1 HISTORICKÝ PREHĽAD

Fascinujúci svet počítačov a ľudí okolo nich je svetom, kde vládne mnoho obchodných, strategických a čisto osobných záujmov. Je to svet, ktorý je do značnej miery závislý na stupni ľudského poznania a z neho vyplývajúcej úrovne vedy, techniky a najrôznejších technológií. Avšak väčšina udalostí zásadného významu je v tomto svete spojená s myšlienkami, ktoré sa zrodili v hlave jedného človeka a vzápätí zmenili spôsob myslenia a práce menších a väčších kolektívov, až sa postupne presadili a zásadne ovplyvnili chod udalostí vo svete počítačov.

V ďalšom texte uvádzame stručný prehľad vývoja operačných systémov.

1.1 Prvé systémy

Prvé počítačové systémy boli veľké stroje, ktoré sa ovládali z konzoly. Programátor (a operátor v jednej osobe) najskôr napísal program, potom ho ručne zaviedol z papierovej diernej pásky alebo diernych štítkov pomocou zariadenia na čítanie týchto médií do pamäte, nastavil štartovaciu adresu a spustil program. Programátor mohol sledovať vykonanie programu a v prípade chyby ho mohol zastaviť a preskúmať obsah pamäťových buniek a registrov a ladiť program priamo z konzoly. Výstup sa buď vytlačil na tlačiarni, alebo sa zapísal do diernych štítkov alebo pásky pre neskoršie vytlačenie.

Časom boli vyvinuté ďalšie hardvérové zariadenia a komponenty. Začala sa bežne používať magnetická páska. Boli vyvinuté asemblery, zavádzacie a spojovacie programy. Vytvárali sa knižnice najčastejšie používaných funkcií.

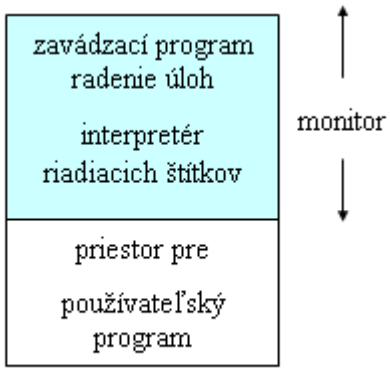
Programy, ktoré vykonávali vstupno/výstupné (V/V) operácie boli veľmi dôležité. Každé nové hardvérové zariadenie malo svoje vlastné charakteristiky a vyžadovalo osobitnú pozornosť pri programovaní. Pre každé zariadenie sa navrhovala špeciálna rutina - **ovládač**, ktorý vedel konkrétne detaily o zariadení ako sú registre, bufre, riadiace bity atď., ako aj spôsob obsluhy zariadenia. Programovanie V/V operácií sa uľahčilo, pretože programátor mohol použiť ovládač zariadenia z knižnice.

Neskôr boli vyvinuté vyššie programovacie jazyky ako sú FORTRAN, COBOL, ALGOL, ktoré značne zjednodušovali programovanie, aj keď manipulácia pre preloženie, spojenie a spustenie programu napísaného v niektorom z týchto jazykov, bola veľmi zložitá a zdĺhavá.

1.2 Dávkové systémy

Manipulácia, spojená so spustením každej úlohy zaberala veľa času. Tu treba pripomenúť, že strojový čas prvých počítačových systémov bol extrémne drahý. Z toho faktu vyplynula snaha pre jeho efektívne využitie. Programátori už nepracovali pri počítači, ale odovzdávali svoje úlohy na magnetických páskach a profesionálni operátori sa starali o spustenie jednotlivých úloh. Ďalšie zefektívnenie využitia strojového času sa zaznamenalo, keď sa úlohy začali zhromažďovať do *dávok*. Dávka pozostávala z niekoľkých úloh, ktoré mali rovnaké požiadavky na prekladač, napr. niekoľko fortranovských programov. Takto sa zmenšil čas pre vykonanie programov, lebo nebolo treba pre každú úlohu znova zavádzať prekladač.

Neskôr sa práca pre spustenie programov zefektívnila použitím ***rezidentného monitora,*** t.j. programu, ktorý bol trvalé uložený v pamäti. Monitor mal na starosti sledovanie priebehu vykonania programu a keď program skončil (normálne alebo s chybou), odovzdal riadenie ďalšiemu programu. **Rezidentný monitor bol vlastne prvý základný operačný systém.**

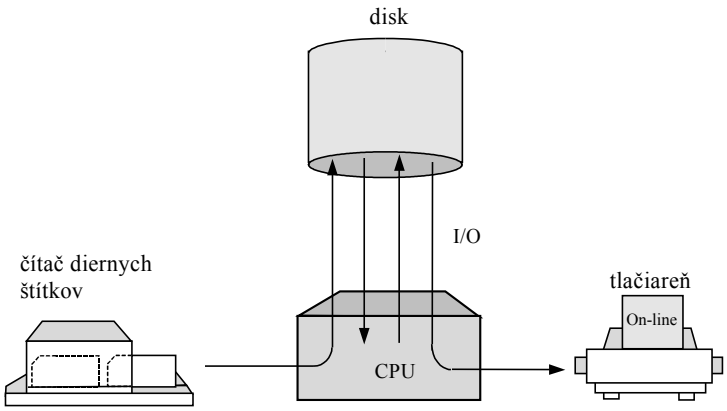


Obr. 1.1 Rozloženie rezidentného monitora v pamäti

Programátor odovzdával pre vykonanie svoj program, dáta, ktoré tento program potreboval a niekoľko riadiacich štítkov, ktoré obsahovali informácie o požiadavkách programu. Riadiace štítky informovali monitor o tom, aký prekladač má spustiť, kde program začína, kde končí a tiež požiadavky pre operátora, napr. ktorú magnetickú pásku má zaviesť atď. Rozloženie rezidentného monitora v pamäti je ukázané na Obr.1.1. Monitor dávkového systému automaticky radil programy pre vykonanie podľa pokynov na riadiacich štítkoch. Najskôr zaviedol program označený na štítku do pamäte a spustil ho. Po ukončení programu prebral znova riadenie a prečítal riadiaci štítok ďalšieho programu. Nedostatkom takéhoto spracovania je nemožnosť interakcie medzi programátorom a programom, počas jeho behu. Program bol pripravený a odovzdaný na spracovanie, ktoré mohlo trvať niekoľko hodín alebo niekoľko dní.

Ďalší vývoj priniesol zrýchlenie spracovania úloh v počiatočnej a záverečnej fáze. Miesto čítačov diernych štítkov, vstup úloh sa vykonával z magnetických pások. Programy a dáta sa načítali z diernych štítkov a nahrali sa na magnetickú pásku v režime *off-line* (t.j. nie v spojení s počítačom). Potom sa páska presunula k počítaču, kde sa nahrané úlohy jedna po druhej spracovali. Výstupy sa tiež nahrávali na magnetickú pásku a potom sa off-line tlačili.

Systém **spooling** (Simultaneous Peripheral Operation On Line) sa začal používať v počítačoch, ktoré mali systém s diskovými perifémymi pamäťami. Princíp spooling-u je znázornený na (Obr. 1.2.).



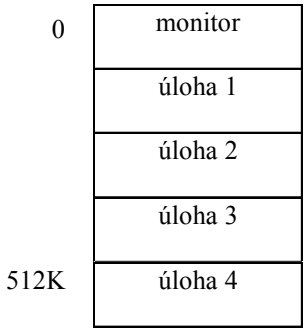
Obr. 1.2 Spooling

Disky značne urýchlili off-line operácie. Problém pri práci s páskami spočíval v tom, že páska je sekvenčné zariadenie a nie je možné, aby sa z jedného jej konca načítaval vstup z diernych štítkov a na druhom konci zároveň zaznamenávali výstupy. Tento problém vyriešil disk, zariadenie s priamym prístupom. S použitím diskov sa dierne štítky začali načítavať z čítača štítkov rovno na disk. Obdobným spôsobom sa spracovali aj výstupy. Miesto na tlačiareň, výstupy sa zaznamenávali na disk a neskôr sa vytlačili. Táto forma spracovania sa nazýva *spooling*. Podstata spooling-u je v tom, že disk plní úlohu veľmi veľkého bufra, do ktorého sa vopred načítajú vstupy úloh a do ktorého sa zapisujú výstupy, ktoré sa vytlačia vtedy, keď výstupné zariadenie bude voľné. Pri spooling-u sa práca procesora prelína s prácou perifémych zariadení a tým sa zvyšuje jeho výkon.

1.3 Multiprogramové dávkové systémy

Využitie spooling-u umožňuje mať niekoľko úloh, pripravených na spustenie. Táto skutočnosť dáva možnosť operačnému systému vybrať si úlohu, ktorú spustí ako ďalšiu tak, aby sa zvýšilo využitie procesora. Tento výber nebol možný v prípade využitia čítačiek diernych štítkov a magnetických pások. Keď sa ako vstupné zariadenie využíva disk a na ňom je niekoľko pripravených úloh, operačný systém môže úlohy plánovať.

Najdôležitejším aspektom plánovania úloh je ***multiprogramovanie***. Základná idea multiprogramovania spočíva v tom, že operačný systém udržiava v pamäti niekoľko úloh naraz (Obr. 1.3).



Obr. 1.3 Pamäť multiprogramového systému

Obyčajne počet úloh v pamäti je omnoho menší ako je počet úloh, ktoré čakajú na spracovanie. Operačný systém vyberie jednu úlohu a spustí ju. Ak táto úloha počas svojho vykonania musí čakať na nejakú udalosť, napr. namontovanie pásky, skončenie V/V operácie alebo inú udalosť, operačný systém prepne na ďalšiu z úloh v pamäti a vykonáva ju. Ak táto úloha tiež musí čakať, prepne sa na ďalšiu atď. Tento spôsob práce zaisťuje, že procesor nebude zaháľať kým sú úlohy, ktoré čakajú na spracovanie.

Multiprogramovanie je prvý prípad, kedy sa operačný systém musí rozhodovať za používateľa. Operačné systémy, ktoré využívajú multiprogramovanie sú zložitejšie. Všetky úlohy, ktoré vstupujú do systému sa ukladajú na disk, kde čakajú na zavedenie do pamäte. Ak je pripravených niekoľko úloh a v pamäti nie je dostatočne veľký priestor pre všetky úlohy, operačný systém musí rozhodnúť, v akom poradí sa budú zavádzať do pamäte. Toto rozhodovanie sa nazýva plánovanie úloh. Keď operačný systém vyberie úlohu na vykonanie, zavedie ju do pamäte. Keďže v pamäti je niekoľko úloh naraz, je potrebná správa pamäte. Ďalej je potrebné vybrať spomedzi pripravených úloh v pamäti tú, ktorej sa pridelí čas procesora. Toto rozhodovanie sa nazýva plánovanie času procesora. Vo všetkých fázach spracovania úloh bežiacich súčasne, je potrebné obmedziť nežiadúce interakcie medzi nimi. Všetky tieto aspekty operačného systému sú vysvetlené v ďalšom texte.

Multiprogramové dávkové systémy využívajú lepšie systémové prostriedky (procesor, pamäť, periférne zariadenia), ale nedávajú možnosť interakcie medzi programom a používateľom počas jeho behu. Ladenie programov je zdĺhavé.

1.4 Systémy so zdieľaným časom

Systémy so zdieľaným časom (time-sharing-ové systémy) sú variantom multiprogramovania, v ktorých každý používateľ komunikuje s počítačom pomocou terminálu. V týchto systémoch sa vykonáva viac úloh súčasne, pričom sa procesor prideľuje každej úlohe na krátky časový interval. Prepínanie medzi úlohami sa deje tak rýchlo, že používateľ má možnosť komunikovať s programom počas jeho behu. V týchto systémoch sa programom, ktoré sú zavedené do pamäte a vykonávajú sa, hovorí *proces*.

Systémy so zdieľaným časom dávajú možnosť, aby počítačový systém bol zdieľaný viacerými používateľmi naraz. Príkazy zadávané z klávesnice pri interaktívnej práci sú obyčajne krátke a spotrebovávajú málo času procesora. Taktiež rýchlosť, s ktorou užívatelia zadávajú tieto príkazy je

nepomerne nižšia, ako je rýchlosť ich spracovávania procesorom. Tieto predpoklady dávajú možnosť prepínať medzi procesmi tak rýchlo, že každý používateľ má zdanie, že disponuje celým časom procesora.

Systémy so zdieľaným časom sú zložitejšie ako systémy, využívajúce multiprogramovanie. Aj v tomto prípade sa v pamäti udržiava viac procesov naraz. Pre dosiahnutie prijateľnej odozvy, je potrebné presúvať úlohy z pamäte na disk a obrátene. Disk slúži ako záložná pamäť operačnej pamäte. Pre efektívne zvládnutie tejto úlohy sa často používajú techniky virtualizácie pamäte. Tieto techniky dovoľujú, aby sa vykonával proces, ktorý nie je celý v pamäti. Hlavnou výhodou systémov s virtuálnou pamäťou je možnosť spúšťať procesy, ktoré sú väčšie, ako je operačná pamäť.

Systémy so zdieľaným časom poskytujú *on-line* súborový systém. Súborový systém je umiestnený na disku a preto operačný systém musí poskytnúť aj správu diskov, ako aj ďalších periférnych zariadení. Pretože v systémoch so zdieľaným časom beží viac procesov naraz, je potrebné zaistiť mechanizmy komunikácie medzi nimi, ako aj mechanizmy synchronizácie ich vykonania v určitých situáciach.

Multiprogramovanie a zdieľanie času sú najpodstatnejšími rysmi moderných operačných systémov a sú základnými témami ďalších kapitol.

1.5 Systémy personálnych počítačov

Personálne počítače sa objavili v 70-tych rokoch. Sú to mikropočítače, ktoré sú omnoho menšie a lacnejšie ako sálové počítače (mainframe), na ktorých sa prevádzkujú systémy so zdieľaným časom.

Operačné systémy personálnych počítačov nie sú ani viacužívateľské, ani so zdieľaným časom. U týchto systémov už nie je na prvom mieste využitie času procesora a periférnych zariadení, ale pohodlie používateľa pri práci so systémom a odozva systému. Najrozšírenejšie operačné systémy používané v personálnych počítačoch (do r.1994) sú MS-DOS a Apple Macintosh. Unix, aj keď pôvodne určený pre veľké sálové počítače, sa tiež rýchlo udomácnil na personálnych počítačoch. Pri návrhu týchto systémov sa využilo mnoho skúseností z predchádzajúcich operačných systémov, pričom v tomto prípade nižšia cena a určenie systému pre osobné použitie v mnohom poznamenalo jeho rysy.

V čase, keď sa vlastnosti veľkých operačných systémov prispôsobovali mikropočítačom, boli vyvinuté výkonnejšie, rýchlejšie a zložitejšie hardvérové systémy, určené pre náročnejšie úlohy. Sú to tzv. personálne pracovné stanice. V dnešnej dobe pracovné stanice už nie sú tak drahé a navyše sú prístupné aj cez počítačovú sieť.

1.6 Paralelné systémy

Väčšina operačných systémov sa objavila v čase, keď sa používali len jednoprocesorové systémy. V dnešnej dobe vývoj smeruje k viacprocesorovým systémom. Takéto systémy majú viac ako jeden procesor. Procesory sú tesne spojené a zdieľajú hodiny, zbernicu, niekedy pamäť a periférne zariadenia. Hlavnými príčinami vývoja týchto systémov je snaha zväčšiť priepustnosť, výkon a spoľahlivosť systému.

Najrozšírenejšie viacprocesorové systémy používajú *symetrický model*. V tomto modeli každý procesor prevádzkuje identickú kópiu operačného systému a tieto kópie komunikujú medzi sebou podľa potreby. *Asymetrický model* na druhej strane udeľuje jednému z procesorov osobitnú úlohu a to riadenie systému. Podriadené procesory dostávajú úlohy od tohoto master-procesora. Rozdiel medzi symetrickým a asymetrickým modelom môže byť výsledkom návrhu hardvéru alebo softvéru.

1.7 Distribuované systémy

Súčasný trend vývoja počítačových systémov smeruje k distribúcii výpočtov medzi viaceré procesory. Na rozdiel od paralelných systémov, procesory distribuovaného systému nezdieľajú hodiny a pamäť. Každý procesor má svoju vlastnú pamäť. Procesory medzi sebou komunikujú cez rôzne komunikačné linky, ako sú napr. rýchle zbernice alebo telefónne linky.

Procesory v distribuovanom systéme môžu byť rôzne podľa výkonnosti a funkcie. Môžu to byť personálne počítače, pracovné stanice, servery, minipočítače alebo veľké sálové počítače.

Distribuované systémy umožňujú zdieľanie prostriedkov, zvýšenie rýchlosti výpočtov, zvýšenie spoľahlivosti a zlepšenie možnosti komunikácie a výmeny dát medzi jednotlivými systémami.

1.8 Systémy reálneho času

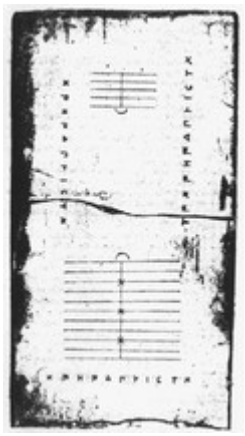
Systémy reálneho času majú špeciálne použitie. Využívajú sa často na riadenie vedeckých experimentov, na riadenie procesov v priemysle, v medicíne pre trojrozmerné zobrazovanie, v automobiloch pre automatické vstrekovanie paliva a v mnohých iných prípadoch. Systém reálneho času má presne stanovené časové obmedzenia pre spracovanie úloh. Na rozdiel od systémov so zdieľaným časom, u systémov reálneho času sa predpokladá, že fungujú správne jedine v prípade, že vrátia správny výsledok v požadovanom čase.

Systémy reálneho času sa delia na *hard real-time* a *soft real-time* systémy. Hard real-time systém garantuje splnenie kritickej úlohy načas. Takéto systémy sa často používajú ako riadiace zariadenia pre presne určené úlohy. Soft real-time systémy nemajú také prísne časové obmedzenia pre splnenie kritických úloh, ale poskytujú im najvyššiu prioritu až do ich ukončenia. Tieto systémy nachádzajú uplatnenie v multimédiách, pri spracovaní virtuálnej reality, vo vedeckých výskumoch a inde.

1.9 Historický vývoj

Nasledujúci historický prehľad naznačuje niektoré myšlienky a udalosti, ktoré zásadným spôsobom ovplyvnili vývoj sveta počítačov.

300 pred n.l. **Počítacia tabuľka** –pomôcka pre rôzne výpočty, najstarší nález tohto typu, pochádza z gréckeho ostrova Salamis

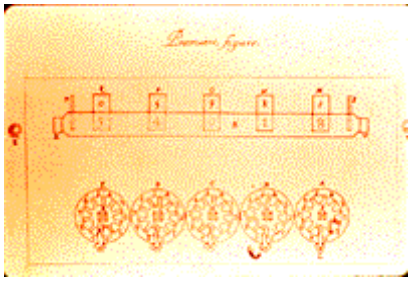


Obr. 1.4 Počítacia tabuľka

antická éra **Abacus -** pomôcka pre výpočty, existujúca v mnohých variantoch, používaná až do dnešných čias. (guľkové počítadlo)

1642 **Kalkulačka Blaisea Pascala** (1623-1662) - "Pascaline", prenosy, akumulácia súčtu, vyrábaná a využívaná v obchodoch, Francúzsko

Obr. 1.5 Princíp kalkulačky *Pascaline* –historický nákres



1804 **Tkáčsky stav Jacquarda**, programovaný pomocou diernych štítkov, revolučný skok vo výrobe kobercov, Francúzsko

1834 **Charles Babbage a lady Augusta Ada zostrojili „Analytical Engine**“ - počítací stroj poháňaný parným strojom, ktorý bol riadený programom na diernych štítkoch, lady Ada navrhuje **princíp podprogramov a** Charles Babbage - **podmienené skoky**; považovaný za prvý „moderný“ počítač, Anglicko

Obr. 1.6 Charles Babbage

1854 **George Boole (Anglicko) publikuje „Laws of Thought“ - základy boole-ovskej algebry** a logiky; Anglicko

1919 **dvojstavový preklápací obvod**, Eccles, Jordan; USA

1928 ENIGMA **- kódovací stroj**; Nemecko

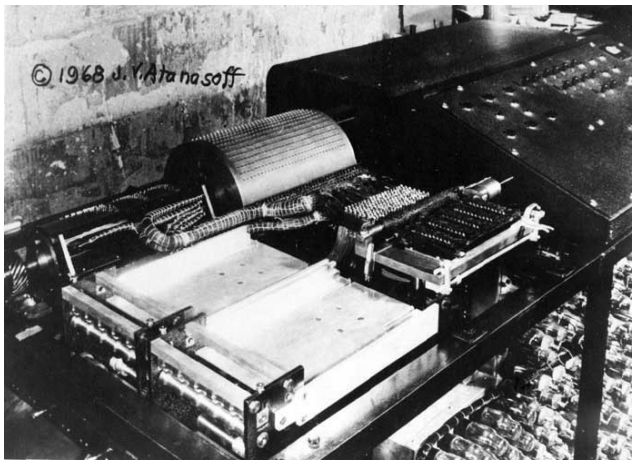
1930 Model 1, **prvý elektromechanický stroj**

1936 **abstraktný model počítača, Alan Turing**; Anglicko

1938 **COLOSSUS, primitívny počítač Alana Turinga**, používaný hlavne pre lúštenie nemeckých šifier

1939 **John Atanasoff - američan bulharského pôvodu, otec moderného elektronického počítača** (Atanasoff-Berry (ABC) computer), ktorý využíva logické operácie, binárne kódovanie, regenerujúcu sa pamäť, autorstvo bolo dokázané až v r.1970 podľa zachovaných rukopisov

Obr. 1.7 John Atanasoff, otec moderného elektronického počítača



Obr. 1.8 Posledná fotografia ABC počítača z roku 1968

1941 **elektromechanický kalkulátor** - 64 slov pamäte, Konrad Zuse; Rakúsko

1944 **MARK 1, Elektromechanický počítač**, Howard Aiken; USA

1945 **ENIAC** (Electrical Numerical Integrator and Computer), Mauchly, Eckert; počítač, využívajúci princíp počítača Johna Atanasoffa,USA

1948 **prvý tranzistor,** Bell Labs; USA

1949 EDSAC, zostrojil Alan Turing, **prvýkrát použitá knižnica podprogramov**

1951 EDVAC, **John von Neumann a kolektív, moderná architektúra (von Neumanova) počítača,** univerzita v Pensilvánií; USA

1952 mikroprogramovanie, Maurice Wilkes

1954 **UNIVAC 1,** prvý počítač pre Ministerstvo obrany USA

MATH_MATIC - kompilovaný programovací jazyk pre UNIVAC 1
FORTRAN, programovací jazyk pre vedecké účely vyvinutý v IBM,

prvý asembler, IBM

1955 **prerušenia a asynchrónne V/V operácie**, prvýkrát použité v počítačoch UNIVAC, firmy Remington Rand

TRIDAC - prvý počítač na tranzistoroch

1956 **V/V kanály a priamy prístup do pamäte**, prvýkrát použité v počítačoch UNIVAC, firmy Remington Rand

1958 ATLAS, **prvý počítač s virtuálnou pamäťou**, Univerzita v Manchesteri; Anglicko

algoritmický programovací jazyk **ALGOL58**

programovací jazyk **LISP**

1959 programovací jazyk **COBOL**

1960 programovací jazyk **ALGOL60**

1964 LAN - **prvá lokálna sieť**

1965 MULTICS, MIT, predchodca operačného systému Unix

1966 operačný systém OS-360, IBM

1968 THE, operačný systém - univerzita Eindhoven, Holandsko, vrstvová štruktúra

1969 laserová tlačiareň

1970 programovací jazyk PASCAL, Nikolaus Wirth; Švajčiarsko

1972 programovacie jazyky C a Smalltalk

1973 UNIX pre široké použitie

1975 **osobný počítač**, Apple, Commodore PET, Radio Shak

1976 MCP - operačný systém pre architektúru s viacerými procesormi

1978 počítač VAX spoločnosti Digital, USA

1979 UNIX 3BSD

programovací jazyk ADA (pomenovaný podľa lady Augusty Ady)

1981 personalný počítač IBM PC, 16 K RAM

1982 programovací jazyk Turbo PASCAL

programovací jazyk Modula2

1984 Apple MacIntosh - prvý osobný počítač s grafickým interfejsom

PostScript

1986 programovací jazyk **C++**

1987 operačný systém OS/2, IBM

1988 Unix workstation

NeXT, **objektovo orientovaný operačný systém**

1990 operačný systém Windows 3.0

1991 operačný systém Linux OS

1992 operačný systém Solaris

1993 operačný systém Windows NT

1999 operačný systém Windows 2000

2003 operačný systém Windows 2003

2 HARDVÉR POČÍTAČOV

Skôr ako budeme hovoriť podrobnejšie o operačných systémoch počítačov, je potrebné zoznámiť sa so základnými princípmi a pojmami z oblasti hardvéru. Pod pojmom hardvér budeme rozumieť technické prostriedky z ktorých sa počítač skladá. Jednou z úloh operačného systému je sprostredkovanie komunikácie medzi hardvérom počítača a jeho používateľmi.

2.1 Základné pojmy

Elektronické súčiastky používané pri konštrukcii číslicových počítačov majú schopnosť nachádzať sa v jednom z dvoch rozlíšiteľných stavov (vedie prúd, nevedie prúd). Tým je dané najmenšie množstvo informácie s ktorým môžu číslicové počítače pracovať - jeden bit, ktorý nadobúda hodnoty označované 0 a 1. Všetky údaje v počítači sú potom reprezentované pomocou postupností bitov rôznej dĺžky. Postupom času sa začali preferovať isté dĺžky pred ostatnými (Tab. 1).

Tab. 1

Pomenovanie	Počet bitov - n	Kardinalita – 2^N
Bajt	8	256
Slovo	16	65536
Dvojslovo	32	4294967296
Štvorslovo	64	2^{64}

Reprezentácia údajov v počítači

Každý počítač je konštruovaný tak, že je schopný vykonávať inštrukcie s údajmi s určitou maximálnou dĺžkou. Ak potrebujeme spracovať údaje s väčšou dĺžkou, musíme ich spracovávať postupne. Napr. sčítanie dvoch 16 bitových slov na počítači, ktorý je schopný spracovávať iba 8 bitové informácie, znamená vykonať operáciu sčítania dvakrát – najskôr sčítame nižších 8 bitov každého slova s prenosom a potom sčítame vyšších osem bitov každého slova, pričom musíme brať do úvahy i prenos. Na počítači, ktorý je schopný spracovávať16 bitové slová, by takáto operácia prebehla ako jedna inštrukcia.

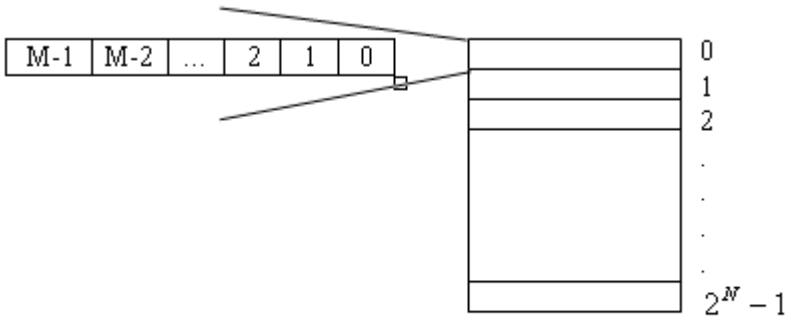
Nezávisle od stupňa vývoja môžeme v každom číslicovom počítači nájsť nasledujúce základné časti:

- **Procesor:** riadi činnosť počítača a vykonáva jeho hlavnú činnosť – spracovanie údajov. V súčasnosti je funkcia procesora často integrovaná do jedného integrovaného obvodu nazývaného mikroprocesor. Keďže cena mikroprocesorov klesá, začínajú sa čoraz viac objavovať počítače s viacerými procesormi.
- **Operačná pamäť:** umožňuje ukladanie údajov a inštrukcií pre procesor. Zvyčajne sa z nej údaje stratia po prerušení napájania.
- **Modul vstupu a výstupu:** slúži pre komunikáciu s ostatnými zariadeniami počítača (vonkajšia pamäť, konzola,...).
- **Systémová zbernica:** umožňuje komunikáciu medzi procesorom, operačnou pamäťou a modulom vstupu a výstupu.

2.2 Operačná pamäť

Operačnú pamäť najlepšie predstavuje údajová štruktúra typu pole. Môžeme ju charakterizovať dvomi údajmi (Obr. 2.1).

- **šírka pamäte** udáva počet bitov jednotlivých prvkov poľa**,**
- **veľkosť pamäte** udáva jej kapacitu, teda koľko prvkov je schopná súčasne uchovávať.



Obr. 2.1 Operačná pamäť so šírkou M a kapacitou 2^N

Veľkosť pamäte je zvyčajne mocnina dvoch (2^N) a jej jednotlivé prvky sú očíslované od nuly po $2^N - 1$. Pri prístupe k prvkom pamäte musíme určiť číslo príslušného prvku – teda vlastne index do poľa prvkov. V počítačovej terminológii sa miesto pojmu index vžil pojem **adresa.**

V súčasnosti býva v bežných počítačoch typu PC k dispozícii od 64 MB (u starých typov PC) do 1 GB operačnej pamäte, pričom M označuje predponu mega ($1024*1024=2^{20}$) a G predponu giga (2^{30}), B označuje bajt.

2.3 Procesor

Procesor je srdcom každého počítača. Jeho úlohou je vyberať inštrukcie z pamäte a vykonávať ich.

2.3.1 Registre procesora

Pre svoju činnosť potrebuje procesor mať k dispozícii pamäť údajov. Pretože prístup k operačnej pamäti je z hľadiska procesora pomalý, má procesor k dispozícii niekoľko pamäťových miest (ich počet je obmedzený kvôli náročnej realizácii) s veľmi rýchlym prístupom, ktoré sa nazývajú **registre.** Registre slúžia na dva účely:

- **Riadiace a stavové registre** používa samotný procesor pre ukladanie informácií potrebných pre jeho činnosť.
- **Všeobecne prístupné registre** sú dostupné programátorovi pre ukladanie ľubovoľných údajov potrebných pre činnosť programu.

2.3.1.1 Riadiace a stavové registre

Medzi informácie dôležité pre spracovanie programu patrí údaj o tom, ktorá inštrukcia bude nasledovať za práve vykonávanou inštrukciou. Na to slúži procesoru register označovaný **PC (program counter - čítač inštrukcií).** Tento register zvyčajne nemôže priamo ovplyvňovať používateľ procesora, je nastavovaný procesorom automaticky. Nepriamo možno jeho hodnotu určiť inštrukciou skoku (či už nepodmieneného alebo podmieneného).

Ďalšiu dôležitú informáciu uchováva **register inštrukcií IR (instruction register).** Je to operačný kód inštrukcie, ktorá sa bude práve vykonávať.

Mnohé algoritmy vykonávané na počítačoch obsahujú podmienený príkaz , keď sa výpočet vetví podľa hodnoty nejakého výrazu. Informáciu o výsledku posledne vyhodnotenej aritmetickej alebo logickej operácie uchováva špeciálny **register príznakov (flag register).** Je možné zistiť napr. paritu hodnoty uloženej v akumulátore (napr. kvôli kontrole prenosu po sériovej linke), ďalej či hodnota v akumulátore je nulová, či došlo pri poslednej aritmetickej operácii k pretečeniu a pod.

2.3.1.2 Všeobecne prístupné registre

Pre programátora je k dispozícii niekoľko registrov ktoré môže použiť podľa svojho uváženia.

Dátové registre slúžia na uloženie údajov potrebných pre činnosť programu. Existuje jeden špeciálne určený register nazývaný **akumulátor (AC)**, ktorý je zdrojom alebo cieľom aritmetických a logických operácií**.**

Adresové registre obsahujú adresy údajov v operačnej pamäti s ktorými sa bude manipulovať. Napr. sú to registre:

- **Indexový register (index register)** je možné použiť pre adresovanie úsekov pamäte vzhľadom na nejaký počiatok, čím môžeme jednoducho implementovať prácu s údajovou štruktúrou typu pole.
- **Ukazovateľ zásobníka (stack pointer)** určuje adresu vrcholu používateľského zásobníka. Menia ho inštrukcie push a pop a je možné ho modifikovať i priamo.

- **Segmentový register (segment pointer)** sa používa v prípade, ak procesor používa segmentové adresovanie pamäte. Tento register uchováva adresu počiatku segmentu.

2.3.2 Vykonávanie inštrukcií

Program vykonávaný procesorom pozostáva z postupnosti inštrukcií uložených v operačnej pamäti. Spracovanie každej inštrukcie procesorom pozostáva z jej prenosu z operačnej pamäte do procesora (fáza výberu, fetch) a z jej analýzy a následného spracovania (fáza spracovania, execution). Tento proces spracovania inštrukcie sa nazýva **inštrukčný cyklus.**

Na začiatku každého inštrukčného cyklu procesor vyberie z operačnej pamäte inštrukciu uloženú na adrese určenej obsahom počítadla inštrukcií (PC) a uloží ju do registra inštrukcií (IR). Nasleduje dekódovanie inštrukcie. Vo všeobecnosti môžeme inštrukcie procesora rozdeliť do štyroch kategórií:

- Inštrukcie prenosu údajov medzi procesorom a operačnou pamäťou.
- Inštrukcie prenosu údajov medzi procesorom a perifémymi zariadeniami.
- Inštrukcie pre aritmetické alebo logické operácie.
- Inštrukcie riadenia toku programu (inštrukcie skoku,...).

Po dekódovaní nasleduje vykonanie inštrukcie a hodnota čítača inštrukcií je nastavená na inštrukciu, ktorá sa má vykonať ako ďalšia v poradí (čo nemusí byť nutne inštrukcia, ktorá nasleduje za už vykonanou inštrukciou v operačnej pamäti).

Uvažujme o jednoduchom procesore, ktorý je schopný spracovávať inštrukcie s dĺžkou 8 bitov. Budeme sledovať vykonanie fragmentu programu pozostávajúceho z dvoch inštrukcií – najprv pripočítame k hodnote akumulátora (AC) konštantnú hodnotu a následne podľa výsledku tejto operácie vykonáme podmienenú inštrukciu skoku, čím dosiahneme vetvenie programu podľa danej podmienky. Začiatok nášho kódu bude na adrese 100. Nech operačný kód inštrukcie pre pripočítanie priamej hodnoty k akumulátoru je 64. Hodnota ktorú máme pripočítať je uložená za operačným kódom. Operačný kód inštrukcie podmieneného skoku bude 192. Tento kód bude testovať príznak zero v registri príznakov a ak je jeho hodnota rovná nule (teda výsledok poslednej aritmetickej operácie nebol rovný nule), vykoná sa skok na adresu, ktorá je uložená za operačným kódom, inak sa pokračuje vo vykonávaní nasledujúcej inštrukcie. Ak uvažujeme, že operačná pamäť procesora má veľkosť 64KB, potrebujeme na jej adresovanie 16 bitové adresy. Vykonávanie týchto dvoch inštrukcií môžeme popísať nasledujúcimi krokmi:

PC obsahuje adresu inštrukcie, ktorá sa má práve vykonať. Inštrukcia je prenesená do IR a PC je nastavené na ďalšiu adresu.

Operačná pamäť			Registre procesora		
Adresa	Hodnota		Register	Hodnota pred	Hodnota po
100	64		PC	100	101
101	5		IR	xx	64
102	192		AC	79	79
103	0		zero	1	1
104	1				
105	78				

Obr. 2.2 Vykonanie kódu programu od adresy 100 - 1. krok

Dekódovaním inštrukcie sa zistí, že jeden z operandov je uložený v pamäti na adrese určenej PC. Operand sa prenesie do aritmeticko-logickej jednotky, adresa PC sa opäť zväčší o jedna.

Operačná pamäť	
Adresa	Hodnota
100	64
101	5
102	192
103	0
104	1
105	78

Registre procesora		
Register	Hodnota pred	Hodnota po
PC	101	102
IR	64	64
AC	79	79
zero	1	1

Obr. 2.3 Vykonanie kódu programu od adresy 100 - 2. krok

Vykoná sa inštrukcia, jej výsledok sa uloží do akumulátora. Podľa výsledku sa nastaví register príznakov.

Operačná pamäť	
Adresa	Hodnota
100	64
101	5
102	192
103	0
104	1
105	78

Registre procesora		
Register	Hodnota pred	Hodnota po
PC	102	102
IR	64	64
AC	79	84
zero	1	0

Obr. 2.4 Vykonanie kódu programu od adresy 100 - 3. krok

Začína ďalší inštrukčný cyklus, PC obsahuje opäť adresu operačného kódu inštrukcie. Ten je prenesený do IR a hodnota PC je nastavená na ďalšiu adresu.

Operačná pamäť	
Adresa	Hodnota
100	64
101	5
102	192
103	0
104	1
105	78

Registre procesora		
Register	Hodnota pred	Hodnota po
PC	102	103
IR	64	192
AC	84	84
zero	0	0

Obr. 2.5 Vykonanie kódu programu od adresy 100 - 4. krok

Dekódovaním sa zistí, že je to inštrukcia podmieneného skoku v závislosti na hodnote príznaku zero. Riadenie programu sa má preniesť na adresu uloženú bezprostredne za operačným kódom inštrukcie ak je hodnota príznaku zero nulová. Pretože táto podmienka je v našom prípade splnená, bude hodnota registra PC naplnená určenou adresou a začne sa nový inštrukčný cyklus na adrese určenej PC.

Operačná pamäť	
Adresa	Hodnota
100	64
101	5
102	192
103	0
104	1
105	78

Registre procesora		
Register	Hodnota pred	Hodnota po
PC	102	256
IR	192	192
AC	84	84
zero	0	0

Obr. 2.6 Vykonanie kódu programu od adresy 100 - 5. krok

Adresa skoku je v operačnej pamäti uložená tak, že najprv je uložený nižší bajt slova (NB) a potom vyšší bajt (VB), teda adresa = 256 * VB + NB.

2.3.3 Prerušenie

Procesor okrem vykonávania inštrukcií musí byť schopný reagovať aj na rôzne asynchrónne sa vyskytujúce udalosti. Zdrojom týchto udalostí môže byť:

1. Samotný procesor – napr. pri pretečení aritmetickej operácie, delení nulou a pod.
2. Časovač – zabezpečuje napr. pravidelné vykonávanie určenej činnosti v presne stanovenom intervale.
3. Periférne zariadenia – oznamujú procesoru ukončenie vstupno-výstupnej operácie, resp. žiadosť o obsluhu.
4. Hardvér počítača – napr. pri chybe parity pamäte, výpadku napájania a pod.

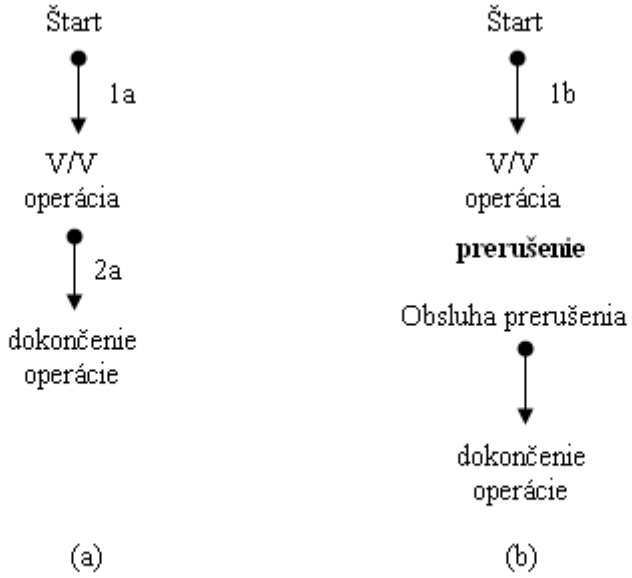
Prerušenia poskytujú možnosť zlepšiť využitie jednotlivých častí počítača. Uvažujme o počítači, ktorý posiela spracované údaje pomocou pomalej (vzhľadom na rýchlosť počítača) komunikačnej linky. Program pre túto činnosť by mohol pozostávať z nasledujúcich krokov:

1. Inicializácia.
2. Spracuj získané údaje.
3. Počkaj, kým komunikačná linka nebude voľná.
4. Pošli údaje.
5. Ak sú ďalšie údaje na spracovanie, pokračuj krokom 2.
6. Koniec.

Z hľadiska použitia prerušení je zaujímavý krok 4. Pre poslanie údajov použijeme rutinu poskytovanú operačným systémom. Jej realizácia závisí od toho, či využijeme systém prerušení poskytovaný príslušným hardvérom. Dve možné riešenia tejto rutiny sú na Obr. 2.7.

Najprv rozoberieme prípad, keď nebudeme používať prerušenia. Činnosť označená 1a sa vykoná po štarte a reprezentuje prípravu vstupno–výstupnej operácie (napr. prenos údajov do pomocnej pamäte, nastavenie registrov, nastavenie príznaku obsadenia komunikačnej linky a pod.). Po nej nasleduje vykonanie samotnej vstupno-výstupnej operácie. V časti 2a sa čaká na jej dokončenie. To sa robí periodickým dotazovaním (polling) periférie, či už sa činnosť ukončila. Po ukončení operácie sa vykonajú úkony potrebné pre dokončenie rutiny, napr. nastavenie príznaku uvoľnenia komunikačnej linky a pod. Je zrejmé, že v takomto prípade sa riadenie programu vráti do hlavnej slučky až po ukončení komunikácie.

Pri využití prerušovacích prostriedkov počítača a periférneho zariadenia by mohla rutina pre obsluhu komunikačnej linky pracovať nasledovne. Prípravná fáza 1b je rovnaká ako aj v rutine bez použitia prerušení. Po vykonaní vstupno-výstupnej operácie sa však riadenie vráti ihneď do hlavnej slučky a tá môže pokračovať spracovaním ďalších údajov.



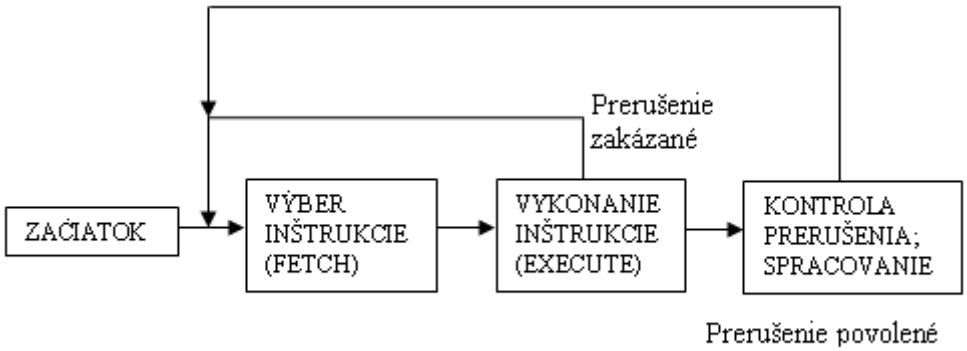
Obr. 2.7 Rutina pre obsluhu komunikačnej linky (a) bez použitia prerušenia, (b) s použitím prerušenia

Ukončenie vstupno-výstupnej operácie oznámi komunikačná linka vygenerovaním signálu prerušenia. Procesor preruší práve vykonávanú činnosť a vstúpi do príslušnej rutiny pre obsluhu prerušenia. V nej sa vykonajú potrebné operácie (napr. nastavenie príznaku uvoľnenia komunikačnej linky, a pod.). Potom sa pokračuje od miesta, kde bolo vykonávanie programu prerušené. V závislosti od pomeru rýchlostí komunikačnej linky a spracovania údajov v hlavnej slučke programu nastanú dva prípady. Ak je rýchlosť posielania údajov do komunikačnej linky väčšia než rýchlosť spracovania údajov programom, prerušenie činnosti nastane počas vykonávania spracovania údajov (bod 2). Potom test v bode 3 nájde komunikačnú linku voľnú a pripravenú na odoslanie ďalších údajov. Ak je však rýchlosť spracovania údajov vyššia než rýchlosť posielania údajov do komunikačnej linky, program vykoná spracovanie údajov a bude čakať na uvoľnenie komunikačnej linky (bod 3). V každom prípade však je zrejmé, že procesor je využitý efektívnejšie, pretože sa môže venovať vykonávaniu výpočtov i v čase keď prebieha prenos údajov do komunikačnej linky.

2.3.3.1 Realizácia mechanizmu prerušení

Mechanizmus prerušení je realizovaný modifikáciou inštrukčného cyklu, ktorý sme si popísali v odseku 2.3.2, podľa Obr. 2.8.

Vidíme, že po vykonaní každej inštrukcie sa testuje, či nevznikla požiadavka na prerušenie. Ak áno, nasleduje jej obslúženie, v opačnom prípade sa pokračuje v ďalšom inštrukčnom cykle. Z tohoto popisu vyplýva, že v skutočnosti nie je reakcia procesora na vzniknutú požiadavku prerušenia okamžitá a doba reakcie závisí od dĺžky práve vykonávanej inštrukcie (resp. od doby, ktorá uplynie od vzniku požiadavky na prerušenie po dokončenie práve vykonávanej inštrukcie). S touto dobou musia preto počítať konštruktéri prerušovacieho podsystému.

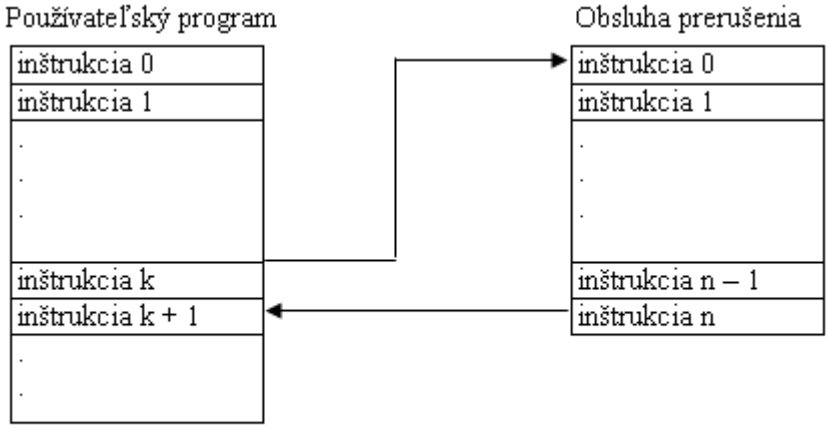


Obr. 2.8 Inštrukčný cyklus akceptujúci prerušenie

2.3.3.2 Spracovanie prerušenia

Každé prerušenie procesora vyvolá celú postupnosť krokov, ktoré treba vykonať pre obsluhu prerušenia:

1. Periférne zariadenie vyšle žiadosť o prerušenie procesora.
2. Procesor dokončí vykonávanie aktuálnej inštrukcie.
3. Procesor testuje existenciu požiadavky na prerušenie, zistí jej prítomnosť. Potvrdí prijatie požiadavky zariadeniu, ktoré o ňu žiadalo, čím umožní zariadeniu ukončiť svoju žiadosť.
4. Procesor musí pripraviť prenos riadenia programu do prerušovacej rutiny. Je potrebné odpamätať adresu inštrukcie, ktorou sa bude pokračovať po obslúžení prerušenia (tá je uložená v registri PC). Na to sa zvyčajne používa zásobník. Odpamätanie ostatných hodnôt registrov sa zvyčajne necháva na prerušovaciu rutinu.
5. Procesor určí adresu prerušovacej rutiny a jej hodnotou naplní register PC. V závislosti od hardvérovej platformy a operačného systému môže byť rôznym zdrojom prerušenia pridelená spoločná prerušovacia rutina, alebo má každý zdroj vlastnú rutinu.
6. Začne sa nový inštrukčný cyklus. V ňom sa vyberie inštrukcia z adresy na ktorú ukazuje register PC (prvá inštrukcia rutiny pre obsluhu prerušenia).
7. Vykonávanie programu pokračuje až po dokončenie obsluhy prerušenia.
8. Prerušovacia rutina obnoví hodnoty registrov.
9. Špeciálnou inštrukciou návratu z podprogramu sa vráti riadenie programu na miesto kde by sa pokračovalo, keby nedošlo k vzniku prerušenia.



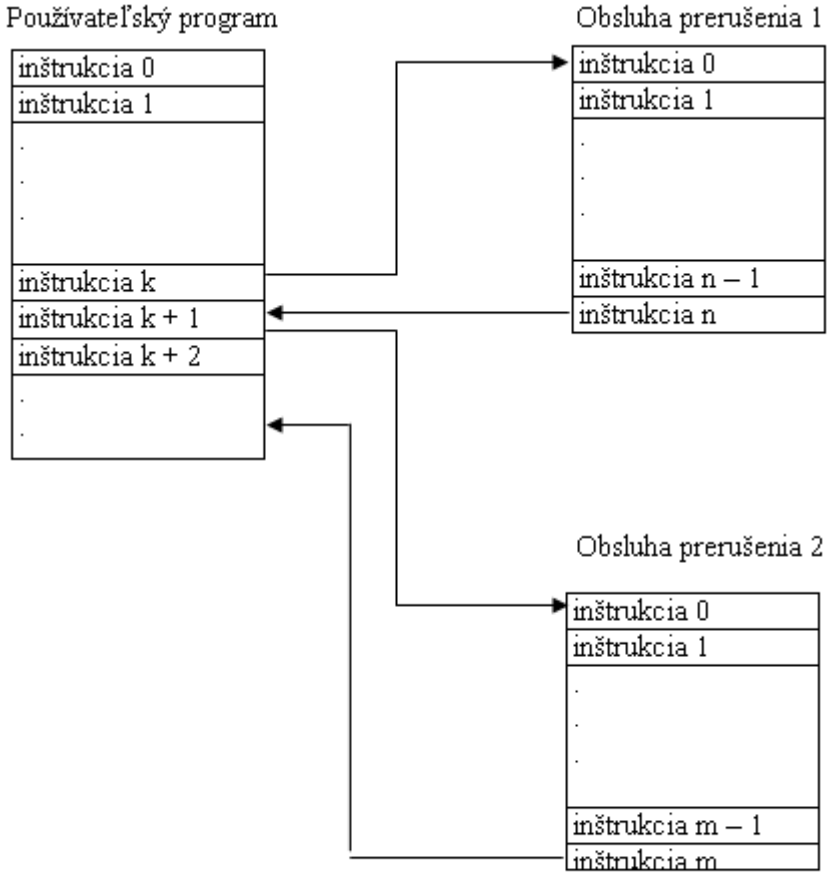
Obr. 2.9 Priebeh riadenie programu v prípade vzniku požiadavky na prerušenie

Pri vytváraní rutín pre obsluhu prerušenia si treba uvedomiť, že je nutné uchovať všetky informácie potrebné pre obnovenie stavu, v akom sa procesor nachádzal pred zavolaním obsluhy prerušenia. V opačnom prípade by dochádzalo k nereprodukovateľnému chybnému správaniu sa programov v dôsledku zmeny stavu procesora po ukončení obsluhy prerušenia.

2.3.3.3 Viacnásobné prerušenie

Doteraz sme neuvažovali prípad, že sa vyskytne požiadavka na prerušenie procesora v dobe, kedy je obsluhovaná iná požiadavka na prerušenie. Existujú dva principiálne rôzne prístupy k tomuto problému.

Prvým prístupom je, že pri výskyte prerušenia sa **zakáže akceptovanie ďalších prerušení** (Obr. 2.10). Periféria žiadajúca o prerušenie v tomto stave musí počkať na dokončenie prebiehajúcej obsluhy prerušenia. Až potom sa povolí akceptovanie ďalších žiadostí o prerušenie (to môže zabezpečiť práve špeciálna inštrukcia pre návrat z prerušenia). Tento prístup je jednoduchý, ale dovoľuje obsluhu prerušení len sekvenčne.

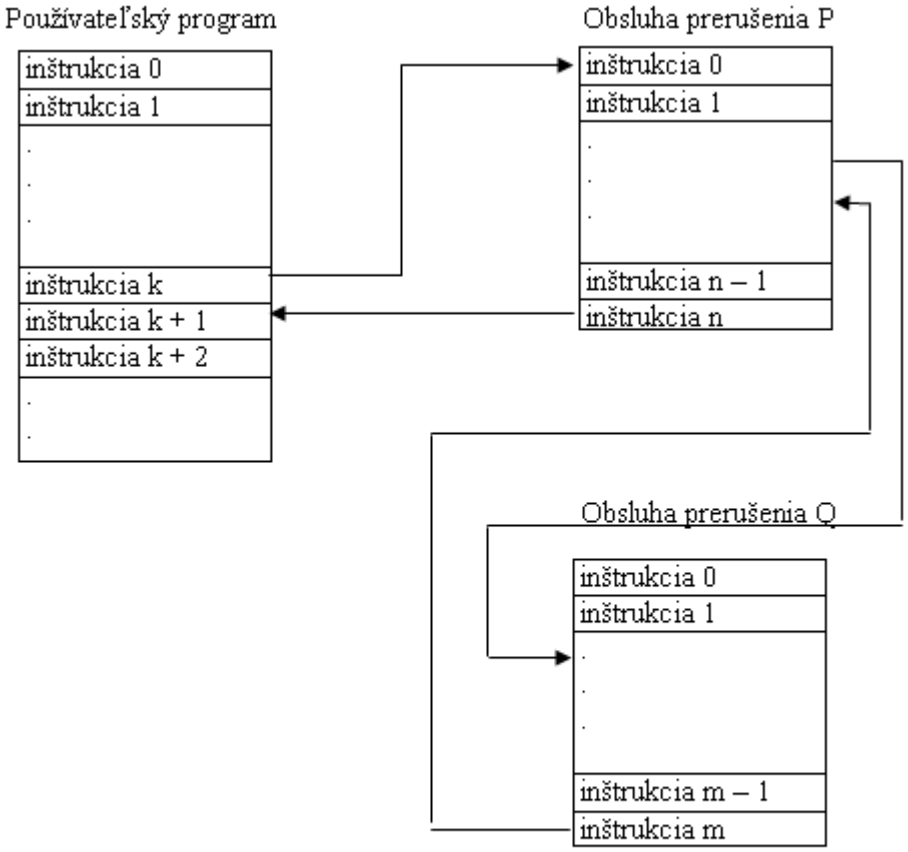


Obr. 2.10 Priebeh riadenia programu v prípade vzniku viacerých požiadaviek na prerušenie, sekvenčné spracovanie

Rôzne periférne zariadenia majú rôzne nároky na rýchlosť odozvy na ich požiadavku prerušenia a rôznu dobu obsluhy vzniknutého prerušenia. V prípade doteraz uvažovaného spôsobu obsluhy viacerých prerušení by sa preto mohlo stať, že pre niektoré zariadenia nebudú ich požiadavky na dobu odozvy na prerušenie splnené, čím môže dôjsť napr. k strate údajov.

Ukázalo sa preto výhodné zvoliť pri spracovaní viacerých výskytov prerušení inú stratégiu. **Jednotlivým prerušeniam pridelíme prioritu**. Potom ak je obsluhované prerušenie s prioritou P, a príde požiadavka na obslúženie prerušenia s vyššou prioritou Q, bude práve vykonávaná rutina pre spracovanie prerušenia s prioritou P prerušená a začne sa vykonávať rutina pre obsluhu prerušenia

s prioritou Q. Po jej ukončení sa riadenie vráti do rutiny obsluhujúcej prerušenie s prioritou P (Obr. 2.11).



Obr. 2.11 Priebeh riadenie programu v prípade vzniku viacerých požiadaviek na prerušenie, spracovanie podľa priorít

2.4 Vstupno-výstupné operácie

Vo väčšine používateľských programov sa vyskytujú požiadavky na komunikáciu s perifémymi zariadeniami. Túto komunikáciu na najnižšej úrovni zvyčajne zabezpečuje operačný systém. Ten zároveň poskytuje používateľským programom určitý komfort a abstrakciu pri práci s perifémymi zariadeniami. Pretože sa ďalej budeme zaoberať princípmi operačných systémov, popíšeme si v ďalšom rôzne možnosti realizácie komunikácie s perifémymi zariadeniami.

Ako vlastne prebieha komunikácia procesora s perifémymi zariadeniami? Pretože v každom systéme sa nachádza zvyčajne viacero perifémych zariadení, v prvom rade musíme byť schopní špecifikovať periférium s ktorou budeme komunikovať. Každé periférne zariadenie by preto mohlo mať pridelenú určitú **adresu.** Procesor a periférne zariadenie si obecne vymieňajú informácie rôzneho charakteru:

- riadiace informácie, napr. požiadavka previnúť magnetickú pásku na začiatok, určenie smeru nasledujúcej V/V operácie, a pod.,
- stavové informácie, napr. stav tlačiarne (pripravená, obsadená, chyba),
- dáta, s ktorými sa má V/V operácia uskutočniť.

Aby bolo možné jednotlivé typy informácií rozlíšiť, je perifémemu zariadeniu priradených **viacero adries**, pričom je pevne stanovené, ktorá adresa slúži pre prenos riadiacich, resp. stavových

informácií a dát. Podobne ako pri operačnej pamäti, existuje teda istý adresný priestor periférnych zariadení.

Periférne zariadenia ukladajú riadiace a stavové informácie do pamäťových miest nazývaných **registre.** Pretože pridelenie adries periférnym zariadeniam vlastne určuje mapovanie adresného priestoru periférií na jednotlivé registre periférnych zariadení, častokrát sa používa pojem **adresa riadiaceho registra**, príp. **adresa stavového registra.**

Inštrukcie procesora pre prácu s perifémymi zariadeniami zahŕňajú operácie ako prenos dát na určenú adresu, resp. prenos dát z určenej adresy z adresného priestoru periférnych zariadení.

V princípe môžeme komunikáciu medzi procesorom a perifériami rozdeliť do troch kategórií:

1. programovo riadený vstup a výstup (ďalej V/V),
2. V/V riadený prerušením,
3. V/V pomocou mechanizmu priameho prístupu do pamäte.

2.4.1 Programovo riadený vstup a výstup

Tento prístup je najmenej náročný na hardvérové možnosti procesora i periférneho zariadenia. Popíšeme si ho na príklade posielania riadku textu na tlačiareň.

1. Prečítaj obsah stavového registra tlačiarne.
2. Ak nastala chyba pri tlači, koniec, oznám chybový stav do volajúceho programu.
3. Ak tlačiareň nie je pripravená prijať znak, pokračuj bodom 1.
4. Zapíš do dátového registra tlačiarne ďalší znak.
5. Zápisom do riadiaceho registra oznám tlačiarni ďalší znak na spracovanie.
6. Počkaj na potvrdenie prijatia znaku tlačiarňou.
7. Pokiaľ sú ďalšie znaky na tlačenie, pokračuj bodom 1.

Vidíme, že komunikácia nie je efektívna z hľadiska využitia procesora, pretože ten vzhľadom na relatívne malú rýchlosť tlačiarne strávi mnoho času v čakacích slučkách (kroky 1-3 a 6 algoritmu). Zvýšenie využitia času procesora môžeme dosiahnuť tým, že čakacie slučky odstránime.

2.4.2 V/V riadený prerušením

Za cenu rozšírenia hardvéru o podsystém prerušení môžeme dosiahnuť zvýšenie efektívnosti využitia času procesora. Táto problematika bola diskutovaná v odseku 2.3.3. V princípe ide o to, že procesor nemusí aktívne čakať na dokončenie V/V operácie, ale je o ňom informovaný perifémym zariadením prostredníctvom prerušenia. So zvyšovaním rýchlosti periférnych zariadení sa však ukázalo, že pre niektoré rýchle periférne zariadenia nie je ani prerušením riadený V/V dostatočne rýchly a bolo nutné hľadať iné riešenia obsluhy periférnych zariadení.

2.4.3 Priamy prístup do pamäte

Konštruktéri periférnych zariadení sú nútení tlakom používateľov vytvárať stále rýchlejšie periférne zariadenia. To však so sebou prináša nasledujúce problémy:

1. Rýchlosť komunikácie s perifémymi zariadeniami je limitovaná rýchlosťou procesora.
2. Procesor strávi mnoho času obsluhou periférnych zariadení.

Pre veľmi rýchle periférne zariadenia je riešením problému komunikácie s procesorom vytvorenie špecializovaného modulu nazvaného **DMA** (Direct Memory Access). Ako už jeho meno

napovedá, je vhodný najmä pre prenos veľkých objemov dát medzi perifériou a operačnou pamäťou. Snaha o odbremenenie procesora vyústila nakoniec do toho, že tento prenos sa deje celkom bez jeho účasti.

Úlohou procesora je len nastaviť údaje potrebné pre prenos dát:

- smer prenosu (z alebo do operačnej pamäte),
- adresu periférneho zariadenia s ktorým sa má komunikácia uskutočniť,
- adresu v operačnej pamäti, ktorá bude použitá na prenos,
- veľkosť prenášaných dát.

Po nastavení údajov a spustení prenosu, procesor pokračuje vo svojej činnosti. Riadenie komunikácie je celé v réžii DMA modulu. Po dokončení prenosu je procesor informovaný pomocou prerušenia.

Prístup do operačnej pamäte je umožnený prostredníctvom systémovej zbernice. V prípade, že prebieha DMA prenos, tak o jej použitie súperia procesor a DMA modul. Ak je zbernica obsadená DMA modulom a procesor potrebuje komunikovať s operačnou pamäťou, musí počkať na jej uvoľnenie. Tým sa rýchlosť spracovania programu procesorom zmenšuje. Toto spomalenie je však akceptovateľné a je to daň za rýchle V/V operácie.

3 ŠTRUKTÚRA OPERAČNÝCH SYSTÉMOV

Operačný systém poskytuje prostredie, v ktorom sa vykonávajú programy. Moderné operačné systémy sa značne líšia jeden od druhého v svojom vnútornom prevedení, pretože pri návrhu nového operačného systému sa predovšetkým berú do úvahy predpoklady, ktoré má spĺňať. Existuje niekoľko pohľadov, podľa ktorých sa dajú hodnotiť operačné systémy (OS). Prvý je podľa služieb, ktoré OS poskytuje, druhý je podľa interfejsu pre používateľa a programátora, tretí sa dá získať posúdením jednotlivých komponentov OS a v spojení medzi nimi. V ďalšom texte sa budeme snažiť uplatniť všetky tri pohľady.

3.1 Komponenty OS

Veľký a zložitý systém sa obyčajne navrhuje rozdelením na menšie časti. Každá z týchto častí by mala byť dobre navrhnutá, s presne definovanými vstupmi, výstupmi a funkciami. Všetky OS nemajú rovnakú štruktúru, ale väčšina moderných operačných systémov pozostáva z komponentov, ktoré sú uvedené ďalej.

3.1.1 Správa procesov

Proces môžeme pokladať za program, ktorého inštrukcie sa vykonávajú procesorom v určitom kontexte. Používateľský program, bežiaci v režime so zdieľaným časom je proces. Obyčajne jedna dávková úloha je tiež proces. Procesom je aj systémová úloha, ktorá obsluhuje tlačiareň. Ďalšia presnejšia definícia bude uvedená v kapitole o procesoch.

Proces potrebuje určité prostriedky pre svoje vykonanie, ako sú napr. čas procesora, pamäť, súbory a V/V zariadenia. Tieto prostriedky sú pridelené procesu buď pri jeho vzniku, alebo počas jeho behu.

Dôležité je uvedomiť si, že program sám o sebe nie je proces, program je pasívna jednotka, je to súbor, ktorý obsahuje inštrukcie, zatiaľ čo proces je aktívna jednotka, kde čítač inštrukcií určuje ďalšiu inštrukciu pre vykonanie. Procesor vykonáva inštrukcie jednu po druhej až do ukončenia procesu. V každom okamihu procesor vykonáva jednu inštrukciu daného procesu. Aj keď je možné, že viacej procesov beží nad jedným programom, každý proces má svoju sekvenciu vykonávania. Opačný jav je tiež bežný, t.j. jeden program počas svojho behu spúšťa viacej procesov.

Proces je základná jednotka práce systému. Systém pozostáva z množiny procesov, z ktorých niektoré sú systémové a ďalšie sú používateľské. Všetky tieto procesy sa vykonávajú paralelne, zdieľaním času procesora.

OS zaisťuje nasledujúce činnosti v súvislosti s riadením procesov:

- tvorba a rušenie systémových a používateľských procesov,
- pozastavenie a opätovné spustenie procesov,
- poskytuje mechanizmy synchronizácie chodu procesov,
- poskytuje mechanizmy komunikácie medzi procesmi,
- poskytuje prevenciu a obsluhu uviaznutí.

Vyššie uvedené činnosti budú popísane podrobnejšie v príslušných kapitolách.

3.1.2 Správa operačnej pamäte

Operačná pamäť (OP) je veľké pole adresovateľných slov alebo bajtov. Slúži pre uloženie a rýchle sprístupnenie dát, ktoré zdieľajú procesor a V/V zariadenia. Procesor (Central Processing Unit) číta inštrukcie z operačnej pamäte počas cyklu zavedenia inštrukcie a počas cyklu zavedenia dát buď číta, alebo zapisuje dáta do pamäte. Operačná pamäť je vlastne jediné pamäťové zariadenie, ktoré procesor môže priamo adresovať. Napr. ak CPU potrebuje spracovávať dáta z disku, tieto dáta musia byť najskôr prenesené do pamäte; inštrukcie ktoré CPU vykonáva, tiež musia byť v pamäti.

Program, ktorý sa má vykonať musí byť zavedený do pamäte a jeho adresy musia byť mapované do fyzických adries. Pri vykonávaní programu a sprístupnení jeho dát v pamäti CPU

generuje práve tieto adresy. Po ukončení programu sa jeho pamäťový priestor uvoľní a môže sa tam zaviesť ďalší program pre vykonanie.

Pre zvýšenie efektívnosti využitia CPU a rýchlosti odozvy počítačového systému je potrebné umiestniť do pamäte niekoľko programov. Pre zvládnutie tejto úlohy existuje viacero techník, ktoré sú postavené na rôznych prístupoch k správe pamäte a závisia na konkrétnej situácii. Výber správy pamäte pre určitý systém závisí od mnohých faktorov, ale najviac od hardvérovej platformy systému.

OS zaisťuje nasledovné činnosti v súvislosti so správou pamäte:

- uchováva informácie o tom, ktoré časti OP sú pridelené a komu,
- rozhoduje, ktoré procesy majú byť zavedené do OP, keď sa táto uvoľní,
- prideľuje a ruší pridelenie pamäťového priestoru podľa potrieb procesov.

Vyššie uvedené činnosti budú popísané podrobnejšie v kapitole o správe pamäte.

3.1.3 Správa diskovej pamäte

Hlavnou úlohou počítačového systému je vykonávať programy. Tieto programy a ich dáta musia byť v operačnej pamäti počas vykonania programu. Pretože táto pamäť obyčajne nestačí pre uloženie všetkých programov a dát a pretože v nej sa tieto nemôžu uložiť trvalo, systém musí poskytovať pre tieto účely sekundárnu periférnu pamäť. Väčšina moderných počítačových systémov používa disky ako on-line periférnu pamäť, kde sa ukladajú programy a dáta. Mnoho programov, vrátane kompilátorov, asemblerov, editorov atď., sú uložené na disku kým nie sú zavedené do operačnej pamäte a potom využívajú disk ako zdrojové aj cieľové zariadenie. Z toho vyplýva, že disková pamäť je často používaná a preto musí byť využívaná efektívne.

OS zaisťuje nasledujúce činnosti v súvislosti so správou diskovej pamäte:

- správu voľného diskového priestoru,
- prideľovanie diskového priestoru,
- plánovanie diskových operácií.

Techniky správy diskovej pamäte budú podrobne rozobrané v kapitole o správe diskových zariadení.

3.1.4 Správa V/V systému

Jedna z úloh OS je schovať pred používateľom špecifické rysy hardvérových zariadení a poskytnúť k nim jednotný prístup.

V/V systém pozostáva zo:

- systému bufrovania a „cache“-ovania,
- štandardného interfejsu k ovládačom zariadení,
- driverov pre jednotlivé hardvérové zariadenia.

Len driver pozná zvláštnosti a príkazy V/V zariadenia, ktoré obsluhuje. Viac podrobností sa čitateľ dozvie v kapitole o správe periférnych zariadení.

3.1.5 Správa súborov

Správa súborov je najviditeľnejšia časť z OS. Počítače môžu ukladať informáciu na niekoľkých rôznych typoch fyzických médií. Najčastejšie používané sú: magnetická páska, magnetický disk, optický disk a pružný disk. Každé z týchto médií má svoje charakteristiky a fyzickú organizáciu. Každé médium je riadené zariadením, ako sú diskové alebo páskové zariadenia, ktoré majú svoje charakteristiky ako sú: rýchlosť, kapacita, metóda prístupu a iné.

Pre pohodlie používateľa OS poskytuje jednotný logický pohľad na periférne pamäťové zariadenia. Pracuje s logickou jednotkou súbor. OS mapuje súbory na fyzické médiá a riadi prístup k nim.

Súbor je množina príbuzných informácií, ktoré sú definované tvorcom súboru. Obyčajne súbory obsahujú programy (v zdrojovom alebo v binárnom tvare) a dáta. Súbory môžu mať voľný formát, ako sú napr. textové súbory alebo môžu byť formátované presne. Súbor pozostáva z postupnosti bajtov, riadkov alebo záznamov, ktorých význam definuje tvorca súboru.

Operačný systém implementuje abstraktný koncept súborov riadením periférnych pamäťových médií (pásky, disky, flopy disky a iné) a ich zariadení.

Súbory sú obyčajne organizované v adresároch pre uľahčenie prístupu k nim. Vo viacužívateľských systémoch je potrebné zaistiť aj kontrolu prístupu k súborom jednotlivých používateľov.

OS je zodpovedný za nasledujúce činnosti v súvislosti so správou súborov:

tvorba a mazanie súborov,

- tvorba a mazanie adresárov,
- dodáva primitíva (základné operácie) pre manipuláciu so súbormi a adresármi,
- mapuje súbory na periférne pamäte,
- zaisťuje zálohovanie súborov na spoľahlivých zariadeniach.

Techniky správy súborov sú uvedené v kapitole o správe súborov.

3.1.6 Systém ochrany

Ak systém je viacužívateľský a dovoľuje paralelnú činnosť viacerým procesom, je potrebné, aby poskytol mechanizmy ochrany procesu pred aktivitami ostatných procesov. To znamená, že OS musí zaistiť, že súbory, pamäťové segmenty, procesor a iné prostriedky systému budú využívané len tými procesmi, ktoré majú na to oprávnenie. Mechanizmus ochrany musí poskytovať prostriedky pre špecifikáciu kontroly a tiež prostriedky na kontrolu a donútenie k ich dodržiavaniu.

3.1.7 Sieťová podpora

Distribuované systémy pozostávajú z množiny procesorov, ktoré nezdieľajú pamäť a hodiny. Každý procesor má svoju vlastnú pamäť a komunikuje s ostatnými procesormi distribuovaného systému pomocou komunikačných liniek, ako sú výsokorýchlostné zbernice alebo telefónne linky. Procesory v distribuovanom systéme sa líšia rozmermi a funkciami. V takomto systéme môžu byť zahrnuté osobné počítače, pracovné stanice, minipočítače a clustre počítačov.

Procesory v distribuovanom systéme sú prepojené komunikačnou sieťou, ktorá môže byť konfigurovaná mnohými spôsobmi. Sieť môže byť úplne prepojená, t.j. každý uzol je spojený s každým, alebo čiastočne prepojená, kedy nejestvuje prepojenie každého uzla s každým. Návrh komunikačnej siete musí rátať aj so stratégiami smerovania balíkov dát a problémami bezpečnosti.

Distribuovaný systém spája fyzicky oddelené, mnohokrát heterogénne systémy do jedného koherentného systému tak, aby poskytol používateľovi prístup k rôznym prostriedkom ktorými disponuje. Prístup k zdieľaným prostriedkom umožňuje zrýchliť výpočty, zväčšiť spoľahlivosť a prístupnosť dát. OS obyčajne zobecňuje sieťový prístup vo forme prístupu k súborom, pričom detaily o umiestnení a prenose súborov zostávajú transparentné pre používateľa.

3.1.8 Interpreter príkazového jazyka

Jednou z najdôležitejších častí OS, je interpreter príkazového jazyka, ktorý poskytuje interfejs medzi používateľom a OS. Niekedy interpreter príkazového jazyka je súčasťou jadra, inokedy (MS DOS) je to samostatný program.

Vo viacužívateľských systémoch po prihlásení sa používateľa, sa spustí program, ktorý číta a interpretuje riadiace príkazy automaticky. Taký program sa nazýva *interpreter príkazového riadku*, alebo interpreter riadiaceho štítku v spracovaní dávkových úloh, alebo *shell* v Unixe atď. Funkcia príkazových interpreterov je veľmi jednoduchá: prečítať príkaz a vykonať ho.

Moderné operačné systémy (MS Windows, MacIntosh, Unix a iné) často ponúkajú grafické interpretery, ktoré sú jednoduché a prístupné širokej vrstve používateľov. Práca pomocou takého interpretera nevyžaduje znalosť jednotlivých príkazov a ich parametrov. Používateľ pomocou myši ukazuje na ikony, ktoré prezentujú programy, súbory alebo systémové funkcie. Podľa umiestnenia kurzora a kliknutím na tlačidlo myši sa dá vyvolať program, vybrať súbor alebo adresár a operáciu nad ním.

Ďalšie interpretery sú určené náročnejším používateľom, pretože tam sa príkazy zadávajú z klávesnice. Tu má používateľ možnosť modifikovať príkaz podľa svojich potrieb, ale musí poznať príkazy a možnosti ich využitia. Typický predstaviteľ tohto typu interpreterov je shell v Unixe.

Príkazový jazyk obsahuje príkazy pre tvorbu a rušenie procesov, tvorbu a mazanie súborov, tvorbu a mazanie adresárov, pre manipuláciu s V/V zariadeniami, OP, sieťovými podpornými programami, ochranou a iné.

3.2 Systémové služby

Systémové služby poskytujú určité služby programom a používateľom týchto programov. Majú za úlohu uľahčiť programovanie. Líšia sa od systému k systému, ale dajú sa klasifikovať niektoré bežné triedy služieb:

- ***Služby pre vykonanie programov*** - umožňujú zaviesť program do pamäte a spustiť ho a tiež dávajú možnosť programu ukončiť sa - normálne alebo s chybou.
- ***Služby pre V/V operácie*** - poskytujú prístup k V/V zariadeniam, pričom tieto sú chránené pred neoprávneným a neadekvátnym prístupom.
- Služby pre narábanie so súbormi - služby pre tvorbu, rušenie, zápis a modifikáciu súborov.
- ***Služby pre komunikáciu*** - poskytujú prostriedky pre komunikáciu medzi procesmi. Týkajú sa komunikácie medzi procesmi, ktoré sa vykonávajú na jednom počítači a medzi procesmi, ktoré sa vykonávajú na rôznych počítačoch, ktoré sú spojené v počítačovej sieti.
- ***Služby pre odhalenie chýb*** - OS musí mať stále prehľad o možných chybách. Chyby sa môžu vyskytnúť v CPU, OP, V/V zariadeniach alebo v užívateľských programoch. OS musí poskytnúť prostriedok pre ošetrenie každej chyby.

V OS existuje aj ďalšia skupina systémových služieb, ktoré sa poskytujú systému pre zefektívnenie jeho činnosti. Sú to:

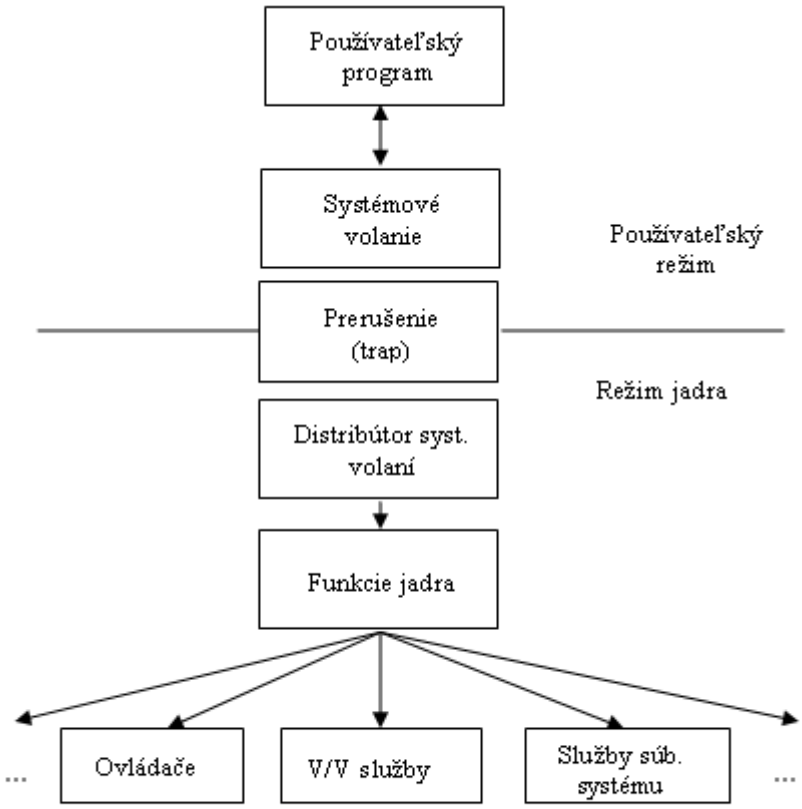
- ***Služby pre prideľovanie prostriedkov*** - OS spravuje mnoho prostriedkov rôznych typov, ako sú: operačná pamäť, procesor, súbory, periférne zariadenia. Napr. pre efektívne využitie CPU, OS poskytuje plánovacie rutiny, ktoré plánujú poradie vykonania procesov, pričom berú do úvahy počet úloh, rýchlosť CPU, počet registrov a iné faktory. Ďalej medzi takéto služby patrí prideľovanie páskového alebo iného zariadenia určitej úlohe, pričom sa urobia záznamy o jeho pridelení do interných tabuliek. Ďalšia služba sa využíva, keď sa tieto záznamy musia vymazať.
- ***Služby pre účtovanie použitých prostriedkov*** - používajú sa na sledovanie počtu a množstva prostriedkov, ktoré užívatelia využívajú. Tieto záznamy sa môžu použiť pre účtovanie alebo pre štatistiku. Štatistika sa môže využiť buď pri rekonfigurácii systému pre zlepšenie jeho výkonu, alebo pre výskumné úlohy.
- ***Služby pre ochranu systému.*** Vlastníci informácie, ktorá je uložená na viacužívateľskom systéme, chcú riadiť jej využitie. To znamená, že ak sa vykonáva viac nezávislých procesov paralelne, nesmú sa navzájom ovplyvňovať. Dôležitá je aj ochrana proti neoprávnenému prístupu k informácii. Služby pre ochranu systému sa využívajú práve v týchto prípadoch.

3.3 Systémové volania

Systémové volania poskytujú interfejs medzi procesom a operačným systémom a umožňujú procesu využívať služby operačného systému pomocou presne definovaného interfejsu, t.j. vstupných a

výstupných parametrov volania. Systémové volania často využívajú špeciálne inštrukcie, ktoré spôsobujú zmenu režimu vykonania z používateľského režimu do chráneného režimu jadra (Obr. 3.1).

Systémové volania sú obyčajne prístupné v asembleri. Niektoré systémy umožňujú systémové volania aj z vyšších programovacích jazykov, kedy sa tieto volania podobajú volaniu preddefinovaných funkcií alebo volaniu podprogramov. V tomto prípade volanie sa uskutočňuje volaním špeciálnej run-time rutiny, ktorá volá systém alebo sa systémové volanie generuje priamo. Niektoré programovacie jazyky - C, C++ a Perl boli navrhnuté ako náhrada asembleru pre systémové programovanie. Tieto jazyky dovoľujú priame volanie systému. Napr. systémové volania v Linuxe sa môžu použiť priamo v programoch, napísaných v C alebo C++.



Obr. 3.1 Princíp systémového volania

Na ilustráciu použitia systémových volaní rozoberieme prípad jednoduchého programu, ktorý číta dáta z jedného súboru a zapisuje ich do iného súboru. Prvý vstup, ktorý tento program bude vyžadovať, budú mená súborov. Ak tieto mená nie sú zadané v príkazovom riadku, potom pre ich interaktívny vstup bude potrebný celý rad systémových volaní najskôr pre výpis navádzacieho textu na obrazovku, potom pre načítanie znakov z klávesnice. V dávkovom režime bude potrebné volať systém pre odovzdanie parametrov (mená súborov) z riadiacich štítkov do programu. Pokiaľ systém pracuje s myšou, oknami a ikonami, bude potrebné otvoriť okno a ponúknuť menu s menami súborov. Používateľ si vyberie zdrojový súbor, potom ďalšia postupnosť systémových volaní otvorí nové dialógové okno, kde sa zapíše meno cieľového súboru.

Po získaní mien súborov musí program otvoriť zdrojový súbor a vytvoriť cieľový súbor pomocou systémových volaní (open, create). Pri týchto operáciách je možné, že vzniknú chyby. Ak napr. zdrojový súbor je chránený proti čítaniu, vzniká potreba vypísať chybovú správu na obrazovku (postupnosť systémových volaní) a ukončiť program s chybou - ďalšie systémové volania. Podobné chyby môžu vzniknúť aj pri vytváraní cieľového súboru - napr. ak súbor s takýmto menom už existuje.

V ďalšej fáze program začne čítať zo vstupného súboru (read) a zapisovať do cieľového (write). Každé čítanie a zápis musí vrátiť kód ukončenia operácie a v prípade chyby ju obslúžiť (ďalšie systémové volania).

Nakoniec, po skopírovaní súboru musí program uzatvoriť obidva súbory (close) a ukončiť program normálne.

Ako je z toho príkladu vidieť, programy využívajú veľmi často systémové volania, ale väčšina používateľov nikdy nevidí tieto detaily. Run-time podpora vo väčšine programovacích jazykov poskytuje jednoduchý interfejs.

Systémové volania sa v rôznych systémoch uskutočňujú rôzne. Často sú pre volanie potrebné ďalšie údaje okrem názvu volanej systémovej rutiny. Pre odovzdanie parametrov operačnému systému sa používajú tri základné metódy. Najjednoduchší spôsob je odovzdať parametre cez registre. Niekedy je ale viac parametrov ako registrov. V takomto prípade sa parametre uložia do bloku (alebo tabuľky) v pamäti a jeho adresa sa odovzdá operačnému systému cez register. Parametre sa môžu odovzdať aj pomocou zásobníka (stack), kde program vloží parametre a operačný systém ich vyberie. Niektoré operačné systémy preferujú odovzdanie pomocou zásobníka alebo bloku, pretože tieto spôsoby neobmedzujú počet a dĺžku odovzdávaných parametrov.

Systémové volania môžeme rozdeliť na 5 hlavných kategórií: riadenie procesov a dávok, práca so súbormi, práca so zariadeniami, správa informácií a komunikácie.

3.3.1 Riadenie procesov a dávok

Do skupiny riadenia procesov patria nasledujúce systémové volania:

- *end, abort* - dáva možnosť procesu ukončiť svoje vykonanie normálne alebo násilne,
- *load, execute* - zavedenie procesu do pamäte a spustenie,
- *create process, terminate process* - tvorba a ukončenie procesu,
- *get process attributes, set process attributes* - dáva možnosť získať atribúty procesu alebo ich nastaviť,
- *wait* - umožňuje procesu čakať určitú dobu,
- *wait event, signal event* - umožňuje procesu čakať na určitú udalosť a signalizovať udalosť,
- *allocate and free memory* - volania pre získanie a uvoľnenie pamäte.

3.3.2 Práca so súbormi

Do skupiny systémových volaní pre riadenie procesov patria nasledujúce volania:

- *create file, delete file - volania pre tvorbu a zrušenie súboru,*
- *open, close* - otvorenie a uzatvorenie súboru,
- *read, write, reposition* - čítanie, zápis a prestavenie pozície ukazovateľa pre nasledovnú operáciu,
- *get file attributes, set file attributes* - získanie a nastavenie atribútov súboru: meno, typ, prístupové práva a ďalšie.

3.3.3 Práca so zariadeniami

Počas svojho behu program často potrebuje dodatočné prostriedky systému - pamäť, pásky, súbory atď. Vo viacužívateľských systémoch je potrebné požiadať o pridelenie zariadenia pred jeho použitím.

Do skupiny systémových volaní pre prácu so zariadeniami patria nasledujúce volania:

- *request device, release device* - žiadosť o pridelenie zariadenia a jeho uvoľnenie,
- *read, write, reposition* - čítanie, zápis a prestavenie pozície,
- *get device attributes, set device attributes* - získanie a nastavenie atribútov zariadenia,
- *logically attach or detach device* - logické pripojenie a odpojenie zariadenia.

3.3.4 Správa informácií

Mnoho systémových volaní existuje jednoducho preto, aby sprostredkovali presun informácií medzi procesom a operačným systémom. K nim patria:

- *get time or date, set time or date* - získanie a nastavenie dátumu a času,
- *get system data, set system data* - získanie a nastavenie systémových dát, ako sú verzia systému, počet pripojených používateľov, veľkosť voľnej pamäte alebo diskového priestoru atď.,
- *get process, file or device attributes* - získanie atribútov procesu, súboru alebo zariadenia,
- *set process, file or device attributes* - nastavenie atribútov procesu, súboru alebo zariadenia.

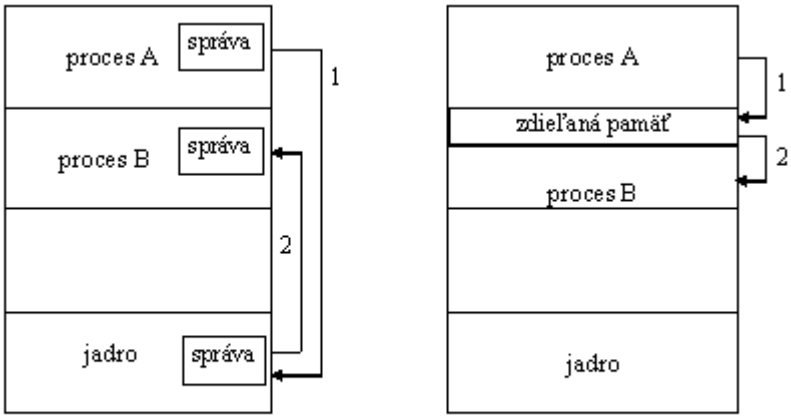
3.3.5 Komunikácie

Komunikačný model výmeny informácií medzi procesmi môže byť uskutočnený na základe komunikácie medzi procesmi, ktorú poskytuje operačný systém. Najčastejšie je to výmena správ. Pred nadviazaním komunikácie je potrebné otvoriť spojenie. Za týmto účelom je treba poznať meno ďalšieho účastníka spojenia (identifikačné číslo procesu) a jeho lokalizáciu (uzol v sieti) .

Ďalší model komunikácie je založený na zdieľanej pamäti. Systémové volania dávajú možnosť procesu požiadať o prístup k pamäťovej oblasti, ktorú vlastní iný proces. Výmena informácií prebieha zápisom a čítaním do tejto zdieľanej oblasti. Táto forma komunikácie je rýchla, ale nehodí sa pre komunikáciu v sieti. Tieto dva komunikačné modely (Obr.3.2) sú implementované vo väčšine operačných systémov.

Systémové volania pre komunikáciu zahŕňajú nasledujúce možnosti:

- *create, delete communication connection* - vytvoriť a zrušiť spojenie,
- *send, receive messages* - zaslať a prijať správu,
- *transfer status information* - prenos informácie o stave spojenia,
- *attach , detach remote device* - pripojenie a odpojenie vzdialeného zariadenia.



Obr. 3.2 Komunikácia medzi procesmi pomocou správ a zdieľanej pamäte

3.4 Systémové programy

Systémové programy poskytujú lepšie prostredie pre vývoj a vykonávanie programov. Niektoré sú jednoduchým interfejsom ku systémovým volaniam, iné sú oveľa komplikovanejšie.

Systémové programy poskytujú prostriedky pre :

- ***Manipuláciu so súbormi a adresármi.***
- ***Získanie stavových informácií*** - sú to informácie o aktuálnom čase, dátume, voľnej pamäti, počte používateľov atď.

- ***Modifikáciu súborov*** - každý operačný systém musí poskytovať editory pre tvorbu a modifikáciu súborov.
- ***Podporu programovacích jazykov*** - kompilátory, asemblery, interpretery bežných programovacích jazykov (C, FORTRAN, COBOL, Pascal, BASIC, LISP). Mnoho z týchto programov je v súčasnosti poskytovaných aj samostatne.
- ***Pre zavedenie a spustenie programov (loader)*** - po preložení programu, sa ten musí zaviesť do pamäte a spustiť. Systém musí poskytovať absolútny, overlay-ový alebo relokovací zavádzací program, linkovací editor a ladiaci program pre vyššie programovacie jazyky.
- ***Komunikácie*** - sú to programy, ktoré dávajú možnosť vytvoriť virtuálne spojenie medzi procesmi, používateľmi a jednotlivými počítačmi v sieti. Patria medzi ne e-mail, ftp, rlogin a iné.
- ***Aplikačné programy*** - väčšina OS poskytuje programy pre riešenie bežných problémov - formátovanie textov, programové balíky pre prácu s grafickými zariadeniami, pre databázové systémy, tabuľkové procesory, hry a mnoho iných aplikácií.
- ***Príkazový interpreter*** - je to najdôležitejší systémový program. Jeho hlavná úloha je prijať a vykonať ďalší príkaz používateľa.

Existujú **dva spôsoby implementácie príkazov**. Prvý je taký, že **príkazový interpreter obsahuje kód na vykonávanie príkazov**. To znamená, že čím väčší počet príkazov obsahuje príkazový jazyk, tým väčší je príkazový interpreter, ale táto implementácia je rýchla.

Druhý spôsob implementácie príkazového interpretera je, **že príkazy sú súbory, ktoré sa zavedú do OP a vykonajú sa ako proces**. Tento prístup je využitý v OS Unix. Táto implementácia je pomalšia oproti predchádzajúcej, pretože zavedenie kódu do pamäte a jeho odštartovanie je zdĺhavejšie ako jednoduchý skok do inej sekcie kódu.

Často pohľad bežného používateľa na určitý operačný systém je podmienený práve poskytovanými systémovými programami a neodzrkadľuje efektívnosť systémových volaní, na ktorých sú tieto postavené. Dôležitú úlohu tu zohráva aj interfejs príkazového interpretera - zadávanie príkazov z klávesnice alebo pomocou ikôn, ale návrh interfejsu nie je priamou funkciou operačného systému.

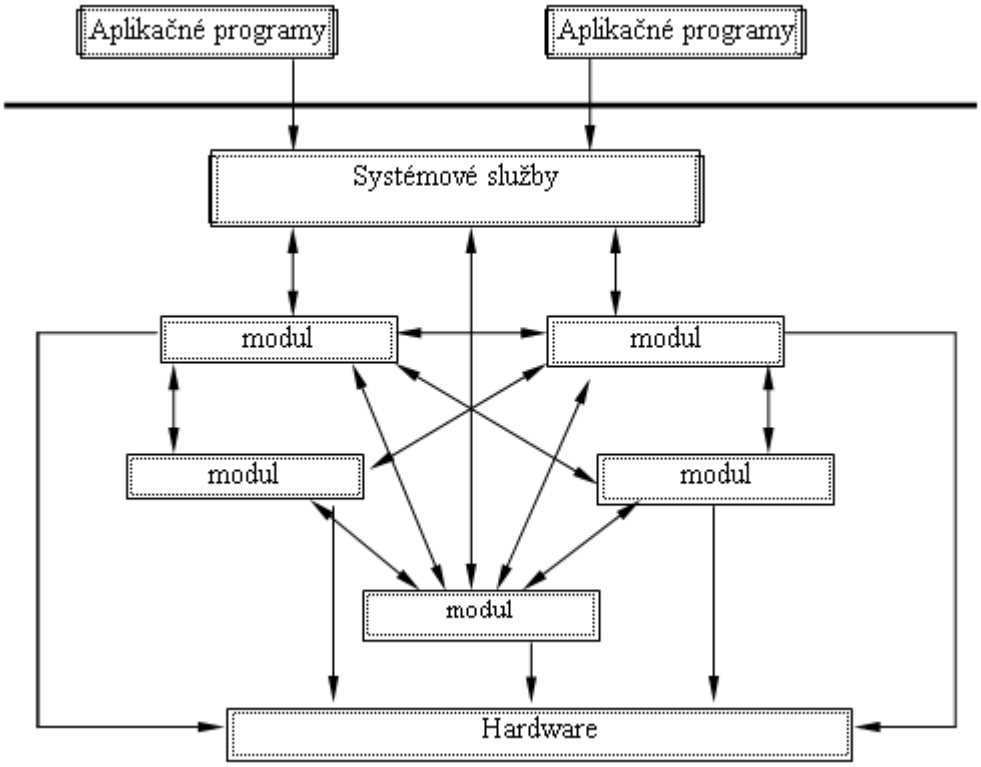
Operačný systém nerozlišuje systémové programy od používateľských pri ich vykonávaní.

3.5 Štruktúra OS

Moderné operačné systémy sú veľké a zložité a musia byť navrhnuté tak, aby fungovali správne a aby sa dali ľahko modifikovať. Bežný prístup je taký, že sa systém rozdelí na menšie moduly, ktoré majú presne stanovené funkcie, vstupy a výstupy. Ďalej uvedieme niektoré spôsoby prepojenia komponentov systému.

3.5.1 Monolitická štruktúra

V systéme s monolitickou architektúrou sú aplikačné programy oddelené od systému. To znamená, že kód operačného systému beží v privilegovanom režime procesora a má prístup k systémovým dátam a hardvéru (HW). Aplikácie bežia v neprivilegovanom režime s obmedzeným prístupom k systémovým dátam. Keď program v užívateľskom režime volá systémovú službu, procesor zachytí volanie a potom prepne volajúci proces do privilegovaného režimu. Keď sa systémová služba ukončí, OS opäť prepne proces do používateľského režimu. Štruktúra monolitického systému je ukázaná na Obr. 3.3. Takáto štruktúra bola typická pre prvé operačné systémy.

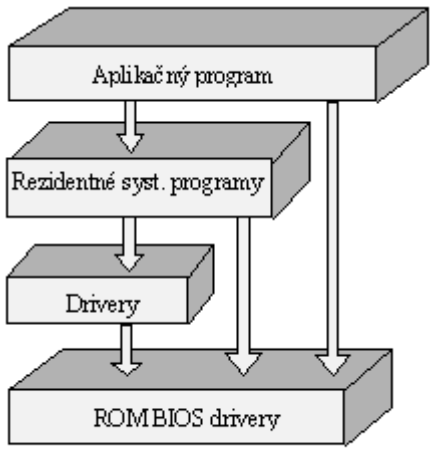


Obr. 3.3 Monolitický operačný systém

3.5.2 Jednoduchá štruktúra

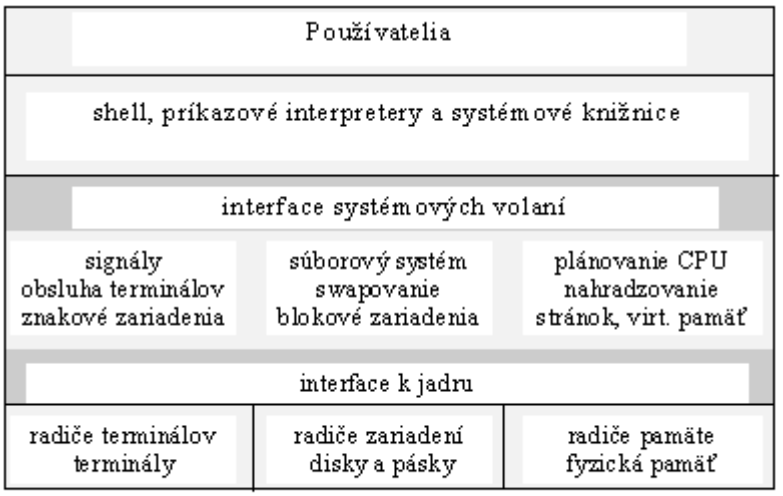
Existuje mnoho komerčných operačných systémov, ktoré majú jednoduchú štruktúru. Obyčajne sa jedná o malé systémy. Typickým príkladom je MS-DOS, ktorý bol najlepšie predávaným operačným systémom na prelome 90-tych rokov. Tento systém nebol zamýšľaný pre také masové rozšírenie, a preto nebola venovaná dostatočne veľká pozornosť starostlivému návrhu jednotlivých modulov. Hardvér, pre ktorý bol systém určený mal tiež obmedzené možnosti.

MS-DOS má určitú štruktúru, ale jeho interfejsy nie sú dobre oddelené (Obr.3.4). Napr. aplikačný program má prístup k základným V/V rutinám a môže zapisovať priamo na disk alebo na obrazovku. Táto voľnosť robí systém zraniteľným, pretože pri chybách je možný pád systému alebo strata dát na disku. Samozrejme MS-DOS je obmedzený aj kvôli možnostiam cieľového HW (pôvodne pre Intel 8088), ktorý neposkytuje HW ochranu a privilegované inštrukcie a tvorcovia nemohli inak vyriešiť tieto problémy.



Obr. 3.4 Štruktúra systému MS- DOS

Ďalší príklad obmedzenej štruktúry je pôvodný UNIX, ktorý na začiatku tiež rátal s obmedzenými HW možnosťami. Pozostáva z dvoch oddelených častí - jadra a systémových programov (Obr.3.5). Všetky moduly pod vrstvou interfejsu patria do jadra. Jadro obsluhuje súborový systém, plánovanie procesora, správu pamäte a ďalšie služby cez systémové volania. To znamená, že veľké množstvo funkcií systému je zahrnutých do jednej vrstvy.

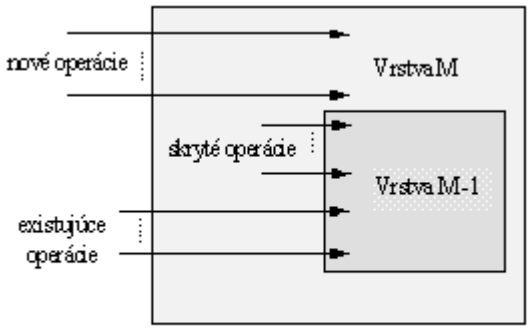


Obr. 3.5 Štruktúra Unix-u

Systémové volania definujú programový interfejs k Unix-u. Množina bežne dostupných systémových programov definuje používateľský interfejs. Programátorský a používateľský interfejs definujú kontext, ktorý je podporovaný jadrom. V súčasnosti je niekoľko verzií Unix-u, kde jadro je rozdelené na ďalšie časti podľa funkcií. AIX (IBM verzia Unix-u) delí jadro na dve časti. Mach (Carnegie Melon University) zredukovali jadro tak, že presunuli všetky menej dôležité časti do vyšších vrstiev, aj do používateľskej. Tak vzniklo tzv. mikrojadro (microkernel).

3.5.3 Viacvrstvová architektúra

Hlavná výhoda viacvrstvovej architektúry systému je jeho modularita. Vrstvy sú rozdelené tak, že každá využíva funkcie (operácie) nižších vrstiev. Vrstva je implementácia abstraktného objektu, čo je vlastne zapuzdrenie dát a operácií s týmito dátami. Typická vrstva operačného systému, napr. vrstva M je ukázaná na Obr.3.6.



Obr. 3.6 Vrstvy operačného systému

Pozostáva z dátových štruktúr a množiny rutín, ktoré môžu byť volané z vyšších vrstiev. Vrstva M na druhej strane môže volať operácie nižších vrstiev. Takýto prístup zjednodušuje ladenie a verifikáciu systému. Prvá vrstva môže byť odladená bez vplyvu zvyšku systému, pretože tá využíva len základný HW. Po odladení prvej vrstvy sa môže ladiť druhá, pričom sa stavia na korektnej funkcii prvej a tak ďalej. Z toho je vidieť, že návrh a implementácia systému sa pri použití viacvrstvovej architektúry značne zjednodušuje.

Každá vrstva je implementovaná pomocou služieb, ktoré poskytujú nižšie vrstvy, pričom nie je potrebné vedieť, ako sú tieto implementované, stačí vedieť, aké operácie poskytujú.

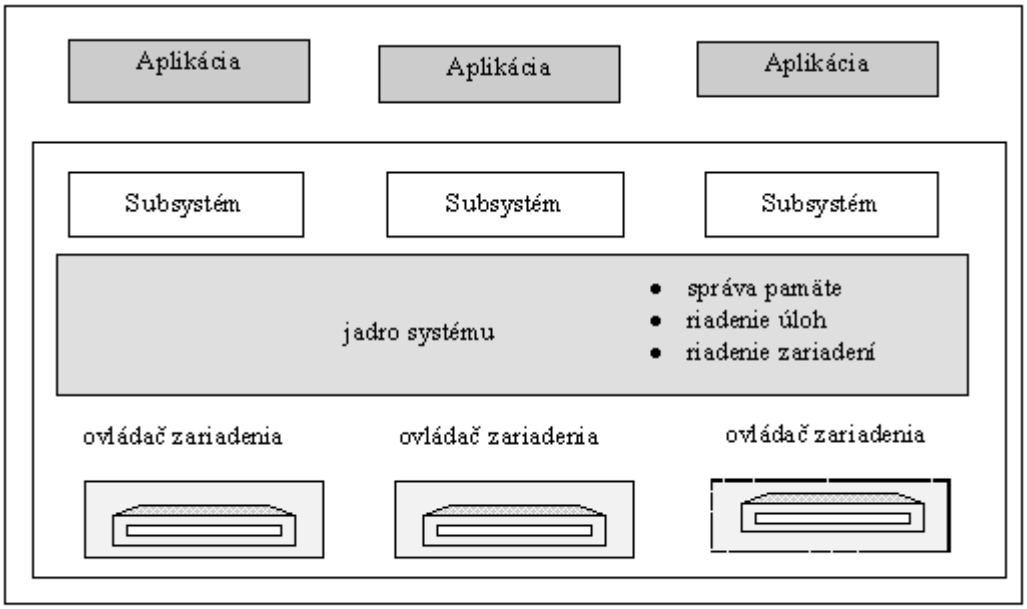
Vrstvová architektúra bola prvýkrát navrhnutá a implementovaná do systému THE - univerzitného operačného systému z Technische Hogeschool Eindhoven - Holandsko. Jeho štruktúra je ukázaná na Obr.3.7.

vrstva 5	používateľské programy
vrstva 4	bufrovanie pre vstup a výstup
vrstva 3	driver operátorskej konzoly
vrstva 2	správa pamäte
vrstva 1	plánovanie procesov
vrstva 0	hardware

Obr. 3.7 Vrstvová štruktúra systému THE

Jeden z hlavných nedostatkov implementácie viacvrstvovej architektúry oproti iným typom je menšia efektívnosť. Napr. ak používateľský program chce vykonať V/V operáciu, musí použiť systémové volanie, ktoré spôsobí programové prerušenie do V/V vrstvy, ktorá volá vrstvu správy pamäte, prejde cez plánovanie procesov a nakoniec sa dostane k HW. V každej vrstve sa parametre musia modifikovať, pravdepodobne sa budú prenášať aj nejaké dáta atď. To znamená, že každá vrstva zanáša dodatočnú réžiu a konečný výsledok je taký, že čas ktorý zaberie takáto operácia, je dlhší ako u systému bez vrstiev.

Tieto obmedzenia v poslednom čase zapríčinili to, že návrhy OS sa uberajú smerom k menšiemu počtu vrstiev. Navrhuje sa menej vrstiev s väčším počtom funkcií, pričom sa využívajú výhody modulárneho programovania a zároveň sa obchádzajú problémy s návrhom veľkého počtu vrstiev a interakcií medzi nimi. Čitateľ si môže všimnúť štruktúru OS/2 a porovnať ju s MS-DOS. OS/2 je priamym následníkom MS-DOS-u, ale pri jeho návrhu sa zobrali do úvahy nedostatky MS-DOS-u. Výsledná architektúra je uvedená na Obr. 3.8.



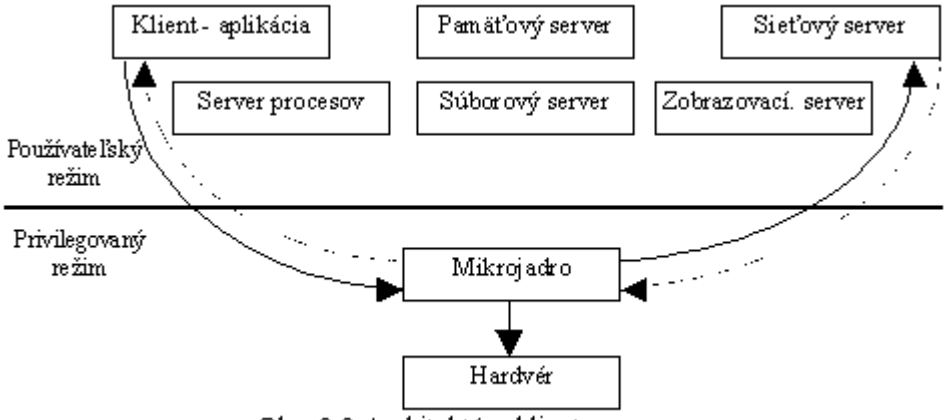
Obr. 3.8 Štruktúra systému OS/2

Novšie verzie Unix-u sú navrhnuté pre progresívnejší HW. Pri tejto HW podpore bolo možné rozdeliť jadro na menšie časti, ako bolo možné u MS-DOS alebo u pôvodnej verzie Unix-u.

3.5.4 Architektúra klient - server

Základná myšlienka tejto architektúry spočíva v rozdelení operačného systému do niekoľkých procesov (serverov), z ktorých každý realizuje jednu sadu služieb - napr. služby pre prácu s pamäťou, služby pre vytváranie alebo plánovanie procesov a iné. Každý server beží v používateľskom režime a čaká v nekonečnej slučke na požiadavky klientov. Klient, ktorý môže byť buď iný operačný systém alebo aplikačný program, žiada o službu tak, že pošle serveru správu. Jadro, ktoré beží v privilegovanom režime doručí správu serveru, server vykoná požadovanú operáciu a jadro vráti výsledok klientovi v inej správe. Tieto operácie sú znázornené na Obr. 3.9.

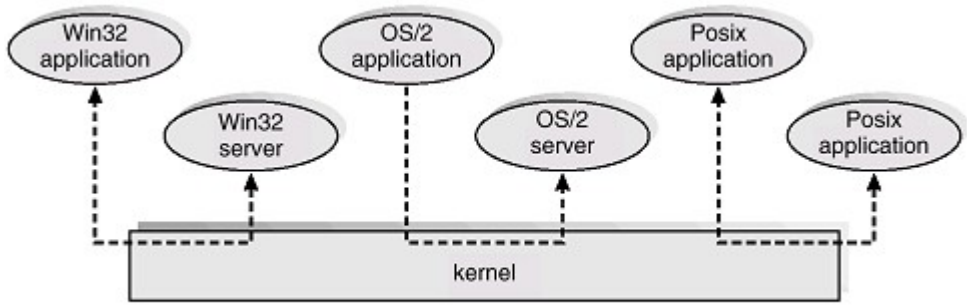
V architektúre klient-server sa využíva mikrojadro, ktoré musí zabezpečiť minimálne tri funkcie pre chod systému: spúšťanie a vykonávanie procesov, správu pamäte a doručovanie správ. Ostatné funkcie operačného systému vykonávajú servery v používateľskom režime. Moderné operačné systémy, ako napr. Mach, OSF, alebo NT (Obr. 3.10) majú architektúru, založenú na mikrojadre. Táto architektúra môže byť využívaná ako v centralizovanom systéme, tak aj v distribuovanom. V druhom prípade systém musí poskytovať aj sieťovú podporu.



Obr. 3.9 Architektúra klient-server

Architektúra *klient-server* má tieto výhody:

- **zvyšuje spoľahlivosť systému**. Každý server beží ako oddelený proces v svojom vlastnom pamäťovom segmente, a preto je chránený pred ostatnými procesmi. Navyše, pretože servery bežia v používateľskom režime, nemôžu zasahovať priamo do HW, ani modifikovať pamäť, v ktorej je uložený riadiaci proces.
- **vedie k použitiu modelu distribuovaných výpočtov**. Pretože počítače v sieti pracujú na základe modelu klient-server a využívajú pre komunikáciu správy, miestne servery môžu ľahko zasielať správy vzdialeným počítačom v záujme klientských aplikácií. Klient nepotrebuje vedieť, či sú jeho požiadavky obsluhované lokálne alebo na vzdialenom počítači.



Obr. 3.10 Architektúra klient-server operačného systému Windows NT

3.5.5 Objektový prístup

Pomerne nový prístup k programovaniu - objektovo orientované programovanie - ovplyvnil aj tvorbu operačných systémov. Niektoré operačné systémy sú celé postavené na objektoch (NextStep), iné ich využívajú v kombinácií s inými modelmi (Windows NT).

Systémové prostriedky, ktoré sú zdieľané viacerými procesmi sú realizované ako objekty a manipuluje sa s nimi pomocou služieb objektu. Tento prístup znižuje dopad zmien, ktoré sa so systémom robia. Ak sa napríklad zmení hardvér, vynútia sa zmeny len tých objektov, ktoré reprezentujú hardvérové zdroje a ich služby. Kód, ktorý takýto objekt len používa, zostane bezo zmien. Ďalšie výhody operačného systému využívajúceho objektový prístup sú:

- Operačný systém manipuluje so zdrojmi jednotným spôsobom.
- Bezpečnosť je poskytnutá jednoduchším spôsobom, pretože všetky objekty sú chránené rovnakým štýlom. Keď sa proces snaží pristúpiť k danému objektu, bezpečnostný systém skontroluje operáciu.
- Objekty poskytujú vhodný a jednotný vzor pre zdieľanie objektov medzi dvomi a viacerými procesmi.

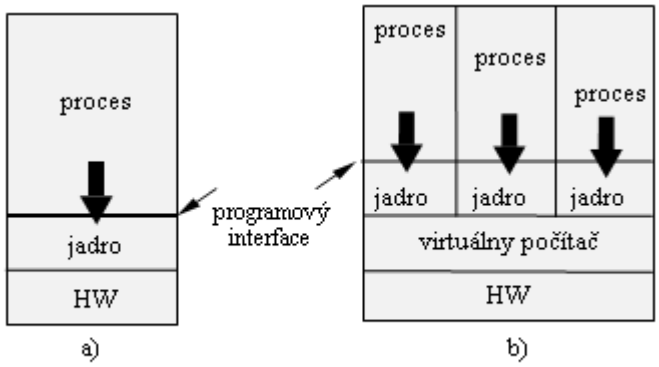
3.6 Virtuálny počítač

Koncepcia počítačového systému je založená na vrstvách. HW je najnižšia vrstva. Jadro bežiace na ďalšej úrovni využíva inštrukcie základného inštrukčného súboru pre tvorbu systémových volaní, ktoré využívajú ďalšie vrstvy. Systémové programy nad jadrom môžu využívať buď systémové volania alebo hardvérové inštrukcie, pomocou ktorých sa budujú ďalšie zložitejšie programy s rozšírenými funkciami.

Niektoré systémy idú ešte ďalej a dovoľujú aplikáciám volať veľmi jednoducho systémové programy. Takto aplikačné programy „vidia“ všetko pod vrstvou systémových programov. Tento pohľad na výpočtový systém vedie ku konceptu virtuálneho počítača. Operačný systém VM od firmy IBM je najlepším príkladom konceptu virtuálneho počítača. Používateľovi je poskytnutý jeho vlastný virtuálny počítač. Na tomto počítači sa môže spustiť ľubovoľný operačný systém alebo SW balík, ktorý beží na nižšej úrovni. V prípade IBM VM používateľ prevádzkuje CMS, čo je interaktívny jednoužívateľský systém. Softvér virtuálneho počítača beží v režime multiprogramovania viacerých virtuálnych systémov na fyzickom počítači.

Koncepcia virtuálnych počítačov je užitočná pri vývoji a ladení operačných systémov, kedy umožní odskúšanie navrhnutého systému na inej platforme.

Na Obr.3.11 sú ukázané modely bežného systému a virtuálneho systému.



Obr. 3.11 Modely systémov a) nevirtuálny systém b) virtuálny

V súčasnej dobe sa virtuálne počítače znova dostavajú do pozornosti, pretože často je potrebné riešiť problém kompatibility systémov. Napr. v blízkej minulosti existovalo veľké množstvo programov pre operačný systém MS-DOS, ktorý bol určený pre systémy s procesorom rady Intel. Veľké firmy, ktoré využívali rýchlejšie procesory, chceli umožniť svojim používateľom spúšťať aj aplikácie z MS-DOS-u. Riešením bolo vytvorenie virtuálneho počítača s procesorom Intel nad existujúcim procesorom. MS-DOS programy bežali v tomto prostredí tak, že ich inštrukcie sa prekladali do vlastného inštrukčného súboru. Výsledkom bol program, ktorý akoby bežal na procesore z rady Intel, pričom skutočný procesor bol celkom iný. Ak procesor bol dostatočne rýchly, program z MS-DOS-u mohol bežať rýchlo aj za predpokladu, že každá inštrukcia sa prekladala do niekoľkých vlastných inštrukcií.

Príkladom využitia virtuálneho počítača je napr. Java Virtual Machine, Portable Standard Lisp, Parallel Virtual Machine a mnoho iných.

4 PROCESY

Každý operačný systém musí byť schopný vykonávať niekoľko programov súčasne. Dokonca aj u systémov, kde používateľ môže spustiť len jeden program, operačný systém musí podporovať aj svoje vnútorné programy. V mnohých aspektoch bežiace programy, či sú to používateľské alebo systémové, sú si podobné a nazývajú sa procesy.

4.1 Proces

Neformálna definícia *procesu* hovorí, že je to program, ktorý sa vykonáva. Vykonanie procesu postupuje sekvenčne, t.j. v každom okamihu sa vykonáva len jedna inštrukcia programu.

V dávkových systémoch je zaužívaný termín *job* a v time-sharing-ových systémoch sa používateľské programy nazývajú *tasky*. Avšak vo väčšine operačných systémov je používaný termín proces, ktorý budeme používať aj v tomto texte.

Proces je niečo viac ako *kód programu*, ktorý sa vykonáva (niekedy nazývaný textový segment). Proces je definovaný aj kontextom, v ktorom sa vykonáva. Do kontextu patria hodnoty *čítača inštrukcií* a *registrov procesora*. Proces zahŕňa aj *zásobník*, ktorý obsahuje dočasne dáta procesu (ako sú parametre podprogramov, návratové adresy a lokálne premenné) a *dátový segment*, ktorý obsahuje globálne premenné.

Zdôrazňujeme, že program sám osebe nie je proces, program je *pasívna* jednotka, je to obsah súboru, ktorý je uložený na disku, zatiaľ čo proces je *aktívna* jednotka, v ktorej čítač inštrukcií určuje, ktorá inštrukcia sa bude vykonávať. K procesu patria aj prostriedky systému, ktoré proces potrebuje pre svoje vykonanie.

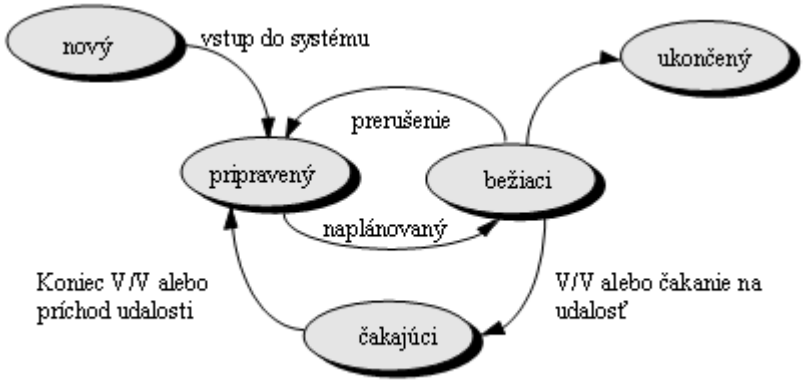
Nad jedným programom sa môže vykonávať viac procesov, ale každý program má svoju sekvenciu vykonávania. Na druhej strane je možné, že jeden proces počas svojho behu vytvorí viacej procesov.

Keď sa množina procesov vykonáva na jednom procesore použitím techniky zdieľania času (time-sharing), hovoríme o paralelnom sekvenčnom vykonaní alebo pseudo-paralelnom vykonaní. Ak počítačový systém má viac ako jeden procesor, potom procesy môžu bežať paralelne.

4.1.1 Stavy procesu

Počas svojho behu proces mení stavy. Každý proces môže byť v jednom z nasledujúcich stavov:

- **Nový** - proces je práve vytvorený.
- **Bežiaci** - vykonávajú sa inštrukcie programu.
- **Čakajúci** - proces čaká na nejakú udalosť, napr. dokončenie V/V operácie alebo na signál.
- **Pripravený** - proces čaká na pridelenie procesora.
- **Ukončený** - proces dokončil svoje vykonanie.



Obr. 4.1 Stavy procesu

Čitateľ sa môže stretnúť aj s inými názvami stavov v rôznych operačných systémoch, ako aj s jemnejším rozlíšením stavov. Stavový diagram procesu je uvedený na Obr. 4.1.

Prechody medzi jednotlivými stavmi procesu môžu nastať v týchto prípadoch:

- **Null ⇒ Nový :** pri vytvorení nového procesu. Napr. spustenie dávkovej úlohy, interaktívne prihlásenie sa nového používateľa, vytvorenie nového procesu operačným systémom pre poskytnutie nejakej služby, alebo keď bežiaci proces vytvorí potomka.

Nový ⇒ Pripravený: OS presúva proces do frontu pripravených procesov, keď je pripravený vytvoriť nový proces. Mnoho systémov obmedzuje počet bežiacich procesov, aby sa predišlo poklesu výkonu systému z dôvodu nedostatku prostriedkov.

Pripravený ⇒ Bežiaci: do stavu bežiaci sa proces dostane vtedy, keď čas procesora, pridelený práve bežiacemu procesu sa vyčerpá a je potrebné vybrať nový proces na spustenie.

Bežiaci ⇒ Ukončený: bežiaci proces skončí sám alebo je ukončený nasilu.

Bežiaci ⇒ Pripravený: najčastejšia príčina pre tento prechod je vyčerpanie času, určeného bežiacemu procesu na neprerušené vykonanie. Tento čas je závislý od algoritmu plánovania.

Bežiaci ⇒ Čakajúci: do tohto stavu sa proces dostane, ak musí čakať na určitú udalosť, napr. na dokončenie V/V operácie, na uvoľnenie zdieľaného systémového prostriedku, na správu od iného procesu atď.

Čakajúci ⇒ Pripravený: keď sa vyskytne udalosť, kvôli ktorej proces bol zablokovaný.

Niektoré systémy pripúšťajú ukončenie procesu zo stavu pripravený alebo zablokovaný.

4.1.2 Udalosti počas behu procesu

Počas behu procesu môžu nastať udalosti, ktoré vyžadujú osobitnú obsluhu.

a) **interné** - tieto udalosti vznikajú v rámci procesu a zapríčinia zmenu stavu procesu:

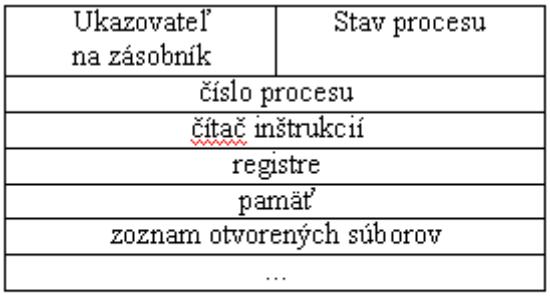
- systémové volanie - skok do jadra,
- chyba - zlá inštrukcia, porušenie oprávnenia k prístupu do pamäte atď.,
- zlyhanie stránky (page fault) - výpadok stránky pri virtualizácii stránkovaním.

b) **externé** - udalosti, ktoré proces neriadi, sú to vonkajšie udalosti a obyčajne ich oznamujú prerušenia, ktoré obsluhuje operačný systém:

- vstup z terminálu (znak),
- ukončenie diskovej operácie,
- prerušenie od časovača.

4.1.3 Riadiaci blok procesu (PCB)

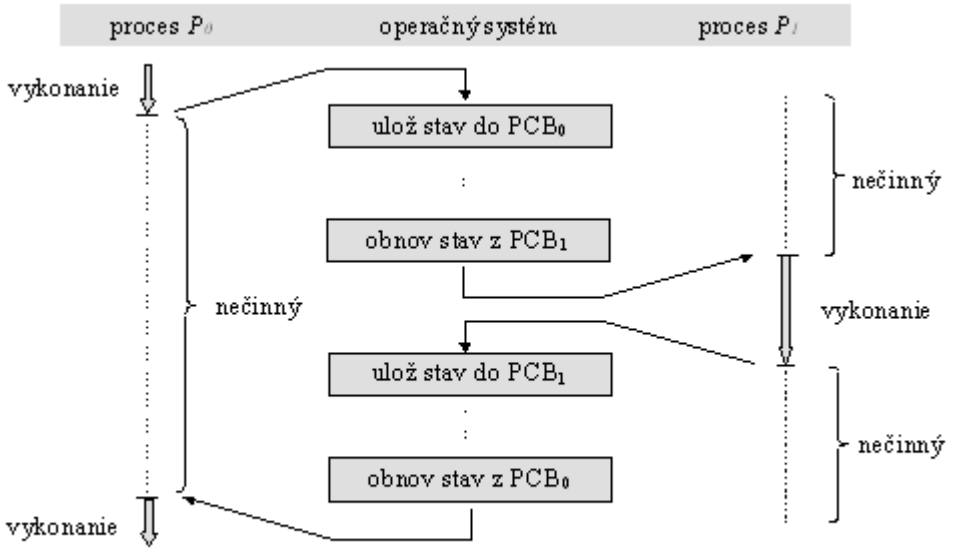
Každý proces je prezentovaný v operačnom systéme dátovou štruktúrou, ktorá sa nazýva *riadiaci blok procesu (Process Control Block*) – Obr. 4.2.



Obr. 4.2 Riadiaci blok procesu

PCB obsahuje mnoho informácií o procese, z ktorých najdôležitejšie sú:

- **ukazovateľ na zásobník procesu,**
- **stav procesu** - nový, pripravený, bežiaci, čakajúci atď.,
- **hodnota čítača inštrukcií** - indikuje adresu inštrukcie, ktorá bude vykonaná ako nasledujúca,
- **registre CPU** - počet a typ registrov sa mení podľa architektúry počítača. Sú to akumulátory, index registre, ukazovatele zásobníkov, univerzálne registre, informácie o podmienených kódov a iné. Obsahy týchto registrov spolu s čítačom inštrukcií sa uchovávajú pri prerušení, aby sa proces mohol neskôr spustiť od inštrukcie, pred ktorou bol prerušený (Obr. 4.3).



Obr. 4.3 Prepínanie kontextu

- **informácie pre plánovanie procesu** - priorita procesu, ukazovatele na fronty pre plánovanie a iné.
- **informácie pre správu pamäte** - hodnoty limitných a bázových registrov, tabuľka stránok alebo segmentov, podľa použitej techniky správy pamäte.
- **účtovacie informácie** - spotrebovaný čas CPU, časové limity pre proces atď.
- **V/V informácie** - obsahujú zoznam V/V zariadení, ktoré sú pridelené procesu, zoznam otvorených súborov atď.

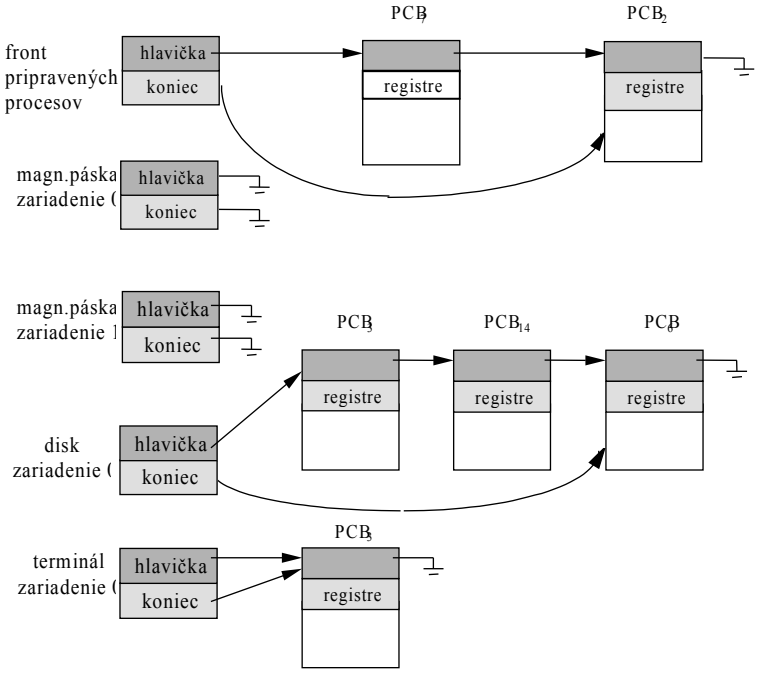
4.2 Plánovanie procesov

Multiprogramovanie bolo zavedené s cieľom zvýšiť efektívnosť využitia procesora. Podstata multiprogramovania je v tom, že v pamäti je viac procesov naraz a procesor prepína medzi nimi. Tak sa znižuje celkový čas vykonania jednej úlohy. Multiprogramovanie dovoľuje aj zdieľanie času. Podstatou *zdieľania času* (time-sharing) je rýchle prepínanie medzi procesmi tak, aby používatelia mohli komunikovať so svojimi programami interaktívne. Systém s jedným procesorom môže spracovávať vždy len jeden proces v danom okamihu. Zostávajúce procesy musia čakať. Plánovanie procesov má za úlohu určiť ktorému procesu bude pridelený procesor pri prepnutí.

4.2.1 Fronty

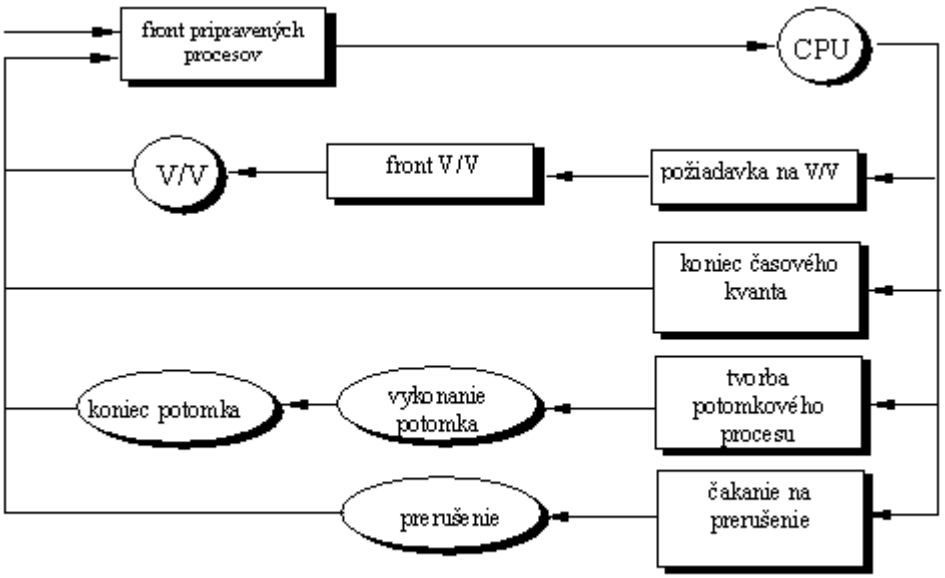
Keď vznikne nový proces, pridá sa do *frontu procesov* (job queue). V tomto fronte sú všetky procesy systému. Procesy, ktoré sú momentálne v operačnej pamäti a sú pripravené pre spustenie, sa radia do frontu *pripravených procesov* (ready queue). Tento front je obyčajne vytváraný ako zreťazený zoznam riadiacich blokov procesov. Hlavička tohto zoznamu obsahuje ukazovatele na začiatok a koniec frontu.

V systéme sa vytvára viacej frontov. Počas vykonania procesu sa môže stať, že proces bude musieť čakať na určitú udalosť napr. ukončenie V/V operácie na niektorom zo zdieľaných zariadení systému - magnetická páska, disk. Pretože v systéme je viac procesov, ktoré môžu mať také isté požiadavky, príslušné zariadenie môže byť obsadené. Potom pred každým zariadením sa vytvára front čakajúcich procesov - *front zariadenia (device queue).* Každé zariadenie má svoj vlastný front (Obr. 4.4).



Obr. 4.4 Front pripravených procesov a fronty periférnych zariadení

Na Obr. 4.5 je reprezentovaný proces plánovania procesov pomocou frontov. Ukázané sú dva typy frontov: front pripravených procesov a množina frontov periférnych zariadení. V krúžkoch sú zobrazené prostriedky, ktoré obsluhujú fronty.



Obr. 4.5 Diagram frontov, ktoré reprezentujú plánovanie procesov

Nový proces je na začiatku umiestnený do frontu pripravených procesov a čaká tam kým mu nie je poskytnutý procesor. Po získaní procesora, proces môže pokračovať jedným z nasledujúcich spôsobov:

- proces požiada o V/V operáciu a bude umiestnený do frontu príslušného zariadenia,
- proces vytvorí nový podproces a čaká na jeho dokončenie,
- procesu môže byť odobratý procesor ako výsledok prerušenia a proces je umiestnený do frontu pripravených procesov,
- skončí časové kvantum pridelené procesu

V prvých dvoch prípadoch proces môže prepnúť zo stavu čakajúci do stavu pripravený a potom je umiestnený do frontu pripravených procesov. Ďalej proces pokračuje v tomto cykle kým sa nedokončí, potom sa zo všetkých frontov odstráni jeho PCB a prostriedky sa uvoľnia.

4.2.2 Plánovače (schedulers)

Počas svojej existencie procesy putujú medzi jednotlivými frontmi. Operačný systém musí vyberať nejakým spôsobom procesy z týchto frontov. Proces výberu vykonáva príslušný *plánovač* (*scheduler*) .

V dávkových systémoch do systému postupuje viac úloh, ako môže byť naraz vykonávaných. Preto sa ukladajú na disk a plánujú sa v dvoch fázach.

- *Dlhodobý plánovač* vyberá z týchto procesov a ukladá ich do pamäte.
- *Krátkodobý plánovač* vyberá z pripravených procesov v pamäti a prideľuje CPU jednému z nich.

Základným rozdielom medzi týmito plánovačmi je frekvencia ich vykonávania. Krátkodobý plánovač sa spúšťa približne raz za 100 milisekúnd. Musí byť veľmi rýchly. Ak mu rozhodnutie o tom, ktorý proces má dostať CPU zaberie 10 milisekúnd, potom 10/(100+10)=9 percent z času CPU je venovaných len rozhodovaniu.

Dlhodobý plánovač sa vykonáva menej často. On v podstate kontroluje *úroveň multiprogramovania (počet procesov v pamäti).* Ak je táto úroveň stabilná, tak sa priemerný počet novovytvorených procesov rovná priemernému počtu ukončených procesov.

Výber procesu dlhodobým plánovačom je dôležitý, pretože väčšina procesov sa dá definovať podľa ich nárokov na V/V alebo CPU. Niektoré procesy sú náročnejšie na čas procesora, iné zas vyžadujú dlhé V/V operácie. Dlhodobý plánovač musí vhodne striedať procesy s rôznymi charakteristikami, aby zaistil efektívne využitie celého systému.

V niektorých systémoch nie sú dlhodobé plánovače. Napr. time-sharing-ové systémy často nemajú dlhodobý plánovač. Tam sa často objavuje iný, dodatočný plánovač, ktorý obstaráva strednú úroveň plánovania. Základná idea spočíva v tom, že niekedy je výhodné znížiť úroveň multiprogramovania odsunutím niektorého z procesov z pamäte na disk v rozpracovanom stave a neskôr ho tam znova vrátiť. Táto metóda sa nazýva *výmena (swapping)* a bude rozobraná podrobnejšie v kapitole o správe pamäte.

4.2.3 Prepínanie kontextu

Pridelenie CPU inému procesu znamená, že stav starého procesu sa musí uschovať a zaviesť stav nového procesu. Táto úloha sa nazýva *prepínanie kontextu.* Táto činnosť je pomocná a počas jej trvania systém nevykonáva užitočnú prácu. Rýchlosť prepínania je závislá na HW. Ovplyvňujú ju rýchlosť pamäte, počet registrov, existencia špeciálnych inštrukcií (ako napr. jedinou inštrukciou uložiť a obnoviť obsahy registrov). Obyčajne sa rýchlosť pohybuje medzi 1 a 1000 μs.

4.3 Operácie nad procesmi

4.3.1 Tvorba procesu

Počas svojej existencie proces môže vytvoriť niekoľko nových procesov pomocou systémového volania. Vytvárajúci proces sa nazýva *rodič* a nový proces *potomok*. Každý z potomkov môže tiež vytvárať nové procesy, a tak vzniká strom procesov (Obr. 4.6).

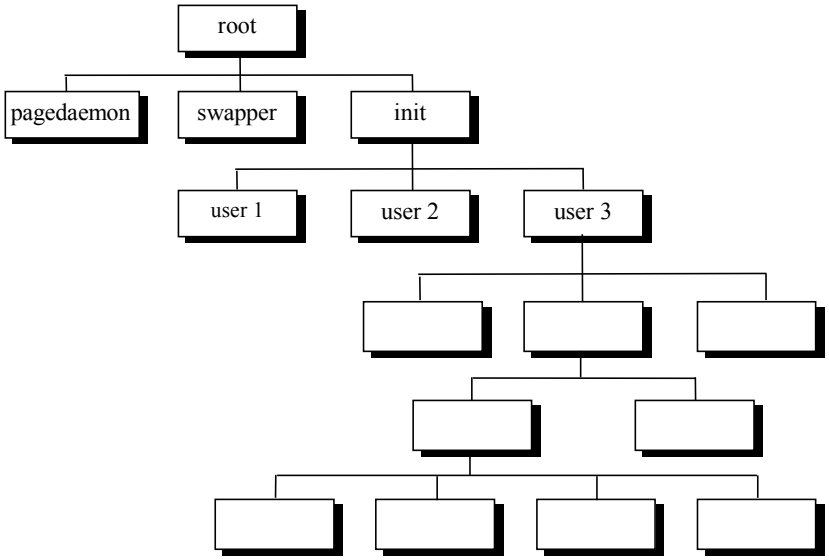
Pre svoju činnosť proces potrebuje prostriedky systému - CPU, pamäť, súbory, V/V zariadenia .Nový proces môže získať prostriedky priamo od OS, alebo môže dostať podmnožinu z rodičovských prostriedkov, alebo tieto budú zdieľané medzi niekoľkými potomkami.

Keď proces vytvorí potomka existujú dve možnosti, čo sa týka jeho pokračovania:

- rodič beží ďalej spolu s potomkovým procesom,
- rodič čaká na ukončenie niektorého z potomkov alebo na ukončenie všetkých potomkov.

V súvislosti s adresným priestorom sú možné tieto varianty:

- potomkový proces je duplikátom rodiča,
- potomkový proces sa vykonáva nad iným programom.



Obr. 4.6 Strom procesov v typickom Unix-ovskom systéme

Príklady:

UNIX: Proces sa vytvára systémovým volaním *fork.* Nový proces je kópiou rodičovského procesu. Iný kód nového procesu sa zavedie novým systémovým volaním - *execve.* Rodič môže vytvoriť viacero potomkov a ak nevykonáva inú činnosť, pomocou systémového volania *wait* môže počkať na ukončenie potomkov.

VAX/VMS: pri tvorbe nového procesu hneď zavádza do pamäte špecifikovaný program.

Windows NT: podporuje obidva modely - nový proces môže duplikovať adresný priestor rodičovského procesu, alebo rodič môže špecifikovať meno programu, ktorý bude vykonávať potomok.

1.3.2 Ukončenie procesu

Proces končí svoju činnosť systémovým volaním - *exit.* Pri ukončení procesu systém vracia jeho prostriedky do fondu voľných prostriedkov.

Proces môže spôsobiť ukončenie iného procesu pomocou systémového volania (napr. *abort*). Obyčajne toto volanie môže použiť iba rodičovský proces. Rodič môže ukončiť potomkový proces z rôznych dôvodov:

- potomkový proces prekročil využitie niektorého z prostriedkov, ktoré mu boli pridelené,
- úloha, ktorú plnil potomkový proces nie je viac potrebná,
- rodičovský proces končí a OS nedovoľuje existenciu potomka bez rodiča.

Veľa systémov nedovoľuje existenciu potomkov po ukončení (normálnom alebo násilnom) práce rodiča.

Príklad: V UNIXe sa môže proces ukončiť volaním *exit* a jeho rodič môže počkať na ukončenie potomka volaním *wait.* Pri tomto volaní sa vracia identifikátor ukončeného procesu, takže rodičovský proces vie, ktorý z potomkov skončil. Avšak ak skončí rodičovský proces, všetci jeho potomkovia sa tiež ukončia.

4.4 Spolupracujúce procesy

Paralelne bežiace procesy môžu byť buď *nezávislé* alebo *spolupracujúce*. Proces je nezávislý, ak nemôže ovplyvniť a nemôže byť ovplyvnený iným procesom. Z toho vyplýva, že každý proces, ktorý nezdieľa dáta s iným procesom je nezávislý a naopak, proces ktorý zdieľa dáta, je spolupracujúci. Je niekoľko dôvodov pre spoluprácu medzi procesmi:

- **Zdieľanie informácií** (napr. súbory) - v tomto prípade je potrebné poskytnúť mechanizmus pre paralelný prístup k príslušnému typu prostriedku.
- **Rýchlosť výpočtu** - ak potrebujeme čo najrýchlejší priebeh nejakého výpočtu, môžeme úlohu rozdeliť na menšie úlohy, ktoré pobežia paralelne. Samozrejme, tento model výpočtu je možný, len ak máme k dispozícii viac procesorov alebo V/V kanálov.
- **Modularita** - ak systém je navrhnutý modulárne tak, že jednotlivé systémové funkcie sa vykonávajú v rôznych procesoch, vzniká potreba spolupráce medzi nimi.
- **Výhoda** - jeden používateľ môže mať viacej spustených úloh naraz, napr. editovanie, tlač a výpočet .

Paralelné vykonanie, ktoré vyžaduje spoluprácu medzi procesmi, vyžaduje aj mechanizmy pre komunikáciu medzi nimi a pre synchronizáciu ich činností.

Klasická úloha, ktorá ilustruje spoluprácu medzi procesmi je *producent-konzument*. Proces *producent* produkuje nejaké informácie a proces *konzument* ich spotrebováva. Obidva procesy zdieľajú spoločný bufer. Proces *producent* zapisuje do bufra, proces *konzument* číta z neho. Úloha synchronizácie spočíva v tom, že konzument sa nesmie pokúšať spotrebovať prvok, ktorý ešte nebol vytvorený, a producent nesmie zapisovať do plného bufra. OS v tomto prípade môže poskytnúť bufer cez mechanizmy komunikácie medzi procesmi, alebo programátor ho vytvorí sám pomocou zdieľanej pamäte.

4.5 Vlákna

Procesy sú charakterizované prostriedkami, ktoré vlastnia a svojim adresným priestorom. Často sa vyskytujú prípady, kedy je užitočné, aby procesy zdieľali prostriedky. Táto situácia je podobná systémovému volaniu *fork*, kedy sa vytvorí nový proces s tým istým adresným priestorom a novým čítačom inštrukcií. Táto koncepcia sa ukázala natoľko užitočná, že väčšina moderných OS poskytuje mechanizmy pre tvorbu vlákien.

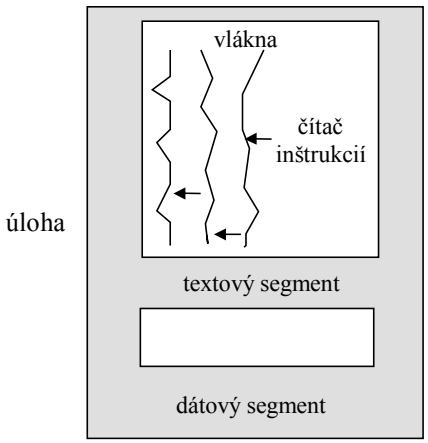
4.5.1 Štruktúra vlákna

Vlákno (thread), niekedy nazývané *odľahčený proces (LWP - Lightweight process)* je základná jednotka pre plánovanie činnosti procesora a pozostáva z:

- čítača inštrukcií,
- sady registrov,
- a zásobníka.

S ostatnými „príbuznými“ vláknami zdieľa kód, dáta a prostriedky (otvorené súbory, signály, atď.). Tradičný proces ako sme ho doteraz poznali, je úloha s jedným vláknom. Tvorba vlákien a prepínanie medzi nimi je „lacnejšie“ ako u tradičných procesov a ochrana pamäte nie je potrebná. Vlákna v rámci úlohy sú ukázané na Obr. 4.7. Každé vlákno vykonáva časť kódu, ale zdieľa adresný priestor s ostatnými „príbuznými“ vláknami.

Vlákna v mnohom fungujú obdobne ako procesy. Vlákno môže byť v stave *pripravený, zablokovaný, bežiaci* alebo *ukončený*. Obdobne len jedno vlákno v danom čase využíva procesor. V rámci procesu sa vlákna vykonávajú sekvenčne a každé vlákno má svoje počítadlo inštrukcií a zásobník. Vlákna môžu vytvárať potomkov a môžu sa zablokovať, kým sa uskutoční systémové volanie. Kým je jedno vlákno zablokované, vykonáva sa iné. Na rozdiel od procesov, vlákna nie sú od seba nezávislé. Pretože vlákna majú prístup k celému adresnému priestoru úlohy, môžu čítať a zapisovať do zásobníkov iných vlákien. Tieto štruktúry nie sú chránené pred zásahom iných vlákien. Takáto ochrana nie je potrebná, pretože vlákna patria jednému používateľovi a sú navrhované za účelom spolupráce v rámci jednej úlohy.



Obr. 4.7 Vlákna v rámci úlohy

4.5.2 Implementácia vlákien

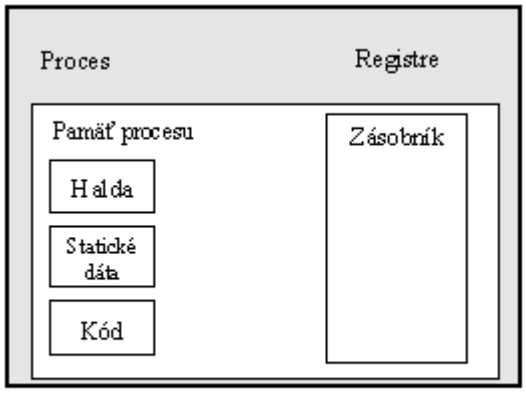
Uvedieme dva základné spôsoby implementácie vlákien:

- Jednoduchšie je vytvorenie **run-time** prostredia (budeme ho nazývať *run-time* modul), ktoré bude zostavené (spojené) spolu s programom. *Run-time* modul je zostavený s používateľským programom a pri inicializácii vykonateľného programu je mu odovzdané riadenie. Taktiež všetky funkcie súvisiace s vláknami sú vykonávané pod „dohľadom“ *run-time* modulu. V tomto prípade sa celý preložený a zostavený program javí z pohľadu operačného systému ako jediný proces. Jedným zo základných požiadaviek *run-time* modulu vzhľadom na hostiteľský operačný systém je získavať (najlepšie cestou programového prerušenia) časové signály. Ak je k dispozícii vhodný časový vstup, nie je už problémom zabezpečiť prepínanie kontextu a plánovanie jednotlivých častí programu. *Run-time* modul len ukladá kontext prerušeného vlákna (registre a ukazovateľ zásobníka) a obnovuje kontext vlákna, ktorému odovzdáva riadenie. Jedným z implementačných problémov v tomto prípade je oblasť zásobníka. Ak nejakému vláknu „podtečie“ zásobník, nutne je prepísaný kontext zásobníka nejakého iného vlákna. To sa čiastočne rieši stráženou oblasťou v zásobníku. Akonáhle *run-time* modul zistí narušenie stráženej oblasti, môže zrušiť celý program alebo vlákna, ktorých sa porucha týka.

Ďalší problém je v tom, že nie vždy je možné takto riešené vlákna celkom „oslobodiť“ od spôsobu riadenia procesu operačným systémom. Preto i v tejto implementácií môže prísť k stavu, kedy zablokovanie jedného vlákna pri vykonávaní V/V operácie zablokuje následne celý proces a teda všetky ostatné vlákna. S využitím *run-time* modulu je riešená implementácia vlákien vo väčšine starších operačných systémov. Ak sú vyvíjané prostriedky, ktoré implicitne existenciu vlákien potrebujú, napr. distribuované výpočtové prostredie - OSF DCE, obsahujú spravidla vlákna implementované uvedeným spôsobom. Ak sú potom príslušné úlohy prevádzkované pod operačným systémom vyššej úrovne, je možné použiť implementáciu vlákien hostiteľského operačného systému.

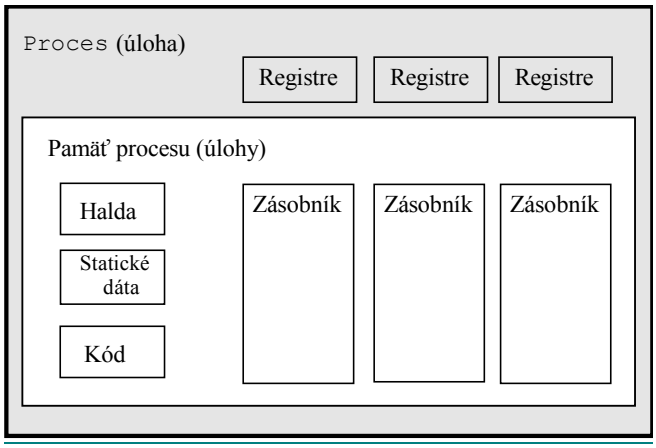
- Iný spôsob predstavuje **implementácia vlákien podporovaná priamo jadrom operačného systému**. V tomto prípade nie je spravidla z pohľadu operačného systému jednotkou plánovania proces. Pojem proces býva obecne nahradený dvoma novými pojmami: *vláknom a úlohou.* Vlákno do značnej miery preberá význam procesu - má vlastný kontext (s registrami a zásobníkom), ale všetky časovo náročné operácie sú vykonávané na úrovni úlohy. Vlákno sa stáva základnou samostatne plánovanou entitou vo vnútri úlohy, ale i celého operačného systému; má prístup ku všetkým častiam úlohy, má len jednoduché základné stavy v ktorých sa môže nachádzať, a môže byť vykonávané paralelne s ostatnými vláknami úlohy. Úloha predstavuje akúsi „obálku“ vlákien jedného programu a z pohľadu jadra operačného systému je to predovšetkým entita dôležitá pre prideľovanie a riadenie požadovaných systémových zdrojov. Pokiaľ je to možné, jadro operačného systému pracuje len s vláknom a len v nevyhnutných prípadoch identifikácie, prideľovania a plánovania systémových zdrojov pracuje s úlohou. Pochopiteľne ako úloha, tak aj vlákna majú v jadre príslušné dátové štruktúry, ktoré sú vzájomne previazané a zabezpečujú vždy jednoznačnú vzájomnú identifikáciu. Spravidla býva táto implementácia vykonaná na systémoch s tzv. *mikrojadrom (microkernel technology).*

Štruktúry jednotlivých spôsobov implementácie sú na Obr. 4.8 až 4.10. V prvom prípade (Obr. 4.8) je zachytená štruktúra „štandardného“ procesu, ktorý si môžeme predstaviť ako jedno vlákno. Vonkajší rámček predstavuje celý kontext procesu s registrami a potrebnými dátovými štruktúrami pre riadenie procesu jadrom. Vnútorný rámček predstavuje rozdelenie pamäte procesu na základné komponenty - oblasť kódu, vymedzený rozsah pamäte pre štatistické dáta a oblasť dynamicky prideľovanej pamäte, nazývanej halda.



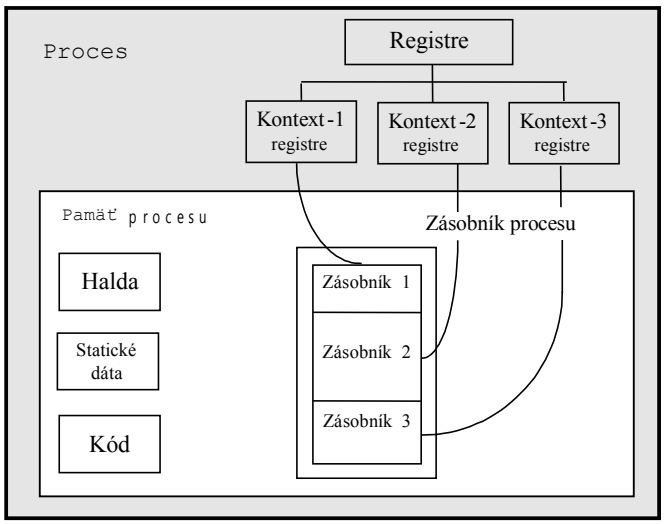
Obr. 4.8 Štruktúra štandardného procesu

Obr. 4.9 znázorňuje stav v operačnom systéme, ktorý podporuje vlákna na úrovni jadra. Tu je kontext procesu delený na časti, ktoré sa týkajú jednotlivých vlákien: samostatné oblasti pre ukladanie registrov a samostatné dátové štruktúry pre jednotlivé vlákna. To všetko má nakoniec obálku (ktorú zabezpečuje úloha), umožňujúcu jadru akýsi globálny pohľad na jednotlivé vlákna, ktoré patria k jednému procesu. V dátovej oblasti je však len jediná podstatná zmena v tom, že každé vlákno má svoj vlastný zásobník. Oblasti kódu, statických a dynamických dát sú spoločné v rámci celého procesu.



Obr. 4.9 Proces s viacerými vláknami (podpora v jadre)

Obr. 4.10 zobrazuje stav, kde sú vlákna podporované *run-time* prostredím vo vnútri procesu. Z pohľadu jadra operačného systému je situácia rovnaká ako na Obr. 4.8, t.j. jeden proces, jeden kontext. Vo vnútri procesu je však zabudovaný mechanizmus, ktorý vie „ukladať" kontext jednotlivých vlákien (predovšetkým sú to hodnoty registrov v okamihu prerušenia vlákna, ale i ostatné interné dátové štruktúry vlákna). Rovnaký mechanizmus delí zásobník na oblasti, využívané jednotlivými vláknami. To znamená, že pri preplánovaní vlákna sa menia nielen hodnoty registrov a základných interných riadiacich štruktúr, ale je nastavená aj iná úroveň zásobníka. Kód, statické dáta a halda sú v tomto prípade zdieľané celkom prirodzene. Pri dôslednej implementácii je však možné na halde vytvoriť kontextovo závislé oblasti so vzťahom ku konkrétnemu vláknu. Inak pochopiteľne platí, že halda je spoločná, ale jednotlivé alokované časti sú známe len vláknu, ktoré ju alokovalo. V tejto implementácii vynikne význam stráženej oblasti zásobníka.



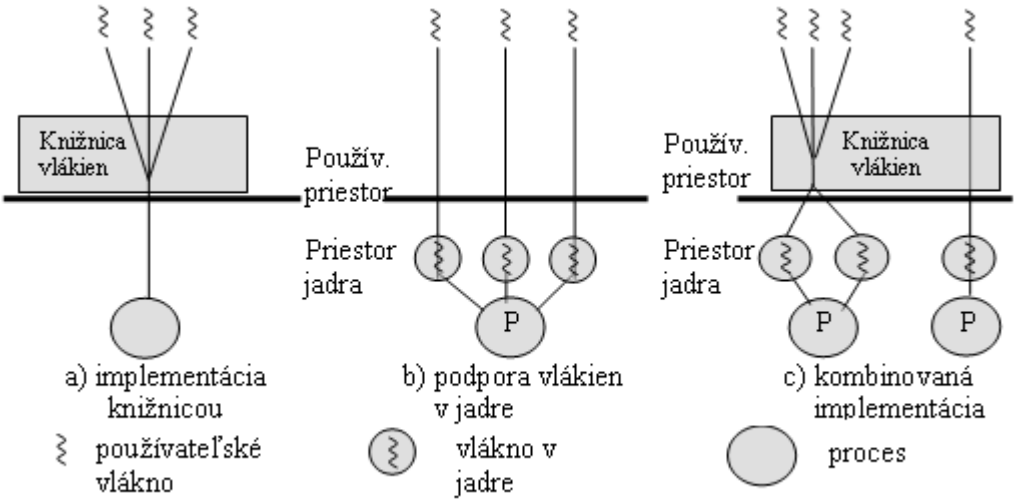
Obr. 4.10 Proces s viacerými vláknami (run-time podpora)

Na Obr. 4.11 sú ukázané všetky možné spôsoby implementácie vlákien, t.j. implementácia knižnicou, implementácia využívajúca podporu v jadre a kombinovaný spôsob, kedy jedna časť vlákien je implementovaná knižnicou a niektoré vlákna majú podporu priamo v jadre.

4.5.3 Základné atribúty vlákna

Pri spracovaní vlákna sa správca (run-time podpora v procese, alebo jadro operačného systému) riadi určitými základnými atribútmi, ktoré popisujú nasledujúce vlastnosti vlákna:

- *Dedičstvo plánovacieho algoritmu:* Novovytvorené vlákno bude používať rovnaký plánovací algoritmus ako vlákno, ktoré ho vytvorilo. Pripomeňme, že úroveň na ktorej je proces odštartovaný, sa považuje za primárne vlákno; inými slovami, akýkoľvek vykonateľný kód je automaticky vláknom.
  - *Plánovací algoritmus*: popisuje, ako bude plánované vykonávanie vlákna vzhľadom na ďalšie vlákna v procese (programe).
  - *Plánovacia priorita* určuje prioritu, ktorá bude uvažovaná pri plánovaní vlákna.
  - *Veľkosť zásobníka* určuje minimálnu požadovanú veľkosť zásobníka vlákna.
  - *Veľkosť stráženej oblasti zásobníka*: Strážená oblasť zásobníka slúži pre detekciu „podtečenia“ zásobníka a nie je za normálnych okolností vláknu dostupná. Ak sa zmení jej obsah, je to indikácia poruchy činnosti vlákna.

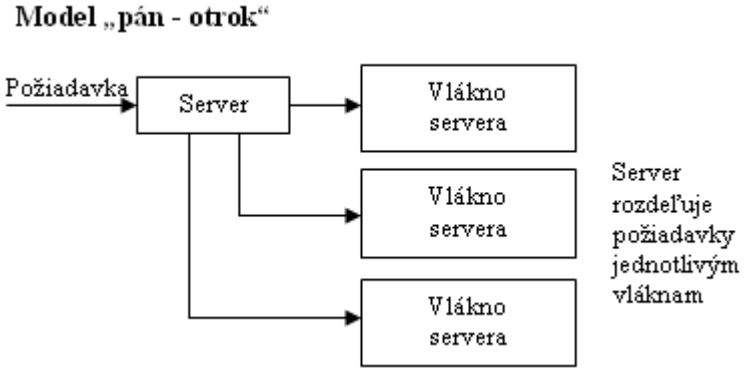


Obr. 4.11 Rôzne spôsoby implementácie vlákien

4.5.4 Modelové situácie použitia vlákien

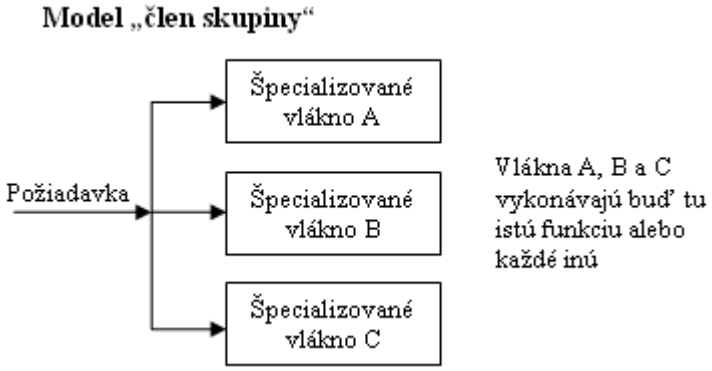
Tento úvod do problematiky vlákien by nebol úplný, keby sme neuviedli najobľúbenejšie modely použitia vlákien, ktorými sú:

1. **Pán/Otrok**. Jedno z vlákien preberá funkciu pána, ktorý rozdeľuje prácu, ostatné čakajú na zadanie práce a odovzdávajú informáciu o jej vykonaní. Zadávateľ - pán, môže buď čakať na dokončenie práce jedného z vlákien poverených vykonaním požiadavky, alebo je vybudovaný front požiadaviek zadávateľa a otroci sa snažia po vykonaní úlohy z fronty vytiahnuť ďalšiu požiadavku. (pozri nasledujúci obrázok).

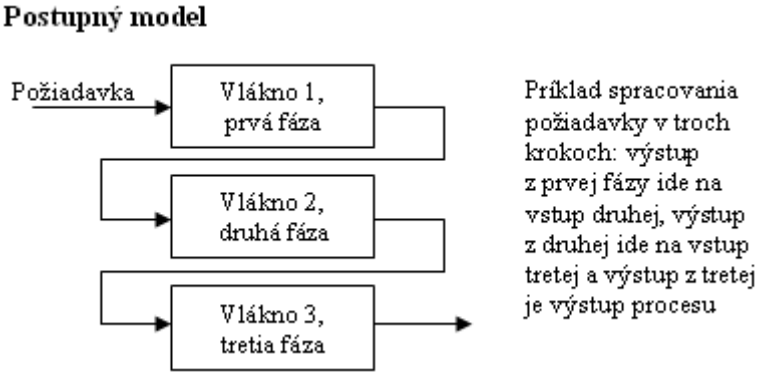


2. **Člen skupiny.** Tu je práca rozdelená tak, že sa *n* vlákien horizontálne delí o vykonanie požiadavky (t.j. každé vlákno má zodpovednosť za splnenie nejakej jej časti). Akonáhle vznikne požiadavka, je vystavená „objednávka“ a každé z výkonných vlákien vykoná jej adekvátnu časť. V tomto prípade nebýva spravidla spracovanie jednej časti požiadavky závislé na spracovaní ostatných častí a v

ideálnom prípade môžu byť jednotlivé časti jednej požiadavky spracované paralelne (pozri nasledujúci obrázok).



3. **Postupný model.** Práca je delená do vertikálnych krokov, pričom každé vlákno je zodpovedné za vykonanie jednej časti úlohy. Jednotlivé kroky sú vykonávané vertikálne tak, že každé vlákno je zainteresované na dokončení práce svojho predchodcu. V tomto prípade je po „rozpracovaní“ požiadaviek každá ďalšia požiadavka vykonávaná paralelne s dokončovaním predchádzajúcej požiadavky (pozri nasledujúci obrázok).



4.5.5 Zhrnutie

Použitie vlákien je možné doporučiť ako uľahčenie riešenia zložitejších požiadaviek a naviac ako možnosť paralelného vykonávania potrebných činností. Príkladom konkrétneho použitia môže byť situácia, kedy je potrebné v sieťovom modeli klient/server zabezpečiť, že jedna časť aplikácie bude trvalo na príjme požiadaviek od klienta a zvyšok sa bude snažiť o ich vykonanie. Základné vlákno - pán, bude zo siete zberať požiadavky na konkrétne spracovanie a ostatné vlákna budú z fronty požiadaviek vyberať svoje úlohy. Podpora vlákien môže byť poskytnutá knižnicou alebo priamo v jadre. Vo všeobecnosti vlákna, implementované knižnicou sú rýchlejšie ako tie, ktoré sú podporované jadrom.

Výpočet systémov podporujúcich technológiu vlákien by bol asi rozsiahly. Z praxe sú známe predovšetkým operačné systémy bývalej firmy Digital (Digital UNIX, ULTRIX a VMS), ktoré všetky vo svojich posledných verziách podporujú vlákna (niektoré priamo na úrovni jadra, iné pomocou knižnice). Vlákna sú podporované i v operačných systémoch Windows NT a Windows 2000 firmy Microsoft, v OS/2 a AIX firmy IBM, v HP-UX firmy Hewlett-Packard, Solaris firmy Sun, IRIX firmy SGI,Linux a iných. Niektoré z uvedených systémov podporujú vlákna na úrovni jadra, iné s pomocou knižnice.

Vlákna, ktoré poskytuje jazyk Jsva spravuje Java Virtual Machine a nie je možné ich zaradiť ani do kategórie implementovaných knižnicou, ani do kategórie implementovaných s podporou v jadre.

5 PLÁNOVANIE PROCESOV

Plánovanie času procesora patrí do základných funkcií operačného systému. Prideľovaním procesora jednotlivým procesom sa práca celého počítačového systému zefektívni.

5.1 Základné princípy

Plánovanie času procesora je základom multiprogramovania. Prepínaním CPU medzi procesmi OS zefektívňuje prácu počítača.

Základnou myšlienkou multiprogramovania je, aby stále bežalo niekoľko procesov, aby sa procesor maximálne využíval. V jednoprocesorových systémoch samozrejme beží v danom čase vždy len jeden proces. Ak v systéme je viac procesov, tie musia čakať, kým sa procesor uvoľní.

Idea multiprogramovania je jednoduchá. Proces sa vykonáva, kým nemusí z určitých dôvodov čakať napr. na dokončenie V/V operácie. V jednoduchom OS (ktorý nevyužíva multiprogramovanie) procesor v tejto dobe bude voľný a nebude vykonávať žiadnu užitočnú prácu. V multiprogramovom systéme sa pokúšame tento čas využiť efektívnejšie. V pamäti je viac programov naraz. Ak bežiaci proces je zablokovaný a musí čakať, OS mu odoberie procesor a pridelí ho ďalšiemu procesu.

Plánovanie prostriedkov patrí medzi najzákladnejšie úlohy OS. Skoro všetky prostriedky sa musia pred použitím naplánovať. Prakticky v operačnom systéme existujú štyri typy plánovania. Tri z nich sa týkajú procesov: *dlhodobé, strednodobé a krátkodobé plánovanie.* Štvrtý typ je *plánovanie obsluhy V/V požiadaviek* a bude rozobratý v kapitolách o správe periférnych zariadení. Plánovanie času procesora ako najzákladnejšieho prostriedku tvorí podstatnú časť návrhu OS a tomuto problému je venovaná táto kapitola.

5.1.1 Cykly práce procesora a periférií

Pozorovaním činností procesov sa zistilo, že vykonanie procesu pozostáva z cyklov práce procesora a čakania na V/V operácie. Práca procesu začína cyklom procesora a pokračuje striedaním cyklu procesora so stavmi čakania na V/V. Proces končí zasa cyklom procesora, aby mohol byť dokončený štandardnými dokončovacími operáciami.

Obr. 5.1 Histogram cyklov CPU

Aby bolo možné ohodnotiť proces, bolo zmerané trvanie cyklov práce procesora. Aj keď od procesu k procesu a od počítača k počítaču sa hodnoty týchto časov môžu značne líšiť, ich frekvenčná krivka má charakter, aký je ukázaný na Obr. 5.1, t.j. exponenciála alebo hyperexponenciála. Z nej môžeme vyčítať, že množstvo krátkych intervalov využitia procesora je veľké a množstvo dlhých intervalov čakania na V/V je malé. Programy viazané prevažne na V/V budú mať veľmi malé periódy využitia procesora a naopak, programy viazané na procesor budú mať tieto periódy veľmi dlhé. Tieto rozdiely sú dôležité pre správny výber plánovacieho algoritmu pre čas procesora.

5.1.2 Preemptívne plánovanie

Rozhodovanie o plánovaní času procesora sa môže urobiť vždy pri jednom z nasledujúcich prechodov:

1. Keď proces prepína zo stavu bežiaci do stavu čakajúci (čakanie na dokončenie V/V operácie alebo čakanie na ukončenie potomka).
2. Keď proces prepína zo stavu bežiaci do stavu pripravený.
3. Keď proces prepína zo stavu čakajúci do stavu pripravený.
4. Keď proces končí.

Pri prechodoch v bodoch 1 a 4 nie je možnosť výberu pre plánovanie. Nový proces sa musí vybrať pre vykonanie.

Keď sa plánovanie vykonáva len v prípadoch 1 a 4 hovoríme, že sa jedná o *nepreemptívne* plánovanie, inak plánovanie je *preemptívne*. Pri nepreemptívnom plánovaní, keď sa raz procesor pridelí procesu, proces sa vykonáva až do svojho ukončenia, alebo pokiaľ nevznikne požiadavka na V/V. Táto metóda plánovania je použitá v OS Microsoft Windows. Pre určité HW platformy je toto jediná možná metóda plánovania, pretože nevyžaduje špeciálny HW (napr. časovač).

Preemptívne plánovanie je náročnejšie. V tomto prípade treba rátať s procesmi, ktoré zdieľajú dáta, a je potrebné udržiavať tieto dáta v konzistentnom tvare pri prepnutí kontextu procesu. To vyžaduje dodatočné synchronizačné prostriedky, ako uvedieme ďalej.

Preemptívne plánovanie má vplyv na návrh jadra operačného systému. Počas spracovania systémového volania jadro vykonáva činnosť v prospech procesu. Počas tohto spracovania je možné, že jadro musí meniť dôležité dáta (napr. V/V fronty). Ak počas tejto činnosti je proces prepnutý, systémové dáta zostanú v nekonzistentnom stave, čo je neprípustný stav. Niektoré OS, vrátane väčšiny verzií Unix-u, riešia tento problém čakaním na ukončenie systémového volania alebo na V/V operáciu, pred prepnutím kontextu procesu.

5.1.3 Dispečer

Ďalší komponent, ktorý sa zúčastňuje na plánovaní času procesora, je *dispečer*. Dispečer je modul, ktorý umožňuje procesoru riadiť procesy, vybrané krátkodobým plánovačom. Jeho funkcie sú:

- prepínanie kontextu,
- prepínanie do používateľského režimu,
- skok na príslušnú adresu po opätovnom spustení programu.

Dispečer musí byť čo najrýchlejší, pretože sa volá pri každom prepnutí procesov.

5.2 Kritéria plánovania

Jednotlivé plánovacie algoritmy majú rôzne vlastnosti a môžu uprednostňovať rôzne skupiny procesov. Pri výbere algoritmu pre danú situáciu musíme mať na vedomí vlastnosti jednotlivých algoritmov.

Pre posudzovanie vlastností plánovacích algoritmov je možné zvoliť rôzne kritéria. Podľa toho, ktoré charakteristiky sú zvolené pre porovnanie algoritmov, je možné získať značné rozdiely pri určovaní optimálneho algoritmu. Používané kritériá sú nasledujúce:

- ***Využitie procesora***. Snaha je zamestnávať procesor čo najviac. Využitie procesora môže byť od 0 do 100 percent. V reálnom systéme využitie procesora by malo byť v hraniciach od 40 % (pre málo zaťažený systém) do 90 % (pre silne zaťažený systém).

- ***Priepustnosť*.** Ak procesor spracováva proces, vykonáva prácu. Jedna z možností merania vykonanej práce je počet ukončených procesov pre danú časovú jednotku. Túto veličinu nazývame priepustnosť.

- ***Čas vykonania***. Časový interval od vzniknutia procesu do jeho ukončenia sa nazýva čas vykonania. Je to súčet času čakania na vstup do pamäte, času stráveného vo fronte pripravených procesov, času vykonávania a času V/V operácií.

- ***Čas čakania***. Algoritmy plánovania neovplyvňujú čas, ktorý proces venuje vykonávaniu V/V operácií, ale len čas, ktorý proces strávi čakaním vo fronte pripravených procesov.

- ***Čas odozvy***. V interaktívnom systéme čas vykonania nemusí byť najvhodnejšie kritérium. Často proces produkuje nejaké výsledky a potom pokračuje vo výpočte ďalších, kým sa predchádzajúce dostanú k používateľovi. Iným ukazovateľom je čas od vystavenia požiadavky do prvej odozvy na túto požiadavku, ktorý nazývame čas odozvy.

Vo všeobecnosti je žiadúce maximalizovať využitie procesora a priepustnosť a minimalizovať čas vykonania, čas čakania a čas odozvy. Avšak niekedy je žiadúce optimalizovať maximálne a minimálne hodnoty namiesto priemerných hodnôt. Napr. ak chceme aby všetci používatelia dostali dobré služby, budeme sa snažiť minimalizovať maximálnu dobu odozvy.

5.3 Plánovacie algoritmy

Plánovanie času procesora rieši problém, ktorému procesu z frontu pripravených procesov má byť pridelený procesor. V nasledujúcom oddieli popíšeme niektoré z týchto algoritmov.

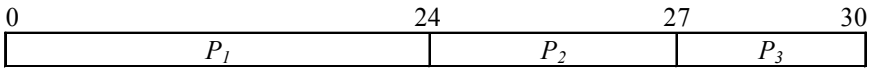
5.3.1 Spracovanie v poradí príchodu (FCFS - First Come, First Served)

Najjednoduchší z algoritmov plánovania je algoritmus spracovania v poradí príchodu. Podľa tohoto algoritmu proces, ktorý požiadal prvý o pridelenie procesora ho dostane ako prvý. Implementácia tohoto algoritmu sa uskutočňuje pomocou frontu FIFO. Keď proces vstúpi do frontu pripravených procesov, jeho riadiaci blok (PCB) sa zaradí na koniec frontu. Keď sa procesor uvoľní, pridelí sa procesu, ktorý je na čele frontu. Bežiaci proces sa odstráni z frontu.

Stredná doba čakania pri použití FCFS je často veľmi dlhá. Predpokladajme, že nasledovná množina procesov vzniká v čase 0 a má požiadavky na čas procesora, ktoré sú zadané v milisekundách (ďalej *ms*):

Proces	Požadovaný čas procesora
P_1	24
P_2	3
P_3	3

Ak procesy prídu v poradí P_1, P_2, P_3 a sú obsluhované v poradí FCFS, potom získame tento diagram:



Pre časy čakania získame nasledujúce hodnoty. Proces P_1 nebude čakať, P_2 bude čakať 24 *ms*, P_3 bude čakať 27 *ms*. Takže priemerná doba čakania je (0 + 24 + 27)/3=17 *ms*. Ak procesy prídu v poradí P_2, P_3, P_1, potom výsledok pre čas čakania bude (6 + 0 + 3)/3=3 *m*s. Podstatné je zníženie času čakania. Takže priemerný čas čakania pri plánovaní podľa poradia príchodu je obecne dosť veľký a mení sa značne podľa požiadaviek procesov na čas procesora.

Algoritmus plánovania v poradí príchodu nie je preemptívny. Keď proces dostane raz pridelený procesor, vykonáva sa až do ukončenia, alebo kým nepožiada o V/V operáciu. Plánovanie procesov

podľa poradia ich príchodu môže neúmerne predĺžiť čas čakania krátkych procesov. Tento algoritmus je ťažko použiteľný v time-sharing-ových systémoch, kde je dôležité, aby každý používateľ získaval čas procesora v pravidelných intervaloch a nie je žiadúce, aby jeden proces zadržal procesor na dlhšiu dobu.

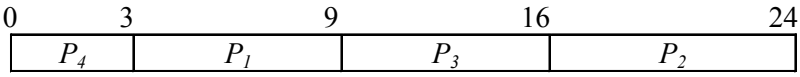
5.3.2 Najkratší proces najskôr (SJF - Shortest Job First)

Tento algoritmus poskytuje k plánovaniu iný prístup. Poradie spracovania procesov sa určuje podľa požadovanej doby obsluhy procesu. Keď sa procesor uvoľní, pridelí sa procesu, ktorý požaduje najmenšiu dobu na svoje dokončenie (ak proces medzitým bol čiastočne spracovaný). Ak dva procesy majú rovnaké požadované doby obsluhy, vyberajú sa podľa poradia príchodu. Čitateľ si určite všimne, že názov algoritmu nezodpovedá celkom jeho popisu, ale v literatúre sa ujal pod týmto názvom, preto je použitý aj v tomto texte.

Pre ilustráciu znova použijeme množinu procesov s rôznymi požiadavkami na čas procesora.

Proces	Požadovaný čas procesora	Doba čakania
P_1	6	3
P_2	8	16
P_3	7	9
P_4	3	0

Podľa algoritmu SJF poradie spracovania procesov bude P_4 , P_1 , P_3 , P_2.



Priemerná doba čakania je (3+16+9+0)/4=7 *ms*. Ak použijeme algoritmus FCFS, priemerná doba čakania bude 10.25 *ms*. Algoritmus SJF je optimálny v tom, že dáva *najlepšie výsledky v priemernej dobe čakania* pre danú množinu procesov.

Nedostatkom tohoto algoritmu je, že je potrebné dopredu vedieť dĺžku požadovanej doby obsluhy. Pri dávkovom spracovaní môžeme použiť časový limit pre spracovanie dávky, ktorý zadáva používateľ. Tak sú užívatelia nútení odhadovať požadované doby presne, pretože to im zaistí rýchlejšiu odozvu. Tento algoritmus sa často používa pri dlhodobom plánovaní.

Aj keď algoritmus SJF dáva optimálne výsledky, nedá sa použiť pre krátkodobé plánovanie, pretože nie je známa dĺžka ďalšej požiadavky procesu na čas procesora. Jeden možný prístup je odhad tejto hodnoty, ktorú nevieme, ale môžeme predpovedať. Očakávame, že ďalšia požiadavka bude mať dĺžku podobnú predchádzajúcim. Potom na základe predpovedaných dĺžok môžeme vybrať proces s najmenšou požiadavkou.

Ďalšia požiadavka sa obyčajne odhaduje ako exponenciálny priemer nameraných dĺžok predchádzajúcich požiadaviek. Nech t_n je dĺžka n-tej požiadavky na čas procesora a nech τ_{n+1} je naša predpoveď dĺžky ďalšej požiadavky. Potom pre α, kde $0 \leq \alpha \leq 1$ definujeme

$$\tau_{n+1} = \alpha . t_n + (1 - \alpha) . \tau_n$$

Tento vzorec definuje exponenciálny priemer. Hodnota t_n obsahuje poslednú informáciu, τ_n obsahuje históriu. Parameter α definuje relatívnu váhu blízkej a vzdialenejšej histórie našich predpovedí.

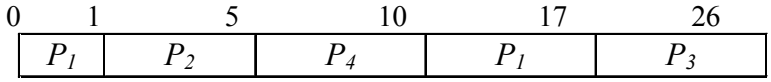
Ak $\alpha = 0$, potom $\tau_{n+1} = \tau_n$ t.j. blízka história nemá vplyv; ak $\alpha = 1$, potom $\tau_{n+1} = t_n$ t.j. vzdialenejšia história nemá vplyv. Najčastejšie $\alpha = 1/2$, čo znamená, že blízka a vzdialenejšia história predpovedí rovnako ovplyvňuje novú predpoveď.

Algoritmus SJF môže byť preemptívny a nepreemptívny. Výber sa robí, keď do frontu pripravených procesov príde nový proces a predchádzajúci sa ešte vykonáva. Nový proces môže mať menšie požiadavky na čas procesora ako zostávajúce požiadavky práve vykonávaného procesu. Preemptívny algoritmus prepne bežiaci proces, zatiaľ čo nepreemptívny ho nechá dobehnúť. Preemptívny algoritmus sa niekedy nazýva *plánovanie podľa najkratšej zostávajúcej doby na vykonanie (SRTF - Shortest Remaining Time First).*

Nasledujúci príklad uvádza 4 procesy, dĺžka doby vykonania a čas príchodu sú uvedené v *ms*.

Proces	Čas príchodu	Požadovaný čas procesora
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Plánovanie procesov podľa SJF bude nasledujúce :



Proces P_1 štartuje v čase 0, pretože je jediný proces vo fronte. Proces P_2 prichádza v čase 1. Zostávajúci čas procesu P_1 (7 *ms*) je väčší ako čas, ktorý požaduje P_2 (4 *ms*), takže proces P_1 je prepnutý a je naplánovaný proces P_2 . Priemerný čas čakania z tohto príkladu je

$$((10 - 1) + (1+1) + (17 - 2) + (5 - 3))/4 = 6.5 \text{ ms}.$$

Nepreemptívne plánovanie by dosiahlo čakaciu dobu 7.75 *ms*.

5.3.3 Prioritné plánovanie

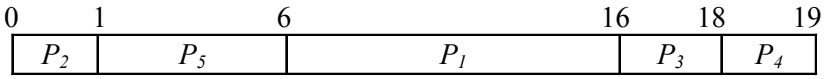
Algoritmus SJF je špeciálny prípad obecného algoritmu plánovania podľa priorít. Podľa algoritmu plánovania podľa priorít, každý proces má pridelenú prioritu a procesor je pridelený procesu s najvyššou prioritou. Procesy s rovnakou prioritou sa plánujú podľa poradia príchodu (FCFS).

Priority patria obyčajne do pevného intervalu celých čísel napr. od 0 do 7 alebo od 0 do 4095. Neexistuje obecne prijaté ustanovenie, že 0 je najnižšia alebo najvyššia priorita. Niektoré systémy používajú menšie čísla pre označenie menšej priority, iné naopak. V tomto texte nižšie čísla budú označovať vyššiu prioritu.

V nasledujúcom príklade predpokladáme, že množina procesov prichádzajúca v čase 0 je P_1, P_2 , ..., P_5 s dĺžkami požadovanej doby obsluhy udanými v *ms*.

Proces	Požadovaný čas procesora	Priorita
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Podľa prioritného plánovania sa procesy budú vykonávať v tomto poradí:



Priemerná doba čakania je 8.2 *ms*. Priority môžu byť definované buď interne alebo externe. Interne definované priority využívajú niektoré merateľné hodnoty, aby určili prioritu procesu, napr. časové limity, požiadavky na pamäť, počet otvorených súborov, pomer priemerných požiadaviek na V/V k priemeru požiadaviek na procesor. Externé priority sa nastavujú podľa kritérií, ktoré sú externé vzhľadom na OS, ako napr. dôležitosť procesu, alebo nejaké pracovné faktory, vyplývajúce z povahy nasadenia.

Prioritné plánovanie môže byť preemptívne alebo nepreemptívne. Keď proces príde do frontu pripravených procesov, jeho priorita sa porovnáva s prioritou bežiaceho procesu. Pri preemptívnom plánovaní proces bude prepnutý, ak priorita nového procesu je vyššia ako jeho. Pri nepreemptívnom plánovaní nový proces sa umiestni na začiatok frontu pripravených procesov.

Hlavný problém v prioritnom plánovaní je nekonečné blokovanie alebo *starvácia*. Proces, ktorý je pripravený na spustenie, ale nedostáva procesor, sa môže pokladať za blokovaný. Pri prioritnom plánovaní je možné, že v silne zaťaženom systéme niektoré procesy s nižšou prioritou budú čakať nekonečne dlho na pridelenie procesora. Obecne môžu nastať dva prípady: buď proces bude niekedy spustený alebo systém spadne a zrušia sa všetky nedokončené procesy s nižšou prioritou.

Riešením tohoto problému je postupné zvyšovanie priority procesov, ktoré dlho čakajú. Napr. ak máme interval priorít od 0 do 127, mohli by sme zvyšovať prioritu čakajúcich procesov každých 15 minút. V takom prípade proces, ktorý má počiatočnú prioritu dokonca 0, bude nakoniec mať najvyššiu prioritu a bude vykonaný, aj keď zvyšovanie priority zaberie viac ako 32 hodín!!

5.3.4 Cyklické plánovanie (Round Robin)

Algoritmus cyklického plánovania (Round Robin - RR) bol navrhnutý špeciálne pre time-sharing-ové systémy. Je podobný algoritmu FCFS, ale *je preemptívny*. Definuje sa malý časový úsek - *časové kvantum (q)*, ktoré je obyčajne od 10 do 100 *ms*. Front pripravených procesov sa spracováva ako cyklický front. Plánovač prideľuje postupne každému procesu z frontu jedno časové kvantum.

Cyklické plánovanie sa implementuje tak, že front pripravených procesov je typu FIFO. Nový proces sa pridáva na jeho koniec. Plánovač vyberá proces vždy zo začiatku frontu, nastavuje časovač na 1 časové kvantum a spúšťa proces.

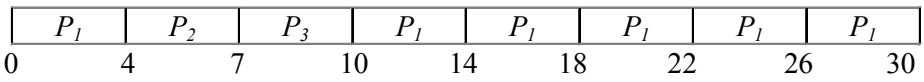
Ďalšia činnosť procesu môže byť nasledovná: proces môže potrebovať procesor na menší čas ako je časové kvantum a v takomto prípade uvoľní dobrovoľne procesor. Plánovač vyberie a spustí ďalší z pripravených procesov. Ak proces potrebuje čas dlhší ako je časové kvantum, po uplynutí kvanta časovač spôsobí prerušenie. Zapamätá sa kontext procesu a proces sa uloží na koniec frontu, z ktorého sa vyberie ďalší pripravený proces.

Priemerná doba čakania pri algoritme RR je niekedy dosť dlhá. Predpokladajme príchod nasledujúcich procesov v čase 0. Požadované doby spracovania každého procesu sú v *ms*.

Proces	Požadovaný čas procesora
P_1	24
P_2	3
P_3	3

$q = 4$ *ms*

Prvé časové kvantum dostane proces P_1, pretože tento proces potrebuje ďalších 20 *ms*, bude prerušený po uplynutí *q*. Potom sa spustí P_2, ale tento proces nespotrebuje celé časové kvantum. Ďalej sa bude spracovávať proces P_3. Keď každý z procesov dostane 1 časové kvantum, príde na rad znova proces P_1. Výsledné poradie vykonávania bude nasledujúce :



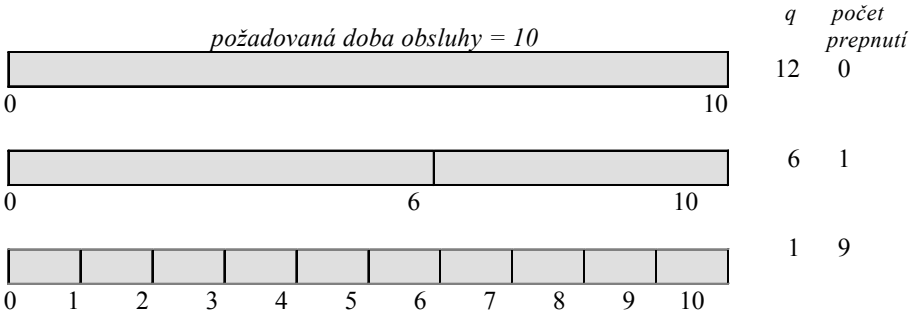
Priemerná doba čakania je 17/3 = 5.66 *ms*.

Algoritmus RR je preemptívny, ak proces potrebuje viac ako 1 časové kvantum, jeho vykonanie je prerušené, kontext zapamätaný a proces odložený na koniec frontu pripravených procesov.

Ak v systéme máme *n* procesov a prideľujeme časové kvantum *q*, potom každý proces dostáva *1/n* -tú časť z času procesora v dávkach najviac po 1*q*. Každý proces čaká nie viac ako *(n-1)xq* časových kvánt, kým príde znovu na rad.

Výkonnosť algoritmu RR silne závisí od veľkosti časového kvanta. V extrémnom prípade, keď *q* je nekonečne veľké, RR bude rovnocenný s FCFS. Ak *q* je veľmi malé, potom výsledný efekt (len teoreticky) by bol, ako keby proces mal pre seba procesor s rýchlosťou *1/n* z rýchlosti skutočného procesora, kde *n* je počet procesov.

V skutočnosti ale musíme zobrať do úvahy čas na prepnutie kontextu procesu. Predpokladajme, že máme len jeden proces s požadovanou dobou obsluhy 10 *ms*. Ak *q* = 12 *ms*, potom prepnutie nebude potrebné. Ak *q* = 6, bude potrebné jedno prepnutie a ak *q* = 1 *ms*, potom bude potrebných 9 prepnutí, čo náležite spomalí proces - pozri Obr 5.2. Čas vykonania tiež závisí od časového kvanta. Priemerná doba vykonania pre množinu procesov sa nemusí nutne skracovať s nárastom časového kvanta. Vo všeobecnosti priemerná doba vykonania sa môže zlepšiť, ak väčšina procesov ukončí svoje vykonanie behom jedného časového kvanta. Napr. ak máme 3 procesy a každý požaduje 10 časových jednotiek a *q* = 1, potom priemerný čas vykonania je 29. Ak *q* = 10, potom priemerná doba vykonania klesne na 20. Ak pridáme k tomu čas potrebný pre prepnutie kontextu, priemerná doba vykonania narastá pre menšie časové kvantum, pretože sa požaduje väčší počet prepnutí kontextu.



Obr. 5.2 Vplyv veľkosti časového kvanta na počet prepnutí

Keď zoberieme do úvahy predchádzajúce protichodné požiadavky, tak nám vychádza, že optimálny je prípad, keď 80% procesov končí svoje vykonanie behom jedného časového kvanta. Najčastejšie sa používa $q = 100\ ms$, čo znamená, že približne 10 - 30 % času je venovaných režijnej práci systému pre prepnutie kontextu.

5.3.5 Plánovanie s viacerými frontmi

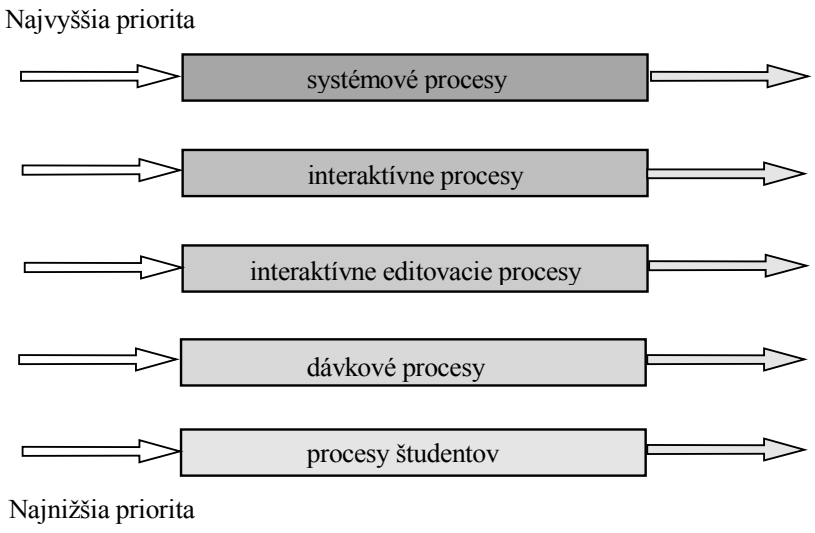
Táto trieda plánovacích algoritmov bola navrhnutá pre situácie, kedy sa procesy dajú ľahko rozdeliť na rôzne skupiny. Napríklad, veľmi často procesy v systéme sa delia na interaktívne a dávkové. Tieto dve skupiny majú odlišné požiadavky na čas odozvy a tiež môžu mať odlišné potreby plánovania.

Plánovanie s viacerými frontmi delí front pripravených procesov na niekoľko frontov, pozri Obr. 5.4. Procesy sú zaraďované do príslušného frontu podľa niektorej vlastnosti procesu - napr. veľkosť, priorita, typ procesu atď. Každý front má svoj plánovací algoritmus, napr. môžu byť použité oddelené fronty pre interaktívne procesy a pre procesy na pozadí. Interaktívne procesy by sa mohli plánovať podľa RR a procesy na pozadí podľa FCFS algoritmu.

Navyše sa tu musí vykonať aj plánovanie medzi frontmi, ktoré sa bežne implementuje ako preemptívne plánovanie s pevnými prioritami. Napr. procesy na popredí vždy majú vyššiu prioritu ako procesy na pozadí.

Na Obr. 5.4 je uvedený príklad plánovania s 5-timi frontmi:
1. Systémové procesy
2. Interaktívne procesy
3. Interaktívne editovacie procesy
4. Dávkové procesy
5. Procesy študentov

Každý front má absolútnu prioritu nad frontmi s nižšou prioritou. Napr. žiadny z dávkových procesov nemôže byť vykonaný, kým fronty pre systémové procesy, interaktívne procesy a interaktívne editovacie procesy nebudú prázdne.



Obr. 5.4 Plánovanie s viacerými frontmi

Iná alternatíva plánovania je rozdelenie času procesora medzi frontmi. Každý front dostáva určitú časť času procesora a delí ju medzi procesy vo fronte. Napr. ak máme dva fronty - pre procesy na popredí a pre procesy na pozadí, potom front procesov na popredí môže dostať 80% času procesora, ktorý môže rozdeliť medzi procesy na základe algoritmu RR, a front procesov na pozadí dostane 20% času procesora, ktorý môže rozdeliť medzi procesy na základe algoritmu FCFS.

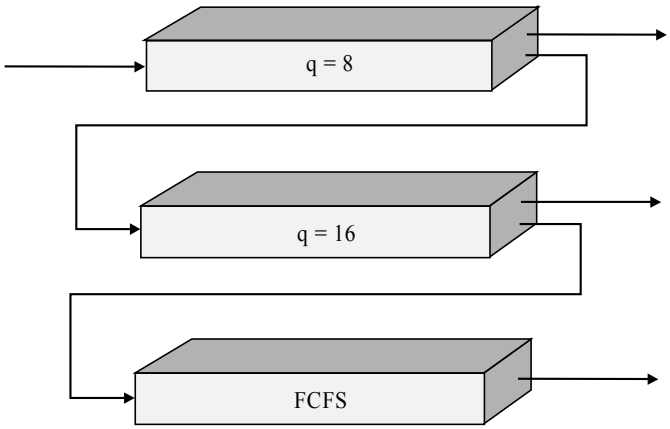
Pri plánovaní s viacerými frontmi sú procesy pevne spojené s jedným frontom a nemenia ho kým sa nedokončia. Je to dané tým, že rozdelenie procesov je urobené za základe charakteristiky procesu, ktorá zostáva nemenná. Tým sa získava výhoda nižšej réžie pri plánovaní, ale všeobecne tento algoritmus je málo flexibilný.

5.3.6 Plánovanie s viacerými frontmi so spätnou väzbou (Multilevel feedback)

Plánovanie s viacerými frontmi so spätnou väzbou dovoľuje procesom pohybovať sa medzi frontmi. Základná myšlienka je rozdeliť procesy podľa ich požiadaviek na cyklus procesora (pozri 5.1.1). Napr. predpokladajme plánovač, ktorý má 3 fronty - očíslované *0, 1, 2* (Obr. 5.6). Plánovač najskôr vykoná všetky procesy z frontu *0*. Keď bude front *0* prázdny, začnú sa vykonávať procesy z frontu *1* a potom procesy z frontu *2*. Ak medzitým príde proces do frontu *0*, preruší vykonanie procesu z frontu *1*. Podobne príchod procesu do frontu *1* preruší vykonanie procesu z frontu *2*.

Proces pripravený na vykonanie sa dáva do frontu *0*. Vo fronte *0* proces dostáva časové kvantum 8 *ms*. Ak sa proces nedokončí v tomto čase, presunie sa na koniec ďalšieho frontu. Ak front *0* je prázdny, vyberie sa na vykonanie prvý proces z frontu *1*. V tomto fronte procesy dostávajú časové kvantum 16 *ms*. Ak sa proces ani za tento čas nedokončí, presunie sa do frontu *2*, kde sa procesy vykonávajú podľa algoritmu FCFS, za predpokladu, že fronty *0* a *1* sú prázdne.

Tento algoritmus uprednostňuje procesy, ktoré majú čas cyklu CPU 8 *ms* alebo menej. Procesy, ktoré potrebujú viac ako 8 *ms*, ale menej ako 24 *ms* sú tiež obslúžené rýchlo, aj keď s menšou prioritou ako kratšie procesy. Dlhšie procesy automaticky klesnú do frontu *2*, kde sú obslúžené v poradí FCFS.



Obr. 5.6 Plánovanie s viacerými frontami so spätnou väzbou

Plánovač používajúci fronty so spätnou väzbou je definovaný pomocou týchto parametrov:

- počet frontov,
- plánovací algoritmus pre každý front,
- metóda, ktorá sa používa pre určenie momentu, kedy proces má byť presunutý do frontu s vyššou prioritou,
- metóda, ktorá sa používa pre určenie momentu, kedy proces má byť presunutý do frontu s nižšou prioritou,
- metóda, ktorá sa používa pre určenie frontu, do ktorého sa zaradí proces, ktorý potrebuje byť obslúžený.

Algoritmus plánovania používajúci fronty so spätnou väzbou je najuniverzálnejší plánovací algoritmus, ale i najzložitejší. Môže byť prispôsobený pre rôzne systémy, ale potrebuje starostlivý výber parametrov, aby sa docielilo optimálne plánovanie.

5.3.7 Porovnanie funkčných vlastností plánovacích algoritmov

Funkčné vlastnosti rôznych plánovacích algoritmov sú rozhodujúce pre výber jedného z nich pre plánovanie procesov. Avšak absolútne porovnanie vlastnosti nie je možné, pretože relatívny výkon konkrétneho plánovacieho algoritmu je závislý od mnohých rôznych faktorov ako napr. distribučná funkcia pravdepodobnosti požadovaných časov na obsluhu rôznych procesov, mechanizmus prepínania kontextu, povaha V/V požiadaviek, výkon V/V systému a ďalších.

V tabuľke 5.1 je uvedené porovnanie funkčných vlastností plánovacích algoritmov za predpokladu Poisson-ovského rozloženia príchodov nových procesov a exponenciálnych požadovaných časov obsluhy. Porovnané algoritmy **FCFS** (spracovanie v poradi príchodu), **RR** (cyklické plánomanie), **SJF** (nejkratší najskôr), **SRTF** (nejkratší zostávajúci čas) a **Multilevel feedback** (s viacerými frontami so spatnou väzbou).

	Proces	A	B	C	D	E	Stredná hodnota
	Čas Príchodu	0	2	4	6	8	
	Čas obsluhy (T_s)	3	6	4	5	2	
FCFS	Čas ukončenia	3	9	13	18	20	
	Čas vykonania(T_r)	3	7	9	12	12	**8,60**
	T_r/T_s	1,00	1,17	2,25	2,40	6,00	
RR, q=1	Čas ukončenia	4	18	17	20	15	
	Čas vykonania(T_r)	4	16	13	14	7	**10,80**
	T_r/T_s	1,33	2,67	3,25	2,80	,350	**2,71**
RR, q=4	Čas ukončenia	3	17	11	20	19	
	Čas vykonania(T_r)	3	15	7	14	11	**10,00**
	T_r/T_s	1,00	2,5	1,75	2,80	5,50	**2,71**
SJF	Čas ukončenia	3	9	15	20	11	
	Čas vykonania(T_r)	3	7	11	14	3	**7,60**
	T_r/T_s	1,00	1,17	2,75	2,80	1,50	**1,84**
SRTF	Čas ukončenia	3	15	8	20	10	
	Čas vykonania(T_r)	3	13	4	14	2	**7,20**
	T_r/T_s	1,00	2,17	1,00	2,80	1,00	**1,59**
Multilevel	Čas ukončenia	4	20	16	19	11	
feedback	Čas vykonania(T_r)	4	18	12	13	3	**10,00**
q=1	T_r/T_s	1,33	3,00	3,00	2,60	1,50	**2,29**
Multilevel	Čas ukončenia	4	17	18	20	14	
feedback	Čas vykonania(T_r)	4	15	14	14	6	**10,60**
$q=2^i$	T_r/T_s	1,33	2,50	3,50	2,80	3,00	**2,63**

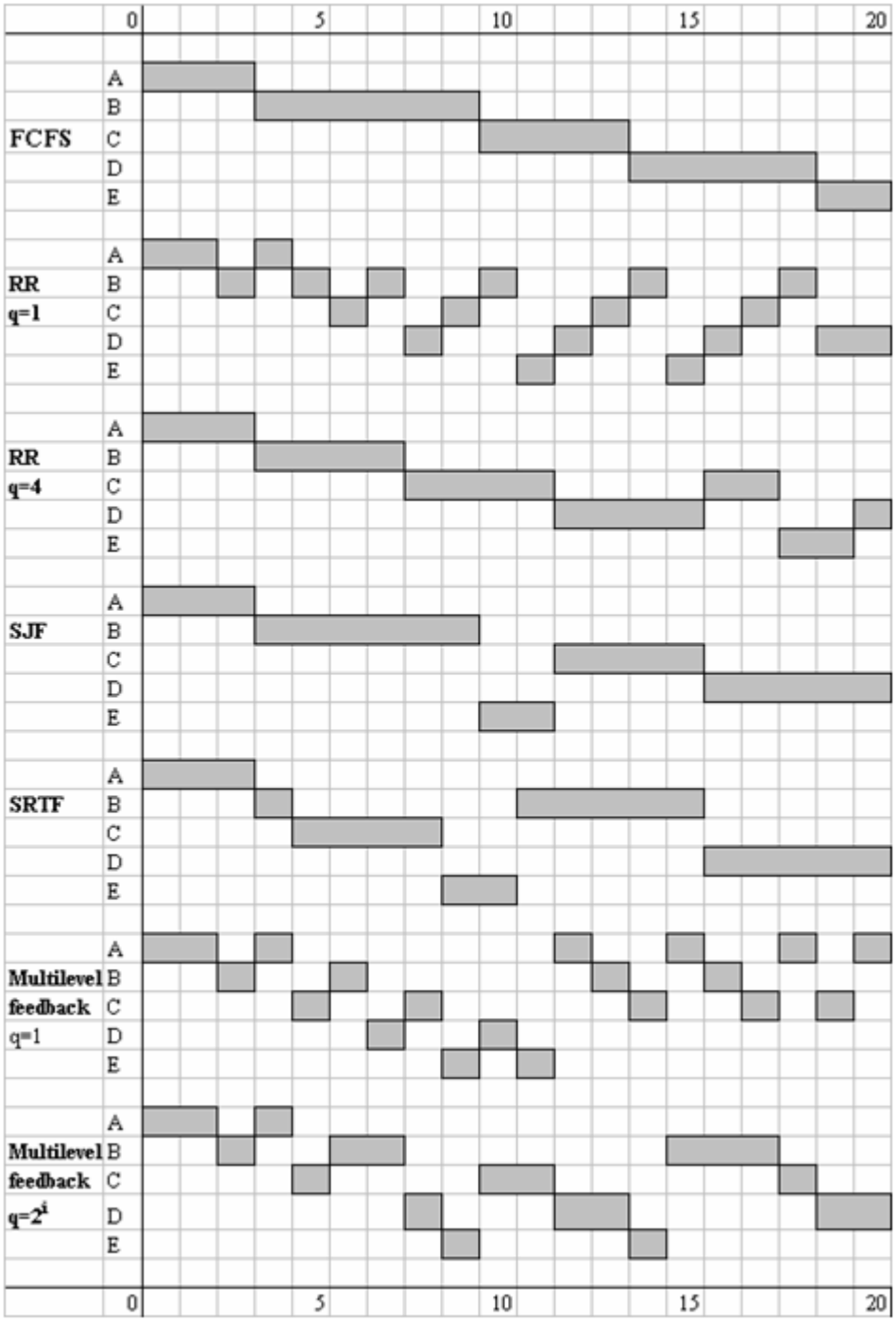
Tabuľka 5.1 Porovnanie funkčných vlastnosti plánovacích algoritmov

$\mathbf{T_s}$ - čas prechodu procesu cez systém; zahŕňa časy čakania a čas vykonania

$\mathbf{T_r}$ - priemerná doba obsluhy; priemerný čas, ktorý proces strávil v stave „bežiaci“

Multilevel Feedback, $q=2^i$ znamená, že čas ktorý strávil proces v *i*-tom fronte je 2^i

Gantov diagram, uvedený na Obr. 5.8 ukazuje pridelenie času procesora procesom podľa jednotlivých plánovacích algoritmov. Tabuľka 5.2 uvádza prehľad vlastností plánovacích algoritmov, podľa ktorých je možne ich porovnať.



Obr. 5.8 Porovnanie plánovacích algoritmov

Tab. 5.2 Porovnanie vlastností plánovacích algoritmov

	Funkcia pre výber	**Preempcia**	**Priepustnosť**	**Doba odozvy**	**Réžia**	**Vplyv na procesy**	**Starvácia**
FCFS	max [*w*]	nie	nie je výrazná	môže byť dlhá, ak požadované časy na obsluhu sú veľmi odlišné	minimálna	penalizuje krátke procesy a procesy, viazané na V/V	nie
RR	cyklicky	áno (po uplynutí *q*)	môže byť malá ak *q* je malé	dobrá pre krátke procesy	minimálna	rovnako spravodlivý ku všetkým procesom	nie
SJF	min[*s*]	nie	vysoká	dobrá pre krátke procesy	môže byť vysoká	penalizuje dlhé procesy	možná
SRTF	min[*s*-*e*]	áno (pri príchode)	vysoká	dobrá	môže byť vysoká	penalizuje dlhé procesy	možná
ML feedback	pozri text	áno (po uplynutí *q*)	nie je výrazná	nie je výrazná	môže byť vysoká	môže uprednostňovať procesy, viazané na V/V	možná

w čas, strávený v systéme do daného okamihu, t.j. čas vykonávania plus čas čakania
e čas strávený v stave „bežiaci“ do daného okamihu
s celková požadovaná doba obsluhy procesu

5.3.8 Plánovanie viacprocesorového systému

Plánovanie procesov v systéme s viacerými procesormi je zložitejšia úloha, ako u jednoprocesorového systému. Aj v tomto prípade boli vyskúšané viaceré algoritmy, ale žiadny nie je ideálny. V ďalšom texte uvedieme stručne problémy, sprevádzajúce plánovanie viacprocesorových systémov.

Procesory v systéme s viacerými procesormi sú väčšinou funkčne *identické* (*homogénne*). To znamená, že procesy môžu byť vykonávané na ľubovoľnom procesore. Ak sú procesory *neidentické* (*heterogénne*), znamená to, že proces môže byť vykonaný len na procesore, pre ktorého inštrukčný súbor bol skompilovaný. To je prípad niektorých distribuovaných systémov.

Pokiaľ máme k dispozícii niekoľko identických procesorov, môžeme použiť stratégiu *zdieľania zaťaženia* (*load sharing*). To znamená, že sa práca delí medzi jednotlivé procesory. Obyčajne sa udržuje jeden front pripravených procesov a keď sa niektorý procesor uvoľní, pridelí sa mu jeden proces z frontu. Podľa tejto schémy sa môžu uplatniť dva prístupy. Podľa prvého každý procesor sa sám stará o svoju prácu. Keď sa uvoľní, preskúma front pripravených procesov a vyberie si proces na vykonávanie. Táto úloha je zložitá, pretože potrebuje synchronizáciu prístupu k spoločným dátovým štruktúram. Druhý prístup je taký, že sa určí jeden procesor ako plánovací a on prideľuje procesom procesory.

Niektoré systémy riešia všetky problémy spojené s plánovaním, V/V a inými systémovými aktivitami tak, že jeden procesor je tzv. *master server.* Tento prípad *asymetrického multiprocesing-u* je jednoduchší ako *symetrický multiprocesing,* kedy sú procesory rovnocenné, lebo len jeden procesor pracuje so systémovými dátami, a tak odpadá potreba synchronizovať k ním prístup.

5.3.9 Plánovanie systémov reálneho času

Systémy reálneho času sa delia na dve skupiny. ***Systémy s pevnými termínmi ukončenia (hard real-time*)** sú také systémy, kde sa požaduje ukončenie zadanej úlohy do určitého pevne stanoveného termínu. Obyčajne sa proces dodáva aj s termínom ukončenia. Plánovač v takomto prípade buď prijme proces a garantuje, že bude ukončený do stanoveného termínu, alebo ho odmietne ako nesplniteľný. Garancia vyžaduje, aby plánovač presne vedel, koľko času vyžaduje určitá systémová funkcia. Garancia nie je možná v systémoch s virtuálnou alebo sekundárnou pamäťou. V týchto prípadoch sa nedá predvídať čas, za ktorý sa vykoná určitý proces. Takže systémy reálneho času sa skladajú zo špeciálnych programov, ktoré bežia na platformách navrhnutých pre príslušné požiadavky systému. Týmto systémom chýba univerzálnosť a plná funkčnosť moderných operačných systémov.

***Systémy s variabilnými termínmi ukončenia (soft real-time*)** nie sú tak obmedzujúce ako tie s pevnými termínmi ukončenia. U týchto systémov sa požaduje, aby kritické procesy dostali vyššiu prioritu ako ostatné procesy. Ak bežný time-sharig-ový systém má aj tieto možnosti, môže sa stať, že prideľovanie prostriedkov bude nespravodlivé a spôsobí spomalenie alebo starváciu niektorých procesov. Ale sú aj prípady, kedy univerzálny systém potrebuje spúšťať náročnejšie úlohy ako sú multimediálne aplikácie alebo vysokorýchlostná interaktívna grafika, ktoré by nepracovali správne v prostredí, ktoré nemá tieto vlastnosti.

Implementácia funkcií pre reálny čas vyžaduje starostlivý návrh plánovača a adekvátne vlastnosti operačného systému. Po prvé, systém musí používať prioritné plánovanie, pričom procesy reálneho času musia mať najvyššiu prioritu a tá nesmie klesať, aj keď obyčajným procesom by sa priorita menila. Po druhé, čas reakcie dispečera musí byť krátky. Čím je čas reakcie dispečera menší, tým rýchlejšie môže byť odštartovaný proces reálneho času. Mnoho operačných systémov, vrátane väčšiny verzií Unix-u, nemôže zaistiť dostatočne malý čas reakcie dispečera, pretože pred prepnutím kontextu procesu musia čakať, ak sa vykonáva V/V operácia alebo systémové volanie. V týchto prípadoch oneskorenie dispečera je veľké, lebo niektoré systémové volania sú zložité a V/V zariadenia sú pomalé.

Riešenie týchto problémov je možné niekoľkými spôsobmi: napr. dovoliť preempciu systémových volaní alebo navrhnúť celé jadro ako prerušiteľné! Posledne spomenutá metóda je použitá v systéme Solaris 2.

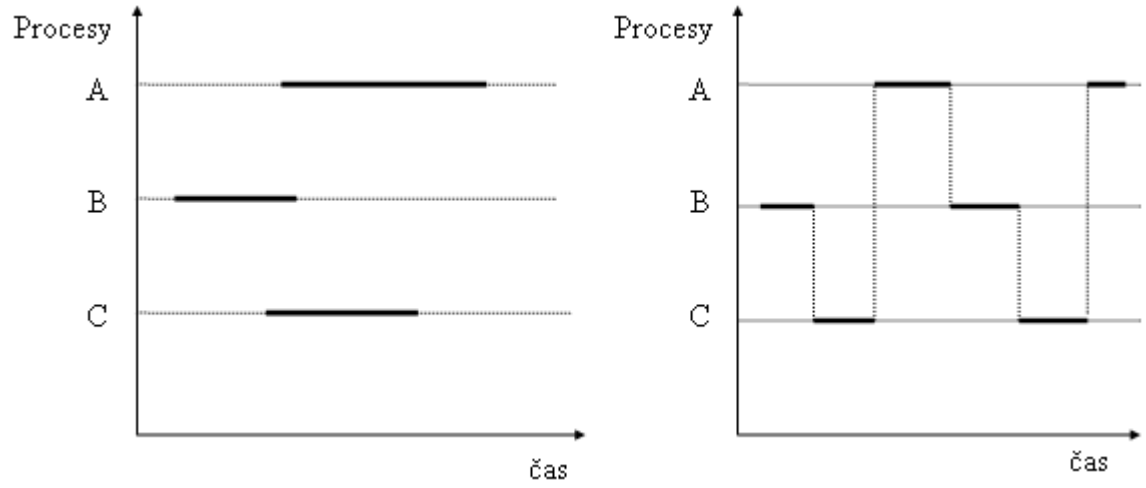
6 SYNCHRONIZÁCIA PROCESOV

Pri svojej spolupráci procesy často zdieľajú prostriedky. Súbežný prístup k dátam alebo prostriedkom môže viesť k nekonzistencii alebo k nesprávnemu použitiu. V tejto kapitole sa budeme zaoberať rôznymi mechanizmami pre synchronizáciu prístupu k zdieľaným prostriedkom, aby bola zaistená ich konzistencia.

6.1 Základné pojmy paralelných procesov

Proces je program, ktorý sa vykonáva a je základnou jednotkou práce operačného systému. Na rozdiel od programu, ktorý je statický, proces je dynamický prvok, pretože vzniká, mení svoj stav a zaniká.

Proces je obyčajne sekvenčný a pozostáva z postupnosti inštrukcií, ktoré sa vykonávajú za sebou. Keď množina procesov beží na jednoprocesorovom systéme a zdieľa čas procesora, hovoríme o *paralelných sekvenčných procesoch* alebo o *pseudoparalelných procesoch*. Ak systém disponuje viacerými procesormi, potom procesy môžu skutočne bežať *paralelne*. Na Obr. 6.1 sú znázornené paralelné a pseudo-paralelné procesy.



Obr. 6.1 a) paralelné procesy b) pseudoparalelné procesy

Procesy, ktoré existujú súčasne v systéme, si môžu uvedomovať existenciu ďalších procesov v rôznej miere.

Proces je ***nezávislý***, ak nemôže byť ovplyvnený a nemôže ovplyvniť iné procesy v systéme. Nezávislý proces je charakteristický tým, že jeho stav nie je zdieľaný žiadnym iným procesom, nemá spoločné prostriedky s inými procesmi, ale súperí o prostriedky systému ako napr. tlačiareň, disk atď. Vykonávanie nezávislého procesu je deterministické a opakovateľné, t.j. výsledok vykonávania je závislý len na vstupnom stave a výsledok vykonávania je vždy rovnaký pre rovnaký vstup. Potencionálne problémy pre tieto procesy sú: vzájomné vylúčenie, uviaznutie a starvácia.

Kooperujúci proces je taký, ktorý môže byť ovplyvnený inými procesmi v systéme, alebo on ich môže ovplyvniť. Kooperujúci proces zdieľa nejaké spoločné prostriedky s inými procesmi a jeho vykonávanie je nedeterministické - s rovnakými vstupnými hodnotami môžeme pri rôznych spusteniach dosahovať rôzne výsledky. Výsledok je neopakovateľný, pretože závisí na relatívnej rýchlosti procesov. Spolupracujúce procesy môžu buď iba zdieľať prostriedky, pričom si neuvedomujú jeden druhého na úrovni identifikačných čísel (PID), alebo môžu spolu komunikovať, kde znalosť PID partnera je potrebná.

6.2 Časová závislosť

Súčasný prístup k prostriedkom systému môže viesť k nekonzistencii dát, čo je neprípustné. Uvedieme príklad.

Príklad:

pozorovateľ
úloha: pozorovať (merať) a evidovať počet pozorovaní.

```
repeat
    Pozorovanie;      1.
    C := C + 1;       2.
until koniec;
```

reportér
úloha: Má v určitých časových úsekoch vypísať počet pozorovaní za daný časový interval.

```
repeat
    if (je čas) then
        begin
         writeln(C);      3.
         C := 0;          4.
        end;
```

Procesy pozorovateľa a reportéra bežia paralelne ako samostatné procesy. Pretože zdieľajú jeden procesor, ich vykonanie môže byť hocikedy prerušené. To znamená, že udalosti 1, 2, 3 a 4 sa môžu vyskytnúť v rôznom poradí, napr. 1, 3, 2, 4. Pri tomto poradí vykonania sa stráca jedno pozorovanie, čo znamená že vznikla *časovo-závislá* chyba. Takéto chyby sa prejavujú len niekedy a závisia od času vykonania procesu.

O *časovej závislosti* procesov (*race condition*) hovoríme v prípadoch, keď výsledok priebehu dvoch alebo viacerých procesov závisí od relatívnej rýchlosti ich vykonania (na poradí, v akom im bol prideľovaný procesor). Časová závislosť je výsledkom explicitného alebo implicitného zdieľania dátových štruktúr alebo periférnych zariadení dvomi alebo viacerými procesmi. Pre riešenie problému časovej závislosti operačný systém musí poskytnúť mechanizmy pre synchronizáciu vykonania paralelných procesov.

6.3 Všeobecné pojmy synchronizácie

Predpokladajme, že máme systém pozostávajúci z množiny procesov $\{P_0, P_1, \ldots, P_{n-1}\}$. Každý proces má vo svojom kóde sekciu, nazvaná *kritická sekcia*, v ktorej môže používať zdieľané prostriedky systému alebo modifikovať zdieľanú informáciu (spoločné premenné, tabuľky, zdieľané súbory a iné). Operačný systém musí zaistiť, že kým jeden proces sa nachádza vo svojej kritickej sekcii, žiadny iný nemá prístup k svojej, v ktorej pracuje s tými istými spoločnými prostriedkami. To znamená, že je potrebná metóda *vzájomného vylučovania* prístupu procesov ku kritickej sekcii. Podstatou tejto metódy je navrhnúť protokol, ktorý budú procesy používať pri požiadavke vstupu do kritickej sekcie, t.j. procesu musí byť povolený vstup. Pre jednoduchosť pri ďalšom vysvetľovaní zavedieme tieto názvy: kód, ktorý obsahuje požiadavku pre vstup do kritickej sekcie budeme nazývať *vstup do kritickej sekcie (KS)*, kód ktorý nasleduje po kritickej sekcii budeme nazývať *výstup z kritickej sekcie (KS)* a zostavajúci kód procesu budeme nazývať *zostávajúca sekcia.*

```
repeat
    Vstup do KS
        kritická sekcia
    Výstup z KS
        zostávajúca sekcia
until false;
```

Riešenie problému kritickej sekcie musí vyhovovať nasledovným podmienkam:

1. **Vzájomné vylúčenie**: ak proces P_i vykonáva svoju kritickú sekciu, žiadny iný proces nesmie vstúpiť do svojej kritickej sekcie.

2. **Postup**: Ak je viac požiadaviek na vstup do kritickej sekcie naraz, jeden z procesov musí dostať povolenie na vstup do kritickej sekcie. Výber procesu, ktorý dostane prístup nesmie trvať nekonečne dlho.
3. **Spravodlivosť**: žiadny proces nesmie nekonečne dlho brániť iným procesom vstúpiť do kritickej sekcie, t.j. každý proces musí mať rovnakú šancu vstúpiť do nej.

Synchronizačné prostriedky vyšších úrovni sú budované na základe atomických operácií. ***Atomická*** operácia je taká operácia, ktorá nemôže byť prerušená, musí sa vykonať len celá. Takými operáciami sú napr. operácie čítania a zápisu, ktoré sú atomické a vzájomné vylúčené.

6.4 Princípy pri synchronizácií

Úlohou synchronizácie je zaistiť vzájomné vylúčenie paralelných procesov, ktoré využívajú zdieľané prostriedky. Prakticky to znamená, že sa rýchlosti procesov musia zosúladiť tak, aby sa časy vykonania ich kritických sekcií neprekrývali. Pri tom sa uplatňujú dva základné princípy: synchronizácia *aktívnym* čakaním a synchronizácia *pasívnym* čakaním.

Synchronizácia ***aktívnym čakaním*** znamená, že sa odsun kritickej sekcie uskutoční vložením pomocných (obyčajne prázdnych) inštrukcií do kódu procesu.

Synchronizácia ***pasívnym čakaním*** znamená, že sa odsun kritickej sekcie uskutoční dočasným pozastavením procesu, kým sa kritická sekcia neuvoľní.

6.5 Prostriedky pre synchronizáciu aktívnym čakaním

6.5.1 Spoločné premenné

V tejto časti popíšeme algoritmy, ktoré pre synchronizáciu vstupu do kritickej oblasti používajú len spoločné premenné. To znamená, že pre serializáciu prístupu k spoločným premenným nie je použitý žiadny iný prostriedok.

Najskôr uvedieme niekoľko algoritmov **pre dva procesy.** K správnemu riešeniu sa dostaneme postupne, pričom pomocou jednotlivých algoritmov poukážeme na niektoré chyby, ktoré sa často vyskytujú pri vývoji paralelných programov.

Procesy sú očíslované P_0 a P_1. Pre jednoduchosť budeme prezentovať algoritmus pre proces P_i a pomocou P_j budeme označovať druhý proces.

Algoritmus č.1

V tomto algoritme procesy používajú jednu spoločnú celočíselnú premennú *turn,* ktorá nadobúda hodnoty 0 alebo1. Na začiatku *turn* má hodnotu 0 (alebo 1). Ďalej je uvedený kód procesu P_i . Kód procesu P_j je analogický.

repeat

```
while (turn <> i)  do  no-op;
```

kritická sekcia

```
turn = j;
```

zostávajúca sekcia

Toto riešenie zabezpečuje, že v danom okamihu bude mať povolený vstup do kritickej sekcie len jeden proces - zaisťuje vzájomné vylúčenie procesov v kritickej sekcii. Nedostatkom tohto algoritmu je striktné striedanie procesov vo vykonávaní kritickej sekcie, pričom chod procesov diktuje pomalší z nich. Tým sa tento algoritmus javí ako konzervatívny.

Algoritmus č.2

V predchádzajúcom algoritme každý proces si pamätal len informáciu o tom, ktorý proces má povolený vstup do KS. Ďalší algoritmus používa pole *flag*, ktoré je definované nasledovne:

var *flag:* ***array*** *[0..1]* ***of*** *boolean;*

Prvky poľa *flag* na začiatku sú inicializované na *false*.

repeat

while *flag[j]* ***do*** *no-op;*
flag[i] := true;

kritická sekcia

flag[i] := false;

zostávajúca sekcia

until*false;*

Toto riešenie je horšie ako predchádzajúce, pretože nezaručuje vzájomne vylúčenie. Ak prepnutie procesov nastane predtým ako *i-tý* proces zmení *flag[i]* na *true,* potom *j-tý* proces môže tiež vstúpiť do kritickej sekcie.

Algoritmus č.3

Prvky poľa *flag* sú inicializované na *false*. Ak *flag[i]* je *true,* to znamená, že proces P_i je pripravený vstúpiť do KS.

repeat

flag[i] := true;
while *flag[j]* ***do*** *no-op;*

kritická sekcia

flag[i] := false;

zostávajúca sekcia

V tomto algoritme proces P_i najskôr nastavuje *flag[i]* na hodnotu *true*, čím naznačuje svoj záujem vstúpiť do KS. Potom P_i skontroluje či proces P_j nie je tiež pripravený vstúpiť do KS. Ak P_j je pripravený, P_i musí počkať kým P_j nenaznačí, že skončil svoju prácu v KS. Potom P_i môže vstúpiť do KS. Po skončení práce v KS proces P_i nastaví *flag[i]* na hodnotu *false,* a tak dovolí ďalšiemu procesu vstup do KS.

Toto riešenie spĺňa podmienku vzájomného vylúčenia, ale tu môže nastať situácia, kedy sa procesy ocitnú v nekonečnej čakacej slučke. Je to situácia, keď prepnutie procesov nastane po nastavení vlastného *flagu* na *true* t.j.:

P_0 nastaví *flag[0] := true;*
P_1 nastaví *flag[1] := true;*

Potom každý z nich bude čakať nekonečne dlho na vstup do KS.

Dekkerov algoritmus

Prvý vyriešil problém kritickej sekcie pre dva procesy holandský matematik Dekker. Procesy zdieľajú dve premenné:

```
var flag: array [0..1] of boolean;
    turn: 0..1;
```

Na začiatku pole *flag* je inicializované na *false*. Hodnota premennej *turn* je ľubovoľná (0 alebo 1).

```
repeat
    flag[i] := true;
    while flag[j] do
        if turn= j then
            begin
                flag[i] := false;
                while turn=j do no-op;
                flag[i] := true;
            end;
        kritická sekcia
    turn := j;
    flag[i] :=false;
```

Pri požiadavke vstupu do KS P_i nastaví *flag[i]* na *true*. Potom testuje, či druhý proces deklaroval svoj záujem o prácu a či sa nachádza v KS. Ak áno, nastaví svoj *flag[i]* na *false* a čaká v slučke, kým sa *turn* zmení na jeho číslo. Po ukončení práce v KS nastaví *turn* na číslo druhého procesu a svoj *flag[i]* na *false*.

Algoritmus pekára

Ďalej uvedieme riešenie problému kritickej sekcie **pre *n* procesov**. Tento algoritmus je známy pod názvom *algoritmus pekára* (*bakery algoritmus*),

Obr. 6.2. Názov vyplýva zo spôsobu predaja v pekárňach a mäsiarstvach (na západe), kde každý zákazník pri vstupe do obchodu dostane číslo. Pri obsluhe sa zachováva poradie príchodu zákazníkov. Tento algoritmus bol vyvinutý pre distribuované prostredie, ale tu budeme brať do úvahy aspekty týkajúce sa centralizovaného systému.

Pri vstupe do obchodu zákazník dostane číslo. Zákazník s najmenším číslom je obslúžený ako prvý. Tento algoritmus nemôže zaistiť, že dva procesy nedostanú rovnaké čísla. Ak nastane takýto prípad, obslúži sa ako prvý proces s menším menom. To znamená, že ak P_i a P_j dostanú rovnaké čísla a i<j, potom ako prvý sa obslúži P_i. Pretože názvy procesov sú jedinečné, algoritmus je deterministický. Spoločné dátové štruktúry sú:

```
var choosing: array [0..n-1] of boolean;     { inicializované na false }
    number: array [0..n-1] of integer;       { inicializované na 0 }
```

Pre jednoduchosť definujeme nasleduj[ce pravidlá:

- *(a,b) < (c,d), ak a<c alebo ak a=c* a *b<d.*
- *max(a_0 ,..., a_{n-1}) je také číslo k , pre ktoré platí k≥ a_i pre i=0, ... ,n-1.*

Štruktúra *i*-tého procesu je ukázaná ďalej.

```
repeat
   choosing[i]:= true;
   number[i]:= max(number[0],number[1],...,number[n  -1])+1;
   choosing[i]:=false;
   for j:=1 to n-1 do
    begin
          while choosing[j]  do no-op;
          while (number[j]<>0)  and ( (number[j],j)<number[i],i))  do no-op;
    end;
          kritická sekcia
   number[i]:= 0;
          zostávajúca sekcia
until false;
```

Obr. 6.2 Štruktúra i-tého procesu v algoritme pekára

6.5.2 Hardvérové prostriedky pre synchronizáciu aktívnym čakaním

V tejto časti popíšeme niektoré inštrukcie základného inštrukčného súboru, ktoré uľahčujú programovanie synchronizačných úloh a zlepšujú efektívnosť systému.

Problém kritickej sekcie v jednoprocesorovom systéme môže byť vyriešený jednoduchým ***zákazom prerušenia*** počas modifikácie hodnoty spoločnej premennej. V tomto prípade budeme mať istotu, že postupnosť inštrukcií pre modifikáciu prebehne bez preempcie (prerušenia vykonania) - to znamená, že sa nevyskytnú neočakávané modifikácie spoločnej premennej.

Toto riešenie má niekoľko nedostatkov. Prvý je, že dovoliť používateľovi používať tak silný prostriedok akým je zákaz prerušenia je nebezpečné pre systém, pretože musíme spoliehať na jeho korektné použitie. V prípade nesprávneho sledu zákazu a povolenia prerušenia môže byť ohrozený plynulý chod systému.

Druhým nedostatokom je, že toto riešenie sa nehodí pre multiprocesorové systémy. Zákaz prerušenia pre všetky procesory by spomalil celý systém a ešte by priniesol dodatočné problémy s hodinami, ak sa tieto aktualizujú pomocou prerušení.

Mnoho procesorov ponúka ***špeciálne inštrukcie***, ktoré dovoľujú používateľovi testovať a modifikovať obsah premennej alebo vymieňať atomicky obsah dvoch premenných. Tieto inštrukcie sú *atomické,* čo znamená, že sa vykonajú vždy ako celok. Ak sa vykonávajú paralelne dve takéto inštrukcie, vždy sa vykonajú sekvenčne v ľubovoľnom poradí.

6.5.3 Inštrukcia *Test-and-Set*

Táto inštrukcia je definovaná nasledovne:

```
function Test-and-Set ( var target: boolean): boolean;
 begin
       Test-and-Set:= target;
       target:= true ;
end ;
```

Použitie tejto inštrukcie môžeme ilustrovať v nasledujúcom príklade (Obr. 6.3). Ak dva procesy potrebujú zaistiť vzájomné vylúčenie v KS, môžu použiť spoločnú premennú *lock,* nastavenú na

začiatku na *false*. Nastavenie hodnoty premennej *lock* prebehne vždy atomicky a vždy len jeden proces dostane povolenie vstúpiť do KS.

```
repeat

    while Test-and-Set(lock) do no-op;

        kritická sekcia

    lock:= false ;

until false ;
```

Obr. 6.1 Požitie inštrukcie Test-and-Set pre dosiahnutie vzájomného vylúčenia

6.5.4 Inštrukcia *Swap*

Táto inštrukcia zaisťuje atomickú výmenu obsahu dvoch premenných a je definovaná nasledovne:

```
procedure Swap ( var a,b: boolean);
  var temp: boolean;
    begin
        temp:= a;
        a:= b;
        b:= temp ;
```

Použitie tejto inštrukcie môžeme ilustrovať v príklade z Obr. 6.4. Ak dva procesy potrebujú zaistiť vzájomné vylúčenie v KS, môžu použiť jednu spoločnú premennú *lock,* nastavenú na začiatku na *false* a jednu lokálnu premennú *key*.

```
repeat
    key:= true;
    repeat
            Swap(lock,key);
    until key=false;
        kritická sekcia

    lock:= false;
        zostávajúca sekcia
until false ;
```

Obr. 6.4 Použitie inštrukcie Swap

Využitie špeciálnych inštrukcií pre zaistenie vzájomného vylúčenia má veľa výhod: dá sa použiť aj pre jednoprocesorový systém, aj pre viacprocesorový, je jednoduché a ľahko sa overuje. Špeciálne inštrukcie sa môžu použiť aj pre viac kritických sekcií, pričom pre každú kritickú sekciu bude definovaná vlastná premenná.

Na druhej strane špeciálne inštrukcie majú aj veľa nedostatkov: sú postavené na princípe aktívneho čakania, čím sa plytvá čas procesora, nie je vylúčená starvácia procesov pri vstupe do kritickej sekcie a takisto hrozí nebezpečenstvo uviaznutia.

6.6 Prostriedky pre synchronizáciu pasívnym čakaním

6.6.1 Semafory

Semafor je synchronizačný prostriedok, ktorý navrhol holandský vedec E.W. Dijkstra v polovici 60-tych rokov. **Semafor je abstraktný dátový typ, ktorý je charakterizovaný svojou hodnotou a operáciami, ktoré sú definované nad ním**. Tieto operácie sú **atomické** a jedine pomocou nich sa môže modifikovať hodnota semafora. Tieto operácie Dijkstra nazval ***P*** (od flámskeho slova *proberen* - testovať) a ***V*** (od flámskeho slova *verhogen* - zvýšiť hodnotu). Klasické názvy pre tieto operácie, zavedené v literatúre sú *wait* a *signal*. Ich sémantika je nasledovná:

wait(S): **while** S≤ 0 **do** no-op;

S:=S - 1;

signal(S): S :=S + 1;

Modifikácie hodnoty semafora sú vykonávané atomicky - to znamená, že kým jeden proces modifikuje hodnotu semafora, žiadny iný proces to nemôže robiť súčasne. Aj testovanie hodnoty semafora (operácia *wait*) sa vykonáva atomicky.

Semafor, ktorý sme popísali sa často nazýva *spinlock* . Tento typ semafora je založený na aktívnom čakaní. Spinlock sa hodí do viacprocesorových systémov, kde je potrebné vyriešiť vzájomné vylúčenie pre krátke časové intervaly a prepínanie kontextu by zabralo veľa času.

6.6.2 Použitie semaforov

Semafory sa dajú použiť pre vyriešenie problému kritickej sekcie **pre *n* procesov**. Procesy zdieľajú jeden semafor - *mutex* (z anglického mutual exclusion - vzájomné vylúčenie), ktorý je inicializovaný na 1. Spôsob použitia semafora pre každý proces je ukázaný na Obr. 6.5.

repeat

wait(mutex);

kritická sekcia

signal(mutex) ;

zostávajúca sekcia

Obr. 6.5 Vzájomné vylúčenie pomocou semaforov

6.6.3 Implementácia semaforov

Hlavný nedostatok synchronizácie pomocou spoločných premenných a semaforov (spinlock-ov) ako bolo vyššie uvedené je, že vyžadujú aktívne čakanie procesu v prípade, že sa ten nemôže dostať do kritickej sekcie. V multiprogramových systémoch je toto vážny nedostatok, pretože proces, ktorý prakticky nerobí nič, dostáva pridelený čas procesora, pričom iné procesy ho môžu využiť efektívnejšie.

Pre prekonanie nedostatkov synchronizácie aktívnym čakaním je možné modifikovať operácie *wait* a *signal*. Keď proces vykonáva operáciu *wait* a zistí, že hodnota semafora nie je kladná, musí čakať. Avšak proces nečaká aktívne, ale zablokuje sa. Proces je umiestnený do frontu, pridružený

k semaforu a stav procesu je zmenený na čakajúci - t.j. proces nebude dostávať pridelený čas procesora, kým nebude znova v stave pripravený.

Proces, ktorý čaká vo fronte semafora, môže byť odblokovaný keď niektorý iný proces vykoná nad týmto semaforom operáciu *signal*. Proces sa reštartuje znova uvedením do stavu pripravený.

Aby sme implementovali semafor podľa tejto definície, musíme ho definovať ako záznam:

```
type semaphore = record
                    value: integer;      { hodnota semafora}
                    L: list of process;  { zoznam procesov}
```

Každý semafor má celočíselnú hodnotu a zoznam čakajúcich procesov. Keď ďalší proces musí čakať vo fronte semafora, zaradí sa do zoznamu. Keď niektorý proces vykoná operáciu *signal* nad týmto semaforom, vyberie sa zo zoznamu jeden proces a uvedie sa do stavu pripravený. V tomto prípade operácie nad semaformi sú definované nasledovne:

```
wait(S):
        S.value := S.value - 1;
        if S.value < 0 then
        begin
            pridaj proces do S.L;
            zablokuj volajúci proces ;

signal(S):  S.value := S.value +1;
            if S.value ≤ 0 then
              begin
                odstráň proces P z S.L;
                odblokuj proces P;
              end;
```

Operácia zablokovania zastaví proces, ktorý ju volal, a operácia odblokovania procesu uvedie do stavu pripravený jeden z procesov, čakajúcich vo fronte semafora.

V tejto implementácii sa objavuje rozdiel oproti klasickej definícii semafora, a to, že tu semafor môže nadobúdať aj záporné hodnoty. Ak hodnota semafora je záporná, jeho absolútna hodnota je vlastne počet procesov, čakajúcich na zvýšenie hodnoty semafora.

Zoznam čakajúcich procesov môže byť ľahko implementovaný ako zreťazený zoznam riadiacich blokov procesov. Jeden zo spôsobov spracovania tohto zoznamu je front FIFO, kde semafor má ukazovatele začiatku a konca frontu. Vo všeobecnosti zoznam môže využívať ľubovoľný typ frontu.

Semafory popísané v predchádzajúcom texte sú známe ako *počítajúce* (*counting*) semafory, pretože ich hodnota sa môže meniť neobmedzene.

6.6.4 Uviaznutie a starvácia

Implementácia semaforov pomocou frontu čakajúcich procesov môže niekedy viesť k situácii, kedy dva alebo viacej procesov čakajú nekonečne dlho na udalosť, ktorú môže vyvolať len jeden z čakajúcich procesov. Táto udalosť je operácia *signal* a takáto situácia sa nazýva *uviaznutie*.

Pre ilustráciu tejto situácie si predstavme systém s dvomi procesmi - P_0 a P_1, ktoré používajú dva semafory S a Q, nastavené na hodnotu 1:

P_0	P_1
wait(S);	*wait(Q);*
wait(Q);	*wait(S) ;*
.	.
.	.
.	.
signal(S);	*signal(Q);*

Predpokladajme, že P_0 vykonal operáciu *wait(S)* a potom P_1 vykonal operáciu *wait(Q)*. Keď P_0 vykoná *wait(Q),* bude musieť počkať, kým P_1 vykoná *signal(Q).* Podobne keď P_1 vykoná *wait(S),* bude musieť počkať, kým P_0 vykoná *signal(S).* Pretože tieto operácie sa nemôžu vykonať, P_0 a P_1 uviaznu. Metódy riešenia uviaznutia budú rozobraté v ďalších častiach.

Iný problém, ktorý je podobný uviaznutiu, je situácia (kedy proces je trvale blokovaný alebo tzv. *starvuje*), ktorá nastáva, keď proces čaká nekonečne dlho, pričom je pripravený pokračovať v práci. Tento problém sa môže vyskytnúť, ak vyberáme procesy z frontu podľa stratégie LIFO.

6.6.5 Binárne semafory

Binárny semafor nadobúda hodnoty len 0 alebo 1. Implementácia binárneho semafora je niekedy jednoduchšia ako implementácia počítajúceho semafora, podľa toho, pre akú HW platformu sa navrhuje.

Implementácia operácií *wait a signal* pre binárny semafor je ukázaná na Obr. 6.6.

```
wait(S):   if S.value = 1 then
            s.value = 0
           else
             begin
                 pridaj proces do S.L;
                 zablokuj  volajúci proces ;
              end;

signal(S): if (S.L je prázdny ) then
              S.value = 1
           else
             begin
                odstráň proces P z S.L;
                odblokuj  proces P;
             end;
```

Obr. 6.2 Definícia operácií wait a signal pre binárny semafor

6.6.6 Monitory

Ďalší synchronizačný prostriedok vyššej úrovne je *monitor.* Monitor je abstraktný dátový typ, ktorý je charakterizovaný zdieľanými dátami a množinou operácií, ktoré sú definované nad tými dátami. Typ monitor pozostáva z deklarácií premenných, ktorých hodnoty definujú stav monitora a z množiny procedúr a funkcií, ktoré implementujú operácie nad monitorom.

```
type monitor-name = monitor
        deklarácia premenných

        procedure entry P1(....);
          begin ... end;

        procedure entry P2(...);
          begin ... end;
                .
                .
                .
        procedure entry Pn(....);
           begin ... end;

          begin
             inicializácia premenných
          end.
```

Obr. 6.7 Syntax monitora

Monitory predstavujú vyšší stupeň abstrakcie ako semafory a sú jednoduchšie a bezpečnejšie pre použitie. Syntax monitora môžeme prezentovať tak, ako je uvedené na Obr. 6.7.

Existujúce implementácie monitorov sú vložené do programovacích jazykov. Najlepšia existujúca implementácia monitora je v jazyku Mesa/Cedar firmy Xerox. V jazyku Java napr. monitor je implementovaný nepriamo tak, že metódy, ktoré pracujú so zdieľanými prostriedkami sú označené kľúčovým slovom *synchronized,* čo znamená, že danú metódu môže vykonávať len jeden proces v danom čase.

Procesy, ktoré využívajú monitor, majú prístup len k procedúram monitora. Procedúry definované v monitore majú prístup len k premenným, ktoré sú definované vo vnútri monitora a k formálnym parametrom. Tieto procedúry sa nesmú vzájomne volať a nesmú byť rekurzívne.

Konštrukcia monitora dovoľuje len jednému procesu byť vo vnútri, takže programátor nemusí explicitne programovať vzájomné vylúčenie vykonania jednotlivých procedúr monitora (pozri Obr. 6.8).

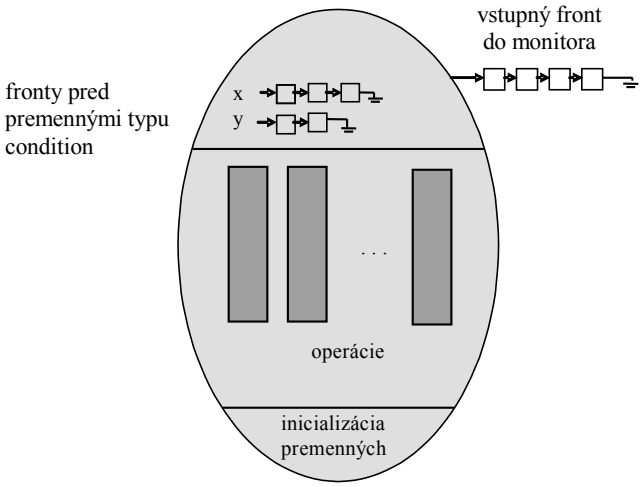
Táto konštrukcia ale nedovoľuje modelovať určité situácie, ktoré sa vyskytujú v synchronizačných úlohách. Vzniká potreba definovať dodatočný dátový typ, ktorý dovoľuje zablokovať proces v prípade, že nie je splnená určitá podmienka. Tento typ sa nazýva *condition* a jediné operácie nad ním sú *wait* a *signal*.

```
var x,y: condition;
```

Volanie operácie *x.wait,* znamená, že keď proces volá túto operáciu, bude zablokovaný, kým iný proces nezavolá operáciu *x.signal.*

Operácia *wait(condition):* uvoľní uzamknutie monitora a „uspí“ proces. Keď sa proces „zobudí“, získa znova prístup k monitoru.

Operácia *signal(condition):* „zobudí“ jeden z procesov čakajúcich na premennú typu *condition*. Ak vo fronte nie je žiadny proces, neurobí nič.



Obr. 6.8 Monitor

Existuje niekoľko rôznych variantov chovania procesov po použití operácie *signal*. Líšia sa v tom, ktorý proces pokračuje vo svojej práci v monitore po zavolaní operácie *signal*. Nech proces *P* zavolal operáciu *signal* nad premennou typu *condition*, pred ktorou je pozastavený proces *Q*. Sú dve možnosti, ktoré vylučujú prítomnosť obidvoch procesov v monitore:

1. Proces *P* buď čaká, kým *Q* opustí monitor, alebo čaká na inú podmienku.
2. Proces *Q* čaká, pokiaľ *P* opustí monitor, alebo čaká na inú podmienku.

V jazyku Concurrent Pascal napríklad je použitý kompromis medzi týmito dvoma alternatívami: keď proces *P* zavolá operáciu *signal*, okamžite opustí monitor a svoju prácu v monitore obnoví proces *Q*. Tento model má nevýhodu, že proces *P* môže zavolať *signal* iba raz v jednom volaní monitora.

Mnoho programovacích jazykov, ktoré zahŕňajú semafory, neponúkajú monitory. V takýchto prípadoch je možné implementovať monitory pomocou semaforov. Pre každý monitor určíme jeden semafor - *mutex*- pre zaistenie vzájomného vylúčenia (inicializovaný na 0 alebo 1). Pred vstupom do monitora proces musí vykonať *wait(mutex)* a po opustení *signal(mutex)*.

Pretože signalizujúci proces musí čakať, kým odblokovaný proces buď opustí monitor, alebo bude čakať na inú podmienku, použijeme ďalší semafor *next*, inicializovaný na 0, pomocou ktorého signalizujúci proces bude blokovať sám seba. Procesy budú používať spoločnú premennú *next-count*, ktorá bude odzrkadľovať počet procesov, ktoré čakajú na semafore *next*. Takže každá procedúra monitora bude obsahovať nasledujúci kód:

```
wait(mutex);
        ...
    telo procedúry
        ...
if next-count > 0 then
                signal(next)
    else
                signal(mutex);
```

Tým je zaistené vzájomné vylúčenie procedúr v monitore.

Implementácia premenných typu *condition* bude vyžadovať zavedenie jedného semafora *x-sem* a jednu celočíselnú premennú *x-count* pre každú premennú *x* tohoto typu, obe inicializované na 0.

Operácia *x.wait* potom bude vyzerať nasledovne:

```
x-count:= x-count + 1;
if next-count > 0 then
            signal(next)
      else
            signal(mutex) ;
wait(x-sem) ;
```

A operácia *x.signal* takto:

```
if x-count > 0 then
      begin
        next-count:=next-count +1;
       signal(x-sem) ;
       wait(next) ;
       next-count:=next-count - 1;
      end;
```

Táto implementácia zodpovedá definícii monitora podľa Hoare-ho a B.Hansena. V niektorých prípadoch nie je potrebná táto všeobecnosť a potom sa dá dosiahnuť väčšia efektívnosť implementácie.

6.7 Problémy synchronizácie v distribuovanom systéme

Synchronizácia v distribuovanom systéme (DS) sa v mnohom podobá synchronizácii v centralizovanom systéme, ale absencia spoločnej pamäte a spoločných hodín ju komplikuje a vyžaduje vyriešenie nasledujúcich problémov:

- usporiadanie udalostí,
- vzájomné vylúčenie,
- zaistenie nedeliteľnosti (atomickosti) transakcie,
- riadenie paralelného prístupu,
- detekciu, prevenciu a obsluhu uviaznutia procesov,
- hlasovanie,
- dosiahnutie dohody.

6.7.1 Usporiadanie udalostí

V mnohých aplikáciách je dôležité usporiadanie udalostí. V centralizovanom systéme je vždy možné určiť, ktorá udalosť nastala skôr, zatiaľ čo v DS to nie je vždy možné. Algoritmy pre úplne usporiadanie udalostí potrebujú buď spoločné hodiny, alebo úplne perfektne zosynchronizované lokálne hodiny. Pretože ani jedny z nich v DS nie súk dispozícií, problém sa rieši pomocou časových pečiatok.

6.7.2 Vzájomné vylúčenie

Problém vzájomného vylúčenia v distribuovanom prostredí je možné riešiť nasledujúcimi spôsobmi:

- **Centralizovaný spôsob**. Pri tomto spôsobe sa medzi procesmi v systéme vyberie jeden proces, ktorý koordinuje prístup ku kritickej sekcii. Každý proces, ktorý potrebuje prístup do KS zašle procesu - koordinátorovi správu s požiadavkou. Vstup procesu do KS je povolený vtedy, keď proces - koordinátor odpovie na požiadavku. Po ukončení práce v KS, proces zašle správu koordinátorovi o uvoľnení prostriedku. Koordinátor radí požiadavky do frontu a

povoľuje vstup do KS vždy tomu procesu, ktorý je na rade v súlade s aplikovaným plánovacím algoritmom.

- **Distribuovaný spôsob**. Tento spôsob je komplikovanejší, pretože rozhodnutie sa musí urobiť na základe dohody so všetkými procesmi.
- **Riadenie pomocou tokenu**. Token je špeciálna správa, ktorá putuje po celom systéme. Proces môže vstúpiť do KS len v prípade, že vlastní token. Pretože token je len jeden, nemôže nastať prípad, že v KS budú dva procesy.

6.7.3 Zaistenie nedeliteľnosti (atomickosti) transakcií

V mnohých prípadoch je potrebné zaistiť, aby určitý sled operácií bol vykonaný buď celý, alebo nebol vykonaný vôbec. Sú to tzv. **transakcie**, pojem z databázových systémov. V DS zaistenie atomickosti transakcie je omnoho komplikovanejšie ako v centralizovanom systéme. Touto úlohou je obyčajne poverený tzv. *transakčný koordinátor* (TK), ktorý sa stará o:

- štart transakcie,
- rozbitie transakcie na podúlohy a ich mapovanie do jednotlivých uzlov,
- koordináciu ukončenia úlohy - ak transakcia prebehla celá v poriadku, potvrdí sa ako úspešná, v opačnom prípade sa zruší.

Každý uzol udržuje svoj *log file* - súbor, kde sa zaznamenávajú všetky vykonané operácie. V prípade neúspechu transakcie sa podľa týchto záznamov vráti počiatočný stav.

6.7.4 Riadenie paralelného prístupu

Pretože každá transakcia je atomická, paralelné vykonanie transakcií v DS musí byť ekvivalentné prípadu, ako keby sa tieto transakcie vykonávali sekvenčne v ľubovoľnom poradí. Táto vlastnosť sa nazýva *serializovateľnosť* a dá sa dosiahnuť tak, že každá transakcia sa vykonáva v KS. Funkciu koordinátora paralelného prístupu v každom uzle zastáva *transakčný manažér*, t.j. proces, ktorý sa stará o riadenie prístupu k lokálnym dátam. V DS každá transakcia môže byť buď lokálna alebo globálna, podľa toho, s akými dátami manipuluje. Prístup k dátam sa môže uskutočniť v dvoch režimoch - zdieľanom a výlučnom, podľa toho, či sa dáta budú len čítať, alebo modifikovať.

Najčastejšie používané metódy pre riadenie paralelného prístupu k dátam v DS sú:

- uzamykanie,
- metóda dátového modelu bez replikácií,
- metóda jediného koordinátora,
- väčšinový (majority) protokol,
- „nespravodlivý“ (biased) protokol,
- metóda primárnej kópie,
- metóda časových pečiatok.

6.7.5 Detekcia, prevencia a obsluha uviaznutia procesov

Prevencia, detekcia a obsluha uviaznutia v DS využíva metódy podobné tým, ktoré sa používajú v centralizovaných systémoch.

1. **Prevencia uviaznutia**

   Využívajú sa nasledujúce metódy:

- metóda globálneho usporiadania systémových prostriedkov, ktoré procesy zdieľajú,
- metóda bankára, rozšírená pre DS,ň
- metóda „wait - die“ (čaká, alebo zanikne),
- metóda „wound - wait“ (čaká, alebo zlikviduje).

2. **Detekcia uviaznutia** - pre detekciu uviaznutia je potrebné udržiavať graf pridelenia prostriedkov, ktorý sa konštruuje pre všetky procesy a prostriedky systému. Uviaznutie v systéme nastalo vtedy, keď v grafe vznikne cyklus, t.j. niekoľko procesov čaká na seba cyklickým spôsobom. V DS každý uzol udržiava svoj lokálny graf, ale stav systému sa dá odvodiť len z grafu, ktorý vznikol zjednotením grafov jednotlivých uzlov. Pri detekcii uviaznutia sa uplatňujú dva prístupy:
   - *centralizovaný prístup* - zamestnáva proces, ktorý je koordinátorom a udržiava priebežne graf pridelenia prostriedkov pre celý systém,
   - *úplne decentralizovaný prístup* - každý uzol udržiava svoj lokálny graf. V prípade uviaznutia sa to zaznamená aspoň v jednom uzle.

6.7.6 Hlasovanie (election, voting)

V mnohých predchádzajúcich prípadoch jeden proces koordinoval danú činnosť v systéme. Problém pri tomto centralizovanom prístupe vzniká vtedy, keď tento proces z nejakého dôvodu zanikne. Vtedy je potrebné zvoliť nového koordinátora a ohlásiť to všetkým procesom. Obyčajne sa tento problém rieši pomocou unikátnych prioritných identifikátorov procesov v systéme. Koordinátor má najväčší identifikátor. Pri jeho zániku sa za koordinátora zvolí iný proces, ktorý má najväčší identifikátor.

6.7.7 Dosiahnutie dohody

Tento problém je v literatúre známy pod názvom problém Byzantských generálov, ktorí sa musia dohodnúť na ďalšom priebehu bitky, pričom si môžu vymieňať správy pomocou poslov. Úspech je zaručený len v prípade, že na nepriateľa zaútočia všetci spoločne, na čom sa musia dohodnúť. Sú dve hlavné príčiny zlyhania dosiahnutia dohody: nepriateľ chytí niektorého posla, alebo niektorý z generálov zradí.

V DS týmito príčinami sú: nespoľahlivý komunikačný podsystém a zlyhanie niektorého procesu, pričom jeho správanie je nepredvídateľné. Prvý prípad sa obyčajne rieši pomocou time-out-ov: ak do určitej doby nepríde odpoveď na vyslanú správu, operácia sa opakuje, kým nedostane odpoveď, alebo inou správou, ktorá upozorňuje na výpadok príslušného uzla. Pre rozpoznanie zlyhania procesu existuje množstvo algoritmov, zložitosť ktorých je závislá od počtu procesov.

6.8 Klasické synchronizačné úlohy

6.8.1 Producent - konzument

V tejto úlohe sa jedná o dva spolupracujúce procesy, ktoré komunikujú cez vyrovnávaciu pamäť obmedzenej veľkosti. Prvý proces produkuje informáciu a vkladá ju do vyrovnávacej pamäte, odkiaľ ju druhý proces vyberá. Aby mohli obidva procesy prebiehať paralelne, ich rýchlosti sa musia zosynchronizovať, t.j. producent musí mať vždy voľné miesto vo vyrovnávacej pamäti pre uloženie informácie a konzument musí mať vždy hotovú položku pre výber. Ak tomu tak nie je, proces, ktorý nemôže pokračovať v činnosti musí počkať - producent na uvoľnenie miesta vo vyrovnávacej pamäti, konzument na uloženie informácie, ktorú môže následne vybrať.

6.8.2 Čitatelia - zapisovatelia

Dátový objekt (súbor alebo záznam) je zdieľaný medzi niekoľkými procesmi. Niektoré procesy môžu len čítať obsah zdieľaného objektu a sú nazývané čitatelia, iné ho môžu modifikovať (t.j. čítať a zapisovať) a sú nazývané zapisovatelia. Synchronizačný problém spočíva v zaistení výlučného

prístupu zapisovateľov. Potom čitatelia môžu pristupovať k objektu paralelne, zatiaľ čo zapisovatelia len po jednom. V opačnom prípade môže vzniknúť chaos!

6.8.3 Obedujúci filozofi

Je päť filozofov, ktorí trávia svoj život jedením a premýšľaním. Filozofi sedia okolo okrúhleho stola, na stole je päť tanierov so špagetami a päť vidličiek. Z času na čas premýšľanie filozofa vyruší hlad a on sa chce najesť. Za týmto účelom potrebuje dve vidličky (špagety sú veľmi klzké!). V danom okamihu filozof môže zobrať len jednu vidličku zo stola a potom sa pokúsi zobrať tú druhú. Samozrejme nemôže zobrať vidličku, ktorú už má jeho sused. Ak dostane obidve vidličky, môže jesť, inak musí nechať vidličku na stole a pokúsiť sa o získanie vidličiek neskôr. Naopak, ak ich dostane, naje sa, položí vidličky na stôl a pokračuje v premýšľaní. Synchronizačný problém tu spočíva v tom, nedovoliť filozofovi držať jednu vidličku a čakať na uvoľnenie druhej, pretože môže nastať uviaznutie. Tento problém sa pokladá za klasický, pretože je to príklad širokej triedy úloh riadenia paralelných procesov, ktoré zdieľajú určitý počet prostriedkov.

6.9 Príklady riešenia niektorých klasických synchronizačných úloh

6.9.1 Úloha obedujúcich filozofov pomocou monitora

```
type dining-philosophers = monitor
var state: array[0..4] of (thinking, hungry, eating);
    self:  array [ 0..4] of condition;

  { keď mám hlad, nastavím svoj stav na hladný, otestujem susedov, či mi nedržia vidličky, a
    ak tomu tak nie je, jem, inak čakám }

        procedure entry pickup(i: 0..4 );
                begin
                        state[i] := hungry;
                        test(i);
                        if state[i]<> eating then self[i].wait;
                end;

{ po ukončení jedla nastavím svoj stav na myslenie a oznámim to susedom}

        procedure entry putdown(i: 0..4);
                begin
                        state[i] := thinking;
                        test((i+4) mod 5 );
                        test((i+1) mod 5 );
                end;

{ ak susedia nejedia a ja mám hlad, môžem jesť!}

        procedure test(k: 0..4);
                begin
                  if  (state[k+4 mod 5] <> eating)
                      and (state[k]=hungry)
                      and (state[k+1 mod 5]<>eating) then
                          begin
                                  state[k]:=eating;
                                  self[k].signal;
                          end;
                end;
```

{inicializácia stavu filozofov - základný stav filozofa je rozmýšľanie!}

```
begin
  for i:=0 to 4 do
      state[i] := thinking;
end;
```

6.9.2 Úloha producent - konzument pomocou semaforov

Riešenie používa tri semafory. Semafor *mutex* zaisťuje vzájomné vylúčenie procesov pri práci s bufrom a je inicializovaný na 0. Semafor *empty* počíta prázdne položky v bufri a je inicializovaný na maximálny počet položiek v bufri. Semafor *full* počíta, koľko položiek v bufri je plných. Na začiatku má hodnotu 0.

Proces producent:

```
repeat
        ...
        vyrobí položku
do nextp
        ...
  wait(empty)  ;
  wait(mutex)  ;
        ...
        vloží nextp do bufra
        ...
  signal(mutex)  ;
  signal(full)  ;
until false ;
```

Proces konzument:

```
repeat
  wait(full)  ;
  wait(mutex)  ;
        ...
  vyberi e jednu položku z  bufra do nextc
        ...
  signal(mutex)  ;
  signal(empty)  ;
        ...
  spracuje položku z  nextc
        ...
until false ;
```

6.9.3 Úloha čitatelia - zapisovatelia pomocou semaforov

Riešenie tejto úlohy má niekoľko variantov. Prvý z nich rieši problém tak, že čitateľ vždy môže začať čítanie, ak už čítajú ďalší čitatelia, a zároveň zapisovateľ chce zapisovať, t.j. zapisovateľ musí počkať až skončia všetci čitatelia. Druhý variant riešenia dovoľuje zapisovateľovi začať zápis ihneď, ako je to možné, t.j. ak po jeho požiadavke na zápis prišla požiadavka aj na čítanie, čitateľ musí čakať do ukončenia zápisu. Obidva varianty môžu spôsobiť *starváciu* - v prvom prípade zapisovateľom, v druhom čitateľom. V ďalšom texte uvedieme riešenie podľa prvého variantu, pričom ponechávame riešenie podľa druhého variantu na čitateľovi.

V tomto riešení problému procesy čitatelia a zapisovatelia zdieľajú nasledovné dátové štruktúry:

```
var mutex,                {zaisťuje vzájomné vylúčenie pri zmene hodnoty premennej
                           readcount}
    wrt: semaphore;       {zaisťuje povolenie čítania}
    readcount: integer;   {počet procesov, ktorí čítajú}
```

Proces čitateľ:

```
wait(mutex) ;
  readcount:= readcount +1 ;

  if readcount = 1 then
wait(wrt) ;
signal(mutex) ;
        ...
        vykoná sa čítanie
        ...
wait(mutex) ;
  readcount:= readcount - 1 ;
  if readcount = 0 then
                signal(wrt) ;
  signal(mutex) ;
```

Proces zapisovateľ:

```
wait(wrt) ;
        ...
        vykoná sa zápis
        ...
signal(wrt) ;
```

7 KOMUNIKÁCIE MEDZI PROCESMI

Spolupracujúce procesy v systémoch so spoločnou pamäťou využívajú pre komunikáciu práve pamäť. Môžu napríklad zdieľať spoločný bufer alebo môžu použiť pomocné systémové programy, ktoré zabezpečujú komunikáciu medzi procesmi (Interprocess Communication - IPC). Najčastejšie používané prostriedky pre komunikáciu medzi procesmi v rámci jedného systému sú správy, zdieľaná pamäť a rúry. Pre komunikáciu procesov bežiacich v distribuovaných systémoch sú vhodné správy, volanie vzdialených procedúr a sokety. V prípade aplikácií, napísaných v Jave, je možné využiť aj volanie metód vzdialených objektov.

7.1 Správy

Spolupracujúce procesy potrebujú komunikovať medzi sebou a synchronizovať svoje činnosti pri využití zdieľaných prostriedkov. Medzi základné prostriedky pre komunikáciu a synchronizáciu medzi procesmi patria správy.

Nad správami sú definované obyčajne aspoň dve základné operácie:

send (** správa **) - operácia vyslania správy,

receive (** správa **) - operácia prijatia správy.

Správy môžu mať buď *pevnú* alebo *variabilnú* dĺžku.

Ak dva procesy komunikujú, musí medzi nimi existovať komunikačná linka. Tu máme na mysli skôr logické vlastnosti a nie hardvérové. Základné otázky, na ktoré treba zodpovedať, sú: ako sa nadviaže spojenie, či sa môže k linke pripájať viac procesov, koľko liniek môže existovať medzi procesmi, aká je dĺžka správy, či je linka jednosmerná alebo obojsmerná.

Existuje niekoľko implementácií prepojenia a operácií *send* a *receive*:

- priama alebo nepriama komunikácia,
  - symetrická alebo asymetrická komunikácia,
  - automatické alebo explicitné bufrovanie,
  - vyslanie kópiou alebo odkazom,
  - pevná alebo variabilná dĺžka správy.

7.1.1 Pomenovanie

Procesy, ktoré komunikujú, musia mať možnosť ako sa na seba odkazovať. Môžu použiť priamu alebo nepriamu komunikáciu.

7.1.1.1 Priama komunikácia

Pri priamej komunikácií každý proces, ktorý potrebuje komunikovať, **musí explicitne pomenovať svojho partnera.** V tomto prípade operácie *send a receive* majú tvar:

***send** (P, správa)* - zasiela správu procesu P,

***receive** (Q, správa)* - prijíma správu od procesu Q.

Komunikačné spojenie v tomto prípade má nasledujúce vlastnosti:

- spojenie sa nadväzuje automaticky medzi každou dvojicou procesov, ktoré chcú komunikovať,
- spojenie sa nadväzuje medzi dvomi procesmi,
- medzi každým párom komunikujúcich procesov existuje jedno spojenie,
- linka môže byť jednosmerná, ale väčšinou je obojsmerná.

Na ilustráciu použitia správ uvedieme príklad riešenia klasického problému producent-konzument.

Proces-producent je definovaný nasledovne**:**

repeat

...

pripraví položku v *nextp*

...

send*(konzument, nextp)*

until false;

Proces-konzument je definovaný nasledovne**:**

repeat

receive*(producent,nextc);*

...

spracuje položku

...

until false;

Táto komunikácia je **symetrická**, t.j. vysielajúci aj prijímajúci proces uvádzajú meno svojho partnera. Variantom tejto schémy komunikácie je **asymetrická** komunikácia, kedy len vysielajúci proces uvádza meno adresáta. Operácie *send* a *receive* majú v tomto prípade tvar:

send *(P, správa)* zasiela správu procesu P,

receive *(id, správa)* prijíma správu od ľubovolného procesu; *id* je premenná, ktorá obsahuje identifikátor procesu, s ktorým bolo nadviazané spojenie.

Nedostatkom oboch spôsobov je obmedzená modularita výsledných procesov. Zmena mena partnera pre komunikáciu by vyžadovala následnú kontrolu všetkých zúčastnených procesov.

7.1.1.2 Nepriama komunikácia

Pri nepriamej komunikácii sú správy zasielané a prijímané z *mailbox*-ov. Na mailbox sa môžeme pozerať abstraktne ako na schránku, do ktorej sa správy môžu posielať a vyzdvihovať. Každý mailbox má svoj unikátny identifikátor. Jeden proces môže komunikovať s ďalšími procesmi cez niekoľko rôznych mailbox-ov. Dva procesy môžu komunikovať cez mailbox, len ak je tento zdieľaný. Operácie *send a receive* majú v tomto prípade tvar:

send *(A, správa)* - zasiela správu do mailbox-u *A,*

receive *(A, správa)* - prijíma správu od mailbox-u *A*.

Komunikačné spojenie v tomto prípade má nasledujúce vlastnosti:

- spojenie sa nadväzuje, len ak procesy majú prístup k zdieľanému mailbox-u,
  - spojenie sa nadväzuje medzi viacerými procesmi,
  - medzi každým párom komunikujúcich procesov je viac komunikačných liniek a každá zodpovedá jednému mailbox-u,
  - linka môže byť jednosmerná alebo obojsmerná.

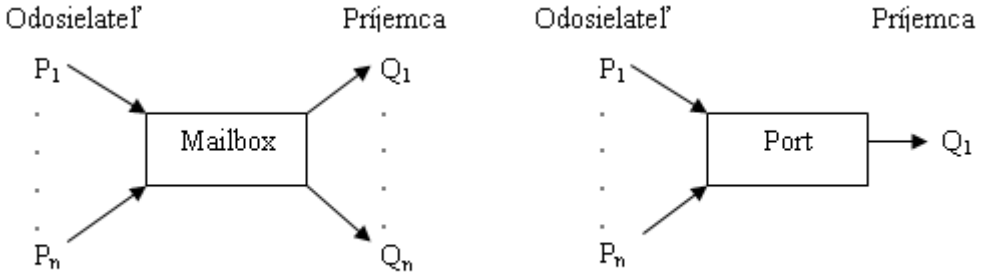
Predpokladajme, že procesy P_1, P_2 a P_3 zdieľajú mailbox A. Proces P_1 zasiela správu do A a P_2 a P_3 preberajú správy z mailboxu. Ktorý proces dostane správu, ktorú P_1 zašle? Tento problém sa dá vyriešiť niekoľkými spôsobmi:

- dovolí sa pripojenie k mailbox-u najviac dvom procesom,
  - v danom čase sa dovolí len jednému procesu vykonať operáciu *receive*,
  - dovolí sa systému, aby rozhodol, ktorý proces dostane správu.

Nepriame adresovanie dovoľuje oddeliť príjemcu od odosielateľa a tak poskytnúť väčšiu flexibilitu pri použití správ. Vzťah medzi odosielateľmi a príjemcami môže byť 1:1, 1:N, N:1 alebo N:N. Pri vzťahu 1:1 je možne ustanoviť privátne komunikačné spojenie medzi dvomi procesmi. Vzťah N:1 je užitočný v komunikáciách klient-server, kedy jeden proces (server) poskytuje službu viacerým procesom

(klientom). V tomto prípade mailbox sa často nazýva **port** (pozri Obr. 7.1). Vzťah 1:N sa využíva v prípadoch, keď je potrebne rozšíriť informáciu medzi viacerými príjemcami.

Mailbox môže patriť buď procesu alebo systému. Ak proces vlastní mailbox (je pripojený k nemu ako súčasť), potom sa rozlišuje medzi vlastníkom (ktorý len dostáva správy z mailbox-u) a používateľom mailbox-u (ktorý len posiela správy do mailbox-u). Pretože každý mailbox má jedného majiteľa, nemôže dôjsť k omylu v tom, kto má preberať správy. Keď majiteľ mailbox-u končí, mailbox sa tiež zruší. Každý proces, ktorý potom posiela správy do tohoto mailbox-u, je upozornený, že mailbox už neexistuje.



Obr. 7.1 Nepriama komunikácia: mailbox a port

Existuje niekoľko spôsobov ako určiť vlastníkov a používateľov mailbox-u. Jedna možnosť je dovoliť procesu deklarovať premennú typu *mailbox*. Proces, ktorý takúto premennú deklaroval, je majiteľom mailbox-u. Každý ďalší proces, ktorý vie meno mailbox-u, ho môže používať.

Ďalšou možnosťou je, že majiteľom mailbox-u je systém. V tomto prípade mailbox nie je pripojený k žiadnemu procesu. Operačný systém poskytuje mechanizmy, ktoré dovoľujú procesu:

- vytvoriť mailbox,
  - posielať a dostávať správy z mailbox-u,
  - zrušiť mailbox.

Obyčajne proces, ktorý vytvoril mailbox, je jeho majiteľom a jedine on má právo dostávať správy cez tento mailbox. Vlastníctvo sa dá presunúť aj iným procesom cez príslušné systémové volanie. Iný spôsob presunutia práva vlastníka na iný proces je tvorba potomkovho procesu, ktorý zdedí prístupové práva otca. Ak všetky procesy, ktoré pracovali s mailbox-om skončia svoju činnosť, operačný systém musí uvoľniť pamäťový priestor používaný mailbox-om.

7.1.2 Bufrovanie

Každá komunikačná linka má nejakú kapacitu, ktorá určuje, aký počet správ sa v nej môže nachádzať. Táto vlastnosť sa dá znázorniť ako front správ, pripojených k linke. Všeobecne existujú tri spôsoby, ako takýto front môže byť implementovaný:

- **Nulová kapacita.** Maximálna dĺžka frontu je 0, t.j. vo fronte nemôže čakať žiadna správa. V tomto prípade vysielajúci proces musí počkať, až prijímajúci proces preberie správu. Procesy sa musia zosynchronizovať pri odovzdávaní správy. Tejto synchronizácií sa hovorí *rendezvous*.
  - **Obmedzená kapacita**. Front má konečnú dĺžku *n*, t.j. môže v ňom čakať najviac *n* správ. Ak front nie je plný, prichádzajúca správa sa doň zaradí a vysielajúci proces môže pokračovať vo svojom vykonaní bez čakania. Ak front je plný, vysielajúci proces je zablokovaný, pokiaľ sa uvoľní miesto.
  - **Neobmedzená kapacita**. Front má prakticky neobmedzenú dĺžku.

O linke s nulovou kapacitou sa niekedy hovorí, že je to systém správ bez bufrovania.

Čitateľ si určite všimol, že v prípadoch bufrovania s nenulovou dĺžkou bufra odosielateľ po uvedení operácie *send* nevie, či správa dorazila k adresátovi. Ak táto informácia je dôležitá pre výpočet, odosielateľ musí dodatočne komunikovať, aby zistil stav. Napríklad proces *P*, posiela správu procesu *Q* a môže pokračovať vo svojom vykonaní, len ak správa bola úspešne doručená.

Proces ***P*** vykoná:

send *(Q, správa)* - zasiela správu *Q*

receive *(Q, potvrdenie)* - prijíma potvrdenie od *Q*

Proces ***Q*** vykoná:

receive *(P, správa)* - prijíma správu od *P*

send *(P, potvrdenie)* - zasiela potvrdenie *P*

Takáto komunikácia sa nazýva *synchrónna*, t.j. procesy synchronizujú operácie *send* a *receive*. Naproti tomu komunikácia je *asynchrónna*, ak iba proces príjemca čaká (je blokovaný), kým nedostane správu.

Existujú špeciálne prípady, kedy komunikácia sa nedá zaradiť do jednej z predchádzajúcich kategórii:

- Proces-odosielateľ sa neblokuje nikdy, ale ak príde nová správa skôr ako predchádzajúca bola vybratá, tak táto je stratená. Výhodou je, že veľké správy sa nemusia viac krát kopírovať. Nedostatkom je, že procesy sa musia explicitne synchronizovať, aby nedochádzalo ku strate správ a tiež pre vzájomné vylúčenie prístupu procesov k bufru.
- Vysielajúci proces je zablokovaný, kým nedostane potvrdenie o prijatí správy.

7.1.3 Problémy pri zasielaní správ

Systém zasielania správ sa veľmi často používa v distribuovanom prostredí, v ktorom sa chyby v komunikácií môžu vyskytnúť oveľa častejšie.

7.1.3.1 Ukončenie procesu

Vysielajúci alebo prijímajúci proces môže byť ukončený pred tým ako poslal alebo prijal správu. To znamená, že môže poslať správu, ktorá nebude nikdy prebratá, alebo proces môže čakať na správu, ktorú nemá kto poslať. Predpokladáme nasledujúce prípady:

1. Proces príjemca P čaká na správu od procesu Q, ktorý bol ukončený, takže P bude zablokovaný donekonečna. V tomto prípade OS musí buď upozorniť P, že Q bol ukončený, alebo ho ukončiť.

2. Proces *P* zasiela správu procesu *Q*, ktorý bol ukončený. Pri automatickom bufrovaní sa nič nedeje, *P* jednoducho pokračuje vo svojom vykonaní. Ak *P* potrebuje potvrdenie o prijatí, musí si to explicitne naprogramovať. Ak je použitý systém bez bufrovania, *P* bude zablokovaný natrvalo. Tu OS tiež musí buď upozorniť alebo ukončiť proces *P*.

7.1.3.2 Stratená správa

Správa od procesu *P* k procesu *Q* sa môže stratiť počas prenosu vplyvom komunikačnej siete. Používajú sa nasledujúce metódy ošetrenia tejto udalosti:

1. OS je zodpovedný za zistenie straty správy a jej opätovného vyslania.
2. Za zistenie straty správy a jej opätovného vyslania je zodpovedný vysielajúci proces.

3. OS je zodpovedný za zistenie straty správy a upozornenie vysielajúceho procesu. Opätovné vyslanie správy je prenechané vysielajúcemu procesu.

Nie vždy je potrebné zisťovať stratu správy. Celý rad protokolov zaisťuje spoľahlivý prenos, takže záleží na tom, aký protokol využívame.

Strata správy sa dá zistiť pomocou *timeout*-ov. Po zaslaní správy sa očakáva potvrdenie jej prijatia. Špecifikuje sa časový interval, kedy sa očakáva príchod potvrdenia. Ak počas tohoto intervalu nepríde potvrdenie, predpokladá sa, že správa sa stratila a vysiela (OS alebo proces) sa opätovne. Ak sa jedná iba o väčšie oneskorenie, je nutné mať nejaký mechanizmus pre správne usporiadanie správ bez opakovania.

7.1.3.3 Poškodená správa

V podstate tento prípad je veľmi podobný predchádzajúcemu, ale tu OS obyčajne zistí poškodenie pomocou rôznych kontrol (kontrolné sumy, parita, Cyclic Redundancy Code a iné) a zabezpečí aj opätovné vyslanie správy.

7.1.4 Synchronizácia pomocou správ

Komunikácia pomocou správ zahŕňa v sebe určitú úroveň synchronizácie medzi dvomi procesmi, pretože príjemca nemôže dostať správu skôr ako ju odosielateľ vyšle. Tu je potrebne špecifikovať čo sa stane ak proces použije operáciu *send* alebo *receive.*

Proces, ktorý použije operáciu *send*, má dve možnosti: buď je zablokovaný kým príjemca nedostane správu, alebo pokračuje v svojej činnosti. Proces, ktorý použije operáciu *receive* má tiež dve možnosti:

- ak správa bola vyslaná skôr, získa ju a pokračuje vo vykonávaní,
- ak správa nebola ešte doručená môže byť zablokovaný kým správa príde alebo pokračuje vo svojej činnosti a viac sa nepokúša ju získať.

Z uvedeného vyplýva, že obidve operácie môžu byť blokujúce alebo neblokujúce. Najbežnejšie sú nasledujúce kombinácie:

- **blokujúca operácia *send*, blokujúca operácia *receive***: aj odosielateľ aj príjemca sú zablokovaní kým správa nie je doručená. Táto kombinácia dovoľuje synchronizáciu činnosti procesov,
- **neblokujúca operácia *send*, blokujúca operácia *receive***: odosielateľ môže pokračovať v práci, príjemca je zablokovaný kým správa nie je doručená. Táto kombinácia je najčastejšia,
- **neblokujúca operácia *send*, neblokujúca operácia *receive***: žiadny proces nie je zablokovaný pri týchto operáciách.

Neblokujúca operácia *send* je najčastejšie implementovaná v programovacích jazykoch. V tomto prípade informáciu o doručení správy je možne získať dodatočnou potvrdzovacou správou.

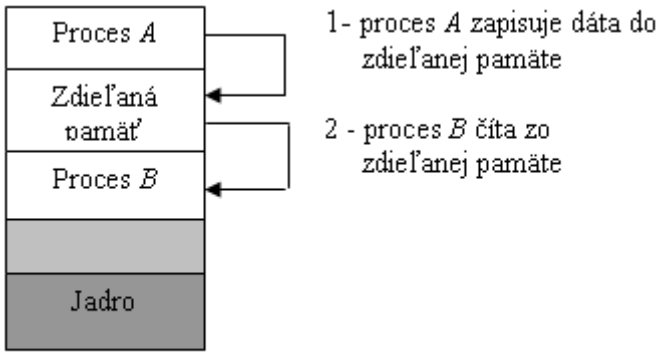
Operácia *receive* je najčastejšie implementovaná ako blokujúca. Pri použití tejto implementácie v distribuovaných systémoch vzniká nebezpečenstvo, že pri strate správy proces príjemca bude zablokovaný donekonečna. Tento problém by sa dal vyriešiť neblokujúcou operáciou *receive*, ale v tomto prípade, ak proces príjemca zavolá *receive* skôr ako proces odosielateľ vyšle správu, tá bude stratená. Riešením by bolo umožniť príjemcovi testovať príchod správy.

7.2 Zdieľaná pamäť

Zdieľaná pamäť poskytuje najrýchlejšiu komunikáciu medzi procesmi. Ten istý pamäťový segment je mapovaný do adresných priestorov dvoch alebo viacerých procesov. Ihneď ako sú dáta zapísané do zdieľanej pamäte, procesy, ktoré majú k nej prístup môžu tieto dáta čítať.

Pri súbežnom prístupe k zdieľaným dátam je potrebné *zaistiť synchronizáciu prístupu.* V tomto prípade zodpovednosť za komunikáciu padá na programátora, operačný systém poskytuje len

prostriedky pre jej uskutočnenie. Problémy spojené so synchronizáciou prístupu budú podrobnejšie vysvetlené v kapitole o synchronizácií procesov. Komunikačný model zdieľanej pamäte je uvedený na Obr. 7.2.



Obr. 7.2 Komunikačný model zdieľanej pamäte

7.3 Rúry

Rúry sú najstarším a najjednoduchším mechanizmom komunikácie medzi procesmi. Procesy komunikujú pomocou bufra, implementovaný jadrom, ktorý má konečnú veľkosť. Dáta sa ukladajú v poradí príchodu (FIFO), pričom jeden z procesov ich zapisuje, druhý ich číta. Rúry poskytujú komunikáciu 1:1 a väčšinu existujú tak dlho ako proces, ktorý ich vytvoril.

Rúry sú obyčajne implementované pomocou systémového volania. Po vytvorení rúry proces môže ihneď do nej zapisovať, pokiaľ je tam dostatok miesta. Ak nie je, proces je zablokovaný kým sa neuvoľní. Obdobne funguje aj proces, ktorý číta z rúry: ak informácia je prítomná, získa ju hneď, inak je zablokovaný. Tradičné unix-ovské rúry sú jednosmerné, ale niektoré modernejšie implementácie UNIX-u poskytujú obojsmerné rúry. Rúry sú bežným komunikačným prostriedkom aj v prostredí systému Windows.

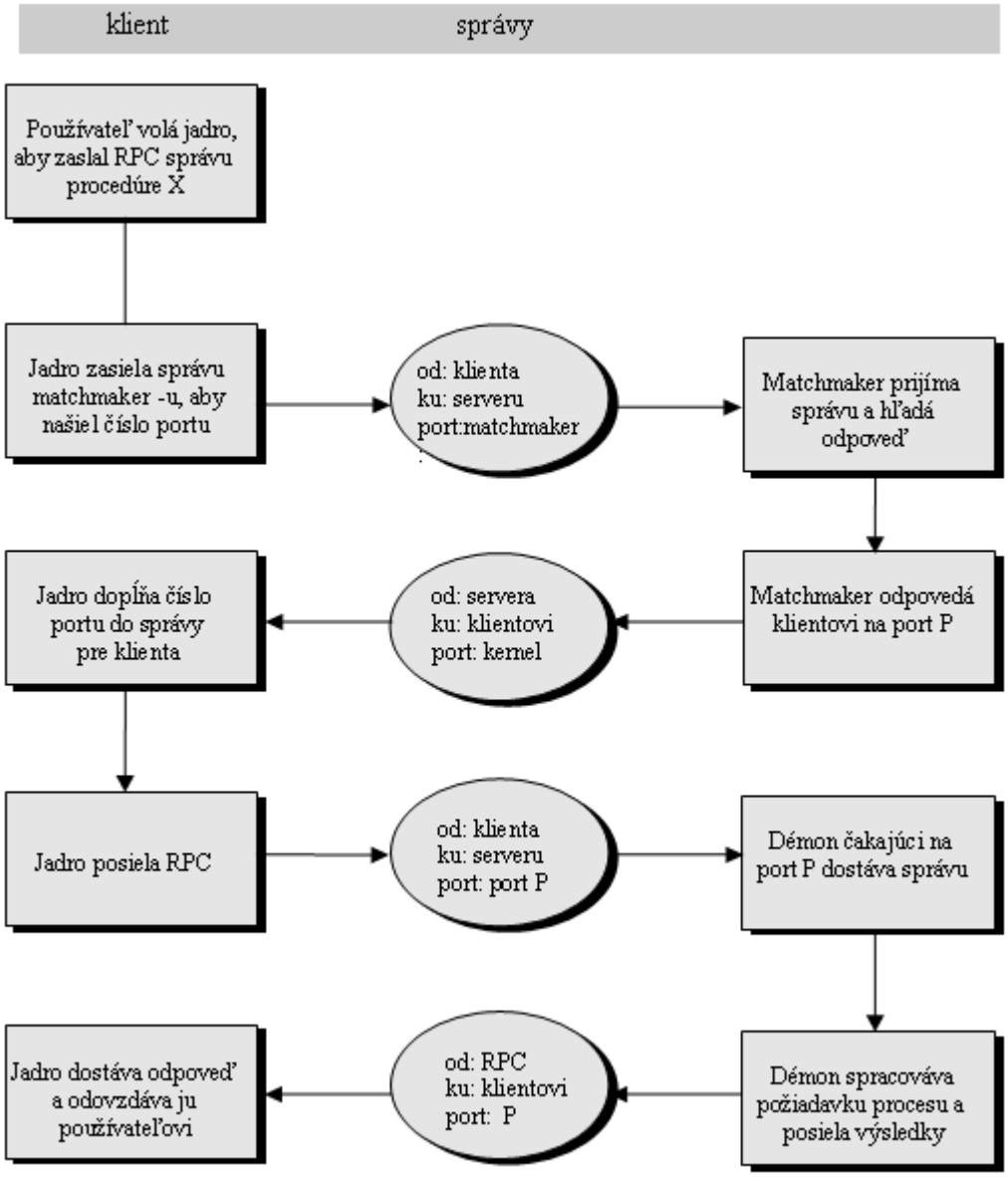
Existujú dva typy rúr: *nepomenované* a *pomenované*. Nepomenované rúry dovoľujú komunikáciu len medzi „príbuznými“ procesmi, pomenované rúry sú použiteľné pre ľubovoľné dva procesy.

7.4 Komunikácie medzi procesmi v distribuovaných systémoch

7.4.1 Volanie vzdialenej procedúry

Synchrónna komunikácia pomocou správ sa dá ľahko rozšíriť do systému volania vzdialenej procedúry (Remote Procedure Call - RPC).

Volanie vzdialenej procedúry je založené na mechanizme, ktorý je podobný mechanizmu volania lokálnej procedúry. Používa sa veľmi často v distribuovaných aplikáciách klient-server. Procesy takejto aplikácie sa vykonávajú na rôznych uzloch siete a komunikujú pomocou zasielania správ. Keď sa zavolá vzdialená procedúra, lokálny systém prenesie požiadavku a parametre volania do vzdialeného systému, ktorý vykoná procedúru. Po ukončení práce vzdialený systém posiela naspäť kód úspešnosti vykonania a výsledky (pozri Obr. 7.3).



Obr. 7.3 Vykonanie volania vzdialenej procedúry (RPC)

Pre zabezpečenie komunikácie medzi procesmi, ktoré sa vykonávajú na rôznych systémoch v sieti, je potrebné správu adresovať na **konkrétny port** (identifikačné číslo koncového bodu spojenia), ktorý je pridelený príslušnej procedúre. Zistenie portu sa uskutočňuje buď *staticky* alebo *dynamicky*.

Pri statickom určovaní adresy portu je tento priradený procedúre ešte pri preklade. Po preložení programu sa číslo portu nedá zmeniť, čo môže byť veľmi nepohodlné po páde systému, keď sa môžu zmeniť čísla portov.

Pri dynamickom zistení čísla portu sa toto číslo zisťuje pred nadviazaním spojenia. Vzdialenému systému sa pošle správa na známy port, na ktorom počúva tzv. *matchmaker* alebo *portmapper*. Je to program, ktorý eviduje služby poskytované príslušným systémom a ich portom a na vyslanú žiadosť o určenie portu odpovedá zaslaním jeho čísla. Tento spôsob je pružnejší, ale vyžaduje o niečo dlhšie „naladenie“ spojenia.

Často sa procesy, ktoré využívajú volanie vzdialených procedúr, vykonávajú na heterogénnych strojoch. Potom vyvstáva problém, ako si majú vymieňať dáta. Za týmto účelom je definovaný

protokol *XDR (eXternal Data Representation protocol),* ktorý zabezpečuje zakódovanie jednotlivých dát.

2.2 Volanie vzdialenej metódy (Remote Method Invocation)

Volanie vzdialenej metódy (RMI) je črta objektového programovacieho jazyka Java, ktorá je podobná RPC. RMI dovoľuje vláknu volať metódu vzdialeného objektu. Objekt je vzdialený, ak sídli v inej JVM (Java Virtual Machine), ktorá môže bežať buď na tom istom počítači, alebo na inom uzli v sieti. Hlavný rozdiel medzi RPC a RMI je, že RPC odovzdáva parametre volania ako obyčajnú dátovú štruktúru. RMI dovoľuje odovzdávanie objektov ako parameter volania vzdialenej metódy.

2.3 Sokety

Soket je koncový bod pre komunikáciu. Procesy komunikujú pomocou dvojice soketov. Sokety využívajú väčšinou architektúry typu klient-server, čo znamená že môžu byť použité pre lokálne alebo sieťové spojenia. Soket pozostáva z IP adresy uzla ku ktorej je pripojené číslo portu.

Známe sieťové služby ako napr. telnet, ftp, http a iné „počúvajú“ na známych portoch. Keď klient požiada o spojenie, priradí sa mu číslo portu. Keďže každé spojenie musí byť jedinečné, ak ďalší klient požiada o spojenie s tým istým serverom, priradí sa mu väčšie číslo portu. Soket existuje, kým ešte je nejaký proces pripojený k nemu, alebo kým ho proces, ktorý ho vytvoril nezruší.

Vytvorenie soketu, pripojenie čísla portu k IP adresy a práca so soketom sa uskutočňuje pomocou systémových volaní.

Komunikácia pomocou soketov, aj keď je efektívna a bežná, je považovaná za nižšiu formu komunikácie medzi procesmi v distribuovanom systéme. Je to hlavne kvôli tomu, že sokety dovoľujú prenos len neštruktúrovaného prúdu bajtov. Vytvorenie potrebnej štruktúry dát je ponechané na aplikáciách, ktoré ich využívajú.

8 UVIAZNUTIE PROCESOV

Proces pre svoje vykonanie potrebuje prostriedky systému - procesor, pamäť, periférne zariadenia, súbory. V multiprogramovom prostredí viac procesov súperí o tieto prostriedky. Proces žiada o prostriedok, ak ten nie je voľný, proces je zablokovaný. Môže sa stať, že stav zablokovania sa nikdy nezmení, pretože žiadaný prostriedok drží iný proces, ktorý tiež čaká na uvoľnenie iného prostriedku. Táto situácia sa nazýva *uviaznutie* (*deadlock)*.

8.1 Model systému

Systém pozostáva z konečného počtu prostriedkov, ktoré sa majú rozdeliť medzi procesy. Prostriedky sú rozdelené na niekoľko typov, z ktorých každý pozostáva z viacerých jednotiek. Napr. pamäť, CPU, súbory, V/V zariadenia (také ako tlačiarne alebo magnetické pásky) sú príkladmi takýchto typov prostriedkov. Systém môže disponovať viacerými prostriedkami daného typu, napr. dva procesory, tri tlačiarne atď. Ak proces žiada o prostriedok daného typu, môže mu byť pridelený ľubovolný z danej triedy. Ak prostriedky triedy nie sú identické, potom musia byť rozdelené do rôznych tried.

Proces môže žiadať o toľko prostriedkov, koľko potrebuje pre splnenie svojej úlohy. Samozrejme, že proces by nemal žiadať o väčší počet prostriedkov, ako systém vlastní.

Za normálnych okolností proces používa prostriedky systému podľa nasledujúcich krokov:

1. Požiada o prostriedok. Ak sa nemôže požiadavke vyhovieť ihneď (ak napr. tento prostriedok práve používa iný proces), potom proces musí čakať na uvoľnenie prostriedku.
2. Použije prostriedok.
3. Uvoľní prostriedok.

Požiadanie o prostriedok sa uskutočňuje pomocou systémových volaní, ktoré už boli popísané. Napr. *request* a *release* pre zariadenia, *open* a *close* pre súbory, *allocate* a *free* pre pamäť.

Množina procesov je v stave **uviaznutia**, ak každý proces z množiny čaká na udalosť, ktorú môže vyvolať len iný proces z tejto množiny. Tu máme na mysli hlavne udalosti požiadania o prostriedok a uvoľnenie prostriedku. Prostriedky môžu byť buď fyzické (tlačiarne, magnetické pásky, pamäť, CPU) alebo logické (súbory, semafory, monitory).

Pre ilustráciu uviaznutia uvedieme nasledujúci príklad: majme tri tlačiarne a tri procesy, každý z ktorých drží jednu pásku, ale pre svoje dokončenie potrebuje ešte jednu. Každý proces čaká na udalosť „uvoľnenie ďalšej pásky“, ktorú môže vyvolať len jeden z ďalších dvoch procesov. V tomto prípade procesy súperia o prostriedky rovnakého typu.

Uviaznutie môže vzniknúť aj pri súperení o prostriedky rôznych typov. Napr. ak proces P_i má pásku a proces P_j má tlačiareň. Ak P_i požaduje tlačiareň a P_j pásku, potom nastane uviaznutie.

8.2 Charakteristika uviaznutia

Uviaznutie je nežiadúci stav. V tomto stave procesy nikdy nekončia a systémové prostriedky sú viazané, čím brzdia prácu ďalších procesov.

8.2.1 Nutné podmienky pre uviaznutie

Uviaznutie môže nastať ak sú splnené nasledujúce štyri (Coffmanove) podmienky naraz v danom čase:

1. **Vzájomné vylúčenie.** Aspoň jeden prostriedok musí byť pridelený výlučne, to znamená, že nemôže byť zdieľaný.

Vlastniť a žiadať. Musí existovať proces, ktorý má pridelený aspoň jeden prostriedok a požaduje ďalšie prostriedky, ktoré sú pridelené iným procesom.

Používanie bez preempcie. Prostriedok nemôže byť odňatý, t.j. proces môže uvoľniť prostriedok jedine dobrovoľne, keď s ním ukončí prácu.

Kruhové čakanie. Musí existovať množina $P_0, P_1, ..., P_n$ čakajúcich procesov takých, že P_0 čaká na prostriedok, ktorý drží P_1 , P_1 čaká na prostriedok, ktorý drží P_2 ,..., P_{n-1} čaká na prostriedok, ktorý drží P_n a P_n čaká na prostriedok, ktorý drží P_0 .

Pre vznik uviaznutia musia platiť **všetky štyri podmienky súčasne**.

8.2.2 Graf prideľovania prostriedkov

Uviaznutie sa dá popísať lepšie pomocou orientovaného grafu, ktorý je nazvaný graf prideľovania prostriedkov. Tento graf pozostáva z množiny vrcholov V a množiny hrán E. Množina vrcholov má dve podmnožiny $P=P_1, P_2, ...P_n$ a $R=R_1, R_1, ...R_n$. Množina P pozostáva zo všetkých procesov systému a množina R zo všetkých typov prostriedkov systému.

Orientovanú hranu z procesu P_i do prostriedku typu R_i označujeme P_i --->R_i a táto hrana znamená, že proces P_i žiada jeden prostriedok typu R_i a momentálne čaká, aby ho dostal. Orientovanú hranu z prostriedku R_i do procesu P_i označujeme R_i--->P_i a táto hrana znamená, že jeden prostriedok typu R_i bol pridelený procesu P_i.

Graficky znázorňujeme procesy krúžkom a prostriedky štvorčekom. Pretože prostriedkov daného typu môže byť viacej ako jeden, vo vnútri každého štvorčeka sú bodky, ktoré znázorňujú počet prostriedkov. Hrana, vyjadrujúca priradenie prostriedku musí ukazovať na príslušnú bodku.

Keď proces požiada o prostriedok, do grafu sa vloží hrana, ktorá smeruje od procesu k prostriedku. Ak sa požiadavka dá uspokojiť, hrana sa hneď pretransformuje na hranu od prostriedku k procesu. Keď proces uvoľní prostriedok, hrana sa zmaže.

Na Obr.8.1 je ukázaný graf prideľovania prostriedkov, kde

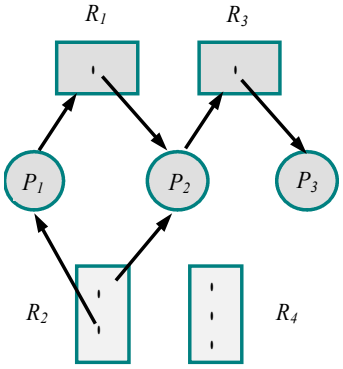
$$P = \{ P_1, P_2, P_3 \},\ R = \{ R_1, R_2, R_3, R_4 \}$$

$$E = \{ P_1 \text{-->} R_1,\ P_2 \text{-->} R_3,\ R_1 \text{-->} P_2,\ R_2 \text{-->} P_2,\ R_2 \text{-->} P_1,\ R_3 \text{-->} P_3 \}.$$

Počty prístupných prostriedkov sú: $R_1 = 1$, $R_2 = 2$, $R_3 = 1$, $R_4 = 3$.

Procesy sú v nasledujúcich stavoch:

P_1 - vlastní jeden prostriedok typu R_2 a čaká na získanie prostriedku typu R_1.
P_2 - vlastní po jednom prostriedku typu R_1 a R_2 a čaká na získanie prostriedku typu R_3.
P_3 - vlastní jeden prostriedok typu R_3.



Obr. 8.1 Graf prideľovania prostriedkov

Pomocou grafu prideľovania prostriedkov, ktorý je zostavený podľa uvedených pravidiel, sa dá ľahko ukázať, že ak graf neobsahuje cyklus, uviaznutie nenastalo, v opačnom prípade uviaznutie je možné.

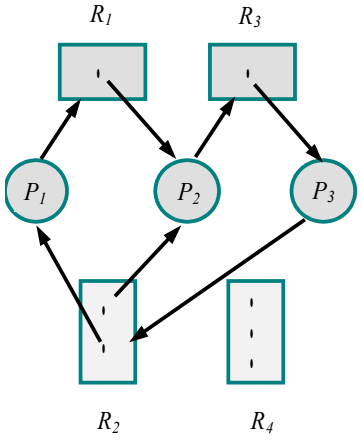
Ak každý prostriedok v systéme je iba jeden, potom existencia cyklu hovorí, že uviaznutie určite nastalo. Ak cyklus obsahuje typy prostriedkov, ktoré sú len po jednom v systéme, to tiež svedčí, že nastalo uviaznutie. *V týchto prípadoch je existencia slučky nutnou a postačujúcou podmienkou pre preukázanie stavu uviaznutia*. Všetky procesy, ktoré sú v cykle, sú uviaznuté.

Ak systém vlastní väčší počet prostriedkov daného typu, potom cyklus je nutnou, ale nie postačujúcou podmienkou pre uviaznutie. Pre ilustráciu tohoto prípadu sa vrátime k situácii na Obr.

8.1. Predpokladajme teraz, že P3 požaduje jeden prostriedok typu R2. Pretože takýto prostriedok nie je voľný, pridá sa hrana P3-->R2 (Obr. 8.2). V tomto okamihu v grafe existujú dva cykly:

$$P_1 \text{-->} R_1 \text{-->} P_2 \text{-->} R_3 \text{-->} P_3 \text{--> } R_2 \text{-->} P_1$$

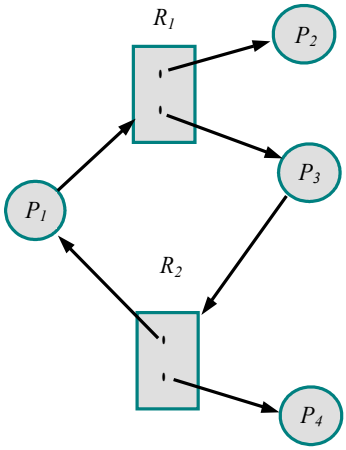
a $P_2 \text{-->} R_3 \text{-->} P_3 \text{-->} R_2 \text{-->} P_2$



Obr. 8.2 Graf prideľovania prostriedkov s uviaznutím

Procesy P_1, P_2 a P_3 sú zablokované - uviaznuté. Proces P_2 čaká na prostriedok R_3, ktorý vlastní proces P_3. Proces P_3 čaká, že buď proces P_1 alebo P_2 uvoľnia prostriedok R_2. A ešte proces P_1 čaká, aby P_2 uvoľnil prostriedok R_1.

Zoberme si ďalší príklad grafu prideľovania prostriedkov na Obr. 8.3. Tu tiež máme cyklus. V tomto prípade ale nenastáva uviaznutie, pretože P_4 po ukončení svojej činnosti uvoľní svoj prostriedok typu R_4, ktorý môže byť potom pridelený P_3 a tým sa cyklus preruší.



Obr. 8.3 Graf prideľovania prostriedkov, ktorý obsahuje cyklus, ale uviaznutie nenastalo

Súhrnom môžeme povedať, že ak v grafe prideľovania prostriedkov nie je slučka, uviaznutie nenastalo. V opačnom prípade uviaznutie môže, ale nemusí nastať.

8.3 Metódy obsluhy uviaznutia

Principiálne máme tri metódy pre obsluhu uviaznutia:

1. Môžeme použiť protokol prideľovania prostriedkov, aby sme zaistili, že uviaznutie *nikdy* nenastane.
2. Môžeme dovoliť systému, aby sa *dostal* do stavu uviaznutia a potom zaistiť jeho *zotavenie.*

3. Môžeme problém *ignorovať* a tváriť sa, že neexistuje. Vo väčšine OS je použité práve toto „riešenie“.

Aby sme zaistili, že sa uviaznutie nevyskytne, musíme zaistiť buď prevenciu alebo vyvarovať sa problému uviaznutia. **Prevencia pred uviaznutím** je množina metód pre zaistenie toho, že aspoň jedna z podmienok pre uviaznutie nebude platiť. Tieto metódy chránia pred uviaznutím tak, že určujú pravidlá, podľa ktorých sa žiada o prostriedky**. Vyvarovanie sa uviaznutiu** na druhej strane vyžaduje, aby operačný systém disponoval s potrebnou dodatočnou informáciou týkajúcou sa prostriedkov, ktoré proces bude požadovať počas svojej existencie. S týmito dodatočnými znalosťami potom môžeme rozhodnúť o každej požiadavke - či bude uspokojená alebo proces musí čakať na jej uspokojenie. Pri spracovaní každej požiadavky systém musí zobrať do úvahy všetky prostriedky, ktoré sú momentálne pridelené procesom, ako aj budúce požiadavky a uvoľnenia prostriedkov.

Ak v systéme nefunguje ani mechanizmus ochrany, ani mechanizmus vyvarovania sa, uviaznutie môže nastať. V každom prostredí systém musí poskytovať algoritmus, ktorý zistí, či uviaznutie nastalo a ak áno, potom poskytnúť aj algoritmus pre zotavenie sa systému z tohoto stavu.

Ak systém neposkytuje ani jeden z už spomenutých prostriedkov, potom je možné, že nastane situácia, kedy nie je možné zistiť, čo sa stalo. Takéto neodhalené uviaznutie môže viesť k poklesu výkonnosti systému kvôli stále väčšiemu počtu prostriedkov, ktoré uviaznuté procesy držia, a stále väčšiemu počtu uviaznutých procesov. Eventuálne sa môže stať, že systém sa úplne zastaví a bude potrebné ho reštartovať manuálne. Aj keď táto metóda nevyzerá byť životaschopná pre riešenie problému uviaznutia, je dosť často používaná. V mnohých systémoch sa uviaznutie vyskytuje zriedkavo (raz za rok), takže toto riešenie je lacnejšie ako prevencia pred uviaznutím alebo vyvarovaním sa pred uviaznutím, alebo detekcia a zotavenie sa. Taktiež sú prípady, kedy systém je v „zmrazenom“ stave, aj keď uviaznutie nenastalo. Je to napr. v systémoch pre reálny čas, keď beží dlhý proces s najvyššou prioritou (alebo pri nepreemptívnom plánovaní) a proces veľmi dlho nevracia riadenie operačnému systému.

8.4 Prevencia pred uviaznutím

Ako sme už poznamenali, aby sa mohlo vyskytnúť uviaznutie, musia platiť súčasne všetky štyri podmienky z časti 8.2.1. Zaistením, že trvale nebude platiť aspoň jedna z podmienok, zaistíme aj prevenciu pred uviaznutím.

8.4.1 Vzájomné vylúčenie

Vzájomné vylúčenie musí platiť aj pre nezdieľateľné prostriedky. Napr. tlačiareň nemôže byť zdieľaná medzi niekoľkými procesmi. Na druhej strane zdieľateľné prostriedky nepotrebujú vylúčenie súčasného prístupu. Pre zdieľateľné prostriedky procesy nečakajú. Vo všeobecnosti sa dá povedať, že prevencia pred uviaznutím sa nedá dosiahnuť zákazom výlučného prideľovania prostriedkov, pretože niektoré prostriedky sú svojou povahou nezdieľateľné.

8.4.2 Vlastniť a žiadať

Aby sme zaistili, že táto podmienka nebude nikdy platiť, musíme zabezpečiť, že kedykoľvek proces bude žiadať o prostriedok, nesmie vlastniť žiadny iný prostriedok.

Jeden z možných spôsobov je, že proces požiada o všetky prostriedky naraz pred svojim zahájením. Tento spôsob sa dá implementovať tak, že systémové volania pre pridelenie prostriedkov budú predchádzať všetky ostatné volania systému.

Iný spôsob je taký, že proces môže žiadať o prostriedok, len ak nevlastní žiadny iný.

Pre znázornenie rozdielov medzi týmito dvomi spôsobmi si zoberme napríklad proces, ktorý kopíruje dáta z magnetickej pásky do súboru na disku, vytriedi ich a potom tlačí výsledok na tlačiarni. V prvom prípade proces bude musieť požiadať o pásku, diskový súbor a tlačiareň hneď na začiatku svojej činnosti. Aj keď tlačiareň potrebuje až na koniec, bude ju držať počas celého svojho vykonania.

V druhom prípade na začiatku proces požiada o pásku a diskový súbor. Skopíruje dáta na disk a uvoľní pásku a súbor. Potom požiada o súbor a tlačiareň a nakoniec uvoľní všetky prostriedky.

Uvedené spôsoby majú dva základné nedostatky. Po prvé je to malé využitie prostriedkov, lebo tie môžu byť pridelené, ale sú dlho nevyužívané. Druhý nedostatok je, že môže nastať starvácia. Proces, ktorý potrebuje často využívaný prostriedok, môže čakať nekonečne dlho, ak aspoň jeden z prostriedkov, ktoré on potrebuje, je sústavne prideľovaný iným procesom.

8.4.3 Zákaz preempcie

Tretia Coffmanova podmienka hovorí, že pridelený prostriedok sa nesmie odobrať. Pre porušenie tejto podmienky môžeme postupovať takto. Ak proces, ktorý vlastní nejaké prostriedky požaduje ďalšie, ktoré mu nemôžu byť pridelené okamžite, potom tie, ktoré vlastní, mu môžu byť odobraté. To znamená, že tieto prostriedky sú implicitne uvoľnené. Odňaté prostriedky sú pridané k prostriedkom, na ktoré proces čaká. Takže proces sa spustí znova keď získa naspäť „staré“ a „nové“ prostriedky, ktoré požadoval.

Iná alternatíva je, že keď proces požiada o nejaké prostriedky, najprv sa testuje, či sú prístupné. Ak áno, pridelia sa. Inak sa testuje, či sú pridelené iným procesom, ktoré tiež čakajú na prostriedky, ktoré držia iné procesy. V takomto prípade sa odoberú prostriedky od čakajúcich procesov a pridelia sa. Ak prostriedky nie sú prístupné a nepatria čakajúcim procesom, potom proces, ktorý žiadal, musí počkať. Pokiaľ proces čaká, niektoré z jeho prostriedkov môžu byť odobraté v prípade, že iný proces o ne požiada. Proces môže pokračovať len keď získa späť odobraté prostriedky a tie, o ktoré žiadal.

8.4.4 Kruhové čakanie

Cestou ako zamedziť platnosť podmienky kruhového čakania, je zoradiť všetky typy prostriedkov a donútiť procesy požadovať prostriedky podľa vzostupného poradia číslovania.

Nech $R = \{ R_1, R_2, ..., R_m \}$ je množina typov prostriedkov. Priradíme každému typu celé číslo, ktoré umožní porovnanie dvoch prostriedkov, aby sme mohli určiť ich poradie podľa zavedeného číslovania. Formálne zadefinujeme funkciu F: R --> N, kde N je množina prirodzených čísel. Napríklad, ak množina typov prostriedkov zahrňuje magnetické pásky, disky a tlačiarne, potom funkcia F môže byť zadefinovaná nasledovne:

F(páska) = 1,

F(disk) = 5,

F(tlačiareň) = 12.

Predpokladáme, že nasledujúci protokol zabezpečí systém proti uviaznutiu:
každý proces žiada o prostriedky len vo vzostupnom poradí číslovania, t.j. ak proces na začiatku požiadal o prostriedok typu R_i, potom môže žiadať len o prostriedky typu R_j, pre ktoré platí $F(R_j) > F(R_i)$. Ak potrebuje proces viac prostriedkov toho istého typu, žiada o všetky naraz. Napríklad, ak funkcia je definovaná dobre, proces ktorý potrebuje pásku a tlačiareň v tom istom čase, najskôr musí žiadať o pásku a potom o tlačiareň.

Alternatívne môžeme požadovať, že kedykoľvek proces požiada o prostriedok typu R_j, musí uvoľniť všetky prostriedky R_i také, že $F(R_i) \geq F(R_j)$.

Ak sa dodržia tieto dva protokoly, podmienka kruhového čakania bude porušená. Môžeme to demonštrovať nasledovne. Majme množinu procesov $\{P_0, P_1, ..., P_n\}$, kde P_i čaká na prostriedok R_j, ktorý vlastní proces P_{i+1}. Pretože proces P_{i+1} vlastní prostriedok R_j a žiada o prostriedok R_{i+1}, môžeme zapísať $F(R_i) < F(R_{i+1})$ pre všetky i. Táto podmienka znamená, že

$$F(R_0) < F(R_1) < ... < F(R_n) < F(R_0).$$

Podľa pravidla tranzitivity platí $F(R_0) < F(R_0)$, čo je nemožné. Takže tu nemôže byť kruhové čakanie.

Funkcia F by mala byť definovaná podľa bežného poradia využitia prostriedkov. Napr. pretože sa páska používa obyčajne pred tlačiarňou, bolo by logické definovať

$$F(\text{páska}) < F(\text{tlačiareň}).$$

8.5 Vyvarovanie sa uviaznutiu

Algoritmy prevencie, ktoré sme prediskutovali v predchádzajúcom odstavci, chránia pred uviaznutím tak, že predpisujú pravidlá, podľa ktorých sa žiada o prostriedky. Tieto pravidlá zaisťujú, že nie je splnená aspoň jedna z podmienok, nutných pre uviaznutie a tým je uviaznutie principiálne nemožné. Uvedené metódy ale majú vplyv na efektívnosť využitia prostriedkov a znižujú priechodnosť systému.

Alternatívnou metódou pre vyhnutie sa uviaznutiu je vyžadovať dodatočnú informáciu o tom, ako budú prostriedky požadované. Napr. ak je v systéme jedna páska a jedna tlačiareň, potom môžeme povedať, že proces P bude požadovať najprv pásku, potom tlačiareň a potom tieto prostriedky uvoľní. Proces Q, zasa bude požadovať najskôr tlačiareň a potom pásku. Na základe informácie o tom, aké prostriedky a v akom poradí budú procesy potrebovať, je možné sa rozhodnúť, či daná konkrétna požiadavka bude uspokojená alebo nie. Pri každej požiadavke sa musí rozhodnúť o jej uspokojení na základe prostriedkov ktoré sú k dispozícii, prostriedkov, ktoré procesy momentálne vlastnia a tých prostriedkov, ktoré procesy budú požadovať a uvoľňovať.

Algoritmy, ktoré zaisťujú vyhnutie sa uviaznutiu sa líšia v tom, koľko a akú informáciu potrebujú. Najjednoduchší model vyžaduje od každého procesu, aby uviedol maximálny počet potrebných prostriedkov z každého typu. Tieto algoritmy potom dynamicky skúmajú stav prideľovania prostriedkov, aby zaistili, že nikdy nenastane kruhové čakanie. Stav prideľovania je daný počtom pridelených prostriedkov, počtom voľných prostriedkov a maximálnym počtom požadovaných prostriedkov.

8.5.1 Stav bezpečný

Stav pridelenia prostriedkov je *bezpečný*, ak systém môže prideliť každému procesu všetky ním požadované prostriedky a pritom sa vyhne stavu uviaznutia. Formálne môžeme povedať, že systém je v bezpečnom stave, keď existuje *bezpečná sekvencia*. Sekvencia procesov $< P_1 , P_2 , ..., P_n >$ je bezpečná sekvencia pre momentálne pridelenie, ak pre každý proces P_i požiadavky, ktoré P_i má, môžu byť uspokojené momentálne prístupnými prostriedkami plus prostriedkami, ktoré vlastnia všetky procesy P_k , pre $k < i$. V tejto situácií, ak prostriedky, ktoré P_i potrebuje, nie sú momentálne prístupné, potom P_i musí počkať, kým procesy P_k ich uvoľnia. Potom ich P_i získa a môže dokončiť svoje vykonanie. Keď P_i skončí, P_{i+1} môže byť dokončený atď. Ak takáto sekvencia procesov neexistuje, stav je *nebezpečný*.

Stav nebezpečný nie je stav uviaznutia, ale stav uviaznutia je stav nebezpečný. Nie všetky nebezpečné stavy vedú do stavu uviaznutia – Obr. 8.7. V bezpečnom stave sa operačný systém môže vyhnúť nebezpečným stavom a stavom uviaznutia. V nebezpečnom stave systém nemôže zaistiť, aby procesy nežiadali o prostriedky a uviaznutie môže nastať.

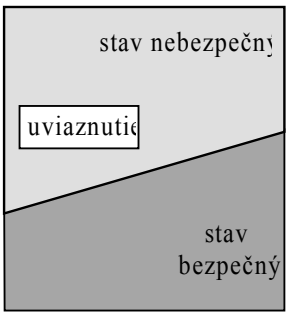
Pre ilustráciu rozoberieme prípad, kedy systém má 12 magnetických pások a 3 procesy: P_0, P_1 a P_2. Proces P_0 požaduje 10 pások, proces P_1 požaduje 4 a proces P_2 - 9 magnetických pások. Predpokladajme, že v čase t_0 proces P_0 vlastní 5 jednotiek, proces P_1 vlastní 2 a proces P_2 - 2 jednotky, 3 jednotky sú voľné.

	Maximálne potreby	Momentálne potreby
P_0	10	5
P_1	4	2
P_2	9	2

V čase t_0 je systém v bezpečnom stave. Postupnosť P_1 , P_0 , P_2 vyhovuje podmienkam bezpečnosti, pretože proces P_1 môže dostať všetky požadované prostriedky a po ukončení ich vráti a systém potom bude mať 5 prístupných pások. Potom proces P_0 môže dostať všetky prostriedky, ktoré požaduje, ukončí sa a nakoniec budú uspokojené aj požiadavky procesu P_2.

Stav systému sa môže vyvinúť aj tak, že prejde do nebezpečného stavu. Napríklad, ak v čase t_1 proces P_2 dostane o 1 pásku naviac, stav systému sa stane nebezpečný. V tomto prípade len proces P_1 dostane všetky požadované prostriedky a po jeho ukončení systém bude mať len 4 prístupné pásky. Pretože proces P_0 má pridelených 5 pások, ale požaduje 10, bude musieť čakať. Obdobne proces P_2 požaduje ďalších 6 pások a tiež musí čakať. Tu už nastalo uviaznutie.

Chyba sa stala pri prideľovaní prostriedkov procesu P_2. Keby sme P_2 nechali počkať, kým ďalšie dva procesy skončia a uvoľnia prostriedky, vyhli by sme sa uviaznutiu.



Obr. 8.7 Stavové priestory stavu bezpečného, nebezpečného a stavu uviaznutia

Na základe koncepcie bezpečného stavu môžeme definovať algoritmus, ktorý zaistí, že systém sa nikdy nedostane do uviaznutia. Na začiatku systém je v bezpečnom stave. Vždy, keď niektorý proces požiada o prostriedok, systém musí rozhodnúť, či mu ten prostriedok pridelí alebo ho nechá čakať. Požiadavke sa vyhovie len vtedy, ak po pridelení prostriedku nevznikne nebezpečný stav.

Pri prideľovaní prostriedkov podľa tejto schémy sa môže stať, že proces bude musieť čakať, aj keď prostriedok je prístupný, čo znamená, že efektívnosť využitia prostriedkov je nižšia ako v opačnom prípade.

8.5.2 Algoritmus bankára

Algoritmus bankára dostal toto meno, pretože sa dá použiť v bankovníctve, kde banka nesmie nikdy požičať celú svoju hotovosť, pretože to môže viesť k stavu, kedy nebude môcť uspokojiť požiadavky svojich klientov.

Každý nový proces pri vstupe do systému musí deklarovať svoje požiadavky pre každý typ prostriedkov. Tento počet samozrejme nesmie prekračovať celkový počet prostriedkov systému. Systém zistí, či uspokojenie požiadaviek procesu ho nedovedie do nebezpečného stavu. Ak tomu tak nie je, proces dostane čo požaduje, inak musí čakať, kým iné procesy neuvoľnia im pridelené prostriedky.

Algoritmus bankára používa nasledovné dátové štruktúry:

- **n** je počet procesov,
    - ***m*** je počet typov prostriedkov v systéme,

- ***prístupné***: vektor s dĺžkou *m*, ktorý obsahuje počty prístupných prostriedkov z každého typu. Ak *prístupné[j]= k*, to znamená, že z prostriedkov typu R_j je k dispozícii *k* jednotiek.
- ***max***: matica *n* x *m* definuje maximálne požiadavky každého procesu. Ak *max[i,j]= k*, potom to znamená, že proces P_i môže požadovať maximálne *k* jednotiek z prostriedkov typu R_j.
- ***pridelené***: matica *n* x *m* definuje počet prostriedkov každého typu, pridelených momentálne procesu P_i . Ak *pridelené[i,j]= k*, potom to znamená, že proces P_i má momentálne pridelených *k* jednotiek prostriedku typu R_j.
- ***zostáva***: matica *n* x *m* , ktorá označuje prostriedky, ktoré ešte musia byť pridelené procesu. Ak *zostáva[i,j]= k*, potom to znamená, že proces P_i potrebuje ešte *k* prostriedkov typu R_j, aby mohol svoju činnosť dokončiť. Všimnite si, že *zostáva[i,j]= max[i,j]- pridelené[i,j].*

Tieto dátové štruktúry môžu meniť v čase svoje rozmery a hodnoty.

Aby sme zjednodušili prezentáciu algoritmu bankára, zavedieme niekoľko pravidiel zápisu.

Nech X a Y sú vektory dĺžky n. Hovoríme, že X≤Y, vtedy a len vtedy ak X[i] ≤ Y[i] pre všetky i = 1, 2, ...n. Napr. ak X = (1, 7, 3, 2) a Y = (0, 3, 2, 1), potom $Y \leq X$. $Y < X$ ak $Y \leq X$ a $Y \neq X$.

Riadky matíc *pridelené* a *zostáva* môžeme brať ako vektory a budeme ich označovať $pridelené_i$ a $zostáva_i$. Vektor $pridelené_i$ špecifikuje všetky prostriedky pridelené procesu P_i a vektor $zostáva_i$ špecifikuje všetky prostriedky, ktoré proces P_i ešte potrebuje dostať do svojho ukončenia.

8.5.3 Algoritmus pre určenie stavu systému

Kroky algoritmu pre určenie stavu systému sú nasledujúce:

1. Nech *pracovné* a *dokončené* sú vektory s dĺžkou *m* resp *n*. Počiatočné hodnoty sú:
   pracovné:= prístupné a *dokončené [i]:= false* pre *i* =1, 2, ..., *n*.

2. Nájdeme *i* také, že *dokončené [i] = false* a *zostáva[i] ≤ pracovné.* Ak také *i* nie je, ideme na krok 4.

3. Priradíme *pracovné := pracovné + pridelené*$_i$
   dokončené [i] :=true

Ideme na krok 2.

4. Ak *dokončené [i] = true* pre všetky *i*, potom systém je v bezpečnom stave!

Tento algoritmus vyžaduje $\boldsymbol{m \times n^2}$ operácií pre nájdenie odpovede o stave systému.

8.5.4 Algoritmus pre vyžiadanie prostriedku

Nech $požiadavka_i$ je vektor požiadaviek procesu P_i . Ak $požiadavka_i[j]= k,$ potom proces P_i požaduje *k* jednotiek prostriedku typu R_j. Keď proces požiada o prostriedky, podniknú sa nasledujúce kroky:

1. Ak $požiadavka_i \leq zostáva[i]$, ideme na krok 2. Inak vzniká chybový stav, pretože proces prekročil svoje maximálne požiadavky.

2. Ak požiadavkai ≤ prístupnéi , ideme na krok 3. Inak Pi musí čakať, pretože prostriedky nie sú prístupné.

3. Predstierame, že systém pridelil požadované prostriedky procesu Pi tak, že modifikujeme stav takto:

   prístupné := prístupné - požiadavka $_i$

   pridelené[i] := pridelené[i] + požiadavka $_i$

   zostáva[i] := zostáva[i] - požiadavka $_i$

Ak výsledný stav je bezpečný, transakcia sa dokončí a proces P_i dostane požadované prostriedky, inak proces musí čakať a obnoví sa pôvodný stav.

Príklad

Máme systém s 5-timi procesmi a tromi prostriedkami typu *A, B, C*. Prostriedok typu *A* má 10 jednotiek, prostriedok typu *B* - 5 jednotiek a typ *C* má 7 jednotiek. V čase t_0 stav systému je nasledujúci:

	pridelené			***max***			***prístupné***		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
$\boldsymbol{P_0}$	0	1	0	7	5	3	3	3	2
$\boldsymbol{P_1}$	2	0	0	3	2	2			
$\boldsymbol{P_2}$	3	0	2	9	0	2			
$\boldsymbol{P_3}$	2	1	1	2	2	2			
$\boldsymbol{P_4}$	0	0	2	4	3	3			

Obsah matice *zostáva* je definovaný ako *max* - *pridelené* a je:

procesy	***zostáva***		
	A	*B*	*C*
$\boldsymbol{P_0}$	7	4	3
$\boldsymbol{P_1}$	1	2	2
$\boldsymbol{P_2}$	6	0	0
$\boldsymbol{P_3}$	0	1	1

Systém je v bezpečnom stave, pretože postupnosť P_1, P_3, P_4, P_2, P_0 vyhovuje kritériám bezpečnosti.

Predpokladajme ďalej, že proces P_1 požaduje ešte jednu jednotku z prostriedkov typu *A* a dve jednotky z prostriedkov typu C, takže $požiadavka_1 = (1, 0, 2)$. Aby sme rozhodli, či môžeme požiadavke vyhovieť, najskôr skontrolujeme či $požiadavka_1 \leq prístupné$ (t.j. $(1, 0, 2) \leq (3, 3, 2)$), čo je splnené. Predpokladáme ďalej, že sme vyhoveli požiadavke a systém prichádza do tohto stavu:

	pridelené			***zostáva***			***prístupné***		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
$\boldsymbol{P_0}$	0	1	0	7	4	3	2	3	0
$\boldsymbol{P_1}$	3	0	2	0	2	0			
$\boldsymbol{P_2}$	3	0	2	6	0	0			
$\boldsymbol{P_3}$	2	1	1	0	1	1			
$\boldsymbol{P_4}$	0	0	2	4	3	1			

Teraz musíme určiť, či tento nový stav je bezpečný. Za týmto účelom vykonáme algoritmus pre určenie stavu systému, pričom zistíme, že postupnosť P_1, P_3, P_4, P_0, P_2 vyhovuje podmienkam bezpečnosti a pridelíme požadované prostriedky procesu P_1.

8.6 Detekcia uviaznutia

Ak do operačného systému nie je zahrnutý algoritmus na prevenciu alebo algoritmus vyhnutia sa uviaznutia, potom uviaznutie môže nastať. V takom prípade systém musí poskytnúť:

- algoritmus, ktorý preskúma stav systému a určí, či nastalo uviaznutie,
- algoritmus pre zotavenie sa z uviaznutia.

Ďalej rozoberieme prípad, kedy systém vlastní len jednu jednotku z každého typu prostriedkov a prípad, kedy týchto jednotiek je viacej.

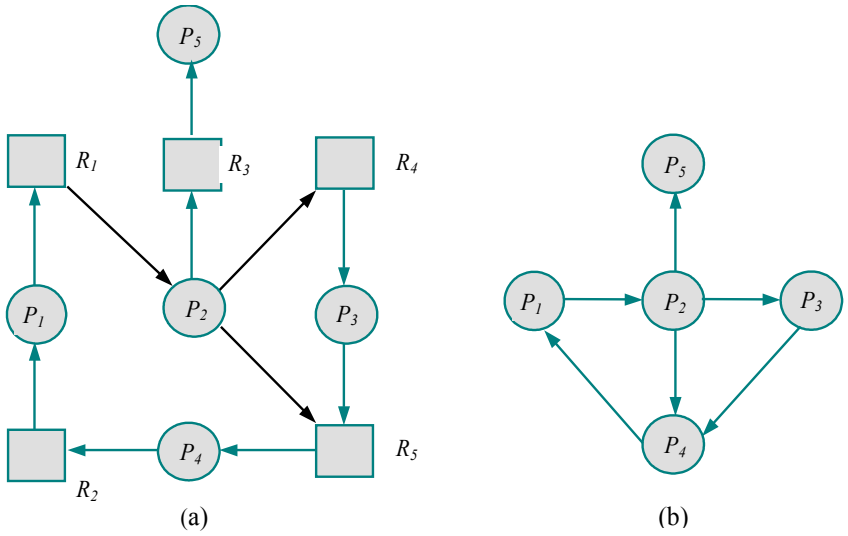
8.6.1 Jedna jednotka z každého typu

Ak všetky typy prostriedkov majú iba po jednej jednotke, môžeme použiť na detekciu uviaznutia variant grafu prideľovania prostriedkov, ktorý sa nazýva *čakací graf*. Takýto graf získame z grafu prideľovania prostriedkov vypustením uzlov, ktoré prezentujú typ prostriedku a spojením príslušných hrán.

Hrana z P_i do P_j v čakacom grafe vyjadruje to, že proces P_i čaká na proces P_j, aby uvoľnil prostriedok, ktorý P_i potrebuje. Hrana P_i -->P_j existuje v čakacom grafe vtedy a len vtedy, keď zodpovedajúci graf prideľovania prostriedkov obsahuje dve hrany P_i -->R_q a R_q -->P_j pre niektorý prostriedok. Na Obr. 8.9 je ukázaný graf prideľovania prostriedkov a zodpovedajúci čakací graf.

Ako v predchádzajúcich prípadoch, uviaznutie existuje vtedy a len vtedy, ak v grafe je cyklus. Pre odhalenie uviaznutia systém musí udržiavať čakací graf a periodicky vyvolávať algoritmus pre hľadanie cyklu v ňom.

Algoritmus pre odhalenie cyklu v grafe vyžaduje rádovo n^2 operácií, kde n je počet uzlov v grafe.



Obr. 8.9 a) Graf prideľovania prostriedkov, b) Čakací graf

8.6.2 Niekoľko jednotiek z každého typu

Systém, ktorý vlastní niekoľko jednotiek z každého typu prostriedkov nemôže použiť čakací graf. V takomto prípade sa používa algoritmus popísaný ďalej. Algoritmus používa niekoľko dátových štruktúr, ktoré sa menia v čase a podobajú sa tým, ktoré využíva algoritmus bankára.

- *prístupné:* vektor s dĺžkou *m*, ktorý označuje počet prístupných prostriedkov z každého typu.
  - *pridelené*: matica *nxm*, ktorá definuje počet prostriedkov z každého typu, ktoré sú momentálne pridelené každému procesu.
  - *zostáva:* matica *nxm*, ktorá definuje počet prostriedkov z každého typu, ktoré ešte požaduje každý proces.

Aj tu, podobne ako v algoritme bankára, budeme riadky matice *pridelené* a *zostáva* považovať za vektory a budeme ich označovať *pridelené[i] a zostáva[i]*. Algoritmus detekcie, ktorý tu popisujeme, jednoducho skúma každú možnú postupnosť prideľovania pre procesy, ktoré ešte majú byť dokončené.

1. Nech *pracovné* a *pridelené* sú vektory s dĺžkou *m* resp *n*.
   Inicializujeme *pracovné := prístupné.*

Pre *i = 1, 2, ..., n* platí *dokončené = false*, ak *pridelené*$_i \neq 0$, inak sa nastaví *dokončené = true.*

2. Nájdeme index *i* pre ktorý platí:
   dokončené = false a *požadované[i]* $\leq$ *pracovné*

   Ak taký index neexistuje, ideme na krok 4.

3. *pracovné := pracovné + pridelené[i]*
   dokončené[i]:= true

   Ideme na krok 2.

4. Ak *dokončené[i] = false* pre niektoré *i,* $1 \leq i \leq n$, potom systém je v stave uviaznutia. Navyše, ak *dokončené[i] = false,* potom práve proces *P i* je uviaznutý.

Tento algoritmus vyžaduje rádovo $\boldsymbol{mxn^2}$ operácií pre odhalenie stavu uviaznutia.

Pre ilustráciu použitia tohoto algoritmu ukážeme systém s 5-timi procesmi a tromi typmi prostriedkov. Prostriedok typu A má 7 jednotiek, prostriedok typu B má 2 jednotky, prostriedok typu C - 6 jednotiek. V čase t_0 je stav systému nasledujúci:

	pridelené			***zostáva***			***prístupné***		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Požadujeme, aby bol systém v bezpečnom stave. Skutočne, ak vykonáme algoritmus zistíme, že sekvencia procesov P_0, P_2, P_3, P_1, P_4 končí tak, že *dokončené[i] =true* pre všetky i.

Ďalej predpokladáme, že proces P_2 požiada dodatočne o jednu jednotku typu C. Matica *pridelené* sa zmení takto:

	zostáva		
	A	*B*	*C*
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

Zistíme, že systém je v stave uviaznutia. Aj keď prostriedky procesu P_0 sa vrátia k prístupným prostriedkom, ich počet nie je dostatočný pre uspokojenie potrieb ostatných procesov. Takže procesy P_1, P_2, P_3 a P_4 sú v stave uviaznutia.

Pri použití algoritmu pre detekciu uviaznutia je potrebné zodpovedať otázku, ako často sa vyskytuje uviaznutie. Ak sa uviaznutie vyskytuje často, algoritmus by sa mal vyvolávať častejšie. Vyvolanie algoritmu pri každej požiadavke prostriedku by viedlo k veľkým režijným stratám času procesora. Menej stratový je variant vyvolania algoritmu napr. raz za hodinu, alebo vždy, keď efektívnosť využitia procesora klesne pod 40%.

8.7 Zotavenie sa z uviaznutia

Keď nastane uviaznutie, je možné postupovať niekoľkými spôsobmi. Jednou z možností je upovedomiť operátora, že tento stav nastal a on zaistí obsluhu. Druhou možnosťou je, že systém sa sám automaticky dostane z uviaznutia. Pre zrušenie uviaznutia môžeme zrušiť jeden alebo viac procesov, aby sme prerušili kruhové čakanie procesov, alebo druhou možnosťou je, že odoberieme jeden alebo viac prostriedkov procesom, ktoré sú v stave uviaznutia.

8.7.1 Ukončenie procesu

Pre zrušenie stavu uviaznutia môžeme postupovať dvoma spôsobmi:

- ukončíme všetky procesy, ktoré sú v stave uviaznutia - táto metóda je radikálna, ale vedie k veľkým stratám, pretože je možné, že niektoré procesy bežali dlhší čas a ich čiastočné výsledky budú stratené. V takomto prípade sa tieto procesy budú musieť neskôr spustiť znova od začiatku.
    - ukončujeme procesy po jednom, kým sa nevylúči cyklus. Táto metóda vedie k veľkým režijným stratám času procesora, pretože vždy, keď sa ukončí proces, sa musí vyvolať algoritmus pre detekciu uviaznutia, ktorý zistí, či systém je stále v tomto stave.

Ukončenie procesu nemusí byť vždy jednoduché. Proces môže napr. práve pracovať so súborom a po jeho ukončení súbor zostane v nekonzistentnom tvare. Tiež po ukončení procesu, ktorý práve tlačí, systém musí resetovať tlačiareň.

Výber procesu, ktorý sa má ukončiť, je obyčajne dosť zložitý. Do úvahy sa berie nielen cena opätovného spustenia procesu, ale aj dodatočné kritériá, ako sú: priorita procesu, dĺžka predchádzajúceho výpočtu, počet a typ prostriedkov ktoré proces vlastní, koľko prostriedkov ešte proces bude potrebovať do svojho ukončenia, či proces je dávkový alebo interaktívny a iné. Podľa toho, ktoré z týchto kritérií sa zoberie ako primárne, dajú sa stanoviť rôzne taktiky pre ukončenie uviaznutých procesov.

8.7.2 Odňatie prostriedku

Keď sa pre zotavenie sa z uviaznutia použije metóda odobratia prostriedku, musia sa zobrať do úvahy tieto fakty:

- výber obete, t.j. procesu ktorému sa odoberie prostriedok. Aj v tomto prípade musíme zobrať do úvahy „cenu“. Faktory, ktoré túto cenu určujú sú: počet prostriedkov, ktoré proces vlastní, a čas, ktorý proces spotreboval do okamihu uviaznutia.
    - návrat späť (rollback) - keď procesu zoberieme nasilu prostriedok, musíme mu umožniť pokračovať z nejakého známeho bezpečného stavu. To znamená, že takéto stavy by sa mali počas behu procesu zaznamenávať, čo nie je vždy možné a výhodné. Najjednoduchšie riešenie je zrušiť proces a spustiť ho znova.
    - pri odoberaní prostriedkov sa môže stať, že odoberieme prostriedok stále tomu istému procesu, čím spôsobíme jeho starváciu. Aby takáto situácia nenastala, je potrebné stanoviť presný počet odobratí prostriedkov jednému procesu.

8.8 Kombinovaný prístup

Výskumníci sa zhodli na tom, že žiadny z uvedených spôsobov ošetrenia stavu uviaznutia nie je dostatočný pre vyriešenie všetkých problémov, ktoré sa vyskytujú pri prideľovaní prostriedkov v operačných systémoch.

Jedno prijateľné riešenie je kombinovať všetky tri prístupy - prevenciu, vyhnutie sa uviaznutiu a detekciu. Toto riešenie je založené na rozdelení prostriedkov na triedy a aplikáciu najvhodnejšieho

spôsobu obsluhy uviaznutia na každú triedu. Uvedieme príklad systému, ktorý má štyri triedy prostriedkov:

- **interné prostriedky** - prostriedky, ktoré využíva systém, napr. riadiaci blok procesu alebo V/V kanály. Prevenciu môžeme docieliť pomocou hierarchického zoradenia prostriedkov.
  - ***operačná pamäť*** - prevenciu môžeme docieliť preempciou, pretože úloha vždy môže byť prerušená, odložená na disk a spustená neskôr.
  - ***prostriedky, ktoré sa prideľujú úlohe,*** ako napr. magnetická páska alebo súbor. Tu môžeme použiť taktiku vyhnutia sa uviaznutiu, pretože proces by mal dopredu požiadať o prostriedky, ktoré bude potrebovať.
  - ***swapovací priestor*** - priestor na záložnej pamäti (disku), ktorý úloha využíva pri výmene. Tu sa využíva predbežné prideľovanie, pretože maximálne nároky úlohy sú obyčajne známe.

9 SPRÁVA PAMÄTE

V tejto kapitole preberieme rôzne spôsoby správy pamäte, od najjednoduchších až po stránkovanie a segmentáciu. Každý prístup má svoje prednosti a nedostatky. Výber spôsobu správy pamäte pre určitý systém závisí od mnohých faktorov, hlavne od HW platformy systému. Ako uvedieme ďalej, mnoho algoritmov správy pamäte vyžaduje HW podporu.

9.1 Pozadie

Pamäť je základným prvkom moderných počítačových systémov. Je to veľké pole adresovateľných slov alebo bajtov. Procesor vyberá inštrukcie z pamäte podľa hodnoty počítadla inštrukcií (Programm Counter - PC). Tieto inštrukcie môžu spôsobiť ďalšie čítanie operandov z pamäte alebo uloženie výsledkov na určité pamäťové miesta.

Typický cyklus vykonania inštrukcie začína výberom inštrukcie z pamäte. Inštrukcia sa dekóduje a v dôsledku toho je možné, že bude potrebné zaviesť ďalšie operandy z pamäte. Po vykonaní inštrukcie nad operandami výsledok sa uloží späť do pamäte. Je zrejmé, že pamäťová jednotka pracuje s prúdom pamäťových adries a nevie, ako tie adresy boli vygenerované (počítadlom inštrukcií, indexovaním, inštrukciou alebo inak) a aký majú význam (inštrukcie alebo dáta).

9.1.1 Pripojenie fyzických adries

Obyčajne sa program nachádza na disku vo vykonateľnom tvare. Program musí byť zavedený do pamäte a vykonaný v rámci procesu. Podľa použitého algoritmu správy pamäte, môže byť proces počas vykonania presúvaný z pamäte na disk. Množina procesov, ktoré sú pripravené na presunutie do pamäte, tvorí vstupný front.

Obyčajný postup je vybrať jeden z pripravených procesov vo fronte a umiestniť ho do pamäte. Počas svojho vykonania proces pracuje s inštrukciami a dátami z pamäte. Keď ukončí svoje vykonanie, jeho pamäťový priestor sa vráti k voľnej pamäti.

Mnoho systémov dovoľuje používateľským procesom sídliť v ľubovoľnej časti fyzickej pamäte. Aj keď adresný priestor počítača začína od 00000, prvá adresa procesu nemusí byť 00000. To znamená, že sa adresy v používateľskom programe menia podľa miesta uloženia v pamäti.

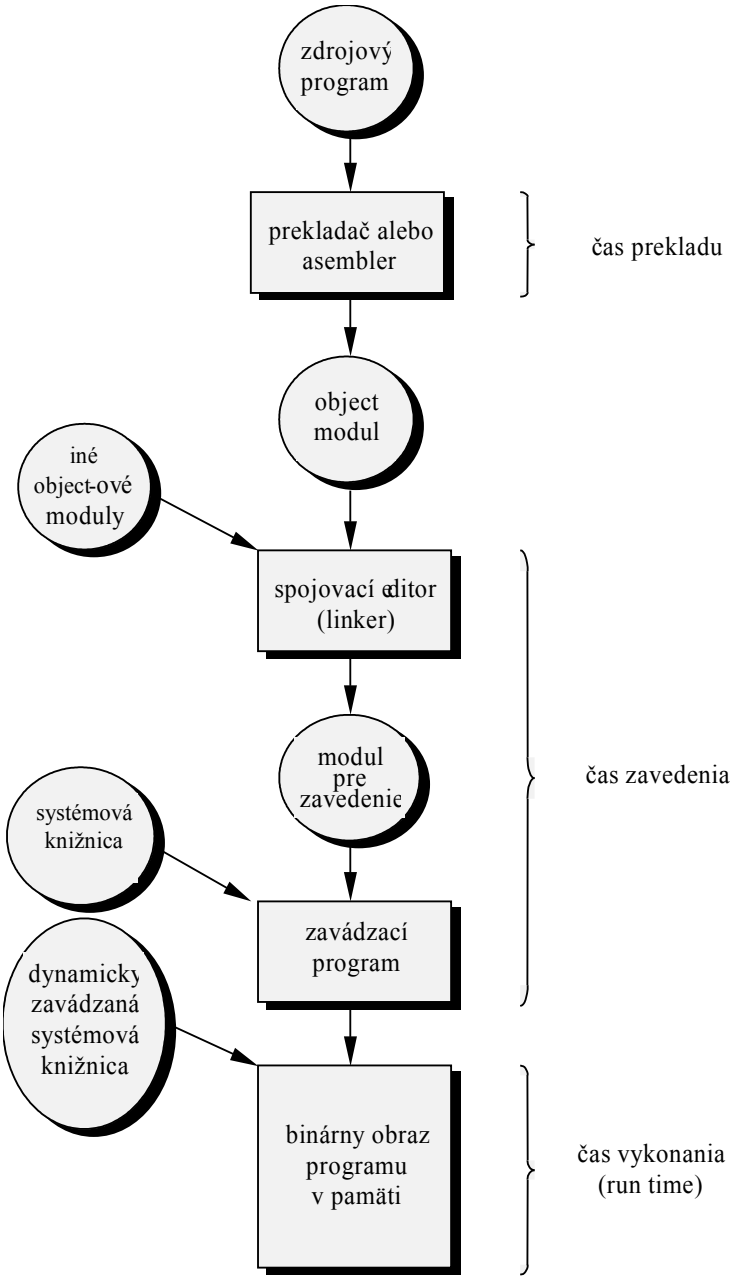
Obyčajne je používateľský program pred spustením spracovaný v niekoľkých etapách. Počas tohoto spracovania sú adresy v programe reprezentované rôznymi spôsobmi (Obr. 9.1).

Adresy v zdrojovom programe sú obyčajne symbolické. Prekladač spája tieto symbolické adresy s relokovateľnými adresami (ako napr. „14 bajtov od začiatku tohoto modulu“). Spojovací editor alebo zavádzací program (loader) premení tieto relokovateľné adresy na absolútne (napr. 74014). Každé pripojenie adries je vlastne mapovanie jedného adresného priestoru do druhého.

Pripojenie inštrukcií a dát k pamäťovým adresám sa môže vykonať v každom z nasledujúcich krokov:

- **Počas prekladu**: Ak počas prekladu je známe, kde v pamäti bude proces umiestnený, generuje sa absolútny kód. Napr. ak vieme, že proces sa uloží od adresy R, potom kód vygenerovaný prekladačom bude obsahovať adresy od R ďalej. Samozrejme, ak sa počiatočná adresa zmení, program sa musí opätovne preložiť. Príklad: programy MS DOS-u typu .COM majú absolútne adresy pridelené počas kompilácie.
    - **Počas zavádzania**: Ak počas prekladu nie je známe, kde v pamäti bude proces umiestnený, generuje sa *relokovateľný* kód. V tomto prípade konečné pripojenie adries je odložené až do zavedenia programu do pamäte. Ak sa zmení počiatočná adresa, je potrebné len opätovne zaviesť používateľský kód, aby sa odzrkadlila táto zmena na adresách programu
    - **Počas vykonania**: Ak proces počas vykonania bude presúvaný z jedného pamäťového segmentu do iného, potom pripojenie adries sa musí uskutočniť až počas behu programu. Pre tento spôsob pripojenia fyzických adries je potrebná HW podpora.

Ďalej rozoberieme, ako sa dajú uvedené metódy pripojenia adries efektívne implementovať, ako aj potrebnú HW podporu pre každú z nich.



Obr. 9.1 Kroky spracovania používateľského programu

9.1.2 Dynamické zavádzanie

Pre efektívnejšie využitie pamäte môžeme použiť dynamické zavádzanie. Pri dynamickom zavádzaní podprogramy sa nezavádzajú do pamäte, kým nie sú volané. Do pamäte sa zavedie hlavný program a spustí sa. Keď hlavný program volá podprogram, najprv sa pozrie či volaný podprogram je v pamäti. Ak nie je, zavolá sa zavádzací program, aby zaviedol požadovaný podprogram a modifikoval tabuľku adries programu tak, aby odzrkadľovala túto zmenu. Potom sa riadenie odovzdá programu, ktorý bol práve zavedený.

Prednosť dynamického zavádzania je, že nepoužívané podprogramy sa nikdy nezavedú do pamäte. Tento spôsob je veľmi užitočný, keď veľké časti programu sú potrebné pre obsluhu zriedkavých prípadov, ako napr. podprogramy pre obsluhu chýb. V takomto prípade celkový rozmer programu môže byť veľký, ale skutočné používaná (a zavedená do pamäte) časť je oveľa menšia.

Dynamické zavádzanie nevyžaduje špeciálnu podporu zo strany operačného systému. Je na používateľovi, aby využil prednosti tejto schémy. Operačný systém môže pomôcť poskytnutím knižničných procedúr pre implementáciu dynamického zavádzania.

9.1.3 Dynamické spojenie

Na Obr. 9.1 sú ukázané aj *dynamicky spojované knižnice*. Väčšina operačných systémov podporuje len *statické spojovanie*, podľa ktorého systémové knižnice jazykov sa berú ako každý iný objektový modul a tak sú zahrňované do binárneho obrazu programu. Koncepcia dynamického spojovania je podobná dynamickému zavádzaniu. Namiesto odsunutia zavedenia na čas vykonania, odsúva sa spojenie. Tento spôsob dynamického spojenia je obyčajne využitý v súvislosti so systémovými knižnicami, ako napr. pri knižniciach podprogramov programovacích jazykov. Bez tejto možnosti by všetky programy potrebovali kópiu svojej jazykovej knižnice (alebo aspoň kópie procedúr volaných v programe) zahrnutú do vykonateľného obrazu. Táto požiadavka plytvá diskovým a pamäťovým priestorom. Pri dynamickom spojovaní je do obrazu programu zahrnutý *stub* (koreň, pahýľ). Stub je malý úsek kódu, ktorý ukazuje, ako lokalizovať príslušnú knižničnú procedúru rezidentnú v pamäti, alebo ako zaviesť knižnicu, ak potrebná procedúra ešte nie je v pamäti .

Keď sa vykonáva *stub*, ten najprv otestuje či potrebná rutina je už v pamäti. Ak nie je, zavedie ju do pamäte. Potom *stub* nahradí sám seba adresou rutiny a vykoná ju. Takže keď sa vykonáva tento kódový segment druhýkrát, knižničná rutina sa vykonáva priamo, bez časových strát pre dynamické spájanie. Podľa tejto schémy všetky procesy, ktoré používajú knižnicu jazyka, vykonávajú len jednu kópiu jej kódu.

Dynamické spájanie na rozdiel od dynamického zavádzania vyžaduje podporu od operačného systému. Ak procesy v pamäti sú chránené pred prístupom iných procesov, potom operačný systém je jediný, ktorý môže otestovať, či potrebná rutina je v adresnom priestore iného procesu, ktorý môže povoliť viacnásobný prístup do toho istého pamäťového priestoru.

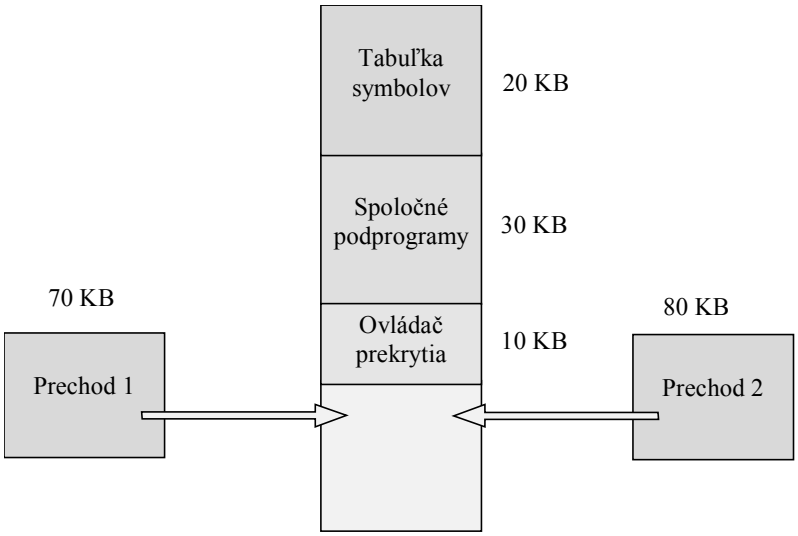
9.1.4 Prekrývanie

Doteraz sme diskutovali len o prípade, kedy celý program a dáta procesu sú počas vykonávania procesu súčasne uložené v pamäti. Veľkosť procesu je obmedzená veľkosťou fyzickej pamäte. Takže proces, ktorý požaduje väčšie množstvo pamäte ako má pridelené, niekedy môže použiť techniku nazvanú prekrývanie (overlay). Idea prekrývania spočíva v tom, že v pamäti sú len tie inštrukcie a dáta, ktoré sú potrebné v danom čase. Keď sú potrebné iné inštrukcie, tie sa zavedú do toho istého miesta, kde boli predchádzajúce.

Ako príklad zoberme dvojprechodový asembler. Počas prvého prechodu sa vybuduje tabuľka symbolov, potom počas druhého prechodu sa generuje strojový kód. Takýto asembler sa dá rozdeliť na kód prechodu 1, kód prechodu 2, tabuľku symbolov a spoločné pomocné podprogramy, ktoré sa používajú počas prechodu 1 a 2. Predpokladajme, že veľkosti týchto komponentov sú nasledovné:

Prechod 1	70 KB
Prechod 2	80 KB
Tabuľka symbolov	20 KB
Spoločné podprogramy	30 KB

Aby sme zaviedli všetko naraz, potrebujeme 200 KB pamäte. Ak máme k dispozícií len 150 K, nemôžeme spustiť proces. Avšak všimnime si, že prechod 1 a 2 nemusia byť súčasne v pamäti. Takže zadefinujeme dve prekrytia: prekrytie A je tabuľka symbolov, spoločné podprogramy a prechod 1, prekrytie B je tabuľka symbolov, spoločné podprogramy a prechod 2.



Obr. 9.2 Prekrytia v dvojprechodovom asembleri

Pridáme ovládač prekrytia a spustíme prekrytie A. Keď sa ukončí, skočí sa na ovládač prekrytí, ktorý načíta do pamäte prekrytie B na miesto, kde bolo prekrytie A, čím ho prekryje a odovzdá mu riadenie. Prekrytie A požaduje len 120 KB pamäte a prekrytie B potrebuje 130 KB pamäte (Obr. 9.2). Takto môžeme spustiť asembler na 150K pamäte. Samozrejme vykonanie bude o niečo pomalšie, lebo bude potrebná jedna V/V operácia navyše pre načítanie prekrytia B.

Kódy jednotlivých prekrytí sa uchovávajú na disku ako absolútne pamäťové obrazy a v prípade potreby ich načítava ovládač prekrytí. Pre spracovanie prekrytí sú potrebné špeciálne relokačné a spojovacie algoritmy.

Tak ako aj v prípade dynamického zavádzania, prekrývanie nevyžaduje špeciálnu podporu od operačného systému. Používateľ ho môže kompletne implementovať pomocou jednoduchých súborov, čítania zo súboru do pamäte a skoku do tejto pamäte, kde sa vykonajú načítané inštrukcie.

Pre programátora je to ťažšia úloha, lebo musí dokonale poznať štruktúru veľkého programu (malý program nepotrebuje prekrytie) a na základe toho rozvrhnúť jednotlivé prekrytia. Kvôli týmto ťažkostiam sa v dnešnej dobe táto technika používa len pre mikropočítače alebo iné systémy s obmedzenou fyzickou pamäťou, ktoré nemajú HW podporu pre progresívnejšie techniky. Niektoré prekladače ponúkajú podporu pre tvorbu prekrytí. Samozrejme, ak máme možnosť, využijeme inú techniku, kde sa tieto problémy riešia automaticky bez účasti používateľa.

9.2 Logický a fyzický adresný priestor

Adresy generované procesorom sa obecne nazývajú *logické adresy* a adresy, ktoré používa pamäťová jednotka (tie ktoré sa zavádzajú do registra pamäťových adries), sa obecne nazývajú *fyzické adresy*.

Ak používame priraďovanie fyzických adries počas prekladu alebo počas zavádzania, potom logické a fyzické adresy sú rovnaké. Avšak pri priraďovaní adries počas behu programu sú logické a fyzické adresy odlišné. Množina logických adries, ktoré sú generované programom, je známa pod pojmom logický adresný priestor; množina fyzických adries, ktoré odpovedajú týmto logickým adresám, je známa pod pojmom fyzický adresný priestor.

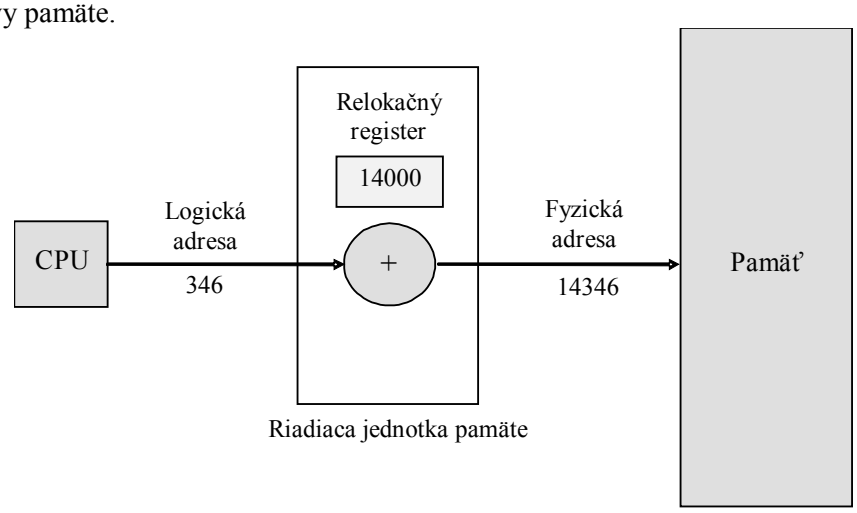
Mapovanie logických adries na fyzické počas behu programu vykonáva jednotka správy pamäte (Memory Management Unit - MMU), ktorá je HW zariadenie. Existuje množstvo rôznych schém mapovania, ktoré budú rozobrané neskôr. Tu pre ilustráciu uvedieme jednoduchú schému mapovania s použitím bázového registra.

Ako je ukázané na Obr. 9.3, táto schéma vyžaduje HW podporu - *bázový* alebo *relokačný* register. Hodnota relokačného registra sa pričíta ku každej adrese, ktorá je generovaná programom. Napr. ak hodnota relokačného registra je 14000, potom pokus používateľa adresovať pamäťové

miesto 0 sa dynamicky pretransformuje na adresu 14000, podobne pokus o dosiahnutie adresy 346 sa mapuje na adresu 14346. Operačný systém MS DOS pre procesory rady Intel 80X86 využíva 4 relokačné registre pri zavádzaní a spustení programov.

Používateľský program, pri použití tejto schémy, nikdy „nevidí“ skutočný adresný priestor. Program môže vytvoriť ukazovateľ na adresu 346, manipulovať s ním, porovnávať s inými adresami stále ako 346. Až keď sa použije ako pamäťová adresa, prepočíta sa relatívne voči bázovému registru. Používateľský program narába s logickými adresami, ktoré sa prevádzajú na fyzické, až keď sa urobí odkaz na danú adresu. Logické adresy sa pohybujú v intervale od (0 po *max)* a fyzické v intervale od R+0 po R+*max*.

Koncepcia logického adresného priestoru, ktorý je mapovaný do fyzického adresného priestoru, tvorí základ správy pamäte.



Obr. 9.3 Dynamická relokácia využívajúca relokačný register

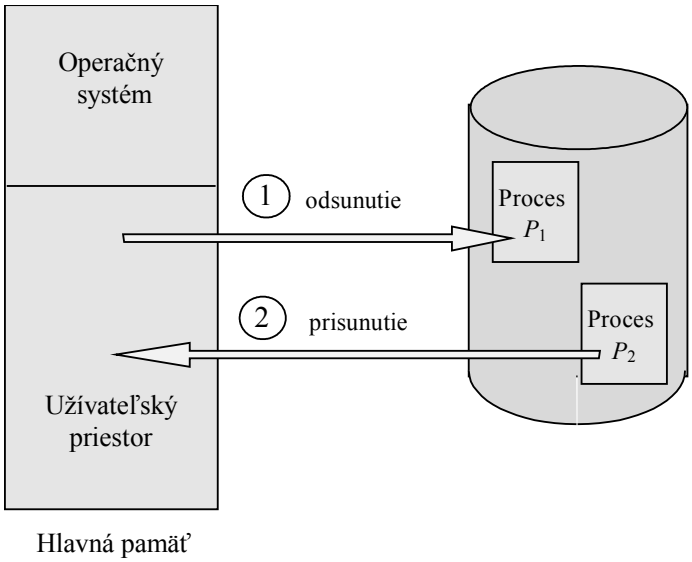
9.3 Swapovanie

Aby proces mohol byť vykonaný, musí byť uložený v pamäti. Počas vykonania ale proces môže byť dočasne odsunutý z pamäte na disk a neskôr znova vrátený do pamäte. Napríklad, predpokladajme multiprogramové prostredie s implementáciou cyklického plánovania. Keď vyprší kvantum procesu, správa pamäte začne s presúvaním procesu a prisúvaním ďalšieho procesu (Obr. 9.4.). Medzitým plánovač bude alokovať časové kvantum niektorému inému procesu v pamäti. Keď každý proces využije svoje kvantum, bude nahradený iným procesom. V ideálnom prípade správca pamäte stačí presúvať procesy tak rýchlo, že vždy keď plánovač preplánováva proces, ten je v pamäti .

Variantom tejto swapovacej techniky je plánovanie na základe priority. Ak do systému príde proces s vyššou prioritou, správca pamäte môže odsunúť proces s nižšou prioritou a zaviesť tam proces s vyššou prioritou. Keď sa tento proces ukončí, zavedie sa znova proces s nižšou prioritou.

Normálne proces, ktorý bol odsunutý, sa prisunie späť do toho istého priestoru, kde bol predtým. Toto obmedzenie je kvôli priradeným adresám. Ak mapovanie bolo uskutočnené počas kompilácie alebo zavádzania, proces sa nemôže zaviesť do iného pamäťového priestoru. Ak mapovanie sa uskutočňuje dynamicky, proces sa môže zaviesť aj inde.

Swapovanie vyžaduje záložný priestor. Je to obyčajne rýchly disk. Ten musí byť dostatočne veľký, aby sa doň zmestili obrazy všetkých používateľských procesov a musí poskytovať priamy prístup k týmto obrazom. Systém udržuje front pripravených procesov, ktorý obsahuje všetky procesy, ktorých pamäťové obrazy sú na disku a sú pripravené na spustenie. Kedykoľvek sa plánovač rozhodne vykonať proces, volá dispečera. Dispečer kontroluje, či ďalší proces z frontu je v pamäti. Ak nie je a nie je voľné miesto v pamäti, dispečer odsúva niektorý z procesov v pamäti a prisúva požadovaný proces. Potom obnoví obsahy registrov a odovzdáva riadenie vybranému procesu.



Obr. 9.4. Swapovanie dvoch procesov pomocou disku ako záložnej pamäte

Je jasné, že čas pre prepnutie kontextu pri takomto swapovaní je veľmi veľký. Pre priblíženie si týchto údajov, predpokladajme používateľský proces s veľkosťou 100 KB a disk so štandardnou rýchlosťou 1 MB/s. Prenos procesu z/do pamäte zaberie:

100/1000 KB za sekundu = 1/10 s = 100 ms

Za predpokladu, že nie je potrebné nastavenie hlavy disku a priemerné oneskorenie je 8 ms, čas pre swapovanie je 108 ms. Pretože je potrebné presúvať aj z pamäte aj do pamäte, celkový swapovací čas je 216 ms.

Pre efektívne využitie procesora je potrebné, aby swapovací čas bol relatívne krátky vzhľadom na čas vykonania procesu. Takže napr. pri cyklickom plánovaní musí byť časové kvantum podstatne dlhšie ako 216 ms.

Všimnime si, že hlavná časť swapovacieho času je čas prenosu. Celkový čas prenosu je priamo úmerný množstvu swapovanej pamäte. Ak máme počítačový systém s 1 MB pamäte a rezidentný OS, ktorý zaberá 100 KB, maximálna veľkosť používateľského procesu je 900 KB. Avšak veľa procesov je menších. 100 KB proces môže byť presunutý za 108 ms, zatiaľ čo 900 KB za 908 ms. Je užitočné vedieť presne dopredu, koľko pamäte používa proces a ktoré časti boli použité. Potom čas swapovania sa dá zredukovať swapovaním len použitých častí.

Na swapovanie sú kladené aj ďalšie obmedzenia. Ak chceme odsunúť proces, musíme mať istotu, že momentálne nevykonáva nič. Veľmi dôležité sú nedokončené V/V operácie. Môže sa stať, že budeme chcieť odsunúť z pamäte proces, ktorý čaká na dokončenie V/V operácie. Ak V/V operácia pristupuje asynchrónne k používateľskej pamäti cez V/V bufre, proces nemôže byť odsunutý. Predpokladajme, že zariadenie je obsadené a V/V operácia je odložená do frontu. Takže, ak odsunieme proces P_1 a prisunieme proces P_2, V/V operácia sa môže pokúsiť použiť pamäť, ktorá teraz patrí procesu P_2. Sú možné dve riešenia tohoto problému:

1. Nikdy neodsúvať proces s nedokončenými V/V operáciami.

2. Vykonávať V/V operácie len cez bufre operačného systému.

Prenosy medzi pamäťou, ktorá patrí operačnému systému, a procesom sa uskutočňujú len keď je proces v pamäti.

Predpoklad, že pri swapovaní nie je potrebné nastavenie hláv disku, potrebuje ďalšie vysvetlenie, ktoré čitateľ nájde v kapitole o správe diskových zariadení**.** Obecne sa dá povedať, že

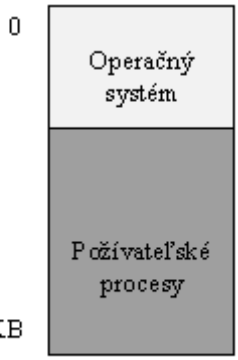
swapovací priestor zaberá na disku zvláštnu časť, ktorá nepatrí do súborového systému, a jeho použitie je navrhnuté tak, aby bolo čo najrýchlejšie.

Modifikácia swapovania je využívaná vo viacerých verziách Unix-u. Swapovanie sa normálne nevykonáva stále. Akonáhle v systéme beží viac procesov, ktoré majú k dispozícií minimálne množstvo pamäte, swapovanie sa spustí a beží, kým nepoklesne záťaž systému.

Personálne počítače postrádajú lepší HW, ktorý umožňuje implementovať progresívnejšie metódy správy pamäte, a preto swapovanie je najvhodnejší spôsob ako spúšťať viac veľkých procesov. Napríklad Microsoft Windows podporuje paralelné vykonávanie procesov v pamäti. Ak vznikne nový proces a v pamäti nie je dosť miesta, starší proces je odsunutý na disk. Tento operačný systém ale neposkytuje úplné swapovanie, pretože čas kedy swapovať určuje používateľ (kliknutím na príslušné okno) a nie plánovač. Ďalší operačný systém z tejto rady - Windows NT využíva progresívnejšie vlastnosti jednotky správy pamäte (Memory Management Unit - MMU), ktoré moderné PC už majú.

9.4 Súvislé prideľovanie pamäte

Do operačnej pamäte sa musia uložiť aj operačný systém aj používateľské procesy. Pamäť je obyčajne rozdelená na dve časti - do jednej sa umiestni rezidentný operačný systém a do druhej sa umiestnia používateľské procesy. Operačný systém môže začínať hneď od adresy 0 - prípad znázornený na Obr. 9.5, alebo môže byť umiestnený na konci. Hlavný faktor, ktorý ovplyvňuje toto umiestnenie, je adresa vektora prerušenia. Pretože vektor prerušenia je častejšie v dolnej časti pamäte, aj operačný systém sa tam častejšie ukladá.



Obr. 9.5 Rozdelenie pamäte

9.4.1 Prideľovanie jedného úseku

Prideľovanie jedného úseku je najjednoduchšia technika správy pamäte pri ktorej sa všetkým procesom prideľuje ten istý úsek (Obr. 9.5). Jadro systému, ako už bolo povedané, je buď na začiatku alebo na konci operačnej pamäte.

Táto technika prideľovania je typická pre monoužívateľské systémy bez paralelného spracovania (CP/M, MS-DOS). Principiálne paralelné spracovanie je možné s použitím techniky swapovania, ale takýto systém by mal príliš veľkú réžiu. V prípadoch, kedy vyhradený úsek je nedostatočný pre používateľský proces, je možné použiť techniku prekrývania segmentov.

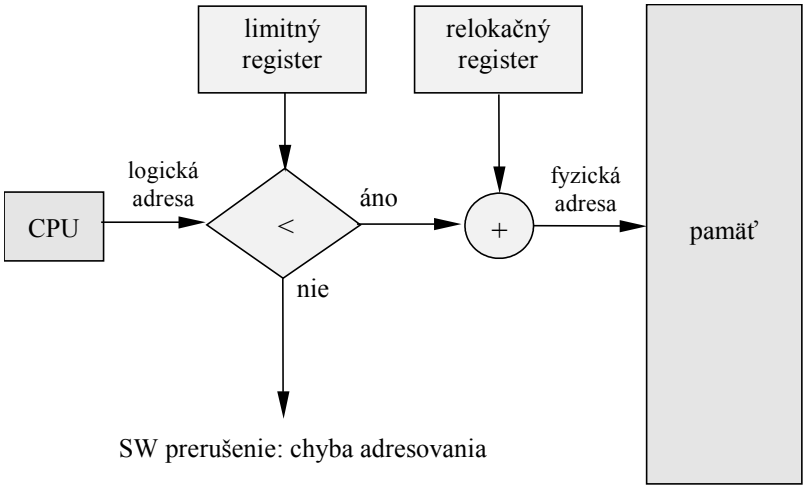
9.4.2 Multiprogramovanie a správa pamäte

Počas svojho historického vývoja tvorcovia operačných systémov sa neustále snažili zvýšiť efektívnosť využitia času procesora jednak pre uspokojenie viacerých používateľov súčasne, jednak pre zaplnenie času, kedy procesy čakajú na dokončenie V/V operácií. Keď v pamäti počítača je viac programov naraz, hovoríme o *multiprogramovaní* a počet procesov, ktoré sa nachádzajú súčasne v pamäti, určuje *úroveň multiprogramovania*.

9.4.3 Prideľovanie viacerých súvislých úsekov s pevnou dĺžkou

Jednoduchým riešením problému zvýšenia stupňa multiprogramovania sa javí rozdelenie pamäte na viac súvislých úsekov s pevnou dĺžkou, kedy každý úsek obsahuje len jeden proces. To znamená, že úroveň multiprogramovania je daná počtom úsekov. Veľkosť úsekov sa určuje buď počas generovania systému, alebo počas jeho zavádzania do pamäte.

Ochrana pamäte pri prideľovaní viacerých úsekov s pevnou dĺžkou musí zabezpečiť ochranu jadra pred náhodným alebo zámerným zásahom zo strany používateľských procesov. Taktiež aj procesy musia byť chránené jeden pred druhým. V tomto prípade ochrana sa dá uskutočniť pomocou *relokačného registra*, ktorý bol spomenutý v časti 9.2 spolu s *limitným registrom.* Limitný register obsahuje oblasť platných adries príslušného procesu. Každá logická adresa sa kontroluje podľa schémy na Obr. 9.6. Ak je adresa menšia ako je adresa uložená v limitnom registri, pripočíta sa k nej obsah relokačného registra, a tak sa získa adresa platná pre proces. Ak je logická adresa väčšia, vygeneruje sa software-ové prerušenie (trap), ktoré oznámi chybu v adresovaní.



Obr. 9.6 HW podpora pre relokačný a limitný register Keď plánovač vyberie proces na vykonanie, dispečer zavedie do relokačného a limitného registra ich aktuálne hodnoty ako súčasť prepnutia kontextu procesu. Táto technika prideľovania pamäte bola pôvodne použitá v systéme IBM OS/360 a je známa pod názvom MFT (Multiprogramming with a Fixed number of Tasks). Okrem ochrany pamäte relokačný register poskytuje aj flexibilnosť v prípadoch, keď sa veľkosť jadra mení. Môžu to byť prípady, kedy sa napr. pridáva nový ovládač do jadra.

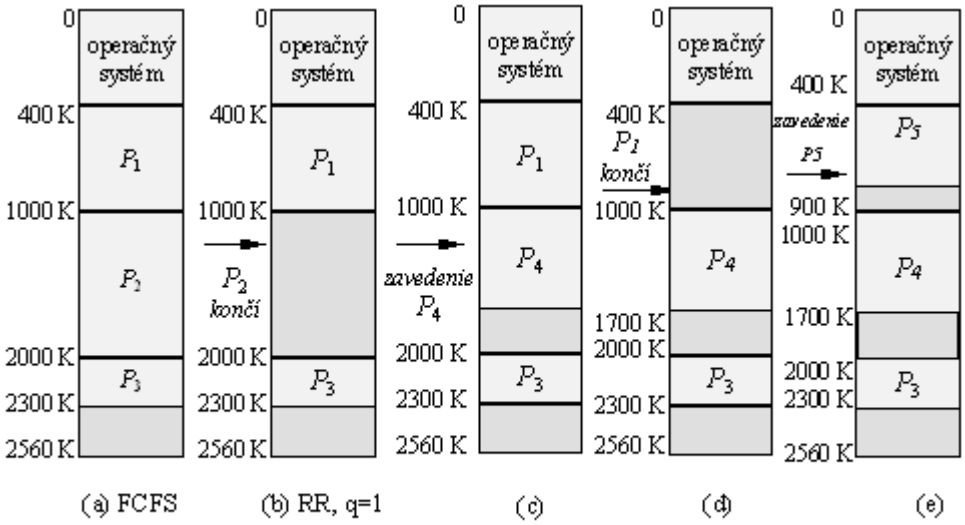
9.4.4 Prideľovanie súvislých úsekov s premenlivou dĺžkou

V praxi sa prideľovanie súvislých úsekov s pevnou dĺžkou ukazuje ako neefektívne, lebo sa veľmi plytvá pamäťovým priestorom pre uloženie menších procesov do väčších úsekov a vzniká vnútorná fragmentácia, t.j. vznikajú úseky pamäte, nepoužiteľné pre ďalší proces. Preto sa používa iný algoritmus prideľovania, kedy sa prideľujú úseky s premenlivou dĺžkou.Na Obr. 9.7 a 9.8 sú ukázané rôzne situácie pri prideľovaní úsekov s premenlivou dĺžkou. Hlavný rozdiel spočíva v tom, že rozmery úsekov sa menia dynamicky s veľkosťou vznikajúcich procesov. Problémom však je komplikovanejší výber vhodného úseku a tiež udržanie informácie o vznikajúcich úsekoch. Tu problém vnútornej fragmentácie neexistuje, ale prideľovaním úsekov s premenlivou dĺžkou vznikajú úseky, ktoré nie sú využiteľné a je potrebné tento problém riešiť *striasaním.*

Technika prideľovania viacerých súvislých úsekov s pevnou dĺžkou (v literatúre je známa pod názvom MVT a pôvodne bola používaná pri spracovaní dvkových úloh) je veľmi blízka inej technike, ktorá sapoužíva v prostredí so zdieľaním času (time-sharing) a nazýva sa čistá segmentácia. Táto technika bude popísaná neskôr.

Na nasledujúcich obrázkoch je uvedený príklad plánovania procesov, keď rezidentný operačný systém zaberá 400 KB z celkovej pamäte, ktorá je 2560 KB. Front pripravených procesov je ukázaný

na Obr.9.7.Ak pre plánovanie procesov použijeme algoritmus FCFS, po pridelení pamäte procesom P_1, P_2 a P_3 vznikne situácia, ktorá je znázornená na Obr 9.8 (a). Zostala „diera“ o veľkosti 260 KB, ktorá nemôže byť použitá pre ďalšie procesy.



Obr. 9.8 Prideľovanie pamäte a dlhodobé plánovanie

V každom okamihu je potrebné, aby operačný systém mal informácie o voľných úsekoch pamäte a o procesoch, ktoré požadujú spracovanie. Z frontu pripravených procesov sa vyberie proces a hľadá sa vhodný úsek pre jeho umiestnenie. Ak sa nájde väčší úsek, proces sa tam umiestni a zvyšok úseku sa zaznamená do zoznamu voľných úsekov. Ak sa nenájde dostatočne veľký úsek, proces musí počkať, kým sa vhodný úsek uvoľní, a medzitým sa z frontu vyberie iný proces pre spracovanie. Keď proces skončí, jeho úsek sa vráti do zoznamu voľných úsekov a skontroluje sa, či úseky v zozname neležia vedľa seba, aby sa mohli spojiť do väčšieho úseku.

Dôležitú úlohu pre efektívnosť prideľovania úsekov pamäte s premenlivou dĺžkou zohráva algoritmus výberu vhodného úseku pre umiestnenie procesu. Existuje veľa riešení tohoto problému. Zoznam voľných úsekov sa prehľadáva podľa niektorého z nasledujúcich algoritmov:

- *Prvý vhodný (First-fit)* - proces sa umiestni do prvého úseku, ktorý je dostatočne veľký. Prehľadávanie môže začať buď na začiatku zoznamu, alebo na mieste, kde skončilo predchádzajúce hľadanie.
  - *Najlepšie vyhovujúci (Best-fit)* - procesu sa pridelí najmenší úsek, do ktorého sa zmestí.
  - *Najhoršie vyhovujúci (Worst-fit)* - opäť ako v prvom prípade sa prehľadáva celý zoznam voľných úsekov, aby sa našiel úsek, v ktorom po uložení procesu zostane najväčšia voľná časť.

Simulácie ukazujú, že prvé dva algoritmy sú lepšie ako algoritmus *worst-fit* čo sa týka času potrebného na prehľadávanie a využitia pamäte. *First-fit* a *best-fit* sú rovnako dobré vo využití pamäte, ale *first-fit* je v obecnom prípade rýchlejší.

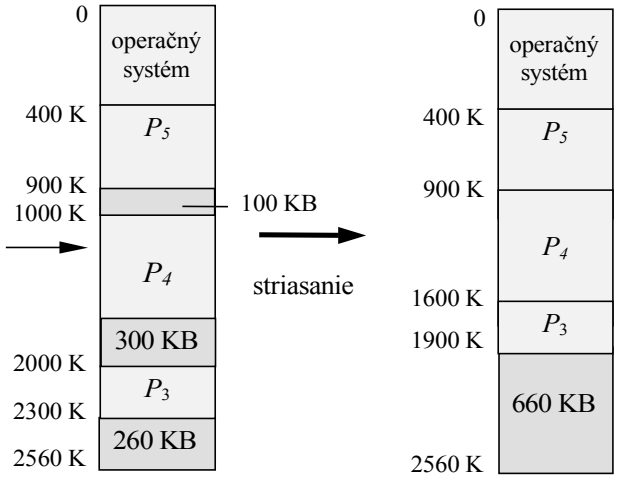
9.4.5 Vonkajšia a vnútorná fragmentácia

Algoritmus prideľovania pamäte po úsekoch s premenlivou dĺžkou spôsobuje *vonkajšiu fragmentáciu*. To znamená, že medzi jednotlivými úsekmi, kde sú umiestnené procesy, zostávajú úseky, ktoré nie je možné použiť pre iné procesy, lebo sú malé. Súčet veľkostí jednotlivých „fragmentov“ je dostatočne veľký, ale problém spočíva v tom, že nie sú súvislé. Príklad z Obr.9.8 je typický, kedy kvôli fragmentácii nie je možné prideliť procesu pamäť, aj keď súčet veľkostí voľných úsekov je väčší ako požiadavka procesu.

Podľa veľkosti pamäte a priemernej veľkosti procesov, externá fragmentácia môže byť dosť veľkým problémom. Napr. štatistická analýza algoritmu first-fit ukazuje, že aj s niektorými

optimalizáciami, pri *N* pridelených blokoch kvôli fragmentácii *sa stráca 0.5*N* blokov. To znamená, že až polovica pamäte môže byť stratená!

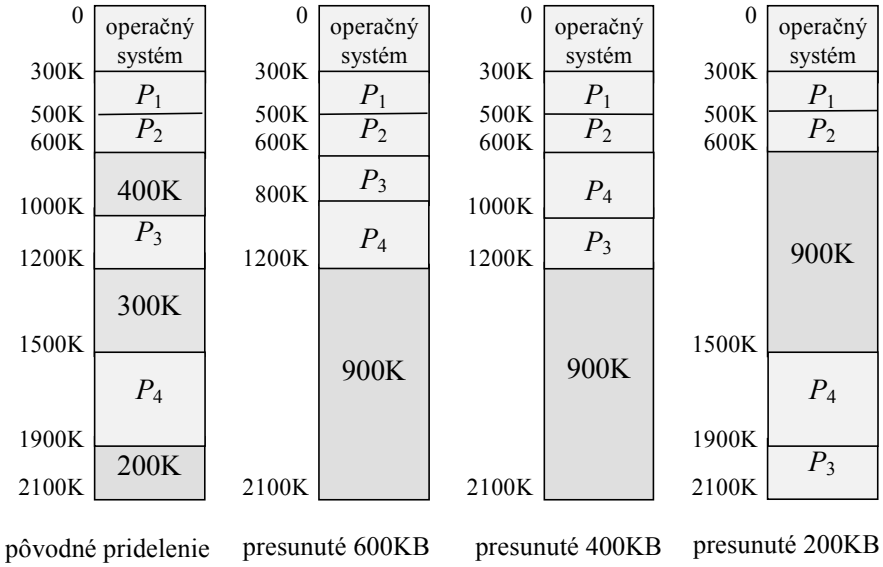
Z uvedeného vyplýva, že fragmentácia sa musí riešiť. Jedno z riešení je *striasanie*. Účelom striasania je spojiť dohromady fragmenty do jedného väčšieho bloku. Napr. situácia z príkladu na obr.9.8(e) sa striasaním dá vyriešiť tak, ako je ukázané na Obr. 9.9.



Obr. 9.9 Striasanie

Striasanie nie je vždy možné. Ak sa proces musí presunúť z jedného miesta v pamäti do druhého, je treba zmeniť aj adresy v programe. Ak adresy programu boli pripájané počas kompilácie alebo počas zavádzania, a nie dynamicky, potom sa striasanie nedá uskutočniť. Striasanie je možné, len ak pripojenie adries programovým inštrukciám a dátam sa deje počas vykonávania programu.

Striasanie sa uskutočňuje v dvoch krokoch - najskôr sa presunú dáta a inštrukcie, potom sa zmenia hodnoty v bázových registroch, aby odzrkadľovali nové umiestnenie programov. Striasanie je náročné na čas, pretože veľké úseky pamäte sa presúvajú inde. Preto je potrebný starostlivý výber algoritmu striasania. Na Obr. 9.10 je ukázaných niekoľko možných riešení danej situácie, pričom rozdiely vo veľkosti presúvaných úsekov sú dosť veľké. Ak presunieme procesy P_3 a P_4 smerom k nižším adresám, získame voľný úsek o veľkosti 900 KB, pričom budeme musieť presunúť 600 KB. Ak presunieme len proces P_4 do voľného úseku pred P_3, budeme musieť presunúť 400 KB, a ak presunieme proces P_3, smerom k vyšším adresám, budeme musieť presunúť len 200 KB.



Obr. 9.10 Porovnanie niekoľkých rôznych spôsobov kompresie pamäte

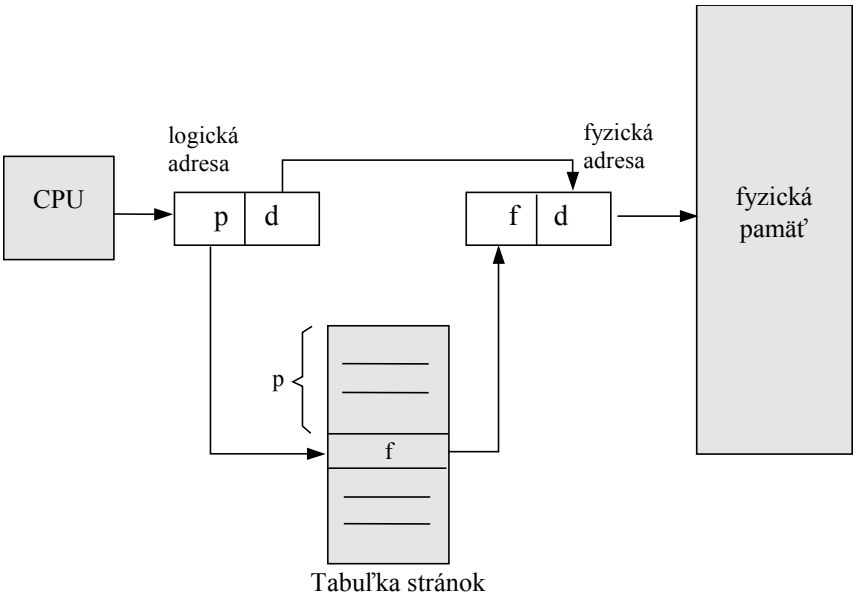
Striasanie sa môže skombinovať so swapovaním. Proces môže byť odsunutý z pamäte, čím sa jeho úsek uvoľní a môže byť pridelený inému procesu. Ak systém disponuje relokačným a limitným registrom, proces môže byť neskôr umiestnený aj inde a pritom sa môže urobiť striasanie. Ak tomu tak nie je, proces sa musí vrátiť na pôvodné miesto.

9.5 Stránkovanie

Iné riešenie vonkajšej fragmentácie je mapovanie súvislého logického adresného priestoru do nesúvislého fyzického priestoru. Táto metóda rieši aj problémy s hľadaním vhodného úseku pamäte pre proces, ako aj hľadanie vhodného úseku pre jeho uloženie na disk.

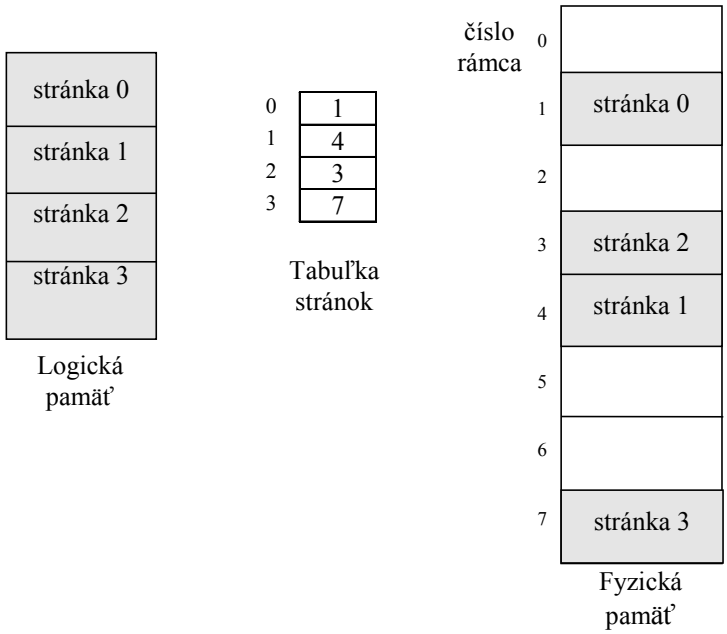
1.1 Princíp

Pri stránkovaní je fyzická pamäť rozdelená na časti s pevnou veľkosťou, nazvaných *rámce*. Logický adresný priestor procesu je rozdelený na rovnako veľké bloky, nazvané *stránky*. Stránka nesúvisí s logickou štruktúrou programu. Keď sa proces vykonáva, jeho stránky sa z disku presunú do voľných rámcov v operačnej pamäti. HW podpora pre stránkovanie je ukázaná na Obr. 9.11.



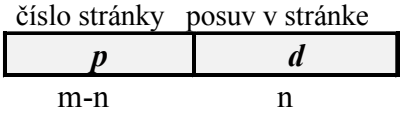
Obr. 9.11 Stránkovací hardvér

Každá adresa vygenerovaná procesorom je rozdelená na dve časti - *číslo stránky (p)* a *posuv* v rámci stránky *(d)*. Číslo stránky sa používa ako index v tabuľke stránok. V tabuľke stránok sú zaznamenané odpovedajúce počiatočné adresy stránok vo fyzickej pamäti. Počiatočná adresa stránky spolu s posuvom v rámci stránky vytvára skutočnú fyzickú adresu, ktorá sa posiela jednotke správy pamäte. Model stránkovania je ukázaný na Obr. 9.12.



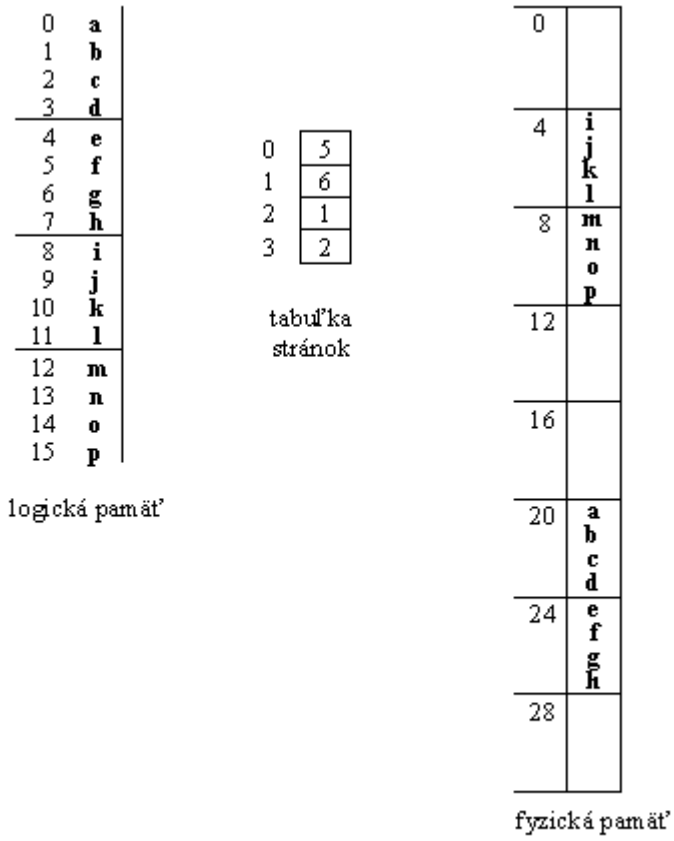
Obr. 9.12 Stránkovací model logickej a fyzickej pamäte

Veľkosť stránky a rámca závisia od HW. Rozmer stránky je obyčajne mocnina 2 a mení sa od 512 bajtov do 8 KB v závislosti od architektúry počítača. Výber mocniny 2 pre veľkosť stránky veľmi uľahčuje výpočet fyzickej adresy. Ak rozmer fyzického adresného priestoru je 2^m a veľkosť stránky je 2^n (bajtov alebo slov), potom vyššie *m-n* bity logickej adresy určujú číslo stránky a *n* nižších bitov určuje posuv v stránke. Logická adresa má nasledovné položky:



kde *p* je index do tabuľky stránok a *d* je posuv v stránke.

Konkrétny, aj keď neskutočný príklad je uvedený na Obr. 9.13. Použitá je stránka s veľkosťou 4 bajty a fyzická pamäť o veľkosti 32 bajtov t.j. 8 stránok. Logická adresa 0 je v stránke 0 a má posuv 0. Keď použijeme index do tabuľky stránok, zistíme, že táto stránka sa nachádza v rámci číslo 5. To znamená, že logická adresa 0 sa mapuje do fyzickej adresy 20 t.j. ((5x4) + 0). Logická adresa 3 (stránka 0, posuv 3) sa mapuje do fyzickej adresy 23 t.j. ((5x4)+3). Logická adresa 4 je v stránke 1 s posuvom 0 a podľa tabuľky stránok sa mapuje do rámca 6. Jej zodpovedajúca fyzická adresa je 24 t.j. ((6x4)+0).



Obr. 9.13 Príklad stránkovania pre 32-bitovú pamäť so 4-bajtovou stránkou

Pozorný čitateľ si určite všimol, že stránkovanie je forma dynamickej relokácie. Každá logická adresa je viazaná cez stránkovací HW na niektorú fyzickú adresu. Vlastné stránkovanie sa dá prirovnať k relokácii, kde pre každý rámec je použitý iný relokačný register.

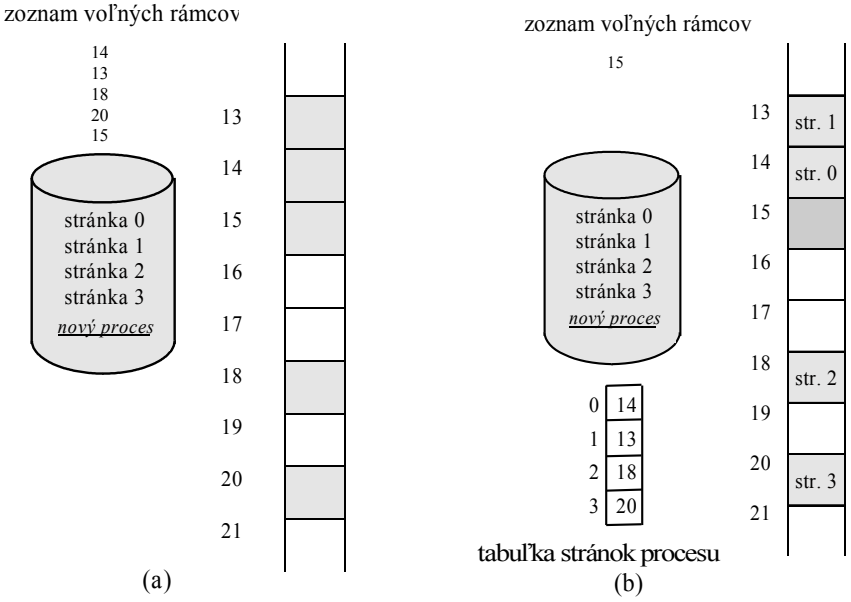
Pri použití stránkovania neexistuje vonkajšia fragmentácia. Každý voľný rámec je pridelený procesu, ktorý ho potrebuje. Ale vnútorná fragmentácia nie je odstránená. Rámce sa prideľujú ako celé jednotky. Ak pamäťové požiadavky procesu nezodpovedajú presne celému násobku stránok, posledná stránka nebude plná. Napr. ak stránka má 2048 bajtov a proces vyžaduje 72766 bajtov, bude potrebovať 35 stránok plus 1086 bajtov. Dostane pridelených 36 rámcov a v poslednej stránke bude nevyužitých 962 bajtov.

Najhorší je prípad, keď proces potrebuje *n* stránok plus 1 bajt. Bude mu pridelených *n+1* stránok, čo znamená, že posledná stránka bude skoro úplne prázdna. Tento príklad navádza k

myšlienke, že je lepšie používať menšie stránky. Na druhej strane, menšie stránky vedú k zväčšeniu tabuľky stránok a k predĺženiu časov prenosov z/na disk (efektívnejšie sú väčšie bloky). Veľkosti stránok sa pohybujú od 512 B do 8 KB. Moderné počítače majú stránky od 2 KB do 4 KB.

Keď sa proces naplánuje pre vykonanie, najskôr sa preskúma jeho veľkosť v stránkach. Ak proces potrebuje *n* stránok, musí byť toľko voľných stránok, aby sa proces mohol uložiť do pamäte. Stránky procesu sa postupne zavedú do voľných rámcov a ich čísla sa zaznamenajú do tabuľky stránok – Obr. 9.14

Dôležitý aspekt stránkovania je jasný rozdiel medzi pohľadom používateľa na pamäť a skutočnou fyzickou pamäťou. Pre používateľa pamäť je súvislá a obsahuje len jeho program. V skutočnosti on dostáva nesúvislý priestor a zdieľa pamäť s inými procesmi. Spojovníkom týchto pohľadov je HW, ktorý vykonáva dynamickú transformáciu adries. Tento proces je skrytý používateľovi a je vykonávaný operačným systémom. Všimnite si, že pri stránkovaní proces z princípu nemôže pristúpiť k stránke, ktorá mu nepatrí. On nemá prostriedky pre adresovanie pamäte mimo tabuľky stránok a táto tabuľka zasa obsahuje len jeho vlastné stránky.



Obr. 9.14 Voľné rámce a) pred pridelením b) po pridelení

Operačný systém spravuje operačnú pamäť a musí stále uchovávať informáciu o pridelených rámcoch, voľných rámcoch atď. Obyčajne sa táto informácia udržiava v tabuľke nazvanej *tabuľka rámcov*. Tabuľka rámcov má jednu položku pre každý rámec, kde je zaznamenané, či je rámec voľný alebo pridelený a ak je pridelený, ktorému procesu patrí.

Dodáme ešte, že operačný systém pre každú logickú adresu musí vytvoriť odpovedajúcu fyzickú adresu. Ak používateľ volá systém (napr. pre V/V operáciu), ako parameter volania dodáva adresu (bufra) a táto adresa musí byť preložená do korektnej fyzickej adresy. Operačný systém udržiava kópiu tabuľky stránok každého procesu, ako aj počítadla inštrukcií a registrov. Tieto kópie sa používajú vždy, keď OS potrebuje prepočítať logickú na fyzickú adresu „ručne“. Používajú sa aj vtedy, keď dispečer musí definovať hardvérovú tabuľku stránok, keď proces dostane pridelený čas procesora. To všetko znamená, že stránkovanie zväčšuje čas prepínania kontextu procesu.

1.2 Štruktúra tabuľky stránok

1.2.1 HW podpora stránkovania

Každý operačný systém má svoju metódu ukladania tabuľky stránok. Väčšinou sa alokuje tabuľka pre každý proces. Ukazovateľ na tabuľku, ako aj hodnoty registrov sú uložené do riadiaceho bloku procesu.

HW implementácia tabuľky stránok sa môže urobiť mnohými rôznymi spôsobmi. Najjednoduchší spôsob je použitie **sady registrov pre tabuľku stránok**. Tieto registre by mali byť veľmi rýchle, aby zefektívnili transformáciu adries. Každý prístup k pamäti musí ísť cez transformáciu adries, takže jej rýchlosť má veľký vplyv na efektívnosť. Dispečer zavádza tieto registre, ako aj ostatné univerzálne registre. Inštrukcie pre zavádzanie a pre modifikáciu týchto registrov sú privilegované, takže len operačný systém môže meniť mapu pamäte.

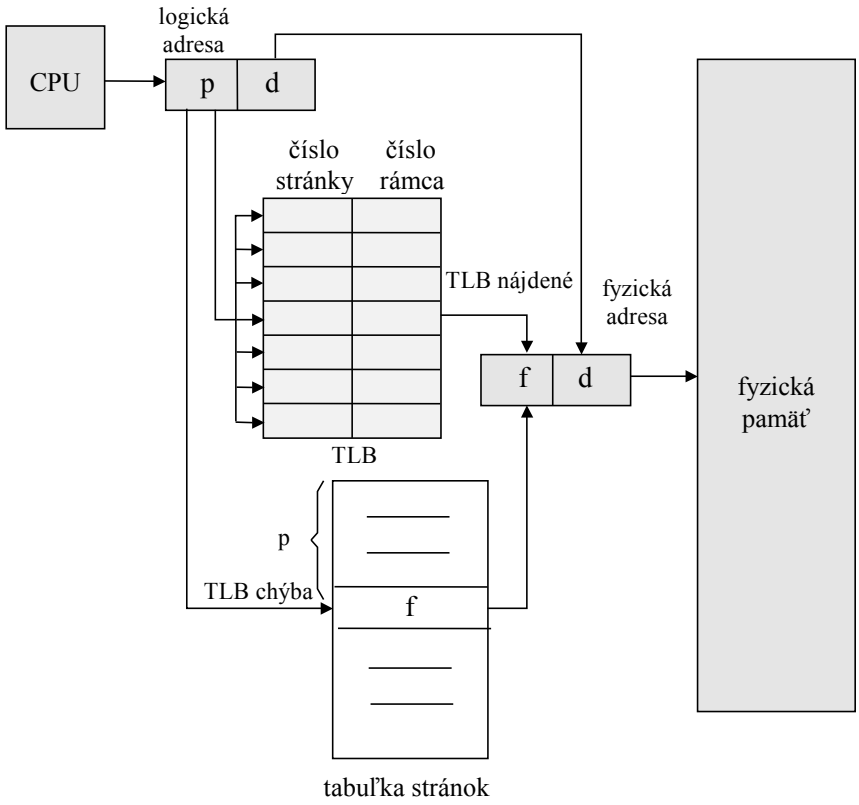
Použitie registrov pre uloženie tabuľky stránok je vhodné vtedy, keď tabuľka je relatívne malá (napr. 256 položiek). Veľa počítačov ale dovoľuje, aby tabuľka bola veľmi veľká, napr. 1000000 položiek. Pre tieto počítače nie je vhodné uloženie tabuľky do registrov. Tam sa **tabuľka stránok udržiava v pamäti a jej adresa sa nachádza v *registri tabuľky stránok*** (Page-Table Base Register, *PTBR)*. Výmena tabuľky stránok potom vyžaduje len zmenu obsahu tohoto registra, čo značne redukuje čas prepnutia kontextu.

Problémom pri tomto prístupe je čas potrebný pre prístup k adrese používateľskej pamäte. Ak chceme prístup k adrese na stránke ***i,*** najprv musíme nájsť adresu rámca, ktorá sa nachádza v tabuľke stránok (jej adresu získame z PTBR) s posuvom od začiatku ***i***. Táto úloha vyžaduje prístup k pamäti. Získaná adresa rámca spolu s posuvom v stránke dáva skutočnú adresu. To znamená, že počet prístupov k pamäti je dvojnásobný, a tým je prístup pomalší.

Štandardné riešenie tohoto problému je **použitie špeciálnej rýchlej asociatívnej pamäte** s veľmi malou prístupovou dobou, nazývanú buď *asociatívna cache pamäť,* alebo *translačné bufre* (Translation Look-aside Buffers - TLB). Asociatívna cache pamäť (TLB) pozostáva z rýchlych registrov, z ktorých každý má dve časti: kľúč a hodnotu. Keď sa prehľadáva tabuľka pre výskyt zadaného kľúča, ten sa porovnáva naraz so všetkými kľúčmi tabuľky. Ak kľúč je v tabuľke, jeho hodnota je výstupom z operácie hľadania. Takéto prehľadávanie je veľmi rýchle, ale HW je drahý. Počet položiek v TLB je od 8 do 2048.

Použitie asociatívnej cache pamäte je nasledovné. V asociatívnych registroch nie sú uložené všetky položky tabuľky stránok. Keď sa vygeneruje logická adresa, použije sa z nej číslo stránky pre prehľadanie asociatívnych registrov pre číslo rámca. Ak hľadanie je úspešné, číslo rámca je skoro ihneď známe a použije sa pre prístup do pamäte. Celá operácia bude o menej ako 10 percent dlhšia ako pri nemapovanom prístupe.

Ak číslo stránky nie je v asociatívnych registroch, musí sa urobiť odkaz na tabuľku stránok. Po získaní čísla rámca sa potom urobí odkaz na požadovanú adresu. Navyše sa do asociatívnych registrov zapíše nová položka - číslo stránky a zodpovedajúce číslo rámca. Ak TLB sú už plné, operačný systém sa musí rozhodnúť, ktorú položku vymení. V tomto prípade sa využívajú nahradzovacie algoritmy, podobné algoritmom nahradzovania stránok pri stránkovaní na žiadosť. Pri prepínaní kontextov procesov sa TLB musia vyčistiť (vymazať). Model stránkovania s TLB je na Obr. 9.15.



Obr. 9.15 Stránkovací HW s TLB

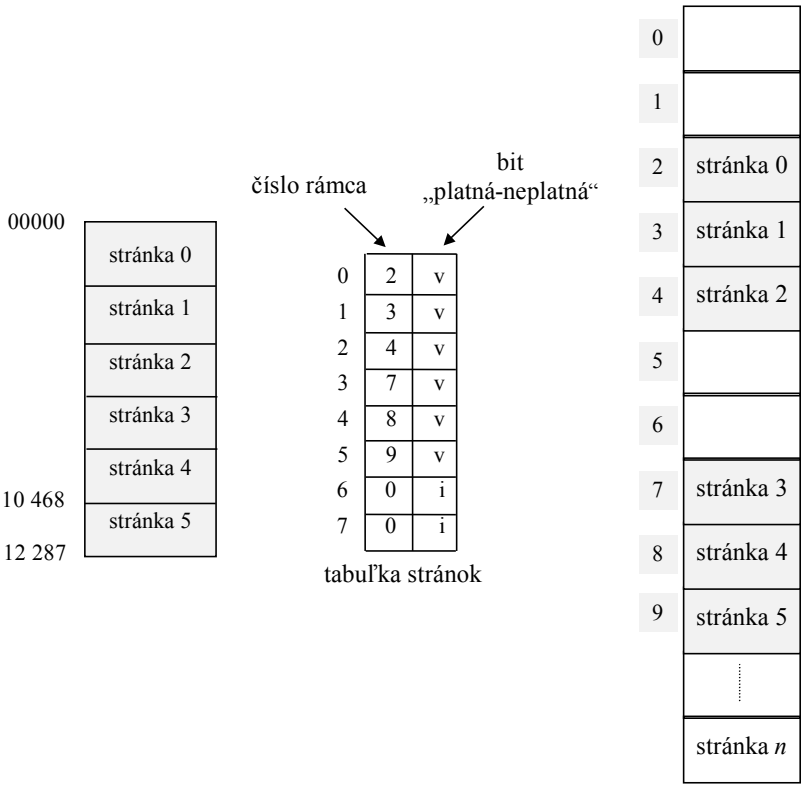
1.2.2 Ochrana

Ochrana pamäte pri stránkovaní sa uskutočňuje pomocou bitov, ktoré sú pripojené ku každému rámcu. Obyčajne sa tieto bity uchovávajú v tabuľke stránok. Jeden bit definuje prístup pre čítanie a zápis, alebo len pre čítanie. Pri každom odkaze na danú adresu sa ide cez tabuľku stránok pre získanie fyzickej adresy, a tým sa skontrolujú bity prístupu. Pokus zapisovať na stránku, ktorá je určená len na čítanie, spôsobí prerušenie pre porušenie ochrany pamäte.

Tento spôsob ochrany sa dá veľmi ľahko prispôsobiť pre dokonalejšiu ochranu, a to pridaním bitov pre každý typ prístupu zvlášť. Zvyčajne sa k týmto bitom pridáva ešte jeden bit pre každú položku tabuľky stránok - bit *platná/neplatná* (valid/invalid).

Keď tento bit je nastavený na „platná“, to znamená, že stránka je v logickom adresnom priestore procesu a je legálnou („platnou“) stránkou. **Ak bit je nastavený na „neplatná“,** to znamená, že stránka nie je v logickom adresnom priestore procesu. Neplatné adresy spôsobujú prerušenie pre ochranu pamäte. Bity platná/neplatná nastavuje operačný systém, aby dovolil alebo zakázal prístup k danej stránke. Napríklad: majme v systéme s 14 bitovou adresou (adresný priestor od 0 po 16383) program, ktorý využíva adresy od 0 po 10468. Pri veľkosti stránky 2 KB budeme mať situáciu, ukázanú na Obr. 9.16.

Adresy v stránkach 0,1,2,3,4 a 5 sú mapované normálne pomocou tabuľky stránok. Každý pokus vygenerovať adresu v stránkach 6 a 7 nájde nastavený bit *platná/neplatná* na *neplatná* a spôsobí prerušenie. Výnimkou je stránka 5, ktorá nie je zaplnená do konca a adresy od 10468 po 12287 sú tiež kvalifikované ako platné, aj keď proces ich nevyužíva. To je výsledkom vnútornej fragmentácie pri veľkosti stránky 2 KB.



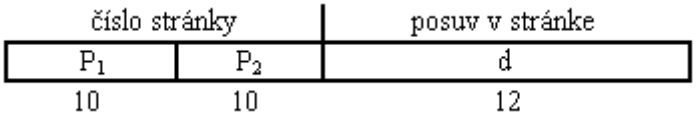
Obr. 9.16 Bit platná (v) alebo neplatná (i) v tabuľke stránok

1.3 Viacúrovňové stránkovanie

Veľa súčasných počítačov podporuje veľmi veľký logický adresný priestor (od 2^{32} po 2^{64}). V takomto prostredí tabuľka stránok je extrémne veľká. Napr. pre systém s 32 bitovým adresným priestorom a stránkou 4 KB (2^{12} bajtov) tabuľka stránok by mala 1 000 000 položiek ($2^{32}/2^{12}$). Pretože každá položka pozostáva zo 4 bajtov, každý proces by mohol vyžadovať len pre tabuľku stránok 4 MB pamäte. Je jasné, že nie je rozumné ukladať takúto tabuľku do súvislého úseku. Jedno riešenie je rozdeliť tabuľku na menšie časti. Pre implementáciu tejto myšlienky existuje niekoľko rôznych spôsobov.

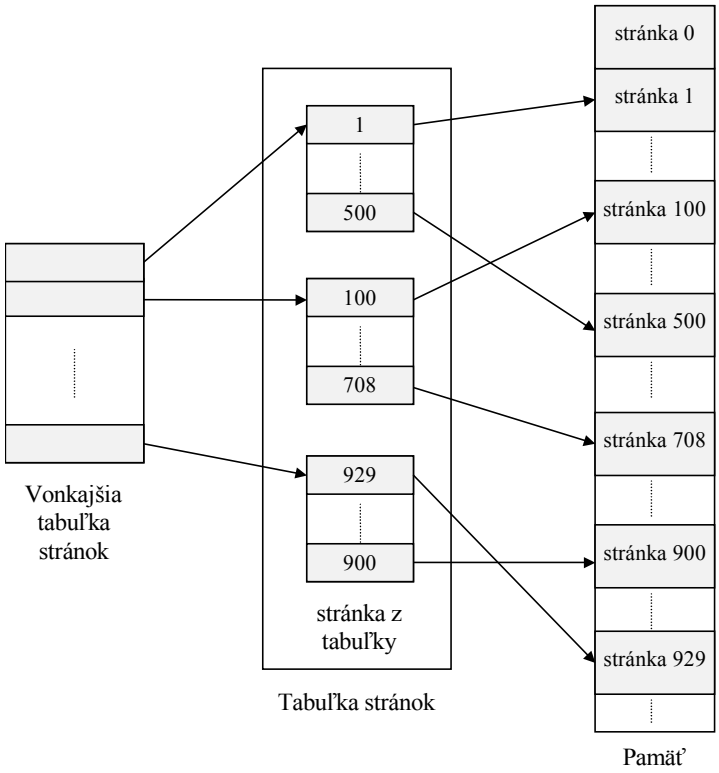
Jeden spôsob je použiť dvojúrovňové stránkovanie, čo znamená, že aj tabuľka stránok je stránkovaná (Obr. 9.17).

Vráťme sa k príkladu 32 bitového počítača so 4 KB stránkou. Logická adresa je rozdelená na číslo stránky (20 bitov) a posuv v stránke (12 bitov). Pretože stránkujeme tabuľku stránok, číslo stránky je rozdelené na 10 bitovú adresu stránky a 10 bitový posuv v stránke. Potom logická adresa vyzerá nasledovne:

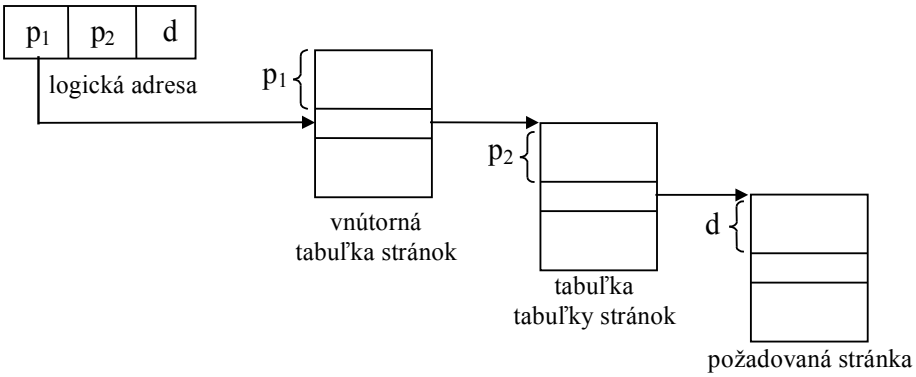


kde P_1 je index do vonkajšej tabuľky stránok a P_2 je posuv v stránke vonkajšej tabuľky stránok. Spôsob prekladu takejto adresy je ukázaný na Obr. 9.18.

Pre systém so 64 bitovým adresným priestorom vyššie uvedený spôsob už nevyhovuje, lebo tabuľka stránok pozostáva z 2^{52} položiek. Tam sa vonkajšia tabuľka stránok delí znova na menšie časti. Sú známe dvoj a trojúrovňové stránkovacie schémy. Pri použití týchto schém je samozrejme potrebné použiť rýchle cache pamäte, ktoré zabraňujú poklesu výkonu kvôli spomalenému prístupu k pamäti.



Obr. 9.17 Dvojúrovňová schéma tabuľky stránok



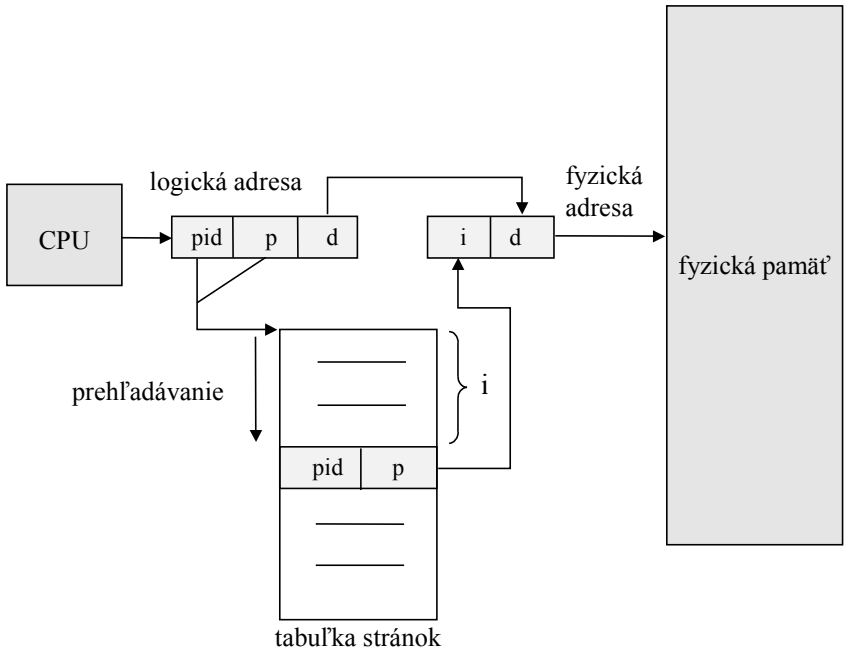
Obr. 9.18 Prevod adries pre dvojúrovňovú 32 bitovú stránkovaciu architektúru

1.4 Invertovaná tabuľka stránok

Zvyčajne každý proces má svoju tabuľku stránok, kde je položka pre každú stránku, ktorú proces používa. Táto reprezentácia je prirodzená, pretože procesy sa odkazujú na stránky cez virtuálne adresy. Operačný systém ich prekladá do fyzických adries. Pretože tabuľka je vytriedená podľa virtuálnych adries, operačný systém je schopný určiť, kde sa v tabuľke nachádza priradená fyzická adresa. Jeden z nedostatkov tejto schémy je, že tabuľka stránok môže mať milióny položiek. Takéto tabuľky zaberajú veľa fyzickej pamäte.

Tento problém sa rieši pomocou *invertovanej tabuľky stránok*. Takáto tabuľka má jednu položku pre každý rámec vo fyzickej pamäti. Každá položka pozostáva z virtuálnej adresy stránky, ktorá je v ňom uložená a informácie o procese, ktorý túto stránku vlastní. Takto v systéme existuje len jedna tabuľka stránok a tá má len jednu položku pre každý rámec fyzickej pamäte. Na Obr. 9.19 sú ukázané operácie nad invertovanou tabuľkou stránok.

Tento spôsob stránkovania je využitý v systémoch IBM RISC 6000, IBM RT a Hewlett-Packard Spectrum Workstation.



Obr. 9.19 Invertovaná tabuľka stránok

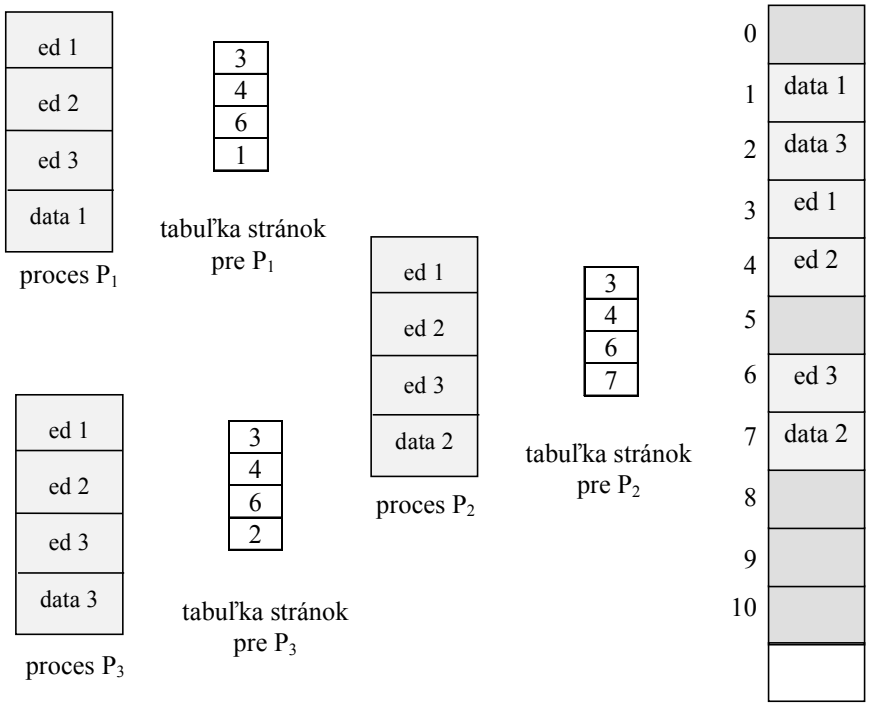
1.5 Zdieľanie stránok

Ďalšou prednosťou stránkovania je možnosť *zdieľania* spoločného kódu. Táto vlastnosť je veľmi dôležitá pre prostredie so zdieľaným časom. Ak so systémom pracuje 40 používateľov a každý používa textový editor (študenti editujú vo *vi* editore !!!) a editor pozostáva zo 150 KB kódu a 50 KB dát, budeme potrebovať 8000 KB pamäte pre uspokojenie používateľov. Ak kód je *reentrantný*, môže byť zdieľaný, ako je ukázané na Obr. 9.20 Obr.. Tam 3 procesy zdieľajú kód editora a dátové stránky sú vlastné.

Reentrantný kód je kód, ktorý nemodifikuje sám seba, a tým sa počas vykonania nemení. Takže jeden alebo viac procesov môže vykonávať ten istý kód súčasne. Každý proces má svoju vlastnú kópiu registrov a dát.

Pri tomto spôsobe využitia sa vo fyzickej pamäti nachádza len jedna kópia editora. Tabuľka stránok každého používateľa ukazuje na tú istú stránku kódu, ale dátové stránky sú odlišné. Takže pre 40 používateľov budeme potrebovať 2150 KB (150 + 40x50), čo predstavuje značnú úsporu pamäte.

V systéme sú obyčajne zdieľané často používané programy, ako sú prekladače, databázové systémy a iné. Aby boli zdieľané, musia mať reentrantný kód.

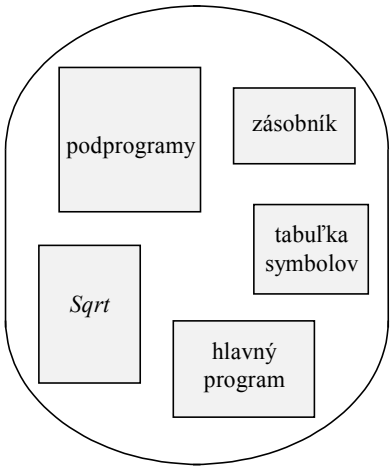


Obr. 9.20 Zdieľanie kódu v stránkovacom prostredí

9.6 Segmentácia

1.1 Princíp

Používateľský pohľad na pamäť sa líši od skutočnej fyzickej pamäte. Používateľ sa nepozerá na pamäť ako na lineárne pole bajtov. Pre používateľa je pamäť množstvo segmentov s variabilnou dĺžkou, ktoré nie sú zoradené (Obr. 9.21).



Obr. 9.21 Používateľský pohľad na program

Programátor sa pri tvorbe programu naň pozerá ako na hlavný program a množinu procedúr, funkcií alebo modulov. V programe sa využívajú aj rôzne dátové štruktúry: tabuľky, polia, zásobníky, premenné atď. Na každý z týchto prvkov odkazujeme menom a nezaujímame sa, kde v pamäti je príslušný prvok uložený.

Segmentácia podporuje tento používateľský pohľad na pamäť. Logický adresný priestor je sada segmentov. Každý segment má dĺžku a veľkosť. Adresa špecifikuje meno segmentu a posuv vo vnútri segmentu a práve tak sa na adresy odkazuje aj používateľ.

Pre jednoduchosť sa segmenty číslujú. Logická adresa pozostáva z čísla segmentu a posuvu v segmente:

<číslo segmentu, posuv>

Keď sa používateľský program prekladá, prekladač automaticky vytvára segmenty na základe vstupného programu. Prekladač Pascalu môže vytvoriť oddelené segmenty pre:

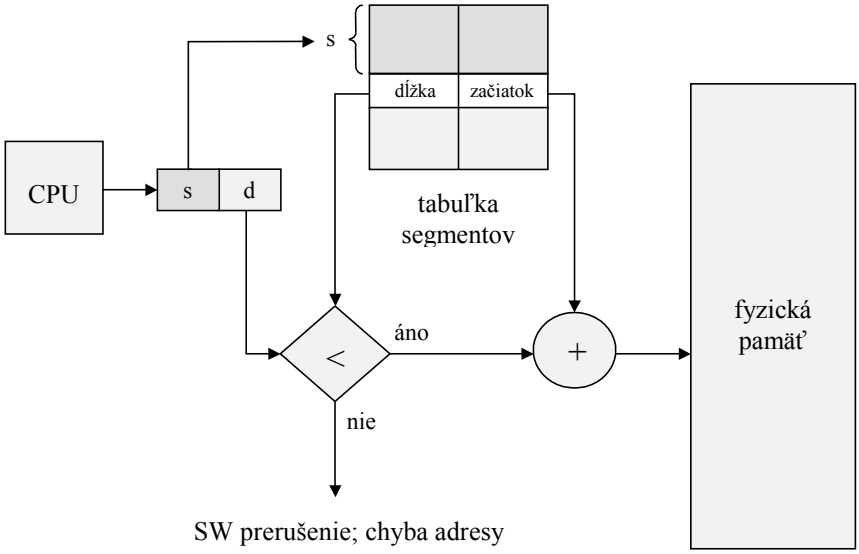
1) globálne premenné,
2) zásobník pre volania podprogramov - pre uloženie parametrov a návratových adries,
1) kódy procedúr a funkcií a
3) lokálne premenné každej procedúry a funkcie.

Zavádzací program priradí všetkým segmentom čísla.

1.2 Hardvér

Používateľ sa môže odkazovať na objekty dvojrozmernou adresou (číslo segmentu, posuv), ale skutočná fyzická pamäť je stále jednorozmerná postupnosť bajtov. Takže musíme mapovať dvojrozmernú používateľskú adresu na jednorozmernú. Toto mapovanie uskutočňujeme pomocou tabuľky segmentov. Každá položka tabuľky segmentov obsahuje bázu a dĺžku segmentu. Báza je počiatočná fyzická adresa segmentu a dĺžka odzrkadľuje dĺžku segmentu.

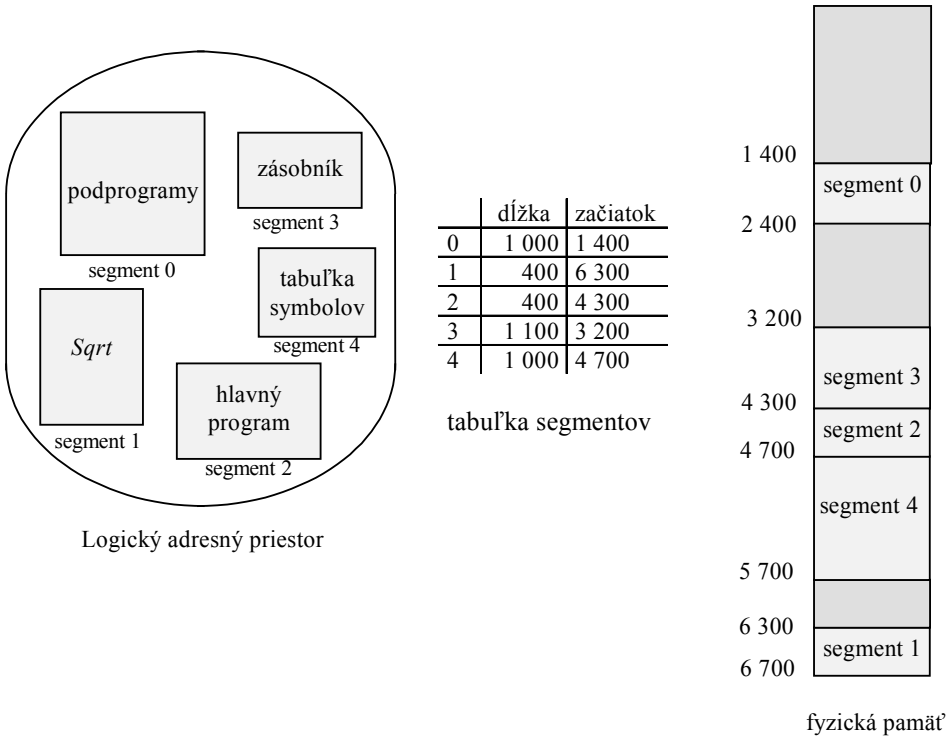
Použitie tabuľky segmentov je znázornené na Obr. 9.22 Obr.. Logická adresa pozostáva z dvoch častí: číslo segmentu - *s,* a posuv v segmente - *d.* Číslo segmentu sa použije ako index v tabuľke. Posuv *d* logickej adresy musí byť medzi 0 a veľkosťou segmentu. Ak tomu tak nie je, vygeneruje sa prerušenie pre pokus o prístup k adrese mimo segmentu. Ak posuv je legálny, pripočíta sa k počiatočnej adrese a získa sa fyzická adresa.



Obr. 9.22 Segmentačný HW

Na Obr. 9.23 je ukázaná situácia, kedy máme 5 segmentov. V tabuľke segmentov je položka pre každý segment, kde je zaznamenávaná dĺžka a začiatočná adresa segmentu. Fyzickú adresu pre bajt 53

zo segmentu 2 spočítame tak, že k počiatočnej adrese segmentu 4300 pripočítame posuv v rámci segmentu: 4300 + 53 = 4353.



Obr. 9.23 Príklad segmentácie

1.3 Implementácia tabuľky segmentov

Segmentácia má úzky vzťah k modelom správy pamäte, ktoré boli prezentované v predchádzajúcom texte, s tým rozdielom, že jeden program môže pozostávať z viacerých segmentov. Segmentácia je zložitejšia ako stránkovanie. Podobne ako tabuľka stránok, tak aj tabuľka segmentov sa môže umiestniť do rýchlych registrov alebo do pamäte. Tabuľka segmentov uchovaná v registroch sa dá spracovať veľmi rýchlo - pripočítanie začiatku a porovnanie s limitnou adresou sa dá vykonať súčasne, a tým ušetriť čas.

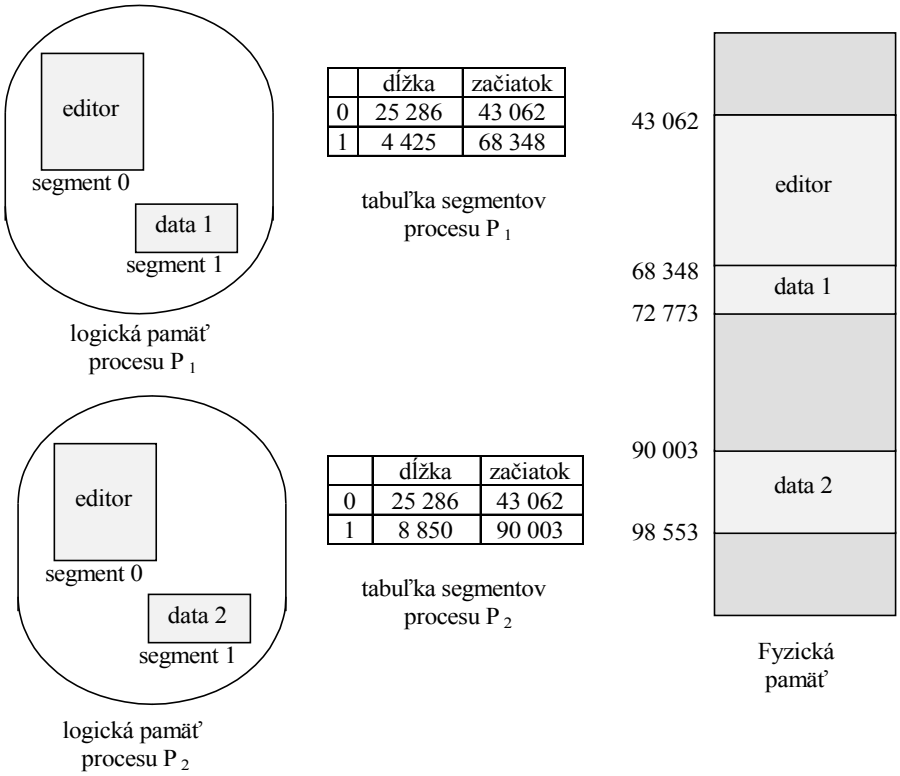
V prípade, že program pozostáva z väčšieho počtu segmentov, nie je výhodné uchovávať tabuľku segmentov v registroch, takže ju musíme uložiť do pamäte. *Register začiatku tabuľky segmentov* (STBR - Segment-Table Base Register) ukazuje na tabuľku segmentov. Počet segmentov v programe môže byť značne odlišný, preto sa používa *register dĺžky tabuľky segmentov* (STLR - Segment-Table Length Register). Pre logickú adresu *(s, d)* sa najprv kontroluje, či číslo segmentu je platné (to znamená, či s<STLR). Potom sa pripočíta číslo segmentu k STBR a získa sa výsledná adresa (STBR + *s)* položky tabuľky segmentov. Táto položka sa načíta z pamäte, potom sa skontroluje dĺžka segmentu a vypočíta sa požadovaná fyzická adresa ako súčet adresy začiatku segmentu a posuvu.

Ako pri stránkovaní, aj toto mapovanie vyžaduje dva odkazy na pamäť pre danú logickú adresu, takže sa počítačový systém spomalí dvakrát, pokiaľ sa táto situácia nerieši. Obyčajne riešenie je použitie sady asociatívnych registrov, ktoré uchovávajú posledne používané položky z tabuľky segmentov. Aj v tomto prípade malá sada asociatívnych registrov môže zredukovať čas potrebný pre prístup k pamäti tak, že ten nebude o viac ako 10 až 15 % väčší ako nemapovaný prístup.

1.4 Ochrana a zdieľanie

Veľká výhoda segmentácie je možnosť pripojenia ochrany k segmentu. Pretože segmenty reprezentujú sémanticky definovanú časť programu, je predpoklad, že všetky položky segmentu sa budú používať rovnakým spôsobom. Napríklad môžeme mať segmenty, ktoré obsahujú len dáta a iné len inštrukcie. V moderných architektúrach inštrukcie sami seba nemodifikujú, takže segment inštrukcií sa môže zadefinovať ako vykonateľný alebo len na čítanie. Mapovací HW kontroluje bity ochrany, ktoré sú pripojené ku každej položke tabuľky segmentov, a takto zabraňuje zápisu do segmentu, ktorý je určený len na čítanie.

Inou výhodou segmentácie je možnosť zdieľania kódu alebo dát. Každý proces má tabuľku segmentov, priradenú k riadiacemu bloku procesu, ktorú dispečer používa na definovanie hardvérovej tabuľky segmentov, keď proces dostane pridelený procesor. Segmenty sú zdieľané vtedy, keď položky v tabuľke segmentov dvoch rôznych procesov ukazujú na tie isté fyzické adresy (Obr. 9.24).



Obr. 9.24 Zdieľanie segmentov v segmentovanom pamäťovom systéme

Zdieľanie sa vyskytuje na úrovni segmentov. To znamená, že každá informácia, ktorá je zadefinovaná ako segment, sa môže zdieľať. Môže sa zdieľať aj viacero segmentov.

Napríklad, predpokladajme použitie textového editora v time-sharing-ovom systéme. Celý editor môže byť veľký a môže pozostávať z viacerých segmentov. Tieto segmenty môžu zdieľať všetci užívatelia a takto namiesto *n* kópií editora potrebujeme len jednu. Pre každého používateľa bude samozrejme potrebný ďalší segment pre uloženie lokálnych premenných, ktorý sa nezdieľa.

Zdieľanie sa môže uskutočniť aj na úrovni podprogramov. Napr. ak balíky bežne používaných podprogramov sa zadefinujú ako segmenty len na čítanie, potom v pamäti bude len jedna fyzická kópia z každého podprogramu, ktorú budú používať všetci užívatelia.

Aj keď zdieľanie segmentov vyzerá jednoduché, sú v ňom schované problémy, ktoré je potrebné vyriešiť. Veľmi často kódové segmenty obsahujú odkazy sami na seba. Napr. podmienený skok obsahuje adresu, kde sa odovzdáva riadenie. Tá adresa je číslo segmentu a posuv v segmente. Ak segment je zdieľaný, všetky procesy musia definovať to isté číslo zdieľaného segmentu. Dátové

segmenty prístupné len pre čítanie a kódové segmenty, ktoré neodkazujú na seba, môžu byť zdieľané aj s rôznymi číslami segmentov.

Tento problém sa rieši zavedením registra čísla segmentu, ktorý obsahuje číslo segmentu, určené čítačom inštrukcií. Adresovanie v segmentoch sa potom realizuje cez tento register. Úplné adresy dát sa pri volaní takýchto podprogramov odovzdávajú pomocou parametrov umiestnených napr. v registroch alebo zásobníkoch.

1.5 Fragmentácia

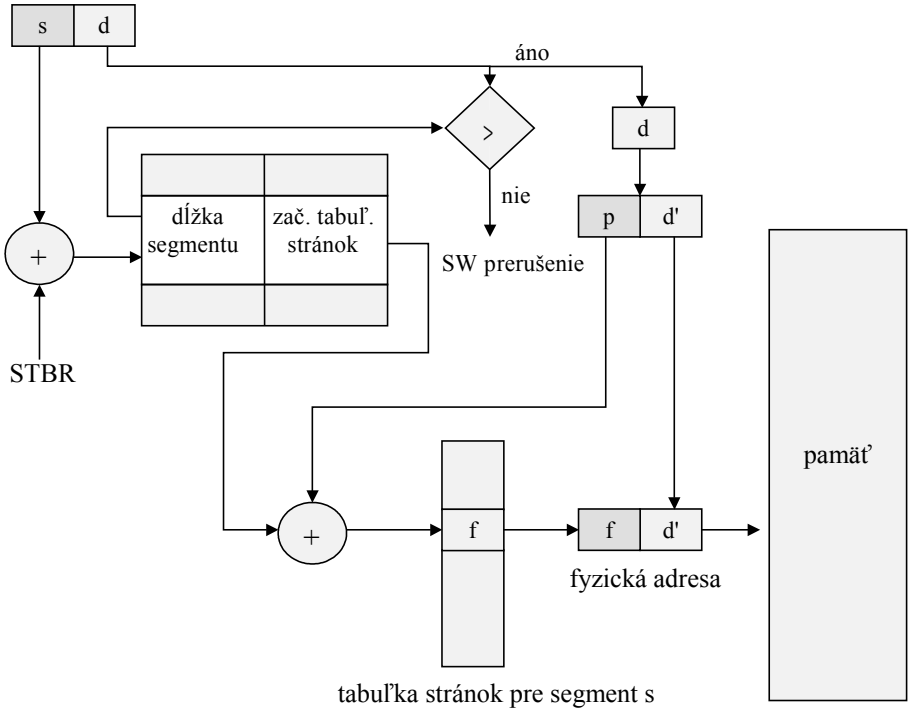
Dlhodobý plánovač musí nájsť a prideliť pamäť pre všetky segmenty programu. Problém prideľovania pamäte pri segmentácii je podobný tomu pri prideľovaní súvislých úsekov s premenlivou dĺžkou a obyčajne sa rieši algoritmami first-fit alebo best-fit.

Segmentácia spôsobuje vonkajšiu fragmentáciu, kedy bloky voľnej pamäte sú príliš malé, aby sa dali využiť. V tomto prípade proces, ktorý sa nedá umiestniť, musí čakať, kým sa uvoľní viac pamäte alebo sa použije striasanie.

9.7 Segmentácia so stránkovaním

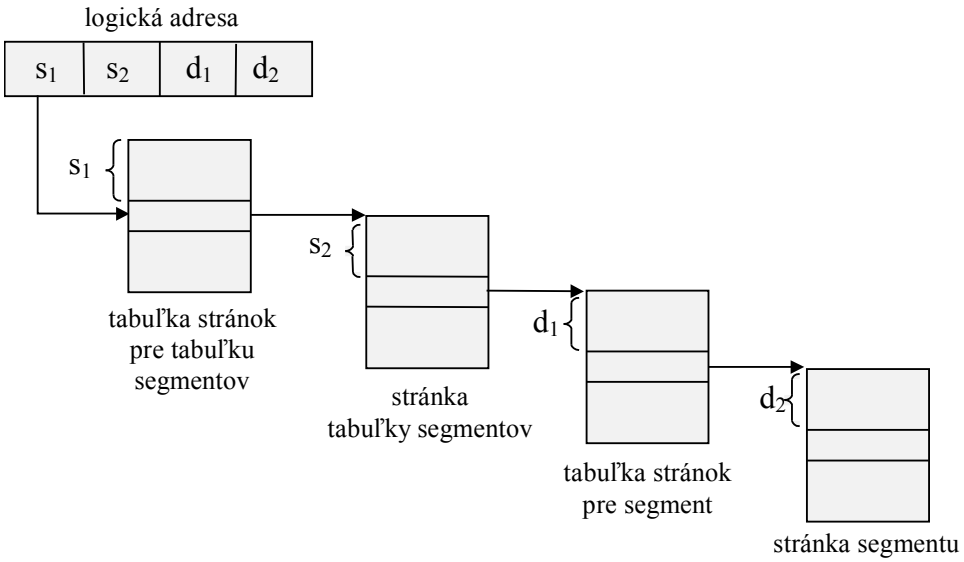
Ako stránkovanie, tak aj segmentácia majú svoje prednosti a nedostatky. Zaujímavý je fakt, že dve najrozšírenejšie rady procesorov Motorola 68000 a Intel 80X86 využívajú rôzne metódy správy pamäte: prvý využíva jednorozmerný adresný priestor t.j. stránkovanie, druhý zakladá správu pamäte na segmentácii. Obidve metódy vlastne kombinujú stránkovanie a segmentáciu.

Na Obr. 9.25 je ukázaný prístup k pamäti systému MULTICS. Schéma, používaná pre odstránenie nedostatkov segmentácie je stránkovanie segmentov. Stránkovanie odstraňuje vonkajšiu fragmentáciu a uľahčuje problém prideľovania pamäte. Adresa je tvorená 18 bitovým číslom segmentu a 16 bitovým posuvom. Posuv je tvorený 6 bitovým číslom stránky a 10 bitovým posuvom v stránke. Číslo stránky je indexom v tabuľke, odkiaľ sa získava číslo rámca. Nakoniec číslo rámca spolu s posuvom tvoria fyzickú adresu. Rozdiel medzi touto tvorbou adresy a čistým segmentovaním je, že tabuľka segmentov neobsahuje počiatočnú adresu segmentu, ale počiatočnú adresu tabuľky stránok príslušného segmentu. Transformácia logickej adresy v MULTICS-e je ukázaná na Obr. 9.26.



Obr. 9.25 Stránková segmentácia na GE 645 (MULTICS)

Každý segment má svoju tabuľku stránok, ale pretože veľkosť segmentu je obmedzená dĺžkou segmentu z tabuľky segmentov, tabuľka stránok nemusí byť úplná.



Obr. 9.26 Preklad adresy v systéme MULTICS

9.8 Záver

Algoritmy správy pamäte pre multiprogramové prostredie predstavujú širokú škálu od jednoduchých metód pre jednoužívateľské systémy po stránkovanú segmentáciu. Určujúci moment pri výbere metódy je HW platforma. Každá adresa vygenerovaná procesorom sa musí skontrolovať či je platná, a potom pretransformovať na fyzickú adresu. Kontrola sa nedá robiť efektívne, ak sa robí programovo. V tomto prípade je správa pamäte viazaná na dostupný HW.

Uvedené metódy správy pamäte (súvislé prideľovanie, stránkovanie, segmentácia a kombinácia stránkovania a segmentácie) sa líšia v mnohých aspektoch. Ďalej uvádzame niektoré dôležité fakty pre porovnanie jednotlivých metód:

- **HW podpora**: jednoduchý bázový register alebo dvojica bázového a limitného registra postačuje pre prideľovanie jedného alebo viacerých úsekov, pokiaľ stránkovanie a segmentácia vyžadujú mapovacie tabuľky pre definovanie mapy adries.
  - **Výkon**: Čím je algoritmus zložitejší, tým je čas potrebný na prepočítanie logickej adresy na fyzickú dlhší. Pre jednoduché systémy je potrebné len porovnávať alebo pripočítavať určité hodnoty k logickej adrese, t.j. operácie sú rýchle. Stránkovanie a segmentácia môžu byť rovnako rýchle, ak tabuľka stránok/segmentov je implementovaná pomocou rýchlych registrov. Ak je tabuľka v pamäti, prístup k používateľskej pamäti môže byť podstatne pomalší. Sada asociatívnych registrov môže zredukovať pokles výkonu na prijateľnú úroveň.
  - **Fragmentácia:** Multiprogramové systémy bežne zlepšujú svoj výkon zvýšením úrovne multiprogramovania. Pre danú sadu procesov môžeme zvýšiť úroveň multiprogramovania len zvýšením počtu procesov umiestnených v pamäti. Pre uskutočnenie tejto úlohy musíme znížiť plytvanie pamäte fragmentáciou. Systémy s pevne stanovenými alokačnými úsekmi, ako napr. systém s prideľovaním jedného úseku alebo so stránkovaním, trpia vnútornou fragmentáciou. Systémy s prideľovaním úsekov s premenlivou dĺžkou, ako sú prideľovanie viacerých úsekov a segmentácia, trpia vonkajšou fragmentáciou.
  - **Relokácia**: Jedno riešenie problému vonkajšej fragmentácie je striasanie. Striasanie zahŕňa presun programu v pamäti tak, že program si nevšimne zmenu. Tento predpoklad vyžaduje dynamickú relokáciu logických adries v čase vykonania. Ak sa adresy relokujú len počas zavedenia, pamäť sa nedá striasať.
  - **Swapovanie:** Každý algoritmus môže zahrňovať aj swapovanie. V intervaloch, ktoré určuje operačný systém a sú obyčajne diktované použitým algoritmom plánovania, proces je

kopírovaný z pamäte do záložnej pamäte a neskôr naspäť do operačnej pamäte. Táto schéma dovoľuje bežať naraz viacerým procesom, ako sa súčasne zmestí do pamäte v danom okamihu.

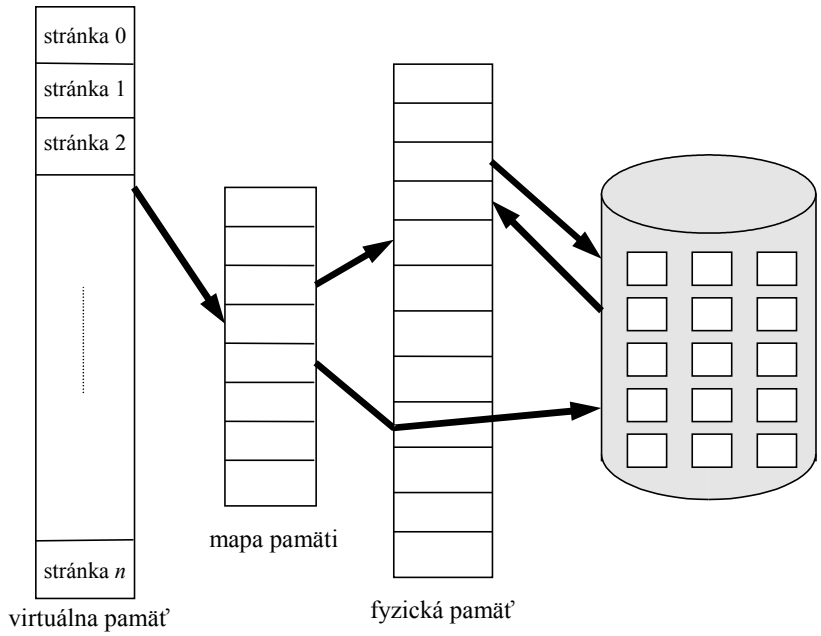
- **Zdieľanie**: Iný prostriedok ako zvýšiť úroveň multiprogramovania je zdieľanie kódu medzi viacerými používateľmi. Zdieľanie sa dá uskutočniť len pri stránkovaní alebo segmentácii, kedy sa zdieľajú malé úseky. Zdieľané programy musia byť navrhnuté starostlivo.
- **Ochrana**: Ak sa používa stránkovanie alebo segmentácia, potom rôzne časti používateľských programov môžu byť deklarované len na čítanie, len na vykonanie alebo na čítanie a zápis. Tieto obmedzenia sú potrebné pri zdieľaní kódu alebo dát a sú obecne užitočné pre uskutočnenie jednoduchej kontroly proti bežným programovým chybám.

10 VIRTUÁLNA PAMÄŤ

Virtuálna pamäť je technika, dovoľujúca vykonanie procesov, ktoré sa nenachádzajú v operačnej pamäti celé. Hlavná výhoda tejto techniky je, že dovoľuje poskytnúť používateľovi veľmi veľkú virtuálnu pamäť, pričom sa využíva malá fyzická pamäť (Obr. 10.1). Virtuálna pamäť zjednodušuje prácu programátora, pretože nie je potrebné vytvárať segmenty pre prekrytie a plánovať poradie ich zavedenia do pamäte. Tieto úlohy preberá správa pamäte.

10.1 Pozadie

Algoritmy správy pamäte, ktoré sme popísali v predchádzajúcej kapitole, majú jeden spoločný rys v tom, že inštrukcie procesu, ktorý sa vykonáva, musia byť vo fyzickej pamäti. Pokiaľ program je väčší ako fyzická pamäť, je možné ho vykonať, len ak využijeme prekrývanie. Táto technika vyžaduje zvýšené úsilie a skúsenosť programátora.

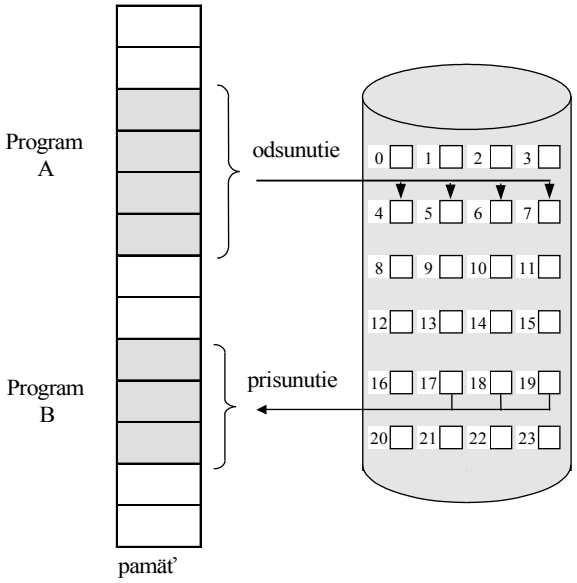


Obr. 10.1 Diagram ukazujúci stav, kedy virtuálna pamäť je väčšia ako fyzická

Virtuálna pamäť sa bežne implementuje pomocou *stránkovania na žiadosť.* Niektoré systémy poskytujú segmentovanie so stránkovaním - segmenty sú rozdelené na stránky. Virtuálna pamäť môže byť implementovaná aj segmentáciou na žiadosť. Táto metóda virtualizácie je použitá v systéme IBM OS/2.

10.2 Stránkovanie na žiadosť

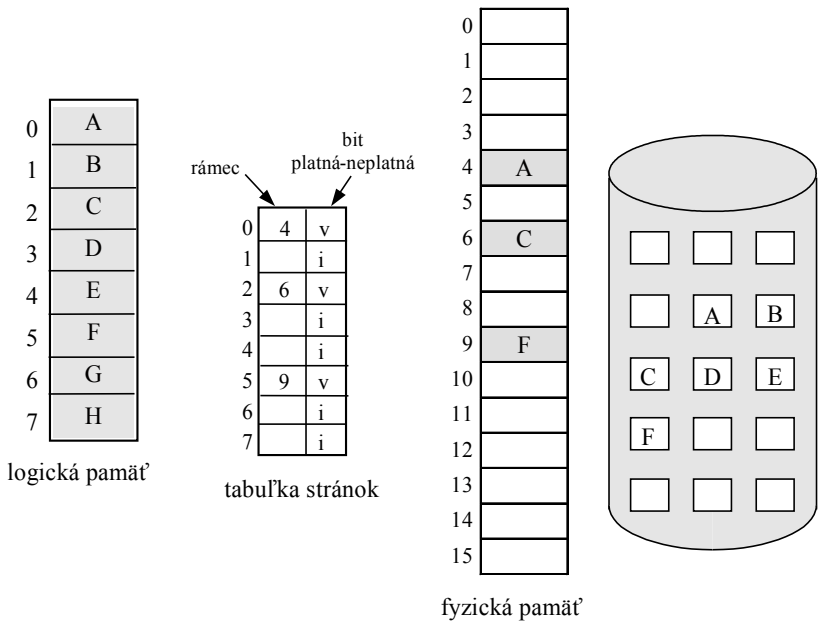
Stránkovanie na žiadosť je podobné systému stránkovania so swapovaním (Obr. 10.2). Procesy sú umiestnené na disku. Keď chceme vykonávať proces, presunieme ho do pamäte. Avšak miesto presunutia celého procesu do pamäte presunieme len jeho časť. Používame tzv. „lenivý“ swapper, ktorý nikdy nepresúva stránku do pamäte, kým nie je potrebná. V tomto prípade, ale použitý termín swapper nie je celkom správny, pretože sa nejedná o presúvanie celého procesu, ako sme doteraz swapovanie chápali. V tomto prípade je presnejší termín stránkovač (pager).



Obr. 10.2 Presun stránok procesu do súvislého diskového priestoru

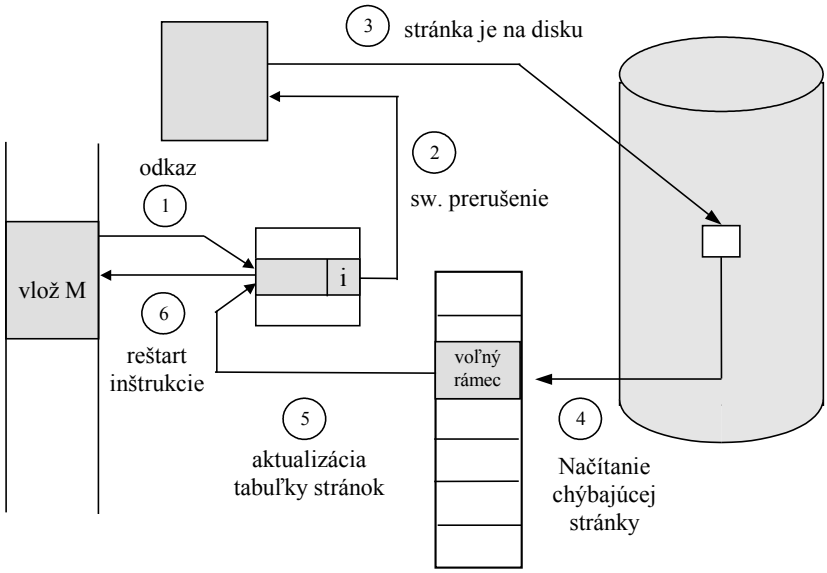
Keď sa proces má zaviesť do pamäte, stránkovač predpokladá ktoré stránky budú potrebné, kým proces nebude znova odsunutý von z pamäte. Miesto presúvania celého procesu, stránkovač presunie do pamäte len vytypované stránky. Takto sa vyhne presúvaniu stránok, ktoré nebudú potrebné a tým sa zníži čas potrebný na výmenu procesov v pamäti.

Pri použití tejto schémy je potrebná HW podpora pre rozlíšenie, ktoré stránky sú v pamäti a ktoré nie sú. Používa sa už zmienený spôsob nastavenia bitu *platná/neplatná (valid/invalid*), ktorý sme popísali v predchádzajúcej kapitole. V tomto prípade keď bit je nastavený na hodnotu *platná*, to znamená, že stránka je platná a je v pamäti. Ak bit je nastavený na *neplatná*, to znamená, že stránka je buď neplatná (nepatrí do adresného priestoru procesu), alebo nie je v pamäti. Položka v tabuľke stránok pre stránku, ktorá je v pamäti, je nastavená ako obyčajne, ale položka stránky, ktorá nie je v pamäti je označená ako *neplatná*, alebo obsahuje adresu stránky na disku. Táto situácia je znázornená na Obr. 10.3.



Obr. 10.3 Tabuľka stránok, keď niektoré stránky nie sú v pamäti

Ak sa proces nikdy neobráti na stránku, ktorá je označená ako neplatná, potom toto označenie nemá žiadny vplyv na proces. To znamená, že ak správne odhadneme, ktoré stránky proces skutočne potrebuje a len tie presunieme do pamäte, proces prebehne tak, ako keby bol celý v pamäti.



Obr. 10.4 Kroky pri obsluhe výpadku stránky

Situácia sa zmení vo chvíli, keď sa proces obráti na stránku, ktorá nie je v pamäti. Obrátenie sa na stránku označenú ako neplatná spôsobí *výpadok stránky(page fault)*. Stránkovací hardvér si počas prekladu adresy všimne, že bit *platná/neplatná* je nastavený na neplatná a spôsobí prerušenie pre neprítomnosť stránky v pamäti. Toto prerušenie má za následok presunutie požadovanej stránky do pamäte. Obsluha prerušenia výpadku stránky prebieha v nasledujúcich krokoch (Obr. 10.4):

2. Skontroluje sa vnútorná tabuľka (obyčajne uchovaná v riadiacom bloku procesu) pre tento proces, aby sa zistilo, či odkaz na danú stránku bol platný.
3. Ak odkaz bol neplatný, proces sa ukončí. Ak odkaz bol platný, ale stránka nie je v pamäti, zaháji sa jej presun.
4. Nájde sa voľný rámec.
5. Odštartuje sa disková operácia pre načítanie požadovanej stránky do určeného rámca.
6. Po skončení operácie načítania sa vnútorná tabuľka modifikuje, aby odzrkadľovala prítomnosť stránky v pamäti.
7. Reštartuje sa inštrukcia, ktorá spôsobila výpadok stránky.

Proces, ktorý spôsobil výpadok stránky, sa reštartuje presne od miesta, kde prerušenie nastalo. Toto je možné, pretože stav procesu sa uchováva - registre, podmienkové kódy, čítač inštrukcií.

V extrémnom prípade je možné odštartovať proces, ktorý nemá v pamäti žiadnu stránku. Po pokuse načítať prvú inštrukciu programu, nastane ihneď výpadok stránky. Po presunutí tejto stránky do pamäte, proces môže pokračovať tak, že bude vyvolávať výpadky stránok dovtedy, kým nebude mať v pamäti všetky stránky, ktoré pre svoje vykonanie potrebuje. Táto metóda sa volá „čisté stránkovanie na žiadosť“ a nikdy nepresúva stránku do pamäte, kým táto nie je požadovaná.

Teoreticky niektoré programy sa môžu odkazovať na niekoľko nových stránok počas vykonania každej inštrukcie (jedna stránka pre inštrukcie, druhá pre dáta), a tak spôsobiť viacej výpadkov stránok pre každú inštrukciu. Táto situácia môže veľmi negatívne ovplyvniť výkon systému. Našťastie analýza bežiacich procesov ukázala, že takéto chovanie procesov je nezvyčajné. Programy sa vo

väčšine prípadov chovajú podľa *princípu lokality* (ktorý sme spomenuli v predchádzajúcom texte) a toto dáva možnosť úspešne aplikovať stránkovanie na žiadosť.

HW prostriedky, potrebné pre stránkovanie na žiadosť, sú rovnaké ako v prípade stránkovania so swapovaním:

- ***Tabuľka stránok****: táto tabuľka obsahuje bit pre označenie platnosti/neplatnosti stránky alebo dodatočné bity pre ochranu prístupu k stránke.*
  - **Periférna pamäť**: táto pamäť uchováva stránky, ktoré nie sú momentálne v operačnej pamäti. Obyčajne je to rýchly disk. Hovorí sa mu *swapovacie zariadenie* a časť disku, určená pre odkladanie stránok procesov, sa nazýva *swapovací priestor.*

Samozrejme okrem HW podpory je potrebná aj značná SW podpora.

Pri stránkovaní na žiadosť je potrebné zobrať do úvahy niektoré zvláštnosti architektúry systému. Už bolo povedané, že po výpadku stránky sa inštrukcia reštartuje. Vo väčšine prípadov sa obsluha výpadku ľahko realizuje. Ale výpadok stránky sa môže vyskytnúť pri každom odkaze na pamäť. Ak sa výpadok vyskytne pri pokuse zaviesť inštrukciu, stránka sa natiahne a inštrukcia sa zavedie znova. Ak sa výpadok vyskytne pri odkaze na operand, inštrukcia sa musí zaviesť znova, znova dekódovať a potom zaviesť požadovaný operand.

Ako najhorší prípad predpokladajme trojadresnú inštrukciu ako napr. sčítanie obsahu adresy *A* s obsahom adresy *B* a uloženie výsledku na adresu *C*. Kroky pre vykonanie tejto inštrukcie sú:

1. Zaviesť a dekódovať inštrukciu.
2. Zaviesť operand *A*.
3. Zaviesť operand *B*.
4. Sčítať *A* a *B*.
5. Uložiť súčet do *C*.

Ak výpadok nastane pri pokuse uložiť výsledok do *C*, musíme zaistiť presun požadovanej stránky do pamäte, upraviť tabuľku stránok a reštartovať inštrukciu. Reštart bude požadovať nové zavedenie inštrukcie, nové dekódovanie, nové zavedenie operandov *A* a *B* a nové sčítanie. Avšak opakovaná práca nepresahuje svojou dĺžkou vykonanie jednej inštrukcie a vykonáva sa len ak nastane výpadok stránky.

Hlavné ťažkosti nastavajú vtedy, ak jedna inštrukcia môže modifikovať niekoľko pamäťových miest. Napr. inštrukcia MVC (systém IBM 360/370) môže presunúť naraz až 256 bajtov z jednej adresy na druhú. Pri vykonaní tejto inštrukcie môže nastať prípad, kedy každý blok znakov je uložený na prelome dvoch stránok a výpadok sa môže vyskytnúť, keď už presun bol čiastočne urobený (t.j. časť adries bola už modifikovaná). Riešenie problému reštartu inštrukcie nie je možné bez dodatočnej HW podpory, kedy sa používa buď mikrokód pre prepočet adries zdrojového a cieľového bloku a v prípade, že má nastať výpadok stránky, ten sa vyvolá ešte pred započatím vykonania inštrukcie, kedy neboli urobené žiadne modifikácie. Druhé riešenie je použitie pomocných registrov, ktoré uchovávajú pôvodný obsah modifikovaných pamäťových adries, aby sa tento mohol vrátiť v prípade výpadku stránky.

10.2.1 Výkonnosť stránkovania na žiadosť

Stránkovanie na žiadosť môže mať významný efekt na výkonnosť počítačového systému. Pre vysvetlenie sa pozrieme na čas prístupu pre pamäť stránkovanú na žiadosť. Väčšina počítačov má *čas prístupu(t_{pr})* k pamäti v intervale od 10 do 200 ns. Pokiaľ sa výpadky stránok nebudú vyskytovať, čas prístupu k pamäti pri stránkovaní na žiadosť bude rovnaký ako v prípade bez stránkovania. Samozrejme pri výpadku stránky sa čas prístupu zmení.

Nech p je pravdepodobnosť výpadku stránky ($0 \leq p \leq 1$). Očakávame, že p je veľmi malé. *Efektívny čas prístupu k pamäti* (t_{ef}) potom bude:

$$t_{ef} = (1-p).t_{pr} + p.t_{vyp}$$

Pre výpočet efektívneho času prístupu musíme vedieť, koľko času potrebujeme pre obsluhu výpadku (t_{vyp}). Výpadok stránky zapríčiní vykonanie nasledujúcich krokov:

1. Prerušenie do operačného systému.
2. Uchovanie obsahu registrov a procesu.
3. Určenie, či sa jedná o výpadok stránky.
4. Kontrola, či odkaz na stránku bol platný a určenie umiestnenia stránky na disku.
5. Zahájenie načítania stránky do voľného rámca, čo znamená:
   a) čakanie vo fronte zariadenia, pokiaľ žiadosť na čítanie bude obslúžená,
   b) čakanie, kým sa čítacie hlavy nastavia,
   c) začiatok prenosu do rámca.
6. Počas čakania sa procesor pridelí inému procesu.
7. Prerušenie od disku - koniec prenosu stránky.
8. Uchovanie registrov a stavu bežiaceho procesu.
9. Obsluha prerušenia od disku.
10. Korekcia tabuľky stránok pre presunutú stránku.
11. Čakanie na opätovné pridelenie procesora.
12. Zavedenie obsahu registrov, procesu a tabuľky stránok a opätovné spustenie inštrukcie, ktorá vyvolala výpadok stránky.

Nie vždy sú potrebné všetky kroky. V každom prípade ale časovo najnáročnejšie sú obsluha výpadku stránky, načítanie stránky a reštart procesu. Prvá a tretia úloha sa dajú zredukovať na niekoľko stoviek inštrukcií pri starostlivom kódovaní a obyčajne zaberajú od 1 do 100 μs. Prepnutie stránky je okolo 24 ms, typické oneskorenie disku je 8 ms a čas pre nastavenie hláv je 15 ms, prenos je 1 ms. To znamená, že celkový čas stránkovania je okolo 25 ms.

Ak je priemerný čas obsluhy výpadku stránky 25 ms a čas prístupu 100 ns, potom efektívny čas prístupu v *ns* je:

$$t_{ef} = (1-p).(100) + p.(25\ 000\ 000) = 100 + 24\ 999\ 900 \,.\, p$$

Z toho je vidieť, že efektívny čas prístupu je proporcionálny pravdepodobnosti výpadkov stránok. Ak sa výpadok stránky vyskytne raz na 1000 pamäťových odkazov, efektívny čas prístupu je 25 μs. To znamená, že stránkovanie na žiadosť zapríčiní 250 krát pomalší chod počítača. Takže ak chceme spomalenie menšie ako 10 %, pre p musí platiť:

$$p < 0.0000004$$

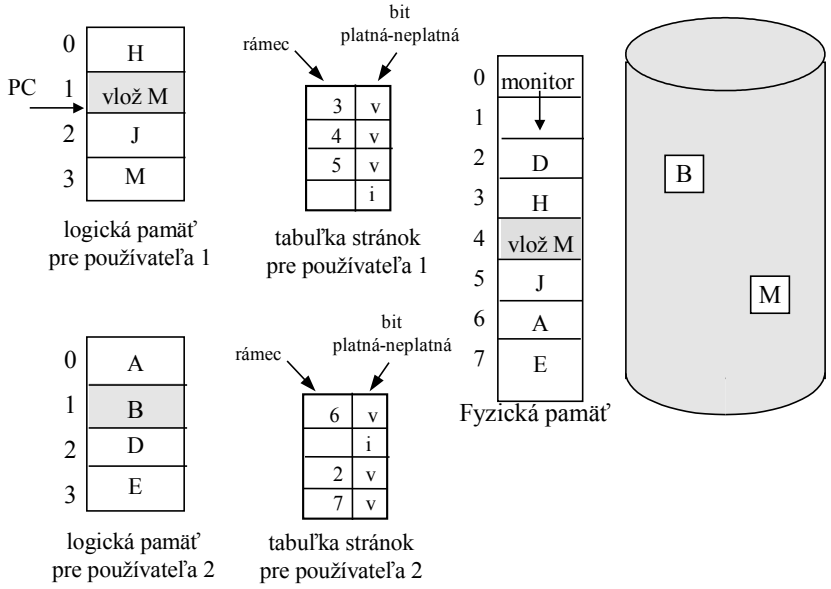
To znamená, že ak chceme udržiavať spomalenie kvôli stránkovaniu na žiadosť na dobrej úrovni, môžeme dovoliť menej ako 1 výpadok na 2 500 000 odkazov na pamäť.

Ďalší aspekt stránkovania na žiadosť je obsluha swapovacieho priestoru. Diskové operácie pre tento priestor sú obyčajne rýchlejšie, pretože sa prideľujú väčšie bloky a pri prideľovaní sa nepoužívajú nepriame spôsoby. Pri odštartovaní procesu sa obyčajne všetky stránky procesu presunú do swapovacieho priestoru. Ak tento priestor je obmedzený, používa sa modifikácia tejto metódy pre binárne súbory - tieto sa prenášajú do pamäte rovno zo súborového systému (nie zo swapovacieho priestoru) a na disk späť sa nezapisujú, pretože počas svojho vykonania sa nemodifikujú. Systém UNIX BSD používa dodatočnú modifikáciu, a to takú, že prvýkrát stránku načítava zo súborového systému a po jej nahradení ju zapisuje do swapovacieho priestoru, odkiaľ ju pri ďalších odkazoch načítava. Tým sa šetrí swapovací priestor a nachádzajú sa v ňom len stránky, ktoré sa skutočne používajú.

10.2.2 Nahradzovanie stránok

Výpadok stránky nie je seriózny problém, ako vyplýva z doteraz uvedeného, pretože každá stránka spôsobuje výpadok nanajvýš raz, keď sa na ňu odkazuje prvý krát. Ale táto úvaha, nie je celkom pravdivá. Predpokladajme, že proces o 10 stránkach skutočne používa len polovicu z nich. Potom stránkovanie na žiadosť ušetrí polovicu V/V pre zavedenie do pamäte stránok, ktoré nebudú nikdy použité. V tejto situácii môžeme zvýšiť úroveň multiprogramovania tak, že spustíme dvakrát viacej procesov. Takže ak máme 40 rámcov, môžeme spustiť 8 procesov miesto pôvodných 4, požadujúcich 10 rámcov. Je však možné, že každý z týchto procesov pre nejakú sadu vstupných údajov náhle bude potrebovať všetkých 10 stránok. Táto situácia nie je veľmi pravdepodobná, ale jej pravdepodobnosť sa zvyšuje so zvýšením úrovne multiprogramovania.

Takú situáciu si ukážeme na Obr. 10.5. Počas vykonania procesu sa vyskytne výpadok stránky. HW spôsobí prerušenie do operačného systému, ktorý testuje vnútorné tabuľky pre určenie príčiny prerušenia. V prípade, že sa jedná o výpadok stránky je potrebné požadovanú stránku zaviesť do pamäte, ale operačný systém zistí, že všetky rámce sú obsadené. V tomto bode sa situácia môže vyriešiť niekoľkými spôsobmi. Jedna z nich je, že ukončíme proces. Avšak toto riešenie je neprípustné, pretože stránkovanie na žiadosť by malo zlepšovať priepustnosť a výkon systému, pričom by malo zostať transparentné pre používateľa.

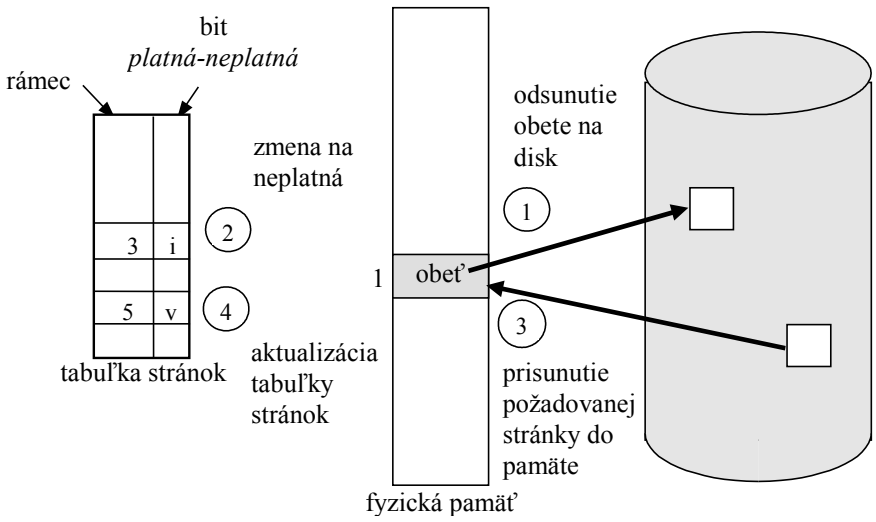


Obr. 10.5 Ukážka potreby nahradzovania stránok v pamäti

Ďalšia možnosť je odsunúť proces na disk, čím sa zníži úroveň multiprogramovania a uvoľnia sa všetky rámce procesu. Táto idea je využívaná v prípadoch zahltenia systému, ako bude spomenuté ďalej. V našom prípade sa používa metóda nazvaná *nahradzovanie stránok.*

Nahradzovanie stránok spočíva v následných činnostiach: Ak nie je voľný rámec, nájdeme taký, ktorý sa momentálne nevyužíva a do neho prisunieme požadovanú stránku. Modifikácia procedúry, ktorá obsluhuje výpadok stránky, teraz bude obsahovať nahradzovanie stránok a bude vyzerať nasledovne (Obr. 10.6):

1. Nájde umiestnenie požadovanej stránky na disku.
2. Nájde voľný rámec:
   a) ak je voľný rámec, použije ho,
   b) ak nie je voľný rámec, použije algoritmus nahradzovania stránok a vyberie *stránku – obeť,*
   c) zapíše stránku - obeť na disk, aktualizuje tabuľku stránok a tabuľku rámcov.
3. Načíta požadovanú stránku do uvoľneného rámca, zmení tabuľku stránok a tabuľku rámcov.
4. Reštartuje používateľský proces.



Obr. 10.6 Kroky pri nahradzovaní stránok

Ak nie sú voľné rámce, vykonajú sa dva prenosy stránok. Tento fakt značne zhoršuje efektívny čas prístupu pre obsluhu výpadku stránky. Riešenie tejto situácie spočíva v použití *bitu modifikácie.* Každý rámec musí mať bit modifikácie, ktorý sa nastaví vtedy, keď sa na stránke zapisuje. Keď sa stránka vyberie pre odsunutie, najskôr sa skontroluje bit modifikácie. Ak je nastavený, to znamená že stránka bola modifikovaná a treba ju zapísať na disk. Ale ak stránka nebola modifikovaná, netreba ju zapisovať, pretože v tom istom tvare ju máme na disku. Táto technika sa úspešne aplikuje hlavne na stránky s prístupom len na čítanie (napr. binárny kód) a takto sa čas potrebný na prenosy stránok znižuje o polovicu.

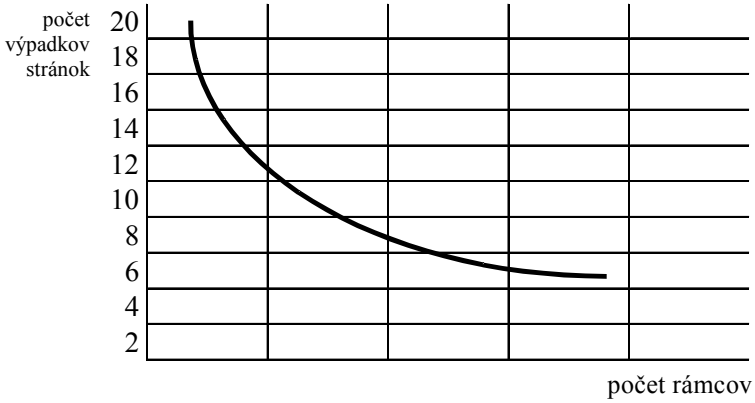
Nahradzovanie stránok je základ techniky stránkovania na žiadosť. Pri implementácii tejto techniky je potrebné vyriešiť dva hlavné problémy: prideľovanie voľných rámcov a nahradzovanie stránok. Návrh algoritmov pre vyriešenie týchto problémov je veľmi dôležitou otázkou a môže značne ovplyvniť výkon celého systému.

10.2.3 Algoritmy nahradzovania stránok

Existuje veľa algoritmov pre nahradzovanie stránok. Kritérium, ktoré majú tieto algoritmy spĺňať, je spoločné - dosiahnuť vhodnou náhradou stránky najmenší počet výpadkov stránok.

Algoritmy nahradzovania stránok sa hodnotia spustením určitého *reťazca odkazov* na pamäť a počítaním výpadkov, ktoré sa vyskytnú pri implementácii príslušného algoritmu. Samozrejme pri tom musíme mať na zreteli veľkosť stránky a počet rámcov, ktorými proces disponuje. Obyčajne s nárastom počtu rámcov počty výpadkov stránok klesajú, kým sa ustália na určitú minimálnu úroveň. Typická krivka závislosti počtu výpadkov od počtu rámcov je uvedená na Obr. 10.7. Pre ilustráciu nahradzovacích algoritmov použijeme jeden reťazec odkazov na stránky pre fyzickú pamäť s tromi rámcami:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1



Obr. 10.7 Závislosť výpadkov stránok od počtu rámcov

10.2.3.1 Algoritmus FIFO

FIFO je najjednoduchší algoritmus nahradzovania stránok. Tento algoritmus priraďuje každej stránke čas jej príchodu do pamäte. Keď sa má niektorá stránka nahradiť, ako obeť sa vyberie stránka, ktorá je najdlhšie v pamäti. Pri tom nie je potrebné presne zapisovať čas, ale stačí vytvoriť FIFO front v pamäti. Potom sa nahradí stránka, ktorá je na začiatku frontu a číslo novej stránky sa pridá na koniec frontu.

V našom príklade na začiatku všetky 3 rámce sú prázdne. Prvé 3 odkazy (7, 0, 1) spôsobia 3 výpadky, pričom sa do voľných rámcov zapíšu požadované stránky. Ďalší odkaz (2) spôsobí nahradenie stránky 7, pretože podľa algoritmu FIFO práve táto stránka má byť nahradená - je na začiatku frontu podľa času príchodu do pamäte. Ďalšie odkazy sa spracovávajú rovnakým spôsobom a konečný výsledok je 15 výpadkov stránok.

Algoritmus FIFO je jednoduchý. Jeho výkonnosť, ale nie je vždy dobrá. Nahradená stránka môže obsahovať inicializačný modul, ktorý už nebude potrebný, ale tiež to môže byť často používaná stránka, ktorá ihneď po odsunutí spôsobí nový výpadok stránky.

	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
rámec 1	7	7	7	2		2	2	4	4	4	0		0	0				7	7	7
rámec 2		0	0	0		3	3	3	2	2	2		1	1				1	0	0
rámec 3			1	1		1	0	0	0	3	3		3	2				2	2	1

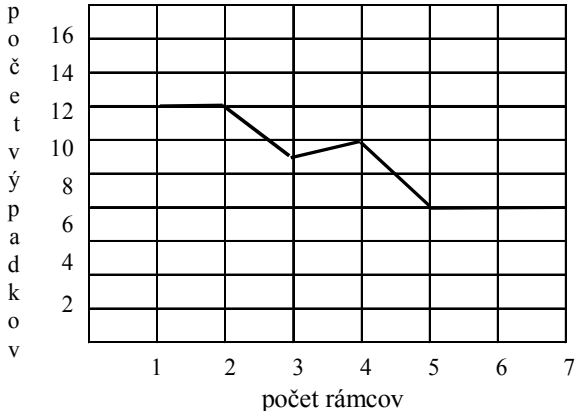
celkom 15 výpadkov

Obr. 10.9 Nahradzovací algoritmus FIFO

Algoritmus FIFO pri výmene stránky, ktorá je často používaná, môže spôsobiť zvýšenie počtu výpadkov. Tento algoritmus preukazuje ešte jednu ďalšiu neočakávanú vlastnosť. Na Obr. 10.9 je ukázaná závislosť počtu výpadkov od počtu rámcov pre proces s reťazcom odkazov:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Z obrázku je vidieť, že počet výpadkov pri použití 4 rámcov je väčší ako počet výpadkov pre 3 rámce. Tento výsledok je známy v literatúre ako *Belady-ho anomália.*



Obr. 10.9 Krivka závislosti výpadkov stránok pre FIFO algoritmus

10.2.3.2 Optimálny algoritmus

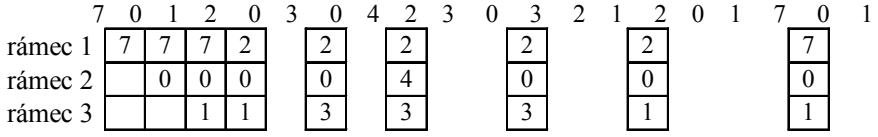
Belady-ho anomália bola objavená pri hľadaní optimálneho nahradzovacieho algoritmu. Tento algoritmus má najmenší počet výpadkov stránok zo všetkých algoritmov. Optimálny algoritmus nepreukazuje Belady-ho anomáliu.

Optimálny nahradzovací algoritmus je algoritmus, ktorý nahradzuje stránku, ktorá nebude potrebná najdlhšiu dobu. Použitie tohto algoritmu zaručuje najmenší možný počet výpadkov stránok pri danom pevnom počte rámcov.

Reťazec odkazov je:

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

a aplikácia optimálneho algoritmu na tento reťazec je nasledovná:



Celkový počet výpadkov stránok - 9

Obr. 10.10 Optimálny nahradzovací algoritmus stránok

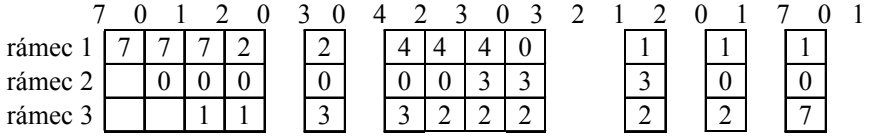
Keď aplikujeme tento algoritmus na našom príklade, získame celkový počet výpadkov stránok 9 (Obr. 10.10). Prvé tri odkazy spôsobia výpadky a zaplnia sa rámce. Odkaz na stránku 2 nahradí stránku 7, pretože táto stránka nebude požadovaná až do 18-teho odkazu, zatiaľ čo stránka 0 bude potrebná v 5-tom odkaze a stránka 1 v 14-tom. Odkaz na stránku 3 nahradí stránku 1, pretože táto stránka bude ako posledná požadovaná z tých, ktoré sú v rámcoch. Keď budeme takto pokračovať s nahradzovaním, získame celkový počet výpadkov 9. S takýmto počtom výpadkov sa optimálny algoritmus ukazuje oveľa lepším ako algoritmus FIFO.

Optimálny algoritmus sa ťažko realizuje, pretože vyžaduje znalosť budúcich odkazov na stránky. Tento algoritmus je užitočný pre porovnávanie pri vývoji nových algoritmov.

10.2.3.3 Algoritmus LRU - najdlhšie nepoužívaná

Optimálny algoritmus je ťažko realizovateľný, ale jeho aproximácia je možná. Hlavný rozdiel medzi algoritmom FIFO a optimálnym algoritmom je v tom, že algoritmus FIFO používa čas, kedy stránka bola zavedená do pamäte a optimálny algoritmus využíva čas, kedy sa má stránka použiť. Ak použijeme blízku minulosť ako aproximáciu pre blízku budúcnosť, budeme nahradzovať stránku,

ktorá nebola použitá najdlhšiu dobu (Obr. 10.11). Tento prístup charakterizuje algoritmus LRU (Least Recently Used - najdlhšie nepoužívaná).



celkom 12 výpadkov

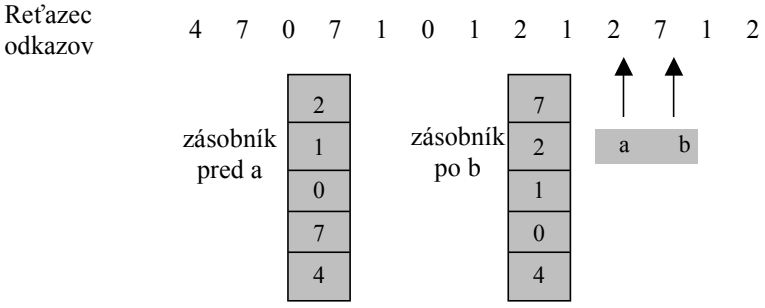
Obr. 10.11 Nahradzovací algoritmus LRU

Pri použití algoritmu LRU ku každej stránke je priradený čas posledného použitia stránky. Keď sa má nahradiť stránka, vyberá sa taká, ktorá najdlhšie nebola používaná.

Výsledok aplikovania tohto algoritmu na reťazci odkazov z nášho príkladu je 12 výpadkov stránok, ako bolo ukázané na Obr. 10.11. Tento algoritmus sa často používa a je považovaný za veľmi dobrý. Hlavný problém je ako implementovať nahradzovanie podľa LRU. Potrebné je určiť poradie rámcov, ktoré je odvodené od času ich posledného použitia. Používajú sa dve implementácie:

- ***Počítadlami****: V najjednoduchšom prípade sa ku každej položke tabuľky stránok pripojí pole, ktoré obsahuje čas posledného použitia a k CPU sa pridávajú logické hodiny alebo počítadlo. Hodnota hodín sa zvyšuje pri každom odkaze na pamäť. Vždy, keď sa odkazuje na stránku, obsah logických hodín sa skopíruje do poľa času použitia pre túto stránku. Potom sa nahradzuje stránka, ktorá má najmenšiu hodnotu „času“. Tento prístup vyžaduje prehľadávanie tabuľky stránok pre nájdenie najdlhšie nepoužívanej stránky a zápis do tabuľky stránok pri každom prístupe k pamäti. Taktiež sa čas musí meniť pri zmene tabuliek stránok.*

  • **Zásobník:** Iný prístup k implementácii algoritmu LRU je uchovanie čísel stránok v zásobníku. Vždy pri odkaze na stránku sa jej číslo presunie na vrch zásobníka a tak na vrchu zásobníka je vždy stránka, na ktorú sa posledne odkazovalo, a na spodku je najdlhšie nepoužívaná stránka (Obr. 10.12).



Obr. 10.12 Použitie zásobníka pre zaznamenanie posledných odkazov na stránku

Najlepšia implementácia takého zásobníka je pomocou obojsmerne zreťazeného zoznamu s ukazovateľmi na hlavu zoznamu a na jeho koniec.

Algoritmus LRU nepreukazuje Belady-ho anomáliu.

10.2.3.4 LRU aproximačné algoritmy

Algoritmus druhej šance

Algoritmus druhej šance (Clock alebo Second chance) je variantom algoritmu FIFO. Využíva tzv. *referenčný bit,* ktorý je pridaný ku každému rámcu. Tento bit sa hardvérovo nastavuje vždy, keď sa so stránkou pracuje. Keď stránka je vybraná na odsunutie podľa algoritmu FIFO, skúma sa jej

referenčný bit. Ak je nastavený na 0, stránka sa odsunie, ak je nastavený na 1, stránka dostáva „druhú šancu“, referenčný bit sa vynuluje a jej čas sa nastaví na momentálny čas. A stránka sa presunie na koniec frontu. Týmto spôsobom stránka, ktorá je stále používaná môže zostať trvalé v pamäti.

Tento algoritmus sa dá implementovať ako kruhový front. Ukazovateľ ukazuje na stránku, ktorá má byť nahradená. Pokiaľ takáto stránka má nastavený referenčný bit, ukazovateľ postupuje ďalej, kým nenájde stránku s nulovým referenčným bitom, pričom nastavuje bity prejdených stránok na 0. Ak všetky referenčné bity sú nastavené, algoritmus druhej šance degeneruje na algoritmus FIFO.

Vylepšený algoritmus druhej šance

Algoritmus druhej šance sa dá vylepšiť použitím dvoch bitov, ktoré sa nastavujú - jeden pri odkaze na stránku (R), druhý pri zápise na stránku (W). Keď zoberieme do úvahy všetky možné kombinácie hodnôt tejto dvojice bitov (v poradí RW), sú tu nasledovné triedy:

00 - na stránku nebolo odkazované a nebola modifikovaná, najlepšia pre nahradenie,

01 - na stránku nebolo odkazované, ale bolo na ňu zapisované, nie je tak dobrá pre nahradenie, lebo sa jej obsah bude musieť zapísať na disk,

10 - na stránku bolo odkazované, ale nebolo na ňu zapisované, pravdepodobne bude v najbližšej dobe znova použitá,

11 - na stránku bolo odkazované a bolo na ňu zapisované, pravdepodobne bude v najbližšej dobe znova použitá a jej obsah sa bude musieť zapísať na disk.

Podľa hodnôt bitov, každá stránka patrí do jednej z týchto tried. Výber stránky na odsunutie sa uskutočňuje obdobne ako u algoritmu druhej šance - vyberie sa prvá stránka z najnižšej triedy, ktorá nie je prázdna.

Tento algoritmus je použitý v správe virtuálnej pamäte systému MacIntosh.

Aproximačný LRU algoritmus pomocou dodatočných referenčných bitov

Presnejšiu informáciu o frekvencii využívania stránky môžeme získať zaznamenávaním hodnoty referenčného bitu v pravidelných intervaloch (napr. každých 100 ms). Za týmto účelom sa dá použiť jeden posuvný register pre každý rámec. Najvyšší bit tohoto registra sa nastavuje pri každom odkaze na stránku. V pravidelných intervaloch časovač vyvoláva prerušenie a odovzdáva riadenie systému. Potom operačný systém posúva všetky referenčné bity o jedno miesto doprava, pričom najnižší bit sa stráca. Takto sa do bitov tohoto registra zaznamenáva história využívania stránky, ktorá slúži pre aproximáciu budúceho použitia stránky. Napr. ak register má hodnotu 00000000, to znamená, že za posledných 8 intervalov stránka nebola použitá. Stránka, ktorá má register s hodnotou 01001001 bola použitá dávnejšie ako stránka, ktorej register má hodnotu 11010110. Ak sa pozeráme na tieto hodnoty ako na celé čísla bez znamienka, potom stránka s najnižšou hodnotou je LRU stránka a tá má byť nahradená. Ak je viacej stránok s rovnakým najmenším číslom, výber jednej z nich sa uskutočňuje na základe FIFO.

Tento algoritmus dáva dobré výsledky, ale vyžaduje HW podporu.

10.2.4 Prideľovanie rámcov

Pri implementácii virtualizácie stránkovaním na žiadosť je treba rozhodnúť o taktike prideľovania voľných rámcov. Problém je zložitejší, keď sa jedná o kombináciu stránkovania na žiadosť s multiprogramovaním. Je samozrejmé, že sa nemôže prideliť viac rámcov ako vlastní systém. Na druhej strane je potrebné prideliť aspoň minimálne množstvo rámcov procesu. Obyčajne čím menej rámcov vlastní proces, tým viac výpadkov stránok vyvoláva, a tým sa spomaľuje chod systému.

Pri prideľovaní rámcov pre proces treba mať na vedomí minimálny počet rámcov, ktorý je určený inštrukčným súborom počítača. Už sme spomenuli, že keď pri vykonaní inštrukcie vznikne

výpadok stránky, inštrukcia musí byť reštartovaná po natiahnutí stránky do pamäte. To znamená, že minimálny počet rámcov, ktoré majú byť pridelené procesu, vyplýva z maximálneho počtu stránok (adries), na ktoré sa pri vykonaní jednej inštrukcie môže proces odvolať.

Minimálny počet rámcov je určený HW architektúrou počítača. Ak máme počítač s inštrukčným súborom, kde každá inštrukcia môže adresovať len jednu bunku v pamäti, potom proces musí mať jeden rámec pre inštrukciu a jeden pre stránku, kde sa môže nachádzať adresa. Ak k tomu zoberieme do úvahy možnosť jednoúrovňového nepriameho adresovania, potom minimálny počet rámcov, ktoré musí mať proces, je 3. Napr. inštrukcia, ktorá je na stránke 8, sa odkazuje na adresu zo stránky 0, kde je nepriamy odkaz na adresu zo stránky 24.

Uvedený príklad je veľmi jednoduchý. Je veľa architektúr, ktoré dovoľujú niekoľko úrovní nepriameho adresovania. Aby sa predišlo extrémnemu prípadu, kedy každá adresa bude nepriamym odkazom na ďalšiu nepriamu adresu, úrovne nepriameho adresovania sú obmedzené.

10.2.5 Algoritmy prideľovania rámcov

Najjednoduchší spôsob prideľovania je ***spravodlivé rozdelenie medzi procesmi,*** t.j. pre m rámcov a n procesov to bude m/n rámcov pre proces. Takéto rozdelenie neberie do úvahy veľkosť procesu a môže viesť k plytvaniu voľných rámcov u malých procesov na jednej strane a k nedostatku rámcov u veľkých procesov na druhej strane.

Tento nedostatok rieši ***proporcionálne prideľovanie***. Každý proces dostáva rámce podľa svojej veľkosti. Ak celkový počet rámcov je m, požiadavky procesu i na pamäť sú s_i , pridelíme a_i rámcov procesu p_i, kde a_i je približne

$$a_i = (s_i / S) \times m$$

kde

$$S = \Sigma s_i$$

m - počet rámcov

n - počet procesov

s_i - požiadavky i-tého procesu

S - požiadavky všetkých procesov

Výsledný počet rámcov sa samozrejme zaokrúhli na celé číslo, ktoré musí byť menšie ako m a väčšie ako je minimálny počet určený inštrukčným súborom.

Uvedené spôsoby rozdeľovania rámcov medzi procesmi neberú do úvahy priority procesov. Je logické, že procesy s vyššou prioritou by mali dostať viac rámcov, aby sa ich vykonanie urýchlilo. Jedno z možných riešení je použiť proporcionálne prideľovanie, kde počet rámcov bude závisieť od rozmerov a priority procesu.

10.2.6 Globálne a lokálne nahradzovanie

Ďalší dôležitý faktor, ktorý ovplyvňuje prideľovanie je množina rámcov z ktorých sa vyberá stránka, ktorá sa bude nahradzovať. Sú možné dva prístupy: *globálny a lokálny*. Pri globálnom nahradzovaní sa stránka - obeť vyberá spomedzi všetkých stránok v operačnej pamäti, bez ohľadu na príslušnosť k procesu. Pri lokálnom nahradzovaní sa stránka - obeť vyberá len spomedzi stránok, ktoré patria procesu, ktorý vyvolal výpadok stránky.

Pri aplikácii lokálneho nahradzovania proces dostane určité množstvo rámcov, ktoré sa ďalej nezväčšuje. Počet pridelených rámcov sa odvodzuje napr. z priority a veľkosti procesu. Pri globálnom nahradzovaní počet rámcov je obmedzený len veľkosťou operačnej pamäte.

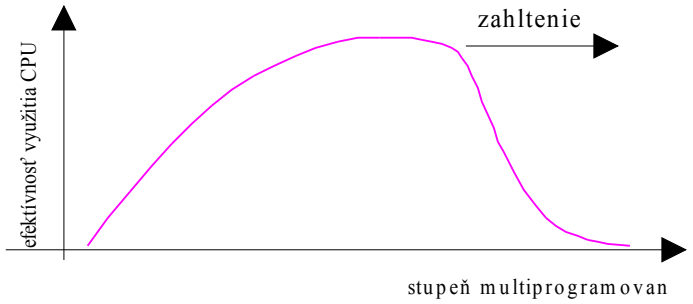
10.2.7 Zahltenie

Ak počet rámcov pridelených procesom, umiestnených v pamäti klesne na hodnotu blízku technickému minimu, vzrastie počet výpadkov stránok. Proces pritom bude často používať všetky stránky, ktoré má v pamäti. V podobnej situácii sa môžu nachádzať všetky procesy v pamäti.

Vždy, keď sa vyberie obeť na odsunutie z pamäte, vzápätí odkaz na tú stránku vyvolá výpadok. V takej situácii sa môže stať, že frekvencia výpadkov prevýši maximálne možnú frekvenciu výmen a procesor venuje až 99% svojho času na riadenie výmen stránok. Takáto situácia sa nazýva ***zahltenie*** *(thrashing)*. Proces je v stave zahltenia ak trávi viac času výmenou stránok ako výpočtom. Zahltenie bolo chorobou prvých operačných systémov, u ktorých nebola prevencia proti tomuto stavu.

Na Obr. 10.13 je uvedený graf závislosti efektívnosti využitia CPU od úrovne multiprogramovania, t.j. od počtu procesov v pamäti. S nárastom počtu procesov aj efektívnosť využitia CPU rastie až dosiahne maximum. Ak sa počet procesov zvyšuje ďalej, výkon systému prudko klesá a nastáva zahltenie. Východisko z tejto situácie je zmenšenie počtu procesov v pamäti.

Zahltenie môže byť obmedzené aplikáciou lokálneho nahradzovania. Vtedy proces, ktorý sa dostane do stavu zahltenia, nemôže odoberať rámce iným procesom a tak rozširovať tento stav. Pretože taký proces stále vyvoláva výpadky stránok, väčšinu času bude vo fronte swapovacieho zariadenia, čím sa zväčší doba obsluhy pre výmenu stránok a výkon poklesne.



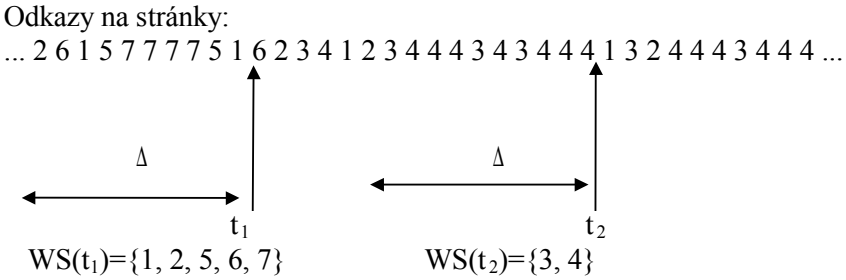
Obr. 10.13 Zahltenie

Aby sa zahltenie nemohlo vyskytnúť, procesy musia mať toľko stránok, koľko potrebujú. Pre odhad potrieb procesu sa aplikuje niekoľko techník. Prvá z nich je založená na ***princípe lokality***. Tento princíp hovorí, že počas svojho vykonania proces má niekoľko lokalít, kedy aktívne využíva len určitú podmnožinu svojich stránok, a tie lokality sa môžu prelínať. Taká lokalita napr. môže byť podprogram, kedy proces pracuje s lokálnymi premennými a podmnožinou globálnych premenných. Všetky „slušné“ programy sa chovajú takýmto spôsobom.

10.2.7.1 Pracovná sada

Model pracovnej sady je založený na princípe lokality. Tento model využíva parameter Δ, ktorý sa používa pre definovanie veľkosti *okna pracovnej sady* (*working-set window)*. Posledných Δ odkazov na stránky tvorí *pracovnú sadu* (*working-set)* procesu. Ak sa stránka aktívne používa v danom čase, patrí do pracovnej sady. Ak stránka nie je ďalej potrebná, odstráni sa z pracovnej sady po Δ časových jednotkách. Takže pracovná sada je aproximácia lokality programu.

Napr. ak máme postupnosť odkazov z príkladu na Obr. 10.14 a ak Δ=10 odkazov, potom pracovná sada v čase t_1 je {1, 2, 5, 6, 7}, ale v čase t_2 sa zmení na {3,4}.



Obr. 10.14 Model pracovnej sady

Presnosť pracovnej sady závisí na výbere parametra Δ. Ak je Δ veľmi malý, nebude pokrývať celú lokalitu, ak je veľmi veľký, bude pokrývať niekoľko lokalít.

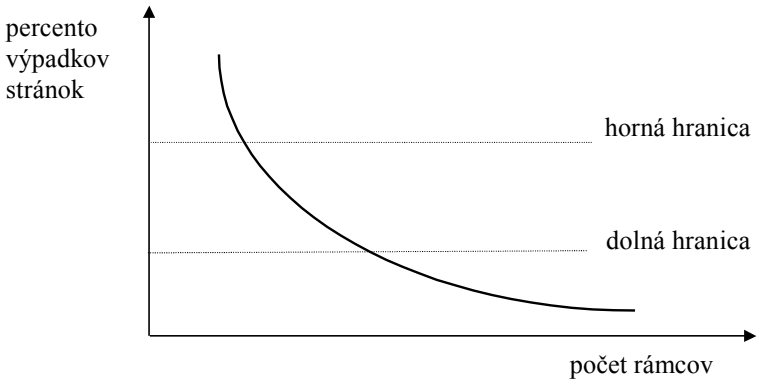
Veľkosť pracovnej sady je veľmi dôležitá. Ak označíme *D* súčet pracovných sád všetkých procesov a *m* je celkový počet rámcov, potom zahltenie môže nastať ak $D > m$.

Použitie pracovnej sady je jednoduché. Operačný systém sleduje pracovné sady procesov, ktoré bežia, a prideľuje im taký počet rámcov, aby v pamäti mohla byť celá pracovná sada. Ak toto nie je možné, vyberie niektorý proces a pozastaví ho. Tým sa uvoľní určitý počet rámcov, ktorý sa poskytne ostatným procesom. Suspendovaný proces sa spustí neskôr.

Využitie pracovnej sady je účinná prevencia proti zahlteniu, pričom sa úroveň multiprogramovania udržuje na maximálne vysokej úrovni. Problémom je sústavné sledovanie pracovných sád procesov, ktoré musí využívať možnosti konkrétnej HW architektúry počítača.

10.2.7.2 Frekvencia výpadkov stránok

Iný spôsob ako zabrániť zahlteniu je sledovať frekvenciu výpadkov stránok procesov. Ak proces vyvoláva príliš často výpadky stránok, to znamená, že potrebuje viac rámcov. Obdobne, ak výpadky sú zriedkavé, proces má veľa rámcov. Na Obr. 10.15 je uvedená závislosť výpadkov stránok od počtu rámcov pre proces. Ak sa proces nachádza nad hornou hranicou výpadkov, treba zvýšiť počet jeho rámcov. Ak sa proces nachádza pod dolnou hranicou výpadkov, treba znížiť počet jeho rámcov.



Obr. 10.15 Frekvencia výpadkov stránok

10.2.8 Úvahy na záver

Typickou vlastnosťou systémov s čistým stránkovaním na žiadosť je veľký počet výpadkov stránok po spustení procesu. To je výsledkom hľadania počiatočnej lokality procesu. Podobný prípad sa môže vyskytnúť aj v inom okamihu počas behu procesu. Prostriedkom ako sa vyhnúť tejto situácii je tzv. predstránkovanie (prepaging). Predstránkovaním sa snažíme zaviesť do pamäte naraz všetky stránky, ktoré proces potrebuje.

V systémoch, ktoré používajú model pracovnej sady, sa uchováva zoznam stránok, ktoré patria do pracovnej sady. Ak proces musí byť pozastavený napr. kvôli V/V operácii, jeho pracovná sada sa odpamätá a po opätovnom spustení procesu sa do pamäte presunie naraz celá pracovná sada.

Veľkosť stránky je faktor, ktorý značne ovplyvňuje výkonnosť stránkovania na žiadosť. Pokiaľ sa operačný systém navrhuje pre existujúci HW, možnosti výberu sú minimálne. Naopak, keď sa navrhuje nový počítač, musí sa vybrať najvhodnejšia veľkosť. Problém spočíva v tom, že neexistuje najlepšie riešenie, pretože výber veľkosti stránky je ovplyvnený mnohými protichodnými požiadavkami. Veľkosť stránky sa pohybuje od 2^9 (512) bajtov do 2^{14} (16384) bajtov.

Faktory, ktoré ovplyvňujú výber veľkosti stránky:

- **Tabuľka stránok** - čím väčšia stránka, tým menšia tabuľka stránok a naopak. Pretože je potrebné pamätať si tabuľky stránok všetkých procesov, spotrebujeme veľkú časť pamäte. Napr. Ak stránka je 1024 bajtov, tabuľka stránok pre virtuálnu pamäť s veľkosťou 4 MB (2^{22}) bude mať 4096 položiek, ale pri veľkosť stránky 8192 bajtov bude mať len 512 položiek!
- **Interná fragmentácia** - obyčajne veľkosť procesov sa nezhoduje s celým násobkom veľkosti stránky a časť poslednej stránky je nevyužitá. V priemere nevyužitá časť sa rovná polovici stránky. Ak stránka je 1024 bajtov, priemerná strata je 512 bajtov, ale pri stránke 8192 bajtov, je strata 4096 bajtov!
- **Čas pre V/V operácie** - čas V/V operácie pre prenos stránky z/na disk pozostáva z času pre nastavenie hláv, z reakčného času a času na prenos dát. Čas pre nastavenie hláv sa pohybuje okolo 20 *ms* a reakčný čas je okolo 8 *ms*. Čas pre prenos 512 bajtov pri rýchlosti 2 MB/sek je 0.2 *ms*. Celkový čas V/V operácie je 28.2 ms, z čoho prenos tvorí len 1%. Pre prenos stránky s dvojnásobnou veľkosťou, celkový čas sa zväčší len na 28.4 ms, ale bude až 56.4 *ms* pre prenos dvoch stránok o 512 bajtov! Takže minimalizácia času V/V operácie vyžaduje väčšiu stránku.
- **Lokality programu** - ak stránka je malá, presnejšie sa môže prispôsobiť lokalitám programu. To má za následok zníženie počtu V/V operácií, pretože bude menej výpadkov stránok a ešte aj menej pridelenej pamäte.
- **Výpadky stránok** - čím je stránka väčšia, tým menej výpadkov sa bude vyskytovať a tým menej prenosov stránok bude potrebných. Pre minimalizáciu výpadkov je potrebná veľká stránka.

Súčasný trend je smerom k zväčšeniu veľkosti stránok. Intel 80386 má stránku 4KB, Motorola 68030 dovoľuje stránku od 256 bajtov do 32 KB! Tento trend je pravdepodobne výsledkom zväčšenia rýchlosti procesorov a zväčšenia kapacity pamäte.

Efektívnosť stránkovania a tým aj celkový výkon systému závisí aj od dátových a programových štruktúr. Napr., zoberme do úvahy typický kód Pascalovského programu pre inicializáciu poľa o veľkosti 128 prvkov:

```
var A: array [1..128] of array [1..128] of integer;
... for j:=1 to 128 do
        for i:=1 to 128 do
            A[i,j]:=0;
```

Prvky poľa sú uložené po riadkoch: *A[1,1], A[1,2],.... A[1,128], A[2,1],... ,A[128,128]*. Ak stránka je128 bajtov, potom na každej stránke je uložený jeden riadok poľa a tento kód bude inicializovať po jednom slove na každú stránku. Ak proces má k dispozícii menej ako 128 stránok, výsledný počet výpadkov stránok bude

128x128 = 16384 výpadkov.

Ak tento kód zmeníme na:

```
var A: array [1..128] of array [1..128] of integer;
... for i:=1 to 128 do
        for j:=1 to 128 do
            A[i,j]:=0;
```

výpadkov bude len 128!

Z uvedeného vyplýva, že starostlivý výber dátových štruktúr môže silne ovplyvniť lokality programu a následne znížiť výpadky stránok. Údajový typ zásobník ma dobrú lokalitu, lebo pracuje stále s položkami na vrchu zásobníka. Rozptyľovacia tabuľka (hash table) má zlú lokalitu, lebo rozptyľuje odkazy. Samozrejme toto kritérium hodnotenia dátových štruktúr je len dodatočné okrem najdôležitejších, ako sú rýchlosť prehľadávania, celkový počet pamäťových odkazov a celkový počet stránok, na ktoré sa odkazuje.

Výber programovacieho jazyka tiež môže ovplyvniť stránkovanie. Napr. jazyk LISP často používa ukazovatele, ktoré majú tendenciu sprístupňovať pamäťové miesta náhodne. Na druhej strane Pascal používa málo ukazovateľov a programy majú lepšiu lokalitu, takže v prostredí s virtuálnou pamäťou sa budú vykonávať rýchlejšie.

Pri stránkovaní na žiadosť je potrebné vyriešiť ešte jeden problém, ktorý vzniká v prípadoch, keď je stránka uzamknutá v pamäti. Tento prípad sa môže vyskytnúť vtedy, keď sa uskutočňuje V/V operácia medzi periférnym zariadením a virtuálnou pamäťou procesu. Môže vzniknúť napr. takáto situácia: V/V operácia odštartuje a požiadavka sa zaradí do frontu príslušného zariadenia, ale procesor je medzitým pridelený inému procesu. Ak tento proces spôsobí výpadok stránky, môže sa stať, že bude nahradená práve stránka, ktorá obsahuje bufer pre prenos dát. Stránka, ktorá sa potom bude nachádzať v tom rámci, už nebude tá pôvodná, s ktorou sa odštartovala V/V operácia! Takéto situácie sa nesmú vyskytovať!

Existujú dve bežné riešenia tohoto problému. Prvé je také, že zakazuje vykonávať V/V operácie do používateľskej pamäte. Dáta sa kopírujú vždy len medzi používateľskou a systémovou pamäťou. V/V operácie sa potom vykonávajú len medzi systémovou pamäťou a V/V zariadením. Druhé riešenie je uzamykať takéto stránky v pamäti, t.j. ak stránka je uzamknutá, nepodlieha nahradzovaniu. Za týmto účelom sa používa jeden bit priradený každému rámcu, ktorý indikuje či je stránka uzamknutá alebo nie. Tento spôsob môže byť nebezpečný, pretože sa môže stať, že sa stránka uzamkne, ale neodomkne nikdy. Systém MacIntosh ponúka mechanizmus na uzamknutie, ale v tomto prípade, t.j. v jednoužívateľskom systéme, aj keby používateľ zneužil tento mechanizmus, šlo by to len na jeho úkor. Vo viacužívateľských systémoch je potrebné opatrnejšie premyslieť, komu sa tento mechanizmus sprístupní.

10.3 Segmentácia na žiadosť

Stránkovanie na žiadosť je pokladané za najefektívnejšiu metódu virtualizácie, ale ako sme v predchádzajúcej časti videli, pre implementáciu potrebuje veľkú HW podporu. Ak takáto podpora chýba, môže sa použiť iná vitrualizačná technika a to *segmentácia na žiadosť*.

Pri segmentování na žiadosť sa program zavádza do pamäte po segmentoch. Predmetom výmeny sú segmenty. Pri odkaze na segment, ktorý nie je v pamäti, sa vygeneruje prerušenie pre *výpadok segmentu (segment fault)*. Správa pamäte nájde pre požadovaný segment vhodný úsek v pamäti a presunie ho tam. Využíva pritom striasanie. Segmentácia na žiadosť má silnejšie väzby na štruktúru programu a minimalizuje vnútornú fragmentáciu. Pravdepodobnosť vzniku zahltenia je tu menšia ako u stránkovania na žiadosť. Nevýhodou tohoto spôsobu správy pamäte je nebezpečenstvo vonkajšej fragmentácie.

Príklad: Operačný systém OS/2, ktorý využíva Intel 80286, nemá podporu pre stránkovanie, a preto prideľuje pamäť procesom po segmentoch, t.j. implementuje segmentáciu na žiadosť. Informácie o každom segmente - veľkosť, umiestnenie, ochrana, sa uchovávajú v tzv. *descriptoroch segmentov*. Descriptor segmentu obsahuje aj bit, ktorý označuje, či je segment v pamäti alebo nie. Pri každom

odkaze na segment správa pamäte skontroluje tento bit a ak segment nie je v pamäti, generuje prerušenie pre výpadok segmentu. Pre určenie vhodného kandidáta na odsunutie sa skontroluje ďalší *bit prístupu (accessed bit),* ktorý sa nastavuje, keď sa na segment odkáže pre čítanie alebo zápis. Pre každý segment v pamäti operačný systém udržuje front. Po každom časovom kvante operačný systém umiestni na začiatok frontu segmenty s nastaveným bitom prístupu a bity ostatných segmentov vynuluje. Takto sa front uchováva ako usporiadaný podľa časov prístupu k segmentom.

Keď sa vyskytne výpadok segmentu, správa pamäte najskôr zisťuje, či je voľný úsek s vhodnou veľkosťou. V prípade potreby sa pamäť môže aj striasť. Ak stále nie je vhodný úsek, pristúpi sa k nahradeniu niektorého zo segmentov v pamäti. Pre nahradenie sa vyberie segment, ktorý je na konci frontu. Ak úsek, ktorý sa uvoľní, je dostatočný pre umiestnenie požadovaného segmentu, ten sa tam presunie a descriptor segmentu sa zaktualizuje. Ináč sa vykoná striasanie a procedúra hľadania vhodného úseku sa opakuje.

Segmentácia na žiadosť má veľkú réžiu a to nevedie k optimálnemu využitiu prostriedkov systému. Ak daná HW platforma neposkytuje podporu pre implementáciu stránkovania na žiadosť, vhodnejšie je nevirtualizovať vôbec.

1 SPRÁVA SÚBOROV

Systém súborov je najviditeľnejšia časť operačného systému. Každý používateľ potrebuje narábať so svojimi dátami, preto často práve správa systému súborov do určitej miery určuje nakoľko, bude daný operačný systém obľúbený u širokej verejnosti.

Súborový systém pozostáva zo súborov a adresárov. Do súborov sa ukladajú dáta. Adresáre sú štruktúry, kde sa ukladajú informácie o súboroch. K problematike súborového systému patrí aj ochrana súborov.

11.1 Súbory

Počítačové systémy ukladajú dáta na niekoľkých typoch médií: na magnetických diskoch, na magnetických páskach, na optických diskoch, na disketách. Periférne zariadenia pre ukladanie dát sú obyčajne stále, ich obsah sa nestráca pri výpadku napätia alebo pri opätovnej inicializácii systému.

Pre jednoduchosť použitia operačný systém poskytuje jednotný logický pohľad na médiá. Operačný systém definuje logickú jednotku pre ukladanie dát, ktorá sa nazýva *súbor*. Súbor je pomenovaná množina príbuzných informácií, ktoré sú zaznamenané na perifémom ukladacom zariadení. Súbor je nezávislý od vlastností konkrétneho zariadenia.

Z hľadiska používateľa je súbor najmenšia časť z logického ukladacieho zariadenia, ktorá mu môže byť pridelená. Súbory môžu byť numerické, textové, alfanumerické a binárne.

Informácia v súbore je definovaná jeho tvorcom. Do súborov je možné uložiť mnoho typov informácií: zdrojové texty, objektové súbory, vykonateľné súbory, numerické údaje, zvukové záznamy atď. Typ súboru určuje jeho štruktúru. Textový súbor je postupnosť znakov usporiadaných v riadkoch, zdrojové súbory pozostávajú z procedúr a funkcií, objektové súbory sú postupnosťou bajtov, usporiadaných v blokoch, vykonateľné súbory sú postupnosťou kódových sekcií, ktoré zavádzací program môže zaviesť do pamäte a vykonať ich.

1.1.1 Atribúty súboru

Súbory majú mená, pretože pre ľudí je prirodzenejšie odkazovať sa na ne pomocou mien. Meno je reťazcom znakov ako napr. „program.c“. Niektoré systémy rozlišujú veľké a malé písmená v menách (Unix), pre iné malé a veľké písmená majú rovnaký význam (MS-DOS). Súbor má aj ďalšie atribúty, ktoré u rôznych operačných systémov sú rôzne, ale väčšinou ide o tieto:

- *Meno* - to je jediná informácia o súbore, ktorá sa uchováva v tvare zrozumiteľnom pre človeka.
    - *Typ* - informácia, potrebná pre systémy, ktoré podporujú rôzne typy súborov.
    - *Umiestnenie* - to je ukazovateľ na zariadenie a miesto súboru na tomto zariadení.
    - *Rozmer* - momentálna veľkosť súboru (v bajtoch, v slovách alebo v blokoch).
    - *Ochrana* - podľa tejto informácie sa kontroluje prístup k súboru, t.j. kto môže súbor čítať, modifikovať, vykonávať atď.
    - *Čas, dátum a identifikácia používateľa*. Táto informácia sa môže uchovávať vzhľadom na tvorbu súboru, na poslednú modifikáciu alebo na posledný prístup k súboru.

Informácie o všetkých súboroch sa uchovávajú v štruktúre adresára, ktorá sa tiež nachádza na disku.

1.1.2 Operácie nad súbormi

Súbor je abstraktný dátový typ. Nad týmto typom sú definované nasledujúce operácie:

- ***Tvorba súboru*** - pri vytvorení nového súboru je potrebné najskôr nájsť pre neho miesto v súborovom systéme. Ďalší krok je vytvorenie položky pre tento súbor v adresári. Tam sa zaznamenajú údaje o mene a umiestnení súboru.
  - ***Zápis do súboru*** - pri tejto operácii sa musí volať systémová služba s menom súboru a dátami, ktoré sa majú zapísať do súboru. Systém prehľadáva adresár, aby našiel údaje o umiestnení súboru. Systém udržiava ukazovateľ pre zápis, ktorý ukazuje, kde sa majú zapísať nasledujúce dáta. Tento ukazovateľ sa aktualizuje po každom zápise.
  - ***Čítanie zo súboru*** - pri čítaní zo súboru je potrebné použiť systémové volanie, ktoré špecifikuje meno súboru a adresu v pamäti, kde sa má uložiť ďalší načítaný blok zo súboru. Aj v tomto prípade systém udržiava ukazovateľ pre čítanie, ktorý ukazuje miesto v súbore, odkiaľ sa načíta ďalší blok. Tento ukazovateľ sa aktualizuje po každom zápise. Mnoho systémov spája ukazovateľ pre čítanie s ukazovateľom pre zápis, pretože súbor sa buď číta alebo zapisuje.
  - ***Zmena pozície v súbore (seek)*** - prehľadáva sa adresár pre príslušný súbor a aktuálna pozícia v súbore sa nastavuje na zadanú hodnotu.
  - ***Zrušenie súboru*** - pre túto operáciu sa musí vyhľadať príslušná položka v adresári, uvoľniť diskový priestor, ktorý súbor zaberal a potom položka súboru sa musí v adresári zrušiť.
  - ***Skrátenie súboru*** - táto operácia ponecháva všetky atribúty súboru okrem dĺžky nezmenené. Dĺžku nastavuje na 0.

Tieto operácie predstavujú minimálny počet požadovaných operácií nad súbormi. Ďalšie bežné operácie sú doplnenie nových dát na koniec súboru a premenovanie súboru. Tieto základné operácie sa môžu kombinovať, aby sa vytvorila iná zložitejšia operácia. Napr. operácia kopírovania súboru vyžaduje vytvorenie nového súboru, načítanie dát zo starého a zápis dát do nového súboru. Operačný systém poskytuje aj operácie, ktoré vracajú informáciu o stave súboru, a ďalšie, ktoré umožňujú zmenu atribútov súboru.

Väčšina operácií, ktoré sme spomenuli zahrňuje prehľadávanie adresára pre nájdenie položky s menom súboru. Aby sa predišlo tomuto sústavnému prehľadávaniu počas práce so súborom, operačný systém udržuje malú ***tabuľku otvorených súborov***, kde sa súbor zapíše pri prvej odvolávke naň. Pri nasledujúcich odkazoch na súbor sa požíva index do tejto tabuľky a prehľadávanie adresára nie je potrebné.

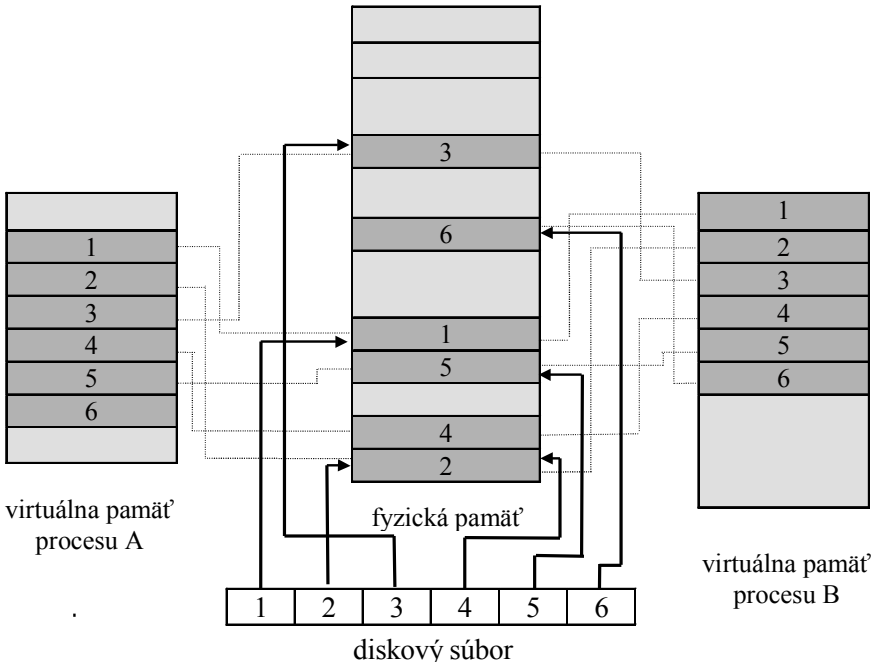
Niektoré operačné systémy automaticky otvárajú súbor, keď sa naň odkáže a po ukončení úlohy, ktorá s ním pracovala, ho automaticky zatvoria. Väčšina operačných systémov ale vyžaduje explicitné otvorenie súboru programátorom pomocou systémového volania *(open)*. Operácia *open* zvyčajne vráti ukazovateľ do tabuľky otvorených súborov, ktorý sa používa pri ďalších operáciách so súborom.

Implementácia operácií otvorenia *(open)* a uzatvorenia *(close)* vo viacužívateľských systémoch je zložitejšia ako v jednoužívateľských. V týchto systémoch viacej používateľov môže otvoriť naraz ten istý súbor. Obyčajne operačný systém používa dve úrovne interných tabuliek. Jedna je ***tabuľka otvorených súborov procesu***, kde sa zaznamenávajú všetky súbory, ktoré daný proces otvoril. Každá položka v tabuľke procesu ukazuje na ***systémovú tabuľku otvorených súborov.*** Táto tabuľka je nezávislá od procesu a ku každému otvorenému súboru uchováva tieto informácie:

- *Ukazovateľ súboru* - ukazuje na pozíciu, kde sa má vykonať ďalšia operácia čítania alebo zápisu. Tento ukazovateľ je jeden pre každý proces, ktorý pracuje so súborom.
  - *Počet procesov, ktoré používajú súbor* - toto číslo sa zväčšuje o 1 pri každom otvorení súboru a po každom uzatvorení sa zmenšuje o 1. Keď dosiahne 0, položka sa odstráni z tabuľky.
  - *Umiestnenie súboru na disku* - väčšina procesov potrebuje modifikovať dáta v súbore. Informácia o umiestnení súboru na disku sa uchováva v pamäti, aby sa nemusela pri každej operácii načítavať z disku.

• *Prístupové práva* – každý proces otvára súbor v niektorom z režimov prístupu. Táto informácia sa uchováva v tabuľke procesu a operačný systém môže povoliť alebo zakázať následne požadovanú V/V operáciu.

Ak je súbor už otvorený a otvorí ho ďalší proces, do tabuľky otvorených súborov procesu sa pridá nová položka, kde sa zaznamená ukazovateľ do súboru (pre čítanie a zápis) a ukazovateľ na príslušnú položku v systémovej tabuľke. Počet procesov, ktoré pracujú s týmto súborom, sa zvýši o 1. Pri uzatvorení súboru jedným procesom sa počet zmenší o 1. Keď počet klesne na 0, položka sa odstráni z tabuľky.



Obr. 11.1 Mapovanie súborov do pamäte

Niektoré viacužívateľské operačné systémy aby umožnili zdieľať súbory, poskytujú aj mechanizmus uzamknutia sekcie otvoreného súboru. Ďalšia možnosť je mapovanie sekcie súboru do pamäte v systémoch s virtuálnou pamäťou (Obr. 11.1). Táto funkcia sa nazýva mapovanie súboru do pamäte (memory mapping) a umožňuje, aby časť virtuálneho adresného priestoru bola logicky spojená so sekciou súboru. Čítanie a zápis tejto časti pamäte sú potom považované za operácie so súborom. Po uzatvorení súboru sa sekcia z pamäte zapíše späť na disk a pamäťový priestor sa uvoľní. Mapovanie sekcií súboru do virtuálnej pamäte uľahčuje aj zdieľanie súboru, pretože na tie stránky/segmenty sa uplatňujú tie isté pravidlá ako pri zdieľaní stránok.

1.1.3 Typy súborov

Pri návrhu systému súborov určitého operačného systému je veľmi dôležité rozhodnúť sa, či operačný systém bude podporovať rôzne typy súborov. Ak bude podporovať viacej typov, bude potrebné každý typ súboru spracovávať príslušným spôsobom. Napr. bežná chyba je pokus o výpis binárneho súboru na obrazovku. Výsledok je nečitateľný. Táto chyba sa môže odstrániť tak, že systém bude mať informáciu, že tento typ súboru sa nedá čítať a nedovolí čítanie.

Najčastejšie sa typy súborov implementujú pomocou rozšírenia mena súboru, umiestneného za bodkou (pozri Obr. 11.2). Napr. v MS-DOS označenie „program.exe“ sa používa pre vykonateľný súbor. Rozšírenie .exe upozorňuje aj operačný systém aj používateľa, o aký typ súboru sa jedná (). Operačný systém potrebuje rozšírenie na to, aby určil, aké operácie sú dovolené nad súborom. Napr. u MS-DOSu sa dajú vykonávať len súbory s rozšírením *.com, .exe* a *.bat*. Prvé dve rozšírenia označujú binárne vykonateľné súbory a tretie označuje dávkový súbor v ASCII formáte. MS-DOS rozlišuje len

niekoľko rozšírení, ale rozšírenia pre označenie typu súboru využívajú aj aplikačné programy. Napr. prekladač asembleru vyžaduje, aby súbory na jeho vstupe mali rozšírenie „*.asm*“. Operačný systém nepodporuje tieto typy, tie slúžia len aplikácii, aby rozpoznala svoje súbory.

Typ súboru	**Bežné rozšírenie**	**Funkcia**
vykonateľný	exe, com, bin alebo žiadne	program, pripravený na vykonanie
cieľový (object) tvar	obj, o	skompilovaný súbor v strojovom kóde, ešte nespojený
zdrojový kód	c, pas, p, f77, asm, a	zdrojový kód v niektorom programovacom jazyku
dávkový	bat, sh	príkazy pre príkazový interpreter
textový	txt, doc	text, dokumenty
textový procesor	wp, tex, rrf a iné	formáty rôznych textových procesorov
knižnice	lib, a	knižnice
pre tlač alebo prehľadanie	ps, dvi, gif	ASCII alebo binárne súbory pre tlač alebo prehľadanie
archívne	arc, zip, tar	skupina súborov spojených do jedného súboru za účelom archivácie

Obr. 11.2 Bežné typy súborov

Napr. operačný systém Aple MacIntosh podporuje dva typy: „text“ a „pict“. Každý súbor má v svojich atribútoch zaznamenané, ktorá aplikácia ho vytvorila. V atribútoch súboru, ktorý bol vytvorený textovým procesorom, je zapísané meno aplikácie, ktorá ho vytvorila. Keď používateľ klikne dvakrát na ikonu tohto súboru, súbor sa automaticky otvorí v textovom procesore, ktorý ho pôvodne vytvoril, a súbor je pripravený na editovanie.

Operačný systém Unix využíva tzv. *magické číslo*, ktoré určuje typ súboru: vykonateľný, dávkový (shell script), postscript atď. Nie všetky súbory majú magické číslo, takže ďalšie spracovanie sa nedá postaviť len na tejto informácii. Unix dovoľuje rôzne rozšírenia, ale tie neoznačujú typy podporované systémom, ale sú skôr určené pre orientáciu používateľa.

1.1.4 Štruktúra súboru

Typy súborov je možné použiť aj pre označenie vnútornej štruktúry súboru. V takomto prípade každému typu odpovedá určitá vnútorná štruktúra súboru. Operačný systém potom môže vykonávať typovú kontrolu operácií nad súborom. Pochopiteľne, čím viacej typov súborov podporuje operačný systém, tým väčšie bude jadro, lebo pre každý typ súboru bude musieť poskytnúť operácie, ktoré jeho vnútornú štruktúru poznajú.

Niektoré operačné systémy podporujú minimálny počet súborových štruktúr. Takýto prístup zvolili tvorcovia MS-DOSu, Unixu a iných OS. Napr. operačný systém Unix pokladá každý súbor za postupnosť bajtov a neinterpretuje ich. Táto schéma je maximálne flexibilná, ale neposkytuje žiadnu podporu. Každý aplikačný program musí obsahovať kód pre interpretáciu vstupného súboru do vlastných štruktúr. Samozrejme operačný systém musí podporovať aspoň štruktúru vykonateľných súborov, aby mohol zaviesť program do pamäte a spustiť ho.

1.1.5 Fyzická štruktúra súborov

Určenie presného umiestnenia dát vo vnútri súboru môže byť komplikované. Diskový systém má presne definovaný rozmer bloku, ktorý je určený veľkosťou sektoru. Všetky diskové operácie sa vykonávajú po blokoch a všetky bloky majú rovnakú veľkosť. Je nepravdepodobné, že rozmer logického záznamu požadovaných dát bude presne súhlasiť s rozmerom fyzického záznamu. Navyše rozmer logického záznamu môže byť aj variabilný. Obyčajne sa tento problém rieši *blokovaním (packing)* určitého počtu logických záznamov do fyzických blokov.

Napr. ako sme už povedali, operačný systém Unix pokladá súbory za postupnosť bajtov. Každý bajt je adresovateľný pomocou jeho posuvu od začiatku súboru. V tomto prípade logický záznam má dĺžku 1 bajt. Systém automaticky „balí“ a „rozbaľuje“ bajty do fyzických blokov podľa potrebnej veľkosti (napr.512 bajtov).

Rozmery logického záznamu a fyzického bloku, ako aj algoritmy „balenia“ určujú koľko logických záznamov sa zmestí do jedného bloku. „Balenie“ robí aplikačný program alebo operačný systém.

Súbor môžeme pokladať za postupnosť blokov. Všetky základné V/V operácie sa vykonávajú po blokoch.

Fyzický priestor na disku sa prideľuje vždy po blokoch. To znamená, že posledný blok súboru nebýva plný a tento diskový priestor je stratený, t.j. existuje ***vnútorná fragmentácia***. Všetky súborové systémy preukazujú vnútornú fragmentáciu a čím väčší je rozmer bloku, tým väčšia je vnútorná fragmentácia.

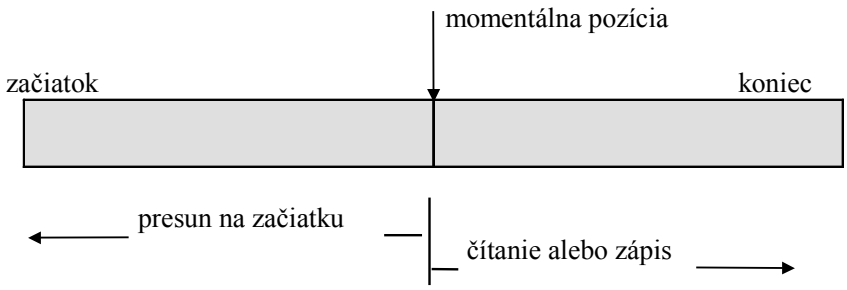
11.2 Metódy prístupu

Existuje niekoľko metód prístupu k informáciám v súboroch. Niektoré systémy ponúkajú len jednu metódu, iné ponúkajú viacej rôznych metód a potom je potrebné rozhodovať, ktorá metóda prístupu je najvhodnejšia pre určitú aplikáciu.

11.2.1 Sekvenčný prístup

Sekvenčný prístup je najjednoduchšou metódou prístupu k súborom. Informácia sa sprístupňuje v poradí, záznam po zázname. Táto metóda prístupu je najrozšírenejšia. Používajú ju napr. editory a kompilátory. Sekvenčný prístup je založený na modeli súboru na magnetickej páske.

Najčastejšie operácie nad súbormi sú čítanie a zápis. Po každej operácii čítania sa ukazovateľ súboru nastavuje na ďalšiu pozíciu. Po zápise na koniec súboru sa ukazovateľ nastavuje na nový koniec súboru, kde sa uskutoční ďalší zápis. Je možne nastaviť ukazovateľ aj na začiatok súboru. Niektoré systémy dovoľujú aj posun o *n* záznamov (*n* je celé číslo) dopredu alebo dozadu (Obr. 11.3).



Obr. 11.3 Sekvenčný prístup k súboru

11.2.2 Priamy prístup

Ďalšia metóda prístupu je *priamy* prístup. Súbor pozostáva z *logických záznamov s pevnou dĺžkou,* ktoré dovoľujú čítanie a zápis záznamov v ľubovoľnom poradí. Tento typ prístupu je založený na diskovom modeli súboru, pretože disk dovoľuje pristupovať k blokom súboru v ľubovoľnom poradí. Súbor je považovaný za postupnosť očíslovaných blokov alebo záznamov. Vďaka tomu môžeme za sebou čítať záznam č.12, potom čítať záznam č.53 atď. Obmedzenia pre poradie operácií čítania alebo zápisu neexistujú.

Priamy prístup sa najčastejšie používa pri potrebe rýchleho prístupu k väčším súborom. Veľmi často je používaný v databázach. Pri požiadavke sprístupnenia určitej informácie sa prepočíta, v ktorom bloku sa požadovaná informácia nachádza, a načíta sa priamo príslušný blok.

Operačný systém ponúka pre priamy prístup operácie *read / write n,* ktoré požadujú číslo bloku *n* ako parameter, alebo operácie *read next/write next* (čítanie/zápis nasledujúceho bloku), čo je v podstate obdoba sekvenčného prístupu. Čítanie/zápis nasledujúceho bloku môžeme dosiahnuť aj iným spôsobom, a to tak, že nastavíme ukazovateľ na určitú pozíciu (*position n*) a potom vykonáme operáciu čítanie.

Číslo bloku, s ktorým sa narába pri priamom prístupe, je *číslo relatívne k začiatku súboru*. Prvý blok v súbore má číslo 0 (niektoré systémy začínajú číslovať bloky od 1), druhý 1 atď., pričom absolútne diskové adresy týchto blokov nemusia ležať vedľa seba.

Ak máme logický blok s dĺžkou *L* a je požadovaný záznam číslo *N*, potom pozícia požadovaného záznamu je *L .(N-1)*. Operácie čítanie, zápis alebo zrušenie určitého záznamu sú jednoduché, pretože logické záznamy majú pevnú dĺžku.

Nie všetky operačné systémy podporujú aj sekvenčný aj priamy prístup. Niektoré systémy požadujú, aby súbor ešte pri vytvorení bol určený pre sekvenčný alebo priamy prístup. Neskôr súbor môže byť sprístupnený len príslušným spôsobom. Čitateľ si určite uvedomuje, že sekvenčný prístup sa dá ľahko simulovať pomocou priameho prístupu, ako je ukázané v nasledujúcom obrázku (Obr. 11.4):

Sekvenčný prístup	Implementácia priamym prístupom
reset	*cp:=0;*
read *next*	***read*** *cp;*
	cp:=cp+1;
write *next*	***write*** *cp;*
	cp:=cp+1;

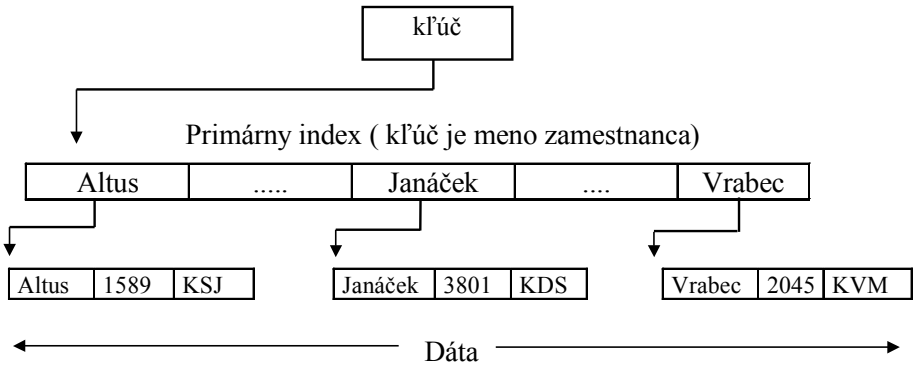
Obr. 11.4 Simulácia sekvenčného prístupu pomocou priameho prístupu

11.2.3 Iné metódy prístupu

Na základe priameho prístupu sa dajú vytvoriť aj iné metódy prístupu k súborom. Ide hlavne o vytvorenie tzv. *indexu k súboru*. Index obsahuje ukazovatele na rôzne bloky v súbore. Pre vyhľadanie určitej položky v súbore sa najskôr prehľadá index a potom sa zistený ukazovateľ použije pre priamy prístup k bloku, kde sa nachádza položka.

Ak súbor je veľký, potom aj index je príliš veľký na to, aby sa uchovával v pamäti. Riešenie je urobiť index indexu. Primárny index obsahuje ukazovatele na sekundárne indexové súbory, ktoré ukazujú na bloky v súbore.

Obr. 11.5 ukazuje organizáciu indexu, ako je implementovaný v systéme VMS. Keď používateľ vytvára indexový súbor, musí sa rozhodnúť, ktorá položka záznamu bude kľúčom. Kľúčov môže byť viacej, ale minimálne musí byť jeden - primárny kľúč. Keď proces zapisuje záznamy do indexového súboru, služba operačného systému vytvára index k tomuto súboru. Index obsahuje kľúč a ukazovateľ na miesto v súbore, kde sa nachádza záznam s týmto kľúčom. Ak používateľ zadefinuje alternatívny kľúč, služba operačného systému vytvára nový index pre tento kľúč.



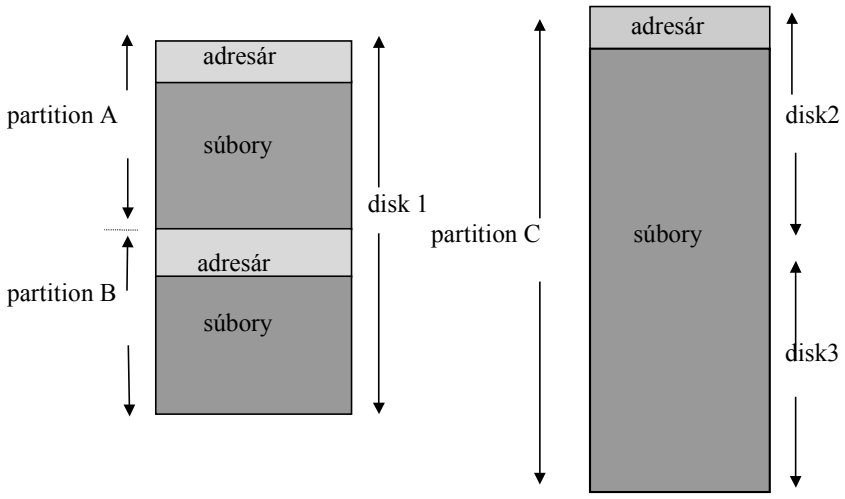
Obr. 11.5 Organizácia prístupu s indexom v systéme VMS

Index-sekvenčná metóda (ISAM), použitá firmou IBM používa malý *master index,* ktorý ukazuje na diskové bloky sekundárneho indexu. Sekundárny index ukazuje na bloky súboru. Súbor sa udržuje utriedený podľa kľúčov. Pre nájdenie určitej položky sa najskôr urobí binárne hľadanie v master indexe, kde sa nájde blok sekundárneho indexu. Tento blok sa načíta a sekundárny index sa opäť prehľadáva binárne pre nájdenie bloku, ktorý obsahuje požadovaný záznam. Nakoniec sa tento blok prehľadáva sekvenčne. Pri použití tejto metódy každý záznam je lokalizovaný svojim kľúčom, použitým pri dvoch čítaniach s priamym prístupom.

11.3 Štruktúra adresárov

Systém súborov počítačového systému musí byť rozšíriteľný. Objem dát stále narastá a je potrebné ich organizovať efektívne. Obyčajne sa súborový systém delí na menšie časti - *minidisky alebo partície* (alebo *partitions* podľa IBM). Každý disk má aspoň jeden minidisk (partition), čo je štruktúra nižšej úrovne, kde sa ukladajú súbory a adresáre. Niektoré systémy dovoľujú rozšíriť *partíciu* cez niekoľko diskov (Obr. 11.6). Takto používateľ má možnosť organizovať svoje dáta logicky, bez ohľadu na fyzické zariadenia na ktoré sú dáta uložené.

Každý minidisk (partition) obsahuje informácie o súboroch. Tieto informácie sú uložené v adresároch zariadenia (Volume Table of Contents). Adresár obsahuje informácie o všetkých súboroch tejto časti a to meno, umiestnenie, veľkosť a typ súboru. Treba si uvedomiť, že adresár sám o sebe môže byť implementovaný mnohými spôsobmi, ktoré budú rozobrané neskôr.



Obr. 11.6 Typická organizácia súborového systému

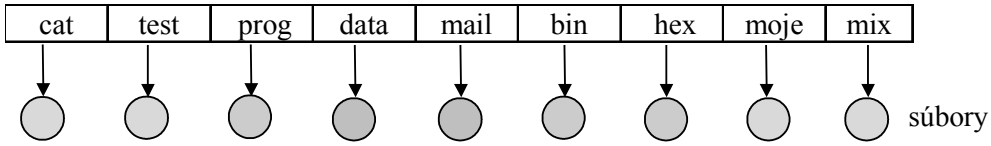
Štruktúra adresára musí byť vybraná s ohľadom na operácie, ktoré sa vykonávajú nad adresármi. Tieto operácie sú:

- **Hľadanie súboru** - adresár sa prehľadáva pre nájdenie položky určitého súboru. Pretože súbory majú symbolické mená, musí byť možné nájsť aj viac mien, obsahujúcich určitý podreťazec.
- **Tvorba súboru** - adresár musí umožňovať pridanie novej položky pri tvorbe nového súboru.
- **Zrušenie súboru** - keď súbor nie je viac potrebný, musíme mať možnosť odstrániť ho z adresára.
- **Výpis adresára** - možnosť vypísať obsah adresára a obsah každého súboru z adresára.
- **Premenovanie súboru** - mená súborov odzrkadľujú ich obsah alebo použitie, takže používateľ musí mať možnosť zmeniť meno a umiestnenie súboru, ak to potrebuje.
- **Prechod celého súborového systému** - je veľa prípadov, kedy je potrebné prejsť celý súborový systém za účelom napr. zálohovania na magnetickej páske.

V nasledujúcich častiach popíšeme najčastejšie sa vyskytujúce logické štruktúry adresárov.

11.3.1 Jednoúrovňový adresár

Jednoúrovňový adresár má najjednoduchšiu štruktúru. Všetky súbory sú v jednom adresári - viď Obr. 11.7.



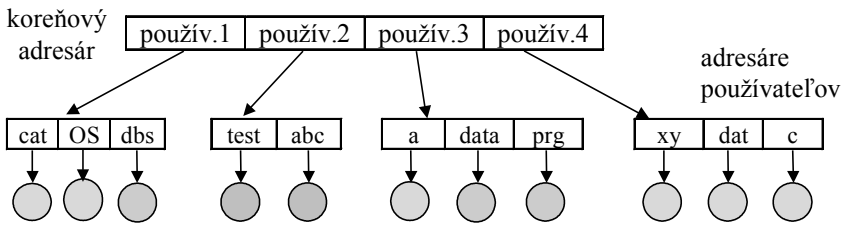
Obr. 11.7 Jednoúrovňový adresár

Štruktúra jednoúrovňového adresára je veľmi jednoduchá, ale je veľmi obmedzujúca v prípade väčšieho počtu súborov alebo viacerých používateľov. Pri väčšom počte súborov je ťažké stále vymýšľať nové mena pre súbory, pretože existujúce mená sa nemôžu opakovať. Obdobný problém nastáva aj v prípade, kedy súbory v tom adresári patria viacerým používateľom a mená sa nesmú opakovať. Potom dvaja užívatelia nemôžu pomenovať svoje súbory rovnako.

Táto štruktúra sa v dnešných počítačových systémoch nepoužíva.

11.3.2 Dvojúrovňový adresár

Štandardné riešenie v prípade viacerých používateľov je prideliť každému používateľovi vlastný adresár. Všetky používateľské adresáre majú rovnakú štruktúru. Keď sa používateľ prihlási, prehľadá sa systémový adresár pre položku s menom používateľa. Táto položka ukazuje na hľadaný adresár používateľa (Obr. 11.8). Keď sa používateľ odvolá na určitý súbor, prehľadáva sa len jeho adresár. Rôzni užívatelia môžu mať súbory s rovnakým menom, ale v rámci jedného používateľského adresára mená musia byť jedinečné.



Obr. 11.8 Dvojúrovňový adresár

Používateľ nemá k hlavnému adresáru prístup a tiež nemôže zdieľať súbory s inými používateľmi, pretože užívatelia nemajú prístup do iných adresároch. Pre umožnenie volania

systémových programov sa prehľadáva viacej adresárov. V adresári hypotetického používateľa „systém“ sa nachádzajú zdieľané systémové programy. Pokiaľ systém nenájde súbor, ktorý používateľ chcel, prehľadáva adresár používateľa „systém“ a ďalšie adresáre. Postupnosť adresárov, ktoré sa prehľadávajú, sa nazýva ***hľadacia cesta***.

Na dvojúrovňový adresár sa môžeme pozerať ako na strom s výškou 2. Koreňom stromu je hlavný adresár, jeho nasledovníkmi sú adresáre používateľov. Nasledovníkmi používateľských adresárov sú súbory. Súbory sú listami stromu.

Meno používateľa, resp. meno jeho adresára a meno súboru tvoria ***úplnú prístupovú cestu*** k danému súboru. Táto cesta jednoznačne lokalizuje súbor. Obyčajne sa cesta zapisuje ako postupnosť mien, oddelených lomítkami. Napr. cesta v OS VMS-e je označovaná nasledovne: *u:[sst.tests]test.c;1* kde *u:* je označenie pre príslušnú partition, *sst* je meno adresára, *test.c* je meno a rozšírenie súboru a za bodkočiarkou je uvedená verzia súboru - v tomto systéme používateľ môže určiť, koľko verzií svojich súborov chce uchovávať. Iné systémy neoddeľujú partition zvláštnym znakom, ako napr. v Unixe: *u/pck/test*. Tu *u* je označenie pre partition, *pck* je meno adresára a *test* je meno súboru.

11.3.3 Stromová štruktúra adresára

Obecnejšou štruktúrou akou je dvojúrovňový adresár je adresár so stromovou štruktúrou s ľubovoľnou výškou. Táto štruktúra dáva možnosť používateľom vytvárať svoje podadresáre a organizovať v nich svoje súbory. Napr. súborový systém MS-DOS-u je organizovaný ako strom. Strom má koreňový adresár. Úplná cesta popisuje cestu od koreňa k súboru.

Adresár alebo podadresár pozostáva z adresárov a súborov. Adresár je jednoducho ďalší súbor, ale interpretuje sa iným spôsobom. Všetky adresáre majú rovnakú vnútornú štruktúru. Jeden bit z každej položky adresára označuje, či položka je súbor alebo adresár.

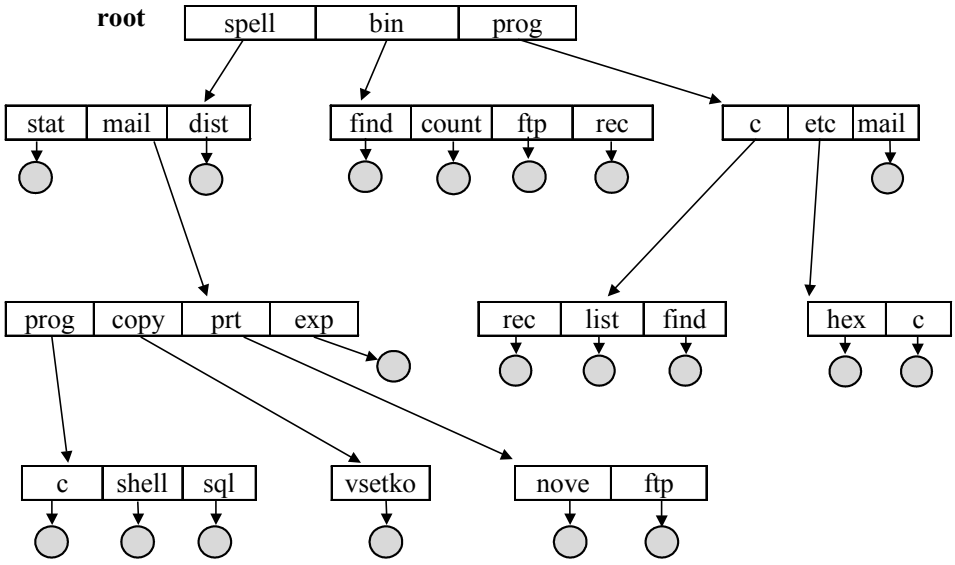
Pri odvolaní sa na súbor, sa prehľadáva najskôr , tzv. *pracovný adresár*, kde používateľ je nastavený. Ak sa tam hľadaný súbor nenájde, hľadá sa v adresároch v poradí, aké je uvedené v prehľadávacej ceste. Zmena adresára je možná systémovým volaním *change directory*.

Pri prihlásení sa používateľa, pracovný adresár sa nastavuje na adresár tohto používateľa alebo prípadne na iný preddefinovaný adresár.

Cesta v stromovej štruktúre sa môže zadávať dvoma spôsobmi: ako ***absolútna***, keď je popísaná od koreňového adresára, napr. *root/spell/mail/prt/ftp,* alebo ako ***relatívna vzhľadom na aktuálny adresár***, napr. ak aktuálny adresár je *spell*, potom relatívna cesta k súboru *ftp* bude *mail/prt/ftp* (Obr. 11.9). Stromová štruktúra dáva možnosť používateľovi usporiadať svoje súbory podľa svojich potrieb.

Zaujímavý je problém zrušenia adresára. Ak adresár je prázdny, jeho položka sa jednoducho vymaže. Ale ak adresár obsahuje súbory aj podadresáre, problém je komplikovanejší. Jeden z možných prístupov je ako u systému MS-DOS - nedovoliť zrušiť adresár, ktorý nie je prázdny. Tento prístup môže byť veľmi nepohodlný, ak adresár obsahuje celý podstrom podadresárov, ktoré bude potrebne jednotlivo spracovať - zmazať súbory, zrušiť podadresáre.

Iný prístup k tomuto problému ponúka operačný systém Unix. Po aplikovaní príkazu *rm* na určitý adresár, sa zruší ako uvedený adresár, tak aj všetky jeho podadresáre. Toto riešenie je pohodlnejšie pre prípad, keď chceme zrušiť celý podstrom, ale veľmi nebezpečné v prípade, že sa pomýlime.



Obr. 11.9 Stromová štruktúra adresára

Prístup k súborom v stromovej štruktúre je možný pomocou ciest. Niektoré systémy dovoľujú používateľom definovať svoje cesty, ktoré sa prehľadávajú po uvedení príkazu.

V stromovej štruktúre cesty môžu byť omnoho dlhšie ako u dvojúrovňovej štruktúry. Aby umožnil používateľom spúšťať programy bez toho, že si budú pamätať dlhé cesty, systém MacIntosh udržiava súbor nazvaný „*Desktop File*“, kde sú uložené mená a cesty všetkých vykonateľných súborov, ktoré sú prístupné. Ak sa použije nový disk alebo disketa, systém automaticky prehľadáva jeho štruktúru a aktualizuje svoj „*Desktop File“*. Tento mechanizmus podporuje vykonanie pomocou dvojitého kliknutia. Systém sa pozrie na hlavičku súboru, kde je zapísaná aplikácia, ktorá súbor vytvorila a po prehľadaní „*Desktop File“* ju spustí a ako vstup berie označený súbor.

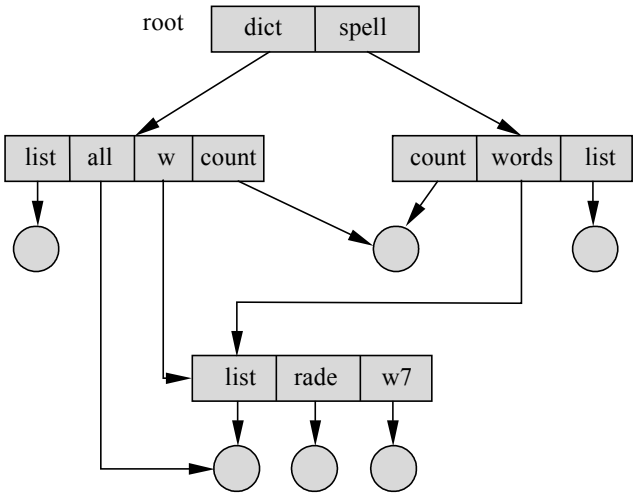
11.3.4 Adresár s acyklickou štruktúrou

Stromová štruktúra adresárov nedovoľuje zdieľať súbory. Zdieľanie znamená, že prístup k súboru budú mať viacerí užívatelia a zmeny, ktoré každý z nich urobí v súbore, budú hneď viditeľné pre ostatných. Zdieľanie je veľmi užitočné pri práci skupiny na spoločnom projekte.

Štruktúra, ktorá dovoľuje tento spôsob zdieľania súborov sa nazýva *acyklická štruktúra* (Obr. 11.10). Táto štruktúra dovoľuje adresárom mať zdieľané podadresáre alebo súbory. Acyklická štruktúra grafu (t.j. neobsahuje cykly) je prirodzeným zobecnením stromovej štruktúry. V prípade, že viacej ľudí pracuje na spoločnom projekte je možné všetky zdieľané súbory dať dohromady do jedného adresára, ktorý bude zdieľaný. Každý z členov týmu bude mať tento zdieľaný adresár ako podadresár svojho domovského adresára.

Zdieľané adresáre a súbory sa implementujú niekoľkými spôsobmi. Jedna z možností, využitá v operačnom systéme Unix je vytvorenie novej položky v adresári, nazvanej *link*. Táto položka je vlastne ukazovateľ na iný súbor alebo adresár, implementovaný ako absolútna alebo relatívna cesta. Pri odkaze na *súbor/adresár* sa prehľadávajú položky adresára. Ak zadaný *súbor/adresár* je *link*, potom sa použije cesta pre lokalizáciu skutočného *súboru/adresára*. *Link-y* sú ľahko identifikovateľné, pretože majú iný formát. Pri prechode celého stromu za účelom zálohovania operačný systém vynecháva linky.

Iný spôsob implementácie zdieľaných súborov je duplikovať všetky informácie o nich v oboch zdieľaných adresároch. Originál a kópia sú k nerozoznaniu. Hlavný problém v tomto prípade zostáva konzistencia súboru v prípade modifikácie jednej z kópií.



Obr. 11.10 Acyklická štruktúra adresára

Adresáre s acyklickou štruktúrou sú flexibilnejšie ako adresáre so stromovou štruktúrou, ale ich spracovanie je zložitejšie. Zoberme si napríklad prípad, keď je potrebné prejsť celý súborový systém. Na zdieľané súbory bude viac rôznych odkazov, ktoré je treba rozlíšiť, pretože chceme pri prechode súbormi prejsť každý z nich iba raz.

Ďalší problém vzniká pri zmazaní zdieľaného súboru. Kedy je možné jeho diskový priestor použiť znova? Jednou z možnosti je zrušenie súboru ihneď ako ho niektorí z používateľov zruší. Tu existuje nebezpečie, že ukazovatele, ktoré zostanú, neukazujú nikde. Alebo ešte horší prípad, kedy diskový priestor zmazaného súboru je použitý znova a starý ukazovateľ ukazuje na nový súbor.

Ak je zdieľanie implementované symbolickými linkami, zmazanie linky neovplyvní súbor. Iná je situácia, keď sa zmaže súbor a zostanú linky k nemu. Tie linky môžu zostať bez zmeny (lebo prehľadávanie celého súborového systému pre ich zrušenie je veľmi nákladné). V operačnom systéme Unix rozhodnutie čo s linkami na neexistujúci súbor je ponechané na používateľa (jednoduché, že!).

Iný prístup k zmazaniu zdieľaného súboru je jeho ponechanie, kým nebude zrušený aj posledný odkaz naň. Implementácia tohoto prístupu je možná pomocou udržiavania zoznamu všetkých odkazov na súbor (adresáre a symbolické linky). Keď sa vytvorí nový súbor alebo adresár, do zoznamu sa zapíše nová položka. Keď sa zmaže súbor alebo adresár, položka sa zruší. Súbor sa potom môže zmazať až keď jeho zoznam odkazov je prázdny. Problém je v tom, že tento zoznam je niekedy dosť veľký. Jedna možnosť redukcie jeho veľkosti je udržiavať len počty odkazov, to znamená, že súbor môže byť zmazaný keď počet odkazov klesne na 0. Tento prístup využíva OS Unix pre nesymbolické linky (*hard links*).

11.3.5 Pripojenie súborového systému

Pripojenie (*mounting*) určitého súborového systému je úkon, ktorý umožňuje prístup k tomuto systému v rámci menného priestoru daného systému. Tak ako súbor musí byť otvorený pred použitím, tak aj súborový systém sa musí najskôr pripojiť, aby bol prístupný procesom.

V OS Unix systém požaduje vedieť meno zariadenia a bod, kde sa pripojí nový súborový systém, tzv. bod pripojenia (*mount point*). Obyčajne bod pripojenia je prázdny adresár. Napr. pripojenie celého hierarchického súborového systému prebieha ešte v čase zavedenia systému v bode „/“ a súborový systém, ktorý obsahuje adresáre používateľov sa môže pripojiť v bode */home*. Pripojenie ďalších súborových systémov je umožnené špeciálnym príkazom a nevykonáva sa automaticky.

Po pripojení nového súborového systému sa skontroluje, či obsahuje očakávaný formát dát. Potom si operačný systém poznamená jeho prítomnosť do adresára, ktorý poskytol bod pripojenia.

V iných operačných systémoch pripojenie súborového systému prebieha bez spoločného bodu pripojenia alebo cez spoločný koreňový adresár.

OS MacIntosh v čase zavedenia systému skúma zariadenia a ak narazí na disk, ktorý ešte nebol zaevidovaný, preskúma ho pre súborový systém. Ak ho nájde, automaticky ho pripojí na úrovni koreňového adresára a ponechá jeho ikonu na ploche.

V operačných systémoch z rodiny Windows (95, 98, NT, 2000) tento problém je vyriešený pomocou rozšírenej dvojúrovňovej štruktúry adresárov – zariadeniam a partíciam sú priradené písmená. Partície majú štruktúru všeobecného grafu. Cesta k určitému súboru sa udáva v tvare *písmeno:\cesta\súbor*. V čase zavedenia tieto operačné systémy automaticky zisťujú všetky zariadenia a pripájajú ich súborové systémy.

11.4 Ochrana

V počítačových systémoch sa kladie veľký dôraz na ochranu informácií pred fyzickým poškodením a neoprávneným použitím.

Ochrana pred fyzickým poškodením sa obyčajne zaisťuje pomocou zálohovania súborov. Mnoho operačných systémov má služobné programy, ktoré v pravidelných intervaloch automaticky kopírujú diskové súbory na magnetickú pásku. V prípade fyzického poškodenia média, na ktorom sú súbory uložené, alebo HW chyby, je možné súbory obnoviť zo zálohy.

Potreba ochrany pred neoprávneným použitím je aktuálna vo viacužívateľských systémoch. V systémoch, ktoré nedovoľujú prístup k súborom iných používateľov, ochrana nie je potrebná. Tento extrém nie je prijateľný, ale ani opačný, kedy sa dovoľuje každému používateľovi prístup ku každému súboru. Všeobecne zaužívaný prístup je stredná cesta, t.j. riadený prístup.

1.1 Typy prístupu

Mechanizmus ochrany poskytuje riadený prístup pomocou definovania typov prístupov k súborom. Prístup k súboru je povolený alebo odmietnutý na základe niekoľkých faktorov, z ktorých jeden je aj typ požadovaného prístupu. Kontrolujú sa tieto požiadavky na operácie nad súbormi:

- Čítanie - čítanie zo súboru.
    - Zápis - zápis alebo modifikácia súborov.
- Vykonanie - zavedenie súboru do pamäte a spustenie.
    - *Doplnenie* - zápis novej informácie na koniec súboru.
    - Zmazanie - zmazanie súboru a uvoľnenie jeho diskového priestoru pre opätovné použitie.
    - Výpis mena a atribútov súboru.

Okrem týchto operácií je možné kontrolovať aj premenovanie, kopírovanie a editovanie súboru. Väčšina systémov implementuje tieto operácie na základe systémových volaní nižšej úrovne, takže ochrana je potom založená na ochrane základnej operácie. Napr. kopírovanie sa dá implementovať ako postupnosť požiadaviek na čítanie a zápis a používateľ, ktorý má právo na čítanie súboru, ho môže aj kopírovať.

Existuje mnoho ochranných mechanizmov. Každý z nich má svoje prednosti a nedostatky. Obecne sa dá povedať, že mechanizmus ochrany sa vyberá podľa určenia systému. Ak je systém malý a určený pre malý počet používateľov, požiadavky na ochranu nie sú tak vysoké ako u viacužívateľského systému s desiatkami rôznych používateľov.

1.2 Zoznam prístupov a skupiny

Najčastejšie sa problém prístupu k súborom rieši na základe identifikácie používateľa. Rozličnú užívatelia potrebujú odlišný prístup k súborom. Najbežnejšie riešenie je pridať k súboru *zoznam prístupov*, ktorý špecifikuje meno a typ prístupu, ktorý je povolený príslušnému používateľovi. Keď

určitý používateľ požaduje prístup k danému súboru, skontroluje sa, či tomuto používateľovi je požadovaný prístup povolený. Ak je povolený, operácia nad súborom sa vykoná, inak sa odmietne.

Napr. v systéme VMS ku každému súboru môže byť pripojený zoznam prístupov, kde sú uvedení užívatelia, ktorí môžu pristupovať k súboru. Tento zoznam vytvára a udržuje vlastník.

Hlavný problém pri použití zoznamov prístupov je ich dĺžka. Napr. ak dovolíme každému používateľovi prístup k danému súboru, zoznam prístupov bude obsahovať zoznam všetkých používateľov systému. Tento nedostatok sa napravuje *rozdelením používateľov na skupiny* a to takto:

- *Vlastník (owner)* - používateľ, ktorý vytvoril súbor.
    - *Skupina (group)* - skupina používateľov, ktorým je dovolené zdieľať súbor.
    - *Ostatní (universe)* - všetci ďalší užívatelia systému.

Takéto rozdelenie umožňuje napr. spolupracovníkom, ktorí patria do s*kupiny* zdieľať určitým spôsobom súbor - čítať, modifikovať a *ostatným*, ktorí nepotrebujú pracovať s týmto súborom umožňuje napr. len čítanie súboru.

Tento spôsob ochrany funguje dobre, keď sa dodržuje rozdelenie používateľov do skupín. Mnoho systémov rieši tento problém tak, že sa vytvárajú skupiny používateľov, ktoré riešia spoločnú úlohu alebo potrebujú zdieľať súbory z iných dôvodov. Tieto skupiny vytvára administrátor systému. Pri zaradení nového používateľa do systému sa mu pridelí jedna alebo viac skupín, kde bude patriť. Prístup k danému súboru sa potom povoľuje na základe príslušnosti žiadateľa ku skupine, ku ktorej patrí vlastník súboru.

Operačný systém Unix rieši problém ochrany na základe rozdelenia používateľov do spomínaných troch skupín, ale ešte odlišuje aj prístup, ktorý sa príslušným skupinám povoľuje: prístup na čítanie, prístup na zápis (modifikáciu) alebo prístup na vykonanie. Pre označenie typov prístupu sa pre každú skupinu používajú 3 polia a každé obsahuje 3 bity: **rwx rwx rwx** . Prvé pole označuje prístup povolený vlastníkovi, druhé pole označuje prístup povolený skupine a tretie - prístup povolený ostatným. Bit označený **r** znamená povolený prístup na čítanie, bit označený **w** znamená povolený prístup na zápis, a bit označený **x** znamená povolený prístup na vykonanie. Absencia príslušného písmena, označovaná pomlčkou (-), znamená nepovolený prístup pre príslušnú operáciu. Napr. zápis rw-rw-r-- pre súbor test.c znamená, že vlastník a príslušník skupiny ku ktorej patrí vlastník, môžu súbor čítať a modifikovať a ostatní môžu súbor iba čítať.

Okrem popísaných spôsobov ochrany sa v histórii vývoja operačných systémov objavili aj iné spôsoby, ale väčšinou nenašli široké uplatnenie a v súčasnosti sa nepoužívajú. Napr. jedným takýmto spôsobom je priraďovanie hesla ku každému súboru(!), alebo heslo ku každému adresáru (systém TOPS-20), alebo heslo(á) k minidisku (systém VM/CMS).

Ďalšie systémy napr. MS-DOS a Macintosh, ktoré boli zamýšľané ako jednoužívateľské neposkytujú skoro žiadnu ochranu.

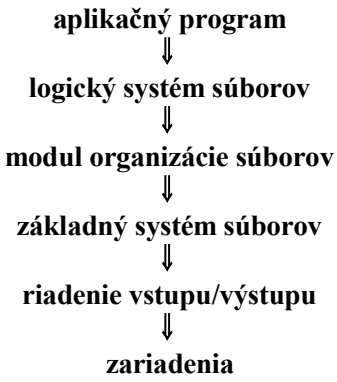
V systémoch so stromovou alebo inou štruktúrou adresárov je dôležitá ešte aj ochrana adresárov. Operácie nad adresármi sa musia ochraňovať trochu inak ako súbory. Je nutné kontrolovať tvorbu a rušenie súborov a navyše tu je potrebné kontrolovať aj prístup pre výpis obsahu adresára. Ochrana adresárov je závislá od štruktúry samotného súborového systému, pretože tá určuje, koľkými cestami sa dá k danému adresáru dostať.

11.5 Implementácia systému súborov

1.1 Celková organizácia systému súborov

Systém súborov poskytuje používateľovi možnosť pohodlne a efektívne spravovať svoje dáta. Pohľad používateľa na súborový systém je len jedna stránka súborového systému. Druhá stránka je spôsob implementácie tohoto systému. Implementácia zahrňuje algoritmy a dátové štruktúry, ktoré sa používajú pre mapovanie logického systému súborov do fyzických ukladacích zariadení.

Súborový systém má viacej vrstiev. Každá vrstva používa vlastnosti nižších vrstiev, aby vytvorila nové vlastnosti, ktoré ponúka vyšším vrstvám (Obr. 11.11).



Obr. 11.11 Vrstvy systému súborov

Najnižšia vrstva sú fyzické zariadenia. Nad ňou je vrstva, ktorá ***riadi V/V operácie.*** Tam patria ovládače zariadení a programy pre obsluhu prerušení. Ovládač zariadenia je program, ktorý „prekladá“ požiadavky systému do príkazov príslušného zariadenia.

Vrstva ***základného systému súborov*** odovzdáva generický príkaz príslušnému ovládaču zariadenia pre čítanie alebo zápis fyzického bloku na disk. Každý fyzický blok na disku je identifikovaný numerickou adresou, napr. mechanika 1, cylinder 73, plocha 2, sektor 10.

Modul organizácie súborov pozná súbory a ich logické a fyzické bloky. Vediac aký spôsob prideľovania diskového priestoru bol použitý a kde je súbor umiestnený na disku, tento modul prekladá logickú adresu bloku do fyzickej adresy a poskytuje ju nižšej vrstve. Logické bloky súboru sú očíslované od 0 (príp. od 1) po *N* a obyčajne číslo fyzického bloku, kde sú dáta, nie je také isté. Modul organizácie súborov zahrňuje aj správcu voľného priestoru na disku.

Logický systém súborov poskytuje používateľovi pohľad na systém súborov a využíva štruktúru adresárov, aby poskytol modulu organizácie súborov informácie o súboroch. Pre tieto účely využíva metadáta. Metadáta zahŕňajú všetky informácie o štruktúre súboru, ktoré sú zaznamenané v **riadiacom bloku súboru** (File Control Block - **FCB**). Logický systém súborov je zodpovedný aj za bezpečnosť súborov.

Keď aplikačný program chce vytvoriť súbor, volá vrstvu logického systému súborov. Logický systém súborov pozná štruktúru adresárov. Pri vytváraní nového súboru načíta príslušný adresár do pamäte, pridá novu položku a zapíše ho späť na disk. Potom logický systém súborov môže zavolať modul organizácie súborov, ktorý preloží logické adresy do fyzických a odovzdá ich nižším vrstvám. Keď sa zaktualizuje adresár, logický systém súborov ho použije, aby vykonal V/V operáciu. Pri otvorení súboru sa prehľadá adresár, aby sa našla položka súboru. Pre zefektívnenie prístupu k súboru sa používa tabuľka otvorených súborov (Obr. 11.12). Po prvom odkaze na súbor sa aplikačnému programu vráti index z tabuľky otvorených súborov, ktorý sa používa pri nasledujúcich operáciách nad súborom (v Unix-e sa nazýva *file descriptor*). Všetky zmeny položky adresára sa zapisujú do tabuľky otvorených súborov, ktorá je v pamäti. Až keď sa súbor uzatvorí, informácie o zmenách v ňom sa zapíšu na disk.

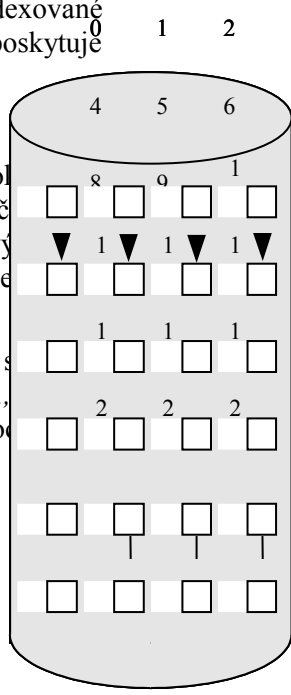
index	meno súboru	prístupové práva	ukazovateľ na blok disku
0	TEST.C	rw-rw-r--	...
1	DOPIS.TXT	rw-------	...
2	.		
.	.		
.	.		
n	.		

Obr. 11.12 Tabuľka otvorených súborov

V súčasnosti existuje mnoho implementovaných súborových systémov. Väčšina operačných systémov podporuje viacero súborových systémov. Napr. CD-ROM disky sú vo formáte High Sierra, ktorý je štandardným formátom odsúhlaseným medzi výrobcami. Windows NT napr. podporuje niekoľko diskových súborových systémov: FAT, FAT32 a NTFS, ako aj súborových systémov floppy diskov, formát CD-ROM a DVD.

1.2 Metódy prideľovania diskového priestoru

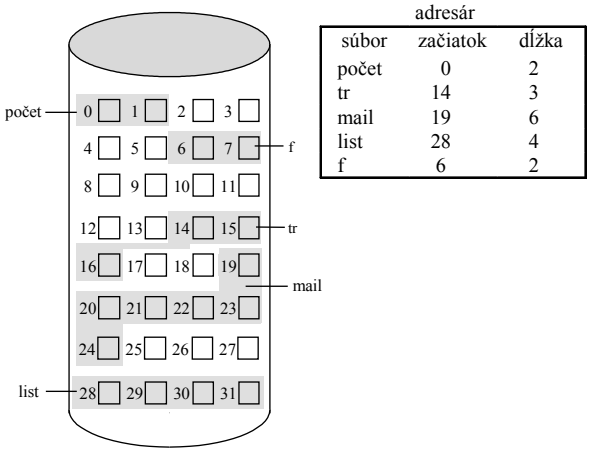
Disk je zariadenie pre ktoré priamy prístup ja najprirodzenejší a umožňuje veľkú flexibilitu pri implementácii systému súborov. Hlavný problém pri tejto implementácii je efektívne prideľovať diskový priestor a umožniť rýchle vyhľadávanie blokov súboru. Najrozšírenejšie sú tri metódy prideľovania diskového priestoru: súvislé prideľovanie, zreťazené prideľovanie a indexované prideľovanie. Každá metóda má svoje prednosti a nedostatky. Zvyčajne operačný systém poskytuje len jednu metódu pre všetky súbory.



1.1 Súvislé prideľovanie

Pri súvislom prideľovaní diskového priestoru sa súbor ukladá postupne na susedných blok disku. Diskové adresy nasledujú za sebou, čo znamená, že nie je potrebný pohyb hláv pre nač ďalšieho bloku. Keď je potrebný pohyb (z posledného sektoru jedného cylindra na prvý ďalšieho cylindra), ten pohyb je minimálny - len jedna stopa. To znamená, že čas pre hľadanie súvisle uloženého súboru je minimálny.

Súvislé uloženie súboru je definované diskovou adresou prvého bloku a dĺžkou. Ak s dlhý *n* blokov a začína na adrese *b*, potom celý súbor bude zaberať bloky *b, b + 1, b+2,....,* Položka súboru v adresári obsahuje adresu prvého bloku a dĺžku prideleného priestoru v blokoc 11.13).



Obr. 11.13 Súvislé pridelenie diskového priestoru

Prístup k súboru, ktorému bol pridelený súvislý priestor je veľmi jednoduchý. Pri sekvenčnom prístupe si systém pamätá adresu bloku, ktorý bol naposledy sprístupnený a keď je potrebné, prečíta sa ďalší blok. Pri priamom prístupe k bloku *i* súboru, ktorý začína od bloku *b*, je potrebné nastavenie na adresu *b+i.*

Problémom pri súvislom pridelení je nájdenie voľného priestoru pre uloženie nového súboru. Problém súvislého prideľovania diskového priestoru je podobný prideľovaniu súvislého pamäťového priestoru. Aj tu sa pri hľadaní vhodného úseku využívajú najčastejšie algoritmy *first-fit* a *best-fit*. Ani jeden z týchto algoritmov nie je lepší z hľadiska času a využitia diskového priestoru, ale *first-fit* je obecne rýchlejší.

Nedostatkom týchto algoritmov je vonkajšia fragmentácia. Ako sa súbory vytvárajú a rušia, diskový priestor sa postupne rozdrobí na malé kúsky. Vonkajšia fragmentácia začína byť problémom, keď aj najväčší súvislý úsek nestačí pre požadované uloženie súboru.

Niektoré staršie systémy používali súvislé prideľovanie pre súborový systém na disketách. Aby predišli stratám kvôli vonkajšej fragmentácií pri kopírovaní súboru na disketu používateľ musel spustiť špeciálny program, ktorý vykonával kompresiu. Tento program prekopíroval na iné zariadenie - disk alebo magnetická páska - celý súborový systém a potom ho prehral späť tak, že voľný priestor bol súvislý. Samozrejme, táto operácia vyžadovala dlhší čas.

Pri súvislom prideľovaní priestoru na disku hlavný problém je v tom, ako určiť koľko miesta bude súbor potrebovať, aby mu to bolo pridelené v čase jeho tvorby. Ak sa používa algoritmus best-fit, súbor sa nebude môcť rozširovať v tomto priestore. Pre zväčšenie súboru existujú dve možnosti. Pri prvej, ak nie je miesto pre rozšírenie súboru, vygeneruje sa chyba a upozorní sa používateľ, že musí spustiť program znova a deklarovať pre súbor viac priestoru. To samozrejme môže viesť k úmyselnému preceňovaniu veľkosti súboru zo strany používateľa, aby sa vyhol nepríjemnostiam pri potrebe zväčšenia súboru. Druhá možnosť je pri potrebe zväčšenia najať nový súvislý úsek na disku, prekopírovať súbor tam a uvoľniť predchádzajúci úsek. Táto operácia sa vykonáva bez účasti používateľa, ale môže tiež zaberať veľa času ak sa zopakuje niekoľkokrát a súbor je väčší.

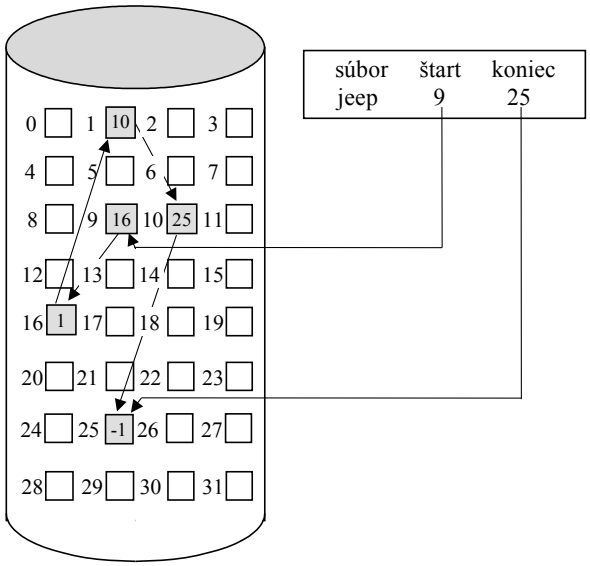
Problém pri rozširovaní súborov v prípade súvislého prideľovania sa rieši tak, že na začiatku sa súboru pridelí súvislý priestor a pri potrebe rozšírenia sa pridelí dodatočný súvislý úsek, ktorý nemusí nutne ležať za prvým. Tento úsek sa nazýva *rozšírenie (extent).* Bloky súboru sa potom zapisujú do položky adresára ako počiatočná adresa a počet blokov plus adresa prvého bloku rozšírenia. Samozrejme problém vonkajšej fragmentácie zostáva, pričom podľa toho, akým spôsobom je vyriešené zadávanie veľkosti rozšírenia, môže byť aj vnútorná fragmentácia. Tá vzniká, keď používateľ zadá väčšie rozmery, ako potrebuje.

1.2 Zreťazené prideľovanie

Zreťazené prideľovanie rieši problémy súvislého prideľovania. Pri tomto spôsobe prideľovania, je každý súbor zreťazeným zoznamom blokov na disku. Položka súboru v adresári obsahuje ukazovateľ na prvý a posledný blok zoznamu (Obr. 11.14). Každý blok obsahuje ukazovateľ na nasledujúci blok. To znamená, že z každého bloku sa určitá časť spotrebuje na uloženie ukazovateľa na ďalší blok.

Tvorba nového súboru je jednoduchá - vytvorí sa položka v adresári a v nej ukazovateľ na prvý blok súboru. Na začiatku je ukazovateľ inicializovaný na hodnotu *nil* a dĺžka súboru je nastavená na 0. Pri zápise nového bloku je potrebné najskôr nájsť voľný blok na disku a zapísať jeho adresu do ukazovateľa predchádzajúceho. Pri čítaní sa jednoducho čítajú bloky podľa ukazovateľov na konci každého bloku.

Vonkajšia fragmentácia tu neexistuje a každý voľný blok na disku sa môže použiť pre uloženie súboru. Tu nie je potrebné deklarovať dopredu veľkosť súboru, pretože rozširovanie je veľmi jednoduché.



Obr. 11.14 Zreťazené prideľovanie diskového priestoru

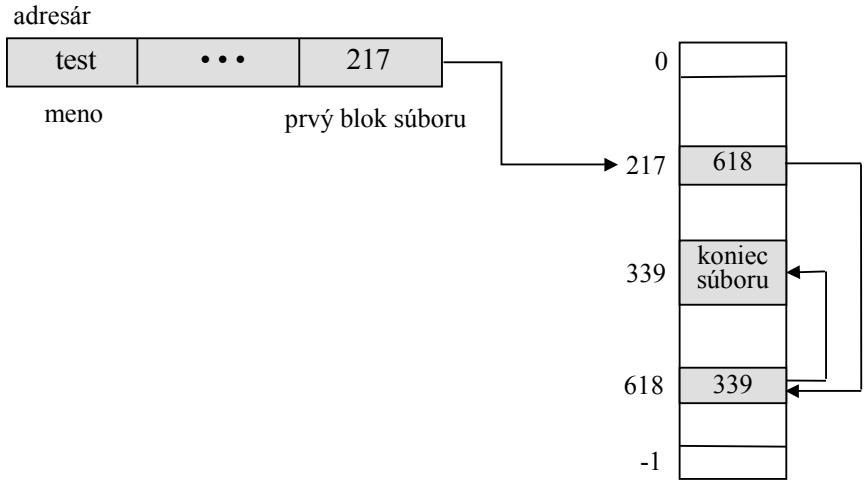
Problémom zreťazeného prideľovania je, že sa dá efektívne využiť len pri sekvenčnom prístupe k súboru. Ak potrebujeme sprístupniť *i-ty* blok, musíme začať vždy od začiatku súboru a blok po bloku sa dostať k *i-temu*. Každý prístup k ukazovateľu vyžaduje čítanie a väčšinou aj hľadanie adresy. Z toho vyplýva, že tento spôsob prideľovania priestoru na disku sa nehodí pre priamy prístup k súboru.

Ďalším nedostatkom je priestor, ktorý zaberá ukladanie ukazovateľov. Ak veľkosť bloku je 512 bajtov a ukazovateľ zaberá 4 bajty, potom 0.78% z diskového priestoru je použitých pre ukazovatele, t.j. stratených.

Problém plytvania miesta pre ukazovatele sa bežne rieši tak, že diskový priestor sa neprideľuje po blokoch, ale po väčších častiach, ktoré obsahujú viac blokov, tzv. *clustre*. Systém si zadefinuje, že napr. cluster bude 4 bloky a ďalej bude narábať s clustrami a nie s blokmi. Výhody z použitia clustrov sú nasledujúce: zmenší sa priestor potrebný pre uloženie ukazovateľov na clustre, zmenší sa pohyb ramienka potrebný pri operáciách so súborom, zmenší sa priestor potrebný pre uloženie zoznamu voľného priestoru. Cena za tieto zlepšenia je nárast vnútornej fragmentácie, pretože cluster je väčší ako blok a predpokladá sa že v priemere polovica posledného clustra/bloku zostáva nevyužitá.

Iným problémom pri prideľovaní diskového priestoru zreťazením blokov je spoľahlivosť. Môže sa stať, že bude poškodený alebo stratený ukazovateľ na ďalší blok, takže do súboru sa dostane omylom cudzí blok. Tento problém sa rieši tak, že buď sa používa dvojitý zreťazený zoznam pre ukazovatele, alebo v každom bloku sa ukladá aj informácia o mene súboru, ktorému blok patrí, a relatívne číslo bloku od začiatku. Samozrejme, že réžia pre uloženie týchto informácií stúpa.

Zaujímavou a efektívnou variáciou prideľovania diskového priestoru zreťazením blokov je použitie *tabuľky pridelenia súboru (File Allocation Table - FAT)*, ktorá bola použitá v systémoch OS/2 a MS-DOS. Jedna sekcia z disku na začiatku každej partition je oddelená pre uloženie FAT tabuľky (Obr. 11.15).



Obr. 11.15 Tabuľka prideľovania FAT

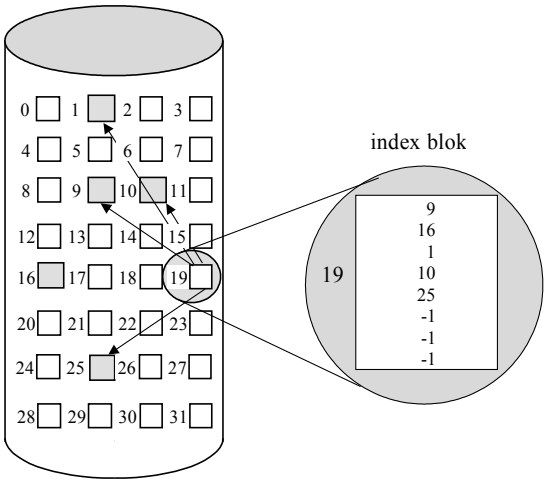
Tabuľka má jednu položku pre každý súbor a je indexovaná podľa čísel blokov. Tabuľka FAT sa používa ako zreťazený zoznam. Položka súboru v adresári obsahuje číslo jeho prvého bloku. Položka v tabuľke s týmto indexom obsahuje číslo ďalšieho bloku súboru atď. Tento reťazec pokračuje kým sa nepríde na koniec súboru, ktorý je označený špeciálnou hodnotou. Pridelenie nového bloku súboru spočíva v nájdení prvej položky s hodnotou 0, ktorá sa nahradí hodnotou konca súboru a tam, kde predtým bol koniec súboru, sa zapíše adresa nového bloku.

Použitie FAT tabuľky vedie k nárastu pohybu ramienka disku, pokiaľ FAT tabuľka nie je v cache pamäti. Najskôr sa ramienko musí nastaviť v tabuľke na index požadovaného bloku a po prečítaní sa musí ešte nastaviť na ten blok. Výhoda je, že priamy prístup je optimalizovaný, pretože adresa každého bloku sa dá prečítať v tabuľke FAT.

1.3 Indexované prideľovanie

Indexované prideľovanie rieši problém vonkajšej fragmentácie a problém s režijným priestorom pre ukazovatele. Zreťazené prideľovanie nedovoľuje efektívny priamy prístup, pretože ukazovatele sú roztrúsené po celom disku a bloky sa môžu sprístupňovať len sekvenčne. Indexované prideľovanie rieši tieto problémy tak, že sústreďuje všetky ukazovatele do jedného bloku do tzv. *index bloku*.

Každý súbor má svoj index blok, kde sú uložené adresy blokov súboru. *i-ta* položka v index bloku ukazuje na *i-ty* blok súboru. Adresár obsahuje adresu index bloku (Obr. 11.16). Keď chceme prečítať *i-ty* blok, použijeme *i-tu* položku z index bloku, aby sme zistili adresu požadovaného bloku.



Obr. 11.16 Indexované prideľovanie diskového priestoru

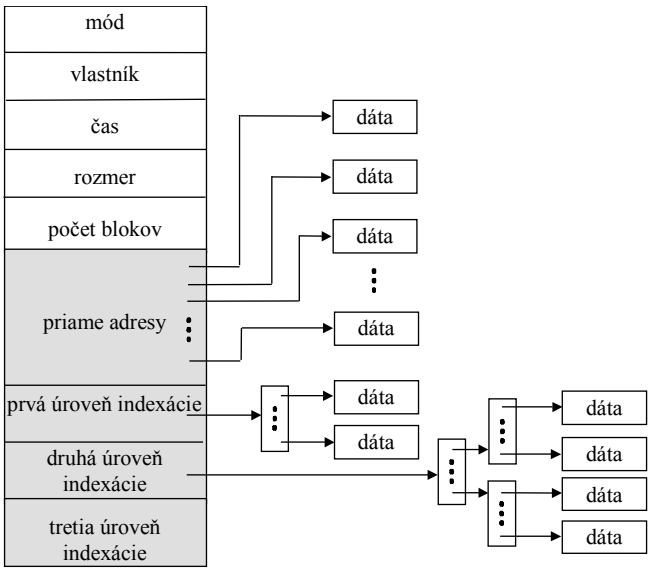
Na začiatku, keď sa vytvára súbor, všetky ukazovatele v index bloku sa nastavujú na hodnotu *nil.* Keď sa zapisuje *i-ty* blok, najskôr sa vyžiada adresa voľného bloku od správcu voľného priestoru a získaná adresa sa zapíše do *i-tej* položky index bloku.

Indexované prideľovanie podporuje priamy prístup a netrpí vonkajšou fragmentáciou.

Nedostatkom indexovaného prideľovania je plytvanie diskového priestoru pre uloženie index bloku, pričom to plytvanie je väčšie ako u zreťazeného prideľovania. Napr. v obecnom prípade, keď súbor má 1 alebo 2 bloky, v prípade zreťazeného prideľovania strata bude len pre 1 alebo 2 ukazovatele. Pri indexovanom prideľovaní sa pre index blok pridelí celý blok, ale z neho sa použijú len 1 alebo 2 ukazovatele a zvyšok bude mať hodnotu *nil*.

Tu je na mieste otázka, aký veľký má byť index blok. Mal by byť čo najmenší, aby sa neplytvalo priestorom, ale pritom by nemal byť moc malý, aby sa do neho mohli zaznamenať adresy blokov aj veľkých súborov. Obecne sa pre vyriešenie týchto požiadaviek používajú tieto mechanizmy:

- ***Reťazovo prepojené index bloky*** - index blok je jeden diskový blok. Aby bola možnosť ukladať veľké súbory, index bloky sa dajú zreťaziť. Napr. index blok môže obsahovať na začiatku informáciu o mene súboru a potom adresy blokov súboru. Posledná položka bude v prípade malého súboru *nil*, a v prípade väčšieho súboru -odkaz na ďalší index blok.
  - ***Viacúrovňový index*** - variantom predchádzajúceho riešenia je použiť index blok, ktorý obsahuje odkazy na ďalšie index bloky, ktoré už obsahujú adresy blokov súboru. Takže prístup k bloku súboru vyzerá tak, že sa najprv použije index prvej úrovne, aby sa dosiahol index druhej úrovne a potom požadovaný blok. Výhodou tejto schémy je, že úrovne indexov sa dajú zväčšovať a tak sa dá adresovať veľké množstvo blokov. Napr. so 4 096 bajtovým blokom (za použitia clustrov) môžeme mať 1 024 ukazovateľov s veľkosťou 4 bajty. Dve úrovne indexácie dovolia adresovať 1 048 576 blokov, čo znamená, že súbor môže mať až 4 GB. Táto veľkosť presahuje kapacitu väčšiny súčasných diskov.
  - ***Kombinovaná schéma*** - iná alternatíva bola použitá v systéme BSD Unix. Používa sa 15 ukazovateľov, ktoré sú uchované v index bloku súboru (*inode*). Prvých 12 ukazovateľov sú priamo diskové adresy blokov súboru. Ďalšie ukazovatele ukazujú na indexy, ktoré sú prvej, druhej a tretej úrovne indexovania (Obr. 11.17). Táto schéma prideľovania dovoľuje pri 32 bitovom ukazovateli adresovať 2^{32} bajtov, alebo 4 GB. Okrem tejto prednosti, táto schéma je veľmi pružná, pretože je štatisticky dokázané, že najviac je malých súborov a práve ich bloky sa tu adresujú priamo, t.j. operácie so súborom sú veľmi rýchle.



Obr.11.17 Štruktúra *inode* v Unixe

1.4 Výkon metód prideľovania diskového priestoru

Kritériá, ktoré sa používajú pre hodnotenie jednotlivých metód prideľovania diskového priestoru, sú efektívnosť ukladania dát a čas prístupu k dátovým blokom.

Súvislé prideľovanie vyžaduje len jeden prístup pre získanie adresy bloku. Pretože je jednoduché udržiavať začiatočné adresy v pamäti, dá sa veľmi rýchlo vypočítať adresa *i-teho* bloku a prečítať priamo tento blok.

Pri zreťazenom prideľovaní môžeme tiež uchovávať adresy nasledujúcich blokov v pamäti a pristupovať k nim priamo. Toto je dobré riešenie pre sekvenčný prístup k súboru, ale nevhodné pre priamy, pretože prístup k *i-temu* bloku bude vyžadovať *i* operácií čítania.

Kvôli uvedeným vlastnostiam týchto metód niektoré systémy podporujú súbory s priamym prístupom, ktoré sú uložené súvisle a súbory so sekvenčným prístupom, ktoré sú uložené zreťazením blokov. Tieto systémy preto vyžadujú pri tvorbe súboru uviesť metódu prístupu, ktorá sa bude neskôr používať. V tomto prípade operačný systém musí poskytnúť aj program konverzie z jedného typu na druhý.

Indexované prideľovanie je zložitejšie. Ak je index blok v pamäti, prístup môže byť priamy. Ale uchovanie index bloku v pamäti vyžaduje nezanedbateľný pamäťový priestor. Ak tento priestor nie je dostupný, potom každá operácia vyžaduje dva prístupy - k index bloku a k dátovému bloku. Pri dvojúrovňovom indexovaní je potrebný ešte jeden prístup navyše. Z toho vyplýva, že výkon indexovaného prideľovania je závislý od štruktúry indexov.

1.3 Správa voľného diskového priestoru

Správa voľného priestoru na disku je potrebná, pretože je potrebné trvalo udržiavať informáciu o zrušených súboroch, aby bol ich priestor znova využitý. Pre tento účel systém udržiava *zoznam voľného priestoru.* Tento zoznam obsahuje adresy všetkých voľných blokov na disku. Zoznam voľných blokov môže byť implementovaný pomocou bitového vektoru alebo zreťazeného zoznamu a jeho modifikáciami. Tieto možnosti sú uvedené ďalej.

1.1 Bitový vektor

Každý voľný blok v bitovom vektore je prezentovaný jedným bitom. Ak blok je voľný, bit má hodnotu 1, ak je pridelený, má hodnotu 0. Napr. predpokladajme, že máme disk na ktorom bloky 1, 7, 8, 9, 10, 12, 17, 22, 24, 28 a 30 sú voľné a ostatné bloky sú obsadené. Bitový vektor pre takýto disk bude nasledovný:

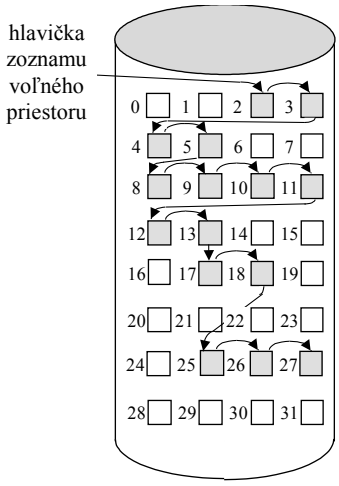
1000001111010000100001010001010000000000000 ....

Hlavnou výhodou tejto metódy spravovania voľného priestoru je jej jednoduchosť a efektívnosť pri hľadaní voľných úsekov. Veľa architektúr poskytuje inštrukcie pre bitovú manipuláciu, ktoré sa efektívne dajú použiť aj na tento účel.

Uchovanie informácií o voľných blokov je neefektívne, ak bitový vektor nie je uchovávaný v operačnej pamäti a práve to je nedostatkom tejto metódy, t.j. veľkosť požadovaného pamäťového priestoru pre uloženie celého bitového vektoru. Pre disky s malou kapacitou je možné použiť bitový vektor, ale je to neúnosné pre disk s kapacitou napr. 1.2 GB s 512 bajtovým blokom, kde bitový vektor bude potrebovať 310 KB, a v prípade clustrovania blokov po 4 bude požadovaný pamäťový priestor 78 KB.

1.2 Zreťazený zoznam

Iný prístup k správe voľného priestoru je uchovávať len adresy voľných blokov ako zreťazený zoznam a na určitej adrese na disku udržiavať ukazovateľ na prvý blok zoznamu. Prvý blok ukazuje na druhý atď. (Obr. 11.18). Prehľadávanie celého zoznamu je časovo náročné, ale našťastie to nie je často požadované. Obyčajne je požadovaný jednoducho voľný blok a poskytuje sa hneď prvý blok zoznamu.



Obr.11.18 Zoznam voľného priestoru na disku

1.3 Zoskupenie

Modifikáciou predchádzajúcej metódy je ukladať adresy *n* voľných blokov do prvého bloku skupiny. Prvých *n-1* blokov je skutočne voľných. Posledný blok z *n-tice* obsahuje adresy ďalších *n* voľných blokov. Výhodou tejto implementácie je, že adresy veľkého počtu voľných blokov sa dajú nájsť rýchlo, namiesto postupného prehľadávania.

1.4 Zoznam, obsahujúci počet voľných blokov

Ďalší prístup k správe voľného priestoru je uchovávať zoznam, obsahujúci začiatok súvislého priestoru a počet voľných blokov v ňom. Takýto prístup je výhodný keď sa používa súvislé prideľovanie alebo clustrovanie. Celková dĺžka zoznamu je menšia ako je u zreťazeného zoznamu.

1.4 Implemenácia adresára

Najrozšírenejšie spôsoby implementácie adresára sú lineárny zoznam a rozptylová (hešovacia) tabuľka.

1.1 Lineárny zoznam

Najjednoduchší spôsob implementácie adresára je lineárny zoznam mien súborov s ukazovateľmi na dátové bloky. Lineárny zoznam položiek adresára vyžaduje lineárne hľadanie požadovanej položky. Naprogramovanie je veľmi jednoduché, ale čas potrebný pre spracovanie požiadavky je veľký. Pri tvorbe nového súboru sa zoznam musí prehľadať, aby bola istota, že už neexistuje súbor s takým menom a potom sa nová položka pridá na koniec zoznamu. Pri zmazaní súboru sa zoznam prehľadáva pre nájdenie mena súboru a potom sa uvoľní pridelený priestor. Aby sa mohla položka v adresári znova použiť, musí sa označiť ako voľná.

Nevýhodou tejto implementácie je lineárne hľadanie pre nájdenie požadovaného súboru. Informácie z adresára sú veľmi často potrebné a pomalý prístup k nemu je viditeľný. Prakticky veľa operačných systémov uchováva v cache pamäti naposledy používané informácie z adresára a tak sa vyhýbajú častému čítaniu informácií z disku. Ak zoznam je utriedený, čas prehľadávania sa skráti. Samozrejme, ak sa zoznam má udržiavať v utriedenom stave, operácie vytvorenia a zmazania súboru budú komplikovanejšie.

1.2 Hešovacia tabuľka

Iná dátová štruktúra, ktorá sa používa pre implementáciu adresára, je hešovacia tabuľka (hash table). Hešovacia tabuľka berie hodnotu, ktorá je vypočítaná z mena súboru a vracia ukazovateľ na meno súboru v lineárnom zozname. Tento mechanizmus značne skracuje čas na prehľadávanie lineárneho zoznamu. Tvorba a zmazanie súboru sú rýchle, aj keď sa musia ošetriť kolízie t.j. situácie, keď dve

mena majú rovnako vypočítané miesta v tabuľke. Hlavnými ťažkosťami s rozptyľovacou tabuľkou sú pevná dĺžka tabuľky a závislosť hešovacej funkcie od rozmeru hešovacej tabuľky.

Napr. majme tabuľku so 64 položkami. Hešovacia funkcia konvertuje mená súborov na celé čísla od 0 po 63. Ak skúsime pridať 65-ty prvok do tejto tabuľky, tá sa bude musieť rozšíriť na 128 prvkov. Následne potrebujeme novú hešovaciu funkciu, ktorá bude mapovať mena súborov v rozsahu od 0 po 128. Mená, ktoré boli už mapované sa musia znova prepočítať. Iný spôsob, je aby každá položka hešovacej tabuľky bola sama o sebe zreťazený zoznam a kolízie by sa riešili pridaním položky do zreťazeného zoznamu.

1.5 Úvahy o efektívnosti využitia a výkone záložných pamätí

Disky sú jedno z úzkych miest počítačových systémov, pretože sú najpomalším komponentom spomedzi hlavných komponentov. Pre prekonanie tohoto nedostatku sa využívajú rôzne techniky, ktoré rozoberieme v ďalšom texte.

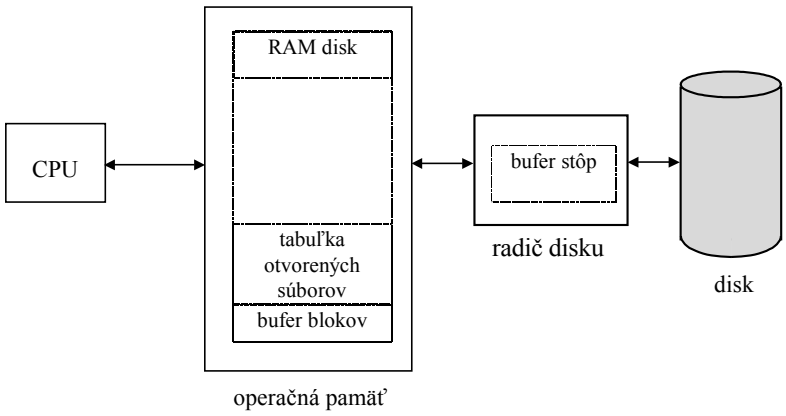
V predchádzajúcich častiach bola vysvetlená metóda zreťazeného prideľovania diskového priestoru a jej zlepšenie pomocou clustrov. Bolo skonštatované, že nedostatkom je vnútorná fragmentácia. Operačný systém BSD UNIX odstraňuje tento nedostatkom tak, že dovoľuje použitie clustera, ktorého veľkosť sa mení s rastom súboru. Veľké clustre sú používané tam, kde budú zaplnené a malé clustre sa používajú pre malé súbory a pre posledný cluster súboru.

Veľkosť ukazovateľa používaného v určitom operačnom systéme je dôležitá z dvoch dôvodov. Po prvé preto, že určuje, aký veľký priestor sa dá adresovať jeho pomocou. Napr. ak je 16 bitový, adresuje 2^{16} bitov (64 KB), 32 bitový adresuje 2^{32} bitov (4 GB) a 64 bitový ukazovateľ adresuje 2^{64} bitov. Taký veľký ukazovateľ zaberá viacej miesta pre uloženie a v zreťazených zoznamoch, indexoch atď. Ďalšími faktormi, ktoré ovplyvňujú výber ukazovateľa sú plánované zmeny vplyvom nových technológií. Napr. prvý MS-DOS podporoval súborový systém len do veľkosti 32 MB. Mal FAT tabuľku s 12-timi položkami a každá položka ukazovala na 8 KB cluster. S nástupom nových technológii a zväčšením kapacity diskov, sa väčšie disky museli deliť na časti (partition) po 32 MB. Neskôr, keď kapacita diskov bežne dosahovala 100 MB, algoritmy a dátové štruktúry museli byť zmenené, aby podporovali väčší súborový systém.

Aj po výbere základných metód práce so súborovým systémom, existuje ešte niekoľko možností ako zlepšiť výkon. Radiče diskov obyčajne majú dostatočné veľkú pamäť, aby mohli uložiť naraz celú stopu. Po nájdení požadovanej adresy sa celá stopa od požadovaného sektora ďalej prečíta do cache pamäte radiča. Potom radič presúva do pamäte požadované bloky. Ďalej tie bloky môžu byť uložené do cache pamäte operačnej pamäte. Niektoré systémy udržiavajú špeciálny úsek v pamäti (*blok cache)*, kde sa ukladajú bloky o ktorých sa predpokladá, že budú v krátkom čase znova požadované. Aj v tomto prípade pre výmenu blokov v tomto úseku sa používa algoritmus LRU, známy z výmeny stránok.

V personálnych počítačoch sa operačná pamäť bežne využíva pre zlepšenie výkonu tak, že sekcie pamäte sa oddelia pre tzv. *virtuálne* alebo *RAM* disky. V tomto prípade ovládač disku preberá požiadavky na vykonanie operácií na disku, ale ich vykonáva na RAM disku. Všetky operácie sa vykonávajú bez rozdielu, ale sú rýchlejšie. RAM disky sa používajú na dočasné uloženie dát, pretože výpadok napätia alebo nové natiahnutie systému ich zničia.

Rozdiel medzi RAM diskom a diskovou cache pamäťou je v tom, že RAM disk spravuje používateľ a cache pamäť je spravovaná operačným systémom. Na Obr. 11.19 sú ukázané rôzne umiestnenia cache pamäte v systéme.



Obr. 11.19 Niekoľko umiestnení cache pamäte pri práci s diskom

1.6 Obnovenie súborového systému

Súbory a adresáre sa uchovávajú v pamäti a na disku a je potrebné venovať pozornosť tomu, aby sa pri poruchách disku alebo zlyhaní systému dáta nestratili alebo neporušila ich konzistencia.

Tabuľka otvorených súborov sa obyčajne udržiava v pamäti a zmeny v adresároch sa v pravidelných intervaloch zapisujú na disk. V prípade zlyhania systému sa môže stať, že aktuálna informácia o adresároch nie je zapísaná na disku, čo zapríčiní nekonzistenciu dát. Obyčajne operačný systém poskytuje program, ktorý po nabehnutí systému kontroluje a koriguje tieto chyby.

Program pre kontrolu konzistencie dát porovnáva dáta v štruktúre adresára s dátovými blokmi na disku. Ktoré problémy vyrieši tento program, je závislé od použitej metódy prideľovania diskového priestoru a spôsobe správy voľného priestoru. V prípade zreťazeného prideľovania sa kontroluje, či každý blok obsahuje ukazovateľ na ďalší a takto sa dá zachrániť celý súbor a obnoviť položka v adresári. Ak sa používa indexové prideľovanie a je stratená položka súboru v adresári, situácia nie je nádejná, pretože len pomocou dátových blokov sa súbor nedá obnoviť. Kvôli tomu v Unix-e sa adresáre uchovávajú v cache pamäti pre čítanie, ale pri zápisoch sa najskôr aktualizujú dáta v štruktúre *inode, inode* sa zapíše na disk a až potom sa samotné dáta zapisujú do súboru.

Iný spôsob ako zabezpečiť dáta proti stratám z rôznych príčin je ***zálohovanie.*** Zálohovanie je uloženie všetkých dát z disku na iné médium, obyčajne na disketu, magnetickú pásku alebo optický disk. Obnovenie stratených súborov alebo celého súborového systému sa potom zaistí z tohoto média. Obyčajne operačný systém vlastní program pre zálohovanie a obnovu dát. V závislostí od podmienok, v ktorých pracuje daný operačný systém a hlavne na dôležitosti dát, ktoré sa nachádzajú na diskoch, sa určuje interval zálohovania. Môže to byť každý deň, môže to byť každý týždeň atď. Aby sa minimalizovalo kopírovanie, zálohovací program kontroluje či súbor bol modifikovaný od dátumu poslednej zálohy a ak tomu tak nie je, ušetrí si nové kopírovanie.

12 SPRÁVA PERIFÉRNYCH ZARIADENÍ

Počítač komunikuje s okolitým prostredím pomocou periférnych zariadení. Štandardné periférne zariadenia sú disk, klávesnica, obrazovka, tlačiareň, myš. Moderné počítačové systémy majú bežne aj CD-ROM, fax, modem, skener. Zoznam periférnych zariadení môže byť ďalej rozšírený podľa toho, na čo sa počítačový systém používa, napr. o kamery, teplomery, merače vlhkosti a mnoho iných.

Správa periférnych zariadení je jednou z hlavných funkcií operačného systému. Medzi periférnymi zariadeniami sú veľké rozdiely, ktoré operačný systém musí skryť pred používateľom. Používateľ pracuje so zariadeniami na logickej úrovni a často pre vykonanie jednej operácie na tejto úrovni je potrebné vydať zariadeniu niekoľko príkazov, obslúžiť niekoľko prerušení, zaistiť obsluhu mimoriadnych situácií atď.

Operačný systém poskytuje interfejs medzi periférnymi zariadeniami a zvyškom systému. Tento interfejs má snahu byť nezávislý od zariadení a byť čo najjednoduchší.

12.1 Klasifikácia vstupno/výstupných zariadení

Periférne zariadenia môžeme klasifikovať podľa niekoľkých kritérií.

Jedná z klasifikácií V/V zariadení je z hľadiska možnosti ich zdieľania medzi procesmi. Delíme ich na:

- ***Zdieľateľné zariadenia*** - môže ich používať viac procesov naraz. Stav takéhoto zariadenia sa dá ľahko odpamätať a potom znova obnoviť. Použitie zdieľateľného zariadenia jedným procesom sa môže prerušiť, jeho stav sa zapamätá a zariadenie môže byť pridelené inému procesu. To je prípad procesora. V prípade operačnej pamäti a disku je kapacita zariadenia rozdelená na časti a každá časť slúži inému používateľovi.
- ***Nezdieľateľné zariadenia*** - tieto zariadenia nemôžu slúžiť viacerým procesom naraz. Stav takéhoto zariadenia sa nedá ľahko obnoviť. Nezdieľateľné zariadenie je napr. tlačiareň. Použitie tlačiarne jedným procesom nemôže byť prerušené, aby bola pridelená inému procesu.

Operačné systémy riešia problém prideľovania nezdieľateľných zariadení obyčajne dvomi technikami. Prvá **je *výlučné pridelenie*** zariadenia. Pred použitím proces musí o pridelenie zariadenia požiadať. Po skončení práce s nim ho uvoľní. Táto technika je jednoduchá, ale môže viesť k uviaznutiu, preto správa týchto zariadení musí implementovať niektorú z metód prevencie alebo vyhnutia sa uviaznutiu.

Druhá technika je ***virtualizácia*** nezdieľateľných zariadení. V tomto prípade sa používateľovi zdá, že je k dispozícii viac zariadení, ako je v skutočnosti. Konkrétna technika virtualizácie závisí od konkrétneho zariadenia. Veľmi často používaná technika pri správe tlačiarní je ***spooling*** (**S**imultaneous **P**eripheral **O**peration **O**n-**L**ine). Podstata je v tom, že keď proces požiada o tlačiareň, jeho požiadavke sa vyhovuje aj v prípade, že tlačiareň nie je voľná. Súbory, ktoré sa majú vytlačiť sa ukladajú do frontu na disk a služobný program zaisťuje ich postupnú tlač.

Ďalšia klasifikácia V/V zariadení je možná podľa toho, či zariadenie slúži k vstupu alebo výstupu dát, a hovoríme o ***vstupných*** alebo ***výstupných*** zariadeniach.

Podľa spôsobu prenosu a prístupu k dátam, môžeme V/V zariadení rozdeliť na tieto skupiny:

- ***blokové zariadenia*** - k tejto skupine patria zariadenia, ktoré ukladajú informáciu po blokoch. Bloky sú adresovateľné a operácie sa vykonávajú po blokoch. Typickými predstaviteľmi tohoto typu zariadení sú disk, CD-ROM, floppy disk.
- ***znakové zariadenia*** - tieto zariadenia pracujú s postupnosťou znakov. Jednotlivé znaky nie sú adresovateľné. K tejto skupine patria napr. terminály, tlačiarne a iné.
- ***iné zariadenia*** - do tejto skupiny patria zariadenia, ktoré nemôžeme charakterizovať ako blokové alebo znakové. Je to napr. časovač a mnoho iných, ktoré nemôžeme zaradiť do predchádzajúcich dvoch skupín.

Ďalšie rozdelenie periférnych zariadení môžeme urobiť na základe ich použitia v počítačovom systéme:

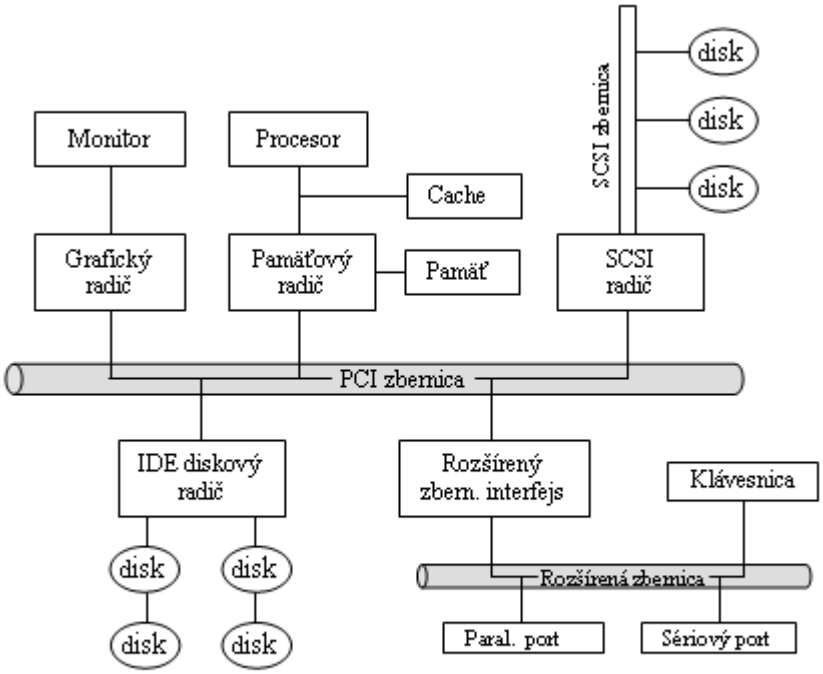
- zariadenia čitateľné pre používateľa (napr. display a tlačiareň),
- zariadenia čitateľné pre počítač (napr. disk, páska, radiče a iné) a
- komunikačné zariadenia (napr. modemy).

12.2 Hardvér V/V zariadení

Počítačový systém pracuje s veľkým množstvom periférnych zariadení, ktoré sa líšia v mnohých aspektoch. Napriek tomu pripojenie a riadenie V/V zariadení podlieha len niekoľkým konceptom.

Zariadenia komunikujú s počítačom zasielaním signálov cez kábel alebo vzduchom. Bod pripojenia zariadenia k počítačovému systému sa nazýva ***port.*** Komunikácia medzi jednotlivými komponentmi systému prebieha po ***zbernici*** (bus). Zbernica je množina spojov a prísne definovaný protokol, špecifikujúci množinu správ a riadiacich signálov, ktoré je možne posielať.

Na Obr. 12.1 je ukázaná typická štruktúra zberníc personálneho počítača. K procesoru a pamäte sú pomocou ***zbernice PCI***, pripojené rýchle zariadenia (disky, grafické karty) a pomocou ***rozšírenej zbernice*** sa pripájajú relatívne pomalšie zariadenia (klávesnica, sériový port, paralelný port).



Obr. 12.1 Štruktúra zberníc personálneho počítača

Radič je elektronická komponenta počítača, ktorá obsluhuje port, zbernicu, alebo zariadenie. Radič vlastne realizuje protokol pripojenia príslušného zariadenia ku zbernici. Niektoré radiče (napr. radič sériového portu) sú jednoduché, iné ako napr. SCSI zbernicový radič sú komplikované.

Operačný systém skoro vždy komunikuje s radičom. Sálové počítače používajú iný spôsob komunikácie s perifériami. Používajú tzv. ***kanály*****,** ktoré sú v podstate špecializované procesory, ktoré preberajú riadenie periférnych zariadení.

Radič má niekoľko registrov, ktoré používa pri komunikácii s procesorom, napr. stavový register, príkazový register, dátový register, adresný register a iné. Komunikácia s procesorom môže prebiehať použitím špeciálnych V/V inštrukcií, ktoré špecifikujú prenos dát na adresu portu.

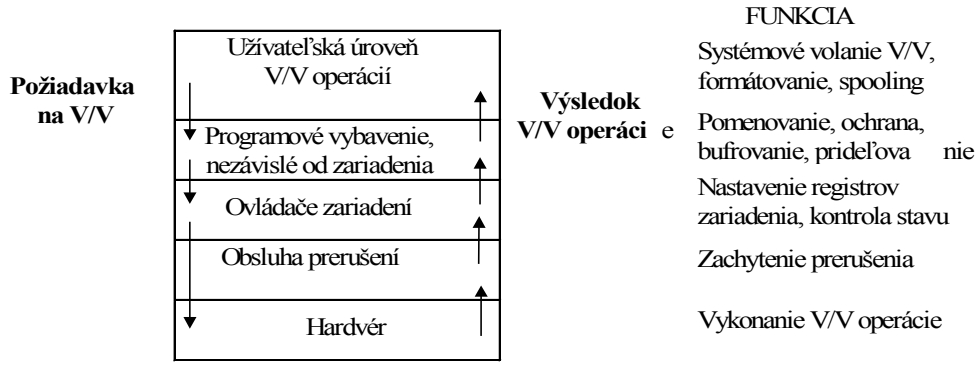
Alternatívny spôsob je mapovanie V/V do pamäte. V tomto prípade registre zariadenia sú mapované do adresného priestoru procesora. Procesor používa štandardné inštrukcie pre čítanie/zápis z/do registrov zariadenia.

Pre vykonanie V/V operácií sa využívajú nasledujúce tri techniky (pre podr. pozri sekciu 2.4):

- **programovo riadený V/V** – procesor vydá príkaz V/V modulu pre vykonanie V/V operácie pre proces a vykonáva aktívne čakanie kým sa operácia dokončí.
- **V/V riadený prerušeniami** - procesor vydá príkaz V/V modulu pre vykonanie V/V operácie pre proces a pokračuje vo vykonávaní ďalších inštrukcií. V/V modul oznámi koniec V/V operácie prerušením, ktoré sa obslúži príslušným spôsobom.
- **priamy prístup k pamäti** (DMA – Direct Memory Accsess) – modul DMA riadi výmenu dát medzi pamäťou a V/V modulom (pozri sekciu 2.4.3). Obvod DMA je špecializovaný procesor, ktorý riadi prenos dát medzi operačnou pamäťou a radičom bez účasti procesora. Priamy prístup k pamäti zrýchľuje prenos dát, pretože dáva možnosť preniesť väčšie množstvo dát naraz. Pri priamom prístupe procesor zadá obvodu DMA adresu v pamäti, kde sa majú zapísať dáta, a počet bajtov, ktoré sa majú preniesť. DMA prenesie zadaný počet bajtov a po ukončení prenosu upozorní na to systém prerušením. Prenos neriadi procesor, ale obvod DMA, takže procesor sa môže venovať inej práci.

12.3 Organizácia programového vybavenia pre správu periférnych zariadení

Štruktúra programového vybavenia periférnych zariadení je rozdelená do niekoľkých vrstiev. Nižšie vrstvy poskytujú služby vyšším. Najvyššia vrstva sú používateľské programy, ktoré využívajú služby nižších vrstiev. Vrstva, ktorá sa nachádza pod používateľským programom zaisťuje interfejs s operačným systémom, ktorý je nezávislý od zariadenia. Ďalšia vrstva ovláda konkrétne zariadenie na úrovni jeho príkazov. Najnižšou vrstvou je obsluha prerušení. Celková štruktúra programového vybavenia je ukázaná na Obr. 12.2.



Obr. 12.2 Vrstvy programového vybavenia pre obsluhu V/V zariadení a ich funkcie

3.1 V/V operácie na používateľskej úrovni

V/V operácie v používateľských programoch sa vykonávajú ako systémové volania, ktoré sú realizované ako knižničné procedúry. Tieto procedúry sú spojené s používateľským programom a neprebiehajú v privilegovanom režime. Systémové volania už boli popísané v kapitole 3, preto tu uvedieme len krátky prehľad volaní pre V/V operácie. Medzi typické operácie patria:

1. **Operácie so zariadeniami:**
   - Žiadosť o pridelenie zariadenia a uvoľnenie zariadenia.
   - Čítanie, zápis, nastavenie pozície.
   - Zistenie a nastavenie atribútov zariadenia.
   - Logické pripojenie a odpojenie zariadenia.
2. **Operácie pre prácu so súbormi:**

- Tvorba a zrušenie súboru.
- Otvorenie a uzatvorenie súboru.
- Čítanie a zápis do súboru.
- Zistenie a nastavenie atribútov súboru.

Ďalšia činnosť, ktorá súvisí s V/V operáciami, je formátovanie. Formátovanie vykonávajú systémové volania, ktoré sú uskutočnené opäť ako knižničné procedúry, t.j. prebiehajú v používateľskom režime. Tieto procedúry zaisťujú napr. prevod reťazcov na binárne hodnoty a naopak.

3.2 Programové vybavenie, nezávislé od zariadenia

Základnou úlohou tejto vrstvy je vykonávať funkcie, ktoré sú spoločné pre všetky periférne zariadenia a poskytovať jednotné rozhranie pre používateľský softvér. Na základe tohoto rozhrania, používateľ môže pracovať napr. so súborom, ktorý je na disku, na floppy disku, na tlačiarni alebo na inom zariadení bez potreby modifikovať program podľa použitého periférneho zariadenia. Medzi hlavné funkcie tejto vrstvi patrí:

- ***Jednotné pomenovanie zariadení.*** To znamená, že meno zariadenia by malo byť číslo alebo reťazec znakov, ktoré sa vytvára jednotným spôsobom pre všetky zariadenia. Napr. v operačnom systéme Unix sa disky, floppy disky alebo iné blokové zariadenia môžu montovať do súborového systému na ľubovolnom mieste, ale vždy sa na ne odkazuje jednotným spôsobom - pomocou cesty.
- ***Ochrana zariadenia***. Ochrana zariadenia nie je súčasťou každého systému. Väčšina mikropočítačových systémov neposkytuje žiadnu ochranu zariadeniam a používateľ môže s nimi robiť všetko. U sálových počítačov je situácia odlišná - tam je väčšinou pre používateľské programy priamy prístup k periférnym zariadeniam zakázaný. Viacužívateľské systémy poskytujú rôznu ochranu. Unix napr. poníma zariadenia ako špeciálne súbory a uplatňuje na ne ochranu ako na súboroch.
- ***Veľkosť bloku, nezávislú od zariadenia***. Rôzne typy diskov majú rôzne veľké sektory. Úlohou nezávislej vrstvy je skryť tieto rozdiely pred používateľom a poskytnúť mu možnosť pracovať s logickým blokom, nezávislým od fyzického sektora. Obdobný problém existuje aj u znakových zariadení. Niektoré pracujú s jednotlivými byte-mi ako napr. klávesnica, iné spracovávajú väčšie celky. Operačný systém zakrýva tieto rozdiely.
- ***Obsluha využitia vyrovnávacích pamäti***. Používateľské programy môžu čítať a zapisovať na periférne zariadenia údaje ľubovoľnej veľkosti. Na úrovni zariadenia sa ale operácie vykonávajú po blokoch. Napr. ak používateľ chce prečítať len časť bloku, operačný systém prečíta celý blok a nepoužité údaje sú pripravené pre ďalšiu požiadavku.
- ***Správa voľného priestoru blokových zariadení***. Udržiava dátové štruktúry - bitovú mapu alebo zreťazený zoznam s adresami voľných blokov, ktoré sa prideľujú novým súborom.
- ***Obsluha chýb***. Chyby vznikajúce pri práci zariadení sa obyčajne obsluhujú na najnižšej možnej úrovni. Napr. chybu pri čítaní bloku dát z disku sa najskôr pokúsi napraviť radič. Ak radič neuspeje, ovládač disku vydá príkaz na nový pokus prečítania bloku. Ak po určitom počte opakovaní operácie čítania sa chyba nenapraví, správa sa pošle nezávislej vrstve programového vybavenia. Ďalej sa chyba buď len ohlási programu, alebo ak je závažnejšia, program sa ukončí.

3.3 Ovládače periférnych zariadení

Ovládač (driver) je program, ktorý preberá od programovej vrstvy nezávislej od zariadenia požiadavky na vykonanie určitých operácií a odovzdáva ich zariadeniu vo forme jemu zrozumiteľných príkazov. Ovládač obyčajne obsluhuje jeden typ zariadení, alebo nanajvýš triedu príbuzných zariadení. Jedine v ovládači sú zabudované konkrétne údaje o zariadení, napr. koľko a akých registrov má radič zariadenia, aké príkazy rešpektuje a ešte celý rad údajov, ktoré sú iné pre každý typ zariadenia.

Operačný systém definuje *spoločnú skupinu služieb*, ktoré požaduje od periférnych zariadení. Najčastejšie sú tieto služby orientované na:

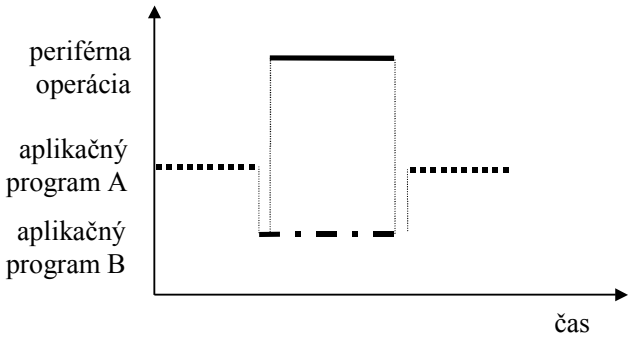
- inicializáciu zariadenia,

- prípravu a spustenie prenosu - pripravuje V/V požiadavku, formátuje dáta, prideľuje systémové bufre, nastavuje registre zariadenia, spúšťa operáciu a ukončuje prenos,
- vytvorenie a zrušenie kanálu (open, close), najčastejšie u komunikačných zariadení,
- nastavenie pozície pre čítanie dát (seek),
- riadiace služby - sú závislé od zariadenia, napr. odstránkovanie, previnutie magnetickej pásky a iné,

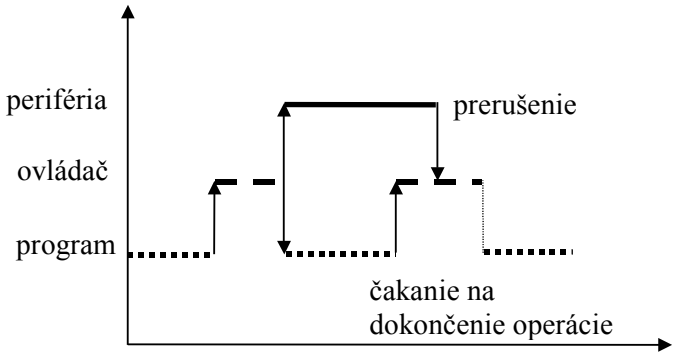
- obsluhu prerušení zo zariadenia,
- obsluhu chybových stavov,
- ukončenie prenosu.

Služby, ktoré ovládač poskytuje sú implementované v jeho vyššej vrstve. Nižšia vrstva obsluhuje prerušenia, ktoré pochádzajú zo zariadenia. Operačný systém požiada ovládač o službu a odovzdáva povel na štart činnosti, parametre požadovanej činnosti a dáta, s ktorými sa má pracovať. Periférna operácia môže prebiehať *synchrónne* (Obr 12.3) alebo *asynchrónne* (Obr. 12.4). Podľa toho, či ovládač využíva synchrónny alebo asynchrónny priebeh perifémych operácií, môže byť navrhnutý ako synchrónny alebo asynchrónny.

Charakteristickým rysom ***synchrónneho ovládača*** je, že prenos dát prebieha synchrónne z hľadiska procesu, ktorý prenos vyvolal. Inými slovami proces požiada o prenos dát a kým sa tento ukončí, čaká. V podstate ovládač pracuje ako volanie podprogramu - návrat z ovládača sa vykoná až potom, keď sa príslušná periférna operácia ukončí.



Obr. 12.3 Synchrónny priebeh perifémej operácie



Obr. 12.4 Asynchrónny priebeh perifémej operácie

Táto koncepcia ovládačov sa hodne používa, pretože sémantika funkcií, ktoré používateľský program používa pre periférne operácie, je veľmi jednoduchá. V tele ovládača sa často využívajú asynchrónne operácie, aj keď ovládač je synchrónny, pretože asynchrónne operácie dávajú možnosť paralelnej práce procesora a periférií. Synchrónne ovládače sú napr. v operačných systémoch MS-DOS a Unix.

Asynchrónne ovládače umožňujú programom požiadať o prenos v predstihu a kým periférna operácia prebieha, vykonávať inú prácu. Samozrejme program musí mať možnosť čakať na dokončenie operácie. Čakanie sa dá realizovať:

- pomocou procedúry ovládača ,
- určením dokončovacieho procesu, ktorý preberie riadenie po ukončení operácie. Dokončovací proces sa deklaruje pred zahájením operácie. Môže to byť len obsluha prerušenia.

Koniec operácie oznámi nastavenie príznaku v stavovom registri radiča alebo prerušenie. V prvom prípade sa vlastne jedná o aktívne čakanie, kedy procesor musí v slučke testovať nastavenie príznaku. V druhom prípade záleží na tom, či v jadre sú implementované synchronizačné prostriedky. Ak nemáme možnosť použiť napr. semafory, ovládač musí čakať aktívne - testovať v slučke, či nastalo prerušenie. Príklad princípu takého ovládača je uvedený na Obr. 12.5.

```
unit OVLADAC_TLACIARNE;

interfejs

var
        STATUS: STAVOVY_REGISTER;
        CNTR: RIADIACI_REGISTER;
        OB: DATOVY_REGISTER;
        KONIEC_OPERACIE: boolean;
    procedure START_PRENOSU(znak: byte);
        begin
          OB:=znak;
           KONIEC_OPERACIE:= FALSE;
          CNTR.REQ0:=1;  {Automaticky sa vykoná aj READYO := 0}
        end;
    procedure CAKAJ_NA_DOKONCENIE;
        begin
          repeat                          {aktívne čakanie na prerušenie}
          until KONIEC_OPERACIE;
        end;
    procedure DOKONCENIE;
        begin
          STATUS_READYO:=1;
          STATUS.ERROR:= ...;
          KONIEC_OPERACIE:=true;
        end;
```

```
procedure TLAC(znak:byte; CHYBA: integer);
        begin
          START_PRENOSU(znak);
          CAKAJ_NA_DOKONCENIE;
          CHYBA:=STATUS.ERROR;
        end;
        begin {inicializácia modulu}
          CNTR.MODEO:=1;
          CNTR.MODEI:=0;
        end.
```

Obr. 12.5 Ovládač tlačiarne, ktorý využíva procedúru na čakanie na prerušenie

Ak jadro ponúka synchronizačné prostriedky, môžeme ich použiť pre implementáciu pasívneho čakania na prerušenie. V príklade na Obr. 12.6, ktorý opäť uvádza principiálne riešenie ovládača tlačiarne, je použitý semafor.

```
unit OVLADAC_TLACIARNE;
interfejs
var
        TLAC_SEM: SEMAPHORE;
        STATUS: STAVOVY_REGISTER;
        CNTR: RIADIACI_REGISTER;
        OB: DATOVY_REGISTER;
procedure START_PRENOSU(znak:byte);
        begin
          OB:=znak;
          CNTR.REQ0:=1;
        {Automaticky sa vykoná aj READYO := 0}
        end;
procedure CAKAJ_NA_DOKONCENIE;
        begin     WAIT(TLAC_SEM);
        end;
procedure DOKONCENIE;                    {vyvolá sa prerušením}
        begin
          STATUS_READYO:=1;
          STATUS.ERROR:= ...;
          SEND(TLAC_SEM);
        end;
```

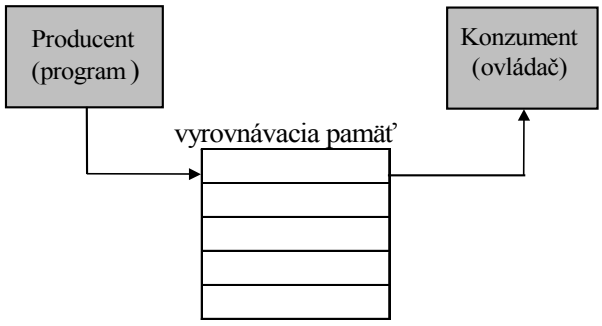
```
procedure TLAC(znak:byte; CHYBA: integer);
        begin
          START_PRENOSU(znak);
          CAKAJ_NA_DOKONCENIE;
          CHYBA:=STATUS.ERROR;
        end;

        begin {inicializácia modulu}
          INIT_SEM(TLAC_SEM, 0);
          CNTR.MODEO:=1;
          CNTR.MODEI:=0;
        end.
```

Obr. 12.6 Ovládač tlačiarne, využívajúci semafor

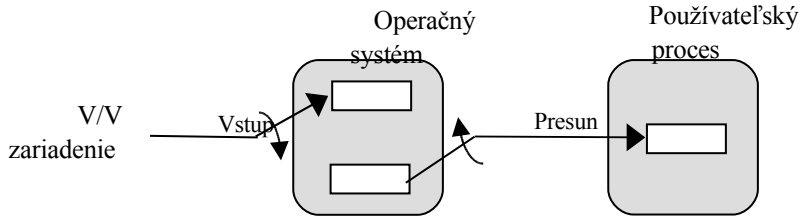
Pri návrhu ovládačov je snaha uskutočniť periférne operácie čo najefektívnejšie. Veľmi často sa k tomu používajú asynchrónne operácie, pričom vykonávaná operácia sa používateľovi môže javiť aj ako synchrónna. Asynchrónny prenos dáva možnosť využiť paralelnú prácu periférnych zariadení a procesora a tak zvýšiť výkonnosť celého systému.

Asynchrónne operácie sa dajú urýchliť vhodným využitím vyrovnávacích pamätí. Ide v podstate o aplikáciu klasickej synchronizačnej úlohy producent - konzument, kedy vyrovnávacia pamäť je v tele ovládača (Obr. 12.7). Napr. pri sekvenčnom čítaní dát z disku sa dáta môžu načítať dopredu do vyrovnávacej pamäte, aby boli pripravené, keď ich program bude potrebovať. Pri zápise dát na disk tiež môžeme použiť princíp producenta a konzumenta. Proces, ktorý požaduje výstup je v úlohe producenta a zapisuje položky, určené pre výstup, napr. bloky po 512 bajtov do vyrovnávacej pamäte. Proces konzument po zaplnení vyrovnávacej pamäte zapisuje dáta na disk.



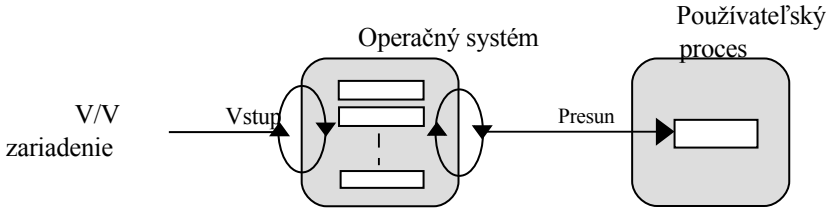
Obr. 12.7 Aplikácia úlohy producent-konzument

Využitie vyrovnávacích pamätí môže byť efektívnejšie pri ich vhodnom usporiadaní. Pri použití jednej vyrovnávacej pamäte môžeme uskutočniť len sekvenčné spracovanie jednotlivých činností: najskôr do pamäte zapíšeme, potom z nej čítame. Paralelné vykonanie zápisu a čítania sa dá uskutočniť pri použití minimálne dvoch vyrovnávacích pamätí - Obr. 12.8.



Obr. 12.8 Paralelné spracovanie pri využití dvoch vyrovnávacích pamätí

Vyrovnávacie pamäte sa môžu usporiadať aj ako cyklický front. V tomto prípade je potrebné použiť synchronizačné prostriedky pre zaistenie prístupu k pamätiam (Obr. 12.9).



Obr. 12.9 Cyklicky front vyrovnávacích pamätí

3.4 Obsluha prerušení

Obsluha prerušení je skrytá v najnižšej vrstve programového vybavenia pre obsluhu periférnych zariadení. Obyčajne sa prerušenia obsluhujú v ovládačoch zariadení. Keď sa prerušenie vyskytne, obslužná rutina vykoná všetky potrebné činnosti. V niektorých systémoch sa zvyšuje hodnota semafora, v iných sa vykoná operácia SIGNAL nad premennou typu *conditional* v monitore. Výsledným efektom je, že proces ktorý bol zablokovaný a čakal na určitú udalosť, môže pokračovať vo svojej činnosti.

12.4 Ovládače jednotlivých zariadení

4.1 Systémové hodiny

Systémový časovač generuje prerušenia v pravidelných intervaloch. Operačný systém využíva tieto prerušenia obyčajne pre splnenie dvoch úloh: pre zdieľanie času v preemptívnom multitaskingu a pre „zobudenie“ procesov, ktoré volali systémovú službu na „uspanie“ procesu na určitú dobu.

Pre obsluhu preempcie obslužná rutina prerušenia sleduje, koľko času ešte zostáva práve bežiacemu procesu a keď jeho čas vyprší, zaistí spustenie plánovača.

Aby sa spiace procesy prebudili, obslužná rutina prerušení musí spracovať zoznam časov, pre ktoré jednotlivé procesy potrebujú byť v neaktívnom stave. Až sa časový interval niektorému z procesov zníži na nulu, presunie sa do frontu pripravených procesov.

4.2 Videopodsystém

Vodeopodsystém u personálnych počítačov pozostáva z videokarty a minotora (obrazovky). Väčšina personálnych počítačov nemá na videokarte špecializovaný grafický procesor. Vďaka tomu je ovládač obrazovky netypický v tom, že nepotrebuje dolnú polovicu ovládača, pretože nemá zariadenie, od ktorého by mohol čakať prerušenie. Horná polovica ovládača jednoducho zapíše potrebné dáta do videopamäte, buď priamo alebo do bufra.

Moderné operačné systémy často túto úlohu riešia odlišným spôsobom, aby dosiahli väčšiu efektívnosť. Prístup do videopamäte zdržuje procesor, pretože pre prístup k nej súperí s obvodmi, ktoré generujú videosignál. Pritom generátor videosignálu má vyššiu prioritu ako procesor, aby nebol narušený obraz. Ovládač preto len preberie od procesu požiadavky na výstup a vo vhodnej chvíli ich zobrazuje, bez toho že by proces na to čakal.

Horná polovica ovládača má veľmi podstatnú úlohu - musí zaistiť virtualizáciu obrazovky, aby k nej mali prístup všetky procesy, ktoré potrebujú vypisovať svoje údaje. Existujú dva spôsoby virtualizácie obrazovky: buď každý proces môže mať svoju vlastnú obrazovku a používateľ volí ktorú z nich chce vidieť, alebo ovládač môže zaistiť rozdelenie obrazovky do jednotlivých okien a každý proces používa svoje vlastné okno. Druhý spôsob virtualizácie obrazovky zaviedla firma Apple na počítačoch MacIntosh a dnes je tento spôsob považovaný za najvýhodnejší. Niektoré operačné systémy kombinujú tieto dve techniky: ponúkajú viacej virtuálnych obrazoviek a každá z nich môže obsahovať viacej okien.

4.3 Klávesnica

Ovládač klávesnice má za úlohu zbierať kódy stlačených kláves, transformovať ich na príslušné kódy a odovzdávať ich programu. Pri tom sa ovládač môže podieľať na konečnej úprave odovzdávaných kódov, ako napr. odstrániť zmazané znaky, pridať konce riadkov ak je to potrebné atď.

Obslužný program klávesnice musí vyriešiť aj ďalší dôležitý problém a to spôsob generovania echa. Môže to byť buď okamžite echo na termináli alebo ako odozva na spracovanie v počítači.

Ovládač klávesnice je štandardný ovládač zariadenia, ale v mnohých prípadoch je potrebné ho výrazne rozšíriť, aby umožnil vkladanie národných znakov. Ovládač priraďuje kódy jednotlivým klávesom a tiež zaisťuje funkciu kontrolných kláves.

4.4 Tlačiareň

Tlačiareň patrí medzi najpoužívanejšie zariadenia počítačového systému, zrejme preto, že sme si ešte stále nezvykli dostatočne dôverovať iným médiám ako je papier. Ako už bolo povedané predtým, pre virtualizáciu tlačiarne sa využíva metóda spooling.

4.5 Disk

Ovládač disku je veľmi dôležitý, pretože disk je kľúčové zariadenie pre výkon celého systému. Základný princíp ovládača disku sa nelíši od obecného ovládača, ale existujú určité rozdiely. Správe disku je venovaná ďalšia kapitola. Tu uvedieme ako príklad dve základné služby disku, ktoré patria do tzv. hornej polovice ovládača a službu pre obsluhu prerušení, ktorá patrí do dolnej polovice ovládača. Príklady sú prebrané zo zdrojových textov operačného systému XINU - jednoduchý školský multitaskingový operačný systém.

Deklarácie:

```
/*disk.h */
enum {          /*diskové operácie*/
  DREAD,                        /*čítanie z disku*/
  DWRITE                        /*zápis na disk*/
};
struct dreg {                                   /* požiadavka */
  int drpid;        /* pid procesu, ktorý vzniesol požiadavku */
```

```
    unsigned long drdba;                    /* disková adresa */
    char *drbuff;                           /* bufer */
    char drop;                              /* operácie */
    int drerror;                    /* chybové stavy, 0 - úspech */
    struct dreg *drnext;        /* ukazovateľ na ďalšiu požiadavku */
  };

  struct dsblk {
    struct dreg *dreglst;               /* front požiadaviek */
    void (*_dwrite)();                  /* zápis na disk */
    void(*_dread)();                    /* čítanie z disku */
    void(*_derror)(void);               /* chyba prenosu */
        };
        extern struct dsblk dstab[];
/*end of file */
```

Tabuľka zariadení:

```
  /* conf.h */
                                                /* mena jednotlivých služieb */
  enum { Init, Open, Close, Read, Write, Seek, Getc, Putc, CNTL };
  #define NDVPROCS  9                           /* počet služieb */

  struct devsw {
    int (*dvproc[NDVPROCS])(struct devsw,...);      /*  služby */
    int dvminor;
                                                /* číslo konkrétneho zariadenia */
    int dvivec,dvovec;                  /* vektor prerušenia */
    void interrupt (*dviint(),(*dvoint);
                                                /*obslužná rutina prerušenia */
    };
    extern struct devsw devtab[];
    /* end of file  */
```

Implementácia služby na čítanie:

```
/* dsk_read  - požiadavka na čítanie z disku */
#include <disk.h>
#include <conf.h>

int dsk_read (struct devsw *dev, char *buff, unsigned snum)
{
  struct dreg *rq;
  int ret=0;                                  /*  bez chyby  */
  char x;

  sdisable(x);
  new(rq);                                    /* vyhradenie pamäte  */
  rq->drdba=disk_address(snum);
  rq->drpid=currpid;
  rq_drbuff->-buff;
  rq->drop=DREAD;
  if (dskenq(&dstab[dev->dvminor], rq)) {
                              /* zaradenie požiadavky do frontu */
  suspend(surrpid);
       /* proces je pozastavený, kým sa jeho požiadavka splní */
  ret=rq->drerror;                                    /* návratový kód */
  }
  dispose{rq};
  restore(x);
  return(ret);
}
/* end of file */
```

Služba na čítanie:

```
/* dsk_write - požiadavka na zápis na disk */
#include <disk.h>
#include <conf.h>
```

```
int dsk_write(struct devsw *dev, char *buff, unsigned snum)
{
  struct dreg *rq;
  char x;

  sdisable(x);
  new(rq);                          /* vyhradenie pamäte  */
  rq->drdba=disk_address(snum);
  rq->drpid=currpid;
  rq_drbuff->-buff;
  rq->drop=DWRITE;
  if (dskenq(&dstab[dev->dvminor], rq));
  restore(x);
  return(0);
  }
  /* end of file */
```

Rutina na obsluhu prerušenia:

```
/* dsk_iohandler.c */
#include <disk.h>
#include <conf.h>

void interrupt dsk_iohandler()
{
  struct dsblk *dsk=&dstab[devtab[DISKDEV].dvminor);
  struct dreq *rq=dsk->dreqlst;
            /* nastavenie na prvú požiadavku frontu požiadaviek */

  rq->drerror=dsk->derror();
          /*nastaví podľa výsledku prenosu */
  if ((dsk->dreqlst=rq->next)!=NULL
                  /*ak front nie je prázdny, spustí ďalšiu operáciu */
  run_disk(dsk);
```

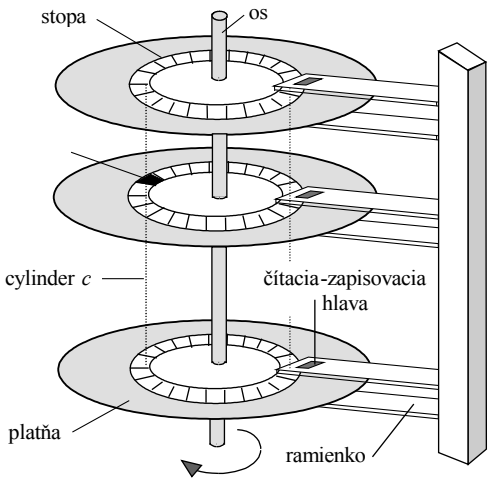
```
  switch (rq->drop) {
          case DREAD:
               /* ak čítal, presunie ďalšiu požiadavku do frontu */
                    ready(rq->drpid, RESCHYES);
                    return;
  case DWRITE:
                         /*ak zapisoval, uvolní používanú pamäť */
                    dispose(rq_buff);
                    dispose(rq);
          }
  }
/* end of file */
```

13 SPRÁVA DISKOVÝCH ZARIADENÍ

Disk je najdôležitejšou periférnou pamäťou moderných počítačových systémov. Predstavuje najnižšiu úroveň súborového systému. V tejto kapitole sú rozobraté algoritmy pre plánovanie pohybu ramienka disku, problémy správy disku a diskové polia.

13.1 Štruktúra disku

Fyzická štruktúra disku je ukázaná na Obr.13.1. Každý disk pozostáva z platní a každá platňa má dva povrchy. Povrch je pokrytý magnetickým materiálom a na ňom sa zapisuje informácia. Motor disku beží na vysokých otáčkach, obyčajne 60 otáčok/sek. Čítanie a zápis vykonávajú hlavy, ktoré sú uložené na posuvnom ramienku. Povrch je logicky rozdelený na *stopy* a stopy sú rozdelené na *sektory.* Informácia sa číta/zapisuje z/na na sektor, ktorý sa práve nachádza pod hlavou. Na povrchu platní sú stovky koncentrických stôp, ktoré obsahujú tisícky sektorov.



Obr. 13.1 Diskový mechanizmus

Informácia na disku sa adresuje pomocou adresy, ktorá ma viac častí, a to číslo zariadenia, povrch, stopa a sektor. Všetky stopy na jednom zariadení, ktoré sa dajú dosiahnuť bez pohybu hláv (t.j. ekvivalentné stopy na všetkých povrchoch), sa nazývajú *cylinder*.

V rámci stopy sa informácia ukladá do sektorov. Sektor je najmenšia jednotka informácie, ktorá môže byť zapísaná alebo prečítaná z disku. Väčšinou je veľkosť sektoru daná hardvérovo. Pohybuje sa od 32 do 4096 bajtov, ale najčastejšie je 512 bajtov. Počet sektorov je od 4 do 32 pre stopu a počet stôp je od 20 do 1500 pre povrch. Pre lokalizáciu sektoru sa používajú špeciálne značky medzi sektormi. Pri operáciách čítania/zápisu sa hlavy posunú nad požadovanú stopu (čas posuvu) a elektronicky sa prepnú na správny povrch. Potom sa musí počkať určitú dobu, kým sa požadovaný sektor dostane pod hlavu (čas reakcie).

V/V prenosy sa uskutočňujú v jednotkách, ktoré pozostávajú z jedného alebo viacerých sektorov a nazývajú sa *bloky*. Adresovanie bloku vyžaduje číslo stopy alebo cylindra, číslo povrchu a číslo sektoru. Takže na disk sa môžeme pozerať ako na trojdimenzionálne pole blokov. Ak *s* je počet sektorov na stopu, *t* je počet stôp pre cylinder, číslo cylindra je *i*, číslo povrchu *j* a sektoru *k*, potom pre číslo bloku *b* platí:

$$b = k + s \times (j + i \times t)$$

Pri tomto mapovaní prístup k bloku *b+1*, ak predtým bol spracovaný blok *b,* vyžaduje posuv iba v prípade, že blok *b* bol posledným blokom jedného cylindra a *b+1* je prvým blokom ďalšieho cylindra. Aj v tomto prípade bude posuv iba o jednu stopu.

13.2 Algoritmy plánovania pohybu ramienka disku

Väčšina programov pri svojej práci intenzívne používa disk, hlavne pre vstupné a výstupné súbory. Rýchlosť diskových operácií silne ovplyvňuje celkový výkon systému a čas na spracovanie úloh. Operačný systém používa rôzne algoritmy pre plánovanie pohybu ramienka disku tak, aby sa priemerný čas prenosu dát skrátil.

Čas, potrebný pre prenos bloku dát, pozostáva z troch zložiek:

- čas pre vystavenie ramienka s hlavami,
  - čas pre nájdenie požadovaného sektoru - to je vlastne čas na otočenie platne tak, aby sa požadovaný sektor dostal pod hlavy,
  - vlastný prenos dát.

Kedykoľvek proces požaduje V/V operáciu, volá systém s parametrami, ktoré špecifikujú presne požadovaný prenos. Sú to:

- *druh operácie* - vstup alebo výstup,
  - *diskovú adresu* - číslo bloku, ktoré príslušný modul zo súborového systému preloží na čísla zariadenia, cylindra, povrchu a sektorov.
  - *adresa v pamäti* - odkiaľ alebo kam sa budú prenášať dáta.
  - *počet bajtov,* ktoré sa majú preniesť.

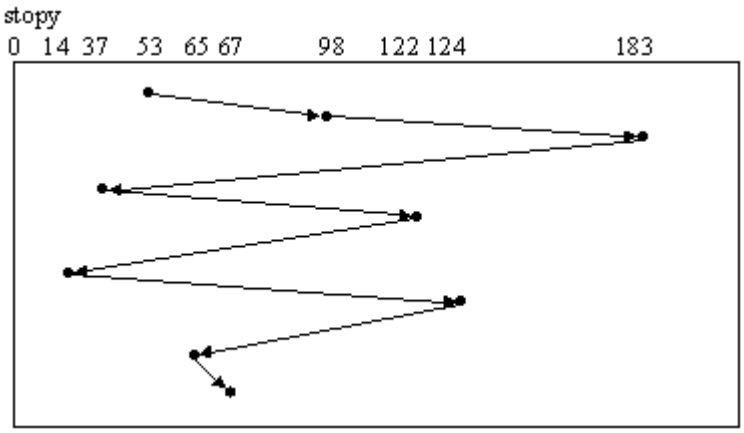
Požiadavky na prenos z/na disk sa radia do frontu. Obsluha jednotlivých požiadaviek znamená, že sa ramienko s hlavami musí nastaviť na požadovanú stopu, potom počkať na sektor a nakoniec preniesť dáta. Poradie obsluhy požiadaviek môže ovplyvniť celkový čas potrebný pre prenos, takže sa používajú rôzne algoritmy pre minimalizáciu času pohybu ramienka disku.

2.1 Plánovanie podľa poradia príchodu (FCFS)

Tento algoritmus obsluhuje požiadavky podľa poradia ich príchodu (First Come, First Served). Je to najjednoduchší algoritmus a stretli sme sa s ním pri výklade plánovania procesov. Vysvetlime algoritmus na základe situácie, kedy front požiadaviek na disk vyzerá nasledovne (uvedené sú čísla požadovaných stôp):

98, 183, 37, 122, 14, 124, 65 a 67.

Ramienko sa na začiatku nachádza na stope 53. Ak zoberieme ako jednotku vzdialenosť medzi stopami, potom od 53-tej stopy k 98-mej vykoná ramienko presun o 45 jednotiek, od 98-mej k 183-tej ďalších 85 jednotiek atď. Celkový presun pre uspokojenie všetkých požiadaviek vo fronte bude *640 jednotiek*. Pohyb ramienka je znázornený na Obr.13.2.

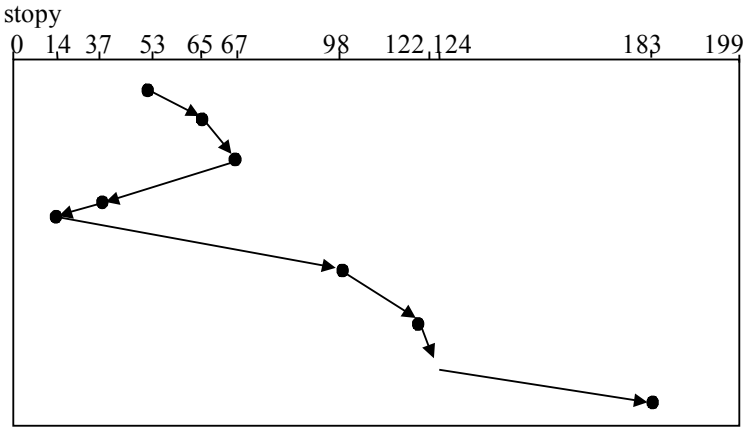


Obr. 13.2 Pohyb ramienka podľa algoritmu FCFS

2.2 Algoritmus najkratšieho presunu

Algoritmus najkratšieho presunu (Shortest Seek Time First - SSTF) obsluhuje najskôr z frontu požiadaviek tú požiadavku, ktorá bude požadovať najmenší pohyb vzhľadom na momentálnu pozíciu ramienka.

Prípad, ktorý sme rozobrali pri predchádzajúcom algoritme, použijeme znova. Tento krát pohyb ramienka bude odlišný. Zo stopy 53 sa ramienko presunie na stopu 65, potom na 67, potom na 37 atď., vyberajúc si vždy najkratší pohyb (Obr.13.3). Celkovo ramienko vykoná presun o *236 jednotiek*, čo znamená, že čas pre obsluhu požiadaviek sa podstatne zlepšil.



Obr. 13.3 Pohyb ramienka podľa algoritmu SSTF

Algoritmus SSTF je obdoba algoritmu SJF z plánovania procesov a tak ako aj algoritmus SJF môže zapríčiniť *starváciu* niektorých požiadaviek. Napr. nech príde požiadavka na stopu 14 a po nej na stopu 183. Ak počas obsluhy stopy 14 príde požiadavka, ktorá je bližšie k 14 ako k 183, požiadavka na 183 bude čakať. Teoreticky tých požiadaviek, ktoré sú bližšie k 14, môže byť viacej a obsluha požiadavky na 183 bude stále odkladaná. Tento prípad je nepravdepodobný, ale je možný.

Algoritmus SSTF je lepší ako FCFS, ale nie je optimálny. Uprednostňuje stopy, ktoré sú uprostred a táto jeho vlastnosť sa dá využiť tak, že na prostredných stopách sa budú umiestňovať bloky, ktoré vyžadujú rýchly a častý prístup.

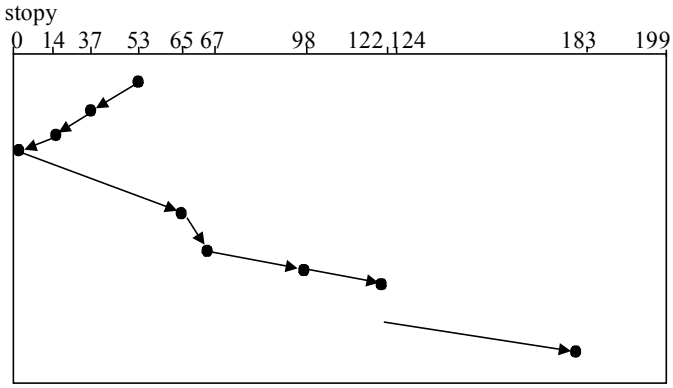
2.3 Algoritmus výťahu

Algoritmus výťahu (SCAN) odzrkadľuje dynamickú povahu požiadaviek. Pohyb ramienka pri použití tohoto algoritmu začína na jednom konci disku a pokračuje k druhému koncu (ako výťah) a potom naspäť, pričom obsluhuje požiadavky na stopy, ktoré sú po ceste pohybu.

Aplikujme algoritmus výťahu na náš príklad, kedy potrebujeme obslúžiť front požiadaviek:

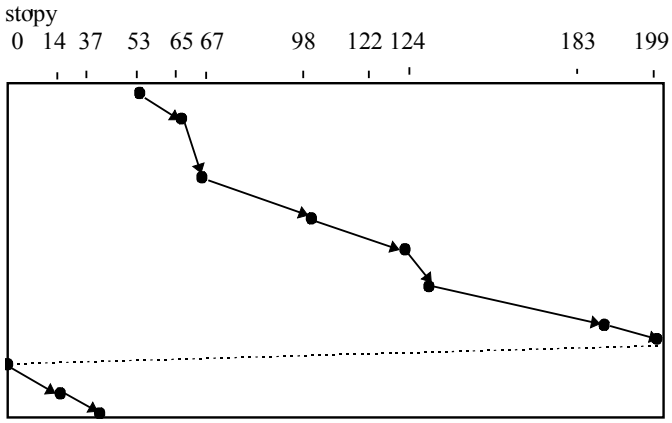
98, 183, 37, 122, 14, 124, 65 a 67.

Najskôr potrebujeme vedieť, ktorým smerom sa pohybuje ramienko a od ktorej pozície. Pozícia bude ako v predchádzajúcich prípadov - 53. Ak sa ramienko pohybuje k 0, obslúži najskôr stopu 37 a 14. Na stope 0 sa pohyb obráti naspäť a budú obslúžené požiadavky 65, 67, 122, 124 a 183 (Obr.13.4). Ramienko vykonáva pohyb po všetkých stopách. Ak príde požiadavka na stopu, ktorá je pred ramienkom v smere pohybu, bude obslúžená hneď, ak je za ním, bude musieť počkať, kým sa ramienko začne pohybovať späť. Maximálna dĺžka čakania požiadavky na obsluhu bude rovná dvojnásobku počtu stôp. Pri použití tohoto algoritmu sú zasa zvýhodnené prostredné stopy, kde maximálna dĺžka čakania na obsluhu je rovná len počtu stôp.



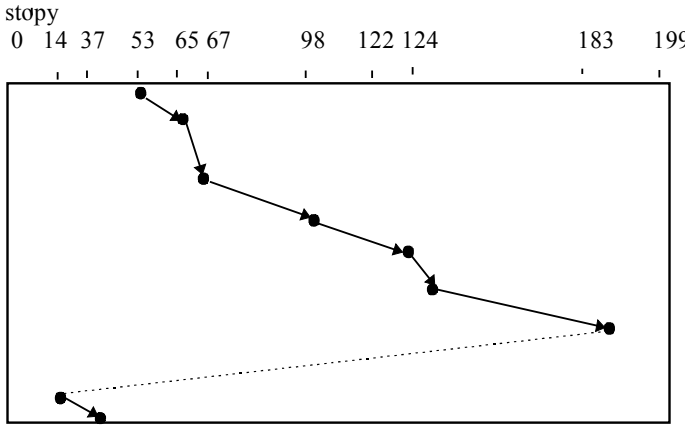
Obr. 13.4 Pohyb ramienka podľa algoritmu výťahu

Variantom algoritmu výťahu je algoritmus C-SCAN (Circular SCAN), ktorý vedie pohyb ramienka rovnako, ale požiadavky sú obsluhované len v jednom smere. V opačnom smere sa ramienko pohybuje naprázdno (Obr.13.5). Pri použití tohoto algoritmu požiadavky na všetky stopy sú obsluhované rovnako spravodlivo.



Obr. 13.5 C-SCAN algoritmus

Existuje aj **ďalší variant algoritmu výťahu (C-LOOK)**, pri ktorom sa ramienko nepohybuje až do konca disku, ale len po najvzdialenejšiu stopu, na ktorú má požiadavku v tom smere. Po dosiahnutí poslednej požadovanej stopy sa pohyb obráti (Obr.13.6).



Obr. 13.6 C-LOOK algoritmus

2.4 Výber algoritmu pre plánovanie pohybu ramienka

Výber algoritmu pre pohyb ramienka je komplikovaný, pretože závisí od mnohých faktorov. Algoritmus najkratšieho presunu je prirodzený a dosť často používaný. Algoritmus výťahu a jeho modifikácie sú vhodné pre systémy, ktoré silne zaťažujú disk. Samozrejme výsledky každého algoritmu sú závislé od počtu a druhu požiadaviek. Napr. ak front požiadaviek obsahuje väčšinu času len jednu nevybavenú požiadavku, potom všetky algoritmy sú prakticky ekvivalentné a dokonca obsluha je vhodná v poradí príchodu.

Požiadavky na diskové operácie závisia aj od metódy prideľovania diskového priestoru. Napr. pri súvislom prideľovaní pohyb ramienka pre čítanie súboru je menší, pretože bloky sú umiestnené vedľa seba. Pri zreťazenom a indexovom prideľovaní sa diskový priestor využíva lepšie, ale pohyb ramienka je väčší, a tým sa predĺži aj čas pre V/V operácie.

Umiestnenie adresárov a indexových blokov na disku je veľmi dôležité, pretože to sú štruktúry, ktoré sa veľmi intenzívne používajú. Umiestnením adresárov na stredných stopách môžeme ušetriť až polovicu pohybu ramienka.

Pretože algoritmus pohybu ramienka je závislý od mnohých faktorov, obyčajne je umiestnený v samostatnom module, aby mohol byť vymenený v prípade potreby. Najčastejšie sa ako východiskový vyberá algoritmus spracovania v poradí príchodu alebo algoritmus najkratšieho presunu. Niekedy výrobcovia diskov dodávajú disky s plánovacím algoritmom, zabudovaným do hardvéru radiča. Tento prístup nie je vždy vhodný, pretože operačný systém môže dodávať požiadavky na diskové operácie už usporiadané podľa svojich kritérií a radič disku ich preusporiada, čím sa efekt správneho výberu stratí.

13.3 Diskové polia

S narastajúcim výkonom počítačov neustále narastajú i požiadavky prevádzkovaných aplikácií na dátový priestor diskov. Aby bolo možné tieto požiadavky uspokojiť, musia sa neustále pridávať ďalšie disky s veľkou kapacitou, čo je finančne nákladné. S počtom diskov priamo úmerne narastá i pravdepodobnosť poruchy niektorého z nich a tak i straty časti dát. Zväčšovanie diskového priestoru je možné vyriešiť buď inštaláciou spoľahlivých a veľmi drahých veľkokapacitných diskov alebo často používanou technológiou diskových polí **RAID** *(Redundant Arrays of Inexpensive Disks).*

Filozofiou RAID je použitie väčšieho množstva lacných, bežne dostupných diskov, na ktorých sú uložené jednak samotné dáta a jednak i pomocné informácie, ktoré v prípade závady na jednom disku (popr. na viacerých) umožní jednoduchú (automatickú) rekonštrukciu zničených dát. Technológia RAID bola štandardizovaná v roku 1987 univerzitnou konferenciou v Berkeley v Kalifornii. Tu vznikol dokument, nazývaný *„RAID Book“*, ktorý popisuje jednotlivé architektúry zostavované z bežných pevných diskov a zavádza jednotnú terminológiu v tejto problematike. Definuje celkom 7 základných úrovní, označovaných RAID 0 až RAID 6 a kombinácie týchto základných úrovní. Kombinácie základných úrovní sa pokúšajú odstrániť nedostatky základných úrovní a sú označované kombináciou ich čísiel (napr. RAID 01).

1.1 Vlastnosti diskových polí

Výhody, ktoré prináša použitie diskových polí sú:

- ***Spoľahlivosť*** *- je to pôvodný cieľ, kvôli ktorému diskové polia vznikli. Ich implementáciou sa stala výmena vadných diskov počas prevádzky samozrejmosťou.*
- **Vysoký výkon** - diskové polia sa spravidla nasadzujú v systémoch, pracujúcich s veľkými objemami dát, čo vyžaduje aj väčší výkon od diskového subsystému. Začínajúc úrovňou RAID 3 je s odolnosťou proti strate dát riešená zároveň i výkonnosť diskového subsystému.
- **Otvorenosť** - a to ako softvérová, tak i hardvérová. Diskový systém by mal umožňovať prechod na iný operačný systém alebo na inú HW platformu bez väčších technických komplikácií.

- **Flexibilita** - možnosť rozčleniť dátový priestor podľa potrieb užívateľov alebo nárokov operačného systému, možnosť zdieľať diskové pole niekoľkými hostiteľskými počítačmi, a to i rôznych hardvérových platforiem s rôznymi operačnými systémami.

Nevýhody - použité disky (RAID 0 až 5) musia mať rovnakú kapacitu a charakteristiky - počet hláv, stôp a sektorov a pokiaľ je to možné i rovnaké prevedenie.

1.2 Spôsoby ovládania pripojených diskov

2.1 Softvérové ovládanie

Jedná sa spravidla o službu operačného systému alebo špeciálny program, ktorý priamo preberá riadenie prístupu k diskom a pracuje s nimi ako s „virtuálnym“ diskovým poľom. Toto riešenie je veľmi nevýhodné, pretože značne zaťažuje samotný operačný systém ďalšou úlohou a znižuje výkonnosť celého systému. Pri výpadku operačného systému je potom väčšie nebezpečie porušenia integrity dát. Významnou slabinou tohoto riešenia je taktiež obmedzenie použitia diskového poľa len na operačný systém, ktorý ho obsluhuje.

1.2 Hardvérové ovládanie.

Hardvérový radič je umiestnený v počítači a jeho logika riadi prístup k dátam, čím sú eliminované nevýhody softvérového riešenia. Výkonnosť je neporovnateľne väčšia a spravidla je možnosť prevádzky pod rôznymi operačnými systémami.

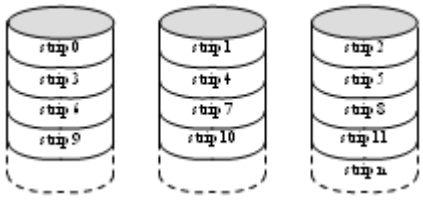
Podľa fyzického umiestnenia delíme tieto ovládania na **interné a externé.** Interný RAID je umiestnený vo vnútornom priestore serveru a externý, je umiestnený v samostatnej technologickej skrini s vlastným zdrojom napätia.

Výhodou **interného RAID** je možnosť použiť radič, ktorý je fyzicky rozložený na niekoľko konštrukčných dosiek vzájomne spojených samostatnou zbernicou. Dosky sú umiestnené do viacerých slotov základnej dosky počítača a tým sa rozšíri dátový kanál medzi operačnou pamäťou počítača a diskovým subsystémom. Nevýhodou je závislosť na technickej spoľahlivosti počítača a závislosť na hardvérovej platforme.

Externý RAID je systém celkom samostatný, na hostiteľský počítač sa pripája väčšinou cez zbernicu SCSI. Jeho kladom je nezávislosť na technických prostriedkoch počítača a HW platformách, čo je výhodné pri prepojovaní diskového poľa na iný počítač, ak dôjde k poruche. Niektoré implementácie diskových polí umožňujú i pripojenie na viacero hostiteľských počítačov zároveň.

1.3 RAID 0 (Disk striping)

Táto úroveň *nezaisťuje bezpečnosť (nejedná sa teda o plnohodnotný RAID),* ale iba zrýchľuje zápis a čítanie dát. Jej použitie je významné v systémoch, ktoré požadujú vysoký výkon a nie je nutné zaisťovať bezpečnosť informácií. Ukladané dáta sa zapisujú ne jednotlivé disky v blokoch (stripoch). Veľkosť jedného bloku je spravidla 64 KB. Súbor je rozdelený na takéto bloky a prvý blok je zapísaný na prvý disk, druhý blok na druhý disk atď. (Obr. 13.7).



Obr. 13.7 RAID 0 (Disk striping)

Výhodou je zvýšenie prenosovej rýchlosti v dôsledku paralelnej práce rovnakých diskov. Zvýšenie rýchlosti zodpovedá počtu diskov v zostave.

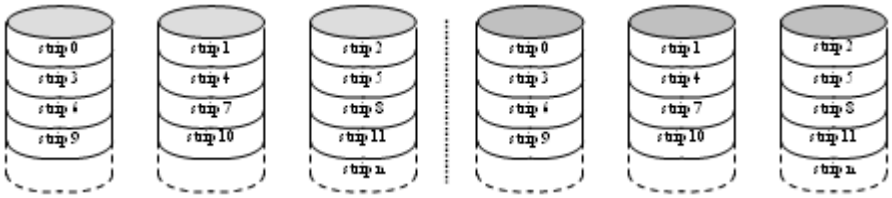
Nevýhodou je to, že pri výpadku jedného disku je stratená tá časť súboru, ktorá je na tomto disku uložená a tým dôjde k porušeniu integrity celého dátového súboru. Veľkosť voľného dátového priestoru sa rovná súčtu veľkostí pripojených diskov.

1.4 RAID 1 (Disk mirroring)

Bezpečnosť dát je zaistená duplicitným zápisom dát na dva rovnaké disky (zrkadlenie dvoch diskov, oblasť jedného sa zrkadlí na druhom - je jeho presnou kópiou). (Obr. 13.9).

Nevýhodou tohoto riešenia je obmedzenie možnosti vykonávať v jednom okamžiku iba jednu operáciu zápisu a výsledná polovičná kapacita diskov. Dochádza tiež k väčšej záťaži procesora a kanálu DMA.

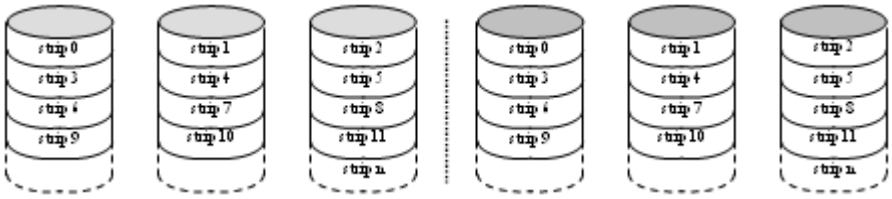
Výhodou je však zase relatívna jednoduchosť tohoto riešenia, nízka cena, jeho častá integrácia priamo do operačných systémov a maximálna bezpečnosť. Prístup k diskom môže byť realizovaný buď jedným (mirroring) alebo dvoma dátovými kanálmi (duplexing). Duplexing je výhodnejší, pretože má vyššiu odolnosť proti poruchám radiča a dátových káblov.



Obr. 13.9 RAID 1 (Disk mirroring)

1.5 RAID 2 (Redundancy through Hamming code)

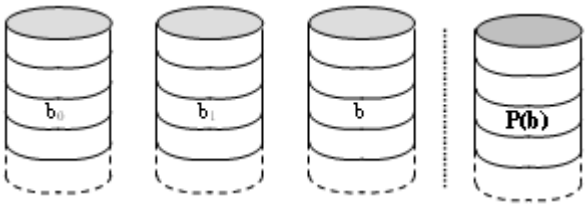
Dáta je možné obnoviť pomocou tzv. **Hammingovho kódu**, ktorý umožňuje detekovať a odstraňovať chyby dát. Súbory sú uložené po bitoch na dátových diskoch a bity Hammingovho kódu sú uložené na kontrolných diskoch. Veľkou nevýhodou tohoto princípu je zaťaženie procesora a časové oneskorenie, spôsobené výpočtom opravených dát pri zápise. Pri čítaní je prenosová rýchlosť približne rovnaká ako u RAID 0. Kontrolné disky zaberajú zhruba 30-50 % celkovej diskovej kapacity. Táto úroveň bola prekonaná vyššími RAID a dnes sa bežne nepoužíva(Obr. 13.11).



Obr. 13.11 RAID 2 (Redundancy through Hamming code)

1.6 RAID 3 (Disk striping with ECC stored as parity)

Aby sa dáta dali obnoviť, to v tomto prípade zaisťuje ukladanie informácie na jeden paritný disk. Dáta sú po bitoch uložené na niekoľkých dátových diskoch. Bežne sa používajú zostavy s dvoma až štyrmi dátovými a jedným paritným diskom (paritný disk zaberie 20-33 % celkového diskového priestoru). Nevýhodou je možnosť vykonávať v jednom okamžiku len jednu V/V operáciu. RAID 3 sa uplatní predovšetkým tam, kde je požadovaný rýchly prístup k jednému veľkému súboru (napr. obrazové aplikácie). Celkom nevhodné je jej používanie u databázových alebo súborových serverov (Obr. 13.13).



Obr. 13.1 RAID 3 (bit-interleaved parity)

Obnova dát po výpadku pri použití parity je veľmi jednoduchá. Predpokladáme použitie diskového poľa, pozostávajúce z piatych diskov, pričom D0 až D3 sú dátové disky a D4 je paritný disk. Parita *i*-teho bitu sa vypočíta podľa nasledovného vzorca:

$$D4(i) = D3(i) \oplus D2(i) \oplus D1(i) \oplus D0(i) \qquad (\oplus \text{ označuje operáciu XOR})$$

Predpokladajme, že vypadne disk D1. Ak k obom stranám predchádzajúcej rovnice pripočítame $D4(i) \oplus D1(i)$, potom získame:

$$D1(i) = D4(i) \oplus D3(i) \oplus D2(i) \oplus D0(i)$$

To znamená, že obsah každej časti (strip) je možne obnoviť na základe obsahu zodpovedajúcich častí (strip-ov) diskov poľa. Tento princíp platí pre RAID úrovne 3 až 6.

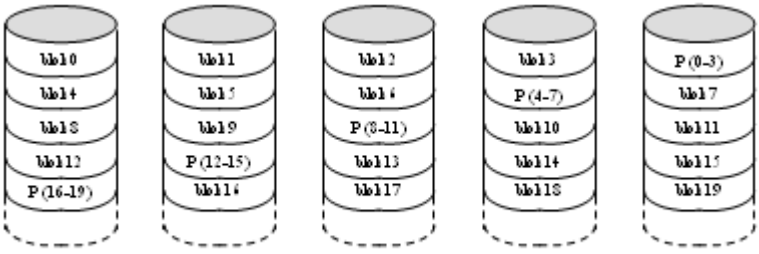
1.7 RAID 4 (Block level parity)

Obnova dát je opäť zaistená paritou, ale hlavný nedostatok úrovne RAID 3, t.j. neschopnosť vykonávať v jednom okamžiku viac I/O operácií, bol odstránený zápisom dát po bajtoch a nie po bitoch (pozri Obr. 13.15). Operácia čítania je niekoľkokrát rýchlejšia (než u jedného disku) a zápis je o niečo pomalší. Redundancia je 1/(N+1), kde N je celkový počet dátových diskov. Jednalo sa o vývojový stupeň RAID 3.

Obr. 13.15 RAID 4

1.8 RAID 5 (Block-level distributed parity)

Vychádza z RAID 4. Obnova dát je zabezpečená paritnou informáciou, uloženou cyklicky medzi dátami na všetkých diskoch súčasne (každý disk rozdelený na 5 častí - jedna časť je parita), v prípade výpadku jedného disku sa jeho dáta dopočítajú. Tým je taktiež odstránený nedostatok predošlých úrovní, t.j. jedinej operácie zápisu. Počet súčasne vykonávaných operácií zápisu je rovný polovici počtu diskov. Veľkosť paritného priestoru je opäť 1/(N+1).(pozri ***17***)



Obr. 13.17 RAID 5

1.9 RAID 6(Dual redundancy)

Jedná sa skôr o teoretickú záležitosť. Špecifikácia bola síce definovaná, ale k praktickému využitiu nedošlo. Paritná informácia je uložená ako pri úrovni RAID 5, ale je umiestnená dvojnásobne - na dvoch rôznych diskoch. Výhodou je možnosť výpadku dvoch diskov súčasne bez toho, aby sa narušila integrita dát. Nevýhodou sú extrémne dlhé časy pri zápise. Veľkosť paritného priestoru je 2/(N+2). Redundantná informácia je ukladaná dvojakým spôsobom.

1.10 RAID 7

Tento zatiaľ posledný vývojový stupeň bol dokončený až po zmieňovanej konferencii v Berkeley. Bol navrhnutý firmou *Storage Computer Corporation*, ktorá je vlastníkom patentu na túto technológiu. Architektúra RAID 7 je celkom odlišná od všetkých predchádzajúcich úrovní. Zatiaľ čo nižšie úrovne sú v spôsobe spracovania dát paralelnými štruktúrami, čo znamená nutnosť inštalácie jedného typu disku s rovnakými parametrami, u RAID 7 hovoríme o asynchrónnej architektúre. Základným princípom asynchrónnej architektúry je vzájomná nezávislosť diskov.

V systéme sú definované tri skupiny diskov – dátové, paritné a stand-by (rezervné). Tieto systémy používa automaticky k rekonštrukcii dát, ak dôjde k havárii disku. Pre odstránenie rizika zlyhania logiky diskového poľa je možné riadiacu logiku zálohovať zdvojením. Inštalované disky môžu byť rôzneho typu, kapacity i formátu a od rôznych výrobcov. Obnova dát je zaistená paritnou informáciou, ktorá umožňuje rekonštrukciu dát i pri výpadku niekoľkých diskov zároveň.

1.11 Ďalší vývoj diskových polí

Úrovne RAID 2, 4 a 6 sa nikdy komerčne nepresadili. Úroveň 5 sa uplatňuje v systémoch, kde nie je možné z technických dôvodov nasadiť RAID 7 (kompatibilita) a je zároveň lacnejší.

13.3 Správa disku

Správa disku zahŕňa ešte niekoľko funkcií, ktoré sú veľmi dôležité pre celý systém. Sú to formátovanie disku, inicializácia systému (booting) a správa chybných blokov na disku.

13.3.1 Formátovanie disku

Po výrobe disk obsahuje len čisté platne. Pred použitím v počítačovom systéme sa disk musí pripraviť, t.j. musí sa rozdeliť na sektory. Tejto operácii sa hovorí *fyzické formátovanie.* Naformátovaný disk pozostáva z množiny sektorov na každej stope, ktoré sú adresovateľné. Každý sektor má hlavičku, ktorá obsahuje číslo sektoru a priestor pre výsledok kontroly pomocou *opravného kódu (ECC).* Po zápise dát sa spočíta kontrolný súčet z bajtov v sektore a uloží sa. Neskôr, pri čítaní dát v sektore sa kontrolný súčet spraví znova a porovná sa s uloženou hodnotou. Ak hodnota nie je rovnaká, znamená to, že sektor môže byť chybný. Výpočet kontrolného súčtu a porovnanie robí automaticky radič disku.

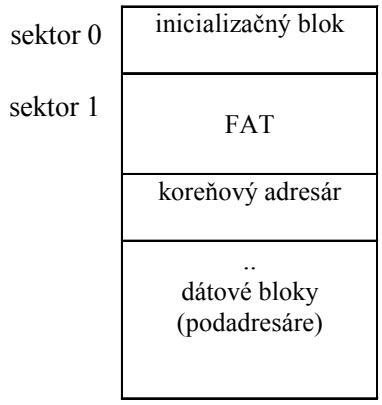
Väčšina diskov sa dodáva výrobcami už fyzicky naformátovaných. Pred začlenením disku do počítačového systému je potrebné ešte ho rozdeliť na časti (partition) a vytvoriť logický formát. Pri logickom formátovaní sa na disk zapíše prázdny adresár a môže sa inštalovať tabuľka FAT alebo štruktúry *inode*, zoznam voľných blokov alebo iné informácie, ktoré sú potrebné pre správu diskového priestoru.

13.3.2 Inicializácia systému

Každý počítač potrebuje po zapnutí alebo resete spustiť *inicializačný program (bootstrap program).* Tento program má za úlohu inicializovať systém po každej stránke - počnúc registrami procesora až po radiče zariadení. Inicializačný program musí vedieť zaviesť jadro operačného systému do pamäte a odštartovať ho.

Adresa inicializačného programu musí byť známa. Niektoré systémy ukladajú väčšiu časť inicializačného programu do pamäte ROM. Je to výhodné, pretože pamäť ROM nepotrebuje inicializáciu a je vždy pripravená. Problém môže byť v tom, že pri zmene inicializačného programu je potrebné zmeniť obvod ROM pamäte. Kvôli tomu väčšina systémov ukladá do ROM pamäti malú časť inicializačného programu a zvyšok sa ukladá do tzv. inicializačných (*boot)* blokov na známej adrese na disku. Disk, ktorý obsahuje takéto bloky, je nazývaný *systémový disk (boot disk).* Bez systémového disku operačný systém nemôže fungovať.

Kód, ktorý je zapísaný v ROM pamäti prenesie dáta z inicializačných blokov do pamäte a odštartuje kód. Program, ktorý je uložený v inicializačných blokoch je zložitejší ako ten, ktorý je uložený v ROM pamäti a je schopný natiahnuť operačný systém aj z iných adries, ktoré nie sú pevne zadané. Tento program zaberá veľmi málo pamäte. Napr. MS-DOS potrebuje pre svoj inicializačný program na disku len 512 bajtov (Obr. 13.19).



Obr. 13.19 Rozloženie informácií na disku v MS-DOS-e

13.3.3 Správa chybných blokov

Hlavy disku sa pohybujú veľmi tesne nad povrchom platní a táto časť zariadenia je veľmi náchylná na chyby. Chyby, ktoré sa vyskytujú môžu byť veľmi vážne a vynútia si výmenu celého média, alebo menšie, ktoré vyradia z použitia jeden alebo viacej blokov. Tieto bloky sa nazývajú chybné (bad) bloky. Často disky prichádzajú ešte od výrobcu s chybnými blokmi.

Na IBM PC kompatibilných počítačoch so zariadeniami s IDE rozhraním, sa chybné bloky odhalia počas formátovania disku. Do FAT tabuľky sa v príslušných položkách zapíše špeciálna hodnota, ktorá upozorní, že blok je chybný a nemá sa používať pre prideľovanie súborom. Ak sa blok stane chybným počas normálnej operácie, musí sa spustiť špeciálny program, ktorý ho lokalizuje a vyradí ho z použitia. Samozrejme dáta, ktoré boli na chybnom bloku sú stratené.

Disky, ktoré sú pripojené pomocou rozhrania SCSI, sa pri obsluhe chybných blokov správajú inteligentnejšie. Radič uchováva na disku zoznam chybných blokov, ktorý sa inicializuje počas fyzického formátovania disku a potom sa stále aktualizuje. Radič dáva stranou aj fond voľných sektorov, ktoré necháva k dispozícii operačnému systému. Každý chybný sektor sa potom môže nahradiť sektorom z tohoto fondu. Pri tomto nahradzovaní sa môže stať, že optimalizácia dosiahnutá správnym výberom algoritmu pohybu ramienka sa stratí. Je to možné napr. v prípade, kedy sa náhradné sektory nachádzajú na začiatku, alebo na konci disku. Tento problém sa niekedy rieši tak, že náhradné sektory sú v každom cylindri. Iné riešenie je umiestniť plánovací algoritmus pre pohyb ramienka do radiča, ktorý pozná umiestnenie chybných blokov na disku.

Oprava chybných blokov sa nerobí plne automaticky, pretože v prípade straty dát sa súbor musí opraviť, čo niekedy vyžaduje účasť používateľa.

13.4 Správa diskového priestoru určeného pre výmenu

Správa diskového priestoru, určeného pre výmenu (swap space) je ďalšou z úloh operačného systému na základnej úrovni. Jednotlivé operačné systémy využívajú diskový priestor pre výmenu odlišným spôsobom. Napr. systémy, ktoré implementujú výmenu, ukladajú na disk obrazy celých procesov, vrátane kódu a dát. Systémy so stránkovaním ukladajú do priestoru na výmenu stránky, ktoré boli nahradené v pamäti. Veľkosť diskového priestoru, ktorý je určený na výmenu, závisí od spôsobu jeho použitia. Na personálnych počítačoch stačí niekoľko megabajtov a na pracovných staniciach a minipočítačoch sú potrebné desiatky až stovky megabajtov.

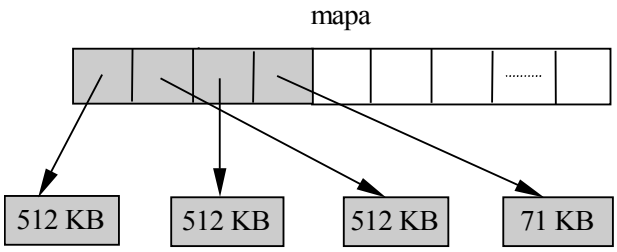
Pri určovaní veľkosti diskového priestoru je bezpečnejšie zvoliť väčšiu hodnotu ako menšiu, pretože ak systému nestačí priestor pre výmenu, môže to zapríčiniť ukončenie procesov alebo úplné spadnutie systému. Väčší priestor pre výmenu ukrojí z diskového priestoru pre súborový systém, ale iný záporný efekt sa nevyskytne.

Diskový priestor pre výmenu môže byť umiestnený *spolu so súborovým systémom* alebo môže mať samostatnú časť - partition. Výmenný priestor môže byť jednoducho väčší súbor, s ktorým sa narába ako s každým iným súborom. Je to jednoduchá implementácia, ale neefektívna. Pri práci s takto pojatým výmenným priestorom sa stráca čas pre lokalizáciu tohoto súboru v adresári, fragmentácia tiež môže spomaliť operácie s ním. Pre zefektívnenie práce s výmenným súborom sa dá použiť rýchla vyrovnávacia pamäť, kde sa uloží informácia o umiestnení blokov.

Častejšie sa výmenný priestor vytvára ako *samostatná časť - partition*. V tejto časti sa nevytvára súborový systém. Pre prideľovanie a uvoľňovanie blokov sa používa iný algoritmus ako pre bežný súborový systém. Tento algoritmus je zameraný na dosiahnutie maximálnej rýchlosti, pričom fragmentácia nie je tak sledovaná, pretože vo výmennom priestore sa dáta veľmi rýchlo vymieňajú a veľmi často sa k nim pristupuje.

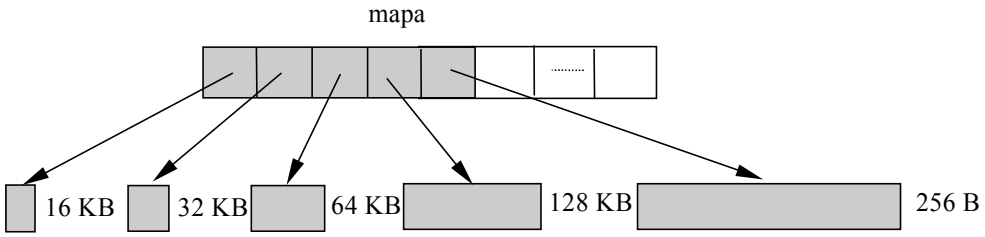
Využitie výmenného priestoru sa vyvíjalo postupne, pričom stále bola snaha zrýchliť operácie a nevykonávať ich dvakrát, ak to nie je potrebné. Jedna z metód zrýchlenia je, že pri spustení procesu sa stránky kódového segmentu načítavajú zo súborového systému priamo do pamäte. Po odsunutí z pamäte sa už zapisujú do výmenného priestoru. Procesy, ktoré zdieľajú stránky v pamäti, ich zdieľajú aj vo výmennom priestore. To znamená, že ak sa odštartujú dva procesy a prvý z nich už má stránky vo výmennom priestore, zdieľané stránky pre druhý proces sa zoberú rovno z výmenného priestoru.

Pre každý proces sa používajú dve mapy, ktoré sledujú prideľovanie blokov procesu - jedna pre textový segment a jedná pre dátový.



Obr. 13.21 Mapa textového segmentu v systéme 4.3 BSD

Textový segment sa nemení, takže sa mu prideľuje priestor po úsekoch po 512 KB, okrem posledného úseku, ktorému sa prideľujú úseky po 1 KB (Obr. 13.21).



Obr. 13.23 Mapa dátového segmentu v systéme 4.3BSD

Mapa dátového segmentu je zložitejšia), pretože dáta sa môžu časom rozrásť. Samotná mapa je pevnej dĺžky, ale obsahuje adresy blokov rôznych veľkostí (Obr. 13.23). Blok mapy s indexom *i* má veľkosť 2^i *x 16 KB*, maximálne však 2 megabajty.

14 POČÍTAČOVÁ BEZPEČNOSŤ

Aktuálnou témou v ostatnom čase je bezpečnosť počítačových systémov. Prvé počítače boli prístupné len z konzoly resp. z pevne pripojených terminálov a prístup k nim mali len oprávnené osoby. Preto i pri návrhu prvých operačných systémov sa nekládol dôraz na hľadisko bezpečnosti. Až s rozšírením počítačov, vznikom lokálnych sietí a Internetu sa ukázala potreba zahrnutia bezpečnostných mechanizmov do operačného systému počítačov.

14.1 Bezpečnostné hrozby

Pre porozumenie typom bezpečnostných hrozieb musíme najprv formulovať požiadavky na bezpečnosť:

Utajenie: požadujeme, aby informácie uložené v počítačovom systéme boli dostupné len oprávneným osobám. Dostupnosťou budeme rozumieť možnosť zobrazenia, tlače, resp. iného spôsobu odhalenia včítane zistenia ich existencie.

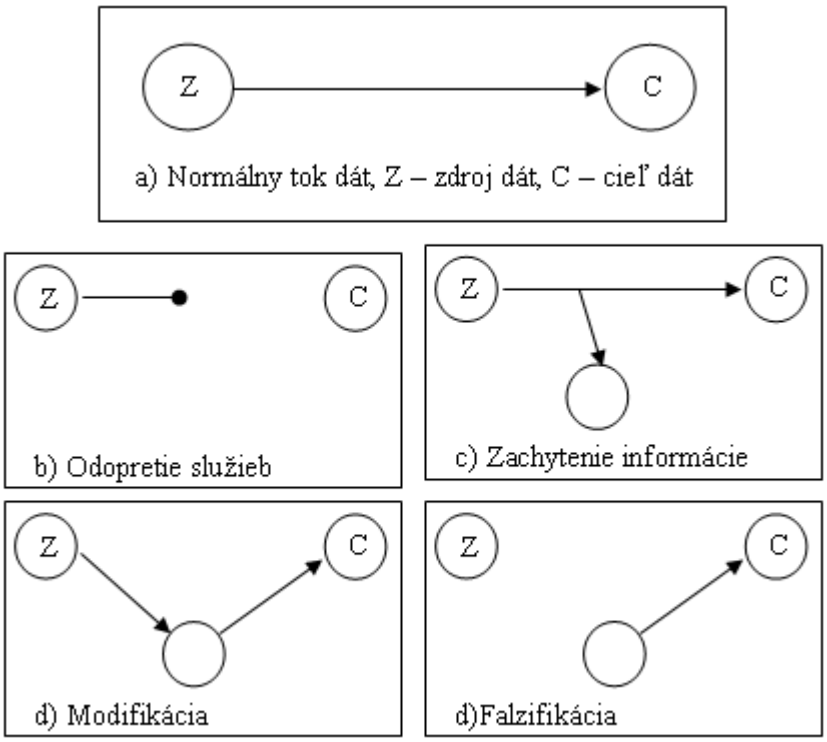
Integrita: znamená podmienku možnosti modifikácie informácií iba oprávnenými osobami. Modifikácia zahŕňa zápis, zmenu, zmazanie a vytvorenie.

Dostupnosť: požadujeme, aby počítačový systém bol dostupný oprávneným používateľom.

Autenticita: znamená že počítačový systém je schopný overiť identitu používateľa.

14.1.1 Typy bezpečnostných hrozieb

Pri charakterizovaní bezpečnostných hrozieb budeme vychádzať z funkcie počítačového systému ako poskytovateľa informácií. Normálne smeruje tok dát od zdroja k cieľu. Bezpečnostná hrozba znamená porušenie aspoň jednej z vyššie uvedených požiadaviek, čím sa tento normálny tok dát poruší. Podľa toho, ktorá z požiadaviek bola porušená môžeme rozdeliť bezpečnostné hrozby do štyroch typov (Obr. 14.1):



Obr. 14.1 Bezpečnostné hrozby

- **Odopretie služieb**: znamená útok na **dostupnosť.** Neoprávnený používateľ zničí alebo znefunkční niektorú súčasť počítačového systému. Môže byť spôsobený napr. zničením niektorej hardvérovej časti počítača, prerušením resp. zahltením komunikačnej linky, prípadne znefunkčnením súborového systému.
- **Zachytenie informácie**: znamená útok na **utajenie.** Neoprávnený používateľ získa prístup k súčasti počítačového systému. Môže sa jednať o odpočúvanie komunikačnej linky, resp. kopírovanie súborov.
- **Modifikácia**: znamená útok na **integritu.** Neoprávnený používateľ manipuluje so súčastami počítačového systému ku ktorým získal prístup. Môže ísť napr. o zmenu obsahu súborov, modifikáciu programov, resp. obsah správ prenášaných komunikačnou linkou.
- **Falzifikácia**: znamená útok na **autenticitu.** Neoprávnený používateľ podvrhne do počítačového systému falošné súčasti – napr. nežiadúce správy do komunikačnej linky, resp. súbory do súborového systému.

Treba poznamenať, že pod pojmom neoprávnený používateľ budeme rozumieť nielen osobu ale i počítačový program, ktorý vykonáva spomínanú činnosť.

14.1.2 Súčasti počítačového systému

Súčasťami počítačového systému budeme rozumieť hardvér, softvér, informácie uložené v počítačovom systéme a komunikačné linky a počítačové siete. V ďalších častiach sa budeme venovať bezpečnostným hrozbám pre jednotlivé súčasti počítačového systému.

1.1 Hardvér

Hlavnou hrozbou pre hardvér počítačového systému je porušenie dostupnosti. Jedná sa o neúmyselné, alebo úmyselné poškodenie hardvéru a jeho krádež. Tu je potrebné zabezpečiť fyzickú bezpečnosť napr. zamedzením prístupu neoprávnených osôb k počítačovému systému.

1.2 Softvér

Softvérom rozumieme operačný systém, pomocné a aplikačné programové vybavenie.

Kľúčovou hrozbou pre softvér je útok na dostupnosť. V mnohých operačných systémoch je veľmi jednoduché neúmyselne, resp. úmyselne znefunkčniť aplikačný program zmazaním niektorej jeho súčasti. Ďalšou hrozbou je modifikácia aplikačného programu útočníkom tak, aby vykonával pre neho nejakú činnosť. Mnohé počítačové vírusy napádajú aplikačné programy týmto spôsobom. Významnou je i hrozba porušenia utajenia, ktorá býva realizovaná neoprávneným kopírovaním aplikačných programov.

1.3 Informácie

Problematika bezpečnosti informácií uložených v počítačových systémoch je veľmi komplexná a týkajú sa jej všetky vyššie spomínané hrozby.

Najcitlivejšie vnímanou je hrozba utajenia, keď neoprávnená osoba získa prístup k informáciám uloženým v počítačovom systéme. Z hľadiska operačného systému resp. služobného programu je však významná i hrozba modifikácie resp. falzifikácie, keď neoprávnená osoba podvrhne konfiguračné súbory pre služobné programy resp. operačný systém, čím modifikuje ich správanie.

1.4 Komunikačné linky a počítačové siete

So vznikom a rozvojom počítačových sietí vznikli aj nové bezpečnostné hrozby. Útoky na počítačové siete resp. komunikačné linky možno rozdeliť na pasívne a aktívne.

K pasívnym útokom patrí odpočúvanie, kedy útočník zachytáva obsah správ prenášaných po komunikačnej linke. Takýto útok je ťažko detekovateľný, pretože nedochádza k modifikácii prenášaných dát. Obranou je zakódovanie prenášaných správ, takže ich odpočúvanie nestačí, útočník musí byť schopný správu i dešifrovať. Takéto typy útokov sú založené na analýze prevádzky na komunikačnej linke.

Pri aktívnych útokoch je obsah správ prenášaných po komunikačnej linke modifikovaný, resp. zachytený a neskôr použitý na získanie neoprávneného prístupu k súčasti počítačového systému. Príkladom aktívneho útoku je aj útok odopretia služieb, kedy útočník zabráni resp. potlačí normálne použitie resp. ovládanie komunikačnej linky. Cieľom útoku môže byť špecifická služba, napr. prístup k autorizačnému serveru a pod.

Zabrániť aktívnym útokom znamená zabezpečiť absolútnu fyzickú ochranu všetkých komunikačných prostriedkov, čo môže byť v praxi ťažko dosiahnuteľné až nemožné. Preto sa kladie veľký dôraz na ich detekciu a možnosť sa z nich zotaviť.

14.2 Ochrana

Zavedenie viacužívateľských operačných systémov umožnilo efektívnejšie využívanie prostriedkov počítača medzi ktoré patria:

- procesor,
- operačná pamäť,
- programy,
- dáta.

Možnosť zdieľania prostriedkov však vedie k možnému vzniku bezpečnostných hrozieb a tým aj k potrebe vzniku ochranných bezpečnostných mechanizmov. V rôznych časoch rôzne operačné systémy poskytovali rôzne stupne ochrany častí počítačového systému:

- žiadna ochrana: akceptovateľné len keď citlivé úlohy sú spúšťané oddelene,
- izolácia: každý proces je spracovávaný oddelene od ostatných procesov, má svoj adresový priestor, súbory a ostatné časti,
- plné zdieľanie alebo žiadne zdieľanie: vlastník časti určí, či bude súkromný alebo verejný, v prvom prípade nie je prístup z ostatných procesov povolený,
- zdieľanie obmedzením prístupu: operačný systém kontroluje prístup pre každú časť počítačového systému a používateľa zvlášť,
- dynamické určovanie prístupu: rozširuje možnosti zdieľania obmedzením prístupu o dynamické prideľovanie prístupových práv,
- limitované použitie: umožňuje určiť nielen používateľov ktorí majú právo k časti pristupovať, ale obmedziť aj spôsob prístupu (napr. možnosť čítať dokument, ale nie vytlačiť, resp. modifikovať ho a pod.).

14.2.1 Ochrana operačnej pamäte

Vo viacužívateľských operačných systémoch je ochrana operačnej pamäte veľmi dôležitá. Prístup do pamäte iného procesu totiž neznamená len možnosť narušenia utajenia, ale i integrity, dostupnosti a autenticity.

Oddelenie adresných priestorov procesov je jednoducho realizovateľné pomocou mechanizmu virtuálnej pamäte. Ak je požadovaná izolácia, potom operačný systém musí iba zabezpečiť, že každá stránka je prístupná len jednému procesu. Pri možnosti zdieľania dát operačný systém môže kontrolovať prístupové práva procesu k zdieľaným dátam.

Mechanizmus virtuálnej pamäte musí byť podporovaný príslušnou hardvérovou platformou.

14.2.2 Riadenie prístupu podľa používateľa

Tento spôsob riadenia prístupu je založený na tom, že každý používateľ pred prácou preukáže svoju identitu. Najčastejšou technikou je preukázanie znalosti prihlasovacieho mena a k nemu prislúchajúceho hesla. Je známe, že tento spôsob je veľmi náchylný k bezpečnostným rizikám. Používatelia môžu heslo zabudnúť, prípadne ho nechtiac prezradiť nepovolanej osobe.

V distribuovanom prostredí je riadenie prístupu buď centralizované alebo decentralizované. Pri centralizovanom prístupe sa používateľ prihlási k nejakej autorizačnej autorite a získa prístup k častiam počítačového systému pre ktoré má oprávnenie.

Decentralizovaný prístup znamená, že je nutné sa ku každej časti prihlasovať samostatne prostredníctvom siete. V každom prípade je však dôležité zabezpečiť bezpečnosť pri prenose prihlasovacích údajov po sieti.

14.2.3 Riadenie prístupu podľa obsahu údajov

Používateľ získa prístup k počítaču a jeho programovému vybaveniu preukázaním sa správnym prihlasovacím menom a prislúchajúcim heslom. Mnohokrát je však potrebné riadiť prístup k údajom aj na úrovni aplikačných programov. Napríklad pracovníci personálneho oddelenia majú prístup k osobným údajom zamestnancov, ale iba niektorí majú prístup k informáciám o mzde.

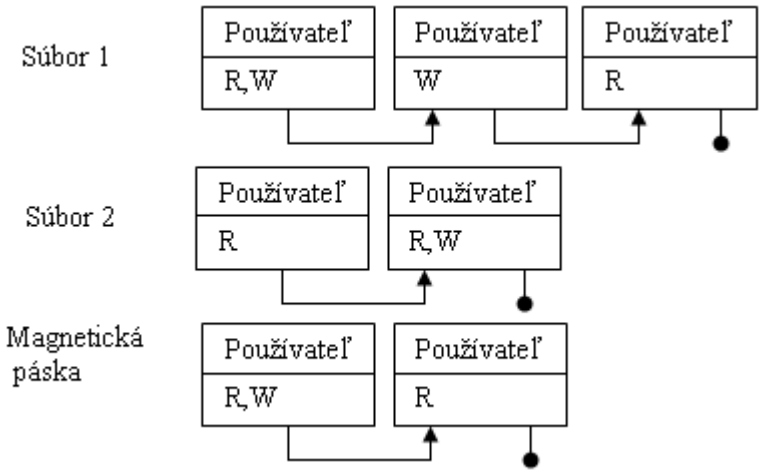
Všeobecný model riadenia prístupu možno popísať prístupovou maticou, ktorá obsahuje subjekty prístupu (entity ktoré pristupujú k objektom), objekty (entity ku ktorým je prístup riadený) a prístupové práva (ktoré určujú spôsob prístupu subjektu k objektu) (Obr. 14.2).

	Súbor 1	Súbor 2	Magnet. páska
Používateľ A	R W	R	
Používateľ B	 W		R W
Používateľ C	R	R W	R

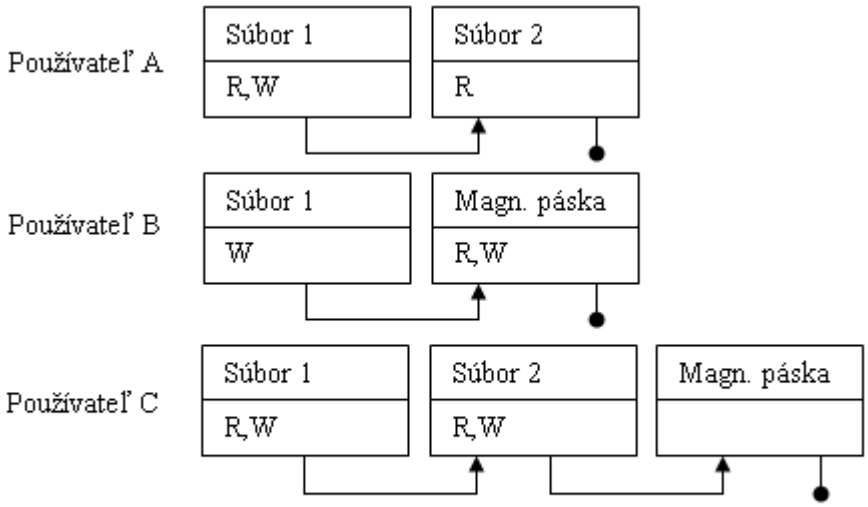
14.2 Prístupová matica, R znamená prístupové právo na čítanie, W prístupové právo pre zápis

V reálnych prípadoch sú prístupové matice riedke a preto sa v praxi členia podľa riadkov alebo stĺpcov.

Pri dekompozícii prístupovej matice podľa stĺpcov dostaneme zoznam prístupových práv pre jednotlivé časti počítačového systému (obr. 14.3, obr. 14.4). Pre každú časť sú definované prístupové práva užívateľov.



Obr. 14.3 Dekompozícia prístupovej matice podľa stĺpcov – zoznam prístupových práv



14.4 Dekompozícia prístupovej matice podľa riadkov – zoznam možností používateľa

14.3 Overenie používateľa

Jedným z problémov bezpečnosti počítačových systémov je identita používateľa. Vo všeobecnosti je možné identitu používateľa preukázať troma spôsobmi: vlastníctvom (používateľ vlastní kľúč, magnetickú kartu, a pod.), znalosťou (používateľ vie prihlasovacie meno a heslo) alebo vlastnosťou (identifikácia pomocou odtlačku prsta, podpisu, sietnice a pod.).

14.3.1 Identifikácia pomocou hesla

Je najčastejšie používaná metóda identifikácie používateľa. Je založená na identifikácii používateľa podľa znalosti. Ako bolo vyššie spomínané, jej nevýhodou je, že má mnohé slabiny. Medzi najznámejšie patrí možnosť zachytenia hesla prenášaného a nezakódovaného po počítačovej sieti, resp. možnosť odhaliť heslo hrubou silou pri získaní prístupu k súboru so zakódovanými heslami.

14.3.2 Jednorázové heslá

Nevýhody identifikácie pomocou znalosti hesla je možné odstrániť použitím jednorázových hesiel. Pri tomto spôsobe existujú heslá v pároch. Počítačový systém pri prihlasovaní náhodne zvolí jednu dvojicu hesiel a pošle používateľovi výzvu s jednou časťou hesla. Ten musí správne odpovedať zadaním druhej časti hesla.

Tento systém môže byť zovšeobecnený tak, že tajomstvom, ktoré zdieľajú počítačový systém a jeho používatelia, je algoritmus pomocou ktorého sa generuje odpoveď na výzvu systému. V takomto prípade počítač generuje náhodnú výzvu a odpoveďou je výsledok algoritmu ktorého vstupom je výzva.

Pokiaľ počítačový systém a používateľ zdieľajú spoločné tajomstvo (heslo), je možné postupovať nasledovne: počítač vygeneruje náhodnú výzvu ktorá je vlastne kľúčom pre zakódovanie hesla pomocou daného algoritmu. Výsledok algoritmu je prenesený do počítačového systému. Keďže ten pozná heslo, kľúč i použitý algoritmus, môže overiť, či je heslo platné aplikovaním algoritmu na vstupné údaje a porovnaním výsledku s tým ktorý poslal používateľ. Keďže výzva je náhodná, prípadné odchytenie komunikácie medzi používateľom a počítačovým systémom neumožní použiť zachytené informácie na neoprávnený prístup k systému.

14.3.3 Identifikácia pomocou biologických vlastností

V tomto prípade ide o zisťovanie niektorých biologických vlastností používateľa a ich porovnanie s údajmi uloženými v počítačovom systéme. Najčastejšie ide o snímanie odtlačkov prstov, ale je možné snímať i iné informácie, ktoré sú špecifické pre každého človeka.

Kombináciou tejto metódy s identifikáciou pomocou prihlasovacieho mena a hesla je možné dosiahnuť vysoký stupeň bezpečnosti.

14.4 Klasifikácia bezpečnosti počítačových systémov

Ministerstvo obrany USA definovalo **štyri kategórie bezpečnosti počítačových systémov A, B, C a D**. Najnižšiu úroveň zabezpečenia predstavuje kategória D ktorá zahŕňa všetky systémy, ktoré nespĺňajú kritériá vyšších kategórií. Do tejto kategórie patrili napr. operačné systémy MS DOS a Windows 3.1.

Systémy kategórie C poskytujú voliteľnú ochranu a dohľad nad používateľom a jeho činnosťou prostredníctvom záznamu jeho činnosti. V rámci kategórie C existujú **dve úrovne – C1 a C2**. Úroveň C1 poskytuje používateľom prostriedky pre ochranu prístupu k súkromným údajom. Väčšina klonov operačného systému UNIX patrí do tejto kategórie.

Množina všetkých prostriedkov slúžiacich pre bezpečnosť počítačového systému sa označuje pojmom Trusted Computer Base (TCB). TCB systému **úrovne C1** riadi prístup používateľov k súborom pomocou mechanizmu prideľovania prístupových práv jednotlivým používateľom alebo skupinám používateľov. Okrem toho sa musí každý používateľ pred začiatkom práce identifikovať pomocou nejakého ochranného mechanizmu alebo heslom. Jednou z úloh TCB je aj ochrana údajov pre overovanie používateľov daného systému.

Úroveň ochrany C2 pridáva možnosť riadenia prístupu na individuálnej úrovni. Správca systému má možnosť sledovať zvolenú činnosť každého používateľa systému, resp. skupiny používateľov. Ďalej TCB musí byť schopné sa chrániť pred modifikáciou jeho kódu alebo údajových štruktúr. Niektoré špeciálne verzie operačného systému UNIX boli certifikované do úrovne ochrany C2.

Kategória bezpečnosti B už poskytuje povinnú úroveň ochrany. K požiadavkám z kategórie C2 pridáva označenie každého objektu operačného systému stupňom zabezpečenia. Na **úrovni B1** je každému používateľovi systému možné individuálne povoliť pristupovať k objektom s istým stupňom zabezpečenia. TCB takisto označuje každý výstup čitateľný pre človeka na vrchu a spodku každej strany stupňom zabezpečenia. Na úrovni procesov je zabezpečená ochrana oddelením adresových priestorov.

Úroveň bezpečnosti B2 pridáva označovanie všetkých systémových zdrojov pomocou stupňa zabezpečenia. Fyzické zariadenia majú priradené minimálne a maximálne stupne zabezpečenia, ktoré systém používa na zabezpečenie bezpečnostných obmedzení uložených na jednotlivé fyzické zariadenia podľa ich fyzického umiestnenia.

Systémy na úrovni **bezpečnosti B3** majú možnosť udržiavať pre každý objekt zoznam používateľov a skupín, ktorí nemajú k nim prístup. TCB musí disponovať mechanizmom monitorovania všetkých udalostí v systéme a možnosťou detekcie porušenia bezpečnostných pravidiel. Tento mechanizmus informuje osobu poverenú dohľadom nad bezpečnosťou systému a ak je to potrebné, zruší neprípustnú operáciu.

Najvyšší stupeň úrovne zabezpečenia poskytujú systémy zaradené do **kategórie A**. Systémy úrovne bezpečnosti A1 sú funkčne zhodné so systémami kategórie B3, ale sú pre ne definované špecifické postupy pre tvorbu a kontrolu bezpečnostných pravidiel. Tým sa dosahuje vyšší stupeň kontroly správnosti implementácie TCB. Takéto systémy sú navrhované a realizované v preverených prevádzkach výlučne prevereným personálom.

14.5 Windows 2000

Ako príklad implementácie vyššie spomínaných princípov bezpečnosti si ukážeme implementáciu riadenia prístupu v operačnom systéme Windows 2000 (W2K), ktorá využíva objektovo-orientovanú technológiu.

W2K poskytuje jednotné riadenie prístupu pre procesy, vlákna, okná, prostriedky IPC a ostatné objekty operačného systému. Prístup je riadený dvoma entitami: prístupový token, ktorý je priradený ku každému procesu a bezpečnostný popisovač priradený ku každému objektu (v zmysle entity operačného systému), ktorý môže vstupovať do medziprocesovej komunikácie. Zarovnať

Pri prihlasovaní používateľa je po overení prihlasovacieho mena a hesla vytvorený proces a tomuto procesu je priradený prístupový token. Súčasťou tokenu je aj identifikátor používateľa (SID, security ID) ktorý ho identifikuje pre potreby bezpečnostného subsystému. Tento prístupový token dedia všetky procesy ktoré používateľ vytvorí.

Prístupový token slúži dvom účelom:

1. Udržuje všetky potrebné bezpečnostné informácie.
2. Umožňuje procesom modifikovať svoje bezpečnostné charakteristiky (kontrolovane) bez ovplyvnenia ostatných procesov pracujúcich pre toho istého používateľa.

Význam druhého bodu je nasledovný: prístupový token obsahuje zoznam práv ktoré má používateľ pridelené, ale pri vytvorení tokena sú deaktivované. Ak proces vytvorený používateľom potrebuje niektoré z prístupových práv, aktivuje ho a potom môže žiadať o službu spojenú s daným právom.

Každý objekt operačného systému má priradený bezpečnostný popisovač, ktorý obsahuje prístupové práva pre jednotlivých používateľov a skupiny používateľov operačného systému. Pri pokuse procesu o prístup k objektu sa porovná SID procesu z jeho tokenu s informáciou uloženou v bezpečnostnom popisovači a podľa toho je prístup k objektu buď povolený alebo zakázaný. Zarovnať

14.5.1 Prístupový token

Prístupový token obsahuje nasledujúce informácie:

- **Security ID:** jednoznačný identifikátor používateľa v rámci celej siete.
- **Group SIDs:** zoznam skupín do ktorých používateľ patrí.
- **Privileges:** zozname bezpečnostne citlivých služieb ktoré môže používateľ používať.
- **Default owner:** ak proces vytvorí nový objekt, táto položka určuje, kto bude určený ako vlastník objektu. Môže to byť SID procesu, ktorý objekt vytvára alebo SID jednej zo skupín do ktorej patrí vlastník procesu.
- **Default Access Control List (ACL):** zoznam prístupových práv pridelených štandardne každému objektu vytvorenému používateľom. Zarovnať

14.5.2 Bezpečnostný popisovač

Bezpečnostný popisovač obsahuje nasledujúce údaje:

Flags: určuje typ a obsah popisovača.

Owner: určuje vlastníka objektu ktorý má všetky práva nad objektom.

System Access Control List (SACL): určuje ktoré typy operácií nad objektom budú generovať správy do logovacieho súboru.

Discretionary ACL (DACL): určuje prístupové práva pre používateľov a skupiny.

Štruktúra ACL (Access Control List) obsahuje hlavičku pevnej dĺžky a premenlivý počet záznamov obsahujúcich SID a masku prístupových práv. Keď sa proces pokúša získať prístup k objektu, správca objektov najprv zistí z prístupového tokenu SID procesu a SID skupín do ktorých proces patrí. Potom prechádza postupne cez ACL a porovnáva SID uložené v DACL s tými získanými

z prístupového tokenu procesu. Ak nájde zhodu, proces má pre prístup k objektu práva určené príslušnou maskou. Ak sa pri prehľadávaní dôjde na koniec zoznamu, je prístup odoprený.

Maska prístupových práv je 32 bitová hodnota. Nižších 16 bitov je určených pre určenie prístupových práv, ktoré sú špecifické pre jednotlivé typy objektov (súbor, proces, semafor, súbor mapovaný do pamäte, a pod.). Vo vyšších 16 bitoch sa nachádzajú nasledujúce príznakové bity:

- *Synchronize*: právo synchronizovať vykonanie kódu s nejakou udalosťou spojenou s objektom. Takýto objekt môže byť použitý vo funkcii wait.
- *Write_owner*: dovoľuje procesu meniť vlastníka objektu.
- *Write_DAC*: dovoľuje procesu meniť DACL a teda bezpečnostné charakteristiky objektu.
- *Read_control*: dovoľuje procesu zisťovať vlastníka a obsah DACL daného objektu.
- *Delete*: proces môže vymazať objekt.
- *Generic_all*: proces má úplný prístup k ojektu.
- *Generic_execute*: proces môže spustiť vykonanie objektu ak je vykonateľný.
- *Generic_write*: proces môže zapisovať do objektu.
- *Generic_read*: proces môže čítať z objektu.

Okrem toho existujú ďalšie dva špeciálne príznaky. Prvým z nich je príznak Access_System_Security, ktorého nastavenie umožňuje procesu modifikovať kontrolné a výstražné mechanizmy objektu. Vzhľadom na možnosť zneužitia však musí byť príslušné právo nastavené i v bezpečnostnom tokene procesu, ktorý o tieto práva žiada.

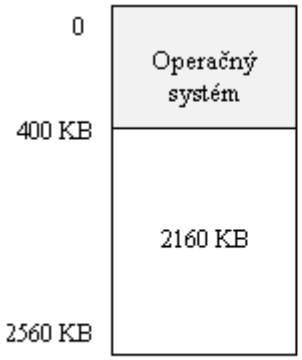
Posledným je príznak Maximum_Allowed, ktorý modifikuje algoritmus prehľadávania DACL. Predstavme si situáciu, keď aplikačný program potrebuje získať pre používateľa prístup k nejakému objektu, ale nemá dopredu špecifikovaný typ prístupu (teda či budeme požadovať iba čítanie z objektu, alebo aj jeho modifikáciu, a pod.). Aplikačný program môže použiť pri žiadosti o prístup k objektu nasledujúce stratégie:

1. Pokúsi sa požiadať o prístup so všetkými právami. Nevýhodou je, že prístup môže byť odmietnutý, hoci v skutočnosti by používateľ požadoval iba práva, ktoré sú aplikácii dostupné.
2. Aplikácia požiada o prístup až po tom ako používateľ špecifikuje požadovaný typ operácie. Nevýhodou je, že rôzne typy prístupu musia byť opakovane požadované, čím sa zvyšuje réžia prístupov.
3. Aplikácia môže požiadať o maximálnu možnú množinu práv. Týmto spôsobom sa odstránia nevýhody predchádzajúcich bodov.

Príznak Maximum_Allowed dovoľuje vlastníkovi objektu definovať masku maximálnych práv pre daného používateľa, čím je možné použiť tretí spôsob žiadosti o prístup k objektu.

Výhodou spomenutého bezpečnostného mechanizmu W2K je jeho transparentnosť a možnosť využitia aj v aplikačných programoch.

OBSAH



front pripravených procesov

proces	požadovaná pamäť	požadovaný čas
P_1	600 KB	10
P_2	1000 KB	5
P_3	300 KB	20
P_4	700 KB	8
P_5	500 KB	15

Obr. 9.7 Príklad plánovania