Operačné systémy

Kapitola 1

Systémové programovanie

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

Jednou z charakteristík, ktorou sa väčšina systémových programov odlišuje od aplikačných programov je *závislosť na počítači (procesore)*.

Aplikačný program sa hlavne sústreďuje na riešenie nejakého problému, pričom používa počítač ako prostriedok. Systémové programy majú podporovať operácie a použitie počítača samotného, nie jednotlivých aplikácií. Preto sa zvyčajne vzťahujú k štruktúre počítača, na ktorom bežia. Napr. asemblery prekladajú mnemonické inštrukcie do strojového kódu, takže formát inštrukcií, adresné módy atď. priamo ovplyvňujú design asemblera. Podobne kompilátory generujú strojový kód berúc do úvahy také hardwarové charakteristiky ako počet a použitie registrov a dostupné strojové inštrukcie. Operačné systémy riadia všetky prostriedky počítačového systému.

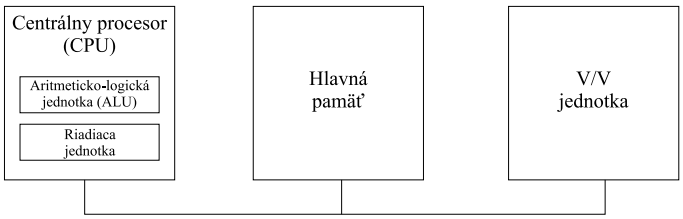
Na druhej strane sú isté aspekty systémového softwaru, ktoré priamo nesúvisia s typom systému, na ktorom pracujú. Napr. všeobecný design a logika asemblera je v základe rovnaká na všetkých procesoroch. Niektoré techniky optimalizácie kódu používané kompilátormi sú nezávislé od počítača. Podobne linkovanie nezávisle asemblerom prekladaných podprogramov zvyčajne nezávisí od použitého počítača.

Okrem operačného systému, ktorý je najzákladnejší systémový program, medzi systémové programy ďalej patria asemblery, kompilátory, makroprocesory, linkre, loadre, editory, debbugovacie systémy.

1.1 Štruktúra počítača

Zjednodušený model typického počítača - ako ho zaviedol v polovici 40-tych rokov 20. stor. matematik John von Neumann - sa skladá z nasledujúcich častí:

- *centrálny procesor (central processing unit)* – pozostáva z riadiacej jednotky, aritmeticko-logickej jednotky a internej pamäte (pracovných registrov – na uchovanie informácie, ktorá má byť rýchlo dostupná)
- *hlavná pamäť* – slúži na uchovávanie informácií a inštrukcií
- *vstupno-výstupná jednotka* – spája počítač s periférnymi zariadeniami



Niektoré registre slúžia na špeciálne účely, napr. *instruction register (IR)* na uloženie práve vykonávanej inštrukcie, *program counter (PC)* na uloženie adresy nasledujúcej inštrukcie, *stack pointer (SP)* na prístup k zásobníku, *stavové slovo procesora – processor status word (PSW)*, ktorý obsahuje informácie o stave súčasného procesu.

1.2 Reprezentácia dát

Počítače slúžia na spracovanie dát. Preto je dôležité vedieť, s akými typmi dát pracujú, aké operácie s nimi môžu vykonávať a ako sú dáta reprezentované v počítači.

Dátový typ je definovaný svojou:

- množinou hodnôt alebo prvkov
- množinou operácií na prvkoch

Na dátový typ sa možno pozerať 3 spôsobmi:

- ako na množinu abstraktných entít a príslušných operácií, ktoré nemajú vzťah k počítaču – *abstraktný dátový typ*
- ako na entity, ktoré definuje a používa nejaký programovací jazyk – *virtuálny dátový typ*
- ako na entity, ktoré sú fyzicky uložené a s ktorými narába hardware počítača – *fyzický dátový typ*

My sa teraz zaujímame o fyzické dátové typy. Všetky dáta sú reprezentované ako skupiny bitov. Vzťah medzi množinou bitov a prvkami typu sa nazýva *kód (kódovanie)*. Použitý kód určuje fyzickú reprezentáciu prvkov dátového typu.

1.2.1 Numerické dátové typy

Počítač pracuje s dvomi hlavnými typmi numerických dát: s celými číslami (integer data types) a číslami v pohyblivej rádovej čiarke (floating point data types).

Najprirodzenejší spôsob reprezentácie nezáporných celých čísel je reprezentácia v dvojkovej sústave.

Pre reprezentáciu záporných celých čísel sú možné tri prístupy:

- *sign and magnitude*: najľavejší bit určuje znamienko čísla (0=kladné, 1=záporné), ostatné bity dávajú absolútnu hodnotu čísla. Nevýhody: 1. dve reprezentácie čisla 0, 2. obvody pre sčítanie čísel sa nedajú použiť pre odčítanie.
- *1's complement (doplnok do 1)*: záporné číslo získame z kladného čísla (ktoré má v najľavejšom bite 0) negáciou po bitoch. Nevýhoda: dve reprezentácie čísla 0. V tomto prípade sa sčítací obvod dá použíť pre odčítanie (pripočíta sa číslo opačné a k výsledku sa pripočíta bit prenosu - "end around carry").
- *2's complement (doplnok do 2)*: záporné číslo vznikne ako negácia kladného čísla po bitoch zväčšená o 1. U tejto reprezentácie už nie sú dve rôzne reprezentácie nuly a sčítací obvod sa dá použiť na odčítanie (bit prenosu - carry bit - sa ignoruje).

Čísla v pohyblivej rádovej čiarke treba previesť do dvojkovej sústavy a zapísať v *normalizovanom tvare*: $(-1)^{znamienko} * mantisa * 2^{exponent}$, kde mantisa je jednoznačne určená v závislosti od použitého formátu (napr. pre VAX: pred desatinnou čiarkou je 0 a bezprostredne za ňou je číslica 1; IEEE štandard požaduje, aby to bolo číslo v tvare "1,zlomok").

V závislosti od požadovaného rozsahu a presnosti čísel potom jednotlivé formáty ukladajú mantisu a exponent do istého počtu bitov. Exponent sa zvyčajne zvýši o nejakú hodnotu N, aby mal kladnú

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

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.

5. **Relatívny mód: adresa**

   Používa sa pre operandy uložené v pamäti, ktoré sú určené adresou (návestím).

   Napr. MOVL A, R10

   hovorí, že sa má obsah pamäťového miesta veľkosti 4 bajty s adresou A presunúť do registra R10.

   Preklad do strojového kódu: pri preklade do strojového kódu sa neuloží priamo adresa A, ale rozdiel medzi adresou A a obsahom PC registra (teda sa prekladá relatívne k PC registru). Na uloženie vypočítaného rozdielu sa vezme najmenší možný priestor (1, 2 alebo 4 bajty), do ktorého sa zmestí. Preklad operandu v relatívnom móde sa potom skladá z 2, 3 alebo 5 bajtov. Prvý bajt (informačný) obsahuje v pravom polbajte F (PC register) a v ľavom polbajte A, C alebo E podľa toho, či rozdiel vojde do 1, 2 alebo 4 bajtov. Nasledujúce 1, 2 alebo 4 bajty obsahujú rozdiel.

   Výhodou takéhoto prekladu je to, že je nezávislý od umiestnenia programu v pamäti. Spomínaný rozdiel je vlastne vzdialenosť pamäťového miesta, s ktorým inštrukcia narába, od tejto inštrukcie a táto vzdialenosť je rovnaká bez ohľadu na to, kde je program umiestnený. Ďalšou výhodou je, že rozdiel je možné vypočítať v čase prekladu z "logických" (relatívnych) adries – program adresujeme od 0 – a netreba poznať adresu, na ktorú bude program do pamäte zavedený.

   Adresa operandu sa vypočíta pri vykonávaní inštrukcie ako súčet obsahu PC registra (to už bude "fyzická" adresa) a rozdielu.

   Nech napr. (relatívna) adresa A je 0002 (hexadecimálne) a nech vyššieuvedená inštrukcia začína na adrese 0142. Na uloženie rozdielu budú potrebné 2 bajty. PC register bude v čase určovania rozdielu (a tiež v čase určovania adresy operandu) ukazovať na bajt nasledujúci za miestom na uloženie rozdielu, takže v našom príklade bude jeho hodnota 0146 (adresa 0142 = kód inštrukcie, 0143 = informačný bajt CF – rozdiel je v 2 bajtoch, 0144 a 0145 = rozdiel). Takže rozdiel je: 0002 - 0146 = FEBC.

   Preklad inštrukcie do strojového kódu: 5A FE BC CF D0

6. **Literál a priamy mód: #číslo alebo #výraz**

   Operandom je priamo hodnota uvedená v inštrukcii. Môže to byť celočíselná konštanta alebo konštanta v pohyblivej rádovej čiarke. Táto konštanta môže byť opísaná číslom alebo výrazom (zvyčajne sa používa len symbol).

   Literál a priamy mód vyzerajú rovnako, líšia sa však prekladom do strojového kódu (veľkosťou miesta na ich uloženie). Pod pojmom literál myslíme celočíselnú konštantu od 0 po 63 (max. 6 bitov) – pri preklade do strojového kódu sa používa len 1 bajt a doň sa priamo zapíše hodnota.

   Príklad: MOVL #25, R11 (do registra R11 vlož číslo 25)

   Preklad do strojového kódu: 5B 19 D0

   Rovnako sme mohli definovať konštantu a potom ju použiť v inštrukcii — preklad do strojového kódu je rovnaký:

   MAX=25 MOVL #MAX, R11

   Priamy mód zaberá 2, 3 alebo 5 bajtov – podľa veľkosti dát, s ktorými narába inštrukcia. Prvý bajt obsahuje vždy 8F a v nasledujúcich 1, 2 alebo 4 bajtoch je uložená konštanta.

   Príklad: MOVL #-2, R11 (do registra R11 vlož číslo -2)

   Preklad do strojového kódu: 5B FF FF FF FE 8F D0 (na konštantu sme použili 4 bajty, lebo inštrukcia MOVL narába s longwordami)

   Ak by sme mali inštrukciu MOVB #-2, R11, preklad by bol 5B FE 8F 90 (kód inštrukcie MOVB je 90, konštanta je uložená do 1 bajtu, pretože inštrukcia MOVB narába s bajtami).

7. **Nepriama adresácia s doplnkom: d(Rn)**

   Adresa operandu sa vypočíta tak, že sa k obsahu registra Rn pripočíta číslo (doplnok) uvedené pred zátvorkou (POZOR! Obsah registra Rn sa nezmení.).

   Doplnok môže byť výraz, ale zvyčajne sa používa len číslo (môže byť kladné aj záporné).

Príklad: MOVL 28(R5), R9

Obsah pamäťového miesta veľkosti 4 bajty s adresou, ktorú vypočítame ako súčet obsahu registra R5 a čísla 28, sa presunie do registra R9.

Preklad do strojového kódu: preklad operandu s doplnkom zaberá 2, 3 alebo 5 bajtov, v závislosti od veľkosti miesta potrebného na uloženie doplnku (prekladač sa snaží uložiť doplnok do najmenšieho miesta, do ktorého sa zmestí). Prvý bajt je informačný – v pravom polbajte obsahuje číslo registra, vzhľadom na ktorý sa adresuje, v ľavom polbajte je A, C alebo E, podľa toho, či doplnok vojde do 1, 2 alebo 4 bajtov. Nasledujúce 1, 2 alebo 4 bajty slúžia na uloženie doplnku v reprezentácii doplnok do 2.

Preklad uvedenej inštrukcie bude: 59 1C A5 D0 (informačný bajt je A5 - adresuje sa vzhľadom k registru R5 a doplnok sa uloží do 1 bajtu, doplnok $28_{10} = 1C_{16}$)

1.3.3 Štruktúra programu

Program v jazyku asemblera má nasledovnú štruktúru:

- deklarácia premenných a konštánt
- definície procedúr a makier
- hlavný program

Deklarácia premenných a konštánt

- *premenné*: pomocou direktívy .BLKx n sa vyhradí miesto pre 'n' bajtov, slov, dlhých slov – podľa toho, či sme namiesto 'x' použili B, W alebo L.

  Pre inicializáciu premenných (vyhradenie miesta spolu s priradením počiatočnej hodnoty) sa používajú direktívy .BYTE zoznam, .WORD zoznam alebo .LONG zoznam, kde 'zoznam' obsahuje hodnoty priradené do vyhradených pamäťových miest oddelené čiarkami.

  Napr. A: .BLKL 10 – vyhradí 10 dlhých slov (40 bajtov) a označí ich adresou A.

  B: .LONG 10,2 – na adrese B sa vyhradia dve dlhé slová, do prvého sa vloží hodnota 10, do druhého hodnota 2.
- *konštanty*: meno = výraz

1.3.4 Niektoré príkazy jazyka asemblera

V názve inštrukcie budeme používať písmená x, y na označenie rozmeru dát, s ktorými narábame (môže to byť B = bajt, W = word, L = longword).

Aritmetické operácie

CLRx	čo	čo:=0
INCx	čo	čo:=čo+1
DECx	čo	čo:=čo-1
MNEGx	čo, kam	aritmetická negácia (kam:=-čo)
ADDx2	čo , kam	kam:= kam + čo
ADDx3	čo1, čo2, kam	kam:= čo2 + čo1
SUBx2	čo, kam	kam:= kam - čo
SUBx3	čo1, čo2, kam	kam:= čo2 - čo1
MULx2	čo, kam	kam:= kam * čo
MULx3	čo1, čo2, kam	kam:= čo2 * čo1
DIVx2	čo, kam	kam:= kam div čo
DIVx3	čo1, čo2, kam	kam:= čo2 div čo1

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.

Nepodmienené skoky

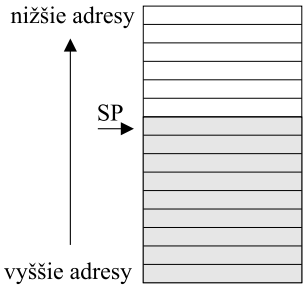
Nepodmienené skoky vždy zmenia obsah PC registra.

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.

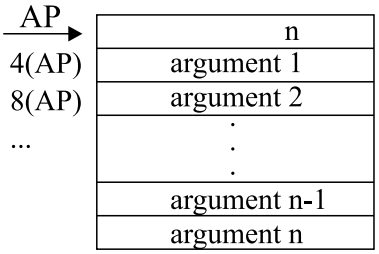
1.3.5 Procedúry

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 deklarácí), 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:



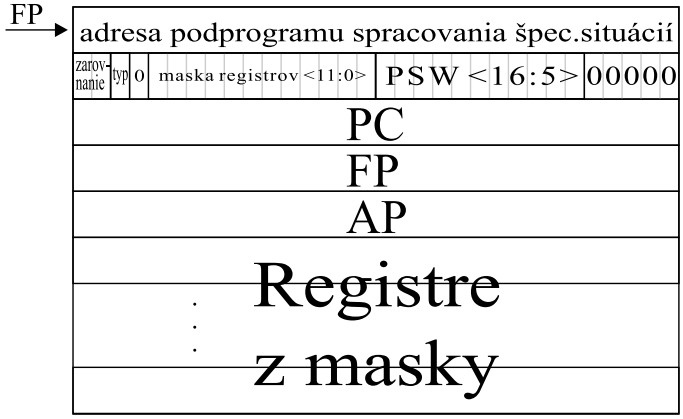
Formát procedúry:

(dátové definície, ak sú)
.ENTRY meno, maska_registrov
príkazy
RET

V maske registrov sú vymenované registre (R2 – R11), ktoré majú byť odložené do zásobníka pri vstupe do procedúry a po jej dokončení obnovené. Maska registrov má tvar: ^M<zoznam_registrov>. Registre AP, FP a PC budú uložené automaticky.

Ďalšou štandardizovanou dátovou štruktúrou pre volanie procedúry je *blok volania (call frame)* – slúži na uchovanie registrov a ďalšej informácie o stave procesu pri volaní procedúry. Je automaticky ukadaný do zásobníka pri oboch spôsoboch volania procedúry.

Formát bloku volania:



Najvrchnejšie dlhé slovo obsahuje adresu podprogramu spracovania špeciálnych situácií (condition handler address). Ak sa v procedúre objaví chyba, sem sa uloží adresa podprogramu, ktorý ju spracuje. Inak je tam uložená 0.

Ďalšie dlhé slovo obsahuje viaceré informácie:

- zarovnanie: 2 bity nadobúdajúce hodnotu 0 – 3, určujúce potrebné zarovnanie v momente volania procedúry (pretože blok volania musí byť vždy uložený od adresy, ktorá je násobkom 4).
- typ volania: 1 bit obsahujúci 0, ak sa vykonalo volanie CALLG, 1, ak sa vykonalo CALLS.
- maska registrov: 12 bitov pre registre z masky (R0 – R11)
- stavové slovo procesora: 16 bitov. Bity 0 – 4 stavového slova procesora sú vždy pred uložením vymazané. Procedúra môže tieto bity nejako nastavovať a indikovať pomocou nich nastatie nejakej podmienky. Po návrate do hlavného programu sa PSW obnoví a uvedené bity slúžia ako príznaky nejakých udalostí.

Na vrch bloku volania ukazuje register R14 = FP (*Frame Pointer*).

Volanie CALLG:

Ako príklad uvedieme procedúru SORT, ktorá má 2 vstupné argumenty: adresu triedeného poľa a dĺžku poľa.

Pri volaní CALLG musíme vyhradiť mieto pre argumenty v časti deklarácí.

```
POLE:       .BLKL 100
ARG_LIST:   .LONG 2             ;počet argumentov
            .ADDRESS POLE       ;direktíva na vyhradenie 4 bajtov a vloženie adresy
DLZ:        .BLKL 1             ;miesto pre dĺžku poľa
```

Volanie procedúry:

```
MOVL DLZKA, DLZ
CALLG ARG_LIST, SORT
```

(do AP registra sa dá adresa uvedená vo volaní ako 1. argument)

Nevýhodou tohto typu volania je to, že argumenty procedúry sú uložené v programe na inom mieste, než je volanie, čo môže spôsobovať neprehľadnosť pri čítaní programu a tiež to, že tento typ nie je vhodný pre rekurzívne procedúry.

Volanie CALLS:

Argumenty sa pred volaním ukladajú do zásobníka a ich počet sa odovzdá procedúre ako argument (hneď po zavolaní procedúry sa toto číslo automaticky zapíše do zásobníka – na vrch zoznamu argumentov – a naň sa nastaví AP register).

Volanie procedúry:

```
PUSHL DLZKA
PUSHAL POLE
CALLS #2, SORT
```

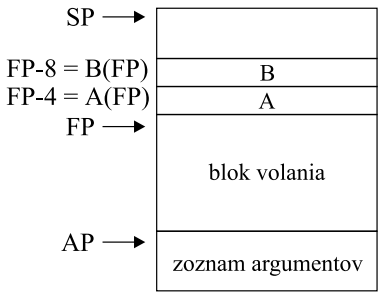
Lokálne premenné:

V zásobníku je možné uchovávať počas behu procedúry lokálne premenné a adresovať ich cez FP register.

Napr. chceme v procedúre PROC používať 2 lokálne premenné – A, B.

```
.ENTRY PROC, ^M<...>
A=-4
B=-8
SUBL2 #8, SP    ;urobiť miesto pre 2 dlhé slová na zásobníku
...
MOVL R0, A(FP)
MOVL R1, B(FP)
...
RET
```

Lokálne premenné adresujeme vzhľadom na FP register, a nie vzhľadom k SP, lebo SP sa môže meniť – zásobník sa môže používať aj na lokálne výpočty.



Návrat z procedúry:

Návrat z procedúry zabezpečuje inštrukcia RET, ktorá zo zásobníka vyberie blok volania (naplní registre PC, AP, FP a registre z masky pôvodnými hodnotami, naplní PSW uloženými údajmi), ak išlo o volanie CALLS vyberie aj zoznam argumentov a príslušne zmení SP (tým automaticky zruší alokáciu

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.

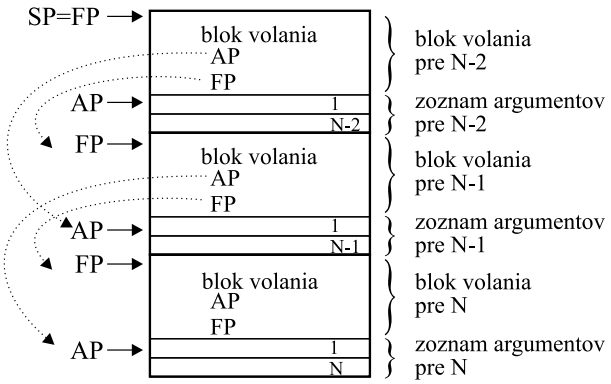
Ako príklad uvedieme výpočet faktoriálu: $N! = N.(N-1)!$, ak $N > 0$, $N! = 1$, ak $N = 0$.

```
        .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

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]
```

telo makra (inštrukcie, direktívy, volania alebo definície makier)

.ENDM [meno]

Parametre makra sú oddelené čiarkami, medzerami alebo tabulátormi. Môžu mať zadanú *implicitnú hodnotu*, ktorá sa dosadí za parameter, ak pri volaní makra nebude daná hodnota tohto parametra. Implicitná hodnota je zadaná tak, že v definícii makra za menom parametra nasleduje rovnítko a hodnota parametra.

Volanie makra:

meno [hodnoty_parametrov]

Príklad: makro na výmenu obsahu dvoch premenných

```
.MACRO VYMEN P1,P2,P3=POM
MOVL V1,V3
MOVL V2,V1
MOVL V3,V2
.ENDM VYMEN
```

Volanie: VYMEN R2,R7 má rozvoj:

```
MOVL R2,POM
MOVL R7,R2
MOVL POM,R7
```

Volanie: VYMEN R2,R7,R11 má rozvoj:

```
MOVL R2,R11
MOVL R7,R2
MOVL R11,R7
```

čiže, ak bola zadaná hodnota parametra, má prednosť pred implicitnou hodnotou danou v definícii makra.

Hodnoty parametrov makra môžu byť zadané dvoma spôsobmi:

- *pozične*: hodnoty pre parametre sú uvedené v takom poradí, ako sú parametre v definícii makra. Ak niektorý parameter (nie posledný) má implicitnú hodnotu, ktorú vo volaní chceme ponechať, musí vo volaní makra byť zadaná „prázdna hodnota“ – tj. idú za sebou 2 čiarky.
- *nepozične*: hodnoty nemusia byť zadané v presnom poradí podľa definície, ale sú zadávané v tvare parameter = hodnota_parametra

Príklad:

.MACRO XX MENO,DLZ=#20,DOL=#0,HOR=#19,TYP=L

má 5 parametrov, z ktorých 4 majú implicitnú hodnotu. Ak chceme volať toto makro a zadať parameter MENO s hodnotou POLE a HOR s hodnotou #100, tak v prípade pozičnej syntaxe použijeme volanie:

XX POLE,,,#100

a pri nepozičnej syntaxi:

XX MENO=POLE,HOR=#100 alebo aj XX HOR=#100,MENO=POLE (nemusíme dodržať poradie parametrov, ako bolo v definícii)

Možná je aj kombinácia pozičného a nepozičného volania, ale vždy musí začať pozičné a potom nepozičné (za ním už pozičnú syntax nemožno použiť):

XX POLE,HOR=#100

Spájanie parametrov:

Niekedy je užitočné spojiť parameter s textom – používa sa na to operátor spojenia: apostrof.

Napr.

.MACRO SUM A,B,C,TYPE

```
ADD'TYPE'3 A,B,C
.ENDM
```

má pri volaní SUM R3,R4,R7,W rozvoj ADDW3 R3,R4,R7.

Ak treba spojiť 2 parametre, medzi ne dáme dva apostrofy:

```
.MACRO XXX A,B,C,OP,TYPE
OP''TYPE'3 A,B,C
.ENDM
```

má pri volaní XXX R3,R4,R7,MUL,B rozvoj MULB3 R3,R4,R7.

Návestia v makrách:

Majme makro na výpočet absolútnej hodnoty premennej:

```
        .MACRO ABS CO,KAM
        MOVL CO,KAM
        BGEQ KON
        MNEGL KAM,KAM
KON:    .ENDM ABS
```

Ak sa toto makro volá len raz, nevznikne problém, ale ak bude volané viackrát, v programe sa vyskytne viacero návestí KON.

Jedno možné riešenie je pridať parameter makra NAV:

```
        .MACRO ABS CO,KAM,NAV
        MOVL CO,KAM
        BGEQ NAV
        MNEGL KAM,KAM
NAV:    .ENDM ABS
```

takže ak pri rôznych volaniach budeme zadávať rôzne hodnoty parametra NAV, konflikt nevznikne – je to ale pre používateľa veľmi „nepohodlné“ riešenie.

Druhou možnosťou je špecifikovať v zozname parametrov makra *lokálne návestia* (majú tvar n$ a platia v úseku medzi dvoma užívateľsky definovanými návestiami), ktoré budú automaticky pri rozvoji makra nahradzované hodnotami, ktoré sa nebudú opakovať – vkladajú sa návestia od 30000$.

```
        .MACRO ABS CO,KAM,?NAV
        MOVL CO,KAM
        BGEQ NAV
        MNEGL KAM,KAM
NAV:    .ENDM ABS
```

Pri volaní ABS A,B vznikne rozvoj:

```
        MOVL A,B
        BGEQ 30000$
        MNEGL B,B
30000$: .ENDM ABS
```

Pri ďalšom volaní sa na miesto parametra NAV vloží 30001$, potom 30002$ atď.

Makrá definujúce makrá:

Ak sa v tele makra nachádza definícia ďalšieho makra, tak „vnútorné“ makro nemožno použiť, pokiaľ sa nezrealizovalo volanie „vonkajšieho“ makra.

Príklad:

```
.MACRO DEF MENO
...
.MACRO MENO A
CLRL A
.ENDM MENO
...
```

```
.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.

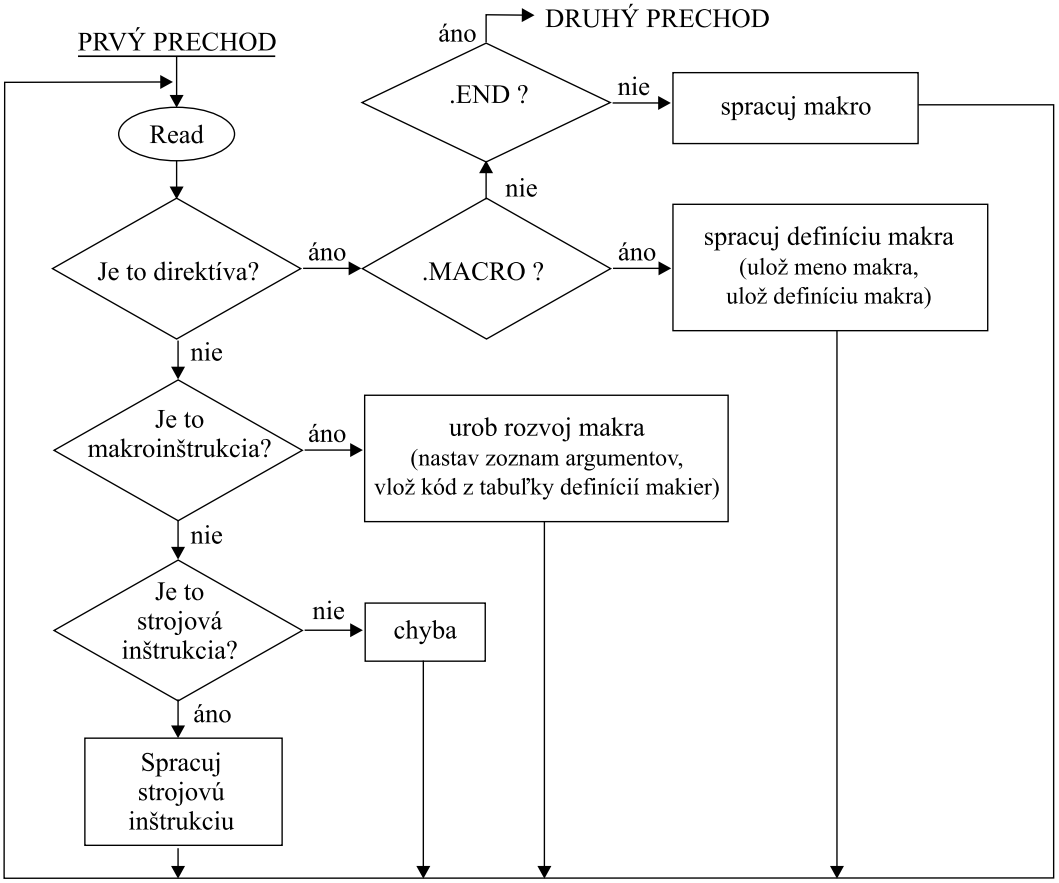
Jednoprechodový makroprocesor

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

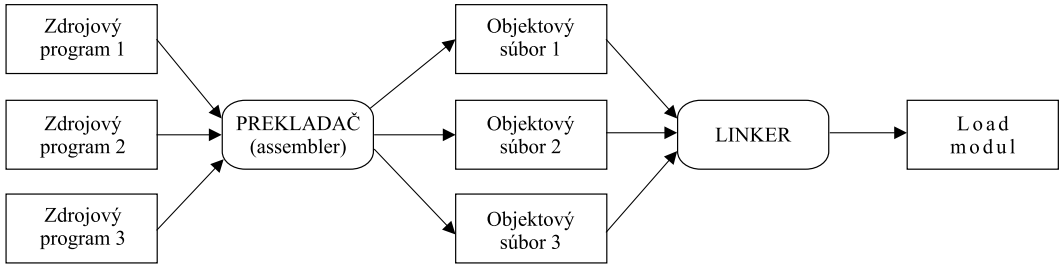
Makroprocesor sa môže pridať ako predprocesor pred asembler, ale je tiež možné implementovať jednoprechodový makroprocesor do prvého prechodu asemblera – výsledok sa nazýva *makroasembler*.

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.

Linker

Pri štarte prvého prechodu asemblera sa nastaví location counter (LC) na 0. Tento krok je ekvivalentný predpokladu, že objektový modul bude umiestnený na (virtuálnej) adrese 0.

Linker, ktorý spája určené moduly do jedného celku, tiež zvyčajne predpokladá, že program začína na adrese 0 (v takomto prípade vytvára „relative load modul"). Keďže na túto adresu možno umiestniť len jeden modul, ostatné musí linker zaradiť zaň. V týchto moduloch musí linker upraviť adresy podľa toho, kde začínajú. K adresám v týchto moduloch sa pripočítava tzv. *relokačný faktor*. Toto je však potrebné len u adries, ktoré nie sú prekladané relatívne, čiže vzhľadom k PC registru.

Pri spájaní modulov musí linker vedieť, ktoré adresy sú v poriadku a ktoré treba relokovať. Túto informáciu mu zapíše asembler do objektového modulu. Ak všetky pamäťové odkazy v module sú vzhľadom k PC registru, nemusí linker robiť žiadne úpravy adries. Takéto moduly nazývame *nezávislé od umiestnenia (position independent code)*.

Ďalej musí linker vyriešiť odkazy medzi modulmi (napr. volanie procedúry definovanej v inom module). Počas prekladu asembler nemôže na miesta týchto odkazov vložiť adresy odkazovaných procedúr (ani relatívne). Návestia (symboly) definované v iných moduloch, než je práve prekladaný modul, sú pre tento modul *externé* (na rozdiel od tých, čo sú definované v súčasnom module, ktoré nazývame *interné* alebo *lokálne*). Asembler uloží informáciu o externých návestiach v objektovom súbore.

Ak k nejakému externému návestiu nenájde linker v ostatných moduloch jeho definíciu, čiže nebude v niektorom module toto návestie definované ako *globálne*, tak vyhlási chybu.

Linker spája separátne adresové priestory objektových modulov do jedného lineárneho adresného priestoru v nasledovných krokoch:

1. Vytvorí tabuľku objektových modulov a ich dĺžok.
2. Na základe tejto tabuľky priradí začiatočné adresy jednotlivým objektovým modulom.
3. Nájde všetky inštrukcie obsahujúce pamäťové adresy a pripočíta k týmto adresám relokačný faktor, rovný začiatočnej adrese modulu, v ktorom sa vyskytuje.
4. Nájde všetky inštrukcie obsahujúce odkazy do iných modulov a naplní tieto odkazy adresami referencovaných objektov.

Štruktúra objektového modulu

Objektový modul (súbor) pozostáva zo šiestich častí:

- **Identifikácia**: meno modulu, čas prekladu, niektoré informácie potrebné pre linker, ako napr. dĺžky jednotlivých častí objektového modulu.
- **Tabuľka globálnych symbolov (Entry point table)**: zoznam symbolov definovaných v module, na ktoré sa môžu odkazovať iné moduly, spolu s ich hodnotami (adresami).
- **Tabuľka externých symbolov (External reference table)**: zoznam symbolov použitých v module, ktoré v ňom nie sú definované, spolu so zoznamom inštrukcií, ktoré ich používajú.
- **Preložený kód (Machine instructions and constants)**: to je jediná časť objektového modulu, ktorá bude nahratá do pamäte na vykonávanie.
- **Tabuľka relokácií (Relocation dictionary)**: zoznam adries, ktoré musia byť relokované pripočítaním relokačného faktora.
- **End-of-module**: adresa začiatku programu – štartovacia adresa (ak ide o hlavný program), prípadne „checksum" na kontorlu chýb pri čítaní modulu.

Väčšina linkerov pracuje v dvoch prechodoch. V prvom prechode linker číta všetky objektové moduly a vyrobí tabuľku názvov objektov a ich dĺžok a tiež *globálnu tabuľku symbolov (global symbol table)* pozostávajúcu zo všetkých globálnych a externých symbolov. V druhom prechode sú objektové moduly čítané, relokované a spojené do jedného modulu.

Loader

Loader umiestňuje load modul do operačnej pamäte a pripraví ho na spustenie. Keďže začiatočná adresa modulu, ktorú predpokladal linker, je zvyčajne rôzna od adresy, na ktorú je program zavedený, loader musí tiež upraviť adresy. Preto musí byť súčasťou load modulu zoznam adries, ktoré treba takto modifikovať. Loader použije uvedenú informáciu na úpravu adries, ale z výslednej podoby strojového kódu ju vymaže. Vykonávanie začina, keď sa urobí skok na štartovaciu adresu programu.

„Čas viazania" (Binding time) a dynamická relokácia

V systémoch so zdieľaním času môžu byť programy umiestnené do pamäte, potom na nejaký čas presunuté na disk a potom opäť nahraté do pamäte. Zvyčajne sa nedá zabezpečiť, aby sa program nahral späť do pamäte na tú istú adresu, ako bol predtým. Ak bol program relokovaný, po opätovnom nahratí do pamäte sú všetky pamäťové odkazy nesprávne. Ak by aj bola ešte dostupná informácia o relokáciách, zaberalo by to mnoho času každý raz po presune programu relokovať všetky adresy.

Problém presúvania zlinkovaných a relokovaných programov súvisí s časom, kedy sa robí „viazanie" (mapovanie) symbolických mien na fyzické adresy. Existuje aspoň 6 možností na „čas viazania" (binding time):

- Keď sa program píše.
- Keď sa program prekladá.
- Keď sa program linkuje, ale pred loadovaním (v tomto a predošlom prípade vzniká „absolute load modul".
- Keď sa program loaduje (nahráva do pamäte).
- Keď sa loaduje bázový register používaný na adresovanie.
- Keď sa vykonáva inštrukcia obsahujúca adresu.

Ak napr. prekladač vytvára priamo „absolute load modul", „viazanie" prebehlo v čase prekladu a program musí byť spustený na adrese, ktorú predpokladal prekladač.

Tu sa vlastne stretávame s dvoma súvisiacimi problémami: prvý – kedy sa symbolické mená mapujú na virtuálne adresy, druhý – kedy sa virtuálne adresy mapujú na fyzické adresy. Až keď prebehnú obe tieto operácie, ukončí sa „viazanie". Keď linker spája separátne adresové priestory do jedného, v skutočnosti vlastne vytvára virtuálny adresný priestor. Relokácia a linkovanie slúžia na namapovanie symbolických mien na určité virtuálne adresy. Toto platí bez ohľadu na to, či systém používa virtuálnu pamäť (kap. 10).

Ak napr. systém používa mechanizmus „run-time" relokačného registra, tak tento register vždy ukazuje na začiatok súčasného programu. K všetkým pamäťovým adresám sa hardwarovo pripočíta obsah relokačného registra, skôr než sa pošlú do pamäte. Keď sa program presunie v pamäti, operačný systém musí zmeniť obsah relokačného registra.

Dynamické linkovanie

Metóda linkovania, ako sme si ju vysvetlili, má tú vlastnosť, že všetky procedúry, ktoré by mohol program volať, sú zlinkované pred spustením programu. Mnoho programov však má procedúry, ktoré sú volané len pri „nezvyčajných" okolnostiach.

Flexibilnejšia je metóda, pri ktorej budú procedúry linkované až pri ich prvom použití. Tento proces je známy ako *dynamické linkovanie*. Umiestnenie preložených modulov na disku je niekde zapamätané (napr. v adresári), takže linker ich môže ľahko nájsť, keď ich bude potrebovať. Keď sa v programe volá procedúra z iného modulu, linker nájde príslušný modul, pridelí mu virtuálnu adresu a vyrieši odkaz na procedúru. Inštrukcia volania procedúry sa opätovne spustí a umožní pokračovanie programu od miesta, kde bol prerušený.

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

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 elektróniek 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.

Najväčšia sila idey „jednej rodiny“ bola súčasne aj jej najväčšou slabosťou: bolo snahou, aby všetok software, vrátane OS, pracoval na všetkých modeloch a bol pre všetky dosť výkonný. OS bol preto enormne veľký a zložitý, s mnohými chybami a nutnosťou nepretržitého toku nových verzií na opravu týchto chýb.

Napriek enormnej veľkosti a problémom, OS/360 a podobné OS 3. generácie uspokojovali väčšinu zákazníkov. Tiež priniesli niektoré kľúčové techniky:

- *multiprogramovanie* (rozdelenie pamäte na niekoľko častí, pričom v každej je iná úloha. Kým jedna úloha čaká na V/V, iná môže využívať CPU. Implikuje to nutnosť špeciálneho hardwaru na ochranu úloh.)
- *spooling* (Simultaneous Peripheral Operation On Line) — znamenalo to schopnosť čítať úlohy zo štítkov na disk hneď ako boli prinesené do počítačovej miestnosti. Hocikedy bola úloha ukončená, OS mohol nahrať novú úlohu z disku do uvoľnenej časti a spustiť ju. Spooling sa využíval aj na výstup. Eliminovala sa tým potreba malého V/V počítača (1401).

OS 3. generácie boli stále batch systémy — čas medzi zadaním úlohy a získaním výsledku bol často niekoľko hodín. Snaha zrýchliť prácu priniesla

- *time-sharing* — každý používateľ má on-line terminál, počítač môže zabezpečovať rýchlu interaktívnu obsluhu pre mnoho používateľov a tiež pracovať na veľkých batch úlohách v pozadí.

Počas 3. generácie nastal veľký vývoj *minipočítačov*, začínajúci DEC PDP–1 (1961). Mal len 4K 18-bitových slov, ale cena (120000 USD) bola menej než 5% ceny 7094, pričom pre niektoré typy nenumerickej práce bol skoro taký rýchly ako 7094. Bol nasledovaný sériou PDP až po PDP–11.

V tomto období vzniká aj OS Unix, ktorý bol vytvorený pre malé PDP–7, neskôr prenesený na malé PDP–11/20, neskôr sa rozšíril na Interdata 7/32, VAX, Motorola 68000, atď.

Štvrtá generácia počítačov (1980–1990)

- prichádza s vývojom LSI obvodov (Large Scale Integration — obvody veľkej integrácie), ktoré majú tisícky tranzistorov na $1cm^2$. Vznikajú *osobné počítače*. Ich architektúra sa nelíšila od triedy PDP–11, ale líšili sa cenou, teraz prístupnou jednotlivcom.
- väčšina softwaru je user-friendly — určený pre používateľov, ktorí nevedeli nič o počítačoch (hlavná zmena oproti OS/360 a jeho zložitému JCL — Job Control Language).
- dominujúce sú 2 operačné systémy: MS-DOS (napísaný Microsoft, Inc. pre IBM PC a iné počítače používajúce Intel 8088 procesor) a Unix (na väčších osobných počítačoch používajúcich Motorola 68000 rodinu procesorov). Unix dominuje najmä na ne-Intelovských počítačoch a pracovných staniciach, a to najmä na tých, ktoré sú založené na RISC-čipoch.
- v polovici 80. rokov zaznamenávame nárast sietí osobných počítačov pomocou *sieťových OS* (network OS) a *distribuovaných OS* (distributed OS).

  V sieťových OS sa používateľ môže prihlásiť na vzdialené počítače, kopírovať súbory z jedného počítača na ďalší. Každý počítač má svoj lokálny OS a vlastných používateľov. Sieťové OS nie sú v zásade odlišné od jednoprocesorových OS, zvyčajne potrebujú kontroler sieťového interface a nejaký nízkoúrovňový software na jeho prevádzku a programy na prevedenie vzdialeného prihlásenia a vzdialeného prístupu k súborom.

  Distribuovaný OS sa javí používateľom ako tradičný uniprocesorový systém, hoci je tvorený viacerými procesormi. Používateľ si nemusí uvedomovať, kde sa budú jeho programy spúšťať alebo kde budú jeho súbory umiestnené — to všetko je zabezpečené automaticky operačným systémom. Distribuované OS vyžadujú viac ako pridanie nejakého kódu k uniprocesorovému OS, pretože distribuované systémy sa zásadne líšia od centralizovaných systémov. Napr., distribuované systémy často umožňujú programom bežať na niekoľkých procesoroch v tom istom čase, čo vyžaduje zložitejšie plánovanie procesov.

2.2 História Unixu

- **1. verzia** (1969) — Ken Thompson z Research Group v Bell Laboratories pre PDP–7. Neskôr sa pridal Dennis Ritchie.
- **2. verzia** (1971) na PDP–11/20 (ekvivalent SM3–20)
- **3. verzia** (1973) — výsledok prepísania hlavnej časti OS (asi 97%) do programovacieho jazyka *C*. Unix bol prenesený na vyššie modely PDP–11 (11/45, 11/70).
- **6. verzia** (1976) — prvá verzia rozšírená mimo Bell Laboratories.
- **7. verzia** (1978) — na PDP 11/70 a Interdata 8/32 — je predchodcom väčšiny moderných systémov Unix. Rýchlo sa adaptoval na ostatných modeloch PDP–11 a počítačoch VAX (verzia pre VAX bola známa ako 32V). Po distribúcii verzie 7 prevzala zodpovednosť a administratívnu kontrolu v distribúcii Unixu od Research Groupu *Unix Support Group* (USG) v AT&T, otcovskej organizácii Bell Laboratories.
- **8. verzia** (1985) bola vyvinutá len pre potreby Bell Labs.
- **System III.** (1982) — 1. externá distribúcia. Zahrňovala charakteristiky verzie 7, 32V a iných systémov Unix vyvinutých inými skupinami než Research Group (zahŕňa charakteristiky systému Unix/RT — systém Unix v reálnom čase a mnohé časti Programmer's Work Bench (PWB).
- **System V.** (1983), **Unix System V. Release 2 (V.2)** (1984)
- **3BSD** Na vývoji systémov na báze Unixu začali pracovať aj ďalšie informatické organizácie — najväčší vplyv medzi nimi mala Kalifornská Univerzita v Berkeley. Jej prvá práca na VAXe bolo pridanie virtuálnej pamäte, stránkovania na žiadosť a substitúcie stránok k 32V. Vzniklo tak 3BSD.
- **4BSD** Vývoj štandardnej verzie 4BSD Unixu pre oficiálne použitie sa rozhodla projektovať Defense Advanced Research Projects Agency (DARPA). Jedným z cieľov tohto projektu bolo udržiavať sieťové protokoly sietí DARPA Internet (TCP/IP).

  V Berkeley sa vytvoril nový užívateľský interface *C-shell*, nový textový editor (ex/vi), kompilátory pre Pascal a Lisp a mnohé nové systémové programy. Unixovský software z Berkeley sa rozširuje pod názvom Berkeley Software Distributions (BSD).
- Nasledovníkmi 3BSD sú 4BSD verzie 4.1BSD, 4.2BSD (1983), 4.3BSD. Verzie 2BSD sú pre počítače PDP–11 (verzia 2.9BSD je ekvivalentom 4.2BSD).

V súčasnosti existuje množstvo OS Unix a podobných: DEC ponúka svoj Unix (ULTRIX) pre VAX, Microsoft prepísal Unix pre Intel 8088 — XENIX, Unix pre PC — LINUX, ďalej existuje Unix firiem Amdahl, Sun, NBI, MassComp, Hewlett-Packard, atď. Väčšina je založená na V7, System III., 4.2BSD alebo System V.

Uvádza sa niekoľko dôvodov veľkej popularity Unixu:

- je napísaný v jazyku vyššej úrovne, vďaka čomu sa dá ľahko pochopiť, zmeniť a preniesť na iný počítač
- styk s používateľom je jednoduchý, ale pritom umožňuje poskytovanie všetkých služieb
- umožňuje skladanie zložitých programov z malých, jednoduchých programov
- používa hierarchický systém súborov
- všetky súbory majú jednotný formát (reťazec bajtov)
- poskytuje jednoduchý a jednotný interface k periférnym zariadeniam
- je to multiužívateľský a multiprocesový systém, t.j. súčasne v ňom môže pracovať viacero používateľov a každý z nich môže súčasne spustiť viacero programov
- zakrýva pred používateľom architektúru počítača, takže sa ľahšie píšu programy, ktoré bežia na rôznych hardwarových implementáciách.
- hoci OS a mnohé riadice programy sú písané v jazyku *C*, Unix poskytuje aj iné jazyky — Fortran, Basic, Pascal, Ada, Cobol, Lisp a Prolog.

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

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)

— uloží si parametre z volania na určité miesto, napr. do registrov počítača a vykoná *TRAP-inštrukciu* (druh chráneného volania procedúry) na spustenie operačného systému. Význam použitia knižničnej procedúry je v tom, že ukryje detaily TRAP-inštrukcie a spôsobí, že systémové volanie vyzerá ako obyčajné volanie procedúry.

Keď OS dostane riadenie po TRAPe, preverí, či sú parametre platné a keď áno, vykoná vyžadovanú úlohu. Po ukončení vloží do registra *stavový kód* a vykoná inštrukciu „návrat z TRAPu“, aby vrátil riadenie knižničnej procedúre. Tá vráti volajúcej procedúre stavový kód ako funkčnú hodnotu (normálnym spôsobom sa knižničná funkcia ukončí).

Množstvo a typy systémových volaní závisia od OS — zvyčajne sú to volania na vytváranie procesov, správu pamäte, čítanie a zápis do súborov, V/V (z terminálu, na tlačiareň a pod.).

Shell

OS je program, ktorý obhospodaruje systémové volania. Editory, prekladače, assemblery, linkre a command interpretre nie sú časťou OS, hoci sú veľmi dôležité a užitočné.

Unixovský *command interpreter* (interpreter príkazov) sa nazýva *shell* a hoci nie je časťou OS, umožňuje využívanie mnohých čŕt OS a slúži ako dobrý príklad, ako môžu byť použité systémové volania. Je to primárny interface medzi používateľom a operačným systémom.

Keď sa používateľ prihlási, shell sa naštartuje. Terminál má nastavený ako štandardný vstup a štandardný výstup. Po spustení vypíše ohlasovací *prompt*, čím oznamuje, že shell čaká na príkazy. Keď používateľ napíše príkaz, shell *vytvorí proces*, ktorý spustí program zodpovedajúci príkazu. Počas jeho behu shell *čaká* na ukončenie. Po skončení znova vypíše prompt. Používateľ môže *presmerovať štandardný výstup* (napr. `date >subor`) alebo štandardný vstup (napr. `sort<subor1 >subor2`). Výstup jedného programu môže byť vstupom druhého (napr. rúra). Ak za príkazom používateľ použije &, shell nečaká na dokončenie príkazu, ale hneď vypíše prompt. Existujú dva hlavné spôsoby na *implementovanie* interpretra príkazov:

- **samotný interpreter obsahuje kód na vykonávanie príkazov**: napr. príkaz na zmazanie súboru — interpreter príkazov skočí na úsek kódu, ktorý nastavuje parametre a vykoná príslušné systémové volanie. V tomto prípade počet príkazov určuje veľkosť interpretra príkazov, lebo každý príkaz vyžaduje vlastný kód.
- **každý príkaz je implementovaný špeciálnym programom**: Interpreter príkazov použije príkaz na identifikovanie súboru, ktorý má byť nahratý do pamäte a vykonaný (tak je to napr. v Unixe). V tomto prípade sa ľahko pridajú do systému nové príkazy, a to vytvorením nových súborov s príslušným menom. Interpreter príkazov je vcelku malý a netreba ho pri pridávaní príkazov meniť. Problémom pri tvorbe interpretra príkazov je, že OS musí poskytovať mechanizmus odovzdávania parametrov z interpretra príkazov systémovým programom.

3.3 Štruktúra OS

Pozrieme sa na OS „zvnútra“. Ukážeme si 4 rôzne štruktúry, ktoré sa skúmali, aby sme získali poznatky o celom spektre možností:

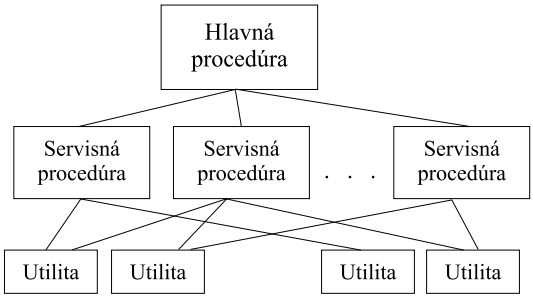
Monolitické systémy

Išlo o na pohľad najvšeobecnejšiu (najjednoduchšiu) organizáciu. Neexistuje tu žiadna štruktúra, celý OS je napísaný ako súhrn procedúr, z ktorých každá môže volať iné, kedykoľvek ich potrebuje. Každá procedúra v systéme má definované rozhranie, čo sa týka parametrov a výsledkov. Na vytvorenie objektového programu — operačného systému — treba skompilovať všetky individuálne procedúry a spojiť ich do jedného objektu linkerom. Každá procedúra je viditeľná pre každú inú.

Aj v monolitickom systéme je možné mať aspoň malú štruktúru, keď služby (systémové volania) sú žiadané uložením parametrov na dohodnuté miesto, ako sú registre alebo zásobník a vykonaním

špeciálnej inštrukcie prerušenia — *kernel call* (supervisor call). Táto inštrukcia prepne počítač z *user mode* do *kernel mode* a odovzdá riadenie operačnému systému. OS zistí parametre volania na určenie, ktoré systémové volanie sa má vyvolať, a na základe toho identifikuje servisnú procedúru, ktorá bude zavolaná (smerník na ňu je v príslušnej položke nejakej tabuľky). Na záver je systémové volanie ukončené a riadenie vrátené užívateľskému programu. Táto organizácia predpokladá základnú štruktúru OS:

1. **hlavný program**, ktorý vyvolá požadovanú servisnú procedúru
2. **množinu servisných procedúr**, ktoré vybavujú systémové volania
3. **množinu utilít**, ktoré pomáhajú servisným procedúram.



Vrstvové systémy

Zovšeobecnením prístupu na predošlom obrázku je organizovať OS ako *hierarchiu vrstiev*. Prvý takto skonštruovaný systém bol THE (vytvorený v Technische Hogeschool Eindhoven, Holandsko — Dijkstra so študentami 1968). Išlo o jednoduchý batch systém pre počítače Elektrologica X8. Systém mal 6 úrovní (vrstiev):

0. alokácia procesora, prepínanie medzi procesmi pri prerušení alebo vypršaní času. Poskytovala bázu pre základné multiprogramovanie CPU (nad úrovňou 0 systém pozostával zo sekvenčných procesov, z ktorých každý mohol byť programovaný bez toho, aby sa vedelo, že na jednom procesore beží viac procesov).
1. správa pamäte a 512K slov bubna (na uchovávanie stránok, pre ktoré nie je miesto v hlavnej pamäti) — nad touto úrovňou sa procesy nemuseli starať o to, či sú v súvislej pamäti alebo na bubne. Sofware z úrovne 1 sa staral, aby potrebné časti boli v pamäti.
2. komunikácia medzi každým procesom a konzolou operátora (nad úrovňou 2 mal každý proces vlastnú operátorskú konzolu)
3. správa V/V-zariadení a buffrovanie toku informácie (nad touto vrstvou sa procesy zaoberali abstraktnými V/V-zariadeniami).
4. užívateľské programy (nemusia sa starať o procesy, pamäť, konzolu V/V)
5. proces systémového operátora

Ďalšie zovšeobecnenie vrstvovej koncepcie bolo v OS MULTICS — bol organizovaný do množiny sústredných kružníc, z ktorých vnútorná bola viac privilegovaná ako vonkajšia. Keď procedúra vo vonkajšej kružnici chcela volať procedúru vo vnútornej kružnici, musela urobiť príslušné systémové volanie (TRAP-inštrukciu, ktorej parametre boli pred vykonaním volania starostlivo preverené, či sú platné).

Virtuálne počítače

Prvé verzie OS/360 boli striktne batch systémy a na žiadosť používateľov začali viaceré skupiny (v IBM aj mimo) vyvíjať time-sharing systémy. Oficiálny IBM time-sharing systém TSS/360 bol uvedený neskoro a bol veľmi veľký a pomalý (bolo možné zapojiť len zopár terminálov). Pozastavený bol po tom, čo jeho vývoj stál 50 miliónov USD. Ale skupina v IBM Scientific Centre v Cambridge vyvinula systém, ktorý je teraz široko používaný. Pôvodne sa tento systém volal CP/CMS, neskôr VM/370. Bol založený

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.

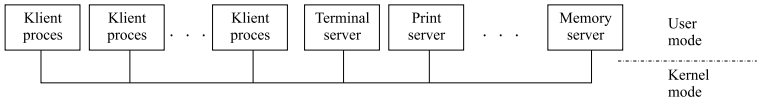
Jadro systému (*monitor viruálneho počítača*) beží na holom hardware a vykonáva multiprogramovanie, pričom poskytuje nie jeden, ale niekoľko virtuálnych počítačov na ďalšej úrovni. Avšak tieto virtuálne počítače nie sú rozšírené počítače (so súbormi a inými „peknými“ črtami), ale sú to presné kópie hardwaru, 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.

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

Kapitola 4

Procesy

Všetok spúšťateľný software na počítači je organizovaný do množstva *sekvenčných procesov* (alebo skrátene *procesov*). Proces je vlastne vykonávaný program vrátane aktuálnych hodnôt registrov, premenných a čítača inštrukcií. CPU sa prepína medzi procesmi, ale pre zjednodušenie si môžeme zo začiatku predstaviť množinu procesov bežiacich pseudoparalelne a až neskôr sa zaoberať tým, ako sa CPU medzi procesmi prepína. Toto rýchle prepínanie sa nazýva *multiprogramovanie*. Vykonávanie procesu musí prebiehať sekvenčným spôsobom (z toho názov „sekvenčný"), t.j. v každom momente sa vykonáva maximálne jedna inštrukcia na účet procesu. Tak, hoci môžu byť dva procesy spojené s tým istým programom, uvažujú sa dve nezávislé sekvencie výpočtu.

4.1 Hierarchia procesov

OS musí poskytovať nejaký spôsob na vytvorenie potrebných procesov. Vo veľmi jednoduchých systémoch alebo systémoch určených len na jednoduché aplikácie, je možné mať všetky procesy, ktoré budú potrebné, prítomné, keď systém nabieha. Avšak vo väčšine systémov je potrebný nejaký spôsob na vytváranie a rušenie procesov podľa potreby počas činnosti systému. V Unixe je proces vytvorený volaním systému `fork`, ktoré vytvorí identickú kópiu volajúceho procesu. Po volaní `fork` rodičovský proces pokračuje vo vykonávaní paralelne s potomkom. Proces môže vytvoriť viacero potomkov. Proces-potomok môže tiež vykonať `fork`, takže je možné mať celý strom procesov. Táto metóda vytvárania procesu sa často nazýva *spawning*. Proces ukončuje sám seba vykonaním volania `exit` alebo môže byť ukončený ako výsledok signálu `kill` od iného procesu. Ako časť bootu je vytvorený proces s identifikačným číslom 0 — *swapper*, ktorý je vždy priradený plánovaču procesov a CPU a ktorý riadi operácie plánovania procesov. Proces 0 vytvorí proces 1 — *init*. Všetky užívateľské procesy sú potomkami procesu init. Tento proces najprv načíta súbor, z ktorého zistí počet terminálov a pre každý terminál vytvorí jeden nový proces *getty*. Proces getty čaká, kým sa niekto neprihlási, t.j. čaká používateľove meno a heslo, ktoré sa stanú argumentami procesu *login*. Ten hľadá toto meno a heslo v súbore `/etc/passwd`. Ak ich nájde, login spustí shell — interpreter príkazov. Používateľ už ďalej komunikuje len so shellom.

V systéme MS-DOS existuje systémové volanie na nahratie určeného binárneho súboru do pamäte a jeho vykonania ako potomka. Na rozdiel od Unixu, v MS-DOSe toto volanie pozastaví rodičovský proces, až kým potomok neskončí, takže proces-rodič a potomok nebežia paralelne.

4.2 Stavy procesov

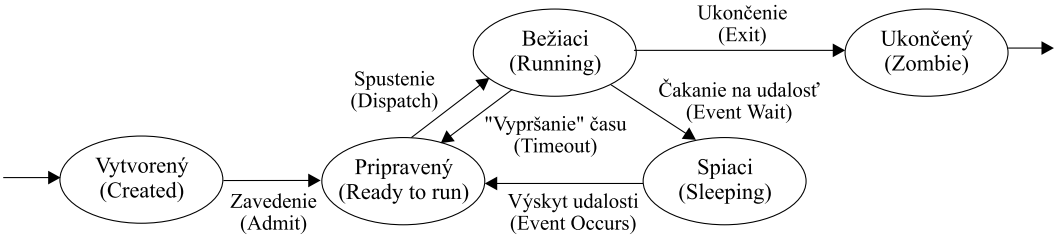
Počas svojej existencie sa proces môže nachádzať v rôznych stavoch. Zmenu stavu procesu spôsobujú rôzne udalosti alebo aktivity.

5-stavový model

Proces sa nachádza v jednom z piatich stavov:

- **Bežiaci (Running)**: práve vykonávaný proces. Ak uvažujeme jednoprocesorový systém, v tomto stave sa môže nachádzať v ľubovoľnom okamihu najviac jeden proces.
- **Pripravený (Ready)**: proces, ktorý je pripravený na vykonávanie hneď, ako sa mu to umožní.
- **Blokovaný alebo Spiaci (Blocked, Sleeping)**: proces, ktorý nemôže bežať kým nenastane nejaká udalosť, napr. ukončenie V/V operácie.
- **Nový alebo Vytvorený (New, Created)**: proces, ktorý bol práve vytvorený, ale ešte nebol zaradený operačným systémom medzi spúšťateľné procesy.
- **Ukončený (Exit, Zombie)**: proces, ktorý bol operačným systémom vyradený spomedzi spúšťateľných procesov, pretože bol z nejakého dôvodu ukončený alebo zrušený. V tomto štádiu už proces nie je možné spustiť, ale tabuľky a informácie spojené s procesom sú ešte k dispozícii napr. pre podporné programy (napr. „účtovací“ program môže potrebovať údaje o spotrebovanom čase procesora a ostatných použitých prostriedkov za účelom „vyúčtovania“).

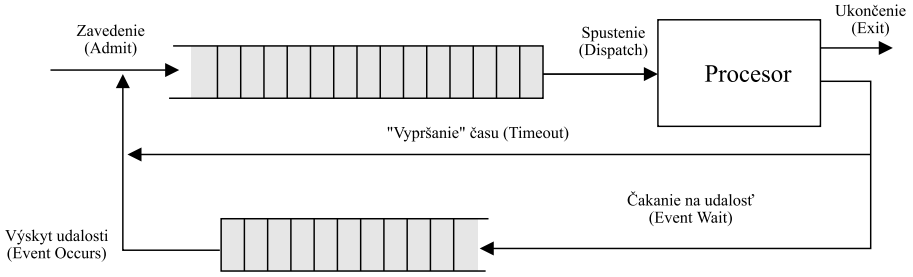
Obrázok ilustruje typy udalostí, ktoré vedú k zmenám stavov procesov.



Možné prechody medzi stavmi sú:

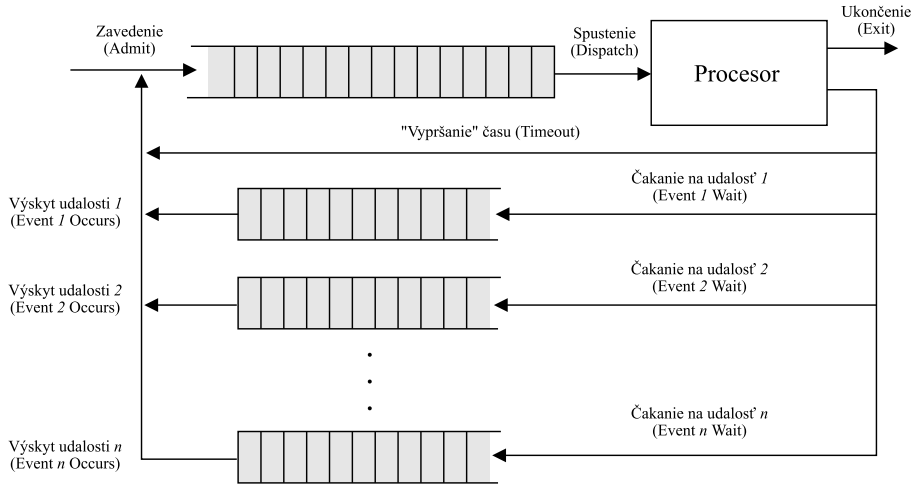
- **Null → Nový**: Je vytvorený nový proces na vykonanie programu.
- **Nový → Pripravený**: Operačný systém presunie proces zo stavu Nový do stavu Pripravený, keď môže prijať ďalší proces. Väčšina systémov má nastavenú nejakú hranicu založenú na počte existujúcich procesov alebo množstve virtuálnej pamäte pridelenej existujúcim procesom. Účelom použitia takejto hranice je zabrániť tomu, aby bolo aktívnych príliš mnoho procesov, že by mohli znížiť výkonnosť systému.
- **Pripravený → Bežiaci**: Keď sa vyberá ďalší proces na spracovanie, operačný systém volí jeden z procesov, ktoré sú v stave Pripravený.
- **Bežiaci → Ukončený**: Bežiaci proces je ukončený operačným systémom, ak bol dokončený alebo zrušený.
- **Bežiaci → Pripravený**: Väčšina multiprogramových operačných systémov prideľuje procesom istý čas na vykonávanie a dôvodom na prechod zo stavu Bežiaci do stavu Pripravený potom je vyčerpanie prideleného času. Môžu byť však aj iné dôvody na pozastavenie bežiaceho procesu – v závislosti od stratégie plánovania procesov na spustenie – napr. ak sa procesy na spracovanie vyberajú podľa priority, dôvodom pozastavenia procesu je, že sa stane pripraveným proces s vyššou prioritou.
- **Bežiaci → Blokovaný**: Proces prechádza do stavu Blokovaný, ak požiadal o niečo, na čo musí čakať (V/V, správa od iného procesu, dokončenie iného procesu a pod.)
- **Blokovaný → Pripravený**: Keď nastala udalosť, na ktorú proces čakal, prechádza zo stavu Blokovaný do Pripravený.
- **Pripravený → Ukončený**: Tento prechod nie je v diagrame vyznačený. V niektorých systémoch môže rodičovský proces ukončiť potomka kedykoľvek. Alebo keď skončí rodičovský proces, môže byť ukončený aj potomok.
- **Blokovaný → Ukončený**: detto ako v predošlom bode.

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.

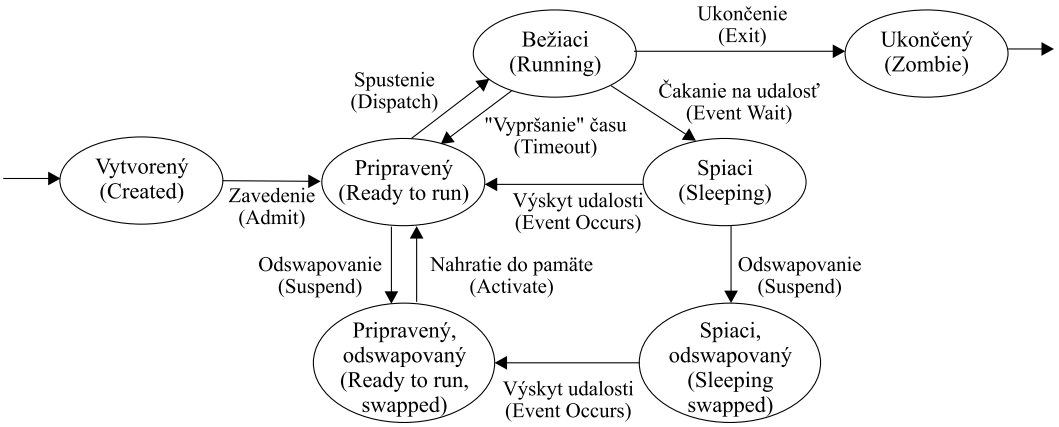
Do modelu stavov procesov musí pribudnúť nový stav – *Odswapovaný (Swapped, Suspended)*. Keď sú všetky procesy v hlavnej pamäti blokované, operačný systém môže niektorý proces previesť do stavu Odswapovaný a presunúť ho na disk.

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

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.

4.3 Popis procesu

Operačný systém riadi procesy a spravuje pre ne systémové prostriedky. Preto musí mať k dispozícii informácie o aktuálnom stave každého procesu a jeho prostriedkoch.

Operačný systém si vytvára a udržuje tabuľky informácií o každej entite, ktorú spravuje:

- **Pamäťové tabuľky**: udržiavajú informáciu o pamäti – o jej pridelení procesom, o ochrane a pod.
- **V/V tabuľky**: sa používajú na správu V/V zariadení. V každom okamihu môže byť V/V zariadenie voľné alebo pridelené nejakému procesu. Ak sa vykonáva V/V operácia, operačný systém musí vedieť o stave operácie, o mieste v hlavnej pamäti, ktoré sa používa ako zdroj alebo cieľ V/V prenosu.
- **Tabuľky súborov**: poskytujú informáciu o súboroch, ich umiestnení na disku, ich aktuálnom stave a iných atribútoch.
- **Tabuľky procesov**: obsahujú informácie potrebné pre správu procesov: kde sú procesy umiestnené a atribúty procesov.

Umiestnenie procesov

Proces pozostáva z vykonávaného programu, množiny dátových miest pre lokákle a globálne premenné a konštanty, zásobníka a množstva atribútov potrebných pre riadenie procesu operačným systémom – množina týchto atribútov sa nazýva *riadiaci blok procesu (process control block)*. Súhrn programu, dát, zásobníka a atribútov sa nazýva *„obraz procesu“ (process image)*. Jeho umiestnenie závisí od použitého typu správy pamäte. V najjednoduchšom prípade je udržiavaný ako súvislý blok pamäte, umiestnený na disku. Aby mohol byť proces spustený, „obraz procesu“ musí byť nahratý do hlavnej pamäte. V moderných operačných systémoch „obraz procesu“ pozostáva z množiny blokov, ktoré nemusia byť uložené súvisle (za sebou). Do hlavnej pamäte je možné zaviesť len niektoré časti procesu, kým ostatné časti ostanú na disku (virtuálna pamäť).

Tabuľka procesov musí obsahovať informáciu o umiestnení „obrazov procesov“.

Atribúty procesu

Operačný systém musí udržiavať množstvo informáci o každom procese. Tieto informácie sú zapísané v riadiacom bloku procesu. Rôzne systémy si organizujú informácie v riadiacich blokoch procesov rôzne. Môžeme však nájsť typické kategórie informácií požadovaných operačným systémom pre každý proces:

- **Identifikácia procesu**: identifikačné číslo procesu (PID), PID rodiča procesu, identifikačné číslo (UID) vlastníka procesu
- **Informácia o stave procesora**: obsah registrov – všeobecných, riadiach, stavových registrov (PSW), stack pointer
- **Informácia o riadení procesu**: prídavná informácia potrebná pre operačný systém pre riadenie a koordinovanie rôznych aktívnych procesov. Sem patrí *informácia o plánovaní a stave procesu* (stav, priorita procesu, identifikácia udalosti, na ktorú proces čaká, informácie pre plánovanie procesov), *informácie o stave V/V* (pridelené V/V prostriedky, zoznam otvorených súborov), *informácie pre „administratívu“* (čas CPU, limity na čas), *informácie o využívanej pamäti* (smerník na tabuľku stránok alebo segmentov), *informácie o komunikácii medzi procesmi* („flagy“, signály, správy používané pre komunikáciu procesov).

Kapitola 5

Synchronizácia a komunikácia procesov

5.1 Synchronizácia procesov

Významným pojmom v OS je *súbežnosť (concurrency)* procesov. Tento pojem zahŕňa množstvo problémov, ako sú komunikácia medzi procesmi, zdieľanie a súťaženie o prostriedky, synchronizácia aktivít procesov a prideľovanie procesorového času procesom.

Existencia súbežnosti vedie k nasledujúcim požiadavkám pri návrhu operačného systému:

- Operačný systém musí byť schopný spravovať rôzne aktívne procesy.
- Operačný systém musí prideľovať a uvoľňovať rôzne prostriedky (procesorový čas, pamäť, súbory, V/V zariadenia) každému aktívnemu procesu.
- Operačný systém musí chrániť dáta a fyzické prostriedky každého procesu pred neúmyselným zásahom od iného procesu.
- Výsledky procesu musia byť nezávislé od rýchlosti vykonávania relatívne k rýchlosti ostatných súbežných procesov.

Interakcia procesov

Spôsoby interakcie procesov môžeme klasifikovať podľa stupňa uvedomenia si existencie ostatných procesov:

- **Procesy si neuvedomujú ostatné procesy.** Sú to nezávislé procesy, ktoré nezamýšľajú pracovať spoločne. Môže ale medzi nimi dochádzať k **súťaženiu** o prostriedky (napr. pristupujú k tomu istému disku, súboru, tlačiarni), ktoré musí riešiť operačný systém.
- **Procesy si nepriamo uvedomujú iné procesy.** Tieto procesy nemusia nutne poznať iné procesy podľa mena, ale napríklad zdieľajú prístup k niektorým objektom (napr. V/V buffer). U takýchto procesov sa prejavuje **kooperácia pri zdieľaní** spoločných objektov.
- **Procesy si priamo uvedomujú iné procesy.** Sú to procesy schopné komunikovať navzájom podľa mena a sú vytvorené, aby spolu vykonávali nejakú činnosť. Dochádza ku **kooperácii** procesov.

Ako príklad problému so súbežnosťou si uveďme *print spooler*: Keď chce proces tlačiť súbor, uloží jeho meno do špeciálneho adresára – *spool directory*. Ďalší proces — *printer daemon* — pravidelne kontroluje tento adresár a keď je tam súbor na vytlačenie, vytlačí ho a vymaže jeho meno z adresára.

Predstavme si, že adresár má veľký (potenciálne nekonečný) počet položiek, očíslovaných $0, 1, 2, \ldots$, pričom v každej môže byť uložené jedno meno súboru. Predstavme si, že existujú dve zdieľané premenné out — ukazuje na ďalší tlačený súbor a in — ukazuje na ďalšiu voľnú položku v adresári.

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.

```
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.

```
#define FALSE 0
#define TRUE 1
#define N 2                                          /* počet procesov */
int turn;
int interested[N];                                   /* všetky hodnoty sú na začiatku 0 */

void enter_region(int process)                       /* číslo procesu: 0 alebo 1 */
{ int other;                                         /* číslo ďalšieho procesu */
  other = 1 − process;
  interested[process] = TRUE;                        /* ukázať, že proces má záujem */
  turn = process;                                    /* nastaviť flag */
  while (turn == process && interested[other] == TRUE);   /* nič */
}

void leave_region(int process)
{ interested[process] = FALSE;                       /* odchod z kritického úseku */
}
```

Obr. 5.3: Petersonovo riešenie

Petersonovo riešenie

Kombináciou myšlienky prepínania s myšlienkou uzamykacích premenných a „varovacích" premenných našiel holandský matematik Dekker riešenie problému vzájomného vylúčenia, ktoré nevyžaduje striktné striedanie. Jeho riešenie je však dosť komplikované, takže sa v praxi nepoužívalo.

V roku 1981 našiel Peterson jednoduchší spôsob na riešenie problému vzájomného vylúčenia (pozri obrázok 5.3). Pred vstupom do kritického úseku proces volá funkciu enter_region() so svojím číslom. Toto volanie spôsobí čakanie, pokiaľ nebude vstup bezpečný.

Predpokladajme, že oba procesy volajú funkciu enter_region() takmer súčasne. Oba ukladajú svoje číslo do premennej turn. Ten, čo ho uloží neskôr, bude na príkaze while čakať.

Všetky uvedené softwarové riešenia vyžadovali činné čakanie. To nielen míňa čas CPU, ale môže mať neželané efekty. Napr., nech sú v počítači 2 procesy: H s vyššou prioritou, L s nižšou prioritou. Proces s vyššou prioritou sa stáva bežiaci hneď, keď je v stave pripravený. V istom momente, keď L je v kritickom úseku a H je v stave pripravený, začína činné čakanie, ale keďže L nemôže získať CPU, kým H beží, L nedostane nikdy šancu opustiť kritický úsek a H čaká do nekonečna. Tento problém sa nazýva *priority inversion problem*.

Riešenia s podporou operačného systému alebo programovacieho jazyka

Uvedieme si prostriedky, ktoré spôsobia zablokovanie, namiesto míňania času CPU, keď nie je možné vojsť do kritického úseku. Jeden z najjednoduchších je pár systémových volaní sleep, wakeup.

Sleep a WakeUp

Sleep spôsobí, že volajúci proces bude zablokovaný, kým ho iný proces „nezobudí". Parametrom volania wakeup je proces, ktorý má byť zobudený. Sleep a wakeup nie sú súčasťou štandardnej $\mathcal{C}$-knižnice, ale pravdepodobne sú prístupné v ľubovoľnom systéme, ktorý má tieto systémové volania.

Ako príklad si uvedieme *problém producenta a konzumenta* (tiež známy ako Problém ohraničeného buffera): Dva procesy zdieľajú ohraničenú pamäť (buffer). Prvý je producent a vkladá doň informácie. Druhý je konzument a vyberá ich. Problém nastane, keď je buffer plný a producent nemôže vkladať, alebo keď konzument nemá čo vyberať.

```
#define N 100                                        /* veľkosť buffra */
int count = 0;

void producer()
{ while (TRUE) {
      produce_item()                                 /* generuj položku */
      if (count == N) sleep();                       /* buffer plný */
      enter_item();                                  /* uložiť do buffra */
      count = count + 1;                             /* zvýš počet položiek v buffri */
      if (count == 1) wakeup(consumer);              /* bol buffer prázdny? */
      }
}

void consumer()
{ while (TRUE) {
      if (count == 0) sleep();                       /* buffer prázdny */
      remove_item();                                 /* vybrať z buffra */
      count = count - 1;
      if (count == N - 1) wakeup(producer);          /* bol buffer plný? */
      consume_item();                                /* spracuj položku */
      }
}
```

Obr. 5.4: Sleep a WakeUp

Označme veľkosť buffra N a počet položiek v buffri count. Producent musí testovať, či count $= N$, konzument, či count $= 0$ (viď. obr. 5.4)

Vráťme sa k „race conditions“: Môže sa stať, že buffer je prázdny a konzument číta premennú count, aby ju testoval. Vtedy sa prepne procesor pre producenta, ktorý vloží do buffra položku a zvýši count na 1. Potom producent volá wakeup(consumer). Ale konzument ešte neuskutočnil volanie sleep, preto sa signál wakeup stratí. Keď zase beží konzument, testuje count, ktoré má nastavené na 0 a zavolá sleep. Časom potom producent zaplní buffer a tiež zavolá sleep, takže oba procesy budú spiace.

Problém teraz spočíval v tom, že sa stratil signál wakeup. Riešením je pridať *wakeup waiting bit*. Keď je poslaný wakeup signál nespiacemu procesu, nastaví sa tento bit. Keď potom proces volá sleep, len sa vynuluje tento bit a proces zostáva nespiaci.

Avšak je možné nájsť príklady s tromi alebo viacerými procesmi, kde ani wakeup waiting bit nie je postačujúci. Môžeme potom pridať ďalší takýto bit (alebo aj viac).

Semafóry

V roku 1965 Dijkstra navrhol používanie celočíselných premenných na počítanie počtu wakeup-ov na budúce použitie. Bol tak zavedený nový typ premennej — *semafór* (nadobúdajúci celočíselné hodnoty väčšie alebo rovné 0).

So semafórmi je možné vykonať dve operácie:

- P (down): Najprv sa vykoná test, či je hodnota kladná. Ak áno, zníži sa o 1 (čím sa minie jeden rezervný wakeup) a pokračuje sa. Ak nie je kladná (t.j. jej hodnota je nulová), volá sa systémové volanie sleep.
- V (up): Najprv zvýši hodnotu semafóru o 1. Ak boli na tomto semafóre nejaké spiace procesy, jeden z nich je systémom vybratý a môže dokončiť operáciu down (po dokončení operácie up na semafóre so spiacim procesom teda bude jeho hodnota opäť 0, ale budeme mať o 1 spiaci proces menej).

```
#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).

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

```
monitor ProducerConsumer;
var full, empty: condition;
    count: integer;

procedure enter;
begin
   if count = N then wait(full);
   enter_item;
   count := count + 1;
   if count = 1 then signal(empty);
end;

procedure remove;
begin
   if count = 0 wait(empty);
   remove_item;
   count := count - 1;
   if count = N - 1 then signal(full);
end;

count := 0;
end monitor;

procedure producer;
begin
   while true do
      begin
         produce_item;
         ProducerConsumer.enter;
      end;
end;

procedure consumer;
begin
   while true do
      begin
         ProducerConsumer.remove;
         consume_item
      end;
end;
```

Obr. 5.6: Monitory

  - Dĺžka: fixná, variabilná

- *Spôsob zaraďovania do fronty*
  - FIFO
  - Priorita

Synchronizácia

Komunikácia medzi procesmi vyžaduje istý stupeň synchronizácie týchto procesov: prijímajúci proces nemôže prijať správu, kým nebola iným procesom poslaná. Musíme tiež špecifikovať, čo sa stane po vykonaní operácie send alebo receive: v oboch prípadoch môže byť proces blokovaný (v prípade operácie send kým nebude správa prijatá, pri receive kým nejaká správa nepríjde) alebo neblokovaný. Bežne sa používajú tri kombinácie, hoci každý systém má zvyčajne implementovanú len jednu alebo dve z týchto kombinácií:

- *Blokovaný send a blokovaný receive*: nazýva sa tiež *rendezvous*
- *Neblokovaný send a blokovaný receive*: pravdepodobne je to najužitočnejšia kombinácia.
- *Neblokovaný send a neblokovaný receive*

Pri multiprogramovaní sa často používa neblokovaný send. Napr. pri požiadavke na vykonanie výstupnej operácie ako napr. tlačenie žiadajúci proces vyšle požiadavku vo forme správy a pokračuje. Pre receive je prirodzenejšia blokovaná verzia. Zvyčajne proces očakávajúci správu potrebuje informáciu z tejto správy, aby mohol pokračovať. Avšak, ak sa správa stratila alebo vysielajúci proces zlyhá ešte pred vyslaním správy, prijímajúci proces bude zablokovaný navždy. Možný je tiež prístup, že pred vykonaním receive proces testuje, či čaká nejaká správa.

Adresovanie

Sú dva spôsoby adresovania:

- **Priame adresovanie**

  Operácia send explicitne pomenuje adresáta. Operácia receive môže byť riešená dvoma spôsobmi:

  - prijímajúci proces explicitne pomenuje odosielateľa správy – hovoríme o *symetrickej komunikácii.*

    Formát operácií send a receive je:

    * send($P, \&s$)
    * receive($Q, \&s$)
  - prijímajúci proces používa implicitnú adresáciu, čiže parameter 'zdroj' v receive bude naplnený identifikačným číslom posielajúceho procesu, keď sa operácia receive vykoná – hovoríme o *nesymetrickej komunikácii.*

    Formát operácií send a receive je:

    * send($P, \&s$)
    * receive($id, \&s$), kde id sa naplní číslom procesu, ktorý správu vyslal.

- **Nepriame adresovanie**

  Správy sa posielajú a prijímajú zo „schránok“ (mailbox). Každá schránka má jednoznačnú identifikáciu. Dva procesy môžu komunikovať, len ak zdieľajú schránku — miesto na buffrovanie istého počtu správ (tento počet je zvyčajne určený pri vytvorení schránky).

  - send($A, \&s$)
  - receive($A, \&s$)

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

```
#include "prototypes.h"
#define N 100                                   /* počet položiek v buffri */
#define MSIZE 4                                 /* veľkosť 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

bytoch, prijímateľ prečíta 1000 bytov, t.j. dostane všetkých 10 správ naraz. Problém nevzniká, ak sa procesy dohodnú, že zapisujú a čítajú správy fixnej veľkosti alebo ukončia správu špeciálnym znakom (napr. *LF*).

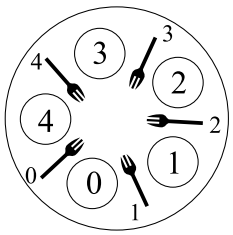
Kapitola 6

Klasické problémy koordinácie procesov

6.1 Problém obedujúcich filozofov

V r. 1965 ho nastolil a vyriešil Dijkstra — na modelovanie procesov, ktoré sa snažia o výlučný prístup k obmedzenému množstvu prostriedkov:

Päť filozofov sedí okolo okrúhleho stola, každý má svoj tanier špagiet, na ich zjedenie potrebuje filozof 2 vidličky. Medzi každými dvoma taniermi je vidlička.



Život filozofa pozostáva z fázy myslenia a fázy jedenia (ostatné aktivity sú tu irelevantné). Keď je filozof hladný, pokúsi sa zobrať ľavú a pravú vidličku (nie naraz) v ľubovoľnom poradí. Ak sa mu to podarí, naje sa, potom položí vidličky a opäť myslí - viď. algoritmy na obr.6.1 a 6.2.

Avšak ak predpokladáme, že naraz všetkých 5 filozofov uchopí ľavú vidličku, tak žiaden nemôže uchopiť pravú vidličku a nastane *uviaznutie*.

```
#define N 5

void filozof(int i)
{ while (TRUE) {
      mysli();
      vezmi_vidlicku(i);                    /* vezmi ľavú vidličku */
      vezmi_vidlicku((i+1)%N);              /* vezmi pravú vidličku */
      jedz();
      poloz_vidlicku(i);                    /* polož ľavú vidličku */
      poloz_vidlicku((i+1)%N);              /* polož pravú vidličku */
  }
}
```

Obr. 6.1: Algoritmus pre filozofa (obvyklé riešenie)

```
#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.

```
#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

```
typedef int semphore;
semaphore mutex = 1;                          /* riadi prístup do rc */
semaphore db = 1;                             /* riadi prístup do databázy */
int rc = 1;                                   /* počet procesov, ktoré čítajú alebo chcú čítať */

void reader()
{ while (TRUE) {
      down(&mutex);                           /* výlučný prístup k rc*/
      rc = rc + 1;                            /* o 1 čitateľa viac */
      if (rc == 1) down(&db);                 /* ak je 1. čitateľ*/
      up(&mutex);                             /* ukonči výlučný prístup k rc */
      read_data_base();                       /* prístup k dátam */
      down(&mutex);
      rc = rc - 1;                            /* o 1 čitateľa menej */
      if (rc == 0) up(&db);                   /* ak to bol posledný čitateľ */
      up(&mutex);
      use_data_read();                        /* nekritická sekcia */
  }
}

void writer()
{ while (TRUE) {
      think_up_data();                        /* nekritická sekcia */
      down(&db);                              /* výlučný prístup do databázy */
      write_data_base();                      /* zmeň dáta */
      up(&db);                                /* ukonči výlučný prístup */
  }
}
```

Obr. 6.4: Riešenie problému čitateľov a zapisovateľov (Courtois 1971) — semafóry

```
var buffer: shared record pool: array[0..n − 1] of item;
                          count, in, out: integer;
            end;

Producent: vkladá novú položku nextp do buffra vykonaním:
region buffer when count < n do
            begin
               pool[in]:= nextp;
               in:= in + 1 mod n;
               count:= count + 1;
            end;

Konzument: vyberá položku z buffra a ukladá ju do nextc:
region buffer when count > 0 do
            begin
               nextc:= pool[in];
               out:= out + 1 mod n;
               count:= count − 1;
            end;
```

Obr. 6.5: Problém obmedzeného buffera (producent/konzument)

Ďalší prostriedok vyššej úrovne na koordináciu procesov je jazykový konštrukt *kritický región* (zavedený Brinch Hansenom a Hoareom, 1972). Umožňuje kontrolu synchronizácie už pri kompilácii: Premenná v typu T zdieľaná viacerými procesmi bude deklarovaná:

var v: shared T;

Premenná v môže byť dostupná len vnútri inštrukcie region nasledujúceho tvaru:

region v **do** S;

čo znamená, že pokiaľ je vykonávaná inštrukcia S, žiaden iný proces nemôže pristupovať k premennej v. Prekladač každej zdieľanej premennej implicitne pridelí binárny semafór a kritický región chrániaci prístup k tejto premennej, uzavrie príkazmi down a up nad týmto semafórom, t.j.

region v **do** S;

je ekvivalentné

down(P);
S;
up(P);

Kritický región sa dá efektívne použiť na riešenie problému kritického úseku, ale nedá sa použiť na riešenie niektorých všeobecných problémov synchronizácie. Na toto zaviedol Hoare (1972) *podmienený kritický región*:

region v **when** B **do** S;

kde B je logický výraz. Keď proces vstúpi do regiónu kritického úseku, vyhodnotí sa výraz B: ak je pravdivý, vykoná sa S. Ak je nepravdivý, proces „uvoľní“ vzájomné vylúčenie a čaká, kým sa B stane pravdivým a neexistuje iný proces v kritickom úseku spojenom s v. Ako príklad na použitie podmieneného kritického regiónu môžeme uviesť problém producenta a konzumenta (obr.6.5).

Za niektorých okolností treba umiestniť synchronizačné podmienky na ľubovoľnom mieste vnútri kritického regiónu (nielen na začiatku). Brinch Hansen navrhol nasledovnú konštrukciu regiónu:

```
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

Kapitola 7

Uviaznutie

Uviaznutie je situácia, keď dva alebo viac procesov čaká na splnenie podmienky, ktorá nikdy nenastane. Skoro každá inštrukcia, v ktorej je procesom povolený výlučný prístup k zariadeniu, súborom a iným objektom je potenciálny zdroj pre *deadlock* (*uviaznutie*).

Množina procesov je v stave *uviaznutia*, keď každý proces z množiny čaká na udalosť, ktorú môže vyvolať len iný proces tejto množiny.

Podmienky uviaznutia:

1. vzájomné vylúčenie – mutual exclusion (aspoň jeden prostriedok môže byť v istom čase využívaný len jedným procesom)
2. postupné získavanie prostriedkov s čakaním – hold and wait (existuje proces, ktorý získava prostriedky postupne, v ľubovoľnom poradí a čaká, kým mu správa prostriedkov nepridelí žiadaný prostriedok, pritom neuvoľní získané prostriedky)
3. nemožnosť prerozdelenia prostriedkov – no preemption (ak proces získa prostriedok, žiaden iný proces nemá právo mu ho odobrať, uvoľniť prostriedok môže len ten proces, ktorý ho získal)

   Pri platnosti týchto 3 podmienok môže, ale nemusí vzniknúť uviaznutie. Aby uviaznutie skutočne nastalo, musí byť splnená štvrtá podmienka:
4. cyklické čakanie – circular wait (existuje množina procesov, v ktorej 1. proces čaká na udalosť generovanú 2. procesom,... n-tý proces na udalosť generovanú 1. procesom)

Prvé 3 podmienky sú nutné, ale nie postačujúce pre vznik uviaznutia. Štvrtá podmienka je vlastne potenciálnym dôsledkom prvých troch podmienok. Čiže za predpokladu, že platia prvé 3 podmienky, môže nastať postupnosť udalostí, ktorá vedie k vzniku *neodstrániteľného* cyklického čakania. Cyklické čakanie v podmienke 4 je neodstrániteľné, lebo platia prvé 3 podmienky. Čiže uvedené 4 podmienky spolu tvoria nutné a postačujúce podmienky pre vznik uviaznutia.

Vo všeobecnosti existujú 4 stratégie na zaobchádzanie s uviaznutím:

- ignorovanie problému (the Ostrich algorithm)
- detekcia a vyvedenie z uviaznutia (deadlock detection and recovery)
- prevencia — neumožnením jednej zo 4 podmienok (deadlock prevention)
- dynamické vyhýbanie sa — starostlivým prideľovaním prostriedkov (deadlock avoidance)

7.1 Ignorovanie („pštrosí prístup“)

Rôzny prístup k tejto metóde:

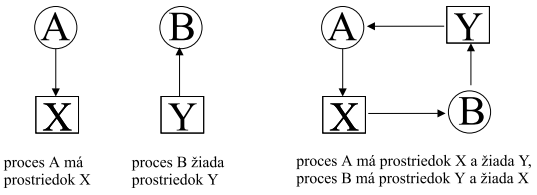
- matematici to považujú za úplne neakceptovateľný prístup
- inžinieri sa pýtajú, ako často takýto problém nastáva a ako vážne uviaznutie je

V Unixe vlastne každá tabuľka v OS (tabuľka procesov, tabuľka i-uzlov — určuje počet otvorených súborov, atď.) reprezentuje konečný (obmedzený) prostriedok a je potenciálnym zdrojom uviaznutia. Unixovský prístup je ignorovať problém uviaznutia s predpokladom, že väčšina užívateľov by preferovala možné uviaznutie pred pravidlami obmedzujúcimi každého užívateľa na používanie 1 procesu, 1 otvoreného súboru atď.

7.2 Detekcia a vyvedenie

Systém sleduje požiadavky na prostriedky a ich uvoľnenie. Periodicky vykonáva algoritmus, ktorý umožňuje zistiť, či nastala podmienka cyklického čakania. Kontrola na vznik uviaznutia sa môže uskutočniť pri každej požiadavke na prostriedok (výhody: skorá detekcia uviaznutia, algoritmus je relatívne jednoduchý, lebo je založený na báze inkrementálnych zmien stavu systému, nevýhoda: frekventovaná kontrola míňa veľa času procesora) alebo menej často, v závislosti od toho, ako pravdepodobný je vznik uviaznutia.

Jednou z možností, ako môže systém sledovať požiadavky na prostriedky a preverovať vznik uviaznutia, je udržiavať graf procesov a prostriedkov.



Na základe požiadaviek procesov na prostriedky a ich uvoľňovania modifikuje graf a sleduje, či sa v ňom nevyskytuje nejaký cyklus.

Iná možnosť je pravidelne preverovať, či existujú procesy, ktoré sú sústavne blokované viac ako istý čas (napr. 1 hod.) — takéto procesy sú potom ukončené.

Všeobecný algoritmus pre detekovanie uviaznutia:

Uvažujme systém s n procesmi a m rôznymi typmi prostriedkov. Definujme nasledujúce matice a vektory:

$$\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$ aktuálne pridelenie prostriedkov procesom

$$\mathrm{P} = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1m} \\ p_{21} & p_{22} & \cdots & p_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nm} \end{pmatrix}$$ požadované prostriedky

$$\mathrm{V} = (v_1, v_2, \cdots, v_m)$$ nepridelené prostriedky

Algoritmus postupuje tak, že označuje neuviaznuté procesy. Na začiatku sú všetky procesy neoznačené. Potom sa vykonávajú tieto kroky:

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.

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.

7.4 Vyhýbanie sa

Pripúšťa platnosť všetkých 4 podmienok, zamedzí sa však ich súčasná platnosť. Takáto metóda menej obmedzuje procesy aj správu prostriedkov než preventívne metódy, je však algoritmicky a aj z hľadiska potrebných dátových štruktúr zložitejšia.

Popíšeme si 2 prístupy k vyhýbaniu sa uviaznutiu:

- nespustiť proces, ak jeho požiadavky na prostriedky môžu viesť k uviaznutiu
- neprideliť procesu ďalšie prostriedky, ak toto pridelenie môže viesť k uviaznutiu

Odmietnutie spustenia procesu

Uvažujme systém s n procesmi a m rôznymi typmi prostriedkov. Definujme nasledujúce matice a vektory:

$$\mathrm{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$ aktuálne pridelenie prostriedkov procesom

$$\mathrm{MP} = \begin{pmatrix} mp_{11} & mp_{12} & \cdots & mp_{1m} \\ mp_{21} & mp_{22} & \cdots & mp_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ mp_{n1} & mp_{n2} & \cdots & mp_{nm} \end{pmatrix}$$ požadované prostriedky

$\mathrm{C} = (c_1, c_2, \cdots, c_m)$ celkové množstvo prostriedkov v systéme

$\mathrm{V} = (v_1, v_2, \cdots, v_m)$ nepridelené (voľné) prostriedky

Matica MP udáva maximálne požiadavky každého procesu na každý prostriedok (mp_{ij} je maximálne množstvo prostriedku j, ktoré bude požadovať proces P_i). Táto informácia musí byť procesmi uvedená vopred.

Platia nasledujúce vzťahy:

1. $c_j = v_j + \sum_{k=1}^{n} a_{kj}$, pre každé j.

   Všetky prostriedky sú buď voľné (nepridelené) alebo pridelené.
2. $mp_{kj} \leq c_j$ pre všetky k, j.

   Žiaden proces nemôže požadovať viac prostriedkov, než je celkovo v systéme.
3. $a_{kj} \leq mp_{kj}$ pre všetky k, j.

   Žiadnemu procesu nemôže byť pridelené viac prostriedkov ľubovoľného typu, než na začiatku požadoval.

Stratégiu predchádzania uviaznutiu, ktorá zabráni spusteniu nového procesu, ak jeho požiadavky na prostriedky môžu viesť k uviaznutiu, definujeme takto: Nový proces P_{n+1} sa spustí len ak platí

$$c_j \geq mp_{(n+1)j} + \sum_{k=1}^{n} mp_{kj} \quad \text{pre každé } j.$$

Čiže proces bude spustený len ak maximálne požiadavky na prostriedky súčasných procesov plus nového procesu nepresiahnu celkové množstvo prostriedkov v systéme. Táto stratégia nie je optimálna,

pretože predpokladá to najhoršie — teda že všetky procesy budú požadovať prostriedky v maximálne ohlasovanej výške naraz.

Odmietnutie pridelenia prostriedkov procesu

Stratégia odmietnutia pridelenia prostriedkov je známa ako *bankárov algoritmus* a bola prvýkrát zavedená u Dijkstru (1965):

Bankár (správca prostriedkov) má k dispozícii isté množstvo peňazí v rôznych menách (isté množstvo prostriedkov rôznych druhov). Do banky prichádzajú zákazníci (procesy), ktorí žiadajú o pôžičku (pridelenie prostriedkov). Zákazníci si v banke otvoria kredity oznámením maximálnych výšok pôžičiek v jednotlivých menách (procesy pred prvou žiadosťou o pridelenie prostriedkov oznámia, koľko ktorých prostriedkov budú nanajvýš požadovať). Potom si zákazníci môžu ľubovoľne až do výšky svojho kreditu požičiavať peniaze s tým, že každú pôžičku v konečnom čase splatia, ak bankár splní svoj záväzok a zákazníkovi v konečnom čase požičia až do výšky jeho kreditu. Ak by bankár požiadavku zákazníka neuspokojil, vyhlasuje úpadok (systém procesov uviazol). Aby sa bankár nedostal do úpadku, musí si stále udržiavať *bezpečný stav* zvyšku kapitálu, t.j. stav, ktorý mu umožní aspoň v jednom poradí uspokojiť požiadavky zákazníkov na pôžičky (je aspoň jedna postupnosť, v ktorej môžu byť všetky procesy dokončené). Pôžičku, ktorá by mu tento stav porušila, nerealizuje, a odsunie ju na neskôr, až sa mu zvýši kapitál. Po každej požiadavke na prostriedok musí správa prostriedkov rozhodnúť, či aj nový stav bude bezpečný. Ak áno, požiadavka sa uspokojí, v opačnom prípade splnenie odloží.

Príklad:

Bankár má celkovo 10 jednotiek danej meny. Uvažujme nasledovné situácie pri rozdelení pôžičiek 4 klientom:

	Max	Má
A	5	1
B	6	1
C	3	2
D	8	4

Voľné: 2
Bezpečný stav

	Max	Má
A	5	1
B	6	2
C	3	2
D	8	4

Voľné: 1
Nebezpečný stav

Ak v situácii, ktorá je naznačená v 1. tabuľke, bankár splní požiadavku klienta B na 1 jednotku danej meny, dostane sa do nebezpečného stavu (síce 1 jednotka, ktorá mu ostala, je dosť pre klienta C a ten po skončení vráti 3 jednotky uvedenej meny, ale tie 3 jednotky nie sú dosť ani pre jedného z klientov A,B,D, takže nie je zaručené, že bankár môže splniť ich požiadavky až do výšky ich kreditov a teda nie je zaručené, že títo klienti vrátia požičané prostriedky). Aj keď nebezpečný stav nemusí viesť k uviaznutiu, lebo je možné, že klienti nebudú potrebovať prostriedky v plnej výške svojho kreditu, bankár sa nemôže na toto spoliehať.

Popíšme si formálne algoritmus zisťovania, či je stav bezpečný. Majme matice A, MP a vektory C, V definované tak, ako bolo uvedené v predošlom bode — A – aktuálne pridelenie jednotlivých typov prostriedkov jednotlivým procesom, MP – maximálne požiadavky procesov na jednotlivé typy prostriedkov, C – celkové množstvo jednotlivých prostriedkov v systéme, V – množstvo momentálne voľných prostriedkov jednotlivého typu. Algoritmus pracuje takto:

1. Inicializuje pomocný vektor **W** rovný vektoru **V**.
2. Nájde také i, že všetky nevybavené požiadavky procesu P_i na prostriedky sú menšie alebo rovné ako množstvo voľných prostriedkov. Teda $mp_{ik} - a_{ik} \leq w_k$ pre každé k. Ak také i neexistuje, systém môže uviaznuť, pretože nie je zaručené pre žiadny proces, že skončí.

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.

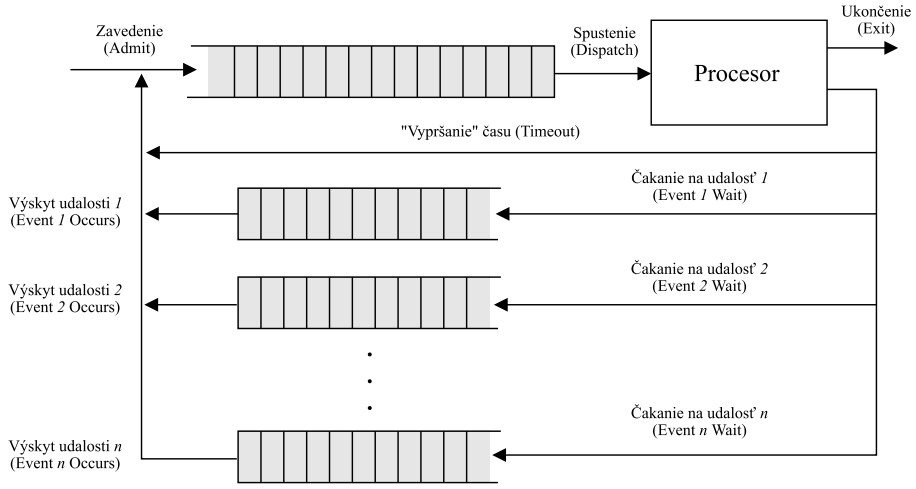
Kapitola 8

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.

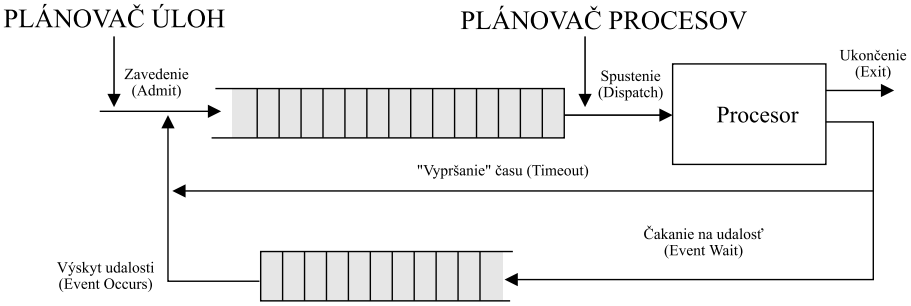
Proces vstupuje do systému zvonku a umiestni sa do zoznamu pripravených procesov. V ňom čaká, pokiaľ nie je vybratý na spracovanie. Keď musí čakať na V/V — zaradí sa do príslušného zoznamu prostriedku. Keď je obslúžený, opäť sa zaradí do zoznamu pripravených procesov. Proces pokračuje v tomto cykle CPU–V/V,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



Plánovač úloh — vyberá zo zadaných úloh nejakú podmnožinu a zavádza ich do systému na spracovanie (vytvára pre ne procesy + PCB a prideľuje procesom prostriedky).
Funkcie:

- Sleduje stav všetkých úloh (ktoré sú v stave „prijatá“ a aj tie, ktoré sa spracovávajú)
- Volí stratégiu, podľa ktorej úlohy vstupujú do systému („prijatá → pripravená“)
- Prideľuje úlohe vybratej na spracovanie potrebné prostriedky (pomocou ďalších modulov)
- Po dokončení úlohy prostriedky uvoľňuje

Plánovač procesov — rozhoduje, ktorým z podmnožiny procesov bude pridelený procesor
Funkcie:

- Sleduje stav procesov — túto funkciu realizuje tzv. dispečer (dispatcher) (sleduje stav procesov, prevádza zmeny stavov procesu, realizuje synchronizáciu a komunikáciu procesov)
- Rozhoduje, ktorému procesu bude pridelený procesor a na aký dlhý časový interval. Túto funkciu realizuje plánovač procesov.
- Prideľuje procesor — túto funkciu plní dispečer
- Uvoľňuje procesor — túto funkciu plní dispečer

Základný rozdiel medzi týmito plánovačmi je frekvencia ich používania:

- Plánovač procesov sa používa s veľkou frekvenciou (ms), teda musí byť veľmi rýchly, aby nevznikali veľké režijné straty. Preto sa nachádza trvale v operačnej pamäti (je súčasťou supervízora)
- Plánovač úloh sa vykonáva s omnoho menšou frekvenciou: vyvoláva sa, keď vstúpi nová úloha do systému, keď je nejaká úloha ukončená, príp. keď čas, počas ktorého procesor nepracuje (je "idle"), presiahne stanovenú hranicu. Plánovač úloh riadi stupeň multiprogramovania (počet procesov v pamäti). Čím viac procesov je vytvorených, tým menšie percento času môže byť každý proces vykonávaný (t.j. viac procesov "súťaží" o to isté množstvo času procesora).

Vo všeobecnosti väčšinu úloh možno klasifikovať ako orientované na V/V alebo orientované na CPU. Je dôležité, aby plánovač úloh vybral dobrý mix úloh medzi orientovanými na V/V a na CPU. Ak všetky úlohy často využívajú V/V, zoznam pripravených procesov by bol takmer prázdny a plánovač procesov by mal málo roboty. Ak všetky úlohy sú orientované na CPU, bude zoznam čakateľov na V/V takmer prázdny a systém bude znovu nevyvážený.

V niektorých systémoch môže byť plánovač úloh minimálny alebo vôbec neexistuje. Stabilita týchto systémov potom závisí buď od fyzických obmedzení (napr. počet možných terminálov) alebo od vlastnej umiernenosti užívateľov (ak služby začnú byť veľmi zlé, niektorí užívatelia sa jednoducho vzdajú a budú sa venovať inej veci).

Niektoré systémy, najmä tie, ktoré majú virtuálnu pamäť, pridávajú stredný stupeň plánovania — *plánovač strednej úrovne* (medium-term scheduler). Základná idea je, že v niektorých prípadoch môže byť výhodné eliminovať procesy v pamäti, a tak redukovať stupeň multiprogramovania. Neskôr proces môže znovu vstúpiť do pamäte a pokračovať od bodu, kde skončil. Toto sa často nazýva *swapping*. Plánovač strednej úrovne vymieňa proces — vyberá ho z pamäti a neskôr ho do nej vráti. Výmena môže byť nevyhnutná na zlepšenie mixu úloh.

8.2 Plánovacie algoritmy

Hlavným cieľom plánovania procesov je prideľovať čas procesora tak, aby sa optimalizovalo jedno alebo viac kritérií správania systému. Vo všeobecnosti je stanovená množina kritérií, podľa ktorých môžu byť jednotlivé plánovacie algoritmy ohodnotené.

Kritériá môžu byť rozdelené do 2 hlavných skupín: *užívateľsky orientované* (týkajú sa správania systému z pohľadu užívateľa alebo procesu) a *systémovo orientované* (dôraz je na efektívnom využití procesora).

Ďalej môžu byť kritériá klasifikované na *vzťahujúce sa na vykonávanie* (sú to kvantitatívne kritériá a zvyčajne môžu byť ľahko merateľné) a *nevzťahujúce sa na vykonávanie* (sú buď kvalitatívne alebo nie sú ľahko merateľné a analyzovateľné).

Uvedieme si niektoré kritériá na porovnanie plánovacích algoritmov.

Užívateľsky orientované, vzťahujúce sa na vykonávanie:

- doba odozvy (response time), t.j. čas od zadania požiadavky po obdržanie odpovede. Cieľom je dosiahnuť nízku dobu odozvy a maximalizovať počet interaktívnych užívateľov, ktorí dostanú akceptovateľnú dobu odozvy.
- doba prechodu procesu systémom (turnaround time), t.j. doba od začiatku zadania procesu až po ukončenie výstupu výsledku. Zahŕňa sa sem doba činnosti procesora aj doba čakania na prostriedky. Je to vhodná miera pre batch úlohy.
- termíny (deadlines): ak sú určené temíny na dokončenie procesov, plánovací algoritmus sa môže snažiť maximalizovať percento dodržaných termínov.

Užívateľsky orientované, ostatné:

- predpovedateľnosť (predictability): daná úloha môže bežať takmer rovnaký čas a za rovnakú "cenu" bez ohľadu na zaťaženie systému. Veľké odchýlky doby odozvy alebo doby prechodu sú pre užívateľov mätúce. Môže to signalizovať veľké kolísanie v pracovnej záťaži systému alebo potrebu optimalizácie systému, aby sa odstránila nestabilita. Kritérium predpovedateľnosti je do istej miery merateľné počítaním odchýliek ako funkcie pracovnej záťaže, ale toto nie je tak jednoduché ako meranie priepustnosti alebo doby odozvy.

Systémovo orientované, vzťahujúce sa na vykonávanie:

- priepustnosť (throughput), t.j. počet procesov spracovaných za jednotku času. Plánovací algoritmus sa snaží maximalizovať tento počet.
- využitie procesora (processor utilization) t.j. percento času, počas ktorého procesor pracuje. Toto využitie je dokonalé, ak v prestoji je procesor po dobu, ktorá nepresiahne 10% z celkovej doby činnosti, všeobecne by táto doba nemala presiahnuť 40%)

Systémovo orientované, ostatné:

- vyváženie prostriedkov (balancing resources): plánovacia stratégia môže udržiavať systémové prostriedky využité. Uprednostnené sú procesy, ktoré znížia zaťaženie preťažených prostriedkov. Toto kritérium zahŕňa aj plánovanie na vyššej a strednej úrovni.

Keď je kritérium porovnania vybraté, vo všeobecnosti je snaha optimalizovať ho. Je žiadúce maximalizovať využitie CPU a priepustnosť alebo minimalizovať čas prechodu alebo odozvy. Vo väčšine prípadov to, čo sa optimalizuje je priemer. V interaktívnych systémoch je dôležitejšie minimalizovať odchýlky (výkyvy) času odozvy ako minimalizovať priemerný čas odozvy.

V príkladoch na plánovacie algoritmy budeme pre jednoduchosť pre každý proces predpokladať, že len 1 raz využíva CPU (a žiadne V/V).

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.

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:

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

- Odstraňuje zo SJF veľké uprednostňovanie kratších úloh pred dlhšími.

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	15	20	14	2.80
5	8	2	13	15	7	3.50
Priemer					8.00	2.14

8.2.2 Preeemptívne (preemptive) plánovacie algoritmy

Bežiaci proces môže byť prerušený a uvedený do stavu "pripravený" operačným systémom. K pozastaveniu procesu môže dôjsť keď príde nový proces alebo periodicky na základe prerušenia od hodín.

FCFS, SJF a prioritné algoritmy tak, ako boli opísané, sú plánovacie algoritmy non-preemptive (bez pozastavenia). Keď je raz pridelený procesor procesu, môže ho proces použiť až pokiaľ si ho neželá uvoľniť (keď končí alebo spracováva V/V). Algoritmus SJF a prioritné algoritmy sa môžu modifikovať na algoritmy s pozastavením — preemptive.

Stratégia SJF s pozastavením — SRT

Keď vstúpi nová úloha, ktorá má kratšiu dobu použitia CPU ako vykonávaná úloha, algoritmus zruší priradenie CPU vykonávanej úlohe, zatiaľ čo SJF bez pozastavenia dovolí, aby vykonávaná úloha dokončila svoju prácu s CPU. SJF s pozastavením je známy ako *SRT* (*Shortest Remaining Time*).

Proces	Čas zadania	Čas spracovania (T_s)	Čas ukončenia	Doba prechodu (T_q)	$\frac{T_q}{T_s}$
1	0	3	3	3	1.00
2	2	6	15	13	2.17
3	4	4	8	4	1.00
4	6	5	20	14	2.80
5	8	2	10	2	1.00
Priemer				7.20	1.59

Prioritná stratégia s pozastavením

Keď vojde úloha do zoznamu pripravených, jej priorita sa porovná s prioritou vykonávaného procesu — ak je vyššia, algoritmus zruší priradenie CPU vykonávaného procesu, kým algoritmus bez pozastavenia len umiestni nový proces na čelo zoznamu.

Stratégia Round-Robin (RR) alebo cyklické plánovanie

- Vhodná pre systémy so zdieľaním času — dosahuje sa rozumná doba odozvy.
- Vytvára sa dojem funkcie viacerých samostatných procesorov, ktoré kontinuálne realizujú jednotlivé procesy, teda dochádza k virtualizácii procesora.

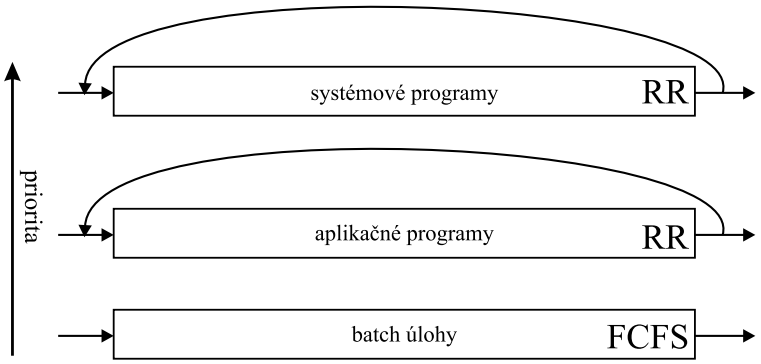
- Definuje sa malá jednotka času — časové kvantum (medzi 10–100 ms). Na implementáciu plánovania RR sa udržuje zoznam pripravených procesov ako FIFO zoznam, teda nové procesy sa zaradia na koniec a plánovač berie 1. proces zo zoznamu a prideľuje mu procesor. Ak do vypršania časového kvanta proces neuvoľní procesor, nastáva prerušenie OS, registre prerušeného procesu sa uložia v jeho PCB a proces sa zaradí na koniec zoznamu. Tomuto javu sa hovorí *process switch*, resp. *context switch*.
- Veľký vplyv na vonkajšie chovanie cyklicky plánovaného systému má veľkosť časového kvanta:
  - ak je veľmi veľké (nekonečno), je to FCFS
  - ak je veľmi malé (blíži sa k nule), tak virtualizáciou procesora sa dosiahne teoreticky ideálny procesor s $1/n$ rýchlosti originálneho procesora, kde n je počet plánovaných procesov. Tento ideálny stav možno dosiahnuť len za predpokladu nulových režijných strát pri odovzdávaní procesora medzi procesmi (pri zmene kontextu), inak sa context switch stane dominantným faktorom. Časové kvantum je preto zvyčajne 10–100 ms.

Možné obmeny algorimu Round-Robin

1. Zaraďovanie podľa *využitia* časového kvanta: ak proces využil celé pridelené kvantum, zaradí sa na koniec zoznamu pripravených procesov. Ak len na 1/2 (napr. čaká na V/V), je zaradený do stredu zoznamu a pod. Táto stratégia je obzvlášť vhodná pre úlohy s veľkými nárokmi na V/V.
2. Proces, ktorý nedočerpal časové kvantum, lebo uskutočnil V/V operáciu, sa po dokončení tejto operácie nezaradí na koniec zoznamu pripravených procesov, ale do pomocného zoznamu (auxiliary queue), ktorý má pri plánovaní procesov prednosť pred zoznamom pripravených procesov. Proces z pomocného zoznamu dostane procesor len na zostávajúci (nevyčerpaný) úsek posledne prideleného časového kvanta.
3. Algoritmus cyklickej obsluhy so *spätnou väzbou*: ak má byť zahájený nový proces, dostane najprv toľko časových kvánt, koľko už obdržala každá z ostatných úloh v systéme a potom pokračuje normálny Round-Robin.
4. *Limitovaný* algoritmus cyklickej obsluhy: úlohy prebiehajú podľa Round-Robin, pokiaľ nevyčerpajú vopred stanovený časový limit. Potom môžu prebiehať, len ak nie sú v systéme iné úlohy.
5. *Selfish Round-Robin* (*SRR*): kombinuje plánovanie na vyššej a nižšej úrovni do jednej rutiny. Pre úlohy, ktoré práve vstúpili do systému je vytvorený „delay queue“ (opozdovací rad), organizovaný ako FCFS. Tam čakajú, kým im nestúpne priorita a prejdú do aktívnej fronty RR.

Stratégia niekoľkých zoznamov (multiple queues alebo multilevel queues)

- Táto stratégia je vhodná pre situácie, kedy sú úlohy ľahko klasifikovateľné do rôznych skupín (napr. úlohy interaktívne a batch — majú rozdielne požiadavky na čas prechodu).
- Rozdeľuje zoznam pripravených procesov do oddelených zoznamov, každý má vlastný algoritmus plánovania.
- Musí však existovať plánovanie medzi zoznamami, to je prioritné plánovanie s pozastavením.



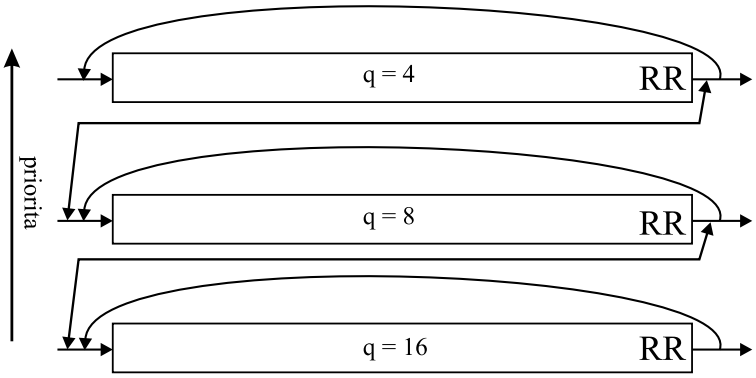
Žiadna úloha zoznamu Batch sa nemôže vykonávať pokiaľ nie sú ostatné zoznamy prázdne. Ak sa do zoznamu Akplikan programy zaradí úloha pokiaľ sa vykováva Batch, vykonávanej úlohe sa odoberie procesor. Iná možnosť je rozdelenie času medzi zoznamami: každý zoznam dostane jednu časť času CPU, ktorú môže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

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.

Kapitola 9

Správa pamäte — modely reálnej pamäte

Operačná pamäť je časť pamäte, ktorá slúži na uchovanie programov a dát, nad ktorými operuje procesor. Patrí medzi zdieľateľné prostriedky: o prístup do operačnej pamäte žiadajú procesy riadené užívateľskými programami, aj procesy riadené systémovými programami. Požiadavky na využitie operačnej pamäte vybavuje *správa* operačnej pamäte (ďalej len správa pamäte).

Funkcie správy pamäte:

1. udržiavanie prehľadu o použitých a nepoužitých miestach v operačnej pamäti
2. rozhodovanie o poradí obsluhovania požiadaviek na pridelenie priestoru v operačnej pamäti a o umiestnení takých priestorov
3. realizácia pridelenia (zápis do záznamov o procese, do tabuliek,...)
4. realizácia uvoľnenia pamäte

Skôr, než sa budeme zaoberať rôznymi typmi správy pamäte, uvedieme si požiadavky, ktoré musí správa pamäte podporovať:

- relokácia
- ochrana
- zdieľanie
- logická organizácia
- fyzická organizácia

Relokácia

V multiprogramovom systéme je pamäť zdieľaná viacerými procesmi. Preto zvyčajne nie je možné dopredu vedieť, kde bude proces v pamäti umiestnený. Tiež často dochádza k presúvaniu procesov v pamäti kvôli swapovaniu. Hardware procesora a operačný systém musia byť preto schopní transformovať odkazy na pamäťové miesta použité v programe na skutočné fyzické adresy v závislosti od aktuálneho umiestnenia procesu v operačnej pamäti.

Ochrana

Každý proces musí byť chránený pred zásahmi iných procesov – či už náhodnými alebo úmyselnými. Čiže do adresného priestoru procesu nesmie pristupovať iný proces bez povolenia.

Ochranu pamäte zabezpečuje procesor (hardware), pretože operačný systém nemôže predvídať všetky odkazy do pamäte, ktoré program urobí. Keby to aj bolo možné, bolo by príliš časovo náročné ochrániť každý program dopredu pred možnými narušeniami ochrany. Čiže preveriť platnosť odkazu do pamäte je možné až keď sa vykonáva inštrukcia.

Zdieľanie

Každý mechanizmus ochrany musí poskytovať možnosť, aby viaceré procesy pristupovali k tej istej časti operačnej pamäte. Správa pamäte teda musí umožňovať kontrolovaný prístup k zdieľaným oblastiam pamäte.

Logická organizácia

Hlavná pamäť počítača je organizovaná ako lineárny alebo jednodimenzionálny adresný priestor, pozostávajúci z postupnosti bytov alebo slov. Táto organizácia však nezodpovedá spôsobu, akým sú zvyčajne konštruované programy. Väčšina programov je organizovaná do modulov, z ktorých sú niektoré nemodifikovateľné (read only, execute only) a niektoré obsahujú dáta, ktoré možno modifikovať. Ak operačný systém a hardware umožňujú efektívne narábať s programami a dátami vo forme modulov, získa sa množstvo výhod:

- Moduly môžu byť písané a kompilované nezávisle, pričom referencie medzi modulmi budú vyriešené počas behu programu.
- Rôznym modulom môže byť priradený rôzny stupeň ochrany.
- Je možné zaviesť mechanizmy umožňujúce zdieľať moduly viacerými procesmi.

Prostriedok, ktorý najviac zodpovedá týmto požiadavkam, je segmentácia – je to jeden z typov správ pamäte, ktoré uvedieme v tejto kapitole.

Fyzická organizácia

Pamäť počítača je organizovaná minimálne do dvoch úrovní: hlavná pamäť (main memory) a prídavná pamäť (secondary memory) - disk. Hlavná pamäť poskytuje rýchly prístup pri relatívne vysokých nákladoch. Naviac, neposkytuje možnosť trvalého uloženia. Disk je pomalšia a lacnejšia pamäť a umožňuje trvalé uloženie údajov. Takže hlavná pamäť sa používa na uloženie programov a dát, ktoré sa práve používajú, kým disk slúži na dlhodobé ukladanie programov a dát.

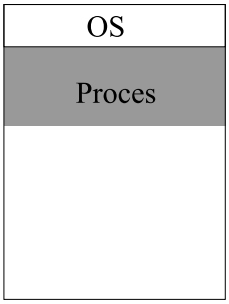
V tejto dvojúrovňovej schéme je hlavným predmetom záujmu organizácia toku informácií medzi hlavnou pamäťou a diskom. Toto je hlavná úloha správy pamäte.

9.1 Typy správy pamäte (historický prehľad)

9.1.1 Jeden súvislý úsek (monoprogramovanie)

Najjednoduchšia schéma správy pamäte je mať v pamäti jeden proces a poskytnúť mu celú pamäť. Fyzický adresný priestor (FAP) možno rozdeliť na množinu po sebe idúcich adries — v každej takejto množine je minimálna a maximálna adresa. Ak je procesu pridelená jediná takáto množina, nazveme ju *úsek*. Operačnému systému je pridelený samostatný úsek (napr. na začiatku operačnej pamäte).

Pri tejto organizácii pamäte môže byť v ľubovoľnom čase v pamäti len jeden proces. Keď užívateľ zadá príkaz, operačný systém nahrá požadovaný program do pamäte a vykoná ho. Po skončení programu vypíše operačný systém "čakací znak" (prompt character) na terminál a čaká na ďalší príkaz, aby nahral do pamäte ďalší proces, ktorý prepíše predošlý.



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

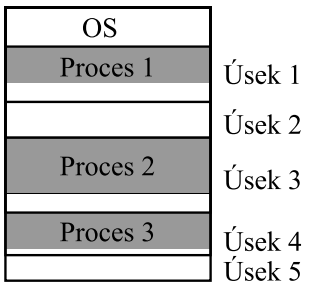
9.1.2 Statické súvislé úseky (Fixed partitions)

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.

Nevýhody: fragmentácia

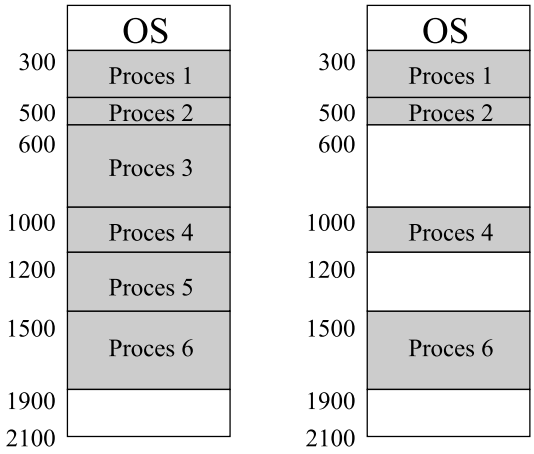
- *vnútorná* fragmentácia: ak proces potrebuje pre svoj beh pamäť s kapacitou K_1 a obdrží úsek s kapacitou K_2 ($K_2 > K_1$), tak $K_2 - K_1$ pamäťových miest je nevyužitých.
- *vonkajšia* fragmentácia: správa pamäte nemôže žiadnemu z pripravených procesov prideliť voľný úsek, lebo žiadny úsek nemá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

9.1.4 Buddy systém

Statické aj dynamické súvislé úseky majú isté nevýhody. Statické úseky limitujú počet aktívnych procesov a môžu využívať pamäť dosť neefektívne, ak je malá zhoda medzi veľkosťami voľných úsekov a veľkosťami procesov. Dynamické úseky majú zložitejšiu obsluhu a vyžadujú réžiu na kompaktáciu. Zaujímavým kompromisom je *buddy systém* (buddy - kamarát, druh).

V tomto systéme sa prideľuje pamäť v blokoch veľkosti 2^k. Na začiatku je celková kapacita pamäte (2^u) uvažovaná ako jeden voľný úsek. Keď proces požaduje pamäťový priestor veľkosti s, pričom platí $2^{u-1} \leq s \leq 2^u$, tak sa procesu pridelí celý úsek veľkosti 2^u. Inak sa úsek rozdelí na dve rovnaké časti veľkosti 2^{u-1}. Ak $2^{u-2} \leq s \leq 2^{u-1}$, tak sa procesu pridelí jedna z dvoch vzniknutých častí. Inak sa jedna z častí opäť rozdelí na dve. Tento proces pokračuje až pokiaľ sa nenájde najmenší úsek veľkosti 2^i, do ktorého sa proces zmestí.

Systém si udržuje zoznamy voľných úsekov podľa veľkosti. Úsek je vymazaný zo zoznamu $(i+1)$ ak sa rozdelí na dve časti veľkosti 2^i, ktoré sa zaradia do zoznamu i. Keď sa v zozname i objavia dva susedné voľné úseky veľkosti 2^i, tak sa spoja, vymažú sa zo zonamu i a zaradia ako voľný úsek veľkosti 2^{i+1} do zoznamu $(i+1)$.

Buddy systém sa používa v niektorých paralelných systémoch ako účinný prostriedok na alokáciu a uvoľňovanie pre paralelné programy.

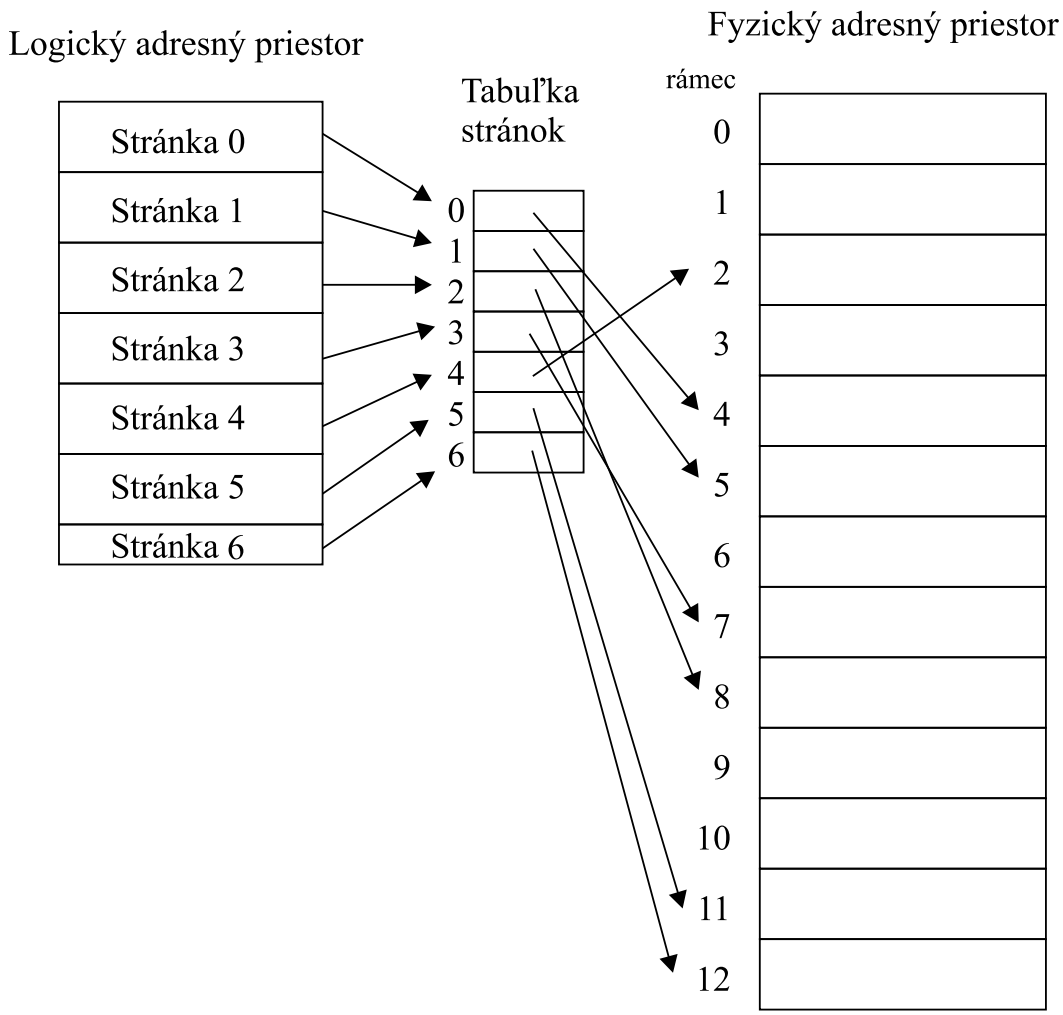
9.1.5 Stránkovanie

Existujú dve triedy algoritmov pre prideľovanie pamäte procesu:

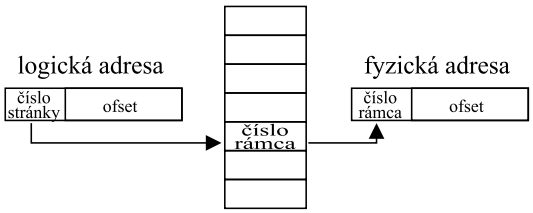
1. algoritmy pridelia procesu vo FAP jediný súvislý priestor — úsek (predošlé metódy)

2. algoritmy pridelia procesu vo FAP priestor, ktorý je z hľadiska postupnosti adries vo FAP súvislý po častiach: ich cieľom je minimalizovať vonkajšiu fragmentáciu.

K 2. skupine patrí aj *stránkovanie*: pamäť sa rozdelí na úseky pevnej dĺžky — *rámce* (dĺžka je vymedzená na úrovni hardware), LAP sa rozdelí na rovnako veľké úseky — *stránky*. Stránkovanie je potom prideľovanie rámcov pamäte stránkam. Kľúčovým problémom stránkovania je vyriešenie spôsobu transformácie adresy z LAP do FAP. Potrebné zobrazenie sa realizuje *tabuľkou stránok* (*page table*, PT). Zobrazenie LAP → FAP prevádza procesor pri interpretácii programu pri každom vstupe do pamäte. Adresa odkazujúca sa na miesto LAP sa rozkladá na dve zložky: vyššie rády adresy definujú stránku, nižšie rády adresu pamäťového miesta v stránke (ofset).



Adresa vo FAP sa rozkladá na číslo rámca a adresu v rámci. Adresa v stránke sa zhoduje s adresou v rámci. Pri zobrazení sa zamieňa číslo stránky číslom rámca, v ktorom je stránka umiestnená. Čísla stránok sa používajú pri prístupe do PT ako index, príslušná položka obsahuje číslo rámca pre túto stránku.



Dĺžka rámca sa volí ako mocnina 2 (IBM/360, ADT 4500: 2K slabík, IBM 370: 2048 alebo 4096B, DEC-10: 512 slov). Ak je to $k = 2^n$, tak dolných n bitov adresy udáva adresu (*offset*) v rámci, resp. stránke. Obsah PT je trvale zobrazený v zázname o procese (PCB), fyzická PT sa plní pri spúšťaní procesu. Z tohto hľadiska využíva stránkovanie vlastne formu dynamickej relokácie.

Správa procesov môže vytvoriť nový proces vtedy, ak jej dá správa pamäte pre jeho vytvorenie dostatočný priestor (rámce) v pamäti. K tomu si správa pamäti udržuje *tabuľku rámcov* (*frame table*, FT). Jednému rámcu zodpovedá jedna položka vo FT, ktorá obsahuje dve zložky: *stavovú* a *definičnú*. Stavová

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.

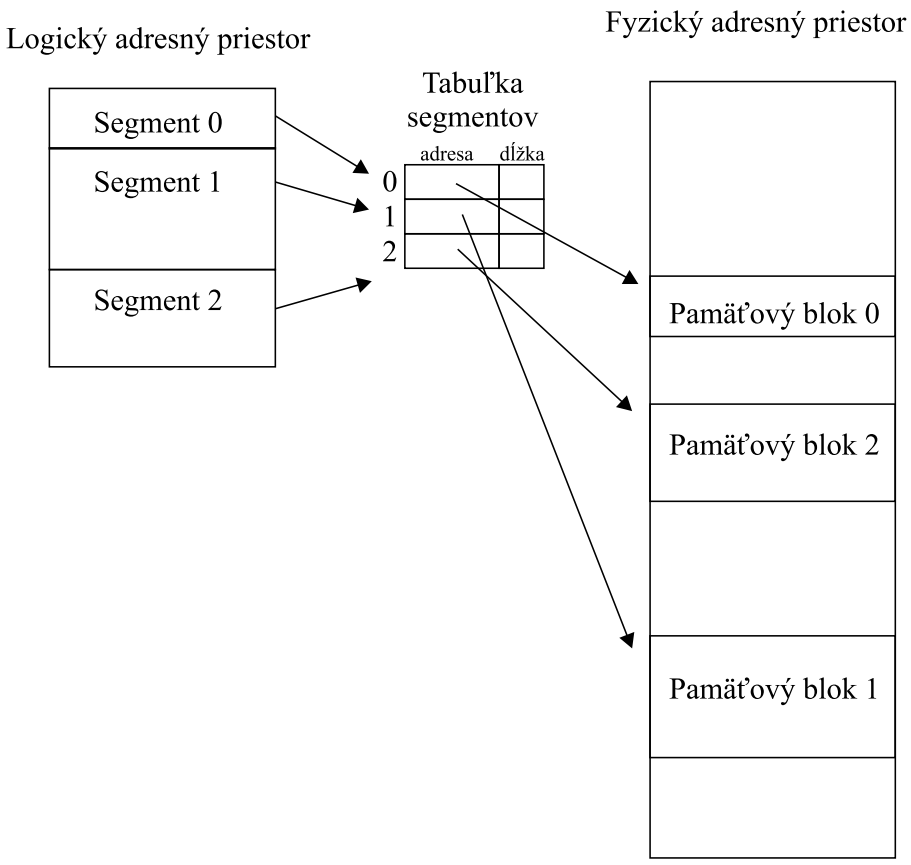
Zdieľateľné stránky

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 dielčí 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.



Pri transformácii logickej adresy na fyzickú sa číslo segmentu použije ako index do tabuľky 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. Tabuľka 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:

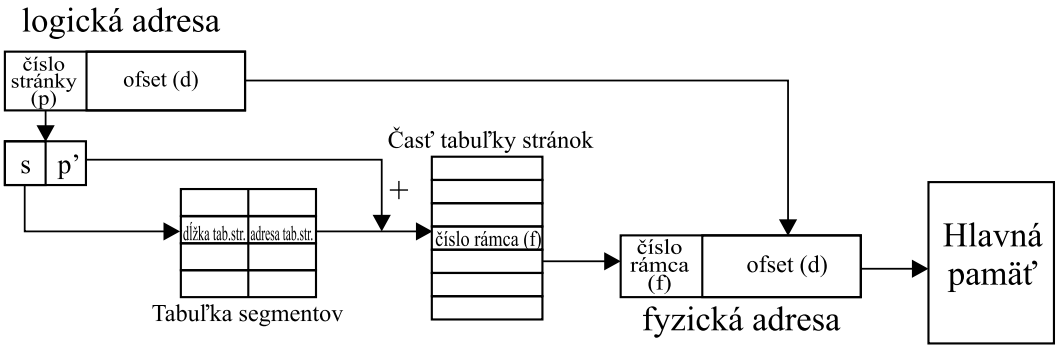
- segmented paging (PT je segmentovaná)
- paged segmentation (segmenty sú stránkované)

Segmentované stránkovanie (IBM 360/67)

24-bitová adresa:

- 12 bitov — číslo stránky: umožňuje 4096 stránok (8KB pre všetky PT, 1 riadok v PT = 2B)
- 12 bitov — offset

Zvyčajne bola väčšina PT prázdna, lebo väčšina programov používa len časť možného adresného priestoru, preto bola PT segmentovaná: horné 4 bity čísla stránky sa považovali za číslo segmentu (16 položiek pre segmenty v tabuľke segmentov), pričom z tabuľky segmentov bol smerník do PT pre tento segment (a tiež dĺžka PT). Týmto spôsobom mohli byť veľké časti tabuľky stránok, ktoré boli nulové „odstránené“ nastavením adresy tabuľky stránok na 0. V najhoršom prípade treba 3 prístupy do pamäti na prístup k adresovanému miestu.

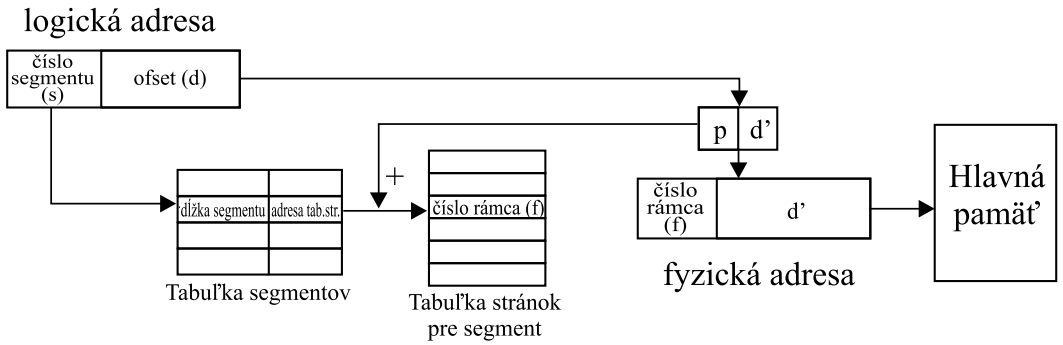


Stránkovaná segmentácia (Multics)

Logická adresa:

- 18 bitové číslo segmentu
- 16 bitový offset

Môže byť veľká vonkajšia fragmentácia alebo čas na hľadanie voľného pamäťového bloku (metódou first-fit alebo best-fit) Preto treba stránkovať segmenty (odstráni to vnútornú fragmentáciu).



Offset v segmente je rozdelený na 6-bitové číslo stránky a 10-bitový offset v stránke.

Kapitola 10

Správa pamäte — modely virtuálnej pamäte

Predošlé algoritmy vyžadovali, aby celý LAP bol v pamäti, čo často nie je nutné. Ak je LAP väčší než FAP, program sa nezmestí do pamäte. Zo začiatku bol tento problém riešený tak, že program sa rozdelil na časti, nazývané *overlays* (*prekryvné segmenty*). Prvý bol spustený overlay 0. Keď bol ukončený, volal ďalší. Niektoré systémy umožňovali umiestniť do pamäte naraz niekoľko segmentov. Segmenty boli uložené na disku a nahrávané do pamäte a z pamäte operačným systémom. Avšak rozdelenie programu na časti bolo úlohou programátora, čo zaberalo mnoho času. Vznikla snaha celú túto prácu zveriť počítaču. Metóda riešiaca tento problém začala byť známa ako *virtuálna pamäť*. Jej základná idea je: Veľkosť programu môže presahovať veľkosť fyzickej pamäte, ktorá je k dispozícii programu. OS drží v pamäti len časti programu, ktoré sa práve používajú, ostatné časti sú na disku.

Väčšina systémov s virtuálnou pamäťou používa metódu stránkovania. Adresy generované programom sa nazývajú *virtuálne adresy* a formujú *virtuálny adresový priestor*. Virtuálna adresa ide do *memory management unit* (MMU), ktorý mapuje virtuálnu adresu na fyzickú adresu. Virtuálny adresný priestor je rozdelený do stránok, operačná pamäť sa delí na rámce rovnakej veľkosti. Na transformáciu adresy sa používa tabuľka stránok. Tá určuje adresu rámcov zodpovedajúcich stránkam pamäte. Každá položka v tabuľke stránok obsahuje naviac *present/absent-bit*, ktorý indikuje, či sa príslušná stránka nachádza v operačnej pamäti. Ak sa nenachádza, generuje sa prerušenie, ktoré sa nazýva *výpadok stránky* (*page fault*), ktoré musí OS vyriešiť zavedením požadovanej stránky do operačnej pamäti a aktualizáciou zodpovedajúcej položky v PT. Ak v operačnej pamäti nie je voľný rámec na zavedenie stránky, treba vybrať *obeť* — stránku, ktorá sa skopíruje na disk a na jej miesto sa zapíše požadovaná stránka. Ak obeť bola od času, čo je v operačnej pamäti, modifikovaná, je nutné jej obsah zapísať na disk. Ak nebola modifikovaná, zápis na disk nie je nutný, nová stránka len prepíše jej obsah.

Stratený čas pri obhospodarovaní výpadku stránky závisí:

- od toho, s akou pravdepodobnosťou sa žiadajú stránky
- od dĺžky stránky (Ak chceme redukovať rozsah tabuľky stránok, použijeme väčšie stránky. Ak chceme redukovať vnútornú fragmentáciu, použijeme malé stránky.)
- od výberu obete

Optimálne je, keď o stránke vieme, ako dlho nebude použitá. Ako obeť potom zvolíme stránku, ktorá najdlhšie nebude potrebná. Tento prístup je však nerealizovateľný.

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

PAMÄŤ: 3 rámce	0	1	2	3	0	1	4	0	1	2	3	4
najmladšia stránka	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
najstaršia stránka			0	1	2	3	0	0	0	1	4	4
	vs	vs	vs	vs	vs	vs	vs			vs	vs	

9 výpadkov stránky

PAMÄŤ: 4 rámce	0	1	2	3	0	1	4	0	1	2	3	4
najmladšia stránka	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
najstaršia stránka				0	0	0	1	2	3	4	0	1
	vs	vs	vs	vs			vs	vs	vs	vs	vs	vs

10 výpadkov stránky

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.

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

pracovná množina (*working set*). Ak má proces počas danej fázy vykonávania v pamäti celú pracovnú množinu, pobeží viac-menej bez výpadkov stránok až kým sa nepresunie do inej fázy vykonávania.

V systémoch so zdieľaním času sa procesy často odsúvajú na disk. Vzniká otázka, čo robiť, keď proces načítame späť. Po technickej stránke to nie je žiadny problém: proces generuje výpadky stránok až kým nenačíta svoju pracovnú množinu. Problémom však je, že mať 20, 50 alebo 100 výpadkov stránok pri každom načítaní procesu do pamäti je veľmi pomalé a navyše dochádza k mrhaniu času CPU (spracovanie výpadku stránky trvá niekoľko ms). Preto sa niektoré systémy so zdieľaním času snažia sledovať pracovnú množinu každého procesu a zabezpečiť, aby bola načítaná vždy pred vykonaním procesu. Tomuto prístupu sa hovorí *model s pracovnou množinou* (*working set model*). Načítanie stránok pred spustením procesu sa tiež nazýva *prepaging*.

Na implementáciu tohto modelu treba nejako sledovať pracovnú množinu procesu. Jednou možnosťou je použiť algoritmus starnutia: každá stránka, ktorá má v horných N bitoch počítadla nejaký bit 1, sa zaradí do pracovnej množiny. Ak nejaká stránka nebola odkazovaná N tiknutí, tak je vyradená z pracovnej množiny. Parameter N sa musí určiť experimentálne pre každý systém, ale výkon systému zvyčajne nie je príliš citlivý na presnú hodnotu N (N sa nazýva aj *okno pracovnej množiny — working set window*).

10.3 Lokálne versus globálne prideľovacie stratégie

Pri nahradzovacích algoritmoch je tiež dôležitá otázka, či sa obeť vyberá spomedzi stránok pridelených procesu alebo spomedzi všetkých stránok v pamäti (lokálne vs. globálne nahradzovanie). Lokálne nahradzovanie zodpovedá prideleniu pevného miesta v pamäti procesu, globálne nahradzovanie dynamicky prideľuje rámce medzi spúšťateľné procesy. V prípade lokálneho nahradzovania treba určiť počet rámcov pre každý proces:

- rovnomerne rozdeliť
- pomerne (podľa veľkosti procesov)

Vo všeobecnosti globálne prideľovanie pracuje lepšie, hlavne v prípadoch, keď sa veľkosť pracovnej množiny počas života procesu mení.

10.4 Problémy pri implementácii

Pri implementácii virtuálnej pamäte sa musíme rozhodnúť pre niektorý z algoritmov na výber obete, lokálne alebo globálne nahradzovanie, resp. či implementovať stránkovanie na žiadosť alebo prepaging. Treba však brať do úvahy aj množstvo praktických otázok, z ktorých niektoré uvedieme:

Zálohovanie inštrukcií

Keď sa program pokúsi o odkaz do stránky, ktorá nie je v pamäti, tak je (strojová) inštrukcia prerušená niekde uprostred a nastane odskok do OS. Po načítaní stránky do pamäte je potrebné inštrukciu vykonať odznova. To niekedy nie je možné. Väčšina inštrukcií pozostáva z viacerých bajtov. Aby OS mohol vykonať inštrukciu znova, musí zistiť, kde sa začína. To možno zabezpečiť napr. tak, že existuje register, do ktorého sa skopíruje PC pred vykonaním inštrukcie, teda riešenie poskytujú tvorcovia CPU. Tiež musí existovať ďalší register, ktorý udržiava informácie o tom, či bol niektorý register autoinkrementovaný alebo autodekrementovaný (takto pracuje napr. PDP 11/45). Iné prístupy sú: Motorola 68010 — mikrokód uloží internú stavovú informáciu do zásobníka, VAX — mikrokód vráti stav počiatku do bodu, než sa začala inštrukcia, RISC — necháva počítač v stave, v akom bol (t.j. všetko musí vyriešiť OS).

Zamykanie stránok v pamäti

Uvažujme nasledovnú situáciu: proces zavolá systémovú procedúru na načítanie časti súboru do buffera vo svojom adresnom priestore. Kým čaká na dokončenie operácie, spustí sa iný proces, ktorý spôsobí

výpadok stránky. Existuje malá, ale nenulová pravdepodobnosť, že sa obetuje stránka so V/V bufferom. Ak V/V zariadenie práve presúva dáta do buffera, časť údajov sa zapíše do buffera a druhá časť prepíše novonačítanú stránku. Jedno riešenie je uzamknúť stránku, ktorá je používaná pre V/V, a tým zabrániť jej odstráneniu z pamäte. Druhé riešenie je, že kernel robí V/V do vlastných bufferov a z nich neskôr údaje skopíruje užívateľovi.

Zdieľanie stránok

Vo veľkých systémoch so zdieľaním času je bežné, že viacero používateľov súčasne spúšťa ten istý program (napr. editor). Bolo by výhodné, keby v pamäti bola len jedna kópia programu, ktorú by všetci zdieľali. Samozrejme, zdieľať možno iba tie stránky, do ktorých sa nesmie zapisovať (read-only), napr. text programu. Dáta alebo zásobník sa zdieľať nedajú. So zdieľanými stránkami môžu vzniknúť nasledovné problémy:

Nech procesy A a B vykonávajú ten istý program a zdieľajú jeho stránky. Ak sa proces A odloží na disk, jeho stránky sa nahradia iným programom, čo spôsobí, že proces B bude generovať množstvo výpadkov stránok, aby vrátil potrebné stránky do pamäte. Podobne, keď A skončí, OS musí vedieť zistiť, že jeho stránky sa naďalej používajú, a teda sa ešte nesmú uvoľniť. Na to potrebujeme nejaké dátové štruktúry, ktoré budú udržiavať informáciu o zdieľaných stránkach.

10.5 Virtualizácia pamäte segmentáciou na žiadosť

- Adresný priestor programu je rozdelený na segmenty — logické časti (procedúry, dáta, zásobník atď.).
- Keď nie je požadovaný segment v pamäti, generuje sa prerušenie nazývané *výpadok segmentu*, t.j. OS musí pre segment nájsť priestor v pamäti (zhusťovaním, odsunom iného segmentu) a zaviesť ho do pamäte.
- Nevýhodou tejto metódy je často zložité prideľovanie pamäti.

10.6 Správa pamäte v Unixe

Pred verziou 3BSD bola väčšina OS Unix založená na swapovaní, t.j. ak existuje viac procesov, ako je možné mať v pamäti, niektoré z nich sú odswapované na disk. Proces je odswapovaný celý (až na zdieľaný text), čiže proces je buď v pamäti alebo na disku.

10.6.1 Swapovanie

(Unix pre PDP-11, Interdata, začiatočné VAX implementácie)
Presun medzi pamäťou a diskom je riadený plánovačom — *swapperom*. Odswapovanie procesu z pamäte na disk sa uskutočňuje, keď nastalo preplnenie pamäte z dôvodu:

- fork potrebuje pamäť pre proces-potomok
- systémové volanie brk požaduje zväčšiť dátový segment
- stack narástol a presiahol vyhradený priestor

Navyše, ak treba z disku do pamäte presunúť proces, ktorý bol na disku príliš dlho, často treba odswapovať na disk iný proces.

Výber obete

- najprv blokované procesy — ak je takých viac, vyberie sa ten, ktorý má najvyšší súčet priority a času v pamäti
- ak nie je žiaden blokovaný proces, vyberá sa pripravený proces podľa spomenutého súčtového kritéria

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.

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.

Kapitola 11

Správa súborov

Všetky počítačové aplikácie potrebujú uchovávať a znovu vyberať informácie. Kým proces beží, isté obmedzené množstvo informácie môže uchovávať vo *vlastnom adresnom priestore*. Kapacita úložného priestoru je tým obmedzená na veľkosť virtuálnej pamäte , čo v niektorých aplikáciách nie je postačujúce (rezervácie leteniek, banky, ...). Druhým problémom je, že keď proces skončí, takto uložená informácia sa stratí. Tretí problém je, že často viaceré procesy pristupujú k tej istej informácii (alebo jej časti) v tom istom čase, takže nie je vhodné mať túto informáciu uloženú v adresnom priestore procesu. Spôsob, akým riešiť tieto problémy, je urobiť túto informáciu nezávislú od procesu: bude uložená na disku alebo inom externom zariadení v jednotkách nazývaných *súbory*.

Súbory sú spravované operačným systémom. To, ako sú štruktúrované, pomenované, ako sa k nim pristupuje, ako sa používajú, ako sú chránené a implementované — to sú hlavné otázky designu operačného systému. Časť operačného systému, ktorá sa zaoberá súbormi je známa ako *file system* (*systém súborov*).

Najprv sa budeme zaoberať súbormi z hľadiska používateľského, potom problémami implementácie.

11.1 Používateľské hľadisko

Súbory poskytujú možnosť ukladať informáciu na disk a znovu ju čítať. Je treba, aby detaily (ako a kde je informácia uložená a ako disky pracujú) boli pred používateľom skryté.

Keď proces vytvára súbor, pomenuje ho. Súbor existuje aj po skončení procesu a iné procesy k nemu môžu pristupovať prostredníctvom jeho *mena*. Mnohé operačné systémy umožňujú dvojdielne meno, v ktorom druhá časť (oddelená bodkou) sa nazýva *prípona (file extension)*. Určuje zvyčajne typ súboru. V Unixe je možné aj viac prípon, napr. prog.c.Z

11.1.1 Typy súborov

Väčšina operačných systémov má rozličné *typy súborov* — podľa druhu pamätanej informácie. Napr. Unix:

- Obyčajné (regular) súbory: obsahujú informáciu vloženú používateľom, aplikačné programy, systémové programy. Môžu obsahovať textovú alebo binárnu informáciu.
- Adresáre: obsahujú informáciu potrebnú na to, aby bolo možné dať súborom symbolické mená. Sú to vlastne obyčajné súbory so špeciálnymi zapisovacími ochrannými privilégiami: len systém môže do nich zapisovať, čítať môžu normálne používatelia.
- Špeciálne súbory: používajú sa na prístup k V/V zariadeniam. Sú dva typy týchto súborov: *blokové* a *znakové*. Každé V/V zariadenie má priradený špeciálny súbor. V/V operácie so špeciálnym súborom inicializujú prenos na zariadenie spojené s týmto súborom.
- FIFO súbory pre pipes: pseudosúbory, ktoré môžu byť otvorené dvomi procesmi na vytvorenie komunikačného kanálu medzi nimi.

Vo väčšine systémov sú obyčajné súbory ďalej delené do rozličných typov v závislosti od ich použitia. Rôzne skupiny sú potom rozlíšené rôznymi príponami, napr. .pas, .c, .asm, .dat, .exe, atď. V niektorých systémoch je to len nepísaná dohoda (Unix), v iných sa prípony berú do úvahy (napr. v MS-DOSe sa dajú spustiť len .com, .exe, .bat súbory. OS TOPS-20 ide tak ďaleko, že pri spúšťaní binárneho súboru sa skontroluje aj jeho zdrojový text. Ak bol modifikovaný po vytvorení binárneho súboru, tak sa automaticky prekompiluje.)

11.1.2 Atribúty súboru

Každý operačný systém pripája k súboru aj ďalšiu informáciu (okrem mena a dát), napr. dátum a čas vytvorenia, veľkosť súboru, t.j. *atribúty*. Atribútmi môžu byť: informácia o ochrane, vlastník súboru, čas posledneného prístupu, čas poslednej modifikácie, ...

11.1.3 Nezávislosť na zariadení

Všetky operačné systémy sa snažia o nezávislosť na zariadení, t.j. aby prístup k súborom bol rovnaký bez ohľadu na to, na ktorom zariadení sa súbor nachádza. Programy pracujúce so súbormi by mali byť schopné zapisovať súbory na disketu, na pevný disk, tlačiareň, terminál bez toho, aby samotný program musel byť pre rôzne prípady rôzne naprogramovaný. Napr. v Unixe je možné „namontovať" (mount) file system (napr. disk) hocikde v systéme súborov, čo umožňuje pristupovať k súborom cez meno (s cestou) bez toho, aby sme sa museli starať, na ktorom disku sa nachádza. Naopak, v MS-DOSe používateľ musí explicitne špecifikovať, na ktorom zariadení sa súbor nachádza (s výnimkou default zariadenia).

11.1.4 Štruktúra (organizácia) súboru

Tri hlavné spôsoby:

1. neštruktúrovaná postupnosť bytov: operačný systém nevie alebo sa nestará, čo je v súbore. Význam musí byť daný používateľskými programami. Používa sa napr. v Unixe, MS-DOSe.
2. postupnosť záznamov (*records*) pevnej dĺžky: Každý záznam má nejakú internú štruktúru. Operácia read číta jeden záznam, write zapisuje jeden záznam. Používa sa napr. v CP/M (128 znakové záznamy).
3. súbor pozostáva zo stromu záznamov (nie nutne pevnej dĺžky): Každý záznam obsahuje kľúč (na pevnej pozícii v zázname). Strom je utriedený podľa kľúča na urýchlenie vyhľadávania. Záznamy môžu byť spracovávané podľa kľúča, aj keď je možná aj operácia získania "ďalšieho záznamu". Pri vkladaní nového záznamu operačný systém rozhodne, kam bude záznam umiestnený. Táto organizácia sa používa napr. v systéme VAX/VMS a v mnohých veľkých sálových počítačoch. Metóda sa nazýva *Indexed Sequential Access Method* (ISAM).

11.1.5 Prístup k súboru

Rané operačné systémy poskytovali len jeden druh prístupu k súboru — *sekvenčný*. Je to prístup používaný pre magnetické pásky. Keď sa začali používať disky, pribudli možnosti čítať bajty alebo záznamy súboru mimo poradia a pristupovať k záznamom podľa kľúča. Prístup v ľubovoľnom poradí (*random access*) je základný prístup pre mnohé aplikácie, napr. databázové systémy.

Na špecifikovanie, kde začať čítať, sa používajú 2 metódy:

- pomocou *pozície*: zmenou aktuálnej pozície — potom nasledujúce operácie read a write operujú na novej pozícii
- pomocou čísla alebo kľúča záznamu

11.1.6 Operácie so súbormi

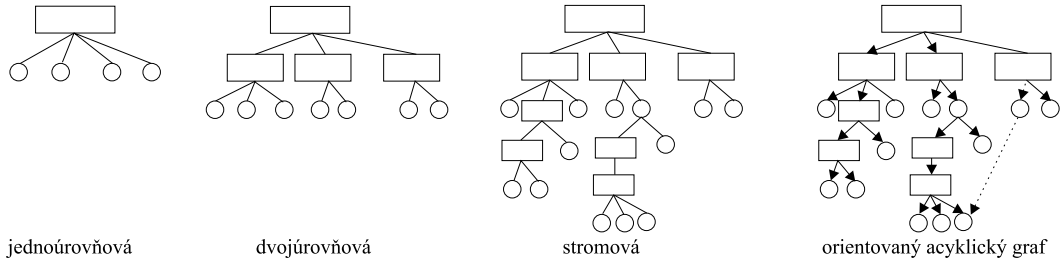
sú rôzne pre rôzne operačné systémy. Všeobecne: create, delete, open, close, read, write, append (pridať na koniec súboru), seek (pre náhodný prístup: zmena ukazovateľa pozície) get attributes, set attributes, rename.

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.

- *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:

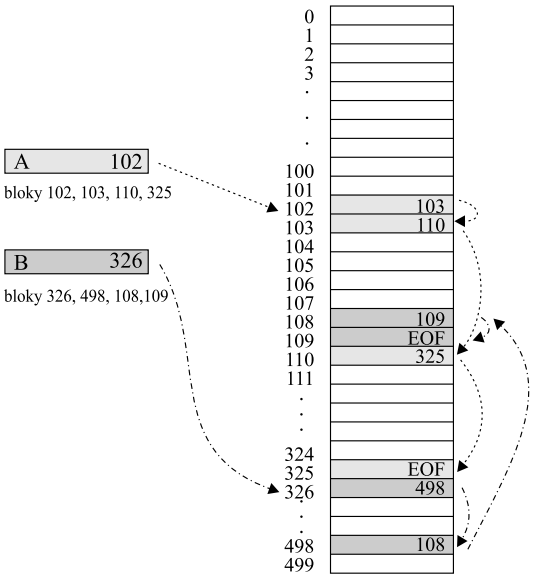
- treba vopred poznať maximálnu veľkosť súboru
- fragmentácia disku (kompaktácia je zvyčajne veľmi „drahá“)

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).

11.3.3 Spájaný zoznam s indexom

Obe nevýhody predošlej metódy sú tu eliminované: smerník bude uložený v špeciálnej tabuľke v pamäti (nie v bloku dát), napr. FAT v MS-DOSe.



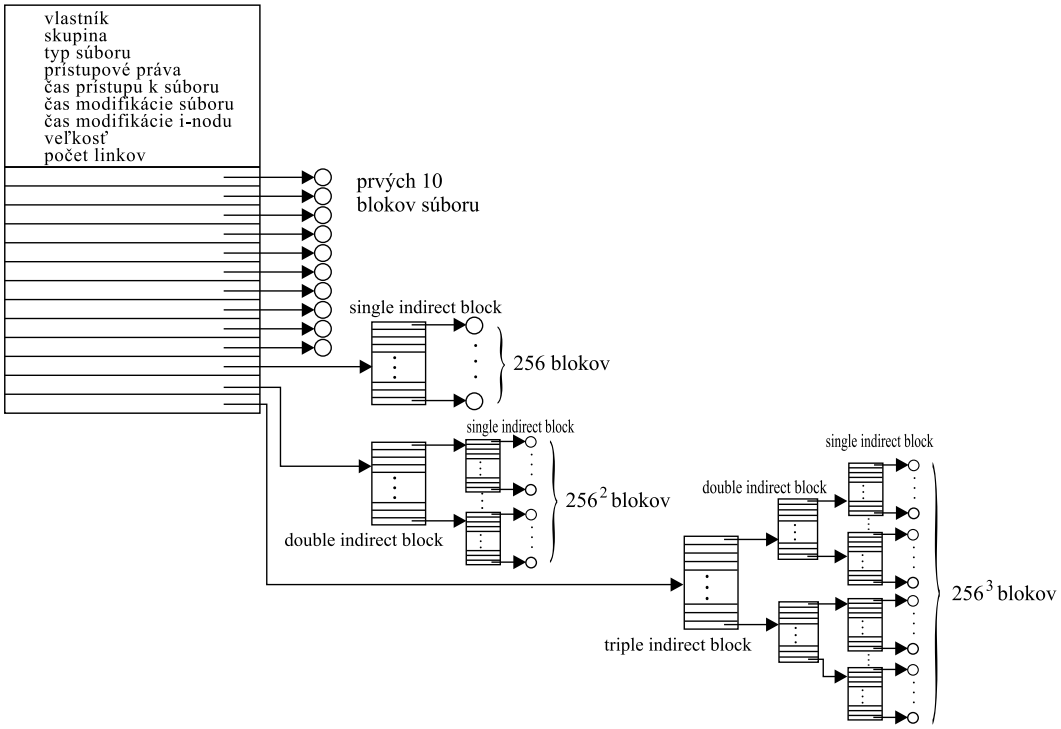
Je tiež ľahší náhodný prístup: netreba pri prechádzaní „reťaze blokov“ prístupy na disk, lebo tabuľka je v pamäti. V adresári sa udržuje len číslo prvého bloku súboru. Toto číslo slúži ako smerník do tabuľky, kde v príslušnej položke nájdeme číslo ďalšieho bloku súboru. To opäť určuje ďalšiu položku tabuľky s číslom ďalšieho bloku súboru, atď.

Nevýhoda:

- Tabuľka musí byť celá v pamäti. Ak je disk veľký, je aj tabuľka príliš veľká. (Napr., ak má disk 500000 1K blokov, t.j. $\approx$ 500M, tak má 500000 položiek minimálne 3-bajtových (na rýchlejšie vyhľadávanie sú vhodnejšie 4 bajty). Tabuľka teda zaberá 1.5–2M.) Ak pritom tabuľka nie je celá v pamäti, ale len jej časť, bude opäť náhodný prístup drahý (kvôli čítaniu častí tabuľky do pamäti).

11.3.4 *i*-node

Podstatou problému so smerníkmi vo FAT je, že sú náhodne porozhadzované v jednej tabuľke. Potenciálne teda treba celú tabuľku FAT, aj keď je otvorený len jeden súbor. Lepšia metóda je udržiavať smerníky pre jeden súbor spolu. Tak je to napr. v *i*-node (index node) v Unixe. *i*-node obsahuje aj ďalšie informácie o súbore:



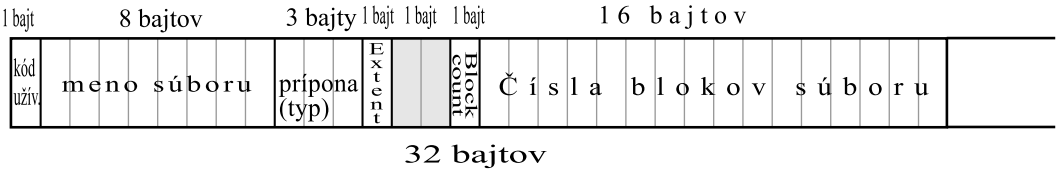
Koncept nepriamych blokových adries bol využitý na zabezpečenie rýchleho prístupu pre veľké aj malé súbory. Súbory, ktoré majú dĺžku do 10 blokov, majú všetky diskové adresy uchované priamo v *i*-node, čím je ľahké ich nájsť. Ak je súbor väčší, tak vezmeme voľný blok a doň uložíme adresy dátových blokov. Ak uvažujeme veľkosť bloku 1K a adresy 32-bitové, tak jednoduchý nepriamy blok môže uchovávať 256 diskových adries. Toto teda stačí na $10 + 256 = 266$ blokov. Pre súbory nad 266 blokov použijeme dvojitý nepriamy smerník, čím možno adresovať $266 + 256^2 = 65802$ blokov. Na ešte väčšie súbory použijeme trojnásobné smerníkovanie, takže celkovo môže mať súbor maximálne 16 gigabajtov. Pri zväčšní diskového bloku na 2K bude každý smerníkový blok obsahovať 512 pointrov, takže maximálna veľkosť súboru narastie až na 128G.

Silou tejto metódy je, že nepriame smerníky sa použijú až vtedy, keď je to naozaj potrebné. Tiež je zaujímavé, že aj pre súbor maximálnej dĺžky potrebujeme nanajvýš 3 pomocné prístupy na disk, aby sme mohli urobiť posun na ľubovoľný byte v súbore. Neberieme pritom do úvahy načítanie *i*-node, ktoré sa urobí pri otvorení súboru a potom sa udržiava v pamäti, až kým súbor nie je zasa zatvorený.

11.4 Implementácia adresárov

Skôr ako môžeme pracovať so súborom, musíme ho otvoriť. Pri otváraní súboru operačný systém použije názov súboru a pomocou neho určí bloky, ktoré súbor tvoria. Mapovanie mien súborov do *i*-node (alebo ekvivalentu) nás vedie k otázke, ako je organizovaný systém adresárov.

Možností je viacero. Začneme od najjednoduchšej, ktorú používa operačný systém CP/M. V ňom existuje len jeden adresár pre všetky súbory, t.j. na to, aby sme našli súbor, stačí prehľadať jeden adresár. Položky v adresári obsahujú aj čísla (adresy) blokov (16), ktoré tvoria súbor.

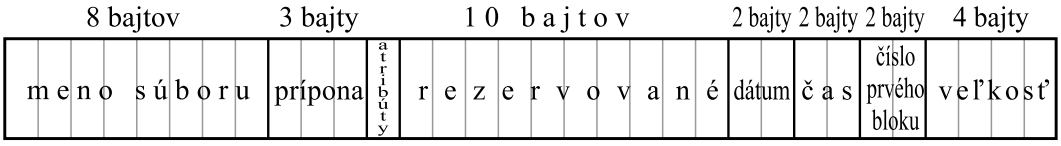


Ak súbor používa viac blokov, ako sa zmestí do jednej položky adresára, súboru sa vyhradia ďalšie položky v adresári. Časť extent sa používa práve v tejto situácii. Hovorí, ktorá položka ide prvá, druhá,

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.

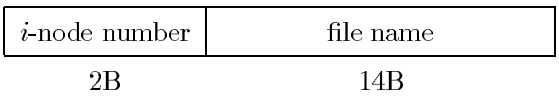
Teraz si preberieme príklady systémov s hierarchickými stromovými štruktúrami adresárov.

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

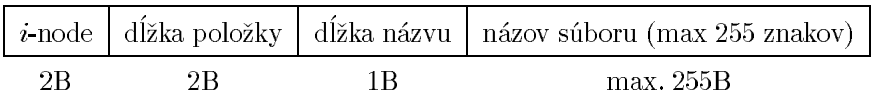


Okrem hlavného adresára, ktorý má pevnú dĺžku, ostatné adresáre sú súbory, a teda môžu obsahovať ľubovoľný počet položiek.

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

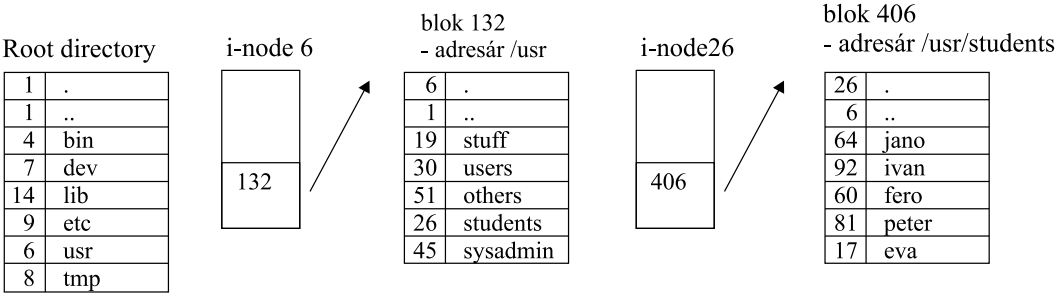


Od BSD 4.2 majú BSD systémy položky premenlivej dĺžky a umožňujú názov súboru až po 255 znakov:



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.

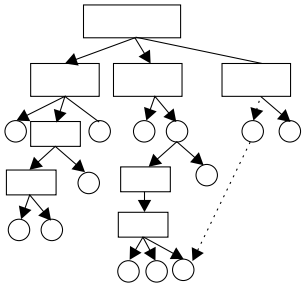


Relatívne názvy súborov hľadáme analogicky, avšak nezačíname z hlavného, ale z aktuálneho adresára.

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.

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

blokov, ktoré logicky patria na disk, ale udržiavajú sa v pamäti.

Na spravovanie cache možno použiť rôzne algoritmy, ale najvšeobecnejší je: Prezrieť pri požiadavke na čítanie, či požadovaný blok nie je v cache. Ak áno, požiadavka na čítanie môže byť uspokojená bez prístupu na disk. Ak blok nie je v cache, najprv sa načíta do cache a potom je skopírovaný tam, kam treba. Ďalšie požiadavky na tento blok sú uspokojené z cache.

Ak je cache pamäť plná, treba nejaké bloky vymazať a zapísať na disk, ak boli modifikované. Táto situácia je podobná stránkovaniu a je možné použiť všetky spomínané nahradzovacie algoritmy. Keďže počet odkazov do cache je relatívne malý, je možné udržiavať bloky v presnom LRU poradí v linkovanom zozname. Avšak v tomto prípade je problém s možnou haváriou systému: ak je nejaký kritický blok (napr. blok *i*-nodu) čítaný do cache a modifikovaný, ale nie znovu zapísaný na disk, havária by mohla nechať systém v nekonzistentnom stave. Ak sa takýto blok zaradí na koniec zoznamu, bude trvať istý čas, kým sa opäť dostane dopredu a bude zapísaný na disk. Naviac, niektoré bloky (napr. double indirect) sa málokedy používajú dvakrát v krátkom časovom intervale. Používa sa preto modifikovaný LRU, ktorý berie do úvahy

- či bude blok pravdepodobne potrebný čoskoro znovu
- či je blok dôležitý pre konzistentnosť systému

Také bloky, ktoré budú pravdepodobne znovu potrebné, idú do LRU radu na koniec a také, ktoré pravdepodobne nebudú čoskoro potrebné, idú dopredu. Bloky dôležité pre konzistentnosť file systému musia byť zapísané na disk hneď, ako boli modifikované, bez ohľadu na to, na ktorý koniec LRU listu boli zaradené.

Aj napriek tejto úprave je neželateľné, aby nejaké dátové bloky boli v cache príliš dlho pred zapísaním (je možné stratiť dáta pri havárii systému, a to aj vtedy, keď používateľ dal príkaz na uloženie). Na riešenie je možné použiť dva prístupy:

- Unix používa systémové volanie sync, ktoré spôsobí zápis modifikovaných blokov na disk. Toto volanie sa vykonáva každých 30 sekúnd (robí to v nekonečnom cykle program update, ktorý beží na pozadí od štartu systému).
- MS-DOS zapíše modifikovaný blok na disk hneď po jeho zapísaní do cache. Také cache sa nazývajú *write-through cache*. Vyžaduje to viac V/V operácií.

Kapitola 12

Správa periférií

Časť operačného systému zabezpečujúca ovládanie periférnych zariadení, sa nazýva *správa periférií*. Jej základné funkcie sú:

- sledovanie stavu všetkých zariadení - pomocou dátovej štruktúry *riadiaci blok jednotky* (*Unit Control Block*, UCB)
- rozhodovanie o prideľovaní periférnych zariadení
- pridelenie periférneho zariadenia procesu
- uvoľňovanie pridelených periférnych zariadení

Modul, ktorý realizuje funkciu sledovania stavu periférnych zariadení sa nazýva *V/V dispečer*. Rozhodovanie, kedy bude V/V zariadenie pridelené žiadajúcemu procesu realizuje *V/V plánovač*. Určuje, ktorá požiadavka bude spracovaná najskôr, v prípade, že na V/V zariadenie čaká viac V/V požiadaviek. Používa na to rôzne stratégie, napr. môže prideliť V/V požiadavkám zariadenia podľa priorít príslušných procesov.

12.1 Klasifikácia periférnych zariadení

Periférne zariadenia môžeme rozdeliť na dve hlavné skupiny:

- V/V zariadenia: Zabezpečujú styk počítača s okolitým prostredím.
  - Vstupné zariadenia: Zvyčajne sú to snímače diernych štítkov, pásky, terminály, skenery, môžu to byť aj prístroje (radary, teplomery).
  - Výstupné zariadenia: Zvyčajne tlačiareň, terminál, dierovač diernych štítkov.
- Vonkajšia pamäť: Zariadenie na uchovávanie informácií.

  Existujú dva typy:
  - Pamäťe so sekvenčným prístupom (*sequential access*): Informácie sú ukladané aj čítané v sekvenčnom poradí. Prístup k položke vyžaduje „lineárne“ hľadanie. Príkladom je magnetická páska.
  - Pamäte s priamym prístupom (*direct access*): Napr. magnetické bubny a magnetické disky.

Periférie môžeme deliť aj podľa prenesenej informácie na základe jedného príkazu na

- *blokové zariadenia*: Uchovávajú informáciu v blokoch, z ktorých každý má svoju vlastnú adresu. Je možné čítať a zapisovať informácie po blokoch, napr. disk. Môžu byť blokovo adresovateľné (disk) alebo neadresovateľné (magnetické pásky).

- *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:

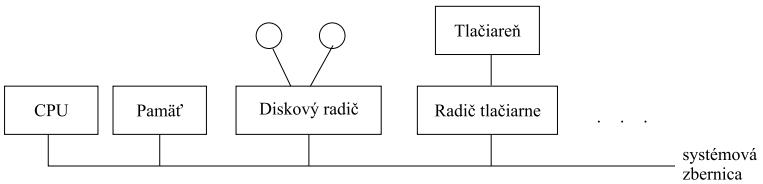
- Pevne prideľované periférne zariadenia (*dedicated*): zariadenie je pridelené úlohe po celú dobu jej trvania. Je to vhodné pre určité typy V/V zariadení, ako napr. snímače štítkov, tlačiarne, ...
- Zdieľané periférne zariadenia (*shared*): Ide o zariadenia používané viacerými procesmi (ako napr. väčšina 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

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.

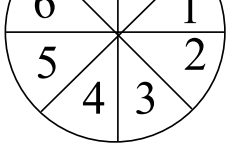
Mnohé radiče, najmä pre blokové zariadenia, umožňujú *priamy prístup do pamäte* (*Direct Memory Access*, DMA). Najprv si vysvetlíme, ako prebieha čítanie bez použitia DMA: Najprv radič číta blok zo zariadenia sériovo, bit po bite, až kým nie je celý blok vo vnútornom bufferi radiča. Ďalej vykoná výpočet 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.

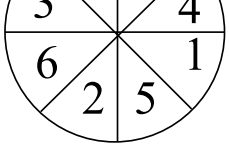
Vzniká otázka, prečo radič používa svoj buffer a nekopíruje bajty priamo do hlavnej pamäte po tom, ako ich získa z disku. Dôvodom je, že keď je začatý diskový prenos, bity prichádzajú z disku konštantnou rýchlosťou bez ohľadu na to, či je radič pripravený alebo nie. Ak by sa radič pokúšal priamo zapísať 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.

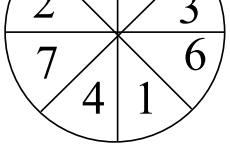
Preskočenie bloku (príp. viacerých) na to, aby mal radič čas na prenos dát do pamäte, sa nazýva *interleaving* (*prekladanie*). Keď sa disk formátuje, bloky sa číslujú na základe „prekladacieho“ faktora. To umožňuje operačnému systému čítať bloky idúce číslovaním za sebou s maximálnou možnou rýchlosťou.



bez prekladania



s jednoduchým prekladaním



s dvojitým prekladaním

12.2.1 Vývoj V/V funkcií

S vývojom počítačových systémov sa vyvíjali ich jednotlivé komponenty. Toto je možné pozorovať aj na vývoji V/V funkcií:

- Procesor priamo riadi periférne zariadenia.
- Je pridaný radič, ale procesor používa programované V/V operácie bez prerušení (procesor zadá príkaz radiču a proces, ktorý V/V operáciu žiadal, činne čaká na jej dokončenie). Procesor je odťažený od znalosti špecifických detailov komunikácie so zariadením.
- K predošlej konfigurácii sú pridané prerušenia (procesor po zadaní V/V príkazu radiču pokračuje vo vykonávaní a je prerušený radičom, keď je operácia dokončená). Nestráca sa čas čakaním na dokončenie V/V operácie a zvýši sa tým výkonnosť procesora.
- Radiču je pridaný priamy prístup do pamäte (DMA).
- Radič je rozšírený a stáva sa samostatným procesorom so špecializovanou množinou inštrukcií pre V/V. Takýto radič sa často nazýva *V/V kanál*. CPU dá príkaz V/V kanálu, aby vykonal V/V program v hlavnej pamäti. V/V kanál číta a vykonáva inštrukcie bez zásahu CPU. Čiže procesor môže zadať sekvenciu V/V aktivít a byť prerušený až po jej dokončení.
- V/V kanál má aj vlastnú lokálnu pamäť a vlastne sa stáva samostatným počítačom. Označuje sa ako *V/V procesor*. Pomocou takejto architektúry je možné riadiť veľké množstvo periférnych zariadení s minimálnym zásahom procesora. Zvyčajne sa toto používa na riadenie komunikácie s interaktívnymi terminálmi.

12.3 V/V sofware

12.3.1 Ciele V/V softwaru

Pri návrhu V/V softwaru sú najdôležitejšie dva ciele: *výkonnosť* (produktivita) a *všeobecnosť*.

Väčšina V/V zariadení je v porovnaní s hlavnou pamäťou a procesorom extrémne pomalá. Jeden spôsob na odstránenie tohto problému je multiprogramovanie, čiže kým nejaké procesy čakajú na dokončenie V/V operácí, vykonáva sa iný proces. Avšak aj pri veľkých operačných pamätiach dnešných počítačov sa môže často stať, že V/V nestačí aktivitám procesora. Na nahratie ďalších pripravených procesov do pamäte, aby sa využil čas procesora, je možné použiť swapovanie, ale to je tiež V/V operácia. Takže hlavné úsilie pri návrhu V/V je nájsť spôsoby na zvýšenie výkonnosti V/V. V ďalšom sa budeme zaoberať zvýšením výkonnosti diskových V/V operácií, lebo je to veľmi dôležitá otázka.

Ďalší dôležitý cieľ je všeobecnosť. V záujme jednoduchosti a zamedzenia chýb je žiadúce, aby sa so všetkými zariadeniami narábalo jednotným spôsobom. To sa týka aj spôsobu, akým procesy vidia V/V zariadenia aj spôsobu, akým operačný systém riadi V/V zariadenia a operácie. Vzhľadom na rozličnosť zariadení, je v praxi ťažké dosiahnuť skutočnú všeobecnosť. Čo sa však dá urobiť, je použiť hierarchický, modulárny prístup k návrhu V/V funkcií. Tento prístup skryje väčšinu detailov V/V zariadení v nízkoúrovňových rutinách, takže procesy a vyššie vrstvy operačného systému vidia zariadenie v termínoch všeobecných funkcií, ako read, write, open, close, lock, unlock.

Dôležitým problémom je ošetrenie chýb. Vo všobecnosti, chyby majú byť čo najviac ošetrené hardwarom. Ak radič objaví chybu pri čítaní, mal by sa ju snažiť opraviť sám. Ak to nedokáže, tak sa o ňu stará device driver (napr. snaží sa prečítať blok znova).

Tieto princípy (ciele) môžu byť dosiahnuté efektívnym spôsobom, keď štrukturujeme V/V software do 4 vrstiev:

1. interrupt handlers (spracovanie prerušení)
2. device drivers (ovládače zariadení)
3. device independent I/O software (V/V software nezávislý od zariadení)
4. user level software (software na užívateľskej úrovni)

12.3.2 Interrupt handlers

Keď sa vyskytne prerušenie, prerušovacia procedúra zabezpečí *odblokovanie* procesu čakajúceho na V/V (up na semafóre, signal na nejakej podmienke monitora, resp. vyslanie správy blokovanému procesu). Tým bude proces, ktorý bol blokovaný, pripravený na vykonávanie.

12.3.3 Device drivers

Všetok kód závislý od zariadenia ide do ovládačov zariadení. Každý ovládač riadi jeden typ zariadenia alebo nanajvýš triedu úzko súvisiacich zariadení.

Ovládačee majú za úlohu prijímať požiadavky od softwaru nezávislého od zariadenia a postarať sa o ich vykonanie, t.j. preložiť požiadavku z abstraktnej formy do konkrétnych termínov (napr. pre ovládač disku to znamená: na ktorom disku je požadovaný blok, preveriť, či motor zariadenia beží, zistiť, či rameno je na správnom cylindri, ...) a rozhodnúť, ktoré operácie radiča sa majú vykonať a v akom poradí. Tieto operácie ovládač zapíše do registrov radiča. Potom môže nastať jedna z dvoch situácií:

- Ovládač musí čakať na to, kým radič preň urobí nejakú prácu, t.j. zablokuje sa až do príchodu prerušenia.
- Ovládač sa nemusí zablokovať (napr. scrollovanie obrazovky)

Po ukončení operácie ovládač (zobudený prerušením alebo vôbec nespiaci) musí zistiť, či nenastali chyby. Ak nie, môže poslať dáta do device-independent softwaru a vrátiť stavovú informáciu o stave volajúcemu procesu. Ak sú ďalšie požiadavky čakajúce na V/V, vyberie sa nejaká a vykoná sa. Ak nečaká žiadna, ovládač sa zablokuje a čaká na požiadavku.

12.3.4 Device-independent I/O software

Veľká časť V/V sofwaru je nezávislá na zariadení. Základná funkcia tejto vrstvy je vykonávať V/V funkcie spoločné pre všetky zariadenia a poskytovať jednotný interface pre používateľský software. Má na starosti mapovanie symbolických mien zariadení do vlastných zariadení. (Napr. v Unixe meno zariadenia (ako /dev/tty0) špecifikuje *i*-node pre špeciálny súbor. Tento *i*-node obsahuje *major device number*, ktoré sa používa na lokalizovanie príslušného ovládača. Obsahuje tiež *minor device number*, ktoré sa odovzdáva ako parameter ovládača na špecifikovanie jednotky, ktorá má byť čítaná alebo zapisovaná.) Ďalej sa stará o ochranu zariadenia pred neoprávneným prístupom, poskytuje jednotnú veľkosť blokov vyšším vrstvám (disky môžu mať rôznu veľkosť sektorov) napr. tak, že zaobchádza s niekoľkými sektormi ako s jedným logickým blokom. Zabezpečuje tiež buffering, stará sa o prideľovanie pevne prideliteľných zariadení a ošetrovanie chýb, ktoré nevie ošetriť ovládač. Do tejto vrstvy patrí aj algoritmus na zistenie voľných blokov na disku pre pridelenie súboru.

12.3.5 User level software

Hoci väčšina V/V software je vnútri operačného systému, malá časť je v knižniciach spojených s používateľskými programami alebo sú to celé programy bežiace mimo kernelu (napr. formátovanie vstupu a výstupu printf sa robí knižničnými funkciami).

Nie všetok V/V software používateľskej úrovne sú knižničné procedúry. Ďalšou dôležitou kategóriou je *spooling system*. (Napr. pre tlač: V systéme je špeciálny proces — *daemon* a špeciálny adresár — *spooling directory*. Ak proces chce tlačiť, najprv generuje celý výstup a uloží tento súbor do spooling adresára. Daemon je jediný proces, ktorý môže používať špeciálny súbor tlačiarne, aby vytlačil súbory z adresára.) Spooling sa používa napr. aj pri presune súborov cez sieť využitím *network daemona* a *network spooling directory*.

12.4 Disky

Čas na čítanie alebo zápis bloku na disk je určený tromi faktormi:

- *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 periférnom 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

  - 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.

Obsah

1 Systémové programovanie **2**
- 1.1 Štruktúra počítača . . . 2
- 1.2 Reprezentácia dát . . . 3
  - 1.2.1 Numerické dátové typy . . . 3
- 1.3 Jazyk asemblera . . . 4
  - 1.3.1 Typy a formát inštrukcií . . . 4
  - 1.3.2 Adresné spôsoby . . . 5
  - 1.3.3 Štruktúra programu . . . 7
  - 1.3.4 Niektoré príkazy jazyka asemblera . . . 7
  - 1.3.5 Procedúry . . . 9
- 1.4 Asembler - prekladač . . . 12
- 1.5 Makrá, makroprocesory . . . 13
- 1.6 Linker a loader . . . 17

2 Úvod do OS, história OS, história Unixu **20**
- 2.1 História operačných systémov . . . 21
- 2.2 História Unixu . . . 23

3 Členenie OS, služby OS **24**
- 3.1 Čo je operačný systém? . . . 24
- 3.2 Koncepcia OS . . . 24
- 3.3 Štruktúra OS . . . 26
- 3.4 Členenie OS . . . 29

4 Procesy **30**
- 4.1 Hierarchia procesov . . . 30
- 4.2 Stavy procesov . . . 30
- 4.3 Popis procesu . . . 34

5 Synchronizácia a komunikácia procesov **35**
- 5.1 Synchroniácia procesov . . . 35
- 5.2 Návrhy na dosiahnutie vzájomného vylúčenia . . . 36
- 5.3 Komunikácia medzi procesmi . . . 41
  - 5.3.1 Pipe (rúra) . . . 44

6 Klasické problémy koordinácie procesov **47**
- 6.1 Problém obedujúcich filozofov . . . 47
- 6.2 Problém čitateľov a zapisovateľov . . . 48

7 Uviaznutie **54**
7.1 Ignorovanie . . . 54
7.2 Detekcia a vyvedenie . . . 55
7.3 Prevencia . . . 57
7.4 Vyhýbanie sa . . . 58

8 Správa procesov a procesora **61**
8.1 Plánovače . . . 61
8.2 Plánovacie algoritmy . . . 63
8.2.1 Nepreemptívne (nonpreemptive) plánovacie algoritmy . . . 64
8.2.2 Preeemptívne (preemptive) plánovacie algoritmy . . . 66
8.3 Policy versus mechanism . . . 68

9 Správa pamäte — modely reálnej pamäte **70**
9.1 Typy správy pamäte (historický prehľad) . . . 71
9.1.1 Jeden súvislý úsek (monoprogramovanie) . . . 71
9.1.2 Statické súvislé úseky (Fixed partitions) . . . 72
9.1.3 Dynamické súvislé úseky (Variable partitions) . . . 73
9.1.4 Buddy systém . . . 74
9.1.5 Stránkovanie . . . 74
9.1.6 Segmentácia . . . 76
9.1.7 Kombinované systémy . . . 77

10 Správa pamäte — modely virtuálnej pamäte **79**
10.1 Nahradzovacie algoritmy . . . 80
10.2 Stránkovanie na žiadosť (demand paging) . . . 81
10.3 Lokálne vs. globálne prideľovacie stratégie . . . 82
10.4 Problémy pri implementácii . . . 82
10.5 Virtualizácia pamäte segmentáciou na žiadosť . . . 83
10.6 Správa pamäte v Unixe . . . 83
10.6.1 Swapovanie . . . 83
10.6.2 Stránkovanie . . . 84

11 Správa súborov **86**
11.1 Použí vateľské hľadisko . . . 86
11.1.1 Typy súborov . . . 86
11.1.2 Atribúty súboru . . . 87
11.1.3 Nezávislosť na zariadení . . . 87
11.1.4 Štruktúra (organizácia) súboru . . . 87
11.1.5 Prístup k súboru . . . 87
11.1.6 Operácie so súbormi . . . 88
11.1.7 Adresáre . . . 88
11.2 Správa priestoru na disku . . . 88
11.3 Implementácia systému súborov . . . 89
11.3.1 Súvislá alokácia . . . 89
11.3.2 Spájaný zoznam blokov na disku . . . 89
11.3.3 Spájaný zoznam s indexom . . . 90
11.3.4 *i*-node . . . 90

11.4 Implementácia adresárov 91
11.5 Zdieľané súbory 93
11.6 Výkonnosť file systému 93

12 Správa periférií **95**
12.1 Klasifikácia periférnych zariadení 95
12.2 Technicé charakteristiky periférnych zariadení 96
12.2.1 Vývoj V/V funkcií 98
12.3 V/V sofware 98
12.3.1 Ciele V/V softwaru 98
12.3.2 Interrupt handlers 99
12.3.3 Device drivers 99
12.3.4 Device-independent I/O software 99
12.3.5 User level software 99
12.4 Disky 99
12.5 Hodiny (clocks) 100