PROGRAMOVÁNÍ

MARTIN DRÁB

JÁDRO SYSTÉMU WINDOWS

KOMPLETNÍ PRŮVODCE PROGRAMÁTORA

TEORIE I RYZE PRAKTICKÉ UKÁZKY
PROGRAMOVÁNÍ OVLADAČŮ JÁDRA
PRÁCE S VLÁKNY A PROCESY, SPRÁVA PAMĚTI
SKRYTÁ ZÁKOUTÍ ÚTROB REGISTRU

CPRESS

Martin Dráb

Jádro systému Windows
Kompletní průvodce programátora

Computer Press, a. s.
Brno
2011

Jádro systému Windows
Kompletní průvodce programátora

Martin Dráb

Computer Press, a. s., 2011. Vydání první.

Jazyková korektura: Zuzana Marková
Sazba: Kateřina Kiszková
Rejstřík: Kateřina Kiszková
Obálka: Martin Sodomka

Komentář na zadní straně obálky: Libor Pácl
Odpovědný redaktor: Libor Pácl
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Computer Press, a. s.,
Holandská 3, 639 00 Brno

Objednávky knih:
http://knihy.cpress.cz
distribuce@cpress.cz
tel.: 800 555 513

ISBN 978-80-251-2731-5
Prodejní kód: K1741
Vydalo nakladatelství Computer Press, a. s., jako svou 4042. publikaci.

Stručný obsah

1.	**Operační systémy**	**17**
2.	**Architektura rodiny operačních systémů Windows NT**	**45**
3.	**Vývoj ovladačů jádra**	**65**
4.	**Synchronizace**	**113**
5.	**Výjimky, přerušení a systémová volání**	**147**
6.	**Správce objektů (Object Manager)**	**201**
7.	**Procesy a vlákna**	**259**
8.	**Správce vstupně/výstupních operací**	**305**
9.	**Správa paměti**	**327**
10.	**Registr**	**393**
11.	**Souborové systémy**	**431**
	Použité zdroje	**461**
	Rejstřík	**463**

Obsah

Úvod **13**
Zdrojové kódy projektů **13**
Co v knize najdete **14**
Zpětná vazba od čtenářů **15**
Zdrojové kódy ke knize **16**
Errata **16**

KAPITOLA 1
Operační systémy **17**

Základní pojmy **17**
Procesor, úrovně oprávnění a systémová volání 17
Virtuální paměť 18
Procesy, vlákna, joby 20
Knihovny DLL a rozhraní Windows API 21
Služby a ovladače 23
Historie Windows **23**
Windows 1 23
Windows 2 24
Windows 3 a OS/2 24
Windows NT 24
Windows 95, Windows 98 a Windows Me 25
Windows 2000 25
Windows XP 26
Windows Vista 26
Windows 7 27
Serverové verze 29
Základní datové struktury užívané v operačních systémech **29**
Pole 30
Spojové seznamy 32
Zásobník 36
Fronta 37
Hašovací tabulky 38
Stromy 40

KAPITOLA 2
Architektura rodiny operačních systémů Windows NT 45

Mikrojádro a monolitický operační systém 45
Windows NT a jeho součásti 47
Vrstva abstrakce hardwaru (Hardware Abstraction Layer – HAL) 47
Tvrdé jádro 48
Ovladače 48
Exekutiva 48
Subsystémy 52
Systémové procesy 55

OKAPITOLA 3
Vývoj ovladačů jádra 65

Co je to ovladač 65
Prostředí pro programování 68
Jak přeložit ovladač 68
Načtení ovladače do jádra 72
Čistý a oficiální způsob 72
Méně známý způsob (nativní funkce NtLoadDriver) 75
Méně známý způsob (nativní funkce NtSetSystemInformation) 79
Jednoduchý příklad: Klasické „Hello World!“ 81
Několik poznámek k ladění ovladačů 83
DbgPrint 84
DbgPrintEx 84
ASSERT 86
KdPrint a KdPrintEx 87
WinDbg 87
Modrá obrazovka smrti 91
Okolnosti vzniku 92
Průběh 92
Nastavení výpisu příčin selhání 94
Zjišťování příčin modrých obrazovek 96
Závěrečný příklad 97
Způsob uchovávání událostí 98
Použité rozhraní pro zachytávání událostí 100
Inicializace a úklid 102
Komunikace s aplikací 106

KAPITOLA 4

Synchronizace 113

Modelový příklad 113
Kritická sekce 115
Vybraná řešení problému kritické sekce 116
Zakázání přerušení 116
Atomické operace 117
Instrukce test and set (TSL) a zámky 117
Známé synchronizační problémy 118
Synchronizační primitiva implementovaná na systémech Windows NT 126
Spinlock 126
Spinlock s frontou (Queued Spinlock) 127
Zásobníkový spinlock s frontou (In Stack Queued Spinlock) 128
Složitější atomické operace (Interlocked operations) 129
Objekt dispatcher 130
Událost (event) 131
Semafor (semaphore) 135
Mutex (Mutant) 136
Rychlé a strážené mutexy (fast mutexes, guarded mutexes) 138
Zámky reader-writer určené pro ovladače (Executive resources) 139
Pushlock 141
Kritická sekce (critical section) 142
Událost na klíč (Keyed Event) 143
Podoba konečného řešení problému s kritickými sekcemi 144
Zámek reader-writer pro uživatelský režim (Slim Reader Writer Lock – SRW Lock) 145
Další synchronizační primitiva? 146

KAPITOLA 5

Výjimky, přerušení a systémová volání 147

Komunikace s hardware 147
Dotazování (polling) 147
Obsluha přerušení (interrupt handling) 148
Obsluha přerušení na architekturách x86 a x64 148
Výjimky 150
Hardwarová přerušení ve Windows a hardwarové priority 155
Hardwarové priority (IRQL) 157
Předdefinované hodnoty IRQL 158
Odložené volání procedur (Deferred Procedure Call – DPC) 160
Využití objektů DPC 162

Pracovní vlákna (worker threads) **169**
Asynchronní volání procedur (Asynchronous Procedure Call – APC) **170**
Systémová volání **171**
Průběh systémového volání 173
SSDTInfo: Získání informací o tabulkách systémových volání **186**
Interrupt Counter: Monitorování přerušení **189**
Syscallmon: Monitorování systémových volání **195**

KAPITOLA 6
Správce objektů (Object Manager) **201**

Požadavky **201**
Objekty exekutivy **202**
Struktura objektu exekutivy **205**
Hlavička (object header) **205**
Hlavička OBJECT_HEADER_NAME_INFO 209
Hlavička OBJECT_HEADER_CREATOR_INFO 210
Hlavička OBJECT_HEADER_HANDLE_INFO 210
Hlavičky OBJECT_HEADER_QUOTA_INFO a OBJECT_HEADER_PROCESS_INFO 211
Struktura OBJECT_CREATE_INFORMATION 212
Příklady **213**
Tělo objektu **215**
Objekty reprezentující typ (struktura OBJECT_TYPE) **218**
Struktura objektu ObjectType 219
Struktura OBJECT_TYPE_INITIALIZER 225
Handle a jejich tabulky **235**
Vlastnosti handle 235
Skutečný význam hodnoty handle 237
Zranitelnosti v bezpečnostním software 239
Detaily struktury HANDLE_TABLE_ENTRY 239
Popis některých funkcí pro práci s handle 240
Jména objektů, adresáře a symbolické odkazy **246**
Adresáře 248
Symbolické odkazy 250
Důležité oblasti jmenného prostoru 251
Relace (sessions) 252
Kradení jmen (object name squatting) 253
Počítání referencí **253**
Příklad: ObjView **256**
Příklad: ObInit **257**

KAPITOLA 7

Procesy a vlákna 259

Obecně o procesech a vláknech 259
Definice 259
Stavy procesů a vláken 262
Plánování 265
Možnosti implementace 270
Procesy a vlákna ve Windows 272
Reprezentace 272
Lokální úložiště vláken (Thread Local Storage) 281
Vznik nových procesů a vláken 282
Plánovač procesů a vláken 284
Chráněné procesy 297
Objekty Job 302
Fibery – vlákna implementovaná čistě v uživatelském režimu 304

KAPITOLA 8

Správce vstupně/výstupních operací 305

Základní přehled 305
Standardní způsob komunikace mezi aplikací a ovladačem 310
Způsoby přenosu dat zprávy 313
Bufferovaná metoda (METHOD_BUFFERED) 314
Metoda přímého vstupu (METHOD_IN_DIRECT) 317
Metoda přímého výstupu (METHOD_OUT_DIRECT) 317
Metoda nulové režie (METHOD_NEITHER) 319
Obsluha požadavků 320
Funkční ovladače 320
Filtry 321
Rychlý vstup a výstup (Fast I/O) 324

KAPITOLA 9

Správa paměti 327

Historický úvod 327
Virtuální paměť 329
Segmentace 332
Stránkování 335
Výběr oběti 338

Příklad implementace virtuální paměti: Intel x86 344
Datové segmenty 346
Kódové segmenty 346
Selektor 347
Stránkování 348
Přidělování bloků paměti proměnlivé velikosti 353
Správa paměti ve Windows NT 355
Struktura virtuálního adresového prostoru jádra 356
Práce s virtuální pamětí 358
Address Windowing Extension (AWE) 367
Paměťově mapované soubory 369
Interní reprezentace struktury virtuálního adresového prostoru 378
MDL (Memory Descriptor List) 380
Práce na haldě 384

KAPITOLA 10
Registr 393

Pohled shora 394
Operace nad registrem 396
Podpora starších 32bitových aplikací na 64bitových platformách 406
Virtualizace (Registry Virtualization) 406
Přesměrování (Registry Redirector) a reflexe (Registry Reflection) 407
Interní struktura 411
Soubory registru (hive) 411
Buňka (cell) 416
Monitorování a filtrování operací nad registrem 424
Kontrola registru na bázi modifikace tabulky systémových volání 425
Kontrola registru pomocí speciálního rozhraní 426

KAPITOLA 11
Souborové systémy 431

FAT 432
Adresáře 437
Dlouhá jména 439
NTFS 440
Bezpečnostní model 441
Hard linky 441
Soft linky (symlinky, symbolické odkazy) 442
Alternativní datové proudy 443

Řídké soubory 444
Defragmentace 444
Komprese a šifrování 447
Žurnálování a transakce 448
Žurnál USN (USN Change Journal) 449
Interní struktura 450
Speciální soubory 456

Použité zdroje 461

Rejstřík 463

Úvod

Tato kniha v jedenácti kapitolách pojednává o různých aspektech jádra operačních systémů rodiny Windows NT. Neklade si však za cíl toto téma zpracovat do nejmenších podrobností; spíše se čtenáři snaží vštípit základní informace a obohatit jej o některé zajímavosti, na něž může při průzkumu jádra narazit.

Kniha se snaží postupy a algoritmy používané v jádře Windows uvést do širších souvislostí teorie operačních systémů. Z tohoto důvodu jsou některé kapitoly rozděleny na dvě části – první se věnuje dané problematice (například synchronizaci či správě paměti) obecně a druhá ukazuje, jaké poznatky a algoritmy se vývojáři a návrháři jádra Windows rozhodli použít. Tímto uspořádáním se kniha snaží ukázat, že mnohé postupy si Microsoft nevymyslel na „zelené louce", ale vychází z teoreticky podložených faktů. Varianty některých zde popsaných algoritmů se nacházejí i v jádrech jiných operačních systémů, například těch založených na Unixu.

Dalším a posledním cílem knihy je naučit čtenáře (pokud bude chtít) pohybovat se v jádře a samostatně zkoumat jeho vnitřní mechanismy a zákonitosti. Z tohoto důvodu kniha zahrnuje i popis v tomto ohledu užitečných nástrojů a snaží se poskytnout neformální základy programování ovladačů. Za tímto účelem spolu s touto publikací vzniká webová stránka *http://www.jadro-windows.cz*, na které naleznete okomentované zdrojové kódy ovladačů jádra, jež prakticky ukazují některé aspekty probírané v jednotlivých kapitolách. Na části těchto zdrojových kódů narazíte i v textu knihy formou výpisů. Dále na zmíněném webu naleznete materiály vhodné pro další rozšiřování znalostí a užitečné nástroje.

Zdrojové kódy projektů

Konkrétně na webové stránce *jadro-windows.cz* naleznete zdrojové kódy následujících projektů:

- **Dllhide** – program demonstruje, jak lze manipulací s interními datovými strukturami procesu skrýt knihovny DLL, které používá.
- **Drv** – tento program na základě argumentů příkazového řádku dokáže načítat a odstraňovat ovladače jádra. Ukazuje, jak tyto operace provádět různými způsoby.
- **Filemaptest** – ukazuje, jakým způsobem je možné využít sdílené paměti vytvořené pomocí paměťově mapovaného souboru a přenášet data mezi dvěma procesy.
- **Hello** – velmi jednoduchý ovladač, který pouze vypíše několik oznámení do debuggeru jádra. Ačkoliv neprovádí prakticky žádné operace, jeho zdrojový kód ukazuje, co musí každý ovladač minimálně umět.
- **Intcount** – ovladač a aplikace, které si kladou za cíl zjistit statistiku vykonávání jednotlivých přerušení. Cílem této ukázky je demonstrovat práci s tabulkou vektorů přerušení a synchronizaci více procesorů.
- **Keyedevent** – jádro Windows implementuje zajímavá synchronizační primitiva, která jsou v této knize označována jako události na klíč (keyed events). Ačkoliv je mohou používat i normální aplikace, příslušné rozhraní není dokumentováno. Projekt `keyedevent` práci s tímto rozhraním zjednodušuje a dokumentuje jej.

- **Listdll** – tento program tvoří protipól k projektu `dllhide`. Na základě argumentů příkazového řádku se snaží zjistit seznam knihoven DLL používaných cílovým procesem. Ukazuje různé způsoby, jak tyto informace zjistit.
- **Logptm** – projekt ukazuje, jak pomocí relativně jednoduchého ovladače jádra monitorovat spouštění a ukončování procesů a vláken a načítání knihoven DLL a dalších spustitelných souborů do paměti. Ovladač k tomuto účelu využívá velmi staré a dokumentované rozhraní.
- **Ntqueryobject** – jednoduchý program, který demonstruje použití nativní funkce `NtQueryObject` ke zjištění různých zajímavých informací.
- **Obinit** – tento projekt ukazuje, jak pomocí techniky DKOH (Direct Kernel Object Hooking) monitorovat přístupy k různým objektům operačního systému (souborům, klíčům registru, procesům, vláknům a dalším).
- **Objview** – projekt, který ukazuje, jakým způsobem je možné implementovat funkce, kterými disponuje utilita `WinObj` ze serveru *www.sysinternals.com*. Jedná se o prohlížeč pojmenovaných objektů existujících v jádře operačního systému.
- **Pproc** – Windows Vista mimo jiné zavádí nový druh procesů – tzv. *chráněné procesy* (protected processes). Cílem projektu `pproc` je umožnit vám libovolný proces označit jako chráněný a opačně. Dále projekt také obsahuje testovací program, který prakticky demonstruje, jaké možnosti a omezení chráněné procesy s sebou přináší.
- **Registrymon** – projekt ukazuje, jak implementovat funkcionalitu podobnou aplikaci `Regmon`, kterou jste mohli dříve nalézt na serveru *www.sysinternals.com*, tedy monitorování operací nad Registrem Windows.
- **SSDTInfo** – systémová volání patří k jednomu z nejdůležitějších mechanismů ve Windows. Projekt `SSDTInfo` ukazuje, jak zjistit zajímavé informace o interních datových strukturách, které implementace tohoto mechanismu používá.
- **Syscallmon** – projekt ukazuje, jakým způsobem lze monitorovat systémová volání.
- **Vad** – bloky alokované a rezervované paměti procesu reprezentuje jádro Windows pomocí struktur VAD (Virtual Address Descriptor). Projekt `vad` demonstruje, jak s těmito strukturami pracovat a získat z nich užitečné informace.

Všechny zdrojové kódy jsou psány v programovacích jazycích C a Object Pascal a vývojových prostředích Delphi XE 2010 a Microsoft Visual Studio. Jednotlivé programy a ovladače, až na výjimky, fungují na 32bitových i 64bitových verzích Windows XP, Windows Server 2003, Windows Vista a Windows 7. Delphi je použito převážně z důvodu snadné tvorby grafického uživatelského rozhraní.

Co v knize najdete

První tři kapitoly knihy tvoří úvod do problematiky operačních systémů rodiny Windows NT. První kapitola nejprve vysvětluje základní pojmy, jako je proces, vlákno, systémové volání či handle. Dále pokračuje stručným popisem jednotlivých verzí Windows; od Windows 1 až po současná Windows 7. Ve své třetí části kapitola popisuje základní datové struktury, mezi které patří pole, spojový seznam, fronta či zásobník a které jádra operačních systémů často využívají k uchovávání různých informací.

Druhá kapitola již trochu sestupuje z teoretických výšin kapitoly první a obecně pojednává o jednotlivých částech jádra Windows a dalších komponentách, bez kterých by operační systém nefungoval. Dočtete se v ní také o službách – programech (a ovladačích), jejichž posláním je vykonávat důležité činnosti na pozadí.

Třetí kapitola popisuje některé postupy a principy programování ovladačů jádra. Dozvíte se, jaké nástroje potřebujete a jak ovladač načíst do paměti jádra. V závěrečné části je popsána struktura ovladače `logptm.sys`. Kapitola se informace snaží podávat méně formálním způsobem; kterým se autor této knihy učil poznávat zákoutí jádra Windows.

Další kapitoly se již věnují jednotlivým aspektům operačního systému, i když u některých nechybí obecný úvod. Ve čtvrté kapitole se dočtete o způsobech řízení přístupu více aplikací (vláken) ke sdíleným prostředkům. Pátá kapitola popisuje mechanismy obsluhy přerušení, odloženého volání procedur a systémových volání. Následující kapitola pojednává o tom, jak jádro Windows využívá principů objektově orientovaného programování.

Sedmá kapitola se věnuje procesům, vláknům a jejich plánování. V osmé se dočtete o způsobech předávání dat mezi aplikací a ovladačem a mezi ovladači navzájem. Jedná se o rozšíření poznatků neformálně sdělených ve třetí kapitole. Devátá kapitola se zabývá správou paměti. Popisuje jak různé operace s virtuální pamětí, které jádro Windows podporuje, tak dává lehce nahlédnout i do interních datových struktur, které správce paměti používá.

Desátá a jedenáctá kapitola popisují formáty, které Windows používají pro ukládání dat na externí média, například pevné disky. Desátá kapitola se věnuje Registru, struktuře určené ukládání nastavení aplikací a celého operačního systému. Popisuje tento formát z hlediska programátora a uživatele. Její podstatná část pojednává i o tom, jak je tato „databáze“ fyzicky uložena na pevném disku. Jedenáctá kapitola popisuje interní datové struktury dvou na Windows nejrozšířenějších souborových systémů – FAT a NTFS.

Zpětná vazba od čtenářů

Nakladatelství a vydavatelství Computer Press stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

redakce PC literatury
Computer Press
Spielberk Office Centre
Holandská 3
639 00 Brno

nebo

sefredaktor.pc@cpress.cz

Zdrojové kódy ke knize

Z adresy *http://knihy.cpress.cz/K1741* si po klepnutí na odkaz Soubory ke stažení můžete přímo stáhnout archiv s ukázkovými kódy.

Errata

Přestože jsme udělali maximum pro to, abychom zajistili přesnost a správnost obsahu, chybám se úplně vyhnout nelze. Pokud v některé z našich knih najdete chybu, ať už v textu nebo v kódu, budeme rádi, pokud nám ji nahlásíte. Ostatní uživatele tak můžete ušetřit frustrace a pomoci nám zlepšit následující vydání této knihy.

Veškerá existující errata zobrazíte na adrese *http://knihy.cpress.cz/K1741* po klepnutí na odkaz Soubory ke stažení.

Operační systémy

První kapitola, jak bývá u tohoto druhu knih dobrým zvykem, je obecným úvodem do problematiky operačních systémů. Nejprve se seznámíte se základními pojmy, kterým je třeba při čtení dalších kapitol alespoň rámcově rozumět. Druhá část je rychlou exkurzí do historie Windows a ve třetí se dozvíte několik základních způsobů, jak si operační systémy pamatují důležité informace, které potřebují pro svůj efektivní běh.

Základní pojmy

Následující odstavce vás letmo provedou základními pojmy, jejichž znalost je nutná pro pochopení obsahu dalších částí knihy. Text této části nezabíhá do přílišných podrobností, ale klade si za cíl, abyste si o jednotlivých pojmech utvořili rámcovou představu. Podrobnější popis většiny z nich najdete v některé z dalších kapitol. S většinou zde vysvětlených termínů a principů se setkáte i při studiu jiných operačních systémů.

Procesor, úrovně oprávnění a systémová volání

Procesor je hlavním nástrojem operačního systému, který tento malý čip využívá k různým vlastním výpočtům a dává jej k dispozici i aplikacím, jež vy jako uživatel spouštíte. Procesor v sobě implementuje řadu mechanismů, kterých operační systém využívá například pro zaručení ochrany hardwaru před zneužitím zákeřným programem či pro ochranu kódu svého jádra před nežádoucími modifikacemi.

Mezi tyto mechanismy patří možnost vykonávat kód programu na několika *úrovních oprávnění*, přičemž každá z nich přesně definuje, které operace jsou povoleny a které zakázány. Omezení se vztahuje například na druhy instrukcí, jež lze na jednotlivých úrovních použít.

Například procesory Intel kompatibilní s architekturou x86 mají v sobě zabudovány čtyři různé úrovně oprávnění. Tyto úrovně se často označují jako *ring módy* (*rings*) a obvykle se číslují od 0 do 3. Čím nižší číslo, tím větší volnost má kód běžící na dané úrovni. Program vykonávaný na úrovni Ring 0 má dovoleno využít veškeré možnosti, které mu instrukční sada procesoru nabízí – může přímo komunikovat s hardware, ovládat chování procesoru či měnit obsah důležitých systémových datových struktur, jako je tabulka vektorů přerušení či globální tabulka segmentů. Úrovně Ring 1 a Ring 2 již některé instrukce vykonávat nepovolují a kódu běžícímu na úrovni Ring 3 procesor neumožňuje žádným způsobem přímo měnit nastavení, která by mohla ovlivnit chování operačního systému. Úroveň Ring 3 například nedovoluje přímou komunikaci s hardware.

Z historických důvodů Windows používají pouze úrovně Ring 0, na které běží veškerý kód jádra a ovladačů, a Ring 3, která slouží pro vykonávání kódu normálních aplikací (též označovaných

jako *procesy*). Obecně se úroveň, která poskytuje nejvyšší možná oprávnění, označuje jako *režim jádra* (*kernel mode*). Naopak úroveň kladoucí nejvyšší omezení na vykonávaný kód se nazývá *uživatelský režim* (*user mode*).

Běžné aplikace samozřejmě nevystačí s úrovní Ring 3, protože potřebují čas od času komunikovat s okolím – číst a zapisovat na pevný disk, posílat data po síti nebo kreslit na obrazovku. Protože úroveň Ring 3 z bezpečnostních důvodů neumožňuje přímou komunikaci s hardware, byl do procesoru implementován mechanismus, který aplikaci umožňuje přejít za určitých podmínek na úroveň Ring 0 a tam vyřídit vše potřebné. Aplikace samozřejmě nemůže sama určit, který blok kódu bude v režimu jádra vykonán; to určuje jádro při zavádění operačního systému.

Aplikace může pouze *požádat* jádro (zavolat systém – provést *systémové volání*), aby pro ni vykonalo příslušnou operaci, jenž není v Ring 3 povolena. Jádro může požadavku aplikace vyhovět, ale také nemusí. Operační systém totiž implementuje vlastní bezpečnostní model, který umožňuje například chránit důležité soubory před neoprávněnou manipulací. Pokud se nějaký program spuštěný uživatelem s příliš nízkými právy pokusí k chráněnému souboru získat přístup, jádro obdrží požadavek na otevření souboru, ale neprovede jej, protože nedůvěryhodným uživatelům není povoleno otevírat chráněné soubory.

Poznámka: Operační systém obvykle neposuzuje důvěryhodnost programů, ale důvěryhodnost (a oprávnění) uživatelů, kteří je spustili nebo pod kterými vykonávají svůj kód. Program `test.exe` běžící pod administrátorským uživatelským účtem může měnit i obsah důležitých systémových souborů a složek. Stejnému programu běžícímu s právy běžného uživatele však operační systém měnit systémové soubory a nastavení nedovolí.
Rozhodování o důvěryhodnosti na základě obsahu binárního souboru aplikace a jejího chování provádějí antivirové programy a jiný bezpečnostní software.

Jakmile jádro zpracování požadavku dokončí, vrátí řízení aplikaci a její kód se začne vykonávat opět na úrovni Ring 3.

Princip popsaný v předchozích třech odstavcích se nazývá *mechanismus systémových volání* a detailně se jím zabývá kapitola 5.

Virtuální paměť

Jedním z úkolů operačního systému je izolovat jednotlivé běžící aplikace od sebe takovým způsobem, aby jedna nemohla (ať úmyslně nebo neúmyslně) škodit druhé, nemá-li k tomu potřebná oprávnění. A aplikace nemůže škodit jiným aplikacím, pokud nedokáže číst a měnit jejich paměť. Aby byla tato podmínka splněna, musí systém každému programu vyhradit oblast paměti, kam nemůže nikdo jiný přistupovat. Jedna z cest, jak toho dosáhnout, vede skrz mechanismus *virtuální paměti*, který je zabudován do většiny moderních procesorů.

Celý princip stojí na myšlence *virtuálních adres*. Drtivá většina kódu (tedy i kód jádra operačního systému) nepracuje přímo s fyzickými adresami (adresami používanými pro přímý přístup do operační paměti), ale s adresami virtuálními. Mapování virtuálních adres na fyzické je uloženo ve speciálních datových strukturách – například *stránkovacích tabulkách*. A s nimi může manipulovat pouze kód běžící v režimu jádra.

Poznámka: S fyzickými adresami pracuje obvykle pouze kód spravující právě datové struktury, které uchovávají informace o překladu virtuálních adres. Běžným aplikacím operační systém (a ani procesor) nedovoluje s fyzickými adresami pracovat vůbec.

Vlastní překlad virtuálních adres na fyzické zajišťuje část procesoru známá pod názvem *memory management unit* (MMU). Protože překlad probíhá na úrovni hardware, je pro aplikaci (a i pro většinu jádra) zcela transparentní. Program si klidně může „myslet", že pracuje přímo s fyzickou pamětí, protože překládání adres je před ním skryto.

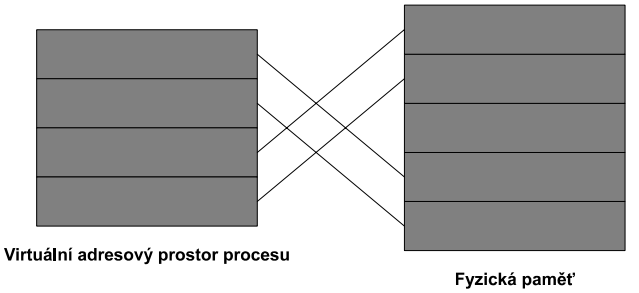
Překlad obvykle není možné definovat pro jednotlivé virtuální adresy, ale pro jejich bloky (například o velikosti 4 KB), které se označují jako *stránky*.

Použití virtuální paměti poskytuje následující výhody:

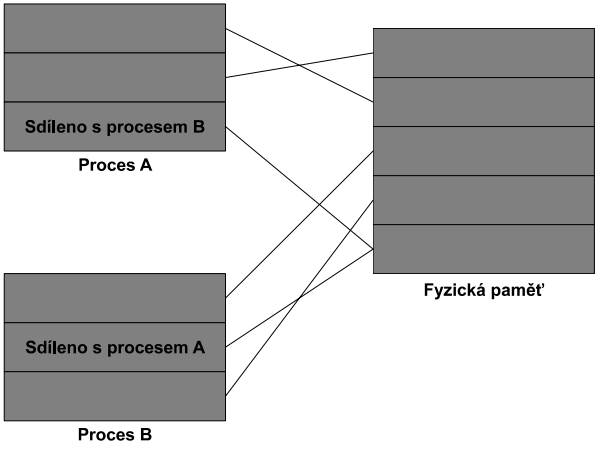
- **Ochrana –** některé procesory umožňují pro každou stránku určit, jaké operace s ní lze provádět. Tímto způsobem operační systém například chrání některé oblasti fyzické paměti proti zápisu. U modernějších procesorů je též možné zakázat na určitých stránkách spouštění kódu. Navíc ne každý blok fyzické paměti musí být viditelný přes nějakou virtuální stránku. Tím lze zamezit přístupu do oblastí, ve kterých jsou uložena citlivá data.
- **Iluze souvislosti** – souvislý blok virtuálních adres je možné namapovat do několika nesouvislých bloků fyzické paměti. Situaci ukazuje obrázek 1.1. Souvislý blok virtuální paměti je namapován do dvou od sebe oddělených souvislých bloků fyzické paměti.
- **Privátní paměť** – každá aplikace může mít rezervován svůj rozsah virtuálních adres, který ostatní aplikace neuvidí. Například 32bitové verze Windows vytvoří každému programu paměťový prostor (též *virtuální adresový prostor* či jen *adresový prostor*) o velikosti 2 GB (od adresy 0x00000000 do adresy 0x7FFFFFFF). Protože jádro může měnit mapování mezi virtuálními a fyzickými adresami za běhu, nevadí, když více aplikací pracuje se stejnými hodnotami virtuálních adres – když aplikace dostane čas na procesoru, operační systém změní mapování tak, aby se na virtuálních adresách objevovala právě její data.
- **Iluze dostatku paměti** – ne každá virtuální adresa musí být namapována na nějakou fyzickou. Operační systém může toto mapování vytvořit až v případě potřeby, která nastává ve chvíli, kdy se z dané adresy pokusí někdo číst, nebo na ni zapisovat. Na 32bitových verzích Windows má tedy aplikace k dispozici paměťový prostor o velikosti 2 GB, jehož valná většina není zpočátku namapována do fyzické paměti. Program může „žít" v iluzi, že má k dispozici celé 2 GB operační paměti, ačkoliv se na počítači nachází třeba jen 512 MB fyzické paměti RAM. Úkolem operačního systému je tuto iluzi co nejvíce podporovat.
- **Sdílení paměti** – na jednu fyzickou adresu lze namapovat více virtuálních adres, každá z nich může povolovat pouze určitý druh přístupů (čtení, zápis, vykonávání kódu). Tato možnost dovoluje uchovávat v paměti pouze jednu kopii dat, která ale může být viditelná ve virtuálním adresovém prostoru několika aplikací, dokonce se může objevit vícekrát v adresovém prostoru jedné aplikace. Sdílená paměť nachází využití zejména při výměně dat mezi adresovými prostory.

Ukázku sdílené paměti vidíte na obrázku 1.2. Adresový prostor procesů A a B sestává ze tří bloků virtuální paměti. Dva z nich jsou do fyzické paměti mapovány tak, aby se nepřekrývaly, takže jsou viditelné pouze z daného adresového prostoru. Stránky třetího bloku jsou však v obou adresových prostorech překládány na stejné fyzické adresy. Oba procesy tak vidí obsah stejné ob-

lasti fyzické paměti a při vhodném nastavení oprávnění stránek může například proces A pozorovat změnu obsahu, kterou proces B způsobí tím, že do sdíleného bloku zapíše nějaká data.



Obrázek 1.1: Vytvoření iluze souvislého bloku paměti



Obrázek 1.2: Sdílení paměti

Více informací o virtuální paměti najdete v kapitole 9.

Procesy, vlákna, joby

Nyní již víte, že každá aplikace má svůj vlastní kousek virtuální paměti – virtuální adresový prostor –, kde si může „žít" a díky mechanismu systémových volání komunikovat s okolním světem (například číst stav klávesnice). Pro správné fungování takové aplikace musí operační systém udělat ale mnohem víc, než vytvořit nový virtuální adresový prostor. A aby OS mohl s aplikacemi jako celky lépe pracovat, byly zavedeny pojmy *proces*, *vlákno* a *job*.

Každá aplikace je reprezentována jednou entitou, která se nazývá *proces*. Proces si můžete představit jako kontejner, který sdružuje všechny potřebné informace. Například:

- Virtuální adresový prostor včetně seznamu rezervovaných a alokovaných bloků virtuální paměti.
- Jedinečný identifikátor procesu. Ve Windows se jedná o číslo PID (*process identifier*). V každém okamžiku mají všechny běžící procesy v systému různá čísla PID.

- Seznam oprávnění, kterými aplikace disponuje. Tato informace je sdružena do struktury (objektu) *tokenu*.
- Entity zodpovědné za vykonávání kódu procesu. Nazývají se *vlákna*.
- Seznam objektů, se kterými proces právě pracuje. Jedná se například o otevřené soubory, klíče registru či bloky sdílené paměti. Tento seznam neobsahuje přímé odkazy na jednotlivé objekty (jejich adresy v paměti jádra), ale tzv. *handle*. Handle k jednotlivým objektům vydává součást jádra, která se nazývá *správce objektů*. Tento mechanismus nepřímých odkazů dovoluje implementovat například zabezpečení a šetří paměť. Více se o něm dočtete v kapitole 6.

Vlastní vykonávání kódu tedy není úkolem procesu, ale jeho vláken. Součást jádra označovaná *plánovač úloh* zajišťuje, aby se jednotlivá vlákna na procesoru střídala, a tím vznikal dojem současného běhu více aplikací, než má počítač k dispozici procesorů.

Od Windows 95 probíhá střídání vláken na procesoru *preemptivně* – vlákna proces střídání na procesoru sice ovlivnit mohou, ale nemají nad ním plnou kontrolu. Plánovač každému z nich přidělí na krátký čas procesor a po jeho uplynutí jej přidělí dalšímu v pořadí. Tento proces se také označuje jako *plánování* a probíhá transparentně vzhledem k vláknům. Každé si může „myslet", že procesor patří jenom jemu a není přidělován jiným entitám.

To je velký rozdíl oproti konceptu *kooperativního plánování*, který využívaly systémy Windows starší než Windows 95. V tomto modelu je procesor vláknu odebrán pouze na jeho explicitní žádost. Pokud se tedy nějaká aplikace dostane do nekonečné smyčky, může způsobit zatuhnutí celého systému, protože její vlákno nikdy nepožádá o odejmutí procesoru.

Stejně jako proces, i vlákno patří mezi velmi složité struktury a zahrnuje v sobě mnoho údajů, mezi které například patří:

- Hodnoty registrů procesoru od okamžiku posledního přeplánování. Ty plánovač načte do registrů, jakmile vláknu přidělí další časový úsek (*time slice*). Tato struktura nese název *CONTEXT* a její obsah je závislý na typu procesoru.
- Jedinečný identifikátor vlákna – TID (*thread identifier*).
- Oprávnění reprezentovaná opět objektem tokenu. Většina vláken touto strukturou nedisponuje, a tak seznam oprávnění dědí z tokenu jejich procesu. Windows tak umožňuje, aby různá vlákna jednoho procesu disponovala různými sadami oprávnění.
- Prioritu, která udává, jak často by měl plánovač pouštět dané vlákno na procesor.

Krom jednotlivých procesů a jejich vláken lze ve Windows pracovat i se skupinami několika procesů jako s nedělitelnými entitami. Tato vlastnost nachází užitek, pokud několik procesů spolupracuje na splnění jedné úlohy. Každou takovou skupinu procesů popisuje jeden *objekt job* a díky němu lze celé skupině nastavit například limity na využití systémových prostředků.

Knihovny DLL a rozhraní Windows API

Kód programů je uložen v tzv. *spustitelných* souborech na pevném disku či jiném médiu. Aplikace nativně psané pro Windows jsou uloženy v souborech ve formátu PE (Portable Executable). Tyto soubory operační systém při vytváření nového procesu načte (namapuje) do nového virtuálního adresového prostoru. Jakmile má nový proces svůj kód v paměti, může jej začít vykonávat.

Soubory formátu PE lze rozdělit na několik druhů:

- **Soubory EXE (přípona .exe)** – obsahují vlastní kód procesu. Při vytváření procesu se mapují jako první do jeho paměťového prostoru.
- **Knihovny DLL (přípona .dll)** – poskytují aplikacím různé užitečné funkce, například pro práci s prvky grafického uživatelského rozhraní či pro komunikaci po síti. Aplikace mohou příslušnou knihovnu DLL namapovat do svého paměťového prostoru a pak volat rutiny, které knihovna poskytuje. Jakmile program již jejích služeb nepotřebuje dále využívat, může příslušnou knihovnu DLL uvolnit z paměti. Knihovny lze tedy do paměti načítat dynamicky – odtud také pochází zkratka DLL – *dynamic link library*.
- **Soubory SYS (přípona .sys)** – obsahují kód ovladačů jádra.

Poznámka: O typu spustitelného souboru nerozhoduje jeho přípona, ale vnitřní formát. Je například možné vytvořit proces, jehož kód obsahuje soubor `ukazka.dll` či ovladač s příponou `.dat`. Například aplikace Adobe Reader v minulosti realizovala své pluginy pomocí knihoven DLL, jejichž jména ale měla příponu `.api`. Přípony uvedené v předchozích odrážkách nemusí být striktně dodržovány, jedná se pouze o nepsanou konvenci.

Spuštění nové aplikace probíhá tak, že operační systém vytvoří nový proces, do jehož adresového prostoru namapuje soubor EXE. Následně se systém podívá, jaké knihovny DLL aplikace ke svému běhu potřebuje a také je namapuje. Během mapování nového souboru formátu PE se vždy zkontroluje, jestli jsou v paměti již načteny všechny knihovny DLL potřebné pro jeho běh, a pokud tomu tak není, systém se je pokusí najít a načíst. Jakmile jsou všechny potřebné soubory načteny v paměti nového procesu, systém spustí nové vlákno, které začne vykonávat kód obsažený v souboru EXE.

Knihovny DLL dávají svému okolí k dispozici (*exportují*) mnoho funkcí. Mezi nejdůležitější knihovny patří `kernel32.dll`, `user32.dll` a `gdi32.dll`. `Kernel32.dll` exportuje rutiny pro práci s procesy, vlákny, soubory, registry a mnoho dalších funkcí pro komunikaci s jádrem operačního systému. `User32.dll` sdružuje rutiny obsluhující prvky grafického uživatelského rozhraní, jako okna, tlačítka, menu a textová pole. `Gdi32.dll` obsahuje funkce pro práci s ikonami a pro kreslení. Téměř všechny rutiny poskytované těmito knihovnami jsou plně zdokumentovány. Protože se málokterá aplikace objede alespoň bez dvou z výše jmenovaných knihoven, tyto funkce se souhrnně označují jako *rozhraní Windows API*.

Další důležitou knihovnou DLL je soubor `ntdll.dll`, jehož úkolem je implementovat tu část mechanismu systémových volání, která musí být umístěna v uživatelském režimu. Knihovna plní úlohu jakéhosi překladatele. Když jiné knihovny (například `kernel32.dll`) potřebují zavolat jádro, obrátí se se svým požadavkem na `ntdll.dll`, která jej přeloží do řeči, jíž jádro rozumí. Ostatní knihovny DLL tedy mohou fungovat nezávisle na konkrétní implementaci mechanismu systémových volání.

Knihovna `ntdll.dll` exportuje mnoho funkcí, jež jsou svoji povahou nízkoúrovňové a jen pár z nich je oficiálně zdokumentováno. Ujal se pro ně název *nativní API funkce*, který bude používán i v této knize.

Služby a ovladače

Windows umožňuje vytvářet speciální druh aplikací, jejichž primárním úkolem je vykonávat důležité úkoly na pozadí, například defragmentaci souborového systému či zálohování. Tyto entity se označují jako *služby* (na unixových systémech *demoni*) a krom absence jakéhokoliv grafického uživatelského rozhraní se od běžných aplikací liší také tím, že jejich běh obvykle neskončí při odhlášení uživatele.

Poznámka: Teoreticky i služby mohou pracovat s prvky grafického uživatelského rozhraní, ale takové chování se silně nedoporučuje. Protože plní nejrůznější systémové úkoly, služby často disponují vysokým oprávněním, dokonce vyšším než administrátor. Absence uživatelského rozhraní činí takový program méně zranitelný proti útokům jiných aplikací.

Služby spravuje součást systému s názvem *správce služeb* (*Service Control Manager* – SCM).

Ovladače jsou jsou spustitelné soubory formátu PE, jejichž kód je určen k vykonávání v režimu jádra. Mohou provádět úkoly, na které normální aplikace běžící v uživatelském režimu procesoru nemají oprávnění. Jedná se například o přímou komunikaci s periferními zařízeními počítače.

Windows interně zacházejí s ovladači jako se speciálním typem služeb. O spouštění a instalaci ovladačů se opět stará správce služeb.

Historie Windows

Historie operačních systémů Windows se začala psát v polovině osmdesátých let, ale některé události, jež vývoj ovlivnily, mají kořeny už v letech sedmdesátých. Cílem této části není přesně popsat, jaké novinky jednotlivé verze přinesly, ale poukázat jen na ty nejdůležitější a demonstrovat, proč dnešní Windows vypadají tak, jak vypadají.

Windows 1

První verze Windows se objevila 20. listopadu 1985. Nejednalo se o plnohodnotný operační systém, ale spíše jen o grafickou nadstavbu systémuMS-DOS, bez kterého Windows 1.0 nefungovaly.

Již tato verze Windows podporovala souběžné vykonávání kódu více aplikací najednou, využíval se kooperativní způsob plánování. Systém obsahoval ovladače videokarty, klávesnice, myši či tiskárny. Aplikace tedy nemusely přímo přistupovat k hardware, jak se stalo dobrým zvykem v prostředí čistého MS-DOS, ale mohly využívat rozhraní, která ovladače poskytovaly. Tato nová rozhraní se rozšířila i do dalších oblastí, jako je správa paměti či práce se soubory. Tím byl položen základní kámen pozdějšího standardního rozhraní Windows API.

Protože se grafické uživatelské rozhraní velmi podobalo rozhraní použitému v operačních systémech od společnosti Apple, Microsoft s touto firmou uzavřel dohodu a implementoval určitá omezení, aby se vyhnul případným soudním sporům. Okna se například nemohla překrývat.

O první verzi Windows trh nejevil příliš velký zájem. Jedním z důvodů může být fakt, že v té době neexistovala větší aplikace, která by striktně vyžadovala grafické uživatelské rozhraní.

Windows 2

Windows 2 vyšly 8. prosince 1987. Byla to první verze, která obsahovala aplikace jako Word, Excel, Kalkulačka či Poznámkový blok. Velkých vylepšení se dočkalo grafické uživatelské rozhraní – okna se mohla překrývat a objevily se pojmy jako maximalizace a minimalizace.

Kvůli vylepšením grafického uživatelského rozhraní došlo k soudnímu sporu mezi Apple a Microsoftem. Společnost Apple se svou žalobou neuspěla, což částečně způsobila předchozí dohoda o grafickém uživatelském rozhraní Windows 1.

V květnu roku 1988 byly vydány dvě verze Windows 2, které využívaly nových vlastností procesorů Intel 8286 a 8386. Byly označovány jako Windows/286 2.1 a Windows/386 2.1. Nové vlastnosti procesorů se projevily zejména na Windows/386. Největší přínos Windows/286 spočíval v tom, že dovolovaly aplikacím využívat více paměti než dříve.

Jádro Windows/386 již běželo v *chráněném režimu* procesoru. Tento speciální režim například umožňuje implementovat ochranu důležitých paměťových struktur před nežádoucí modifikací a zabránit přímé komunikaci běžných programů s hardware. Předchozí verze Windows běžely čistě v *reálném režimu* procesoru, kde neexistují žádné mechanismy, pomocí kterých by bylo možné zajistit ochranu paměti či zamezit neautorizované komunikaci s periferními zařízeními. Kód normálních aplikací byl vykonáván ve speciálním režimu procesoru zvaném *virtuální režim 8086*, který emuloval prostředí reálného režimu.

Windows 3 a OS/2

Třetí verze Windows byla vydána 22. května 1990 a brzy zaznamenala velký úspěch. V krátké době se jí prodalo několik milionů kopií. Prodeje byly vysoké mimo jiné díky tomu, že se systém předinstalovával do nových počítačů. Windows 3 je zároveň poslední verze systému, u které byla garantována stoprocentní zpětná kompatibilita s aplikacemi určenými pro prostředí MS-DOS.

Již od druhé poloviny 80. let (tedy souběžně s vývojem prvních verzí Windows) spolupracovali Microsoft a IBM na vývoji nového operačního systému, jenž se měl stát nástupcem MS-DOS (Disk Operating System). Ani Windows 3 totiž bez podpory DOSu nefungovaly. Tento nový operační systém byl označen OS/2 a plně využíval možností chráněného režimu procesorů Intel 8286. OS/2 již podporoval preemptivní plánování a *swapování* (ukládání nepoužívaných oblastí paměti na disk, čímž se uvolnila fyzická paměť pro další použití).

Spolupráce obou firem se začala chýlit ke svému konci po vydání a obrovském úspěchu Windows 3. Obě společnosti se dohodly na tom, že IBM bude pokračovat ve vývoji aktuální verze OS/2 (OS/2 2.0), která se měla stát nástupcem Windows 3, a Microsoft začne pracovat na OS/2 3.0, který později nahradí verzi vyvíjenou IBM.

Záhy i tato dohoda skončila. V IBM dále pracovali na vývoji druhé verze OS/2, zatímco Microsoft budoucí OS/2 3.0 přejmenoval na Windows NT (New Technology) a jeho kód od základů přepsal.

Windows NT

Ačkoliv jsou Windows NT v mnoha ohledech daleko pokročilejší než jejich předchůdci, nejedná se o úplně novou technologii, jak by se z názvu mohlo zdát. Hlavním návrhářem tohoto systému se stal Dave Cutler, který dříve pracoval u společnosti Digital Equipment Corporation (později odkoupené firmou Compaq, která je dnes součástí společnosti Hewlet Packard) například na vývoji operačního systému VMS. Cutler z VMS převzal mnoho mechanismů a postupů.

Podobnost obou systémů dosáhla takové úrovně, že si jí všimli zaměstnanci společnosti DEC a Microsoftu hrozila další žaloba. Soudní spor nakonec neproběhl, obě firmy se dohodly mimosoudně a Microsoft mimo jiné dodal druhé firmě prostředky k tomu, aby Windows NT mohly běžet i na procesoru Alpha, jehož výrobcem byl právě Digital Equipment Corpration.

Procesor Alpha v první polovině 90. let patřil k nejrychlejším. Výkon byl ale vykoupen vysokou cenou. Postupem času jej začaly svým výkonem dohánět levnější procesory od Intelu. Výkonový rozdíl klesl natolik, že se nevyplatilo drahý procesor kupovat, a tak pomalu skončila i podpora Windows NT.

Windows NT přinesly mnoho inovací. Jednalo se o plně 32bitový operační systém s podporou virtuální paměti a preemptivního plánování, který však stále umožňoval spustit i 16bitové programy pro MS-DOS. Mezi další vylepšení patřil nový souborový systém NTFS a implementace vlastního bezpečnostního modelu, který dovoloval uživatelům nastavovat různá oprávnění, a tím povolit či zakázat provádění specifických operací.

Windows NT se postupně dočkaly čtyř verzí: Windows NT 3.1 (27. červenec 1993), Windows NT 3.5 (21. září 1994), Windows NT 3.51 (30. května 1995) a Windows NT 4.0 (24. srpna 1994).

Windows 95, Windows 98 a Windows Me

Než došlo ke sloučení vývojových linií Windows 3 a Windows NT, byly vydány ještě tři následníci Windows 3 a jeden následník Windows NT.

Windows 3 postupně následovaly Windows 95 (24. srpen 1995), Windows 98 (25. červen 1998) a Windows Me (14. září 2000). Tyto systémy měly některé rysy společné s Windows NT (chráněný režim procesoru, podpora běhu 32bitových aplikací, virtuální paměť, swapování), ale v mnohém se též lišily.

Ačkoliv podporovaly běh 32bitových aplikací, část kódu jádra zůstávala od dob Windows 3 stále 16bitová. Nebyl implementován žádný bezpečnostní model, díky kterému by například bylo možné ochránit soubory jednoho uživatele před neautorizovaným přístupem jiného. Windows 95/98/Me podporovaly pouze souborové systémy FAT12, FAT16 a FAT32. Úroveň izolace jednotlivých aplikací také nebyla tak vysoká jako u Windows NT – všechny běžící programy používaly společné kopie systémových knihoven DLL. Běžná aplikace dokonce mohla číst paměť jádra.

16bitový kód jádra a menší míra izolace jednotlivých běžících procesů se staly jednou z příčin nižší stability těchto operačních systémů.

Windows 2000

17. února 2000 vyšly další Windows založené na NT technologii. Původně měly Windows 2000 sjednotit obě vývojové větve (Windows 95/98/Me a Windows NT), což se kvůli obtížnosti takového projektu podařilo až u Windows XP. Windows 2000 už však převzaly mnoho prvků z druhé vývojové linie, například aplikace Internet Explorer 5 a Windows Media Player a podporu souborového systému FAT32. Přinesly také následující novinky:

- **Šifrovaný souborový systém (Encrypted File System – EFS)** – nová verze NTFS (3.0) krom symbolických odkazů přinesla i podporu šifrování. Automatické šifrování a dešifrování obsahu souborů zajišťuje ovladač souborového systému, tudíž pro normální aplikace probíhá vše transparentně. Data zapisovaná na disk jsou automaticky šifrována a při čtení opět dešifrována.

- **Obecné ovladače pro USB zařízení** – od Windows 2000 není nutné instalovat speciální ovladače pro většinu zařízení s rozhraním USB, jako jsou flash disky či externí pevné disky.
- **Ochrana systémových souborů (Windows File Protection) a Kontrola integrity (System File Checker)** – operační systém v reálném čase kontroluje, zda nebyl poškozen či smazán některý z důležitých souborů, a pokud se tak stane, pokusí se provést jeho opravu. Oprava se provádí ze zálohy v podadresáři `dllcache` systémového adresáře (standardně `C:\Winnt\system32`), případně z instalačního média. Uživatel může o kontrolu integrity systému požádat i explicitně.
- **Podpora pro analýzu příčin selhání systému** – již dřívější verze Windows při zjištění kritické chyby zobrazily neslavně proslulou *modrou obrazovku smrti* (Blue Screen Of Death – BSOD) a ukončily svůj běh. Windows 2000 přináší možnost při detekci kritické chyby počítač automaticky restartovat, což je užitečné zejména pro servery, které by měly běžet pokud možno nepřetržitě. Navíc se při detekci kritické chyby na disk vypíše část obsahu fyzické paměti (rozsah výpisu záleží na konkrétním nastavení). Z obsahu paměti v době havárie je často možné zjistit, proč k selhání došlo.

Windows XP

Windows XP, které se objevily na trhu 25. října 2001, konečně spojily obě vývojové linie, ačkoliv mnoho prvků z linie Windows 9x bylo zakomponováno již do Windows 2000. Jádro bylo převzato z Windows 2000 a vylepšeno.

Nové implementace se dočkal mechanismus systémových volání. Do Windows 2000 se pro přechod do režimu jádra využívalo softwarového přerušení číslo 0x2e. Windows XP pro změnu Ring módu používají nové instrukce SYSENTER a SYSEXIT, které byly navrženy právě pro tyto účely. Starý mechanismus ale z důvodů zpětné kompatibility stále funguje.

Vývojáři se také zaměřili na rychlost startu systému (bootování) a kladli si za cíl snížit tento čas pod hranici třiceti sekund. Proto implementovali například technologii Prefetch, která monitoruje, jaké soubory se při startu systému používají a optimalizuje jejich umístění na disku tak, aby jejich načtení trvalo co nejkratší dobu.

Další novinkou je možnost lokálního přihlášení více uživatelů na jeden počítač. V dřívějších verzích mohl být k počítači přihlášen pouze jeden uživatel. Technologie *rychlého přepínání uživatelů* (*fast user switching*) umožňuje přepínat mezi pracovním prostředím jednotlivých přihlášených uživatelů, aniž by se museli odhlašovat.

Windows Vista

Na následníka Windows XP si museli zákazníci počkat až do 30. listopadu 2006. Vývoj operačního systému provázely velké problémy a od některých slibovaných funkcí nakonec vývojáři upustili. Tento osud postihl například nový souborový systém WinFS, který měl tvořit databázovou nadstavbu nad NTFS. I přes tyto komplikace Windows Vista přinesly řadu novinek nejen z oblasti bezpečnosti.

Díky mechanismu *Kontroly uživatelských účtů* (User Account Control – UAC) uživatelé již nemusí disponovat administrátorskými právy po celou dobu své práce. Pokud nějaký program pro svůj běh vyžaduje vyšší oprávnění, systém se zeptá, zda mají být tato oprávnění poskytnuta.

I na dřívějších verzích Windows bylo samozřejmě možné pracovat pod uživatelským účtem s omezenými právy, nebylo však možné některému programu oprávnění (způsobem dostatečně

jednoduchým pro uživatele) zvýšit. Navíc mnohé programy nebyly vytvořeny tak, aby se s omezeným oprávněním vypořádaly. Jedná se pravděpodobně o dědictví z vývojové linie Windows 9x, kde v podstatě existoval jen jeden druh uživatele – administrátor, jenž měl oprávnění k libovolné činnosti. A tak si většina uživatelů zvykla pracovat pod administrátorským účtem, ač jeho výhod nepotřebovala.

Práce pod účtem s omezenými právy přináší zvýšenou bezpečnost. Škodlivé programy, které se do systému dostanou například kvůli zranitelnosti internetového prohlížeče, se nemohou nakopírovat do systémových složek či zapsat data do důležitých oblastí registru, protože k tomu nemají oprávnění. Proto je práce pod účtem s omezenými právy velmi výhodná a s použitím programů, které si rozumí s mechanismem UAC, i prakticky možná.

Další změny se týkají ochrany samotného jádra a kvůli zpětné kompatibilitě jsou uplatňovány převážně na 64bitových verzích operačního systému.

Škodlivé modifikace důležitých struktur jádra monitoruje komponenta PatchGuard (Windows Kernel Patch Protection). PatchGuard sestává z mnoha kontrolních rutin, které se spouští při různých událostech a kontrolují, zda datové struktury a kód jádra nebyly modifikovány nepovoleným způsobem. Pokud je taková modifikace nalezena, PatchGuard uměle vyvolá modrou obrazovku smrti a počítač je třeba restartovat.

Kontrolní rutiny se vykonávají při různých událostech, tudíž není možné ochranu jádra za běhu systému snadno vypnout. PatchGuard ale často zjistí škodlivou modifikaci až za určitou dobu po jejím vzniku. Tato doba se pohybuje obvykle v řádu minut. Škodlivý kód tedy na nějakou dobu může jádro modifikovat, ale musel by předvídat, kdy se spustí další kontrolní rutina, a příslušně se podle toho zachovat. A zjistit, kdy dojde ke spuštění kontrolní rutiny, je obtížné.

Kromě technologie PatchGuard zavádějí 64bitové Windows Vista další stupeň ochrany – při standardním nastavení mohou být do jádra načteny pouze ovladače digitálně podepsané jednou z důvěryhodných certifikačních autorit. Tím se zvyšuje pravděpodobnost, že do jádra bude vpuštěn pouze ověřený (a tedy legitimní) kód.

Na 32bitových verzích Windows Vista digitální podpis u ovladačů není nutnou podmínkou pro načtení do jádra. Při detekci přítomnosti nepodepsaného ovladače systém pouze zobrazí varovný dialog.

Další vylepšení Windows Vista se týká zvýšení odolnosti proti útokům typu přetečení zásobníku či přetečení haldy (stack overflow, heap overflow). Pro úspěšné použití těchto technik útočník ve většině případů potřebuje vědět, na jakých adresách se nachází paměťové struktury jako halda či zásobník, nebo znát umístění některých systémových knihoven DLL. U předchozích verzí Windows byly tyto hodnoty pevně stanoveny. Windows Vista přichází s jejich randomizací – knihovny DLL se do paměti načítají na náhodné virtuální adresy a adresy hald a zásobníků též nejsou voleny pevně. Útočník tak dopředu neví, kde v paměti se nachází datové struktury a knihovny, které pro úspěšné provedení útoku potřebuje. Tato technika randomizace adres se označuje zkratkou ASLR (Address Space Layout Randomization).

Windows 7

Windows 7 se na trh dostaly 21. října 2009. Vlastnímu uvedení předcházelo několik měsíců veřejného testování, čímž si ještě nedokončený operační systém získal velkou popularitu u běžných uživatelů. Co se týče samotného jádra, krom vyššího stupně modularity a výkonnostních

optimalizací nepřináší Windows 7 žádné převratné novinky. Změny, které stojí za zmínku, naleznete v následujícím seznamu:

- **Méně zamykání –** kód operačního systému je paralelně vykonáván mnoha vlákny. Jak se dočtete v kapitole 4, běh vláken je někdy nutné synchronizovat, aby nedošlo k porušení konzistence dat, která mezi sebou sdílejí. Například situace, kdy vlákno čte oblast paměti, do níž ve stejném okamžiku jiné vlákno zapisuje, by neměla nikdy nastat, protože první vlákno pravděpodobně přečte blok, obsahující data zčásti původní a zčásti nově zapsaná – tedy pravděpodobně nesmyslná. Z toho plyne, že k některým datům musí v každém časovém okamžiku přistupovat pouze omezený počet vláken. Toto omezení platí zejména při změnách dat, které může obvykle v každém okamžiku provádět nejvýše jedno vlákno.

  K vynucení této podmínky lze využít například princip *zamykání*. Oblast se sdílenými daty je opatřena *zámkem*, který je na počátku odemčený. Pokud chce nějaké vlákno s takto hlídanými daty pracovat, musí jej zamknout. Zamknout je možné pouze odemčený zámek. Pokus o zamčení zámku, který jíž zamkl někdo jiný, končí pozastavením běhu snažícího se vlákna. Jakmile nad daty provede požadovanou operaci, vlákno zámek opět odemkne, čímž případně probudí některé z vláken, která se pokusila zamknout již zamčený zámek. Probuzené vlákno zámek opět zamkne a může s daty pracovat.

  Zamykání samozřejmě omezuje míru paralelního běhu vláken v operačním systému, což může vést až k znatelné ztrátě výkonu. Proto je důležité používat zámky jen tam, kde to situace opravdu vyžaduje a zamykat na co nejkratší dobu, aby nedocházelo k zbytečnému blokování vláken. A jádro Windows 7 bylo optimalizováno i v tomto směru, což by se mělo projevit na vyšší rychlosti celého operačního systému.

- **Podpora strojů až s 256 procesory –** Windows si pro každé vlákno pamatují, na jakých procesorech může běžet. – tzv. *afinitu.* Tento údaj je uložen jako 32bitové (na 64bitových verzích operačního sytému 64bitové) celé číslo, jehož každý bit reprezentuje jeden procesor. Pokud je bit nastaven na jedničku, vlákno může být plánováno na příslušném procesoru. Tato reprezentace afinity omezuje maximální počet procesorů na 32 (resp. 64). Windows 7 zavádí tzv. *skupiny procesorů*. Afinita vlákna potom neobsahuje informace o jednotlivých procesorech, ale o skupinách procesorů. Protože maximální velikost skupiny procesorů je čtyři (osm na 32bitových verzích), Windows 7 mohou pro svůj běh využívat až 256 procesorů.

- **Slučování časovačů –** operační systém umožňuje aplikacím provádět některé akce periodicky. Proces může například zapisovat do souboru každých 15 milisekund. Jádro pro tyto účely poskytuje objekty zvané *časovače* (*timer*). Každý takový požadavek na periodické vykonávání určitého kódu je reprezentován jedním časovačem. Od Windows 7 dochází k tzv. *slučování* časovačů – pokud například jedna aplikace vykonává určitou akci s periodou 8 milisekund a druhá s periodou 16 milisekund, je zbytečné vytvářet dva časovače, protože obě akce může zajistit časovač s periodou 8 milisekund. Díky tomuto opatření se šetří paměť jádra a zatěžuje se méně procesor.

- **Snížení četnosti dotazů od UAC –** Microsoft vyslyšel uživatele Windows Vista, kteří si stěžovali na příliš časté dotazy od UAC a možnosti této komponenty byly ve Windows 7 rozšířeny. V základním nastavení se UAC nyní neptá při spouštění digitálně podepsaných programů od společnosti Microsoft. Tyto dotazy patřily ve Windows Vista k těm nejčastějším.

- **Optimalizace startovacího procesu** – proces zavádění operačního systému byl upraven tak, aby maximálně využíval možností paralelního běhu několika vláken na vícejádrových procesorech, které se dnes nachází téměř v každém osobním počítači. Tato optimalizace by se měla pozitivně projevit na době bootování.

Serverové verze

Operační systémy rodiny Windows NT byly původně koncipovány především pro nasazení na serverech či ve firmách, ne pro použití na běžných osobních počítačích. Ke změně došlo s příchodem Windows 2000 a Windows XP. Od té doby Microsoft vydává i speciální serverové verze, které se od systémů určených pro domácí použití liší především absencí některých aplikací a osekaným uživatelským rozhraním. Jádro serverových systémů se odlišuje jenom mírně. Tabulka 1.1 ukazuje párování mezi systémy určenými pro servery a systémy určenými pro domácí uživatele.

Tabulka 1.1: Serverové verze operačního systému Windows a jejich příslušnost k verzím pro domácí použití

Serverový systém	Systém pro domácí použití
Windows 2000 Server	Windows 2000
Windows Server 2003	Windows XP
Windows Server 2003 R2	Windows XP Service Pack 2
Windows Server 2008	Windows Vista
Windows Server 2008 R2	Windows 7

Základní datové struktury užívané v operačních systémech

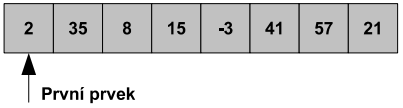
Operační systém si potřebuje pamatovat mnoho informací, například seznam běžících procesů, údaje o všech načtených ovladačích či seznam volných a alokovaných bloků fyzické paměti. S každým druhem informací je potřeba zacházet trochu odlišným způsobem. Například pokud nějaká součást jádra požádá o alokaci fyzické paměti, je žádoucí velmi rychle najít potřebné bloky (u fyzické paměti se též nazývají *rámce*, u virtuální paměti *stránky*) a přidělit je volajícímu. Přidělování virtuální paměti by mělo splňovat podobná kriteria. Na druhou stranu není v drtivé většině případů nutné rychle vědět, jaké procesy používají určitou knihovnu DLL.

Z předchozího odstavce plyne, že různé informace je třeba reprezentovat různým způsobem, aby s nimi mohl operační systém a aplikace pracovat co nejefektivněji. Následující odstavce popisují několik možností, jak reprezentovat soubor údajů. Tyto reprezentace se nazývají *datové struktury* a jednotlivé údaje v nich obsažené se často označují jako *prvky*.

Upozornění: Pro čtení dalších kapitol knihy není nutné do detailů pochopit, jak dále popisované datové struktury fungují. Ale měli byste získat rámcový přehled o jejich výhodách, nevýhodách a při jakých příležitostech a na jaká data jsou vhodné.

Pole

Pole je datová struktura, která slouží k uchovávání prvků stejného druhu. Všechny prvky jsou uloženy za sebou v souvislém bloku paměti. Ukázkový příklad vidíte na obrázku 1.3.



Obrázek 1.3: Konkrétní příklad datové struktury pole

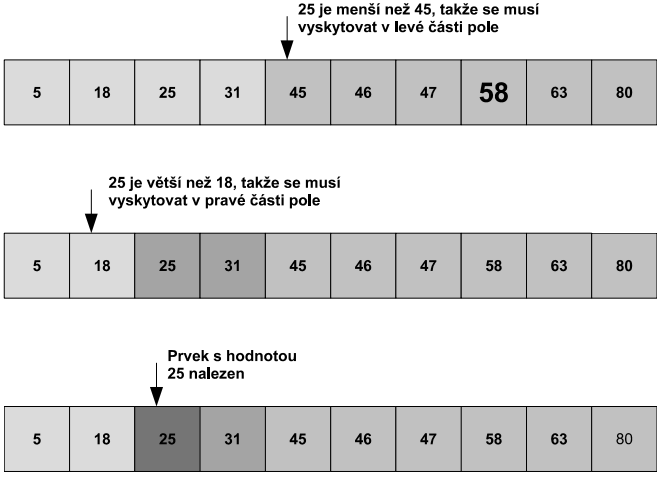
Výhodou pole je velmi rychlý přístup k N-tému prvku (tato vlastnost se též označuje jako *náhodný přístup*). Pokud známe adresu začátku pole, adresa N-tého prvku je určena vzorcem:

```
adresa_N_teho_prvku = adresa_pole + N * velikost_prvku
```

Číslu N se také říká *index*. Obsah prvku na určitém indexu tedy získáme v jediném kroku nezávisle na velikosti pole.

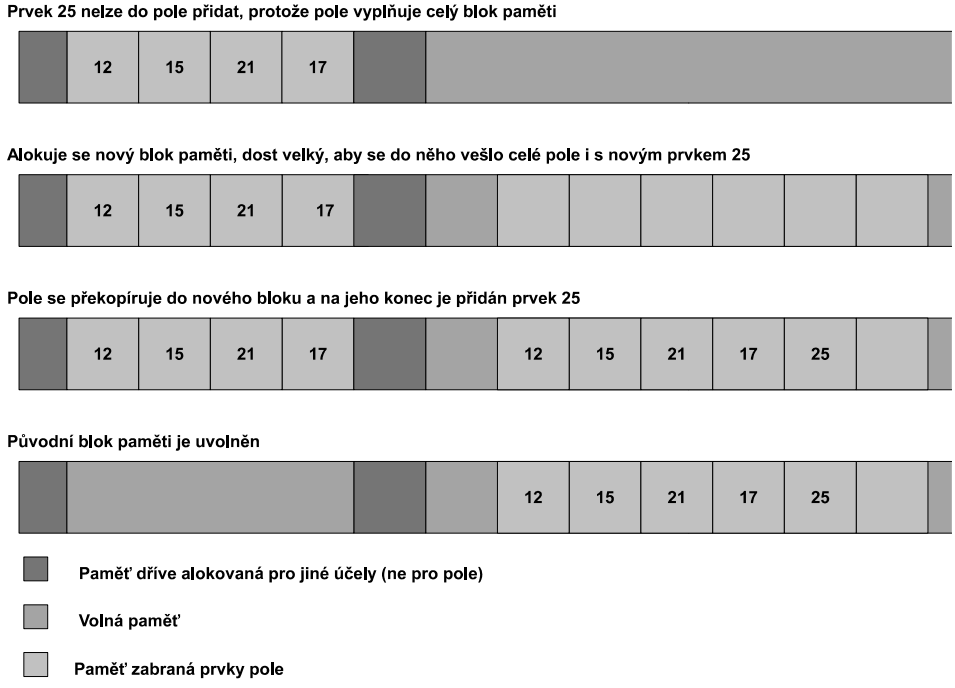
Pokud je pole navíc setříděné, je možné rychle provést test na existenci prvku s určitou hodnotou. Tato operace již závisí na velikosti pole. Je nutné provést nejvýše tolik operací, kolik je dvojkový logaritmus z počtu prvků ve struktuře.

Na obrázku 1.4 vidíte, jak se obvykle při testování existence prvku v setříděném poli postupuje. Tato metoda se nazývá *půlení intervalů*. Nejprve se otestuje hodnota prvku uprostřed pole. Pokud je hledaná hodnota menší, musí se nacházet před ním. Pokud je větší, musí, díky faktu, že pole je v tomto příkladu setříděné vzestupně, ležet za ním. A stejný postup se aplikuje na příslušnou polovinu pole – tedy provede se test, zda se její prostřední prvek nerovná hledané hodnotě. Po každém testování se interval, ve kterém se hledaný prvek může nacházet, zmenší na polovinu. Pokud zdegeneruje na interval tvořený jedním prvkem, který není shodný s hledanou hodnotou, hledaný prvek v poli není obsažen. Obrázek 1.4 konkrétně ukazuje, jak v setříděném poli, jehož prvky tvoří čísla, probíhá test na existenci hodnoty 25.



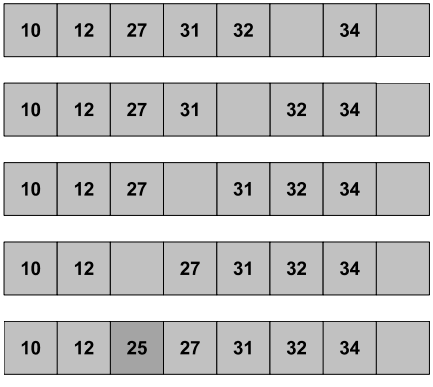
Obrázek 1.4: Ukázka průběhu algoritmu na vyhledávání hodnoty v setříděném poli

Tím jsou však výhody pole téměř vyčerpány. Velkým problémem může být fakt, že se nachází v jednom souvislém bloku paměti. Postupným přidáváním nových prvků může dojít k naplnění celého bloku. Potom je nutné alokovat větší souvislý blok a obsah celého pole do něho překopírovat. A kopírování velkých bloků paměti již není časově zanedbatelné, probíhá-li relativně často. Tento případ demonstruje obrázek 1.5.



Obrázek 1.5: Přidávání nového prvku s hodnotou 25 do pole, které již zaplnilo celý souvislý blok paměti

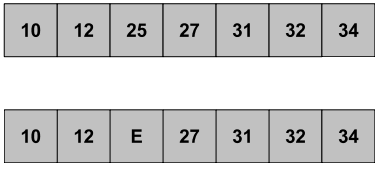
Navíc se může stát, že volná paměť je rozdrolena do malých kousků, které sice v součtu dávají dostatek místa pro větší blok, ale souvislý blok dostatečné velikosti neexistuje.



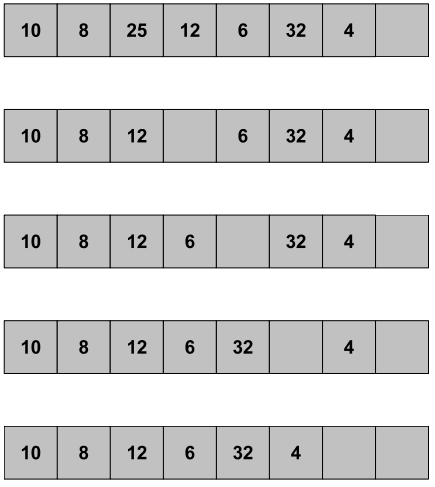
Obrázek 1.6: Přidávání doprostřed pole

Přidávání nového prvku může být ještě pomalejší, není-li přidáván na konec, ale někam doprostřed. Tento případ může nastat například u setříděných polí (viz obrázek 1.6). Pro nový prvek je třeba uvolnit místo a to je možné provést jen překopírováním prvků ležících za ním o jednu pozici dozadu. Na obrázku 1.6 vidíte tuto operaci schématicky znázorněnou.

Odebírání prvku je možné provést nezávisle na velikosti pole, pokud není nutné datovou strukturu udržovat celistvou a může obsahovat *díry* – prvky se speciální hodnotou, která reprezentuje volné místo. Pokud si není možné díry dovolit, je při odebrání prvku nutné posunout jiné prvky tak, aby byla vzniklá díra zacelena. Příklad takového mazání vidíte na obrázcích 1.7 a 1.8.



Obrázek 1.7: Mazání prvku z pole, které si může dovolit díry



Obrázek 1.8: Mazání prvku z pole, které si nemůže dovolit díry

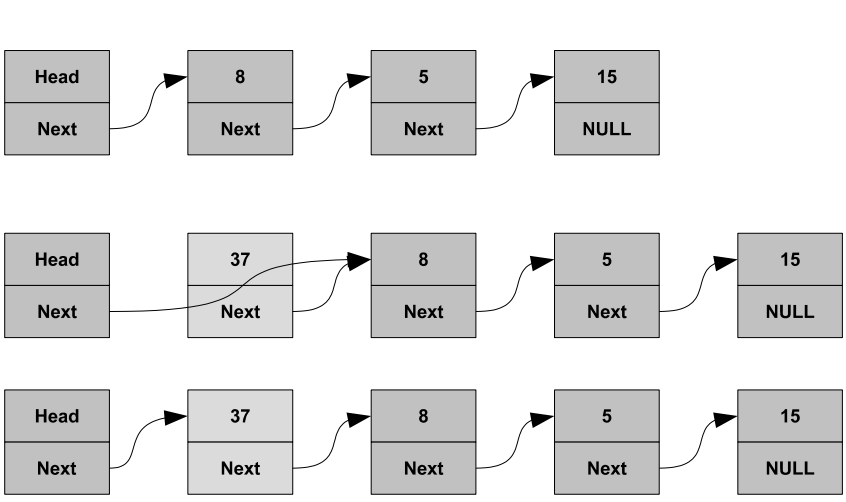
Pole se využívají především při implementaci složitějších datových struktur jako zásobník, fronta či hašovací tabulka. Hodí se na uchovávání dat, která se příliš často nemění – nedochází k častému přidávání nových a mazání starých prvků. Jak je totiž vidět z předchozích odstavců, přidávání či mazání prvků může způsobit i kopírování celého pole, což je operace pomalá, ačkoliv probíhá celá v operační paměti počítače.

Spojové seznamy

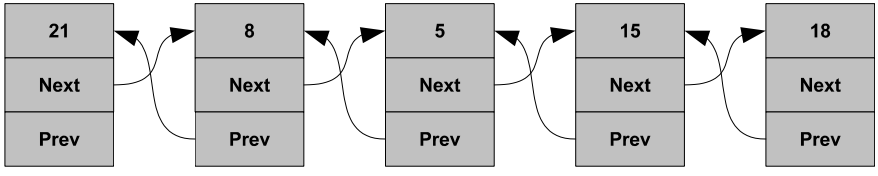
Spojový seznam ukládá každý svůj prvek v odděleném bloku paměti. Tím je eliminována potřeba velkých volných souvislých úseků paměťových stránek, pokud datová struktura obsahuje velké množství prvků. Každý paměťový blok, který obsahuje jeden prvek seznamu, v sobě zároveň uchovává další informace o poloze ostatních prvků. Podle množství těchto informací se spojové

seznamy rozdělují na několik druhů. Čím více informací navíc se uchovává, tím lépe lze s daným seznamem pracovat. Spojové seznamy můžeme rozdělit do následujících kategorií:

- **Lineární (jednosměrné)** – každý prvek obsahuje adresu následujícího prvku v seznamu (viz obrázek 1.9). Do lineárního spojového seznamu se snadno přidává na začátek a snadno ze začátku odebírá. Poslední prvek má adresu následujícího prvku seznamu vyplněnou hodnotou pro neplatný ukazatel (NULL, Nil). Pokud je známa adresa nějakého prvku, snadno lze za něho přidat další prvek.
- **Obousměrné** – každý prvek obsahuje odkaz (adresu) na svého následníka a předchůdce. I do obousměrného spojového seznamu se snadno přidává na začátek a snadno ze začátku odebírá. Dále je možné snadno přidat další prvek za nebo před jiný prvek se známou adresou. Přidávání a mazání, ačkoliv nezávisí na délce (počtu prvků) seznamu, je však pomalejší než u lineárního seznamu – kromě adresy následníka je nutné měnit i adresu předchůdce. Ukázku obousměrného seznamu vidíte na obrázku 1.10.



Obrázek 1.9: Jednosměrný spojový seznam



Obrázek 1.10: Obousměrný spojový seznam

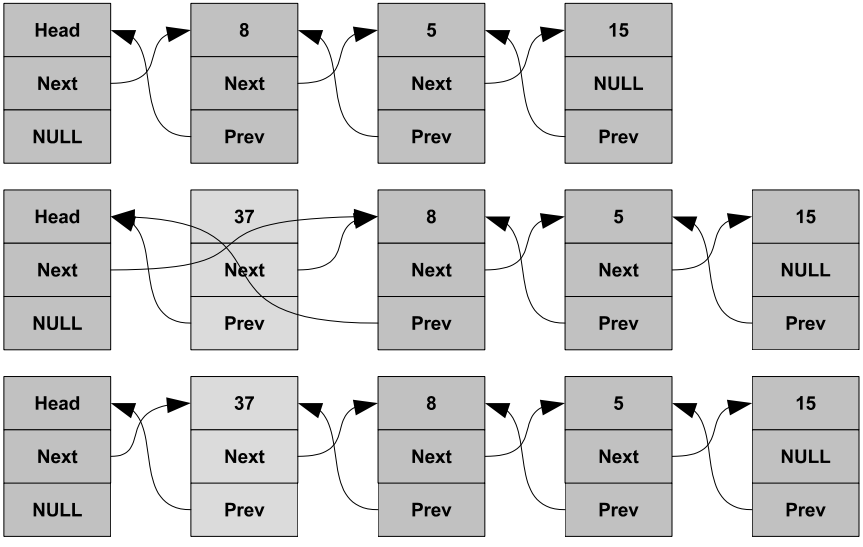
Jednosměrné i obousměrné seznamy je možné ještě obohatit následujícím způsobem:

- Ve většině případů si stačí pamatovat jenom adresu prvního prvku v seznamu. To je plně dostačující pro všechny operace, které je s touto datovou strukturou možné provádět. Pro prázdný seznam bude mít tato adresa hodnotu neplatného ukazatele (NULL, Nil). Z toho vyplývá, že obslužný kód musí v některých případech obsahovat speciální příkazy pro práci s prázdným seznamem a pro práci s neprázdným seznamem, což jej činí složitějším a ná-

chylnějším k chybám. Nutnosti separátního kódu pro obsluhu prázdného a neprázdného seznamu se lze vyhnout zavedením tzv. *hlavy*. Hlava je speciální prvek, který neobsahuje žádnou hodnotu, ale umožňuje obsluhovat spojový seznam stejným způsobem, ať už obsahuje nějaké skutečné prvky nebo ne. Stačí pouze nadefinovat, že prázdný seznam je seznam, který obsahuje pouze hlavu a že hlava bude vždy prvním prvkem. Operace přidávání a mazání do seznamu s hlavou jsou znázorněny na obrázku 1.11.

- Definujeme-li jako následníka posledního prvku seznamu první prvek, dostaneme spojový seznam, který se nazývá *cyklický*. Tato vlastnost má význam zejména u obousměrných seznamů, protože umožňuje dostat se z prvního prvku na prvek poslední přečtením pouze jediné adresy. Díky tomu je možné přidávat i na konec nezávisle na délce seznamu. Příklad cyklického obousměrného seznamu vidíte na obrázku 1.12.

Přidání prvního prvku (provedení operací v opačném pořadí zajistí odebrání prvního prvku)



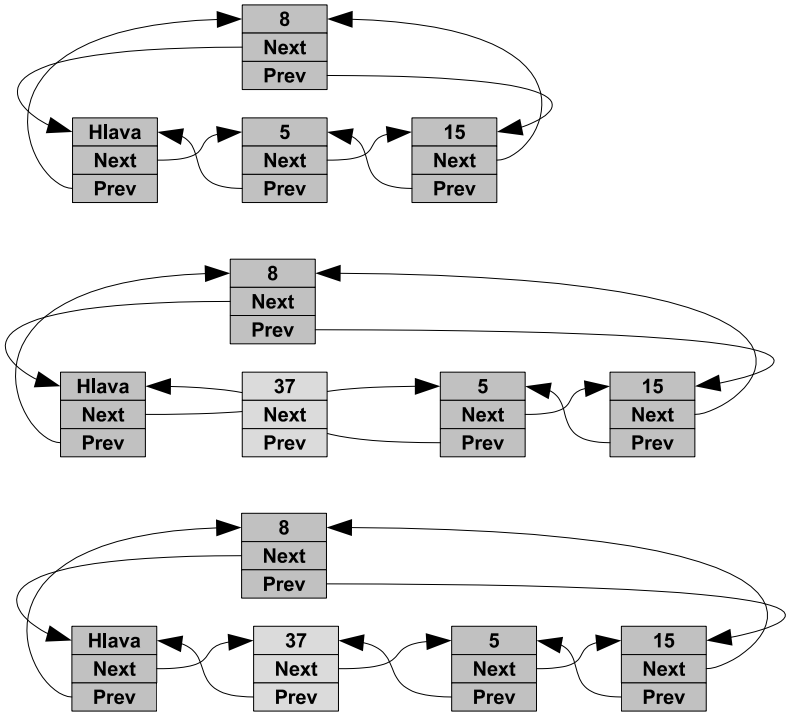
Obrázek 1.11: Operace s obousměrným seznamem s hlavou

Obě výše popsané vlastnosti lze kombinovat. Může tedy vzniknout například obousměrný cyklický seznam s hlavou, se kterým se velmi dobře pracuje a využívá se nejen v jádru Windows.

Spojové seznamy se hodí na reprezentaci souborů dat, které se hodně mění a operace přidávání a mazání probíhají na prvních nebo posledních prvcích. Na rozdíl od pole však není možné rychle zjistit hodnotu N-tého prvku od začátku (či od konce) – spojové seznamy neumožňují rychlý náhodný přístup. Je nutné postupně projít všechny předcházející (nebo následující) prvky, protože adresa N-tého prvku je uložena pouze v jeho předchůdci (či následníkovi).

Windows obousměrným cyklickým seznamem s hlavou reprezentují například seznam běžících procesů či seznam ovladačů načtených do jádra. Seznam služeb je reprezentován prostým obousměrným seznamem. Jednosměrné spojové seznamy se využívají například ve strukturách správce paměti. Stejně jako pole i spojové seznamy nachází uplatnění při implementaci pokročilejších datových struktur jako zásobník, fronta či hašovací tabulka.

Přidání prvního prvku (provedení operací v opačném pořadí zajistí odebrání prvního prvku)



Obrázek 1.12: Operace s obousměrným cyklickým seznamem

Protože druhů spojových seznamů je relativně mnoho a tato pasáž může být pro čtenáře nepřehledná, tabulka 1.2 ukazuje, které operace lze snadno provádět nad daným druhem spojového seznamu.

Tabulka 1.2: Výhodné operace nad různými variantami spojových seznamů

Operace	Přidání				Odebrání				Přístup	
Spojový seznam	Na začátek	Na konec	Za známý prvek	Před známý prvek	Prvního prvku	Posledního prvku	Předchozího prvku	Následujícího prvku	Na první prvek	Na poslední prvek
Lineární	Ano	Ne	Ano	Ne	Ano	Ne	Ne	Ano	Ano	Ne
Lineární cyklický	Ano	Ne	Ano	Ne	Ano	Ne	Ne	Ano	Ano	Ne
Obousměrný	Ano	Ne	Ano	Ano	Ano	Ne	Ano	Ano	Ano	Ne
Obousměrný cyklický	Ano	Ano	Ano	Ano	Ano	Ano	Ano	Ano	Ano	Ano

Zásobník

Zásobník je datová struktura pro uchovávání souboru prvků, která podporuje pouze následující operace:

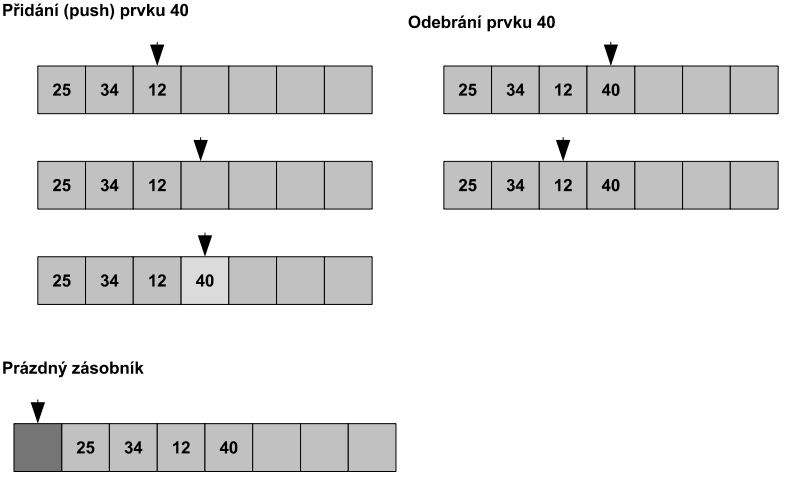
- Vložení nového prvku (`push`).
- Odebrání naposledy vloženého prvku (`pop`).
- Test prázdnosti (`empty`).

Je vidět, že nad takovouto strukturou lze provádět méně operací než s polem či se spojovým seznamem. Protože tyto operace tvoří podmnožinu operací nad oběma výše popsanými strukturami, je možné je pomocí pole či spojového seznamu implementovat.

Implementace prostřednictvím pole může vypadat například takto:

- K poli si zavedeme proměnnou `PosledniPlatny`, ve které si budeme pamatovat index posledního platného prvku. Na začátku, kdy je zásobník prázdný, má tato proměnná hodnotu –1.
- Z předchozího bodu vyplývá implementace operace `empty`. Zásobník je prázdný právě tehdy, když je hodnota proměnné `PosledniPlatny` rovna –1.
- Operace `push` znamená zvýšení hodnoty proměnné `PosledniPlatny` o jedničku a uložení nového prvku do slotu s příslušným indexem.
- Operace `pop` se implementuje jako snížení hodnoty proměnné `PosledniPlatny` o jedničku a vrácení prvku s indexem rovným její předchozí hodnotě.

Průběh všech operací graficky znázorňuje obrázek 1.13.



Obrázek 1.13: Implementace zásobníku pomocí pole

Tato implementace v sobě nese výhody a nevýhody pole jako takového. Pokud by se do zásobníku přidávalo hodně prvků, může dojít k naplnění celé kapacity pole. V takovém případě bude potřeba alokovat větší blok paměti a překopírovat do něho obsah původního pole. Na druhou stranu, pokud není potřeba měnit velikost pole, tak operace `push` a `pop` znamenají pouze inkrementaci a dekrementaci jedné proměnné a jeden zápis nebo čtení.

Zásobník je možné snadno realizovat i přes spojový seznam a ani není nutné zakládat pomocnou proměnnou.

- Zásobník je prázdný právě tehdy, když je spojový seznam prázdný.
- Operace `push` spočívá ve vložení příslušného prvku na začátek nebo konec seznamu. Zde záleží na tom, jaký druh seznamu je pro implementaci zásobníku zvolen, tedy jaké operace vkládání jsou výhodné. U necyklických seznamů se vyplatí přidávat na začátek, u cyklických obousměrných je možné přidávat i na konec.
- Operace `pop` znamená odebrání prvku ze začátku nebo z konce seznamu. Opět záleží na volbě konkrétního spojového seznamu. U necyklických seznamů se vyplatí odebírat první prvek, u cyklických obousměrných je možné odebírat i z jejich konce.

U implementace zásobníku spojovým seznamem nehrozí žádné přetečení pole, takže všechny operace budou mít stále stejnou časovou složitost. Na druhou stranu přidávání a odebírání prvku bude několikrát pomalejší než v případě pole, protože se manipuluje s adresami, nejde jen o zvýšení či snížení hodnoty jedné proměnné.

Fronta

Fronta je datová struktura, která podporuje následující operace:

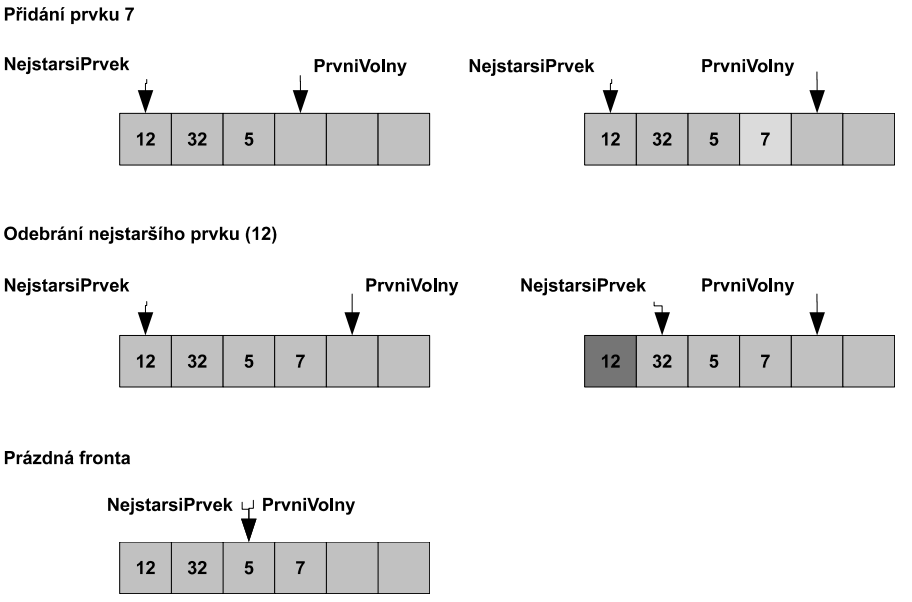
- Vložení nového prvku (`insert`).
- Odebrání prvku, který se ve frontě nachází nejdelší dobu – byl ze všech prvků vložen nejdříve (`removefirst`).
- Test, zda je struktura prázdná (`empty`).

Stejně jako zásobník, i frontu je možné implementovat jak pomocí pole, tak pomocí spojového seznamu.

Pomocí pole lze tuto datovou strukturu realizovat například následujícím způsobem:

- Zavedeme dvě pomocné proměnné `NejstarsiPrvek` a `PrvniVolny`, které budou udávat index nejstaršího prvku a index slotu, který bude při příští operaci `insert` zaplněn. Na počátku obě proměnné ponesou hodnotu 0 (v prázdné frontě neexistuje žádný nejstarší prvek a první volný slot pole má index 0). Předpokládáme pole velikosti N s možnými hodnotami indexů od 0 do N-1.
- Fronta je prázdná právě tehdy, když se hodnoty obou pomocných proměnných rovnají.
- Při operaci `insert` se do indexu, jehož hodnotu obsahuje proměnná `PrvniVolny`, uloží nový prvek a proměnná `PrvniVolny` se zvýší o jedničku modulo `N`. Pokud by `PrvniVolny` dosáhl stejné hodnoty jako `NejstarsiPrvek`, znamená to, že došlo k přeplnění pole. V takovém případě je nejprve potřeba alokovat nový větší blok paměti a přidat do něho všechny prvky z původní fronty. Při přidávání se postupuje přesně podle tohoto bodu, a tedy je zaručeno, že k opětovnému přeplnění nemůže dojít (nově alokovaný blok pro pole je větší než původní).
- Při operaci `removefirst` se proměnná `NejstarsiPrvek` zvýší o jedničku modulo `N` a jako prvek se vrátí obsah slotu s indexem rovným její původní hodnotě. Tuto operaci není možné provést, pokud je fronta prázdná – jsou-li hodnoty obou pomocných proměnných shodné.

Znázornění implementace fronty pomocí pole vidíte na obrázku 1.14.



Obrázek 1.14: Implementace fronty pomocí pole

Implementace pomocí pole má stejné výhody a nevýhody jako u zásobníku. Operace `insert` a `removefirst` znamenají jen zvyšování a snižování indexů nejstaršího prvku a prvního volného slotu, což je velmi rychlé. Opět ale může dojít k přeplnění pole a k problémům s tím spojeným.

K implementaci fronty se obzvláště hodí obousměrný cyklický spojový seznam. Přináší stejné výhody a nevýhody jako v případě implementace zásobníku – časová složitost všech operací je předvídatelná (nemůže dojít k anomálii přeplnění pole), ale práce s adresami je pomalejší. Operace lze implementovat například následovně:

- Fronta je prázdná právě tehdy, když seznam neobsahuje žádný prvek.
- Operace `insert` se realizuje přidáním nového prvku na konec seznamu. Přidávání na konec patří u cyklického obousměrného spojového seznamu mezi rychlé operace.
- Operace `removefirst` spočívá v odebrání prvního prvku seznamu. Odebrání prvního prvku lze u každého spojového seznamu (tedy i u obousměrného cyklického) realizovat v jednom kroku.

Při implementaci spojovým seznamem tedy není potřeba zavádět žádné pomocné proměnné.

Hašovací tabulky

Výše popsané datové struktury dokáží velmi rychle zvládat přidávání či odebírání prvků, ale mají jednu podstatnou nevýhodu – neumí efektivně vyhledávat podle obsahu. Výjimku tvoří setříděné pole, ale jeho příprava vyžaduje nějaký čas a přidávání do takové struktury také není zadarmo.

Představte si, že pole či spojový seznam obsahuje soubory, které se nachází v adresáři `C:\Windows` a vy byste rádi zjistili, jestli existuje soubor `C:\Windows\explorer.exe`. Souborový systém nemusí vracet seznam souborů nějakého adresáře setříděný, takže metodu půlení intervalů nelze použít. Pole ani spojový seznam nenabízí žádnou možnost, jak existenci souboru `explorer.exe` rychle ověřit. Je nutné projít všechny soubory v seznamu a u každého se ptát, zda se nejmenuje `explorer.exe`. Naštěstí jsou známy i datové struktury, které existenci určitého prvku dokáží ověřit v jediném kroku, a hašovací tabulka patří mezi ně.

Hašovací tabulka je datová struktura, která umožňuje vkládat, vyhledávat a mazat prvky v konstantním čase – tedy délka trvání těchto operací nezávisí na počtu prvků, které tabulka obsahuje. Prvky se vyhledávají podle části jejich obsahu, která se nazývá *klíč*. Klíčem do hašovací tabulky obsahující informace o souborech v adresáři mohou být třeba jejich jména.

Všechny prvky hašovací tabulky se uchovávají v poli, jehož položkám se říká *sloty*. Při operacích nad hašovací tabulkou se klíč transformuje pomocí *hašovací funkce* na index slotu, nad kterým se provede daná operace (například při přidávání se do něho zapíše hodnota nového prvku). Protože lze zajistit, aby operace nad jednotlivými sloty probíhaly v konstantním čase, všechny operace s hašovací tabulkou probíhají též v konstantním čase. Doba transformace klíče na index do pole je nezávislá na velikosti tabulky či počtu obsažených prvků.

Jak bylo řečeno, úkolem hašovací funkce je transformovat hodnotu klíče na index slotu, který bude pro prvky s tímto klíčem využíván. Hašovací funkce by měla splňovat následující podmínky:

- Transformace klíče musí proběhnout rychle.
- Pro stejnou hodnotu klíče musí vracet vždy stejný index slotu.
- Musí jednotlivé hodnoty klíče překládat na indexy slotů rovnoměrně. Na každý index by se mělo přeložit (mapovat) přibližně stejné množství možných hodnot klíče. Tuto podmínku nemusí být snadné dodržet, protože do hašovací tabulky často nejsou ukládána náhodná data, která se vyznačují různými hodnotami klíče. Hašovací funkci je tedy rozumné přizpůsobit povaze dat, se kterými bude hašovací tabulka pravděpodobně pracovat.

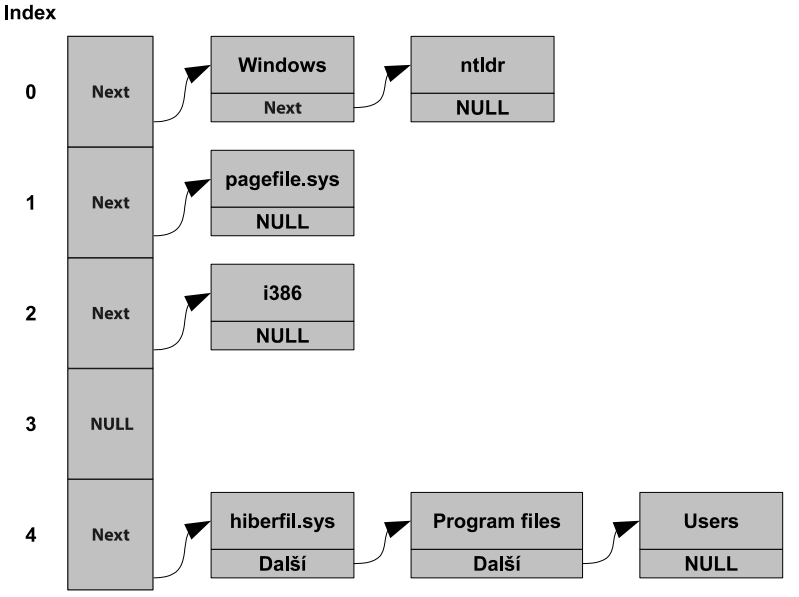
Jak napovídá třetí vlastnost, může dojít k situaci, kdy hašovací funkce transformuje několik klíčů s různou hodnotou na stejný index slotu. Tomuto problému se nelze vyhnout, protože množina všech hodnot klíče bývá obvykle mnohem větší než počet slotů v poli hašovací tabulky. Například klíče tvořené řetězci mohou nabývat v podstatě nekonečně mnoha hodnot, pole tvořící hašovací tabulku musí respektovat paměťová omezení v mnohem větší míře. Situace, kdy dojde k překladu různých klíčů na stejný index slotu, se nazývá *kolize*.

Nemožnost eliminovat kolize znamená, že je nutné se nějak vypořádat s faktem, že jeden slot může obsahovat více prvků. Existuje několik možností, jak tento problém řešit. Jednou z metod často používaných v jádru Windows, je *řetězení prvků*.

Podstata metody spočívá v tom, že každý slot hašovací tabulky je tvořen spojovým seznamem prvků, jejichž klíč byl hašovací funkcí transformován na příslušný index. Při provádění operací nad takovým slotem je vždy nutné projít všechny prvky spojového seznamu a najít ten, kterého se příslušná operace týká. Třetí jmenovaná vlastnost hašovací funkce přitom zajišťuje, že délka spojového seznamu bude pro každý slot přibližně stejná. Práce s takto konstruovanou hašovací tabulkou je stále velmi rychlá, pokud spojové seznamy neobsahují mnoho prvků. Jakmile délka seznamů překročí určitou mez, je třeba vytvořit novou hašovací tabulku s větším počtem slotů. Další nevýhodou této metody je nutnost alokovat blok paměti při každém přidání nového prv-

ku, což o ostatních přístupech k řešení kolizí neplatí. Na druhou stranu řetězení prvků umožňuje plnohodnotně prvky z hašovací tabulky odstranit.

Ukázku hašovací tabulky s řešením kolizí pomocí řetězení prvků vidíte na obrázku 1.15.



Obrázek 1.15: Hašovací tabulka s metodou řešení kolizí pomocí řetězení prvků

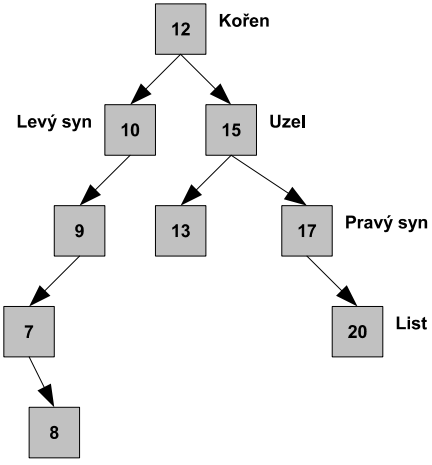
Stromy

Hašovací tabulky dokáží s daty pracovat velmi rychle, ale existují situace, na které se vůbec nehodí. Představte si, že máte v hašovací tabulce uloženy informace o souborech a složkách v adresáři `C:\Windows`. Velmi snadno a rychle zjistíte, zda existuje soubor s určitým jménem, ale pokud chcete znát odpověď na otázku *„Které soubory a složky v C:\Windows mají jména mezi aaa.txt a explorer.exe?“* (předpokládá se řazení jmen podle abecedy), musíte projít celou tabulku, což bude časově náročné, protože adresář `C:\Windows` obsahuje mnoho položek.

Hašovací tabulky se nehodí na tzv. *intervalové dotazy* – dotazy na přítomnost prvků v určitém rozsahu hodnot klíče. Naopak stromy jsou na intervalové dotazy velmi vhodné a ani standardní operace jako vkládání, vyhledávání na shodu či mazání, ač jsou časově závislé na velikosti struktury, na nich neprobíhají nijak pomalu – počet kroků nutný k jejich provedení je logaritmický vzhledem k počtu prvků ve stromě.

Svou vnitřní strukturou se strom mírně podobá lineárnímu spojovému seznamu. Každý prvek (označovaný též jako *uzel*) se nachází v separátním bloku paměti. U lineárních spojových seznamů má každý prvek krom posledního v sobě uložen odkaz na následníka. U stromu může mít jeden uzel následníků více a ti se označují jako *synové*. Uzlům, které žádné následníky nemají, se říká *listy*. Stejně jako lineární spojový seznam, i strom obsahuje uzel, který nepatří mezi následníky žádného jiného uzlu. Tento uzel se označuje jako *kořen*. Příklad takového stromu vidíte na obrázku 1.16.

Existuje mnoho různých variant stromů. Protože jich je většina relativně složitá a jejich detailní popis není tématem této knihy, v této kapitole bude zmíněn pouze jeden z nejjednodušších, kterým je *binární vyhledávací strom*.



Obrázek 1.16: Ukázka struktury stromu (binární vyhledávací strom)

Binární vyhledávací strom je strom s následujícími vlastnostmi:

- Každý uzel má nejvýše dva následníky. Tyto uzly se někdy označují jako *levý následník* a *pravý následník* či *levý syn* a *pravý syn*.
- Pro každý uzel platí, že jeho hodnota je větší než hodnoty všech uzlů v *levém podstromě* (ve stromě, jehož kořenem je levý následník) a menší než hodnoty všech uzlů v *pravém podstromě* (ve stromě, jehož kořenem je pravý následník).

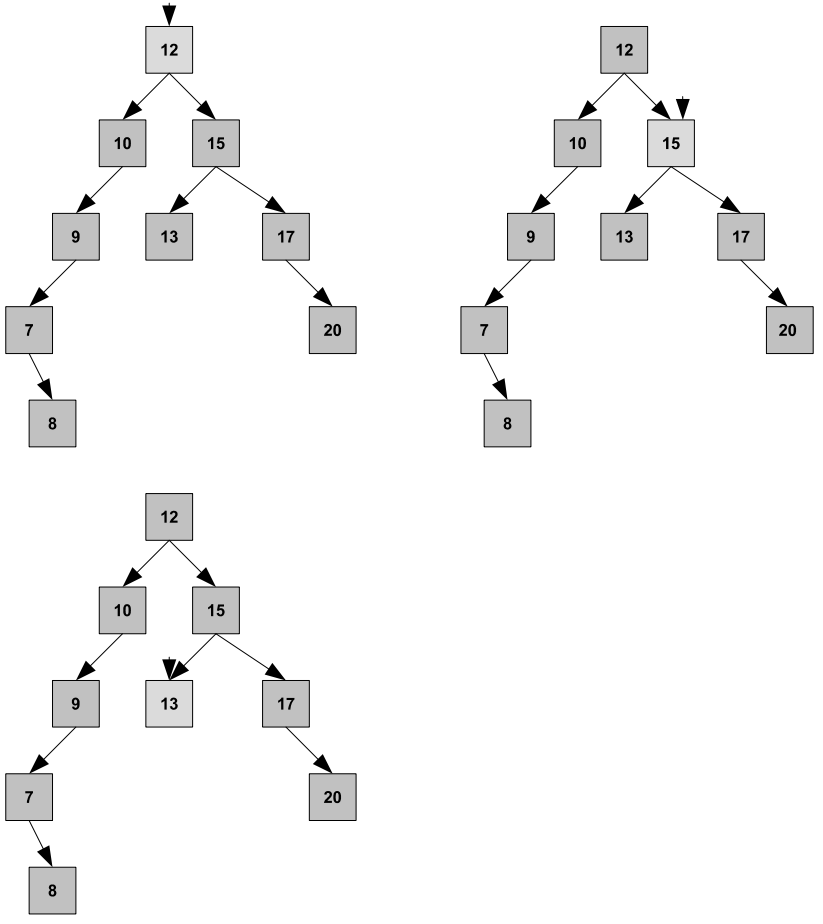
Vyhledávání uzlu s určitou hodnotou probíhá následujícím způsobem:

1. Za aktuální uzel zvolíme kořen.
2. Je-li hodnota aktuálního uzlu rovna hodnotě, kterou hledáme, vyhledávání končí a jako výsledek vrací adresu aktuálního uzlu.
3. Je-li hodnota, kterou hledáme, menší než hodnota v aktuálním uzlu, zvolíme za aktuální uzel jeho levého následníka a pokračujeme krokem (2). Pokud aktuální uzel nemá levého následníka, vyhledávání končí neúspěchem – uzel s hledanou hodnotou se ve stromě nenachází.
4. Je-li hledaná hodnota větší než hodnota aktuálního uzlu, zvolíme za aktuální uzel jeho pravého následníka a pokračujeme krokem (2). Pokud aktuální uzel nemá pravého následníka, vyhledávání končí neúspěchem – uzel s hledanou hodnotou se ve stromě nenachází.

Ukázku průběhu vyhledávání v binárním vyhledávacím stromě vidíte na obrázku 1.17.

Operace přidávání nového uzlu a mazání existujícího uzlu jsou založeny na vyhledávání. Při přidávání se postupuje úplně stejně jako při vyhledávání. Pokud je nalezen uzel se stejnou hodnotou jako má nově přidávaný uzel, operace skončí neúspěchem (v základní variantě binárního vyhledávacího stromu nemohou existovat dva uzly se stejnou hodnotou klíče). Pokud se uzel se stejnou hodnotou ve stromě nenachází, vyhledávání skončí na uzlu, který nemá levého nebo

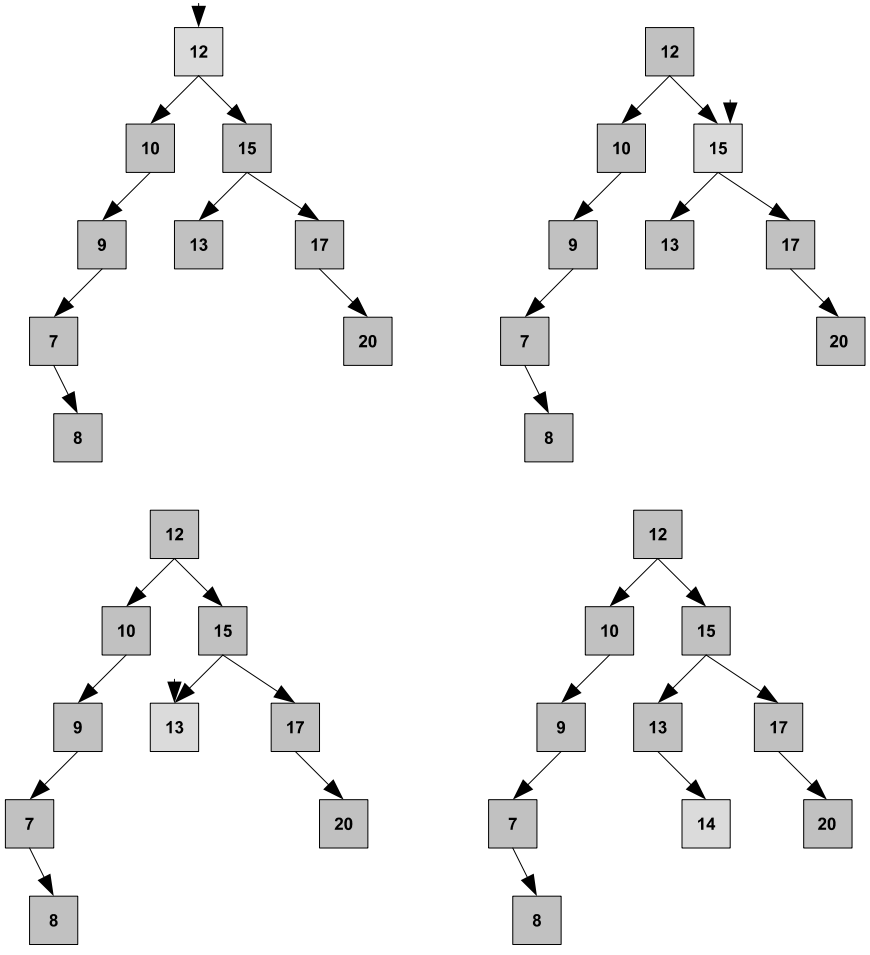
pravého následníka (nemusí mít ani jednoho). Nový uzel je přidán jako levý (resp. pravý) následník, záleží na jeho hodnotě vzhledem k hodnotě jeho předchůdce (otce). Přidávání nového uzlu zachycuje obrázek 1.18.



Obrázek 1.17: Průběh vyhledávání hodnoty 13 v binárním vyhledávacím stromě

Mazání již existujícího uzlu zpočátku probíhá stejně jako přidávání, další část operace je však již složitější, ale stále probíhá v logaritmickém počtu kroků vzhledem k počtu uzlů ve stromě.

Upozornění: Test existence, přidávání a mazání probíhají u binárních vyhledávacích stromů v logaritmickém počtu kroků vzhledem k velikosti struktury pouze v případě, že jsou jednotlivé prvky přibližně rovnoměrně rozprostřeny do celého stromu. Při plnění této datové struktury nevhodnou posloupností dat může dojít k degeneraci na spojový seznam (každý uzel má pouze jednoho následníka), což se odrazí na časovém průběhu všech operací, pro jejichž rychlost se stromy využívají.



Obrázek 1.18: Přidání nového uzlu s hodnotou 14 do binárního vyhledávacího stromu

Tip: Existují však rozšíření, která, aplikovaná na binární vyhledávací strom, garantují, že k degeneraci na spojový seznam nikdy nedojde a že časová náročnost operací zůstane vždy logaritmická vzhledem k počtu uzlů. Takové stromy se označují jako *samovyvažující* (*self-balancing*) a jedná se například o červenočerné stromy či stromy AVL.

Architektura rodiny operačních systémů Windows NT

První kapitola byla spíše obecným úvodem do problematiky operačních systémů. Kapitola druhá se mnohem více zaměřuje na detailnější popis architektury operačních systémů založených na NT technologii, ačkoliv o několik odstavců s obecnými informacemi také nebudete ochuzeni.

V první části se seznámíte s jedním z nejzásadnějších rozhodnutí, které musí návrháři operačního systému udělat hned na počátku celého tvůrčího procesu. Musí se rozhodnout, jak pokročilé funkce budou implementovány v jeho jádře. Toto rozhodnutí má dopady nejen na celkovou obtížnost navrhování a programování, ale i na výkon, bezpečnost a stabilitu výsledného díla.

Další části kapitoly se již zaměří výhradně na architekturu Windows NT. Dozvíte se, z jakých součástí se celý operační systém skládá, k čemu jednotlivé komponenty slouží a jak na sobě závisí.

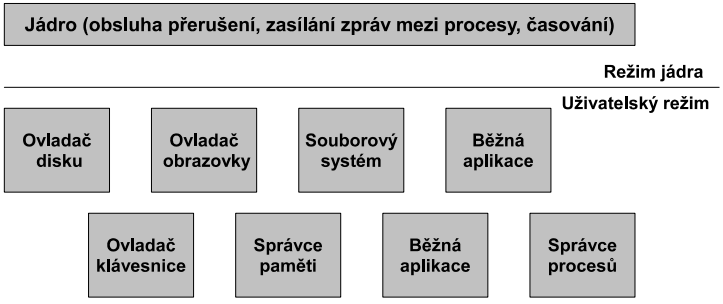
Mikrojádro a monolitický operační systém

Od rozhodnutí, jak moc pokročilé funkce bude umět samotné jádro, se odvíjí další postup při návrhu a implementaci. Existují dvě základní možnosti, jak se rozhodnout – buď bude jádro umět minimum, nebo naopak skoro všechno. Obě varianty mají svá pro a proti a výsledný operační systém většinou leží někde mezi těmito možnostmi – samotné jádro disponuje pokročilými funkcemi, ale některé vlastnosti jsou stále implementovány prostřednictvím aplikací uživatelského režimu.

Jak již název napovídá, systémy založené na *mikrojádru* (*mikrokernel*), obsahují velmi malé jádro, které implementuje pouze některé základní mechanismy jako virtuální paměť, plánování vláken, obsluhu výjimek a posílání zpráv mezi procesy. Ostatní komponenty (souborové systémy, síťová komunikace, správce procesů) běží v uživatelském režimu jako aplikace.

Mezi výhody architektury založené na mikrojádru patří vyšší stabilita a menší nároky na programátorské schopnosti vývojářů systému. Protože většina součástí běží v uživatelském režimu, selhání jedné z nich nemusí znamenat pád celého systému – například není možné narušit integritu jádra nechtěným přepsáním jeho datových struktur v ovladači souborového systému. Při případné chybě v součásti systému často stačí příslušnou komponentu restartovat a systém může pokračovat ve své činnosti. Malé jádro tedy znamená méně kritického kódu (kódu, který mů-

že způsobit selhání celého systému), čímž se zjednodušuje programování celého projektu. Schéma systému založeného na mikrojádru znázorňuje obrázek 2.1.



Obrázek 2.1: Schéma operačního systému založeného na mikrojádru

Mikrojádro v podstatě slouží jenom k plánování vláken a k předávání zpráv mezi procesy, reprezentujícími jednotlivé součásti systému. Protože jsou všechny pokročilé funkce implementovány v uživatelském režimu, ke komunikaci mezi procesy dochází velmi často, což vede k velkému počtu přechodů mezi uživatelským režimem a režimem jádra a k častým změnám virtuálního adresového prostoru. Obě tyto operace, zvláště pak ta druhá, jsou náročnější na výkon procesoru. Z tohoto důvodu mohou být systémy založené na mikrojádru pomalejší než systémy monolitické, ačkoliv architektura malého jádra je z teoretického pohledu mnohem čistší a elegantnější.

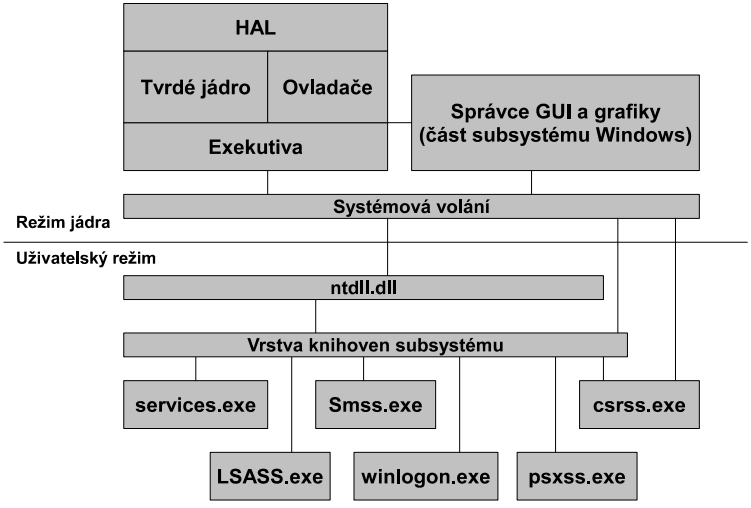
Monolitické operační systémy jsou přesným opakem mikrokernelů. Disponují velkým jádrem obsahujícím většinu komponent nutných pro běh celého systému – souborové systémy, správu procesů, síťovou komunikaci či bezpečnostní model. Obvykle všechny komponenty jádra sdílejí jeden virtuální adresový prostor, což zrychluje jejich vzájemnou komunikaci, ale na druhou stranu správce procesů například může nechtěně poškodit datové struktury nutné pro správné fungování ovladače souborového systému, což zcela určitě povede k pádu celého systému.

Velké jádro tedy s sebou přináší teoreticky nižší stabilitu a bezpečnost, ale vyšší výkon. Větší množství kritického kódu zvyšuje obtížnost implementace celého projektu. Schéma architektury monolitického operačního systému vidíte na obrázku 2.2.

Obrázek 2.2: Schéma monolitického operačního systému

Windows NT a jeho součásti

Operační systémy rodiny Windows NT patří spíše do druhé skupiny. Schéma jejich architektury najdete na obrázku 2.3. Většina pokročilých funkcí (souborové systémy, síťová komunikace, správa paměti a procesů) běží v režimu jádra. Systém pro svoji plnohodnotnou činnost vyžaduje běh pouze několika málo procesů, které zajišťují správu služeb či dovolují přihlášení uživatele přes grafické rozhraní. Všechny součásti jádra a ovladače zařízení sdílí stejný virtuální adresový prostor.



Obrázek 2.3: Architektura Windows NT

Následující odstavce vás postupně seznámí s jednotlivými komponentami systému, které vidíte na obrázku 2.3. O většině z nich velmi podrobně pojednávají další kapitoly této knihy.

Vrstva abstrakce hardwaru (Hardware Abstraction Layer – HAL)

Windows jsou psány tak, aby nebylo obtížné upravit celý systém pro běh na dalších platformách – například na nových procesorech či základních deskách. Přenositelnost patří mezi důvody, proč je celé jádro naprogramováno převážně v jazyce C, kterému někteří přezdívají „přenositelný Assembler". Pouze velmi malou část kódu psali vývojáři v Assembleru, jazyku striktně závislém na konkrétním procesoru či rodině procesorů. Pro přenesení Windows na novu architekturu tedy stačí přepsat některé části jádra a systémových knihoven DLL, které zapouzdřují mechanismy, jejichž implementace závisí na konkrétním druhu procesoru (přepínání kontextu vláken, obsluha stránkovacích tabulek, systémová volání).

Jedním z cílů při navrhování nového operačního systému bezesporu je usnadnit programátorům psaní aplikací a ovladačů. Programátoři běžných aplikací by například neměli řešit odlišnosti různých typů procesorů. To platí i o většině ovladačů jádra. Z tohoto důvodu se architektura Windows dělí do několika vrstev. Každá vrstva v sobě skrývá některé specifické vlastnosti a problémy a vrstvám vyšší úrovně poskytuje obecná rozhraní.

HAL se v architektuře Windows nachází na nejnižší úrovni a jejím úkolem je odstínit ostatní části operačního systému a aplikace od specifik hardware, jako jsou různé modely procesorů. Odtud také pochází název *Vrstva abstrakce hardware* (*Hardware Abstraction Layer*). Vyšším

vrstvám poskytuje například rutiny pro komunikaci s periferními zařízeními. Kód HAL se nachází v souboru `hal.dll` v systémovém adresáři.

Tvrdé jádro

Nad HAL se nachází tenká vrstva, jež implementuje relativně jednoduché mechanismy, kterých využívají vyšší vrstvy jádra operačního systému pro stavbu složitějších struktur. Kód tvrdého jádra je pomocí vrstvy HAL odstíněn od většiny specifik hardware, a tudíž zde Assembleru najdete jen málo.

Mezi mechanismy implementované v této součásti patří:

- Algoritmus plánování vláken na procesoru.
- Odložené volání procedur (Deferred Procedure Call – DPC).
- Základní synchronizační primitiva jako událost (event), spinlock, semafor či pushlock.
- Práce s hardwarovými přerušeními.
- Část obsluhy systémových volání.

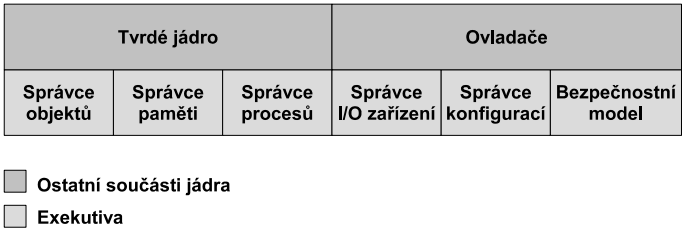
Ovladače

Ovladače umožňují správci vstupně/výstupních zařízení, který spolu s dalšími komponentami tvoří vrstvu v této knize označovanou jako *exekutiva*, komunikovat s různými typy hardware. Jedná se o spustitelné soubory formátu PE a jejich jméno většinou obsahuje koncovku `.sys`. Bližší informace se dozvíte v kapitolách 3 a 8.

Exekutiva

Exekutiva využívá tvrdého jádra k realizaci mnohem složitějších mechanismů, kterých mohou přes systémová volání nepřímo využívat obyčejné aplikace běžící v uživatelském režimu. Obsahuje ještě méně platformně specifického kódu než tvrdé jádro.

Exekutiva se skládá z několika navzájem oddělených částí (viz obrázek 2.4). Ačkoliv se všechny nachází ve stejném adresovém prostoru, a tudíž by například správce objektů mohl přímo manipulovat s interními datovými strukturami, které náleží správci paměti, komunikují spolu pomocí přesně definovaných rozhraní. Každá komponenta dává k dispozici sadu rutin, které může volat libovolný kód běžící v režimu jádra. Názvy některých funkcí a způsob realizace některých mechanismů připomínají techniky používané v objektově orientovaném programování.



Obrázek 2.4: Interní struktura exekutivy

Následující odstavce krátce charakterizují jednotlivé součásti. Podrobněji se jimi zabývají další kapitoly této knihy.

Správce objektů (*Object Manager*) umožňuje vývojářům psát kód jádra podobně, jako by programovali v jazyce podporujícím konstrukty OOP. Správce objektů například umožňuje jednotným způsobem objekty (otevřené soubory, klíče registru, paměťově mapované soubory) vytvářet a odstraňovat a obsahuje jednotný mechanismus řízení přístupu. Zjednodušeně lze říci, že správce objektů v sobě zapouzdřuje všechny principy, které platí pro většinu druhů objektů (při odstraňování objektu dochází k uvolnění paměti nezávisle na jeho typu, při vytváření nového objektu zase dochází k alokaci paměti, ať už se jedná o otevřený soubor, nebo klíč registru). Tuto součást exekutivy podrobně popisuje kapitola 6.

Správce paměti (*Memory Manager*) se stará o všechny činnosti, které nějak souvisí s virtuální a fyzickou pamětí. Přiděluje volné rámce fyzické paměti, zajišťuje správné mapování mezi virtuálními a fyzickými adresami a obsluhuje výpadky stránky. Vyřizování požadavků na přidělování a uvolňování bloků virtuální paměti o proměnlivé velikosti patří též do jeho kompetencí. Dále obsahuje bezpečnostní mechanismy jako ASLR (*Address Space Layout Randomization*). Podrobné informace najdete v kapitole 9.

Správce vstupně/výstupních zařízení (*I/O Manager*) zajišťuje většinu funkcí kolem ovladačů jádra a zařízení, ať už fyzických nebo virtuálních. Díky této součástí může jádro za běhu načítat nové ovladače do svého paměťového prostoru a uvolňovat ty, které aktuálně nepotřebuje. Správce vstupně/výstupních zařízení též dovoluje ovladačům komunikovat mezi sebou. Komunikace probíhá přes objekty zvané *zařízení* (*device*), které si pro jednoduchost můžete představit jako kanály nebo roury mezi jednotlivými ovladači. Podrobněji se jimi zabývá kapitola 8.

Správce procesů (*Process Manager*) má na starosti spouštění, běh a ukončení procesů a vláken. Umožňuje ostatním součástem systému a běžným aplikacím zjistit informace o právě běžících procesech, měnit jejich prioritu, násilně je ukončovat a provádět další zajímavé akce. Této součásti se detailně věnuje kapitola 7.

Windows patří mezi operační systémy, jejichž cílem je plnit co největší množství různorodých úkolů a přizpůsobit své chování co nejvíce požadavkům uživatele. Nastavení různých aspektů systému musí být někde uloženo a systém k němu musí mít snadný přístup. A způsob ukládání a práce s konfiguracemi je parketa pro součást exekutivy s názvem *správce konfigurací* (*Configuration Manager*).

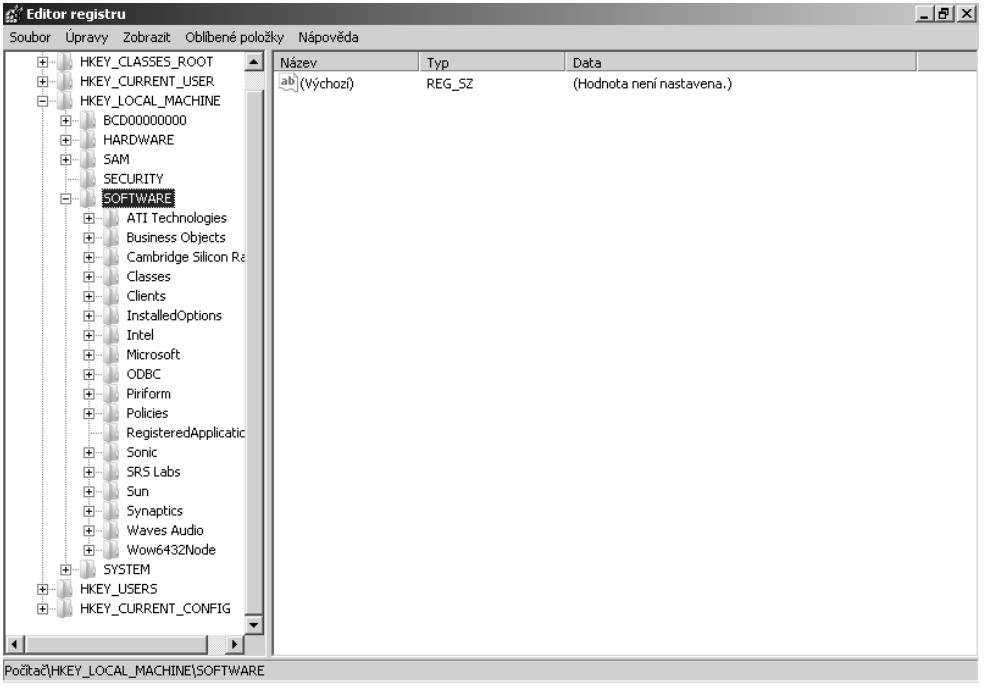
Na rozdíl od operačních systémů založených na Unixu, které většinu nastavení ukládají do textových souborů, Windows své konfigurace ukládá v binární podobě – do tzv. registru. Registr funguje jako malá databáze, rychle se v něm vyhledává a manipuluje s jednotlivými položkami. Na rozdíl od textových souborů, dobře čitelných pro obyčejného člověka, pro prohlížení obsahu registru potřebujete speciální programy. Jedním z nich je Editor registru (regedit.exe), který vidíte na obrázku 2.5.

Vývojáři Windows se pro binární formu ukládání konfigurací rozhodli pravděpodobně z toho důvodu, že textové soubory se hůře a pomaleji počítačově zpracovávají a zabírají více místa.

Logická struktura registru je téměř totožná s adresářovou strukturou na pevném disku. Adresáře se v této terminologii registru nazývají *klíče* a souborům se říká *hodnoty*. Každý klíč může obsahovat libovolné množství dalších klíčů (podadresářů) a hodnot (souborů). Pouze hodnoty mohou obsahovat data.

V dřevních dobách operačního systému MS–DOS a v raných verzích Windows se pro ukládání nastavení používaly soubory s příponou `.ini`. Každý takový soubor se skládá ze *sekcí*. Každá sekce může obsahovat *záznamy*. Tvar záznamu vypadá následovně:

```
<název_položky>=<hodnota>
```



Obrázek 2.5: Editor registru – regedit.exe

I v dnešní době některé součásti Windows a některé aplikace používají pro ukládání nastavení soubory INI. Ukázkový příklad takového souboru vidíte ve výpisu 2.1.

Výpis 2.1: Ukázkový soubor INI

```
[386Enh]
woafont=dosapp.fon
EGA80WOA.FON=EGA80WOA.FON
EGA40WOA.FON=EGA40WOA.FON
CGA80WOA.FON=CGA80WOA.FON
CGA40WOA.FON=CGA40WOA.FON
[drivers]
wave=mmdrv.dll
timer=timer.drv
[mci]
```

Soubor INI z výpisu 2.1 se skládá ze tří sekcí: `386Enh`, `drivers` a `mci`. První sekce obsahuje několik záznamů, například hodnota položky `CGA40WOA.FON` je `CGA40WOA.FON`. V sekci `drivers` se nachází pouze dva záznamy a sekce `mci` je prázdná. Pokud se rozhodnete říkat sekcím *klíče*, položkám záznamů *hodnoty* a hodnotám položek *data hodnot*, zjistíte, že logická struktura souboru INI a registru je velmi podobná. Na rozdíl od registru však soubory INI samy od sebe neumožňují hierarchické uspořádání sekcí (klíčů), což však lze nasimulovat zavedením pravidel pro jejich pojmenování. Rozdíl ve fyzické struktuře je samozřejmě obrovský.

Podrobný popis správce konfigurací naleznete v kapitole 10.

Operační systém také musí zajistit, aby každý mohl provádět jen operace, na které má dostatečná oprávnění. To je úkolem *bezpečnostního modelu* (*security model*), který umožňuje pro každého uživatele nastavit, jaké činnosti smí a nesmí provádět. Díky bezpečnostnímu modelu může administrátor systému určit, k jakým objektům mají jednotliví uživatelé přístup. Množina objektů přitom není omezena jenom na soubory a klíče registru; oprávnění lze nastavit i u procesů, vláken, nebo třeba synchronizačních primitiv. Bezpečnostní model Windows dále definuje sadu oprávnění, které uživatel potřebuje, aby mohl vykonávat určitou činnost. Do této kategorie patří například oprávnění vytvořit stránkovací soubor či povolení načíst ovladač do jádra.

Celá exekutiva je implementována v hlavním modulu jádra (`ntoskrnl.exe` nebo `ntkrnlpa.exe`). Tento soubor formátu PE exportuje velké množství funkcí. Jejich názvy mají pevně danou strukturu. Jméno se skládá z předpony, která určuje, do jaké kategorie rutina patří, a několika slov, jež popisují její účel. Příklady vidíte v tabulce 2.1.

Tabulka 2.1: Ukázky názvů některých rutin

Předpona	Příklady názvů rutin	Popis
Cm	`CmRegisterCallback`, `CmRegisterCallbackEx`	Rutiny patří k rozhraní, které ovladačům umožňuje monitorovat operace nad registrem. O tuto funkci se stará správce konfigurací.
Ex	`ExAllocatePoolWithTag`, `ExFreePoolWithTag`	Rutiny různého druhu, které exekutiva poskytuje ostatním. Některé slouží pro alokaci a uvolňování paměti, jiné třeba pro synchronizaci.
Ke	`KeAcquireSpinLock`, `KeDelayExecutionThread`	Poskytované komponentou Tvrdé jádro. Určené pro práci s vlákny, obsluhu přerušení a výjimek a synchronizaci.
Rtl	`RtlInitUnicodeString`, `RtlCopyMemory`	Různé pomocné funkce pro práci s řetězci, bloky paměti a dalšími datovými strukturami.
Mm	`MmProbeAndLockPages`	Rutiny exportované správcem paměti.
Nt	`NtCreateEvent`, `NtOpenProcess`, `NtClose`, `NtShutdownSystem`, `NtTerminateProcess`	Tyto funkce mohou být volané z uživatelského režimu přes mechanismus systémových volání. Rutiny jádra s jinou předponou obyčejné aplikace volat nemohou.
Ob	`ObReferenceObjectByName`, `ObDereferenceObject`	Rutiny správce objektů.
Ps	`PsTerminateSystemThread`, `PsCreateSystemThread`	Exportováno správcem procesů.
Se	`SeAccessCheck`	Umožňují využít funkcí bezpečnostního modelu.
Zw	`ZwClose`, `ZwCreateKey`, `ZwMapViewOfSection`	Význam je téměř ekvivalentní rutinám s předponou Nt. Na rozdíl od nich jsou však Zw funkce určeny pro volání pouze z režimu jádra.

Subsystémy

Tabulka 2.2 ukazuje, že některé rutiny exekutivy jsou přes mechanismus systémových volání a nízkoúrovňovou knihovnu `ntdll.dll` přístupné i obyčejným programům běžícím v uživatelském režimu. Drtivá většina rutin exportovaných touto knihovnou DLL však není oficiálně dokumentovaná, a tudíž by je programátoři neměli používat, pokud nutně nemusejí. Tyto nativní funkce se totiž mohou měnit v závislosti na verzi Windows.

Nad `ntdll.dll` se nachází vrstva knihoven, která tvoří součást prostředí pro obyčejné aplikace – tzv. *subsystému*. Rozhraní exportované touto vrstvou je již dobře dokumentováno, což znamená, že i v budoucích verzích Windows bude pravděpodobně fungovat stejně a maximálně jej Microsoft rozšíří. Vývojáři jej mohou používat ve svých aplikacích. Windows obsahuje dva subsystémy – Windows a POSIX. Každý z nich dává programátorům aplikací k dispozici trochu odlišnou množinu funkcí obsažených v `ntdll.dll`. Například POSIX umožňuje vytvářet nové procesy pomocí rutiny `fork`, dobře známé z prostředí Unixu. Subsystém Windows takový způsob vytváření procesů neposkytuje. Nutnost implementovat POSIX také donutila vývojáře zavést do souborového systému NTFS hardlinky.

Subsystém POSIX

POSIX je zkratka z anglického sousloví *portable operating system interface based on Unix* (přenositelná systémová rozhraní založená na Unixu) a označuje sadu mezinárodních standardů, které popisují aplikační rozhraní v operačních systémech založených na Unixu. Pokud by výrobci operačních systémů tyto standardy implementovali, rozhraní různých operačních systémů by byla definována stejně a stejně by se také chovala, což by ušetřilo práci programátorům aplikací, kteří by neměli tolik problémů s přenositelností na jiné platformy.

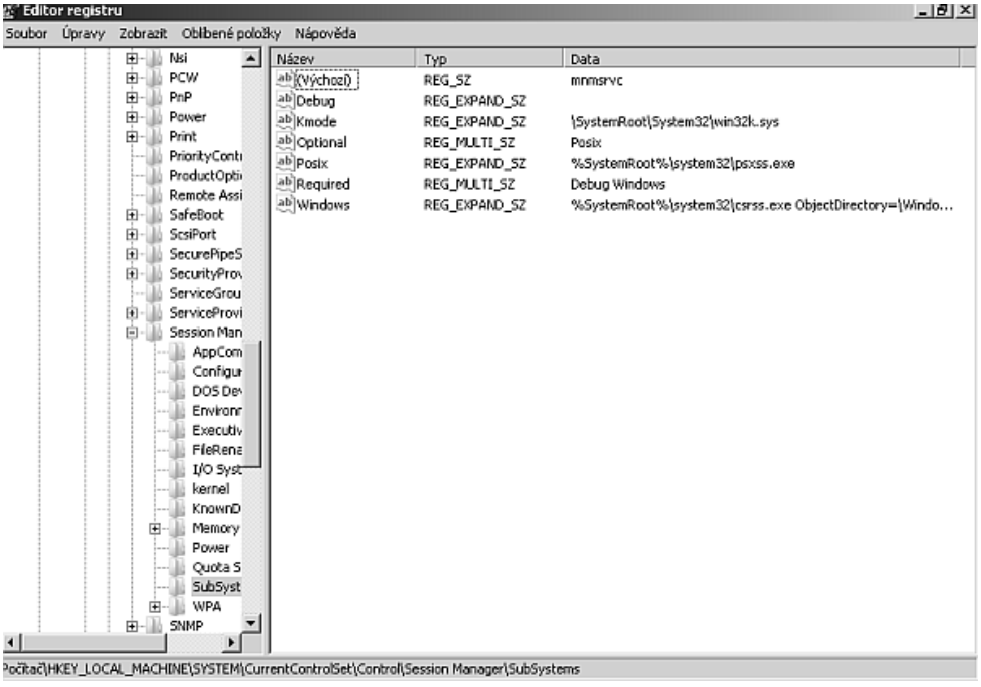
Z historických důvodů Windows původně ze všech těchto standardů implementovaly pouze POSIX 1. Windows Vista a Windows Server 2008 s sebou přináší implementaci standardů POSIX v podobně SUA (Subsystem for Unix–based Application – subsystém pro aplikace založené na operačním systému Unix). Tato rozšířená varianta subsystému POSIX implementuje kolem 2 000 funkcí různých unixových aplikačních rozhraní a obsahuje kolem tří set unixových programů. Subsystém SUA je dostupný na Windows Server 2008 a v Ultimate a Enterprise edicích Windows Vista.

Na rozdíl od subsystému Windows, bez kterého celý operační systém nemůže fungovat, POSIX se spouští jenom tehdy, pokud si uživatel přeje spustit proces určený pro tento subsystém. Konfigurace obou subsystémů je uložena v registru pod klíčem

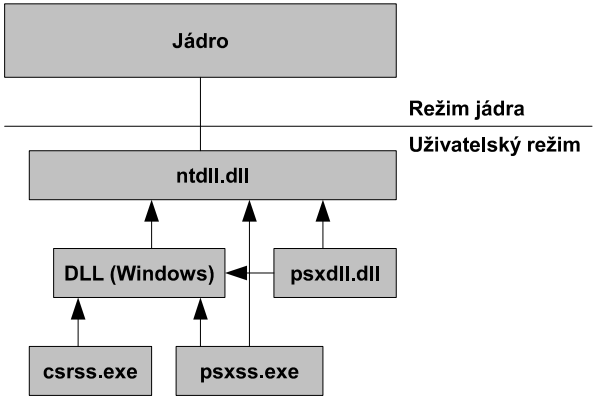
```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Subsystems
```

jehož obsah vidíte na obrázku 2.6. Z obrázku je patrné, že hlavním procesem subsystému POSIX je `psxss.exe`, který se nachází v systémovém adresáři. Také vidíte, že hlavní proces subsystému Windows nese název `csrss.exe` a že subsystémy mají i komponentu běžící v režimu jádra. Jedná se o ovladač `win32k.sys`, který se stará o správu grafického uživatelského rozhraní a grafiky vůbec.

Na tomto místě může vyvstat otázka, proč každý subsystém nemá svůj vlastní ovladač jádra. Důvodem je vysoká duplicita kódu. Vývojáři Windows se rozhodli tomuto nebezpečí předejít, a proto subsystém POSIX využívá funkcí (a knihoven DLL) poskytovaných subsystémem Windows včetně ovladače `win32k.sys`. Situaci znázorňuje obrázek 2.7. Kdyby tomu tak nebylo, musely by existovat dvě sady knihoven DLL a dva různé ovladače pro obsluhu grafického uživatelského rozhraní. Kód obou sad knihoven a ovladačů by však byl na mnoha místech totožný, což je v programování nežádoucí, protože dochází ke zvýšení rizika vzniku chyb.



Obrázek 2.6: Výpis klíče Session Manager\SubSystems



Obrázek 2.7: Závislost subsystému POSIX na subsystému Windows

Subsystém Windows

Struktura subsystému Windows je složitější než v případě POSIXu. Skládá se z následujících komponent:

- Hlavní proces subsystému `csrss.exe` a knihovny DLL, které používá.
- Ovladač jádra `win32k.sys` a drivery grafické karty a videa.

- Vrstva knihoven DLL zajišťující překlad volání dokumentovaných funkcí Windows API na volání nativních (a často nedokumentovaných) rutin z knihovny `ntdll.dll`, která je zodpovědná za volání jádra. Mezi tyto knihovny patří `kernel32.dll`, `user32.dll`, `gid32.dll` či `advapi32.dll`.

Mezi úkoly hlavního procesu subsystému patří vykreslování oken konzolových aplikací a podpora 16bitových programů určených původně pro operační systém MS–DOS. `Csrss.exe` také dostane oznámení, kdykoliv dojde k spuštění nového či ukončení již běžícího procesu nebo vlákna. Ve svém adresovém prostoru si uchovává vlastní kopii seznamu všech běžících procesů a vláken, aby pro ně mohl vyřizovat požadavky, jež mohou vzniknout při volání některých funkcí Windows API.

Tip: Informace o tom, že `Csrss.exe` ve své paměti uchovává vlastní kopii seznamu běžících procesů a vláken, byla zveřejněna na počátku roku 2009 na diskusním fóru Sysinternals (*http://forum.sysinternals.com*). Bezpečnostní programy zaměřené na hledání skrytých hrozeb v počítači (například skrytě běžících procesů) mohou této skutečnosti využít ve svůj prospěch a zvýšit svoji efektivitu. Existence kopie seznamu běžících procesů a vláken uvnitř `Csrss.exe` pravděpodobně nebyla příliš známa ani mezi autory malware a zejména rootkitů – programů, které se zaměřují na skrývání přítomnosti škodlivého softwaru v systému.

V dobách předcházejících Windows NT 4 do hlavního procesu subsystému patřily i moduly, které se starají o grafické uživatelské rozhraní a grafické kreslení vůbec. Vývojáři se tímto uspořádáním pravděpodobně chtěli přiblížit elegantní struktuře operačních systémů založených na mikrojádru. Taková struktura však vyžadovala velmi častá systémová volání a přepínání kontextu procesů. Ovladače grafické karty a obrazovky totiž běžely v režimu jádra, takže pokud nějaká aplikace chtěla například překreslit okno, bylo nutné přepnout kontext na hlavní proces subsystému, kde sídlil správce grafického uživatelského rozhraní, a následně se přepnout do režimu jádra, aby ten mohl doručit požadavek na překreslení ovladači grafické karty.

Systémová volání a přepínání kontextu mezi procesy patří mezi časově náročné operace, což mělo negativní dopad na výkon systému jako celku. Situace se výrazně nezlepšila ani po provedení mnoha optimalizací (byl upraven plánovač procesů a `csrss.exe` dokonce mohl číst některé oblasti paměti jádra, aby se ušetřilo kopírování). Vývojáři proto ve Windows NT 4 přesunuli správce grafického uživatelského rozhraní a ostatní grafické funkce do ovladače `win32k.sys`, čímž se snížil počet systémových volání a přepínání procesů. Jak již víte, jádro systému sdílí společný paměťový prostor a může komunikovat s ovladači grafické karty bez použití systémových volání.

Ovladač `win32k.sys` se tedy stará o grafické uživatelské rozhraní; spravuje okna, tlačítka, textová pole, obsluhuje myš a klávesnici. Umožňuje také kreslení různých grafických útvarů, jako jsou body, přímky či křivky. Ovladač se postará, aby se požadavky na vykreslování dostaly k správným ovladačům grafiky, tudíž aplikace, jež využívají GUI, vůbec nemusí (a většinou ani nepotřebují) vědět, jakou grafickou kartou počítač disponuje a jak se s ní zachází. `Win32k.sys` se chová podobně jako HAL – smazává rozdíly mezi různými kusy hardware.

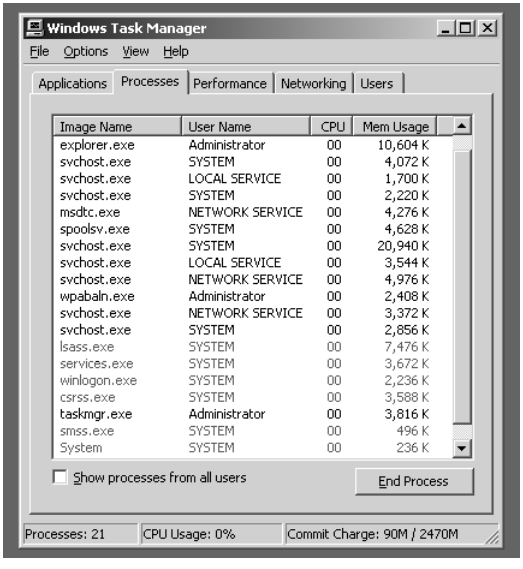
Bránu k ovladači `win32k.sys` tvoří knihovny `gdi32.dll` a `user32.dll`. První z nich obsahuje rutiny pro kreslení různých grafických útvarů (body, čáry, křivky) a druhá exportuje funkce pro práci s prvky uživatelského rozhraní, jako jsou okna, menu či ikony.

Mezi velmi důležité knihovny DLL také patří `kernel32.dll`, jenž exportuje vybrané části exekutivy (práce s procesy a vlákny, správa paměti, synchronizace), a `advapi32.dll`, která obsahuje rozhraní pro práci se službami a bezpečnostním modelem.

Systémové procesy

Windows patří mezi monolitické operační systémy, které se vyznačují velmi velkým a složitým jádrem, jež disponuje řadou pokročilých funkcí. Přesto běh systému závisí na několika procesech, jejichž ukončení (ať je způsobeno úmyslně, nebo v důsledku softwarové chyby) znamená restartování počítače. Tyto kritické procesy se obvykle nazývají jako *systémové*.

Seznam právě běžících procesů můžete vidět v programu Správce úloh, který spustíte například pomocí známé klávesové zkratky Ctrl+Alt+Del. Na obrázku 2.8 vidíte okno programu. Systémové procesy, o kterých pojednávají další odstavce, jsou vyznačeny šedou barvou.



Obrázek 2.8: Správce úloh – běžící procesy

Tip: Správce úloh vám nedovolí systémové procesy násilně ukončit. Pokud se o to pokusíte, program zareaguje hláškou „Přístup odepřen". Zajímavá je zde metoda, jakou Správce úloh rozhoduje, zda cílový proces je systémový. Až do Windows Vista totiž nejsou tyto procesy chráněny žádným bezpečnostním mechanismem, který by bránil jinému procesu, jenž disponuje administrátorským oprávněním, je ukončit. Správce úloh má v sobě „natvrdo" zakódovány názvy procesů, které považuje za systémové. Tohoto faktu mohou snadno zneužít tvůrci malware. Zkuste si například pojmenovat nějaký program `winlogon.exe`, spustit jej a následně se jej pokuste násilně ukončit pomocí Správce úloh.

Nečinné procesy (System Idle Processes)

Jedná se pouze o pseudoproces, který nevykonává žádnou činnost v uživatelském režimu. Na disku nenajdete žádný soubor, který by obsahoval jeho kód a data, ani žádné knihovny DLL, jenž by využíval. Úkolem Nečinných procesů je spotřebovávat čas procesoru, když žádná součást operačního systému ani žádná aplikace nemá co na práci. Jádro vytvoří tento proces během raných fází inicializace operačního systému. PID Nečinných procesů je vždy roven nule.

Proces System

Proces System také nevykonává žádný kód v uživatelském režimu, a tudíž nepoužívá žádné knihovny DLL a nenajdete žádný soubor s příponou `.exe`, který by jej reprezentoval. Už ale nejde čistě o pseudoproces; v jeho kontextu běží skupina vláken známá pod označením *pracovní vlákna* (*worker threads*). Jádro tato vlákna vytvoří během bootovacího procesu a jejich úkolem je vykonávat činnosti, jenž jim někdo zadá. Pracovní vlákna stráví většinu času čekáním, až jim nějaká součást jádra (nebo ovladač) určí, co mají vykonat. Jakmile práci dokončí, čekají na další zadání.

Poznámka: Kontext procesu System je také vhodné využít, pokud ovladač potřebuje pracovat s objekty jádra přes handle. Před vydáním Windows 2000 nebylo možné chránit ovladačem vytvořená handle k objektům před zneužitím procesem, v jehož kontextu byla vytvořena. Protože v kontextu procesu System není vykonáván žádný uživatelský kód a tento proces je přístupný pouze aplikacím s velmi vysokým oprávněním (většinou se jedná o služby), k zneužití handle vytvořených v kontextu tohoto procesu dojít nemůže. Více o principech kolem handle a objektů jádra se dočtete v kapitole 6.

Jádro využije služeb pracovních vláken ve chvíli, kdy potřebuje vykonat úkol, který je časově náročný (a tudíž by mohl brzdit výkon systému, kdyby nebyl proveden asynchronně), nebo nemůže být splněn za aktuálních podmínek. V takovém případě jádro předá úkol na bedra této speciální skupině vláken. Vlákna většinou dokončují zpracování požadavků od hardware, které nebylo možné provést při obsluze přerušení. Dále je využívá například správce paměti pro zapisování „špinavých" (dirty) stránek z vyrovnávací paměti na disk.

Poznámka: Zde můžete namítat, že každá součást jádra a každý ovladač si mohou vytvořit vlastní vlákna, aby asynchronně vykonávala výše popsané úlohy. Pracovní vlákna se tak mohou jevit jako zbytečná práce pro vývojáře Windows. Díky nim však mají programátoři o něco lehčí práci a ovladače mají jednodušší strukturu. A jednodušší struktura ve výsledku znamená méně chyb. Navíc, kdyby si každý modul jádra vytvářel vlastní skupinu pracovních vláken, celkový počet takových entit by mohl dosahovat daleko větších čísel než současný počet pracovních vláken. Taková vlákna by strávila většinu svého času čekáním na zadání práce a zbytečně by zabírala operační paměť.

Spolu s Nečinnými procesy je System jediným procesem s pevně daným číslem PID. Toto číslo má v jeho případě od Windows 2000 hodnotu 4. V předcházejících verzích neslo hodnotu 1.

Správce relací (Session Manager, Smss.exe)

`Smss.exe` je spuštěn v poslední fázi startu jádra a jedná se o první proces, který vykonává kód v uživatelském režimu. Provede poslední fázi inicializace operačního systému, který je pak připraven na přihlášení uživatelů.

Jak název napovídá, hlavním úkolem tohoto procesu je vytváření *relací*. Relaci si můžete představit jako ohradu, ze které ten, kdo je v ní uzavřen, nevidí ven a zároveň nikdo nevidí dovnitř. Tyto ohrady se využívají pro oddělení prostoru jednotlivých uživatelů. Díky nim například každý uživatel může mít namapované disky pod jinými písmeny. Jednotlivé relace se označují číslem.

`Smss.exe` nejprve vytvoří relaci 0, která se též označuje jako *konzolová relace*. V rámci ní běží systémové procesy, služby a všechny procesy uživatele, který se přihlásí jako první. Pro každého dalšího uživatele vytvoří správce další relaci.

Pro každou novou relaci správce spustí jednu kopii procesu `winlogon.exe`. `Smss.exe` je též zodpovědný za inicializaci hlavního procesu subsystému Windows – `csrss.exe`. Pro relaci 0 se místo Winlogonu spouští `wininit.exe`.

Jakmile dokončí svoji práci, správce relací navždy hlídá hlavní proces subsystému Windows a všechny kopie programu `winlogon.exe`. Pokud zjistí, že některý z těchto procesů byl neočekávaně ukončen, vyvolá modrou obrazovku smrti s kódem STATUS_CRITICAL_SYSTEM_PROCESS_DIED

Procesy podílející se na přihlašování uživatele (winlogon.exe, LSASS.exe)

Úkolem procesu `winlogon.exe` je umožnit uživateli přihlášení pomocí grafického uživatelského rozhraní. Může autentizovat uživatele různými způsoby, nejčastěji se ale stále používá zadání uživatelského jména (login) a hesla. Jednotlivé metody autentizace jsou implementovány v oddělených knihovnách DLL.

Winlogon v sobě má zabudovanou jednoduchou metodu ochrany proti programům, které se snaží zachytit údaje, jež uživatel zadal při pokusu o přihlášení. Většinou se jedná o tzv. *keyloggery* – programy, které zaznamenávají stisky jednotlivých kláves. Aby takový program nemohl přesně určit, kdy uživatel zadává přihlašovací údaje, může být Winlogon nastaven takovým způsobem, že před zobrazením přihlašovací obrazovky (viz obrázek 2.9) je třeba stisknout speciální kombinaci kláves. Obrazovku, která uživatele vyzývá k stisku této kombinace kláves, vidíte na obrázku 2.10; její podoba se může mírně lišit v závislosti na verzi operačního systému.

Obrázek 2.9: Přihlašovací obrazovka

Obrázek 2.10: Výzva pro stisknutí aktivační kombinace kláves

Jako výchozí kombinace je nastaven známý „trojhmat" Ctrl+Alt+Del. Trik spočívá v tom, že jakmile Winlogon zjistí, že tato kombinace byla stisknuta, tuto informaci pohltí a zobrazí okno, kam uživatel vyplní přihlašovací údaje. Přesněji, o stisku aktivační kombinace se dozví okno s názvem *SAS Window*, které je též zodpovědné za její pohlcení.

Pokud tedy keylogger neoperuje v samotném jádře systému nebo neinfiltroval do adresového prostoru procesu `winlogon.exe` (například v podobě knihovny DLL), nemůže stisk aktivační kombinace přímo detekovat, a tedy přesně zjistit, kdy uživatel zadává přihlašovací údaje.

Jakmile uživatel zadá login a heslo, Winlogon tyto údaje odešle procesu `lsass.exe`, kde proběhne jejich ověření. Pokud vše dopadne dobře (uživatel zadal správné jméno a heslo), `lsass.exe` zjistí, jakými oprávněními uživatel disponuje, a vytvoří tzv. *token*. Jedná se o objekt, kterým se pak uživatel prokazuje, potřebuje-li doložit, že má oprávnění k provedení určité operace.

Pokud má uživatel administrátorská práva a je-li zapnut mechanismus UAC (User Account Control), `lsass.exe` vytvoří tokeny dva. První z nich v sobě obsahuje všechna oprávnění, kterými uživatel disponuje. Druhý obsahuje jenom vybraná z nich. S pomocí druhého tokenu pak Winlogon vytvoří procesy, které provedou inicializaci prostředí. Token s omezenými právy se použije, kdykoliv uživatel spustí nějakou aplikaci, jež nevyžaduje administrátorská práva.

Pokud se uživatel pokusí spustit program vyžadující oprávnění administrátora, UAC se může dotázat (záleží na nastavení systému), zda chce daný program, který vyžaduje administrátorská práva, a tudíž by mohl být pro systém nebezpečný, spustit. Pokud dá uživatel spuštění zelenou, použije se pro daný program token, který obsahuje všechna uživatelova oprávnění.

Obrázek 2.11: Dialog programu Winlogon

Winlogon zůstává aktivní i po úspěšném přihlášení uživatele. Kromě toho, že je jeho povinností zajistit i odhlášení, stále čeká na onu aktivační klávesovou kombinaci a v případě, že zjistí její stisknutí, zobrazí dialog, který vidíte na obrázku 2.11. Winlogon také může být nakonfigurován tak, že spustí rovnou Správce úloh.

Jakmile Winlogon obdrží tokeny od LSASS, může být zahájena inicializace pracovního prostředí uživatele. Tato činnost již nespadá do pravomocí tohoto procesu, ale do kompetence procesů, jejichž názvy jsou uvedeny v hodnotě `Userinit` v klíči

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon.`

Winlogon přečte obsah této hodnoty a pokusí se všechny nalezené procesy spustit.

Upozornění: Jediný proces, který by se měl za všech okolností starat o inicializaci pracovního prostředí uživatele, je `userinit.exe` v systémovém adresáři. Pokud hodnota obsahuje i názvy dalších procesů, jedná se pravděpodobně o malware.

Inicializace pracovního prostředí spočívá v podstatě jen ve spuštění startovacích skriptů a předání řízení shellu, jehož název `userinit.exe` nalezne pod hodnotou `Shell` ve stejném klíči, jako Winlogon hledal hodnotu `Userinit`. Výchozím shellem pro všechny uživatele je Průzkumník Windows (`explorer.exe`), který zodpovídá za zobrazení pracovní plochy a umožňuje uživateli mimo jiné procházet adresářovou strukturu a spouštět další programy.

Procesy pro podporu služeb (services.exe, svchost.exe)

Mnoho součástí Windows je implementováno jako služby – programy běžící na pozadí, které ve většině případů nepotřebují (a ani nevyhledávají) interakci s uživatelem a jejichž běh nezávisí na tom, zda je přihlášen či nikoliv.

Poznámka: Odhlášení uživatele znamená automatické ukončení všech procesů, které běží pod jeho účtem.

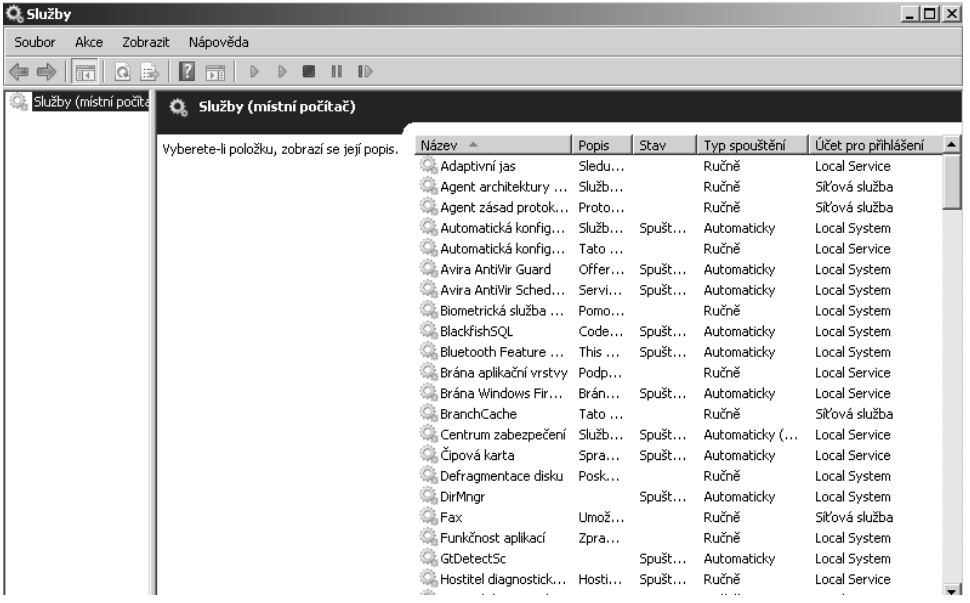
Interně systém Windows mezi služby řadí i ovladače jádra, ačkoliv ty nepatří mezi entity, o které se správce služeb (Service Control Manager – SCM), reprezentovaný procesem `services.exe`, příliš stará. Jeho hlavním úkolem je instalovat, spouštět a zastavovat služby určené pro běh v uživatelském režimu a posílat jim další druhy příkazů.

Službu tvoří obyčejný spustitelný soubor formátu PE s příponou `.exe`, který navíc obsahuje speciální kód pro komunikaci se správcem. Na rozdíl od normálních aplikací, více služeb může sdílet virtuální adresový prostor jednoho procesu. Vše záleží na nastavení konkrétní entity. Kód služeb, které nemají vlastní proces, je vykonáván v kontextu instancí procesu `svchost.exe`, který je pro jejich „hostování“ přímo určen.

Pro pohodlnou kontrolu služeb je připravena aplikace `services.msc`, která nejen zobrazí seznam všech nainstalovaných služeb a jejich aktuální stav, ale umožňuji s nimi i manipulovat. Uživatelské rozhraní tohoto programu vidíte na obrázku 2.12.

Konfigurace všech služeb se nachází v registru pod klíčem

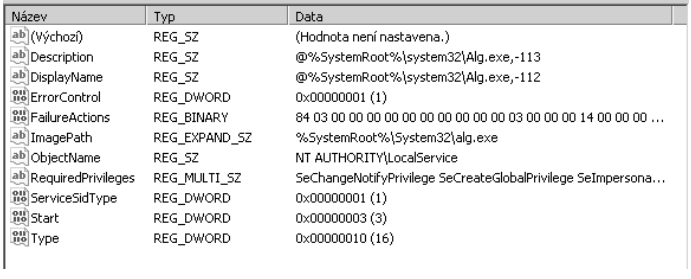
`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services.`



Obrázek 2.12: Rozhraní snap-inu services.msc

Každá nainstalovaná služba má zde vyhrazený jeden podklíč. Ten nese její interní jméno a může obsahovat další klíče s informacemi o jejím nastavení. Na obrázku 2.13 vidíte příklad podklíče pro službu s interním jménem ALG. Mezi nejdůležitější hodnoty patří:

- `Description`
- `DisplayName`
- `ErrorControl`
- `ImagePath`
- `Start`
- `Type`



Obrázek 2.13: Ukázka klíče s nastavením služby

Hodnoty `Description` a `DisplayName` mají pouze informativní charakter. Jejich obsah zobrazuje snap-in `services.msc` ve sloupcích **Název** a **Popis**. Pokud je hodnota `DisplayName` prázdná, nebo vůbec neexistuje, utilita zobrazí ve sloupci **Název** interní jméno.

Položka `ErrorControl` určuje, jak se má správce zachovat, když při spouštění služby dojde k chybě. Možné hodnoty a popis chování správce naleznete v tabulce 2.2.

Tabulka 2.2: Význam hodnot ErrorControl

Hodnota	Konstanta	Význam pro SCM
3	SERVICE_ERROR_CRITICAL	Chyba se zaznamená do systémového protokolu. SCM restartuje počítač a pokusí se postupovat podle poslední známé dobré konfigurace (last known good state). Pokud k chybě dojde i během tohoto pokusu, správce ohlásí chybu a nebude pokračovat v načítání dalších služeb.
0	SERVICE_ERROR_IGNORE	Chyba služby je úplně ignorována. SCM nezapíše nic ani do systémového ptotokolu.
1	SERVICE_ERROR_NORMAL	Správce služeb zapíše informaci o chybě do systémového protokolu, ale dál pokračuje ve své činnosti.
2	SERVICE_ERROR_SEVERE	SCM zapíše zprávu o chybě do systémového protokolu. Pokud k chybě došlo během obnovování z poslední známé dobré konfigurace, pokračuje ve spouštění dalších služeb. Jinak SCM restartuje počítač a pokusí se obnovit nastavení z poslední známé dobré konfigurace.

Hodnota `ImagePath` v sobě uchovává název souboru služby. Pokud není hodnota přítomna, SCM předpokládá, že soubor služby se nachází v systémovém adresáři a jmenuje se `X.exe`, nebo `X.sys`, kde `X` je interní název služby.

Podle hodnoty `Start` se SCM rozhodne, kdy může být služba spuštěna. Tabulka 2.3 popisuje hodnoty, kterých tato položka může nabývat.

Tabulka 2.3: Význam hodnot položky Start

Hodnota	Konstanta	Popis
0	SERVICE_BOOT_START	Platí pouze pro ovladače jádra. Příslušný ovladač je načten do jádra během rané fáze inicializace.
1	SERVICE_SYSTEM_START	Platí pouze pro ovladače jádra. Systém načte služby s touto hodnotou položky `Start` během volání funkce `IoInitSystem`, která se nachází v hlavním modulu jádra.
2	SERVICE_AUTO_START	Služba je spuštěna správcem služeb během jeho inicializace. To se děje krátce před vyzváním uživatele, aby zadal své přihlašovací údaje.
3	SERVICE_DEMAND_START	Služba není spuštěna automaticky, ale libovolný program může o její dodatečné spuštění požádat správce služeb. Platí jak pro služby běžící v uživatelském režimu, tak pro ovladače.
4	SERVICE_DISABLED	Službu nelze spustit. Platí i pro ovladače.

Položka `Type` určuje, zda se jedná o službu běžící v uživatelském režimu, nebo o ovladač jádra. Možné hodnoty vidíte v tabulce 2.4.

Pro podklíče, které reprezentují ovladače jádra, platí stejná pravidla jako u služeb běžících v uživatelském režimu. Ovladače ale většinou nepotřebují položky `Description` nebo `DisplayName`.

Tabulka 2.4: Hodnoty položky Type a jejich význam

Hodnota	Konstanta	Popis
0x20	SERVICE_WIN32_SHARE_PROCESS	Služba může sdílet kontext procesu s jinými službami.
0x10	SERVICE_WIN32_OWN_PROCESS	SCM službu spustí jako proces. Kontext není sdílen s jinými službami.
0x1	SERVICE_KERNEL_DRIVER	Služba je ovladač jádra.
0x2	SERVICE_FILE_SYSTEM_DRIVER	Služba je ovladač souborového systému. Tuto hodnotu ve svém klíči mají například ovladače `fastfat.sys` a `ntfs.sys`.
0x100	SERVICE_INTERACTIVE_PROCESS	Tato hodnota se používá, pokud běží služba v uživatelském režimu – tedy ve vlastním procesu, nebo sdílí proces s dalšími službami – a může být logickým součtem přidána k některým z příslušných hodnot zmiňovaných výše. Použití této hodnoty instruuje správce, aby službě umožnil používat grafické uživatelské rozhraní.

Způsob a čas spuštění služby lze tedy ovlivnit hodnotou položky `Start` v jejím registrovém klíči. Toto nastavení lze zjemnit přiřazením služby do určité skupiny. Výčet všech dostupných skupin včetně pořadí, v jakém budou procházeny při spouštění jednotlivých služeb, se nachází v klíči

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder`.

Pro ovladače je možné určit i pořadí spouštění jednotlivých entit ve skupině. Slouží k tomu hodnota `Tag`. Pořadí spouštění jednotlivých entit v jedné skupině na základě jejich tagu je uloženo v klíči

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GroupOrderList`

Některé služby mohou svoji činnost vykonávat pouze za předpokladu, že běží služby jiné. Například služba *Zvuk systému Windows* je závislá na službách *Koncové vytváření služby Windows Audio*, *Vzdálené volání procedur* a *Služba Plánovač multimédií*. Pokud nějaká z nich není aktivní, nelze spustit ani *Zvuk Systému Windows*.

Služby mohou být závislé nejen na jiných službách, ale i na celých skupinách. Systém může spustit službu závislou na skupině, pokud je alespoň jedna entita z této skupiny aktivní.

Programátoři mají pro práci se službami k dispozici sadu funkcí z knihovny `advapi32.dll`. Tyto rutiny jsou napojeny na proces `services.exe`. Nejdůležitější z nich naleznete v tabulce 2.5.

Tabulka 2.5: Některé funkce Windows API pro práci se službami

Název funkce	Popis
`OpenSCManagerW`	Správce si seznam služeb reprezentuje v databázi, která je uložena v registru. Aby libovolný program mohl s touto databází pracovat, musí k ní získat přístup. A k tomu slouží tato funkce.
`CreateServiceW`	Nainstaluje novou službu do databáze SCM. Umožňuje nastavit všechny potřebné parametry jako jméno, popis, podmínky spuštění a uživatelský účet, pod kterém nová služba poběží. Služby standardně běží pod účtem LocalSystem.
`OpenServiceW`	Pokud chce aplikace pracovat s již nainstalovanou službou, musí k ní získat přístup. Tato akce se provádí právě přes funkci `OpenService`.
`StartService`	Spustí zadanou službu nebo načte ovladač do jádra.
`DeleteService`	Odinstaluje službu z databáze. Správce smaže všechny záznamy v registru. Vymazání z databáze však neznamená ukončení běhu služby, pokud je aktivní.
`ControlService`	Umožňuje zasílat službám zprávy včetně signálu na ukončení činnosti.
`EnumServiceStatusW`	Zjistí informace o nainstalovaných službách.
`CloseServiceHandle`	Když program získává přístup k databázi správce služeb či přímo k jednotlivým entitám, dochází k alokaci systémových prostředků. Tyto prostředky je potřeba po ukončení práce s příslušnou entitou opět uvolnit, což provede právě tato funkce.

Vývoj ovladačů jádra

Cílem této knihy není pouze poskytnout čtenářům teoretické informace o vnitřním uspořádání a fungování jádra operačních systémů rodiny NT, ale také ukázat, jak těchto znalostí prakticky využít. Z tohoto důvodu v sobě některé kapitoly zahrnují i ukázkové zdrojové kódy ovladačů jádra.

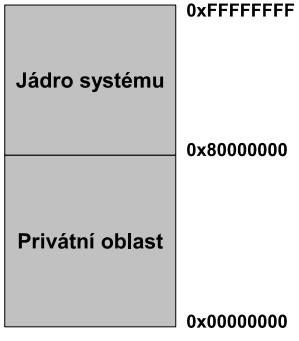
Tato kapitola si klade za cíl poskytnout takové informační pozadí, abyste ukázkovým zdrojovým kódům rozuměli, uměli si je přeložit do binární podoby a vyzkoušet. Dozvíte se podrobnější informace o tom, jak se ovladače liší od běžných aplikací, jaké prostředí pro jejich programování Microsoft nabízí a jak probíhá komunikace mezi ovladačem a kódem běžícím v uživatelském režimu.

Tato kapitola podává informace neformálním způsobem. Širší souvislosti se dozvíte v dalších částech knihy.

V závěru naleznete ukázkový zdrojový kód jednoduchého ovladače jádra, který demonstruje praktickou aplikaci znalostí, jež kapitola obsahuje.

Co je to ovladač

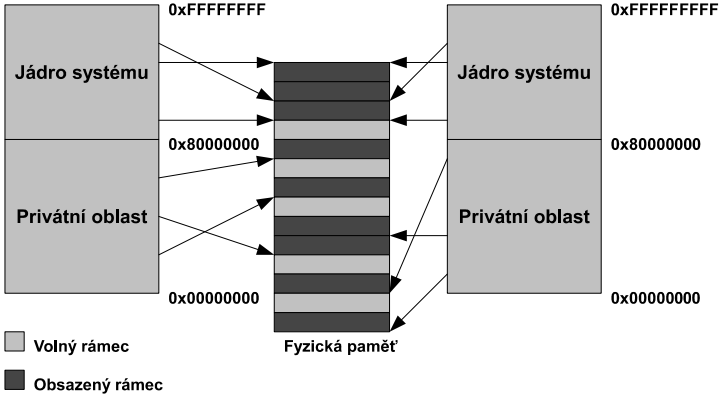
Již víte, že ovladače se na disku nacházejí ve spustitelných souborech formátu PE, většinou s příponou `.sys`. Na rozdíl od běžných aplikací, kód ovladačů není vykonáván v separátních virtuálních adresových prostorech, ale sdílí prostor společně s jádrem operačního systému. Přesněji, tento adresový prostor je namapován do prostoru každého běžícího procesu. Situaci vidíte na obrázku 3.1.



Obrázek 3.1: Zjednodušená struktura virtuálního adresového prostoru procesu

Privátní část adresového prostoru procesu (tedy oblast, kam ostatní procesy nemají přímý přístup) se nachází mezi adresami 0x00000000 a 0x7FFFFFFF. Na adresy 2 GB a vyšší je do všech

adresových prostorů namapována oblast jádra, která však z uživatelského režimu není přístupná. Naopak privátní části adresových prostorů z režimu jádra přístupné jsou. Obrázek 3.2 ukazuje mapování několika virtuálních adresových prostorů do fyzické paměti. Vidíte na něm, že horní část adresových prostorů je pokaždé namapována na stejné fyzické adresy.



Obrázek 3.2: Mapování několika virtuálních adresových prostorů do fyzické paměti

Upozornění: Zde načrtnutá struktura adresového prostoru procesu platí pro většinu 32bitových konfigurací a má pouze ilustrační charakter. Na některých konfiguracích je počátek prostoru jádra posunut až na adresu 3 GB (0xC0000000) a u 64bitových verzí operačního systému vypadá struktura celého prostoru úplně jinak. Bližší informace naleznete v kapitole 9, která se věnuje správě paměti.

Ovladače tedy nemají k dispozici žádný speciální kontext procesu, kterému by systém předával řízení, když by potřeboval vykonat jejich kód. Ten vykonává vlákno, které je zrovna naplánováno na procesoru. Jejich kód a data (proměnné) jsou namapovány ve virtuálním adresovém prostoru každého procesu na stejném místě. Z tohoto důvodu na konkrétním kontextu procesu (a vlákna) nezáleží.

Existuje však několik pravidel, pomocí kterých dokážete určit, v kterém kontextu kód ovladače poběží:

- Přechod z uživatelského režimu do režimu jádra v rámci systémového volání nemění kontext vlákna ani procesu.
- Kód vykonávaný některým z pracovních vláken (jedná se většinou o asynchronní zpracování odpovědi hardware na dřívější požadavek) běží vždy v kontextu procesu System.
- Obsluha výjimek a přerušení probíhá v kontextu vlákna, které se nacházelo na procesoru v okamžiku, kdy k příslušné události došlo.

Ovladačům většinou nezáleží na tom, v kontextu jakého procesu se jejich kód vykonává. Ve výjimečných případech se však mohou přepnout do libovolného běžícího procesu. Slouží k tomu funkce `KeAttachProcess`, `KeStackAttachProcess`, `KeUnstackDetachProcess` a `KeDetachProcess` exportované hlavním modulem jádra (`ntoskrnl.exe`). Tyto rutiny mění pouze kontext procesu; aktuální vlákno si při volání `KeAttachProcess` či `KeStackAttachProcess` „přivlastní“ zadaný proces

a ke svému pravému vlastníkovi se vlákno vrátí přes `KeDetachProcess` či `KeUnstackDetach Process`.

`KeAttachProcess` se od své zásobníkové varianty (`KeStackAttachProcess`) liší v tom, že pokud vlákno „navštívilo" cizí proces, musí se před návštěvou dalšího cizího procesu vrátit do kontextu svého vlastníka. Jinými slovy, mezi dvěma voláními `KeAttachProcess` se musí nacházet volání `KeDetachProcess`. Pro druhou dvojici funkcí toto neplatí. Volání však musí být párová – na každé volání `KeAttachProcess` (resp. `KeStackAttachProcess`) musí existovat volání `KeDetachProcess` (resp. `KeUnstackDetachProcess`).

Poznámka: Jak uvádí druhá kapitola, pro ochranu prostředí ovladačů slouží proces System. Jak ale vyplývá z odrážek výše, ovladače nemohou běžet pouze v kontextu tohoto procesu.

Další rozdíl mezi ovladači a běžnými aplikacemi skrývá práce s pamětí. Aplikace vidí všechnu paměť virtuálně – přistupují na různé virtuální adresy a je na operačním systému, aby zajistil existenci potřebných mapování do fyzické paměti. Pokud například program přistupuje do stránky, která se nachází ve stránkovacím souboru na disku, systém tuto stránku nahraje do fyzické paměti a vytvoří potřebné mapování. Tyto operace „v zákulisí" jsou pro aplikaci (až na případné malé časové zpoždění) neviditelné.

Ovladače také pracují téměř výhradně s virtuální pamětí. Jejich programátoři si ale musí uvědomovat, co se za virtuálními adresami skrývá. Za určitých okolností si totiž výpadek stránky (nastává, pokud požadovaná stránka virtuální paměti není mapována do fyzické paměti) nelze dovolit.

K takové situaci dochází například při obsluze přerušení. Představte si, že při obsluze přerušení disku dojde k výpadku stránky. Procesor vyvolá nové přerušení s číslem 0xE, které informuje systém, že nastal výpadek stránky. Operační systém se pokusí chybějící stránku nahrát ze stránkovacího souboru. Protože se stránkovací soubor nachází na pevném disku, jádro toto zařízení požádá o načtení příslušných dat. Jakmile disk splní požadavek, vyvolá další přerušení. Nezapomeňte, že v našem příkladu došlo k výpadku stránky právě při obsluze přerušení disku.

Takovýmto způsobem by mohlo snadno dojít k zatuhnutí celého systému. Proto Windows obsahují bezpečnostní opatření, která jejich běh preventivně ukončí, pokud nějaký ovladač udělá akci, jejíž provedení by za daných podmínek mohlo ohrozit chod celého systému a způsobit i poškození hardware. Takovou akcí je třeba výpadek stránky při obsluze drtivé většiny přerušení.

Z předchozích odstavců vyplývá, že za určitých okolností si programátor ovladače musí být jist, že k výpadu stránky `nemůže` dojít. Proto ovladače mohou alokovat a používat paměť dvou druhů:

- *Paměť stránkovaného fondu (PagedPool)* sestává ze stránek virtuální paměti, které mohou být uloženy na disk do stránkovacího souboru. Tento druh paměti se tedy chová úplně stejně jako privátní část adresového prostoru procesu uživatelského režimu.
- *Paměť nestránkovaného fondu (NonPagedPool)* sestává ze stránek virtuální paměti, které operační systém za žádných okolností neodloží do stránkovacího souboru na disk. Při přístupu na tyto adresy k výpadku stránky *nemůže* dojít. Paměť z nestránkovaného fondu byste měli používat s rozvahou. Využití velkého množství fyzické paměti pro účely tohoto fondu může negativně ovlivnit výkon celého systému, protože narušuje mechanismy virtuální paměti – ubírá fyzickou paměť, kterou je možné využít k načtení bloků ze stránkovacího souboru.

Upozornění: Výpadek stránky nastává, kdykoliv procesor není schopen určit, na jakou fyzickou adresu má požadovanou virtuální adresu přeložit. Tedy i v případě přístupu na adresu, na které není alokována žádná paměť. Před takovými výpadky stránky vás neuchrání ani nestránkovaný fond. Přístup na neplatnou virtuální adresu svědčí o tom, že někde v programu je chyba. U odladěného kódu by k takovému chování docházet nemělo. Pokud se váš ovladač chová správně, k výpadku stránky při přístupu do oblasti nestránkovaného fondu dojít opravdu nemůže.

Tip: Windows umožňují alokovat paměť i z dalších oblastí. Dokumentovány jsou fondy `PagedPoolMustSucceed` a `NonPagedPoolMustSucceed`, pro které systém vyhrazuje velmi malou oblast virtuální paměti (například 64 kilobajtů). Charakteristickým znakem těchto fondů je, že každá alokace musí proběhnout úspěšně. Pokud se tak nestane – například proto, že ve fondu není dostatek volné paměti – běh systému skončí modrou obrazovkou smrti. Ovladače a jádro by měly používat paměť z těchto zdrojů, pouze pokud je neúspěch alokace neslučitelný s dalším během systému.

Posledním velkým rozdílem mezi ovladači a aplikacemi běžícími v uživatelském režimu je moc, kterou nad systémem zástupci těchto kategorií mají. Na rozdíl od normálního programu ovladač může:

- Přímo komunikovat s periferními zařízeními.
- Ovlivňovat chování systému jako celku a dočasně narušovat základní mechanismy, mezi které patří plánování vláken na procesoru a fungování bezpečnostního modelu.

Na druhou stranu chyba v kódu ovladače způsobí pád celého operačního systému, kdežto chyba, kterou uděláte při programování běžné aplikace, většinou skončí násilným ukončením příslušného procesu ze strany Windows. Programování ovladačů je v tomto ohledu mnohem náročnější a dostupných nástrojů pro jejich pohodlné ladění také není mnoho.

Prostředí pro programování

Microsoft pro vývoj ovladačů poskytuje zdarma balík Windows Driver Kit (WDK), ve kterém najdete překladač, potřebnou dokumentaci a řadu užitečných nástrojů. WDK neobsahuje žádné sofistikované vývojové prostředí, ale pro kompilaci ovladačů z příkazové řádky, kterýžto způsob bude používán i v této knize, plně postačuje.

Ovladače můžete vytvářet i v příjemném prostředí Microsoft Visual Studia. I pak budete ale potřebovat WDK kvůli knihovnám a hlavičkovým souborům, kde jsou deklarovány konstanty a rutiny, jež ovladače jádra mohou používat.

Jak přeložit ovladač

Krom zdrojového kódu je k překladu potřeba dalších dvou souborů. Soubor MAKEFILE obsahuje obecné informace pro překladač a jeho obsah odkazuje na výchozí soubor s těmito údaji. Tento odkaz je zapsán následovně:

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Všechny informace týkající se přímo vašeho ovladače, jako například seznam souborů se zdrojovým kódem či seznam knihoven, se nachází v souboru SOURCES, jehož formát vidíte na výpisu 3.1.

Výpis 3.1: Formát souboru SOURCES

```
TARGETNAME=<jméno>
TARGETTYPE=<typ_výsledného_souboru>
SOURCES= <seznam_souborů_se_zdrojovým_kódem>
INCLUDES= <adresáře_s_hlavičkovými_soubory>
```

Jedná se v podstatě jen o přiřazení hodnot do proměnných prostředí, které se pak používají při překladu. Výpis 3.1 obsahuje pouze ty nejdůležitější z nich. Jejich význam a formát hodnot shrnuje tabulka 3.1.

Tabulka 3.1: Formát a význam několika základních proměnných v souboru SOURCES

Proměnná	Formát hodnoty	Význam pro překladač
TARGETNAME	Název výsledného souboru (bez přípony)	Udává název souboru (bez přípony), který linker vytvoří z přeloženého zdrojového kódu
TARGETTYPE	DRIVER, LIBRARY nebo PROGRAM	Určuje typ výsledného spustitelného souboru. Pro ovladače platí hodnota DRIVER a pro aplikace běžící v uživatelském režimu PROGRAM. Pokud chcete vytvořit knihovnu DLL, použijte hodnotu LIBRARY.
SOURCES	Seznam souborů. Oddělovačem je mezera. Pokud seznam zasahuje do více řádků, musí na konci každého z nich být zpětné lomítko „\“	Seznam všech souborů, které obsahují zdrojový kód ovladače.
INCLUDES	Seznam adresářů. Formát je stejný jako u proměnné SOURCES.	Seznam adresářů, ve kterých bude překladač vyhledávat hlavičkové soubory. Adresáře s hlavičkovými soubory WDK se prohledávají automaticky.

Spuštění prostředí překladače a následný překlad zdrojových kódů ovladače můžete provést následovně:

- V nabídce **Start** v sekci **Programy** (ve Windows Vista a Windows 7 se tato položka jmenuje **Všechny programy**) vyberte položku **Windows Driver Kits**.
- Zobrazí se seznam všech verzí WDK, které máte nainstalovány. Vyberte tu nejaktuálnější – v době psaní této kapitoly se jednalo o **WDK 7600.16385**.
- Prostředí překladače se skrývá pod volbou **Build Environments**.
- Nyní vyberte operační systém, pro který chcete ovladač zkompilovat. Novější verze OS umožňují ovladačům využívat nové funkce. Na druhou stranu i ovladač přeložený pro Windows 7 můžete bez problémů rozchodit na Windows XP, pokud nevyužívá specifik novější verze.

  Pokud chcete přeložit ovladač například pro Windows 7, vyberte položku **Windows 7**.
- Nyní je třeba vybrat architekturu procesoru. Překladač WDK dokáže kompilovat do instrukční sady procesorů x86, AMD64 (x64) a Ithanium (ia64). Architektura x86 implikuje 32bitový operační systém. Pokud chcete ovladač používat na 64bitových Windows, zvolte x64.

Dále se musíte rozhodnout, zda chcete svůj ovladač přeložit v prostředí tzv. `Free build`, nebo `Checked build`. Prostředí `Free build` je ekvivalentní modu `Release` z aplikace Microsoft Visual Studio; překladač provádí všechny dostupné optimalizace kódu a ignoruje makra jako `ASSERT`, `KdPrint` či `KdPrintEx`. Prostředí `Checked build` slouží pro testování. Překladač neprovádí optimalizace, které činí kód binárky méně čitelným pro disassemblery, a makra `ASSERT`, `KdPrint`, `KdPrintEx` a další provádějí svoji normální činnost: `ASSERT` v případě vyhodnocení zadaného výrazu na FALSE pošle zprávu debuggeru s informacemi o tom, na jakém řádku jakého souboru zdrojového kódu k selhání došlo, `KdPrint` a `KdPrintEx` se překládají na volání `DbgPrint` a `DbgPrintEx`. Prostředí dále definuje symbol DBG na hodnotu 1, čehož lze využít při podmíněném překladu.

> **Tip:** možnostem ladění ovladačů včetně použití maker `ASSERT`, `KdPrint`, `KdPrintEx`, `DbgPrint` a `DbgPrintEx` se věnuje část „Několik poznámek k ladění ovladačů" níže v této kapitole.

Pokud ovladač chcete testovat, použijte nastavení `Checked build`. Pokud si myslíte, že již může být nasazen do „ostrého provozu", přeložte jej pomocí `Free build`.

Rozhodnete-li se například pro architekturu x64 a chcete-li do ovladače přilinkovat i dodatečné ladicí informace, zvolte položku **x64 Checked Build Environment**.

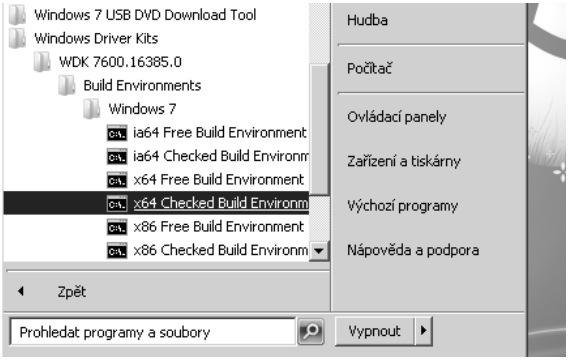
Dosavadní postup vidíte na obrázku 3.3.

- Spustí se konzole Příkazového řádku nastavená pro účely překladače WDK. Nyní se pomocí standardních příkazů přesuňte do adresáře, kde se nachází soubory MAKEFILE a SOURCES vašeho ovladače.
- Příkazem **build** zahájíte překlad a následné linkování. Překladač a linker vypisují průběžný stav do konzole. Chování příkazu lze ovlivnit mnoha parametry příkazové řádky. Některé možnosti ukazuje tabulka 3.2.

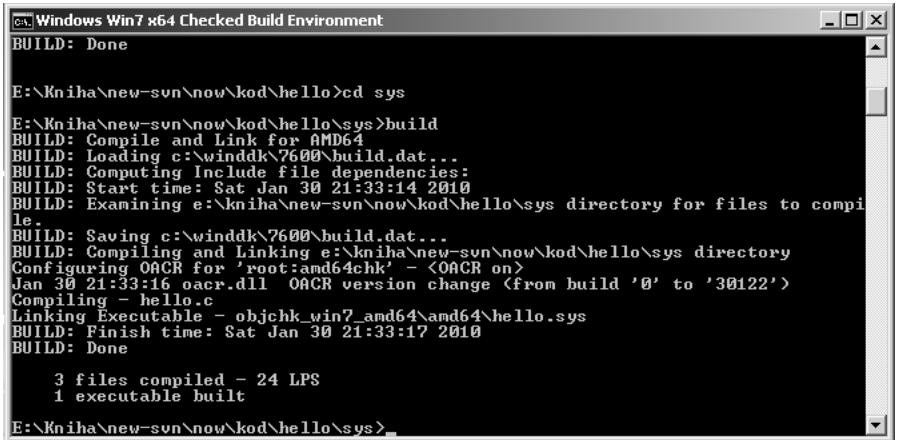
Tabulka 3.2: Ovlivnění chování příkazu build pomocí příkazové řádky

Příkaz	Popis
/c	Před zahájením překladu odstraní všechny existující objektové soubory.
/C	Před zahájením překladu odstraní soubory statických knihoven .lib.
/g	Při výpisu varování, chyb a výsledků překladu jednotlivé aspekty odlišuje barvami. Chyby jsou tisknuty červeně, varování žlutě a počet vytvořených spustitelných souborů zeleně.
/h	Nevypisuje podrobné informace o překladu do konzole.
/w	Do konzole jsou zobrazována i varování.
/T	Zobrazí úplný strom závislostí.
/$	Zobrazí úplný strom závislostí hierarchicky.

- Pokud překlad a linkování proběhly úspěšně, vytvoří se ve složce se souborem MAKEFILE nový adresář, v němž najdete i spustitelný soubor vašeho ovladače. Úspěšný překlad a linkování vidíte na obrázku 3.4.
- V případě, že se při překladu či linkování vyskytly problémy (viz obrázek 3.5), překladač (resp. linker) vypíše hlášení o neúspěchu a vytvoří dva nové textové soubory. Soubor s příponou `.wrn` obsahuje varování. Ta nebrání v úspěšném dokončení překladu a linkování, ale zvlášť při programování ovladačů je žádoucí, aby se žádná neobjevovala. Do souboru s příponou `.err` překladač (resp. linker) vypíše informace o nalezených chybách. Jedná se například o špatnou syntaxi jazykových konstrukcí či hlášení o použití neexistující proměnné či konstanty.

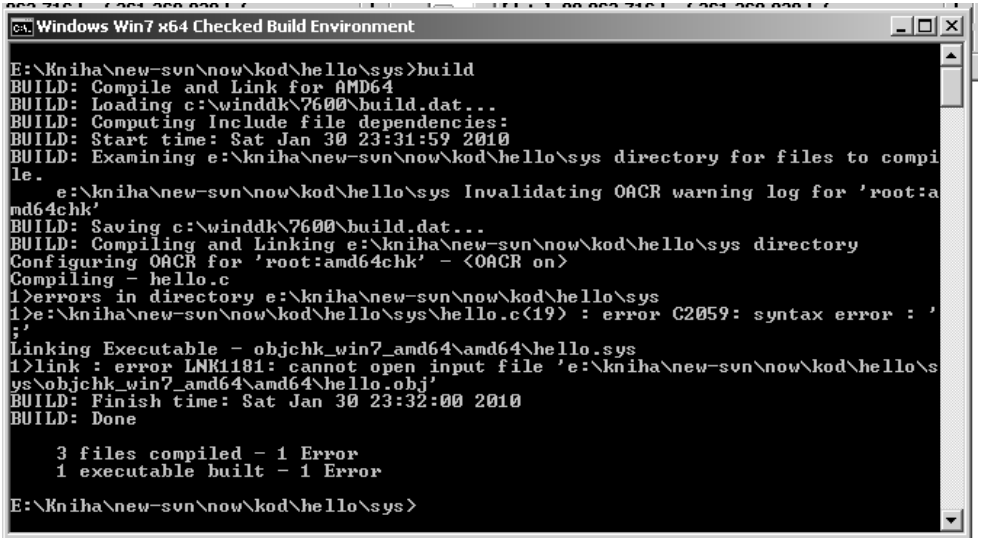


Obrázek 3.3: Výběr nastavení prostředí překladače WDK



Obrázek 3.4: Výstup při úspěšné kompilaci

Poznámka: Balík Windows Driver Kit se za dob, kdy nejnovější verze operačního systému nesla název Windows Server 2003, nesl jméno Driver Development Kit (DDK).



Obrázek 3.5: Během kompilace (linkování) byly nalezeny chyby

Tip: U ovladačů, které tvoří součást malware, se občas stává, že uniknou jejich verze určené pro ladění – tedy obsahují odkaz na soubor `.pdb` s ladicími informacemi. Tento odkaz je uložen ve formě absolutní cesty a může poskytnout cenné vodítko nejen k identifikaci typu malware (podle jména souboru ovladače to často není možné, protože může být generováno náhodně), ale i ke zjištění základních informací o autorovi. Vše závisí na tom, jaká jména autor volil pro složky, které se v odkazu nachází.

Načtení ovladače do jádra

Spustitelný soubor ovladače není možné (za běžného nastavení) načíst do jádra systému klasickým poklepáním, jak tomu je v případě běžné aplikace. Na rozdíl od spuštění programu se může jednat o relativně složitou operaci a existuje více cest, jak ji provést.

Čistý a oficiální způsob

Nejjednodušší způsob načtení ovladače do jádra spočívá ve vytvoření služby, která daný ovladač popisuje (pamatujte, Windows interně reprezentují ovladače jako služby) a jejím následném spuštění. Ukázkový kód vidíte na výpisech 3.2 a 3.3. Jedná se o techniku plně dokumentovanou a „zdvořilou" k operačnímu systému. Pravděpodobně nenajdete mnoho legitimních důvodů, proč ji nepoužít.

Pro vytvoření nové služby je nejprve nutné získat potřebná oprávnění k databázi služeb, kterou spravuje správce služeb (SCM). K tomuto účelu slouží funkce `OpenSCManagerW`, která jako své argumenty bere název počítače, k jehož databázi služeb chcete získat přístup, název konkrétní databáze a přístupová práva, která chcete získat. Pokud jako první dva parametry dostane hodnotu NULL, rutina se pokusí zajistit požadovaná oprávnění k právě používané databázi na lokálním počítači. Pro instalaci služby je nutné oprávnění SC_MANAGER_CREATE_SERVICE. Pokud se podaří

přístup získat, funkce vrátí handle databáze, kterým se bude proces prokazovat při volání dalších rutin pro komunikaci se SCM.

Po úspěšném získání oprávnění následuje vlastní vytvoření služby – volání funkce `CreateServiceW`. Tato rutina má mnoho parametrů, protože pokrývá veškeré možnosti nastavení všech typů služeb. Pro ovladače má význam druhý parametr (interní jméno), pátý parametr, který určuje typ služby (pro ovladač je rezervována hodnota SERVICE_KERNEL_DRIVER) a šestý a osmý parametr. Šestým parametrem ovlivňujete, kdy může být služba spuštěna. V ukázkovém kódu se předává hodnota SERVICE_DEMAND_START, která určuje, že ovladač popsaný touto službou bude načten do jádra „na požádání" libovolného programu s dostatečným oprávněním. Osmý parametr specifikuje jméno souboru ovladače.

Pokud volání `CreateServiceW` uspěje, službu ovladače se podařilo úspěšně nainstalovat. Nyní je třeba uklidit prostředky alokované jak při vlastní instalaci, tak při získávání přístupu k databázi služeb. Tento úkol patří rutině `CloseServiceHandle`, která je zodpovědná za uvolnění prostředků spojených s libovolným handle od služby (která vrací například `CreateServiceW`) či databáze služeb (které vzniká při úspěšném volání `OpenSCManagerW`).

Výpis 3.2: Vytvoření služby ovladače

```
BOOL scmInstallDriver (PWCHAR DriverName, PWCHAR FileName)
{
  BOOL ret = FALSE;
  SC_HANDLE hservice = NULL;

  hservice = CreateServiceW(hmanager_inst, DriverName, NULL,
SERVICE_ALL_ACCESS, SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START,
SERVICE_ERROR_NORMAL, FileName, NULL, NULL, NULL, NULL, NULL);
  ret = hservice!= NULL;
  if (ret)
    CloseServiceHandle(hservice);

  return ret;
}
```

Výpis 3.3 obsahuje zdrojový kód rutiny `scmLoadDriver`, která jako parametr vezme interní jméno ovladače a pokusí se jej načíst do jádra systému.

Pro načtení ovladače do jádra je třeba získat oprávnění pro spuštění služby, která jej reprezentuje. Aby mohla získat dané oprávnění, musí nejprve funkce `scmLoadDriver` získat přístup do databáze služeb. Tentokrát postačí oprávnění SC_MANAGER_CONNECT, které umožňuje pracovat s jednotlivými službami.

Po úspěšném připojení k databázi služeb se `scmLoadDriver` pokusí získat ke službě ovladače právo SERVICE_START pomocí volání `OpenServiceW`. Bez tohoto oprávnění není možné službu spustit.

Spuštění služby ovladače se projeví načtením příslušného souboru s příponou `.sys` do jádra. Samotnou akci provádí rutina `StartServiceW`. Tato funkce krom handle cílové služby bere i další dva argumenty, jež umožňují spouštěné entitě předat parametry obdobně jako se předávají parametry příkazového řádku při vytváření nového procesu. Ovladače tento mechanismus nepodporují, a proto na hodnotách těchto argumentů nezáleží.

Výpis 3.3: Načtení ovladače do jádra

```
BOOL scmLoadDriver (PWCHAR DriverName)
{
  SC_HANDLE hservice = NULL;
  BOOL ret = FALSE;

  hservice = OpenServiceW(hmanager_connect, DriverName, SERVICE_START);
  ret = hservice!= NULL;
  if (ret) {
    ret = StartServiceW(hservice, 0, NULL);
    CloseServiceHandle(hservice);
  }

  return ret;
}
```

Pokud již ovladač není v jádře potřeba, může být z paměti odstraněn. Postup, který demonstruje rutina `scmUnloadDriver` na výpisu 3.4 se velmi podobá předchozím; `scmUnloadDriver` nejprve získá přístup k databázi služeb, následně k službě se zadaným interním jménem a pošle jí řídící příkaz SERVICE_CONTROL_STOP, kterým požaduje okamžité zastavení činnosti. K vyslání tohoto příkazu je nutné získat k cílové entitě oprávnění SERVICE_STOP.

Výpis 3.4: Uvolnění ovladače z jádra

```
BOOL scmUnloadDriver (PWCHAR DriverName)
{
  SERVICE_STATUS ss;
  SC_HANDLE hservice = NULL;
  BOOL ret = FALSE;

  hservice = OpenServiceW(hmanager_connect, DriverName, SERVICE_STOP);
  ret = hservice!= NULL;
  if (ret) {
    ret = ControlService(hservice, SERVICE_CONTROL_STOP, &ss);
    CloseServiceHandle(hservice);
  }

  return ret;
}
```

Protože služby reprezentují vysoce privilegované aplikace a ovladače, není pro ně příkaz k zastavení činnosti závazný. Za určitých okolností by zastavení nějaké služby mohlo znamenat ohrožení pro celý systém. Jak se toto „odmítnutí“ implementuje u ovladačů, najdete na příkladu Hello World dále v této kapitole.

Odinstalování služby (funkce `scmUninstallDriver` na výpisu 3.5) spočívá pouze v získání dostatečných oprávnění k dané službě a k volání rutiny `DeleteService`.

Výpis 3.5: Smazání služby ovladače

```
BOOL scmUninstallDriver (PWCHAR DriverName)
{
  SC_HANDLE hservice = NULL;
  BOOL ret = FALSE;

  hservice = OpenServiceW(hmanager_connect, DriverName, DELETE);
  ret = hservice!= NULL;
  if (ret) {
    ret = DeleteService(hservice);
    CloseServiceHandle(hservice);
  }
  return ret;
}
```

Upozornění: Správce služeb umožňuje odstranit ze systému i službu, která je právě aktivní. Funkce `DeleteService` pouze smaže klíče registru, které vytvořilo volání `CreateService` při instalaci. Služba nedostane žádnou informaci o tom, že již v systému není nainstalována a může běžet až do restartu počítače.

Zdrojové kódy rutin uvedených na výpisech 3.2 až 3.5 naleznete na internetových stránkách knihy v projektu `drv` v souboru `scmDrivers.c`.

Web: www.jadro-windows.cz/projekty/drv

Méně známý způsob (nativní funkce NtLoadDriver)

Mezi méně známé způsoby načítání ovladače do jádra systému patří použití nativní funkce `NtLoadDriver`. Před seznámením s podrobnostmi této metody je třeba uvést několik základních fakt o nativních funkcích Windows API.

Již víte, že nativní funkce exportuje knihovna `ntdll.dll`, která má na starost komunikaci s jádrem. Na rozdíl od dokumentovaných rutin Windows API na vyšších vrstvách, které vrací nenulovou hodnotu v případě úspěchu a nulu, když daná operace selže, nativní funkce většinou pracují s návratovou hodnotou typu NTSTATUS. Jedná se o celé číslo bez znaménka, jehož význam vysvětluje tabulka 3.3.

Upozornění: Mnoho funkcí dostupných ovladačům jádra též vrací hodnotu NTSTATUS. Při zkoumání jejich významu si pečlivě přečtěte, jaké návratové kódy mohou vrátit. Rozdělení návratových hodnot může být ošidné. I když funkce vrátí hodnotu spadající do kategorie „úspěch“, nemusí to znamenat, že daná operace byla úspěšně provedena. Úspěch operace v obecném případě garantuje pouze návratová hodnota STATUS_SUCCESS.

Tabulka 3.3: Význam hodnot NTSTATUS

Hodnota či rozsah hodnot	Význam
0x00000000 – 0x3FFFFFFF (úspěch)	Operace byla úspěšně provedena. Ideální případ nastává při návratové hodnotě 0, které odpovídá konstanta STATUS_ SUCCESS.
0x40000000 – 0x7FFFFFFF (informace)	Během provádění operace došlo k události, která její úspěch významně neovlivnila, ale přesto stojí za povšimnutí.
0x80000000 – 0xBFFFFFFF (varování)	Během provádění operace se vyskytl problém, který nepřímo brání jejímu dokončení. To nastává například v případě, že výsledky operace nelze zapsat do bufferu předávaného v parametru volání, protože je příliš malý.
0xC0000001 – 0xFFFFFFFF (chyba)	Při pokusu o provedení operace došlo k chybě.

Poznámka: Pro pohodlnější testování, do které kategorie určitá hodnota NTSTATUS patří, můžete využít některá z následujících maker:

- NT_SUCCESS – vyhodnotí se jako TRUE, pokud zadaná hodnota patří do kategorie „úspěch" či „informace". V ostatních případech vrátí FALSE.
- NT_INFORMATION – nabývá hodnoty TRUE právě tehdy, když zadaná hodnota patří do kategorie „informace".
- NT_WARNING – pokud hodnota spadá do kategorie „varování", vrátí TRUE. V jiném případě vrací FALSE.
- NT_ERROR – vyhodnotí se na TRUE pouze v případě hodnoty z oblasti „chyba".

Další rozdíl mezi standardními rutinami Windows API a nativními funkcemi spočívá v práci s řetězci. Standardní Windows API pracuje s tzv. *nulou ukončenými řetězci*. Takové řetězce jsou tvořeny posloupností znaků ukončenou znakem s hodnotou nula a jejich výhoda spočívá v tom, že není nutné si explicitně pamatovat jejich délku (v případě potřeby se vypočítá jako rozdíl adresy počátku řetězce a adresy koncového nulového znaku). Nativní funkce pracují s řetězci reprezentovanými strukturami UNICODE_STRING a ANSI_STRING. Tyto struktury v sobě uchovávají nejenom obsah celého řetězce, ale i jeho délku v bajtech. Protože jádro Windows pracuje téměř výhradně s řetězci ve formátu Unicode, používají se v drtivé většině případů struktury UNICODE_STRING.

Načtení ovladače pomocí volání `NtLoadDriver` lze rozdělit do několika kroků. Nejprve je třeba v registru manuálně vytvořit klíč, který bude reprezentovat službu našeho ovladače a vyplnit hodnoty `Start`, `Type` a `ImagePath`. Tyto operace dělá funkce `ntldInstallDriver` z výpisu 3.6. Odinstalování ovladače spočívá ve smazání klíče služby z registru (rutina `ntldUninstallDriver`).

Výpis 3.6: Načtení a uvolnění ovladače z jádra

```
BOOL ntldInstallDriver (PWCHAR DriverName, PWCHAR FileName)
{
  DWORD start = SERVICE_DEMAND_START;
```

```
  DWORD type = SERVICE_KERNEL_DRIVER;
  BOOL ret = FALSE;
  HKEY driverkey = NULL;
  LONG res = 0;
  UNICODE_STRING uFullFileName;
  DWORD FileNameSize = (DWORD) (wcslen(FileName) + 1) * sizeof(WCHAR);

  res = RegCreateKeyExW(serviceskey, DriverName, 0, NULL, 0, KEY_ALL_ACCESS,
    NULL, &driverkey, NULL);
  ret = res == ERROR_SUCCESS;
  if (ret) {
    _PrepareFullName(FileName, L"\\??\\", &uFullFileName);
    if (ret) {
      res = RegSetValueExW(driverkey, L"ImagePath", 0, REG_SZ,
        (PVOID)uFullFileName.Buffer,
        uFullFileName.Length + sizeof(WCHAR));
      ret = res == ERROR_SUCCESS;
      if (ret) {
        res = RegSetValueExW(driverkey, L"Start", 0, REG_DWORD, (PVOID)&start,
          sizeof(start));
        ret = res == ERROR_SUCCESS;
        if (ret) {
          res = RegSetValueExW(driverkey, L"Type", 0, REG_DWORD, (PVOID)&type,
            sizeof(type));
          ret = res == ERROR_SUCCESS;
          if (!ret)
            RegDeleteKeyW(serviceskey, DriverName);
        } else RegDeleteKeyW(serviceskey, DriverName);
      } else RegDeleteKeyW(serviceskey, DriverName);

      _FreeFullName(&uFullFileName);
    }

    RegCloseKey(driverkey);
  }

  return ret;
}
BOOL ntldUninstallDriver (PWCHAR DriverName)
{
  BOOL ret = FALSE;
  LONG res = 0;

  ret = DeleteRegistryKey(serviceskey, DriverName);
  return ret;
}
```

Po jeho úspěšném vytvoření stačí název klíče předat jako parametr nativní funkci `NtLoadDriver`. Protože tato rutina akceptuje řetězec ve formátu UNICODE_STRING, nejprve je nutné nulou ukončený řetězec Unicode (typ `PWCHAR`) do této podoby převést. Konverze se provádí pomocí procedury `RtlInitUnicodeString`, která pro zadaný nulou ukončený řetězec vytvoří strukturu UNICODE_STRING, jež jej popisuje. Potřebné definice vidíte ve výpisu 3.7. Výpis 3.8 ukazuje přímé použití `NtLoadDriver` k načtení ovladače do jádra a `NtUnloadDriver` pro jeho uvolnění.

Výpis 3.7: Definice datových typů a konstant používaných nativními funkcemi Windows API

```
#define STATUS_SUCCESS          0x00000000L
#define STATUS_UNSUCCESSFUL     0xC0000001L
typedef struct _UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
} UNICODE_STRING, *PUNICODE_STRING;

typedef VOID (NTAPI *RTLINITUNICODESTRING)(PUNICODE_STRING UnicodeString,
  PWCHAR WideString);
typedef NTSTATUS (NTAPI *NTLOADDRIVER)(PUNICODE_STRING DriverName);
typedef NTSTATUS (NTAPI *NTUNLOADDRIVER)(PUNICODE_STRING DriverName);
```

Rutina `NtUnloadDriver` je párová k `NtLoadDriver`. Provádí přesně opačnou operaci. Za parametr též bere název klíče služby ovladače. Obě nativní funkce indikují úspěch či neúspěch celé operace hodnotou NTSTATUS.

Výpis 3.8: Načtení a uvolnění ovladače

```
BOOL ntldLoadDriver (PWCHAR DriverName)
{
  BOOL ret = FALSE;
  UNICODE_STRING uFullName;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  ret = _PrepareFullName(DriverName, NTLD_NAME_PREFIX, &uFullName);
  if (ret) {
    status = _NtLoadDriver(&uFullName);
    ret = status == STATUS_SUCCESS;

    _FreeFullName(&uFullName);
  }

  return ret;
}

BOOL ntldUnloadDriver (PWCHAR DriverName)
{
  BOOL ret = FALSE;
```

```
  UNICODE_STRING uFullName;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  ret = _PrepareFullName(DriverName, NTLD_NAME_PREFIX, &uFullName);
  if (ret) {
    status = _NtUnloadDriver(&uFullName);
    ret = status == STATUS_SUCCESS;
    _FreeFullName(&uFullName);
  }

  return ret;
}
```

Funkce pro práci s ovladači pomocí nativních funkcí `NtLoadDriver` a `NtUnloadDriver` naleznete na internetových stránkách knihy v projektu `drv` v souboru `ntregDrivers`.c.

Web: http://www.jadro-windows.cz/projekty/drv

Poznámka: Zde ukázaný postup je téměř ekvivalentní k práci s ovladačem pomocí SCM. Správce služeb při instalaci služby ovladače vytvoří potřebné klíče a její spuštění provádí též přes volání `NtLoadDriver`. Jediný rozdíl mezi oběma postupy tkví v tom, že při manuálním vytvoření potřebných klíčů registru se SCM o nové službě nedozví. Seznam služeb se totiž nachází v paměti procesu `services.exe` a registr slouží pouze jako jeho trvalé úložiště. SCM nekontroluje, zda nějaká aplikace manuálně přidala klíče a hodnoty odpovídající instalaci nové služby. Z tohoto důvodu je načtení ovladače jádra přímo pomocí volání `NtLoadDriver` méně viditelné a často tento postup najdete ve škodlivých programech.

Tip: Pro instalaci ovladače musí být v klíči služby přítomné buď hodnoty `Type`, `Start` a `ImagePath`, nebo `Name`, `Type` a `Start`. Pokud není položka `ImagePath` přítomna, systém předpokládá, že se soubor ovladače nachází v systémovém adresáři pod jménem `<Obsah_položky_Name>.sys`.

Méně známý způsob (nativní funkce NtSetSystemInformation)

Nativní funkce `NtSetSystemInformation` slouží k úpravě různých aspektů systému. Mezi takto konfigurovatelná „nastavení“ patří i načtení nového ovladače do jádra. Odstranění ovladače z paměti tato rutina neumožňuje. Definici funkce vidíte na výpisu 3.9, ukázku volání pak na výpisu 3.10.

Výpis 3.9: Definice nativní funkce Windows API NtSetSystemInformation

```
typedef enum _SYSTEM_INFORMATION_CLASS {
  SystemBasicInformation,
  SystemProcessorInformation,
  SystemPerformanceInformation,
```

```
  SystemTimeOfDayInformation,
  ...
  SystemExtendServiceTableInformation,  // 38
  ...
  SystemMemoryListInformation,
  SystemFileCacheInformationEx,
  MaxSystemInfoClass
} SYSTEM_INFORMATION_CLASS, *PSYSTEM_INFORMATION_CLASS;

typedef NTSTATUS (NTAPI *NTSETSYSTEMINFORMATION)
  (SYSTEM_INFORMATION_CLASS SystemInformationClass, PVOID Buffer,
    ULONG Length);
```

`NtSetSystemInformation` akceptuje tři parametry. První parametr určuje, jaké nastavení systému chce volající změnit. Druhý parametr obsahuje adresu bloku paměti s novými hodnotami a třetí parametr udává délku tohoto bloku.

Pro načtení ovladače do jádra slouží hodnota prvního parametru 36 známá též jako `SystemLoadAndCallImage`. Systém předpokládá jako druhý parametr strukturu SYSTEM_LOAD_AND_CALL_IMAGE, která se skládá pouze z řetězce Unicode reprezentovaného pomocí struktury UNICODE_STRING udávajícího název a umístění souboru ovladače. Úspěch operace se dozvíte z návratové hodnoty NTSTATUS.

Výpis 3.10: Načtení ovladače pomocí volání NtSetSystemInformation

```
BOOL ntssiLoadDriver(PWCHAR DriverName)
{
  BOOL ret = FALSE;
  UNICODE_STRING uFullName;
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  ret = _PrepareFullName(DriverName, NTSSI_NAME_PREFIX, &uFullName);
  if (ret) {
    status = _NtSetSystemInformation(SystemExtendServiceTableInformation,
      &uFullName,
      sizeof(UNICODE_STRING));
    ret = status == STATUS_SUCCESS;
    _FreeFullName(&uFullName);
  }

  return ret;
}
```

Upozornění: Ovladač načtený tímto způsobem se celý nachází ve stránkované paměti a nemůže standardními cestami provádět některé běžné úkony jako například vytváření virtuálních zařízení za účelem komunikace s aplikacemi běžícími v uživatelském režimu.

Práce s ovladači pomocí utility drv

Výše popsané metody práce s ovladači jádra naleznete na internetových stránkách knihy implementované v konzolové aplikaci `drv`. Pomocí tohoto programu a příkazového řádku Windows můžete snadno instalovat vlastní ovladače a načítat je do jádra systému. Program se ovládá následujícími příkazy:

```
drv -scm install <jmeno_sluzby> <jmeno_souboru>
drv -scm load <jmeno_sluzby>
drv -scm unload <jmeno_sluzby>
drv -scm uninstall <jmeno_sluzby>
drv -ntld install <jmeno_sluzby> <jmeno_souboru>
drv -ntld load <jmeno_sluzby>
drv -ntld unload <jmeno_sluzby>
drv -ntld uninstall <jmeno_sluzby>
drv -ntssi load <jmeno_souboru>
```

První parametr udává, kterou metodu má program použít (`scm` – správce služeb, `ntld` – nativní funkci `NtLoadDriver`, `ntssi` – nativní funkci `NtSetSystemInformation`). Podle hodnoty druhého se utilita rozhoduje, jakou akci provést. Nativní API funkce `NtSetSystemInformation` podporuje pouze načtení ovladače do jádra, ostatní způsoby vyžadují před vlastním načtením vytvoření služby, která ovladač popíše.

Web: http://www.jadro-windows.cz/projekty/drv

Jednoduchý příklad: Klasické „Hello World!“

V mnoha publikacích narazíte na ukázkové příklady, pro které se vžilo označení „Hello World!“ Cílem ukázek tohoto typu je většinou nějakým zajímavým způsobem vypsat ono anglické sousloví na obrazovku. Například zobrazením textu v dialogovém okně. O tento typ ukázky nebudete ochuzeni ani v této knize, protože dobře poslouží k ilustraci základní architektury ovladače jádra.

Web: http://www.jadro-windows.cz/projekty/hello

Ovladač `hello.sys` se skládá ze tří částí – souboru MAKEFILE se standardním obsahem, souboru SOURCES s instrukcemi pro překladač balíku WDK (výpis 3.11) a souboru `hello.c`, který obsahuje vlastní zdrojový kód (viz výpis 3.12).

Překladač se ze souboru SOURCES dozví, že má přeložit zdrojový kód v souboru `hello.c` (řádek 3) jako ovladač (řádek 2) se jménem `hello` (řádek 1). Linker ovladačům automaticky do jména souboru doplní příponu `.sys`.

Výpis 3.11: Soubor SOURCES

```
TARGETNAME=hello
TARGETTYPE=DRIVER
SOURCES= hello.c
```

Soubor `hello.c` obsahuje dvě rutiny – `DriverEntry` a `DriverUnload`. `DriverEntry` je analogií k funkcím `main` a `WinMain` známých z prostředí programování běžných aplikací. Systém této funkci předá řízení během načítání ovladače do jádra. Úkolem podprogramu je provést nezbytnou inicializaci, aby po svém načtení do jádra mohl ovladač okamžitě fungovat.

Výpis 3.12: Soubor hello.c

```
#include <ntddk.h>

VOID DriverUnload(PDRIVER_OBJECT DriverObject)
{
  DbgPrint ("Good Bye\n");
  return;
}

NTSTATUS DriverEntry (PDRIVER_OBJECT DriverObject,
  PUNICODE_STRING RegistryPath)
{
  NTSTATUS status = STATUS_UNSUCCESSFUL;

  DbgPrint("HELLO WORLD!\n");
  DriverObject->DriverUnload = DriverUnload;

  status = STATUS_SUCCESS;
  return status;
}
```

Inicializace nemusí vždy proběhnout úspěšně. Proto pokud `DriverEntry` vrátí hodnotu NTSTATUS odpovídající chybě, načtení ovladače do jádra skončí nezdarem.

`DriverEntry` akceptuje dva parametry – adresu struktury DRIVER_OBJECT, která reprezentuje ovladač jako entitu v jádře a obsahuje veškeré informace a nastavení, a úplný název klíče služby, jež ovladač reprezentuje.

Poznámka: Pokud ovladač dostanete do jádra pomocí výše popsané nativní funkce `NtSetSystemInformation`, rutina `DriverEntry` obdrží oba parametry nastavené na hodnotu NULL. Takto zavedený ovladač nemá žádnou strukturu DRIVER_OBJECT, která by jej v jádře reprezentovala, a ani službu, jež by jej zastupovala ve správci služeb. Navíc je celý soubor ovladače tvořen stránkovanou pamětí, jak uvádí poznámka výše.

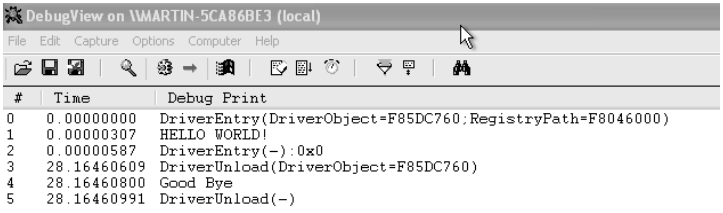
Krom ladicího výpisu „HELLO WORLD!“ provádí rutina `DriverEntry` velmi důležitou věc – umožňuje ovladač uvolnit z paměti za běhu operačního systému. Při povídání o službách jste se dozvěděli, že okolí jim může zasílat různé požadavky, kterým služby ale nemusí vyhovět. Něco podobného platí i pro ovladače. Nemusí dovolit své odstranění z paměti jádra.

Ovladač lze dynamicky uvolnit z jádra právě tehdy, když má nastavenou proceduru, které systém předá řízení během odstraňování z paměti. Tato rutina se často nazývá `DriverUnload` a její úkol je opačný k úkolu funkce `DriverEntry` – uvolnit všechny prostředky, jenž ovladač používal během své přítomnosti v jádře. Adresa této rutiny se musí nastavit do položky `DriverUnload` struktury DRIVER_OBJECT. Na rozdíl od `DriverEntry`, která má přímý vliv na úspěch zavedení ovladače do jádra, tato „uklízecí" procedura nemůže nijak ovlivnit úspěch či neúspěch jeho odstranění.

Protože inicializace ovladače `hello.sys` spočívá pouze v nastavení „uklízecí" procedury a ve vypsání známého anglického sousloví, `DriverUnload` nemá co uklízet a pouze vypíše řetězec „Good Bye".

Ovladače nemohou snadno zobrazovat textové řetězce na obrazovku například pomocí dialogových oken. Nejjednodušším způsobem je použití ladicích funkcí, mezi které patří `DbgPrint`, jež pošle zadaný řetězec debuggeru jádra, je-li v systému přítomen.

Volání `DbgPrint` lze monitorovat i s pomocí jednoduché aplikace `DebugView`, kterou naleznete na serveru `www.sysinternals.com`, nebo internetových stránkách této knihy. Pro zachytávání volání `DbgPrint` je nutné program spustit s administrátorskými právy a v menu **View** zaškrtnout položky **Capture Kernel** (zachytávat události jádra) a **Enable Verbose Kernel Output** (povolit čitelné zobrazování ladicích výpisů jádra). Po načtení ovladače `hello.sys` do jádra a jeho následném odstranění uvidíte v bílém poli programu něco podobného obrázku 3.6. K práci s ovladačem můžete využít program `drv.exe` popsaném na konci předchozího oddílu. Ukázku zavedení ovladače do jádra a jeho následného odstranění pomocí této utility vidíte na obrázku 3.7.



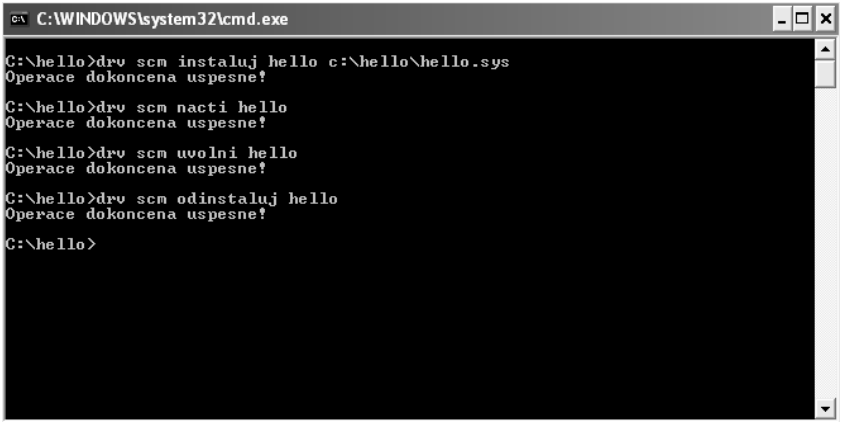
Obrázek 3.6: Projevy ovladače hello.sys v programu DebugView

Několik poznámek k ladění ovladačů

K ladění běžných aplikací většinou stačí program zvaný `debugger`, který umožňuje na specifických místech běh aplikace zastavit, krokovat či prohlížet aktuální hodnoty proměnných, což velmi pomáhá při odhalování nejrůznějších chyb. Debugger je aplikace jako každá jiná – běží v uživatelském režimu a pouze využívá podpory pro ladění, kterou systém Windows disponuje.

Pro ladění ovladačů obyčejný debugger nestačí, protože ovladače běží v režimu jádra a některé akce, které provádějí, snadno kontrolovat nelze. Metody na odhalování chyb ale existují.

Mezi nejúčinnější metody patří prevence. Než vámi vytvořený ovladač pustíte do jádra operačního systému, pečlivě si pročtěte celý zdrojový kód a přemýšlejte, za jakých okolností bude systém jeho jednotlivé části vykonávat a jestli nemůže dojít k uváznutí (deadlock) či špatné manipulaci se sdílenými daty. Nedělejte unáhlené předpoklady, že některé kusy kódu fungují správně, protože jsou krátké a provádějí pouze jednoduché operace. Chyba se často vyskytuje právě v nich. Ačkoliv se tato technika může jevit jako velmi zdlouhavá a otravná, překvapivě ušetří spoustu času a vede k odhalení řady malých chyb.



Obrázek 3.7: Použití drv.exe k manipulaci s ovladačem hello.sys

Další způsob, jak se vyhnout dlouhému ladění, spočívá v přesunutí složitosti řešeného problému z režimu jádra do uživatelské aplikace. Struktura ovladače se tak velmi zjednoduší a ladění aplikace lze provést standardními postupy. Ovladač pak funguje pouze jako prodloužená ruka aplikace – umožňuje jí provádět jinak zakázané operace.

Prevence není nikdy stoprocentně účinná. Nemusí odhalit problémy složitějšího charakteru, které se projeví jen za velmi specifických podmínek. Jedná se například o chyby, jež se objeví pouze při specifickém pořadí plánování vláken na procesoru. Pro jejich odhalení je potřeba použít prostředky umožňující chování ovladače sledovat v reálném čase.

DbgPrint

Na rozdíl od obyčejných programů ovladače nemohou jednoduše zobrazovat dialogová okna se zprávami, aby oznamovaly svůj aktuální stav. Mohou ale využívat rutiny `DbgPrint`, která (podobně jako funkce `OutputDebugString` z rozhraní Windows API) pošle zadaný text debuggeru.

Debuggery jádra umožňují při ladění ovladačů používat podobné postupy jako při ladění aplikací. Dokáží umisťovat breakpointy a prohlížet obsah paměti. Jejich ovládání je méně přívětivé, protože nemohou využívat většiny služeb jádra. Musí totiž běžet na nižší úrovni než většina ovladačů, aby mohly kontrolovat jejich běh. Proto některé mechanismy jádra obcházejí a činí operační systém méně stabilním.

Pro monitorování volání funkce `DbgPrint` nepotřebujete plnohodnotný debugger jádra. Postačující práci odvede i utilita `DebugView`, která pouze zobrazuje výstupy z funkcí, jako je právě `DbgPrint` či `OutputDebugString` a jejíž grafické uživatelské rozhraní jste viděli na obrázku 3.6.

DbgPrintEx

Tato rutina plní stejnou úlohu jako DbgPrint, navíc dovoluje volajícímu určit, jaký typ danou zprávu posílá a jak je tato zpráva závažná; zda jde o pouhou informaci, nebo došlo k závažnému problému. Deklaraci rutiny vidíte na výpisu 3.13.

Výpis 3.13: Deklarace funkce DbgPrintEx

```
NTSTATUS __cdecl DbgPrintEx(
  ULONG ComponentId,
  ULONG Level,
  PCSTR Format,
  ... arguments);
```

Hodnota parametru `ComponentId` určuje typ ovladače, který zprávu zasílá. Možné hodnoty vidíte v tabulce 3.4. Parametr `Level` určuje závažnost zprávy. Povoleny jsou libovolné hodnoty, doporučuje se ale používat pouze následující:

- **DPFLTR_ERROR_LEVEL (0)** – závažná chyba,
- **DPFLTR_WARNING_LEVEL (1)** – varování,
- **DPFLTR_TRACE_LEVEL (2)** – oznámení o vykonávání určité části kódu (například určité funkce),
- **DPFLTR_INFO_LEVEL (3)** – nezávažné oznámení jiného druhu.

Ostatní parametry mají stejný význam jako v případě funkce `DbgPrint` či jiných rutin určených pro formátování řetězců (například `printf`). Funkce indikuje úspěch vrácením hodnoty typu NTSTATUS.

Tabulka 3.4: Hodnoty parametru ComponentId funkce DbgPrintEx a jejich význam

Konstanta	Název komponenty	Popis
DPFLTR_IHVVIDEO_ID	IHVVIDEO	Ovladač videa.
DPFLTR_IHVAUDIO_ID	IHVAUDIO	Ovladač zvuku.
DPFLTR_IHVNETWORK_ID	IHVNETWORK	Síťový ovladač.
DPFLTR_IHVSTREAMING_ID	IHVSTREAMING	Ovladač pracující s proudem dat (například dekódující proud zvukových dat).
DPFLTR_IHVBUS_ID	IHVBUS	Ovladač sběrnice.
DPFLTR_IHVDRIVER_ID	IHVDRIVER	Jiný typ ovladače.

Windows dovolují jednotlivé zprávy na základě hodnot parametrů `ComponentId` a `Level` filtrovat; debuggeru doručují jen zprávy s určitými kombinacemi těchto hodnot. Konkrétní nastavení filtrování lze provést buď přímo v debuggerem modifikací daného nastavení přímo v paměti jádra, nebo změnou hodnot klíče HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Debug Print Filter. Tento klíč obsahuje hodnoty typu DWORD. Název hodnoty odpovídá názvu komponenty (viz druhý sloupeček tabulky 3.4), data jsou interpretována jako bitová maska určující, jak závažné zprávy od daného typu ovladače (závažnost zprávy se posuzuje podle hodnoty parametru `Level`) bude systém zasílat. Změna tohoto klíče se do nastavení filtrování promítne až po restartu počítače, protože si operační systém tento klíč čte pouze během svého startu. Změna přímo v paměti jádra se projeví okamžitě.

Rozhodování, zda bude zpráva zaslána debuggeru, probíhá následovně:

- Z hodnoty parametru `Level` systém vypočítá bitovou masku. Hodnota mezi 0 a 31 včetně odpovídá masce s pouze jedním bitem nastaveným na jedničku. Jedná se o danou mocninu dvojky. Například hodnota 5 znamená masku 0x00000020 = 32 = 2^5. Ostatní hodnoty parametru `Level` nejsou nijak transformovány a systém je interpretuje přímo jako bitovou masku.
- Následně jádro provede logický součin (operace AND) masky z předchozího kroku s maskou nastavení filtrování zpráv příslušné komponenty logickým součtem (operace OR) zkombinovanou s maskou WIN2000. Maska příslušné komponenty je při startu systému načtena z registru z výše jmenovaného klíče a může být později přímo upravena debuggerem jádra. Pokud je výsledkem nenulové číslo, zpráva je debuggeru poslána. V opačném případě zaniká. Pokud bitová maska pro danou komponentu neexistuje, jádra její hodnotu uvažuje jako nulovou a po operaci logického součtu nabude hodnoty masky WIN2000.

Poznámka: Maska WIN2000 má ve výchozím natavení hodnotu 0x1. Protože je vždy zkombinována s maskou natavení filtrování určité komponenty, zajišťuje, že nejnižší bit odpovídající úrovni DPLFLTR_ERROR_LEVEL bude vždy nastaven na jedničku, a tak hlášení o závažných chybách budou odeslány debuggeru i v případě, že nastavení filtrování pro příslušnou komponentu by takové jejich odesílání nepovolovalo.

Od Windows Vista se volání `DbgPrint` chová stejně jako volání `DbgPrintEx` s parametrem `ComponentId` DFLTR_DEFAULT_ID a hodnotou `Level` DPLFLTR_INFO_LEVEL. Na starších verzích operačního systému `DbgPrint` vždy posílá zprávy debuggeru.

Tip: Program `DebugView`, který naleznete na stránce www.sysinternals.com, zobrazuje zprávy nehledě na nastavení filtrování v operačním systému. Při zaškrtnuté volbě **Pass Through** v menu **Options** všechny obdržené zprávy přeposílá dále debuggeru jádra a k filtrování vůbec nedochází.

ASSERT

Makro ASSERT slouží testování invariantů a různých podmínek během vykonávání kódu ovladače. Jeho syntaxe je následující:

```
ASSERT(Vyraz)
```

Vyhodnotí-li se obsah výrazu `Vyraz` na TRUE, makro nedělá nic. V opačném případě pošle zprávu debuggeru jádra. V této zprávě najdete název souboru zdrojového kódu a číslo řádky, kde se dané makro nachází, a také text výrazu, který se vyhodnotil na FALSE. Makro slouží k ověření, že při vykonávání daného kódu platí ještě další podmínky, které ale nejsou testovány pomocí konstrukcí if, protože platnost těchto podmínek je dána implicitně – jinak řečeno, ve finální verzi ovladače takové podmínky platí vždy.

Ve finální verzi ovladače se tedy všechna použití makra ASSERT vyhodnotí na TRUE, takže nedochází k odesílání zpráv debuggeru. Makro má výše popsanou sémantiku pouze v případě překladu ovladače v prostředí `Checked build`, kdy má symbol DBG hodnotu 1. V prostředí `Free build`, kde symbol DBG má hodnotu 0, nebo není vůbec definován, se makra ASSERT nezahr-

nují do vygenerovaného binárního souboru `.sys`. Překladač se chová tak, jako by ve zdrojovém kódu vůbec nebyla napsána.

KdPrint a KdPrintEx

Tato makra se v případě překladu zdrojového kódu v prostředí `Checked build` chovají jako rutiny `DbgPrint` a `DbgPrintEx`. Pokud ovladač překládáte v prostředí `Free build`, překladač tato makra ignoruje. Chová se tedy podobně jako v případě makra `ASSERT`. Na rozdíl od volání ostatních funkcí a maker, seznam parametrů pro `KdPrint` a `KdPrintEx` musíte uvádět ve dvojitých kulatých závorkách, například takto:

```
KdPrint(("Testovaci ladici vypis\n"));
```

Důvodem této nutnosti je implementace těchto maker v hlavičkových souborech balíku Windows Driver Kit.

WinDbg

Pokud výpisy z `DbgPrint` k identifikaci problému nestačí, můžete zkusit plnohodnotný debugger jádra. Jedním z nich je program WinDbg, který naleznete v balíku Debugging Tools For Windows, jenž se standardně instaluje společně s WDK. Tento debugger umožňuje za běhu prohlížet a upravovat obsah paměti jádra. Krokování a breakpointy v režimu jádra však nepodporuje lokálně. Abyste mohli ovladač krokovat, musíte jej pustit na jiném (třeba i virtuálním) počítači, na který se potom například pomocí pojmenované roury či sériového portu připojíte.

Protože plně nepodporuje lokální ladění jádra, nemusí tato aplikace ani obcházet mechanismy, čímž nesnižuje stabilitu celého systému. Grafické uživatelské rozhraní debuggeru vidíte na obrázku 3.8.

Pokud hledaná chyba ovladače způsobuje modrou obrazovku smrti, systém většinou na disk ukládá obsah paměti jádra v okamžiku zjištění problému – tzv. crash dump. WinDbg dokáže tyto výpisy paměti při selhání analyzovat, a tak můžete vědět přesně, kde chyba nastala, aniž byste potřebovali druhý stroj, i když třeba jen virtuální.

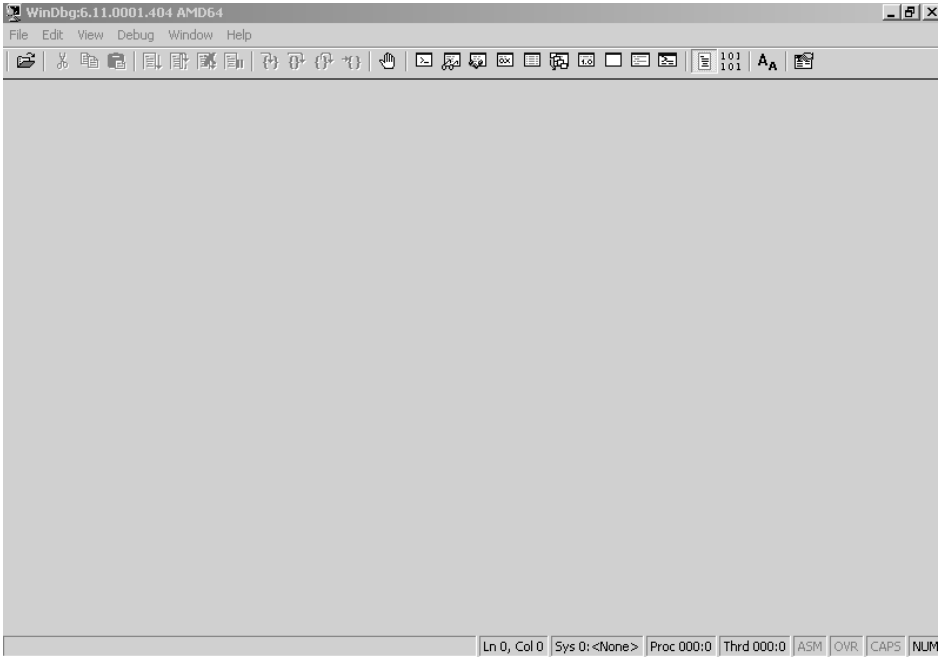
Obsluha WinDbg není nejpříjemnější – program se ovládá podobně jako příkazová řádka. Pro diagnostiku většiny problémů s ovladači ale vystačíte jen s několika málo příkazy, které popisují následující odstavce.

WinDbg umí pracovat s údaji v souborech `.pdb`, které obsahují například názvy a umístění proměnných, datových typů a podprogramů. Tyto informace jsou dostupné i pro hlavní moduly jádra (`ntoskrnl.exe`, `hal.dll`, `win32k.sys`) a nachází se na stránkách Microsoftu.

Pro korektní nastavení ladicích symbolů stačí v menu **File** zvolit položku **Symbol Path...** a do textového pole vyplnit adresu serveru, která zní

```
srv*DownstreamStore*http://msdl.microsoft.com/download/symbols
```

WinDbg z tohoto serveru stahuje ladicí informace k těm souborům, jež zrovna potřebuje. Po správné konfiguraci symbolů můžete začít s vlastní analýzou souboru crash dump.



Obrázek 3.8: Uživatelské rozhraní programu WinDbg

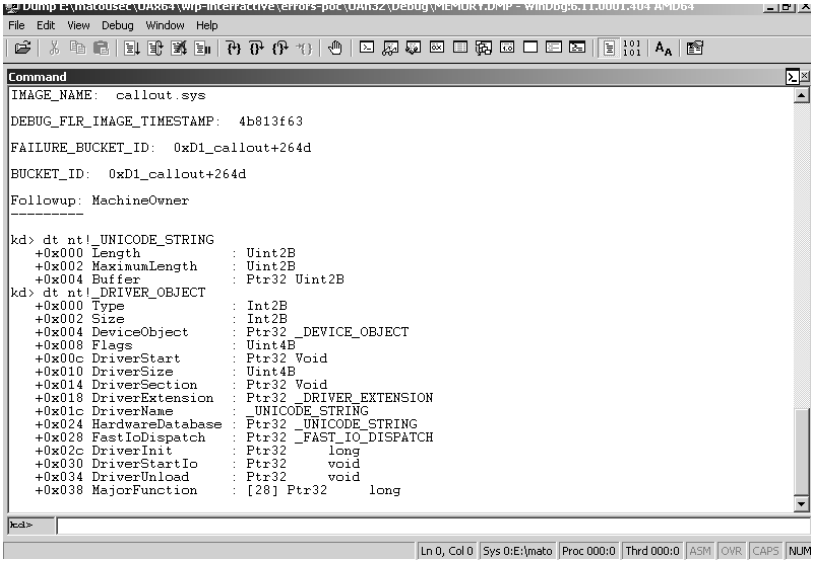
Poznámka: Pokud váš počítač není během práce s debuggerem připojen k Internetu, můžete si ladicí informace k většině ovladačů od Microsoftu stáhnout na disk a do textového pole ve WinDbg vyplnit jejich umístění.

Po otevření souboru crash dump (**File – Open Crash dump...**) WinDbg automaticky vykoná příkaz `!analyze`, který provede zběžný průzkum výpisu paměti. Dozvíte se kód chyby, která modrou obrazovku způsobila, a název ovladače, v němž vznikla.

Podrobnější informace vypíše příkaz `!analyze` s parametrem `-v`. Debugger zobrazí krátký text, který popisuje nejčastější příčiny vzniku dané chyby a význam parametrů. Program také vypíše hodnoty registrů procesoru a stav zásobníku volání v době pádu. Tak se dozvíte, v jaké rutině k chybě došlo. WinDbg se snaží všechny adresy v paměti překládat pomocí ladicích symbolů na jména proměnných a funkcí. Pokud najde i zdrojový kód ovladače, zobrazí číslo řádky, na kterém došlo k volání určité rutiny. Pro některé chyby zobrazuje přímo výpis zdrojového kódu.

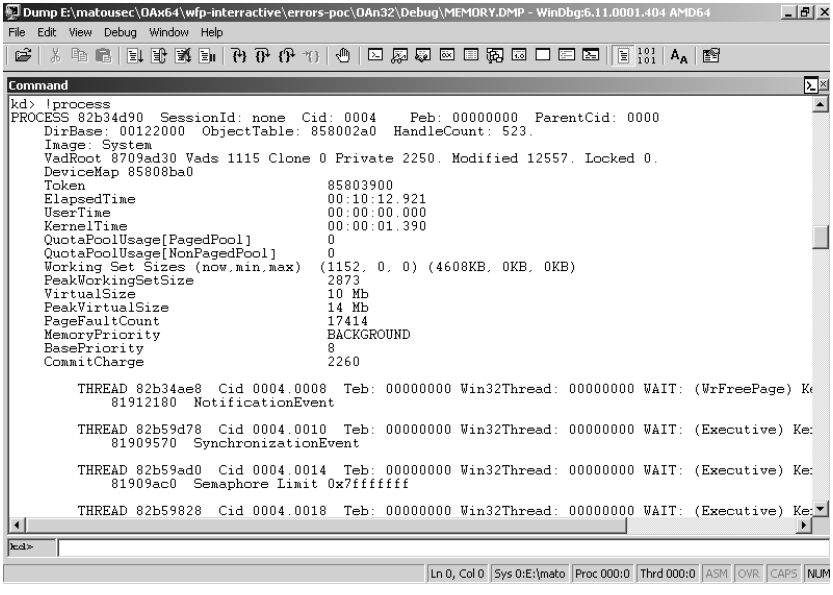
Pokud informace poskytnuté příkazem `!analyze -v` nejsou dostatečné, můžete použít některé z následujících příkazů:

- `dt <typ> [adresa]` – zobrazí strukturu zadaného datového typu. Pokud uvedete i nepovinný parametr `adresa`, pokusí se obsah paměti na této adrese interpretovat jako obsah položek zadaného datového typu. Tento příkaz se hodí nejenom na zjišťování obsahu paměti, ale i na zkoumání datových struktur samotného operačního systému. Podmínkou je mít správně nastavenou cestu k souborům se symboly. Na obrázku 3.9 vidíte několik ukázek použití.



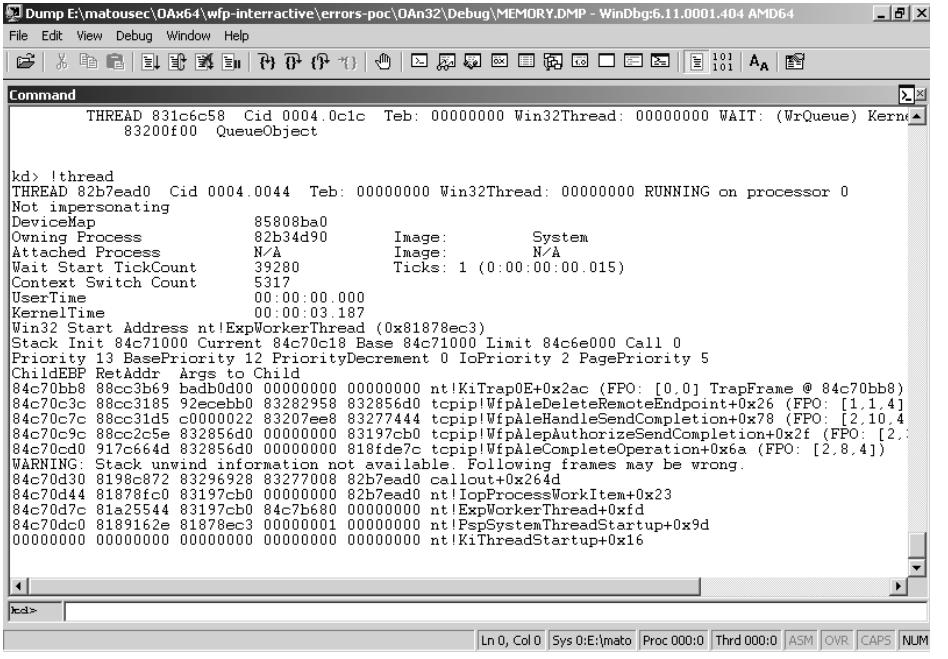
Obrázek 3.9: Příklad použití příkazu dt

- `!process` – zobrazí informace o běžících procesech v době selhání (viz obrázek 3.10).

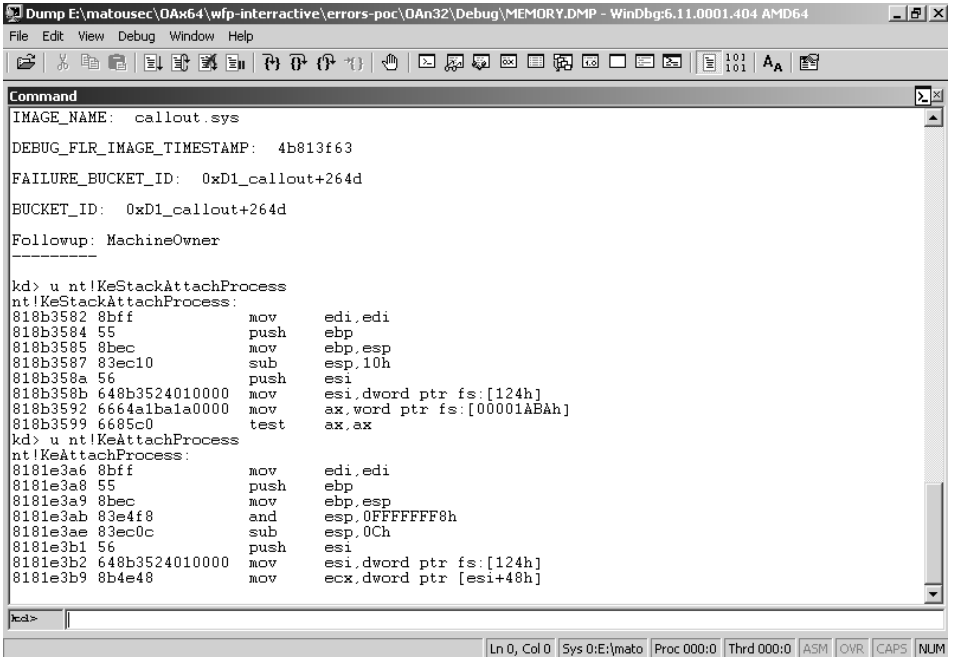


Obrázek 3.10: Příkaz !process

- `!thread` – zobrazí informace o vláknu, které selhání způsobilo. Dozvíte se i údaje o procesu, jemuž patří. Ukázkový výpis vidíte na obrázku 3.11.

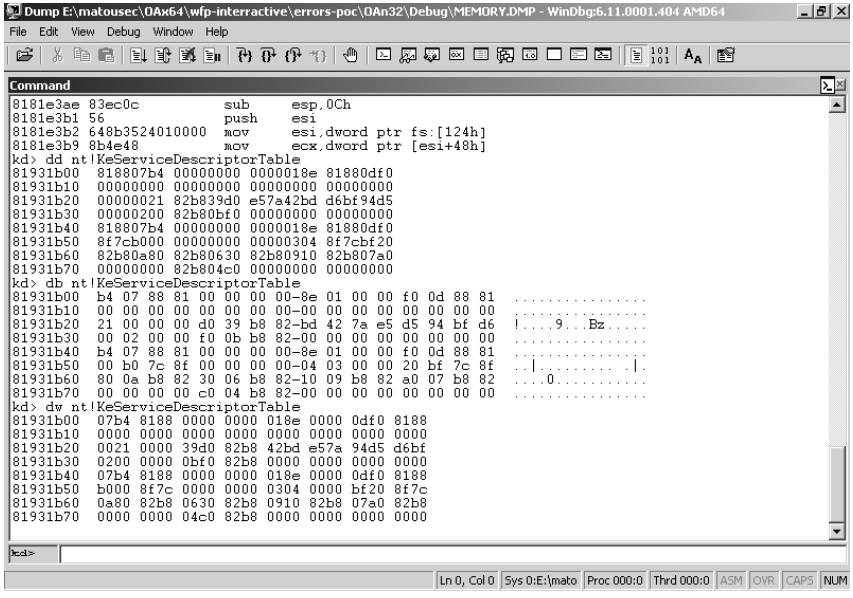


Obrázek 3.11: Příkaz !thread



Obrázek 3.12: Příkaz u

- `u <adresa>` – interpretuje obsah paměti na zadané adrese jako instrukce procesoru. Jako adresu můžete zadat i jméno funkce. Pokud je cesta k souborům PDB dobře nastavena, program si jméno na adresu sám převede. Příklad najdete na obrázku 3.12.
- `db`, `dw`, `dd` – vypíše obsah paměti na zadané adrese jako posloupnost bajtů, slov či dvojslov (viz obrázek 3.13).



Obrázek 3.13: Příkazy db, dw a dd

Tip: Seznam všech příkazů a jejich podrobný popis naleznete v nápovědě k programu. Zde jsou uvedeny pouze ty nejčastěji používané.

Modrá obrazovka smrti

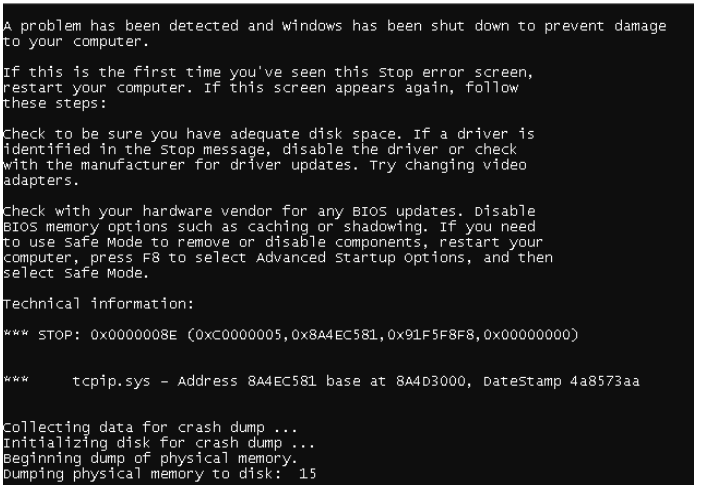
Modrou obrazovku smrti (Blue Screen Of Death – BSOD) již někteří uživatelé pamatují z dob Windows 9x/Me. Systém tuto obrazovku zobrazí, pokud dojde k události neslučitelné s pokračováním v jeho činnosti. Tento oddíl se vám pokusí přiblížit okolnosti jejího vzniku a procesy, které probíhají během vypisování bílého textu na modré pozadí a po něm. Konkrétní ukázku takové modré obrazovky vidíte na obrázku 3.14.

Text modré obrazovky obsahuje informace o tom, co mohlo selhání způsobit a jak by se mu dalo do budoucna předejít. Protože příčin pádu může být velmi mnoho, tyto rady mají velmi obecný charakter, a tudíž nejsou příliš užitečné.

Naopak velmi užitečné informace se nachází v dolní části obrazovky, kde mimo chybového kódu za textem STOP:někdy naleznete i název ovladače, jenž problém pravděpodobně způsobil.

Do horní části obrazovky systém někdy velkými písmeny oddělenými znakem podtržítka vypíše slovní označení pro daný kód chyby.

O probíhajícím výpisu obsahu paměti do stránkovacího souboru systém informuje na posledních řádcích. Pokud se tyto informace neobjeví, ukládání do stránkovacího souboru neproběhlo. Výpis stavu paměti při selhání lze následně použít k odhalení příčiny.



Obrázek 3.14: Modrá obrazovka smrti

Okolnosti vzniku

Když systém zjistí, že není něco v pořádku, vyvolá modrou obrazovku a ukončí svoji činnost. Důvodem může být jak selhání pevného disku či poškození konzistence samotného jádra, ale i drobná chyba v cizím ovladači, ze které by se systém teoreticky mohl zotavit. Včasným zastavením činnosti ale chce předejít možnému poškození hardware.

Pro detekci takových nebezpečných situací má systém zabudováno několik mechanismů. Například správce paměti dokáže označováním bloků paměti, se kterými pracuje, zjistit, že se nějaký ovladač pokusí uvolnit oblast, která mu nepatří. Dále jádro kontroluje, jak rychle probíhají určité kritické operace. Pokud zjistí, že některá z nich trvá moc dlouho, usoudí, že došlo k uváznutí v nekonečné smyčce, a ukončí svoji činnost.

Průběh

Systém vyvolá modrou obrazovku pomocí interních funkcí, mezi které patří `KeBugCheck` a `KeBugCheckEx`. Těchto rutin mohou využít i programátoři ovladačů, pokud usoudí, že za určitých okolností systém prostě nemůže pokračovat v činnosti.

Poznámka: Někdy je vhodné modrou obrazovku smrti vyvolat, i když to okolnosti nevyžadují. Některé viry totiž při vypínání počítače normálním způsobem kontrolují, zda jsou stále zapsány v registrech a na pevném disku tak, aby se znovu spustily při příštím startu. Pokud však běh počítače ukončíte modrou obrazovkou, malware si nemůže ověřit prakticky nic.

Hned po zavolání některé z výše zmíněných rutin systém provede několik kroků k tomu, aby bylo možné zaznamenat informace o selhání na disk. Nejprve pošle všem procesorům krom aktuálního požadavek na zablokování. Jestli se je zablokovat skutečně podařilo, už ale nekontroluje. Jádro se totiž nachází v nedefinovaném stavu, kdy již nelze použít žádných synchronizačních primitiv, protože by mohlo dojít k dalšímu selhání, jež by zamezilo výpisu dat o příčině toho původního na bezpečné místo na pevném disku. Systém si zároveň zajistí naprostou exkluzivitu aktuálního procesoru – aktuálně vykonávaný kód nemůže být ničím přerušen.

Jakmile má aktuální vlákno zaručen exkluzivní přístup k procesoru a k veškerému hardware, přepne monitor do textového režimu a vypíše známý text o tom, že došlo k selhání. Systém následně zapíše informace o chybě na disk, a pokud uživatel má zakázán automatický restart při selhání, procesor přejde do vykonávání nekonečné smyčky.

Během vypisování obsahu modré obrazovky se dostanou ke slovu i jiné ovladače než hlavní modul jádra, který obsahuje rutiny `KeBugCheck` a `KeBugCheckEx`. Každý ovladač si může zaregistrovat rutinu, jíž jádro předá řízení těsně po výpisu většiny textu na obrazovku. Rutina má ale vzhledem k nekonzistentnímu stavu systému velmi omezený repertoár možností. Základní omezení jsou tato:

- Nelze alokovat paměť – ani stránkovanou ani nestránkovanou.
- Nelze přistupovat ani k již alokované stránkované paměti. Nekonzistentní stav systému nedovoluje obsluhovat výpadky stránek.
- Není možné použít žádných synchronizačních mechanismů.
- Nelze volat drtivou většinu funkcí exportovaných jádrem.

Co tedy taková rutina dělat může? Jediná užitečná možnost spočívá v předání důležitých informací perifernímu zařízení, například pevnému disku, což Windows opravdu udělají, jsou-li tak nakonfigurovány.

Jakmile systém zjistí všechny dostupné informace o selhání, nashromážděná data zapíše na disk, aby se na ně uživatel mohl později podívat. Údaje zapisuje do stránkovacího souboru, který se následně označí speciálním příznakem, aby při příštím startu systém věděl, že obsah souboru není bezcenný.

Během bootovací sekvence systém zkontroluje, zda se ve stránkovacím souboru nenachází informace o předchozím selhání. Pokud ano, uloží je do předem nastaveného adresáře.

Z výše popsaného vyplývají následující důsledky:

- Pokud stránkovací soubor nemáte zapnutý, informace o selhání systému se na disk neuloží. Proto je rozumné u stránkovacího souboru nastavit alespoň minimální velikost, která se například u Windows XP rovná dvěma megabajtům.
- Nemělo by se stávat, aby se vypisované informace do stránkovacího souboru nevešly. Tudíž mějte tento soubor dost velký, aby k tomu nedocházelo. Pokud chcete vypisovat jen nejnutnější informace o selhání, bohatě stačí nastavit velikost na 2 MB. Pokud při selhání chcete vypisovat obsah celé paměti jádra, resp. veškeré fyzické paměti, stránkovací soubor by měl být tak velký, aby tyto informace pojal – výpis celé paměti jádra zabere často i pár set megabajtů.
- Pro vyjmutí informací o selhání ze stránkovacího souboru je nutné systém úspěšně nastartovat do použitelného stavu, přinejhorším do nouzového režimu. Informace o selhá-