Operačné systémy

- *seek time* (čas presunu hlavy na príslušný cylinder)
- *rotational delay* (čas posunu sektoru pod hlavu)
- *transfer time* (čas prenosu)

Pre väčšinu diskov je dominantný seek time, takže jeho redukovanie môže významne zlepšiť výkonnosť systému.

Požiadavky na prácu s diskom sa zaraďujú do *radu požiadaviek*. Ak sa spracovávajú v poradí, v akom prišli, t.j. stratégiou FCFS (First Come First Served), hľadanie na disku je náhodné, a tak dostávame dlhé časy. Aby sa čas hľadania minimalizoval, treba plánovať prácu s diskom. Treba teda urobiť analýzu a preorganizovanie požiadaviek tak, aby bolo možné nájsť najefektívnejšie poradie ich vykonávania. Možné stratégie sú:

- SSTF (Shortest Seek Time First): Prvá sa bude vykonávať požiadavka, pre ktorú treba minimálny pohyb ramena s čítaco-zapisovacími hlavami. Problémom je, že pre často používaný disk sa môže stať, že rameno bude v strede disku väčšinu času (malé presuny) a požiadavky na okrajoch budú dlho čakať. Tým je zhoršený čas odozvy.
- SCAN alebo tiež *elevator* (prehľadávanie): Pohyb hlavy najprv v jednom smere, pričom sa vykonajú všetky požiadavky, ktoré cestou „stretne“. Potom sa hlava pohybuje v opačnom smere. Zmena smeru teda nastane, ak v danom smere nie je viac požiadaviek alebo ak hlava narazí na okraj disku. Na zistenie súčasného smeru pohybu hlavy stačí jeden bit.
- C-SCAN (cyklické prehľadávanie): hlava sa hýbe len jedným smerom. Ak už v tomto smere nie sú žiadne požiadavky alebo narazí na okraj, „skokom“ sa vráti na začiatok.
- N-step SCAN: Hlava sa hýbe dopredu a dozadu ako v metóde SCAN, ale obsluhuje len požiadavky, ktoré čakali, keď začal pohyb daným smerom. Požiadavky, ktoré prídu potom, sa zaraďujú, aby boli optimálne vybavené pri ceste späť.

Vykonávaciu dobu možno podstatne zredukovať, ak je na perifémom zariadením zaznamenaných niekoľko kópií každej vety, t.j. vo viacerých blokoch. Teda pri čítaní je veta určená niekoľkými alternatívnymi adresami a operácie sa realizujú s „najbližším“ dostupným blokom. Tomuto prístupu sa hovorí *folding*. Koľkokrát sa zväčší počet kópií, toľkokrát sa skráti efektívna vybavovacia doba tejto vety, ale práve toľkokrát sa zmenší kapacita pamäte.

12.5 Hodiny (clocks)

Hodiny sú základom pre činnosť ľubovoľného systému so zdieľaním času z rôznych dôvodov: určujú čas, zabraňujú procesu, aby si monopolizoval čas CPU a pod. Software hodín má zvyčajne formu ovládača zariadenia, hoci hodiny nie sú blokové ani znakové zariadenie.

Software hodín

Hardware hodín len generuje v daných intervaloch prerušenia. Driver hodín má zvyčajne tieto funkcie:

- Udržovať čas: Pri každom tiku sa zväčší počítadlo, ktoré určuje počet tikov od 12 a.m. 1. 1. 1970. Sú tu tri prístupy:
  - Počítadlo má 64 bitov — to značí náročné pripočítavanie.
  - Tik je každú sekundu, takže 32 bitov stačí na 136 rokov.
  - Tiky možno počítať relatívne od času bootovania — počítadlo bude mať 32 bitov.
- Zabraňuje procesu dlho bežať: vždy pri naštartovaní procesu sa inicializuje počítadlo na časové kvantum v tikoch od hodín. Pri každom prerušení od hodín ovládač hodín zníži počítadlo o 1. Keď počítadlo dosiahne nulu, ovládač hodín vyvolá plánovač, aby spustil ďalší proces.
- Administratíva CPU: Treba procesom sledovať čas používania CPU:
  - počítadlom sekúnd, ktoré je pri prerušení niekde odložené a opäť nahraté

1.3.2 Adresné spôsoby

Adresný spôsob (adresný mód) je spôsob špecifikácie umiestnenia operandov. Až na niekoľko výnimiek môže byť ľubovoľný adresný mód použitý s ľubovoľnou inštrukciou. Skoro všetky adresné spôsoby môžu špecifikovať aj dáta aj cieľový operand.

Operand môže byť v registri, v pamäti alebo v samotnej inštrukcii.

Popíšeme si niekoľko základných adresných spôsobov a súčasne uvedieme, ako sa tieto adresné spôsoby prekladajú do strojového kódu.

1. **Registrový mód: Rn**

   Určuje, že operandom je všeobecný register.

   Napr. inštrukcia presunu dlhého slova (MOVL): MOVL R3, R7

   hovorí, že sa má obsah registra R3 presunúť (skopírovať) do registra R7.

   Preklad do strojového kódu: inštrukcia MOVL má kód D0 (v šestnástkovej sústave) - čiže zaberá 1 bajt. Operand v registrovom móde sa tiež prekladá do 1 bajtu, pričom v pravom polbajte je číslo registra (0-F) a v ľavom polbajte je 5 (určuje, že ide o registrový mód).

   Takže preklad uvedenej inštrukcie je: 57 53 D0 (adresy rastú smerom sprava doľava).

2. **Nepriamy registrový mód: (Rn)**

   V registri Rn je pamäťová adresa operandu (obsah registra Rn je smerník do pamäte na operand).

   Napr. MOVL (R3), R7

   hovorí, že sa má obsah pamäťového miesta veľkosti 4 bajty, ktorého adresa je v registri R3, presunúť do registra R7.

   Preklad do strojového kódu: inštrukcia MOVL má kód D0, nepriama registrová adresácia má v ľavom polbajte operandu číslo 6, pravý polbajt udáva číslo registra: 57 63 D0.

   Ak by sme použili operáciu presunu bajtu MOVB (R3), R7 – tak sa obsah pamäťového miesta veľkosti 1 bajt, ktorého adresa je v registri R3, presunie do najpravejšieho bajtu (najnižšie rády) registra R7.

3. **Autoinkrementový mód: (Rn)+**

   V registri Rn je adresa operandu (obsah registra Rn je smerník do pamäte na operand), po určení adresy sa obsah registra automaticky zvýši.

   Napr. MOVL (R3)+, R7

   hovorí, že sa má obsah pamäťového miesta veľkosti 4 bajty, ktorého adresa je v registri R3, presunúť do registra R7. Po určení adresy prvého operandu sa obsah registra R3 automaticky zvýši o 4 (pretože sme použili inštrukciu narábajúcu s dlhými slovami = 4 bajty) - čiže bude obsahovať adresu nasledujúceho dlhého slova.

   Tento adresný spôsob je významný pre prácu s poľami.

   Preklad do strojového kódu: v ľavom polbajte operandu je číslo 8, pravý polbajt udáva číslo registra: 57 83 D0.

4. **Autodekrementový mód: -(Rn)**

   Obsah registra Rn sa najprv automaticky zníži (o 1, 2 alebo 4 – podľa použitej inštrukcie) a až potom sa použije ako adresa operandu.

   Napr. MOVL -(R3), R7

   hovorí, že sa má obsah registra R3 znížiť o 4 a potom sa má obsah pamäťového miesta veľkosti 4 bajty (longword), ktorého adresa je v registri R3, presunúť do registra R7.

   Tento adresný spôsob možno použiť pre prácu s poľami v opačnom poradí.

   Preklad do strojového kódu: v ľavom polbajte operandu je číslo 7, pravý polbajt udáva číslo registra: 57 73 D0.

- *znakové zariadenia*: Prenos informácie sa realizuje na základe toku znakov (bez umožňovania nejakej blokovej štruktúry), napr. terminály, riadkové tlačiarne, snímač a dierovač diernej pásky, network interface, myš a pod. Tieto zariadenia nie sú adresovateľné a neumožňujú operáciu vyhľadania (seek).

Niekedy môže periférne zariadenie pracovať podľa zadaného príkazu v blokovom alebo znakovom režime.

Táto klasifikácia nie je dokonalá, niektoré zariadenia jej nevyhovujú, napr. hodiny nemajú adresovateľné bloky ani negenerujú či neakceptujú tok znakov, len vyvolávajú prerušenia v definovaných časových intervaloch.

Podľa techniky prideľovania rozdeľujeme periférne zariadenia na:

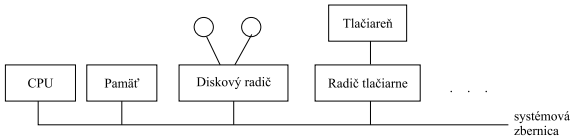
- Pevne prideľované periférne zariadenia (*dedicated*): zariadenie je pridelené úlohe po celú dobu jej trvania. Je to vhodné pre určité typy V/V zariadení, ako napr. snímače štítkov, tlačiarne, ...
- Zdieľané periférne zariadenia (*shared*): Ide o zariadenia používané viacerými procesmi (ako napr. väčšina pamätí s priamym prístupom). Treba riešiť otázky riadenia: Ak dva procesy žiadajú čítanie z toho istého disku, treba rozhodnúť, ktorej požiadavke bude vyhovené ako prvej. Stratégie rozhodovania môžu byť založené na stanovení priorít alebo na snahe po čo najlepšej efektivite systému a pod.
- Virtuálne periférne zariadenia (*virtual*): Niektoré periférne zariadenia, ktoré treba pevne prideliť (napr. tlačiareň) možno previesť napr. pomocou techniky „spooling“ na zdieľané periférne zariadenia.

12.2 Technické charakteristiky perifémych zariadení

Periférie zvyčajne pozostávajú z *mechanickej* a *elektronickej* časti. Často je ich možné oddeliť a umožniť tak modulárnejší a všeobecnejší design. Elektronický komponent sa nazýva *riadiaca jednotka* alebo *radič* (*device controller, adapter*). Na mini- a mikropočítačoch má často podobu karty s plošnými spojmi, ktorá sa vkladá do počítača. Mechanický komponent je zariadenie samotné.

Karta radiča má zvyčajne konektor, do ktorého sa zapája kábel vedúci k príslušnému zariadeniu. Mnoho radičov môže ovládať niekoľko identických perifémych zariadení.

Na rozdiel medzi radičom a zariadením upozorňujeme preto, že operačný systém skoro vždy má do činenia s radičom, nie so zariadením. Takmer všetky mikro a minipočítače používajú model jednej zbernice na komunikáciu medzi CPU a radičmi.



Veľké počítače používajú iný model, s viacerými zbernicami a špecializovanými V/V procesormi, nazývanými *V/V-kanály*. Tie vykonávajú „kanálové“ programy, ktoré slúžia na prenos dát medzi V/V zariadením a operačnou pamäťou a sú špecializované výhradne na V/V-operácie.

Interface medzi radičom a zariadením je často veľmi nízkoúrovňový interface. Napr. disk môže byť formátovaný do 8 sektorov po 512 bajtov na stopu, avšak to, čo skutočne prichádza z disku, je sériový tok bitov, začínajúci preambulou (*preamble*), potom 4096 bitov sektoru a napokon *checksum* alebo error-correcting code (ECC). Preambula je vytvorená pri formátovaní disku (obsahuje cylinder, číslo sektoru, jeho veľkosť a podobné dáta). Úlohou radiča je premeniť tok bitov na blok bajtov a vykonať opravu

Nepodmienené skoky

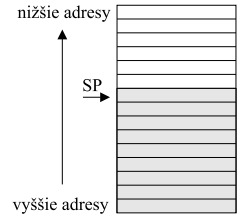
Nepodmienené skoky vždy zmenia obsah PC registra.

V preklade do strojového kódu sa u inštrukcií BRB a BRW ukladá opäť rozdiel medzi návestím a PC registrom, pri BRB sa uloží do 1 bajtu (celá inštukcia zaberá 2 bajty), pri BRW sa uloží do 2 bajtov (celá inštukcia zaberá 3 bajty). Pri inštrukcii JMP sa môže použiť na určenie cieľa ľubovoľný adresný mód (okrem priameho a literálu) – preklad potom závisí od použitého adresného módu.

Práca so zásobníkom

Zásobník je súvislé pole dátových miest používané na uloženie dočasných dát a informácie súvisiacej s volaním procedúr. Dátové položky sú do zásobníka vkladané a zo zásobníka vyberané metódou LIFO (last in first out). Na posledne vloženú položku zásobníka ukazuje premenná nazývaná *stack pointer - SP* (na VAXe je to register R14). Po zavedení programu do pamäte operačný systém automaticky vyhradí blok pamäte v adresnom priestore používateľa a nastaví SP.

Na VAXe zásobník rastie smerom k nižším adresám.



Inštrukcie pre prácu so zásobníkom:

PUSHL	čo	vlož do zásobníka dlhé slovo	≡ MOVL čo, -(SP)
POPL	kam	vyber zo zásobníka dlhé slovo	≡ MOVL (SP)+, kam
PUSHR	#^M<zoznam_registrov>	ulož do zásobníka registre z masky od registra s najvyšším číslom po najnižšie	
POPR	#^M<zoznam_registrov>	vyber zo zásobníka dlhé slová a daj do registrov z masky od registra s najnižším číslom po najvyššie	
PUSHAx	adr	ulož do zásobníka adresu adr	(x=B,W,L)

Poznámka: pre vloženie a vybratie dát iného rozmeru ako longword treba použiť inštrukcie MOVx čo, -(SP) a MOVx (SP)+, kam, kde x je rozmer dát, s ktorými narábame.

1.3.5 Procedúry

Procedúry umožňujú rozdeliť riešenie úlohy na časti, ktoré sú ľahšie modifikovateľné a odladiteľné.
VAX asembler poskytuje 2 volania procedúr:

- CALLG adresa_zoznamu_argumentov, meno
- CALLS počet_argumentov, meno

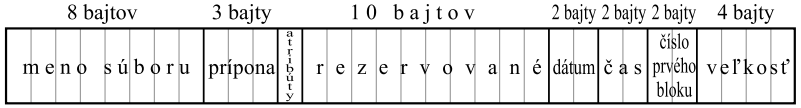
Oba spôsoby používajú *zoznam argumentv*, líšia sa však v tom, kde je tento zoznam uložený: v prípade CALLG (Call General) je to hocikde v pamäti (napr. naň vyhradíme miesto na začiatku programu - v časti deklarácií), u CALLS (Call Stack) sa uloží zoznam argumentov do zásobníka. V oboch prípadoch na zoznam argumentov ukazuje register R12 = AP (*Argument Pointer*).

Formát zoznamu argumentov:

atď. Časť *block count* hovorí, koľko z potenciálnych 16 diskových blokov je použitých. Posledný blok súboru nemusí byť plný, takže operačný systém nemá spôsob, ako určiť presnú veľkosť súboru v bytoch, uchováva informáciu o veľkosti súboru v blokoch.

Teraz si preberieme príklady systémov s hierarchickými stromovými štruktúrami adresárov.

V MS-DOSe má položka v adresári 32 bytov rozdelených nasledovne:



32 bajtov

Okrem hlavného adresára, ktorý má pevnú dĺžku, ostatné adresáre sú súbory, a teda môžu obsahovať ľubovoľný počet položiek.

Štruktúra adresárov Unixu je veľmi jednoduchá. V System V má každá položka 16 bytov (maximálna dĺžka mena súboru je 14 znakov):

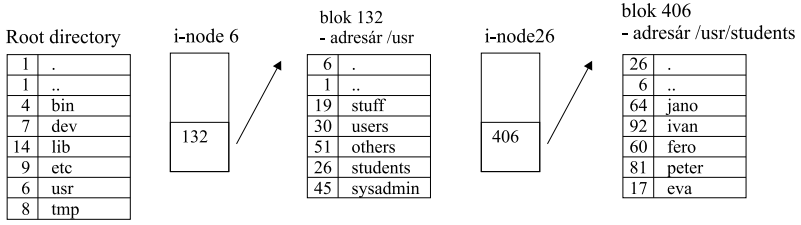
i-node number	file name
2B	14B

Od BSD 4.2 majú BSD systémy položky premenlivej dĺžky a umožňujú názov súboru až po 255 znakov:

i-node	dĺžka položky	dĺžka názvu	názov súboru (max 255 znakov)
2B	2B	1B	max. 255B

Keď otvárame súbor, musíme podľa názvu súboru nájsť bloky, ktoré ho tvoria. Uvažujme napr. súbor /usr/students/fero a operačný systém Unix (algoritmus je v podstate ten istý pre všetky hierarchické systémy adresárov):

Najprv musíme nájsť *hlavný adresár* (*root directory*), ktorého adresa (*i*-node) je na pevnom mieste disku. V hlavnom adresári nájdeme položku usr, čím určíme *i*-node pre /usr. Z tohto *i*-node systém nájde adresár /usr a v ňom hľadá položku students. Keď ju nájde, má *i*-node pre adresár /usr/students. Z tohto *i*-node možno nájsť adresár /usr/students a v ňom hľadanú položku fero. *i*-node pre tento súbor je načítaný do pamäti a uložený v nej až dovtedy, kým súbor nie je zatvorený.



Relatívne názvy súborov hľadáme analogicky, avšak nezačíname z hlavného, ale z aktuálneho adresára.

Každý adresár obsahuje položky s názvom „.“ a „..“ (aktuálny a rodičovský adresár). Tieto názvy spolu s adresami zodpovedajúcich *i*-nodes sa vytvoria pri vytvorení adresára. Preto je možné použiť aj názvy ../others/prog.c: procedúra nájde v pracovnom adresári *i*-node pre rodičovský adresár a hľadá v ňom others. Na uchovanie týchto názvov nie je treba žiaden špeciálny mechanizmus, sú to jednoducho ASCII reťazce.

určenie týchto hodnôt používa premennú LC = *Location counter*, ktorá funguje počas prekladu tak, ako PC za behu programu. Asembler vždy zvyšuje hodnotu LC o dĺžku inštrukcie, takže LC vždy obsahuje adresu nasledujúcej inštrukcie.

Podľa počtu prechodov cez zdrojový text rozlišujeme asemblery:

- dvojprechodové
- jednoprechodové

Dvojprechodový asembler

1. prechod: jeho úlohou je prejsť vstupný text, priradiť miesto každej inštrukcii a tým definovať hodnoty návestí. Vytvára *tabuľku symbolov*, do ktorej zapíše všetky nájdené symbolické mená spolu s ich hodnotami alebo adresami a prípadne ďalšou informáciou (premenná lokálna, globálna, externá).

Postup pri vytváraní tabuľky symbolov je nasledovný: na začiatku prvého prechodu sa nastaví LC na 0. Postupne asembler číta riadky zdrojového textu, ak riadok obsahuje návestie, zapíše ho do tabuľky symbolov spolu s aktuálnou hodnotou LC. Ak v tabuľke symbolov už symbol s rovnakým názvom existuje, vypíše chybu „Viacnásobne definovaný symbol“. LC zvýši o dĺžku inštrukcie a opakuje uvedený postup, až kým nepríde na koniec programu.

Na zistenie dĺžky inštrukcie a tiež overenie platnosti inštrukcie je potrebné prehľadať *tabuľku kódov inštrukcií* – obsahuje meno inštrukcie, jej ekvivalent v strojovom kóde, prípadne informáciu o formáte a dĺžke inštrukcie.

Prvý aj druhý prechod asemblera môžu ako vstup používať zdrojový program, ale je výhodnejšie, ak prvý prechod vytvorí upravený zdrojový program, ktorý sa potom stane vstupom pre druhý prechod. Upravený program obsahuje zdrojové riadky spolu s ich adresou, indikátormi chyby, môžu tu byť uložené aj smerníky do tabuľky kódov inštrukcií (pre kód inštrukcie) a tabuľky symbolov (pre každý použitý symbol), aby nebolo nutné opätovné prehľadávanie týchto tabuliek v druhom prechode.

2. prechod: druhýkrát sa prechádza vstupný (príp. upravený) text a robí sa preklad do strojového kódu. Ak sa v inštrukcii vyskytne symbol, dosadí sa jeho numerická hodnota alebo adresa z tabuľky symbolov.

Jednoprechodový asembler

Pri jednoprechodovom asembleri sa číta zdrojový text iba raz a v tomto jednom prechode sa vyrába tabuľka symbolov aj prekladá do strojového kódu. K problémom dochádza pri priradení numerických hodnôt symbolom (návestiam), ktoré sa v programe definujú neskôr, ako sa použijú. Tento problém možno riešiť tak, že sa vytvorí linkovaný zoznam nedefinovaných návestí. Po ukončení čítania vstupného textu sa len doplnia hodnoty návestí na miesta označené uvedeným zoznamom.

1.5 Makrá, makroprocesory

Makro je pomenovaná skupina inštrukcií, ktoré sa vložia do kódu na mieste, kde sa makro použije (volá).

Definícia makra môže byť daná programátorom v programe, v ktorom sa používa alebo môže byť v knižnici makier, ktorá je prístupná jednému alebo viacerým používateľom.

Proces nahradenia výskytu mena makra – *volania makra* – príslušnými príkazmi, sa nazýva *rozvoj makra* (macro expansion). Rozvoj makra nemusí byť pri každom volaní rovnaký, lebo v makre je možné použiť aj parametre.

V porovnaní s procedúrami je použitie makier nevýhodnejšie z hľadiska dĺžky výsledného kódu (lebo každé volanie makra vedie k vloženiu jeho tela na miesto volania, kým procedúry potrebujú v pamäti len jednu kópiu svojho kódu), ale je výhodnejšie z časového hľadiska (pri volaní procedúry vznikajú časové straty na vytvorenie prepojenia medzi programovými modulmi – napr. uloženie bloku volania, ktoré pri makrách nie sú).

Definícia makra:

.MACRO meno [zoznam_parametrov]

11.1.6 Operácie so súbormi

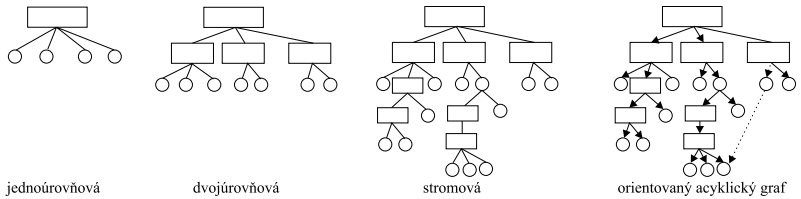
sú rôzne pre rôzne operačné systémy. Všeobecne: create, delete, open, close, read, write, append (pridať na koniec súboru), seek (pre náhodný prístup: zmena ukazovateľa pozície) get attributes, set attributes, rename.

11.1.7 Adresáre

Slúžia na udržiavanie prehľadu o uložení súborov. V mnohých operačných systémoch aj adresáre sú súbory.

Hierarchické systémy adresárov

1. Jeden adresár pre všetkých používateľov — *jednoúrovňová organizácia*: Ide o najjednoduchší spôsob, sú možné konflikty pri pomenovaní súborov. Použitie: primitívne mikropočítačové operačné systémy.
2. Po jednom adresári pre každého používateľa — *dvojúrovňová organizácia*
3. *stromová štruktúra*
4. *orientovaný acyklický graf*: Jeden súbor môže mať niekoľko mien a prístupových ciest. Umožňuje to zdieľanie súborov.



Mená ciest (path names)

- absolútne: od koreňa, napr. /usr/users/jano. Jeden znak je oddeľovač: v Unixe „/“, v MS-DOSe „\“ alebo „/“, v Multicse „>“.
- relatívne: vzhľadom na *working directory* (current directory). Prvý znak je iný ako oddeľovač. Tiež je možnosť použiť „.“, „..“.

Operácie s adresármi

V Unixe: create, delete, opendir, closedir, readdir, rename, link, unlink.

11.2 Správa priestoru na disku

Voľné bloky

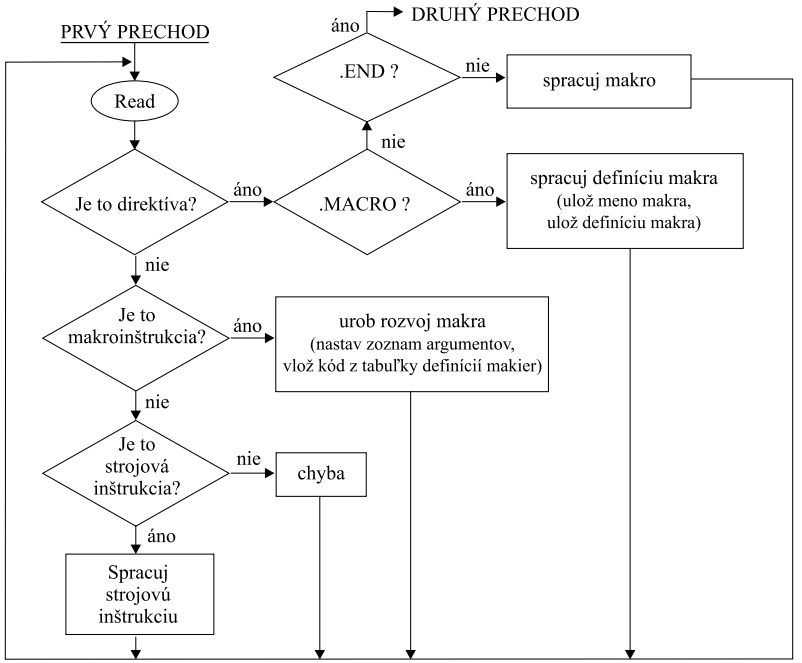
Operačný systém si musí udržiavať *prehľad o voľných blokoch* na disku. Na to je možné použiť viaceré metódy:

- *Spájaný zoznam voľných blokov*:

  Nevýhoda: na vyhradenie n blokov treba n prístupov na disk. Alternatívou je preto zoznam skupín blokov.

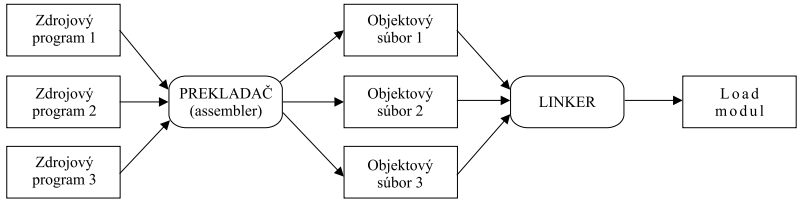
Makroprocesor sa môže pridať ako predprocesor pred asembler, ale je tiež možné implementovať jednoprechodový makroprocesor do prvého prechodu asemblera – výsledok sa nazýva *makroasembler*.

Toto spojenie vylučuje náklady na vytváranie prechodných súborov a tiež mnohé činnosti nie je potrebné implementovať dvakrát (čítanie zdrojového riadku, testovanie typu príkazu, ...).



1.6 Linker a loader

Väčšina programov pozostáva z viacerých procedúr. Kompilátory a asemblery zvyčajne prekladajú vždy len jednu procedúru a preložený výstup uložia na disk. Pred tým, ako je možné spustiť program, musia byť nájdené všetky potrebné preložené procedúry a musia byť správne spojené. Výsledný modul je potom zavedený do pamäte.



Úlohou *linkera* je spojiť separátne preložené procedúry do jedného modulu, zvyčajne nazývaného *load module*. *Loader* potom nahrá load modul do pamäte. Tieto funkcie sú často kombinované.

Preloženie každej procedúry ako separátnej entity má výhodu v tom, že pri zmene v niektorej procedúre stačí prekompilovať len zmenenú procedúru (aj keď treba vykonať nanovo linkovanie), a nie všetky, ako by to bolo nutné, ak by kompilátor čítal sériu procedúr a priamo vyrábal spúšťateľný program.

Swapovanie do pamäte

Každých niekoľko sekúnd swapper prezerá zoznam odswapovaných procesov, aby zistil, či nie je nejaký pripravený. Ak áno, vyberie sa taký, čo je najdlhšie na disku. Swapper preverí, či je naň v pamäti miesto. Ak nie, treba odswapovať jeden alebo viac procesov z pamäte na disk. Tento algoritmus sa opakuje, až kým nenastane jedna z udalostí:

1. žiaden proces na disku nie je pripravený
2. pamäť je plná procesov, ktoré boli práve do nej nahraté, takže nie je možné uvoľniť miesto. (Proces nemôže byť odswapovaný z pamäte, ak v nej nie je aspoň dve sekundy).

Evidencia voľného miesta

na disku a v pamäti — linkovaný zoznam voľných úsekov

10.6.2 Stránkovanie

(od verzie 3BSD, 4BSD aj System V implementujú demand paging — stránkovanie na žiadosť)

Stačí, aby „user structure“ a tabuľka stránok boli v pamäti a proces môže byť naplánovaný na spracovanie. Požadované stránky sú nahrávané do pamäte dynamicky. Ak „user structure“ a PT nie sú v pamäti, proces nemôže bežať, kým ich swapper nenahrá do pamäte.

Berkeley Unix nepoužíva model s pracovnou množinou alebo inú formu predstránkovania, lebo keďže VAX nemá „reference“ bity, je ťažké sledovať používané stránky.

Stránkovanie je implementované sčasti hlavným kernelom a sčasti novým procesom — *page daemon* (proces č. 2). Ten je periodicky štartovaný a kontroluje, či je nejaká robota, ktorú má urobiť. Ak je počet voľných stránok v pamäti príliš nízky, naštartuje akcie na uvoľnenie viac rámcov.

Hlavná pamäť v 4BSD pozostáva z 3 častí:

- kernel
- core map (kernel a core map nie sú nikdy odstránkované)
- zvyšná pamäť — delí sa na rámce

Core map obsahuje informácie o obsahu rámcov (pre každý rámec jednu položku). Ak napr. rámce majú 1K a položky v core map 16B, tak core map zaberá menej ako 2% pamäte. Prvé dve položky sa používajú, ak je rámec voľný: obsahujú smerníky do zoznamu voľných rámcov. Ďalšie tri položky sa používajú na určenie miesta na disku, kde je stránka uložená. Ďalšie tri položky dávajú číslo položky v tabuľke procesov pre proces, ktorému stránka patrí. Posledná položka obsahuje flagy, potrebné pre stránkovací algoritmus.

Ak nastane page fault, OS berie prvú stránku zo zoznamu voľných stránok a požadovanú stránku nahrá do nej. Ak však nie je voľný rámec, proces je pozastavený, kým page daemon neuvoľní rámec.

Nahradzovací algoritmus

je vykonávaný page daemonom. Každých 250 ms je daemon zobudený, aby zistil, či počet voľných rámcov je aspoň lotsfree (systémový parameter, zvyčajne aspoň 1/4 pamäte). Ak je počet stránok menší, začne presúvať stránky z pamäte na disk. Ak je väčší, zaspí.

Page daemon používa modifikovanú verziu „hodinového“ algoritmu. Základný „hodinový“ algoritmus prejde všetky stránky a vynuluje „usage bit“. V 2. prechode každá stránka, ktorá nebola od 1. prechodu referencovaná, je po zapísaní zaradená do zoznamu voľných stránok.

Pretože prechody trvali príliš dlho, bol algoritmus modifikovaný na *two-handed clock algorithm*. Predná ručička nuluje „usage bit“, zadná preveruje jeho nastavenie. Ak sú ručičky príliš blízko, iba veľmi často používané stránky majú šancu byť použité medzi prechodom prvej a druhej ručičky. Ak sú priďaleko (napr. 359°), dostaneme pôvodný hodinový algoritmus.

2.1 História operačných systémov

Všimneme si generácie počítačov, aby sme videli, ako vyzerali ich operačné systémy.

Prvý skutočne digitálny počítač zostrojil anglický matematik Charles Babbage (1792–1871). Nikdy nepracoval správne kvôli svojmu čisto mechanickému designu.

Prvá generácia počítačov (1949-1955)

- do 2. svetovej vojny — malý pokrok v konštrukcii počítačov
- v polovici 40. rokov — niekoľko úspešných pokusov — počítače s použitím elektroniek (Howard Aiken v Harvarde, John von Neumann v Princetone, J. Presper Eckert a William Mauchley v Pensylvánii, Konrad Zuse v Nemecku)
- išlo o veľmi mohutné zariadenia: napr. ENIAC vážil 30 ton, bol postavený v bývalom leteckom hangári, mal 18000 elektrónok v bloku rozmerov 30×3 metre a bol chladený dvoma vyradenými leteckými motormi
- každý počítač navrhla, vytvorila, programovala a udržovala jedna skupina ľudí, programovalo sa v strojovom jazyku, neexistovali programovacie jazyky (ani assembler), ani OS. Väčšina úloh boli náročné matematické výpočty.
- začiatkom 50. rokov sa začali používať dierne štítky

Druhá generácia počítačov (1955–1965)

- začína sa zavedením tranzistorov. Počítače začínajú byť dostatočne spoľahlivé, aby sa mohli začať vyrábať a predávať.
- po prvý raz sa začínajú oddeľovať návrhári, tvorcovia, operátori, programátori a udržovací personál.
- objavili sa programovacie jazyky (assembler, Fortran)
- zo začiatku boli pri spracovaní veľké časové straty operátorov (ktorí mali na starosti načítanie sady diernych štítkov, príp. prekladača, výstupy,...). Snaha o ich redukciu viedla k zavedeniu *batch systémov*: po nazhromaždení úloh sa tieto načítali na magnetickú pásku použitím malého, relatívne nie veľmi drahého počítača (napr. IBM 1401), ktorý bol dobrý na čítanie štítkov, kopírovanie pások, tlač, ale nie na numerické výpočty. Na výpočty bol použitý iný, drahší počítač (napr. IBM 7094). Po zhromaždení úloh bola páska previnutá a prenesená do počítačovej miestnosti. Operátor nahral špeciálny program (predchodcu dnešných operačných systémov), ktorý načítal úlohu a spustil ju. Výstup sa ukladal na ďalšiu pásku. Keď bol celý batch vykonaný, operátor vyňal obe pásky a výstupnú preniesol do iného počítača (IBM 1401) na výpis off-line (t.j. bez spojenia s hlavným počítačom).
- počítače sa používali zväčša na vedecké a inžinierske výpočty, zvyčajne boli vo Fortrane a assembleri. Typický OS bol FMS (the Fortran Monitor System) a IBSYS (IBM OS pre 7094)

Tretia generácia počítačov (1965–1980)

- Na začiatku 60. rokov už mala väčšina výrobcov počítačov dve rozdielne línie produktov — na jednej strane to boli vedecké počítače (ako 7094) používané na numerické výpočty vo vede a strojárstve, na druhej strane to boli obchodné počítače (ako 1401) široko použiteľné na triedenia a tlač bankami a poisťovňami. Vývoj a urdžiavanie dvoch rozdielnych línií bolo pre výrobcov drahé a okrem toho viacero zákazníkov potrebovalo zo začiatku malý počítač, ale neskôr väčší, ktorý by mohol spúšťať všetky ich staré programy, ale rýchlejšie.
- IBM sa pokúsilo vyriešiť oba tieto problémy zavedením System/360 — série sotwarovo kompatibilných počítačov v rozsahu od počítača veľkosti 1401 až po výkonnejšie ako 7094. Líšili sa len v cene a výkone (maximálnej pamäte, rýchlosti procesora, počtu povolených V/V-zariadení, atď.). Boli vyvinuté na spracovanie vedeckých aj obchodných výpočtov. 360 bola prvá línia počítačov s použitím integrovaných obvodov.

10.1 Nahradzovacie algoritmy

FIFO (First-In-First-Out Page Replacement)

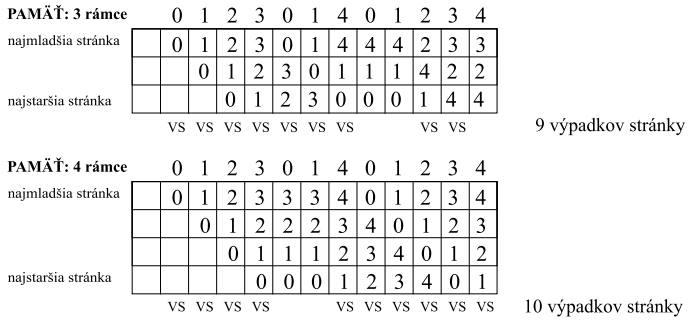
Pomerne nenáročný na implementáciu: OS si udržiava zoznam všetkých stránok, ktoré sú práve v pamäti. Na začiatku zoznamu je „najstaršia“ stránka, nové stránky sa pridávajú na koniec zoznamu. Pri výbere obete sa vyberá stránka, ktorá je na čele zoznamu a nová stránka je zaradená na koniec zoznamu. Výhody:

- Stránka nie je odstránená hneď po svojom zavedení do pamäte.

Nevýhody:

- Stránka, ktorá je v operačnej pamäti sústavne používaná bude vybraná ako obeť, lebo je „najstaršia“ (aj keď bude hneď potrebná).

Môžu sa vyskytnúť pozoruhodné vedľajšie javy, napr. FIFO anomália (Beladyho): za istých podmienok sa môže stať, že pri zväčšení operačnej pamäte (zväčší sa počet rámcov) dostaneme horšiu efektivitu (zväčší sa počet výmen). Napr. ak má operačná pamäť 3 rámce a program 5 stránok, ktoré sú požadované v nasledovnom poradí: 0 1 2 3 0 1 4 0 1 2 3 4



NRU (Not-Recently-Used Page Replacement)

Väčšina počítačov s virtuálnou pamäťou má ku každej stránke pridelené dva bity: R — *referenced* a M — *modified*. Bit R sa nastaví vždy, keď sa stránka číta alebo sa do nej zapisuje. Bit M sa nastaví vtedy, keď sa do stránky zapisuje (Ak je M bit "obete" 0, nemusí sa táto stránka nahrávať na disk - lebo sa nezmenila a teda na disku existuje jej kópia - len sa prepíše novou stránkou). Keďže sa tieto bity modifikujú pri každom použití pamäte, je dôležité, aby sa to robilo rýchlo, teda hardwarovo. Keď je bit hardwarom nastavený na 1, už sa hardwarovo nemení, až kým ho OS nezmení softwarovo. Bity R a M možno využiť nasledovne: Pri odštartovaní procesu sa všetky bity vynulujú OS. Periodicky (napr. pri prerušení od hodín) sa bit R vynuluje, aby sme odlíšili stránky, ktoré sa nedávno použili od použitých dávnejšie. Keď sa vyskytne výpadok stránky, OS rozdelí všetky stránky do 4 kategórií na základe súčasnej hodnoty R a M bitov:

Trieda 1 — neodkazované, nezmenené (R=0, M=0)

Trieda 2 — neodkazované, zmenené (R=0, M=1)

Trieda 3 — odkazované, nezmenené (R=1, M=0)

Trieda 4 — odkazované, zmenené (R=1, M=1)

Algoritmus potom vyberá stránky náhodne z neprázdnej triedy s najnižším číslom.

rôzne softwarové objekty, ktoré sú riadené operačným systémom. Najdôležitejšími sú *procesy* a *súbory*.

Procesy

Kľúčovým pojmom v každom OS sú *procesy*, t.j. programy vo vykonávaní: spustiteľný program, dáta programu, zásobník, program counter, stack pointer, ostatné registre a informácia potrebná na beh programu.

Proces pozostáva zo svojho *adresného priestoru* (core image) a položky v *tabuľke procesov* (pole alebo zoznam štruktúr, jedna pre každý existujúci proces). Informácia v tabuľke procesov je potrebná pri pozastavení procesu (vyčerpanie času CPU, proces s vyššou prioritou a pod.) na reštartovanie od toho istého stavu, kde bol proces zastavený (napr. pozícia v otvorených súboroch, obsahy registrov, atď.).

Hlavné systémové volania pre správu procesov sú *volania na vytvorenie a ukončenie procesu* (napr. Command Interpreter vytvorí proces na vykonanie programu, t.j. proces — potomok). Ďalej sú to volania: požiadavka na viac pamäte, uvoľnenie nepoužívanej pamäte, čakanie na ukončenie procesu (potomka), ...

Niekedy treba bežiacemu procesu doručiť informáciu, na ktorú nečaká. Vtedy OS vyšle procesu *signál*. Ten spôsobí, že je proces pozastavený, uloží svoje registre do zásobníka a odštartuje beh špeciálnej procedúry na *ošetrenie signálu* (signal handling procedure). Po jej dokončení je proces reštartovaný.

Signály sú softwarovou analógiou hardwarových prerušení a môžu byť generované množstvom dôvodov (mnoho prerušení detekovaných hardwarom, napr. vykonanie ilegálnej inštrukcie, použitie zlej adresy je tiež konvertované do signálov). Signály sa používajú aj na rýchlu komunikáciu medzi procesmi.

V multiprogramovom systéme je nevyhnutné udržiavať informáciu o tom, ktorý používateľ vlastní proces. Každému používateľovi je pridelený *uid* (user identification), zvyčajne 16- alebo 32-bitové celé číslo. Každému procesu je priradený uid jeho vlastníka. Používatelia sa delia do skupín (tímy pracujúce na projekte, katedry,...), z ktorých každá má pridelený *gid* (group identification). Uid a gid sa používajú aj pri ochrane informácií v počítači.

Súbory

Ďalšia veľká skupina systémových volaní sa vzťahuje na *systém súborov* (file system). Zvyčajne sa používajú na vytvorenie, zmazanie, čítanie a zápis do súborov. Pred čítaním musí byť súbor otvorený, po čítaní zatvorený.

Väčšina OS používa na uchovávanie súborov koncept *adresára*. Systémové volania sa potom používajú na vytvorenie a zrušenie adresárov, uloženie súboru do adresára, vymazanie súboru z adresára.

Ak majú viacerí používatelia prístup k tomu istému počítaču, je dôležité zabezpečiť prostriedky na *ochranu súborov*. Tie sa líšia pre rôzne OS. V Unixe je napr. každému súboru a adresáru priradený 9-bitový binárny kód ochrany, zložený z troch 3-bitových polí (owner, group, world), každé obsahujúce bity pre read(r), write(w) a execute, resp. search(x). Pri otváraní súboru sa *preverujú prístupové práva*. Ak je prístup povolený, systém vráti celé číslo — *file descriptor*, ktorý sa používa pre ďalšie operácie. Ak je prístup zakázaný, vráti sa kód chyby. V Unixe a MS-DOSe je deskriptor 0 priradený štandardnému vstupu (zvyčajne klávesnica), 1 štandardnému výstupu (terminál), 2 štandardnému chybovému výstupu (terminál).

Unix a MS-DOS umožňujú použitie prostriedku, ktorý sa týka procesov aj súborov — *pipe* (rúra). Je to druh pseudosúboru, ktorý sa používa na prepojenie dvoch procesov. Ak proces A chce poslať dáta procesu B, zapíše ich do rúry, ako by to bol výstupný súbor. Proces B číta z rúry ako by to bol vstupný súbor. Teda komunikácia medzi procesmi vyzerá ako čítanie a zápis do obyčajných súborov. Ako príklad uveďme `ls -l|grep Jan` (výpis súborov z januára) alebo `cat f1 f2|sort`.

Systémové volania

Užívateľské programy komunikujú s OS a žiadajú o služby OS prostredníctvom *systémových volaní*. Každému systémovému volaniu zodpovedá knižničná procedúra (ktorú môže užívateľský program volať)

zložka určuje, či je rámec voľný alebo či obsahuje stránku niektorého procesu (tam je zakódovaná identifikácia tohto procesu). Definičná zložka obsahuje číslo stránky, ktorá je umiestnená v rámci. Niekedy sa zobrazenia LAP procesov vo FAP môžu čiastočne prekrývať. Vzniká tým zdieľanie podprogramov, dát atď. Stránkovanie umožňuje zobraziť viac logických adresových priestorov do jedného, spoločného FAP, tým sa uľahčuje multiprogramovanie a prijateľným spôsobom rieši problém fragmentácie (pripustením vnútornej fragmentácie sa minimalizuje vonkajšia fragmentácia). Stránkovanie odstraňuje nutnosť kompaktovania. Vyžaduje si však nákladnejšie technické vybavenie a predlžuje priemernú dobu prístupu k informáciám uloženým v operačnej pamäti počítača.

Špecifickým problémom je ochrana stránky: používajú sa bity ochrany spojené so stránkami (R/W, RO), uchovávané v PT.

Implementácia tabuľky stránok

1. množina vyhradených registrov: Plánovač procesov nahráva ich obsah tak, ako sa nahráva obsah ostatných registrov. Inštrukcie na modifikáciu týchto registrov sú privilegované, takže ich môže meniť len OS. Toto sa používalo napr. v XDS-940: 8 stránok po 2048 slov, NOVA BID: 32 stránok po 1024 slov, Sigma 7: 256 stránok, teda bolo treba 8–256 registrov. Táto technika sa dá použiť, len keď je PT pomerne malá, ale DEC-10 má 512 stránok, IBM 370 až 4096 stránok, takže nie je možné používať registre.

2. PT je uchovávaná v hlavnej pamäti a ukazuje do nej *Page Table Base Register* (PTBR):

   - Výmena tabuľkových stránok vyžaduje len zmenu tohto registra.
   - Problémom je čas na prístup k užívateľskej pamäti: Ak chceme dosiahnuť pozíciu i, najprv pristúpime do PT, tam nájdeme číslo rámca a určíme fyzickú adresu. Potom pristúpime na túto adresu. Ide teda o 2 prístupy do pamäte, takže nastáva isté spomalenie.
   - Štandardným riešením je použitie špecifickej, malej HW pamäte, nazývanej *associative registers* alebo *cache*. Tieto registre obsahujú len niektoré položky z PT. Číslo stránky sa najprv hľadá v cache. Ak sa nájde, máme priamo číslo rámca. Ak sa nenájde, treba pristúpiť do pamäte — do PT a vyhľadať číslo rámca. Potom sa tento pár pridá do cache, takže ďalšíkrát sa vyhľadá veľmi rýchlo. Pri použití 8–16 asociatívnych registrov asi 80–90% percent času nájdeme žiadané číslo stránky v asociatívnych registroch.

Zdieľateľné stránky

Ďalšou výhodou stránkovania je možnosť zdieľania spoločného kódu medzi viacerými procesmi (napr. editory, kompilátory, DB-systémy atď.) Podmienkou je, aby tento kód bol *reentrantný*, t.j. nemodifikoval sám seba počas výpočtu. Potom ho viac procesov môže vykonávať v tom istom čase, pričom každý proces má vlastnú kópiu registrov a dátovej oblasti.

9.1.6 Segmentácia

V uvedených technikách boli všetky činnosti s operačnou pamäťou pre používateľský program „neviditeľné“. Vždy sme predpokladali lineárny a súvislý adresný priestor. Teraz sa zaoberáme otázkou, či je možný iný spôsob prístupu k adresnému priestoru, ktorý vedie k efektívnejšiemu využitiu pamäte a uľahčuje programovanie. Program je rozdelený na *segmenty* — logické zoskupenie informácií (napr. podprogramy alebo dátové oblasti), t.j. logické časti adresného priestoru. Segmenty nemusia mať rovnakú veľkosť, ale je daná maximálna veľkosť segmentu. *Pamäťový blok* je deliči priestor FAP, ľubovoľne dlhý. *Segmentovanie* je prideľovanie pamäťových blokov segmentom.

Členenie programu na segmenty môže previesť programátor manuálne alebo kompilátor automaticky (prvý pre globálne premenné, druhý pre stack, tretí pre funkcie, štvrtý pre lokálne premenné). Každý odkaz na adresu v pamäti musí obsahovať určenie segmentu a adresu v segmente.

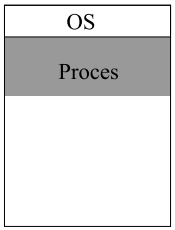
3.4 Členenie OS

Operačný systém delíme na 4 základné správy:

- správa procesov a procesora
- správa operačnej pamäte
- správa súborov
- správa periférií

Každá správa má nasledujúce základné funkcie:

- sledovať stav časti systému, ktorú má na starosti
- rozhodovať alebo plánovať prideľovanie spravovaného prostriedku
- prideľovať prostriedok
- uvoľňovať prostriedok



Takáto technika správy pamäti je typická pre jednoduché monoprogramové OS (FMS (Fortran Monitoring System pre 7094), mikropočítačové systémy, napr. CP/M).

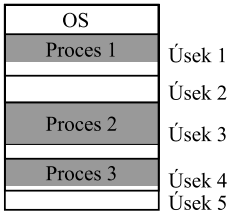
9.1.2 Statické súvislé úseky (Fixed partitions)

Operačná pamäť sa pri generovaní alebo zavádzaní systému rozdelí na pevný počet úsekov, ktoré sa počas behu OS nemenia. Do každého úseku môže byť zavedený jeden proces.

Úseky môžu byť buď rovnakej veľkosti alebo rôznej veľkosti. V prípade použitia úsekov rovnakej veľkosti, každý proces, ktorého veľkosť je menšia alebo rovná veľkosti úseku môže byť zavedený do ľubovoľného voľného úseku. Využitie pamäte je však v tomto prípade veľmi neefektívne.

Ak sú veľkosti úsekov rôzne, sú dve možnosti, ako prideliť procesu úsek pamäte: triviálna správa pamäte prideľuje procesu prvý voľný úsek s dostatočnou kapacitou, t.j. používa algoritmus „prvý vyhovujúci“ (*first-fit*). Iná možnosť je, že sa procesu pridelí ten voľný úsek, ktorý svojou kapacitou najmenej prevyšuje kapacitu pamäti požadovanú procesom, t.j. použitím algoritmu „najlepšie vyhovujúci“ (*best-fit*).

Pri použití stratégie best-fit môže mať každý úsek v pamäti vlastný zoznam, do ktorého sa zaraďujú prichádzajúce procesy čakajúce na tento úsek (veľkosťou je to najmenší úsek, do ktorého sa vojdú). Nevýhodou tohto prístupu je, že sa môže stať, že zoznam pre veľký úsek je prázdny, ale zoznam pre menší úsek je plný, a tak procesy zaradené v tomto zozname musia čakať, aj keď sú v pamäti voľné úseky. Preto je zrejme vhodnejšie zaraďovať procesy do jedného zoznamu a prideľovať im úseky podľa stratégie best-fit z momentálne neobsadených úsekov.



Pre transformáciu logickej adresy na fyzickú (zobrazenie LAP → FAP, LAP = logický adresný priestor, FAP = fyzický adresný priestor) sa najčastejšie používa mapovací register. Obsah mapovacieho registra (ten zodpovedá adrese 0 LAP) sa definuje až pri spúšťaní procesu.

Aby sa zabezpečila ochrana obsahu pamäti aj za úsekom s bežiacim procesom (pamäť pred týmto úsekom je chránená mapovacím registrom), je treba pre každý proces ešte druhý mapovací register (*hraničný register*). Ten obsahuje adresu za posledným pamäťovým miestom úseku.

Kľúčovým problémom návrhu prevádzkovej verzie operačného systému je voľba počtu a kapacity úsekov. Na ňu má vplyv predovšetkým charakter úloh riešených na danom počítači. Musí umožniť spracovanie aj práce s maximálnymi požiadavkami.

pamäte sa teda vyberie pripravený proces.

Takže do modelu stavov procesov pribudnú vlastne dva stavy:

- **Blokovaný, odswapovaný**: proces na swap disku čakajúci na nejakú udalosť.
- **Pripravený, odswapovaný**: proces na swap disku pripravený na vykonávanie hneď, ako bude nahratý do hlavnej pamäte.

Pribudli aj nové prechody medzi stavmi:

- **Blokovaný → Blokovaný, odswapovaný**: Ak nie sú žiadne pripravené procesy, aspoň jeden blokovaný proces je odswapovaný, aby uvoľnil miesto v pamäti. Toto odsúvanie je možné robiť aj keď sú pripravené procesy, ale je zlá výkonnosť systému.
- **Blokovaný, odswapovaný → Pripravený, odswapovaný**: ak nastala udalosť, na ktorú proces čakal. Všimnime si, že to vyžaduje, aby mal operačný systém prístup k informácii o stave odswapovaných procesov.
- **Pripravený, odswapovaný → Pripravený**: Keď v pamäti nie je žiadny pripravený proces, operačný systém nahrá nejaký proces do pamäte, aby vykonávanie pokračovalo. Môže sa tiež stať, že proces v stave Pripravený, odswapovaný má vyššiu prioritu ako pripravené procesy v pamäti. Operačný systém môže rozhonúť, že je dôležitejšie nahrať procesy s vyššou prioritou, než minimalizovať swapovanie.
- **Pripravený → Pripravený, odswapovaný**: Zvyčajne operačný systém preferuje odswapovanie blokovaných procesov. Niekedy môže byť potrebné odsunúť aj pripravený proces, napr. ak je to jediný spôsob, ako uvoľniť dostatočne veľký úsek pamäte. Alebo operačný systém sa môže rozhodnúť odswapovať pripravený proces s nižšou prioritou radšej ako blokovaný proces s vyššou prioritou, ak predpokladá, že blokovaný proces sa skoro stane pripraveným.
- **Nový → Pripravený, odswapovaný**: Keď je vytvorený nový proces, môže byť zaradený do Zoznamu pripravených procesov alebo do Zoznamu pripravených odswapovaných procesov (keď nie je v pamäti dosť miesta pre nový proces).

Samotný proces má kontrolu nad niektorými stavovými prechodmi na užívateľskej úrovni:

1. Proces môže vytvoriť nový proces, ktorý začína v stave Nový. Na ďalší prechod novovytvoreného procesu (zo stavu Nový do stavu Pripravený) má už vplyv len operačný systém.
2. Proces môže vykonať systémové volanie, čím prejde zo stavu Bežiaci do stavu Blokovaný. Nemá však už vyplyv na to, kedy (a či vôbec) sa vráti zo systémového volania. Rôzne udalosti môžu spôsobiť, že proces prejde do stavu Ukončený (predčasné ukončenie procesu).
3. Proces môže dobrovoľne skončiť systémovým volaním `exit`.

Všetky ostatné prechody sú riadené operačným systémom podľa určitých pevných pravidiel.

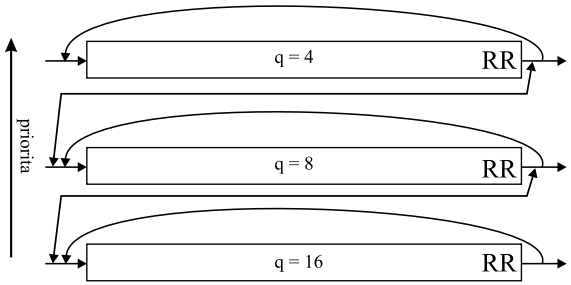
Žiadna úloha zoznamu Batch sa nemôže vykonávať pokiaľ nie sú ostatné zoznamy prázdne. Ak sa do zoznamu Akplikan programy zaradí úloha pokiaľ sa vykováva Batch, vykonávanej úlohe sa odoberie procesor. Iná možnosť je rozdelenie času medzi zoznamami: každý zoznam dostane jednu časť času CPU, ktorú môže plánovať medzi procesmi v zozname.

Stratégia niekoľkých zoznamov s premiestnením (multilevel feedback queues)

Normálne v plánovacom algoritme s niekoľkými zoznamami úlohy zostávajú priradené jednému zoznamu. Zoznamy s premiestnením umožňujú, aby úloha prechádzala z jedného zoznamu do druhého na základe stanovených kritérií.

Príklad:

V tomto príklade je základná idea v oddelení úloh, ktoré majú rôzne charakteristiky vzhľadom k intervalom použitia CPU.



Nová úloha sa zaraďuje do prvého zoznamu. Tento používa stratégiu RR s určitým časovým kvantom. Ak úloha plne vyčerpá pridelené kvantum (teda neopustí procesor dobrovoľne napr. kvôli V/V operácii), je zaradená do zoznamu s nižšou prioritou, ale dlhším kvantom. Takýmto spôsobom môže postupne klesať dole. Naopak, úlohy v zoznamoch s nižšou prioritou, ktoré nedočerpajú pridelené kvantum, budú zaradené do zoznamu s vyššou prioritou. Teda ak úloha požaduje veľa času CPU, prechádza do zoznamu s nižšou prioritou, ale interaktívne úlohy alebo úlohy intenzívne vo využívaní V/V prostriedkov zostávajú v zoznamoch s vysokou prioritou.

Vo všeobecnosti sa takýto plánovač definuje na základe nasledovných parametrov:

- počet zoznamov
- plánovací algoritmus pre každý zoznam
- metóda, ktorá určuje, kedy sa úloha presunie do zoznamu s nižšou prioritou
- metóda, ktorá určuje, kedy sa úloha presunie do zoznamu s vyššou prioritou (keď je úloha príliš dlho v zozname s nízkou prioritou, môže sa premiestniť do zoznamu s vyššou prioritou)
- metóda, ktorá určuje, do ktorého zoznamu sa zaradí úloha, keď vstupuje do zoznamu pripravených procesov

8.3 Policy versus mechanism (princípy a pravidlá rozhodovania versus mechanizmus)

Doteraz sme predpokladali, že všetky procesy v systéme patria rôznym užívateľom, a teda „súťažia“ o CPU. Niekedy sa však môže stať, že jeden proces má mnoho procesov-potomkov, ktoré bežia pod jeho riadením (napr. proces pre správu databázového systému má veľa potomkov, každý pracuje na rôznej požiadavke alebo vykonáva nejakú špecifickú funkciu: zaradenie do fronty, prístup na disk a pod.) a je


```
enter_region:
        tsl register, flag      ! skopíruj flag do register, nastav flag = 1
        cmp register,#0         ! je flag = 0 ?
        jnz enter_region        ! ak je flag <> 0, je uzamknuté — čakaj
        ret                     ! návrat do volajúcej funkcie — vstup do
                                ! kritického úseku
leave_region:
        mov flag,#0             ! vlož 0 do flag
        ret                     ! návrat
```

Obr. 5.1: Inštrukcia TSL

```
P0:  while (TRUE) {
       while (turn != 0);       /* wait */
         critical_section();
         turn = 1;
         noncritical_section();
     }

P1:  while (TRUE) {
       while (turn != 1);       /* wait */
         critical_section();
         turn = 0;
         noncritical_section();
     }
```

Obr. 5.2: Striktné striedanie procesov P_0 a P_1

Softwarové riešenia

Tieto riešenia zvyčajne predpokladajú elementárne vzájomné vylúčenie na úrovni prístupu do pamäte (simultánny prístup na to isté pamäťové miesto je sériovaný správou pamäte), inak nie je potrebná žiadna podpora na úrovni hardwaru, operačného systému alebo programovacieho jazyka.

Uzamykacie premenné

Máme jednu zdieľanú *uzamykaciu* premennú, inicializovanú na hodnotu 0. Keď chce proces vstúpiť do kritického úseku, najprv testuje zámok. Ak má tento hodnotu 0, nastaví ho na 1 a vojde do kritického úseku. Ak je hodnota zámku 1, proces čaká. Môže však nastať rovnaká chyba ako v prípade spooler adresára.

Striktné striedanie

Algoritmy procesov pozri na obrázku 5.2. Celočíselná premenná `turn` je inicializovaná na 0. Proces P_i môže vstúpiť do kritického úseku len vtedy, keď je premenná `turn` nastavená na i, v opačnom prípade čaká (while cyklus). Pri opúšťaní kritického úseku proces prepne premennú `turn` na hodnotu, ktorá umožní vstup druhému procesu. Takýmto spôsobom sa procesy striedajú vo využívaní kritického úseku. Ak je jeden proces rýchly a druhý pomalý, môže sa stať, že pomalý proces, pracujúci momentálne vo svojej nekritickej časti, bráni vstupu do kritického úseku rýchlemu procesu (premenná `turn` je nastavená tak, že vstúpiť môže len pomalý proces). Porušuje sa tým 3. podmienka pre problém vylúčenia.

8.2.1 Nepreemptívne (nonpreemptive) plánovacie algoritmy

Keď proces prejde do stavu "bežiaci", vykonáva sa až kým neskončí alebo sa sám zablokuje (napr. čaká na V/V alebo požaduje službu operačného systému).

Stratégia FCFS (First Come First Served)

Vhodná aj pre plánovač úloh, aj pre plánovač procesov.

- Poradie obsluhy požiadaviek je dané poradím ich príchodu.
- Implementácia sa realizuje pomocou radu FIFO (jednoduché).
- Zvyčajne dosť malá výkonnosť.

Proces	Čas zadania	Čas spracovania (T_s)	Čas spustenia	Čas ukončenia	Doba prechodu (T_q)	$\frac{T_q}{T_s}$
1	0	3	0	3	3	1.00
2	2	6	3	9	7	1.17
3	4	4	9	13	9	2.25
4	6	5	13	18	12	2.40
5	8	2	18	20	12	6.00
Priemer					8.60	2.56

Okrem doby prechodu procesu systémom v tabuľke vidíme aj *normalizovanú dobu prechodu (normalized turnaround time)* – podiel doby prechodu k dobe spracovania. Táto hodnota udáva relatívne opozdenie procesu. Zvyčajne čím je dlhší čas spracovania procesu, tým väčšie opozdenie je možné tolerovať. Minimálna možná hodnota tohto podielu je 1 (proces bol spustený hneď ako bol zadaný), rastúce hodnoty zodpovedajú klesajúcej úrovni obsluhy procesu.

Priemerná doba prechodu vo FCFS vo všeobecnosti nie je minimálna a môže dosť variovať.

FCFS lepšie pracuje pre dlhšie procesy ako pre kratšie. Majme takýto príklad:

Proces	Čas zadania	Čas spracovania (T_s)	Čas spustenia	Čas ukončenia	Doba prechodu (T_q)	$\frac{T_q}{T_s}$
1	0	1	0	1	1	1
2	1	100	1	101	100	1
3	2	1	101	102	100	100
4	3	100	102	202	199	1.99
Priemer					100	26

Normalizovaná doba prechodu pre proces 3 je netolerovateľná: celkový čas, ktorý proces strávi v systéme, je 100 krát väčší ako požadovaný čas vykonávania. Toto nastane vždy, keď malý proces príde tesne za veľkým procesom. Na druhej strane vidíme aj na tomto extrémnom príklade, že dlhé procesy "dopadli" celkom dobre. Proces 4 má síce dobu prechodu takmer dvojnásobnú oproti procesu 3, ale jeho normalizovaná doba prechodu (vyjadrujúca dobu čakania) je menšia ako 2.

Hoci monitory poskytujú ľahký spôsob na dosiahnutie vzájomného vylúčenia, nie je to ešte dostačujúce — potrebujeme spôsob na zablokovanie procesov, keď nemôžu byť vykonávané. Na to sú tu zavedené premenné typu *podmienka* (condition variables) spolu s dvoma operáciami na nich wait a signal. Keď procedúra monitora zistí, že nemôže pokračovať, vykoná wait na nejakej premennej typu podmienka — tým bude volajúci proces zablokovaný. To súčasne umožní inému procesu, ktorý predtým nemohol vstúpiť do monitora, aby doň vstúpil. Tento druhý proces môže zobudiť spiaci proces vykonaním signal na premennej typu podmienka, na ktorej spiaci proces čaká. Aby sme zabránili tomu, že by boli v monitore dva aktívne procesy v tom istom čase, potrebujeme pravidlo, ktoré určuje, čo sa vlastne stane po signal-e:

- Hoare navrhoval nechať zobudený proces bežať a druhý proces pozastaviť.
- Brinch Hansen požadoval, aby proces vykonávajúci signal opustil ihneď monitor, t.j. signal sa smie vyskytnúť len ako posledný príkaz procedúry monitora (budeme používať tento návrh — je koncepčne jednoduchší a ľahší na implementáciu).

Ak sa signal vykoná na premennej, na ktorú čaká viac procesov, len jeden z nich bude oživený (určený systémovým plánovačom).

Aj použitie monitora si demonštrujeme na probléme producenta a konzumenta (obr. 5.6).

Wait a signal sú podobné sleep a wakeup, ale je tu jeden rozdiel: sleep a wakeup môžu zlyhať, pretože jeden proces sa pokúša „zaspať" a druhý zase „zobudiť". S monitormi sa to nemôže stať — automatické vzájomné vylúčenie zabezpečuje, že keď je napr. producent v monitore a zistí, že buffer je plný, je schopný dokončiť wait operáciu bez obavy, že plánovač môže prepnúť na konzumenta pred jej ukončením.

Na realizovanie monitorov potrebujeme programovací jazyk, ktorý ich má zabudované (napr. Concurrent Euclid, 1983), kým na realizáciu semafórov stačí pridať dve assembler-rutiny do knižnice — užívateľské programy potom môžeme písať v Pascale alebo v jazyku $\mathcal{C}$.

Ďalší problém s monitormi a semafórmi je, že boli vyvinuté na riešenie problému vzájomného vylúčenia na 1 alebo viac CPU, ktoré majú všetky prístup k spoločnej pamäti. Avšak v distribuovanom systéme pozostávajúcom z viacerých CPU (každý so svojou vlastnou pamäťou), spojených lokálnou sieťou, sú tieto prostriedky nepoužiteľné. Je navyše potrebné niečo na výmenu informácií medzi počítačmi (výmena správ).

5.3 Komunikácia medzi procesmi

Posielanie správ

Tento spôsob komunikácie používa primitívy (operácie) send a receive, ktoré sú systémovými volaniami a môžu byť ľahko pridané do knižničných procedúr (podobne ako semafóry):

- send(cieľ, &správa)
- receive(zdroj, &správa)

Pri návrhu systému posielania správ je treba vyriešiť množstvo otázok, ktorými sa budeme zaoberať v ďalšom výklade:

- *Synchronizácia*
  - Send: blokovaný, neblokovaný
  - Receive: blokovaný, neblokovaný, test na prítomnosť správy
- *Adresovanie*
  - Priame: symetrické, nesymetrické
  - Nepriame: statické, dynamické, vlastníctvo
- *Formát*
  - Obsah

3. Ak také i existuje, predpokladajme, že proces P_i požiada o všetky potrebné prostriedky a skončí. Algoritmus označí proces P_i ako ukončený a pripočíta všetky jeho prostriedky k vektoru W. Čiže $w_k = w_k + a_{ik}$ pre každé k.
4. Opakuje kroky 1 a 2, kým nenastane jedna zo situácií: všetky procesy sú označené ako ukončené – čo znamená, že začiatočný stav bol bezpečný – alebo kým nenastane uviaznutie – teda začiatočný stav nebol bezpečný.

V praxi je bankárov algoritmus takmer nepoužiteľný, pretože je ťažké očakávať od procesov, že budú vopred poznať množstvo potrebných prostriedkov. Ďalšie obmedzenie tohto algoritmu je v tom, že uvažuje fixný počet prideľovaných prostriedkov, a tiež žiadny proces nesmie skončiť bez uvoľnenia prostriedkov.


```
#include "prototypes.h"
#define N 100                                   /* počet položiek v buffri */
#define MSIZE 4                                 /* velkosť správy */
typedef int message[MSIZE];

void producer(void)
{ int item;
  message m;
  while (TRUE) {
      produce_item(&item);
      receive(consumer, &m);                    /* čakanie na prázdnu správu */
      build_message(&m, item);
      send(consumer, &m);
  }
}

void consumer(void)
{ int item, i;
  message m;
  for (i = 0; i < N; i++) send(producer, &m);   /* N prázdnych */
  while (TRUE) {
      receive(producer, &m);
      extract_item(&m, &item);
      send(producer, &m);                       /* späť prázdnu */
      consume_item(item);
  }
}
```

Obr. 5.7: Problém producenta/konzumenta s posielaním správ

1. Označ každý proces, ktorý má v matici A nulový riadok.
2. Inicializuj pomocný vektor **W** rovný vektoru **V**.
3. Nájdi index i taký, že proces i je neoznačený a i-ty riadok v P je menší alebo rovný **W**. Čiže $p_{ik} \leq w_k$, pre $1 \leq k \leq m$. Ak taký riadok neexistuje, ukonči algoritmus.
4. Ak bol taký riadok nájdený, označ proces i a pripočítaj príslušný riadok matice A k **W**. Teda $w_k = w_k + a_{ik}$. Vráť sa na krok 3.

Uviaznutie nastáva vtedy a len vtedy, ak po ukončení algoritmu existujú neoznačené procesy. Každý neoznačený proces je uviaznutý. Stratégiou tohto algoritmu je nájsť proces, ktorého požiadavky na prostriedky môžu byť uspokojené dostupnými prostriedkami. Ďalej algoritmus predpokladá, že tomuto procesu budú prostriedky pridelené a že proces skončí a vráti všetky prostriedky. Potom algoritmus hľadá ďalší proces, ktorý môže byť uspokojený.

Príklad:

Matica pridelených prostriedkov

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Matica požiadaviek (ešte potrebných prostriedkov)

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Vektor nepridelených (ešte voľných) prostriedkov

R1	R2	R3	R4	R5
0	0	0	0	1

Algoritmus pracuje takto:

1. Označí P4, lebo P4 nemá pridelené žiadne prostriedky.
2. Nastaví **W** = (0 0 0 0 1).
3. Požiadavka procesu P3 je menšia alebo rovná ako **W**, preto označí P3 a nastaví **W** = **W** +(0 0 0 1 0) = (0 0 0 1 1).
4. Skončí.

Procesy P1 a P2 sú neoznačené, čiže sú uviaznuté.

Keď operačný systém detekuje uviaznutie, treba ho nejako riešiť. Možné sú viaceré prístupy:

- Zrušiť všetky uviaznuté procesy.
- Vrátiť všetky uviaznuté procesy do nejakého definovaného *kontrolného bodu (checkpoint)* (v ktorom je stav procesu zapísaný do súboru) a reštartovať všetky procesy. Čiže systém musí poskytovať mechanizmus návratu programu (rollback) a reštartovania. Problémom tohto prístupu je, že sa opätovne môže objaviť pôvodné uviaznutie.
- Vybrať proces spomedzi uviaznutých procesov (obeť), ktorý bude ukončený. Ak sa uviaznutie odstránilo, možno pokračovať. Ak nie, je nutné vybrať ďalšiu obeť. Pri výbere obete hrá úlohu viacero faktorov: priorita, rozpracovanosť, počty a druhy pridelených prostriedkov, súvislosť procesu s ostatnými procesmi, atď.
- Postupne prerozdeľovať prostriedky, kým sa neodstráni uviaznutie. Proces, ktorému boli odňaté prostriedky, sa musí vrátiť do bodu pred pridelením odňatých prostriedkov.


```
#define N 5
#define LEFT (i-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;                    /* vzájomné vylúčenie pri práci s poľom state */
semaphore s[N];                         /* inicializované na 0, pre filozofov, nie pre vidličky */

void philosopher(int i)
{ while (TRUE) {
      think();
      take_forks(i);                    /* uchop obe vidličky alebo prejdi do stavu blokovaný */
      eat();
      put_forks(i);                     /* polož obe vidličky */
  }
}

void take_forks(int i)
{ down(&mutex);                         /* vojdi do kritického úseku */
  state[i] = HUNGRY;                    /* zaznač fakt, že filozof i je hladný */
  test(i);                              /* skús chytiť vidličky */
  up(&mutex);                           /* výstup z kritického úseku */
  down(&s[i]);                          /* zablokuj sa, ak vidličky neboli voľné */
}

void put_forks(int i)
{ down(&mutex);                         /* vojdi do kritického úseku */
      state[i] = THINKING;              /* filozof i dojedol */
      test(LEFT);                       /* pozri, či ľavý sused môže jesť */
      test(RIGHT);                      /* pozri, či pravý sused môže jesť */
      up(&mutex);                       /* výstup z kritického úseku */
}

void test(int i)
{ if (state[i] == HUNGRY && state[LEFT] != EATING
                         && state[RIGHT] != EATING {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

Obr. 6.3: Problém obedujúcich filozofov

```
region v do
  begin
    S1;                          /* vykoná sa po vstupe do krit. regiónu; nemusí tam byť nič */
    await(B);
    S2;
  end;
```

pričom príkaz await(B) vyhodnotí B. Ak je B nepravdivé, čaká sa, kým je B pravdivé a nie je žiaden proces v kritickom úseku spojenom s v.

V prípade čitateľov a zapisovateľov problém vyžaduje, aby keď je zapisovateľ pripravený, mohol zapisovať hneď, ako je to možné. Teda čitateľ môže vojsť do svojho kritického úseku, len keď v kritickom úseku nie je žiadny zapisovateľ a ani nie sú pripravení žiadni zapisovatelia (obr.6.6).


```
var v: shared record
                 nreaders, nwriters: integer;
                 busy: boolean;
               end;

procedure open_read;
begin
  region v do
    begin
      await(nwriters = 0);
      nreaders := nreaders + 1;
    end;
end;

procedure close_read;
begin
  region v do
    begin
      nreaders := nreaders - 1;
    end;
end;

procedure open_write;
begin
  region v do
    begin
      nwriters := nwriters + 1;
      await((not busy) and (nwriters = 0));
      busy := true;
    end;
end;

procedure close_write;
begin
  region v do
    begin
      nwriters := nwriters - 1;
      busy := false;
    end;
end;

begin
  busy := false;
  nreaders := 0;
  nwriters := 0;
end.
```

Obr. 6.6: Problém čitateľov/zapisovateľov

```
#define N 5
semaphore vidlicka[N]

void filozof(int i)
{ while (TRUE) {
      mysli();
      down(&vidlicka[i]);                          /* vezmi ľavú vidličku */
      down(&vidlicka[(i+1)%N]);                    /* vezmi pravú vidličku */
      jedz();
      up(&vidlicka[i]);                            /* polož ľavú vidličku */
      up(&vidlicka[(i+1)%N]);                      /* polož pravú vidličku */
  }
}
```

Obr. 6.2: Algoritmus pre filozofa (obvyklé riešenie) - s použitím semafórov

Možnosti riešenia:

(a) Môžeme dovoliť maximálne 4 filozofom, aby si sadli k stolu.

(b) Dovolíme, aby filozof uchopil vidličky, len ak sú obe voľné.

(c) Asymetrické riešenie: 1 filozof uchopí najprv ľavú vidličku a potom pravú, iný zase naopak.

(d) Môžeme modifikovať program tak, že po chytení ľavej vidličky program preverí, či je pravá k dispozícii. Ak nie, filozof položí ľavú vidličku a chvíľu počká — potom proces opakuje. Môže sa však stať, že všetci filozofovia naraz uchopia ľavú vidličku, naraz ju položia, počkajú, opäť naraz uchopia, atď. Stav, keď program pokračuje do nekonečna, ale zlyhá bez akéhokoľvek postupu sa nazýva *vyhladovanie* (*starvation*).

(e) Mohli by sme nechať filozofov čakať náhodný čas (nie ten istý) — pravdepodobnosť, že by nastala opísaná situácia, je veľmi malá. Niekedy však potrebujeme algoritmus, ktorý funguje vždy a nezlyhá kvôli nepravdepodobnej postupnosti náhodných čísel.

(f) Zaviesť binárny semafor — keď niektorý filozof ide jesť, musí vykonať operáciu down, po položení vidličiek vykoná up. Teda len 1 filozof môže jesť v ľubovoľnom čase (kým teoreticky môžu jesť 2).

(g) Riešenie umožňujúce maximálny paralelizmus pre ľubovoľný počet filozofov: Použijeme pole state na udržiavanie informácie, či filozof je, myslí alebo je hladný (pokúša sa chytiť vidličky). Filozof môže prejsť do stavu *jediaci*, len keď žiaden z jeho susedov neje. Susedia filozofa i sú definovaní makrami LEFT a RIGHT (viď. obr.6.3).

Poznámka: Uvedené riešenie zabráni uviaznutiu, ale môže viesť k vyhladovaniu (UKÁŽTE!).

6.2 Problém čitateľov a zapisovateľov

Problém 5 filozofov je užitočný na modelovanie procesov, ktoré sú konkurujúce vo výlučnom prístupe k obmedzenému množstvu prostriedkov, ako páskové jednotky alebo iné V/V zariadenia. Problém čitateľov a zapisovateľov (r. 1971, Courtois) modeluje prístup do bázy dát. Predstavme si veľkú bázu dát (napr. rezervačný systém v aerolíniách) s množstvom procesov, ktoré do nej môžu zapisovať a čítať z nej. V istom čase môže databázu čítať viac procesov, ale ak 1 proces zapisuje do databázy, žiaden iný proces do nej nemá prístup. Riešenie problému pomocou semafórov vidíme na obr.6.4.

Prvý čitateľ, ktorý získa prístup do databázy, vykoná down na semafóre databázy. Až keď posledný čitateľ dočíta, vykoná up a uvoľní blokovanému zapisovateľovi (ak nejaký je), vstup do databázy. V tomto riešení čitatelia majú väčšiu prioritu ako zapisovatelia.

Metóda detekcie a vyvedenia z uviaznutia sa používa často v batch systémoch, kde je ukončenie a reštartovanie procesu zvyčajne akceptovateľné.

7.3 Prevencia

Prevencia je neumožnenie jednej zo 4 podmienok uviaznutia:

1. **Vzájomné vylúčenie** — prostriedok nie je výlučne pridelený jednému procesu. To môže spôsobiť chaos, napr. pri tlači. Riešením je *spooling* — viaceré procesy môžu generovať výstup v tom istom čase. Jediný proces, ktorý žiada o tlačiareň je tlačový daemon, ktorý nikdy nepožaduje iné prostriedky. Tým eliminujeme uviaznutie pre tlačiareň. Avšak nie všetky zdieľané prostriedky môžu používať spooling (napr. tabuľka procesov). Ďalej uviaznutie môže vzniknúť pri zapĺňaní priestoru disku určeného na spooling v prípade, že tlačový daemon je naprogramovaný tak, že začína tlač, až keď je k dispozícii celý výstup.
2. **Postupné získavanie prostriedkov** — mohli by sme žiadať, aby proces pred začatím vykonávania získal všetky prostriedky, ktoré bude potrebovať. Problémom je, že mnohé procesy nevedia, koľko prostriedkov budú potrebovať počas behu. Ďalej, prostriedky nie sú využívané optimálne. Iná možnosť je požadovať od procesu žiadajúceho prostriedok, aby uvoľnil všetky prostriedky, ktoré práve drží. Až keď je požiadavka úspešná, môže dostať späť pôvodné prostriedky.
3. **Nemožnosť prerozdelenia prostriedkov** — dať možnosť odňať prostriedok procesu. Táto metóda môže byť používaná hlavne pre prostriedky, ktorých stav môže byť ľahko uložený (CPU registre, pamäťový priestor). Možné prístupy:
   - Ak proces držiaci nejaké prostriedky žiada iné prostriedky, ktoré nie sú voľné, tak musí uvoľniť prostriedky, ktoré má a ak to bude potrebné, vyžiadať si ich znova spolu s požadovanými novými prostriedkami.
   - Ak proces žiada prostriedky, ktoré nie sú voľné, hľadá sa, či ich nedrží iný proces, ktorý čaká na ďalšie prostriedky. Ak áno, prostriedky sa čakajúcemu procesu odoberú a pridelia žiadajúcemu. Ak nie, žiadajúci proces čaká a zatiaľ mu môžu byť odobrané prostriedky.
4. **Cyklické čakanie** — môže byť eliminované viacerými spôsobmi.
   - Pravidlo, ktoré hovorí, že proces môže mať v danom momente len jeden prostriedok. Ak potrebuje ďalší, musí prvý uvoľniť (nie je možné napr. pre proces, ktorý potrebuje kopírovať veľký súbor z pásky na tlačiareň)
   - Očíslovať všetky prostriedky, potom môžu procesy žiadať prostriedok kedykoľvek, ale v numerickom poradí. Preto nemôže nastať uviaznutie. (V ľubovoľnom momente má jeden z priradených prostriedkov najväčšie číslo. Proces, ktorý má tento prostriedok, nikdy nežiada o už pridelený prostriedok. Buď skončí alebo žiada o prostriedky s vyšším číslom — všetky sú vtedy dostupné. Keď skončí, uvoľní svoje prostriedky — vtedy nejaký iný proces drží prostriedok s najvyšším číslom atď.)
   - Obmena: nepožadujeme striktne, že prostriedky môžu byť žiadané len v rastúcom poradí, ale to, že proces nesmie žiadať prostriedok s nižším číslom, než tie, čo drží. Ak napr. proces žiadal prostriedok s č. 9 a 10, potom oba uvoľnil, vlastne môže žiadať od začiatku — nie je dôvod, aby nemohol žiadať prostriedok s č. 1.

Aj keď usporiadanie prostriedkov rieši problém uviaznutia, nie je prakticky možné nájsť usporiadanie, ktoré by úplne vyhovovalo všetkým procesom.

Ak vylúčime jednu z prvých troch podmienok uviaznutia, hovoríme o *nepriamej metóde prevencie uviaznutia*, kým *priama metóda prevencie uviaznutia* znamená zabránenie výskytu cyklického čakania.

Vzťah medzi posielajúcimi a prijímajúcimi procesmi môže byť: one-to-one ("súkromný" komunikačný kanál medzi dvoma procesmi), one-to-many (užitočné pre aplikácie, kde jedna správa má byť rozoslaná – *broadcast* – viacerým procesom), many-to-one (užitočné pre vzťah klient/server, kedy jeden proces poskytuje službu mnohým ďalším procesom. Schránka sa v tomto prípade nazýva tiež *port*.), many-to-many.

Priradenie procesov ku schránkam môže byť statické alebo dynamické. Porty sú často staticky spojené s príslušným procesom, čiže port je vytvorený a priradený procesu permanentne. Podobne vzťah one-to-one je zvyčajne definovaný staticky a permanentne. Keď je mnoho odosielateľov, pripojenie odosielateľa ku schránke môže byť dynamické (na tento účel slúžia napr. primitívy *connect* a *disconnect*).

Tiež je dôležitá otázka vlastníctva schránky. V prípade portu je schránka zvyčajne vytvorená a vlastnená prijímajúcim procesom. Takže, keď tento proces skončí, schránka je zrušená. Vo všeobecnosti môže operačný systém poskytovať službu na vytváranie schránok. Schránka môže byť chápaná ako vlastníctvo procesu, ktorý ju vytvoril, a teda zaniká pri ukončení procesu, alebo je schránka vlastníctvom operačného systému a na jej zrušenie treba použiť explicitný príkaz.

Formát správ

Správy môžu byť pevnej (fixnej) alebo premenlivej (variabilnej) dĺžky.

Typický formát správ variabilnej dĺžky je: "header", obsahujúci informáciu o správe – typ správy, identifikáciu cieľa, identifikáciu odosielateľa, dĺžku správy, riadiacu informáciu, napr. priorita, poradové číslo správy a pod. – a "body", vlastný obsah správy.

Zaraďovanie správ

Najjednoduchší spôsob zaraďovania správ je FIFO – first-in-first-out, čo však nemusí byť postačujúce, ak sú niektoré správy dôležitejšie ako ostatné. V takom prípade je možné zaviesť priority správ na základe typu správy alebo určenia odosielateľa. Ďalšia možnosť je umožniť prijímajúcemu procesu prezrieť zoznam čakajúcich správ a vybrať, ktorá správa bude prijatá ako nasledujúca.

Problémy designu pre posielanie správ

Posielanie správ má niektoré problémy, ktoré sa neobjavujú u semafórov alebo monitorov, hlavne ak komunikujúce procesy sú na rôznych počítačoch prepojených sieťou. Napr., správy sa môžu v sieti stratiť: je možné, aby sa odosielateľ a adresát dohodli, že hneď po prijatí správy sa posiela špeciálna „potvrdzovacia správa" — *acknowledgement message* (ak ju odosielateľ nedostane do istého času — pošle správu znova).

Systém správ musí tiež riešiť otázku, ako sú procesy pomenované, aby ich určenie bolo jednoznačné — zvyčajne proces@počítač alebo počítač:proces.

Riešenie problému producenta a konzumenta pomocou posielania správ je uvedené na obr 5.7.

Predpokladajme, že všetky správy majú rovnakú veľkosť a že odoslané, ale zatiaľ neprijaté správy sú bufrované automaticky operačným systémom. Konzument začne tým, že pošle producentovi N prázdnych správ. Kedykoľvek má producent položku k dispozícii pre konzumenta, vezme 1 prázdnu správu a pošle späť plnú. Týmto spôsobom celkový počet správ v systéme zostáva konštantný, teda môžu byť uložené v danom pamäťovom priestore. Ak producent pracuje rýchlejšie ako konzument, všetky správy sa naplnia a producent bude blokovaný a čaká na prázdnu správu od konzumenta. Ak pracuje rýchlejšie konzument, situácia je opačná.

5.3.1 Pipe (rúra)

V Unixe sa komunikácia medzi užívateľskými procesmi realizuje aj prostredníctvom *pipe*, čo sú vlastne mailboxy s tým rozdielom, že pipe neudržuje hranice správ. Ak teda odosielateľ pošle 10 správ po 100

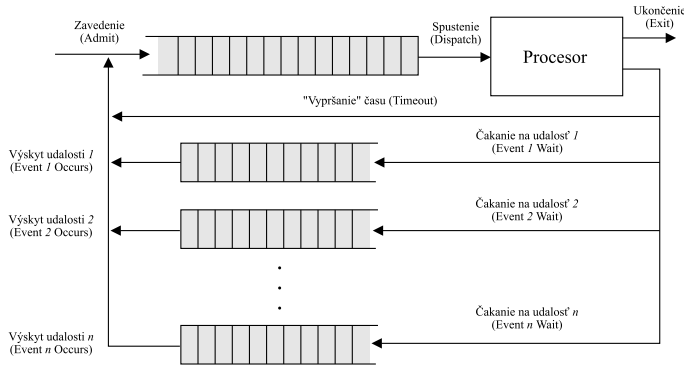
Kapitola 8

Správa procesov a procesora

Jedným z najdôležitejších princípov moderných OS je multiprogramovanie, teda rôzne programy, ktoré sa nachádzajú v pamäti v tom istom čase, môžu zdieľať CPU. Toto zvyšuje využitie CPU a *priepustnosť* (*throughput*) systému, t.j. množstvo úloh realizovaných v danom časovom intervale.

Cieľom multiprogramovania je mať v ľubovoľnom okamihu nejaký proces bežiaci (vykonávaný), aby sa maximalizovalo využitie CPU. V monoprocesorovom systéme môže byť bežiaci maximálne jeden proces, ostatné musia čakať na CPU. Pripravené procesy, ktoré čakajú na spracovanie, sa udržiavajú v zozname nazývanom *zoznam pripravených procesov (Ready queue)*. Tento zoznam nemusí byť nutne rad FIFO, ale vzhľadom na rôzne plánovacie algoritmy to môže byť rad s prioritami, strom alebo aj neusporiadaný zoznam. V systéme sú aj ďalšie zoznamy — *zoznamy prostriedkov*, t.j. zoznamy procesov čakajúcich na daný prostriedok. Každý prostriedok má svoj vlastný zoznam.

Proces vstupuje do systému zvonku a umiestni sa do zoznamu pripravených procesov. V ňom čaká, pokiaľ nie je vybratý na spracovanie. Keď musí čakať na V/V — zaradí sa do príslušného zoznamu prostriedku. Keď je obslúžený, opäť sa zaradí do zoznamu pripravených procesov. Proces pokračuje v tomto cykle CPU–V/V, až kým neskončí a neopustí systém.



8.1 Plánovače

OS má množstvo plánovačov. Pre plánovanie CPU sú 2 hlavné plánovače:

- *Plánovač úloh* (plánovač vyššej úrovne, job scheduler, long-term scheduler), tj plánovač na úrovni správy úloh
- *Plánovač procesov* (plánovač nižšej úrovne, CPU scheduler, process scheduler, short-time scheduler), t.j. plánovač na úrovni prideľovania procesora

```
#define N 100                                   /* veľkosť buffra */
typedef int semaphore                           /* semafóry sú špecializáciou typu int */
semaphore mutex = 1;                            /* riadi prístup do kritického úseku */
semaphore empty = N;                            /* počíta prázdne položky v buffri */
semaphore full = 0;                             /* počíta plné položky v buffri */

void producer()
{ while (TRUE) {
      produce_item();                           /* produkuj položku */
      down(&empty);                             /* zníž počítadlo empty */
      down(&mutex);                             /* vstup do kritického úseku */
      enter_item();                             /* vlož položku do buffra */
      up(&mutex);                               /* von z kritického úseku */
      up(&full);                                /* zvýš počítadlo full */
      }
}

void consumer()
{ while (TRUE) {
      down(&full);                              /* zníž počítadlo full */
      down(&mutex);                             /* vstup do kritického úseku */
      remove_item();                            /* vezmi položku z buffra */
      up(&mutex);                               /* von z kritického úseku */
      up(&empty);                               /* zvýš počítadlo empty */
      consume_item();                           /* spracuj položku */
      }
}
```

Obr. 5.5: Semafóry

Operácie down a up sú vykonané ako jednoduché nedeliteľné atomické akcie. Počas ich vykonávania nemá žiaden iný proces prístup k semafóru.

Aj tento prístup je demonštrovaný na probléme producenta a konzumenta (obr. 5.5). Semafóry sú v ňom použité dvoma spôsobmi: jednak na ošetrenie problému plného alebo prázdneho buffera a ďalej na zabezpečenie vzájomného vylúčenia pri prístupe do buffera (ktorý je zdieľaný).

Semafóry riešia problém strateného wakeup-u. Zvyčajný spôsob ich realizácie je implementovať operácie down a up ako systémové volania, pričom sú znemožnené prerušenia počas ich vykonávania.

V algoritme sa používa semafór *mutex* (inicializovaný na hodnotu 1), ktorý zabezpečuje, aby do kritického úseku mohol vstúpiť vždy len jeden proces. Tento semafór nadobúda len hodnoty 1 alebo 0, preto sa nazýva *binárny semafór*.

Monitory

Semafóry sú veľmi primitívne prostriedky na riadenie koordinácie procesov (je dosť zložité písať správne algoritmy). Prostriedkami vyššej úrovne sú *monitory* (navrhnuté v roku 1974 – Hoare a 1975 – Brinch Hansen). Monitor je množina procedúr, premenných a dátových štruktúr zjednotených do špeciálneho druhu modulu alebo balíka.

Procesy môžu volať procedúry monitora kedy chcú, ale nemôžu priamo pristupovať k vnútorným dátovým štruktúram monitora z procedúr deklarovaných mimo monitora. Dôležitá vlastnosť, ktorá robí monitor užitočným na dosiahnutie vzájomného vylúčenia, je, že len jeden proces môže byť aktívny v monitore v ľubovoľnom momente (kompilátor môže obsluhovať volania procedúr odlišne od iných volaní procedúr — zvyčajne sa na to využíva binárny semafór).

Stratégia SJF (Shortest Job First)

Uprednostňuje riešenie kratších požiadaviek (s kratším predpokladaným časom spracovania) pred dlhšími, čím minimalizuje doby čakania.

Proces	Čas zadania	Čas spracovania (T_s)	Čas spustenia	Čas ukončenia	Doba prechodu (T_q)	$\frac{T_q}{T_s}$
1	0	3	0	3	3	1.00
2	2	6	3	9	7	1.17
3	4	4	11	15	11	2.75
4	6	5	15	20	14	2.80
5	8	2	9	11	3	1.50
Priemer					7.60	1.84

Táto stratégia je optimálna v zmysle, že dáva minimálny priemerný čas čakania pre daný súbor úloh. Skúsenosť ukazuje, že ak sa preferuje krátka úloha pred dlhšou, redukuje sa čas čakania krátkej úlohy viac než rastie čas čakania dlhšej úlohy. Preto priemerný čas čakania (a teda aj doba prechodu) klesá. Problém však je poznať dĺžku nasledujúcej požiadavky na CPU.

Táto stratégia sa dá použiť na plánovanie úloh, kedy odhad dĺžky spracovania zadáva zadávateľ úlohy. V tomto prípade je potrebné rozhodnúť, ako penalizovať úlohy, ak odhadovaná doba spracovania bude prekročená (cenou strojového času, ukončenie úlohy, odsunutie úlohy na koniec zoznamu pripravených úloh a pod.)

Alebo je možné robiť odhad času ďalšieho použitia CPU na základe predošlých použití. To je vhodné pre plánovanie procesov.

Priorita

- SJF stratégia je špeciálny prípad všeobecného algoritmu plánovania podľa priority ($p = 1/r$, p = priorita, r = dĺžka použitia CPU).
- Každá úloha má priradenú prioritu a CPU sa prideľuje úlohe s najvyššou prioritou.
- Úlohy s tou istou prioritou sa plánujú podľa FCFS.
- Priority sa môžu definovať interne alebo externe. Priority definované interne používajú isté merateľné veličiny na výpočet priority procesu (napr. obmedzenia času, požiadavky na pamäť, počet otvorených súborov atď.). Priority definované externe sa určujú na základe kritérií vzdialených od OS, napr. koľko sa platí za použitie počítača, katedra, ktorá zadáva úlohu a iné externé faktory.
- Dôležitým problémom plánovania podľa priority je nebezpečenstvo trvalého zablokovania úloh s nižšími prioritami v prípade, že sa systém zahltí požiadavkami na spracovanie s vyššími prioritami. Jedným možným spôsobom riešenia tohto problému je *starnutie* (*aging*). To je technika, ktorá zvyšuje prioritu úloh, ktoré dlho čakajú v systéme.

Strategia Highest response-ratio next (HRN)

(ratio značí pomer, podiel odpovedí)

- Priorita úlohy nie je len funkciou času použitia CPU, ale aj času čakania.
- Dynamické priority v HRN sú určené vzťahom:

$$\text{priorita (t.j. response-ratio)} = \frac{\text{čas čakania} + \text{čas spracovania}}{\text{čas spracovania}}$$

Pri out = 4 a in = 7 platí, že položky 0–3 sú prázdne (súbory boli vytlačené), 4–6 sú naplnené. Predpokladajme, že prakticky simultánne sa procesy A a B rozhodnú zaradiť súbor do tlače. Podľa „zákona schválnosti“ sa môže stať toto:

- Proces A číta premennú in a uloží hodnotu 7 do svojej lokálnej premennej next_free_slot.
- Nastane prerušenie od časovača a procesor sa prepne na proces B.
- Proces B číta premennú in, získa hodnotu 7, uloží meno tlačeného súboru do položky 7 a zvýši hodnotu premennej in na 8.
- Znova beží proces A, prezrie premennú next_free_slot, nájde hodnotu 7, teda zapíše meno tlačeného súboru do položky 7 (premaže meno od procesu B) a zvýši in na 8. Teda súbor, ktorý žiadal vytlačiť proces B nebude nikdy vytlačený.

Podobné situácie, kde dva alebo viac procesov číta alebo zapisuje zdieľané dáta a výsledok závisí od toho, v akom poradí procesy prebiehajú, sa nazývajú *race conditions* (časová závislosť procesov). Možnosť, ako predísť problémom v situáciách so zdieľaním prostriedkov, je nájsť spôsob, ako zakázať viac ako jednému procesu čítanie a zápis zdieľaných dát v tom istom čase. Inak povedané, potrebujeme *vzájomné vylúčenie* (mutual exclusion). Je to spôsob, ako zabezpečiť, že keď jeden proces používa zdieľané premenné, ostatné procesy toto nebudú mať dovolené. Problém predídenia „race conditions“ môže byť formulovaný abstraktne: časť času proces vykonáva interné výpočty a iné činnosti, ktoré nevedú ku konfliktom. Niekedy však proces môže pristupovať k zdieľanej pamäti alebo súborom, čo môže viesť ku konfliktom — táto časť programu sa nazýva *kritický úsek* (critical section). Ak nebudú nikdy dva procesy naraz vo svojich kritických úsekoch, zabráni sa vzniku *race conditions*.

Kritériá, ktoré musia platiť, aby bol vyriešený problém vylúčenia (podmienka na vylúčenie *race conditions* nepostačuje na zabezpečenie toho, aby súbežné procesy kooperovali správne a vhodne používali zdieľané dáta):

1. Žiadne dva procesy nemôžu byť súčasne vo svojich kritických úsekoch spojených s tým istým zdieľaným prostriedkom.
2. Pokiaľ proces do kritického úseku vstúpi, v konečnom čase z neho vystúpi.
3. Ak nie je proces v kritickom úseku, nebráni iným procesom do neho vstúpiť.
4. Každý z procesov žiadajúci vstup do kritického úseku bude uspokojený v konečnom čase.
5. Nie sú žiadne predpoklady o relatívnej rýchlosti procesov alebo počte procesorov.

5.2 Návrhy na dosiahnutie vzájomného vylúčenia

Hardwarové riešenia

Znemožnenie prerušenia

Ide o najjednoduchšie riešenie — po vstupe do kritického úseku znemožniť všetky prerušenia a umožniť ich až po odchode z kritického úseku, vrátane prerušení od hardwaru. Nie je však vhodné dať takúto možnosť užívateľským procesom. Naviac, ak má počítač 2 alebo viac CPU, tak toto znemožnenie prerušenia sa týka len jedného CPU, ostatné z nich budú pokračovať normálne a pristupovať do zdieľanej pamäte. Je to vhodné riešenie pre samotný kernel (jadro systému), kým updatuje premenné alebo zoznamy.

Špeciálna inštrukcia - TSL

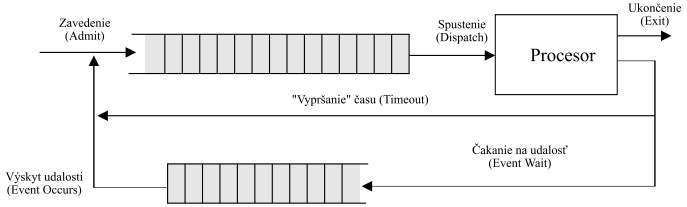
Mnohé počítače majú inštrukciu *Test and Set Lock* (TSL). Tá číta obsah daného pamäťového slova do registra a uloží na jeho adresu hodnotu rôznu od 0 (napr. 1). Operácie čítania slova a ukladania doň sú nedeliteľné (vykonané v jednom inštrukčnom cykle).

Na to, aby sme pomocou TSL inštrukcie koordinovali prístup do zdieľanej pamäte, použijeme zdieľanú premennú flag. Keď má flag nulovú hodnotu, ľubovoľný proces ju môže nastaviť na 1 použitím inštrukcie TSL a potom čítať alebo zapisovať do zdieľanej pamäte. Keď takúto činnosť ukončí, nastaví flag na 0 použitím inštrukcie MOVE.

možné, že hlavný proces vie, ktorý z potomkov je najdôležitejší a ako by mali byť potomkovia zaradení. Avšak žiadny zo spomenutých plánovačov neakceptuje vstup z užívateľských procesov, ktorý sa týka rozhodovania plánovania. Preto plánovač nemôže urobiť najlepší výber.

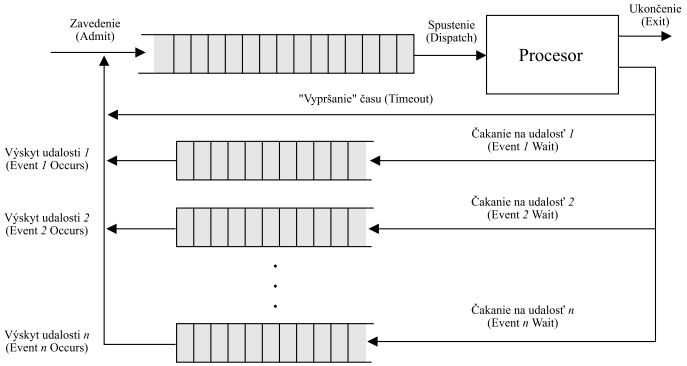
Riešením tohto problému je oddeliť plánovací mechanizmus od plánovacej „policy“ (pravidiel), t.j. plánovací mechanizmus je nejako parametrizovaný a parametre môžu byť nastavené užívateľskými procesmi (napr. systémové volanie, ktorým môže proces nastaviť a zmeniť priority svojich potomkov, t.j. rodič môže riadiť plánovanie potomkov, aj keď sám nerobí plánovanie). Teda mechanizmus je v kerneli, ale „policy“ je na základe nastavení z užívateľského procesu.

Nasledujúci obrázok ukazuje, ako môže byť realizované zaraďovanie procesov.



Keď je proces vpustený do systému, zaradí sa do *Zoznamu pripravených procesov (Ready queue)*. Proces na spracovanie sa vyberá z tohto zoznamu (môže to byť napr. FIFO zoznam). Proces opustí procesor buď keď je ukončený alebo sa zaradí do Zoznamu pripravených procesov (bol pozastavený napr. z dôvodu vyčerpania prideleného času) alebo do *Zoznamu blokovaných procesov (Blocked queue)* (čaká na nejakú udalosť). Zo Zoznamu blokovaných procesov sa proces presúva do Zoznamu pripravených procesov, keď nastala udalosť, na ktorú čakal.

Ak by bol len jeden Zoznam blokovaných procesov, tak keď nastane nejaká udalosť, operačný systém musí prehľadať celý zoznam, aby našiel proces čakajúci na túto udalosť. Vo veľkých operačných systémoch v tomto zozname môže byť stovky až tisíce procesov. Preto je efektívnejšie mať viacero takýchto zoznamov, jeden pre každú udalosť. Potom keď táto udalosť nastane, všetky procesy zaradené v príslušnom zozname môžu byť presunuté do zoznamu pripravených procesov.



Swapovanie procesov

Mnohé operačné systémy umožňujú presunutie procesov (alebo ich častí) z hlavnej pamäte na disk – swapovanie procesov – za účelom zlepšenia výkonnosti systému. Napríklad, môže nastať situácia, kedy všetky procesy, ktoré sa nachádzajú v pamäti, sú blokované (čakajú na V/V) a procesor „zaháľa“. Do pamäte však už nemožno zaviesť ďalšie procesy (Nový → Pripravený), lebo v nej nie je miesto. Riešením môže byť odsunutie nejakého blokovaného procesu na (swap) disk a tým sa uvoľní pamäť.

Avšak aj swapovanie je vstupno-výstupná operácia a preto je možné, že sa situácia ešte zhorší, a nie zlepší. Ale pretože diskové V/V operácie sú najrýchlejšie v systéme (v porovnaní s páskovými V/V či výstupmi na tlačiareň), swapovanie zvyčajne zvýši výkonnosť.

Do modelu stavov procesov musí pribudnúť nový stav – *Odswapovaný (Swapped, Suspended)*. Keď sú všetky procesy v hlavnej pamäti blokované, operačný systém môže niektorý proces previesť do stavu Odswapovaný a presunúť ho na disk.

Pri presúvaní procesov z disku späť do pamäte je nevýhodné presúvať blokované procesy, pretože tie stále nie sú pripravené na vykonávanie. Ak však nastala udalosť, na ktorú čakal niektorý z odsunutých procesov, proces prestáva byť blokovaný a je potenciálne pripravený na vykonávanie. Na presun späť do

Nevýhody: fragmentácia

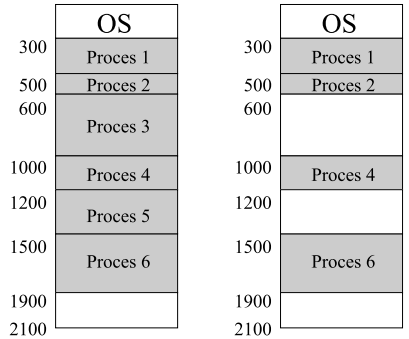
- *vnútorná* fragmentácia: ak proces potrebuje pre svoj beh pamäť s kapacitou K_1 a obdrží úsek s kapacitou K_2 ($K_2 > K_1$), tak $K_2 - K_1$ pamäťových miest je nevyužitých.
- *vonkajšia* fragmentácia: správa pamäte nemôže žiadnemu z pripravených procesov prideliť voľný úsek, lebo žiadny úsek nemá dostatočnú kapacitu (aj keď spojenie voľných úsekov by požadovanú kapacitu malo).

Vnútornú fragmentáciu je možné minimalizovať použitím stratégie best-fit, vonkajšiu fragmentáciu možno minimalizovať na úrovni plánovača úloh: ten vyberá zmes úloh tak, aby ich požiadavky najlepšie pokryli existujúce úseky. Touto metódou sa však nedajú dosiahnuť zaručené úspechy.

Systém s úsekmi pevnej dĺžky je postačujúci pre systémy s dávkovým spracovaním. Avšak pre systémy so zdieľaním času je typické, že v nich je zvyčajne viac používateľov než pamäte pre ich procesy. Procesy, ktoré sa nezmestia do pamäte, musia byť odložené na disk a odtiaľ opäť presunuté do pamäte (swapovanie). Pre systémy so swapovaním sa používajú úseky s premennou dĺžkou.

9.1.3 Dynamické súvislé úseky (Variable partitions)

Správa pamäte vytvára úseky operačnej pamäte podľa požiadaviek procesov podľa toho, ako prichádzajú.



Ak má 7. proces požiadavku na úsek s kapacitou 200K, tak správa pamäte používajúca stratégiu best-fit mu pridelí úsek s adresami 1900–2100, so stratégiou first-fit pridelí úsek od adresy 600 po adresu 800 (800–1000 bude voľné).

Algoritmus first-fit môže mať z hľadiska celkového využitia pamäte lepšie vlastnosti ako best-fit, ktorý zanecháva menšie voľné nepridelené úseky, a tým zvyšuje pravdepodobnosť vonkajšej fragmentácie. Pridelenie úsekov operačnej pamäte podľa požiadaviek procesov odstraňuje vnútornú fragmentáciu, ale vedie ku zvyšovaniu vonkajšej fragmentácie.

Správa pamäte často vytvára úseky o kapacite rovnej násobku základnej pamäťovej prideľovacej jednotky (IBM/360, ADT 4500 to sú 2K slabík, PDP11, SM-4: 32 slov po 16 bitoch). To síce zvyšuje vnútornú fragmentáciu, ponecháva však voľné časti pamäti zmysluplných dĺžok (lebo evidencia malých voľných úsekov je veľmi zložitá).

Ak sa ako 7. úloha objaví úloha s požiadavkou na pridelenie úseku so 450K pamäte, tak ju plánovač nezaháji, aj keď je v pamäti 900K voľných, lebo nie je voľný úsek dostatočnej kapacity (nastala vonkajšia fragmentácia). Je možné posunúť úseky v operačnej pamäti tak, aby vznikol súvislý voľný priestor. Tomu sa hovorí *kompaktovanie (defragmentácia)*. Je to časovo náročná operácia a vykonáva sa, až keď sa detekuje vznik vonkajšej fragmentácie. Ak očakávame, že väčšina procesov bude počas behu rásť, môžeme procesu pri načítaní do pamäti prideliť trochu viac pamäte ako momentálne potrebuje.

Správa pamäte si musí viesť prehľad o voľných úsekoch. Často sa používa forma viazaného zoznamu (na začiatku voľného úseku je informácia o jeho dĺžke a smerník na ďalší voľný úsek) alebo bitové mapy.

na poznatkoch, že timesharing systém poskytuje jednak multiprogramovanie, jednak rozšírený počítač s omnoho viac vyhovujúcim interfacom ako holý hardware. Základom VM/370 je úplne oddeliť tieto dve funkcie.

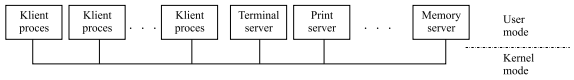
Jadro systému (*monitor viruálneho počítača*) beží na holom hardware a vykonáva multiprogramovanie, pričom poskytuje nie jeden, ale niekoľko virtuálnych počítačov na ďalšej úrovni. Avšak tieto virtuálne počítače nie sú rozšírené počítače (so súbormi a inými „peknými“ črtami), ale sú to presné kópie hardwaru, vrátane kernel/user módu,V/V, prerušení atď. Pretože každý virtuálny počítač je identický s hardwarom, na každom môže bežať ľubovoľný OS, ktorý bude bežať priamo na hardwari: napr. na jednom OS/360 pre batch procesy, na inom jendoužívateľský interaktívny systém CMS (Conversational Monitor System).

Keď CMS program vykoná systémové volanie, to je odovzdané operačnému systému v jeho vlastnom virtuálnom počítači, nie VM/370. CMS potom vykoná normálne hardwarové V/V operácie na čítanie svojho virtuálneho disku alebo čo už vyžadovalo volanie. Tieto V/V inštrukcie sú vykonané systémom VM/370, ktorý ich vykoná ako časť svojej simulácie reálneho hardwaru.

Vykonaním kompletnej separácie funkcie multiprogramovania a poskytovania rozšíreného počítača môže každá časť byť jednoduchšia, flexibilnejšia a ľahšie spravovateľná a udržiavateľná.

Klient-server model

VM/370 posunul veľkú časť kódu tradičného operačného systému do vyššej úrovne, CMS. Avšak je to stále rozsiahly program, lebo simulovanie množstva virtuálnych 370-ok nie je tak jednoduché. Trendom moderných OS je vziať ideu presúvania kódu do vyšších úrovní ešte viac a „zmazať“ (presunúť) čo najviac z operačného systému, a teda ponechať len minimálny *kernel*. Zvyčajný prístup je implementovať väčšinu funkcií OS v užívateľských procesoch. Na požiadanie o službu, napr. čítanie bloku súboru, užívateľský proces (*klient-proces*) posiela požiadavku *server-procesu*, ktorý vykoná úlohu a pošle späť odpoveď. V tomto modeli všetko, čo robí kernel, je udržiavanie komunikácie medzi klientami a serverom.

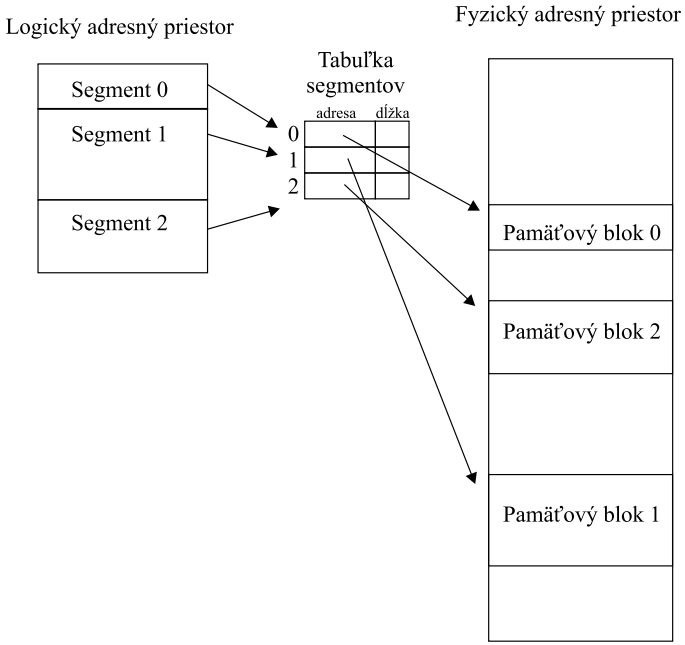


Rozdelením operačného systému na časti, z ktorých každá má na starosti len nejakú časť systému — správa súborov, procesov, terminálu, pamäte — sa každá časť stáva menšou a ľahšie spravovateľnou. Naviac, keďže všetky servre bežia ako user-mode procesy (nie v kernel-móde), nemajú priamy prístup k hardwaru. Teda, ak sa napr. vyskytne chyba vo file-serveri, môže spadnúť služba, ale zvyčajne to nespôsobí „spadnutie“ celého počítača.

Ďalšou výhodou tohto modelu je jeho prispôsobiteľnosť pre distribuované systémy. Ak klient komunikuje so serverom vysielaním správ, nepotrebuje vedieť, či správa je spracovaná lokálne, v jeho vlastnom počítači alebo je posielaná cez sieť serveru na vzdialenom počítači.

Predošlý obrázok, ktorý ukazoval, že kernel má na starosti len posun správ z klientov do serverov a späť nie je úplne realistický. Niektoré funkcie OS (napr. nahratie inštrukcie do registrov fyzických V/V-zariadení) je ťažké (príp. nemožné) robiť z užívateľských programov. Sú dva možné spôsoby ako riešiť tento problém:

- mať nejaké rozhodujúce server-procesy (napr. I/O device drivery) bežiace v kernel móde s kompletným prístupom k hardwaru, ale ktoré komunikujú s ostatnými procesmi prostredníctvom normálnych mechanizmov správ.
- zabudovať minimálne množstvo mechanizmu do kernelu, ale ponechať princípy a pravidlá rozhodnutia na serveri v používateľskom priestore. Napríklad kernel môže rozpoznať, že správa poslaná na nejakú špeciálnu adresu znamená vziať obsah správy a uložiť ho do registrov V/V-zariadení pre nejaký disk a začať diskové čítanie. Kernel nepreveruje byty správy, či sú platné a či majú zmysel, len ich kopíruje do registrov zariadenia (zvyčajne sa ale preveruje, či je proces „autorizovaný“ na vyslanie takej správy.



Pri transformácii logickej adresy na fyzickú sa číslo segmentu použije ako index do tabulky adries segmentov (Segment Map Table) - obsahuje začiatočné adresy všetkých segmentov v pamäti a ich veľkosti. Potom sa porovná ofset s veľkosťou segmentu - ak je väčší, tak je adresa neplatná. Fyzická adresa sa získa ako súčet začiatočnej adresy segmentu v pamäti (adresa prideleného pamäťového bloku) a ofsetu.

Transformácia adresy sa robí automaticky (procesorom) počas behu programu. Tabulka segmentov je trvale súčasťou záznamu o procese. Rovnako ako stránky, aj segmenty je možné zdieľať viacerými procesmi, čo však môže prinášať problémy pri adresovaní.

Nevýhody: Súvislé ukladanie segmentov do FAP a premenná dĺžka segmentov vedie k rovnakým problémom s transformáciou pamäte do dynamicky tvorených súvislých úsekov. Správa pamäte musí segmentom prideľovať bloky premennej dĺžky, čím vzniká nebezpečenstvo vonkajšej fragmentácie. Istou prednosťou segmentácie je, že segmenty obvykle požadujú kratšie bloky pamäte, než by požadoval nesegmentovaný program.

Hlavný rozdiel medzi stránkovaním a segmentáciou je v tom, že segment je „logická“ jednotka, má ľubovoľný rozsah a je „viditeľný“ v používateľskom programe, zatiaľčo stránka je „fyzická”“jednotka informácie pevného daného rozsahu, používa sa iba v module prideľovania pamäte a v používateľskom programe ju nie je „vidieť“.

9.1.7 Kombinované systémy

Aj stránkovanie aj segmentovanie majú svoje výhody aj nevýhody. Je ich možné kombinovať na vylepšenie:

- segmented paging (PT je segmentovaná)
- paged segmentation (segmenty sú stránkované)

Kapitola 3

Členenie OS, služby OS

3.1 Čo je operačný systém?

OS plní dve v základe „nesúvisiace“ funkcie:

OS ako rozšírený počítač

Architektúra (množina inštrukcií, organizácia pamäte, V/V, štruktúra zbernice) väčšiny počítačov na úrovni strojového jazyka je primitívna a „nepohodlná“ pre program, najmä pre V/V. Na upresnenie sa pozrime ako je relizovaný V/V z floppy disku použitím NEC PD765 controller čipu, ktorý sa používa pre IBM PC a mnohé ďalšie osobné počítače.

PD765 má 16 príkazov, každý je špecifikovaný nahratím 1–9 bytov do registrov zariadenia: pre čítanie, zápis, pohyb hlavy, ... Najpoužívanejšie príkazy READ a WRITE vyžadujú po 13 parametrov spakovaných do 9 bytov (určujú adresu diskového bloku, počet sektorov na stope, nahrávací mód,...) Keď je operácia ukončená, čip vráti 23 stavov a chybové polia spakované do 7 bytov. Programátor floppy disku musí byť oboznámený, či motor je zapnutý alebo vypnutý. Ak je vypnutý, musí byť zapnutý (s dlhým časovým oneskorením) predtým, než je možné presúvať dáta. Aj bez toho, aby sme skutočne šli do detailov, vidíme, že bežný programátor nebude chcieť presne ovládať programovanie floppy disku (alebo pevného disku, čo je úplne odlišná, rovnako zložitá úloha), ale bude chcieť *jednoduchú abstrakciu vyššej úrovne*, ktorou sa bude zaoberať. V prípade disku touto abstrakciou je, že disk obsahuje množinu pomenovaných súborov. Každý súbor môže byť otvorený, číta sa, zapisuje, zatvorí sa. Detaily sa v abstrakcii prezentovanej používateľovi neobjavia.

Program, ktorý skrýva detaily pred používateľom, je operačný systém. Z tohto pohľadu je funkciou OS predkladať používateľovi ekvivalent *rozšíreného* alebo *virtuálneho počítača*, ktorý je možné ľahšie programovať ako hardware.

OS ako správca prostriedkov

Použitie OS ako programu, ktorý poskytuje používateľom vhodný interface je pohľad zhora-dole. Operačný pohľad (zdola-hore) je, že OS riadi všetky časti komplexného systému, t.j. má na starosti riadenie prideľovania procesov, pamäte, V/V-zariadení rôznym programom, ktoré o ne žiadajú. Keď má systém viacero používateľov, je treba zabezpečiť správu a ochranu pamäte, V/V-zariadení. Tiež sa zabezpečuje evidencia používania prostriedkov.

3.2 Koncepcia OS

Interface medzi OS a užívateľskými programami je definovaný množinou „rozšírených inštrukcií“, ktoré OS vykonáva — sú známe ako „systémové volania“. Systémové volania vytvárajú, rušia a používajú

Vylepšenia FIFO

Aby sme sa vyhli obetovaniu síce starej, ale intenzívne používanej stránky, možno tiež použiť R a M bity. Postupujeme tak, že najprv obetujeme najstaršiu stránku z triedy 0. Ak taká nie je, hľadáme stránky zo triedy 1, 2, 3.

Z algoritmu FIFO je odvodený aj *algoritmus druhej nádeje* — opäť preveríme najstaršiu stránku ako potenciálnu obeť: ak má bit $R = 0$, odstránime ju hneď. Ak má $R = 1$, t.j. bola nedávno použitá, tak bit R vynulujeme a stránku zaradíme na koniec zoznamu, ako keby práve prišla do pamäte. Ak udržujeme zoznam kruhový, tak namiesto zaraďovania na koniec zoznamu, sa len o jednu stránku posunie pointer v zozname. Toto sa často nazýva *hodiny*. Ak sa intenzívne pracuje so stránkami, degraduje sa tento algoritmus na FIFO.

LRU (Least-Recently-Used Page Replacement)

Je založený na predpoklade, že stránky, ktoré sa počas niekoľko posledných inštrukcií intenzívne používali, sa pravdepodobne budú intenzívne používať aj naďalej. A naopak, stránky, ktoré sa už dlho nepoužívajú, sa ešte dlho nebudú používať. Teda keď vznikne výpadok stránky, obetujeme stránku, ktorá sa najdlhšie nepoužívala. To je však veľmi „drahé“. Ak by sme to chceli plne implementovať, potrebovali by sme zoznam stránok v pamäti, zoradený podľa toho, ako dávno boli stránky použité a tento zoznam by sme museli upravovať pri každom odkaze do pamäti. Presúvanie prvkov v zozname je časovo náročná operácia a buď by sme museli použiť špeciálny hardware alebo nájsť nejakú lacnejšiu softwarovú aproximáciu. Budeme sa zaoberať 2. možnosťou, konkrétne algoritmom nazývaným

NFU (Not Frequently Used Page Replacement)

Ku každej stránke máme priradené softwarové počítadlo, ktoré je na začiatku vynulované. Pri každom prerušení od hodín OS prechádza všetky stránky v pamäti a k počítadlu pripočíta obsah R bitu (až potom ho vynuluje). Teda počítadlo udržiava informáciu o tom, ako často sa stránka používa. Keď nastane výpadok stránky, tak obetujeme stránku s najmenším počítadlom. Pri tejto realizácii vzniká problém, že sa „nikdy na nič nezabúda“. Môže sa napr. stať, že na začiatku intenzívne používame nejaké stránky, a teda majú vysoké počítadlo. Keď sa potom začnú používať iné stránky (časti) programu, budú mať nízke počítadlo, takže padnú za obeť aj napriek tomu, že sa momentálne intenzívne používajú. Tento nedostatok možno odstrániť malou úpravou a dostaneme algoritmus nazývaný *starnutie* (*Aging*). Nastanú tieto zmeny:

- Pred pripočítaním bitu R sa počítadlo posunie o 1 bit doprava.
- Bit R sa pripočíta k najľavejšiemu, nie k najpravejšiemu bitu. Keď sa potom vyskytne výpadok stránky, obetujeme stránku s najmenším počítadlom (ak nejaká stránka nebola odkazovaná, napr. počas posledných 4 tikov, bude mať zľava 4 vedúce 0, teda nižšiu hodnotu ako počítadlo stránky, na ktorú sa neodkazovalo posledné 3 tiky).

10.2 Stránkovanie na žiadosť (demand paging) versus model s pracovnou množinou (working set model)

Pri stránkovaní na žiadosť nemá proces pri spustení žiadnu stránku v pamäti. Hneď, ako sa CPU pokúsi vykonať (načítať) prvú inštrukciu, vznikne výpadok stránky a OS načíta stránku s prvou inštrukciou. Zvyčajne hneď nasledujú ďalšie výpadky stránok kvôli zásobníku, globálnym údajom a po chvíli má proces načítané všetky stránky, ktoré práve potrebuje a beží s relatívne malým počtom výpadkov stránok. Samozrejme, je možné napísať testovací (trashing) program, ktorý by systematicky načítaval všetky stránky vo veľmi veľkom adresnom priestore, čím by používal také množstvo stránok, že by pre ne nestačila pamäť a dochádzalo by k častému vymieňaniu stránok. V praxi však väčšina procesov používa relatívne malú časť svojich stránok. Tejto množine stránok, ktorú proces momentálne používa hovoríme

Kapitola 2

Úvod do operačných systémov, história operačných systémov, história Unixu

Software počítača môžeme rozdeliť na dva druhy programov: *systémové programy*, ktoré riadia operácie samotného počítača a *aplikačné programy*, ktoré riešia užívateľské úlohy.

Najzákladnejším zo všetkých systémových programov je *operačný systém*, ktorý riadi všetky prostriedky počítača a poskytuje bázu, na ktorej môžu byť napísané aplikačné programy. Slúži ako interface medzi užívateľom a hardwarom. Moderný počítačový systém pozostáva z 1 alebo viac procesorov, hlavnej pamäte, hodín, terminálov, diskov, V/V-zariadení,... — je to komplexný systém. Každý programátor nemôže tvoriť programy so znalosťou všetkých spomenutých komponentov a ich použitia. Bolo preto treba nájsť spôsob, ako ochrániť programátorov od spletitosti hardwaru, a to vytvorením vrstvy softwaru na vrchu „holého" hardwaru, ktorá bude riadiť všetky časti systému a poskytuje používateľovi interface alebo viruálny počítač, ktorý je ľahké programovať — *operačný systém*.

Členenie počítačového systému na vrstvy (zdola nahor):

- hardware
  - *fyzické zariadenia* (integrované obvody, káble,...)
  - *mikroprogram* — primitívny software, ktorý priamo riadi fyzické zariadenie, zvyčajne je umiestnený v read-only pamäti. Je to vlastne interpreter interpretujúci inštrukcie strojového jazyka (ako MOVE, ADD, JUMP) ako sériu malých krokov.
  - *strojový jazyk* — množina inštrukcií, ktoré interpretuje mikroprogram. Na niektorých počítačoch je implementovaný v hardware. Má okolo 50–300 inštrukcií (presun dát, aritmetika, porovnávanie). Na tejto úrovni sú V/V-zariadenia riadené ukladaním hodnôt do špeciálnych registrov zariadení. Strojový jazyk nie je priamo časťou holého počítača, ale výrobcovia ho vždy popisujú vo svojich manuáloch.
- software
  - *operačný systém*, ktorého hlavnou funkciou je skryť túto spletitosť a dať programátorovi vhodnejšiu množinu inštrukcií na prácu.
  - *systémové programy* — dôležité je, aby tieto programy neboli časťou OS, hoci zvyčajne sú dodávané výrobcom počítača. OS je časť softwaru, ktorá beží v kernel-móde alebo v supervisor-móde. Je chránený hardwarom pred zásahom používateľa. Kompilátory a editory bežia v užívateľskom móde.
  - *aplikačné programy* — napísané používateľom na riešenie konkrétnych problémov

Keď page daemon beží, ručičky rotujú, kým nevznikne aspoň lotsfree voľných položiek.

Ak sa často stránkuje a počet voľných rámcov je stále nižší ako lotsfree, swapper odsunie nejaké procesy na swap-disk.

Swapovací algorimus pre 4BSD

Swapper zistí, či existuje proces, ktorý je „idle" viac než 20 sekúnd. Ak áno, tak ten, čo je idle najdlhšie, je odswapovaný. Ak nie, preveria sa 4 najväčšie procesy a odswapovaný je ten, ktorý je v pamäti najdlhšie. Toto sa prípadne opakuje, až kým nie je dosť miesta.

Každých pár sekúnd swapper preveruje, či existuje nejaký pripravený proces na disku. Každý proces na disku má priradenú hodnotu, ktorá je funkciou toho, ako dlho je odswapovaný, jeho veľkosti, nice a toho, ako dlho spal pred odswapovaním. Táto funkcia je váhovaná, aby sa zvyčajne nahral do pamäte proces, ktorý je najdlhšie odswapovaný, avšak iba ak nie je priveľký (presun veľkého procesu je drahý, a teda sa nesmie robiť často). Swapper nahrá do pamäte len „user structure" a tabuľku stránok. Ostatné časti sú stránkované podľa potreby.

Stránkovanie pre System V

je veľmi podobné 4BSD. Sú tu však dva zaujímavé rozdiely:

1. Používa originálny „one-handed clock algorithm". Stránka sa zaraďuje do zoznamu voľných rámcov, ak sa nepoužíva v n nasledujúcich prechodoch.
2. Namiesto jednoduchej premennej lotsfree System V používa dve premenné min a max. Ak počet voľných rámcov klesne pod min, uvoľňuje sa pamäť dovtedy, kým nie je voľných aspoň max rámcov.

.ENDM DEF

Rozvoj volania DEF ZMAZ je:

...

.MACRO ZMAZ A

CLRL A

.ENDM ZMAZ

...

takže po tomto už môžeme použiť ZMAZ R5 a rozvoj bude CLRL R5.

Ak voláme DEF CISTI, zadefinuje sa makro CISTI a môžeme použiť volanie CISTI R5, ktoré má takisto rozvoj CLRL R5.

Makroprocesor:

Makroprocesor je program, ktorý má tieto funkcie:

1. nájsť a uložiť definície makier
2. nájsť volania makier a rozvinúť ich s dosadením parametrov

Makroprocesor môže byť program funkčne nezávislý od asemblera, výstup z makroprocesora (program v jazyku asemblera, v ktorom sa nevyskytujú makrá) je potom vstupom do asemblera.

Podľa počtu prechodov zdrojovým textom rozlišujeme dva typy makroprocesorov:

- dvojprechodové
- jednoprechodové

Dvojprechodový makroprocesor

1. prechod: jeho úlohou je prejsť vstupný text a uložiť nájdené definície makier. Názvy makier ukladá do *tabuľky mien makier* spolu so smerníkom na telo makra, uložené v *tabuľke definící makier*. V tabuľke definícií makier je uložený najprv tzv. prototyp makra, čiže zoznam parametrov aj s implicitnými hodnotami, aby bolo možné použiť aj nepozičné volanie makra. V tomto prechode sa tiež robia rozvoje systémových makier.

2. prechod: číta zdrojový text a vytvára výstupný text nasledovne: ak ide o inštrukciu alebo direktívu, riadok zdrojového textu sa skopíruje do výsledného textu. Ak sa nájde volanie makra, do výsledného textu sa budú kopírovať riadky z tabuľky definícií makier (čiže telo makra). Podľa smerníka v tabuľke mien makier sa nájde definícia makra v tabuľke definícií, pripraví sa *pole zoznamu parametrov makra*, ktoré sa naplní hodnotami parametrov z volania makra a môžu sa do výsledného textu kopírovať riadky z tela makra, do ktorých sa dosádzajú parametre z uvedeného poľa.

Ak je v tele makra volanie ďalšieho makra, pole zoznamu parametrov a aktuálna pozícia v tabuľke definící makier sa uložia do zásobníka, pripraví sa pole zoznamu parametrov pre vnorené makro, nájde sa jeho definícia a vkladá sa telo tohto makra. Keď je rozvoj vnoreného makra dokončený, zo zásobníka sa obnoví stav pred vnoreným rozvojom a pokračuje sa v rozvoji vonkajšieho makra.

Dvojprechodový makroprocesor nevie spracovať vnorené definície. Problém je v tom, že definícia vnútorného makra sa objaví až v druhom prechode makroprocesora – pri rozvoji definujúceho makra. Teda táto nová definícia nie je zapísaná v tabuľke mien a definící makier a preto keď sa vyskytne volanie nového makra, nebude možné urobiť jeho rozvoj. Bolo by v takomto prípade nutné zopakovať oba prechody makroprocesora.

Jednoprechodový makroprocesor

Jednoprechodový makroprocesor v rámci jedného prechodu zdrojovým textom ukladá definície makier a robí aj rozvoje makier. Jedinou požiadavkou je, aby vždy definícia makra predchádzala jeho volaniu. Dokáže (podobne ako dvojprechodový makroprocesor) spracovať vnorené volania makier a tiež makrá definujúce iné makrá.

Makroasembler

- *Index blocks*, t.j. spájaný zoznam diskových blokov: Každý blok obsahuje toľko adries (t.j. smerníkov) k voľným blokom, koľko sa doň zmestí a smerník na ďalší takýto blok. Ak máme bloky veľkosti 1K a 16-bitové adresy blokov, tak v každom bloku môže byť 511 adries voľných blokov $((1024 : 2) - 1 = 511)$. Disk veľkosti 20M (t.j. 20K blokov veľkosti 1K) potom bude potrebovať cca 40 blokov na uchovanie všetkých 20K diskových adries blokov $((20 \cdot 1024) : 511 \doteq 20 \cdot 2 = 40)$.

  Nevýhody:

  – zlý prehľad o súvislých voľných oblastiach
  – problémom je, ako značiť, v ktorých položkách sú smerníky na voľné bloky a ktoré položky v poslednom indexovom bloku sú prázdne

- *Bitová mapa*: Disk s N blokmi potrebuje mapu s N bitmi, kde 1 = obsadený a 0 = voľný (alebo naopak). Potom 20M disk (s blokmi veľkosti 1K) potrebuje 20K bitov na mapovanie adries blokov, t.j. 3 bloky $((20 \cdot 2^{10}) : (8 \cdot 2^{10}) \doteq 3)$. Pokles oproti metóde „index blocks“ nastáva preto, lebo metóda bitovej mapy používa 1 bit na 1 blok, kým metóda „index blocks“ na to potrebuje 16 bitov. Jedine ak je disk takmer plný, tak schéma spájaného zoznamu bude požadovať menej blokov ako bitová mapa.

  Ak máme v operačnej pamäti dosť miesta na udržanie celej bitovej mapy naraz, je metóda bitovej mapy výhodnejšia. Ak však len jeden blok pamäti môže byť rezervovaný na uchovávanie informácie o voľných blokoch na disku a disk je takmer plný, tak spájaný zoznam bude lepší. Keď je v operačnej pamäti len jeden blok bitovej mapy, môže sa stať, že v ňom nenájdeme žiadne voľné bloky, takže treba pristupovať na disk a čítať zvyšok bitovej mapy, kým pri spájanom zozname pri načítaní jedného bloku do pamäte je možné alokovať 511 diskových blokov (získame 511 voľných blokov) pred ďalším nutným prístupom na disk na čítanie ďalšieho bloku zo zoznamu.

Diskové kvóty (quotas)

V multiužívateľskom operačnom systéme je často mechanizmus na zavedenie diskových kvót, t.j. stanovenie maximálneho množstva priestoru na disku pre používateľa a maximálneho počtu súborov.

11.3 Implementácia systému súborov

Implementácia súborov rieši problém, ktoré bloky disku sú pridelené súboru.

11.3.1 Súvislá alokácia

Najjednoduchším spôsobom je prideliť súboru súvislý blok dát na disku (postupnosť za sebou idúcich blokov).

Výhody:

- ľahká implementácia (v adresári je uložená začiatočná adresa a veľkosť súvislého bloku prislúchajúceho súboru)
- celý súbor môže byť z disku čítaný naraz v 1 operácii

Nevýhody:

- treba vopred poznať maximálnu veľkosť súboru
- fragmentácia disku (kompaktácia je zvyčajne veľmi „drahá“)

11.3.2 Spájaný zoznam blokov na disku

Prvé slovo v každom bloku je smerník na ďalší blok (v adresári je uložené číslo prvého bloku). Veľmi pomalý je náhodný prístup, napr. pri posune na bajt $32768 = 32K$ treba prejsť cez $32768 : 1022 \doteq 33$ blokov (1 blok má $1K = 1024B$, pričom 2B zaberá smerník). Tiež môže byť problémom, že počet dát v bloku nie je mocnina 2 (mnohé programy čítajú a zapisujú v blokoch veľkosti mocniny 2).

miesta pre lokálne premenné).

Vrátenie hodnôt a príznakov:

Na VAXe je konvencia, že ak ide o funkciu, hodnota funkcie sa vráti v registri R0 (v prípade dát vyššej presnosti v R0 a R1).

Na uloženie príznakov (napr. či sa úloha úspešne vykonala, či nastali nejaké špeciálne situácie) sú dohodnuté dve miesta: register R0 alebo podmienkové bity – tie boli pred uložením do zásobníka, do bloku volania, vynulované. Procedúra ich môže nastaviť a po návrate do hlavného programu (po naplnení PSW) je možné ich otestovať.

Rekurzia:

Rekurzívne procedúry nemôžu mať dáta uložené staticky (.LONG, .BLKx, ...), ale všetky lokálne premenné musia byť uložené v zásobníku tak, že premenné z jedného volania nie sú modifikované ďalším rekurzívnym volaním.

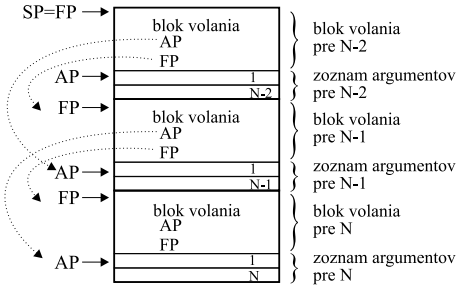
Ako príklad uvedieme výpočet faktoriálu: $N! = N.(N-1)!$, ak $N > 0$, $N! = 1$, ak $N = 0$.

```
        .ENTRY FAKT, ^M<R2>
        MOVL #1, R0             ;výsledok bude v R0 - je to funkcia
        MOVL 4(AP), R2          ;N daj do R2
        BEQL VON                ;končíme, keď N = 0
        SUBL3 #1, R2, -(SP)     ;do zásobníka daj N-1
        CALLS #1, FAKT          ;rekurzívne volanie procedúry
        MULL2 R2, R0            ;N.(N-1)!
VON:    RET
```

Hlavný program:

```
.BEGIN FAKTORIAL
   ⋮
PUSHL N
CALLS #1, FAKT
   ⋮
RET
.END FAKTORIAL
```

Po niekoľkonásobnom volaní rekurzívnej funkcie bude zásobník vyzerať takto:



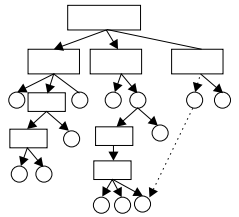
1.4 Asembler - prekladač

Asembler je program, ktorý prekladá zdrojový program v jazyku asemblera do strojového kódu. Okrem strojového kódu vytvára ďalšie informácie, ktoré potom využije linker a loader (viď. kap. Linker a loader). Výsledkom prekladu je *objektový modul*.

Počas prekladania asembler priraďuje symbolickým výrazom ich numerické hodnoty a adresy. Na

11.5 Zdieľané súbory

Často je potrebné, aby viacero používateľov zdieľalo ten istý súbor. Preto je vhodné, aby sa zdieľaný súbor akoby vyskytoval súčasne v rozličných adresároch (resp. aby jeden súbor mohol mať viacero mien). Strom súborov potom vyzerá nasledovne:



Spojenie medzi adresárom B a zdieľaným súborom nazývame *link*. Zdieľanie súborov je užitočné, aj keď s implementáciou sú problémy.

Bližšie si vysvetlíme implementáciu linkov v OS UNIX. Link možno implementovať dvoma spôsobmi:

- *priamy link (hard link)*: v adresári sa vytvorí položka pre link obsahujúca meno (linku) a číslo i-node zdieľaného súboru (čiže "nové" meno súboru sa odkazuje na ten istý i-node, ktorý má "pôvodný" súbor).
- Link v adresári B bude realizovaný ako špeciálny súbor (typu link), ktorý obsahuje názov zdieľaného súboru. To je *symbolický link (symbolic link)*.

Obe tieto metódy majú svoje „vedľajšie účinky". V prvom prípade, keď sa B pripojí k zdieľanému súboru, v i-node ostáva ako vlastník uvedený C. Vytvorenie linku nemení vlastníka, iba sa v i-node zvýši počítadlo linkov, takže systém vie, koľko položiek v adresároch na súbor ukazuje. Ak C vymaže súbor (len on ako vlastník to môže urobiť), tak stojíme pred problémom: Ak pri vymazaní súboru zároveň uvoľníme i-node, tak B bude ukazovať na nedefinovaný i-node. Keď sa neskôr tento i-node pridelí nejakému súboru, bude B ukazovať na zlý súbor. Systém totiž vie z počítadla linkov len to, že i-node (a teda súbor) sa ešte používa. Ale nemá možnosť nájsť všetky súbory, ktoré sa na tento i-node odkazujú, aby ich mohol tiež vymazať. Smerníky späť z i-node do adresára sa nemôžu uchovávať v i-node, lebo týchto smerníkov môže byť ľubovoľne veľa. Jediné, čo môže systém urobiť je, že pri vymazaní súboru v C nechá i-node nedotknutý s počítadlom 1 (B ho používa).

Teda sme v situácii, že B je jediný používateľ, ktorý má položku adresára pre súbor vlastnený C-čkom. Ak systém robí účtovanie diskového priestoru, tak súbor sa naďalej účtuje používateľovi C, a to až dovtedy, kým aj B nevymaže súbor. Tým sa zníži počítadlo na 0 a súbor aj i-node uvoľníme.

U symbolických linkov tento problém nie je, pretože iba skutočný vlastník súboru má aj smerník na i-node. Ostatní majú iba názov súboru. Keď vlastník vymaže súbor, tento sa skutočne zruší. Ak v zápätí použijeme symbolický link, tak dôjde k chybe, lebo súbor už neexistuje. Vymazanie symbolického linku pritom nijako nevplýva na súbor.

Problém, ktorý máme pri symbolickom linku, je réžia navyše. Najprv musíme nájsť a načítať súbor obsahujúci meno súboru, z neho musíme načítať názov zdieľaného súboru a znova analyzovať a prechádzať po jednotlivých zložkách, až kým nenájdeme i-node. To všetko vyžaduje nové a nové prístupy na disk. Navyše, na symbolický link potrebujeme i-node a ďalší diskový blok na uloženie názvu súboru.

Ďalší problém s linkami je, že súbor má dva alebo viac názvov. Programy, ktoré štartujú v danom adresári a hľadajú všetky súbory v tomto adresári a všetkých jeho podadresároch, nájdu zdieľané súbory viackrát. To môže byť problém napr. pri archivovaní súborov, lebo dostaneme viacnásobné kópie.

11.6 Výkonnosť file systému

Prístup na disk je omnoho pomalší ako do pamäte. Väčšina systémov sa snaží redukovať počet potrebných prístupov na disk. Najčastejšie na to používaná technika je *block cache* alebo *buffer cache*. Je to súhrn

Presuny a konverzie

MOVx	čo,kam	presun: kam:=čo
CVTxy	čo,kam	rozšírenie/skrátenie reprezentácie dát s doplnením znamienkového bitu
MOVZxy	čo,kam	rozšírenie/skrátenie reprezentácie dát s doplnením 0
MOVAx	náv,kam	presun adresy dát rozmeru x (kam:=adresa náv)

Skoky

Príkaz skoku môže spôsobiť, že do PC registra sa načíta nová adresa, teda sa nebude vykonávať nasledujúca inštrukcia. Väčšina príkazov skoku sú podmienené skoky, ktoré menia PC v závislosti od podmienky na dátach. VAX (a mnohé iné počítače) používa jednobitové príznaky nazývané *podmienkové bity (condition codes)* na zaznamenanie vlastností operandov inštrukcií – tieto príznaky sú súčasťou stavového slova procesora (PSW). Podmienené skoky testujú tieto príznaky, aby zistili, či treba meniť PC.

Podmienkové bity:

- N – Negative: N=1, ak výsledok operácie bol záporný
- Z – Zero: Z=1, ak výsledok operácie bol nula
- V – Overflow: V=1, ak nastalo pretečenie (výsledok presiahol vyhradený priestor)
- C – Carry: ak operácia mala prenos alebo záporný prenos v najľavejšom bite

Podmienkové bity sú automaticky nastavované vzhľadom na výsledok väčšiny operácií (napr. pri operácii sčítania sa nastavia podľa výsledku operácie, pri operácii prenosu sa nastavia podľa prenášaného čísla, pri operácii nulovania sa vždy nastaví N na 0, Z na 1, V na 0).

Niekedy je treba urobiť takéto nastavenie pre nejakú premennú alebo register v inom čase ako po vykonaní operácie alebo treba vyjadriť vzťah medzi dvoma porovnávanými hodnotami.

Na to slúžia dva príkazy:

TSTx	čo	test na nulu
CMPx	čo1, čo2	porovnanie operandov

Operácia TSTx nastaví Z a N bity podľa obsahu operandu (bity V a C vynuluje).

Operácia CMPx porovná operandy ako celé čísla v doplnku do 2 aj ako bezznamienkové čísla a podľa výsledku porovnania nastaví Z, N a C bity (vlastne robí porovnanie rozdielu čo1-čo2 s nulou – obsah operandov čo1 a čo2 sa pritom nezmení!):

Z=1, ak $čo1 = čo2$

N=1, ak $čo1 < čo2$ v doplnku do 2

C=1, ak $čo1 < čo2$ ako bezznamienkové čísla

Napr. ak $A = 6A_{16}$, $B = 94_{16}$, tak operácia CMPB A,B nastaví Z na 0 ($A \neq B$), N na 0 ($A - B \not< 0$, a teda $A \not< B$, lebo A je kladné a B je záporné - ako znamienkové čísla v doplnku do 2), C na 1 ($A < B$ bezznamienkovo) a V na 0.

Podmienené skoky

Na základe nastavenia podmienkových bitov podmienené skoky buď naplnia PC novou adresou (operand náv) alebo bude program pokračovať nasledujúcou inštrukciou.

BEQL	náv	ak rovné	– ak Z=1
BNEQ	náv	ak nerovné	– ak Z=0
BGTR	náv	ak väčšie	– ak N=0 a zároveň Z=0
BGEQ	náv	ak väčšie alebo rovné	– ak N=0
BLSS	náv	ak menšie	– ak N=1
BLEQ	náv	ak menšie alebo rovné	– ak N=1 alebo Z=1

Pri preklade do strojového kódu sa ukladá (podobne ako u relatívneho adresného módu) rozdiel medzi návestím náv a PC registrom – tu sa však tento rozdiel vždy ukladá do 1 bajtu (preklad celej inštrukcie podmieneného skoku tak zaberá 2 bajty) – takže je možné skákať len na návestia vzdialené 128 bajtov pred alebo 127 bajtov za aktuálnou pozíciou.

chýb, ak treba. Blok bajtov sa zvyčajne ukladá do buffera v radiči a až po preverení checksum je blok kopírovaný do pamäte.

Každý radič má niekoľko *registrov*, ktoré sa používajú na komunikáciu s CPU. U niektorých počítačov tieto registre sú časťou normálneho adresového priestoru pamäte (*memory-mapped I/O*), napr. PDP-11 má rezervované adresy od 0160000 po 0177777. Iné počítače (vrátane IBM PC) používajú špeciálny adresný priestor pre V/V, pričom každý radič má určenú nejakú jeho časť.

Operačný systém vykonáva V/V pomocou zapísania príkazov do registrov radičov, napr. radič floppy diskov IBM PC akceptuje 15 príkazov (ako read, write, seek, format, ...). Parametre príkazov sa tiež zapisujú do registrov radičov. Keď bol príkaz prijatý, CPU opustí radič a robí svoju prácu. Keď je príkaz vykonaný, radič spôsobí prerušenie, aby CPU mohol prijať výsledok operácie a stav zariadenia čítaním informácií z registrov radičov.

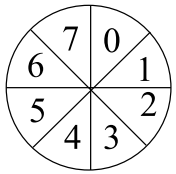
Mnohé radiče, najmä pre blokové zariadenia, umožňujú *priamy prístup do pamäte* (*Direct Memory Access*, DMA). Najprv si vysvetlíme, ako prebieha čítanie bez použitia DMA: Najprv radič číta blok zo zariadenia sériovo, bit po bite, až kým nie je celý blok vo vnútornom bufferi radiča. Ďalej vykoná výpočet checksumu, aby zistil, či sa pri čítaní nevyskytli nejaké chyby. Potom spôsobí prerušenie. Keď operačný systém začne bežať, môže čítať blok z buffera radiča po bajtoch alebo slovách v cykle.

Cyklus CPU na čítanie bajtov z radiča míňa veľa času CPU. DMA bol zavedený na to, aby oslobodil CPU od tejto práce nízkej úrovne. V tomto prípade CPU dáva radiču okrem diskovej adresy bloku aj pamäťovú adresu, kam má byť blok uložený a počet bajtov, ktorý má byť prenesený. Po tom, ako radič prečíta blok do svojho buffera a preverí checksum, kopíruje prvý bajt do hlavnej pamäte na určenú adresu, inkremetuje DMA adresu a dekrementuje DMA počítadlo bajtov. Tento proces sa opakuje, pokiaľ DMA počítadlo nebude 0. Vtedy radič spôsobí prerušenie. Operačný systém už nemusí kopírovať blok do pamäte.

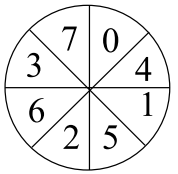
Vzniká otázka, prečo radič používa svoj buffer a nekopíruje bajty priamo do hlavnej pamäte po tom, ako ich získa z disku. Dôvodom je, že keď je začatý diskový prenos, bity prichádzajú z disku konštantnou rýchlosťou bez ohľadu na to, či je radič pripravený alebo nie. Ak by sa radič pokúšal priamo zapísať dáta do pamäte, musia ísť cez systémovú zbernicu, ktorá môže byť zamestnaná iným prenosom a radič bude musieť čakať. Ak príde z disku ďalšie slovo pred tým, než bolo predošlé uložené do pamäte, radič ho bude musieť niekam uchovať. Ak je zbernica príliš zaťažená, môže radič potrebovať množstvo slov na uloženie a na to bude treba množstvo administrácie. Ak sa blok uloží do vnútorného buffera, zbernica nie je potrebná, až kým nezačne DMA.

Dvojkrokový proces bufferingu významne vplýva na čas vykonávania V/V. Kým sú dáta prenášané z radiča do pamäte, pod hlavu disku sa dostane ďalší sektor a do radiča prichádza nový tok bitov. Jednoduché radiče nedokážu naraz vykonávať vstup aj výstup, a teda počas prenosu dát do pamäte by sa stratila informácia z ďalšieho sektoru. Toto možno riešiť tak, že radič bude schopný čítať len každý druhý blok, takže čítanie celej stopy bude požadovať dve otáčky.

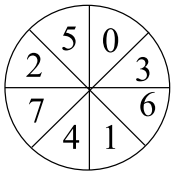
Preskočenie bloku (príp. viacerých) na to, aby mal radič čas na prenos dát do pamäte, sa nazýva *interleaving* (*prekladanie*). Keď sa disk formátuje, bloky sa číslujú na základe „prekladacieho“ faktora. To umožňuje operačnému systému čítať bloky idúce číslovaním za sebou s maximálnou možnou rýchlosťou.



bez prekladania



s jednoduchým prekladaním



s dvojitým prekladaním

hodnotu a ukladá sa ako bezznamienkové číslo. Napr. v štandarde IEEE vo formáte "single precision" sa používa na uloženie reálneho čísla 32 bitov, z toho 1 bit je na znamienko, 8 bitov na zvýšený exponent (pôvodný exponent sa zvýši o 127, čiže pôvodný exponent mohol byť v rozsahu -127 až 128) a 23 bitov na zlomkovú časť mantisy. Čísla vo formáte "double precision" sa ukladajú do 64 bitov, z nich je na zvýšený exponent vyhradených 11 bitov (pôvodný exponent sa zvýši o 2047) a na zlomkovú časť mantisy sa používa 52 bitov.

1.3 Jazyk asemblera

Jazyk asemblera (asembler) je mnemonický jazyk, ktorý nahrádza inštrukcie strojového jazyka mnemonikami (symbolmi).

Na rozdiel od jazykov vyššej úrovne nie je asembler prenositeľný, lebo je úzko spätý so strojovým jazykom daného počítača, s jeho architektúrou.

Tým však programátor môže plne využiť všetky výhody architektonických čŕt počítača. Programy v jazyku asemblera majú minimálny čas vykonávania a efektívne využívajú systémové prostriedky.

1.3.1 Typy a formát inštrukcií

Základné informácie o programovaní v jazyku asemblera si uvedieme pre asembler počítača VAX.

VAX asembler používa 3 typy inštrukcií:

- *strojové inštrukcie (výkonné)* - tie, ktoré sú prekladané do strojového kódu a vykonávajú nejaké operácie
- *direktívy (nevýkonné)* - riadiace informácie pre prekladač (napr. na rezervovanie miesta pre premenné), začínajú bodkou
- *makroinštrukcie* - pseudoinštrukcie zavedené používateľom

Strojové inštrukcie môžeme ďalej rozdeliť na 4 základné skupiny:

- *prenos dát*
- *aritmetické a logické operácie*
- *riadenie programu* - rozhodovania a skoky
- *vstupno-výstupné inštrukcie*

Formát inštrukcie:

[Návestie :] KódOperácie [Operand(y)] [;Komentár]

Zvyčajne posledný operand je *cieľový* – teda ten, do ktorého sa uloží výsledok operácie.

VAX asembler používa 16 registrov veľkosti 32 bitov (= 4 bajty = dlhé slovo–*longword*):

R0 - R11 sú všeobecné registre (používané na ukladanie medzivýsledkov)
R12 = AP – Argument Pointer
R13 = FP – Frame Pointer
R14 = SP – Stack Pointer
R15 = PC – Program Counter

  - udržiavaním smerníka do tabuľky procesov a zvyšovaním priamo počítadla v položke pre proces

- Ošetrovanie systémového volania alarm vyvolávaného používateľskými procesmi.
- Poskytovanie timerov pre časti systému (watchdog timer), napr. ak sa 3 sekundy nič nedeje s floppy diskom, vypne sa motor.
- Monitorovanie a štatistiky.