s přímým vysíláním a příjmem datových paketů. V rámci soustavy TCP/IP není tato vrstva blíže specifikována, neboť je závislá na použité přenosové technologii. Nejčastěji používaný protokol této vrstvy je Ethernet, ethernetová vrstva. Tabulka 4.2 popisuje ethernetový rámec. Popis a význam jednotlivých položek lze nalézt [21] [22].



Obr. 4.1: Obalování dat.

Tab. 4.1: Ethernetový rámec

Velikost v bytech:	7	1	6	6	2	46-1500	4
Popis:	preambule	SFD	cílová MAC adresa	zdrojová MAC adresa	typ / délka	data (payload)	CRC

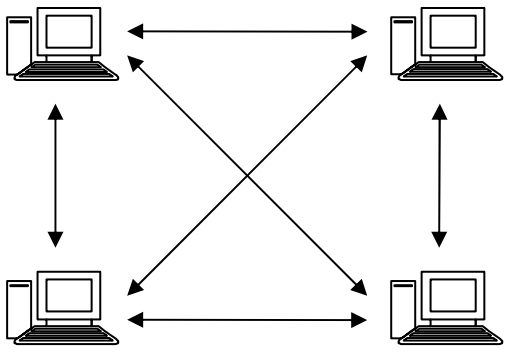
Tab. 4.2: IP hlavička

Bitový offset	Bity 0–3	4–7	8–15	16–18	19–31
0	Verze	Délka hlavičky	Typ služby	Celková délka	
32	Identifikace			Příznaky	Fragment Offset
64	TTL	Protokol	Kontrolní součet		
96	Cílová adresa				
128	Zdrojová adresa				
160	Volby				
160/192+	Data				

4.2 Architektura sítě

4.3.1 Peer to peer

Architekturou sítě je myšlen způsob propojení jednotlivých entit na stejných vrstvách. Buď jsou si jednotlivé entity rovny, architektura peer to peer. Nebo je jedna entita nadřazené druhé a pak se mluví o architektuře Client-server, viz následující podkapitola. Příkladem architektury Peer to peer jsou výměnné sítě (P2P). Čistá architektura Peer to peer vůbec nezná pojem Server [23]. Obrazek 4.2 popisuje tuto architekturu.



Obr. 4.2: Peer to peer architektura

Pro peer to peer síťové aplikace pracující s rodinou protokolů TCP/IP je vhodné namísto transportního protokolu TCP použít UDP. To proto, že UDP je nespojový protokol. Není potřeba vytvářet a udržovat spojení s každým klientem. Adresace na úrovni transportní protokolu probíhá skrze čísla UDP portu. UDP hlavičku popisuje tabulka 4.3 a význam jednotlivých položek lze nalézt [24].

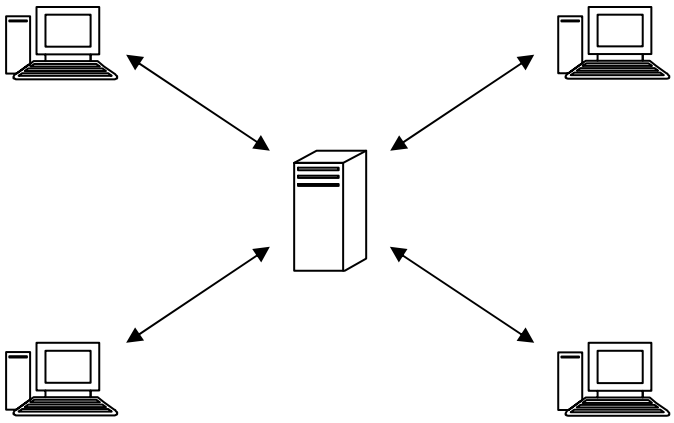
Tab. 4.3: UDP hlavička

Bitový offset	Bity 0 - 15	16 - 31
0	Zdrojový port	Cílový Port
32	Délka	Kontrolní součet
64	Data	

4.3.2 Client-server

Pokud mezi entitami existuje nějaký vztah nadřazenosti na odpovídajících vrstvách, tak se mluví o Client-server architektuře. Příklad může být http protokol. Klient posílá požadavky na stránky která chce uživatel navštívit (příkaz GET) a server požadavek zpracuje a stránky poskytne.

K implementaci této architektury v rodině protokolů TCP/IP se používá protokol TCP. Oproti UDP poskytuje spolehlivý přenos dat. Spojový charakter TCP protokolu je také výhodou, stačí jen jednou navázat spojeni se serverem a pak jen zasílat aplikační data. Více informací o TCP protokolu lze nalézt [25].



Obr. 4.3: Client-server architektura

Tab. 4.4: TCP hlavička

Bitový offset	Bity 0–3	4–7	8–15	16–18	19–31
0	Zdrojový port			Cílový port	
32	Sekvenční číslo				
64	Potvrzovací číslo				
96	Datový offset	Rezervováno	Příznaky	Velikost okna	
128	Kontrolní součet			Urgent ukazatel	
160	Volby (nepovinné)				
160/192+	Data				

4.3 Získání MAC adresy

Z předchozích podkapitol a prostudování hlaviček jednotlivých protokolů si lze udělat představu jaké informace jdou vyčíst z jednotlivých protokolů. Nejčastěji se jedná o IP adresu příjemce a odesílatele. Dále často požívanými údaji bývají čísla portů. Situace se získaní MAC adresy odesílatele je ale komplikovanější. MAC adresu lze sice vyčíst z hlavičky ethernetového rámce. Ale tato MAC adresa nemusí být adresou odesílatele. Pokud se odesílatel nachází mimo LAN, tedy například za Gateway (branou), tak položka MAC adresa v ethernetovém rámci bude obsahovat MAC adresu této brány a ne skutečného odesílatele. Tento jev je dán dynamickým přiřazováním MAC adres k IP adresám, konkrétně pak protokolem ARP.

Řešením může být přidání MAC adresy odesílatele do uživatelských dat. Je zřejmé, že odesílatel musí svou MAC adresu nejprve získat . K tomu lze použit NetBIOS. NetBIOS je softwarový interface (API) pro přístup k síťové komunikaci na úrovni relační (model ISO/OSI) vrstvy [26]. NetBIOS nepracuje s adresami síťových uzlů ze síťové vrstvy, ale místo něj využívá logických jmen. OS Windows Vista a Windows Server 2008 toto rozhraní již neobsahuje.

K tomuto rozhraní se přistupuje pomocí volání funkce Netbios (tabulka 4.5). Funkce má pouze jeden argument a to ukazatel na strukturu NCB. Tabulka 4.6 popisuje strukturu NCB velmi zjednodušeně. To proto, že většina položek v této struktuře je závislá na typu příkazu, těch je 24 a výpis všech možných variant je nad rámec tohoto textu. Podrobnější popis lze nalézt [27]. Následující příklady lépe vysvětlují práci s rozhraním NetBIOS.

Tab. 4.5: Funkce Netbios.

Syntaxe:	Popis poměných:
void Netbios(__in PCNB pcnb);	Ukazatel na NCB strukturu.
Návratová hodnota:	Funkce nemá návratovou hodnotu.
Funkce se nachází v netapi32.dll.	

Tab. 4.6: Struktura NCB.

Syntaxe:	Popis poměných:
STRUC NCB	
.ncb_command RESB 1	Číselný kód, který reprezentuje typ příkazu. Jejich seznam lze nalézt [27].
.ncb_retcode RESB 1	Specifikátor návratového kódu. Jejich seznam a význam lze nalézt [27].
.ncb_lsn RESB 1	Identifikátor lokálního relačního čísla.
.ncb_num RESB 1	Specifikátor čísla jména lokální sítě.
.ncb_buffer RESD 1	Ukazatel na buffer. Význam je závislý na typu příkazu.
.ncb_length RESW 1	Velikost bufferu v bytech.
.ncb_callname RESB 16	Jméno volaného uzlu.
.ncb_name RESB 16	Jméno lokálního uzlu.
.ncb_rto RESB 1	Doba vypršení pro operace příjmu.
.ncb_sto RESB 1	Doba vypršení pro operace odesílání.
.ncb_post RESD 1	Adresa na funkci která bude volána po ukončení asynchronního příkazu.
.ncb_lana_num RESB 1	Specifikátor LAN adapteru.
.ncb_cmd_cplt RESB 18	Stejný význam jako ncb_retcode. Specifikátor návratového kódu. Jejich seznam a význam lze nalézt [27].
.ncb_reserve10 RESB 18	Rezervované místo. Na 32-bit architektuře je rezervováno 10B.
.ncb_event RESD 1	Handle na objekt události. Proměnná slouží k práci se synchronními nebo asynchronními příkazy.
ENDSTRUC	

Příklad, získáni seznamu LAN adapteru:

```
[section .data class=DATA use32 align=16]

      ncb:        resb  NCB_size
      lenum:      resb  LANA_ENUM_size

[section .code use32 class=CODE]

      mov byte [ncb + NCB.ncb_command],NCBENUM
      mov dword [ncb + NCB.ncb_buffer],lenum
      mov word [ncb + NCB.ncb_length],LANA_ENUM_size
      push dword ncb
      call [Netbios]
```

V datovém segmentu je alokováno místo v paměti pro struktury NCB a LANA_ENUM. Ncb a lenum jsou ukazatele na tyto struktury.

Příkazem je kód reprezentovaný konstantou NCBENUM. Příkaz slouží k získání seznamu LAN adaptéru. Funkce Netbios po zavolání vyplní strukturu lenum seznamem všech LAN adapteru které se na PC vyskytují. Délka toho bufferu je definována konstantou LANA_ENUM_size. Strukturu LANA_ENUM lze nalézt v tabulce 4.7.

Pokud má PC pouze jeden adapter tak pole ve struktuře má pouze jeden záznam a proměnná iLenght obsahuje hodnotu jedna. V případě více adaptéru je nutné zvolit ten správný adapter. Cílem volání této funkce je získání čísla adaptéru přes které PC komunikuje.

Tab. 4.7: Struktura LANA_ENUM.

Syntaxe:	Popis poměných:
STRUC LANA_ENUM	
.iLength RESB 1	Počet LAN adapteru.
.lana RESB 255	Pole s platnými čísly LAN adaptéru.
ENDSTRUC	

Příklad, resetování LAN adapteru:

```
;vymazání struktury ncb
mov ecx,NCB_size
xor eax,eax
mov edi,ncb
rep
  stosb

;resetování lana
mov byte [ncb + NCB.ncb_command],NCBRESET
mov byte [ncb + NCB.ncb_lana_num],03
push dword ncb
call [Netbios]
```

Před tím než je možno použít NCBASTAT k získaní informací o konkrétním LAN adaptéru je potřeba tento adaptér resetovat. Obecně platí, že před každým příkazem pro konkrétní adapter (specifikován položkou ncb_lana_num) je nutné tento adaptér resetovat.

První blok příkazu v tomto příkladu vymaže strukturu ncb. Po minulém volání, příkaz NCBENUM, obsahuje struktura irelevantní informace a může dojít k selhání volání této funkce.

Následně je zvolen typ příkazu NCBRESET a číslo LAN adapteru. Toto číslo bylo získáno v minulém příkazu.

Příklad, získání informací o LAN adapteru:

```
[section .data class=DATA use32 align=16]

      adap:       resb  ADAPTER_STATUS_size

[section .code use32 class=CODE]

;vymazáni struktury ncb
mov ecx,NCB_size
xor eax,eax
mov edi,ncb
rep
stosb

;vyplnění struktury adap
mov byte [ncb + NCB.ncb_callname],0x2A
mov byte [ncb + NCB.ncb_command],NCBASTAT
mov byte [ncb + NCB.ncb_lana_num],03
mov dword [ncb + NCB.ncb_buffer],adap
mov word [ncb + NCB.ncb_length],ADAPTER_STATUS_size
push dword ncb
call [Netbios]
```

V datovém segmentu je nejprve alokováno místo pro strukturu ADAPTER_STATUS, adap je ukazatel na alokované místo v paměti pro tuto strukturu. Popis této struktury lze nalézt [28]. Poté je vymazána instance struktury ncb.

Jméno volaného uzlu musí být "*", kód 0x2A je ASCII kód znaku “*“. Příkaz je NCBASTAT a slouží k získání informací o LAN adapteru. Číslo adaptéru je 3. Do proměnné ncb_buffer je nahrán ukazatel adap, tedy ukazatel na instanci struktury ADAPTER_STATUS. Velikost je pak uložena do proměnné ncb_length.

Po zavolání funkce Netbios je vyplněna struktura adap. Prvních 6B obsahuje MAC adresu.

Příklad, zformátování výsledku:

```
[section .data class=DATA use32 align=16]

      my_mac:           resb  20
      my_fmac:          DB    "[%02x-%02x-%02x-%02x-%02x-%02x]",0
      SEND_TEXT:        resb  67
      SEND_fTEXT:       DB    "[*%02x-%02x-%02x-%02x]:%51s",0

[section .code use32 class=CODE]

;zkopírováni a zformátovaní MAC adresy do řetězce my_mac
xor eax,eax
mov   byte al,[adap+5]
push dword eax
mov   byte al,[adap+4]
push dword eax
mov   byte al,[adap+3]
push dword eax
mov   byte al,[adap+2]
push dword eax
mov   byte al,[adap+1]
push dword eax
mov   byte al,[adap]
push dword eax
push dword my_fmac
push dword my_mac
call [wsprintf]

;zkopírováni a zformátovaní MAC adresy do řetězce SEND_TEXT
push dword name
mov   byte al,[adap+5]
push dword eax
mov   byte al,[adap+4]
push dword eax
mov   byte al,[adap+3]
push dword eax
mov   byte al,[adap+2]
push dword eax
push dword SEND_fTEXT
push dword SEND_TEXT
call [wsprintf]
```

Kód v příkladě nejprve zkopíruje a poté zformátuje MAC adresu ze struktury adap do nulou ukončeného řetězce my_mac. Výstupní formát MAC adresy bude: "[xx-xx-x-xx-xx-xx]", kde xx reprezentují jednotlivé byty MAC adresy.

V druhém bloku dojde k připravení vysílací zprávy. První část tvoří zkrácený zápis MAC adresy a konec zprávy pak obsahuje jméno uživatele. Pokud je například MAC adresa

00:15:AF:12:AB:CB a ukazatel name ukazuje na řetězec “xnovak00“, tak řetězec na nějž ukazuje SEND_TEXT bude vypadat následovně:

```
[*AF-12-AB-CB]:                                    xnovak00
```

V levé části je MAC adresa a vpravo pak jméno uživatele. Funkci wsprintf popisuje tabulka 4.8. Významy jednotlivých formátovacích položek lze nalézt [29].

Tab. 4.8: Funkce wsprintf.

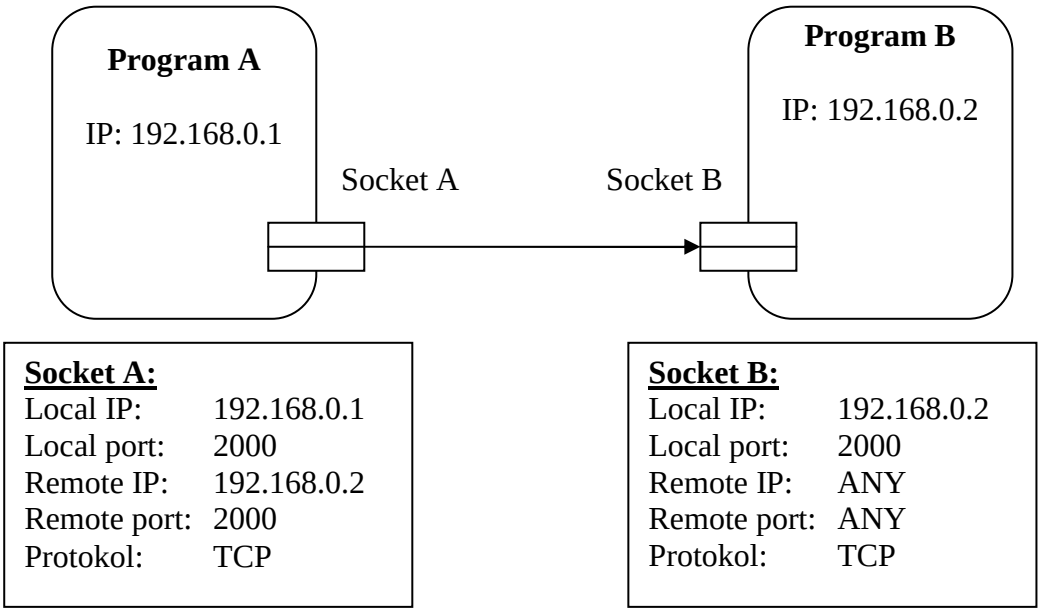
Syntaxe:	Popis poměných:
int wsprintf(LPTSTR lpOut, LPCTSTR lpFmt, ...);	 Ukazatel na buffer který obdrží zformátovaný výstup. Ukazatel na nulou ukončený řídící řetězec. Jeden nebo více nepovinných argumentů. Počet závisí na lpFmt parametru.
Návratová hodnota:	V případě úspěchu vrací funkce počet znaků uložených ve výstupním bufferu. V případě selhání funkce vrací menší počet znaku než je očekáváno (je možno volat GetLastError pro dodatečné informace).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

4.4 Internetové sockety

V následujících kapitolách bude často používán pojem internetové sockety, lze je chápat jako základní prvky při síťovém programování a je možné je definovat jako rozhraní sítí. Z pohledu programátora je socket struktura, jenž obsahuje jednoznačnou adresu síťového rozhraní (socketu) v rámci celé sítě. Pokud je to účelné, tak také obsahuje adresu socketu ze kterým program skrze svůj socket komunikuje. Typ adresy závisí na použité rodině protokolů. V textu bude použita rodina protokolů IPv4, adresu socketu určuje IP adresa a číslo portu (UDP nebo TCP).

Další dělení socketu je z hlediska spojovosti spojení a typu použitého protokolu. Je potřeba říci, že ačkoliv se text diplomové práce bude zabývat jen protokolem IPv4, je socket nezávislý na typu protokolu.

Dva programy které spolu komunikují pomocí socketů popisuje obrázek 4.4.



Obr. 4.4: Komunikace skrz sockety.

Z obrázku lze vidět, že program B je server, který očekává spojení na portu 2000. Položky remote IP a remote port obsahují hodnotu ANY, což znamená, že socket přímá jakékoliv spojení. Socket programu A má položky remote IP a remote port nastaveny na port a IP adresu socketu B. Oba sockety používají protokol TCP z rodiny protokolu IP.

5 Architektura peer-to-peer.

5.1 Inicializace socketu

Při implementaci architektury peer-to-peer je nejprve potřeba inicializovat socket. K tomu slouží funkce WSAStartup, tab. 5.1.

Tab. 5.1: Funkce WSAStartup.

Syntaxe:	Popis poměných:
int WSAStartup(WORD wVersionRequested, LPWSADATA lpWSAData);	 Nejvyšší verze socketu jenž může být použita. Ukazatel na WSADATA strukturu.
Návratová hodnota:	V případě úspěchu vrací funkce nulu, jinak chybový kód.
Funkce se nachází v Ws2_32.dll.	

Struktura použitá v druhém argumentu funkce je definována následovně:

Tab. 5.2: Struktura WSADATA.

Syntaxe:	Popis poměných:
STRUC WSAdata	
.wVersion RESW 1	Verze Windows socketu.
.wHighVersion RESW 1	Nejvyšší verze Windows socketu, jenž je podporována.
.szDescription RESB WSADESCRIPTION_LEN+1	ASCIIZ řetězec pro popis implementace socketu.
.szSystemStatus RESB WSASYS_STATUS_LEN+1	ASCIIZ řetězec pro popis významných stavů socketu nebo konfigurační informace.
.iMaxSockets RESW 1	Od verze winsock 2.0 nemá význam, zachováno jen kvůli zpětné kompatibilitě.
.iMaxUdpDg RESW 1	Od verze winsock 2.0 nemá význam, zachováno jen kvůli zpětné kompatibilitě.
.lpVendorInfo RESD 1	Od verze winsock 2.0 nemá význam, zachováno jen kvůli zpětné kompatibilitě.
ENDSTRUC	

Příklad, inicializace socketu:

Je importována funkce WSAStartup z knihovny Ws2_32.dll.

```
Extern WSAStartup
import WSAStartup Ws2_32.dll
```

Dále je potřeba v datovém segmentu alokovat místo na strukturu WSAdata.

```
WSAdata1:   resb  WSAdata_size
```

Konstanta WSAdata obsahuje počet bytů struktury. Tato konstanta je definována v souboru win32n.inc. V kódovém segmentu pak proběhne volání funkce:

```
push dword WSAdata1
xor eax,eax
mov ax, 0x0202
push dword eax
call [WSAStartup]
test eax,eax
jnz near .Finish2
```

První argument funkce měl hodnotu 0x0000 0202. Je tedy požadována verze 2.2. Více informací ohledně verzí socketu lze nalézt [12].

Funkce WSAStartup tedy naplní strukturu WSAdata1 která bude potřeba dále.

5.2 Vytvoření socketu

Socket se vytváří pomocí funkce socket. Tato funkce má tři parametry. První specifikuje rodinu protokolů. Pro rodinu protokolů IPv4 je definována konstanta AF_INET (0x2). Druhý parametr definuje typ vytvářeného socketu, například konstanta SOCK_STREAM využívá spojové spolehlivé spojení transportního protokolu TCP. Konstanta SOCK_DATAGRAM pak využívá nespojový nespolehlivý přenos transportním protokolem UDP. Posledním parametrem funkce socket je typ transportního protokolu, např. konstanta IPPROTO_TCP pro TCP, nebo IPPROTO_UDP pro UDP.

Všechny tři parametry jsou na sobě závislé, tedy pokud je požadováno spojení protokolem TCP, tak druhý a třetí bude mít hodnotu SOCK_STREAM resp. IPPROTO_TCP. První argumentem pak může být AF_INET nebo AF_INET6. Literatura [13] pak podrobně popisuje funkci socket.

Tab. 5.3: Funkce socket.

Syntaxe:	Popis poměných:
SOCKET WSAAPI socket(int af, int type, int protocol);	 Specifikace rodiny protokolů (pro TCP/IP bude použita konstanta AF_INET). Typ vytvářeného socketu. Specifikace transportní vrstvy.
Návratová hodnota:	V případě úspěchu vrací funkce identifikátor vytvořeného socketu.
	V případě neúspěchu vrací funkce hodnotu INVALID_SOCKET (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Příklad, vytvoření socketu:

```
push dword IPPROTO_UDP
push dword SOCK_DGRAM
push dword AF_INET
call [socket]
jmp eax,INVALID_SOCKET
je near .Finish2
mov [RecvSock],eax
```

Zdrojový kód v příkladu vytvoří socket pro nespolehlivý nespojový přenos pomocí protokolu UDP v IPv4. Pokud nastane chyba, tak je funkcí vrácena konstanta INVALID_SOCKET a program skočí na návěští .Finish2, v opačném případě funkce vrací identifikátor právě vytvořeného socketu, ten je pak zkopírován do proměnné `[RecvSock]`.

5.3 Přijímací socket.

Přijímacím socketem je myšlen socket, který přijímá UDP datagramy. Po inicializaci přijímacího socketu je socket potřeba "pojmenovat". Tedy socketu přiřadit instanci struktury sockaddr. Tato struktura obsahuje informace na jakém UDP portě bude přijímací socket naslouchat, z jaké IP adresy bude datagramy přijímat. Dále pak specifikuje rodinu protokolů.

Struktura sockaddr má více verzí, to kvůli existenci více protokolových rodin a taky kvůli jejich historickému vývoji. Například pro protokol IP existuji dvě základní verze, jsou to IPv4 a IPv6. Dále tato struktura existuje v obecné verzi jako již zmíněná sockaddr nebo jako sockaddr_in. Struktura sockaddr je společná pro všechny sockety a předává se jako parametr funkcím pro práci se sockety. Pro práci s internetovými protokoly se kvůli pohodlí používá struktura sockaddr_in. Velikost obou struktur musí být samozřejmě stejná. Obě struktury jsou velké 16 bytů. Strukturu sockaddr_in je popsána tabulkou 5.4.

Tab. 5.4: Struktura sockaddr_in.

Syntaxe:	Popis poměných:
STRUC sockaddr_in	
.sin_family RESW 1	Specifikace rodiny protokolů.
.sin_port RESW 1	Port socketu.
.sin_addr RESD 1	IP adresa socketu (IPv4).
.sin_zero RESB 8	Nevyužitá část (zachováno kvůli kompatibilitě se strukturou sockaddr).
ENDSTRUC	

Přiřazení struktury typu sockaddr je pak provedeno voláním funkce bind. Funkce je detailněji popsána tabulkou 5.5.

Tab. 5.5: Funkce bind.

Syntaxe:	Popis poměných:
int bind(	
SOCKET s,	Identifikátor socketu.
const struct sockaddr* name,	Ukazatel na strukturu sockaddr_in.
int namelen	Velikost struktury na kterou ukazuje druhý argument funkce.
);	
Návratová hodnota:	V případě úspěchu vrací funkce nulu.
	V případě neúspěchu vrací funkce hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Posledním krokem při vytvoření přijímacího socketu je volání funkce recvfrom. Tato funkce čte příchozí data a současně ukládá adresu vysílacího socketu, uloží adresu přijatého datagramu. Funkce pracuje v blokujícím režimu. Znamená to, že pokud ve frontě není žádný

datagram, tak funkce recvfrom zablokuje provádění programu (případně vlákna). Tuto funkci lze také použít pro spojově orientovaný přenos. Detailněji je popsána tabulkou 5.6.

Tab. 5.6: Funkce recvfrom.

Syntaxe:	Popis poměných:
int recvfrom(__in SOCKET s, __out char* buf, __in int len, __in int flags, __out struct sockaddr* from, __inout int* fromlen);	 Identifikátor socketu. Ukazatel na buffer pro příchozí data. Velikost bufferu. Množina příznaků pro modifikaci volané funkce. Viz lit. [18]. Nepovinný parametr, ukazatel na strukturu typu sockaddr. Funkce zde uloží parametry vysílacího socketu. Nepovinný parametr, ukazatel na velikost struktury na kterou ukazuje předchozí parametr.
Návratová hodnota:	V případě úspěchu vrací funkce počet přijatých bytů.
	V případě neúspěchu vrací funkce hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

V následujícím příkladu jsou použity funkce htonl a htons. Obě slouží ke změně endianity (způsob uložení čísel) z formátu Little-endian na Big-endian. Architektura x86 ukládá data do pamětí ve formátu Little-endian. Ale většina funkcí (například Bind), pracuje s čísly portů a IP adres ve formátu Big-endian. Proto je potřeba před uložení těchto čísel jejich konverze. Funkce htonl a htons jsou popsány tabulkami 5.7, 5.8.

Tab. 5.7: Funkce htonl.

Syntaxe:	Popis poměných:
u_long htonl(__in u_long hostlong);	16-bit číslo v endianilitě Little-endian.
Návratová hodnota:	16-bit číslo v endianilitě Big-endian.
Funkce se nachází v Ws2_32.dll.	

Tab. 5.8: Funkce htons.

Syntaxe:	Popis poměných:
u_short htons(__in u_short hostshort);	8-bit číslo v endianilitě Little-endian.
Návratová hodnota:	8-bit číslo v endianilitě Big-endian.
Funkce se nachází v Ws2_32.dll.	

Příklad, přijímací socket:

```
; Datový segment
[section .data class=DATA use32 align=16]

sin2:             resb  sockaddr_in_size
SenderAddr:       resb  sockaddr_in_size
SenderAddr_size   dd    sockaddr_in_size
RECV_TEXT:        resb  50
TEXT_size         dd    50
port              dd    5000
```

Nejprve je potřeba alokovat místo v paměti na dvě instance struktury sockaddr_in. Sin2 obsahuje adresu přijímacího socketu a SenderAddr adresu vysílacího socketu. Proměnná TEXT je buffer do kterého jsou nahrána data z přijatého paketu. Proměnná port pak obsahuje číslo portu na kterém bude přijímací socket naslouchat.

```
; Kódový segment
[section .code use32 class=CODE]

;vyplnění struktury sockaddr_in
mov word [sin2 + sockaddr_in.sin_family],AF_INET
push dword [port]
call [htons]
mov dword [sin2 + sockaddr_in.sin_port],EAX
push dword INADDR_ANY
call [htonl]
mov dword [sin2 + sockaddr_in.sin_addr],EAX
```

Struktura sin2 je vyplněna následovně, rodina protokolů je použita AF_INET, port je nastaven na hodnotu 5000 a IP adresa na hodnotu INADDR_ANY, tedy socket přijímá datagramy ze všech IP adres.

Samozřejmě je potřeba napřed socket [RecvSock] inicializovat a vytvořit. Před voláním funkce bind. Viz příklad v minulé kapitole.

```
; pojmenování socketu
push dword sockaddr_in_size
push dword sin2
push dword [RecvSock]
call [bind]
cmp eax,SOCKET_ERROR
je near .Finish2
```

Volání funkcí bind pak dojde k přiřazení struktury sin2 socketu [RecvSock]. Pokud funkce bind vrátí chybu, tak program skočí na návěští .Finish2 .

```
; příjem dat
push dword SenderAddr_size
push dword SenderAddr
push dword 0
push dword [TEXT_size]
push dword RECV_TEXT
push dword [RecvSock]
call [recvfrom]
cmp eax,SOCKET_ERROR
je near .Finish2
```

Voláním funkce recvfrom socket očekává příjem dat. Pokud nějaký datagram přijde, tak jej funkce uloží do přijímacího bufferu (ukazatel TEXT) a adresu odesílatele pak uloží do struktury SenderAddr. Pokud datagram nepřijde, tak se běh programu zastaví. Funkce recvfrom pracuje v blokujícím režimu.

5.4 Vysílací socket.

Při programování vysílacího socketu se poustupuje obdobně jako v případě přijímacího socketu. Není potřeba volat funkci bind a místo funkce recvfrom je volána funkce sendto k odeslaní datagramu. Funkce sendto je detailněji popsána tabulkou 5.9.

Tab. 5.9: Funkce sendto.

Syntaxe:	Popis poměných:
int sendto(	
__in SOCKET s,	Identifikátor socketu.
__in const char* buf,	Ukazatel na buffer pro odchozí data.
__in int len,	Velikost bufferu.
__in int flags,	Množina příznaků pro modifikaci volané funkce.
__in const struct sockaddr* to,	Nepovinný parametr, ukazatel na strukturu typu sockaddr která obsahuje adresu cílového socketu.
__in int tolen	Velikost struktury na kterou ukazuje předchozí parametr.
);	
Návratová hodnota:	V případě úspěchu vrací funkce počet odeslaných bytů., ten ale může být menší než udává parametr len.
	V případě neúspěchu vrací funkce hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Pro snadnější manipulaci s IP adresami slouží funkce inet_addr. Převádí IPv4 adresu z řetězce znaků na 32-bit číslo.

Tab. 5.10: Funkce inet_addr.

Syntaxe:	Popis poměných:
unsigned long inet_addr(	
__in const char* cp	Ukazatel na nulou ukončený řetězec znaků který obsahuje IP adresu ve standardní (tečkové) notaci.
);	
Návratová hodnota:	V případě úspěchu vrací funkce 32-bit binární hodnotu IP adresy.
	V případě neúspěchu vrací funkce hodnotu INADDR_NONE, například v případě, když parametr cp neobsahuje legitimní IP adresu, třeba "a.b.c.d"
Funkce se nachází v Ws2_32.dll.	

Příklad, vysílací socket:

```
; Datový segment
[section .data class=DATA use32 align=16]

SEND_TEXT:  resb  50
TEXT_size   dd    50
SendSock    dd    0
sin1:       resb  sockaddr_in_size
badr:       DB    "10.255.255.255",0
port        dd    5000
```

Nejprve se v datovém segmentu deklarují potřebně proměnné. Proměnná badr obsahuje broadcast adresu sítě, port pak číslo portu. SEND_TEXT je ukazatel na nulou ukončený řetězec, který obsahuje data k odeslání. Struktura sin1 obsahuje adresu příjemce a SendSock je identifikátor vysílacího socketu. Vysílací socket je samozřejmě nejprve potřeba inicializovat a vytvořit.

```
; Kódový segment
[section .code use32 class=CODE]

;vyplnění struktury sockaddr_in
mov word [sin1 + sockaddr_in.sin_family],AF_INET
push dword [port]
call [htons]
mov dword [sin1 + sockaddr_in.sin_port],EAX
push dword badr
call [inet_addr]
mov dword [sin1 + sockaddr_in.sin_addr],EAX
```

Struktura sin1 je vyplněna takto: rodina protokolů je AF_INET, číslo portu 5000 a IP adresa cíle je broadcast adresa.

```
; odeslání dat
push dword sockaddr_in_size
push dword sin1
push dword 0
push dword [TEXT_size]
push dword SEND_TEXT
push dword [SendSock]
call [sendto]
cmp eax,SOCKET_ERROR
je near .Finish2
```

Voláním funkce sendto se odešle datagram. Data tohoto datagramu reprezentuje ukazatel TEXT. Datagram se odešle skrz socket SendSock na adresu kterou obsahuje struktura sin1.

Dále je testována návratová hodnota funkce, pokud dojde k chybě, tak je instrukcí JE skočeno na návěští .Finish2. Není ale kontrolován počet odeslaných bytů, může tedy nastat situace kdy je odeslán prázdný datagram.

5.5 Uzavření socketu.

Pro zavření existujícího socketu slouží funkce closesocket. Jejím jediným parametrem je identifikátor socketu, který má být uzavřen.

Tab. 5.11: Funkce closesocket

Syntaxe:	Popis proměnných:
int closesocket(SOCKET s);	Identifikátor socketu.
Návratová hodnota:	V případě úspěchu vrací funkce nulu.
	V případě neúspěchu vrací funkce hodnotu INVALID_SOCKET (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Příklad, zavření socketu:

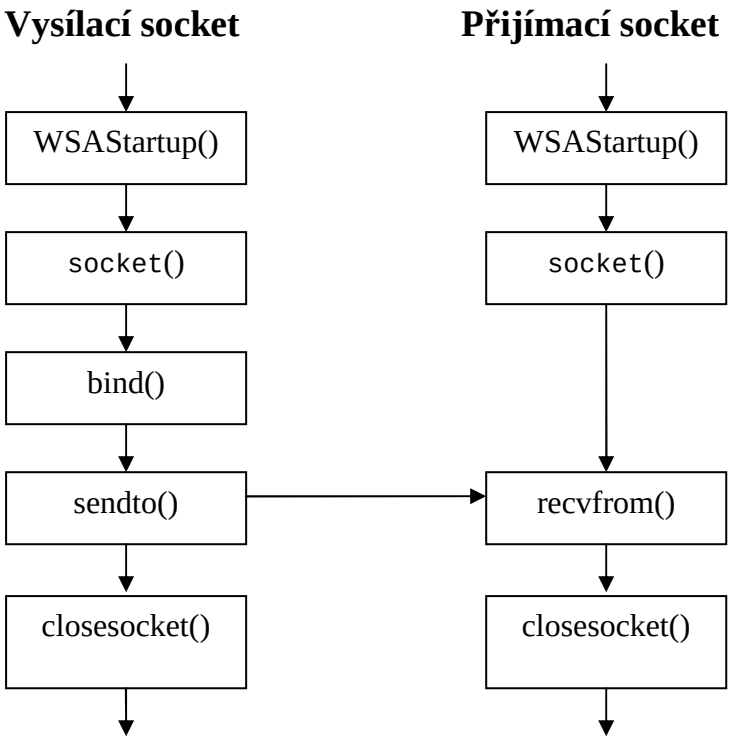
```
push dword [soc1]
call [closesocket]
```

Kód v příkladě uzavře socket [soc1].

5.6 Závěr kapitoly.

Cílem kapitoly bylo popsat funkce a postupy potřebné k naprogramování síťové aplikace pracující v architektuře peer-to-peer s použitím nespolehlivé a nespojové služby pomocí protokolů IP a UDP.

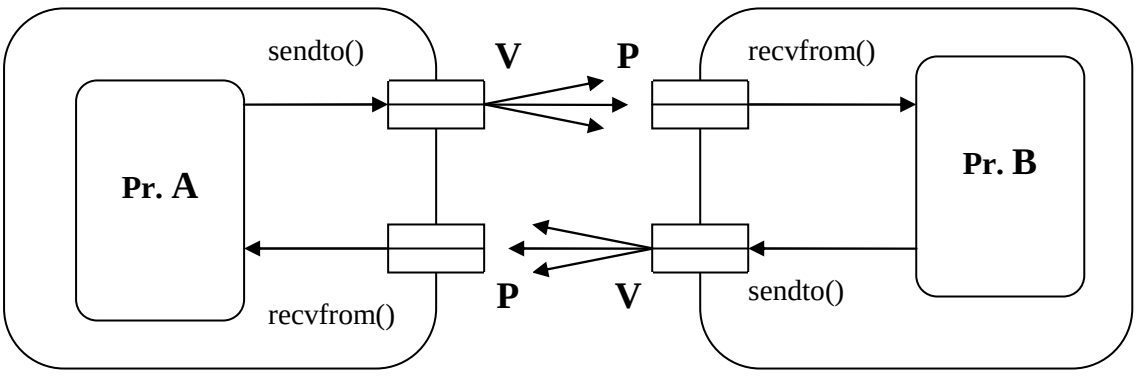
Výsledné chování komunikace mezi vysílacím a přijímacím socketem zobrazuje následující blokové schéma:



Obr. 5.1: Blokové schéma komunikace mezi sockety.

Nejprve dojde k inicializaci a pak vytvoření socketu, vysílací socket volá funkci bind a funkcí sento odešle datagram. Přijímací socket volá jen funkci recfrom pro příjem datagramu. Na konci se oba sockety uzavřou voláním funkce closesocket.

Na obrázku 5.2 je pak zobrazena komunikace mezi programy A a B. Každý program má dva sockety. Jeden pro příjem (označený jako **P**) a druhý pro vysílání (označený jako **V**). Trojice šipek pak symbolizuje, že je datagram vyslán na všesměrovou adresu.



Obr. 5.2: Schéma práce socketu v reálné aplikaci.

6 Architektura client-server.

6.1 Inicializace a vytvoření socketu.

Inicializace socketu je úplně stejná jako v architektuře peer-to-peer a vytvoření socketu probíhá podobně, jen jsou funkci předány jiné parametry. Protokoly TCP a IP je vytvořen socket pro spojový spolehlivý přenos.

Příklad, vytvoření socketu:

```
push dword IPPROTO_TCP
push dword SOCK_STREAM
push dword AF_INET
call [socket]
cmp eax,INVALID_SOCKET
je near .Finish
mov [soc1],eax
```

Zdrojový kód v příkladu vytvoří socket pro spolehlivý spojový přenos pomocí protokolu TCP v IPv4. Pokud nastane chyba, tak je funkcí vrácena konstanta INVALID_SOCKET a program skočí na návěští .Finish, v opačném případě funkce vrací identifikátor právě vytvořeného socketu, ten je pak zkopírován do proměnné [soc1].

6.2 Navázání spojení, server.

Při navazování spojení je nutno rozlišovat, zda program volání inicializuje, nebo pouze volání očekává. V dalším částí textu bude pojmem server myšlen program, který volání očekává. V případě obrázku 4.1 by se jednalo o Program B. Klientem pak program, jenž volání inicializuje.

První krokem při navázání spojení ze strany serveru je zavolání funkce bind, jak již bylo řečeno, funkcí bind je socketu přiřazena instance struktury sockaddr. Tato instance struktury pak udává z jaké adresy bude server přijímat požadavky na navázání spojení.

Po funkci bind v logickém celku navazování spojení následuje funkce listen, tab. 6.1. Funkce listen přepíná socket do naslouchajícího stavu, systém vytvoří frontu požadavků na připojení. Její délka je určena druhým parametrem funkce. Pokud je fronta plná, další připojení bude odmítnuto.

Tab. 6.1: Funkce listen.

Syntaxe:	Popis poměných:
int listen(SOCKET s, int backlog);	 Identifikátor socketu. Maximální délka fonty nevyřízených spojení.
Návratová hodnota:	V případě úspěchu vrací funkce nulu. Jinak vrátí hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Funkce accept pak vybere požadavek na spojení z fronty požadavku a potvrdí ho. Funkce pracuje v tzv. blokujícím režimu. Pro každé přijaté spojení je vytvořen nový socket,

identifikátor socketu je také návratovou hodnotou funkce. Původní socket slouží pouze k navazování spojení. Druhý parametr funkce accept je ukazatel na instanci struktury sockaddr_in. Tato instance bude vyplněna při volání funkce a bude obsahovat informace o socketu jenž inicializoval spojení. Instance struktury pak bude obsahovat mimo jiné IP adresu a port socketu klienta.

Tab. 6.2: Funkce accept.

Syntaxe:	Popis poměných:
SOCKET accept(SOCKET s, Struct sockaddr* addr, int* addrlen);	 Identifikátor socketu. Ukazatel na strukturu sockaddr_in. Ukazatel na velikost struktury na kterou ukazuje druhý argument funkce.
Návratová hodnota:	V případě úspěchu vrací funkce identifikátor nového socketu socketu.
	V případě neúspěchu vrací funkce hodnotu INVALID_SOCKET (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Příklad, navázání spojení (server):

Nejprve je potřeba alokovat místo v paměti na instance struktury sin1 a sin2. Dále pak definovat proměnné len, soc1 a soc2. Soc1 a soc2 slouží k uložení identifikátorů socketů. Len bude obsahovat délku struktury sockaddr_in.

```
; Datový segment
[section .data class=DATA use32 align=16]

sin1:       resb        sockaddr_in_size
sin2:       resb        sockaddr_in_size
len         dd          sockaddr_in_size
soc1        dd          0
soc2        dd          0
```

Zdrojový kód pro navázání spojení ze strany serveru vypadá následovně:

```
; Kódový segment
[section .code use32 class=CODE]

;vyplnění struktury sockaddr_in
mov word [sin1 + sockaddr_in.sin_family],AF_INET
push dword 2000
call [htons]
mov word [sin1 + sockaddr_in.sin_port],EAX
push dword INADDR_ANY
call [htonl]
mov dword [sin1 + sockaddr_in.sin_addr],EAX

; pojmenování socketu
push dword sockaddr_in_size
push dword sin1
push dword [soc1]
call [bind]
cmp eax,SOCKET_ERROR
```

```
      je near .Finish

      ; přepnutí socketu do naslouchajícího stavu
      push dword 5
      push dword [soc1]
      call [listen]
      cmp eax,SOCKET_ERROR
      je near .Finish

      ; čekání na požadavek o navázání spojení
      push dword len
      push dword sin2
      push dword [soc1]
      call [accept]
      cmp eax,INVALID_SOCKET
      je near .Finish
      mov [soc2],eax
```

Na začátku prvního bloku se vyplní struktura sin1. Následně je volána funkce bind pro pojmenování socketu, dále je pak testována návratová hodnota pro případný výskyt chyby.

V dalším bloku dochází k volání funkce listen, přepnutí socketu do naslouchajícího režimu. Fronta požadavku na spojení může mít maximálně pět požadavků, ostatní jsou odmítnuty. Opět je testována návratová hodnota funkce kvůli případnému výskytu chyby.

V posledním bloku dochází k volání funkce accept. Pokud funkce nevrátí návratový chybový kód, tak se do proměnné [soc2] uloží identifikátor nově vytvořeného socketu.

6.3 Navázání spojení, klient.

Navázaní spojení ze strany klienta je o poznání jednodušší. Slouží k tomu jediná funkce s názvem connect. Funkční prototyp funkce connect je velmi podobný funkci accept. Connect navazuje spojení na socketu s, adresa cílového socketu je pak popsána strukturou typu sockaddr.

Tab. 6.3: Funkce connect.

Syntaxe:	Popis poměných:
int connect(SOCKET s, const struct sockaddr* name, int namelen);	 Identifikátor socketu. Ukazatel na strukturu sockaddr_in. Velikost struktury na kterou ukazuje druhý argument funkce.
Návratová hodnota:	V případě úspěchu vrací funkce nulu.
	V případě neúspěchu vrací funkce hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Příklad, navázání spojení (klient):

V datovém segmentu je alokován prostor pro instanci struktury sockaddr_in.

```
sin1: resb  sockaddr_in_size
```

V kódovém segmentu pak dojde k volání funkce connect.

```
mov word [sin1 + sockaddr_in.sin_family], AF_INET
mov word [sin1 + sockaddr_in.sin_port],2000
mov dword [sin1 + sockaddr_in.sin_addr], EDX

push dword sockaddr_in_size
push dword sin1
push dword [soc1]
call [connect]
cmp eax, SOCKET_ERROR
je near .Finish
```

Instance struktury sockaddr_in je inicializována na adresu serveru, port serveru a použitou rodinu protokolu. IP adresa severu je uložena v registru EDX. Dolní byte registru EAX obsahuje nejlevější byte IP adresy serveru. Horní byte pak obsahuje nejpravější byte IP adresy serveru.

Instrukcí push se uloží data na zásobník a instrukcí call pak dojde k volání funkce connect. Je testována návratová hodnota funkce, pokud dojde k chybě, tak je instrukcí JE skočeno na návěští .Finish.

6.4 Přenos dat.

Data jsou odeslána pomocí funkce send a přijímána pomocí recv. Funkce popisují tabulky 6.4 a 6.5. Funkce recv pracuje, stejně jako accept v tzv. blokujícím režimu.

Tab. 6.4: Funkce send.

Syntaxe:	Popis proměnných:
int send(	
SOCKET s,	Identifikátor socketu.
const char* buf,	Ukazatel na buffer pro odchozí data.
int len,	Velikost bufferu.
int flags	Příznak specifikující způsob volání funkce.
);	
Návratová hodnota:	V případě úspěchu vrací funkce počet odeslaných bytů.
	V případě neúspěchu vrací funkce hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Tab. 6.5: Funkce recv.

Syntaxe:	Popis proměnných:
int recv(	
SOCKET s,	Identifikátor socketu.
char* buf,	Ukazatel na buffer pro příchozí data.
int len,	Velikost bufferu.
int flags	Příznak specifikující způsob volání funkce.
);	
Návratová hodnota:	V případě úspěchu vrací funkce počet přijatých bytů.
	V případě neúspěchu vrací funkce hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

Příklad, přenos dat (odeslání dat):

```
push dword 0
push dword [txt_lenght]
push dword txt
push dword [soc1]
call [send]
cmp eax,SOCKET_ERROR
je near .Finish
```

Ukazatel txt ukazuje na ASCIIZ řetězec o délce [txt_lenght], který obsahuje data k odeslání. Po volání funkce je kontrolována návratová hodnota kvůli případné chybě.

Příklad, přenos dat,příjem dat:

```
push dword 0
push dword 100
push dword buf
push dword [soc1]
call [recv]
```

Pomocí funkce recv je očekáván příjem dat na socketu [soc1], data se uloží do bufferu buf, který má velikost 100 bytů.

6.5 Deaktivace a uzavření socketu.

Funkce shutdown slouží k deaktivaci příjmu nebo odesílaní dat. Oproti funkci closesocket však neuzavírá socket. To může být užitečné pokud vznikne požadavek na nové spojení v průběhu běhu programu.

Druhý parametr této funkce specifikuje operace, které nebudou dále podporovány, 0 pro deaktivaci přijmu dat, 1 pro deaktivaci odesílání dat a 2 pro deaktivaci příjmu i odesílaní dat.

Pro uzavření socketu je stále potřeba volat funkci closesocket.

Tab. 6.6: Funkce shutdown

Syntaxe:	Popis proměnných:
int shutdown(__in SOCKET s, __in int how);	 Identifikátor socketu. Specifikuje typ operace který nebude nadále podporován.
Návratová hodnota:	V případě úspěchu vrací funkce nulu.
	V případě neúspěchu vrací funkce hodnotu SOCKET_ERROR (je možno volat WSAGetLastError pro získání chybového kódu).
Funkce se nachází v Ws2_32.dll.	

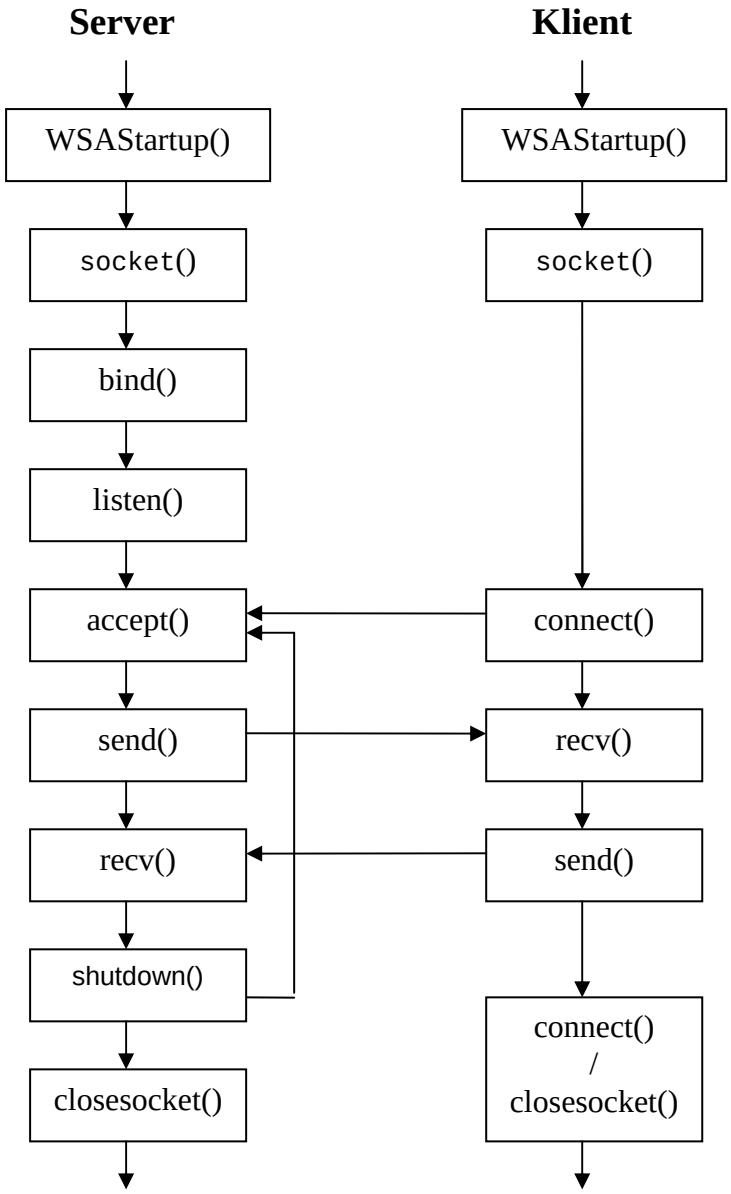
Příklad, deaktivace socketu:

```
push dword 2
push dword soc2
call [shutdown]
```

Kód v příkladu uzavře spojení na [soc2].

6.6 Závěr kapitoly.

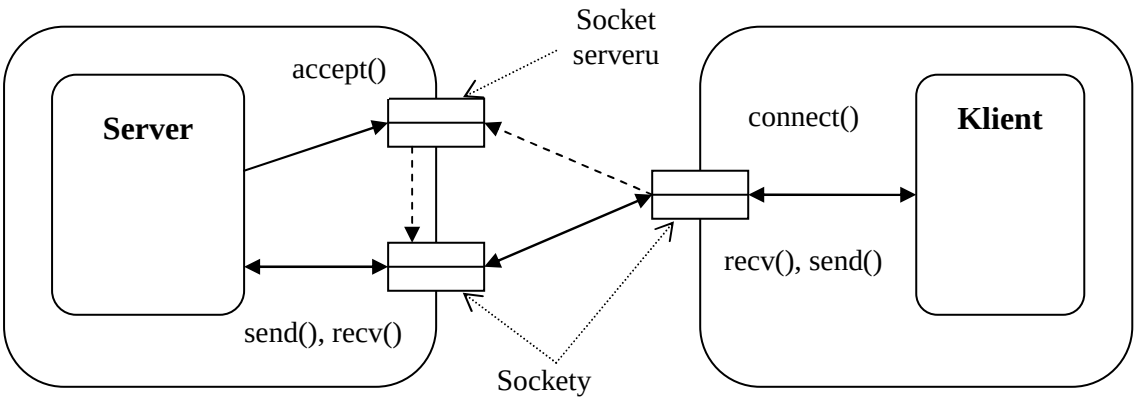
Výsledné chování komunikace mezi programy A a B podle obrázku 4.1 popisuje následující blokové schéma:



Obr. 6.1: Schéma komunikace mezi serverem a klientem.

Na začátku oba programy inicializují a vytvoří sockety. Server pak navíc provede volání funkce bind. Následně je socket serveru je přepnut do naslouchajícího stavu funkcí listen, server pak očekává požadavek na spojení (accept). Klient pomocí funkce connect zašle tento požadavek. Pak následuje výměna dat pomocí funkcí send a recv. Nakonec server uzavře socket jenž byl vytvořen pouze pro komunikaci s klientem. Klient svůj socket uzavírat nemusí, může jej dalším voláním funkce connect použít pro komunikaci s jiným serverem.

Z pohledu vytváření socketu pak vše lépe vysvětluje obrázek 6.2.



Obr. 6.2: Schéma vytváření socketu při komunikaci.

Na schématu jsou vidět celkem tři sockety. První, označený jako server socketu, slouží ke zpracování požadavku na spojení. Požadavek na spojení pošle klient funkcí connect skrz svůj socket. Funkce accept pak vytvoří poslední socket zobrazený na schématu. Socket serveru slouží pouze k přijímaní požadavku na spojení a nově vytvořený socket pak na přenos dat (funkce send a recv).

7 Multitasking

7.1 Terminologie

Multitasking je schopnost operačního systému řešit více úloh současně. Multitasking se dělí na kooperativní a preemptivní. Pří kooperativní multitaskingu se jednotlivé běžící úlohy samy střídají (bez účasti OS) ve využití procesoru. Každá úloha tedy sama předává řízení zpět operačnímu systému. V případě havárie jedné úlohy pak dochází k havárií celého operačního systému. Tento druh multitaskingu byl použit v OS Windows 3.x. Novější typ multitaskingu je preemptivní multitasking. V tomto případě o přidělování a odebírání procesoru rozhoduje operační systém. Preemptivní multitasking je použit v systémech MS Windows od verze 95.

Multitaskingem je tedy řešena otázka více úloh na jeden operační systém. Místo abstraktní pojmu úloha se používá konkrétnější termín proces. Proces je objekt vlákna vytvořený operačním systémem při načítání spustitelného souboru. Například každý spustitelný soubor je v OS Windows charakterizován procesem [4].

Proces má přístup k následujícím prostředkům [4]:

-K vlastní virtuální paměti, tuto paměť vytvoří OS.
-Handly souborů které proces otevřel.
-Seznamy dynamických knihoven.
-Podprocesy.

Podprocesy jsou další procesy vytvořené hlavním procesem. Důležité je však říci, že jednotlivé podprocesy jsou vytvořeny v adresovaném prostoru hlavního procesu. Tedy mezi procesy je možno sdílet data. Je ovšem nutno nějakým způsobem ošetřit možnost současného přístupu dvou procesů k těmto sdíleným datům (například pomocí semaforů, událostí, kritických sekcí nebo mutexů). Podprocesy jsou označovány jako vlákna.

7.2 Vlákna

K vytvoření vlákna slouží funkce CreateThread, je obsažena v knihovně kernel32.dll. Tuto funkci popisuje tabulka 7.1. CreateThread vytvoří objekt vlákna který spustí funkci na kterou ukazuje 4. parametr (lpStartAddress). Tato funkce se nazývá funkce vlákna. Funkcí vlákna lze předat jednu proměnou, parametr lpParameter funkce CreateThread. O tom kdy se funkce vlákna spustí rozhoduje parametr dwCreationFlags, hodnota nula pak znamená, že se funkce vlákna spustí okamžitě (ihned po volání funkce CreateThread).

Tab. 7.1: Funkce CreateThread.

Syntaxe:	Popis poměných:
HANDLE WINAPI CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes, SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCreationFlags, LPDWORD lpThreadId);	Ukazatel na _SECURITY_ATTRIBUTES strukturu. Tato struktura obsahuje přístupové atributy, viz lit. [17]. Velikost zásobníků v bytech. Pokud je hodnota nula, tak je použita defaultní velikost zásobníku (tedy velikost rodičovského vlákna). Ukazatel na funkci vlákna. Při spuštění vlákna je volána tato funkce. Ukazatel na proměnou jenž bude předána vláknu. Řídící příznaky vytvoření vlákna. Hodnota nula znamená, že vlákno se vytvoří ihned po zavolání funkce CreateThread. Hodnota konstanty CREATE_SUSPENDED pak znamená, že vlákno bude spuštěno až voláním funkce ResumeThread. Ukazatel na proměnou do které bude nahrán identifikátor vlákna. Pokud bude mít hodnotu NULL, tak nebude nahrán žádný identifikátor.
Návratová hodnota:	V případě úspěchu vrací funkce handle nového vlákna. V případě selhání funkce vrací NULL (je možno volat GetLastError pro dodatečné informace).
Funkce se nachází v kernel32.dll.	

Existují tři základní možnosti jak vlákno ukončit. První a ta nejjednodušší možnost je, že funkce vlákna přirozeně ukončí svůj běh. Tedy v těle této funkce dojde k volání instrukce RET. Druhou možností je volání funkce ExitThread. Tato funkce je volána v těle funkce vlákna. Třetí možností je pak volání funkce TerminateThread. Touto funkci může mateřské vlákno ukončit běh dceřiného vlákna. Tedy hlavní proces takto může ukončit běh některého z podprocesů. Obě funkce na ukončení vlákna popisují tabulky 7.2 a 7.3.

Tab. 7.2: Funkce ExitThread.

Syntaxe:	Popis poměných:
VOID WINAPI ExitThread(__in DWORD dwExitCode);	Specifikátor výstupního kódu.
Návratová hodnota:	Funkce nemá návratovou hodnotu.
Funkce se nachází v kernel32.dll.	

Tab. 7.3: Funkce TerminateThread.

Syntaxe:	Popis poměných:
BOOL WINAPI TerminateThread(__inout HANDLE hThread, __in DWORD dwExitCode);	 Handle vlákna které bude ukončeno. Specifikátor výstupního kódu.
Návratová hodnota:	V případě úspěchu vrací funkce nenulovou hodnotu. V případě selhání funkce je návratová hodnota nula (je možno volat GetLastError pro dodatečné informace).
Funkce se nachází v kernel32.dll.	

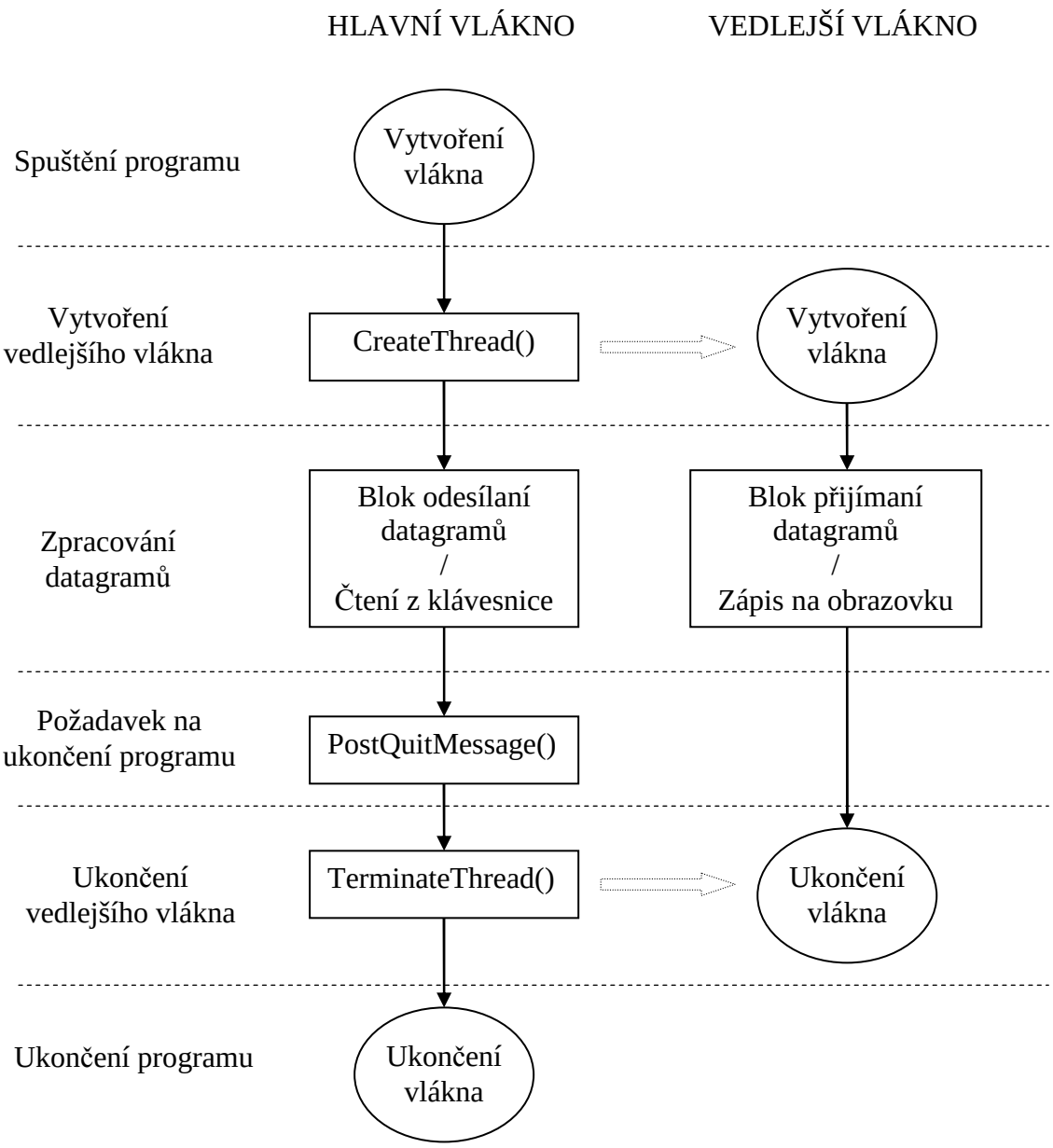
V tělech nekonečných smyček je vhodné volat funkci sleep. Tato funkce uvolňuje systémový čas. Ten pak může operační systém přiřadit jinému vláknu nebo procesu.

Tab. 7.4: Funkce Sleep.

Syntaxe:	Popis poměných:
VOID WINAPI Sleep(__in DWORD dwMilliseconds);	 Počet milisekund po které bude aktuální vlákno pozastaveno.
Návratová hodnota:	Funkce nemá návratovou hodnotu.
Funkce se nachází v kernel32.dll.	

7.3 Programovaní vláken

Cílem této podkapitoly je na praktickém příkladě vysvětlit programování vláken v jazyce symbolických instrukcí s použitím funkcí z Win32 API. Cílem je vytvořit program který bude ve dvou vláknech přijímat a odesílat UDP datagramy. Blokové schéma takového programu je zobrazeno na obrázku 7.1.



Obr. 7.1: Blokové schéma programu.

Spuštěním programu dojde k vytvoření hlavního vlákna programu. Následně je funkcí CreateThread vytvořeno “vedlejší“ vlákno, tj. vlákno pro příjem a zobrazení dat přijatého datagramu. Volat funkci CreateThread je vhodné například v proceduře okna, v konstruktoru. Následují dva bloky pro zpracování datagramu. Hlavní vlákno má za úkol čtení znaků z klávesnice (konkrétně pak z editačního pole), vyplnění a odeslání datagramu. Vedlejší vlákno pak má za úkol příjem datagramu a zobrazení datové části datagramu na obrazovku (konkrétně do posuvného seznamu). Pokud uživatel program ukončí, tak destruktor volá funkci PostQuitMessage, tedy běh programu vyskočí ze smyčky zpracování zpráv a jsou volány funkce pro korektní ukončení programu. Například funkce pro uzavření socketu. Jednou z těchto funkcí je pak TerminateThread. Vedlejší vlákno je touto funkci ukončeno. Funkcí ExitProcess je nakonec ukončeno i hlavní vlákno. V popsané terminologii je “hlavní vlákno“ procesem.

Příklad, spuštění vlákna:

```
push dword [iThread]
push dword 0
push dword [hListbox]
push dword RCthread
push dword 0
push dword 0
call [CreateThread]
mov [hThread],eax
```

Kód v příkladu vytvoří a okamžitě spustí vlákno. Do proměnné [iThread] je nahrán identifikátor vlákna a do proměnné [hThread] pak handle vlákna. Spuštění vlákna znamená volání funkce RCthread. Táto funkce má jediný parametr a to je [hListbox], tedy handle posuvného seznamu ve kterém bude přijatý datagram zobrazen.

Příklad, ukončení vlákna:

```
push dword 0
push dword [hThread]
call [TerminateThread]
```

Proměnná [hThread] obsahuje handle vlákna jenž bude ukončeno.

Příklad, funkce vlákna:

```
[section .code use32 class=DATA]

      RECV_TEXT:        resb  50
      TEXT_size         dd    50
```

RECV_TEXT je pole o velikosti 50 bytů alokovaných pro příjem a zobrazení datagramu. TEXT_size je pak proměnná která obsahuje hodnotu 50, tedy velikost předchozího pole.

```
[section .data class= CODE use32 align=16]

      RCthread:
            RCthread_PARAMBYTES EQU 4
            enter 0,0
                        ; vytvoření socketu
                        ; vyplnění struktury sockaddr_in
                        ; pojmenovaní socketu

            .Recv:
                        ; příjem dat

                  ; zobrazení dat
                  push dword RECV_TEXT
                  push dword [TEXT_size]
                  push dword LB_ADDSTRING
                  push dword [ebp+8]
                  call [SendMessage]
                  push dword 200
                  call [Sleep]
                  jmp near .Recv

            .Finish:
                  xor eax,eax
```

```
                leave
      ret RCthread_PARAMBYTES
```

V příkladu je vytvořena funkce, která má jeden parametr a žádné lokální proměnné. Parametrem funkce je pak handle na spojový seznam, kde se mají přijaté data zobrazit. Tato funkce bývá označována jako funkce vlákna. Běh programu je dán vykonáváním funkce main, obdobně je běh vlákna dán vykonáváním funkce vlákna. V tomto případě je funkcí vlákna funkce RCthread.

V těle funkce pak nejprve dojde vytvoření socketu, vyplnění struktury sockaddr_in a pojmenovaní socketu. V příkladu je to naznačeno pouze komentáři. Detailně je postup prostudován v kapitole 5.

Blok začínající návěštím .Recv a končící instrukcí jmp near .Recv slouží k příjmu dat (opět jen vyznačeno komentářem) a zobrazení dat. Přijímaná data jsou uložena do pole RECV_TEXT. Pomocí funkce SendMessage jsou pak zobrazena v posuvném seznamu, tento seznam je jednoznačně identifikován handlem, tedy parametrem funkce [ebp+8]. Po té následuje pozastavení vlákna na 200ms.

7.4 Mutex

Mutex je jedna z metod synchronizace pro práci s vlákny a procesy. Používá se ale výhradně pro synchronizaci procesů. Každý mutex má svého vlastníka, vlastníkem je proces. Mutex může mít jen jednoho vlastníka. Této vlastnosti se pak využívá mimo jiné pro zamezení vícenásobného spuštění programu.

Tab. 7.5: Funkce CreateMutex.

Syntaxe:	Popis poměných:
HANDLE WINAPI CreateMutex(LPSECURITY_ATTRIBUTES lpMutexAttributes, BOOL bInitialOwner, LPCTSTR lpName);	 Ukazatel na strukturu definující přístupové práva. Specifikace vlastníka mutexu. Při nenulové hodnotě se stává volající vlastníkem mutexu. Ukazatel na řetězec specifikující jméno mutexu.
Návratová hodnota:	V případě úspěchu vrací funkce handle nově vytvořeného mutexu. V případě selhání funkce vrací funkce hodnotu NULL (je možno volat GetLastError pro dodatečné informace).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v kernel32.dll.	

Příklad, zamezení vícenásobného spuštění programu:

```
[section .data class=DATA use32 align=16]

      mutex_name:       DB "xchat_window_mutex",0

[section .code use32 class=CODE]

      push dword mutex_name
      push dword 1
```

```
      push dword NULL
      call [CreateMutex]
      call [GetLastError]
      cmp eax,ERROR_ALREADY_EXISTS
      je .Finish
```

V příkladu je vytvořen mutex s názvem "xchat_window_mutex". Volající proces se stává vlastníkem mutexu. Následně dochází k volání funkce GetLastError. Touto funkcí je získán chybový kód posledně volané funkce. Pokud je tento chybový kód roven hodnotě konstanty ERROR_ALREADY_EXISTS. Tak došlo k vícenásobnému spuštění aplikace a je proveden skok na návěští .Finish.

Tab. 7.6: Funkce GetLastError.

Syntaxe:	
DWORD WINAPI GetLastError(void);	
Návratová hodnota:	Funkce nemá návratovou hodnotu.
Funkce se nachází v kernel32.dll.	

8 Dialogové okna

8.1 Zdroje

Při tvorbě dialogových oken existuje několik typů předdefinovaných dialogů. Je tu jistá podobnost s předdefinovanými prvky, jakým je například tlačítko. V takovém případě nevytváříme novou třídu okna pro tlačítko, ale použijeme třídu předdefinovanou. Při tvorbě dialogových oken je situace podobna, pokud programátor bude chtít použít některé z předefinovaných dialogových oken tak použije některou z funkci pro jejich tvorbu. Například MessageBox, tabulka 8.1. Vzhled a chování okna pak určuje 4. argument (uType). Možnosti pro nastavení jsou však velmi omezené, v zásadě se omezují jen počty tlačítek v dialogu. Nejde kupříkladu vložit na dialogové okno edit box nebo navrhnout vlastní uspořádání a velikosti jednotlivých ovládacích prvků. Následující dva příklady pak demonstrují použití této funkce.

Tab. 8.1: Funkce MessageBox

Syntaxe:	Popis poměných:
int MessageBox(	
HWND hWnd,	Handle vlastníka okna. Pokud je NULL, dialogové okno nemá vlastníka.
LPCTSTR lpText,	Ukazatel na nulou ukončený řetězec ze zprávou která bude zobrazená v dialogovém okně.
LPCTSTR lpCaption,	Ukazatel na nulou ukončený řetězec s titulkem dialogového okna.
UINT uType	Typ dialogového okna. Je definován množinou příznaků která určuje chovaní a vzhled dialogového okna. Podrobnější informace lze nalézt v [19].
);	
Návratová hodnota:	V případě úspěchu vrací funkce kód stisknuté klávesy. Kódy kláves lze nalézt např. v [19]. V případě selhání funkce vrací funkce hodnotu 0 (je možno volat GetLastError pro dodatečné informace).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v user32.dll.	

Obr. 8.1: Předdefinované dialogové okno 1

Příklad, vytvoření předdefinovaného modálního dialogového okna 1:

```
[section .data class=DATA use32 align=16]

    cap_thread:  DB "Error",0
    mes_mutex:   DB "Program lze spustit jen v jediné instanci.",0

[section .code use32 class=CODE]

    push dword MB_OK
    push dword cap_mutex
```

```
        push dword mes_mutex
        push dword 0
        call [MessageBox]
```

Příkladu je vytvořeno dialogové okno podle obrázku 8.1. Styl okna je MB_OK, je vytvořeno jediné tlačítko “OK“. Handle vlastníka okna je nastaven na nulu, což znamená že toto dialogové okno vlastníka nemá.

Obr. 8.2: Předdefinované dialogové okno 2

Příklad, vytvoření předdefinovaného modálního dialogového okna 1:

```
[section .data class=DATA use32 align=16]

    cap_thread: DB "Error,thread.",0
    mes_mutex: DB "You can run this program only in one instance.",0

[section .code use32 class=CODE]

    push dword MB_OK | MB_ICONINFORMATION
    push dword cap_mutex
    push dword mes_mutex
    push dword NULL
    call [MessageBox]
```

Příkladu je vytvořeno dialogové okno podle obrázku 8.2. Oproti minulému příkladu byl navíc použit styl MB_ICONINFORMATION který vytvoří ikonu ze symbolem informace.

Lze vidět, že možnosti funkce MessageBox jsou omezené, pro složitější dialogové okna je nutno použit zdroje (resources). Zdroje jsou grafické data nejčastěji uložena ve spustitelném *.exe souboru. Pomocí zdrojů lze vytvořit vlastní kurzory, ikony, bitmapy, znakové řetězce, nabídky a dokonce i klávesové zkratky. V následujícím textu budou zdroje použity pro tvorbu dialogových oken.

Na vytvoření zdrojů lze použit některý z nástrojů pro editaci zdrojů, například Visual Studio .NET. Zdroje však lze vytvářet v libovolném textovém editoru. Textový soubor pak mívá příponu RC. Po kompilaci pak RES. Kompilace je popsána v kapitole 1.4.

Jazyk pro tvorbu zdrojů je velmi intuitivní a snadno pochopitelný. Soubor se zdroji začíná definicí konstant. Následuje identifikátor zdroje a vlastnosti zdroje. Jednotlivé zdroje jsou tímto identifikátorem jednoznačné identifikovány. Identifikátor zdroje a jeho handle jsou dva rozdílné identifikátory. Pokud nějaká funkce potřebuje znát handle prvku dialogového okna, tak je potřeba jej nejprve zjistit. K tomu slouží funkce GetDlgItem.

Obr. 8.3: Vytvořené dialogové okno

Příklad, soubor se zdroji:

```
      //-------------identifikátory------------
      #define WS_SYSMENU           0x00080000L
      #define WS_CHILD             0x40000000L
      #define WS_VISIBLE           0x10000000L
      #define WS_BORDER            0x00800000L
      #define WS_TABSTOP           0x00010000L
      #define BS_DEFPUSHBUTTON     0x00000001L
      #define SS_CENTER            0x00000001L

      //----------------dialog-----------------
      DIAL1 DIALOG 130, 80, 120, 40
      STYLE WS_SYSMENU
      CAPTION "UDP Communication"
      {

      //textové pole, identifikátor 11
      CONTROL "",11, "edit", WS_CHILD | WS_BORDER | WS_VISIBLE | WS_TABSTOP,
15, 20, 50, 10

      //tlačítko, identifikátor 12
      CONTROL "OK!",12, "button", BS_DEFPUSHBUTTON | WS_CHILD | WS_VISIBLE |
WS_TABSTOP, 75, 20, 35, 10

      //staticky text, identifikátor 13
      CONTROL "Enter your login name:", 13, "static", WS_CHILD | WS_VISIBLE
| SS_CENTER, 15, 7, 90, 20

      }
```

Skript v příkladě vytvoří zdroj dialogového okna podle obrázku 8.3. Skript začíná blokem s definicemi konstant. Následuje identifikátor dialogu, textový řetězec “DIAL1“. Pouze identifikátory dialogů mohou být textové řetězce, identifikátory ostatních zdrojů jsou čísla. Tento řetězec bude potřeba později při vytváření tohoto dialogového okna v programu. Direktiva DIALOG pak říká, že se jedná o dialogové okno. Následují rozměry okna. Levý horní roh okna je na souřadnicích [130,80] a má okno má rozměry 120×40. Jednotky jsou v pixelech.

Direktivou STYLE se definuje styl okna. Konstanta WS_SYSMENU vytvoří systémové menu na dialogu. Titulek okna je nastaven direktivou CAPTION. Následují složené závorky. V těchto závorkách jsou definovány jednotlivé prvky dialogu.

První prvek je textové pole s identifikátorem 11. Styl WS_CHILD je povinný pro všechny prvky dialogu. Styl WS_VISIBLE znamená, že textové pole je viditelné. WS_BORDER vytvoří okraje v textovém poli. WS_TABSTOP pak definuje chování prvku, kdy po stisku klávesy TAB dojde ke změně zaměření mezi prvky jenž mají definovaný tento styl. Rozměry mají stejný význam jako v případě dialogového okna, s tím rozdílem, že levý horní roh textového pole je počítán od levého horního bodu dialogového okna a ne obrazovky.

Druhý prvek je tlačítko s identifikátorem 12. Tlačítko obsahuje text “OK!“. Styl BS_DEFPUSHBUTTON definuje standardní vlastnosti tlačítka.

Posledním prvkem je statický text který je stylem SS_CENTER vycentrován na střed.

8.2 Vytvořeni dialogového okna

K vytvoření okna lze použit funkce DialogBoxParam a CreateDialogParam. Rozdíl mezi těmito funkcemi je v tom, že DialogBoxParam vytváří modální dialogové okno a CreateDialogParam nemodální. Modální okna mají tu vlastnost, že program musí čekat dokud není okno zavřeno. Naproti tomu dialogové okna umožňují uživateli, aby se mohl přepínat mezi jednotlivými okny programu. Funkce jsou popsány tabulkami 8.2 a 8.3.

Tab. 8.2: Funkce DialogBoxParam

Syntaxe:	Popis poměných:
INT_PTR DialogBoxParam(	
HINSTANCE hInstance,	Handle instance programu.
LPCTSTR lpTemplateName,	Ukazatel na nulou ukončený řetězec který obsahuje identifikátor dialogu.
HWND hWndParent,	Handle na rodiče nebo vlastníka vytvářeného okna. Pokud žádné mateřské okno neexistuje, tak je předána hodnota NULL.
DLGPROC lpDialogFunc,	Ukazatel na proceduru dialogu.
LPARAM dwInitParam	Ukazatel na data, která budou předána jako LPARAM zprávy WM_INITDIALOG.
);	
Návratová hodnota:	V případě úspěchu vrací funkce handle objektu. V případě selhání, parametr hWndParent je neplatný, vrací funkce hodnotu 0. Pokud funkce selže z nějakého jiného důvodu, tak je návratovou hodnotou -1 (je možno volat GetLastError pro dodatečné informace).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v User32.lib.	

Tab. 8.3: Funkce CreateDialogParam

Syntaxe:	Popis poměných:
HWND CreateDialogParam(	
HINSTANCE hInstance,	Handle instance programu.
LPCTSTR lpTemplateName,	Ukazatel na nulou ukončený řetězec který obsahuje identifikátor dialogu.
HWND hWndParent,	Handle na rodiče nebo vlastníka vytvářeného okna. Pokud žádné mateřské okno neexistuje, tak je předána hodnota NULL.
DLGPROC lpDialogFunc,	Ukazatel na proceduru dialogu.
LPARAM dwInitParam	Ukazatel na data, která budou předána jako LPARAM zprávy WM_INITDIALOG.
);	
Návratová hodnota:	V případě úspěchu vrací funkce handle okna právě vytvořeného dialogu. V případě selhání funkce vrací NULL (je možno volat GetLastError pro dodatečné informace).
Implementováno v ANSI a Unicode verzi. Funkce se nachází v User32.lib.	

Příklad, vytvoření modálního dialogového okna:

```
[section .data class=DATA use32 align=16]

      I_Dial:          DB          "DIAL1",0

[section .code use32 class=CODE]

      push dword 0
      push dword DialProc
      push dword NULL
      push dword I_Dial
      push dword [hInstance]
      call [DialogBoxParam]
```

V datovém segmentu je deklarován řetězec “DIAL1“. Tento řetězec musí být shodný s identifikátorem dialogu.

Handle instance programu obsahuje proměnná [hInstance] a DialProc je ukazatel na proceduru dialogu, viz následující podkapitola.

Voláním funkce DialogBoxParam dojde k vytvoření modálního dialogového okna podle zdroje vytvořeného v předchozí kapitole. Při linkování výsledného souboru je samozřejmě nutné zkompilovaný soubor se zdroji přilinkovat.

8.3 Procedura dialogu

Procedura dialogu má svou analogii v proceduře okna. Rozdíly jsou minimální. Jedním z rozdílu je zaslání zprávy WM_INITDIALOG namísto WM_CREATE pří vytváření dialogu respektive okna. Zprávy určené pro dialogové okno nejprve zpracuje interní procedura dialogu, následuje volání procedury dialogu. Pokud procedura dialogu vrátí nulu, tak je opět volána interní procedura dialogu a dokončí zpracování zprávy.

Pro ukončení hlavního okna byla použita funkce PostQuitMessage. Dialogové okno se ukončí voláním funkce EndDialog. Má o jeden parametr více, handle na okno které se má ukončit.

Tab. 8.4: Funkce EndDialog

Syntaxe:	Popis poměných:
BOOL EndDialog(HWND hDlg, int nResult);	 Handle na dialogové okno jenž má být ukončeno. Specifikátor návratové hodnoty který bude výstupem funkce na vytvoření dialogového okna.
Návratová hodnota:	V případě úspěchu vrací funkce nenulovou hodnotu. V případě selhání funkce vrací hodnotu 0 (je možno volat GetLastError pro dodatečné informace).
Funkce se nachází v User32.lib.	

Příklad, procedura dialogu:

```
DialProc:
    DialProc_PARAMBYTES EQU 16
    enter 0,0       ;push EBP

         ; 1. parametr: handle   [ebp+8]
         ; 2. parametr: wMsg     [ebp+12]
         ; 3. parametr: wParam   [ebp+16]
         ; 4. parametr: lParam   [ebp+20]

   mov eax,dword [ebp+12]
   cmp eax,WM_INITDIALOG   ;je v eax zprava WM_INITDIALOG?
       je near .Create
   cmp eax,WM_CLOSE        ;je v eax zprava WM_CLOSE?
       je near .Destroy
   cmp eax,WM_PAINT        ;je v eax zprava WM_PAINT?
       je near .Paint
   cmp eax,WM_COMMAND      ;je v eax zprava WM_COMMAND?
       je near .Command
   jmp near .Finish

   .Create:
         ;návěští pro konstruktor
         jmp near .Finish

   .Destroy:
         ;návěští pro konstruktor
         push dword 1
         push dword [ebp+8]
         call [EndDialog]
         jmp near .Finish

   .Command:
         ;obslužná rutina pro zprávu WM_COMMAND
         jmp near .Finish


   .Paint:
         ;nastaveni fontu editu
         push dword 1
         push dword [hFont]
         push dword WM_SETFONT
         push dword 11
         push dword [ebp+8]
         call [SendDlgItemMessage]
         ;nastaveni fontu tlacitka
         push dword 1
         push dword [hFont]
         push dword WM_SETFONT
         push dword 12
         push dword [ebp+8]
         call [SendDlgItemMessage]
         ;nastaveni fontu statickeho textu
         push dword 1
         push dword [hFont]
         push dword WM_SETFONT
         push dword 13
         push dword [ebp+8]
         call [SendDlgItemMessage]
         jmp near .Finish
```

```
      .Finish:
            xor eax,eax
            leave
   ret DialProc_PARAMBYTES
```

V příkladu je vytvořena funkce se 4 parametry, handle dialogu, typ zprávy, wParam a lParam. V těle funkce se nejprve rozhoduje o jaký typ zprávy se jedná a podle toho je proveden skok na příslušné návěští. Pokud není zpráva v proceduře obsloužena, tak je proveden skok na návěští .Finish. Na tomto návěští se vynuluje obsah registru eax a provedou se operace nutné pro správné ukončení funkce.

Blok instrukcí na návěští .Create slouží jako konstruktor, tedy blok instrukcí na tomto návěští se provede jen jednou a to při vytvoření okna. Podobně blok na návěští .Destroy se provede jen jednou a to při ukončení okna. Na konci bloku dat na tomto návěští se provede volání funkce EndDialog pro ukončení dialogu. Druhý parametr funkce EndDialog informuje jak skončil dialog. Lze například díky tomuto parametr určit, jaké tlačítko na dialogu bylo stisknuté.

Návěstí .Command slouží k obsluze zpráv, které vyniknou při interakci s ovládacími prvky dialogového okna. Může se jednat například o stisk tlačítka.

Zpráva WM_PAINT je volána když dojde k překreslení okna, například při překrytí jiným oknem. Na návěští .Paint se provede nastavení fontů na všech prvcích dialogu. Tento font nemůže být nastaven v konstruktoru, protože pak text na těchto prvcích bliká. Nebo v horším případě se prvek vůbec nezobrazí. Typ fontu může být nastaven už přímo v souboru se zdroji, ale aby bylo zřejmé, jakým způsobem dochází k adresaci prvku na dialogu, je font nastaven dynamicky, při běhu programu.

Font se nastaví zaslání zprávy WM_SETFONT všem prvkům. Místo dříve používané funkce SendMessage je použita funkce SendDlgItemMessage, tabulka 8.5. To proto, že nejsou známy handle jednotlivých prvků na dialogu. Je samozřejmě možné tyto handle zjistit, ale mnohem elegantnější je použít funkci SendDlgItemMessage a adresovat je pomocí handle dialogu a identifikátoru prvku. Handle dialogu je znám, je v [epb+8]. Hodnota 1 v parametru lParam funkce SendDlgItemMessage znamená, že font se změní okamžitě po příjmu zprávy. Proměnná [hFont] obsahuje handle fontu.

Tab. 8.5: Funkce SendDlgItemMessage

Syntaxe:	Popis poměných:
LRESULT SendDlgItemMessage(	
HWND hDlg,	Handle dialogového okna.
int nIDDlgItem,	Identifikátor dialogového prvku.
UINT Msg,	Identifikátor zprávy (druh zprávy).
WPARAM wParam,	Dodatečné informace o zprávě.
LPARAM lParam	Dodatečné informace o zprávě.
);	
Návratová hodnota:	Návratový kód je závislý na výsledku zpracování zprávy a také na jejím typu.
Implementováno v ANSI a Unicode verzi.	
Funkce se nachází v user32.dll.	

9 Závěr

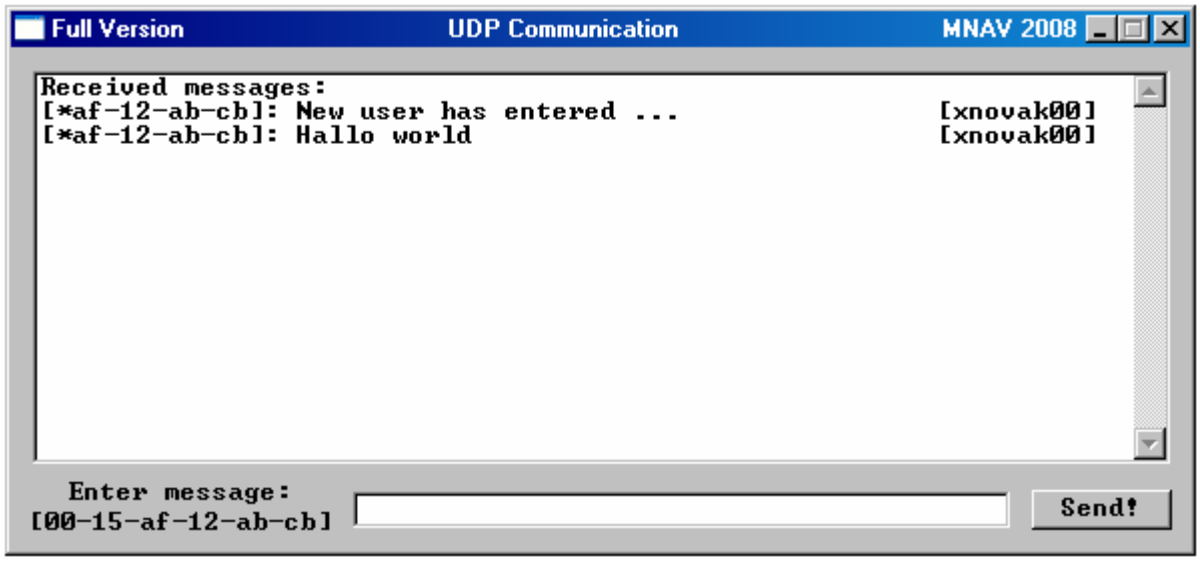
Cíle diplomové lze rozdělit do čtyř bodů. Prvním bodem je nastudování základů 32bitového popř. 64bitového programování v jazyce symbolických instrukcí (dále jen JSI) ve Windows. Kapitoly číslo 1, 2 a 3 se zabývají nastudováním této problematiky. První kapitola v textu diplomové práce popisuje programově aplikační rozhraní Win32, nástroje pro práci s tímto rozhraním v JSI, tvorbou dynamicky linkovaných knihoven a vytváření funkcí v JSI. Druhá kapitola pojedná o základní práci s okny. Jelikož většina prvků, například tlačítka nebo nabídky, jsou z pohledu programovaní ve Win32 okny, je tato kapitola detailněji prostudována. V závěru kapitoly je uvedeno několik příkladů. Poslední kapitola, která se zabývá základy programování v JSI, je věnována zprávám, smyčce zpracování zpráv a proceduře okna.

Dalším bodem v zadání diplomové práce je seznámení se s programátorskými možnostmi pod Win32 se zaměřením na prostředky sloužící ke komunikaci mezi počítači. V kapitolách číslo 4,5 a 6 je vypracován tento bod zadání. Nejprve je v teoreticky zaměřené kapitole prostudována rodina protokolů TCP/IP, architektura client-server a architektura peer to peer. A dále pak možnosti vytváření těchto architektur s použitím transportních protokolů TCP a UDP. Následující dvě kapitoly se zaměřují na prostředky Win32 slouží k vytvoření těchto architektur. Kapitola 5 se zabývá prostředky Win32 sloužícími k vytvoření architektury Peer to peer pomocí protokolu UDP. Kapitola 6 pak prostředky Win32 k vytvoření Client to server architektury skrz protokol TCP.

Třetím bodem zadání je přehledně zpracovat vybrané úlohy v JSI s odpovídajícím popisem a komentářem. Každá z probíraných kapitol ukončuje svůj výklad praktickým příkladem na probírané téma. Jak již bylo řečeno v úvodu, je to proto, aby text diplomové práce mohl sloužit studentům předmětu MNAV jako studijní opora pro počítačová cvičení. Z tohoto důvodu je zvolen použitý styl výkladu, kdy je nejprve teoreticky prostudována látka, následuje popis funkcí z Win32 API v přehledných tabulkách a závěr kapitol je věnován popsaným a okomentovaným příkladům.

Poslední bod zadání je připravit části úloh pro počítačová cvičení. Pro splnění tohoto bodu byla vytvořena vzorová aplikace v JSI. Tato vzorová aplikace je jednoduchý program pro komunikaci uvnitř sítí LAN. Aplikace používá prostředků Win32 a rodiny protokolů TCP/IP. Aplikace je koncipovaná v architektuře Peer to peer. Jednotlivé zprávy jsou jako UDP datagramy vysílány na adresu sítě. Ostatní uživatelé tyto datagramy přijímají. Komunikace probíhá uvnitř sítí LAN na UDP portu 5000. Jednotliví uživatelé jsou odlišovaní na základě VUT jména (např. xnovak00) a MAC adresy síťového rozhraní. Kapitoly číslo 7 a 8 se zabývají programátorskými technikami, které v předchozích kapitolách nebyly dosud prostudovány, ale byly použity ve vzorové aplikaci. Jedná se o kapitolu zabývající se multitaskingem. Jádrem kapitoly je prostudování prostředků Win32 API pro práci s vlákny. Vzorová aplikace totiž pracuje ve dvou vláknech, kdy hlavní vlákno má na starosti ošetření ovládacích prvků programu a vysílání zpráv. Vedlejší vlákno zpracovává zprávy od ostatních uživatelů. V poslední kapitole je probrána tvorba dialogových oken pomocí tzv. zdrojů (Resource script). Hlavní okno vzorové aplikace lze vidět na obrázku 9.1.

Tímto jsem přesvědčen o tom, že bylo splněno zadání diplomové práce, a to ve všech jeho bodech.



Obr. 9.1: Hlavní okno vzorové aplikace

9 Literatura

[1] SIMON R, J., GOUKER, M. a BARNES, B., C. *Win32 API - průvodce vývojáře (svazek 1-3)*, UNIS publishing s.r.o, Brno 1997, ISBN 80-86097-06-4

[2] NÁDBELA, J. *Velký počítačový slovník*. Computer Media, Kralice na Hané 2004, ISBN: 80-86686-56-6

[3] ERICKSON, J. *Hacking, umění exploitace*, ZonerPress, Brno 2003, ISBN 80-86815-21-8

[4] PIROGOV, V. *Mistrovství v jazyce Assembler - Programování*, disassembling, analýza kódu. Computer Press. Brno 2006, ISBN: 80-251-0888-0

[5] JIROVSKÝ, V. *Vademecum správce sítě*. Grada, Praha 2001, ISBN 80-7169-745-1

[6] MAREK, R. *Učíme se programovat v jazyce Assembler pro PC*. Computer Press, Brno 2003, ISBN 80-7226-843-0

[7] BALÍK, M. *Počítače a jejich periférie - počítačová cvičení*. Elektronické skriptum FEKT VUT Brno.

[8] ORSÁG, F. *Pokročilé asemblery – studijní opora*. Elektronické skriptum FIT VUT Brno.

[9] MSDN, *Overview of the Windows API*. 3/27/2008, dostupné na WWW: http://msdn.microsoft.com/en-us/library/aa383723(VS.85).aspx

[10] MSDN, *About Window Classes*. dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms633574.aspx

[11] KYLE, M. *Win32 Window Hierarchy and Styles*. 29/9/1993, dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms997562.aspx

[12] MSDN, *WSAStartup Function*. 5/15/2008 , dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms742213.aspx

[13] MSDN, *Socket Function*. 5/15/2008, dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms740506.aspx

[14] MSDN, *WNDCLASSEX Structure*. dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms633577.aspx

[15] MESSMER, H., P. a DEMBOWOSKI, K. *Velká kniha hardware, architektura, funkce, programování*. Computer Press. Brno 2005, ISBN: 80-251-0416-8

[16] MSDN, *GetStockObject*. dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms533223(VS.85).aspx

[17] MSDN, *SECURITY_ATTRIBUTES Structure*. 4/24/2008, dostupné na WWW: http://msdn2.microsoft.com/en-us/library/aa379560(VS.85).aspx

[18] MSDN, *recvfrom Function*. 5/15/2008,
dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms740120.aspx

[19] MSDN, *MessageBox Function*.
dostupné na WWW: http://msdn2.microsoft.com/en-us/library/ms645505.aspx

[20] WIKIPEDIE, *Internet Protocol version 4*. dostupné na WWW: http://en.wikipedia.org/wiki/IPv4

[21] SAMURAJ, *TCP/IP - model, encapsulace, paket vs. rámec*. 16/08/2007, dostupné na WWW: http://www.samuraj-cz.com/clanek/tcpip-model-encapsulace-paketu-vs-ramec

[22] WIKIPEDIE, *Ethernet*. dostupné na WWW: http://en.wikipedia.org/wiki/Ethernet

[23] WIKIPEDIE, *Peer-to-peer*. dostupné na WWW: http://cs.wikipedia.org/wiki/Peer-to-peer

[24] WIKIPEDIE, *User Datagram Protocol*. dostupné na WWW: http://en.wikipedia.org/wiki/User_Datagram_Protocol

[25] WIKIPEDIE, *Transmission_Control_Protocol*. dostupné na WWW: http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[26] MALLAT, J. *NetBIOS*. 04/08/2004, dostupné na WWW: http://hps.mallat.cz/view.php?cisloclanku=2004080202

[27] MSDN, *NCB Structure.* 2/28/2008, dostupné na WWW: http://msdn.microsoft.com/en-us/library/bb870902.aspx

[28] MSDN, *ADAPTER_STATUS Structure.* 2/28/2008, dostupné na WWW: http://msdn.microsoft.com/en-us/library/bb870890.aspx

[29] HEROUT, P. *Učebnice jazyka C*. Kopp. České Budějovice 2001, ISBN: 80-858228-21-9