INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

OPERAČNÍ SYSTÉMY

SYSTÉM SOUBORŮ

doc. Dr. Ing. Oldřich Kodym

Ostrava 2013

ISBN 978-80-248-3053-7

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

7. SYSTÉM SOUBORŮ

OBSAH KAPITOLY:

Koncepce systému souborů, základní atributy souboru.

Operace se soubory.

Metody přístupu k souboru.

Organizace souborů na odkládacím zařízení.

Ochrana souborů a řízení přístupu k nim.

MOTIVACE:

Chod operačního systému využívá mnoha standardních mechanizmů známých z jiných oblastí řízení i běžného života. Pro většinu uživatelů je systém souborů nejviditelnější součástí operačního systému. Provádí mechanismy pro on-line ukládání a přístup k programům a datům jak operačního systému, tak i všech uživatelů výpočetního systému. Grafické uživatelské rozhraní operačního systému dnes zobrazuje především právě soubory, ať už obsahují uložená data nebo vlastní aplikace. Výběrem souboru je pak specifikována akce, která má být provedena. S ohledem na množství souborů v dnešních výpočetních systémech (i stovky tisíc) hraje významnou úlohu jejich organizace.

CÍL:

Systém souborů – koncepce, atributy, operace

Metody přístupu k souborům – sekvenční, přímý, další metody

Adresářová struktura

Ochrana souborů, přístupová práva

1. ÚVOD

Pro většinu uživatelů je systém souborů nejviditelnější součástí operačního systému. Provádí mechanismy pro on-line ukládání a přístup k programům a datům jak operačního systému, tak i všech uživatelů výpočetního systému.

Systém souborů se skládá ze dvou oddělených částí: z množiny *souborů*, v každém jsou uložena nějaká data a z *adresářové struktury*, která organizuje všechny soubory v systému a podává o nich informace. Některé operační systémy mají ještě třetí součást: *partitions*, které užívají k logickému nebo fyzickému oddělení velkých adresářových struktur.

V této kapitole si popíšeme různé aspekty souborů a různé adresářové struktury. Budeme také diskutovat cesty k ochraně souborů, která je nutná v prostředí, kde k souboru může současně přistupovat více uživatelů a kde je většinou žádoucí kontrola kým a jak může být soubor užit. V závěru ukážeme některé aspekty sdílení souborů více procesy.

2. KONCEPCE SYSTÉMŮ SOUBORŮ

Počítače mohou uchovávat informace na různých mediích, jako jsou magnetické disky, magnetické pásky, optické disky, magnetooptické disky apod. Pokud má být počítačový systém použitelný, musí operační systém provádět jednotný logický pohled na ukládané informace.

Operační systém musí abstrahovat od fyzických aspektů ukládacího zařízení a definovat logickou datovou jednotku – *soubor*. Soubor je operačním systémem mapován na fyzické zařízení. Tato zařízení jsou obvykle *energeticky nezávislá*, tzn. jejich obsah zůstává nezměněn po vypnutí počítače a robootování systému.

Soubor je pojmenovaná kolekce příbuzných informací, která je uložena na odkládacím zařízení. Z uživatelského hlediska je soubor nejmenší částečka logického odkládacího zařízení. Tzn. data nemohou být uložena na odkládací prostor, pokud nejsou v souboru. Obecné soubory reprezentují programy a data. Datové soubory mohou být numerické, alfabetické, alfanumerické nebo binární. Soubory mohou mít volnou formu, jako např. textové soubory, nebo mohou mít přísné uspořádání.

V globálu je soubor sekvence bitů, bytů, řádek nebo záznamů, jejichž význam je dán tvůrcem a uživatelem souboru. Koncepce souboru je proto neobyčejně obecná.

Informace v souboru jsou definované jeho tvůrcem. Množství rozdílných druhů informací musí být uloženo v souborech: zdrojové texty programů, objektové formy programů, spustitelné programy, číselná data, text, záznamy, grafické obrázky, zvuky atd.

Každý soubor má jistou definovanou strukturu příslušnou jeho typu. *Textový soubor* je sekvence znaků organizovaná do řádek, odstavců a stránek; *zdrojový soubor* je sekvence procedur a funkci; *objektový soubor* je sekvence bytů organizovaná do bloků srozumitelných linkeru; *spustitelný soubor* je sekvence kódových sekcí (stránek, segmentů), které dokáže loader zavést do paměti a spustit.

2.1 Atributy souborů

Soubor je pojmenován pro pohodlí lidských uživatelů a ti se na něj odvolávají prostřednictvím jeho jména. Jméno souboru je většinou řetězec znaků, jako např. "priklad.c". Některé systémy rozlišují mezi velkými a malými písmeny ve jménech souborů, zatímco jiné je přijímají jako ekvivalentní. U dalších systémů lze tuto vlastnost nastavit.

Je-li soubor pojmenovaný, stává se nezávislý na procesu, uživateli a také systému, který ho vytvořil. Jeden uživatel může například vytvořit soubor "priklad.c", zatímco jiný uživatel

může editovat tento soubor po zadání jeho jména. Vlastník souboru ho může uložit na floppy disk nebo magnetickou pásku a může tento soubor přečíst na jiném systému, kde tento soubor může být stále pojmenován "priklad.c".

Soubor má v různých systémech různé atributy, typicky však tyto:

- **Jméno**. Symbolické jméno souboru v lidsky čitelné podobě.
- **Typ**. Tato informace je potřebná v systémech, které podporuji různé typy souboru. Může sloužit pro přiřazení standardní aplikace, kterou je soubor zpracováván.
- **Lokace**. Ukazatel na zařízení a na umístění souboru na tomto zařízení.
- **Velikost**. Součástí tohoto atributu je aktuální velikost souboru (v bytech, slovech nebo blocích) a jeho maximální možná velikost.
- **Ochrana**. Informace o ochraně přístupu k souboru definují, kdo ho může číst, kdo spouštět, kdo do něj zapisovat atd.
- **Datum, čas a uživatelská identifikace**. Tyto informace mohou být uloženy po vytvoření (1), poslední modifikaci (2) a posledním užití (3) souboru. Tato data mohou být užitečná pro ochranu, bezpečnost a monitoring systému.

Informace o všech souborech jsou uloženy v adresářové struktuře vytvořené na odkládacím zařízení. Tyto záznamy mohou zabírat od 16 po více než 1000 bytů na každý soubor. V systémech s mnoha soubory může velikost adresáře dorůstat megabytových rozměrů. Protože adresáře, stejně jako soubory, musejí být energeticky nezávislé, musejí být uloženy na odkládacím zařízení a do paměti zaváděny část po části, podle potřeby.

2.2 Operace se soubory

Soubor je *abstraktní datový typ*. Pro důkladnou definici souboru musíme zavést operace, které mohou být se soubory prováděny. Operační systém provádí systémová volání pro vytvoření, zápis, čtení, přemístění v souboru, smazání a vypuštění souboru. Upřesněme, co musí OS provádět během každé z těchto šesti elementárních operací. Potom pro nás bude velmi jednoduché pochopit, jak mohou být implementovány různé příkazy vyšší úrovně, jako např. přejmenování souboru apod.

- **Vytvoření souboru**. K vytvoření souboru jsou nutné dva kroky. Nejdříve je třeba najít pro soubor dostatečně velký prostor na odkládacím zařízení. Jak tento prostor alokovat si popíšeme v kapitole 8.3. Za druhé musí být vytvořen záznam pro nový soubor v adresáři. Do tohoto záznamu je mj. uloženo jméno souboru a jeho uložení v systému souborů.
- **Zápis souboru**. K zápisu souboru je prováděno systémové volání, které vyžaduje specifikaci jména souboru, informace, která má být do souboru uložena a místo v souboru, kam se má ukládat. Po přijetí jména souboru systém vyhledá adresář a v něm informaci o umístění souboru. Systém musí uložit *ukazatel zápisu* na místo v souboru, kam má být zápis proveden. Ukazatel zápisu musí být správně nastaven kdykoli má dojít k zápisu do nějakého souboru.
- **Čtení souboru**. Ke čtení souboru je prováděno systémové volání, které vyžaduje specifikaci jména souboru a informaci o tom, kde v paměti je uložen následující blok souboru. Opět je vyhledán patřičný adresář a v něm patřičný záznam, podle něj je nastaven *ukazatel čtení* na místo, od kterého má být provedeno následující čtení. Kdykoli má nastat čtení souboru, je aktualizována pozice ukazatele čtení. Protože většinou je soubor buď čten, nebo zapisován, mnoho systémů užívá pouze jeden ukazatel *ukazatel aktuální pozice*. Obě operace, čtení i zápis, užívají týž ukazatel, čímž je šetřen prostor a redukována složitost systému.

- **Přemístění v souboru**. Je vyhledán adresář a v něm patřičný záznam a ukazatel aktuální pozice je nastaven na patřičnou hodnotu. Přemístění pozice příští vstupní nebo výstupní operace v souboru nepotřebuje volat žádnou I/O operaci. Tato akce je známa jako *seek.*
- **Smazání souboru**. Abychom mohli smazat soubor, musíme vyhledat adresář, ve kterém soubor leží. Potom uvolníme všechen úložný prostor, který byl souboru přirazen (ten může byt ihned přirazen jiným souborům jiných uživatelů) a smažeme položku záznam o daném souboru v adresáři.
- **Vypuštění souboru**: Existují situace, kdy uživatel chce ponechat nastaveny všechny atributy souboru a pouze vyprázdnit jeho obsah. Spíše než nutit uživatele soubor smazat a potom opětovně vytvořit, umožňuje tato funkce zachovat všechny atributy soubory (s výjimkou velikosti), která je nastavena na nulu.

Těchto šest základních operací představuje minimální sadu nutných instrukcí. Další potřebné příkazy, jako jsou např. připojení nových dat na konec souboru (*append*) a přejmenování souboru (*rename*) mohou být implementovány kombinací jiných operací se soubory. Např. operace kopie (*copy*) souboru, nebo kopírování souboru na jiné I/O zařízení (tiskárna, páska, displej atd.) může být definována jako vytvoření nového souboru a čtení starého a zápis do nového souboru.

Potřebujeme také sadu operací, které umožní uživateli číst a nastavovat atributy souboru. Např. potřebujeme operaci, která umožní uživateli zjistit velikost souboru nebo operaci, která mu dovolí nastavit vlastníka souboru.

Většina operací se soubory v sobě zahrnuje vyhledání patřičného záznamu v adresáři. Pro zjednodušení (aby nebylo nutné při každém přístupu k souboru soubor v adresáři vyhledávat) implementují některé OSy jako první operaci se souborem operaci *open*. OS si pak vede jen malou tabulku obsahující informace o všech otevřených souborech (*Tabulka otevřených souboru – Open-File Table*). Jestliže je potom prováděna další operace s otevřeným souborem, užije se index do této tabulky a neprovádí se žádné vyhledávání. Pokud již nebude soubor dále aktivně využíván, proces užije operaci *close* a OS vymaže záznam o souboru v tabulce otevřených souboru.

Některé systémy provádí operaci *open* automaticky, když je soubor poprvé použít. Operace *close* je potom užita v okamžiku ukončení procesu, díky kterému byla vyvolána operace *open*. Přesto mnoho systémů vyžaduje explicitní otevření souboru programem pomoci systémového volání *open* před tím, než bude soubor užit. Operace *open* vyžaduje jméno souboru. Potom vyhledá požadovanou položku v adresáři a zkopíruje ji do tabulky otevřených souborů v případě, že bezpečností aspekty umožňují soubor užít. Volání *open* většinou vrací ukazatel do tabulky otevřených souborů. Tento ukazatel, ne tedy jméno souboru, je užit při všech I/O operacích, čímž je zamezeno jakémukoli dalšímu hledání v adresářích a je zjednodušeno volání dalších operací.

Implementaci systémových volání *open* a *close* v multiuživatelských systémech, jako je např. Unix, je poněkud komplikovanější. V těchto systémech může totiž více uživatelů chtít současně přistupovat k témuž souboru. Nejčastěji má systém interní tabulky dvou úrovní. Má tabulku per-proces, ve které jsou všechny soubory, které mají otevřené jednotlivé procesy. Je v ní uložen např. ukazatel do souboru, kde bude provedena příští operace *write* nebo *read*.

Každá položka v tabulce per-proces má také ukazatel do systémové tabulky otevřených souborů. Tato tabulka obsahuje informace o souborech, které jsou nezávislé na procesech, tj. např. umístění souboru na disku, bezpečnostní informace a velikost souboru.

Je-li soubor otevřen nějakým procesem a jiný proces na něj provede volání *open*, je přidána další položka do tabulky per-proces s ukazatelem na soubor a do systémové tabulky otevřených souborů. Systémová tabulka otevřených souborů mívá většinou *čítač otevření,*

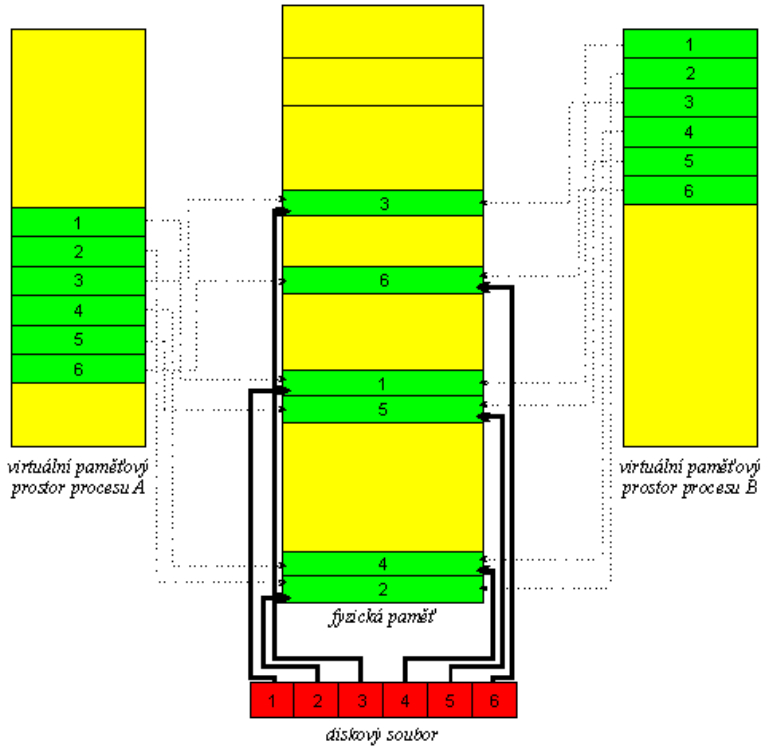
který udává počet procesů, které mají daný soubor otevřený. Každé volání *close* sníží hodnotu *čítače otevření* o 1. Je-li pak hodnota *čítače otevření* 0, není již soubor užíván žádným procesem a jeho položka je odebrána z tabulky otevřených souborů. Systém tedy udržuje nejrůznější informace asociované s otevřenými soubory:

- **Ukazatel do souboru (File pointer).** Systémy, které nemají implementovánu relativní adresu ve voláních **read** a **write**, musí udržovat ukazatel příštího čtení nebo zápisu v souboru. Každý proces, který má otevřený daný soubor má vlastní ukazatel do souboru, takže tento ukazatel nemůže být uchováván spolu s atributy souboru.
- **Čítač otevření.** Je-li soubor uzavřen, musí OS uvolnit položku v tabulce otevřených souboru. Protože soubor může být otevřen více procesy, systém s uvolněním položky z tabulky musí čekat, dokud poslední z nich soubor neuzavře. Tento čítač uchovává počet procesů, které mají daný soubor otevřený. Je-li jeho hodnota 0, může být položka z tabulky otevřených souborů smazána.
- **Umístěni souboru na disku**. Většina operací chce modifikovat obsah souboru. Informace potřebné k alokaci souboru na disku jsou uchovávány v paměti, aby nebylo nutno číst informace z disku před každou operací.

Některé OSy provádí mechanismy pro víceprocesový přístup k souboru, aby mohly být sekce souboru sdíleny více procesy, a to tím, že mapují jednotlivé sekce souboru do virtuální paměti. Tato funkce se nazývá *memory mapping* (*mapování souboru do paměti*) a dovoluje asociovat část adresového prostoru virtuální paměti se sekcemi souboru. Čtení a zápis do této oblasti paměti představuje čtení a zápis do souboru, což značně ulehčuje užití souboru.

Uzavření souboru vyvolá uložení všech virtuálně namalovaných sekcí souboru zpět na disk a vymazání souboru z virtuální paměti procesu. Různé procesy mohou mapovat týž soubor ve virtuální paměti na týchž stránkách a je tak umožněno sdílení souboru.

Zápis do souboru libovolným procesem modifikuje sdílená data ve virtuální paměti a může ho provést libovolný proces, který má nemapovány sekce sdíleného souboru do své virtuální paměti. Vzpomeneme-li našich znalosti o virtuální paměti, musí být jasné, jak je implementováno sdílení sekcí souboru ve virtuální paměti. Virtuální paměť každého procesu, který sdílí soubor, obsahuje odkazy na stejné stránky fyzické paměti – na stránky, ve kterých je uložena kopie sdíleného souboru. Tento mechanismus ilustruje následující obrázek. Přístup procesu ke sdílenému souboru je samozřejmě koordinován systémem, aby byl zajištěn vzájemně jedinečný přístup k souboru.



Obrázek 1 - Mapování souboru do virtuální paměti

2.3 Struktura souboru

Typy souboru mohou být užity také k identifikaci jejich vnitřní struktury. Jak již bylo uvedeno, zdrojový a objektový soubor mají odlišné struktury, které očekává program, jež s nimi pracuje. Další soubory musí mít strukturu vyžadovanou přímo operačním systémem. Např. OS vyžaduje určitou strukturu spustitelného souboru, ze které je mu jasné, kam zavést program do paměti a kde je první instrukce tohoto programu.

Některé OS rozšiřují tuto myšlenku na celou množinu systémem podporovaných struktur souborů a množinu speciálních operací, které s těmito soubory manipulují. Tato myšlenka s sebou přináší určitou nevýhodu v těžkopádnosti přílišné velikosti kódu operačního systému. Pro každou podporovanou strukturu souboru musí OS obsahovat kód, který s touto strukturou umí pracovat. Další problémy mohou nastat, podporuje-li OS kompletně všechny struktury všech souborů. Nová aplikace vytvořená pod takovýmto OS přinese problémy v případě, že bude vyžadovat organizaci dat v souboru způsobem, který OS nezná a nepodporuje.

Uvažujme např. OS, ve kterém existují pouze dvě struktury souboru: textový soubor (sekvence ASCII znaků oddělovaná CR a LF) a spustitelný binární soubor. Jestliže chci jako uživatel definovat v takovémto systému krytovaný soubor (pro zabezpečení jeho obsahu před nepatřičnými uživateli), nenajdeme žádnou vhodnou strukturu pro takovýto soubor. Krytovaný soubor není pouhý ASCII text, ale jedna se spíše o náhodnou sekvenci bitů – tedy binární soubor, který není spustitelný. Chci-li krytování provést v tomto systému, musím systémovou podporu struktury souboru buď obejít nebo užít spatně, případně modifikovat kryptovací algoritmus či od krytování úplně upustit.

Některé OS definují a podporují určitý minimální počet struktur souborů. Je tomu tak například u OS Unix, MS DOS aj. Unix bere každý soubor jako sekvenci 8mi bitových bytů a nijak tyto byty na úrovni OS neinterpretuje. Toto schéma zajišťuje maximum flexibility, ale minimální podporu. Každý aplikační program musí obsahovat kód pro podporu struktury „svých“ vstupních souborů. Přesto všechny OS musí podporovat minimálně jednu strukturu souboru – spustitelný soubor, který OS musí být schopen zavést do paměti a spustit.

Jiný příklad podpory struktur souboru můžeme naleznout u OS počítačů Macintosh, který kromě struktury spustitelného souboru podporuje ještě další dvě: *resource fork* a *data fork*. Resource fork obsahuje uživatelsky závislé informace, např. texty v tlačítkách programu. Zahraniční uživatel pak může vyžadovat tyto texty ve svém rodném jazyce. Potom stačí užít nástroje OS Macintosh, který umožňuje modifikaci dat v resource forku.

Data fork obsahuje kód programu a data, která uživatel nemůže modifikovat. K uskutečnění stejné úlohy pod Unixem či MS DOSem, musí *programátor* přepsat a rekompilovat zdrojový text programu, nebo vytvořit vlastní uzivatelsky modifikovatelný soubor.

Podstatou této ukázky je poukázat na výhodnost toho, když OS podporuje struktury souboru, které jsou často používané a tím ušetřit programátorům značnou námahu.

2.4 Vnitřní struktura souboru

Najít interně nějaký offset uvnitř souboru může být pro OS komplikované. Připomeňme, že diskové systémy mají často definovanou všeobecně známou velikost bloku dat určenou velikostí sektoru. Všechny diskové I/O operace pracují v jednotkách jednoho bloku (fyzický záznam) a všechny bloky jsou stejné velikosti. Je nepravděpodobné, že velikost fyzického záznamu bude přesně stejná jako velikost požadovaného logického záznamu. Logické záznamy jsou navíc proměnné délky. *Packing* (*uložení*) logických záznamů do fyzických je možným řešením tohoto problému.

Např. OS Unix definuje jednoduše všechny soubory jako sled bytů. Každý byte v souboru je individuálně adresovatelný pomocí jeho offsetu od začátku (nebo konce) souboru. V tomto případě je délka logického záznamů 1 byte. OS podle potřeby automaticky ukládá a vybírá tyto logické záznamy z fyzických diskových bloků (řekněme velikosti 512 B na blok).

Délka logického záznamu, délka fyzického záznamu a technika packingu určují, kolik logických záznamů se vejde do jednoho fyzického bloku. Uložení může provést buď uživatelský program, nebo operační systém.

V jiném případě může být soubor definován jako sled bloků. Všechny základní I/O operace pracují na těchto blocích. Konverze logického záznamu do fyzického bloku je potom relativně jednoduchý softwarový problém.

Poznamenejme, že přidělování diskového prostoru po blocích má za následek částečné nevyužití posledního bloku. Jestliže každý blok disku má velikost 512 B, potom soubor o velikosti 1949 B bude uložen do čtyř bloků o celkové velikosti 2048 B a v posledním bude 99 B nevyužitých. Celkový objem nevyužitých bloků bývá označován jako *interní fragmentace*. Všechny systémy souborů se ji snaží zabránit – větší velikost bloku znamená vetší interní fragmentaci.

Vzhledem k tomu, že nejčastěji je jako odkládací zařízení používán pevný disk (splňuje podmínku energetické nezávislosti), připomeňme, že jde o zařízení blokové. To znamená, že nejmenší jednotka dat, se kterou disk dokáže pracovat je jeden blok. Je-li třeba změnit na disku jeden jediný bit, znamená to načtení celého bloku do paměti počítače, modifikaci tohoto bitu a následně opět zápis celého bloku zpět. Reálná velikost fyzického bloku u starších disků typu HDD je obvykle 512 B, u novějších disků HDD a SSD to je obvykle 4 KB. Logický blok se může skládat z několika bloků fyzických (počet je většinou mocnina dvou).

3. METODY PŘÍSTUPU K SOUBORŮM

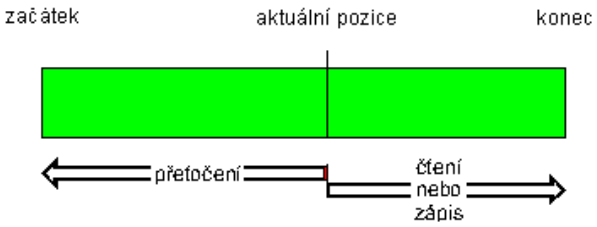
Soubory uchovávají informace. Jestliže má být taková informace užita, musí být přistoupeno k souboru, který ji obsahuje a tento soubor musí být zaveden do paměti. Existuji nejrůznější cesty, jak získat informaci uloženou v souboru. Některé systémy podporuji pouze jednu

metodu přístupu, zatímco jiné, jako např. systémy IBM podporuji vetší počet různých přístupových metod a výběr té pravé při přístupu k souboru je závažný problém.

3.1 Sekvenční přístup

Sekvenční přístup je nejjednodušší metoda přístupu. Informace v souboru jsou zpracovávány jeden záznam za druhým. Tento přístup je velice častý. Užívají jej např. překladače, editory apod.

Operace prováděné na souboru jsou čtení a zápis. Operace čtení čte následující část souboru a automaticky posune ukazatel následující I/O operace. Stejně tak append, operace zápis provede zápis informaci na konec souboru a automaticky posune ukazatel následující I/O operace na nový konec souboru. V některých systémech program může přeskočit dopředu nebo dozadu o *n* záznamů, pro nějaké celočíselné *n* (nejčastěji pro *n* = 1). Sekvenční přístup je vidět na následujícím obrázku. Je založen na modelu pásky se souborem, která pracuje jako zařízení se sekvenčním přístupem.



Obrázek 2 - Sekvenční přístup k souboru

Vhodným příkladem je soubor obsahující neformátovaný text, kde jednotlivými záznamy jsou řádky textu. Každý řádek je ukončen speciálním znakem (CR, LF nebo CR+LF – závisí na konkrétním operačním systému). Tento speciální znak pak slouží pro oddělení jednotlivých záznamů, jinými slovy určuje množství dat při čtení souboru. Při zápisu je za právě zapsaný záznam doplněn automaticky. Výhodou sekvenčního přístupu k souboru je možnost různé délky jednotlivých záznamů v souboru.

3.2 Přímý přístup k souboru

Jinou metodou je *přímý přístup* (nebo *relativní přístup*). Soubor sestává z *logických záznamů* pevné délky, což umožňuje programu číst nebo zapisovat záznamy rychle, ne v pořadí. Metoda přímého přístupu je založena na diskovém modelu souboru, protože disky umožňují náhodný přístup k libovolnému bloku souboru. Při přímém přístupu je soubor představován očíslovanou sekvencí bloků nebo záznamů.

Přímý přístup k souboru umožňuje číst nebo zapisovat libovolný blok souboru. Tzn., můžeme přečíst blok 14, potom blok 53 a na závěr zapsat blok 7. Nejsou tu žádná omezení na pořadí čtení nebo zápisu.

Soubory s přímým přístupem umožňuji rychlý přístup k velkému množství informací. Takto jsou často implementovány databáze. Když dorazí dotaz na nějakou položku databáze, vypočítáme, který blok požadovanou informaci obsahuje, a potom přečteme přímo požadovaný blok pro zjištění dotazovaného údaje.

Uvažujme například systém rezervace letenek. Můžeme uložit všechny informace o určitém letu (např. letu 713) do bloku identifikovaného číslem letu. Potom počet volných sedadel v letu 713 je uložen v bloku 713. K uložení dalších informací většího rozsahu, jako např. jmen cestujících, můžeme užít hashovaní funkci na jména cestujících nebo užít nějaké indexování k určení čísla bloku pro zápis a vyhledávání.

Operace se soubory musí být modifikovány tak, aby obsahovaly číslo bloku jako svůj parametr. Tj. užíváme funkci *read n*, kde *n* je číslo načítaného bloku spíše než *read next* a stejně tak *write n* namísto *write next*. Alternativní řešení je ponechat operace *read next* a *write next* s jejich původním významem a přidat operaci *position file to n*, kde *n* je číslo bloku. Potom má sekvence příkazů *position file to n*, *read next* stejný význam jako operace *read n*.

Číslo bloku, které používá uživatel v interakci s operačním systémem, se nazývá *relativní číslo bloku*. Relativní číslo bloku je relativní index bloku od začátku souboru. Relativní číslo prvního bloku souboru je tedy 0, dalšího 1 atd., zatímco absolutní adresa prvního bloku na disku může být 14703 a druhého 3192. Užití relativních čísel bloků umožňuje operačnímu systému rozhodnout, kde bude soubor na disku fyzicky uložen (alokační problém) a pomáhá zamezit uživateli přistupovat k blokům disku, které neobsahuji část jeho souboru. Některé OSy definují relativní číslo prvního bloku 0, jiné 1.

Je-li logický záznam délky *D*, potom žádost o záznam *N* je převeden na I/O žádost o *D* bytů od místa *D*(N-1)* od začátku souboru. Neboť jsou logické záznamy pevné délky, je velmi jednoduché zapsat, přečíst nebo vymazat záznam.

Ne všechny OSy podporují jak sekvenční, tak přímý přístup k souboru. Některé umožňuji pouze sekvenční, jiné pouze přímý přístup. Jiné systémy vyžadují, aby v okamžiku vytvoření souboru bylo dopředu definováno, zda se bude jednat o soubor s přímým nebo sekvenčním přístupem a potom je možno přistupovat k souboru pouze způsobem definovaným v okamžiku jeho vytvoření.

Poznamenejme, že je velmi jednoduché simulovat sekvenční přístup na soubor s přímým přístupem. Jestliže jednoduše definujeme proměnnou *ap*, která udává aktuální pozici v souboru, potom je možno simulovat sekvenční operace tak, jak ukazuje následující tabulka:

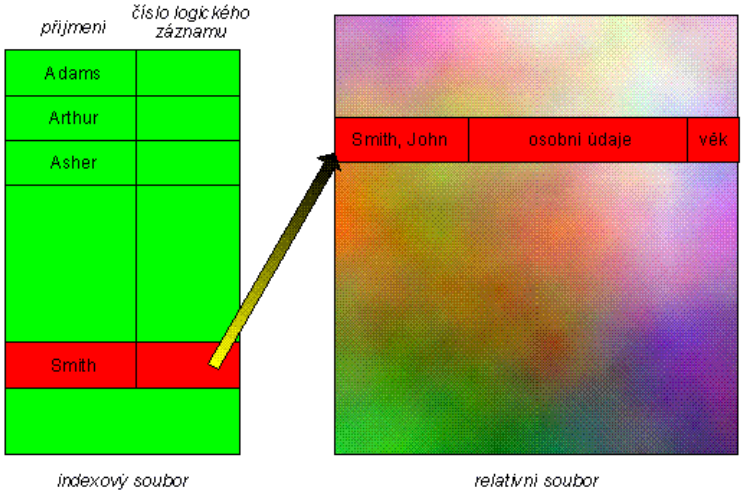
Sekvencni operace	Implementace pri primem pristupu
reset	*ap* := 0;
read next	read ap ap := ap + 1;
write next	write ap ap := ap + 1;

3.3 Další metody přístupu

Další metody přístupu k souboru mohou být vytvořeny na základě metody přímého přístupu. Tyto metody většinou spočívají v konstrukci *indexu* pro každý soubor. Index, stejně jako rejstřík na konci knihy obsahuje ukazatele na různé bloky. Pro nalezení nějaké položky v souboru nejdříve prohledáme index a potom užijeme ukazatel pro přímý přístup vedoucí k nalezení položky.

Například, soubor maloobchodních cen knih může obsahovat ISBN kód knihy a k němu asociovanou cenu. Každá položka sestává z 13-ti místného ISBN kódu a třímístné ceny. Celkem každá položka zabírá 16 bytu. Jestliže náš disk má blok velikost 1024 bytů, můžeme do bloku uložit 64 položek. Soubor o 120 000 položkách zabere 2 000 bloků (tedy 2 miliony bytů). K uložení souboru uspořádaného podle ISBN, můžeme sestavit index, obsahující ISBN první položky v každém bloku. Tento index bude mít 2 000 položek po 13-ti číslech tj. 26 000 bytů a bude uložen v paměti.

K nalezení ceny určité knihy stačí prohledat index, potom budeme přesně vědět, ve kterém bloku se informace o ceně nachází, a tento blok užít. Popisovaná struktura umožňuje prohledávat velké soubory dat malým počtem I/O operací.



Obrázek 3 - Příklad indexového a realtivního souboru

U velkých souborů, může být i sám index příliš velký na to, aby byl uchováván v paměti. Možným řešením je vytvoření indexového souboru. Primární indexový soubor potom obsahuje ukazatele do sekundárního indexu v souboru s daty.

Např. IBM indexuje souboru metodou ISAM (Indexed Sequential Access Method), která užívá malý master index, který ukazuje na diskové bloky sekundárního indexu. Sekundární index ukazuje na konkrétní bloky dat. Soubory jsou ukládány uspořádané podle definovaného klíče. K nalezení určité položky je třeba nejprve binárně prohledat master index, který ukáže číslo bloku sekundárního indexu. Tento blok je načten, opět binárně prohledán a je nalezeno číslo bloku, který je sekvenčně prohledán. Při této organizaci přístupu k souboru je jeho libovolná položka nalezena maximálně po dvou přímých čteních. Následující obrázek ukazuje obdobnou situaci, jak je implementována v operačním systému VMS.

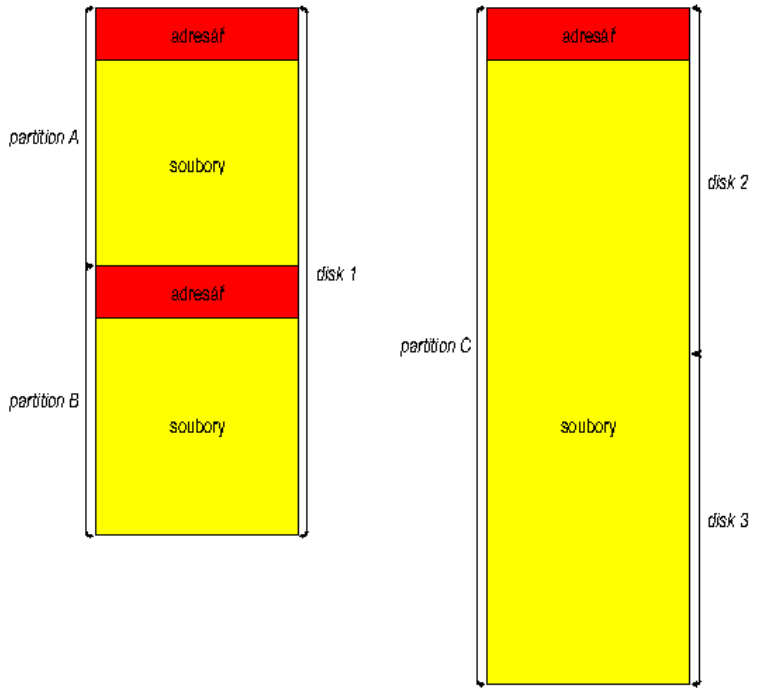
4. ADRESÁŘOVÁ STRUKTURA

Systém souborů počítače může být značně rozsáhlý. Velké systémy uchovávají tisíce souborů na mnohaset gigabytových discích. Pro management takového množství souborů je třeba je nějak organizovat.

Organizace souborů je většinou prováděna ve dvou úrovních. V první úrovni je systém souborů rozdělen na *partitions*, známých ve světě IBM jako *minidisks*, či jako *volumes* na počítačích PC či Macintosh.

Nejčastěji obsahuje každý disk v systému minimálně jednu partition, ve které jsou uloženy soubory a adresáře. Někdy jsou partitions užity k rozdělení většího disku na několik nezávislých oblastí. S každou oblastí systém zachází jako se samostatným odkládacím zařízením. Jiné systémy umožnují vytvářet vetší partitions než je fyzický rozsah disku a seskupovat několik disků do jednoho logického zařízení.

Z výše uvedených důvodů uživatel pracuje vždy jen s logickou adresářovou strukturou a logickým systémem souborů a zcela ignoruje problémy fyzické alokace disku souborem. Z tohoto pohledu představují partitions *virtuální disky*.



Obrázek 4 - Typická organizace systému souborů

Každá partition obsahuje informace o souborech na ní uložených, což je druhá úroveň systému souborů. Tyto informace jsou uloženy v záznamech *adresáře zařízení (device directory)* nebo *volume table.*

Adresář zařízení (častěji označován pouze slovem adresář) uchovává informace jako jméno, lokace, velikost a typ pro všechny soubory na zařízení. Následující obrázek ukazuje typickou strukturu systému souborů.

Na adresář je možno pohlížet jako na tabulku symbolů, která převádí jména souborů na odpovídající záznamy. Musíme být schopni vložit do adresáře novou položku, smazat některou stávající, vyhledat záznam v adresáři odpovídající souboru daného jména.

V následující kapitole rozebereme datové struktury, které mohou být užity při implementaci adresářové struktury. Nyní si popíšeme různá schémata definice logické struktury systému adresářů.

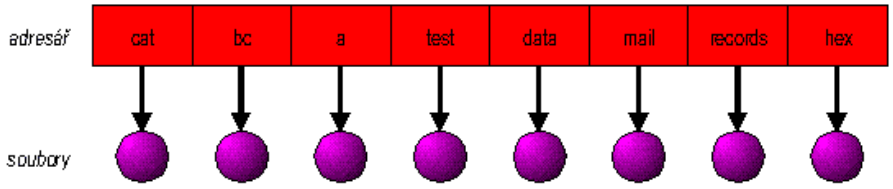
Definujme nejdříve operace, které potřebujeme s adresářem (jeho položkami) provádět:

- **Vyhledání souboru**. Musíme být schopni prohledávat adresářovou strukturu a každý soubor v ní. Protože soubory mají symbolická jména a podobná jména mohou označovat soubory, které jsou si blízké, je třeba mít možnost vyhledat všechny soubory, jejichž jméno obsahuje zadaný řetězec.
- **Vytvoření souboru**. Musíme mít možnost vytvořit nový soubor a uložit ho do adresáře.
- **Smazání souboru**. Jestliže už není soubor více zapotřebí, je třeba mít možnost ho z adresáře odstranit.
- **Výpis adresáře.** Musíme mít možnost získat seznam souborů, které jsou v adresáři a také vypsat všechny informace v záznamu každého souboru adresáře.
- **Přejmenovat soubor.** Protože jméno souboru reprezentuje uživatelem popsaný obsah souboru, musí existovat možnost změnit jméno souboru, jestliže se změní jeho obsah nebo způsob užití.

- **Pohyb systémem souborů**. Je žádoucí mít možnost přístupu do každého adresáře a ke každému souboru v něm uloženému. Pro bezpečnost systému souborů je také vhodné ukládat v pravidelných intervalech obsah a strukturu celého systému souborů. Toto ukládání často spočívá v kopírování všech souborů na magnetickou pásku. Touto technikou se vytvoří záložní kopie pro případ havárie systému, nebo prostě proto, že soubory už dále nebudou používány. V tomto případě mohou být soubory zkopírovány na pásku, smazány z disku, čímž vznikne prostor pro soubory jiné.

4.1 Adresář jedné úrovně

Nejjednodušší adresářová struktura je jednoúrovňová. Všechny soubory jsou uloženy v jednom adresáři. Tato struktura vyžaduje minimální podporu a je dobře pochopitelná (viz následující obrázek).



Obrázek 5 - Jednoúrovňová adresářová struktura

Jednoúrovňová struktura adresáře má zřejmé nevýhody v případě, když je v systému velké množství souborů, případně má systém více uživatelů. Protože všechny soubory jsou uloženy ve stejném adresáři, musí mít různá jména. Jestliže dva různí uživatelé nazvou své soubory *test*, je toto pravidlo jedinečnosti jména porušeno.

Přesto, že jsou jména souborů volena tak, aby vystihovala jejich obsah, jsou často limitována délkou. MS DOS umožňuje jméno souboru maximálně 11-ti znakové – včetně přípony reprezentující typ souboru – (tečka oddělující jméno a příponu není v adresářové položce uložena, doplňuje ji souborový systém). Unix připouští pro jméno souboru maximálně 255 znaků – tečka oddělující příponu je v tomto případě součástí jména.

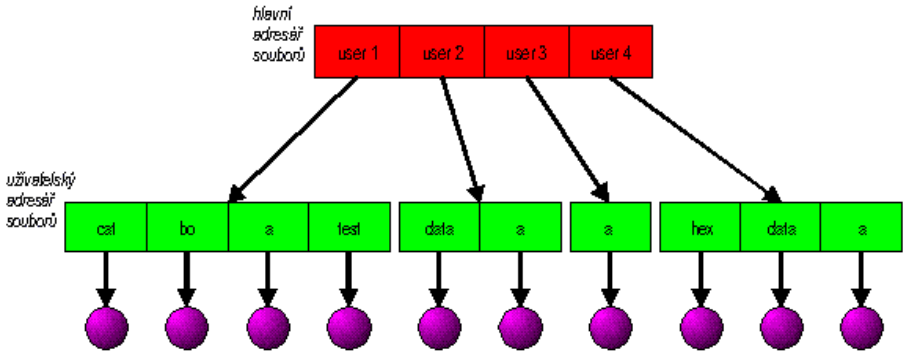
I v případě, že systém bude mít pouze jednoho uživatele a poměrně malý počet souborů, nastanou potíže se zapamatováním všech jmen všech souborů, aby nový soubor měl vždy jedinečné jméno. Není neobvyklé, že uživatel má v systému stovky souborů a stejně mnoho v systému jiném. Sledovat v této adresářové struktuře všechny soubory je zřejmě nemožné.

4.2 Dvouúrovňový adresář

Hlavní nevýhodou adresáře jedné úrovně je zmatek mezi soubory a jejich jmény od více uživatelů. Řešením tohoto problému je vytvořit samostatný adresář pro každého uživatele.

Ve dvouúrovňové adresářové struktuře má každý uživatel svůj vlastní *uživatelský adresář souborů (user file directory – UFD)*. Každý UFD má stejnou strukturu, ale obsahuje pouze soubory patřící danému uživateli.

Když se spustí uživatelská úloha nebo se uživatel přihlásí do systému, je prohledán *hlavní adresář souborů (master file directory – MFD)*. MFD indexuje podle uživatelského jména nebo čísla účtu a každá jeho položka ukazuje na patřičný UFD hledaného uživatele.



Obrázek 6 - Dvouúrovňová adresářová struktura

Jestliže uživatel požaduje nějaký soubor, je prohledán pouze jeho UFD. Různí uživatelé mohou tedy mít soubory stejných jmen (viz soubor *a* nebo *data*).

Vytvoří-li uživatel nový soubor, je prohledán pouze jeho uživatelský adresář, není-li v něm soubor stejného jména. Při mazání souboru OS prohledává pouze aktuální UFD, takže není možno smazat soubor stejného jména jiného uživatele.

Uživatelské adresáře musí být možno vytvářet i mazat. Nového uživatele počítače vytváří speciální systémový program, který vytvoří nový uživatelský adresář a přidá ukazatel na něj do hlavního adresáře. Spouštět tento program smí pouze administrátor systému. Způsob alokace diskového prostoru pro uživatelský adresář bude ještě diskutován.

Dvouúrovňová struktura adresáře řeší problém kolize jmen, nicméně má stále mnoho nedostatků. Tato struktura naprosto izoluje uživatele od všech ostatních. To je výhoda, jsou-li jednotliví uživatelé systému na sobě naprosto nezávislí. Je to však vážný nedostatek, pokud uživatelé chtějí spolupracovat na nějaké úloze a přistupovat společně k nějakým souborům. To u některých systémů prostě není možné a tyto systémy neumožňují jakýkoliv přístup k souborům jiného uživatele.

Jestliže OS připouští přístup k souborům jiného uživatele, uživatel musí mít možnost pojmenovat soubor v jiném uživatelském adresáři. K jednoznačné identifikaci souboru ve dvouúrovňové struktuře adresářů slouží jméno uživatele a jméno souboru.

Dvouúrovňová struktura adresářů představuje strom výšky 2. Kořenem toho stromu je hlavní adresář. Jeho potomkem jsou uživatelské adresáře a jejich potomky pak jednotlivé soubory. Soubory tvoří listy stromu. Specifikujeme-li jméno uživatele a jméno souboru, specifikujeme cestu stromem od kořene (hlavni adresář) k listu (specifikovaný soubor). Jméno uživatele a jméno souboru tvoří tedy *úplnou cestu*. Ke každému souboru vede úplná cesta. K adresování souboru z jiného adresáře je třeba znát plnou cestu k němu.

Jestliže např. uživatel A chce přistoupit k souboru jmena *test*, který má ve svém uživatelském adresáři, může se na něj odkazovat pouze jménem souboru *test*. Chce-li přistoupit k souboru *test* uživatele B (s uživatelským adresářem *userb*), musí se na něj odkazovat plnou cestou */userb/test*. Každý systém má svou vlastní syntaxi k pojmenovávání souborů uložených jinde než v uživatelově adresáři.

Syntakticky musí být také upravena specifikace partition, na které se soubor nachází. V OS MS DOS je partition specifikována písmenem následovaným dvojtečkou. Zde by výše diskutovaný odkaz mohl vypadat např.: *C:\userb\test*. Jiné systémy užívají i mnohem komplikovanější způsoby odkazování na soubory. V operačním systému VMS může např. odkaz na soubor login.com vypadat následovně: *u:(sst.jdeck)login.com;1*, kde *u* je jméno partition, *sst* je jméno adresáře, *jdeck* jméno podadresáře a *1* je číslo verze.

Další systémy vkládají jméno partition do plné cesty. První položka v cestě je jméno partition, za kterým následuje jméno adresáře a souboru, např. */u/pbg/test*, kde *u* specifikuje partition, *pbg* adresář a *test* jméno souboru.

Speciální situace nastává v případě systémových souborů. Tyto soubory, představující část OS (zavaděče, linkery, překladače, knihovny atd.) jsou také definovány jako soubory. Jestliže je OS zadán nějaký příkaz, soubor je načten zavaděčem do paměti a spuštěn. Mnoho příkazů představuje jméno souboru, který má být spuštěn. Protože ještě nemáme zavedeny žádné systémové adresáře, bude spouštěný soubor vyhledáván v uživatelském adresáři. Nyní můžeme řešit tento problém pouze tím, že nakopírujeme veškeré systémové soubory do každého uživatelského adresáře, což však představuje obrovské nároky na diskový prostor. (Jestliže by systémové soubory zabíraly 5 MB, tak při 12 uživatelích, by na jejich rozkopírování padlo 5 * 12 = 60 MB diskového prostoru).

Standardní řešení spočívá v jemné modifikaci vyhledávací procedury. Jeden speciální uživatelský adresář je definován tak, že obsahuje systémové soubory (např. user 0). Kdykoliv je přistupováno k nějakému souboru, OS prohledá uživatelský adresář. Jestliže je soubor nalezen, je užit. Pokud nalezen není, OS automaticky prohledá speciální uživatelský adresář systémových souborů. Pořadí prohledávaných adresářů při hledání souborů se nazývá *vyhledávací cesta (search path)*. Rozšířením této myšlenky může být, že vyhledávací cesta obsahuje neomezený seznam adresářů. Toho využívá Unix a MS DOS.

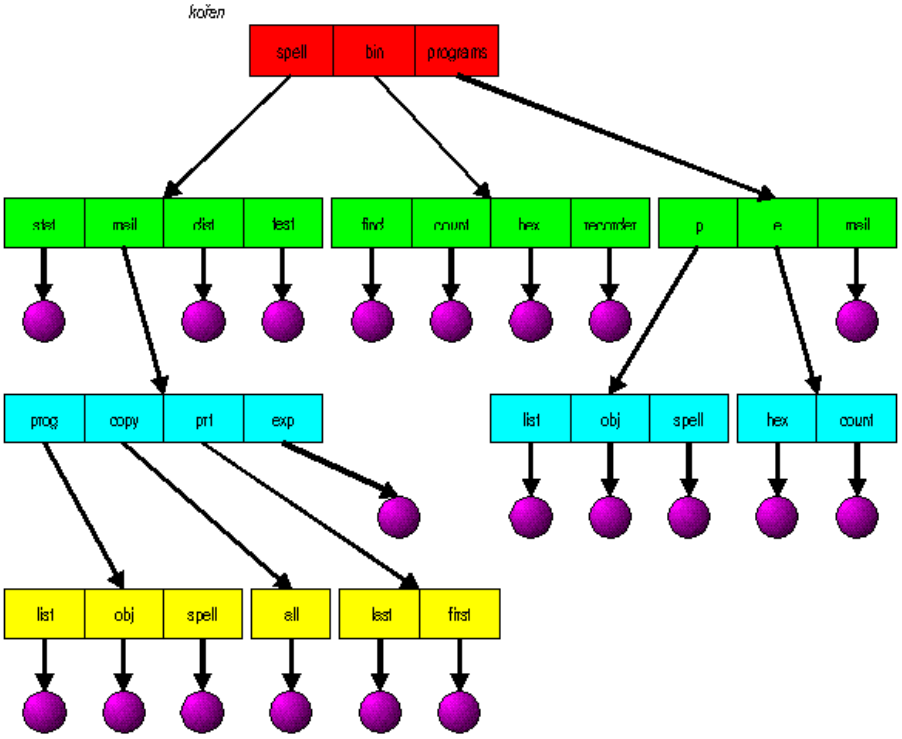
4.3 Stromová struktura adresářů

Pokud jsme ukázali dvouúrovňovou strukturu adresářů jako dvouúrovňový strom, je jen přirozené rozšířit adresářovou strukturu na strom libovolné výšky. Tato generalizace umožňuje uživatelům vytvářet své vlastní podadresáře a organizovat tak své soubory. MS DOS je např. strukturován jako strom. Strom je velmi častá adresářová struktura. Strom má kořenový adresář. Ke každému souboru vede jednoznačná cesta od kořenového adresáře přes všechny podadresáře až k cílovému souboru.

Adresář (nebo podadresář) obsahuje množinu souborů adresářů nebo podadresářů. Adresář je jednoduše soubor, se kterým OS nakládá speciálním způsobem. Všechny adresáře mají stejný vnitřní formát. Jeden bit v každé položce adresáře identifikuje, zda je daná položka soubor (0) nebo podadresář (1). K vytvoření a smazání adresáře užívá systém speciální volání.

Při běžné práci má každý uživatel definován *aktuální adresář (current directory)*. Aktuální adresář obsahuje většinou souboru týkající aktuálního zájmu uživatele. Jestliže se uživatel odkazuje na soubor, je prohledán aktuální adresář. Jestliže potřebný soubor není v aktuálním adresáři nalezen, potom musí uživatel, buď specifikovat vyhledávací cestu, nebo změnit aktuální adresář na adresář, který obsahuje požadovaný soubor.

Změna aktuálního adresáře na jiný je prováděna systémovým voláním, které vyžaduje jako parametr jméno adresáře a provede předefinování aktuálního adresáře. Uživatel může tedy změnit svůj aktuální adresář kdykoliv potřebuje. Po provedeném volání **change directory** všechna volání **open** hledají specifikovaný soubor v aktuálním adresáři.



Obrázek 7 - Stromová adresářová struktura

K inicializaci aktuálního adresáře uživatele dojde, když systém spustí uživatelovu úlohu, nebo když se uživatel přihlásí do systému. Operační systém tehdy prohledá soubor uživatelských účtů, aby nalezl položku týkající se daného uživatele (s informacemi o jeho uživatelském účtu). Mezi těmito informacemi je i ukazatel do uživatelova domácího adresáře. Tento ukazatel je zkopírován do lokální proměnné uživatele, která specifikuje uživatelův aktuální adresář.

Cesta k souboru může být dvojího druhu: *absolutní (absolute path name)* nebo relativní *(relative path name)*. Absolutní cesta začíná kořenovým adresářem a pokračuje až k souboru. Relativní cesta definuje cestu z aktuálního adresáře. Budeme-li uvažovat strukturu adresářů z minulého obrázku a je-li aktuální adresář *root/spell/mail*, potom relativní cesta *prt/first* odkazuje na stejný soubor jako absolutní cesta *root/spell/mail/prt/first*. Pro možnost procházení adresáři směrem vzhůru (k nadřazeným adresářům) existují adresáře speciálního jména, které je reprezentují: . (tečka) je tento, aktuální (pracovní) adresář a .. (dvě tečky) je adresář danému adresáři nadřazený.

Umožníme-li uživateli vytvářet podadresáře, umožníme mu vytvořit ukládací strukturu jeho souborů. Tak může vytvořit adresáře pro různé typy souborů (např. samostatný adresář si může uživatel vytvořit pro uložení textu této přednášky, další pro programy, který bude obsahovat zdrojové programy, adresář bin pro jejich spustitelné formy atd. atd.).

Zajímavým problémem ve stromové struktuře adresářů je problém smazání adresáře. Je-li adresář prázdný, můžeme jednoduše smazat jeho položku v nadřazeném adresáři. Pokud ovšem mazaný adresář prázdný není a obsahuje nějaké soubory nebo podadresáře, je možno s ním naložit dvěma způsoby. Některé systémy (např. MS DOS) nedovolí smazat adresář, který není prázdný – tj. uživatel musí nejprve smazat všechny soubory, které aktuální adresář obsahuje. Pokud adresář obsahuje podadresáře, je třeba tento algoritmus rekurzívně aplikovat i na ně. To může znamenat pro uživatele značnou námahu.

Druhé řešení, které umožňuje např. Unix příkazem **rm -r** je, že dojde ke smazání všech souborů i podadresářů, které mazaný adresář případně obsahuje. Toto řešení je na jedné straně příjemnější, nicméně představuje pro systém i značné nebezpečí, neboť jedním příkazem

může být nechtěně smazán celý strom adresářů nebo jeho významná část. Dojde-li k tomu, nastává nudná operace kopírování smazaných souborů z pásky na disk (pokud ovšem takovou pásku se zálohou máme :-).

Při stromovém uspořádáni adresářové struktury je velmi jednoduché pracovat se souborem jiného uživatele. Např. pokud uživatel B chce pracovat se souborem uživatele A, jednoduše zadá cestu k tomuto souboru, ať už relativní nebo absolutní. Případně může změnit svůj aktuální adresář na adresář uživatele A a přímo přistupovat ke všem jeho souborům.

Některé systémy umožnují uživatelům vytvářet vlastní vyhledávací cesty. Potom uživatel B může do této cesty definovat 1. svůj adresář; 2. systémový adresář; 3. adresář uživatele A. Pokud se jméno souboru v adresáři uživatele A neshoduje se jménem nějakého souboru v adresáři uživatele B nebo v systémovém adresáři, může se na tento soubor uživatel B odkazovat jednoduše pouze prostřednictvím jeho jména.

Poznamenejme, že u stromové uspořádané struktury adresářů může být cesta k souboru (např. spustitelnému) velmi dlouhá. Aby uživatelé mohli spouštět programy, aniž by si museli pamatovat cesty ke spustitelným souborům, bývá jejich vyhledávání automatizováno. MacOS např. obhospodařuje soubor zvaný 'Desktop File', který obsahuje jméno a cestu ke všem spustitelným programům v systému. Jestliže je k systému připojen nový disk resp. souborový systém, nebo je připojena síť, OS vyhledá v nových adresářích spustitelné soubory a aktualizuje patřičné informace. To je podpora pro dvojklik, který je v MacOS užit, tak jak již bylo zmiňováno. Dvojklik na ikonu souboru vyvolá načtení jeho atributu creator a v 'Desktop File' je potom vyhledána cesta k programu, který je spuštěn a otevře soubor, na který byl proveden dvojklik. Podobný mechanismus existuje i v jiných operačních systémech

4.4 Acyklický graf adresářů

Představme si dva programátory, kteří pracují na společném projektu. Soubory související s tímto projektem nechť jsou uloženy ve speciálním podadresáři, aby se nemíchaly s ostatními soubory obou programátorů. Protože jsou oba programátoři za projekt odpovědni stejnou měrou, oba chtějí, aby podadresář s projektem byl v jejich adresáři. Tento adresář musí být *sdílen (shared)*.

Sdílený adresář nebo soubor existuje v systému souborů na dvou nebo i více místech najednou. Poznamenejme, že sdílený soubor nebo adresář není totéž co dvě kopie souboru či adresáře. Pokud by programátor provedl změnu v jedné kopii, tato změna by se neprojevila v kopii druhé.

Je-li soubor sdílen, fyzicky existuje pouze *jednou*, takže všechny změny provedené libovolným uživatelem jsou okamžitě viditelné i všemi ostatními. Tento mechanismus je vhodný i pro sdílení adresářů. Nově vytvořený soubor ve sdíleném adresáři je viditelný i pro všechny ostatní.

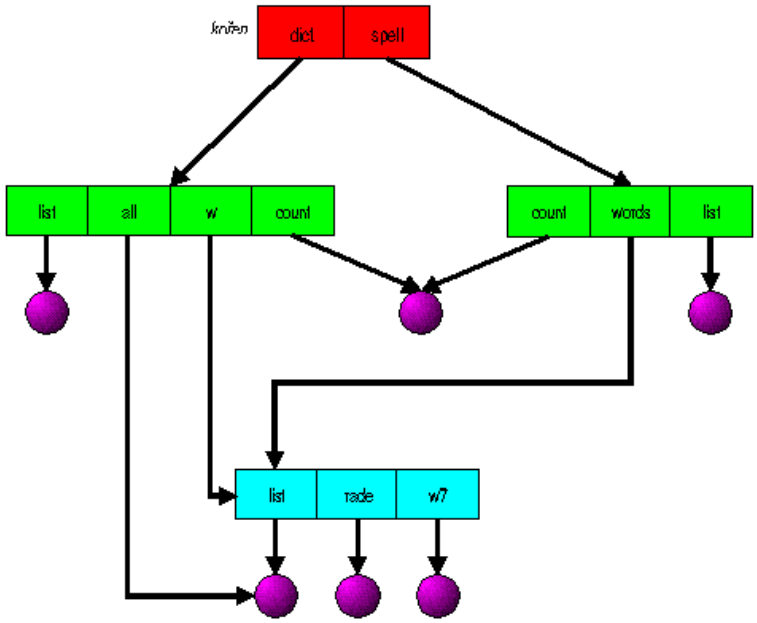
Stromová struktura adresářů sdílení neumožnuje. Strukturou, ve které je sdílení možné je *acyklický graf*. Týž soubor nebo podadresář může být ve dvou různých adresářích současně. Acyklický graf (tedy graf, ve kterém nejsou žádné kružnice) je přirozeným rozšířením stromové struktury adresářů.

U pracovní skupiny lidi mohou být potom všechny společné soubory umístěny ve sdíleném adresáři. Tento adresář je potom sdílen ve všech uživatelských adresářích všech členů pracovní skupiny.

Sdílené soubory a podadresáře mohou být implementovány nejrůznějšími způsoby. Častou cestou, kterou jdou i mnohé Unixy je vytvoření nové položky v adresáři, zvané *link*. Link je ve skutečnosti ukazatel na jiný soubor nebo adresář. Link může být implementován pomocí absolutní nebo relativní cesty (*symbolický link*).

Když dojde odkaz na soubor, prohledáme adresář. Patřičná položka je označena jako link a získáme z ní odkaz na skutečný soubor nebo adresář. Link užijeme k získání cesty ke skutečnému souboru. Linky je možno jednoduše identifikovat formátem položky adresáře, nebo je možno definovat je jako speciální typ v OS, který typy rozeznává.

Linky jsou ignorovány při přenosu stromu adresářů, aby byla zachována acyklická struktura systému.



Obrázek 8 - Acyklický graf adresářové struktury

Jiná implementace sdílených souborů spočívá v duplikaci všech informací o souboru ve všech adresářích, které soubor sdílí (nebo adresář). Tam jsou všechny položky identické. Tento přístup vede k identitě mezi originálem a linkem a to s sebou přináší určité obtíže při správě sdílených dat.

Acyklický graf adresářů je flexibilnější struktura než běžný strom adresářů, je však také mnohem komplexnější a je třeba při její implementaci vyřešit mnohé problémy. V takové struktuře může vést k souboru více absolutních cest. V důsledku toho mohou různá jména odkazovat na tentýž soubor. Jestliže potom chceme projít celý systém souborů (kvůli vyhledávání, vytvoření statistiky souborů nebo kvůli backupu systému), nemáme nejmenší zájem na tom, abychom sdílené struktury procházeli více než jednou.

Další problém s sebou nese mazání. Kdy můžeme skutečně uvolnit prostor alokovaný sdílenému souboru? Jedna možnost je provést to vždy, když některý uživatel "smaže" sdílený soubor. Tento postup však za sebou může zanechat ukazatele na neexistující soubor. A ještě hůře, jestliže zbylé ukazatele obsahují diskové adresy na prostor, který už obsadil jiný soubor, tyto ukazatele mohou odkazovat dovnitř úplně jiného souboru.

V systému, kde je sdílení implementováno pomocí symbolických linků, je řešení tohoto problému jednodušší. Smazání linku nijak neovlivní sdílený soubor, pouze je link odstraněn z adresáře. Jestliže je smazán sám soubor, je uvolněn prostor, který zabíral na disku a ponechány visící ukazatele. Mohli bychom tyto ukazatele vyhledat a smazat, ovšem pokud není udržován seznam těchto ukazatelů, bylo by hledání velmi obtížné a zdlouhavé. Stejně dobře můžeme tyto prázdné odkazy nechat být do doby, než bude proveden pokus o jejich užití. V této situace můžeme určit, že soubor, na který link odkazuje, již neexistuje a ukončit chybou rezoluci jména souboru. S odkazem na link je potom naloženo stejně jako s odkazem na neexistující soubor. (V tomto případě by měl tvůrce systému bezpečně vyřešit co dělat, je-

li soubor smazán a jiny soubor téhož jména opětovně vytvořen a je užit symbolický odkaz na původní soubor.) V Unixu symbolické linky přetrvávají a je na uživateli, jak se zařídí.

Jiný přístup k mazání linku je zachovat soubor, dokud nejsou smazány všechny linky na něj. Pro tento přístup musíme mít implementován mechanismus, který zjistí, zda právě mazaný linky je či není poslední. Můžeme udržovat seznam všech odkazů na soubor. Jestliže je vytvořen nový link, přidá se nová položka do seznamu. Pokud se link maže, smažeme položku ze souboru odkazu. K fyzickému smazání souboru může dojít, je-li jeho seznam linků prázdný.

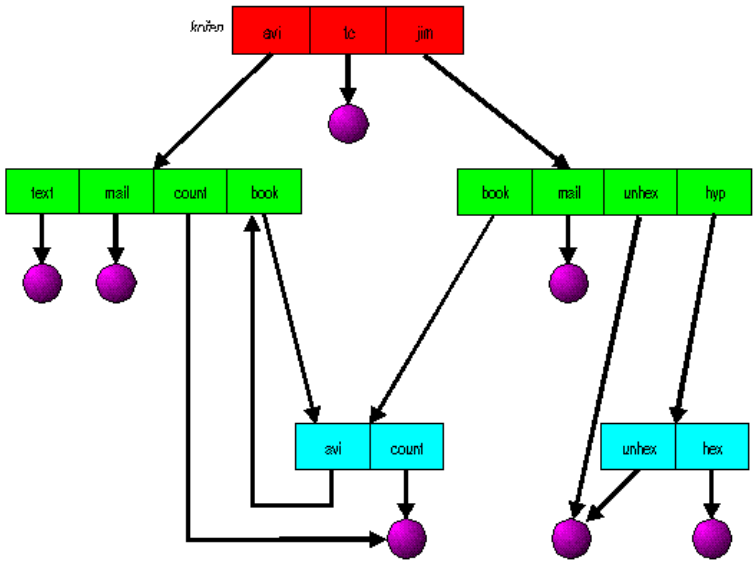
Potíž tohoto přístupu je v proměnlivé a potenciálně značné velikosti seznamu linků. Ve skutečnosti nemusíme uchovávat celý seznam odkazů, stačí znát jejich aktuální *počet*. Nový odkaz inkrementuje čítač linků, smazání odkazu ho dekrementuje. Je-li čítač roven 0, je možno soubor fyzicky smazat.

Unix implementuje tento přístup u nesymbolických linků (*hard links*), kde je čítač referencí součástí informačního bloku souboru, tzv. Inodu.

Aby se některé operační systémy vyhnuly výše uvedeným problémům, prostě sdílení souboru neimplementují (např. MS DOS).

4.5 Obecný graf adresářů

Vážný problém při užití acyklického grafu adresářů je zajistit, že se do něj nevetře žádná kružnice. Jestliže jsme začali s dvojúrovňovým adresářem a umožnili jsme uživatelům vytvářet podadresáře, vznikla stromová struktura adresářů. Je snadné nahlédnout, že vytvořením podadresářů se nijak nenaruší podstata stromové struktury. V okamžiku, kdy ale přidáme linky, situace se změní velice významně. Systém souborů tvoří obecný graf, tak jak je vidět na následujícím obrázku.



Obrázek 9 - Obecný graf adresářů

Výhodou acyklických grafů je relativní jednoduchost algoritmu průchodu grafem a snadné určení počtu linků na soubor nebo adresář. Potřebujeme se vyhnout vícenásobnému procházení sdílených oblastí v acyklickém grafu zejména z hlediska výkonu. Jestliže hledáme soubor ve sdíleném adresáři a nenalezneme ho, je zcela zbytečné, abychom ho díky dalšímu linku na něj prohledávali znovu. Byl by to ztracený čas.

Jestliže připustíme kružnice v grafu adresářové struktury, stejně tak chceme zabránit vícenásobné prohledávání sdílených oblastí ze zmiňovaných důvodů. Špatně vytvořený

algoritmus průchodu grafem vsak může velmi jednoduše skončit v neomezeném cyklu. Jednou z možností jak to řešit je stanovit maximum možných průchodů adresářem během prohledávání.

Další problém nastává, chceme-li zjistit, jestli je možné soubor smazat. V acyklickém grafu je 0 v čítači odkazu nutnou i postačující podmínkou k tomu, aby mohl být soubor smazán. Pokud se ovšem v grafu kružnice mohou objevit, čítač odkazů může být různý od 0 a přesto na soubor nepovede žádný jiný odkaz. Může totiž dojít k tomu, že adresář bude odkazovat sám na sebe. V tomto případě je nutné užít *inventarizaci volných míst (garbage collection)* ke zjištění, zda byl smazán poslední odkaz na soubor a soubor může být odstraněn z disku.

Garbage collection umožňuje procházet systémem souborů a označit vše, co může být užito. V důsledku toho označí vše, co není v prvním seznamu, jako volný prostor k užití. Obdobného mechanismu lze užít i k tomu, aby libovolný průchod či hledání v systému souborů navštívilo vše právě jednou. Garbage collection je ovšem velmi časově náročná a bývá užita jen velmi zřídka.

Garbage collection je nutná pouze v případě, že se v grafu mohou objevit kružnice. Acyklická struktura je tedy pro management mnohem přijatelnější. Jediný problém u ní je, zabránit tomu, aby nový link nevnesl do grafu kružnici. Jak zjistíme, že nový link uzavírá kružnici? Existují algoritmy, které v grafu kružnici objeví. Ty jsou ale časově náročné. Obecně, stromová struktura adresářů je mnohem častější, než struktura acyklických grafu.

5. OCHRANA

Jsou-li informace uloženy ve výpočetním systému, je důležité zajistit jejich ochranu před fyzickým poškozením (*spolehlivost – reliability*) a před neoprávněným užitím (*ochrana – protection*).

Spolehlivost je zajišťována především pravidelným zálohováním souboru. Mnoho počítačů má systémový program, který automaticky nebo na přání operátora kopíruje souboru z disku na pásku v pravidelných intervalech (jednou za týden, měsíc apod.).

Systém může být poškozen hardwarovým problémem (chyba při čtení nebo zápisu), výkyvem napětí, havárií čtecí hlavy, nečistotou, teplotními extrémy nebo vandalismem. Soubory mohou být smazány náhodně. Chyba softwaru, který obsluhuje systém souborů, může také vést ke ztrátě dat. Spolehlivostí systému se ještě budeme věnovat podrobněji.

Ochrana může být prováděna mnoha způsoby. U malého jednouživatelského systému je možno provádět ochranu pouze tak, že vyjmeme disketu z mechaniky a zamkneme ji ve stole. Ve víceuživatelských systémech je však seriózní ochrana zapotřebí.

5.1 Typy

Potřeba ochrany souborů má smysl pouze v případě, že uživatel může přistupovat k souborům jiných uživatelů. Pokud tuto možnost nemá, ochrana není zapotřebí. Jedna možnost absolutní ochrany je naprostý zákaz přístupu k cizím souborům. Použitelná je přibližně stejně jako druhý extrém, tj. naprosto uvolnit všechen přístup ke všem souborům. To co potřebujeme je *řízený přístup (controlled access)*.

Mechanismus ochrany provádí řízený přístup pomocí limitovaného typu přístupu k souboru. Přístup je umožněn nebo odepřen v závislosti na různých faktorech. Jeden z nich je požadovaný typ přístupu. Systém může rozlišovat mezi různými přístupy k souboru:

- **Čtení**. Čtení souboru.
- **Zápis**. Zápis do souboru, nebo jeho přepis.

- **Spuštění**. Zavedení souboru do paměti a jeho spuštění.
- **Přípis na konec.** Zápis nových informací na konec souboru.
- **Smazání**. Smazání souboru a uvolnění prostoru na disku pro další užití.
- **Výpis**. Výpis jména souboru a dalších informaci.

Další operace, jako přejmenování, kopírování nebo editování souborů mohou být také zvlášť řízeny. V mnoha systémech jsou však tyto operace vyšší úrovně implementovány systémovým programem, který využívá volání nižší úrovně a ochrana je prováděna pouze na této nižší úrovni. Kopírování může být implementováno jako sekvence požadavku čtení. V tomto případě uživatel s právem čtení souboru ho také může kopírovat, tisknout atd.

Již bylo navrženo mnoho mechanismů ochrany. Každý má své výhody a nevýhody a je třeba vybrat ten, který vyhovuje nejlépe zamýšleným aplikacím. Malý výpočetní systém s malým počtem uživatelů jistě nepotřebuje stejné zabezpečení jako velký sdílený počítač určený k vědeckým účelům. I problematika ochrany bude ještě diskutována podrobněji.

5.2 Seznam přístupů (Access List) a skupiny (Groups)

Nejčastější způsob ochrany souborů je vytvořit přístup závislý na identifikaci uživatele. Různí uživatelé mohou vyžadovat různý přístup k souboru nebo adresáři. Nejobecnější schéma implementace uživatelsky závislého přístupu k souborům je připojit ke každému souboru nebo adresáři seznam přístupů (*access list*), specifikující uživatelské jméno a typ přístupu povolený pro daného uživatele. Jestliže pak uživatel chce užít soubor, OS vyhledá patřičný záznam v seznamu přístupu asociovaném se souborem a podle něj bud přístup uživateli povolí nebo odepře.

Nejzávaznější nevýhoda tohoto přístupu je velikost seznamu přístupů. Jestliže bychom chtěli umožnit čtení souboru všem uživatelům, musíme do seznamu všechny zaznamenat s právem čtení. Z toho vyplývají dva nežádoucí důsledky:

- Tvorba access listu je pro správce nudná a zdlouhavá
- Adresář doposud pevné délky ji má nyní proměnlivou, což znesnadňuje management odkládacího prostoru

Řešením tohoto problému je užití minimalizované verze access listu. Nejčastější způsob zmenšení access listu spočívá v zavedení nejčastěji tří tříd přístupu uživatele k souboru. Tyto třídy jsou:

- **Vlastník (Owner)**. Uživatel, který vytvořil soubor, se stává jeho vlastníkem.
- **Skupina (Group)**. Množina uživatelů, kteří sdílí soubor, a potřebují k němu současný přístup, se nazývá skupina nebo pracovní skupina.
- **Ostatní (Universe)**. Tento vesmír tvoří všichni ostatní uživatelé systému.

Pro příklad uvažujme spisovatelku Šárku, která píše vědeckou publikaci. Využívá k tomu také práce tří doktorandů (Josefa, Davida a Jiřího). Text knihy je uložen v souboru s názvem *kniha*. Ochrana souboru kniha je potom následující:

- Šárka musí mít možnost provést se souborem kniha cokoliv.
- Josef, David a Jiří mohou soubor pouze číst a zapisovat do něj, nesmí mít možnost soubor smazat.
- Všichni ostatní uživatelé musí mít možnost soubor číst. Šárka chce, aby si uživatelé mohli knihu číst a vyjadřovat se k ní.

Pro provedení této ochrany musíme vytvořit novou skupinu, např. *text*, do které budou patřit Josef, David a Jiří. Jméno skupiny text musí být asociováno se souborem *kniha* a přístupová

práva musí být nastavena tak, jak jsme výše definovali. Takovéto skupiny uživatelů dnešní operační systémy běžně podporují.

Aby tato ochrana dobře fungovala, musí být kontrolována. Tuto kontrolu provádí různé operační systémy různě. V OSu Unix mohou být skupiny vytvářeny, modifikovány připadně rušeny pouze správcem systému. Ve VMS může mít každý soubor svůj access list (zde známý jako *access control list*), v němž je seznam uživatelů, kteří mohou k souboru přistupovat. Tento seznam může vytvořit a modifikovat vlastník souboru.

V minimalizované ochraně, tak jak ji zavádíme, jsou k definici ochrany třeba pouze tří položky (vlastník, skupina, ostatní). Tyto položky bývají kolekcemi bitů, které definují přístup. V Unix obsahuje každá položka 3 bity známé jako **rwx**, kde **r** znamená právo čtení, **w** právo zápisu a **x** právo spouštění. Jednoduchou aritmetikou lze ověřit, že pro zajištění ochrany každého souboru je třeba 9 bitů.

Vrátíme-li se k Šárce, tak ona jako vlastník souboru *kniha* bude mít nastaveny všechny tří bity, skupina *text* bity **r** a **w** a ostatní pouze bit **r.**

Na závěr dodejme, že toto minimalizované schéma ochrany není tak obecné jako *access list*. Uvažme, že by Šárka chtěla ukončit spolupráci s Josefem a vyřadit ho ze skupiny *text*. Při popsaném mechanismu ochrany to udělat nemůže.

5.3 Další cesty ochrany

Existují i jiné cesty k problému ochrany souborů. Jednou z nich je např. zpřístupnit každý soubor heslem. Stejně jako je přístup k počítači velmi často zabezpečen heslem, může tak být kontrolován i přístup ke všem souborům počítače. Jestliže jsou hesla vybírána vhodně a často měněna, je to bezpečná cesta k zajištění přístupu k souborům jen těm, kteří smějí a tudíž heslo znají.

Přesto má tento přístup řadu nevýhod. První, která asi čtenáře napadne je ohromné množství hesel, které si musí pamatovat uživatel takto zabezpečeného systému a tím se toto schéma stává poněkud nepraktickým. Pokud by jedním heslem byly chráněny všechny soubory, tak po kompromitaci tohoto hesla šťouravým uživatelem by všechna ochrana přišla vniveč.

Některé systémy (např. TOPS-20) umožnují chránit heslem podadresář. To je cesta středem obou výše uvedených extrémů a více méně řeší jejich nedostatky.

Ve víceúrovňové adresářové struktuře je třeba definovat nejen ochranu souborů, ale i podadresářů. Operace s adresáři, které je třeba chránit, jsou jiné povahy než operace se soubory. Je třeba kontrolovat vytváření a mazání souborů v adresáři. Stejně tak je třeba některým uživatelům zabránit jakémukoliv přístupu do adresáře (tedy i jeho výpisu). Někdy může být signifikantní i pouhé jméno souboru. Tedy v cestě, které adresuje požadovaný soubor, musí být umožněn přístup do všech podadresářů i k souboru. V systémech, kde může vést k souboru více cest (užitím acyklických nebo obecných grafů adresářů) může mít uživatel různé přístupy k témuž souboru podle toho, jakou cestu k němu zvolí.

6. SÉMANTICKÁ KONZISTENCE

Sémantická konzistence je důležité kritérium při vývoji systému souborů, který umožňuje jejich sdílení. Jedná se o následující problém. Když jeden uživatel změní obsah sdíleného souboru, jak se tato změna projeví u ostatních uživatelů?

Pro následující diskusi uvažujme, že sekvence užití souboru uživatelem (tedy jeho čtení či zápis) je vždy ohraničena příkazy **open** a **close**. Sekvenci příkazů mezi operacemi **open** a **close** nazvěme *užití souboru (file session)*. Dva nejčastější přístupy k sémantické konzistenci si ukážeme na následujících příkladech.

- Pokud uživatel zapíše něco do otevřeného souboru, je tento zápis okamžitě viditelný i pro všechny ostatní uživatele tohoto souboru, kteří ho mají tou dobou také otevřený.
- Je užito typu sdílení, kde uživatelé sdílí ukazatel do skutečného souboru. Tzn., posunutí ukazatele jedním uživatelem zaznamenají i všichni ostatní. Soubor zde existuje pouze jednou a na něj jdou všechny přístupy bez ohledu na jeho původu.

Tato sémantika vede k implementaci, kde soubor má jeden fyzicky obraz, ke kterému je přistupováno jedinečně. To může vést k tomu, že uživatelské procesy budou čekat, než budou moci ke sdílenému souboru přistoupit.

V multiprogramovém prostředí si mohou různé procesy konkurovat v získání konečného počtu zdrojů. Proces požaduje zdroje; jestliže tyto zdroje nejsou v danou chvíli dostupné, je proces přesunut do stavu *čekající*. Může se stát, že čekající proces svůj stav již nikdy nezmění, protože zdroje, na které čeká, drží jiné čekající procesy. Tato situace je nazývána *deadlock (zablokování)*. Částečně jsme se o této problematice zmínili v předcházející kapitole v souvislosti se semafory.

Snad nejlepší ilustrací zablokování můžeme nalézt v jednom kansaském zákonu ze začátku století: Jestliže se k sobě vzájemně blíží dva vlaky na křížení kolejí, oba musí zcela zastavit a žádný se nesmí opětovně rozjet, dokud druhý neodjede.

V této kapitole si ukážeme metody, které může užít operační systém pro řešení deadlocku. Poznamenejme přesto, že řada současných operačních systémů prevenci deadlocku neprovádí. Tyto mechanismy budou zřejmě doplněny časem, až bude problematika deadlocku aktuálnější. Některé vývojové trendy k této situaci jistě povedou: čím dál větší množství procesů na počítači, čím dál větší množství zdrojů (včetně více CPU) atd.

7. MODEL SYSTÉMU

Systém obsahuje konečný počet zdrojů, které mohou být distribuovány mezi konkurující si procesy. Zdroje jsou rozdělené do tříd dle typu a každá třída obsahuje určitý počet identických instancí. Takovou třídou je např. paměťový prostor, cykly CPU, soubory a I/O zařízení. Jestliže má systém 2 CPU, potom zdroj typu *CPU* má 2 instance. Stejně tak, např. zdroj typu *tiskárna* může mít 5 instancí apod.

Jestliže proces požaduje instanci určitého typu zdroje, jeho žádost uspokojí alokace jakékoliv instance daného typu. Pokud tomu tak není, instance nejsou identické a třída zdroje není definována korektně. Např. nechť systém obsahuje dvě tiskárny. Tyto dvě tiskárny mohou být zahrnuty do téže třídy, jestliže nezáleží na tom, která tiskárna tiskne který tisk. Přesto, jestliže jedna tiskárna je v devátém poschodí a druhá v přízemí, je zřejmé, že tyto dvě tiskárny navzájem ekvivalentní nejsou.

Proces musí o zdroj požádat před jeho užitím a musí ho po užití uvolnit. Proces může požadovat tolik zdrojů, kolik jich vyžaduje uskutečnění zpracovávané úlohy. Počet požadovaných zdrojů většinou nepřekročí celkový počet zdrojů v systému. Proces nemůže požadovat tři tiskárny, jsou-li v systému pouze dvě.

Při běžném vykonávání operace proces může užít zdroj pouze dle následující sekvence:

- **Dotaz**: Jestliže nemůže být dotaz vyplněn okamžitě (např. proto, že daný zdroj právě využívá jiný proces), potom dotazující proces musí čekat, než zdroje může nabýt.
- **Užití**: Proces může pracovat se zdrojem (např. je-li zdrojem tiskárna, proces na ni může tisknout)
- **Uvolnění**: Proces uvolní zdroj k užití procesy jinými.

Dotaz a uvolnění jsou systémová volání. Příkladem mohou být volání **dotaz** a **uvolnění zařízení**, **otevření** a **uzavření souboru**, **alokování** a **uvolnění paměti**. Dotaz a uvolnění jiných zdrojů mohou být provedeny pomocí operací *cekej* a *signal* na semaforu. Pročež OS před každým užitím zdroje kontroluje, zda daný proces vznesl dotaz na tento zdroj a jestli mu byl zdroj přidělen. Jedna systémová tabulka vede evidenci o každém zdroji, zda je volný nebo alokovaný, případně kterým procesem. Jestliže se proces dotazuje na zdroj, který je právě alokovaný, může být zařazen do fronty čekajících na tento zdroj.

Říkáme, že množina procesů je ve stavu zablokována, jestliže každý proces v množině čeká na událost, kterou může vyvolat pouze jiný proces z téže množiny. Událost, se kterou se zde budeme setkávat zejména je obsazení zdroje a jeho uvolnění. Zdrojem může být buď fyzické zařízení (tiskárna, magnetopásková mechanika, paměťový prostor a čas CPU) nebo logické zařízení (např. soubor, semafor a monitor). Ovšem i jiné typy událostí mohou vést k deadlocku (např. IPC facility).

K ilustraci deadlocku předpokládejme systém se třemi magnetopáskovými mechanikami. Nechť v něm existují 3 procesy a každý užívá jednu mechaniku. Jestliže za daného stavu každý proces vznese dotaz na další mechaniku, všechny tři se dostanou do deadlocku. Každý z nich čeká na událost „magnetopásková mechanika je uvolněna“, která může být vyvolána pouze jiným čekajícím procesem. Tento příklad ilustruje deadlock vyvolaný procesy ucházejícími se o týž typ zdroje.

Deadlock může být také vyvolán různými typy zdrojů. Uvažujme např. systém s jednou tiskárnou a jednou magnetopáskovou mechanikou. Nechť proces Pi užívá magnetopáskovou mechaniku a proces Pj tiskárnu. Jestliže Pi požádá o tiskárnu a Pj o páskovou mechaniku nastává deadlock.

8. CHARAKTERISTIKA DEADLOCKU

Je zřejmé, ze deadlock je nežádoucí. V deadlocku procesy nemohou dokončit své spuštění a systémové zdroje jsou obsazené a ani jiné procesy nemohou tyto zdroje alokovat. Než budeme diskutovat různé metody řešení deadlocku, popišme rysy, které ho charakterizují.

8.1 Nutné podmínky

Deadlock může nastat, jestliže jsou v systému současně splněny následující 4 podmínky:

- **Vzájemná jedinečnost**: Minimálně jeden zdroj v systému musí existovat jako nesdílitelný – tj. pouze jeden proces může v danou chvíli užívat tento zdroj. Jestliže se na zdroj tou dobou dotazuje jiný proces, je pozdržen do doby, než se zdroj uvolní.
- **Drží a čeká**: Musí existovat proces, který užívá alespoň jeden zdroj a čeká na přidělení dalšího zdroje, který je užíván jiným procesem.
- **Nepreemptivnost**: Zdroj nesmí byt preemptivně plánován, tzn. zdroj může být uvolněn pouze dobrovolně po ukončení úlohy procesem, který ho na požádání dostal.
- **Kruhové čekání**: Musí existovat množina {P0, P1, ..., Pn-1} čekajících procesů taková, že P0 čeká na zdroj užívaný procesem P1, P1 čeká na zdroj užívaný procesem P2, ..., Pn-1 čeká na zdroj užívaný procesem Pn a Pn čeká na zdroj užívaný procesem P0 (tj. Pi čeká na zdroj užívaný procesem Pi+1 mod n).

Zdůrazněme, že pro vznik deadlocku musí nastat všechny uvedené podmínky současně, přičemž podmínka kruhového čekání implikuje podmínku drž a čekej. Uvedené podmínky tedy nejsou na sobě navzájem nezávislé. Jak však ještě uvidíme, je výhodné definovat podmínky všechny čtyři.

8.2 Graf alokace zdrojů

Deadlock může být precizně popsán pomocí orientovaného grafu zvaného graf alokace zdrojů. Tento graf sestává z množiny vrcholů V a množiny hran H. Množina vrcholů sestává ze dvou podmnožin:

- P = {P1, P2, ..., Pn} – množina všech procesů v systému a
- R = {R1, R2, ..., Rm} – množina všech zdrojů v systému.

Orientovaná hrana od procesu Pi ke zdroji Rj je zaznamenána jako Pi -> Rj a znamená, že proces Pi požaduje instanci zdroje Rj a čeká na její přidělení. Orientovaná hrana od zdroje Rj k procesu Pi je zaznamenána jako Rj -> Pi a znamená, že instance zdroje Rj byla alokována procesu Pi. Orientovaná hrana Pi -> Rj se nazývá *hrana dotazu* a orientovaná hrana Rj -> Pi *hrana přidělení*.

Graficky prezentujme každý proces Pi jako kruh a každý zdroj Rj jako obdélník. Protože každý zdroj Rj může sestávat z více instancí, každou instanci definujme jako tečku uvnitř obdélníku. *Hrana dotazu* tedy dle uvedených předpokladů ukazuje na celý obdélník Rj, zatímco *hrana přidělení* musí ukazovat na konkrétní tečku uvnitř obdélníku.

Pokud se proces Pi dotazuje na instanci zdroje Rj, je do grafu alokace zdrojů přidána hrana dotazu Pi R Rj. Když je dotaz vyplněn, hrana dotazu je *okamžitě* transformována na hranu přidělení Rj R Pi. V okamžiku, kdy proces Pi uvolní zdroj Rj je hrana přidělení smazána.

Na obr. 10 je zobrazena následující situace.

Množiny *P*, *R* a *H*:

```
P = {P1, P2, P3}
R = {R1, R2, R3, R4}
```

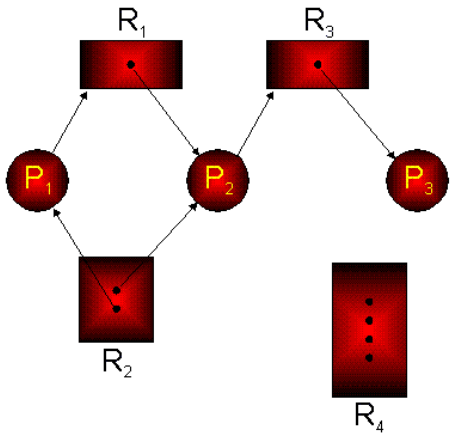
```
H = { P1 -> R1, P2 -> R3, R1 -> P2, R2 -> P2, R2 -> P1, R3 -> P3}
```

Instance zdrojů:

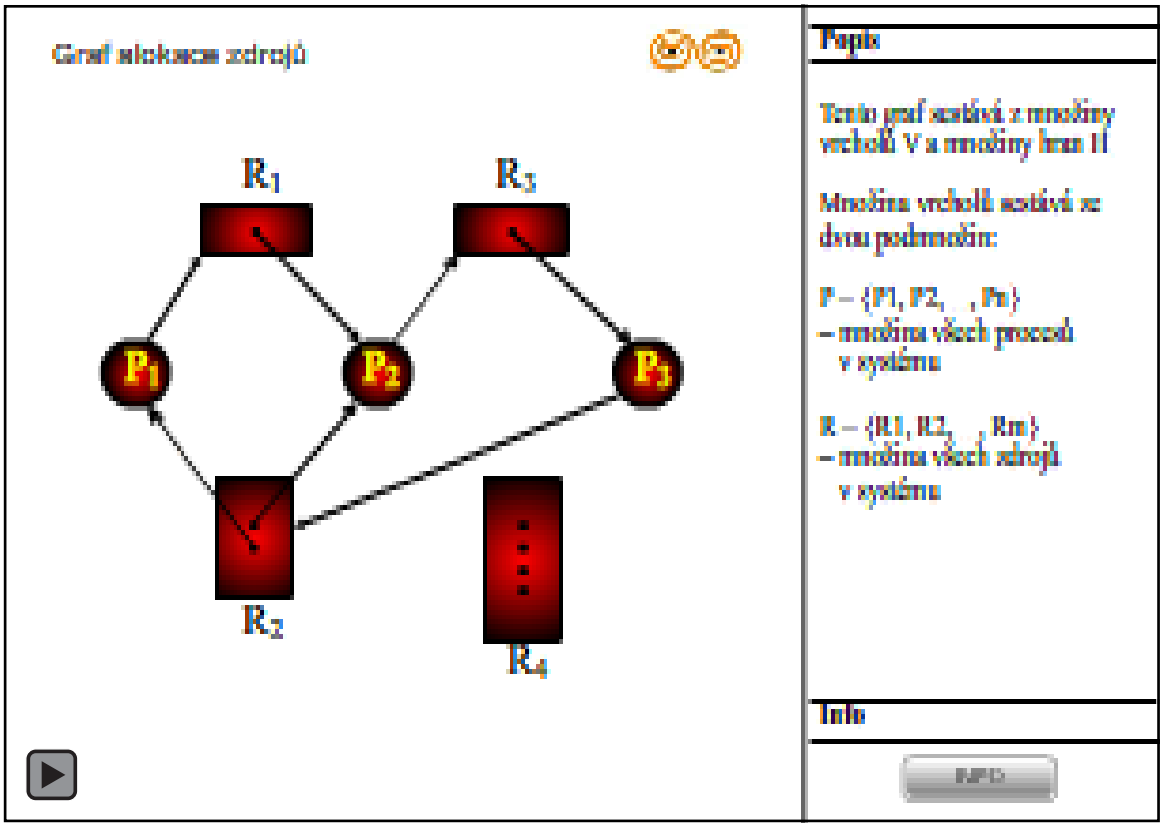
- Jedna instance zdroje R1
- Dvě instance zdroje R2
- Jedna instance zdroje R3
- Tři instance zdroje R4

Stav procesu:

- Proces P1 drží jednu instanci zdroje R2 a čeká na instanci zdroje R1.
- Proces P2 drží jednu instanci zdroje R1 a R2 a čeká na instanci zdroje R3.
- Proces P3 drží jednu instanci zdroje R3.



Obrázek 10 - Graf alokace zdrojů



Animce 7.1 Graf alokace zdrojů

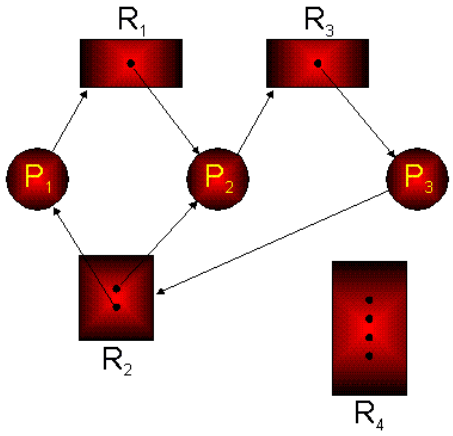
Díky definici grafu alokace zdrojů je velmi snadné ukázat na případný deadlock v systému. Pokud graf neobsahuje žádnou kružnici, není žádný proces zablokován. Na druhé straně, jestliže graf kružnice obsahuje, deadlock může nastat.

Pokud by každý typ zdroje obsahoval právě jednu instanci, kružnice v grafu by automaticky znamenala deadlock. Každý proces v takové kružnici by byl zablokován. V tomto případě je kružnice v grafu nutnou i postačující podmínkou deadlocku.

Jestliže každý zdroj obsahuje několik instancí, potom kružnice v grafu nemusí nutně znamenat deadlock. Zde je kružnice pro vznik deadlocku podmínkou nutnou, ne však postačující.

Pro ilustraci tohoto mechanismu se vraťme k obr. 10. Nechť dále proces P3 požádá o instanci zdroje R2. Protože žádná instance tohoto zdroje není volná, je hrana P3 -> R2 přidána do grafu. V tomto případě existují v grafu dvě kružnice (viz obr. 11)

```
P1 -> R1 -> P2 -> R3 -> P3 -> R2 -> P1
P2 -> R3 -> P3 -> R2 -> P2
```



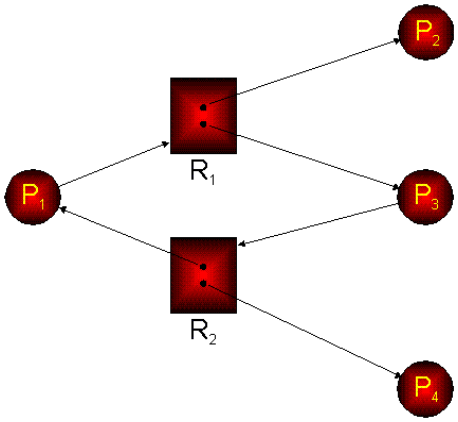
Obrázek 11 - Graf alokace zdrojů s deadlockem

V tomto případě jsou procesy P1, P2 a P3 zablokovány. Proces P2 čeká na zdroj R3, který drží proces P3. Proces P3 čeká na zdroj R2, jehož dvě instance drží procesy P1 a P2 a proces P1 čeká na proces P2, který drží zdroj R1.

Nyní si obraťme pozornost ke grafu alokace zdrojů na obr. 12. V tomto grafu se také vyskytuje kružnice

```
P1 -> R1 -> P3 -> R2 -> P1
```

přesto ovšem k deadlocku nedochází. Proces P4 může totiž uvolnit instanci zdroje R2, ta může být přidělena procesu P3 a kružnice je přerušena.



Obrázek 12 - Graf alokace zdrojů s kružnicí bez deadlocku

Shrňme zjištěné poznatky: Pokud graf alokace zdrojů neobsahuje žádnou kružnici, potom v systému není žádný deadlock. Pokud v grafu kružnice je, v systému deadlock být může i nemusí. Tento závěr bude důležitý v okamžiku, kdy budeme řešit problém deadlocku v systému.

9. METODY OBSLUHY DEADLOCKU

Principiálně existuji 3 metody řešení deadlocku:

- Můžeme v systému užít protokol, který zajistí, že *nikdy* deadlock nenastane.
- Můžeme systému dovolit, aby deadlock nastal a potom ho vyřešit.
- Můžeme celý problém deadlocku ignorovat a předstírat, že k němu nikdy nedojde. Toto řešení zatím užívá většina OSu, včetně Unixu.

Upřesněme krátce každou metodu a v následujících kapitolách ukážeme konkrétní algoritmy.

K zajištění, že deadlock v systému nikdy nenastane, může systém užít buď schéma *deadlock–prevention* nebo schéma *deadlock–avoidance*.

Deadlock–prevention je množina metod, které zajišťují, že minimálně jedna z nutných podmínek (viz kapitola 6.3.1) nenastane. Tato metoda spočívá v omezení možnosti vytváření žádostí o zdroje.

Deadlock–avoidance naopak požaduje, aby OS měl k dispozici další rozšiřující informace o tom, které zdroje bude proces požadovat, a užíval jich za běhu procesu. Díky těmto rozšířeným informacím může systém u každé žádosti o přidělení zdroje rozhodnout, je-li ho možno bezpečně přidělit či nikoli. K bezpečnému vyřešení žádosti o přidělení zdroje potřebuje systém informace o volných instancích zdroje, o instancích alokovaných jiným procesům, o budoucích žádostech o instance tohoto zdroje a o budoucích uvolněních těchto instancí.

Nevyužívá-li systém buď schéma deadlock–prevention nebo schéma deadlock–avoidance obecně může dojít k deadlocku. V takovémto prostředí může systém provádět detekci možného deadlocku nastane-li, pak ho vyřešit.

Jestliže systém nemá žádné zabezpečení proti deadlocku, a tedy neprovádí žádný mechanismus pro detekci a odstranění deadlocku, tehdy může nastat situace, kdy se systém dostane do deadlocku a nemá žádný mechanismus k tomu, aby zjistil, co se stalo. Takovýto nedetekovaný deadlock vede ke snížení výkonu systému, protože zdroje jsou drženy procesy, které nemohou byt ukončeny, a tak se další a další procesy, které nárokují blokované zařízení,

dostávají do deadlocku. Nakonec může systém zcela přestat fungovat a vyžaduje manuální restartování.

Ačkoliv se poslední zmiňovaná metoda nemusí zdát právě optimální pro řešení problému deadlocku, je přece jen užita některými operačními systémy. V mnoha systémech deadlock nastává velmi zřídka (řekneme jednou za rok) a tudíž se nevyplatí drahá režie jeho obsluhy. Deadlock avoidance, deadlock prevention či deadlock detection musí být spuštěny neustále. Navíc existují situace, kdy systém zamrzne, aniž by byl v deadlocku. Představme si například proces spuštěný v reálném čase s maximální prioritou (nebo jakýkoliv proces v systému s nepreemptivním plánováním), který nikdy nevrátí řízení operačnímu systému.

10. DEADLOCK PREVENTION

Jak jsme již uvedli v kapitole 6.3.1, má-li nastat deadlock, musí být splněny současně všechny čtyři tam uvedené podmínky. Zajistíme-li, aby alespoň jedna z nich nenastala, provedeme dostatečnou prevenci deadlocku. Co je proto nutné?

10.1 Vzájemná jedinečnost

Podmínka vzájemné jedinečnosti musí být dodržena u nesdělitelných zdrojů. Např. tiskárnu nemohou najednou využívat dva procesy. Sdílené zdroje na druhé straně podmínku vzájemné jedinečnosti nevyžadují, a proto *nemohou* vyvolat deadlock. Jako ukázku takového zdroje můžeme označit soubor, určený pouze ke čtení. Jestliže z tohoto souboru chce číst současně více procesů, OS jim to může s klidem dovolit. Proces nikdy nemusí čekat na sdílitelný zdroj. V globálu ovšem není možné provádět prevenci deadlocku popřením podmínky vzájemné jedinečnosti, protože některé zdroje prostě sdílitelné nejsou.

10.2 Drží a čeká

K zajištění, aby v systému nikdy nenastala podmínka drží a čeká, musíme garantovat, že kdykoli požádá proces o zdroj, nesmí držet žádné jiné zdroje. Řešením problému drží a čeká, může být protokol, který umožňuje procesu alokovat všechny potřebné zdroje před tím, než bude spuštěn. Implementaci tohoto protokolu lze jednoduše provést tak, že systémovým voláním alokace zdrojů procesu dáme přednost před všemi ostatními.

Jiný protokol umožňuje procesu alokovat zdroje pouze tehdy, nemá-li aktuálně žádné v držení. V takovém případě může zdroje požadovat a užít. Před tím, než může požádat o další zdroje, však musí již alokované zdroje uvolnit.

Pro přiblížení rozdílu mezi těmito dvěma protokoly uvažujme proces, který kopíruje data z pásky na disk, soubor na disku uspořádá a vytiskne výsledek na tiskárně. Při užití prvního protokolu bude mít proces alokovanou tiskárnu po celou dobu jeho spuštění, ačkoli ji evidentně potřebuje až na konci. Druhá metoda umožňuje alokovat procesu pouze disk a pásku. Potom musí obě zařízení uvolnit a požádat o opětovné přiděleni disku spolu s tiskárnou. Po zkopírování souboru do tiskárny proces uvolní obě zařízení a ukončí se.

Tyto protokoly mají dvě hlavní nevýhody:

- Může dojít k *nízkému využití zdroje*, protože mnoho zdrojů může být alokováno a nevyužito po dlouhou dobu. V uvedeném příkladu při druhém protokolu můžeme uvolnit pásku a diskový soubor a potom opětovně požádat o soubor a tiskárnu pouze v případě, když máme jistotu, že naše data zůstala na disku v tom stavu, v jakém jsme je zanechali. Jestliže si jisti být nemůžeme, musíme na počátku požádat o všechny zdroje v obou případech.

- Může dojít k *umoření*. Proces, který požaduje nějaký populární zdroj, může čekat neomezeně dlouho, když nejméně jeden ze zdrojů, které požaduje, je neustále alokován jiným procesem.

10.3 Nepreemptivnost

Třetí nutná podmínka deadlocku je nepreemptivní plánování alokovaných zdrojů. K zajištění toho, že tato podmínka nikdy nenastane, můžeme užít následující protokol: Pokud proces, který má alokovaný nějaký zdroj, požaduje ještě další zdroj, nemůže zdroj alokovat ihned (tj. musí čekat), než budou všechny zdroje, které má přidělené preemptivně uvolněny. Tzn. všechny před tím alokované zdroje jsou implicitně uvolněné a přidané do seznamu zdrojů, na které proces čeká. Proces může být restartován pouze v případě, že může znovu nabýt svých starých zdrojů, stejně jako zdroje nového, o který původně žádal.

Jinak, jestliže proces požaduje zdroje, nejdříve zkontrolujeme, jestli jsou volné. Pokud ano, přidělíme mu je. Jestliže volné nejsou, zjistíme, nejsou-li alokovány procesům čekajícím na jiné zdroje. Pokud ano, požadované zdroje preemptivně uvolníme a přidělíme žádajícímu procesu. Jestliže zdroje nejsou volné, ani nejsou v držení čekajícího procesu, žádající proces musí čekat. Zatímco tento proces čeká, může být jím požadované zařízení preemptivně uvolněno, ovšem pouze v případě, že o to požádá jiný proces. Proces může být obnoven pouze v případě, že alokuje nové zdroje, o které žádal a případně znovu nabyde zdrojů, o které přišel, zatímco čekal.

Tento protokol je často používán u zdrojů, jejichž stav může být snadno uložen a potom opětovně obnoven (např. registry CPU nebo paměťový prostor). Nemůže být ovšem aplikován globálně na všechny zdroje (např. stěží na tiskárnu).

10.4 Cyklické čekání

Jedna z cest, jak zajistit, že nikdy nedojde k cyklickému čekání je definovat absolutní uspořádání všech typů zdrojů a vyžadovat, aby se procesy dotazovaly na zdroje ve vzestupné posloupnosti.

Nechť R = {R1, R2, …, Rm} je množina typů zdrojů. Přiraďme každému zdroji jedinečné celé číslo. Tato čísla nechť definují uspořádání na množině R. Definujeme tedy vzájemně jednoznačnou funkci F: R -> N, kde N je podmnožina množiny přirozených čísel. Obsahuje-li například množina typů zdrojů magnetopáskové mechaniky, pevné disky a tiskárny, funkce F může být definována následovně:

```
F(magnetopaskova mechanika) = 1,
F(pevny disk) = 5,
F(tiskarna) = 12.
```

Nyní můžeme definovat následující protokol prevence deadlocku:

Každý proces může požadovat zdroje systému pouze v uspořádané posloupnosti definované enumerace. Tzn. proces může nejdříve požádat o zdroj Ri. O další zdroj Rj může pak požádat pouze v případě, že F(Ri) < F(Rj). Jestliže požaduje více instancí téže třídy zdroje, musí tak učinit *jediným* dotazem. V souladu s uvedeným očíslováním zdrojů, požaduje-li proces současně páskovou mechaniku a tiskárnu, musí nejprve požádat o pásku a potom o tiskárnu.

Stejně tak můžeme jednoduše požadovat, aby kdykoliv se proces dotazuje na instanci zdroje Rj musel před tím uvolnit všechny zdroje Ri, pro které platí F(Ri) >= F(Rj).

Jestliže jsou užity oba dva předešlé protokoly, potom cyklické čekání nemůže nastat. Toto tvrzení lze dokázat sporem. Nechť {P0, P1, ..., Pn-1} je množina cyklicky čekajících procesů, kde Pi čeká na zdroj Ri, který je držen procesem Pi+1 mod n (je užito modulo indexování, takže proces Pn čeká na zdroj Rn, který drží proces P0). Potom, jestliže Pi+1 drží zdroj Ri a

zároveň se dožaduje zdroje Ri+1, musí pro všechna i platit, že F(Ri) < F(Ri+1) => F(R0) < F(R1) < ... < F(Rn) < F(R0). Z tranzitivity potom plyne, že F(R0) < F(R0), což je SPOR.

Poznamenejme, že funkce F by měla být definována podle pořadí důležitosti zdrojů v systému. Např. magnetická páska bývá většinou užita před užitím tiskárny, takže by mělo platit F(magnetopásková mechanika) < F(tiskárna).

11. DEADLOCK AVOIDANCE

Algoritmus deadlock–prevention, diskutovaný v předcházející kapitole, předchází deadlocku omezením možnosti toho, jak mohou procesy žádat o zdroje. Tato omezení zajišťují, že nenastane minimálně jedna z nutných podmínek deadlocku. Možný negativní dopad těchto mechanismů na OS je ve snížení jeho výkonu a propustnosti.

Jiná metoda prevence deadlocku je požadovat více informací o tom, jak budou procesy vyžadovat zdroje. Např. v systému s jednou páskovou mechanikou a jednou tiskárnou můžeme říci, že proces P požádá nejprve o pásku a potom o tiskárnu nežli oba zdroje uvolní. Proces Q nechť naopak nejprve požaduje tiskárnu a potom pásku. S těmito znalostmi úplného pořadí žádostí a uvolněni zdrojů každým procesem můžeme pro každý dotaz rozhodnout, má-li proces čekat či nikoli.

Pro rozhodnutí, zda aktuální dotaz má být vyplněn nebo čekat bez nebezpečí následného deadlocku, jsou třeba informace o volných zdrojích systému, i o zdrojích aktuálně alokovaných různým procesům, které tyto procesy posléze uvolní.

Různé algoritmy se liší v množství a typech požadovaných informací. Nejjednodušší a velmi užitečný model požaduje, aby každý proces deklaroval *maximální počet* zdrojů každé třídy, které může potřebovat. Jsou-li tyto informace dostupné pro každý proces, lze sestavit algoritmus, který zajistí, že se systém nikdy neocitne v deadlocku. To je cesta prevence deadlocku metodou deadlock–avoidance.

Algoritmus deadlock–avoidance dynamicky ohodnocuje stav alokace zdrojů, aby se ujistil, že nemůže nastat podmínka cyklického čekání. *Stav* alokace zdrojů je definován jako počet volných a alokovaných zdrojů a maximální počet požadavků procesu.

11.1 Bezpečný stav

Stav je bezpečný, jestliže systém může alokovat zdroje každému procesu (v mezích maximálního počtu) v nějakém pořadí a zároveň nenastane deadlock. Formálně, systém je v bezpečném stavu, jestliže existuje *bezpečná sekvence*. Sekvence procesu <P1, P2, ..., Pn> je bezpečná pro aktuální stav alokace, jestliže každému procesu Pi mohou být přiděleny požadované zdroje ze zdrojů volných + zdrojů držených procesy Pj, kde j < i. Za této situace, pokud zdroje požadované procesem Pi nejsou momentálně volné, potom Pi čeká, dokud nejsou ukončeny všechny procesy Pj. V momentě, kdy dokončeny jsou, Pi může dostat všechny požadované zdroje, splnit svou úlohu, zdroje uvolnit a skončit. Je-li proces Pi ukončen, může dostat všechny požadované zdroje proces Pi+1 atd. Pokud v systému neexistuje bezpečná sekvence, říkáme o něm, že není bezpečný.

Deadlock není bezpečný stav, nicméně ne všechny stavy, které nejsou bezpečné, musí být deadlockem. Stav, který není bezpečný, může však k deadlocku vést. Tak dlouho jak je systém v bezpečném stavu, může se vyvarovat nebezpečí včetně deadlocku. V nebezpečném stavu nemůže systém zabránit procesům, aby nevyžadovaly zdroje tak, aby k deadlocku došlo. Nebezpečný stav může vyvolat chování procesu.

Pro ilustraci si představme systém s 12 magnetickými páskami a třemi procesy P0, P1 a P2. Nechť proces P0 požaduje 10 pásek, P1 nejvýše 4 a proces P2 9 pásek. Dále nechť v čase t0

má proces P0 alokováno 5 pásek a procesy P1 a P2 po dvou páskách (tzn. v systému jsou ještě 3 volné pásky).

	Maximum potřebných pásek	Počet přidělených pásek
P0	10	5
P1	4	2
P2	9	2

V čase t0 je systém v bezpečném stavu. Sekvence procesů <P1, P0, P2> dodrží všechny podmínky bezpečnosti. Proces P1 alokuje pro sebe všechny potřebné pásky, vykoná svou úlohu, bude ukončen a vrátí pásky systému. V ten moment je v systému 5 volných pásek. Všechny je alokuje proces P0 a může dokončit svou úlohu. Když vrátí pásky systému, může být dokončen i proces P2.

I tento systém však může jednoduše přejít do nebezpečného stavu. Nechť v čase t1 proces P2 alokuje další pásku. V ten moment se systém dostává do nebezpečného stavu. Za této situace může získat všechny potřebné pásky pouze proces P1. Když P1 skončí a vrátí všechny pásky, jsou v systému dostupné pouze 4. Neboť P0 má alokovaných 5 pásek a potřebuje 10, snaží se získat dalších 5. Ty ale nejsou v systému dostupné, takže P0 musí čekat. Stejně tak proces P2 potřebuje ke svému ukončení dokonce dalších 6 pásek a nastává deadlock.

Chybou bylo povolení alokace jedné pásky procesem P2. Pokud bychom proces P2 zadrželi ve stavu *čekající*, dokud nebude některý z procesů P0 nebo P1 ukončen, k deadlocku by nedošlo.

Za přispění mechanismu bezpečného stavu můžeme definovat algoritmus prevence deadlocku. Základní myšlenka tohoto algoritmu je jednoduchá: zajistit, aby systém byl neustále v bezpečném stavu.

Na počátku systém v bezpečném stavu pochopitelně je. V okamžiku, kdy proces žádá zdroj, který je aktuálně volný, musí systém rozhodnout, zda procesu zdroj přidělí, nebo ho nechá čekat. Zdroj může být procesu přidělen pouze v případě, že systém i po přidělení zůstane v bezpečném stavu.

Je zřejmé, že tento mechanismus, kdy proces mnohdy musí čekat na zdroj, který je aktuálně volný může vést ke snížení výkonu systému.

12. DETEKCE DEADLOCKU

Jestliže systém nevyužívá žádného mechanismu pro prevenci deadlocku, může deadlock v systému nastat. V takovém prostředí musí systém provádět:

- algoritmus ohodnocující stav systému, který je schopen odhalit nastalý deadlock a
- algoritmus, který nastalý deadlock vyřeší.

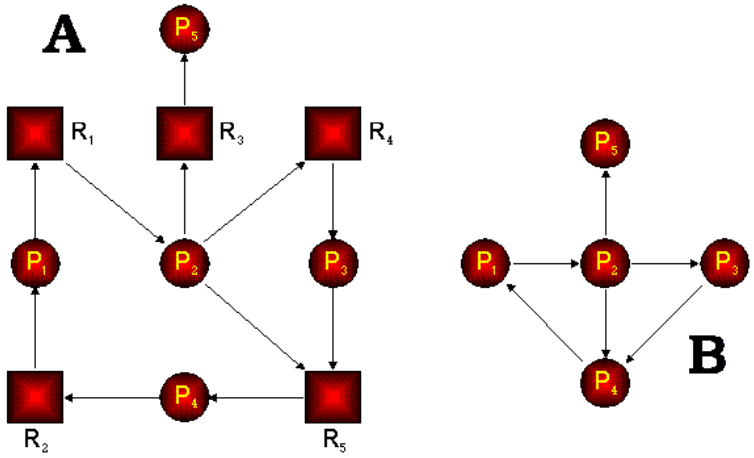
V následující diskusi rozebereme tyto dva požadavky v systému s jednou instancí každého zdroje i v systému s více instancemi. Podotkněme, že odhalení a vyřešení deadlocku vyžaduje režii, která neobsahuje jen čas systému nutný k obsluze datových struktur a spuštění algoritmu detekce, ale také potenciální ztráty vzniklé při nápravě deadlocku.

12.1 Jedna instance v každé třídě zdroje

Jestliže všechny třídy zdrojů mají pouze jednu instanci, je možno k detekci deadlocku užít speciální tvar grafu alokace zdrojů, zvaný *graf čekání*. Tento graf můžeme získat z grafu alokace zdrojů odebráním všech uzlů zdrojů a kolabováním příslušných hran.

Precizněji, hrana *Pi* -> *Pj* v grafu čekání znamená, že proces *Pi* čeká na to, až proces *Pj* uvolní zdroje, které *Pi* potřebuje. Hrana *Pi* -> *Pj* v grafu čekání existuje pouze v případě,

jestliže v analogickém grafu alokace zdrojů existuji hrany *Pi -> Rq* a *Rq -> Pj* pro zdroj *Rq*. Na obr. 13 jsou vidět jak graf alokace zdrojů i graf čekání popisující stejnou situaci.



Obrázek 13 - Stav systému: (A) Graf alokace zdrojů (B) Graf čekání

Stejně jako u grafu alokace zdrojů, deadlock je v systému tehdy a jen tehdy, obsahuje-li adekvátní graf čekání kružnici. Pro detekci deadlocku musí systém *udržovat* graf čekání a periodicky *spouštět algoritmus*, který hledá v grafu kružnici.

Algoritmus detekující kružnici v takovémto grafu vyžaduje sekvenci n^2 operací, kde *n* je počet vrcholů grafu.

12.2 Více instancí v každé třídě

Schéma užívající graf čekání není aplikovatelné pro odhalení deadlocku v systému s více instancemi v jednotlivých třídách. Popíšeme si algoritmus použitelný za této situace. Algoritmus užívá analogických datových struktury jako Bankéřův algoritmus:

- *Volny*: vektor délky *m* indikující počet volných instancí každého typu zdroje
- *Alokace*: matice typu [*n,m*] definující počet zdrojů každé třídy aktuálně alokovaných každému procesu.
- *Zadost*: matice typu [*n,m*] definující aktuální žádost každého procesu. Jestliže *Zadost[i,j] = k*, potom proces *Pi* požaduje dalších *k* instancí zdroje třídy *Rj*.

Relace menší než (<) nechť je definována stejně jako v kapitole 6.5.3. Pro zjednodušení zápisu opět definujme i-ty řádek matic *Alokace* a *Zadost* jako *Alokacei* a *Zadosti*. Algoritmus prostě prohledává všechny možné sekvence alokace zdrojů procesy, které mají být dokončeny.

1. Nechť *Prace* je vektor délky *m* a *Konec* je vektor délky *n*. Inicializujme *Prace = Volny* a pro *i* = 1, 2, …, *n* přiraď *Konec[i] := false* jestliže *Alokacei* <> 0, jinak *Konec[i] := true*.
2. Vyhledej *i* pro které platí obě nesledující podmínky:

   Konec[i] = false

   Zadosti L Prace

   Jestliže takové *i* neexistuje, přejdi na bod 4.
3. Prace := Prace + Alokacei

   Koneci := true

Přejdi na bod 2.

4. Jestliže *Konec[i] = false* pro nějaké *i*, 1 <= *i* <=; *n*, potom systém je v deadlocku. Navíc, jestliže *Konec[i] = false*, potom je proces *Pi* zablokován.

Tento algoritmus požaduje sekvenci $m * n^2$ operací pro detekci deadlocku v systému.

Může se zdát podivné, že přidělíme požadované zdroje procesu *Pi* (v kroku 3) okamžitě poté, co zjistíme, že *Zadosti* <= *Prace* (v kroku 2b). Víme, že *Pi* není aktuálně v deadlocku (protože *Zadosti* <= *Prace*). Takže postavíme optimistický předpoklad, že *Pi* nebude vyžadovat další zdroje k dokončení svého úkolu, a že tedy brzy vrátí všechny přidělené zdroje zpět systému. Jestliže je tento náš předpoklad chybný, může později nastat deadlock. Ten bude následně detekován algoritmem detekce deadlocku.

Pro ilustraci tohoto algoritmu uvažujme systém s pěti procesy *P0* až *P4* a třemi zdroji typu *A, B, C*. Zdroj typu *A* má 7 instancí, zdroj typu *B* 2 a zdroj typu *C* 6 instancí. Nechť v čase *t0* je systém v následujícím stavu:

	Alokace			Max			Volny		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

V této situaci není systém v deadlocku. Skutečně, pokud spustíme výše uvedený algoritmus, potom sekvence < *P0, P2, P3, P1, P4*> vrací výsledek *Konec[i] = true* pro všechna *i*.

Uvažujme, že proces *P2* požádá o jednu další instanci zdroje třídy *C*. Matice *Zadost* potom vypadá následovně:

	Zadost		
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

V takové situaci je systém v deadlocku. Můžeme uvolnit zdroje držené procesem *P0*. Počet volných zdrojů však i přesto nedostačuje k dokončení žádného jiného procesu. Deadlock nastal a obsahuje procesy *P1, P2, P3* a *P4*.

12.3 Užití algoritmu detekce deadlocku

Kdy má být spuštěn algoritmus detekce deadlocku? Odpověď na tuto otázku je podmíněna dvěma faktory:

- Jak *často* k deadlocku pravděpodobně dochází?
- *Kolik* procesů bude deadlockem postiženo v případě, že nastane?

Pokud k deadlocku dochází často, potom je třeba algoritmus detekce spouštět často. Zdroje alokované procesům v deadlocku budou zablokované, dokud nebude deadlock odstraněn. Počet procesů v deadlocku se může časem zvětšovat.

Deadlock může nastat pouze v případě, že některý proces vytvoří žádost, která nemůže být okamžitě vyřízena. Je možné, že tato žádost vyvolá řetězec čekajících procesů. Jedna extrémní možnost je spouštět algoritmus detekce vždy, když nemůže být žádost libovolného procesu okamžitě vyřízena. V takovém případě můžeme odhalit nejen celou množinu procesů v deadlocku, ale i jeho původce. (Ve skutečnosti každý deadlockovaný proces vyvolá jedno spojení v kružnici grafu alokace zdrojů, takže procesy vyvolají deadlock společně.)

Je-li v systému mnoho různých tříd zdrojů, může jedna žádost vyvolat více kružnic v grafu alokace zdrojů. Případná kružnice v grafu je uzavřena poslední žádostí a tedy „vyvolána“ jedním identifikovatelným procesem.

Samozřejmě, spouštěni algoritmu detekce deadlocku po každé žádosti může vyvolat značně nároky na výpočetní čas. Méně náročná alternativa je vyvolávat algoritmus v malých pravidelných intervalech (např. jednou za hodinu) nebo když využití CPU klesne např. pod 40 %. (Deadlock může ochromit systém do té míry, že výrazně poklesne využití CPU.)

Je-li algoritmus detekce spouštěn v libovolném čase, může v grafu alokace zdrojů existovat velké množství kružnic. Obecně pak nejsme schopni určit, který z množství zablokovaných procesů deadlock vyvolal.

13. NÁPRAVA DEADLOCKU

V okamžiku, kdy algoritmus detekce zjistí přítomnost deadlocku v systému, jsou různé alternativy další cennosti systému. Jedna možnost je informovat správce systému, že nastal deadlock a nechat ho odstranit deadlock manuálně. Druhou možností je ponechat automaticky nápravu na systému. Tato automatická náprava může být provedena dvojím způsobem. Nejjednodušší možností je ukončit jeden nebo více cyklicky čekajících procesů. Druhá možnost je preemptivně ukončit alokaci nějakého zdroje deadlockovaným procesem.

13.1 Ukončení procesu

K eliminaci deadlocku ukončením některého zablokovaného procesu je možno užít jednu ze dvou metod. V obou případech systém uvolní a získá všechny zdroje alokované ukončovaným procesům.

- **Ukončení všech deadlockovaných procesů**: Ukončit všechny procesy v kruhu cyklického čekání je sice jednoduché, ale mnohdy drahé. Vypočet nejednoho procesu mohl trvat i značně dlouho než se proces dostal do deadlocku, a celá tato výpočetní doba je najednou ztracena a výpočet procesu musí začít znovu.
- **Ukončení jednoho nebo více procesů, dokud není deadlock eliminován**: Tato metoda vyžaduje značnou režii, neboť po každém ukončeni každého procesu musí být spuštěn algoritmus detekce deadlocku ke zjištění, nachází-li se systém stále v deadlocku.

Poznamenejme, že ukončení procesu nemusí být vždy jednoduchá věc. Jestliže proces pravé provádí změnu nějakého souboru, jeho ukončení by mohlo zanechat soubor v nekonzistentním stavu. Podobná situace nastává, jestliže proces právě provádí tisk. V takovém případě musí systém provést reset tiskárny a nastavit ji do bezpečného stavu před tím, než povolí tisk další.

Jestliže je užita metoda ukončení pouze některých procesů, potom je třeba z množiny cyklicky zablokovaných procesů vybrat ten (nebo ty), které mají být ukončeny, aby se deadlock ze systému odstranil. To je politické rozhodnutí podobného charakteru jako plánovaní CPU. Celý problém se točí kolem toho, jak vybrat proces(y), jejichž ukončení bude pro systém nejméně náročné. Na konkretizaci této obecné formulace má vliv mnoho faktorů, např.:

- Jakou má proces prioritu
- Jak dlouho je již proces zpracováván systémem a jaká doba je ještě třeba k jeho dokončení
- Kolik zdrojů a jakého typu má proces alokováno (nedají-li se tyto zdroje preemptivně uvolnit)
- Kolik dalších zdrojů bude proces ještě potřebovat ke svému dokončení

- Kolik procesů je třeba ukončit
- Je proces interaktivní nebo dávkový

13.2 Preemptivní uvolnění zdroje

Při eliminaci deadlocku preemptivním uvolněním zdroje postupně uvolňujeme zdroje alokované zablokovaným procesům, přidělujeme je jiným, dokud není deadlock odstraněn.

Jestliže chceme tuto metodu užít, je třeba vyřešit tři problémy:

- **Vybrat oběť**: Který zdroj a který proces mají být preemptivně rozděleni? Stejně jako u ukončení procesu musíme určit nejméně náročné pořadí preemptivních uvolňování. Faktory náročnosti zde mohou být počet zdrojů, které drží zablokovaný proces a množství systémového času, které již spuštěný proces spotřeboval.
- **Zabezpečení**: Jestliže uvolníme preemptivně zdroj, musíme proces, který o něj přišel, uvést do bezpečného stavu. Tento proces přišel o zdroj nutný k jeho výpočtu. Tento nedostatek je třeba ošetřit a umožnit procesu další bezpečný výpočet. Protože obecně je velmi těžké stanovit, co to je bezpečný stav konkrétního procesu, je nejjednodušší provést absolutní návrat procesu, tj. ukončit proces a restartovat ho.
- **Umořeni**: Jak zajistit, aby nedošlo k umoření? To jest, jak zaručit, aby zdroje nebyly stále preemptivně uvolňovány a opětovně přidělovány témuž procesu? V systému, kde je výběr oběti veden pouze požadavky maximální efektivity se může stát, že bude obětován stále jeden a tentýž proces. Tento proces ovšem nikdy nedokončí svou úlohu a nastává umoření procesu. Je tedy třeba zajistit, aby byl proces obětován pouze konečně krát. Výhodnou cestou je zařadit počet navrácení procesu do faktorů, podle kterých je vybírán obětovaný proces.

14. KOMBINOVANÝ PŘÍSTUP K ŘEŠENÍ DEADLOCKU

Systémoví programátoři namítají, že žádná ze základních metod řešeni deadlocku (prevention, avoidance, detection) není sama vhodná pro řešení všech problémů alokace zdrojů v operačním systému. Jedna z možností je kombinovat tyto tři základní metody a pro každou třídu zdrojů užít metodu optimální.

Tento přístup je založen na myšlence, že zdroje mohou být rozděleny do hierarchicky uspořádaných tříd. Technika uspořádání zdrojů je užita na třídy. Potom může být pro každou třídu zdrojů užita nejpřijatelnější technika řešení deadlocku.

Je jednoduché ukázat, že pokud je v systému aplikována tato strategie, není vystaven problému deadlocku. Deadlock nemůže postihnout více než jednu třídu zdrojů, protože je užita technika uspořádání zdrojů a každá třida má asociován algoritmus obsluhy deadlocku.

K ilustraci popisovaného přístupu užijeme systém obsahující nesledující třídy zdrojů:

- **Interní zdroje**: zdroje užívané systémem (např. process control block)
- **Centrální paměť**: paměť vyhrazená pro uživatelské úlohy
- **Zdroje procesu**: aplikovatelné zdroje (např. pásky, tiskárny apod.) a soubory
- **Swapovací prostor**: úložný prostor pro uživatelské procesy

Kombinované řešeni deadlocku pro výše uvedené pořadí tříd zdrojů užívá následujících postupů pro jednotlivé třídy:

- **Interní zdroje**: Lze užít *deadlock prevention* popření podmínky cyklického čekání užitím uspořádání tříd zdrojů, nedochází k rozhodování mezi více žádostmi v jednom okamžiku

- **Centrální paměť**: Lze užít *deadlock prevention* užitím preemptivní prevence, protože každá úloha může být vyswapována z paměti a tato paměť může být preemptivně přidělena jiné úloze.
- **Zdroje procesu**: Lze užít *deadlock avoidance*, neboť potřebné rozšiřující informace je možno získat.
- **Swapovací prostor**: Lze užít *předběžné přidělení (preallocation)*, protože maximální požadavky jsou předem známy.

Tento příklad ilustruje, jak mohou být základní metody obsluhy deadlocku kombinovány v uspořádané struktuře zdrojů, aby bylo dosaženo efektivního řešení deadlocku.