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

Obsah

KAPITOLA 4

Synchronizace 113

KAPITOLA 5

Výjimky, přerušení a systémová volání 147

KAPITOLA 7

Procesy a vlákna 259

KAPITOLA 8

Správce vstupně/výstupních operací 305

KAPITOLA 9

Správa paměti 327

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.

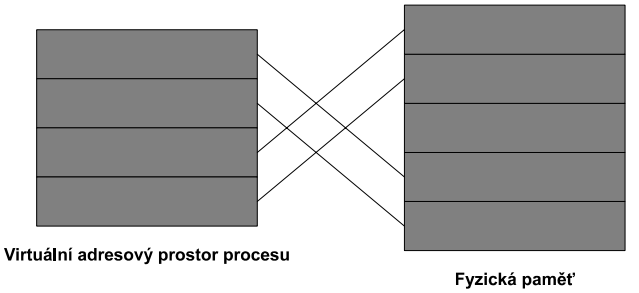
Překlad obvykle není možné definovat pro jednotlivé virtuální adresy, ale pro jejich bloky (například o velikosti 4 KB), které se označují jako *stránky*.

Použití virtuální paměti poskytuje následující výhody:

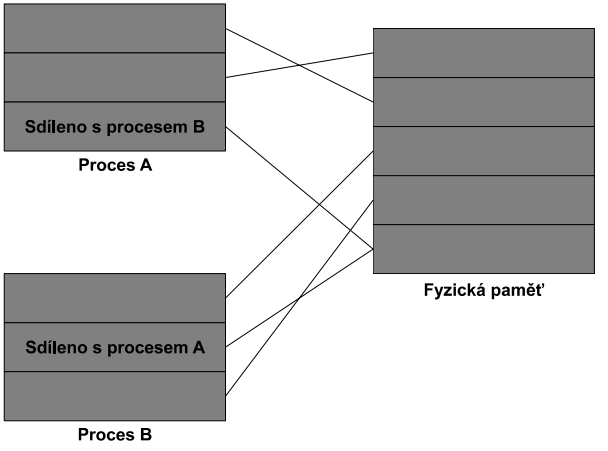
- **Ochrana –** některé procesory umožňují pro každou stránku určit, jaké operace s ní lze provádět. Tímto způsobem operační systém například chrání některé oblasti fyzické paměti proti zápisu. U modernějších procesorů je též možné zakázat na určitých stránkách spouštění kódu. Navíc ne každý blok fyzické 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

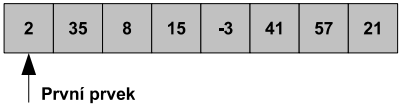
Operační systém si potřebuje pamatovat mnoho informací, například seznam běžících procesů, údaje o všech načtených ovladačích či seznam volných a alokovaných bloků fyzické paměti. S každým druhem informací je potřeba zacházet trochu odlišným způsobem. Například pokud nějaká součást jádra požádá o alokaci 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

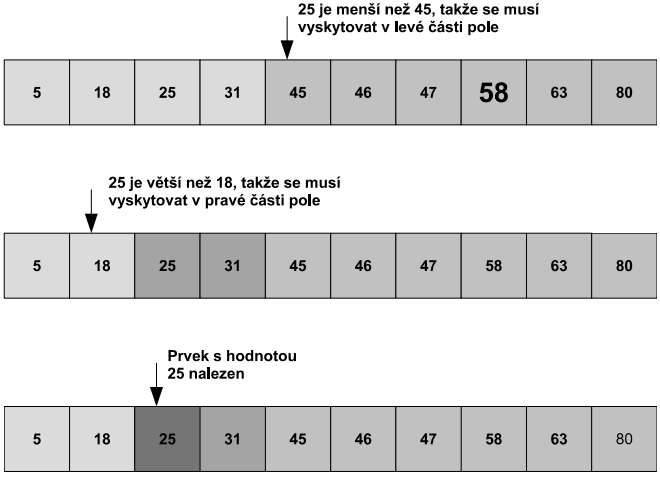
Výhodou pole je velmi rychlý přístup k N-tému prvku (tato vlastnost se též označuje jako *náhodný přístup*). Pokud známe adresu začátku pole, adresa N-tého prvku je určena vzorcem:

```
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.

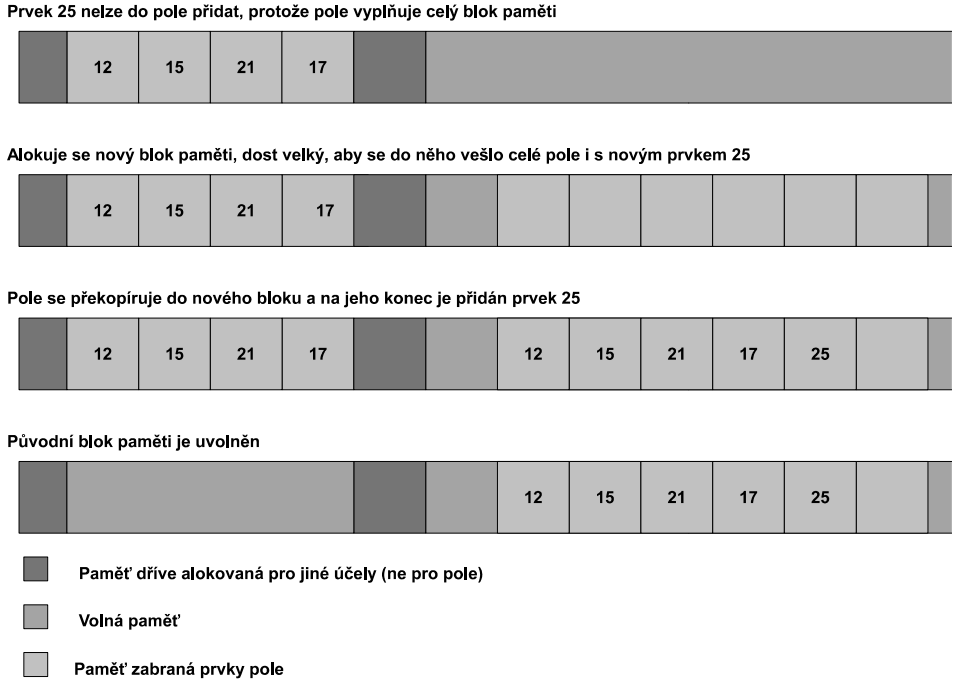
Pokud je pole navíc setříděné, je možné rychle provést test na existenci prvku s určitou hodnotou. Tato operace již závisí na velikosti pole. Je nutné provést nejvýše tolik operací, kolik je dvojkový logaritmus z počtu prvků ve struktuře.

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.



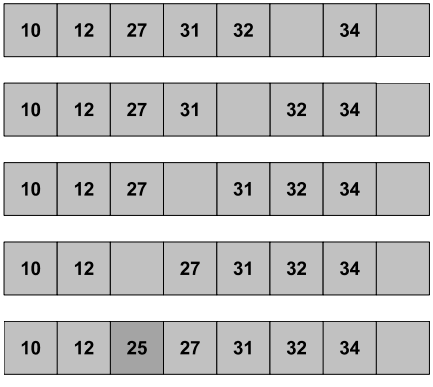
Obrázek 1.4: Ukázka průběhu algoritmu na vyhledávání hodnoty v setříděném poli

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

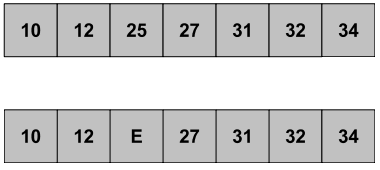
Navíc se může stát, že volná paměť je rozdrolena do malých kousků, které sice v součtu dávají dostatek místa pro větší blok, ale souvislý blok dostatečné velikosti neexistuje.



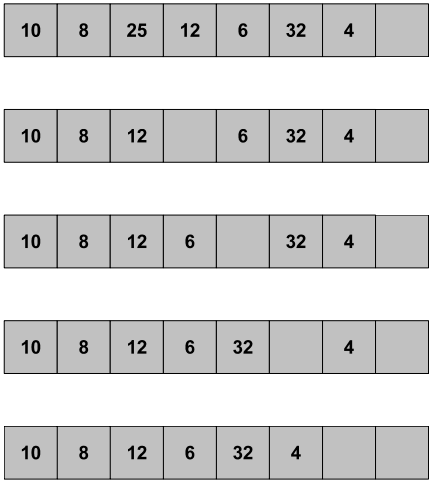
Obrázek 1.6: Přidávání doprostřed pole

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

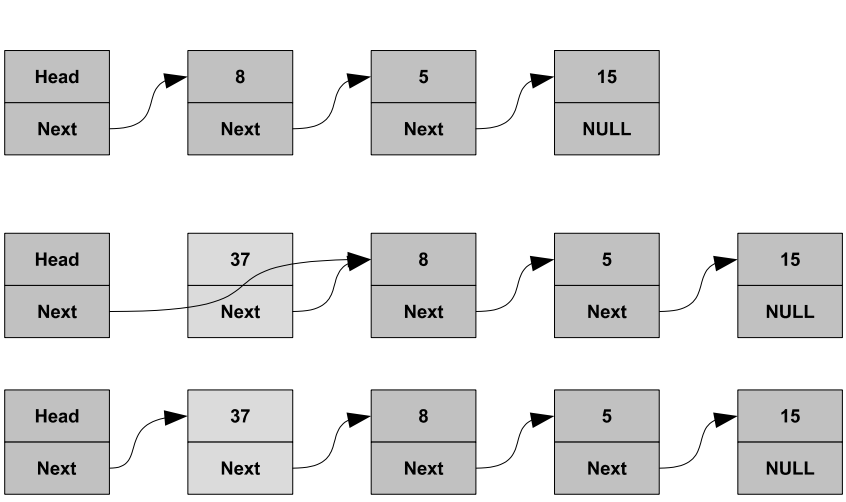
Pole se využívají především při implementaci složitějších datových struktur jako zásobník, fronta či hašovací tabulka. Hodí se na uchovávání dat, která se příliš často nemění – nedochází k častému přidávání nových a mazání starých prvků. Jak je totiž vidět z předchozích odstavců, přidávání či mazání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

Spojové seznamy

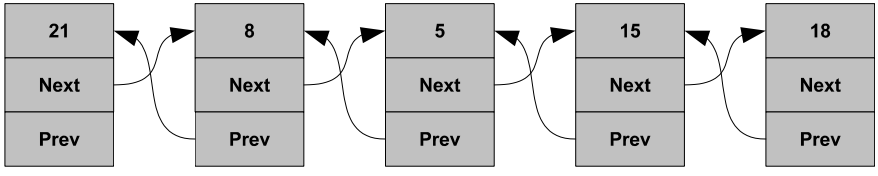
Spojový seznam ukládá každý svůj prvek v odděleném bloku paměti. Tím je eliminována potřeba velkých volných souvislých úseků paměťových stránek, pokud datová struktura obsahuje velké množství prvků. Každý paměťový blok, který obsahuje jeden prvek seznamu, v sobě zároveň uchovává další informace o poloze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

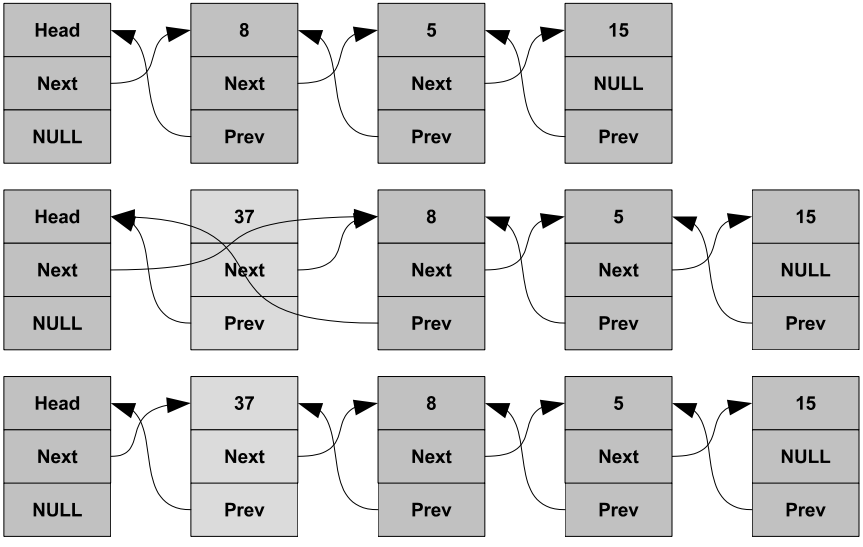
Jednosměrné i obousměrné seznamy je možné ještě obohatit následujícím způsobem:

- 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)



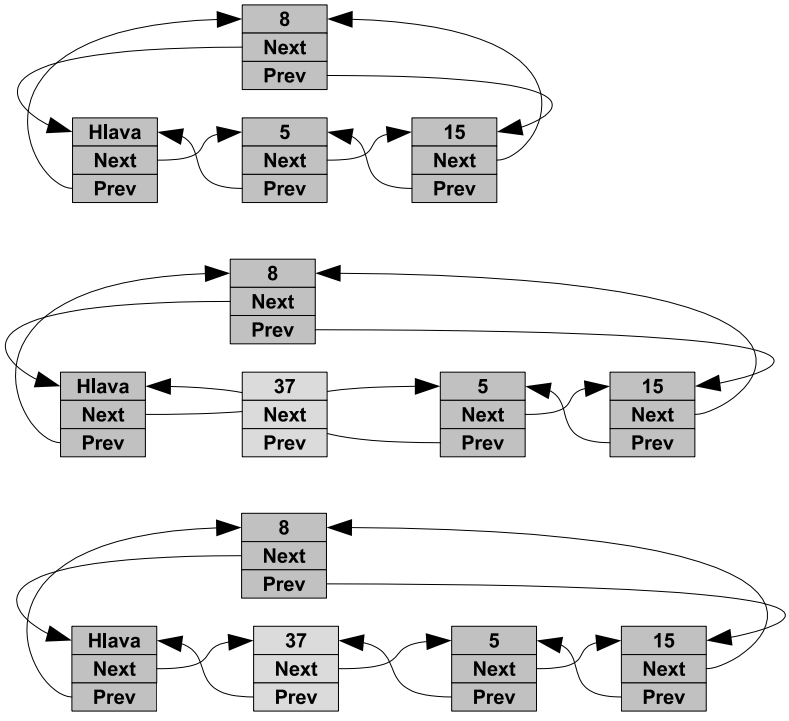
Obrázek 1.11: Operace s obousměrným seznamem s hlavou

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:

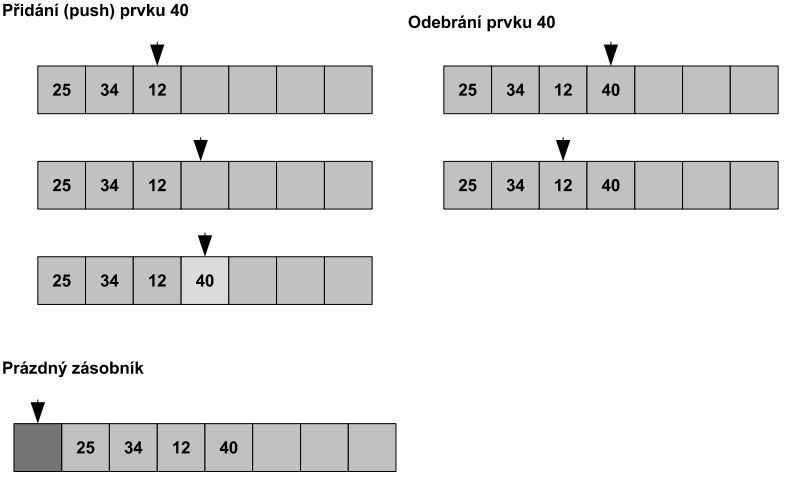
- Vložení nového prvku (`push`).
- Odebrání naposledy vloženého prvku (`pop`).
- Test prázdnosti (`empty`).

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:

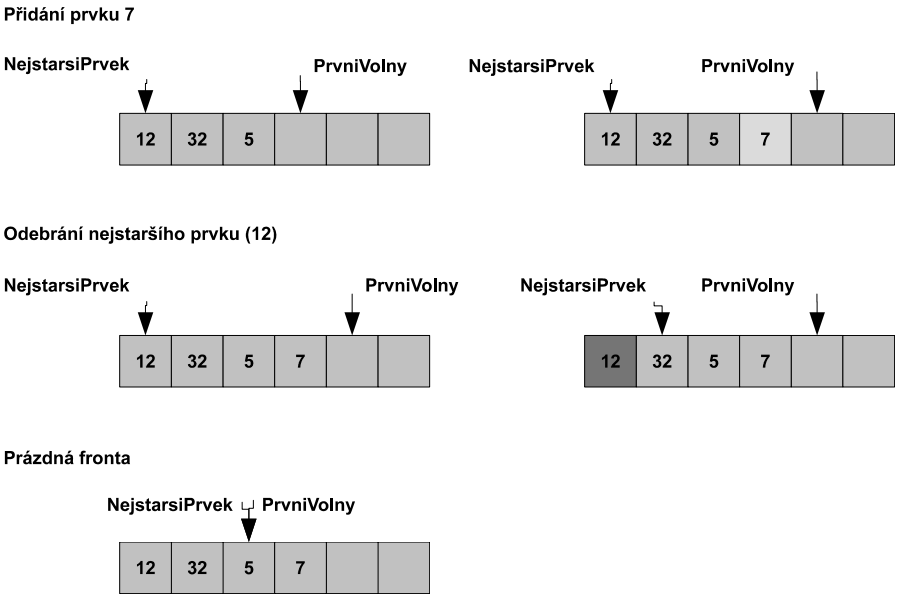
- Vložení nového prvku (`insert`).
- Odebrání prvku, který se ve frontě nachází nejdelší dobu – byl ze všech prvků vložen nejdříve (`removefirst`).
- Test, zda je struktura prázdná (`empty`).

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.

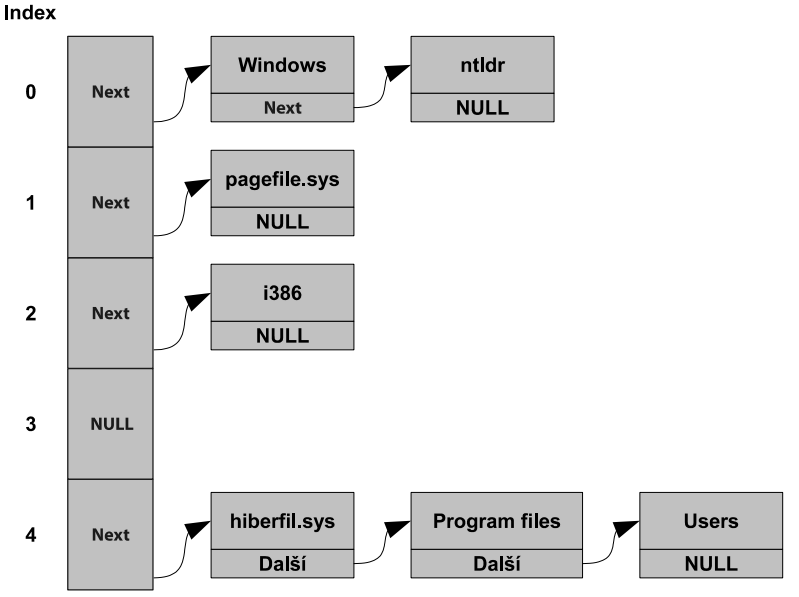
Jak napovídá třetí vlastnost, může dojít k situaci, kdy hašovací funkce transformuje několik klíčů s různou hodnotou na stejný index slotu. Tomuto problému se nelze vyhnout, protože množina všech hodnot klíče bývá obvykle mnohem větší než počet slotů v poli hašovací tabulky. Například klíče tvořené řetězci 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

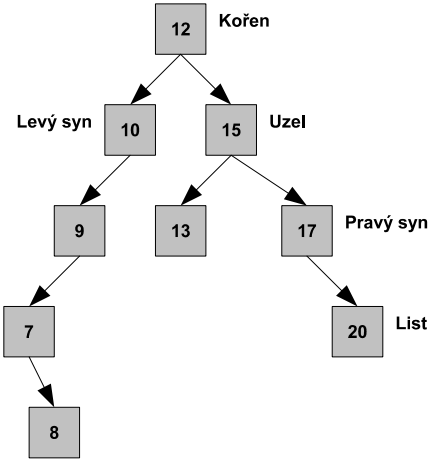
Stromy

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).

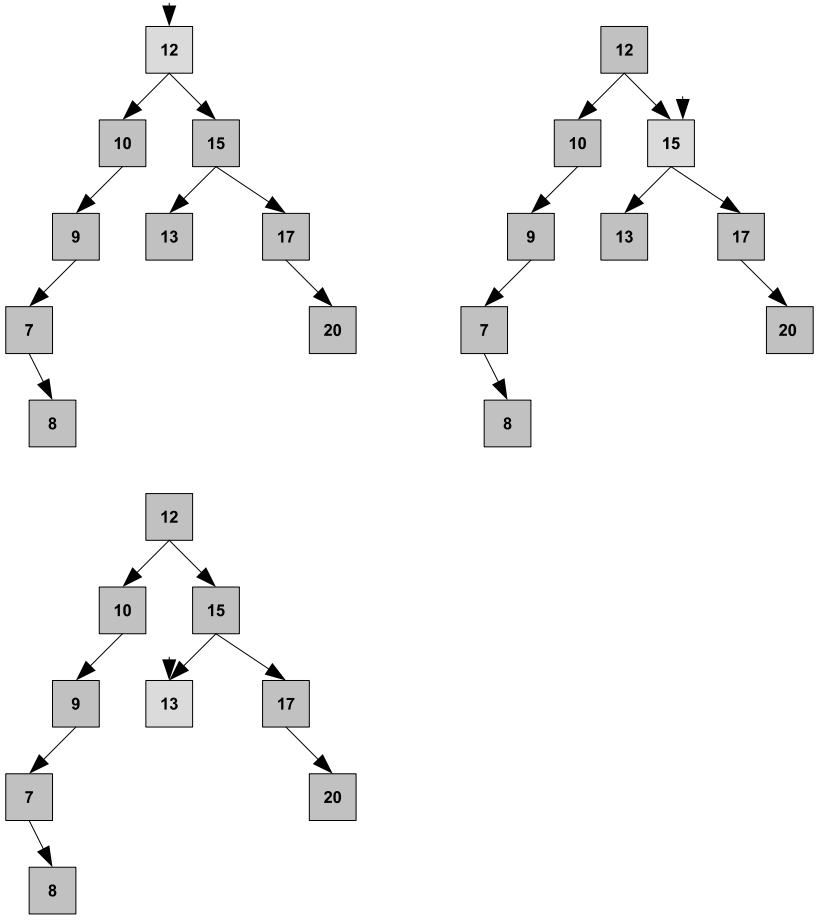
Vyhledávání uzlu s určitou hodnotou probíhá následujícím způsobem:

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

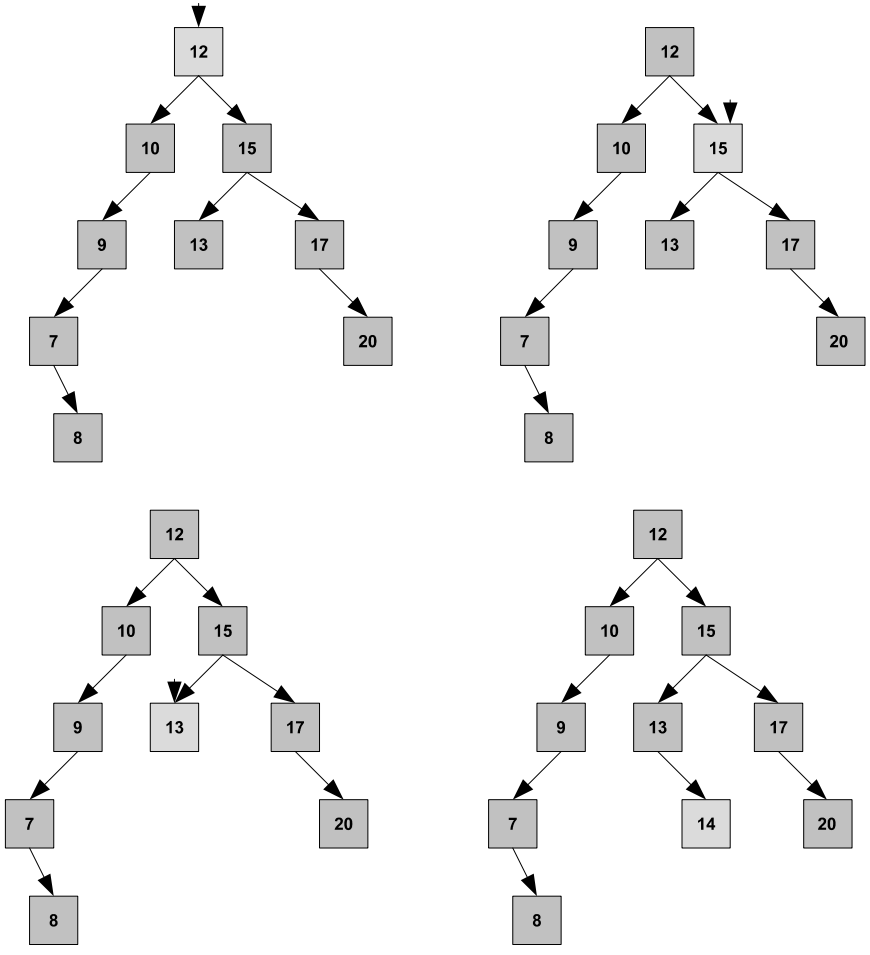
pravého následníka (nemusí mít ani jednoho). Nový uzel je přidán jako levý (resp. pravý) následník, záleží na jeho hodnotě vzhledem k hodnotě jeho předchůdce (otce). Přidávání nového uzlu zachycuje obrázek 1.18.



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.

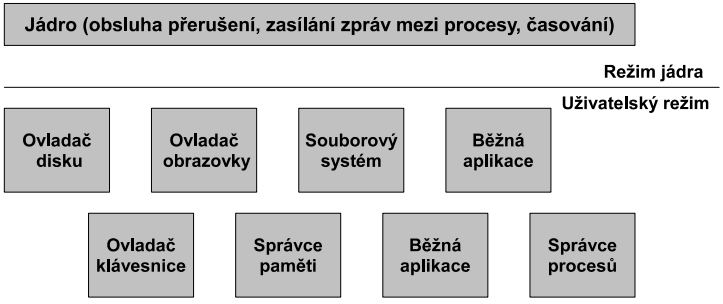
Mikrojádro a monolitický operační systém

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