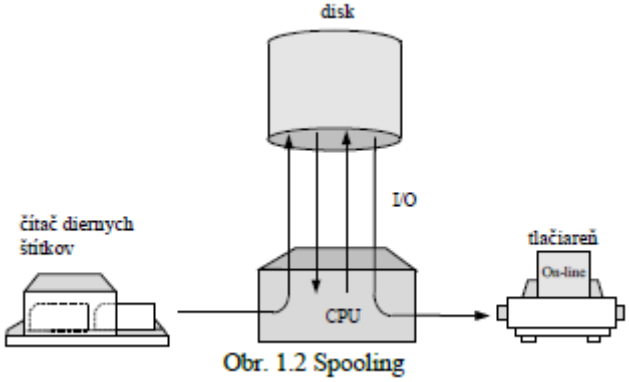
1. Ktoré zariadenie využíva spooling,

   Odpoveď: 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.

2. Posielanie správ v distribuovanom systéme,

   Odpoveď:
   Vid 23.
   RPI
   RMI
   SOKETY

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.

2.2 Volanie vzdialenej metódy (**R**emote **M**ethod **I**nvocation**)**

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.

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.

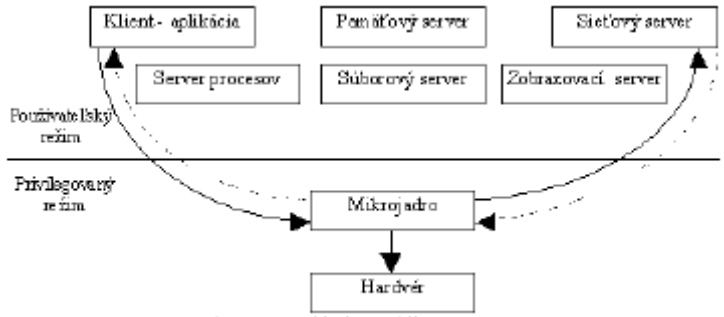
Procesory v distribuovanom systéme môžu byť rôzne podľa výkonnosti a funkcie. Môžu to byť personálne počítače, pracovné stanice, servery, minipočítače alebo veľké sálové počítače.

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

3. Popis architektúru: klient-server, vrstvovú arch. OS,

Odpoveď: 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.

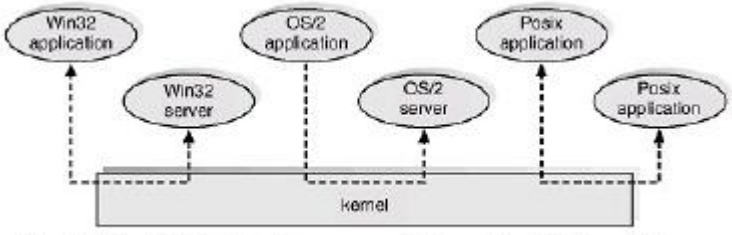


Obr. 3.9 Architektúra klient-server

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

4. Definujte proces,

Odpoveď:

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.

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.

5. Nakreslite stavový diagram procesu,

Odpoveď: 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 vykonávanie,

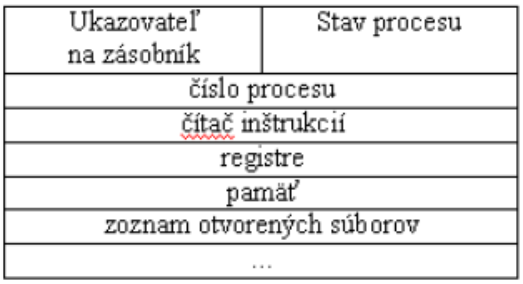
Obr. 4.1 Stavy procesu

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** Þ **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.

6. Riadiaci blok procesu (PCB),

Odpoveď: Každý proces je prezentovaný v operačnom systéme dátovou štruktúrou, ktorá sa nazýva riadiaci blok procesu (Process Control Block).



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

- ukazovateľ na zásobník procesu,
- stav procesu – nový, pripravený, bežiaci, čakajúci atď.
- hodnota čitača 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.

7. Akú úlohu rieši dlhodobý plánovač a akú krátkodobý plánovač,

Odpoveď: 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 častonemajú 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.

8. Popis vlákien,

Odpoveď: 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.

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.

9. Vlákna, porovnanie s procesmi, implementácia,

Odpoveď: Viď predošlá úloha,

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).

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 procesodštartovaný, 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.

Modelové situácie použitia vlákien:
• Pán/ Otrok,
• Člen skupiny,
• Postupný model,

10. Napíš 2 spôsoby implementácie vlákna,
Odpoveď: Viď predošlá úloha,

11. Popíšte algoritmy plánovania procesov FCFS a RR, ich výhody a nedostatky,
Odpoveď: **P**lánovanie času procesora patrí do základných funkcií operačného systému. Prideľovaním procesora jednotlivým procesom sa práca celého počítačového systému zefektívni.

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.

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.

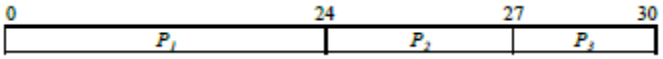
Spracovanie v poradí príchodu (FCFS - First Come, First Served)

Najjednoduchší z algoritmov plánovania je algoritmus spracovania v poradí príchodu. Podľa tohto 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 *P1, P2, P3* a sú obsluhované v poradí FCFS, potom získame tento diagram:



Pre časy čakania získame nasledujúce hodnoty. Proces P1 nebude čakať, P2 bude čakať 24 ms, P3 bude čakať 27 ms. Takže priemerná doba čakania je (0 + 24 + 27)/3=17 ms. Ak procesy prídu v poradí P2, P3, P1, potom výsledok pre čas čakania bude (6 + 0 + 3)/3=3 ms. 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.

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.

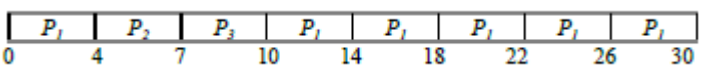
Plánovač vyberie a spustí ďalší z pripravených procesov. Ak proces potrebuje čas dlhší ako je časové kvantum, po uplynutí kvanta časovač spôsobí prerušenie. Zapamätá sa kontext procesu a proces sa uloží na koniec frontu, z ktorého sa vyberie ďalší pripravený proces.

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 P1, pretože tento proces potrebuje ďalších 20 ms, bude prerušený po uplynutí q. Potom sa spustí P2, ale tento proces nespotrebuje celé časové kvantum. Ďalej sa bude spracovávať proces P3. Keď každý z procesov dostane 1 časové kvantum, príde na rad znova proces P1. Výsledné poradie vykonávania bude nasledujúce :



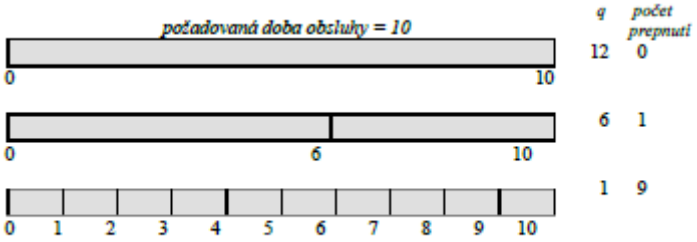
Priemerná doba čakania je 17/3 = 5.66 ms.

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 1q. 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.

12. Popíš cyklické plánovanie (Round-Robin),
    Odpoveď: Vid predošlá úloha

13. Prečo je dôležitá synchronizácia procesov, vysvetliť aké prostriedky existujú číže aktívnym a pasívnym čakaním, vymenovať ich, a vybrať si jeden a ten popísať,
    Odpoveď: 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.

    Synchronizácia aktívnym čakaním

    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 *P0* a *P1*. Pre jednoduchosť budeme prezentovať algoritmus pre proces *Pi* a pomocou *Pj* budeme označovať druhý proces.

    //Algoritmus č1, Algoritmus č2, Algoritmus č3, Dekkerov algoritmus, Algoritmus pekára,
    //Inštrukcia Test-and-Test, Inštrukcia Swap,

    Synchronizácia pasívnym čakaním

    //Semafory, Monitory,

    Podrobný popis niektorých prostriedkov sa nachádza v ďalších 5tich otázkach.

14. Algoritmus pekára,
    Odpoveď:

    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 Pi a Pj dostanú rovnaké čísla a i<j, potom ako prvý sa obslúži Pi. 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 $k \geq a_i$ 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;
```

15. Hardvérové prostriedky pre synchronizáciu aktívnym čakaním,

Odpoveď:

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.

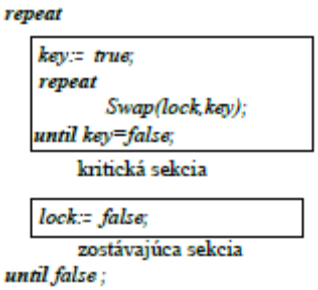
Mnoho procesorov ponúka ***špeciálne inštrukcie***, ktoré dovoľujú používateľovi testovať a modifikovať obsah premennej alebo vymieňať atomicky obsah dvoch premenných. Tieto inštrukcie sú *atomické*, čo znamená, že sa vykonajú vždy ako celok. Ak sa vykonávajú paralelne dve takéto inštrukcie, vždy sa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

16. Synchronizácia cez SWAP,

Odpoveď: 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.



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.

17. Popíšte semafor – štruktúra, operácie, použitie,

Odpoveď:

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;

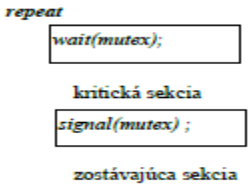
S:=S - 1;

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.

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.



Obr. 6.5 Vzájomné vylúčenie pomocou semaforov

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.

18. Hlavný princíp monitora,

Odpoveď: 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).

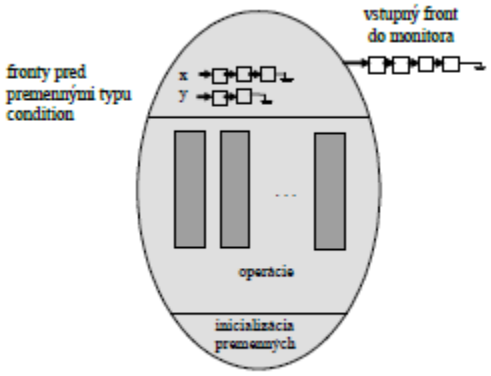
Táto konštrukcia ale nedovoľuje modelovať určité situácie, ktoré sa vyskytujú v synchronizačných úlohách. Vzniká potreba definovať dodatočný dátový typ, ktorý dovoľuje zablokovať proces v prípade, že nie je splnená určitá podmienka. Tento typ sa nazýva *condition* a jediné operácie nad ním sú *wait*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

19. Prostriedky na komunikáciu procesov,

Odpoveď:

Správa

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.

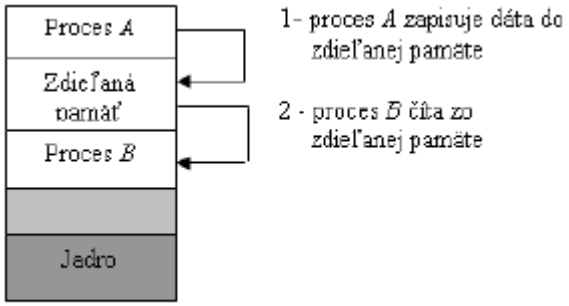
Správy môžu mať buď *pevnú* alebo *variabilnú* dĺžku. Ak dva procesy komunikujú, musí medzi nimi existovať komunikačná linka. Tu máme na mysli skôr logické vlastnosti a nie hardvérové. Základné otázky, na ktoré treba zodpovedať, sú: ako sa nadviaže spojenie, či sa môže k linke pripájať viac procesov, 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,

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 vysvetlené v kapitole o synchronizácií procesov. Komunikačný model zdieľanej pamäte je uvedený na Obr. 7.2.



Obr. 7.2 Komunikačný model zdieľanej pamäte

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.

20. Komunikácia medzi procesmi. Jeden typ popísať aj do podrobná,

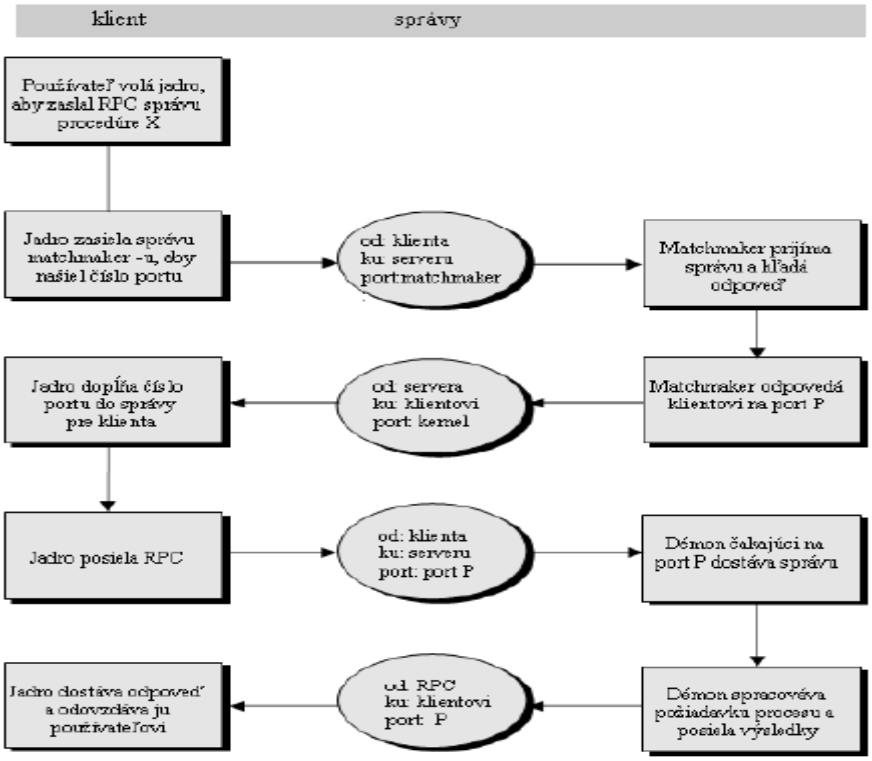
Odpoveď: Viď predošlá úloha.

21. Popis komunikáciu medzi procesmi na základe volania vzdialenej procedúry RPC,

Odpoveď:

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.

22. Coffmanove podmienky - nutné podmienky na uviaznutie,

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

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 *P0 , P1 ,..., Pn* čakajúcich procesov takých, že *P0* čaká na prostriedok, ktorý drží *P1* , *P1* čaká na prostriedok, ktorý drží *P2 ,..., Pn-1* čaká na prostriedok, ktorý drží *Pn* a *Pn* čaká na prostriedok, ktorý drží *P0* . Pre vznik uviaznutia musia platiť všetky štyri podmienky súčasne.

23. Bola zadaná úloha a matice z algoritmu Bankára. Určí, či je podľa algoritmu Bankára proces v bezpečnom stave,

Odpoveď: 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 *Rj* 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 *Pi* môže požadovať maximálne *k* jednotiek z prostriedkov typu *Rj*.

· ***pridelené***: matica *n* x *m* definuje počet prostriedkov každého typu, pridelených momentálne procesu *Pi* . Ak *pridelené*[*i,j*]= *k*, potom to znamená, že proces *Pi* má momentálne pridelených *k* jednotiek prostriedku typu *Rj*.

· ***zostáva***: matica *n* x *m* , ktorá označuje prostriedky, ktoré ešte musia byť pridelené procesu. Ak *zostáva*[*i,j*]= *k*, potom to znamená, že proces *Pi* potrebuje ešte *k* prostriedkov typu *Rj*, 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* £ *X*. *Y* < *X* ak *Y*£ *X* a *Y*¹*X*. Riadky matíc *pridelené* a *zostáva* môžeme brať ako vektory a budeme ich označovať *pridelenéi* a *zostávai* . Vektor *pridelenéi* špecifikuje všetky prostriedky pridelené procesu *Pi* a vektor *zostávai* špecifikuje všetky prostriedky, ktoré proces *Pi* ešte potrebuje dostať do svojho ukončenia.

24. Fázy spracovania programu, kedy je možné mu priradiť pamäťové adresy,

Odpoveď: 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).

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).

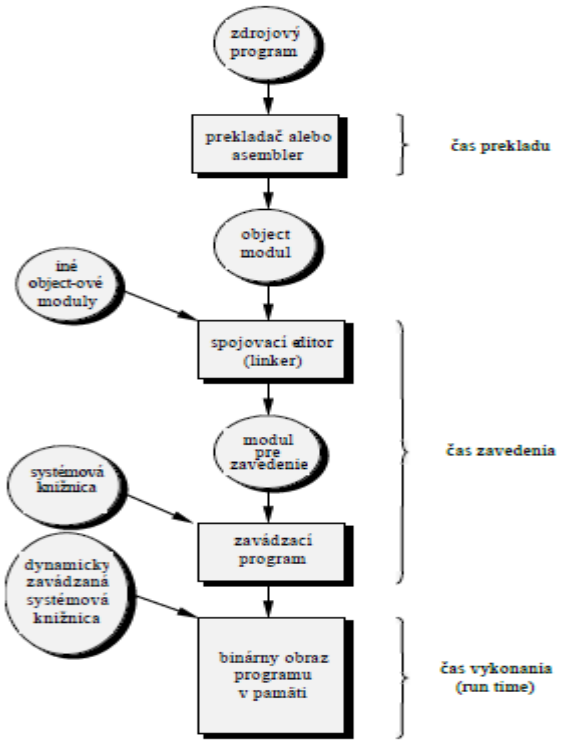
Adresy v zdrojovom programe sú obyčajne symbolické. Prekladač spája tieto symbolické adresy s relokovateľnými adresami (ako napr. „14 bajtov od začiatku tohoto modulu“). Spojovací editor alebo zavádzací program (loader) premení tieto relokovateľné adresy na absolútne (napr. 74014). Každé pripojenie adries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

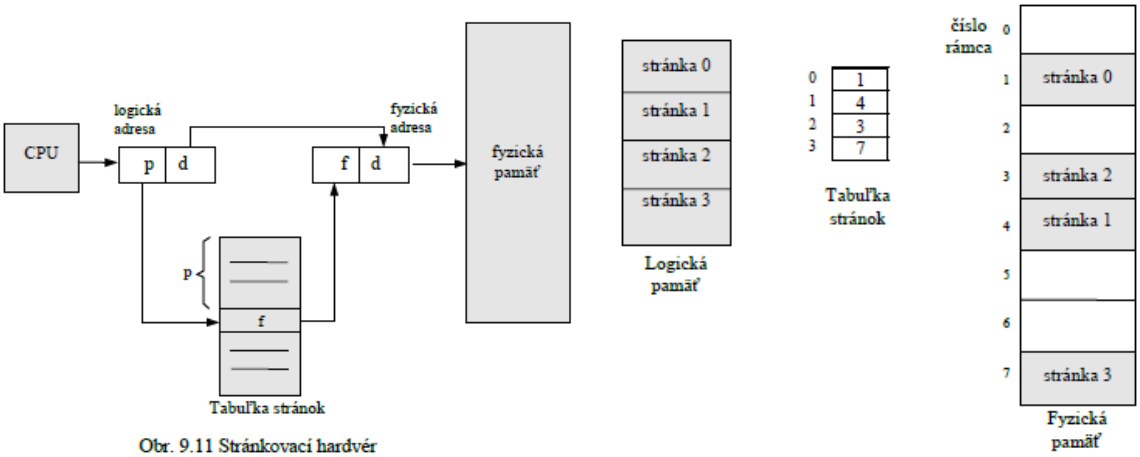
25. Stránkovanie,

Odpoveď: 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.

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.

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.



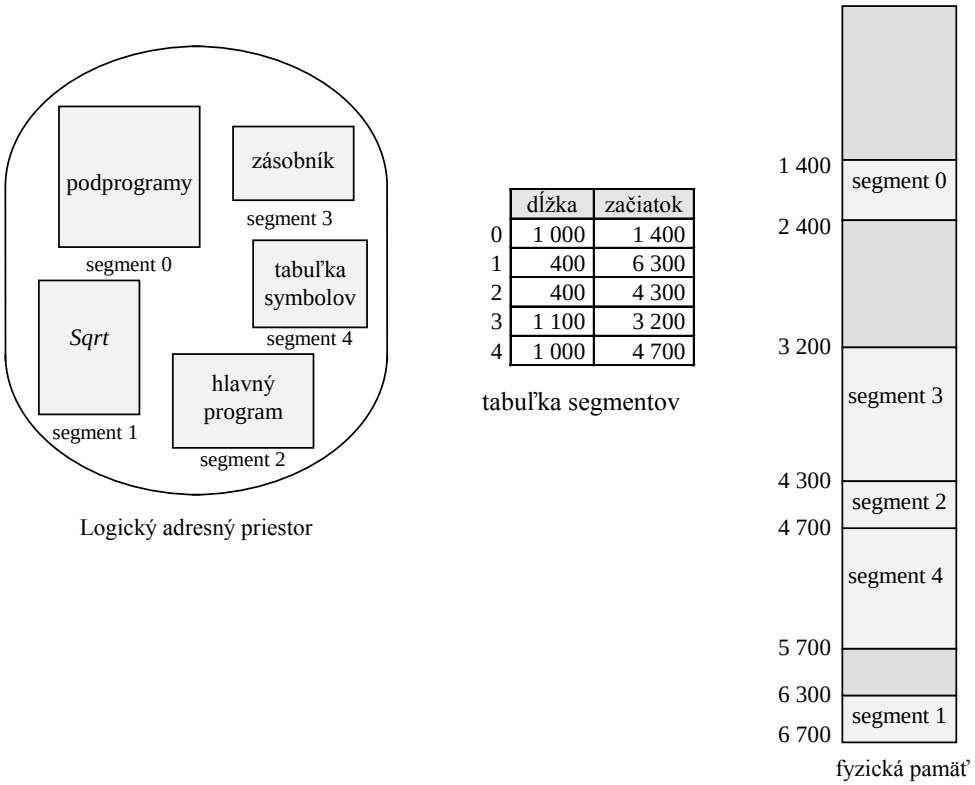
Obr. 9.11 Stránkovací hardvér

26. Popíšte transformáciu logickej adresy na fyzickú pri použití segmentácie,

Odpoveď: 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.

Užívateľ sa môže odkazovať na objekty dvojrozmernou adresou (číslo segmentu, posuv), ale skutočná fyzická pamäť je stále jednorozmerná postupnosť bajtov. Takže musíme mapovať dvojrozmernú užívateľskú 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.

Na obr.8.23 je ukázaná situácia, kedy máme 5 segmentov. V tabuľke segmentov je položka pre každý segment, kde je zaznamenáva 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.



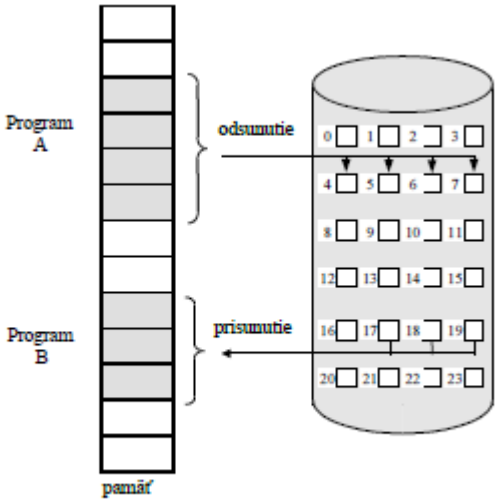
Obr. 8.23 Príklad segmentácie

27. Virtuálna pamäť - stránkovanie na požiadanie

Odpoveď: *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.

Stránkovanie na žiadosť

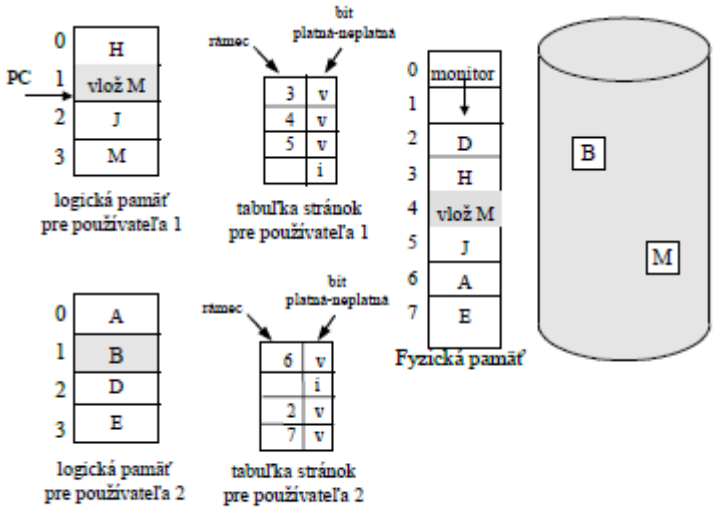
Stránkovanie na žiadosť je podobné systému stránkovania so swapovaním (Obr. 10.2). Procesy sú umiestnené na disku. Keď chceme vykonávať proces, presunieme ho do pamäte. Avšak miesto presunutia celého procesu do pamäte presunieme len jeho časť. Používame tzv. „lenivý“ swapper, ktorý nikdy nepresúva stránku 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

28. Popíš nahradzovanie stránok pri stránkovaní na žiadosť,

Odpoveď: 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.



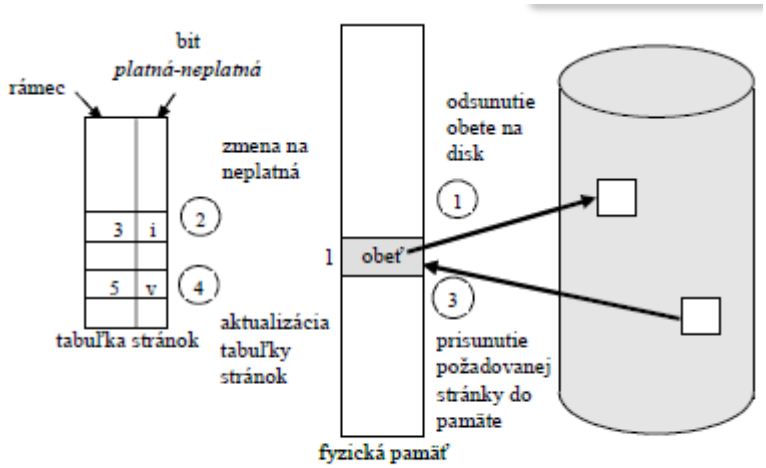
Obr. 10.5 Ukážka potreby nahradzovania stránok v pamäti

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.



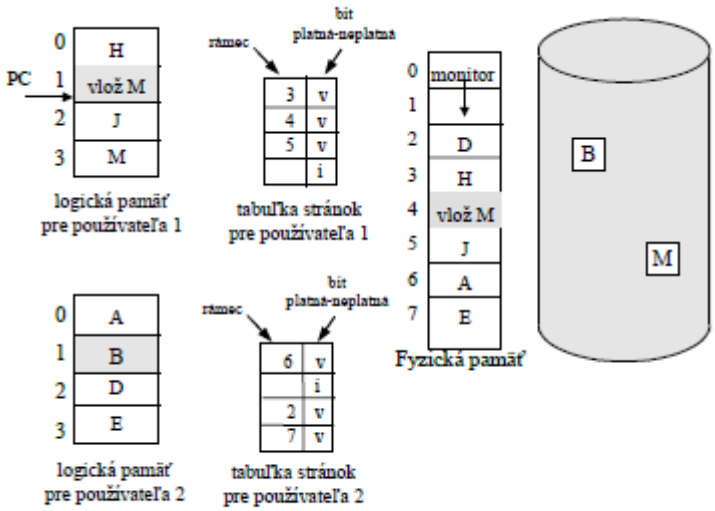
Obr. 10.6 Kroky pri nahradzovaní stránok

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.

29. Nahradzovaní algoritmus stránok v pamäti FIFO,

Odpoveď: 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. Potomstránkovanie 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

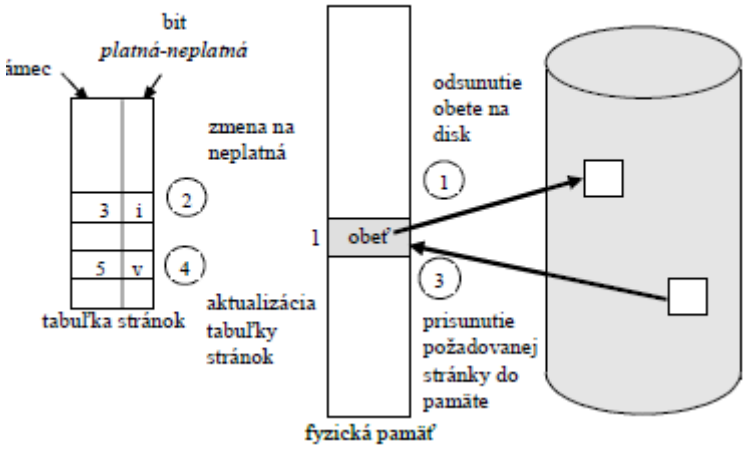


Obr. 10.5 Ukážka potreby nahradzovania stránok v pamäti

Ďalšia možnosť je odsunúť proces na disk, čím sa zníži úroveň multiprogramovania a uvoľniasa 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 rámcov4. Reštartuje používateľský proces.



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.

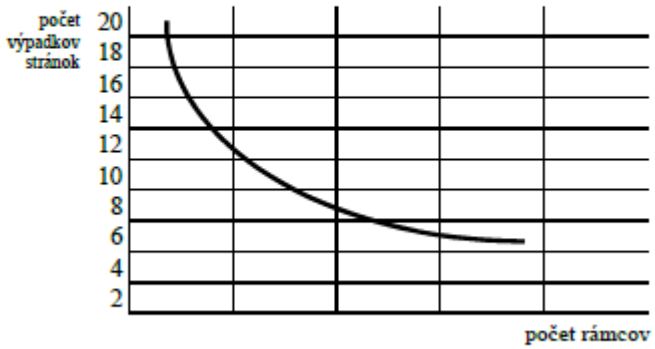
Nahradzovanie stránok je základ techniky stránkovania na žiadosť. Pri implementácii tejto techniky je potrebné vyriešiť dva hlavné problémy: prideľovanie voľných rámcov a nahradzovaniestránok. Návrh algoritmov pre vyriešenie týchto problémov je veľmi dôležitou otázkou a môže značne ovplyvniť výkon celého systému.

30. Algoritmy nahradzovanie stránok,

Odpoveď: 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

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

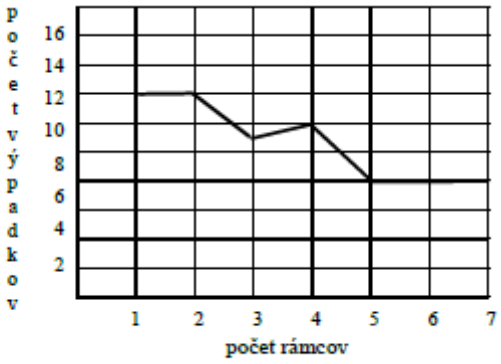
celkom 15 výpadkov

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

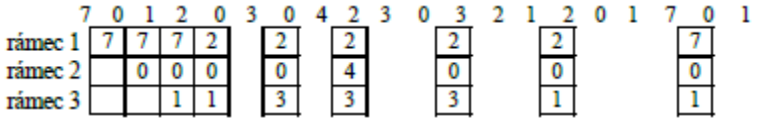
Optimálny algoritmus

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

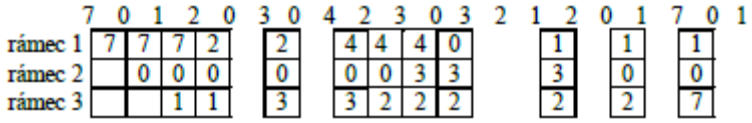
Obr. 10.10 Optimálny nahradzovací algoritmus stránok

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.

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).



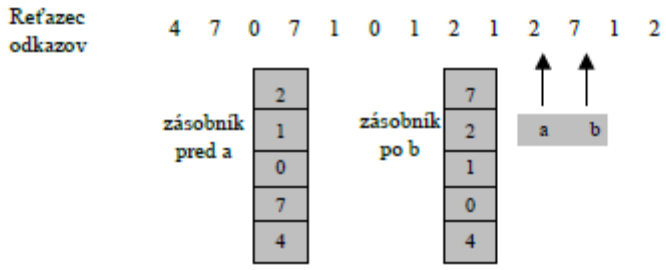
celkom 12 výpadkov

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.

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.

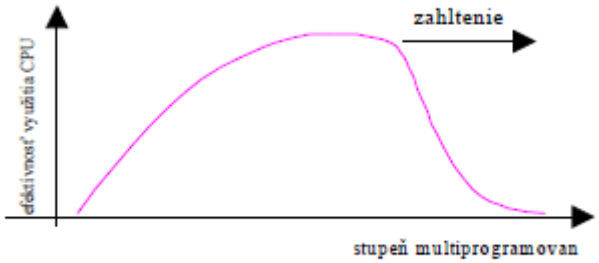
31. Nahradzovací algoritmus stránok v pamäti FIFO,
    Odpoveď: Viď predošlá otázky,

32. Bola daná tabuľka procesov a hodnoty: čas začiatku procesu a posledného použitia. Pri zavádzaní nového procesu, ktorý proces bude uvoľnený z pamäte podľa algoritmu,
    a) Algoritmus FIFO,
    b) LRU,
    c) Modifikovaný 2. Šance
    Odpoveď: Viď otázka č.24

33. Čo je zahltenie, kedy vzniká a ako sa mu predchádza,
    Odpoveď: 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.

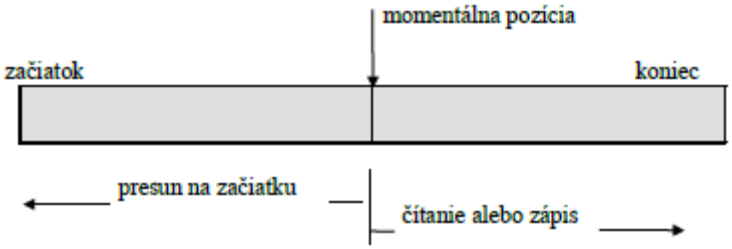
34. Ktoré metódy prístupu k súborom poznáte,

Odpoveď: 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.

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

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):

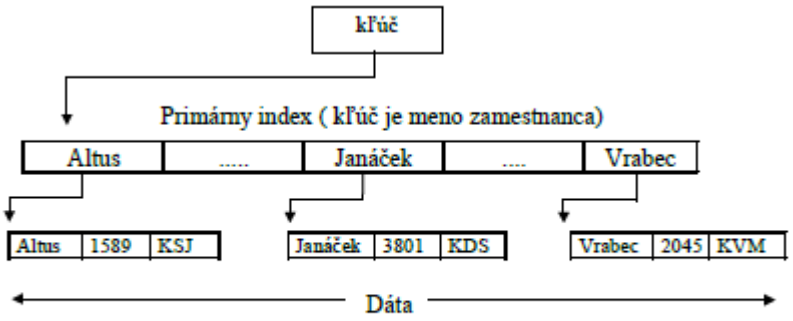
Sekvenčný prístup	Implementácia priamym prístupom
reset	*cp:=0;*
read next	***read*** *cp;*
	cp:=cp+1;
write next	***write*** *cp;*
	cp:=cp+1;

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.

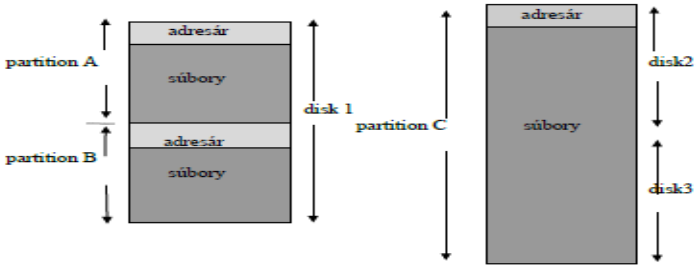


Obr. 11.5 Organizácia prístupu s indexom v systéme VMS

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.

35. Štruktúra adresárov,

Odpoveď: 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.

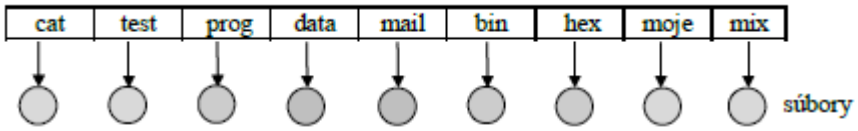
· **Tvorba súboru** - adresár musí umožňovať pridanie novej položky pri tvorbe nového súboru.· **Zrušenie súboru** - keď súbor nie je viac potrebný, musíme mať možnosť odstrániť ho z adresára.

· **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.

Jednoúrovňový adresár

Jednoúrovňový adresár má najjednoduchšiu štruktúru. Všetky súbory sú v jednom adresári - viď Obr. 11.7.

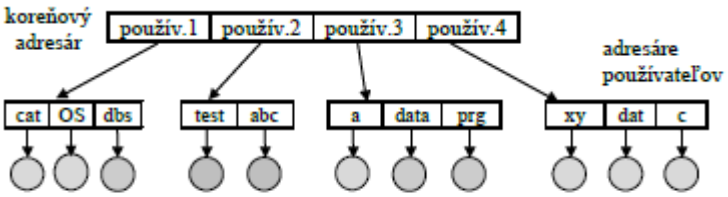


Obr. 11.7 Jednoúrovňový adresár

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.

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.

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.

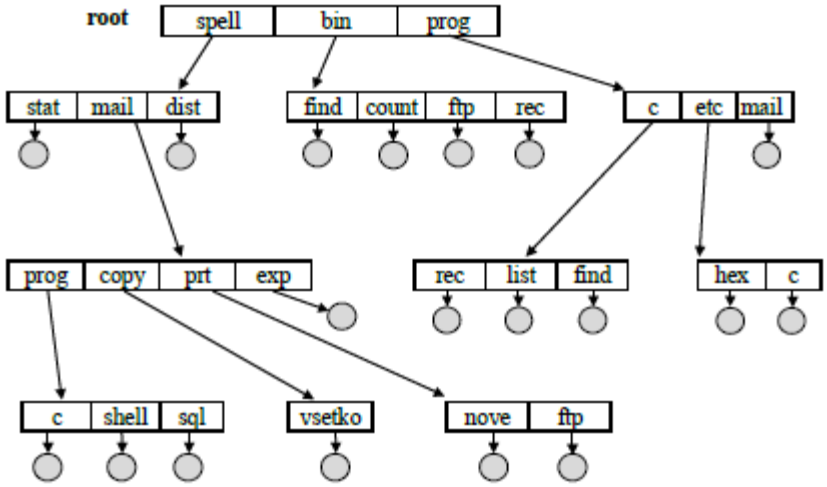
Pri odvolaní sa na súbor, sa prehľadáva najskôr , tzv. *pracovný adresár*, kde používateľ je nastavený. Ak sa tam hľadaný súbor nenájde, hľadá sa v adresároch v poradí, aké je uvedené v prehľadávacej ceste. Zmena adresára je možná systémovým volaním *change directory*.

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.

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.

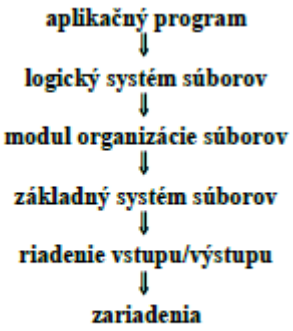
Štruktúra, ktorá dovoľuje tento spôsob zdieľania súborov sa nazýva *acyklická štruktúra* (Obr. 11.10). Táto štruktúra dovoľuje adresárom mať zdieľané podadresáre alebo súbory. Acyklická štruktúra grafu (t.j. neobsahuje cykly) je prirodzeným zobecnením stromovej štruktúry. V prípade, že viacej ľudí pracuje na spoločnom projekte je možné všetky zdieľané súbory dať dohromady do jedného adresára, ktorý bude zdieľaný. Každý z členov tímu 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.

36. Ako je organizovaný systém súborov? Aké sú úlohy jednotlivých vrstiev,

Odpoveď: 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.

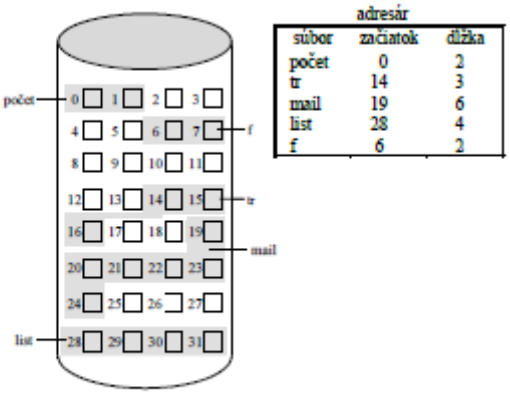
37. Ktoré metódy prideľovanie voľného priestoru súborom na disku poznáte,

Odpoveď: 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.

Súvislé prideľovanie

Pri súvislom prideľovaní diskového priestoru sa súbor ukladá postupne na susedných blokoch na disku. Diskové adresy nasledujú za sebou, čo znamená, že nie je potrebný pohyb hláv pre načítavanie ďalšieho bloku. Keď je potrebný pohyb (z posledného sektoru jedného cylindra na prvý sektor ďalšieho cylindra), ten pohyb je minimálny - len jedna stopa. To znamená, že čas pre hľadanie blokov súvisle uloženého súboru je minimálny.

Súvislé uloženie súboru je definované diskovou adresou prvého bloku a dĺžkou. Ak súbor je dlhý *n* blokov a začína na adrese *b*, potom celý súbor bude zaberať bloky *b, b + 1, b+2,...., b+n-1*. Položka súboru v adresári obsahuje adresu prvého bloku a dĺžku prideleného priestoru v blokoch (Obr. 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 bestfit, 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.

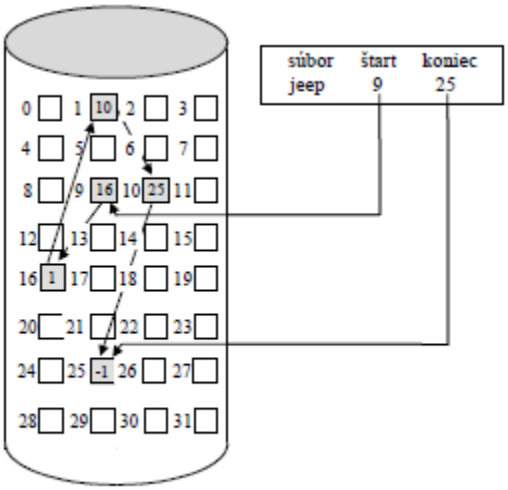
Problém pri rozširovaní súborov v prípade súvislého prideľovania sa rieši tak, že na začiatku sa súboru pridelí súvislý priestor a pri potrebe rozšírenia sa pridelí dodatočný súvislý úsek, ktorý nemusí nutne ležať za prvým. Tento úsek sa nazýva *rozšírenie (extent).* Bloky súboru sa potom zapisujú do 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.

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

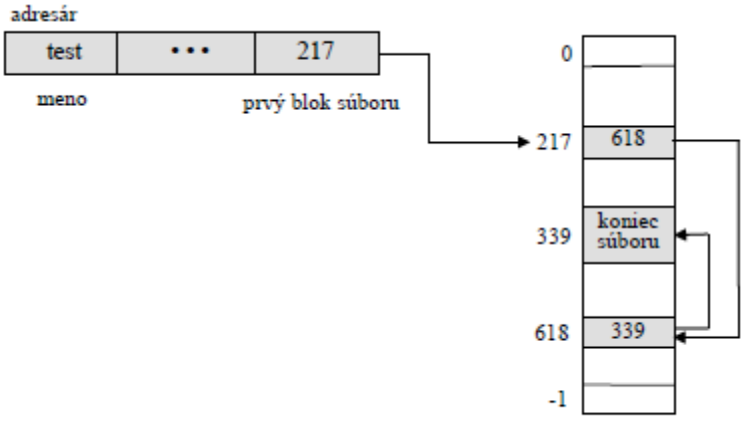
Problémom zreťazeného prideľovania je, že sa dá efektívne využiť len pri sekvenčnom prístupe k súboru. Ak potrebujeme sprístupniť *i-ty* blok, musíme začať vždy od začiatku súboru a blok po bloku sa dostať k *i-temu*. Každý prístup k ukazovateľu vyžaduje čítanie a väčšinou aj hľadanie adresy. Z toho 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

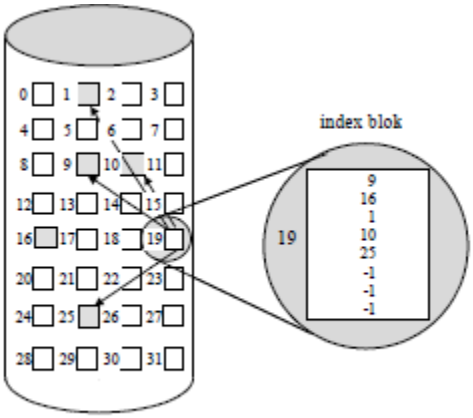
Tabuľka má jednu položku pre každý súbor a je indexovaná podľa čísel blokov. Tabuľka FAT sa používa ako zreťazený zoznam. Položka súboru v adresári obsahuje číslo jeho prvého bloku. Položka v tabuľke s týmto indexom obsahuje číslo ďalšieho bloku súboru atď. Tento reťazec pokračuje kým sa nepríde na koniec 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.

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.

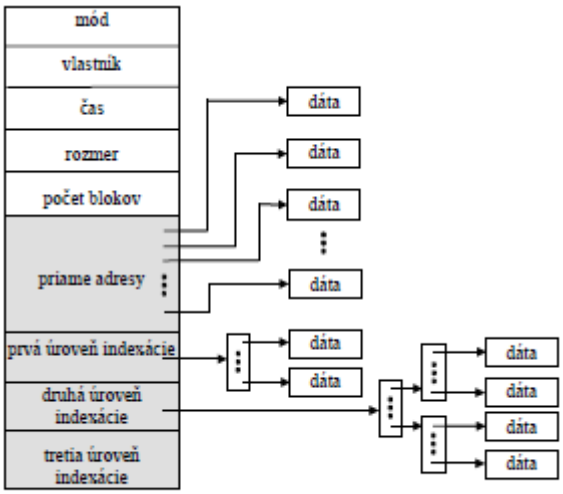
Indexované prideľovanie podporuje priamy prístup a netrpí vonkajšou fragmentáciou.

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 232 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

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.

38. Popíšte FAT tabuľku. K čomu slúži,
Odpoveď: Viď predošlá otázka

39. Čím sa zaoberá prideľovanie diskového priestoru,
Odpoveď: Viď úloha č. predpredošla

40. Aké sú spôsoby implementácie adresára (jeden popíšte),
Odpoveď: Najrozšírenejšie spôsoby implementácie adresára sú lineárny zoznam a rozptylová (hešovacia) tabuľka.

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.

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.

41. Algoritmy pohybu ramienka po disku,

Odpoveď: 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.

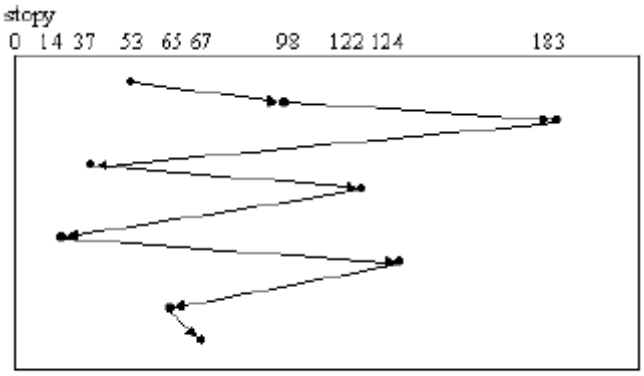
Požiadavky na prenos z/na disk sa radia do frontu. Obsluha jednotlivých požiadaviek znamená, že sa ramienko s hlavami musí nastaviť na požadovanú stopu, potom počkať na sektor a nakoniec preniesť dáta. Poradie obsluhy požiadaviek môže ovplyvniť celkový čas potrebný pre prenos, takže sa používajú rôzne algoritmy pre minimalizáciu času pohybu ramienka disku.

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.

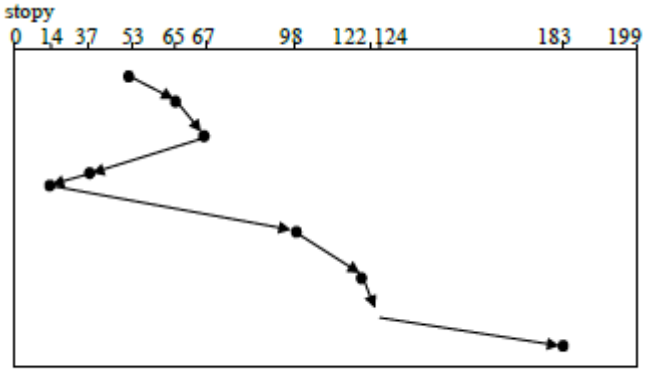
Ramienko sa na začiatku nachádza na stope 53. Ak zoberieme ako jednotku vzdialenosť medzi stopami, potom od 53-tej stopy k 98-mej vykoná ramienko presun o 45 jednotiek, od 98-mej k 183-tej ďalších 85 jednotiek atď. Celkový presun pre uspokojenie všetkých požiadaviek vo fronte bude *640 jednotiek*. Pohyb ramienka je znázornený na Obr.13.2.



Obr. 13.2 Pohyb ramienka podľa algoritmu FCFS

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.

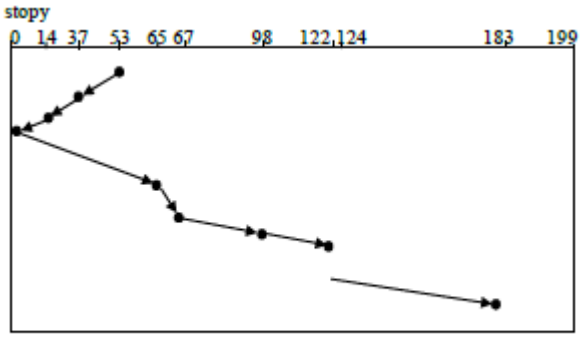
Algoritmus SSTF je lepší ako FCFS, ale nie je optimálny. Uprednostňuje stopy, ktoré sú uprostred a táto jeho vlastnosť sa dá využiť tak, že na prostredných stopách sa budú umiestňovať bloky, vyžadujú rýchly a častý prístup.

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:

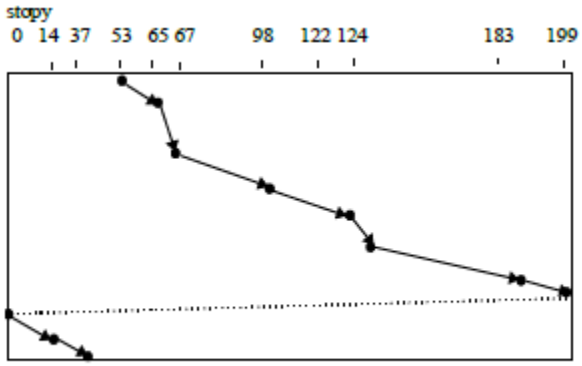
98, 183, 37, 122, 14, 124, 65 a 67.

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.



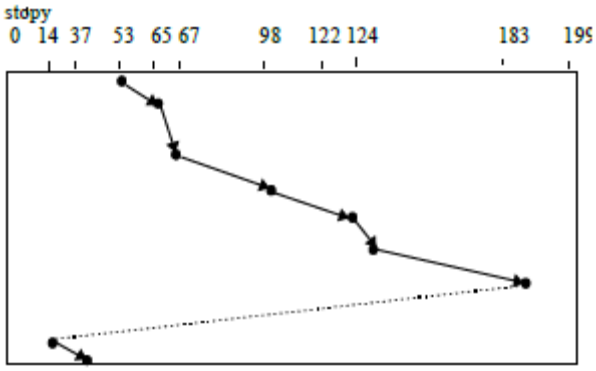
Obr. 13.4 Pohyb ramienka podľa algoritmu výťahu

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

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.

42. Bezpečnosť OS,

Odpoveď: **A**ktuálnou 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.

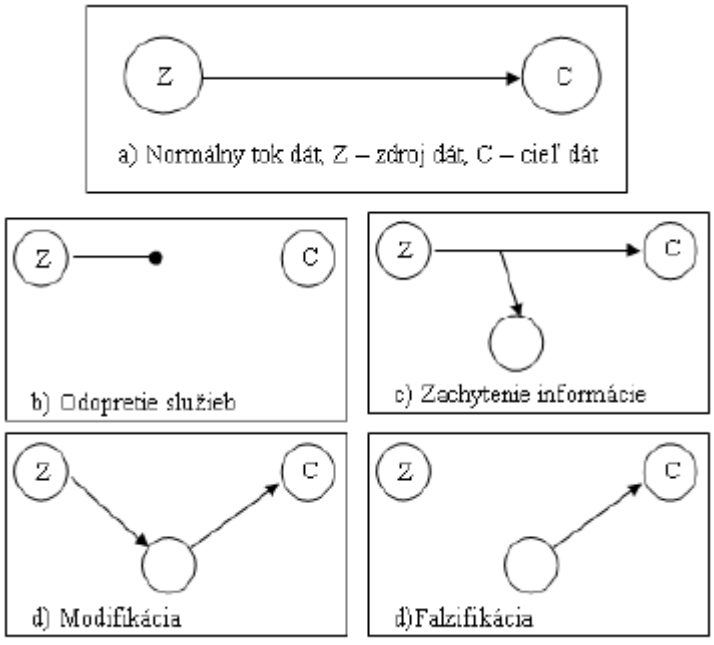
Bezpečnostné hrozby

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.

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.

43. Uveďte aspoň dva spôsoby implementácie zoznamu voľných blokov na disku,

Odpoveď: 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.

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:

1000001111010000100001010001010000000000 ....

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.

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.

44. Mame log. adresný priestor z 8 stranok po 1024 slov, ktorý sa mapuje do fyz. priestoru s 32 rámcami. Koľko bitov ma log. adresa? Koľko bitov ma fyz. adresa,

Odpoveď:

Fyzická adresa: Vynásobíme počet rámcov (32) veľkosťou rámca (veľkosti stránky, teda 1024 slov). Slovo je 16bitové, tj. fyzická adresa ma dĺžku 32*1024*(2^16).

Logická adresa: Bude to obdobne ako u fyzickej akurát namiesto 32 dosadíme 8.

slovo (word) - 16bit = 2 bajty

45. Napíšte malý script, ktorý Vám vypíše procesy, ktoré vykonáva užívateľ, ktorého meno zadávate ako parameter,

Odpoveď:

```
#!/bin/bash
ps –ef | grep "^$1"
```

46. Uveďte základne systémové volania pre správu procesov a ich úlohy,

Odpoveď:

1. príkaz **exit() –** normálne ukončenie procesu,
2 . príkaz **kill()** – násilné ukončenie procesu,
3. príkaz **abort()** – abnormálne ukončenie procesu,
4. príkaz **wait()** – čakanie na ukončenie potomka,

47. Čo urobia príkazy,

```
ps –ef| grep $USER
cd
PATH=$PATH
enz
echo $PATH
cd $HOME
```

Odpoveď:

ps –ef | grep $USER	- vypíše procesy užívateľa uloženého v premennej USER
cd	- zmena pracovného adresára
PATH=$PATH	- nastavenie cesty na aktuálny adresár
env	- výpis systémových premenných – SHELL,TERM, SSH_CLIENT, SSH_TTY,USER,MAIL,PATH,PWD,LOGNAME,...
echo $PATH	- vypíše na obrazovku hodnotu premennej PATH (aktuálneho adresára)
cd $HOME	- zmení adresár na cestu uloženú v premennej HOME (domovský adresár)

48. Máme procesy čas vykonávania RR konštánt,

a) Požiadavky	čas. príchodu	FCFS RR	FSJ	
b) 50	0	50	125	50
c) 30	5	80	90	125
d) 5	15	85	25	55
e) 25	25	110	100	95
f) 15	35	125	75	70