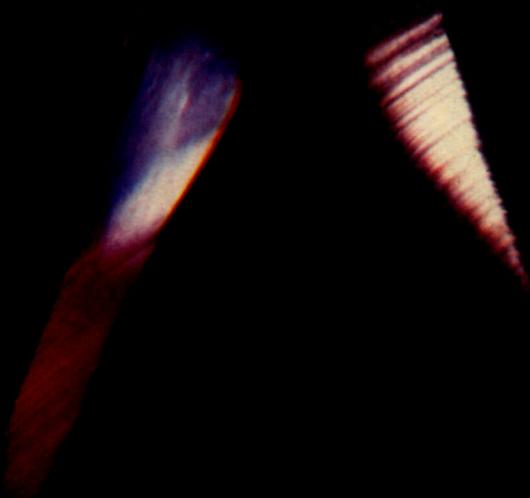


ONDŘEJ ČADA

OPERAČNÍ SYSTÉMY

- nepostradatelná příručka každého odborníka ■
- zevrubný popis struktury, činnosti i služeb ■
- přehled nejrozšířenějších operačních systémů ■





Ondřej Čada

Operační systémy

Copyright © Ondřej Čada, 1993

Photo © Vladimír Svoboda, Cover Design © Karel Kárász, 1993

Ondřej Čada

Operační systémy

V knize použité názvy programových produktů, firem apod. mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Veškerá práva vyhrazena. Žádná část této publikace nesmí být reprodukována, uchovávána v rešeršním systému, nebo přenášena jakýmkoli způsobem včetně elektronického, mechanického, fotografického či jiného záznamu bez předchozí dohody a písemného svolení nakladatelství.

ISBN 80-85623-44-7

Rád bych poděkoval všem autorům knih vyjmenovaných v dodatku D i všem autorům mnoha dalších knih, na které jsem zapomněl. Bez nich by tato knížka nikdy nemohla vzniknout.

Rád bych také poděkoval všem svým učitelům a zvláště Jirkovi Hořejšovi za to, že mě naučili vše, co se nyní snažím neuměle předat čtenářům. Přitom je však nutné mít na paměti, že všechny chyby a nedostatky této knihy jsou výhradně mým vlastním dílem.

Autor.

Obsah

Díl první	21
1. Úvod	15
1.1 Přehled obsahu knihy	16
1.2 Příklady operačních systémů	16
2. Co je to operační systém	23
3. Struktura operačního systému	25
3.1 Vyšší jazyk	25
3.2 Jednoduché a krátké zdrojové texty	27
3.3 Objektový přístup	28
3.4 Vrstvená struktura	29
3.5 Návrh 'běžícího' systému	29
3.6 Ošetření výjimek	31
4. Operační paměť	33
4.1 Přidělování paměti po blocích	34
4.1.1 Informace o blocích	35
4.1.2 Fragmentace paměti	37
4.1.3 Alokační strategie	38
4.2 Přesunování bloků	39
4.2.1 Kooperativní přesunování bloků	43
4.2.2 Transparentní přesunování bloků	47
4.3 Virtuální paměť	50
4.3.1 Potřebné technické vybavení	51
4.3.2 LRU a pseudo-LRU	57
4.3.3 Virtualizace paměti	58
4.3.4 Implementace správce paměti	61
4.3.5 Virtualizace adres	63
4.3.6 Další informace	64
4.4 Segmentace	64

4.5 Ochrana paměti	66
5. Procesy a procesor	69
5.1 Co je to multitasking	69
5.2 Princip multitaskingu	73
5.2.1 Přepínání programů	73
5.2.2 Kontext	75
5.2.3 Kooperativní multitasking	78
5.2.4 Preemptivní multitasking	81
5.2.5 Sdílení času	83
5.2.6 Implementace	83
5.3 Správa procesů	85
5.3.1 Správa front	86
5.3.2 Tabulka procesů	88
5.3.3 Stavy procesu	89
5.3.4 Nulový proces	91
5.3.5 Plánování	92
5.3.5.1 Služby 'resched' a 'ready'	93
5.3.5.2 Služby 'resume' a 'suspend'	97
5.3.5.3 Služba 'kill'	99
5.3.5.4 Služba 'create'	101
5.3.6 Priority a sdílení času	106
5.3.7 Správa času	107
5.3.7.1 Služba sleep	108
5.3.7.2 Služba wakeup	109
5.4 Synchronizace procesů	110
5.4.1 Semafora	111
5.4.1.1 Binární semafory	112
5.4.1.2 Obecné semafory	114
5.4.1.3 Semafora v multiprocesorovém prostředí	118
5.4.2 Kritické sekce bez semaforů?	119
5.4.3 Zprávy	121
5.4.4 Deadlock	124
5.5 Správa úloh	125
6. Ovladače periferií	127

6.1	Vstupní a výstupní zařízení	127
6.1.1	Vyhrazená zařízení	128
6.1.2	Sdílená zařízení	131
6.1.3	Společná zařízení	131
6.2	Ovladače zařízení	132
6.2.1	Klasické ovladače	133
6.2.1.1	Logický systém ovladačů	136
6.2.1.2	Přístup programů ke službám	138
6.2.1.3	Horní polovina	140
6.2.1.4	Obsluha přerušení	145
6.2.1.5	Dolní polovina	148
6.2.2	Servery	151
6.3	Postavení ovladačů v operačním systému	152
6.4	Obrana proti deadlocku	152
6.4.1	Úplné vyhrazení prostředků	153
6.4.2	Hierarchické přidělování prostředků	154
6.4.3	Bankéřův algoritmus	156
6.4.4	Detekce deadlocku	159
6.5	Ovládání konkrétních zařízení	162
6.5.1	Sdílená data	163
6.5.2	Systémový časovač	164
6.5.2.1	Pozastavení hodin	165
6.5.2.2	Obslužná rutina přerušení časovače	169
6.5.2.3	Preempce a bezpečnost	171
6.5.3	Obrazovka, klávesnice, myš	172
6.5.3.1	Obrazovka	173
6.5.3.2	Klávesnice	175
6.5.4	Tiskárna	175
6.5.5	Disk	176
6.5.5.1	Služby ovladače	178
6.5.5.2	Potřebná data	179
6.5.5.3	Práce s frontou požadavků	181
6.5.5.4	Horní polovina ovladače	187
6.5.5.5	Dolní polovina ovladače	190
6.6	Ovladače a bezpečnost	192
6.6.1	Příčiny nebezpečí	192
6.6.2	Nebezpečné příkazy	194

6.6.3 'Callback' rutiny	194
6.6.4 Přístup k paměti	195
6.6.5 Změna média	195
6.6.6 Výpadek systému	196
7. Systém souborů	199
7.1 Systém souborů	200
7.2 Systém adresářů	201
7.3 Formátované soubory	202
7.4 Sdílení souborů	202
7.5 Soubory a bezpečnost	203
8. Sítě	207
8.1 Co je to lokální síť	207
8.1.1 Technické vybavení lokální sítě	208
8.1.2 Programové vybavení lokální sítě	210
8.1.2.1 Opět vrstvená struktura	210
8.1.2.2 Cesta dat sítí	212
8.1.2.3 Umístění síťového software v systému	217
8.2 Globální sítě	218
8.3 Implementace lokální sítě	219
8.3.1 Linková vrstva	223
8.3.1.1 Funkce ovladače	227
8.3.1.2 Společné deklarace	234
8.3.1.3 Horní polovina	237
8.3.1.4 Dolní polovina	241
8.3.2 Síťová vrstva	245
8.3.2.1 Návrh protokolu	245
8.3.2.2 Struktura síťové vrstvy	249
8.3.2.3 Společné deklarace	250
8.3.2.4 Rozhraní pro vyšší vrstvy	252
8.3.2.5 Vstupní proces	253
8.3.2.6 Výstupní proces	258
8.3.2.7 Časovači proces	261
8.4 Sítě a bezpečnost	262
8.4.1 Lokální sítě	262
8.4.2 Globální sítě	263

9. Systém služeb	265
9.1 Jaké služby potřebujeme	265
9.2 Implementace služeb	268
9.2.1 Služby systému	268
9.2.2 Klasické knihovny	270
9.2.3 Servery	271
9.2.4 Sdílené knihovny	272
9.2.5 Objekty	273
9.3 Obsah služeb	274
9.3.1 Textové služby	274
9.3.2 Národní prostředí	275
9.3.3 Databáze	279
9.3.4 Komunikace programů	280
9.3.4.1 Schránka	282
9.3.4.2 Inteligentní schránka	283
9.3.4.3 Spojení dat	284
9.3.4.4 'Drag and drop'	284
9.3.4.5 Datové služby	285
Díl druhý	287
10. Nástup grafiky	289
11. Struktura grafického systému	291
12. Okna	295
12.1 Co je to okno	295
12.2 Okna mimo obrazovku?	297
12.3 Hierarchie oken	297
12.4 Překrývající se okna	300
12.4.1 Vlastnictví oken	300
12.4.2 Správné překreslování	301
12.4.3 Překreslení okna	302
13. Uživatelské rozhraní	305
13.1 Okna	307

13.2 Ikony	316
13.3 Nabídky	317
13.4 Dialogová okna	319
13.5 Výstražné dialogy	324
13.6 Indikátory	324
13.7 Kurzory	326
Díl třetí	329
14. Pohled uživatele	331
14.1 Interpret příkazů	331
14.2 Příkazy	334
14.3 Aplikace	335
14.4 Vývojové prostředí	336
Dodatky	337
A. Přehled operačních systémů	339
Amiga DOS	339
ATARI DOS	339
CP/M	340
DOS EC	341
EPOC	341
GEM	342
MACH	342
Macintosh System 7	343
MINIX	344
MS DOS (PC DOS)	344
MS Windows	344
Multics	345
NeXTStep	346
OS/2	347
OS 360/370	347
Solaris	347
TOS	348
UNIX (XENIX, VENIX, AIX, A/UX, HP/UX ...)	348

Windows NT	349
XINU	350
tabulka	350
B. Jazyk C	355
B.1 Syntaktické odlišnosti PASCALu a C	355
B.2 Datové typy	356
B.3 Deklarace	357
B.4 Operátory	359
B.5 Příkazy	360
B.5.1 Příkaz if	360
B.5.2 Příkaz for	361
B.5.3 Příkazy while, do	361
B.5.4 Příkaz switch	361
B.5.5 Příkaz goto	362
B.5.6 Příkazy break a continue	362
B.5.7 Příkaz return	362
B.6 Funkce	363
B.6.1 Hlavičky funkcí	363
B.7 Preprocesor	364
B.8 Inicializace	364
C. Strojově závislá rozšíření Turbo C	367
C.1 Funkce typu interrupt	367
C.2 Práce na strojové úrovni	368
C.2.1 Přístup k registrům	368
C.2.2 Přístup k paměti	368
C.2.3 Strojový kód	368
D. Literatura	371
Rejstřík	373

Seznam obrázků

obr. 1: vrstvená struktura XINU.	29
obr. 2: přidělování bloků paměti	34
obr. 3: údaje o blocích, první způsob	35
obr. 4: údaje o blocích, druhý způsob	36
obr. 5: fragmentace paměti	37
obr. 6: setřesená paměť	40
obr. 7: jak funguje handle	44
obr. 8: paměť v EPOCu	49
obr. 9: překlad adres	53
obr. 10: překlad adres / stránkování	54
obr. 11: odebrání stránky	55
obr. 12: přidělení nové stránky	56
obr. 13: přepnutí kontextu	77
obr. 14: stavy procesu	90
obr. 15: služby správce procesů	92
obr. 16: zásobník po vytvoření procesu	105
obr. 17: síť se dvěma zónami	212
obr. 18: topologie sítě XINU.	220
obr. 19: tok dat v síti	223
obr. 20: stavy vstupní rutiny ovladače	229
obr. 21: stavy výstupní rutiny ovladače	232
obr. 22: vrstvený grafický systém	291
obr. 23: okno na obrazovce	295
obr. 24: totéž okno po přemístění	296
obr. 25: hierarchie oken	298
obr. 26: okna EPOCu na velké obrazovce	299
obr. 27: okno v NeXTstepu	308
obr. 28: titulek a podtitulek (EPOC)	309
obr. 29: změna velikosti okna v EPOCu	310
obr. 30: vertikální posuvník na Macintoshi	314
obr. 31: panely v okně EPOCu	315
obr. 32: panely v Nisu a mnoho ikon	316
obr. 33: ikony EPOCu	317
obr. 34: běžná nabídka EPOCu	317
obr. 35: hierarchická nabídka na Macintoshi	318
obr. 36: dialog na Macintoshi	320
obr. 37: rozevírací nabídka v EPOCu	321
obr. 38: dialog se seznamem souborů na Macintoshi	322
obr. 39: indikátor postupu kopírování na Macintoshi	325
obr. 40: kurzory programu ResEdit	326

1. Úvod

V této knize se budeme zabývat operačními systémy. Ačkoli se samozřejmě seznámíme s řadou existujících operačních systémů, s jejich strukturou, vlastnostmi, výhodami i nevýhodami, hlavním úkolem této knížky je vysvětlit, **co to vlastně operační systém je**, jak by měl vypadat a jaké služby by měl zajišťovat. Přehled častěji užívaných operačních systémů nalezne případný zájemce v dodatku A.

Jak uvidíme později, je sám termín 'operační systém' dost široký a každý autor jej používá v trochu jiném významu. Všichni se shodují v tom, že základní součástí operačního systému je řada programů a ovladačů nezbytných pro provoz počítače. Často se však za součást operačního systému považuje ještě další software: grafický systém, používaný u všech moderních osobních počítačů pro komunikaci s uživatelem, a větší či menší množství aplikačních programů, dodávaných výrobcem jako součást operačního systému (typickým příkladem může být program pro kopírování souborů nebo jednoduchý textový editor).

V této knize budeme pojmem 'operační systém' používat převážně v nejužším významu (definici operačního systému, podle které budeme postupovat, čtenář nalezne v následující kapitole). V samostatných oddílech se však budeme zabývat i ostatním programovým vybavením, dodávaným obvykle společně s operačním systémem (tj. grafickým systémem, připojenými aplikačními programy a podobně). Tam, kde nebude hrozit nedorozumění, pak použijeme pojmem 'operační systém' i v některé z jeho širších interpretací - napíšeme např. 'součástí operačního systému NeXTStep je DTP systém $T_E X$ ' namísto správnějšího ale delšího i méně přehledného 'firma NeXT automaticky dodává společně s operačním systémem NeXTStep také DTP systém $T_E X$ '.

1.1 Přehled obsahu knihy

Kniha má několik relativně samostatných dílů; každý z nich popisuje jinou část programového vybavení.

- První díl je nejdůležitější součástí knihy a obsahuje podrobný popis vlastního operačního systému a jeho hlavních součástí (jako je správce paměti, správce procesů apod.). Zvláště podrobně se budeme věnovat realizaci multitaskingu - tedy správě procesů.
- Ve druhém dílu se budeme zabývat grafickými systémy. Ukážeme si, jaké služby grafický systém musí zajistit, rozlišíme vlastní grafický systém a grafické uživatelské rozhraní, a seznámíme se s několika existujícími systémy.
- Třetí díl pak bude obsahovat stručné shrnutí aplikačních programů, dodávaných současně s operačním systémem. Uvidíme, že modernější operační systémy bývají obvykle doprovázeny větším množstvím aplikací a stručně probereme několik konkrétních příkladů, od systémů které jsou doprovázeny jen nejnutnějším minimem programů (jako např. MS DOS nebo CP/M) až po systémy doprovázené luxusní paletou řady aplikací (jako je NeXTStep nebo EPOC).
- Na konci knihy najeznete rozsáhlé dodatky, ve kterých je například stručný popis programovacího jazyka C nebo přehled nejznámějších operačních systémů.

1.2 Příklady operačních systémů

Přehled řady dnes užívaných operačních systémů čtenář nalezne v dodatku A. Některé operační systémy však budeme v textu používat častěji pro ilustraci popisovaných faktů; je proto vhodné se s těmito 'reprezentačními' příklady stručně seznámit ihned:

MS DOS pravděpodobně všichni čtenáři této knihy celkem dobře znají. Jedná se o příklad nejjednoduššího možného jednouživatelského systému (ve smyslu definice, kterou použijeme v následující kapitole, je dokonce sporné, jedná-li se vůbec o operační systém - odborníci MS DOS často označují jako 'glorifikovaný zavaděč programů').

XINU je velmi jednoduchý školní multitaskový operační systém, který popsal pan Douglas Comer ve své výborné knize [XINU]. Ačkoli existují implementace tohoto systému, nelze jej v žádném případě označit za 'používaný'. Jeho výhodou však je jednoduchost, přehlednost a především dostupnost jeho zdrojových textů¹. XINU se proto ideálně hodí pro ilustraci vnitřních mechanismů operačních systémů a budeme se s ním setkávat především v prvním dílu knihy.

PC-XINU, ST-XINU jsou omezené implementace jádra operačního systému XINU pro počítače IBM PC a ATARI ST, které před lety vytvořil autor této knihy - jednu jako diplomovou práci a druhou ze zájmu o to, jak náročný bude přenos jádra systému do jiného prostředí (potřebné úpravy zabraly méně než hodinu času).

Poznamenejme, že příklady uvedené v této knize nejsou přesně autentickým zdrojovým kódem XINU, PC-XINU nebo ST-XINU. Některé technické detaily, které usnadňovaly implementaci ale snižovaly by přehlednost, jsou vypuštěny. Z jednotlivých funkcí je pro zvýšení přehlednosti také úplně vypuštěna kontrola správnosti parametrů; ta by v praxi samozřejmě byla bezpodmínečně nutnou.

¹Zdrojové texty UNIXu jsou sice také dostupné; UNIX však v žádném případě nelze označit za jednoduchý a přehledný.

Macintosh System 7 je pravděpodobně nejdokonalejším kooperativním² operačním systémem. Navíc je vybaven velmi kvalitním grafickým uživatelským rozhraním; z dnes dostupných operačních systémů disponuje lepším uživatelským rozhraním pravděpodobně jen NeXTStep. Tento systém proto budeme využívat pro příklady týkající se kooperativního multitaskingu a společně s NeXTstepem jako referenční systém při diskusi uživatelského rozhraní.

UNIX je zřejmě nejlepším operačním systémem pro výkonné stroje, u kterých lze předpokládat práci s více terminály zároveň nebo zapojení do počítačové sítě. Jedná se o velký a poměrně komplikovaný víceuživatelský operační systém, jehož ovládnutí je v klasických implementacích dost náročné. Často se proto budeme zmíňovat o moderních operačních systémech, které přebírají všechny služby UNIXu, ale doplňují je vlastními nadstavbami, které umožňují programátorům i uživatelům systém snadněji zvládnout.

NeXTStep, vyvinutý firmou NeXT, je příkladem právě takového moderního operačního systému. Jeho jádrem je v zásadě léty prověřený UNIX; sám NeXTStep však nabízí daleko více: uživatelům a správcům intuitivní uživatelské rozhraní s řadou standardních aplikací a programátorům plně objektové a velmi bohaté vývojové prostředí. Budeme jej užívat jako příklad nejmodernějšího operačního systému pro výkonné mikropočítače.

EPOC je jiným příkladem moderního operačního systému, pokrývajícím trochu jinou skupinu požadavků. Zatímco NeXTStep je určen pro výkonné stolní počítače, vyvinuli odborníci firmy PSION systém EPOC pro ovládání kapesních počítačů a tomu samozřejmě přizpůsobili i jeho strukturu a vlastnosti. My si systému EPOC povšimneme hlavně

²Podrobné vysvětlení technických pojmu (jako 'preemptivní' nebo 'kooperativní') čtenář samozřejmě nalezne v dalším textu. Pro orientaci můžeme uvést, že kooperativní systém je takový systém, ve kterém současný běh více programů je sice možný, ale musí jej do značné míry zajistit samy programy (v kooperaci s operačním systémem - odtud jméno 'kooperativní'). Operační systém bez jejich podpory by na to 'nestačil'. Špatně napsaný aplikační program může tedy narušit funkci celého systému.

v souvislosti s technikou tzv. transparentního přesunování bloků při správě paměti.

Díl první

Co je to operační systém

"Zvláště bychom chtěli poděkovat studentům v M.I.T.... za jejich konstruktivní kritiku i za to, že to vůbec přežili."

S.E.Madnick, J.E.Donovan v úvodu knihy [OS]

"Operační systém by měl být jednoduchý, elegantní a snadno použitelný"

Ken Thompson v roce 1969, kdy začínal pracovat na UNIXu

"Tato kapitola si všímá pouze obecných otázek ideální struktury ... praxe není zdaleka tak jednoduchá a elegantní."

Úvod ke kapitole o operačních systémech v [HOŘ]

2. Co je to operační systém

Laik často neví, co si má pod pojmem 'operační systém' vlastně představit. Ani odborníci ale - jak jsme se již zmínili - nemají v této věci úplné jasno.

Dosti často se setkáme s definicí, která dobře odpovídá první intuitivní představě o tom, co to vlastně operační systém je:

"Operační systém je programové vybavení nezbytné pro provoz počítače."

Při podrobnějším rozboru však taková definice začne vykazovat nepříjemnou nejasnost: jaké programové vybavení je proboha nezbytné a bez kterého by to přece jen nějak šlo? Navíc, co to vlastně znamená 'provoz počítače'? Chci-li na počítači tisknout noviny, je pro mě absolutně nezbytné mít nějaký programový systém pro DTP; těžko však řeknu, že tento program je součástí operačního systému. Pokud bychom naopak brali 'provoz počítače' minimalisticky, stačí programové vybavení sestávající z jedné jediné instrukce - totiž skoku na sebe sama. Počítač vybavený takovým programem nám asi nebude příliš mnoho platný, nicméně nepochybňě bude v provozu.

Raději se proto takovýmto pojmem vyhneme a zkusíme se na počítač podívat z trochu jiné stránky. Pro běžného uživatele je počítač především klávesnice, obrazovka a tiskárna, a pak ještě nějaké bedýnky okolo. My jsme ale odborníci, a proto víme, že základem počítače je procesor, nějaká paměť a nějaké I/O procesory (říkáme jim 'kanály')³, které zprostředkovují komunikaci se zařízeními, která jsou k počítači připojena - mezi jiným i se zmíněnou klávesnicí a obrazovkou.

Na všechno, co jsme doposud vyjmenovali, se můžeme dívat jako na prostředky, jichž jednotlivé programy tak či onak využívají ke své práci. Jestliže máme počítač, na kterém běží jen jeden jediný program, je vše bez problémů:

³Téměř všechny systémy Apple Macintosh a nejjednodušší počítače ostatních výrobců nemají kvůli snížení ceny kanály a všechnu práci musí zvládnout sám mikroprocesor. Princip je ale naprostě stejný, pouze namísto kanálů máme přímo ovládací registry jednotlivých vstupních a výstupních zařízení.

program využívá prostředky tak, jak se mu právě zachce. Pokud však má na našem počítači běžet programů několik najednou, je situace složitější: programy se o využívání prostředků musí nějak dohodnout. Některé prostředky může v jednom okamžiku využívat pouze jeden jediný program - příkladem je mikroprocesor nebo třeba tiskárna. Jiné prostředky sice mohou sloužit více programům najednou, tyto programy si však musí rozdělit 'sféry vlivu' - každý může dostat 'kousek' operační paměti nebo třeba jedno okno na obrazovce. Jen velmi málo prostředků může sloužit všem programům bez jakéhokoli omezení; takovým prostředkem jsou např. systémové hodiny.

Chceme-li problémy, nastíněné v minulém odstavci, nějak vyřešit, potřebujeme někoho, kdo se bude o prostředky starat; někoho, kdo bude stát nad všemi programy a prostředky jím bude přidělovat a v případě potřeby i odebírat. Potřebujeme zkrátka správce prostředků. Můžeme ovšem také říci, že potřebujeme operační systém. Došli jsme tak k daleko lepší definici, které se i nadále budeme držet:

"Operační systém je správce prostředků."

Ve zbývajících kapitolách tohoto dílu proto postupně probereme jednotlivé prostředky nebo skupiny prostředků a ukážeme si, jakým způsobem se o ně může operační systém postarat a jak mohou sloužit jednotlivým programům. Jedinou výjimkou je hned následující kapitola, která se zabývá obecnou strukturou operačního systému 'z nadhledu'.

3. Struktura operačního systému

Operační systém jakéhokoli počítače - má-li za něco stát - je poměrně dost rozsáhlým programem. Žádný člověk - nebo téměř žádný, aby se neurazil ani pan Seymour Cray⁴ - si nedokáže udržet přehled o funkci programu, jehož zdrojový kód má desítky nebo stovky tisíc řádků ve vyšším jazyce a stovky až tisíce řádků v assembleru - což je právě tak typický případ průměrného operačního systému. Jestliže chceme, aby se nám podařilo operační systém rychle vytvořit, aby byl spolehlivý a výkonný a aby jeho údržba nebyla přehnaně náročná, musíme využít všech možností, které tvorbu velkých programových balíků usnadní.

Zkušenější programátoři po přečtení následujících odstavců pravděpodobně namítnou, že všechny popsané techniky zvětšují a zpomalují výsledný kód. To je pravda jen do jisté míry - není pochyb o tom, že naprogramujeme-li jednoduchý problém jednou v assembleru a podruhé v objektově orientovaném jazyce s důsledným využíváním dodatečné vazby (late binding), bude první implementace daleko menší a rychlejší. Jedná-li se však o složitější problém, bude výsledek právě opačný - implementace v assembleru bude složitější, nespolehlivá, a po odladění (spojeném s mnohonásobným 'záplatováním') také daleko pomalejší a větší.

3.1 Vyšší jazyk

Ještě neuběhlo mnoho času od dob, kdy se operační systémy psaly v assembleru. Dodnes je zvykem psát v assembleru velmi podstatnou část operačního systému; to ale přináší daleko více nevýhod než výhod.

⁴Seymore Cray, tvůrce superpočítače Cray 1 a většiny počítačů firmy Control Data je programátorem, o němž se vyprávějí legendy. Jedna z nich tvrdí, že pan Cray 'namačkal' operační systém počítače CDC7600 po jeho prvním zapnutí do paměti prostřednictvím čelního panelu počítače, a to zpaměti.

Operační systémy

Ještě v době, kdy vznikal operační systém UNIX, však prakticky neexistoval vyšší jazyk, který by bylo možné použít pro tvorbu operačního systému. Jakkoli je nepochybná pravda, že ve FORTRANu se dá napsat všechno, hodí se tento jazyk pro psaní systémových programů skutečně velmi málo; ostatní starší programovací jazyky - jmennujme ALGOL nebo COBOL - jsou pro tento účel již zcela nepoužitelné. Modernější PASCAL profesora Wirtha je výtečným prostředkem pro zápis algoritmů; doopravdy v něm programovat však hraničí s nemožností⁵.

Průlom v tvorbě operačních systémů nastal ve chvíli, kdy v Bell Laboratories vznikl programovací jazyk C. Tento jazyk vytvořili programátoři pro programátory, a podle toho také vypadá - je efektivní, stručný, silný a velmi přehledný. Navíc je snadno optimalizovatelný; většina překladačů jazyka C díky tomu generuje kód, který není o mnoho horší než kód vytvořený přímo v assembleru⁶. Použitelnost jazyka C byla ihned prokázána velmi praktickým způsobem - byl do něj převeden operační systém UNIX.

Postupem času se ukázalo, že jakkoli je jazyk C oproti assembleru pro tvorbu systémových programů výrazným pokrokem, je pořád poměrně slabý v odhalování nechtěných chyb, omylů a překlepů. Nejprve proto došlo k rozšíření jazyka normou ANSI o řadu vesměs bezpečnostních prvků (jako jsou funkční prototypy, typ void nebo modifikátor const), a později byl jazyk C rozšířen o možnost práce s objekty (Objective C, C++). Naprostá většina příkladů v této knize je psána v ANSI C (čtenář, který jazyk C nezná, nalezne základní informace v dodatku B). Některé příklady, převzaté z implementace operačního systému XINU na počítače třídy IBM PC, využívají určitých rozšíření překladače Turbo C; stručné shrnutí těchto rozšíření je v dodatku C.

⁵Různé modifikace PASCALu - jako je např. Think PASCAL pro Apple Macintosh nebo Turbo PASCAL pro počítače třídy IBM PC - mají s původním PASCALEm asi tolik společného, jako Škoda 130 rallye speciál se škodovkou, kterou máte doma - totiž karosérii, a i ta je jen podobná. Jedná se o podivné kódy, ve kterých sice programovat lze (někdy velmi dobře), ovšem za cenu dost nekonzistentního jazyka a absolutní nepřenositelnosti.

⁶To nemusí být pravda na počítačích, osazených mikroprocesory Intel - jejich instrukční kód je totiž velmi nešťastně navržen a jeho automatická optimalizace je mimořádně obtížná.

-Struktura operačního systému-

Použití vyššího jazyka výrazným způsobem usnadní ladění i údržbu operačního systému. Jestliže se navíc vyhneme používání nepřenositelných konstrukcí⁷, není velký problém vytvořit operační systém nezávislý na počítači, na kterém jej budeme provozovat. Dobrým příkladem velmi dobře přenositelného operačního systému je právě UNIX; tvůrci dnešních operačních systémů právě výhodu přenositelnosti znovaobjevují, takže dalšími příklady mohou být vesměs moderní operační systémy jako je Solaris, NeXTStep a prý i Windows NT (podle zprávy Locusu, o které se zmiňujeme v dodatku A, to však s přenositelností NT asi nebude tak horké).

3.2 Jednoduché a krátké zdrojové texty

Přehlednost zdrojového kódu operačního systému se podstatným způsobem zvýší, jestliže jednotlivé úkoly rozčleníme takovým způsobem, aby ucelené části kódu, které je zpracovávají, nebyly příliš dlouhé. V ideálním případě by neměl být zdrojový kód žádné z funkcí delší než jedna stránka; studium takového systému je pak velmi jednoduché.

Jedná se vlastně o předobraz objektového přístupu - namísto komplikované funkce, jejíž činnost nestačíme sledovat (v horším případě se mylně domníváme, že stačíme), máme jednoduchou funkci, která využívá dobře definovaných základních operací. Obdobným způsobem při OOP využíváme dobře definovaných primitivních objektů pro konstrukci složitějších.

Dobrým příkladem tohoto přístupu je operační systém XINU.

⁷Přesněji řečeno, nepřenositelné části kódu - které z principiálních důvodů vždy budou součástí každého operačního systému - můžeme shrnout do samostatného modulu, odděleného od zbytku systému.

3.3 Objektový přístup

Poměrně moderní přístup k programování - OOP - samozřejmě nezůstal bez vlivu na tvorbu operačních systémů. Jejich tvůrci mají k dispozici i vhodné programátorské prostředky - objektové nadstavby jazyka C: mezi aplikačními programátory velmi rozšířené C++ nebo o něco lépe navržené (i když možná nepatrně méně pohodlné) Objective C.

Operační systém, skládající se z řady více či méně nezávislých správců, je snad nejtypičtějším příkladem programu skutečně předurčeného pro objektový přístup. Hledíme-li na jednotlivé složky systému jako na 'černé skříňky', po kterých můžeme požadovat určité služby, získáme daleko lepší přehled. Objektový operační systém přináší i řadu dalších výhod: daleko jednodušší údržbu a možnost snazší modifikace a rozšiřování systému.

Nesmírnou výhodou pak je objektový systém pro programátory. Dědičnost objektů totiž podstatným způsobem zvyšuje flexibilitu systémových služeb - programátoři mohou jejich funkci v případě potřeby velmi mírně modifikovat, aniž by přitom museli připravovat jiný kód než právě kód zmíněné modifikace. Naprostě bezproblémové využití modifikované služby v aplikaci pak je automaticky zajištěno prostředky pro komunikaci mezi objekty.

Z dnešních operačních systémů využívají výhod objektového přístupu zřejmě pouze dva: EPOC a NeXTStep⁸. Zatímco NeXTStep je vytvořen v Objective C, EPOC je naprogramován ve speciální objektové nadstavbě jazyka C, kterou vyvinula pro vlastní potřeby firma PSION (tato nadstavba je zhruba srovnatelná s Objective C, její konkrétní výrazové prostředky jsou však poměrně neohrabané).

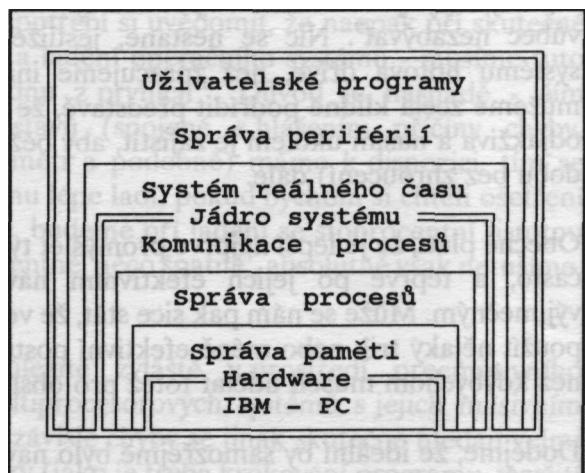
⁸Třetím příkladem objektového operačního systému by podle některých informací měl být PenPoint firmy Go Corporation. I kdyby tomu tak skutečně bylo, nemusíme se jím zvlášť zabývat - systém PenPoint zvolna upadá v zapomnění společně se všemi systémy řízenými perem (není ostatně divu - zásadní výhodu klávesnice proti peru prokazuje již vznik mechanických psacích strojů dávno a dávno před tím, než se zrodila vůbec sama myšlenka počítače).

3.4 Vrstvená struktura

Zajištění lepšího přehledu napomůže i uspořádání jednotlivých součástí operačního systému do jakýchsi vrstev, pro které platí, že nižší vrstva nevyužívá žádných služeb vyšších vrstev.

Není velký problém takového uspořádání docílit. Operační systém se z principu do podobných vrstev rozpadá - např. správce paměti musí být k dispozici téměř všem ostatním složkám, sám však nepotřebuje vědět nic např. o přidělování procesoru.

Na obr. 1 vidíme vrstvenou strukturu operačního systému PC-XINU. Aby byl obrázek jasnější, je v něm jako nejnižší vrstva uvedeno i technické vybavení (hardware), jako nejvyšší vrstva pak aplikační programy - nic z toho samozřejmě není součástí operačního systému, alespoň ve smyslu naší definice.



obr. 1: vrstvená struktura XINU

Takové uspořádání zmenší počet vnitřních závislostí, které musíme mít při návrhu operačního systému na paměti.

3.5 Návrh 'běžícího' systému

Začátečník někdy při návrhu systému udělá tu chybu, že postupuje zdánlivě logickým způsobem - představí si nejprve zavádění systému do paměti a jeho inicializaci, rozvrhne potřebné služby a prostředky, a potom věnuje svou pozornost situaci, kdy již operační systém běží a doplňuje návrh vším potřebným.

To je zásadní chyba. Navrhne-li operační systém tímto způsobem, bude velmi pravděpodobně umožňovat pohodlnou a přehlednou inicializaci bez použití jakýchkoli speciálních triků - což je samo o sobě samozřejmě dobře - ale při běhu bude poněkud těžkopádný; struktury ideální pro jednoduchý start systému si mohou vynutit komplikovanější řešení běhových záležitostí.

Proto je nejlepší navrhovat operační systém nejprve pro běh a inicializací se vůbec nezabývat⁹. Nic se nestane, jestliže bude podstatná část práce na systému hotova dříve, než zpracujeme inicializaci; funkci služeb systému můžeme zcela klidně podřídit představě, že operační systém na počítači běží odjakživa a naším úkolem je zajistit, aby běžel co nejfektivněji (a co nejdelší dobu bez zhroucení) dále.

Obecně platí, že je lepší nejprve promýšlet ty situace, ke kterým dochází velmi často, a teprve po jejich efektivním návrhu se vrátit k situacím spíše výjimečným. Může se nám pak sice stát, že ve výjimečné situaci budeme muset použít nějaký trik nebo méně efektivní postup; to je ale daleko menší chyba, než kdybychom museli udělat totéž pro obsluhu situace zcela běžné.

Dodejme, že ideální by samozřejmě bylo navrhnout celý operační systém tak, že každá situace bude ošetřena přehledně a přitom nejfektivnějším možným způsobem; zkušení programátoři nám však dají za pravdu v tom, že i přes všechny výhody, které přináší vrstvená struktura, objektové programování a podobné prostředky, se to bohužel v praxi u větších projektů podaří jen málokdy. Je proto lepší při návrhu postupovat tak, aby všechny případné nedostatky 'vyplavalny' tam, kde budou relativně nejméně vadit.

⁹Toho využijeme i pro udržení rozsahu této knihy v rozumných mezích - až na naprosté výjimky (mezi které patří např. popis služby 'create', která vytváří proces) se totiž budeme zabývat právě během operačního systému a inicializaci jednotlivých funkčních celků ponecháme stranou.

3.6 Ošetření výjimek

Ošetření nejrůznějších výjimek a chybových stavů, do kterých by se operační systém v žádném případě neměl dostat, patří mezi oblasti, kterým bychom se podle minulého odstavce měli věnovat při návrhu systému až někdy nakonec.

To je naprostá pravda; je však zapotřebí si uvědomit, že naopak při skutečné implementaci - tj. programování a ladění operačního systému - musíme tuto úlohu zpracovat jako vůbec jednu z prvních¹⁰. Důvod je nasnadě - čím luxusnější ošetření chybových stavů (spojené s hlášením příčiny chyby, případným výpisem operační paměti a podobně) máme k dispozici, tím se jednotlivé části operačního systému lépe ladí; pokud bychom si chtěli ošetření chybových stavů nechat na konec, budeme při ladění se stoprocentní jistotou každou chvíli v situaci, kdy je evidentně 'něco špatně', absolutně však netušíme, co by to mohlo být.

Ošetření výjimek je velmi důležité zvláště v prostředí preemptivního multitaskingu nebo dokonce multiprocesorových systémů s jejich masivním paralelismem; nejrůznější časově závislé chyby se jinak skutečně hledají velmi obtížně, a tradiční ladicí prostředky (jako je třeba krovování programu, využití breakpointů a v některých případech dokonce i ladicí výpisy) mohou dokonale selhat, protože změní časové souslednosti jednotlivých paralelních akcí.

¹⁰Nemusíme se snad ani výslovňě zmiňovat o tom, že tak rozsáhlý projekt jakým je operační systém musíme ladit po částech; s výhodou k tomu využijeme vrstvené struktury - dříve, než začneme implementovat služby některé vrstvy, musíme mít pokud možno co nejdokonaleji odladěny všechny služby všech nižších vrstev.

Operační systémy

4. Operační paměť

Správce paměti patří mezi nejdůležitější - i když ne nejkomplikovanější - moduly každého operačního systému. Dobře realizovaná správa paměti umožní ostatním prvkům operačního systému efektivní práci; je-li správce paměti naopak implementován nešikovným způsobem, nebude ani operační systém jako celek stát za mnoho.

Hlavním úkolem správce paměti je:

- Přidělovat operační paměť jednotlivým procesům, když si ji vyžádají;
- Udržovat informace o paměti, o tom, která část je volná a která přidělená (a komu);
- Zařazovat paměť, kterou procesy uvolní, opět do volné části;
- Odebírat paměť procesům, je-li to zapotřebí.

Kromě toho - umožňuje-li to technické vybavení počítače - by měl správce paměti zajistit i **ochranu paměti** - žádný proces by neměl mít přístup k paměti jiného procesu nebo operačního systému, jestliže mu to 'vlastník' paměti explicitně nepovolí.

V následujících odstavcích si nejprve popíšeme typické strategie správce paměti: přidělování paměti po blocích, přidělování paměti po blocích s dynamickým přemísťováním bloků a virtuální paměť se stránkováním na žádost. Je vhodné si uvědomit, že v konkrétních operačních systémech mohou být využity i jiné strategie, nebo strategie které jsou kombinací těch zde popsánych. Velmi triviální operační systém bez podpory multitaskingu může např. přidělovat prostě celou operační paměť aktivnímu programu. Některé operační systémy mohou využívat technické vybavení pro překlad adres (popsané v odstavci o virtuální paměti) jen pro dynamické přemísťování bloků paměti.

Kapitolu zakončíme podrobnějším rozborem různých možností zabezpečení ochrany paměti.

Operační systémy

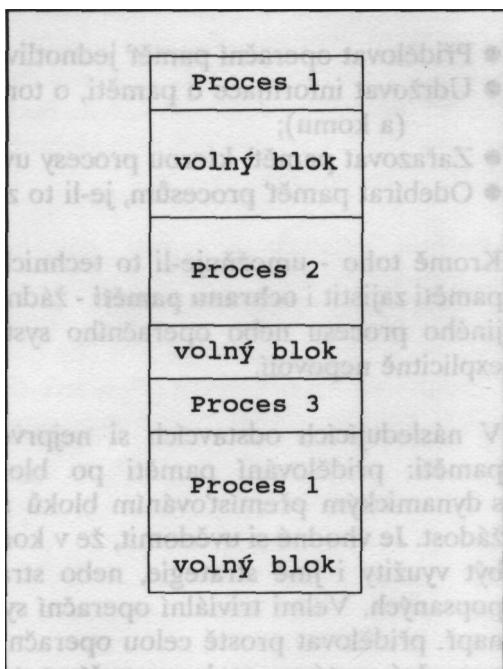
4.1 Přidělování paměti po blocích

Většina jednoduchých operačních systémů využívá právě tuto strategii - jako příklady můžeme jmenovat MS DOS nebo XINU. Její základní princip je velmi jednoduchý: každý proces musí vědět, kolik operační paměti bude potřebovat a musí si tuto paměť od operačního systému explicitně vyžádat. Operační systém - respektive správce paměti - takový požadavek buď splní přidělením bloku požadované velikosti, nebo zamítne, a v tom případě je úkolem procesu vzniklý problém nějak vyřešit (např. předčasným ukončením práce).

Na obr. 2 vidíme příklad operační paměti rozdělené na několik bloků. Operační paměť používají tři procesy; proces 1 má přiděleny dva bloky paměti, zbyvající dva bloky mají jen po jednom bloku.

Operační systém samozřejmě potřebuje vědět, který blok je přidělen kterému procesu. Navíc v průběhu práce, při stálém přidělování a uvolňování bloků paměti přestane být dosud nepřidělená paměť souvislá a stane se také skupinou bloků různé velikosti. Správce paměti musí udržovat také informace o těchto volných blocích, aby je mohl využít pro splňování požadavků procesů.

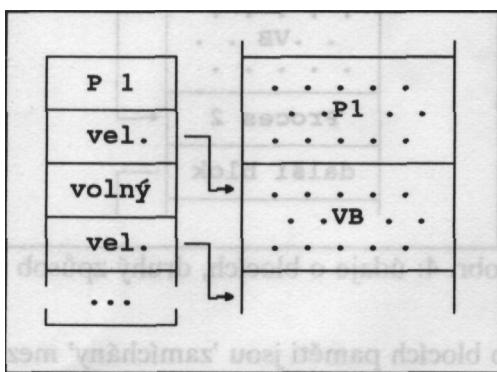
Z toho důvodu je nutné někde udržovat tabulky obsahující informace o přidělených i volných blocích operační paměti. Podíváme se proto podrobněji na oba častěji užívané způsoby udržování těchto informací.



obr. 2: přidělování bloců paměti

4.1.1 Informace o blocích

Správce paměti musí o každém bloku vědět nejméně dvě věci: jeho délku a jeho vlastníka. Je-li informace o vlastníkovi paměťového bloku prázdná, znamená to, že blok je volný a může být přidělen. Adresu dalšího bloku nemusíme explicitně uvádět, protože bloky na sebe samozřejmě v operační paměti navazují; adresu následujícího bloku tedy snadno spočítáme na základě délky a adresy aktuálního bloku. Správce paměti pak již potřebuje pouze znalost adresy prvního bloku, pak může snadno se spojovým seznamem bloků pracovat.



obr. 3: údaje o blocích, první způsob

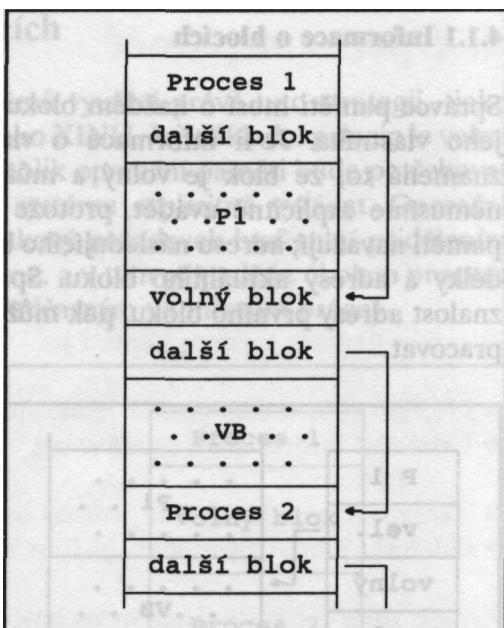
Správce paměti může samozřejmě tyto údaje udržovat někde uvnitř vlastních datových struktur (ilustraci tohoto způsobu vidíme na obr. 3); toto řešení však má jednu podstatnou nevýhodu: pro údaje musíme vyhradit dostatek paměti. Velikost potřebné paměti je však úměrná počtu přidělených bloků paměti; jejich počet však pochopitelně není možné určit předem. S jistotou samozřejmě víme, že bloků paměti bude méně, než je v paměti

k dispozici bytů; tolik prostoru pochopitelně není možné pro systémové tabulky vyhradit. Vyhradíme-li však méně místa, riskujeme, že ve výjimečném případě, kdy budou procesy požadovat velké množství malých bloků, vyhrazená paměť nepostačí.

Obvyklým a poměrně efektivním řešením tohoto problému je přidělit na žádost procesu vždy blok o několik bytů větší a do tohoto volného místa uložit informace o bloku samotném i o adrese bloku následujícího. V operační paměti tak vznikne spojový seznam volných i alokovaných bloků; správce paměti pak při každém požadavku tento seznam prochází a zpracovává jeho položky - jednotlivé bloky.

Malý úsek takového spojového seznamu vidíme na obr. 4. Vytečkované oblasti znázorňují vlastní paměťové bloky (tj. tu část bloků, kterou dostanou procesy přidělenou jako paměť), 'P1' označuje blok přidělený procesu číslo 1. 'VB' označuje volný blok.

Obrázek ilustruje i to, že tyto pomocné údaje bývají obvykle uloženy před vlastním blokem, a nikoli za ním. Proces, který požadoval přidělení paměti, tedy dostane ukazatel na začátek vytečkovaného prostoru a vůbec 'neví', že paměťový blok vlastně zabírá i několik bytů před tímto ukazatelem.



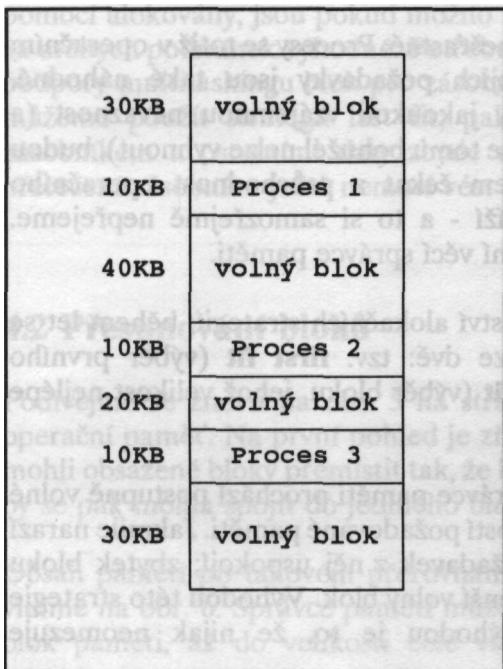
obr. 4: údaje o blocích, druhý způsob

Toto řešení má samozřejmě také svou nevýhodu. Systémové informace o blocích paměti jsou 'zamíchány' mezi paměťové bloky přidělené jednotlivým procesům. Jestliže technické vybavení počítače, na kterém operační systém provozujeme, neumožnuje zabezpečení ochrany paměti, je poměrně velké riziko, že některý chybný program systémové informace poškodí. V takovém případě prakticky není možné pokračovat v činnosti operačního systému, protože poškozením spojového seznamu bloků paměti je vyřazen vlastně celý správce paměti - uvědomme si, že bez informací z hlaviček bloků není možné ani bloky postupně projít; tím méně pak lze alokovat další paměť nebo uvolnit některý z přidělených bloků.

4.1.2 Fragmentace paměti

Systém přidělování paměti po blocích však má ještě jednu principiální nevýhodu, která se projeví i v případě, že žádný z provozovaných programů žádnou chybu neobsahuje. Dochází při něm totiž k tzv. **fragmentaci paměti**.

Již jsme se zmínili o tom, že při běhu operačního systému s přidělováním paměti po blocích se volný úsek paměti postupně rozpadá na více oddělených bloků. Správce paměti samozřejmě dokáže spojit dva volné bloky, které na sebe navazují, do jediného; to však nestačí - často dojde k situaci, kdy jsou dva volné bloky odděleny blokem přiděleným některému procesu.



obr. 5: fragmentace paměti

strategie, jak uvidíme hned v následujícím odstavci; nemůžeme se jí však nikdy zbavit úplně, pokud zachováme jednoduchý systém přidělování bloců paměti.

Správce paměti pak pochopitelně není schopen splnit požadavek na přidělení většího množství paměti, než je velikost největšího z volných bloků, bez ohledu na to, že celková velikost volné paměti (tj. součet všech volných bloků) může být i několikanásobně větší.

Tento problém ilustruje obr. 5, na kterém vidíme přidělené i volné bloky v operační paměti velikosti 150KB. Na první pohled je zřejmé, že dokud některý ze tří procesů, které mají přidělenou paměť, svůj blok paměti neuvolní, nemůže správce paměti nikomu přidělit více než 40KB. Rozsah volné paměti je přitom daleko větší - celých 120KB.

Fragmentaci můžeme do jisté míry omezit volbou vhodné alokační

4.1.3 Alokační strategie

Zamyslíme-li se nad problémem fragmentace paměti, napadne nás asi brzy velmi jednoduchá strategie, která by dokázala fragmentaci úplně zamezit: stačilo by uvolňovat bloky v opačném pořadí, než ve kterém byly přidělovány procesům, tj. jako první vždy uvolnit ten blok, který byl alokován jako poslední. Na první pohled je samozřejmě vidět, že tato strategie není v praxi použitelná - představme si, že poslední alokovaný blok bude zapotřebí ještě dlouho, zatímco všechny předchozí by již dávno mohly být volné. Je však vhodné si uvědomit, že zásadní nedostatek této strategie spočívá v tom, že by předepisovala jednotlivým procesům, jak se smí a nesmí chovat, navíc v závislosti jeden na druhém.

Takové závislosti jsou však mimořádně nešťastné. Procesy se totiž v operačním systému objevují zcela náhodně a jejich požadavky jsou také náhodné. Vneseme-li proto mezi různé procesy jakoukoli vzájemnou návaznost (a v dalších kapitolách uvidíme, že někdy se tomu bohužel nelze vyhnout), budou procesy nutně muset na sebe navzájem čekat a průchodnost operačního systému¹¹ se drastickým způsobem sníží - a to si samozřejmě nepřejeme. Alokační strategie proto musí být vnitřní věcí správce paměti.

Je možné vypracovat nepřeberné množství alokačních strategií; během let se však jako Životaschopné ukázaly pouze dvě: tzv. **first fit** (výběr prvního dostatečně velkého bloku), a tzv. **best fit** (výběr bloku, jehož velikost nejlépe odpovídá požadované velikosti).

Strategie first fit je zcela přímočará: správce paměti prochází postupně volné bloky a porovnává jejich velikost s velikostí požadované paměti. Jakmile *naráží* na blok, který je dostatečně velký, požadavek z něj uspokojí; zbytek bloku samozřejmě zůstane v seznamu jako menší volný blok. Výhodou této strategie je jednoduchost a rychlosť; její nevýhodou je to, že nijak neomezuje fragmentaci.

¹¹Průchodnost operačního systému určuje - přibližně řečeno - míru toho, jak moc operační systém v průměrném případě 'zabrzdí' proces.

Při strategii best fit naproti tomu správce paměti projde všechny volné bloky, a z těch, které jsou dostatečně velké, vyhledá nejmenší. Z něj pak již standardním způsobem uspokojí požadavek. Účelem strategie best fit je zachovat velké volné bloky co nejdéle 'nerozdrobené' a dělit především ty menší. Díky tomu se fragmentace skutečně trochu sníží.

Pro upmost dodejme, že v literatuře se často můžeme setkat i s pojmem 'strategie **last fit**'. Při této strategii se procházejí všechny volné paměťové bloky podobně jako při strategii best fit, ale požadavek se uspokojí z posledního bloku, který je dostatečně velký. Z hlediska fragmentace paměti je tato strategie dokonale rovnocenná strategii first fit, je však o něco málo složitější a o dost pomalejší. Používá se jen proto, že paměťové bloky, které jsou její pomocí alokovány, jsou pokud možno na co nejvyšších adresách; to může být za určitých podmínek výhodné. Potřebujeme-li např. v operačním systému bez podpory multitaskingu blok pro zásobník (který obvykle roste směrem dolů), můžeme použít strategii last fit; pak je velká pravděpodobnost, že mezi zásobníkem a ostatními daty zbyde nějaký úsek volné paměti a případné přetečení zásobníku ještě nemusí vést ke zhroucení programu¹².

4.2 Přesunování bloků

Podívejme se znovu na obr. 5 na straně 37, který ukazuje fragmentovanou operační paměť. Na první pohled je zřejmé, že by nám pomohlo, kdybychom mohli obsazené bloky přemístit tak, že by ležely těsně vedle sebe. Volná paměť by se pak mohla spojit do jediného bloku zabírajícího celých 120KB.

Obsah paměti po takovém přerovnání bloků - říkáme mu často **setřásání** - vidíme na obr. 6. Správce paměti může bez problémů přidělit libovolně velký blok paměti, až do velikosti celé volné paměti, tj. do 120KB. Problémů

¹²V takovém případě se samozřejmě poškodí systémové informace o blocích paměti, jsou-li uloženy mezi bloky podle druhého principu, který jsme poznali v odstavci 4.1.1, a v důsledku toho se dříve či později zhroutí správce paměti. Programy pod operačním systémem bez podpory multitaskingu však často mohou dokončit svou práci i ve chvíli, kdy část operačního systému není v pořádku.

v odstavci 4.3); nebudeme proto pravděpodobně automatické přesunování bloků používat¹⁴.

Procesy, které pracují pod operačním systémem, jehož správce paměti využívá automatického přesunování bloků, proto obvykle musí splňovat určité konvence, které zajistí, že přesun bloku paměti procesu 'neuskodí'. Máme k dispozici v zásadě tři možnosti řešení tohoto problému:

- Procesy musí pro přístup do paměti dodržet určité adresovací konvence, které zajistí přemístitelnost bloku (typicky se bude jednat o povinné bázování vhodným registrem, konkrétní řešení samozřejmě závisí na adresovacích módech použitého procesoru).
- Procesy musí dodržovat určité konvence na vyšší úrovni - na úrovni vlastního algoritmu. Proces může být např. povinen před přístupem do paměti zjistit dotazem u správce systému momentální adresu bloku a pak po celou dobu práce s tímto blokem paměť 'zamknout' - tj. zakázat jeho přemístění.
- Operační systém může procesu zaslat zprávu ve chvíli, kdy blok paměti přemísťuje. Proces pak na základě této zprávy přepočítá všechny své ukazatele, které do bloku míří, na správné hodnoty podle nové adresy bloku.

Každá z možností má své výhody i nevýhody. První metoda je nepochyběně nejlepší, závisí však do značné míry na technickém vybavení - u některých procesorů může velmi podstatným způsobem redukovat možnosti adresování v paměťových blocích. Musí jí být také přizpůsoben použitý překladač při tvorbě procesů ve vyšším jazyce.

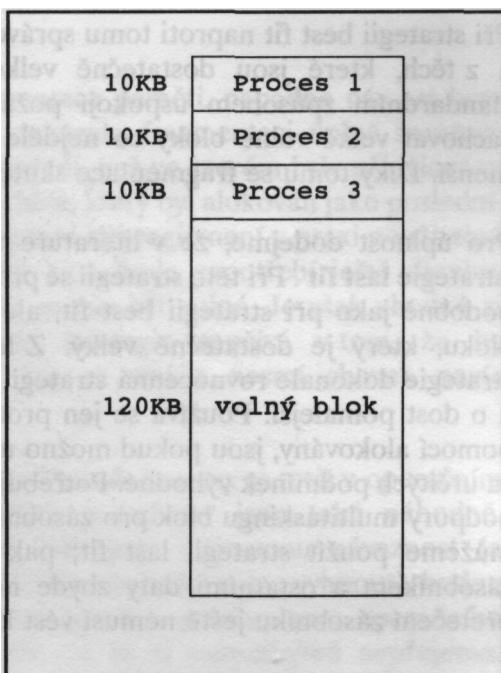
Druhý způsob klade nejmenší nároky na správce paměti, o to více práce však mají programátoři, kteří vytvářejí procesy. Podle názoru autora této knihy se jedná o nejhorší variantu; autor totiž vytváří programy pro systém Apple Macintosh, který právě tohoto systému využívá.

¹⁴ Přesněji řečeno, nebudeme používat samotné automatické přesunování bloků. Systém virtuální paměti bez stránkování musí někdy bloky v paměti přesunovat také.

s fragmentací jsme se tedy zbavili; bohužel však za to musíme zaplatit poměrně značnou cenu: správce paměti musí paměťové bloky přesunovat, což je samozřejmě Časově náročné (zvláště jedná-li se o větší úseky paměti), a jednotlivé procesy musí počítat s tím, že přidělená paměť nemusí zůstat stále na stejně adrese. Jak uvidíme, jedná se o relativně složitý problém.

Přesunování bloků samo o sobě samozřejmě zabere nějaký čas, to však není zase příliš tragické. Na řadě počítačů má správce paměti k dispozici specializovaný mikroprocesor (nejčastěji jej nazýváme **blitter**¹³), který zajišťuje velmi rychlé přesuny dat v paměti, takže časová ztráta není tak velká. Ani na systémech, kde není blitter k dispozici, nehrází vážnější problémy, protože setrásání paměti je zapotřebí provádět pouze ve chvíli, kdy některý proces požaduje příliš velký blok - a k tomu nedochází tak často.

Daleko horším problémem je automatické přesunování bloků z hlediska procesů. V ideálním případě by samozřejmě mělo být přesunování bloků pro procesy zcela transparentní; to však lze zajistit jen tehdy, když technické vybavení počítače umožňuje tzv. překlad adres. Máme-li však k dispozici překlad adres, můžeme implementovat virtuální paměť, která nejen zamezuje fragmentaci, ale přináší i řadu dalších výhod (seznamíme se s nimi



obr. 6: setřesená paměť

¹³Název 'blitter' vznikl spojením a zkrácením slov 'bit block transfer' (přesun bitových bloků). Blitter je totiž obvykle schopen pracovat nejen s celými byty, ale i s jejich částmi - tj. s byty. To je proto, že nejvíce práce blitter zastane při provádění grafických operací - tj. vlastně při přesunování údajů ve videopaměti).

Třetí metoda není příliš vhodná pro běžné aplikace, dobré však poslouží jako doplněk první metody pro programy, které z nějakých důvodů musí nutně pracovat s absolutními adresami¹⁵ (typicky se jedná o ovladače periferních zařízení). V případě jejího využití je však nutné vhodným způsobem zvolit způsob, kterým správce paměti procesu oznámí přesunutí bloku paměti - klasický aparát zpráv, se kterým se seznámíme později, není vhodný, protože zpráva by mohla přijít příliš pozdě (tedy po pokusu o práci s pamětí na původním místě).

Příkladem operačního systému, jehož správce paměti využívá přesunování přidělených bloků, může být systém počítačů Macintosh nebo EPOC. Systém Macintosh využívá výhradně druhé metody - říkejme jí **kooperativní**, protože procesy musí s operačním systémem kooperovat, operační systém EPOC pracuje s kombinací první a třetí metody - nazveme ji **transparentní**, protože běžné procesy¹⁶ se jí nemusí zabývat.

Dodejme, že správce paměti může volit jednu ze dvou variant: bloky mohou být vyhrazovány libovolným způsobem tak, jak jsme předpokládali až dosud, nebo lze alokovat bloky jednoho procesu 'vedle sebe', bez přerušení 'cizím' blokem. V takovém případě budeme celou skupinu bloků jednoho procesu nazývat sekce. Oba operační systémy, o kterých jsme se zmínili, pracují se sekczemi, využívají tedy druhou variantu. Podívejme se nyní na správce paměti v obou systémech podrobněji.

¹⁵Při použití kterékoli metody samozřejmě musí procesy pracovat s adresami relativními: jakákoli adresa uvnitř paměťového bloku je určena offsetem v bloku a proměnnou adresou bloku samotného.

¹⁶Tj. takové, které využívají pouze standardních služeb a v ideálním případě jsou naprogramovány ve vyšším jazyce, jehož překladač potřebné konvence dodržuje automaticky (automatické dodržení konvencí kooperativní metody není možné).

4.2.1 Kooperativní přesunování bloků

V tomto odstavci popíšeme podrobněji relevantní služby konkrétního operačního systému, totiž operačního systému počítačů Apple Macintosh. To však není na úkor obecnosti, protože jakýkoli jiný operační systém, který využívá obdobné strategie správy paměti, musí nutně fungovat obdobným způsobem - jediné, co by se tedy vlastně mohlo změnit, jsou jména konkrétních systémových služeb. Na druhé straně je nutné si uvědomit, že tento odstavec nemůže sloužit jako učebnice programování Macintoshe - řada vlastností jeho správce paměti, které nejsou na této úrovni důležité, je zde opomíjuta.

Operační systém počítačů Apple Macintosh zavádí pro práci s pamětí nový prvek - tzv. **handle**. Handle je vlastně dvojitý ukazatel; chceme-li tedy pracovat s blokem paměti obsahujícím třeba celá čísla, deklarujeme

```
int **int_handle;
```

Podobně můžeme samozřejmě vytvořit handle na libovolný typ, včetně složených typů. Handle můžeme i přetypovat; práce s ním je velmi podobná práci s normálním ukazatelem, až na to, že potřebujeme vždy 'o hvězdičku víc' - tj. kdekoli bychom použili obyčejný ukazatel, použijeme namísto toho výraz '(*handle)'.

Jestliže máme handle deklarovaný, musíme si vyžádat od operačního systému přidělení bloku paměti. Použijeme k tomu systémovou funkci 'NewHandle', jejímž parametrem je velikost požadovaného bloku. Funkce vrátí buď handle přiděleného bloku, nebo nulovou hodnotu jako informaci, že není k dispozici dostatek paměti¹⁷.

Pak můžeme s pamětí volně pracovat již popsaným způsobem. Chceme-li tedy např. vynulovat první číslo v bloku paměti, můžeme zapsat

```
**int_handle=0;
```

¹⁷Nebo dost velký blok - jak uvidíme dále, může ve skutečnosti u operačního systému Apple Macintosh k fragmentaci dojít, i když je oproti správě paměti bez přesunování bloků silně omezena.

chceme-li vynulovat celý blok paměti, můžeme zapsat

```
for (i=0;i<VELIKOST_BLOKU;i++)
    (*int_handle)[i]=0;
```

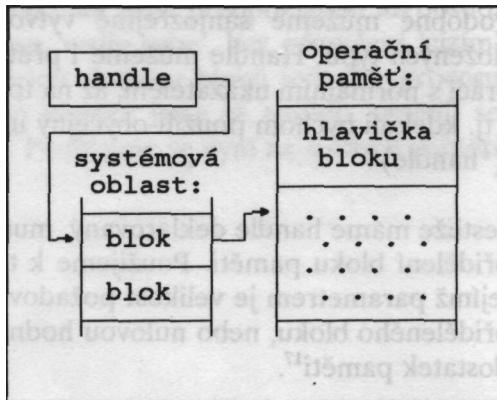
Zde však již narázíme na první problém, který handle přináší. Zkušený programátor totiž cyklus z minulého příkladu napíše jinak, daleko efektivnějším způsobem:

```
{
    int *ptr=*int_handle,*end=*int_handle+VELIKOST_BLOKU;
    while (ptr<end)
        *ptr++=0;
}
```

V tom se však skrývá nebezpečí, pro jehož pochopení si musíme nejprve vysvetlit, co to vlastně handle ve skutečnosti je.

Ilustrace na obr. 7 ukazuje, že operační systém počítačů Apple Macintosh vlastně kombinuje obě metody z odstavce 4.1.1 - handle ukazuje do tabulky v systémové oblasti, ale informace o blocích jsou uloženy v blocích samých.

Tabulka v systémové oblasti slouží právě k přemístování bloců: správce paměti blok přemístí a jeho novou adresu uloží na patřičné místo v této tabulce. Jestliže pak proces přistupuje k paměti pouze přes dvojitou dereferenci handle¹⁸, je vše v pořádku. Jakmile si však zapamatujeme skutečnou adresu bloku nebo uvnitř něj - jak tomu bylo právě v minulém



obr. 7: jak funguje handle

¹⁸Vzhledem k obtížné tvorbě tvarů používáme pojmenování 'handle' v této knize jako slovo nesklonné. Programátoři jej však samozřejmě veselé skloňují: "vytvořil jsem hendl, tomu hendlu jsem přiřadil blok a pak jsem s tím hendlem něco zvoral".

příkladě - riskujeme, že správce paměti blok přemístí a náš ukazatel bude ukazovat někam úplně jinam, v lepším případě do nepřiřazené paměti, v horším do bloku, který patří někomu úplně jinému (v odstavci 4.5 uvidíme, že operační systém Macintoshe nedisponuje žádnou ochranou paměti).

Na první pohled by se zdálo, že stačí používat důsledně handle a dvojitou dereferenci a všechny problémy zmizí. To je samozřejmě pravda, bohužel pravda nerealizovatelná. První problém spočívá v tom, že dvojitá dereference je přece jen výpočetně náročná, a nemůžeme si proto dovolit ji používat ve složitějších výpočtech a v nejvnitřnejších cyklech. Druhý - a při stále se zvyšujícím výkonu počítačů důležitější - problém je v tom, že řada funkcí a služeb požaduje ukazatel na data, a nikoli handle. Všechny tyto funkce samozřejmě jsou (nebo by alespoň měly být) naprogramovány tak, aby jejich vlastní průběh byl bezpečný; to však bohužel nestačí. Představme si třeba následující volání funkce 'BlockMove', která přesunuje data v paměti:

```
BlockMove(*int_handle,*int_handle+10,10*sizeof(int));
```

Na první pohled se zkopíruje obsah prvních deseti čísel v bloku do druhých deseti čísel. Překladač však musí nejprve vyhodnotit argumenty funkce, uložit je na zásobník a pak funkci zavolat. Co když správce paměti přesune blok v době mezi vyhodnocením prvního a druhého parametru?

Operační systém počítačů Macintosh proti tomuto nebezpečí bojuje dvěma způsoby:

- Kooperativní podstata tohoto operačního systému sama o sobě znemožňuje správci paměti cokoli dělat, není-li zavolána některá ze systémových služeb (již to stačí k zabránění problémů v obou konfliktních příkladech, které jsme uvedli). Operační systém navíc povoluje setřásání paměti jen při volání těch systémových služeb, které to nutně potřebují; jestliže tedy proces žádnou z nich nevolá, může si být jist, že všechny bloky paměti zůstávají na původním místě.
- Proces má k dispozici služby 'HLock' a 'HUnlock'. První z těchto služeb 'zamkne' handle - to znamená, že správce paměti nesmí blok určený pomocí zamčeného handle přesunovat. Proces tedy rnůže zamknout

blok paměti a pracovat s ním zcela standardním způsobem pomocí běžných ukazatelů; po ukončení práce jej opět odemkne služba 'HUnlock'.

Díky tomu je vlastně možné handle v praxi vůbec používat; musíme si však uvědomit několik velmi závažných důsledků, které více či méně degradují výkon celého systému a snižují tak výhody plynoucí z automatického přemísťování bloků:

- Možnost setřásat paměť jen při volání některých funkcí do jisté míry snižuje výkon celého systému. Paměť totiž pak musí být setřesena ve chvíli, kdy je to opravdu nutně zapotřebí, a všechny procesy musí čekat. Kdyby mohl operační systém setřásat paměť kdykoli, mohl by využívat volných chvil (např. když všechny procesy čekají na akci uživatele nebo na ukončení práce s diskem) k částečnému setřesení paměti; žádný čas by tak nebyl ztracen a vynucené úplné setřesení by buď nenastalo vůbec, nebo alespoň po mnohem delší době.
- Ještě horší důsledky má možnost zamykání bloků paměti. Zamčený blok není možné setřást vůbec; pokud tedy některý proces své bloky zamyká na dlouhou dobu, narůstá fragmentace stejně jako na systému bez automatického přesunování bloků.
- Daleko nejvýznamnějším důsledkem popsaných vlastností správy paměti je to, že správa paměti omezuje multitasking (nebo naopak je omezována multitaskingem). Uvědomme si, že proces 'si je jist', že se žádný z jeho bloků paměti nepřemístí, pokud proces nevolá žádnou ze skupiny 'nebezpečných' služeb. To ale znamená, že správce paměti nemůže paměť setřást ani v případě, že by to některý jiný proces nutně potřeboval!

Tento problém lze řešit několika způsoby - buď omezením multitaskingu (tak, že žádný jiný proces nesmí běžet nebo alespoň volat služby, které by vedly k setřesení paměti, dokud první proces

předpokládá, že jeho bloky zůstanou beze změny¹⁹⁾, nebo omezením setřásání paměti (a samozřejmě tedy růstem fragmentace). Operační systém počítačů Apple Macintosh dělá obojí; kromě omezení multitaskingu navíc setřásá pouze bloky patřící každému procesu uvnitř jeho sekce, zatímco sekce v rámci celé operační paměti nesetřásá vůbec. To samozřejmě vede k neomezené fragmentaci 'mezi procesy'.

Jakkoli má kooperativní systém přemísťování bloků paměti řadu nevýhod, se kterými jsme se právě seznámili, je daleko lepší než jednoduchý systém přidělování bloků bez přemísťování - alespoň v tom smyslu, že daleko lépe využívá operační paměť počítače, z hlediska programátorů je kooperativní systém velmi složitý a v kombinaci s neexistující ochranou paměti i nebezpečný.

4.2.2 Transparentní přesunování bloků

Podobně, jako jsme pro ilustraci kooperativního systému využili operační systém počítačů Apple Macintosh, využijeme tentokrát operační systém EPOC.

Zatímco kooperativní systém Macintoshe setřásá pouze bloky paměti uvnitř sekcí, patřících jednotlivým procesům, pracuje EPOC právě opačně: starost o fragmentaci uvnitř sekcí ponechává procesům, setřásá však tyto sekce v rámci celé operační paměti²⁰⁾. Správce paměti v EPOCu navíc může dynamicky měnit velikost jednotlivých sekcí - zvětší je, potřebuje-li proces více paměti, a zmenší je kdykoli je to možné (tj. kdykoli je uvolněn poslední blok v sekci). Operační paměť je tedy využita velmi dobře.

"Jinými slovy, operační systém může 'nechat chvíli běžet' jiné procesy pouze v rámci těch služeb, které mohou setřásat paměť.

²⁰⁾Pro úplnost dodejme, že proces si může vyžádat i přidělení bloku, který leží mimo jeho 'sekci'; takový blok pak je automaticky setřásán společně se sekci jednotlivých procesů.

EPOC využívá architektury mikroprocesorů řady Intel 80x86, které jsou základem počítačů PSION²¹ pro transparentní přesunování bloků. Tyto mikroprocesory jsou navrženy tak, že každý přístup do paměti je automaticky bázován jedním z tzv. segmentových registrů; EPOC tedy může přesunout sekci paměti a patřičným způsobem změnit obsah odpovídajícího segmentového registru. Proces nemusí vůbec o ničem vědět - jakýkoli přístup do paměti bude díky automatickému bázování jistě veden na správné místo.

Proces samozřejmě musí dodržet také řadu konvencí; na rozdíl od konvencí kooperativní metody je však jejich dodržení poměrně snadné a při psaní programů ve vyšším jazyce je možné jejich dodržení zautomatizovat na úrovni překladače. Všechny konvence by se daly vyjádřit stručným imperativem 'nepoužívat segmentové registry'; EPOC však není natolik kategorický:

- Segmentový registr nesmí obsahovat nic jiného, než adresu kterékoli sekce. Je tedy možné ukládat do segmentových registrů adresy sekcí, získané od operačního systému nebo třeba od jiných procesů (při sdílení paměti).
- Adresy sekcí naopak nesmí být ukládány jinde než v segmentových registrech. Adresy sekcí v segmentových registrech totiž správce paměti při setřásání automaticky upravuje, zatímco adresy uložené např. v jiném registru by se mohly stát neplatnými.

Všechny běžné programy tyto konvence snadno dodrží a nepotřebují je žádným způsobem obcházet²²; mohou proto pracovat s pamětí zcela standardním

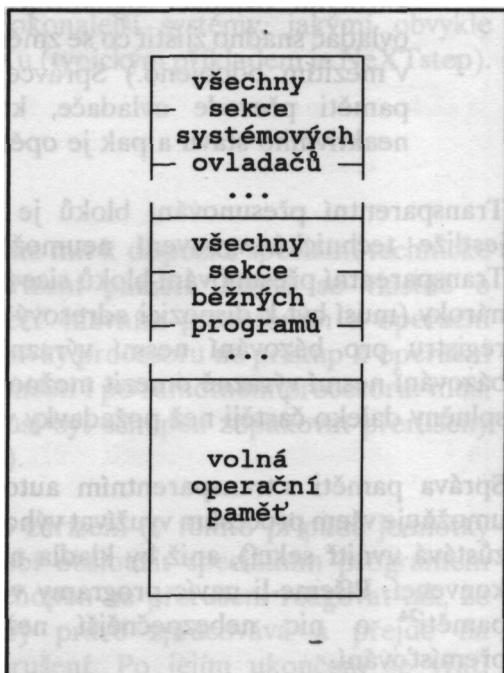
²¹Přesněji řečeno, jedná se o mikroprocesory V30 firmy NEC, které mají oproti svému vzoru (Intel 8086) řadu drobných výhod - např. plnou instrukční sadu reálného módu práce mikroprocesoru Intel 80286.

²²Ti, kdo znají mikroprocesory řady Intel 80x86 pravděpodobně namítou "a co když proces potřebuje větší data než 64KB?". Odpověď je jednoduchá: nestává se to. Přpomeňme, že EPOC je speciální operační systém vytvořený pro obsluhu kapesních počítačů; ty mají řadu specifických vlastností. Jednou z nich je také využívání elektronických obvodů na místě disket a pevných disků stolních počítačů; EPOC s tím počítá a podporuje zpracování dat přímo na 'disku' (který je na to dostatečně rychlý), zatímco operační paměť se používá nejvýše jako buffer nebo cache. Kromě toho pro nutné případy nabízí EPOC i služby pro práci s většími paměťovými bloky.

způsobem pomocí ukazatelů a setřásání je pro ně skutečně dokonale transparentní.

Existuje však skupina programů - nejčastěji se jedná o ovladače periferních zařízení - které pro komunikaci potřebují znát skutečnou adresu svých dat v paměti, nezávisle na obsahu segmentových registrů. Často tyto ovladače musí při případném přemístění bloku provést i řadu dalších akcí - změnu svých vnitřních proměnných, preprogramování registrů vstupních a výstupních zařízení a podobně.

EPOC se proto k systémovým ovladačům chová trochu jiným způsobem, než k ostatním programům:



obr. 8: paměť v EPOCu

- Sekce paměti ovladačů se alokují na nižších adresách než sekce paměti běžných programů (podívejme se na obr. 8, který schematicky znázorňuje rozdělení paměti v EPOCu). Díky tomu je jen zřídkakdy zapotřebí přemísťovat sekce, které patří systémovým ovladačům - pouze tehdy, když je některý z ovladačů instalován nebo odstraněn (nebo když alokuje nebo uvolňuje paměť).
- Každý ovladač musí být schopen na vyžádání operačního systému přejít do neaktivního stavu (v terminologii EPOCu tzv. 'hold'). Dalším příkazem operačního systému pak je příkaz pro ukončení neaktivního stavu ('resumé'); jednou z řady informací, které jsou součástí tohoto příkazu, je i nové umístění sekce ovladače v operační paměti. (Tento aparát je univerzální a operační systém jej využívá v řadě případů - např. vypneme-li počítač, převede EPOC nejprve všechny ovladače do neaktivního stavu a pak teprve skutečně vypne. Při zapnutí pak-

každému ovladači signalizuje ukončení neaktivního stavu²³, takže ovladač snadno zjistit co se změnilo - ovládané zařízení např. mohlo být v mezitím odpojeno.) Správce paměti pak prostě před setrásáním paměti převede ovladače, kterých se setřesení bude týkat, do neaktivního stavu a pak je opět aktivuje.

Transparentní přesunování bloků je ideami mechanismus správy paměti, jestliže technické vybavení neumožňuje implementaci virtuální paměti. Transparentní přesunování bloků sice na technické vybavení také klade určité nároky (musí být k dispozici adresový režim s bázováním, vyhrazení určitého registru pro bázování nesmí výrazně omezit využití procesoru, povinné bázování nesmí výrazně omezit možnosti adresace), tyto požadavky jsou však splněny daleko častěji než požadavky virtuální paměti.

Správa paměti s transparentním automatickým přesunováním bloků navíc umožňuje všem procesům využívat výhody omezené fragmentace (fragmentace zůstává uvnitř sekcí), aniž by kladla na programátora zátěž komplikovaných konvencí. Příseme-li navíc programy ve vyšším jazyce, není ani bez ochrany paměti²⁴ o nic nebezpečnější než normální přidělování bloků bez přemístování.

4.3 Virtuální paměť

Za nejdokonalejší strategii správy paměti lze považovat tzv. virtuální paměť. V tomto odstavci se postupně seznámíme s nezbytnými předpoklady pro implementaci virtuální paměti i s jejím mechanismem, od nejjednodušších variant (jako např. virtuální paměť operačního systému 7.x počítačů Apple

²³Počítače PSION při vypnutí pouze přejdou do stavu, kdy 'nepočítají'; obsah operační paměti, registrů a stav všech procesů však zůstane beze změny. Tato funkce sice klade nemalé nároky na korektní funkci operačního systému a všech ovladačů (zatímco je počítač 'vypnuto', může se měnit konfigurace ...), kvalitní kapesní počítač se však bez ní pochopitelně nemůže obejít.

²⁴Poznamenejme, že počítače PSION s operačním systémem EPOC jsou ochranou paměti vybaveny.

Macintosh) až po zatím zřejmě nejdokonalejší systémy, jakými obvykle disponují operační systémy na bázi UNIXu (typickým příkladem je NeXTStep).

43.1 Potřebné technické vybavení

Pro implementaci virtuální paměti musíme mít k dispozici speciální technické vybavení, kterému se říká **jednotka řízení paměti**. Jedná se vlastně o specializovaný procesor, který stojí 'mezi' hlavním procesorem a operační pamětí počítače a sám zpracovává požadavky procesoru na přístup k operační paměti²⁵. Navíc požadujeme určité schopnosti i po samotném procesoru: musí být schopen zpracovávat přerušení a musí být schopen zopakovat přerušený přístup do paměti (nebo celou instrukci).

- **Přerušení** je signál nějakého vnějšího zařízení (v tomto případě jednotky řízení paměti), který je zapotřebí obsloužit speciálním programem ihned. Procesor tedy musí být schopen na přerušení reagovat tak, že dočasně přeruší program, který právě zpracovává a přejde na zpracování obslužné rutiny přerušení. Po jejím ukončení se vrátí k přerušenému programu.
- **Zopakování přístupu do paměti** je základem celé virtualizace. Pokusí-li se totiž mikroprocesor pracovat s pamětí, která neexistuje (je pouze virtuální, doslova zdánlivá), odpoví na to jednotka řízení paměti přerušením. Správce paměti ve spolupráci s jednotkou řízení paměti přerušení obslouží tím způsobem, že onu neexistující paměť doplní (zanedlouho uvidíme jak). Procesor pak musí být schopen zopakovat pokus o přístup do paměti - která již nyní existuje - aby mohl přerušený program pokračovat v práci.

Jednotka řízení paměti musí být schopna minimálně zajistit ochranu paměti, jinak není možné virtuální paměť vůbec implementovat. Velmi důležitou funkcí jednotky řízení paměti je i překlad adres - bez něj by byl systém virtuální paměti nesmírně těžkopádný a v praxi nepoužitelný. Pro rozumnou efektivitu

²⁵Moderní mikroprocesory jsou obvykle integrovány i s jednotkou řízení paměti na jedený čip.

Operační systémy

virtuální pamětí však obvykle požadujeme po jednotce řízení paměti ještě jednu službu - stránkování.

- **Ochrana paměti** se budeme podrobněji zabývat v odstavci 4.5; zatím pouze určíme, na jakém principu pracuje:

Správce paměti předá jednotce řízení paměti vhodným způsobem informace ze svých tabulek - tj. který úsek paměti patří kterému procesu. Správce procesů, o kterém budeme hovořit v další kapitole, zajistí, aby správce paměti 'věděl', který proces právě běží; správce paměti předá i tuto informaci jednotce řízení paměti.

Při zpracování každého požadavku na přístup k operační paměti pak jednotka řízení paměti ověří, má-li aktivní proces právo s tímto úsekom paměti pracovat. Jestliže tomu tak není, jednotka řízení paměti procesoru přístup k paměti neumožní a namísto toho vyvolá výjimku. Správce paměti, který výjimku obsluhuje, pak může zajistit vše potřebné.

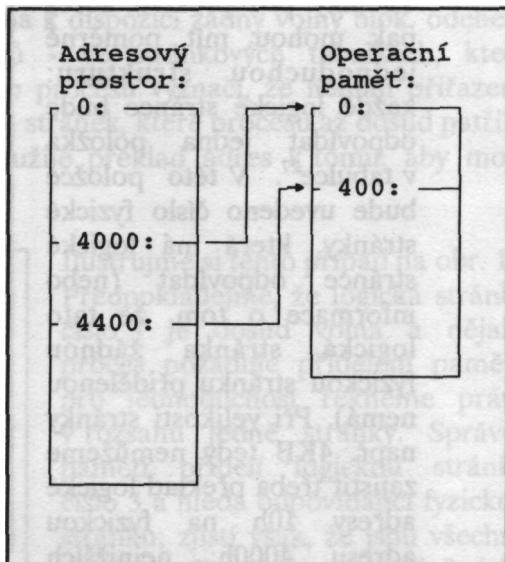
- **Překlad adres** umožňuje (přibližně řečeno) přiřadit libovolnému úseku operační paměti libovolné adresy. Správce paměti může vytvořit tabulky, které určují nejen komu který blok paměti patří a jak je velký, ale i na které adrese v operační paměti má ležet.

Překlad adres nám umožní používat celý adresový prostor procesoru s tím, že kterékoli jeho části můžeme přiřadit skutečnou operační paměť.

Můžeme mít tedy například blok velikosti 1KB, který pro program leží na adrese 4000h - program jej může naprosto běžným způsobem používat; na adrese 4002h nalezne třetí byte bloku a podobně. Správce paměti však pomocí tabulek oznámil jednotce řízení paměti, že tento blok má v operační paměti ležet na adrese 0. Jednotka řízení paměti pak bude pracovat tak, že kdykoli zachytí požadavek na přístup k některé adrese v rozmezí 4000h až 43FFh, předá jej operační paměti, ale nejprve od adresy odečte hodnotu 4000h. Tuto situaci ilustruje obr. 9.

Abychom mohli dobře rozlišit obě adresy, které se účastní překladu adres, budeme adresy, kterou používá program a která leží v adresovém prostoru procesoru říkat **logická adresa**. Adrese v operační paměti, po překladu adres, budeme naproti tomu říkat **adresa fyzická**.

V minulém příkladu se tedy logická adresa 4000h přeložila na fyzickou adresu 0h, logická adresa 4002h na fyzickou adresu 2h a podobně.



obr. 9: překlad adres

- **Stránkování** na první pohled vypadá jako omezení: požaduje, aby bloky, které se účastní překladu adres, byly složeny z tzv. stránek pevné velikosti.

Adresový prostor procesoru se tak rozpadá na řadu stránek - první z nich začíná na (logické) adrese 0, druhá na adrese <velikost stránky>, třetí na adrese <2*velikost stránky> a tak dále. Těmto stránkám budeme říkat **logické stránky**, podobně jako adresy v tomto prostoru nazýváme logickými adresami.

Obdobným způsobem rozdělíme na jednotlivé stránky operační paměť - první z nich bude začínat na (fyzické) adrese 0, druhá na adrese <velikost stránky>, třetí na adrese <2*velikost stránky> a tak dále. Těmto stránkám budeme samozřejmě říkat **fyzické stránky**²⁶.

starší literatuře, zvláště z prostředí střediskových počítačů, se často setkáme s alternativním názvem **stránkový rám**.

Operační systémy

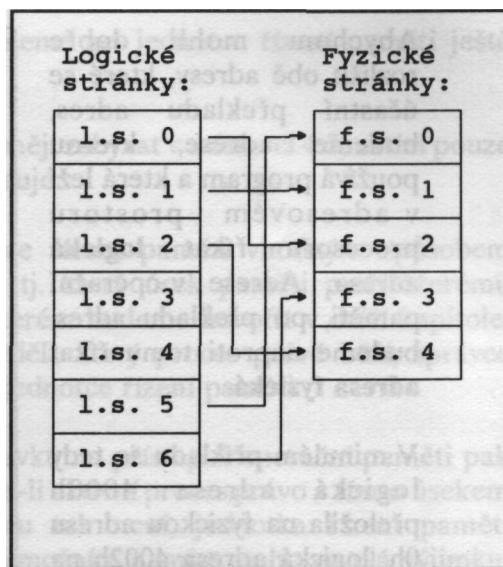
Tabulky pro překlad adres pak mohou mít poměrně jednoduchou strukturu: každé logické stránce bude odpovídat jedna položka v tabulce²⁷. V této položce bude uvedeno číslo fyzické stránky, která má logické stránce odpovídat (nebo informace o tom, že tato logická stránka žádnou fyzickou stránku přidělenou nemá). Při velikosti stránky např. 4KB tedy nemůžeme zajistit třeba překlad logické adresy 10h na fyzickou adresu 4000h; nejnižších dvacát bitů logické i odpovídající fyzické adresy

musí být totožných. Systém překladu adres se díky tomu výrazně zjednoduší; stránkování přinese i další výhody, které poznáme níže.

Příklad vzájemného přiřazení stránek vidíme na obr. 10. První logické stránce je přiřazena fyzická stránka číslo 2, logická stránka číslo 3 nemá vůbec žádnou fyzickou stránku přiřazenu apod. Připomeňme si ještě jednou: fyzické stránky reprezentují paměť, zatímco logické pouze adresový prostor.

Základy

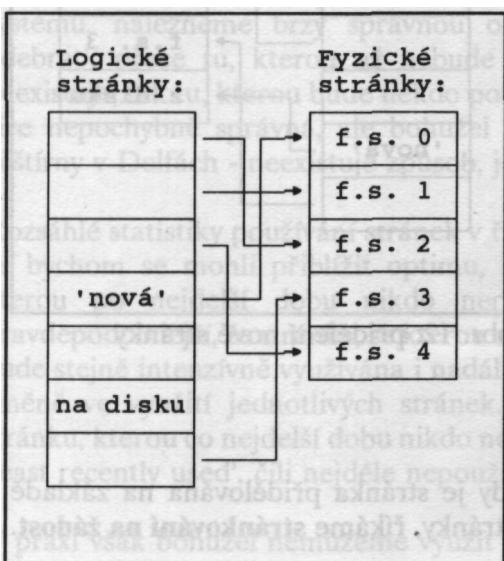
Základní princip virtuální paměti je poměrně jednoduchý. Správce paměti má k dispozici odkládací prostor na nějakém rychlém velkokapacitním paměťovém médiu, zpravidla na pevném disku. Dojde-li k situaci, že správce potřebuje



obr. 10: překlad adres / stránkování

²⁷Ve skutečnosti to není možné pro příliš vysoký počet logických stránek, stránkové tabulky proto bývají víceúrovňové. Na úrovni abstrakce, na které se zatím pohybujeme, to však není důležité.

někomu přidělit paměť a přitom nemá k dispozici žádný volný blok, odebere prostě paměť některému z procesů - ve stránkových tabulkách, které odpovídají logickým stránkám tohoto procesu vyznačí, že nemají přiřazené žádné fyzické stránky, obsah fyzických stránek, které procesu až dosud patřily, uloží do odkládacího prostoru a využije překlad adres k tomu, aby mohl stránky přidělit novému žadateli.



obr. 11: odebrání stránky

paměti, že požadovaná adresa neexistuje - této situaci budeme říkat výpadek stránky - a vyvolá výjimku, kterou obslouží správce paměti. Ten reaguje stejně, jako kdyby si proces vyžádal přidělení nové paměti: nalezne pro něj nějaké volné fyzické stránky (nejsou-li volné, odebere je již známým způsobem někomu jinému), využije překlad adres k tomu, aby tyto stránky připojil k logickým stránkám procesu a zapíše do nich údaje z odkládacího prostoru. Proces pak může pokračovat stejným způsobem, jako by se s jeho pamětí nikdy nic nedělo.

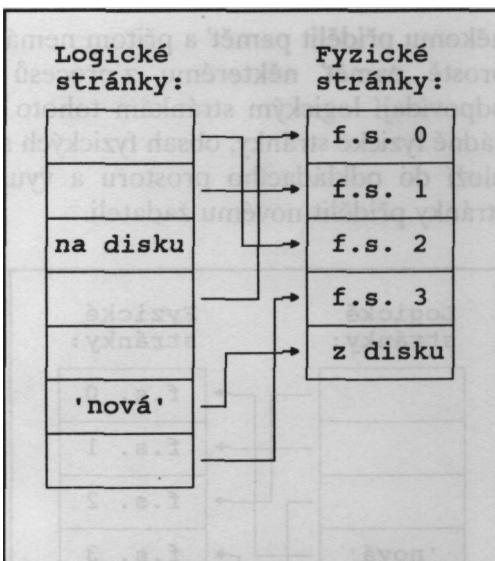
Vraťme se opět k našemu příkladu: původní vlastník fyzické stránky číslo 0 (pracoval s ní pochopitelně prostřednictvím logických adres z logické stránky

Ilustrujme si tento případ na obr. 10. Předpokládejme, že logická stránka číslo 3 je dosud volná a nějaký proces požaduje přidělení paměti; pro jednoduchost řekněme právě v rozsahu jedné stránky. Správce paměti přidělí logickou stránku číslo 3 a hledá odpovídající fyzickou stránku; zjistí však, že jsou všechny obsazeny. Zapíše proto obsah fyzické stránky číslo 0 do odkládacího prostoru a pozmění potřebným způsobem stránkové tabulky; výsledek vidíme na obr. 11.

Dojde-li později k tomu, že se původní vlastník pokusí pracovat se svou pamětí (kterou již vlastně nemá), detekuje jednotka řízení

Operační systémy

číslo 5, podívejme se znovu na obr. 10) se pokusí o přístup ke své paměti. Jednotka řízení paměti to samozřejmě detekuje, a protože žádná taková paměť není k dispozici (momentálně platí situace z obr. 11, kde logická stránka 5 žádou fyzickou stránku nemá připojenou), vyvolá výjimku. Výjimku obsluhuje správce paměti; ten se pokusí vyhledat nějakou volnou fyzickou stránku. Nenalezně ji, a proto musí někomu jednu odebrat - zvolí např. fyzickou stránku číslo 4. Nejprve její obsah uloží do odkládací oblasti, potom do ní z odkládací oblasti přečte údaje, které byly ještě nedávno uloženy ve fyzické stránce číslo 0. Pak upraví stránkové tabulky tak, jak ukazuje obr. 12.



obr. 12: přidělení nové stránky

Poznamenejme, že této druhé části, kdy je stránka přidělována na základě pokusu o použití adresy uvnitř logické stránky, říkáme **stránkování na žádost**.

Na virtuální paměti tedy není v zásadě nic složitého. Musíme však ještě vyřešit řadu 'technických detailů': správce paměti musí mít k dispozici nějaký klíč, podle kterého bude volit stránku, kterou má odebrat. Musí také rozhodnout, kam na odkládací prostor zapsat její obsah a musí toto místo rychle najít při obnovování stránky z disku.

V praxi je třeba zpochybnit i představu, kterou jsme v tomto odstavci rozvinuli - že totiž správce paměti reaguje na požadavky na přidělení bloku prací se stránkami. Kromě řady jiných nevýhod by to znamenalo, že velikost přidělovaných bloků paměti by musela být dělitelná velikostí stránky; to by ale bylo velmi nepraktické. V následujících odstavcích se proto na implementaci virtuální paměti podíváme podrobněji.

4.3.2 LRU a pseudo-LRU

Nejprve se podíváme na princip výběru stránky, kterou budeme uvolňovat v případě, že je zapotřebí vyhradit nějakou další paměť a všechny fyzické stránky již jsou přiřazeny.

Zamyslíme-li se nad problémem z hlediska efektivity celého operačního systému, nalezneme brzy správnou odpověď na otázku 'kterou stránku odebrat': přece tu, kterou už nebude nikdo potřebovat, a jestliže taková neexistuje, tak tu, kterou bude někdo potřebovat co nejpozději. Odpověď je to sice nepochyběně správná, ale bohužel nepoužitelná, protože - snad kromě věštíny v Delfách - neexistuje způsob, jak tento údaj o stránkách zjistit.

Rozsáhlé statistiky používání stránek v řadě operačních systémů však ukázaly, že bychom se mohli přiblížit optimu, kdybychom vybírali vždy tu stránku, kterou po nejdelší dobu nikdo nepoužil. V průměru je totiž daleko pravděpodobnější, že stránka, se kterou až dosud každou chvíli někdo pracoval, bude stejně intenzívne využívána i nadále a naopak, než že by došlo k výrazné změně ve využití jednotlivých stránek. Algoritmus, který vybere právě tu stránku, kterou co nejdelší dobu nikdo nepoužil, se jmenuje LRU (z anglického 'least recently used', čili nejdéle nepoužitá)²⁸.

V praxi však bohužel nemůžeme využít ani algoritmus LRU - představme si, jak bychom jej asi realizovali. Zda některá stránka byla nebo nebyla použita neví nikdo jiný než jednotka řízení paměti. Ta by musela při každém použití některé stránky - tedy vlastně při každém přístupu do paměti - generovat přerušení. Obslužná rutina přerušení by musela nejprve tento mechanismus vypnout (aby nebyla sama neustále přerušována) a pak by přemístila právě použitou stránku na konec fronty stránek, připravených k odebrání. Jinou alternativou je, že by obslužná rutina přerušení uložila někam do stránkových tabulek čas, kdy byla stránka naposledy použita a při odebírání stránky by se pak prohledaly všechny stránky a zvolila by se ta s 'nejstarším' časem. Na první pohled je zřejmé, že počítač by neměl čas téměř na nic jiného než na stránkování.

²⁸ Pro výběr stránky samozřejmě existuje i řada dalších algoritmů; algoritmus LRU se však alespoň zatím ukazuje jako nejvhodnější.

Operační systémy

Volíme proto algoritmy, které jsou jednodušší, méně zatěžují počítač a jejich výsledky se přitom blíží výsledkům algoritmu LRU. Takovým algoritmem potom říkáme **pseudo-LRU** algoritmy.

Nejčastěji používaný pseudo-LRU algoritmus pracuje následujícím způsobem:

- Součástí stránkových tabulek je tzv. used bit - speciální bit, který jednotka řízení paměti nastaví na jedničku při každém přístupu k odpovídající logické stránce (a tedy také k jí přiřazené stránce fyzické). To lze snadno zajistit bez jakékoli degradace výpočetního výkonu; snad všechny dnešní jednotky řízení paměti také s 'used' bitem pracují.
- Operační systém pravidelně v určitém intervalu prochází stránkové tabulky a nuluje všechny 'used' byty. Nalezne-li již některý 'used' bit nulový, znamená to, že tato stránka nebyla po celý interval použita (jinak by jednotka řízení paměti její 'used' bit nastavila), a je tedy dobrým kandidátem na odebrání.

Tento algoritmus má poměrně velmi dobré výsledky. Poznamenejme bez detailního výkladu, že jej lze mírně vylepšit využitím tzv. strategie sdružování dvojic - při výběru stránky k odebrání se díváme na dvojici po sobě následujících stránek jako na jednu velkou stránku. Podrobnější rozbor tohoto vylepšení případný zájemce najde v [OS].

4.3.3 Virtualizace paměti

Všechny stránky mají stejnou velikost, jsou tedy jako stvořeny k ukládání na disk do rychle zpracovatelné struktury s pevnou velikostí záznamu a s přímým přístupem. Nejjednodušší implementace virtuální paměti pak může vyhradit na disku místo třeba pro celý adresový prostor procesoru (máme-li ovšem k dispozici takovou kapacitu disků) a skutečnou operační paměť využívat vlastně jen jako cache.

Program potom může použít kteroukoliv adresu z celého adresového prostoru; jestliže se 'strefí' do logické stránky, která nemá svůj ekvivalent v operační

paměti, přidělí mu operační systém nějakou fyzickou stránku (buď volnou, nebo ji někomu odebere způsobem, se kterým jsme se již seznámili).

Prakticky žádný program však nemůže takovou obrovskou kapacitu operační paměti využít (snad vyjma několika specializovaných programů, na nichž pracují 'opravdoví programátoři' v Jet Propulsion Labs v Kalifornii). Je tedy zbytečné rezervovat obrovský rozsah disku pro zálohování operační paměti, která se nikdy nevyužije.

Naopak můžeme samozřejmě použitelný adresový prostor omezit podle velikosti odkládacího prostoru na disku - tak již dostaneme celkem životaschopný systém virtuální paměti, jaký je např. součástí operačního systému 7.x počítačů Apple Macintosh.

Komplexnější operační systémy (typicky systémy postavené na bázi UNIXu) však v praxi většinou postupují trochu lepším způsobem:

- Na začátku práce má program k dispozici celý adresový prostor, ale žádnou fyzickou paměť (všechny logické stránky tedy mají nastaven přepínač, určující přiřazení fyzické stránky, na 'neexistenci').
- Kdykoli se pokusí program použít nějakou adresu v logické stránce, která nemá a ani dosud neměla svůj ekvivalent v operační paměti (ze začátku tedy pokusí-li se program použít jakoukoli adresu), dojde k výpadku stránky a jednotka řízení paměti samozřejmě vyvolá přerušení. V něm se operační systém pokusí vyhledat nějakou dosud nevyužitou fyzickou stránku a přidělit ji programu. Pokud se mu to podaří, ukončí ihned přerušení a umožní tak programu pokračovat v přerušené práci - požadovaná adresa již ovšem má svůj ekvivalent v operační paměti.
- Ve chvíli, kdy operační systém nemá žádnou volnou fyzickou stránku, kterou by mohl programu přidělit, musí nějakou stránku uvolnit. K výběru fyzické stránky, která má být uvolněna, nejspíše využije strategii LRU, popsanou o pár odstavců výše. Původní obsah této fyzické stránky je zapotřebí zapsat do odkládací oblasti; tentokrát však logická adresa neurčuje místo na disku. Řešením je alokace vyrovnávací paměti na disku, s tím, že obsah fyzické stránky se uloží na disk (tam, kde je

Operační systémy

zrovna místo), stránka se 'staré' logické stránce odebere a ke 'staré' logické stránce se v tabulce stránek zaznamená, kde přesně na disku je obsah stránky uložen. Pak již nic nebrání tomu, aby byla uvolněná fyzická stránka přidělena 'nové' logické stránce.

- Jestliže potřebuje program použít data v logické stránce, která již dříve byla v operační paměti, ale fyzická stránka jí byla během času odebrána, proběhnou v podstatě opět předchozí dva body (nebo jen první z nich, je-li k dispozici nějaká volná fyzická stránka); jedinou změnou je to, že obsah přidělené stránky se před návratem do přerušeného programu inicializuje daty z disku, jejichž adresa byla uložena v tabulce stránek, když se fyzická stránka logické stránce odebírala podle minulého bodu. Nakonec se ještě může uvolnit místo na disku, kde byl uložen obsah stránky²⁹.

Je vidět, že při této strategii není na disku obsazeno více místa, než je nezbytně třeba; přitom je tato strategie prakticky stejně efektivní, jako výše navržená strategie udržování obrazu celého adresového prostoru na disku (ta by dokonce v některých případech nebyla použitelná vůbec, jak brzy uvidíme při popisu virtualizace adres).

Systém stránskování tedy umožňuje, aby programy pracovaly s 'dojmem', že mají k dispozici daleko více operační paměti, než kolik jí je doopravdy v systému instalováno. Tato paměť je ovšem jen zdánlivá - cizím slovem **virtuální**. Proto tento způsob 'rozšíření' paměti nazýváme virtualizací paměti.

Výhody, plynoucí z virtualizace paměti, jsou zřejmé: nadále nejsou žádné problémy s rozsahem dat, programátor může celkem pokojně pracovat s polem 1000 x 1000 x 1000 (ne, že bychom takový styl programování doporučovali). Odpadají také jakékoli problémy s překryvy (overlay) - program může být prakticky libovolně velký. Počítač přitom nemusí mít příliš velkou operační paměť - méně paměti pouze sníží efektivitu systému (je nutné častěji

²⁹Ale také se uvolňovat nemusí. Operační systém může případně využít toho, že program obsah stránky pouze četl, ale neměnil a při odebíráni této fyzické stránky ji 'uloží' na totéž místo na disku - ve skutečnosti si ovšem čas potřebný na ukládání ušetří, protože tam již nezměněná stránka je. Běžné jednotky řízení paměti tomuto přístupu napomáhají tím, že sledují, byly-li obsah stránky použit pro zápis (a tedy se změnil a bude nutné jej znova uložit na disk).

'prehazovat' stránky mezi diskem a operační pamětí). Celý systém s virtuální pamětí tedy může být poměrně levný; přitom umožní zpracování úloh, které by nebylo možné na daleko dražším stroji s několika sty megabytů operační paměti (pro dostatečně efektivní pokrytí virtuálních čtyř gigabytů - což je obvyklá velikost adresového prostoru - stačí několik, nejvýše několik desítek megabytů).

43.4 Implementace správce paměti

Nejjednodušší a poměrně efektivní implementace virtuální paměti využívá správce paměti s přidělováním bloků, jak jsme jej poznali v odstavci 4.1 - tedy nejjednodušší algoritmus vůbec³⁰. Nepracuje však nad reálnou operační pamětí, jako tomu bylo předtím, ale nad celým adresovým prostorem procesoru (nebo nad jeho částí, je-li rozsah virtuální paměti omezen).

O pokrytí adresového prostoru operační paměti se stará druhá úroveň správce paměti, která s první úrovní vůbec žádným způsobem nesouvisí. Správce paměti se v takovém případě vlastně rozpadá na dvě zcela samostatné části: správce virtuální paměti a vlastní správce paměti, který přiděluje procesům požadované bloky. Správce virtuální paměti přitom zajistí, že celý zbytek operačního systému - včetně vlastního správce paměti - se může chovat stejně, jako kdyby byl celý adresový prostor osazen skutečnou operační pamětí.

Samozřejmě, že na úrovni vlastního správce paměti dochází k fragmentaci. Fragmentován je však pouze adresový prostor, nikoli skutečná paměť, protože logickým stránkám v 'dírách' fragmentovaného adresového prostoru samozřejmě nejsou přiděleny fyzické stránky. Vzhledem k rozsahu adresového prostoru (u běžných procesorů se jedná řádově o gigabyty) nám jeho fragmentace nevadí, protože i když fragmentace 'spolkne' hodně, zůstane vždy dostatečně velký blok ještě volného adresového prostoru.

³⁰Samozřejmě vyjma přidělování celé paměti, používaného v nejjednodušších jednoprogramových a jednouživatelských systémech.

Na tomto principu je možné doplnit správce virtuální paměti i k operačnímu systému, který sám o sobě jednotku řízení paměti nevyužívá³¹. Tak existují např. komerční programy zajišťující virtuální paměť pro počítač ATARI TT, jehož operační systém ATARI TOS ani zdaleka nevyužívá možností použitého mikroprocesoru Motorola 68030. Stejným způsobem je vlastně vyřešen i systém virtuální paměti operačního systému 7.x pro počítače Apple Macintosh, o kterém jsme se již zmínili - de facto se jedná o externí program, ačkoliv je standardně dodáván se systémem.

Správce virtuální paměti, doplněný takovýmto způsobem k operačnímu systému zvenku, je samozřejmě lepší než žádný; toto řešení však má řadu nevýhod. Operační systém totiž může ve spolupráci se správcem virtuální paměti zajistit řadu velmi šikovných služeb - jmenujme některé z nich:

- Služeb správce virtuální paměti může využívat systém pro práci se soubory pro rychlý a efektivní přístup k datům na disku. Uvědomme si, že stačí vytvořit umělé stránkové tabulky obsahující odkazy na sektory disku, ve kterých je uložen soubor, a můžeme pohodlně pracovat s 'operační pamětí' - ve skutečnosti budeme pracovat se zvoleným souborem.
- Správce paměti může automaticky zvětšovat paměťové bloky jestliže jejich uživatelé překročí hranice bloků. To je mimořádně výhodné pro bloky obsahující zásobník.
- Ovladače periferních zařízení mohou také využívat virtuální paměť s tím, že po dobu přenosu dat mezi pamětí a periferií odpovídající stránku 'zamknou' (tj. upozorní správce virtuální paměti, že jím stránku nesmí odebrat).
- Správce paměti může být implementován odlišným (a daleko jednodušším) způsobem - přečtěme si následující odstavec.

³¹To se netýká MS DOSu - v jeho případě totiž není rozsah použitelné paměti omezen velikostí instalované fyzické paměti, ale chybami operačního systému samotného.

4.3.5 Virtualizace adres

Jednotku řízení paměti můžeme využít ještě daleko lepším způsobem než pro samotnou virtualizaci paměti. Představme si, že systém stránkování, popsaný v minulých odstavcích, používáme v multitaskovém operačním systému. Ihned se nám objevuje jedno nepříjemné omezení: ačkoli nám stránkování umožnilo (virtuálně) rozšířit operační paměť na celý adresový prostor, musí se v něm pořád vedle sebe nějak srovnat všechny procesy.

Operační systém se přitom musí starat o spoustu věcí - relokace zaváděných programů, poměrně složité spojování se sdílenými knihovnami a podobně. Oč by bylo jednodušší, kdyby měl každý proces vlastní adresový prostor, ve kterém by si mohl dělat, co se mu zachce.

Pokud si ještě jednou vzpomeneme na překlad adres, popsaný na straně 52, nabídne se nám řešení samo: stačí, když bude mít každý proces vlastní tabulku stránek. Spojíme-li tento 'vynález' se stránkováním na žádost, nemusí se vlastní správce paměti téměř o nic starat, všechnu práci zajistí správce virtuální paměti: fyzické stránky se přidělují tomu, kdo je právě potřebuje a při odebírání se ukládají do bufferů na disku. Protože neexistuje pevné přiřazení mezi adresami na disku a logickými stránkami (vytváří se dynamicky až při práci systému), nemá správce virtuální paměti o nic více práce, než když zajišťoval stránkování na žádost pro jediný program (respektive pro celý operační systém jako pro jediný program). Po upozornění správce procesů, že se jiný proces stává aktivním, pouze musí zajistit výměnu tabulek popisujících stránky.

Tento jednoduchý trik zajistí, že každý proces může používat vlastní adresový prostor, který se nijak nekříží (nebo kříží, ale přesně tak, jak procesy požadují) s adresovým prostorem ostatních procesů. Proto pro něj používáme termín virtualizace adres.

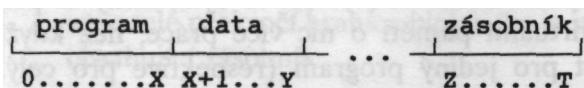
4.3.6 Další informace

Případný zájemce o podrobnosti související s virtuální pamětí a jednotkou řízení paměti (jako je např. problematika hierarchických stránkových tabulek, implementace sdílení adresového prostoru více procesy a podobně) nalezne velmi detailní rozbor v [68030].

4.4 Segmentace

Na konec se - kvůli omezenému rozsahu této knihy jen velmi stručně - zmíníme o **segmentaci**. Právě popsaný mechanismus stránkování je před uživatelskými programy dokonale skrytý - programy prostě pracují s logickým adresovým prostorem, a jeho překlad do fyzické paměti je vůbec nezajímá.

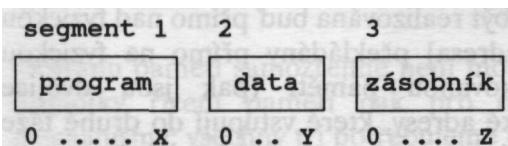
Uvědomme si ale, že existuje řada celkem rozumných situací, kdy by se vyplatilo, aby program měl přímý přístup k mechanismu překladu adres. Představme si zcela klasickou situaci, kdy je na nejnižších adresách umístěn program, nad ním data a nakonec zásobník:



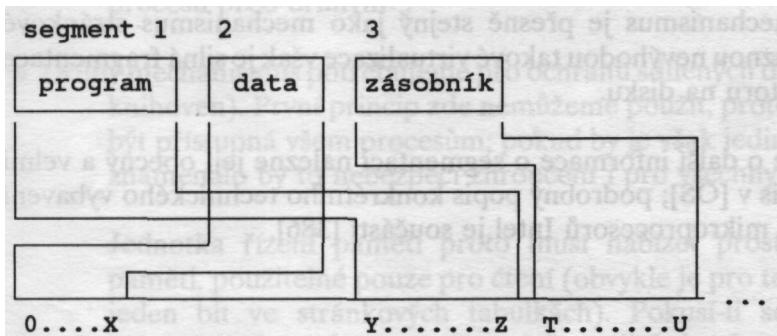
Požaduje-li program více dat, posune se hranice mezi daty a nevyužitou pamětí (adresa Y) nahoru. Zásobník naopak roste dolů, adresa Z se tedy snižuje. Budou-li tedy požadavky programu na paměť stále narůstat, může nakonec dojít k tomu, že na sebe 'narazi' horní hranice dat a dolní hranice zásobníku. Sama virtualizace tomu nemůže zabránit - hovořili jsme přece celou dobu o adresách; s využitím virtualizace jsme tedy mohli mít na mysli adresy v logickém adresovém prostoru.

Pokud by však program měl k dispozici aparát pro překlad adres, mohl by pro interpretaci 'datových' adres použít jiné překladové tabulky, než které používá pro interpretaci adres 'zásobníkových'. Pak by samozřejmě ke zmíněnému problému nemohlo dojít - zásobník by byl de facto v jiném adresovém prostoru než data, a kolize by byla vyloučena.

Popsaným způsobem právě funguje segmentace. Při ní program nepracuje s jednoduchými adresami, ale s dvojicemi [segment,adresa]. Jednotka řízení paměti se pak automaticky postará o to, aby dva segmenty nemohly navzájem kolidovat, a to ani v případě, že program použije v obou segmentech totožné adresy. Segmentace tedy přináší programu možnost explicitně využívat několika nezávislých adresových prostorů. Běžná struktura programu v segmentovaném prostředí proto vypadá trochu jinak:



Kdykoli pak chce program pracovat např. s daty, musí kromě jejich adresy určit, že se jedná o segment číslo dvě. Podobně pro práci se zásobníkem je zapotřebí specifikovat segment číslo tři. Mapování všech tří segmentů do skutečné operační paměti (nebo do virtuálního adresového prostoru) je přitom samozřejmě lhostejné, a může vypadat třeba takto:



Data tak mohou růst poměrně velmi daleko - od adresy X až k adrese Y - bez jakýchkoli problémů; stačí změnit velikost segmentu. Ale ani případ, kdy by data narostla nad tuto mez není kritický; stačilo by přemapovat datový segment jinam (třeba na adresu U+1), zkopirovat tam jeho obsah a přidělit mu tam daleko více paměti.

Existují různé přístupy k segmentaci. V nejobecnějším případě by mohly být všechny adresy, které program používá, dvourozměrné - jedna část adresy by

Operační systémy

určovala segment, druhá by pak byla adresou uvnitř segmentu. Takové řešení však má jednu nevýhodu: adresy jsou zbytečně velké. V praxi se proto obvykle používají pomocné registry, které identifikují požadované segmenty; součástí adresy je pak pouze několikabitové číslo registru. Mikroprocesory Intel jdou dokonce ještě dál: informace o požadovaném segmentu není vůbec součástí adresy; namísto toho má každá operace 'přidělen' standardní segment, se kterým pracuje.

Jak jsme se zmínili, může segmentace být realizována buď přímo nad fyzickou pamětí (pak jsou dvojice [segment,adresa] překládány přímo na fyzickou adresu), nebo na virtuální stránkovovanou pamětí (pak jsou dvojice [segment,adresa] překládány na logické adresy, které vstupují do druhé fáze překladu adres při běžné virtualizaci, kterou známe z předcházejícího textu). Je zřejmé, že druhá metoda je flexibilnější, vyžaduje však komplikovanějšího správce paměti.

Segmentace je ideální technické vybavení pro realizaci správce paměti s transparentním setřásáním bloků. Sama segmentace při dobré navržené jednotce řízení paměti dokonce umožňuje i realizaci virtuální paměti na úrovni segmentů (její mechanismus je přesně stejný jako mechanismus stránkové virtualizace); výraznou nevýhodou takové virtualizace však je silná fragmentace odkládacího prostoru na disku.

Případný zájemce o další informace o segmentaci nalezne její obecný a velmi vyčerpávající popis v [OS]; podrobný popis konkrétního technického vybavení pro segmentaci u mikroprocesorů Intel je součástí [386].

4.5 Ochrana paměti

Základní princip ochrany paměti jsme již popsali; jedná se však o tak důležitou oblast, že si zaslouží samostatný odstavec.

Ochrana paměti zajišťuje, aby žádný proces nemohl poškodit data jiného procesu nebo dokonce samotného operačního systému. V multitaskovém

operačním systému je toto zabezpečení a jeho 'neprůstřelnost' jedním z nejdůležitějších faktorů - musíme si uvědomit, že obvykle zároveň běží řada procesů, z nichž některé jsou velmi důležité i pro ostatní uživatele (např. správce sítě, řada počítačů umožňuje také zpracování vzdálených procesů, tj. na 'mém' počítači může běžet program mého kolegy z vedlejší kanceláře). Pokud by jediný špatně napsaný program mohl zavinit zhroucení ostatních procesů, byla by práce takového systému velmi málo efektivní - jeho uživatelé by totiž většinu času trávili obnovováním přerušených procesů a zničených dat.

Ochrana paměti samozřejmě není možné zajistit bez jednotky řízení paměti. Jednotky řízení paměti pak pro ochranu paměti nabízejí obvykle tři mechanismy; všechny tři potřebujeme (nemohou se tedy vzájemně zastoupit):

- První mechanismus je vlastně vedlejším efektem virtualizace adres: paměť, která není vůbec dosažitelná, nemůže být poškozena. Fyzické stránky jednoho procesu jsou pro druhý proces nepřístupné - buď nejsou v jeho stránkových tabulkách, nebo je jejich obsah bezpečně uložen v odkládacím prostoru. Tak je zajištěna ochrana paměti jednoho procesu před druhým³².
- Druhý mechanismus potřebujeme pro ochranu sdílených dat (např. sdílených knihoven). První princip zde nemůžeme použít, protože data musí být přístupná všem procesům; pokud by je však jeden z nich poškodil, znamenalo by to nebezpečí zhroucení i pro všechny ostatní.

Jednotka řízení paměti proto musí nabízet prostředky k označení paměti, použitelné pouze pro čtení (obvykle je pro tento účel vyhrazen jeden bit ve stránkových tabulkách). Pokusí-li se takovou paměť některý z procesů číst, zpřístupní mu ji jednotka řízení paměti bez jakýchkoli problémů. Jestliže se však proces pokusí tuto paměť modifikovat, generuje jednotka řízení paměti přerušení stejně jako

³²Na případnou námitku pozorného čtenáře 'jak lze tuto ochranu zajistit, nepoužíváme-li virtualizaci adres' zní odpověď 'nijak'. Běžné jednotky řízení paměti neumožňují zajistit ochranu paměti jednoho procesu před druhým jinak než virtualizaci adres; to je možné pouze u některých speciálních jednotek řízení paměti (příkladem může být ochrana paměti, kterou má k dispozici operační systém EPOC na počítačích PSION).

v případě výpadku stránky. Operační systém na takové přerušení obvykle reaguje velmi jednoduchým způsobem - násilně ukončí 'provinilý' proces.

- Konečně třetí mechanismus slouží k ochraně systémových dat před procesy. Každý moderní procesor rozlišuje, běží-li právě některá rutina operačního systému nebo běžný proces. Jednotka řízení paměti musí nabízet prostředky k označení stránek, které jsou 'volné pro kohokoli', a stránek, které jsou k dispozici jen operačnímu systému. Při přístupu do paměti pak jednotka řízení paměti porovná přístupová práva stránky s momentálním režimem procesoru a pokud je běžný proces operovat s pamětí vyhrazenou operačnímu systému, generuje opět přerušení.

Ochrana systémových dat před procesy by teoreticky samozřejmě mohla být zajištěna pomocí prvního mechanismu; bylo by to však nepraktické, protože k 'přepínání' mezi operačním systémem a běžným procesorem dochází daleko častěji než k přepínání mezi jednotlivými procesy. Některé jednotky řízení paměti však dokáží pracovat zároveň se dvěma tabulkami stránek, jedna z nich se použije pro přístupy k paměti běží-li operační systém, druhá slouží běžným procesům. S takovou jednotkou řízení paměti (je např. součástí mikroprocesorů Motorola) má návrhář operačního systému na vybranou, použije-li pro ochranu systému první nebo třetí mechanismus.

Dodejme, že zajišťuje-li jednotka řízení paměti segmentaci, je zapotřebí, aby na úrovni segmentů nabízela stejné bezpečnostní služby. Speciálně tedy musí být možné segment označit jako použitelný pouze pro čtení, nebo pouze pro přístup z operačního systému. Samozřejmostí je ošetření délky segmentů - pokud je program pracovat v některém segmentu s adresou, která není součástí segmentu, musí dojít k chybě (a ne k překladu na adresu ve zcela jiném segmentu, jak tomu je u mikroprocesorů Intel 8086 a 80186!).

5. Procesy a procesor

Správa procesů a procesoru je jedním z nejkomplikovanějších prvků operačního systému (jedná-li se o komplexní multitaskový systém - správa procesů např. v MS DOSu je samozřejmě poněkud triviální).

Z toho důvodu bude i tato kapitola poměrně rozsáhlá. Probereme v ní postupně řadu věcí, které se správou procesů souvisejí:

- Nejprve si vysvětlíme samotný pojem 'multitasking', který jsme dosud používali v intuitivním smyslu, a ukážeme si různé možnosti jeho zajištění. Vysvětlíme si také, jaké má multitasking výhody proti jednoúlohovým systémům.
- Pak se seznámíme s technickými principy multitaskingu (kooperativního i preemptivního) a korektně definujeme další pojmy (např. 'proces'), které jsme prozatím používali bez zcela jasného významu.
- V další části se zaměříme na správu procesů a ukážeme si její realizaci v běžném multitaskovém operačním systému.
- Na konci kapitoly věnujeme několik odstavců velmi důležité oblasti synchronizace procesů.

5.1 Co je to multitasking

Základní definice multitaskingu je jednoduchá: multitaskový počítač (respektive operační systém, protože multitaskový operační systém lze vytvořit pro každý počítač) je **takový, který umožňuje současný běh několika programů**.

Zamysleme se nejprve nad tím, jestli takovou funkci počítače vůbec potřebujeme. Multitasking přináší některé výhody i některé nevýhody; pokusme se je zde shrnout. Nejprve se podíváme na výhody:

—Operační systémy—

- Multitasking umožňuje kdykoli přejít k jinému programu, potřebujeme-li jej 'na chvíli' použít, aniž bychom byli nuceni přerušovat rozdělanou práci.
- Multitasking usnadňuje implementaci činností, které z principiálních důvodů musí probíhat paralelně s ostatními činnostmi počítače (dobrým příkladem je správa počítačové sítě). V multitaskovém operačním systému prostě takové činnosti zajistí samostatný proces a není pro ně zapotřebí vytvářet nový komplikovaný aparát.
- Jednotlivé programy mohou v multitaskovém prostředí lépe kooperovat. Zatímco v jednoúlohovém prostředí si mohou programy nejvýše předávat údaje prostřednictvím souborů, mohou dva paralelně běžící programy navzájem přímo ovlivňovat svou činnost.
- Multitasking je nutnou podmínkou pro realizaci víceuživatelského systému. Má-li jediný počítač zpracovávat požadavky několika uživatelů, musí být vybaven multitaskovým operačním systémem, protože žádný z uživatelů samozřejmě nebude chtít čekat až budou ostatní hotovi.
- Multitasking umožňuje daleko lepší využití výpočetní kapacity systému. Uvědomme si, že například při práci s textovým editorem počítač naprostou většinu času jen čeká až stiskneme klávesu; tuto dobu by stejně dobře mohl vyplnit prací na nějakém složitějším výpočtu (jehož výsledky budeme potřebovat až dopříjem text). Obdobným příkladem je program, který pracuje s daty na disku - např. překladač při zpracování rozsáhlého projektu. V době, kdy pracuje řadič nebo mechanika disku, se procesor může opět věnovat nějaké jiné činnosti, zatímco v klasickém jednoúlohovém systému by zahálel.

Multitaskový operační systém však samozřejmě přináší i některé nevýhody:

- Je-li systém špatně navržen, může současný běh většího množství programů degradovat funkci toho programu, se kterým uživatel přímo

komunikuje³³ - běh programu se neúnosně zpomalí, zvýší se doba odezvy na uživatelské příkazy a podobně. Takovým způsobem se často chovají zvláště kooperativní systémy (viz níže).

- I multitaskový systém, který nemá nectnost popsanou v minulém bodě, má samozřejmě nějakou vlastní režii; ta se přidá k času potřebnému pro zpracování samotných programů. V krajiném případě by mohlo dojít i k tzv. **zahlcení** systému - naprostá většina výpočetní kapacity by byla spotřebována na režii systému a programy by nebyly zpracovávány vůbec nebo téměř vůbec. I to je známka nesprávného návrhu systému; na rozdíl od minulého bodu toto nebezpečí hrozí spíše u preemptivních operačních systémů.
- Multitaskový operační systém je samozřejmě daleko složitější a tedy i větší a dražší než systém jednoúlohový. Multitaskový operační systém často také potřebuje větší konfiguraci - rozsáhlejší operační paměť, větší kapacitu disků a výkonnější procesor.

Je vhodné poznamenat, že požadavky na rozsáhlou konfiguraci bývají spíše ukázkou jistého megalomanství návrhářů operačních systémů (spojeného s bezpečným vědomím, že operační paměť je dnes již velmi levná, stejně jako kapacita disků), nikoli vlastností multitaskingu jako takového - minimální konfigurace, na které dobře pracuje multitaskový operační systém EPOC zahrnuje 512 kilobytů operační paměti (z toho 384KB je paměť ROM, obsahující EPOC a jen 128KB je RAM), zádnou vnější paměť a mikroprocesor NEC V30 s taktem pouhých 3.84MHz.

- Není-li multitaskový operační systém spojen s kvalitním systémem zabezpečení, zvyšuje nepříjemným způsobem riziko ztráty dat. Při zhroucení jednoúlohového systému přijdeme pouze o výsledky práce

³³Automaticky předpokládáme, že operační systém je uzpůsoben pro interaktivní ovládání. Tak tomu samozřejmě nemusí být vždy - operační systémy střediskových počítačů bývají např. uzpůsobeny pro zpracování dávkové, kdy uživatel se svým programem za běhu nijak nekomunikuje a získá až jeho výsledky. Tam pak je tato nevýhoda irrelevantní.

programu, který zhroucení zavinil³⁴. Jestliže však některý program 'shodí' multitaskový operační systém, jsou ztraceny výsledky práce všech procesů, které v té době běžely.

Zrekapitulujeme-li výhody i nevýhody multitaskingu, dojdeme snadno k následujícím závěrům:

- Multitaskový operační systém musí být velmi pečlivě navržen, jinak je lepší zůstat u jednoúlohového systému (jehož kvalita nebo nekvalita nemá na práci počítače ani zdaleka takový vliv).
- Multitaskový operační systém se nevyplatí pro systémy, které jsou používány striktně jednoúčelově - máme-li např. mikropočítač, na kterém vedeme pouze databázi zaměstnanců a nechceme po něm nic jiného (ani psaní dopisů, ani účetnictví, ani hrání počítačových her ...), byla by investice do multitaskového operačního systému zcela zbytečná.
- Multitaskový operační systém by měl být bezpečný. Jestliže tomu tak není, musíme to mít na paměti a pracovat s ním 'opatrн', běží-li nějaký důležitější program - což samozřejmě výhody systému do značné míry degraduje.
- Není pravda, že se multitaskový systém nehodí pro méně výkonné počítače - jak dokumentuje systém EPOC, stačí, aby byl systém vhodně navržen. Na druhou stranu je samozřejmé, že přeneseme-li systém navržený a optimalizovaný pro počítače s desítkami megabytů operační paměti a se stovkami megabytů diskové kapacity na IBM PC/XT, bude jeho užitečnost minimální.

³⁴ Až na naprosté výjimky bývá operační systém odladěn tak dobře, že sám zhroucení systému nezaviní - to je téměř vždy výsledkem chyby některého programu.

5.2 Princip multitaskingu

Naprostá většina dnešních počítačů je vybavena jediným procesorem³⁵. Není proto technicky možné, aby na nich doopravdy běželo najednou několik programů; operační systém musí současný běh programů nějak 'simulovat'. Podívejme se, jakým způsobem toho lze docílit.

(Pro doplnění poznamenejme, že obdobné techniky musí používat i operační systémy nejmodernějších počítačů osazených více procesory. Tam je totiž zase zapotřebí rozdělovat výpočetní zátěž rovnoměrně mezi jednotlivé procesory; k tomu by ale nedošlo, kdyby každému procesoru byl napevno přidělen jeden proces.)

5.2.1 Přepínání programů

Operační systém s přepínáním programů (**task switching**) je přímým předchůdcem kooperativního multitaskingu. Vznikl na interaktivních systémech na základě poměrně jednoduché úvahy:

Jednou z nejzřetelnějších výhod multitaskingu pro toho, kdo pracuje s jednouživatelským interaktivním operačním systémem, je první z výhod, popsaných na straně 70 - totiž možnost kdykoli přejít ke zpracování jiného programu, který uživatel právě potřebuje, aniž by byl nucen přerušit práci.

Prvním krokem k dosažení této výhody bylo tzv. **vzájemné volání**. Vzájemné volání není vlastností operačního systému, ale jednotlivých programů: ty mohou umožnit spuštění jiného programu a práci s ním; po ukončení tohoto programu se obnoví stav původního programu a uživatel s ním může pracovat dále. Vzájemné volání bylo rozšířeno u programů pro Apple Macintosh a dnes je běžné u programů pro MS DOS (kde se místo přímého volání požadovaného

³⁵Máme na mysli univerzální procesor, který zpracovává programy. Moderní počítače jsou samozřejmě osazeny řadou specializovaných procesorů, které zajíšťují některé často prováděné činnosti. Typickým příkladem může být blitter (viz strana 40) nebo I/O procesory (kanály).

Operační systémy

programu obvykle volá příkazový interpret - je to o něco pohodlnější pro uživatele, zabere to však víc operační paměti).

Vzájemné volání však má nejméně jednu zásadní nevýhodu: chceme-li pouze provést jiný program a pak se vrátit k momentálnímu stavu, poslouží nám dobře. Není však k ničemu, jestliže potřebujeme mezi oběma programy přepínat, aniž bychom jeden či druhý ukončili (např. příšeme nějaký dokument, do kterého přebíráme řadu informací z databáze - nejpohodlnější by bylo přepínat mezi editorem a databázovým programem).

Návrháři operačních systémů se tedy pokusili přinést uživatelům zmíněnou výhodu, aniž by museli vytvořit plně multitaskový operační systém (jehož návrh, jak víme, je dost náročnou záležitostí). Výsledkem byl operační systém, který dokáže zavést a spustit více programů, a uživatel pak mezi nimi může přepínat. Zvolený program zcela normálně běží; ostatní programy jsou dočasně potlačeny a jen čekají, až je uživatel zvolí.

Existují dvě skupiny operačních systémů s přepínáním programů: systémy s omezeným přepínáním a systémy s neomezeným přepínáním.

- **U systémů s omezeným přepínáním programů** je možné přepínat jen mezi jedním 'normálním' programem - říká se mu hlavní program - a několika speciálními programy vytvořenými zvláštním způsobem výhradně pro přepínání. Tyto speciální programy se obvykle nazývají 'pomůcky' (**accessories**) a uživatel bývá omezen i ve volbě pomůcek, které bude mít při práci s počítačem k dispozici (nejčastěji to lze určit při startu systému, ne však později).

Jakkoli tyto operační systémy kladou řadu velmi silných omezení, je práce s nimi nepochybňě daleko pohodlnější než práce s klasickými jednoúlohovými operačními systémy, zvláště je-li omezené přepínání spojeno se vzájemným voláním. Příkladem operačního systému s omezeným přepínáním programů může být GEM nebo nejstarší verze operačního systému počítačů Apple Macintosh (před zavedením programu MultiFinder).

- **Systémy s neomezeným přepínáním** naproti tomu umožňují spuštění několika 'normálních' programů a přepínání mezi nimi. Kdykoli

samořejmě můžeme přepnout do příkazového interpretu a spustit další program (je-li dostatek volné paměti).

Z hlediska současného přístupu ke službám více programů jsou systémy s neomezeným přepínáním prakticky rovny multitaskovým operačním systémům (nemají však jejich ostatní výhody, zato jsou mnohem jednodušší).

Dobrým příkladem operačního systému s neomezeným přepínáním programů jsou starší verze operačního systému počítačů Apple Macintosh po zavedení MultiFinderu (návod k použití MS DOSu verze 5.0 udává, že i MS DOS 5.0 nabízí uživateli neomezené přepínání programů; autor této knihy však po jeho pečlivém studiu došel k názoru, že v tomto bodě se zmíněný návod mylí).

Zatímco princip vzájemného volání je zřejmý, princip přepínání programů tak jasný není. Popíšeme jej proto podrobněji. Při tomto popisu se seznámíme s pojmem 'kontext'; tento pojem je pro další výklad velmi důležitý, a proto mu věnujeme samostatný odstavec.

522 Kontext

Představme si, že na našem počítači běží nějaký program. Chceme tento program přerušit, ale takovým způsobem, abychom mohli po nějakém čase jeho běh opět obnovit; program přitom musí pokračovat 'jako by se nic nestalo'.

Je zřejmé, že bychom toho mohli dosáhnout tak, že bychom někam na vnější paměť uložili kompletní stav počítače: obsah celé operační paměti, obsah registrů procesoru, nastavení různých zařízení, stav pomocných procesorů (jako je blitter nebo kanály) a tak dále. Obnovení běhu přerušeného programu by pak vlastně spočívalo pouze v obnovení stavu počítače; přerušený program by již pokračoval automaticky (součástí uloženého stavu by samozřejmě byla i informace 'právě je zpracováván program na adrese té a té').

Operační systémy

V praxi pochopitelně není možné takto postupovat. Řada kroků je zbytečná - nemusíme ukládat obsah operační paměti, jestliže zajistíme, aby paměť, kterou program používá³⁶ zůstala až do jeho obnovení beze změny; nemusíme se starat o zařízení, která program nepoužívá, a podobně. Některé kroky jsou dokonce nemožné - těžko bychom např. uvedli do původního stavu tiskárnu, na kterou mezitím jiný program vytiskl několik řádek textu. Situaci si také můžeme podstatným způsobem zjednodušit, vyžádáme-li si od programu, aby před přerušením přešel do přesně definovaného stavu: může ukončit práci se všemi periferními zařízeními i s obrazovkou (takže stav kanálů a blitteru nemusíme ukládat), může ukončit všechny rozpracované výpočty v pohyblivé řádové čárce (takže nás nezajímá ani stav aritmetického koprocessoru) a podobně. Primějeme-li navíc program k tomu, aby nespolehal na obsah registrů procesoru po obnovení práce, zbyde jen naprosté minimum údajů které musíme uložit.

Jestliže důsledně využijeme všech popsaných možností, zůstane nám nakonec pouze nastavení zásobníku (tedy obsah patřičného registru) a obsah obrazovky (není-li sdílení obrazovky vyřešeno jiným způsobem - viz druhý díl knihy)³⁷. Umístění bloků paměti si nemusíme zvlášť pamatovat, protože program svou paměť 'zná' a po obnovení práce ji bude korektně používat. To již není tolik údajů; můžeme snadno vytvořit tabulku, ve které budou tato data uložena pro každý program, který je momentálně 'rozpracován'.

Pro údaje uložené v tabulce budeme používat pojem **kontext**³⁸ a 'programu', který má v paměti svůj kontext, budeme říkat **proces** (často se ve stejném smyslu používá i pojem **task**).

Předpokládejme, že jeden z procesů je právě aktivní, a na základě nějakého signálu uživatele je zapotřebí přepnout na jiný proces. Operační systém vyčká, až proces zavolá službu, kterou oznamuje že je v 'přerušitelném' stavu (každý

i paměť, ve které je uložen sám program, i paměť pro jeho data).

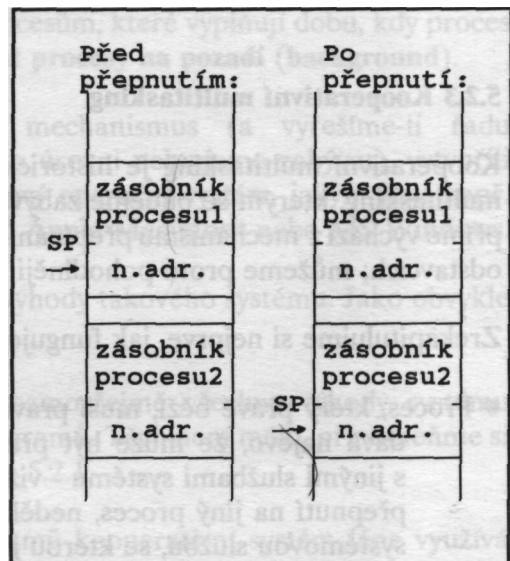
³⁷Naznačený problém s tiskárnou prozatím vyřešíme - po vzoru většiny operačních systémů s přepínáním programů - nejjednodušším možným způsobem: budeme jej ignorovat.

³⁸Striktně vzato je kontextem procesu kromě údajů uložených v tabulce i jeho paměť. Pro další výklad však je pohodlnější definovat 'kontext' takto.

proces je povinen *to* pravidelně dělat³⁹). Pak operační systém jen uloží do tabulky na místo odpovídající dosud aktivnímu procesu adresu vršku jeho zásobníku (přečte ji z patřičného registru procesoru). Nakonec systém zjistí z tabulky adresu zásobníku nového procesu a zavede ji do patřičného registru procesoru. Pak ukončí službu a od té chvíle běží nový proces; říkáme, že došlo k **přepnutí kontextu (context switch)**. Budeme také používat pojmu **odebrání procesoru** (o starém procesu, který běžel a již neběží) a pojmu **přidělení procesoru** (o novém procesu, který je nyní aktivní).

Uvědomme si, jak přepnutí kontextu probíhá: služba systému, kterou dává proces na vědomí že je v 'přerušitelném' stavu, je na první pohled docela obyčejným podprogramem. Jakmile však má dojít k přepnutí kontextu, chová se tato služba zcela neobvyklým způsobem - vrátí se totiž na jiné místo, než ze kterého byla vyvolána.

Podívejme se na obr. 13. Těsně před přepnutím kontextu obsahuje zásobník procesu číslo 1 (který byl až dosud aktivní) na vrcholku návratovou adresu pro návrat ze systémové služby zpět do kódu procesu. Dříve, než dojde k návratu, však přepnutí kontextu změní zásobník. Ve chvíli návratu je tedy aktivní zásobník procesu číslo 2, na jehož vrcholu je samozřejmě také návratová adresa, ale tentokrát do procesu číslo 2 - a tam se také systémová služba ve skutečnosti vrátí.



obr. 13: přepnutí kontextu

³⁹Skutečné operační systémy mívají tuto službu kombinovanou s řadou dalších systémových služeb, téměř vždy mezi ně patří např. služba pro čtení znaku z klávesnice. Tak je nepřímo zajištěno, že se proces pravidelnému volání služby nevyhne; je ovšem nutné před jejím zavoláním přejít do stavu umožňujícího přerušení (o to se samozřejmě zčásti postará hotový kód na systémových knihovnách).

Operační systémy

Po nějakém čase samozřejmě bude situace opačná a systém bude přepínat z nějakého jiného procesu na proces číslo 1. Pak se teprve analogickým způsobem použije návratová adresa uložená na jeho zásobníku, a z lokálního hlediska procesu číslo 1 se vlastně teprve pak služba systému ukončí.

Dodejme, že při vytváření procesu je samozřejmě nutné uměle vytvořit zásobník 'jako kdyby' došlo k přerušení procesu před jeho první instrukcí. To ovšem není nic složitého - na zásobník nového procesu se prostě uloží adresa jeho první instrukce a adresa tohoto zásobníku se zapíše do kontextu.

5.2.3 Kooperativní multitasking

Kooperativní multitasking je historicky mladší než dokonalejší preemptivní multitasking, kterým se budeme zabývat v příštím odstavci. Jeho princip však přímo vychází z mechanismu přepínání programů popsaného v předcházejících odstavcích; můžeme proto pohodlněji navázat.

Zrekapitulujme si nejprve, jak funguje přepínání programů:

- Proces, který právě běží, musí pravidelně volat systémovou službu, kterou dává najevo, že může být přerušen (tato služba bývá kombinována s jinými službami systému - viz pozn. 39). Dokud si uživatel nevyžádá přepnutí na jiný proces, nedělá tato služba nic (nebo pouze provede systémovou službu, se kterou je kombinována).
- Vyžádá-li si uživatel přepnutí procesů, zajistí zmíněná služba při nejbližším vyvolání přepnutí kontextu. Po jejím ukončení tedy již běží nový proces, a dosud aktivní proces čeká, až bude znova aktivován.

Předpokládejme, že 'přerušovací' služba je kombinována mimo jiné také se službou pro čtení znaku z klávesnice. Jestliže uživatel o žádné přepínání procesů nestojí, bude služba skutečně jen číst znak z klávesnice; to však může trvat velmi dlouho - dokud uživatel nestiskne nějakou klávesu. Uvědomme si, že rychlosť i toho nejpomalejšího procesoru mnohonásobně převyšuje rychlosť nejrychlejších písárek; procesor se tedy z jeho hlediska velmi dlouhou dobou 'fláká', nedělá nic jiného než že čeká, až se něco stane.

Pozorného čtenáře již asi napadlo totéž, co napadlo tvůrce kooperativních multitaskových operačních systémů: tento volný čas můžeme využít tak, že přepneme na jiný proces, který 'má co dělat' (kopíruje třeba soubory z diskety na pevný disk). Přitom nastavíme požadavek na aktivaci prvního procesu (toho, který čeká na stisk klávesy - nadále mu budeme říkat **proces na popředí /foreground/**); druhý proces tedy poběží jen 'chvilku', a jakmile poprvé sám zavolá 'přerušovací' službu, dojde opět k přepnutí kontextu a první proces bude znova čekat na příchod znaku z klávesnice. Není-li dosud žádný k dispozici, může se opět aktivovat jiný proces, který 'má co dělat' (tentýž jako minule nebo jiný), a vše se opakuje. Těmto procesům, které vyplňují dobu, kdy proces na popředí na něco čeká, budeme říkat **procesy na pozadí (background)**.

Naprogramujeme-li právě popsaný mechanismus (a vyřešíme-li řadu technických detailů, kterými se na této úrovni nebude zábývat), vytvořili jsme skutečný **kooperativní multitaskový operační systém**, jakým jsou např. všechny novější verze systému počítačů Apple Macintosh nebo MS Windows.

Je vhodné rozebrat ihned výhody a nevýhody takového systému. Jako obvykle se nejprve podíváme na jeho výhody:

- Kooperativní operační systém má samozřejmě všechny výhody systému s neomezeným přepínáním programů. Těch není málo, připomeňme si jejich shrnutí na konci odstavce 5.2.1.
- Oproti systému s přepínáním programů kooperativní systém lépe využívá procesor, protože může dobu, kdy procesor čeká, vyplnit zpracováním jiného procesu.

Bohužel, nevýhod tohoto systému je daleko víc a jsou poměrně **závažné**:

- Zpomalení procesu na popředí není pouhým rizikem, ale naprostou jistotou. Uvědomme si, že průměrná doba odezvy na stisknutí klávesy bude odpovídat průměrnému intervalu mezi dvojím voláním 'přerušovací' funkce z procesu na pozadí - záleží tedy na tom, jak je tento proces naprogramován. Téměř jistě však bude tato doba delší, než by se uživateli počítače líbilo.

- Důsledkem právě popsané nevýhody je i to, že nemůžeme multitaskový aparát použít na realizaci paralelních úloh (správa sítě, komunikace prostřednictvím sériového rozhraní apod.). Jakmile proces přestane být aktivním, může do jeho další aktivace uplynout poměrně hodně času; to si však zmíněné úlohy nemohou dovolit.
- Představme si, že v aktivním procesu je chyba, která vede k nekonečné smyčce. Aktivní proces pak nikdy nezavolá 'přerušovací' službu a celý operační systém se všemi ostatními procesy má smůlu. Kooperativní multitaskový systém proto není a v principu nemůže být bezpečný.
- Programátoři, kteří vytvářejí aplikace pro kooperativní systém, musejí dodržovat řadu konvencí, z nichž některé výrazným způsobem ztěžují práci. Snad nejnepříjemnější je nutnost jakoukoli časově náročnější Činnost rozdělit do kratších úseků, oddělených voláním 'přerušovací' služby (tentо požadavek samozřejmě vyplývá z první nevýhody). Zmíněné konvence jsou natolik omezující, že velké množství programátorů raději rezignuje na výhody multitaskingu a označí své aplikace jako 'foreground only' - zakáží tedy operačnímu systému, aby takový proces aktivoval na pozadí. Máme-li většinu programů tohoto typu (a u počítačů Apple Macintosh tomu tak skutečně je), degraduje se kooperativní multitaskový systém vlastně na pouhý systém s neomezeným přepínáním programů.
- Jak uvidíme v dalších odstavcích, nemá kooperativní multitaskový systém proti preemptivnímu systému ani výhodu jednoduchosti. Složitost obou druhů systémů je zhruba stejná; o něco komplikovanější přepínání kontextu v preemptivním systému zdaleka vyvází složitost procesů, které pracují pod kooperativním systémem (viz minulá nevýhoda - stejné konvence pochopitelně platí i pro systémové procesy) a nutnost použít odlišný aparát pro realizaci paralelních úloh (viz druhá nevýhoda).

5.2.4 Preemptivní multitasking

Zamyslíme-li se podrobně nad nevýhodami kooperativních multitaskových operačních systémů, vidíme, že základem všech jejich nevýhod je právě kooperativnost - tj. to, že na multitaskingu musí s operačním systémem spolupracovat každý proces.

Chceme-li se těchto nevýhod zbavit, potřebujeme operační systém, který dokáže přerušit každý proces - i takový, který tomu nijak nenapomůže. Potřebujeme operační systém, který umožní důležitějším procesům - jako je proces na popředí nebo procesy obsluhující síť či jinou komunikaci - získat procesor přednostně, jakmile dají systému na vědomí, že jej mohou upotřebit. Potřebujeme zkrátka preemptivní - česky snad nejlépe 'předbíhací' - operační systém.

Jak ale docílit toho, aby bylo možné přerušit kterýkoli proces na kterémkoli místě, aniž by nám s tím proces pomáhal? Podívejme se zpátky do textu, kde jsme narazili na nutnost spolupráce procesu: bylo to asi v polovině odstavce 5.2.2, když jsme minimalizovali rozsah kontextu.

Dejme se tedy tentokrát jinou cestou, vzdejme se jedné výhody - velmi malého kontextu - za získání výhody daleko větší - preemptivního multitaskingu. Zařadíme snad tedy do kontextu obsah všech registrů mikroprocesoru, stav aritmetického koprocessoru, stav blitteru, stav I/O procesorů, stav všeho možného? To by přece jen nebylo možné; musíme tedy vymyslet nějakou 'fintu', jak udržet kontext v rozumném rozsahu, aniž bychom museli spolupracovat s procesem.

Řešení je překvapivě jednoduché: do kontextu uložíme pouze stav procesoru (a případného aritmetického koprocessoru) a s ostatními prvky systému povolíme pracovat vždy jen jedinému procesu. Jen jediný proces tedy smí obsluhovat blitter, jen jediný proces smí komunikovat s I/O procesorem, který se stará o komunikaci prostřednictvím sériového rozhraní a podobně. Stav těchto prvků samozřejmě nemusíme nikam ukládat - v případě, že je jejich vlastník přerušen, nic se s nimi nebude dít, takže po obnovení práce je vlastník nalezne v nezměněném stavu.

Operační systémy

Tento přístup - tj. vyhrazení prostředků pro jediný proces - však má i jednu významnou nevýhodu: přináší totiž možnost zablokování (tento pojem podrobněji vysvětlíme v odstavci 5.4.4). Moderní operační systémy proto často zakazují užívání takovýchto prostředků všem procesům kromě speciálně vyhrazených procesů systémových; ty pak ostatním procesům nabízejí služby, které nepřímo umožní využití toho kterého zařízení (takovýmto procesům se říká **servery** a ještě se jimi budeme podrobněji zabývat). Servery pochopitelně udržují informace o každém ze svých **klientů** (tj. procesů, které požadují jejich služby) zvlášť; při přepínání kontextu tedy žádný problém nenastává.

Vytvoříme-li tedy servery pro obsluhu některých prostředků, zavedeme-li povinné vyhrazení ostatních prostředků jen pro jeden proces a upravíme-li přepínání kontextu pro uložení kompletních informací o procesoru, máme vše připraveno pro implementaci preemptivního multitaskového operačního systému. Přepnutí kontextu za těchto podmínek totiž můžeme vyvolat kdykoli se nám zachce - v rámci kterékoli ze systémových služeb nebo třeba v rámci obslužné rutiny libovolného přerušení:

- Je zapotřebí zajistit obsluhu sítě a odpovídající server není aktivní? Nevadí - technické vybavení nás na tuto situaci upozorní přerušením a v rámci jeho obslužné rutiny operační systém přidělí síťovému serveru procesor⁴⁰.
- Proces na popředí čeká na znak, takže není aktivní (běží nějaký proces na pozadí), a uživatel stiskl nějakou klávesu? Není třeba se bát dlouhé odezvy - stisknutí klávesy generuje přerušení a v rámci jeho obslužné rutiny operační systém přidělí procesor procesu na popředí.
- Pracuje již aktivní proces příliš dlouho, aniž by zavolal jakoukoli systémovou službu a umožnil tak přepnutí kontextu? Žádný problém - součástí technického vybavení každého počítače je časovač obvod, který generuje přerušení v pravidelných intervalech. V rámci tohoto přerušení může operační systém přidělit procesor procesu, který je právě na řadě.

⁴⁰V dalších kapitolách uvidíme, že v praxi se ovládání periferních zařízení - mezi které patří samozřejmě i síť - řeší trochu složitějším, ale efektivnějším způsobem.

5.2.5 Sdílení času

V poslední poznámce minulého odstavce jsme narazili na princip moderních operačních systémů, který zajišťuje iluzi skutečně současného běhu několika procesů bez zřetelného 'střídání'. Jedná se o **sdílení času** (anglický název **time slicing** vystihuje situaci o něco přesněji).

Sdílení času je v podstatě nesmírně jednoduché: operační systém prostě nedovolí žádnému procesu běžet déle než po určitý časový interval. Jestliže proces běží příliš dlouho, odebere mu operační systém v rámci přerušení časovače procesor a přidělí jej dalšímu procesu. Teprve když se vystřídají všechny procesy, dostane procesor znovu původní proces.

Vzhledem k tomu, že doba, po kterou smí proces běžet, je z hlediska pomalých lidských smyslů velice krátká (obvykle se jedná o desítky až stovky milisekund), je iluze současného běhu více procesů zcela dokonalá.

Poznamenejme, že v průměrném případě se sdílení času neuplatní příliš často - běžný program totiž v přiděleném časovém intervalu stihne udělat vše, co sám udělat mohl, a vyvolá některou ze systémových služeb zajišťujících komunikaci s uživatelem, práci s diskem a podobně. Součástí těchto služeb bývá čekání na nějakou událost, takže operační systém pro dobu čekání stejně přidělí procesor někomu jinému.

5.2.6 Implementace

Implementace přepínání kontextu je poměrně jednoduchá. V případě kooperativního systému se skutečně nejedná o nic jiného, než o změnu obsahu ukazatele zásobníku; budeme se proto zabývat přepínáním kontextu v preemptivním systému (ani to však není příliš složité).

Jednou z nejjednodušších implementací je implementace použitá v operačním systému PC-XINU - rozšíření jazyka Turbo C umožnila napsat celou rutinu ve vyšším jazyce (ti, kdo neznají Turbo C, mohou nahlédnout do dodatku C).

Operační systémy

Vstupními parametry služby 'ctxsw' jsou adresy ukládacího prostoru pro starý kontext ('lastsva') a pro nový kontext ('newsva'). Kontext je deklarován

```
typedef unsigned kontext[2];
```

a oba ukazatele jsou typu 'unsigned *'. Implementace totiž na ukládací prostor umísťuje pouze ukazatel zásobníku (ten je u mikroprocesorů řady Intel 80x86 čtyřbytový, složený ze dvou dvoubytových registrů SS a SP). Celý zbytek kontextu se uloží na zásobník (a při přidělení procesoru se odtamtud obnoví); k tomu je využita funkce typu 'interrupt', která v Turbo C kontext ukládá a obnovuje automaticky:

```
/* ctxsw.c - ctxsw */
#include <kernel.h>
#include <proc.h>

void interrupt ctxsw(void)
{
    lastsva[0]=_SP;
    lastsva[1]=_SS;
    _SP=newsva[0];
    _SS=newsva[1];
}

/* end of file */
```

Nejčastěji se přepínání kontextu implementuje v assembleru. Pak je běžné ukládat celý kontext do ukládací oblasti a zásobník nevyužívat. Jako příklad můžeme uvést implementaci rutiny 'ctxsw' v ST-XINU (tam se zásobník používá pouze pro uložení stavového registru a registru A0). Funkce je volána prostřednictvím aparátu přerušení a ukončena instrukcí RTE; to automaticky uloží na zásobník (a zase z něj obnoví) i obsah stavového registru. Snad ještě vhodné poznamenat, že u mikroprocesorů Motorola je ukazatelem zásobníku registr A7⁴¹:

⁴¹Pro ty, kdo znají dobře mikroprocesory Motorola dodejme, že systém ST-XINU vůbec nepoužívá uživatelský režim mikroprocesoru, celou dobu se tedy pracuje pouze s ukazatelem zásobníku AT.

```
; ctxsw.s - ctxsw

global ctxsw

ctxsw: move.l a0,-(a7)
       move.l lastsva,a0
       move.l d0-d7/a1-a7,a0
       move.l newsva,a0
       move.l a0,d0-d7/a1-a7
       move.l (a7)+,a0
       rte
```

Zajímavým způsobem je přepínání kontextu řešeno u moderních mikroprocesorů řady Intel 80x86. Velmi orientačně řešeno, integrovaná jednotka řízení paměti udržuje i kontexty jednotlivých procesů; k přepnutí kontextu pak dojde zcela automaticky při volání funkce uložené v adresovém prostoru jiného než aktivního procesu.

Základní myšlenka není nezajímavá, její konkrétní implementace se však příliš nepodařila. Celý systém tzv. bran, který zmíněné efekty umožňuje, je velmi komplikovaný (zájemce nalezne všechny podrobnosti v [386]) a neefektivní; např. srovnání systémů XENIX V/286 a SYSTEM V/AT ukázalo více než čtyřnásobnou dobu odezvy druhého z nich vůči prvnímu při volání jádra; XENIX V využívá přitom 'tradiční' přepínání kontextu, zatímco SYSTEM V pracuje s automatickým přepínáním kontextu prostřednictvím zmíněných 'bran' (viz [UXPC]).

5.3 Správa procesů

"Operační systém, to je něco jako socialistická ekonomika: samé plánování a samá fronta"

Roderik Plevka

Podobně, jako se správce paměti staral o obsluhu operační paměti, sleduje správce procesů procesy v systému a řídí jejich chování. Správce procesů

plánuje, který proces bude aktivován a také rozhoduje, bude-li právě aktivní proces přerušen.

Aby mohl tyto úlohy řešit efektivním způsobem, udržuje správce několik front procesů, odpovídajících jednotlivým situacím, ve kterých proces na něco čeká. Z těchto front pak správce vybírá proces, který bude aktivován při odstranění příčiny čekání. Efektivní práce s frontami je proto základním požadavkem pro efektivitu celého správce procesů - podívejme se na ni proto podrobněji.

5.3.1 Správa front

Správce front musí být schopen udržovat větší počet prioritních front procesů a nejméně jednu frontu speciálního typu, kterou nazýváme delta list.

Fronta jako taková je objekt, na který můžeme aplikovat následující služby:

- **vytvoření fronty** zajistí vznik nového objektu typu 'fronta', se kterým můžeme dále pracovat. Když jej již nepotřebujeme, můžeme jej zrušit službou **zrušení fronty**.
- **umístění do fronty** je operace, při které se do fronty zařadí nový objekt (v našem případě proces, respektive jeho identifikační číslo, pod kterým je proces správci systému znám). Objekt nemusí být nutně zařazen na konec fronty - to závisí na konkrétním typu fronty, se kterou pracujeme.
- **odebrání z fronty** je jednoduchá operace, která vrátí první objekt ve frontě a z fronty jej odstraní.

V operačních systémech se používají fronty dvou typů:

- **Prioritní fronta** - při umístění do fronty je dalším parametrem tzv. priorita procesu. Správce front pak proces zařadí do fronty před všechny procesy s nižší prioritou a za všechny procesy s prioritou stejnou nebo vyšší.

- Fronta typu **delta list** slouží k ukládání procesů, které cekají na uplynutí časového intervalu.

Při umístění do fronty je dalším parametrem časový interval, po kterém má být proces opět aktivován. Správce front pak proces zařadí za všechny procesy, jejichž časový interval je kratší, a před všechny procesy s delším intervalem.

Intervaly uložené v této frontě musí být pravidelně zkracovány (nejčastěji na základě přerušení časovače). Aby správce front nemusel frontu procházet celou, ukládá u každého procesu pouze rozdíl jeho intervalu a intervalu předchozího. Pak stačí, zkracuje-li pouze interval prvního procesu; po jeho vynulování může proces z fronty vyjmout a pokračovat s dalším procesem. Proto se tato fronta nazývá 'delta list'.

Fronty je samozřejmě možné implementovat jako spojové seznamy; existuje však jednodušší řešení. V následujícím textu uvidíme, že žádný proces nemůže být zároveň ve více frontách; stačí proto vytvořit tabulku, ve které bude pro každý proces odkaz na předchůdce a následníka, a realizovat fronty pomocí této tabulky. Pak je také možné vyžádat si odstranění určitého procesu z fronty, aniž bychom se starali o to, ve které frontě doopravdy proces je.

Nebudeme se zde zabývat detailem implementace (ta je ostatně velmi jednoduchá); pro lepší srozumitelnost dalších příkladů ale uvedeme služby správce front v XINU⁴²:

```
/* q.h - firstid,firstkey,isempty,lastkey,nonempty */

isempty(q)          /* je fronta prázdná? */
nonempty(q)          /* je fronta neprázdná? */
firstkey(q)          /* nejmenší \ priorita ve frontě ... */
lastkey(q)           /* největší / ... nebo čas v delta listu */
firstid(q)           /* první proces ve frontě */

enqueue(p,q);       /* zařazení procesu p na konec fronty q */
```

⁴²Oproti skutečným zdrojovým textům XINU budeme pro lepší přehlednost v příkladech vypouštět některé nepodstatné technické detaile (zde je to např. potřeba určovat někdy frontu pomocí její hlavičky, jindy pomocí jejího posledního prvku).

Operační systémy

```
insert(p,q,key); /* zařazení s prioritou key */
insertd(p,q,time);/* zařazení do delta listu */
dequeue(p); /* odstranění procesu p z fronty */
getfirst(q); /* zjištění prvního procesu ve frontě */
getlast(q); /* zjištění posledního procesu ve frontě */
newqueue(void); /* vytvoření nové fronty */

/* end of file */
```

5.3.2 Tabulka procesů

Operační systém musí udržovat tabulku procesů, ve které jsou uloženy všechny potřebné (i nepotřebné) informace o každém procesu.

Musí zde být samozřejmě priorita procesu, ukládací oblast pro jeho kontext a několik dalších údajů, se kterými se ještě seznámíme. Součástí tabulky procesů navíc bývá několik údajů, které operační systém k ničemu nepotřebuje, usnadňují však případnou statistiku a mohou sloužit i pro ladící účely - jedná se obvykle o jméno procesu, informace o způsobu, jakým byl vytvořen a době jeho vytvoření a podobně.

Tabulka procesů v operačním systému XINU vypadá takto:

```
/* proc.h */

/* stavy procesu (viz odstavec 5.3.3) */

#define PRCURR  '\01'      /* proces právě běží */
#define PRFREE   '\02'     /* volné místo v tabulce procesů */
#define PRREADY  '\03'     /* proces je v ready frontě */
#define PRRECV   '\04'     /* proces čeká na zprávu */
#define PRSLEEP  '\05'     /* proces čeká na uplynutí časového
                           /* intervalu */
#define PRSUSP   '\06'     /* proces je suspendován */
#define PRWAIT   '\07'     /* proces čeká na semafor

struct pentry {
    unsigned char pstate;    /* stav procesu: PRCURR, ap. */
    short pprio;             /* priorita */
    unsigned long svarea;    /* ukládací oblast pro kontext */
    short psem;               /* semafor pro stav waiting */
    short pmsg;              /* přijatá zpráva */
```

```

short phasmsg;           /* nenula je-li zpráva validní */
unsigned *pbase;          /* zásobník */
unsigned pstklen;         /* jeho délka */
unsigned *plimit;         /* jeho limit */
char pname[PNMLEN];      /* jméno procesu */
short pargs;              /* počet parametrů */
void *paddr;              /* startovní adresa */

};

extern struct pentry proctab[];

/* end of file */

```

Tabulka procesů je umístěna v poli 'proctab'. XINU pak používá index do tohoto pole jako identifikační číslo procesu.

5.3.3 Stavy procesu

Podívejme se nyní na operační systém z hlediska procesu. Proces nejprve vznikne, pak je mu třeba přidělen procesor a proces se nějakou dobu o procesor střídá s dalšími procesy na základě sdílení času. Pak proces chvíli komunikuje s uživatelem - takže většinu doby čeká na stisknutí klávesy. Potom si proces vyžádá nějaké služby od některých serverů; přitom se může stát, že proces bude čekat, než mu server pošle zprávu. Nakonec se třeba proces 'rozhodne' čekat až do zítřka - vyžádá si tedy od operačního systému aby jej za několik hodin aktivoval a 'usne'.

Aby se v tom správce procesů vůbec vyznal, udržuje u každého procesu informaci o jeho **stavu**. Prostudujme si obr. 14 - jsou tam uvedeny všechny stavy procesu v běžném systému (konkrétně se jedná o XINU) a jsou tam vyznačeny i jejich možné změny.

Po vytvoření se proces ocitne ve stavu **suspended**. To je speciální stav 'procesů', o které nemá nikdo zájem'. Ze stavu 'suspended' se proces dostane jen na explicitní požadavek jiného procesu (nebo operačního systému). Proces se může do stavu 'suspended' vrátit, opět na základě explicitního požadavku.

Procesy, které mohou být vybrány pro přidělení procesoru, jsou ve stavu **ready**. Správce procesů všechny takové procesy udržuje v prioritní tzv. ready frontě;

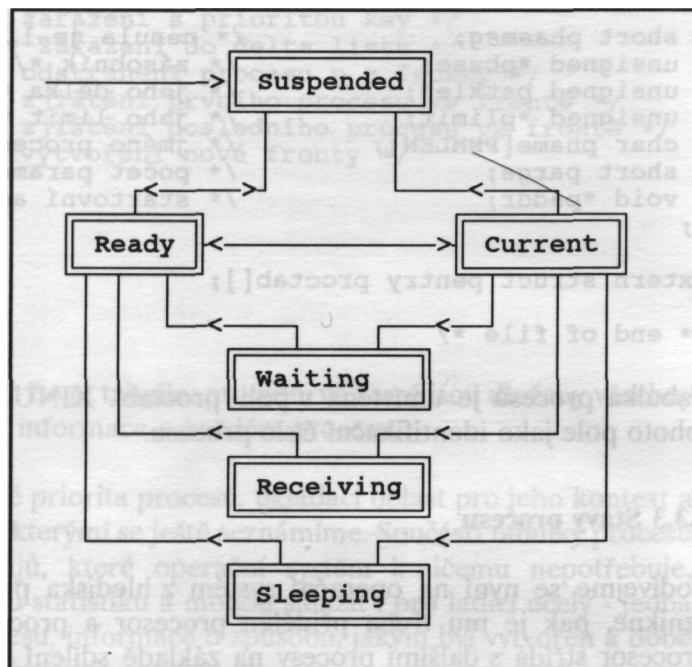
Operační systémy

jakmile je zapotřebí některému procesu přidělit procesor, vybere se prostě první proces z ready fronty.

Proces, kterému je přidělen procesor, je odstraněn z ready fronty a dostane se do stavu **current** - to je proces, který skutečně běží. Ve stavu 'current' může být v jednom okamžiku samozřejmě jen jediný proces.

Ze stavu 'current' se může proces dostat několika způsoby; součástí všech je pochopitelně přepnutí kontextu a přidělení procesoru prvnímu procesu z ready fronty:

- Jestliže byl procesu odebrán procesor, protože proces čeká na nějakou periferní operaci, dostane se proces do stavu **waiting**. V tomto stavu jsou procesy uloženy v tolka frontách, na kolik typů událostí lze čekat⁴³.
- Čeká-li proces na přijetí zprávy od jiného procesu, dostane se do stavu **receiving**. Procesy v tomto stavu nejsou umístěny v žádné frontě; jak uvidíme níže při popisu aparátu zpráv, nebylo by to také k ničemu dobré.



obr. 14: stavy procesu

⁴³Podrobněji si stav 'waiting' vysvětlíme v odstavci zabývajícím se synchronizací procesů v oddílu o tzv. semaforech.

- Čeká-li proces na uplynutí určitého časového intervalu, dostane se do stavu **sleeping**. Všechny procesy v tomto stavu jsou uloženy ve společném delta listu.

Z kteréhokoli stavu je navíc možné proces zrušit (tedy vyřadit ze systému). Složitější operační systémy mívají navíc stav **dead**, do kterého je přemístěn proces po požadavku na zrušení; proces v tomto stavu zůstane, dokud není vyřazení ze systému ukončeno. V XINU je vyřazení procesu ze systému rychlou jednorázovou událostí, takže stav 'dead' zde není zapotřebí.

5.3.4 Nulový proces

Představme si, že jsou všechny procesy, až na jedený, ve stavech 'suspended', 'waiting', 'sleeping' nebo 'receiving', a tu najednou ten jediný proces (který až dosud pochopitelně běžel bez jakéhokoli přerušení) potřebuje čekat na stisknutí klávesy.

Správce procesů jej převede do stavu 'waiting', odebere mu procesor - a co s tím procesorem bude dělat? Zdá se, že budeme muset přeprogramovat funkci 'ctxsw' tak, aby bylo možné říci 'nikomu procesor nepředávat' (např. nulovou hodnotou v ukazateli 'newsval'). Funkce pak může procesor třeba pozastavit, pokud to jeho technické vybavení umožňuje ...⁴⁴

Takové řešení je v principu možné, má však řadu nevýhod. Služby správce procesů - počínaje samotnou funkcí 'ctxsw' - se nepříjemně zkomplikují. Pokud bychom v době, kdy neběží žádný proces, chtěli místo čekání dělat 'něco rozumného' (např. počítat nějaké statistiky využití systému), zkomplikovaly by se služby ještě víc. Obvykle se proto používá jiné řešení - tzv. **nulový proces**.

Operační systém při inicializaci vytvoří naprosto normální proces s velmi nízkou prioritou (zaručeně nižší, než je priorita všech uživatelských procesů). Tento proces pak nechá normálně běžet; jeho nízká priorita však zajistí, že mu

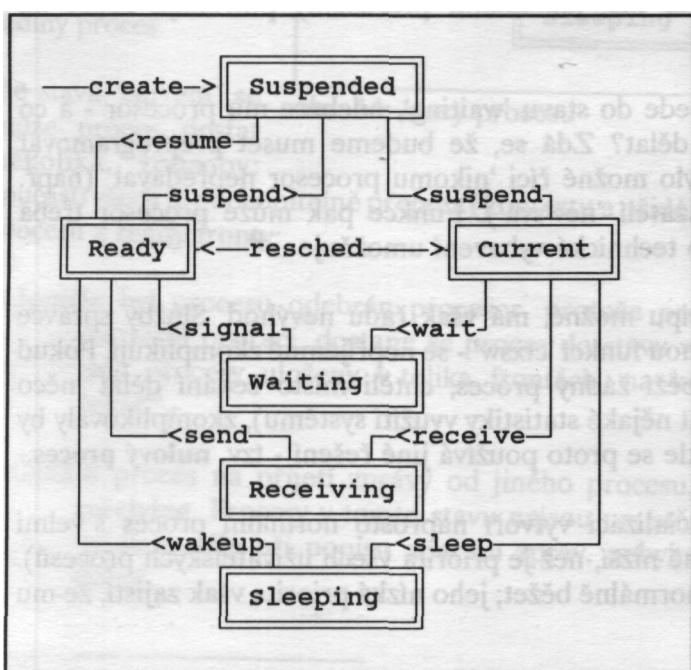
⁴⁴Pozastavený procesor nedělá nic, dokud nedojde k přerušení. Pak začne opět normálně pracovat.

Operační systémy

bude procesor přidělen pouze v případě, kdy všechny ostatní procesy na něco čekají a žádný z nich běžet nemůže. Obsah tohoto procesu bývá velmi triviální, často jen zmíněné pozastavení procesoru ve věčném cyklu. Jediným omezením pak je to, že tento proces nesmí nikdy přejít do jiného stavu než 'ready' nebo 'current'; ono to ale také není zapotřebí.

53.5 Plánování

Musíme si uvědomit, že přepnutí kontextu - respektive služba 'ctxsw', jak jsme se s ní seznámili v odstavci 5.2.6 - je sice základním stavebním kamenem multitaskingu, ani zdaleka však není celým multitaskingem. Operační systém - respektive správce procesů - totiž musí udržovat všechny potřebné informace o procesech, na jejich základě musí rozhodovat, kterému procesu (pomocí služby 'ctxsw') přidělit procesor a kterému ne.



obr. 15: služby správce procesů

Podívejme se na obr. 15. Podobně jako na minulém obrázku vidíme i zde jednotlivé stavy procesů; tentokrát však zde jsou vyznačena i jména služeb správce procesů, které zajistí přechody mezi jednotlivými stavy.

Povšimněme si, že zde nikde není uvedena služba 'ctxsw' - ta je totiž na příliš nízké úrovni a musí být 'obalena' složitější službou

s komplexnější funkcí, (totiž službou 'resched').

Řadu těchto služeb využívá sám plánovač procesů (služba 'resched' je například jádrem všech ostatních služeb zajišťujících přepnutí kontextu). Služby, jejichž jména vidíme na obr. 15, jsou navíc k dispozici ostatním částem operačního systému. Ovladače periferií pak intenzivně využívají služby 'wait' a 'signal' (my se s nimi seznámíme zanedlouho), pro komunikaci se servery budou nejčastěji používány služby 'send' a 'receive' a podobně.

V dalších odstavcích si popišeme funkci služeb z horní poloviny obrázku (až po službu 'resched'); ostatním službám se budeme věnovat později v souvislosti s časováním a se synchronizací procesů. Vzhledem k tomu, že konkrétní příklad často objasní víc než několikastránkové popisy, doplníme popis každé služby její implementací v operačním systému XINU.

5.3.5.1 Služby 'resched' a 'ready'

Základem správce procesů je služba pro 'přeplánování' (reschedule), v XINU nazvaná 'resched'. Ta vezme v úvahu stav procesů a je-li to zapotřebí, zajistí vlastní přepnutí kontextu. Podívejme se na její implementaci v XINU. Je zde použit malý trik - číslo aktivního procesu systém zná (je uloženo v globální proměnné 'currpid'), proto nemusí být v jeho položce v tabulce procesů uložen stav 'current'; namísto toho je tam před voláním služby 'resched' uložen stav, do kterého má aktivní proces po odebrání procesoru přejít:

```
/* resched.c - resched */
#include<conf.h>

#include<proc.h>
#include <q.h>

/*
 * resched - reschedule to highest priority ready process
 * Notes: upon entry, currpid gives current process id.
 *        Proctab[currpid].pstate gives correct NEXT state for
 *        current process if other than PRCURR.
 */
```

```
/*
int resched(void)
{
    register struct pentry *optr; /* ptr to old process entry */
    register struct pentry *nptr; /* ptr to new process entry */

    if ((optr=&proctab[currpid])->pstate==PRCURR) {
        if (lastkey(rdyqueue)<optr->pprio)
            return(OK);
        optr->pstate=PRREADY;
        insert(currpid, rdyqueue, optr->pprio);

        nptr=&proctab[ (currpid=getlast(rdyqueue)) ];
        nptr->pstate=PRCURR;
        preempt=QUANTUM;
        lastsва=(unsigned *) &(optr->svarea);
        newsва=(unsigned *) &(nptr->svarea);
        ctxsw();
        return(OK);
    }
/* end of file */
```

Funkce 'resched' nejprve ověří, je-li vůbec zapotřebí přepínat kontext: je-li 'budoucí' stav aktivního procesu stále 'current' a je-li priorita aktivního procesu větší než největší priorita procesů v ready frontě, může funkce klidně skončit.

Není-li tomu tak, a 'budoucí' stav je stále 'current' je nutné 'přeplánovať' - funkce přeradí dosud aktivní proces do ready fronty.

Zbývající činnost funkce již je jednoduchá: zjistí, kterému procesu má být přidělen procesor a nastaví jeho stav na 'current'. Následující příkaz nastaví čítací intervalu pro sdílení času na plnou hodnotu; dosáhne-li tento čítací při dekrementaci v obslužné rutině přerušení časovače nulové hodnoty, bude opět vyvolána funkce 'resched', takže procesu bude procesor odebrán (existuje-li jiný proces se stejnou nebo vyšší prioritou).

Pak již funkce jen vyvolá vlastní přepnutí kontextu. To tedy znamená, že funkce 'resched' byla vyvolána v rámci starého procesu, ukončena však již je v rámci procesu nového⁴⁵.

(Zkušenější programátor si na tomto místě pravděpodobně uvědomí, že služba 'resched' musí být realizována jako nepřerušitelná. Pokud by totiž došlo 'uprostřed' služby k přerušení - v rámci kterého by samozřejmě mohla být služba 'resched' volána znova - mohlo by dojít k poškození systémových tabulek. PC-XINU tento problém řeší velmi jednoduchým a účinným způsobem: po celou dobu práce kterékoli ze systémových služeb je zakázáno přerušení. Nemusíme se proto na zmíněný problém zatím ohlížet, a můžeme se soustředit na rozbor ostatních úkolů, které je zapotřebí splnit pro implementaci preemptivního multitaskingu. V kapitolách zabývajících se vstupem a výstupem a kritickými sekcmi se k této problematice ještě vrátíme a rozebereme ji podrobněji.)

Uvědomme si také, že díky tomu, že funkce 'insert' zařazuje proces do fronty až za všechny procesy se stejnou prioritou, zajistí pouhé volání funkce 'resched' v pravidelných intervalech střídání všech procesů s nejvyšší prioritou přesně tak, jak to požaduje sdílení času. Tomuto mechanismu cyklické záměny aktivních procesů se někdy také říká '**round robin**'.

Jakkoli je služba 'resched' již na dost vysoké úrovni, aby ji bylo možné přímo používat (jak jsme se již zmínili, je volána v rámci obslužné rutiny přerušení časovače, vyprší-li čítač pro sdílení času), je v praxi nejčastěji používána v jednom konkrétním kontextu - totiž při přidělování procesoru některému z procesů. Vyplatí se proto vytvořit jednu pomocnou službu, interní pro správce procesů; tato služba se bude jmenovat 'ready' a bude implementována takto:

⁴⁵Je-li ovšem 'nový' proces opravdu nový, tj. takový, který dosud neběžel, nebude funkce 'resched' vlastně ukončena vůbec. Zásobník nového procesu je totiž uměle připraven tak, jako by byl nový proces přerušen službou 'ctxsw' před první instrukcí - po přepnutí kontextu se tedy ze služby 'ctxsw' nevrátíme do funkce 'resched', ale přímo na začátek nového procesu.

Operační systémy

```
/* ready.c - ready */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>

/*
 * ready -- make a process eligible for CPU service
 */
int ready(short pid,short resch)
{
    register struct pentry *pptr;

    (pptr=&proctab[pid])->pstate=PRREADY;
    insert(pid,rdyhead,pptr->pprio);
    if (resch)
        resched();
    return(OK);

/* end of file */
```

Funkce této služby je celkem zřejmá - její parametr 'pid' je identifikační číslo procesu, který má být přeřazen do stavu 'ready'. Funkce nejprve nastaví správně jeho stavovou informaci v tabulce procesů, pak jej zařadí do ready fronty a vyvolá přeplánování.

Zbývá jen vysvětlit význam parametru 'resch', to však není žádná záhada. Správce procesů někdy musí zařadit do stavu 'ready' několik procesů najednou. Pokud by pro zařazení každého z nich použil službu 'ready' tak, jak jsme ji právě popsali, došlo by po zařazení každého z procesů k přeplánování. To ale znamená, že hned po zařazení prvního z procesů by byl správce zbaven procesoru a není jisté, kdy by jej znova dostal; po celou dobu by však ostatní procesy čekaly na zařazení, ačkoli jejich prioritá by mohla klidně být vyšší než prioritá prvního zařazeného procesu.

Právě proto byl zaveden parametr 'resch': uložíme-li do něj nulovou hodnotu, zařadí se proces do ready fronty a k přeplánování nedojde. Teprve při zařazování posledního ze skupiny procesů zavoláme službu 'ready's nenulovou hodnotou v parametru 'resch', takže dojde k přeplánování.

Procesy a procesor

Pro lepší čitelnost zdrojového textu se obvykle pro požadavek přeplánování používá symbolická konstanta 'RESCHYES' (definovaná jako jednička), a pro požadavek pouhého zařazení do ready fronty symbolická konstanta 'RESCHNO' (definovaná jako nula).

5.3.5.2 Služby 'resume' a 'suspend'

Jak již víme, je každý čerstvě vytvořený proces ve stavu 'suspended'. Chceme-li jej spustit (tj. převést do stavu 'ready', odkud jej při nejbližším přeplánování může vybrat správce procesů k přidělení procesoru), musíme na něj použít službu 'resume'. Je zřejmé, že služba 'resume' bude obsahovat jen o málo více než vyvolání služby 'ready'; podívejme se na její implementaci:

```
/* resume.c - resume */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*
 * resume -- unsuspend a process, making it ready; return the
 *           process' priority
 */
int resume(short pid)

    struct pentry *pptr;
    int prio;

    if ((pptr=&proctab[pid])->pstate!=PRSUSP)
        return(SYSERR);
    prio=pptr->prio;
    ready(pid,RESCHYES);
    return(prio);
}

/* end of file */
```

Implementace dává za pravdu předpokladům - funkce pouze zkontroluje, nesnažíme-li se náhodou uvolnit proces, který není suspendován, a zajistí vrácení priority uvolněného procesu.

Operační systémy

Snad jediným zajímavým detailem v implementaci této služby je použití proměnné 'prio'. Uvědomme si, že služba 'ready' s parametrem 'RESCHYES' zajistí přeplánování - dříve, než služba skončí, se tedy může stát spousta věcí; může dojít k mnohonásobnému přepnutí kontextu a ostatní procesy mohou změnit obsah systémových tabulek. Po ukončení služby 'ready' se proto již nemůžeme spolehnout na hodnotu 'pptr->pprio'; namísto toho je zapotřebí zapamatovat si tuto hodnotu před vyvoláním služby 'ready' v lokální proměnné a pak ji jen vrátit.

Služba 'suspend', která převede proces do stavu 'suspended', je trochu komplikovanější - musí být totiž schopna korektně zpracovat proces v kterémkoliv ze stavů 'ready' nebo 'current'. Její implementace vypadá takto:

```
/* suspend.c - suspend */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*
 * _____
 * suspend -- suspend a process, placing it in hibernation
 * _____
 */
int suspend(short pid)

    struct pentry *pptr;
    int prio;

    if (pid==NULLPROC ||
        (pptr=&proctab[pid])->pstate!=PRCURR &&
        pptr->pstate!=PRREADY)
        return(SYSERR);
    prio=pptr->pprio;
    if (pptr->pstate==PRREADY) {
        dequeue(pid);
        pptr->pstate=PRSUSP;
    } else {
        pptr->pstate=PRSUSP;
        resched();
    }
    return(prio);

/* end of file */
```

Příkaz 'if' na začátku funkce pouze ověřuje, nejedná-li se o nulový proces a je-li skutečně suspendovaný proces ve stavu 'ready' nebo ve stavu 'current'. Potom služba rozliší oba případy: je-li proces ve stavu 'ready', stačí jej odstranit z ready fronty (a samozřejmě změnit jeho stav na 'suspended').

Jestliže však proces právě běží (to mimochodem znamená, že proces suspendoval sám sebe), musíme provést přeplánování, kterým se procesu odebere procesor. Nyní využijeme trik služby 'resched' a ve stavové informaci aktivního procesu jí předáme požadovaný budoucí stav - totíž 'suspended'.

Je vhodné si uvědomit, že přeplánování samozřejmě opět zapříčiní Velmi dlouhé trvání' služby 'resched' a tedy i služby 'suspend'. Přesně řečeno, volá-li proces službu 'suspend' sám na sebe, vrátí se mu z této služby řízení teprve poté, co jej někdo opět uvolní.

5.3.5.3 Služba 'kill'

Služba 'kill' musí vyřadit proces ze systému. Kromě změn uvnitř správy procesů tedy musí zajistit několik dalších úkolů (jako je uvolnění paměti, ve které byl zásobník procesu).

Další komplikací pro službu 'kill' je to, že ji můžeme použít na proces v libovolném stavu. Služba proto musí stav procesu detekovat a zařídit se podle něj.

```
/* kill.c - kill */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sem.h>
#include <mem.h>

/*
 * kill -- kill a process and remove it from the system
 */
int kill(short pid)
{
```

Operační systémy

```
struct pentry *pptr; /* points to proc. table for pid */

if ((pptr=&proctab[pid])->pstate==PRFREE)
    return(SYSERR);
--numproc;
freemem(pptr->plimit,pptr->pstklen);
switch (pptr->pstate) {
    case PRCURR:
        pptr->pstate=PRFREE; /* suicide */
        resched();
        /* never returns */
    case PRWAIT:
        semaph[pptr->psem].semcnt++; /* fall thru */
    case PRSLEEP: /* fall thru */
    case PRREADY:
        dequeue(pid); /* fall thru */
    default:
        pptr->pstate=PRFREE;
}
return(OK);

/* end of file */
```

První příkaz 'if' pouze ověří, nesnažíme-li se zrušit neexistující proces. Pak služba 'kill' sníží hodnotu globální proměnné 'numproc', ve které operační systém XINU udržuje počet procesů. Dalším krokem je uvolnění paměti, ve které byl uložen zásobník procesu.

Zbývající příkaz 'switch' je v tomto případě nejpohodlnější číst od konce:

- Po odstranění libovolného procesu ('default:') musíme označit patřičnou položku v tabulce procesů jako volnou - uložíme tedy do ní hodnotu 'PRFREE'.
- Jestliže byl rušený proces ve stavu 'sleeping' nebo 'ready', znamená to, že byl umístěn v nějaké frontě. Musíme jej tedy nejprve z fronty odebrat.
- Pro proces, který byl ve stavu 'waiting', platí totéž; navíc však musíme snížit o jedničku informaci o počtu procesů, které čekají na společnou událost (více v odstavci o semaforech).

Nejzajímavější je zrušení aktivního procesu. V takovém případě jen označíme položku v tabulce procesů jako volnou pomocí hodnoty 'PRFREE' a zajistíme přeplánování. Správce procesů při něm procesu odebere procesor a od té chvíle se již o tento proces nemusíme starat, protože nikdy procesor nedostane zpátky (je zrušen, nemůže se tedy dostat do ready fronty). Z hlediska tohoto rušeného procesu tedy služba 'resched' nikdy neskončí (přesněji řečeno, proces přestane existovat dříve, než by služba mohla skončit).

5.3.5.4 Služba 'create'

Vytvoření nového procesu je trochu komplikovanější služba. Uvědomme si, co všechno je zapotřebí zajistit:

- Vyhrazení položky v tabulce procesů a vyplnění informací, které jsou v ní uloženy. Proces se vytváří ve stavu 'suspended'; ušetříme si tak jeho zařazování do front a přeplánování.
- Vyhrazení paměti pro zásobník procesu. K tomu správce procesů samozřejmě využije služeb správce paměti; konkrétní realizace tedy závisí na tom, jak vypadá správa paměti.
- Na zásobníku je nutné vytvořit záznam 'jako by' byl proces přerušen službou 'ctxsw' bezprostředně před jeho první instrukcí, takže po prvním přidělení procesoru se proces spustí opravdu od začátku. V XINU se navíc na zásobník uloží ještě adresa standardní návratové rutiny 'INITRET'; jestliže pak proces skončí instrukcí 'RET', bude vyvolána funkce na adresě 'INITRET' (v ní je obvykle vyvolána služba 'kill', takže proces skutečně skončí).
- Paměť pro data procesu není zapotřebí vyhrazovat - proces si ji v případě potřeby vyžádá od správce paměti sám.
- Program, který bude proces zpracovávat, musí být již uložen v operační paměti. Služba 'create' tedy program nezavede, ale pouze vytvoří proces na základě již zavedeného programu. Komplexní služba pro

—Operační systémy—

zavedení programu (ta samozřejmě bude na vyšší úrovni než v plánovači procesů) bude službu 'create' sama využívat.

V operačním systému XINU je služba 'create' implementována následujícím způsobem. Jejími parametry jsou adresa programu, podle nějž bude proces pracovat, požadovaná velikost zásobníku, požadovaná priorita procesu a parametry procesu; ty služba 'create' uloží na zásobník tak, že budou pro proces přístupné podle běžných konvencí jazyka C⁴⁶.

Služba 'create' je - podobně jako funkce 'ctxsw' - do značné míry strojově závislá. Tentokrát sice nebude zapotřebí explicitně využívat rozšíření Turbo C, setkáme se s nimi však 'implicitně' - pořadí ukládání údajů na zásobník odpovídá pořadí, ve kterém registry procesoru ukládá a čte funkce typu 'interrupt' použitá pro přepínání kontextu. Musíme si také uvědomit, že příklad ukazuje implementaci služby 'create' v PC-XINU; v ST-XINU i v originálním XINU služba samozřejmě vypadá trochu jinak.

Neuvádíme (triviální) implementaci pomocných funkcí 'getstk' a 'newpid'. První z nich využije služeb správce paměti pro vyhrazení paměťového bloku zadané velikosti pro zásobník a vrátí jeho adresu (nebo nulu, není-li dost volné paměti). Druhá vyhledá volné místo v tabulce procesů a vrátí odpovídající index (tedy budoucí identifikační číslo procesu), nebo nulu, je-li tabulka procesů zaplněna.

Význam hodnoty 'INITRET' jsme si již ozrejmili - jedná se o standardní návratovou hodnotu pro případ, že proces skončí instrukcí 'RET'. Další hodnoty 'INITFF', 'INITES' a 'INITDS' určují, co má být při startu procesu zavedeno do registrů procesoru FLAGS, ES a DS. Ostatní registry procesoru budou vynulovány.

⁴⁶Parametry procesu jsou poměrně důležité. Uvědomme si, že často dává velmi dobrý smysl vytvořit více procesů na základě jediného programu - interpreter uživatelských příkazů by například mohl při každém požadavku uživatele na kopírování souborů vytvořit proces, který kopírování zajistí. Takových procesů může být více najednou (vyžadá-li si uživatel další kopírování dříve, než je první ukončeno); všechny budou samozřejmě zpracovávat jedený program (který také bude v operační paměti pouze jednou) a budou se lišit pouze parametry udávajícími co a kam je zapotřebí kopírovat.

```

/* create.c - create */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <mem.h>

/*-----*
 *  create - create a process to start running a procedure
 *-----*/
int create(void (*procaddr)(), unsigned ssize, short priority,
           char *name, short nargs, unsigned *args)

    int pid;          /* stores new process id */
    struct pentry *pptr; /* pointer to proc. table entry */
    int i;
    unsigned *saddr;   /* stack address */

    if (priority<1 || priority>=MAXPRIO ||
        ((saddr=getstk(ssize))==0) || (pid=newpid())==SYSERR)
        return(SYSERR);
    numproc++;
    (pptr=&proctab[pid])->pstate=PRSUSP;
    for (i=0;i<PNMLEN && (pptr->pname[i]==name[i])& i++);
    pptr->pprio=priority;
    pptr->plimit=saddr; /* lowest stack extent */
    pptr->pbase=saddr+ssize-sizeof(*saddr);
    pptr->pstklen=ssize;
    pptr->pargs=nargs;
    pptr->paddr=procaddr;
    saddr=pptr->pbase; /* stack grows down */
    for (i=0;i<nargs;i++) /* machine dependent; copy args ... */
        *saddr-=args[i]; /* ... onto created process' stack */
    *(unsigned long *)--saddr=(unsigned long)INITRET;
                                /* push on return address (32bit) */
    saddr--;
    /* after INITRET addr ... */
    *saddr-=INITFF; /* ... push on init flags */
    *(unsigned long *)--saddr=(unsigned long)procaddr;
                                /* push on process addr (32bit) */
    saddr--;
    for (i=0;i<4;i++)
        *saddr-=0; /* AX - DX set to zeros */
    *saddr-=INITES;
    *saddr-=INITDS;
    for (i=0;i<3;i++)
        *saddr-=0; /* SI,DI,BP set to zeros */
    pptr->svarea=(unsigned long)++saddr;

```

Operační systémy

```
    /* stack points to last saved */
    return(pid);
}

/* end of file */
```

První příkaz 'if' zkонтroluje, je-li možné vůbec proces vytvořit. Nejprve ověří, je-li požadovaná priorita v rozsahu přípustných priorit systému XINU; pak se pokusí alokovat blok paměti pro zásobník a nakonec zkusí vyhradit místo v tabulce procesů. Pokud se cokoli z toho nepodaří, funkce skončí a vrátí symbolickou hodnotu 'SYSERR' jako informaci, že proces se nepodařilo vytvořit⁴⁷.

Potom funkce 'create' inkrementuje obsah globální proměnné 'numprod', ve které je uložen počet procesů v systému.

Následujících osm řádků zdrojového textu je věnováno vyplnění položky v tabulce procesů.

Zbytek služby 'create' vytváří na zásobníku požadovaný záznam. Podívejme se postupně na jednotlivé kroky při tvorbě tohoto záznamu; pomůže nám obr. 16.

Nejprve na zásobník zkopírujeme všechny parametry procesu, a pak 'pod ně' uložíme adresu 'INITRET'. Tato část zásobníku je na obrázku označena jedničkou. Ti, kdo znají konvence volání funkcí v jazyce C mohou potvrdit, že se jedná o standardní záznam, který na zásobník ukládá volající. Proces je tedy vlastně uměle přiveden do stejného stavu, jako kdyby byl volán z místa těsně před adresou 'INITRET' se všemi parametry.

Další úsek zásobníku, označený na obrázku dvojkou, je standardní záznam ukládaný mikroprocesorem na zásobník při přerušení. Pro nás je důležité, že funkce typu 'interrupt' v Turbo C je ukončena instrukcí pro návrat z přerušení, která tento záznam interpretuje správným způsobem (tj. uloží hodnotu 'INITFF' do stavového registru a předá řízení na adresu, uloženou v záznamu - spustí tedy vlastně proces).

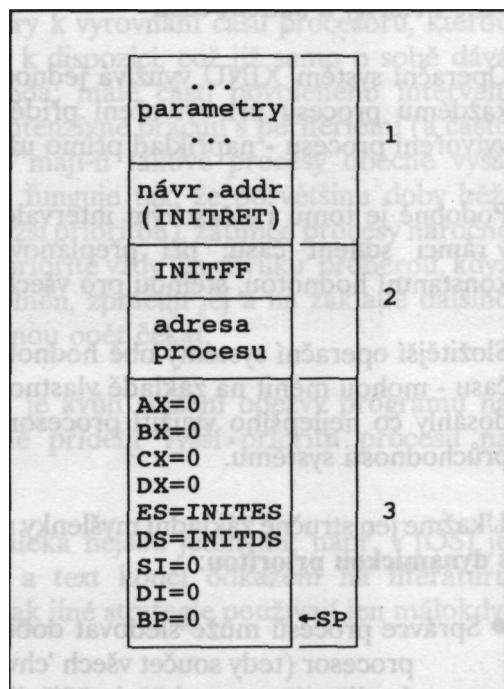
⁴⁷Mimochodem, v tomto místě je ve funkci 'create' drobná chyba. Dokážete ji odhalit dříve, než si na konci tohoto odstavce přečtete v čem spočívala?

Zbývající úsek zásobníku, označený trojkou, obsahuje registry procesoru tak, jak by je uložila funkce typu 'interrupt' - to samozřejmě znamená, že při ukončení funkce 'ctxsw', která je typu 'interrupt', budou hodnoty automaticky uloženy do správných registrů. Hodnota ukazatele zásobníku, označená na obrázku 'SP', je zapsána do ukládací oblasti pro kontext v tabulce procesů; odtud ji pak funkce 'ctxsw' při přepínání kontextu odeberε a uloží do skutečného ukazatele zásobníku.

A chyba, o které jsme se zmínili v poznámce 47? Představme si, že se podaří alokovat paměť pro zásobník, ale pak pomocná funkce 'newpid' nenajde žádné volné místo v tabulce procesů. Služba 'create' skončí, ale blok paměti zůstane alokován - což je jistě špatně. Chytrý správce paměti sice po nějakém čase zjistí, že tento blok nikomu neslouží a opět jej automaticky uvolní; i v tom případě ale zůstane paměť určitou dobu zbytečně vyhrazena. Příkaz 'if' by tedy měl v takové situaci využít službu 'freemem' pro uvolnění paměti:

```
if (priority<1 || priority>=MAXPRIO ||  
    ((saddr=getstk(ssize))==0) || (pid=newpid())==SYSERR) {  
    if (saddr) freemem(saddr,ssize);  
    return(SYSERR);  
}
```

a navíc je zapotřebí proměnnou 'saddr' při deklaraci inicializovat nulou (proč asi?).



obr. 16: zásobník po vytvoření procesu

53.6 Priority a sdílení času

Operační systém XINU využívá jednoduchý systém statických priorit, kdy je každému procesu při vytvoření přidělena priorita (někým, kdo si vyžádal vytvoření procesu - například přímo uživatelem), která se nadále již nemění.

Podobně je tomu i s časovým intervalom, který má každý proces k dispozici v rámci sdílení času: při přeplánování je čítač 'preempt' inicializován konstantní hodnotou, stejnou pro všechny procesy.

Složitější operační systémy obě hodnoty - tj. prioritu i délku intervalu sdílení času - mohou měnit na základě vlastností a chování jednotlivých procesů, aby dosáhly co nejlepšího využití procesoru a periferních zařízení a co nejvyšší průchodnosti systému.

Ukažme jen stručně základní myšlenky několika nejčastěji využívaných systémů s **dynamickou prioritou**:

- Správce procesů může sledovat dobu, po kterou již měl proces k dispozici procesor (tedy součet všech 'chvilek', kdy byl proces ve stavu 'current'). Jestliže již proces běžel příliš dlouho, tj. je-li tato doba delší než určitá hranice, která může být jedním z parametrů procesu, sníží se priorita tohoto procesu.

Důsledkem takovéto strategie je, že ty nejnáročnější procesy běží velmi dlouho (ale to se u nich ostatně dalo očekávat), zatímco kratší procesy jsou hotovy relativně dříve (protože 'staré' rozsáhlé procesy již mají nižší prioritu).

Poznamenejme, že sledování celkového času práce procesu je velmi šikovná věc a hodí se mimo jiné i k automatickému zrušení procesů, které běží mnohem déle, než jejich tvůrce očekával (což téměř vždy znamená, že v programu je chyba - například nekonečný cyklus).

- Při 'spravedlivé' strategii správce procesů při přepínání kontextu sleduje, vyčerpal-li aktivní proces celý interval sdílení času nebo ne. Každému procesu pak přidělí tím větší prioritu, čím menší část tohoto intervalu proces ve skutečnosti využil.

Tato strategie vede do jisté míry k vyrovnaní času procesoru, kterou má v dohromady každý proces k dispozici, což již samo o sobě dává smysl. Je zde ale i další výhoda: malé části povoleného intervalu obvykle čerpají procesy, které intenzívne pracují s periferiemi (a často na ně tedy čekají). Je dobré, mají-li takové procesy obecně vyšší prioritu - operační systém pak funguje tak, že po většinu doby běží výpočetně náročné procesy (s nižší prioritou), zatímco procesy náročné na vstup/výstup se díky vysoké prioritě vždy 'na chvílku' proberou, když je jejich poslední požadavek splněn, zpracují jej a na základě dalšího požadavku na vstup/výstup začnou opět čekat.

- V interaktivním operačním systému je kvůli snížení odezvy programu na akce uživatele obvykle vhodné přidělit vyšší prioritu procesu na popředí.

Popsané strategie samozřejmě ani zdaleka nejsou jedinými; např. v [OS] je uvedeno osm alternativních strategií a text končí odkazem na literaturu obsahující další možnosti. V praxi se však jiné strategie používají jen málokdy.

5.3.7 Správa času

Součástí správce procesů je kromě správy front i poměrně jednoduchá správa času. Jejím úkolem je umožnit procesům, aby se vzdaly možnosti přidělení procesoru po nějakou předem určenou dobu. Po uplynutí této doby musí správa času 'uspané' procesy opět aktivovat.

Správce času převede proces, který jej požádal o 'uspání', do stavu 'sleeping', a zařadí jej do delta listu. V pravidelných intervalech (na základě přerušení časovače) pak správce sleduje delta list, není-li již načase z něj odebrat některé procesy a převést je opět do stavu 'ready'.

53.7.1 Služba sleep

Služba 'sleep' převede proces do stavu 'sleeping'. Patří mezi nejjednodušší služby vůbec - zařadí pouze dosud aktivní proces do delta listu ('clockqueue'), změní jeho stav na 'sleeping' a vyvolá přeplánování.

Služba používá globální proměnnou 'currpid', se kterou jsme se již seznámili a která obsahuje číslo aktivního procesu a nastavuje globální proměnné 'slnempty' a 'slstop', které podrobněji popíšeme zanedlouho v rámci služby 'wakeup' (vysvětlíme si i smysl poznámky u nastavení proměnné 'slstop'):

```
/* sleep.c -- sleep */

#include "kernel.h"
#include "proc.h"
#include "q.h"
#include "sleep.h"

/*
 * sleep -- delay the caller for a specified time
 */
void sleep(unsigned for_time)
{
    if (for_time!=0) {
        insertd(currpid,clockqueue,for_time);
        slnempty=1;
        slstop=&firstkey(clockqueue); /* ! implementace ! */
        proctab[currpid].pstate=PRSLEEP;
        resched();
    }
}

/* end of file */
```

Čas, po který chce proces 'spát', a který předá službě 'sleep' pomocí parametru 'for_time', je určen v časových jednotkách, které závisejí na konkrétní implementaci. V nejjednodušším případě může časová jednotka odpovídat intervalu přerušení časovače; chceme-li použít delší časovou jednotku, můžeme interval prodloužit jednoduchým dělením - v odstavci, zabývajícím se ovladačem přerušení časovače se s touto technikou seznámíme podrobněji.

5.3.7.2 Služba wakeup

Služba 'wakeup' není k dispozici ostatním vrstvám operačního systému, jako tomu je s ostatními službami správce procesů. Namísto toho je tato služba automaticky volána v rámci obslužné rutiny přerušení časovače (znovu na ni narazíme později při popisu ovladačů zařízení, až se budeme věnovat ovládání systémového časovače).

Služba 'wakeup' zajistí převedení prvního procesu z delta listu a všech dalších procesů, které mají v delta listu svou 'delta' nulovou (to znamená, že čekaly na stejný okamžik, jako první proces) do ready fronty. Časovač samozřejmě službu 'wakeup' volá až po uběhnutí potřebného času; k tomu slouží dvě globální proměnné: nenulová hodnota v proměnné 'slnempty' informuje ovladač přerušení, že existuje nějaký 'spící' proces, ukazatel 'slstop' pak obsahuje adresu 'delta' prvního procesu v delta listu - tedy zbyvajícího času do chvíle 'probuzení' tohoto procesu.

```
/* wakeup.c -- wakeup */

#include "kernel.h"
#include "proc.h"
#include "q.h"
#include "sleep.h"

/*
 * wakeup -- called only when delta list is nonempty!
 */
void wakeup(void)
{
    while (nonempty(clockqueue) && firstkey(clockqueue)==0)
        ready(getfirst(clockqueue), RESCHNO);
    if (slnempty==nonempty(clockqueue))
        slstop=&firstkey(clockqueue); /* ! implementace ! */
    resched();
}

/* end of file */
```

Implementace služby je celkem zřejmá: nejprve je zapotřebí 'probudit' všechny procesy, jejichž 'delta' je nulová - služba tedy všechny takového procesy převede do ready fronty (aniž by prozatím zajistila přeplánování).

Operační systémy

Potom je zapotřebí ihned naplnit proměnné 'slnempty' a 'sltop'; uvědomme si, že v rámci služby 'resched' může být spuštěn jiný proces a při jeho práci by mohlo dojít k dalšímu přerušení časovače - v tu chvíli však již musí obě proměnné obsahovat korektní údaje. Uložíme tedy do proměnné 'slnempty' informaci o tom, je-li delta list neprázdný, a je-li tomu tak, nastavíme i ukazatel 'sltop'.

Implementace zde (kvůli lepší přehlednosti) využívá potenciálně nebezpečné techniky - 'firstkey' je makro, které se převede na 'delta' první položky v delta listu; je tedy zcela korektní použít před ním operátor '&' pro zjištění adresy této položky. Pokud bychom však přeprogramovali službu 'firstkey' jako funkci, byl by výraz zcela chybný.

5.4 Synchronizace procesů

Často se stává, že různé procesy pracují nad společnou datovou strukturou - mějme např. frontu, do které všechny procesy občas ukládají statistické údaje o svém běhu. Operační systém pak tyto údaje z fronty odebírá a nějak zpracovává. Při implementaci takových procesů zjistíme, že existuje krátký časový úsek při vlastním zpracování fronty, kdy systém ukazatelů (nebo vzájemných odkazů v tabulkách), který frontu tvoří, nenív konzistentní stavu, protože uložení údaje do fronty již započalo, ale není ještě dokončeno.

Pokud by ale ve chvíli, kdy je jeden z procesů v takovémto stavu, došlo k přeplánování, riskujeme, že druhý proces se pokusí do fronty zapsat svůj údaj dříve, než první proces uvede frontu do konzistentního stavu. Druhý proces v takovém případě údaje ve frontě velmi pravděpodobně zničí a může se i sám zhroutit. První proces jej pak bude následovat ve chvíli, kdy se pokusí zápis do poničené fronty dokončit.

Takovým úsekům programu říkáme **kritické sekce** a, musíme nějakým způsobem zajistit, aby se nemohly dva procesy dostat do kritické sekce zároveň. Teoreticky by asi proces, který vstupuje do kritické sekce, mohl všechny ostatní procesy suspendovat a po jejím ukončení je opět uvolnit; na první pohled však je vidět, že to nebude nejšťastnější řešení.

Při práci operačního systému však můžeme narazit i na jiný problém - již jsme se o něm několikrát zmínili: některý proces musí čekat na pomalé periferní zařízení. V předcházejících odstavcích jsme si ukázali, jak lze dobu čekání efektivně využít pro práci ostatních procesů; dosud jsme se však neseznámili s žádným aparátem, který by dokázal rozumným způsobem zajistit samotné čekání. Opět by jistě bylo možné proces např. po dobu čekání suspendovat; nebylo by to však ani efektivní, ani programátorský pohodlné.

Je tedy zřejmé, že potřebujeme nějaký aparát pro **synchronizaci procesů**. Běžné operační systémy používají dva alternativní mechanismy - aparát semaforů a systém zpráv. Každý z nich se dobře hodí pro trochu jinou třídu problémů. V následujících odstavcích se s oběma seznámíme a ukážeme si i jejich implementaci v XINU.

5.4.1 Semafor

Základní myšlenka semaforů je poměrně jednoduchá: dobrá, jestliže tedy nesmějí dva procesy zároveň vstoupit do kritické sekce, vytvoříme programovými prostředky semafor, který postavíme na její začátek. Za normálních okolností bude na semaforu 'zelená'; jakmile však kolem něj projde do kritické sekce první proces, změní se barva na červenou a ostatní procesy kolem semaforu nebudou moci projít. Jestliže se o to některý proces pokusí, bude mu odebrán procesor, dokud bude na semaforu červená. Teprve ve chvíli, kdy první proces z kritické sekce odejde, změní se barva semaforu zpět na zelenou a Čekající proces - je-li takový - se vrátí do stavu 'ready'.

Ukažme si možnost implementace takového semaforu a jeho potenciální nevýhody.

5.4.1.1 Binární semafory

Takto definovaný semafor nazýváme binárním semaforem, protože má dva stavů: může být buď uzavřen, nebo otevřen. Je možné jej naprogramovat např. pomocí služeb 'lock' a 'unlock'. Parametrem obou služeb je adresa záznamu, obsahujícího dvě položky: byte, který slouží jako vlastní semafor a před použitím je nastaven na nulu, a frontu, která je před použitím vyprázdněna. Obě služby by pak mohly pracovat například následujícím způsobem:

- Služba 'lock' nejprve zjistí hodnotu semaforu a nastaví jej na jedničku. Tato operace musí být realizována jako nepřerušitelná, aby v době mezi testem hodnoty a nastavením na jedničku nemohl se semaforem pracovat jiný proces.

Jestliže byl původně semafor nulový, služba 'lock' již nedělá nic dalšího a bez problémů skončí. Pokud však semafor původně obsahoval jedničku, uloží služba 'lock' identifikační číslo aktivního procesu do fronty semaforu a pak jej suspenduje⁴⁸ (služba 'lock' je samozřejmě volána z aktivního procesu; ten tedy z tohoto hlediska suspenduje sám sebe).

- Služba 'unlock' nejprve ověří, je-li fronta semaforu neprázdná. Jestliže je tomu tak, vybere z ní první proces a převede jej (pomocí služby 'resume') do stavu 'ready'. Pokud naproti tomu fronta byla prázdná, vynuluje služba 'unlock' semafor.

Každý proces pak prostě před vstupem do kritické sekce zavolá službu 'lock' a po ukončení kritické sekce službu 'unlock' (parametrem služeb je samozřejmě semafor, spojený s kritickou sekcí). Implementace bude skutečně fungovat tak, jak bychom potřebovali:

⁴⁸V praxi by suspendování nebylo šikovné, lepší je zavést nový stav 'waiting'. Tak to také uděláme později, až budeme popisovat obecné semafory; prozatím používáme již vysvětlené služby pro lepší srozumitelnost.

První proces vstoupí do kritické sekce bez problémů, služba 'lock' rychle proběhne a ihned skončí. Jejím vedlejším efektem však bude nastavení semaforu na jedničku (na 'červenou').

Dokud první proces nevystoupí z kritické sekce, bude každý další proces, který zavolá službu 'lock', zařazen do fronty semaforu a suspendován - z hlediska takového procesu vlastně služba 'lock' neskončí.

Jakmile první proces vystoupí z kritické sekce, zavolá službu 'unlock'. Ta uvolní první z procesů, čekajících ve frontě semaforu - z hlediska tohoto procesu tedy právě v tu chvíli služba 'lock' skončí⁴⁹ a proces vstoupí do kritické sekce. Semafor zůstává 'červený', takže jakýkoli další proces, který by zavolal službu 'lock', bude pozastaven a zařazen do fronty.

Teprve ve chvíli, kdy vystoupí z kritické sekce poslední z procesů a fronta je prázdná, nastaví služba 'unlock' semafor opět na nulu.

Binární semafory je možné použít pro ochranu kritických sekcí i v řadě dalších případů; existují však i úlohy, které jejich pomocí řešit nelze vůbec nebo jen velmi obtížně.

Představme si, že dva procesy spolupracují třeba takovým způsobem, že první z nich počítá nějaké údaje a předává je druhému, který je formátuje do 'hezké' podoby a prezentuje uživateli. Předpokládejme, že procesy si budou předávat data prostřednictvím sdílené paměti o velikosti 512 byteů.

Pomocí binárních semaforů bychom mohli procesy synchronizovat tak, aby druhý proces - říkejme mu **konzument** - vždy počkal, až první proces - ten nazveme **producentem** - zapíše do sdílené paměti všech 512 byteů. Potom by opět musel čekat producent do chvíle, kdy konzument údaje z paměti odebere. Tak by se oba procesy mohly pravidelně střídat.

⁴⁹Nebo o něco později, totiž tehdy, až bude procesu přidělen procesor. K tomu samozřejmě nemusí nutně dojít hned při prvním přeplánování po jeho zařazení do ready fronty. Z hlediska synchronizace je to však lhostejné, protože ostatní procesy v ready frontě nemají se semaforem ani s kritickou sekcí nic společného (jinak by nebyly v ready frontě, ale čekaly by na semafor).

Operační systémy

Takový postup nám však přestane vyhovovat například ve chvíli, kdy bude producent vydávat hodnoty ve velmi nepravidelných intervalech. Dejme tomu, že producent dokáže vytvořit 510 bytů dat a pro získání zbývajících dvou bytů bude muset počítat třeba několik desítek minut. Jistě by bylo lepší, aby si v takovém případě mohl konzument odebrat již hotová data a zpracovat je.

Přrogramujme tedy producenta i konzumenta tak, aby si předávali údaje po jediném bytu. Problém, o kterém jsme se zmínili v minulém odstavci, skutečně nehrozí; je zde ale opačný nedostatek: pracují-li oba procesy hladce a srovnatelnou rychlostí, vznikne neúměrným způsobem režie při neustálém (byť kratičkém) čekání jednoho na druhého.

5.4.1.2 Obecné semafory

Pro řešení takovýchto problémů je ideální zobecnit semafory tak, že může nabývat nejen hodnot 0 a 1, ale libovolných celočíselných hodnot. Velikost čísla v semaforu tak může udávat, kolikrát je 'předplacena' služba 'lock', tj. kolikrát je možné ji provést, aniž bychom museli proces pozastavit. V opačném případě může velikost záporné hodnoty v semaforu určovat, kolikrát se na semafor vlastně čeká.

Semaforům tohoto druhu říkáme **obecné semafory (counting semaphores)**. Obecný semafor je kombinací fronty a celočíselné hodnoty, podobně jako binární semafor byl kombinací fronty a hodnoty binární. Pro realizaci obecných semaforů je zvykem používat služby 'wait' a 'signal'; popišme si jejich funkci:

- Služba 'wait' nejprve sníží hodnotu semaforu o jedničku. Je-li výsledná hodnota menší než nula, uloží služba 'wait' identifikační číslo aktivního procesu do fronty semaforu, převede aktivní proces do stavu 'waiting' a vyvolá přeplánování - při něm je samozřejmě procesu odebrán procesor (připomeňme implementaci služby 'resched' na straně 93).
- Služba 'signál' nejprve zvýší hodnotu semaforu o jedničku. Je-li výsledná hodnota menší nebo rovna nule, vybere služba první proces z fronty semaforu a převede jej (pomocí služby 'ready') do stavu 'ready'.

Vidíme, že implementace obecných semaforů je překvapivě ještě jednodušší, než implementace semaforů binárních. Podívejme se na skutečný zdrojový program⁵⁰ realizující obecné semafory v XINU. Typ 'semafor' je připraven takto:

```
/* sem.h */

#define SFREE '\x01'
#define SUSED '\x02'

struct sentry {
    signed short semcnt,           /* hodnota semaforu */
                squeue;          /* fronta */
};

extern struct sentry semaph[]; /* semafory */

/* end of file */
```

Služba 'wait' je pak implementována takto:

```
/* wait.c -- wait */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*
 * wait -- make current process wait on semaphore
 */
void wait(int sem)
{
    register struct sentry *sptr;
    register struct pentry *pptr;

    sptr=&semaph[sem];
    if (--(sptr->semcnt)<0) {
        (pptr=&proctab[currpid])->pstate=PRWAIT;
        pptr->psem=sem;
```

⁵⁰Jako obvykle oproti originálu zjednodušený o některé nepodstatné technické detaily.

```
enqueue(currpid,sptr->squeue) ;  
resched() ;  
}  
  
/* end of file */
```

Jediným rozdílem proti popisu chování služby 'wait', jak jsme jej před chvílí uvedli, je to, že číslo semaforu se uloží i do tabulky procesů (do položky 'psem'). To je proto, že často nám nestačí vědět, že proces čeká na nějaký semafor (to zjistíme ze stavu 'waiting'), ale potřebujeme znát i o který semafor se jedná. Procházení front všech semaforů a vyhledání inkriminovaného procesu by přitom bylo zbytečně zdlouhavé.

Ani implementace služby 'signál' nebude velkým překvapením:

```
/* signal.c -- signal */  
  
#include <conf.h>  
#include <kernél.h>  
#include <proc.h>  
#include <q.h>  
#include <sem.h>  
  
/*  
 * signal -- signal a semaphore, releasing one waiting process  
 */  
void signal(int sem)  
{  
    register struct sentry *sptr;  
  
    sptr=&semaph[sem];  
    if (sptr->semcnt++<0)  
        ready(getfirst(sptr->squeue),RESCHYES);  
    return(OK);  
}  
  
/* end of file */
```

Zajištění efektivní implementace vztahu producent-konzument je s pomocí obecných semaforů celkem jednoduché. Ke sdílené paměti připojíme dva semafory; první z nich (označme jej třeba A) inicializujeme O a druhý (B)

inicializujeme velikostí sdílené paměti. Kód producenta pak bude vypadat přibližně takto:

```
for (;;) {
    char data=vytvoř_data();

    wait(B);
    ulož_do_sdílené_paměti(data);
    signal(A);
```

Konzument bude naprogramován velmi podobně:

```
for (;;) {
    char data;

    wait(A);
    data=čti_ze_sdílené_paměti();
    signal(B);
    zpracuj_data(data);
}
```

Nyní je sdílená paměť využita nejlépe, jak to je vůbec možné. Budou-li oba procesy pracovat stejně rychle, stačí konzument odebírat data tak rychle, jak je bude producent vytvářet a k prostopojům nedojde vůbec. Jestliže se například konzument trochu zdrží, umožní hodnota semaforu B producentovi generovat data bez přerušení, dokud nezaplní celou sdílenou paměť; teprve pak bude pozastaven a přinucen čekat na konzumenta. Jestliže potom poběží pomaleji naopak producent, bude konzument pozastaven teprve tehdy, až bude sdílená paměť prázdná.

Poznamenejme, že obecné semafory mohou samozřejmě sloužit k hlídání kritických sekcí stejně snadno, jako binární semafory; (záporná) hodnota semaforu navíc udává počet procesů, které čekají na uvolnění semaforu.

Operační systémy

Nyní je asi také zřejmé, proč byl ve službě 'kill' na straně 100 inkrementován semafor, na který rušený proces čekal:

```
case PRWAIT:  
    semaph[pptr->psem].semcnt++; /* fall thru */
```

Služba 'kill' totiž mimo jiné odstraní proces z kterékoli fronty, ve které právě je; tedy i z fronty semaforu. Přitom je ovšem samozřejmě nutné upravit hodnotu semaforu odpovídajícím způsobem.

5.4.1.3 Semafore v multiprocesorovém prostředí

Připomeňme požadavek, který jsme stanovili při popisu služby 'lock': mezi testem hodnoty semaforu a jeho nastavením (u obecného semaforu mezi zjištěním hodnoty a dekrementací) nesmí v žádném případě se semaforem pracovat nikdo jiný (jinak by mohlo dojít k současnému vstupu obou procesů do kritické sekce).

Vytváříme-li operační systém pro počítač, osazený jediným procesorem, je to poměrně jednoduché - stačí před testem hodnoty semaforu *zakázat* přerušení, a po změně hodnoty semaforu jej opět povolit (jak víme, v PC-XINU je přerušení zakázáno dokonce po celou dobu práce kterékoli služby systému). Vážnějším problémem je implementace semaforů v operačním systému, který bude pracovat v multiprocesorovém prostředí - tam je nutné využít speciálních prostředků.

Většina procesorů pro podobné účely nabízí instrukci 'TAS' ('Test And Set'), nebo jinou instrukci s podobnou funkcí. Tato instrukce sama zajistí obě činnosti, potřebné pro realizaci služby 'lock' - tj. test 'nulovosti' proměnné v operační paměti a následné nastavení této proměnné na jedničku (nebo na jinou nenulovou hodnotu). Instrukce přitom pracuje s pamětí takovým způsobem, že po dobu její práce nemůže se stejnou proměnnou pracovat jiný procesor. Chceme-li tedy semafore naprogramovat 'bezpečně', musíme sestoupit na úroveň assembleru a použít instrukce 'TAS' (zbytek služby 'lock' - tj. obsluha fronty - samozřejmě může být nadále napsán ve vyšším jazyce).

5.4.2 Kritické sekce bez semaforů?

Zamyslíme-li se podrobněji nad aparátem kritických sekcí, uvědomíme si, že je značně 'defenzivní': současný vstup dvou procesů do kritické sekce nemusí nutně vést k poškození sdílených dat; pouze k němu vést může. Kritické sekce hlídané semafory však 'pro jistotu' současnemu vstupu dvou procesů apriori zabraňují.

Některé moderní mikroprocesory nabízejí technické prostředky pro implementaci jiného mechanismu přístupu ke sdíleným datům⁵¹, ve kterém se procesům současný vstup do kritické sekce povolí; zamezí se jim pouze dělat takové změny, které by vedly ke zničení dat.

Princip tohoto mechanismu je následující:

- Nejprve si proces, který vstupuje do 'kritické sekce', musí vytvořit vlastní kopii sdílených dat. Není přitom zapotřebí kopírovat celou datovou strukturu, nad kterou bude proces pracovat - stačí vytvořit kopii těch jejích částí, které se v rámci 'kritické sekce' budou měnit⁵².
- Pak proces provede požadovanou operaci; obsah sdílených proměnných však nemění, a namísto toho provede změnu nad svou kopíí těchto proměnných.
- Srovnáním obsahu sdílených proměnných s původní hodnotou proces zjistí, jestli se sdílená data v paměti nezměnila od té doby, kdy je poprvé četl (tj. neoperoval-li nad nimi mezitím jiný proces).

⁵¹V operačním systému určeném výhradně pro počítače s jedním procesorem samozřejmě můžeme tento aparát realizovat bez speciálních technických prostředků, s využitím zákazu přerušení - stejně jako tomu bylo u semaforů bez využití instrukce 'TAS'.

⁵²Přesně řečeno: proces musí vytvořit kopie všech sdílených údajů, které bude později měnit, a také kopie všech sdílených údajů, které bude jen číst, ale mohlo by je mezitím změnit jiný proces. V praxi se obě tyto skupiny většinou kryjí (tj. jedinou sdílenou proměnnou proces čte i mění a zároveň ji může měnit jiný proces).

Operační systémy

- Jsou-li data nezměněna, uloží se do nich nové hodnoty z vlastní kopie a proces může 'kritickou sekci' opustit.
- Pokud někdo data změnil, zopakuje proces celý postup od prvního bodu.

Je zřejmé, že třetí a čtvrtý bod musí být proveden jako jediná nedílná operace
- pro multiprocesorové systémy tedy musí nastoupit speciální služby technického vybavení.

Popsaný mechanismus budeme ilustrovat jednoduchým příkladem využívajícím speciální instrukce 'CAS' moderních mikroprocesorů řady Motorola 680x0 (komplexnější příklady případný zájemce naleze v [68030]). Tato instrukce je - podobně jako instrukce 'TAS' - 'nepřerušitelná jiným procesorem'; její funkce je však daleko komplexnější než funkce instrukce 'TAS'.

Instrukce 'CAS' pracuje se třemi argumenty: se dvěma datovými registry - označme je 'Do' a 'Dn' - a s adresou v paměti. Instrukce předpokládá, že registr 'Do' obsahuje původní hodnotu sdílené proměnné a registr 'Dn' obsahuje hodnotu, na kterou chceme sdílenou proměnnou nastavit. Sama sdílená proměnná pak je určena adresou.

Instrukce nejprve zjistí, je-li obsah sdílené proměnné roven obsahu registru 'Do' (porovná obě hodnoty a nastaví přepínače podle výsledku). Jsou-li obě hodnoty stejné, uloží obsah registru 'Dn' do sdílené proměnné, v opačném případě uloží obsah sdílené proměnné do registru 'Do'.

Předpokládejme, že úkolem 'kritické sekce' je inkrementovat sdílený čítač 'cnt'. Odpovídající úsek programu pak může vypadat například takto:

```
move.w      cnt,d0      ; původní hodnota čítače
loop: move.w      d0,d1      ; do D1 se uloží ...
      addq.w      #1,d1      ; ... nová hodnota
      cas.w       d0,d1,cnt   ; vlastní akce
      bne.s       loop       ; nepodařilo se - znova
```

Instrukce 'CAS' zde zjistí, má-li čítač dosud stejnou hodnotu, jakou měl při provedení první instrukce 'MOVE'. Je-li tomu tak, uloží do něj inkrementovanou hodnotu a jsme hotovi. Jestliže se však hodnota čítače změnila (tj. jeden či více paralelních tašku jej mezitím inkrementovalo), uloží

instrukce 'CAS' novou hodnotu do registru 'D0', takže ji můžeme ihned znovu inkrementovat a pokusit se ji znovu uložit.

Poznamenejme, že obdobným způsobem (zaměníme-li pouze inkrementaci dekrementací) nám instrukce 'CAS' mikroprocesoru Motorola nebo obdobná instrukce jiného procesoru umožní velmi snadno implementovat službu 'wait' obecného semaforu v multiprocesorovém prostředí. Implementace obecného semaforu pomocí instrukce 'TAS' naproti tomu není triviální - musíme vlastně test a inkrementaci obecného semaforu uzavřít do kritické sekce chráněné (jiným) binárním semaforem realizovaným pomocí instrukce 'TAS'.

5.4.3 Zprávy

Semafora velmi dobrým způsobem slouží při 'neosobní' komunikaci mezi procesy, kdy se vzájemně synchronizované procesy nemusí znát navzájem. Vztah producent-konzument je například určen sdílenou pamětí, se kterou jsou svázány dva semafory; producent přitom nepotřebuje vědět, kdo je konzumentem a naopak.

Existují však případy, kdy je zapotřebí synchronizovat procesy 'jmenovitě'. Představme si opět interpret uživatelských příkazů, který pro realizaci těchto příkazů vytváří samostatné procesy. Uživatel si vyžádá kopírování nějakých údajů na disketu, která není naformátovaná. Interpret vytvoří dva procesy - jeden pro formátování diskety, druhý pro kopírování dat. Nějakou dobu oba procesy poběží paralelně - dokud bude druhý z nich načítat údaje do svých vnitřních bufferů - pak ale musí druhý proces počkat na ukončení formátování.

Taková situace se samozřejmě dá ošetřit pomocí semaforů; není to však ani pohodlné ani elegantní. Daleko lépe zde poslouží mechanismus zpráv. Jeho princip je jednoduchý: kterýkoli proces může jinému procesu poslat zprávu. Kterýkoli proces si také může vyžádat přečtení zprávy, kterou obdržel. Jestliže dosud žádnou zprávu nedostal, operační systém jej pozastaví do chvíle, kdy se tak stane.

Různé operační systémy nabízejí různý luxus pro předávání zpráv. Vzhledem k tomu, že zprávy jsou ideálním prostředkem pro komunikaci se servery, bývají

moderní operační systémy (EPOC, NeXTStep) vybaveny poměrně složitými aparáty zpráv, ve kterých se zprávy odeslané některému procesu řadí do front, obsahem zprávy může být téměř cokoli a podobně.

My si ukážeme implementaci velmi jednoduchého systému zpráv v operačním systému XINU. Jeho princip je nicméně stejný, jako princip dnešních luxusních mechanismů; je na něm pouze nabalenou méně přídavných služeb.

Obsahem zprávy v XINU může být jen jediné číslo. Není proto zapotřebí pro zprávy vytvářet nějaký speciální typ. Podívejme se tedy rovnou na implementaci služby 'receive', která procesu vrátí zprávu kterou dostal:

```
/* receive.c - receive */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*
 * receive - wait for a message and return it
 */
int receive(void)
{
    struct pentry *pptr;

    if ((pptr=&proctab[currpid]) ->phasmmsg==0) {
        pptr->pstate=PRRECV;
        resched();
    }
    pptr->phasmmsg=0;
    return(pptr->pmsg);
}

/* end of file */
```

Služba nejprve zjistí, má-li proces k dispozici nějakou zprávu (a tedy je položka 'phasmmsg' v tabulce procesů nenulová). Jestliže tomu tak není, převede proces do stavu 'receiving' a zajistí přeplánování; jinak zprávu smaže a její obsah vrátí volajícímu.

Jestliže proces neměl k dispozici žádnou zprávu, je pozastaven uvnitř funkce 'resched'; ta z hlediska tohoto procesu skončí teprve tehdy, až mu někdo

zprávu pošle (a přeřadí jej přitom zpět do ready fronty). Je tedy zcela korektní to, že po návratu ze služby 'resched' funkce čte zprávu a vrací ji volajícímu.

Implementaci služby 'send' si pozorný čtenář dokáže již asi představit:

```
/* send.c - send */

#include<conf.h>
#include <kernel.h>
#include <proc.h>

/*
 *  send -- send a message to another process
 */
int send(short pid,short msg)
{
    struct pentry *pptr;

    if ((pptr=&proctab[pid])->pstate==PRFREE) ||
        (pptr->phasmmsg!=0)
        return(SYSERR);
    pptr->pmsg=msg;
    pptr->phasmmsg=1;
    if (pptr->pstate==PRRECV)
        ready(pid,RESCHYES);
    return(OK);
}

/* end of file */
```

Služba nejprve ověří, zda přijímající proces vůbec existuje a jestli již nějakou zprávu nemá (XINU neumožňuje vytvářet fronty zpráv). Jestliže oba testy projdou, uloží služba zprávu do tabulky procesů a pokud cílový proces byl ve stavu 'receiving', přeřadí jej do stavu 'ready', aby si zprávu mohl ihned přečíst.

5.4.4 Deadlock

Synchronizace procesů přináší nebezpečí **zablokování (deadlock)**. K němu může dojít v případě, že dva či více procesů čeká na sebe navzájem. Vypreparovanou podstatu zablokování můžeme ukázat na příkladu dvou procesů, které pracují se dvěma semafory, A a B, inicializovanými na jedničku:

```
proces_1()
{
    wait(A);
    wait(B);
    jakási_činnost();
    signal(B);
    signal(A);
}

proces_2()
{
    wait(B);
    wait(A);
    jakási_činnost();
    signal(A);
    signal(B);
}
```

Oba procesy se dostanou až k zavolání druhé služby 'wait'⁵³ a dále už nikdy nepokročí ani jeden z nich (čtenář, který neví proč, si může znova prostudovat odstavec 5.4.1.1).

Samozřejmě, že nikdo není takový blázen, aby programoval procesy právě popsaným způsobem (a je-li, zablokování si zaslouží). Semaforu (respektive jimi hlídané kritické sekce) však v našem příkladě sloužily pouze jako ilustrace libovolného prostředku, ke kterému má přístup pouze jediný proces; často ostatně používáme právě semafory k zajištění výhradního přístupu k zařízení, která není možné sdílet. V reálném operačním systému by se pak třeba mohlo stát, že proces číslo 1 chce pracovat s disketovou mechanikou a s tiskárnou, zatímco proces číslo 2 chce pracovat s tiskárnou a s disketovou mechanikou (v tomto pořadí). Výsledkem bude opět zablokování.

Ideální řešení již vlastně známe - zmínili jsme se o něm v odstavci 5.2.4. Stačí, když nedovolíme procesům, aby si vyhrazovaly zařízení a namísto toho vytvoříme pro každé zařízení speciální systémový proces - server - který bude služby zařízení zprostředkovávat ostatním procesům. Se serverem mohou

⁵³Předpokládáme, že procesy se rozběhly najednou a že běží paralelně, takže zatímco první proces provede operaci 'wait(A)', provede druhý proces operaci 'wait(B)'. Pokud by kterýkoli z procesů stačil provést oba své 'waity' dříve, než se rozběhne druhý proces, k zablokování by samozřejmě nedošlo.

komunikovat třeba všechny procesy najednou; není proto zapotřebí nic vyhrazovat a zablokování nehrozí.

5.5 Správa úloh

V některých složitějších operačních systémech - obvykle se jedná o systémy uzpůsobené pro dávkové zpracování programů - spolupracuje správce procesů úzce s dalším správcem, který např. v XINU není vůbec realizován - se **správcem úloh**.

Za běžných okolností se totiž požadavek na spuštění programu nebo sady na sebe vzájemně navazujících programů - tzv. **úloha (job)** - stane součástí operačního systému dříve, než se vytvoří odpovídající procesy. Velmi markantní je to právě v dávkových operačních systémech. Pak musí existovat správce úloh, který se o ně stará podobným způsobem, jako správce procesů o procesy: udržuje informace o jednotlivých úlohách, spravuje fronty, ve kterých úlohy čekají na zpracování nebo na vyřazení ze systému a podobně. Můžeme říci, že správce úloh realizuje plánování na vyšší úrovni.

Správce úloh vybírá úlohy ke zpracování. Pro tyto úlohy pak s využitím služeb správce procesů vytvoří procesy, o které se nadále stará správce procesů. Algoritmy, které používá správce úloh pro výběr úlohy ke zpracování, jsou poměrně komplikované; v této knize se jimi nebudeme podrobně zabývat, především proto, že její čtenáři se pravděpodobně budou nejčastěji setkávat s interaktivními operačními systémy, ve kterých hraje roli správce úloh uživatel. Případný zájemce nalezne podrobný rozbor správce úloh například v [OS].

Operační systémy

6. Ovladače periferií

Počítač komunikuje s okolním světem prostřednictvím periferních zařízení. Těchto zařízení tedy samozřejmě musí využívat i jednotlivé procesy. Jak jsme se již zmínili v úvodu, není v moderním - a tedy multitaskovém - operačním systému z mnoha důvodů možné nechat procesy, aby si mezi periferiemi 'dělaly, co je napadne'; operační systém musí obsahovat poměrně komplikovanou správu zařízení. Budeme se jí zabývat v této kapitole.

6.1 Vstupní a výstupní zařízení

Vstupním nebo výstupním zařízením počítače se může stát téměř cokoli. Nejčastěji se samozřejmě setkáme s diskovou jednotkou, klávesnicí, obrazovkou a tiskárnou, u modernějších počítačů také s myší; výjimkou však není ani plotter, digitizér, lokální síť, osvitová jednotka nebo scanner.

Počítač však může řídit i soustruh, traktor, tank nebo střelu země-vzduch. Podobně jako vstupní zařízení může sloužit třeba kamera, vlhkoměr, radiový přijímač nebo teploměr (což je častým případem právě ve spojení se zmíněnou střelou země-vzduch).

Z hlediska operačního systému jsou však přesto prese všechno vstupní i výstupní zařízení velmi podobná jedno druhému. U střediskových počítačů nebo moderních mikropočítačů bývají zařízení řízena speciálními I/O procesory, které nazýváme kanály; z hlediska operačního systému pak ovládání zařízení není ničím jiným, než komunikací s jiným procesorem.

Hlavním úkolem operačního systému ostatně není ani tak zařízení přímo ovládat⁵⁴, jako zajistit jejich korektní přidělování jednotlivým procesům - pro

⁵⁴To je samozřejmě pravda pouze do jisté míry - rozumný operační systém pochopitelně procesům nabízí již hotové služby pro ovládání zařízení na daleko vyšší úrovni, než jakou nabízí zařízení samo o sobě. Tyto služby však nemusí být nutně součástí vlastního operačního systému; namísto toho mohou být soustředěny např. na sdílených knihovnách.

Operační systémy

srovnání si můžeme uvést např. správce paměti, který operační paměť jednotlivým procesům pouze přiděluje, aniž by se jakkoli staral o to, co do přidělené paměti budou procesy zapisovat.

Mimochedem, je vhodné si uvědomit, že neexistuje principiální rozdíl mezi vstupním či výstupním zařízením na jedné straně a procesorem nebo pamětí na straně druhé - v obou případech se jedná o prostředky, které jednotlivé procesy potřebují a operační systém jim je vhodným způsobem přiděluje.

Zařízení mohou být v principu tří typů: vyhrazené zařízení, sdílené zařízení a společné zařízení. Podívejme se na ně nyní podrobněji:

6.1.1 Vyhrazená zařízení

Vyhrazené zařízení je takové zařízení, které nemůže sloužit více procesům najednou. Snad nejtypičtějším příkladem takového zařízení je tiskárna: pokud bychom umožnili přístup k tiskárně více procesům najednou nebo třeba jen střídavě v kratších intervalech, bude samozřejmě jediným výsledkem zmatený výstup, sestavený z promíchaných částí výstupů jednotlivých procesů.

Operační systém proto musí obsahovat pro každé vyhrazené zařízení jeho správce; ten může využít jedné ze dvou možných technik:

- **Vyhrazování zařízení** je základní a nejjednodušší mechanismus. Při něm správce prostě některému procesu zařízení přidělí (obvykle tomu, který s požadavkem přijde nejdříve), a ostatním procesům jeho používání nepovolí, dokud první proces zařízení neuvolní.

Pozornější čtenář si možná povšiml, že tento mechanismus je velmi podobný kritickým sekčím, popisovaným v odstavci 5.4; to není doslova pravda - je totiž přesně stejný. Na kritické sekce se můžeme dívat jako na velmi jednoduchého 'správce', který řídí přidělování zařízení - jímž jsou zde sdílená data - jednotlivým procesům. Naopak, implementace správce vyhrazeného zařízení je obvykle pouze trochu 'opentlenou' kritickou sekcí: zařízení je přiřazen semafor, první proces, který o zařízení požádá, tento semafor nastaví službou 'wait' na červenou a

každý další proces, který požádá o přidělení téhož zařízení, bude v rámci služby 'wait' pozastaven. Teprve poté, kdy první proces svou práci se zařízením ukončí a uvolní jej pomocí služby 'signal', bude uvolněn první z procesů, které na zařízení čekají. Podrobný popis tohoto mechanismu je ostatně součástí popisu obsluhy kritických sekcí na straně 112; pouze namísto služeb 'wait' a 'signal' jsou tam použity služby 'lock' a 'unlock' aparátu binárních semaforů.

Je tedy zřejmé, že správce vyhrazeného zařízení se při této technice nijak 'nepředře' a jeho programátor také ne. To jsou výhody, vyhrazování zařízení však má i jednu velmi podstatnou nevýhodu - může totiž dojít k zablokování, jehož princip jsme si vysvětlili v odstavci 5.4.4. Zopakujme jen stručně: představme si, že první proces si vyhradí tiskárnu a pak si chce vyhradit i disketovou mechaniku, zatímco druhý proces si vyhradil disketovou mechaniku a pak požádá i o tiskárnu. Oba procesy pak budou čekat na věky věků - nebo alespoň do chvíle, kdy operátor jeden z nich násilně ukončí (nebo vypne počítač).

Správce vyhrazených zařízení může zařízení přidělovat podle různých algoritmů chytře, tak, aby k zablokování nemohlo dojít (některé z těchto algoritmů si ukážeme v dalších odstavcích). Nevýhodou tohoto řešení je to, že i ten nejlepší ze zmíněných algoritmů přidělování zařízení nepříjemným způsobem omezuje, čímž snižuje výkon celého systému. Operační systémy pro střediskové počítače a moderní operační systémy pro mikropočítače proto využívají jinou techniku:

Virtualizace vyhrazených zařízení je podobný postup, jako virtualizace paměti, se kterou jsme se již seznámili. Stejně, jako jsme při virtualizaci paměti přiměli programy 'věřit', že je k dispozici daleko více paměti než doopravdy, přimějeme je při virtualizaci zařízení k 'věře', že namísto jediného vyhrazeného zařízení jich je k dispozici libovolně mnoho⁵⁵, takže každý proces, který o takové zařízení požádá, může dostat svoje.

⁵⁵Přesněji řečeno, počet virtuálních zařízení může být velmi vysoký, ale je samozřejmě omezen - podobně jako je velikost virtuální paměti omezena kapacitou odkládacího disku.

S touto technikou jsme se vlastně již v jiném kontextu také setkali v odstavci 5.2.4: stačí přidělit zařízení napevno jednomu systémovému procesu (serveru), který jej bude ovládat a ostatním procesům nabídně služby, které jim využití zařízení zprostředkuje nepřímo. Virtualizace pak spočívá v tom, že proces může snadno nabízet své služby třeba všem ostatním procesům najednou bez jakéhokoli vyhrazování.

Čtenář v tomto okamžiku může snadno namítnout, že taková virtualizace nám nebude nic platná, pouze nás dostane do stejného stavu, jako kdyby žádný správce neexistoval: výstup všech procesů bude na zařízení nesmyslně promíchán. Jenže to právě není pravda.

Server se totiž může snadno postarat o vhodné rozdělení výstupu jednotlivých procesů. Konkrétní technika samozřejmě záleží na konkrétním zařízení - na obrazovce je možné přidělit každému procesu okno, pro zvukový výstup je asi nejkorektnější zvuk vysílaný jednotlivými procesy smíchat dohromady a pro tiskárnu (a jí podobná zařízení) je zapotřebí použít techniky zvané **spooling**. Při ní se výstup jednotlivých procesů ukládá do souborů na disku. Tepřve ve chvíli, kdy proces 'tisk' zakončí (to se pozná tak, že proces uvolní 'tiskárnu'), je soubor uzavřen a předán jinému systémovému procesu, který tyto soubory postupně tiskne.

Je asi zřejmé, že principiálně je virtualizace daleko lepším řešením než vyhrazování zařízení; někdy se však docela prostě nevyplatí.

Stejně jako všechny ostatní 'inteligentní' mechanismy správy prostředků (například virtualizace paměti nebo preemptivní multitasking) je totiž virtualizace zařízení poměrně náročná: vyžaduje vlastní proces, spotřebuje nějakou operační paměť, pro spooling potřebujeme hodně místa na disku a podobně.

Nemá tedy valný smysl virtualizovat zařízení, která procesy využívají velmi zřídkakdy nebo zařízení, která až na naprosté výjimky stejně využívají jen jeden z procesů. Příkladem zařízení prvního typu může být třeba generátor DTMF

tónů⁵⁶ u kapesních počítačů PSION - jistěže jej může využívat kterýkoli proces, ale pravděpodobnost toho, že by se s takovým požadavkem 'srazily' dva procesy najednou (a druhý z nich by tedy musel být zablokován), je extrémně nízká. Příkladem zařízení druhého typu by mohl být třeba termostat a ovladač topení - velmi pravděpodobně bude pouze jediný proces sledovat změny teploty a ostatní procesy se o vyhrazení tohoto zařízení ani nebudou pokoušet.

6.1.2 Sdílená zařízení

Sdílená zařízení jsou taková zařízení, která mohou svou kapacitu rozdělit na části a každou části sloužit jinému procesu. Typickým případem je operační paměť nebo magnetický disk.

Sdílené zařízení samozřejmě není zapotřebí vyhrazovat; musí však existovat správce zařízení, který se bude starat o jeho rozdělení na jednotlivé části a o přidělování těchto částí procesům.

6.1.3 Společná zařízení

Společná zařízení naproti tomu mohou sloužit bez jakýchkoli problémů libovolnému počtu procesů najednou. Tato zařízení při vhodné technické realizaci jako jediná skutečně nepotřebují správce. Obvykle se jedná o jednoduchá vstupní zařízení; příkladem mohou být např. hodiny reálného času, které jsou součástí většiny počítačů, nebo mikrofon.

Společná zařízení *se* budou obejdou zcela bez správce, nebo je jejich správce naprostě triviální. Jestliže chce tvůrce operačního systému nabídnout procesům širší služby, než by zařízení mohlo zajistit samo o sobě (např. služby 'budíku' spojené s hodinami reálného času), vytvoří obvykle pro správu zařízení server; ten se k zařízení může chovat stejně, jako by se jednalo o zařízení vyhrazené.

je to 'pípání', které se v civilizovaných zemích používá při telefonování na místo našeho vytáčení čísel.

Operační systémy

Není proto zapotřebí se společnými zařízeními nadále zvlášť zabývat (většina autorů tato zařízení z popsaného důvodu nebude v úvahu).

6.2 Ovladače zařízení

Ačkoli je pravda, že hlavním úkolem správce zařízení je starat se o jeho přidělováním jednotlivým procesům a ne zařízení přímo ovládat, existuje několik velmi dobrých důvodů, proč v praxi nelze ani ovládání zařízení nechat na jednotlivých procesech:

- Často je zařízení ovládáno nepřímo prostřednictvím serveru nebo je virtualizováno. Pak operační systém samozřejmě zařízení ovládat musí.
- I v případě, že tomu tak není, je nutné nabídnout aplikačním programátorům vyšší úroveň ovládání zařízení, než jaké lze obvykle docílit přímým ovládáním. Jak již víme z poznámky 54 (a jak uvidíme podrobně v kapitole 9), je možné (a dokonce výhodnější) většinu služeb vyšší úrovni soustředit na sdílené knihovny; není však možné tak realizovat celý ovladač - důvodem je zabezpečení systému.
- Ten, kdo přímo ovládá periferní zařízení, totiž musí mít přístup k potenciálně nebezpečným službám operačního systému i technického vybavení (jako je např. mechanismus přerušení). Řada periferních zařízení navíc může při špatném řízení zapříčinit zhroucení operačního systému; existují dokonce zařízení, u kterých lze špatným řízením docílit zničení technického vybavení.

Není proto možné nechat procesy, aby samy ovládaly jednotlivá zařízení v žádném případě - ovladače zařízení musí být součástí operačního systému.

Nejprve se budeme zabývat klasickými ovladači zařízení, které musí být součástí každého operačního systému. Potom se vrátíme k serverům a ukážeme si, co se jejich použitím na koncepci klasických ovladačů mění a co naopak zůstává beze změny.

6.2.1 Klasické ovladače

Zamysleme se nejprve nad tím, jak by měl vypadat ovladač zařízení z hlediska procesu, který jej bude využívat:

- Ovladač samozřejmě musí nabízet služby dostatečně silné na to, aby bylo možné využít plně všech možností zařízení. Budeme-li např. prostřednictvím ovladačů zařízení pracovat i se soubory (což není nikterak neobvyklé, jak uvidíme v dalších kapitolách), je nutné mít k dispozici nějaký způsob detekce konce souboru nebo zjištění jeho velikosti.
- Zároveň však by měly být všechny ovladače v maximální možné míře navzájem podobné, alespoň z hlediska rozhraní mezi ovladačem a procesem, který jej využívá. Smysl tohoto požadavku je jasný - pokud by se nám například podařilo zajistit, že všechny ovladače budou nabízet stejné služby, je velmi snadné psát univerzální programy, které budou teprve při spuštění spojeny s konkrétním zařízením pro vstup a pro výstup.

Je tedy zapotřebí vytvořit takovou skupinu služeb, která bude dobře sloužit pro přístup ke kterémukoli zařízení; mezi nimi přitom musí být alespoň jedna 'speciální' služba, která zpřístupní neobvyklé a výjimečné vlastnosti konkrétního zařízení, samozřejmě za tu cenu, že programátor musí typ zařízení sám detektovat.

Většina operačních systémů se shodla na následující (nebo velmi podobné) skupině služeb (tentto konkrétní příklad je převzat z operačního systému XINU):

- Služba **Init** zajistí inicializaci zařízení na začátku práce celého systému nebo po jeho restartování. Bývá obvykle bez parametrů a může vracet stav zařízení (tj. podařilo-li se jej inicializovat nebo ne).
- Řada zařízení požaduje navíc před čtením provést jakousi 'menší' inicializaci nebo přípravu - např. měřicí systém je zapotřebí zapnout (a po

ukončení načítání hodnot zase vypnout). U jiných zařízení - typicky se jedná o zařízení pro komunikaci - zase před zahájením vlastní komunikace bývá nutné navázat spojení, vytvořit komunikační 'kanál', kterým bude komunikace probíhat. K tomu všemu je určena služba **Open**, zařízení, která tuto službu nepotřebují, ji mohou snadno implementovat jako prázdnou; nic se tedy nestane, když ji program 'pro jistotu'⁵⁷ zavolá.

Služba 'open' navíc může snadno sloužit k virtualizaci zařízení - její volání vytvoří nové virtuální zařízení, které bude samo nadále sloužit programu. Nejtypičtějším případem zde bývá ovladač disku; jeho služba 'open' obvykle vytvoří logické zařízení odpovídající souboru.

Parametrem služby 'open' bývá obvykle jméno požadovaného virtuálního zařízení nebo kanálu; služba vrací nové číslo zařízení, které bude program nadále používat (nebo nezměněné číslo původního zařízení tam, kde služba 'open' není zapotřebí).

- Po ukončení práce se zařízením připraveným pomocí příkazu 'open' je často zapotřebí vytvořený kanál opět uzavřít nebo zrušit virtuální zařízení. K tomu slouží služba **Close**. Jejím jediným parametrem bývá číslo zařízení.
- Pro vlastní výměnu dat mezi zařízením a programem jsou určeny služby **Read**, **Write**, **Getc** a **Putc**. První dvě předávají celé bloky dat, druhé dvě postupují po jediném bytu.
- Služba **Seek** je dokladem toho, že u velkého množství zařízení se můžeme 'vrátit' a znova si přečíst data, která jsme již jednou četli nebo můžeme nějakou skupinu údajů přeskočit a číst až za nimi. Služba sděluje ovladači relativní pozici dat, která chce program číst, v rámci celého datového bloku.

⁵⁷Respektive pro případ, že bude při spuštění spojen se zařízením, které před použitím službu 'open' vyžaduje.

- Konečně pro speciální případy, které není možné pokrýt dosud popsanými službami, je k dispozici služba **Cntl.** Její funkce není definována, a záleží prostě na typu konkrétního zařízení.

Dříve, než se podíváme na způsoby implementace jednotlivých služeb, se však soustředíme ještě na funkci celého ovladače z 'druhé strany', tedy z pohledu vlastní komunikace s konkrétním zařízením.

Základní princip komunikace s typickým zařízením vypadá takto - jako příklad zvolíme zařízení výstupní (pro vstupní by se však nic podstatného nezměnilo, pouze aktivace zařízení by přešla z ovladače na připojený vstup):

- Ovladač na základě požadavku některého procesu zapíše data na vhodné místo v paměti a pak zařízení aktivuje, tj. předá mu příkaz 'odešli data z té a té adresy'.
- Zařízení po nějakém čase data odešle; pak dá ovladači na vědomí, že je možné připravit další data. Jestliže ovladač již odesal vše, co bylo zapotřebí, jsme hotovi; jinak ovladač připraví další dávku dat a zařízení opět aktivuje.
- Tento postup se opakuje, dokud není požadavek procesu zcela splněn (nebo dokud nedojde k nějaké chybě, která jeho splnění znemožní).

Je zřejmé, že se jedná o komunikaci typu producent/konzument, kterou jsme poznali již v odstavci o synchronizaci procesů. Víme, že má-li taková komunikace probíhat efektivně a bez časových ztrát, i když jsou oba procesy (zde příprava dat programem na straně jedné a jejich odesílání zařízením na straně druhé) různě rychlé a pracují nestejnoměrně (tak tomu v daném případě skutečně bývá), je zapotřebí použít semaforů. To ale znamená, že ovladač musí být složen ze dvou samostatných programů, které spolu popsaným způsobem komunikují - první z nich slouží jako producent (dosud jsme mu říkali ovladač), a druhý je konzumentem (obvykle je umístěn v obslužné rutině přerušení, kterým zařízení oznamuje odeslání dat).

Vidíme, že ovladač běžného zařízení má dvě části. První z nich - tzv. horní polovina - je volána procesy (stává se tedy vlastně součástí volajícího procesu)

Operační systémy

a zajišťuje předávání údajů do sdílené paměti pro výstup a odebírání údajů ze sdílené paměti pro vstup. Horní polovina se zařízením přímo nekomunikuje, až na jedinou výjimku - aktivaci zařízení na samém začátku výstupu. Druhá část ovladače - tzv. dolní polovina - pak zajišťuje synchronizaci mezi zařízením a horní polovinou ovladače.

Podívejme se na obě poloviny a na problematiku s nimi související podrobněji.

6.2.1.1 Logický systém ovladačů

Představme si, že by uživatelské programy volaly přímo funkce, které tvoří horní polovinu ovladačů jednotlivých zařízení. Takový přístup by nebyl právě praktický - program by pracoval 'natvrdo' s jediným pevně zvoleným zařízením, a pro použití jiného zařízení by bylo zapotřebí jej znova přeložit.

Operační systémy proto musí obsahovat **tabulku zařízení**, která mapuje jméno nebo identifikační číslo zařízení na jednotlivé obslužné rutiny. Ve skutečných operačních systémech je tato tabulka samozřejmě vytvářena dynamicky při startu systému, nebo - jako třeba v EPOCu - může být měněna při zavádění a odstraňování jednotlivých ovladačů. My pro lepší přehlednost ukážeme použití pevné tabulky, jaká je součástí např. operačního systému XINU (to samozřejmě znamená, že chceme-li změnit konfiguraci počítače, musíme operační systém XINU znova přeložit - což by v praxi bylo poněkud nepraktické). Dalším zjednodušením bude to, že programy se budou na jednotlivá zařízení odkazovat prostřednictvím jejich čísel; to nám umožní tabulku přímo indexovat číslem požadovaného zařízení.

Ve skutečném operačním systému bychom samozřejmě namísto tabulky použili spojový seznam instalovaných ovladačů a namísto indexace vyhledání zadaného jména v tomto seznamu; to by zkomplikovalo jednotlivé rutiny, jejichž zdrojový kód si ukážeme, aniž by to cokoli měnilo na principu logického systému ovladačů.

Tabulka zařízení 'devtab' je tedy v XINU deklarována takto:

```
/* conf.h */
```

```

/* jména jednotlivých služeb - např. dvproc[Getc](i) */
enum {Init,Open,Close,Read,Write,Seek,Getc,Putc,Cntl};
#define NDVPROCS 9           /* počet služeb */

struct devsw {
    int (*dvproc[NDVPROCS])(struct devsw*,...);/* služby */
    int dvminor;          /* číslo konkrétního zařízení */
    int dvivec,dvovec;    /* vektory přerušení */
    void interrupt (*dviint)(),(*dvoint);/* obsl. rutiny */
};

extern struct devsw devtab[];

/* end of file */

```

Jak je vidět, je celé zařízení popsáno poměrně jednoduchou strukturou. Položky 'dvivec', 'dvovec', 'dviint' a 'dvoint' nás nyní nemusí zajímat - určují přerušení, jejichž prostřednictvím zařízení s procesorem komunikuje, a obslužné rutiny pro tato přerušení. Operační systém tyto položky využije pouze při inicializaci, kdy zajistí, aby přerušení volalo odpovídající rutinu (této problematice se budeme podrobněji věnovat v odstavci 6.2.1.4).

Pole 'dvproc' obsahuje adresy všech devíti služeb, o kterých jsme se zmiňovali v minulém odstavci. Pro větší přehlednost jsou přiřazena pomocí enum-u čísla jednotlivých služeb odpovídajícím identifikátorům - službu 'seek' ovladače zařízení číslo 3 tedy zavoláme příkazem

```
devtab[3].dvproc[seek](&devtab[3],pos);
```

Parametry služeb jsou samozřejmě rozdílné; prvním parametrem kterékoli z nich však je ukazatel na položky tabulky 'devtab', ze které byla služba vybrána. Služby tak mají přístup ke všem polím, která mohou potřebovat; obvykle se samozřejmě jedná o pole 'dvioblk' a 'dvminor' - viz níže:

Poslední položkou je číslo konkrétního zařízení 'dvminor'. To umožňuje řídit jediným ovladačem několik zařízení - všechna taková zařízení pak budou mít obsah všech položek v tabulce zařízení identický, vyjma jediné položky - právě 'dvminor'. V ní bude uloženo číslo, podle kterého ovladač (tj. služby z pole 'dvproc') zjistí, se kterým zařízením má právě pracovat.

6.2.1.2 Přístup programů ke službám

Originální systém XINU (stejně jako implementace ST-XINU) byl vlastně pouhou knihovnou služeb, která se spojila s přeloženými uživatelskými programy a vzniklý program se spustil jako celek. Díky tomu stačilo 'zabalit' volání služeb pro vstup a výstup do jednoduchých funkcí, které byly uživatelským programům přímo k dispozici.

Ve skutečném operačním systému, který dokáže uživatelské programy vytvářet a zavádět za běhu, tento způsob samozřejmě nepřipadá v úvahu. Nejjednodušším řešením je využít prostředků, které nabízí procesor pro volání služeb systému (jako je např. instrukce SVC u počítačů IBM360/370 nebo instrukce TRAP u mikroprocesorů Motorola), a připravit tzv. dispečera služeb. Ten přijme požadavek uživatelského programu a sám volá odpovídající službu.

PC-XINU samozřejmě využívá aparát přerušení, který pro tento účel nabízí mikroprocesory řady Intel 80x86. Ukažme si zdrojový kód dispečera služeb vstupu a výstupu, který byl připojen na přerušení číslo 0x61:

```
/* h_61.c -- handler61 */

#include <conf.h>
#include <kernel.h>

extern unsigned p_seg;

void interrupt (*memo61)();

void interrupt handlers61(unsigned bp,unsigned di,
                           unsigned si,unsigned ds,
                           unsigned es,unsigned dx,
                           unsigned cx,unsigned bx,
                           unsigned ax)

char ah,al;
struct devsw *devptr;

enable();
ah=ax>>8;
al=ax;
devptr=&devtab[al];
switch (ah) {
    case 0x00:
```

```

        ax=devptr->dvproc[Init] (devptr) ;
        break;
case 0x01:
        ax=devptr->dvproc[Open] (devptr,bx,dx) ;
        break;
case 0x02:
        ax=devptr->dvproc[Close] (devptr) ;
        break;
case 0x03:
        ax=devptr->dvproc[Read] (devptr,bx,cx) ;
        break;
case 0x04:
        ax=devptr->dvproc[Write] (devptr,bx,cx) ;
        break;
case 0x05:
        ax=devptr->dvproc[Seek] (devptr,dx) ;
        break;
case 0x06:
        ax=devptr->dvproc[Getc] (devptr) ;
        break;
case 0x07:
        ax=devptr->dvproc[Putc] (devptr,cx) ;
        break;
case 0x08:
        ax=devptr->dvproc[Cntl] (devptr,cx,bx,dx) ;
        break;
default:
        ax=SYSERR;
}

/* end of file */

```

Implementace dispečeru je celkem samozřejmá: nejprve povolíme přerušení, které instrukce INT (v tomto případě poněkud zbytečně) zakázala. Pak rozložíme obsah registru AX do dvou čísel - vyšší, které odpovídá obsahu registru AH, je číslem požadované služby, zatímco nižší, které odpovídá obsahu registru AL, je číslem zařízení, jehož ovladač má službu provést.

Služba se pak vyvolá uvnitř příkazu 'switch', parametry se jí předají z patřičných registrů. Návratová hodnota je uložena do registru AX a řízení se vrátí programu, který dispečer vyvolal.

Operační systémy

Abychom mohli služby ovladačů zařízení pohodlně volat i z vyšších jazyků, musíme vytvořit knihovny pomocných funkcí; ty budou samozřejmě velmi jednoduché. Ukažme si příklad implementace služby 'read' na takové knihovně:

```
/* Lread.c - knihovní funkce pro volání služby 'read' */
unsigned read(int dev,void *buffer,unsigned len)
{
    if (dev>=DEVICES) return(BAD_DEVICE);
    _AX=0x0300|dev;
    _BX=buffer;
    _CX=len;
    geninterrupt(0x61);
    /* return AX */
}
/* end of file */
```

Funkce nejprve ověří, je-li číslo zařízení korektní, a pak naplní registry a zavolá prostřednictvím dispečeru požadovanou službu. Příkaz 'return' nemusí být (v Turbo C) explicitně uveden, protože vynecháme-líjej, vrací každá funkce hodnotu registru AX.

V dalším textu knihy budeme předpokládat, že takovéto knihovny mají k dispozici i vyšší vrstvy operačního systému, a mohou tedy samy používat služby 'init', 'open', ... , 'cntl' pro vstup a pro výstup.

6.2.1.3 Horní polovina

Služby horní poloviny komunikují s dolní polovinou ovladače na základě vztahu producent-konzument nad sdílenou pamětí (které v tomto kontextu obvykle říkáme 'buffer'). Při výstupu dat z počítače je horní polovina ovladače v postavení producenta a dolní polovina pracuje jako konzument; při čtení dat je tomu právě naopak.

Horní polovina ovladače vlastně sestává z devíti (v našem případě - obecně jich samozřejmě může být různý počet) služeb 'init'...'cntl' a z údajů v bloku 'dvioblk'. Konkrétní příklad i velmi jednoduchého ovladače by byl příliš rozsáhlý; uvedeme proto jen základní algoritmy některých služeb.

Implementace ostatních služeb a detailní implementace všech pomocných funkcí se od uvedených příkladů v zásadě neliší; je však samozřejmě z technických důvodů daleko komplikovanější.

Ukážeme si implementaci nejdůležitějších služeb horní poloviny ovladače zcela obecného zařízení, komunikujícího s počítačem po jednotlivých bytech - dobrým příkladem může být třeba sériové rozhraní. Ukážeme také, jak může být využito pole 'dvmino' - náš obecný ovladač bude schopen řídit několik zařízení stejněho typu.

Naprostá většina ovladačů má nějaká 'soukromá' data, která potřebuje pro svou práci - jedná se o buffery, kam jsou ukládána přenášená data a podobně. Deklarace těchto dat pro náš ovladač by mohla vypadat následovně:

```
/* driver.h */

#define IBUFLEN 256 /* velikost vstupního bufferu */
#define OBUFLLEN 256 /* velikost výstupního bufferu */

struct dev1 { /* data pro jedno zařízení */
    int ihead, itail; /* začátek/konec vstupní fronty */
    char ibuf[IBUFLEN]; /* vstupní buffer */
    SEM isem; /* vstupní semafor */
    int ohead, otail; /* začátek/konec výstupní fronty */
    char obuf[OBUFLLEN]; /* výstupní buffer */
    SEM osem; /* výstupní semafor */
    void (*run_odev)(void); /* aktivace výstupního zařízení */
    void (*stop_odev)(void); /* deaktivace výstupního zařízení */
    void (*wrt_odev)(char); /* zápis znaku na výst. zařízení */
    char (*rd_idev)(void); /* čtení znaku ze vstup. zařízení */
};

extern struct dev1 ddevs[];

/* end of file */
```

Hlavními položkami datové struktury 'dev1', která popisuje stav zařízení, jsou vstupní a výstupní buffer 'ibuf' a 'obuf', do kterých se ukládají načtené a odesílané znaky.

Oba buffery jsou realizovány jako tzv. cyklické buffery - dojdeme-li při práci s takovým bufferem na jeho konec, pokračujeme zase od jeho začátku. Pro

každý buffer potom potřebujeme dva ukazatele - jeden určuje místo, kde v bufferu začínají validní data, a druhý ukazuje, kde začíná volné místo. Tyto ukazatele jsou uloženy v položkách 'ihead', 'itail', 'ohead' a 'otail'.

Semafory 'isem' a 'osem' slouží pro synchronizaci mezi horní a dolní polovinou ovladače na základě známého mechanismu producent-konzument.

Funkce 'run_odev' povolí výstupnímu zařízení, aby přerušením hlásilo, že 'nemá co na práci'. Zařízení nemá k dispozici žádná data, a proto vygeneruje přerušení, které samozřejmě bude obsluženo dolní polovinou ovladače - ta pak data skutečně odešle.

Funkce 'stop_odev' naopak výstupnímu zařízení generování přerušení zakáže. Výstupní zařízení v takovém případě samozřejmě dokončí případný přenos dat, pokud nějaký provádí, ale jeho ukončení již nebude hlásit přerušením. Pak zůstane zařízení v klidu, dokud nebude opět zavolána služba 'run_odev'.

Funkce 'wrt_odev' výstupnímu zařízení předá znak k odeslání. Zařízení znak odešle (což může trvat delší dobu), a potom generuje přerušení (má-li to povoleno předchozím voláním funkce 'run_odev').

Funkce 'rd_idev' naproti tomu přenos dat prostřednictvím vstupního zařízení nevyvolá; pouze z tohoto zařízení přečte znak, který již byl načten. Vstupní zařízení nemá nikdy zakázané přerušení; kdykoli načte nějaký znak, generuje přerušení a jeho obslužná rutina pak použije funkci 'rd_idev' ke zjištění načteného znaku.

Implementace funkcí 'run_odev', 'stop_odev', 'wrt_odev' a 'rd_idev' samozřejmě závisí podstatným způsobem na konkrétním zařízení; nemá proto smysl, abychom k nim uváděli zdrojové texty. Budeme prostě předpokládat, že v odpovídajících položkách již jsou umístěny ukazatele na správné funkce.

Struktury 'devl' tvoří pole 'ddeps'; v něm je každému zařízení, které má ovladač k dispozici, věnována jedna položka. Ovladač tedy prostě bude používat číslo konkrétního zařízení 'dminor jako index do pole 'ddeps'.

Služba pro čtení jednoho znaku ze zařízení (její adresa bude uložena do položky 'dvproc[Getc]') je velmi jednoduchá:

```
/* _getc.c */

#include <conf.h>
#include <driver.h>

int _getc(struct devsw *dev)
{
    struct dev1 *dta=&ddevs[dev->dvminor];
    char ret,x;

    sdisable(x);
    wait(dta->isem);
    ret=dta->ibuf[dta->itail++];
    if (dta->itail==IBUflen)
        dta->itail=0;
    restore(x);
    return(ret);
}

/* end of file */

```

Funkce nejprve nastaví ukazatel 'dta' na strukturu odpovídající požadovanému zařízení a po provedení makra 'sdisable' (k němu se zanedlouho vrátíme) zajistí vlastní čtení dat:

Služba 'wait', volaná na vstupní semafor, pozastaví proces na tak dlouho, dokud nebudou k dispozici nějaká data (vstupní semafor tedy musí být na počátku inicializován nulou). Službu 'signál' na tento semafor zavolá dolní polovina ovladače po přijetí dat - to uvidíme v odstavci 6.2.1.5. Potom bude proces znova aktivován a data odebere ze začátku fronty ve vstupním bufferu; odebraná data uloží do proměnné 'ret'. Pak je ještě zapotřebí zajistit korektní funkci cyklického bufferu případným přechodem z jeho konce na začátek. To je vše; funkce vrací načtená data.

Dosud jsme ignorovali makra 'sdisable' a 'restore'. Bez nich by však funkce nepracovala korektně - ukažme si nejprve, k jaké chybě by mohlo dojít.

Nesmíme totiž zapomínat, že se pohybujeme v preemptivním multitaskovém operačním systému. Pokud by však došlo k přeplánování např. mezi odebráním dat ze vstupní fronty a úpravou ukazatele (v příkazu 'if (dta->itail ...)'), mohl by nový aktivní proces jistě zavolat sám službu 'getc' na totéž zařízení. Pak by ovšem nalezl v položce 'itail' nekorektní hodnotu!

Celý obsah služby '_getc' je tedy kritickou sekcí. Mohli bychom samozřejmě pro její ohlídání použít dalšího semaforu, jak jsme se to naučili v odstavci 5.4.1.1⁵⁸; ušetříme si však práci tím, že prostě zakážeme přerušení. Po dobu zákazu přerušení samozřejmě k přeplánování nemůže dojít; můžeme si to dovolit proto, že doba, po kterou bude přerušení zakázáno, bude velmi krátká - služba '_getc' se nikde a ničím nezdržuje (případné přeplánování uvnitř služby 'wait' nepřináší žádné problémy; pokud se čtenář domnívá, že ano, může si nalistovat podrobný rozbor takovéto situace v odstavci 6.2.1.4 na straně 145).

S makry 'sdisable' a 'restore' se zde setkáváme poprvé⁵⁹. Vysvětlíme si proto jejich funkci podrobněji:

Makro 'sdisable' uloží do proměnné, která je jeho parametrem, informaci o tom, zda je nebo není povoleno přerušení. Pak přerušení zakáže. Makro 'restore' pak ověří, je-li zapotřebí přerušení opět povolit (tedy bylo-li povoleno v okamžiku provedení makra 'sdisable'), a v kladném případě tak učini⁶⁰.

Výstupní služba '_putc' (jejíž adresa bude samozřejmě uložena do položky 'dproc[Putc]') je jen nepatrн komplikovanější:

```
/* _putc.c */
#include <conf.h>
#include <driver.h>

void _putc(struct devsw *dev, char ch)
    struct dev1 *dta=&ddevs[dev->dvminor];
```

⁵⁸V multiprocesorovém systému bychom to tak udělat dokonce museli.

⁵⁹Připomeňme znova, že služby PC-XINU žádný mechanismus pro zákaz přerušení nepotřebovaly, protože v PC-XINU je pro zjednodušení automaticky přerušení zakázáno po celou dobu práce kterékoli služby systému (což samozřejmě snižuje průchodnost systému, takže pokud by bylo PC-XINU voleno pro praktickou aplikaci, bylo by vhodné jej v tomto smyslu přeprogramovat).

⁶⁰Implementace této funkce pro mikroprocesory řady Intel 80x86 je uvedena v dodatečích v odstavci C.2.3.

```

char x;

wait(dta->osem);
sdisable(x);
dta->obuf[dta->ohead++]=ch;
if (dta->ohead==OBUFLEN)
    dta->ohead=0;
restore(x);
dta->run_odev();
}

/* end of file */

```

Funkce nejprve nastaví ukazatel 'dta' na strukturu odpovídající požadovanému zařízení a pomocí služby '**wait**' a výstupního semaforu počká, dokud nebude v bufferu místo (výstupní semafor tedy musí být na počátku inicializován ne nulou nebo jedničkou, ale velikostí výstupního bufferu). Po návratu ze služby '**wait**' v bufferu jistě místo je; proces tedy na konec fronty v bufferu uloží zadaný znak a zajistí korektní funkci cyklického bufferu případným přechodem z jeho konce na začátek. Nakonec funkce aktivuje zařízeníslužbu '**run_odev**'; zařízení tedy při nejbližší příležitosti (pravděpodobně tedy ihned) generuje přerušení. Dolní polovina ovladače, která je obslužnou rutinou tohoto přerušení, pak data odešle.

Uložení dat do bufferu a úprava ukazatele '**ohead**' je opět kritickou sekcí; tuto část kódu proto musíme ochránit před případným přeplánováním pomocí maker '**sdisable**' a '**restore**'.

6.2.1.4 Obsluha přerušení

Jak již dávno víme, nejsou dolní poloviny ovladačů ničím jiným, než obslužnými rutinami přerušení. Obsluha přerušení je však záležitostí, na kterou stojí zato se podívat podrobněji. Jsou zde dva základní problémy: především musíme vyřešit vlastní instalaci procedur psaných ve vyšším jazyce jako obslužných rutin přerušení; potom se musíme nějak ubránit tomu, aby při zpracování obslužné rutiny přerušení nebylo přerušení vygenerováno znova - při dostatečném počtu takto vnořených rutin by totiž mohl přetéci zásobník a aktivní proces by se

Operační systémy

zhroutil⁶¹. Při struktuře operačního systému, kterou zde popisujeme, je navíc nutné věnovat zvláštní pozornost nulovému procesu.

Podívejme se nyní na jednotlivé problémy podrobněji:

- Instalace **obslužných rutin přerušení** je v Turbo C, které používáme pro naše příklady, proveditelná velmi snadno - stačí vytvořit funkci typu 'interrupt' a její adresu uložit do příslušného vektoru přerušení.

Pokud bychom však měli k dispozici překladač, který disponuje pouze standardními prostředky podle normy ANSI, nepřipadá podobné řešení v úvahu. Pak musíme napsat jednoduchou obslužnou rutinu přerušení v assembleru; ta pouze zjistí, k jakému přerušení došlo a sama zavolá odpovídající 'obslužnou rutinu' ve vyšším jazyce - ta v tomto případě může samozřejmě být zcela běžnou funkcí.

- O to, aby nedošlo ke **vnořenému volání** obslužné rutiny přerušení, se stará každý rozumný procesor sám - při vstupu do obslužné rutiny přerušení totiž další přerušení automaticky zakáže. Při návratu z obslužné rutiny přerušení je obvykle obnoven původní stav.

Tento mechanismus se nám hodí; musíme si ovšem uvědomit, co z něj přesně vyplývá: jestliže bude obslužná rutina přerušení provozována při zakázaném přerušení, nesmí trvat příliš dlouho⁶² - dolní poloviny ovladačů tedy musí být poměrně jednoduché.

⁶¹To samozřejmě platí pro procesory typu mikroprocesoru Intel 8086, které mají pouze jeden zásobník, společný pro procesy i pro obslužné rutiny přerušení. Moderní procesory jsou vybaveny více zásobníky, právě pro zamezení této možnosti. Implementace operačního systému pro takový procesor však nic nemění na tom, že vnořenému volání obslužných rutin přerušení se musíme vyhnout.

⁶²Jinak by mohlo dojít k chybě při komunikaci se zařízeními, která by načetla v době zakázaného přerušení dva nebo více znaků: první načtený znak ještě může 'počkat', až bude přerušení opět povoleno; další znaky však není kam uložit - k bufferu se nedostaneme, protože přerušení je zakázáno a nelze tedy aktivovat dolní polovinu ovladače.

Dalším důsledkem je to, že žádná ze služeb systému nesmí přerušení explicitně povolit. Pokud by totiž takovou službu použila obslužná rutina přerušení, mohlo by dojít ke vnořenému volání - přerušení by bylo povolené ještě před ukončením jeho obslužné rutiny. Služby systému, které musí přerušení zakazovat (protože obsahují kritické sekce) tedy musí nejprve uložit momentální stav a na konci jej obnovit - tak, jak to dělají např. makra 'sdisable' a 'restore'.

Ze zdrojových textů horních polovin ovladačů je jasné, že bychom potřebovali, aby dolní poloviny ovladačů volaly službu 'signal'. Ta však vyvolá přeplánování - je vůbec korektní volat přeplánování při zakázaném přerušení?

První otázka může znít, nebude-li trvat přeplánování 'moc dlouho' a nedojde-li tedy k chybě, na kterou upozorňuje poznámka 62. Jestliže však po přeplánování poběží dál tentýž proces, který běžel před ním, je vše v pořádku, protože bude následovat rychlé ukončení obslužné rutiny přerušení a tedy opětovné povolení přerušení. Stav přerušení však je součástí kontextu; dojde-li tedy v rámci přeplánování k výměně kontextu, bude stav přerušení odpovídat novému procesu. Bude-li přerušení v novém procesu povoleno, je dobré. Bude-li však přerušení v novém procesu zakázáno, znamená to, že proces se vzdal procesoru uvnitř obslužné rutiny přerušení a nyní tedy obslužnou rutinu přerušení rychle ukončí a přerušení opět povolí (je to vlastně přesně stejná situace, jako kdyby při přeplánování ke změně kontextu nedošlo).

Dobrá; je-li tomu tak, nemůže přece jen dojít k mnohonásobnému vnoření obslužné rutiny přerušení? Odpověď zní - nemůže. Každý proces totiž může být přerušen pouze jednou - dokud tento proces běží a obslužná rutina přerušení není ukončena, je přerušení zakázáno. V rámci jednoho procesu a jeho zásobníku tedy ke vnořenému volání obslužných rutin přerušení nemůže dojít.

- Poslední problém může nastat díky **nulovému procesu**. Z odstavce 5.3.4 víme, že nulový proces musí být vždy k dispozici v ready frontě a nesmí proto v žádném případě volat službu 'wait', 'receive' a podobně. Zde se však právě skrývá nebezpečí: uvědomme si, že obslužná rutina

přerušení probíhá vlastně v kontextu toho procesu, který byl zrovna náhodou aktivní ve chvíli, kdy bylo přerušení vyvoláno - speciálně tedy může obslužná rutina přerušení probíhat v kontextu nulového procesu.

Z toho vyplývá, že dolní polovina ovladače sice smí vyvolat přeplánování (jak jsme si ukázali před chvilkou), musí však používat jen takové služby, které zanechají volající proces v ready frontě.

To je také důvod, proč je komunikace producent-konzument v ovladači výstupního zařízení řešena poněkud netradičním způsobem. Dolní polovina ovladače je konzument znaků, a měla by proto používat služby 'wait'; to si však nemůžeme dovolit. Jednoduchým trikem jsme proto z dolní poloviny ovladače udělali producenta volného místa v bufferu a z horní jeho konzumenta; nyní službu 'wait' používá horní polovina a vše je v naprostém pořádku.

6.2.1.5 Dolní polovina

Dolní polovina ovladače je instalována jako obslužná rutina přerušení generovaného zařízením, které ovladač řídí. Ukažme si obě obslužné rutiny odpovídající příkladu ovladače, který jsme uvedli v odstavci 6.2.1.3; nejprve popíšeme o něco jednodušší dolní polovinu ovladače vstupního zařízení. Ta je volána vždy, když vstupní zařízení přijme znak. Obslužná rutina přerušení je určena vždy pro jediné konkrétní zařízení - může tedy znát jeho číslo jako konstantu DEV^{63} .

V rutině použije dosud nepopsanou službu 'scount', jejímž parametrem je semafor. Služba nedělá vůbec nic zázračného - vrátí prostě hodnotu obecného semaforu. Ovladač tuto hodnotu potřebuje znát, aby zjistil, je-li ještě volné místo v bufferu.

Zdrojový text rutiny vypadá takto:

⁶³To samozřejmě platí pouze v našem modelu s pevnou tabulkou zařízení - dolní polovina ovladače, který byl instalován při inicializaci systému nebo za běhu, musí mít pro informaci o zařízení vyhrazenou statickou proměnnou, kterou naplní při instalaci ovladače.

```

/* in_iohandler.c */

#include <conf.h>
#include <driver.h>

#define DEV XYZ /* číslo zařízení */

void interrupt in iohandler()
{
    struct dev1 *dta=&ddevs[devtab[DEV].dvminor];

    if (scount(dta->isem)==IBUFLEN) /* buffer je plný ... */
        rd_idev();
    else {
        dta->ibuf[dta->ihead++]=rd_idev();
        if (dta->ihead==IBUFLEN)
            dta->ihead=0;
        signal(dta->isem);
    }
}

/* end of file */

```

Rutina nejprve nastaví ukazatel 'dta' na strukturu odpovídající požadovanému zařízení. Pak zjistí, je-li v bufferu ještě místo; vzhledem k tomu, že po přijetí každého znaku zvýšila hodnotu semaforu, zatímco horní polovina ovladače při odebrání znaku hodnotu semaforu sníží, stačí nyní porovnat momentální hodnotu semaforu s velikostí bufferu⁶⁴. Je-li buffer plný a nikdo jej nečte, nedá se nic dělat - přijaté znaky se zahazují (trochu 'chytréjsí' ovladač by mohl oznámit vysílajícímu, že již nemá místo odesláním vhodného znaku pomocí ovladače výstupního zařízení; sám by naopak mohl podobný znak interpretovat a po jeho přijetí ovladač výstupního zařízení zastavit).

Jestliže je v bufferu ještě místo, uloží na něj rutina přijatý znak a po případné úpravě ukazatele v cyklickém bufferu to oznámí horní polovině ovladače 'signal'em.

⁶⁴Mohli bychom namísto toho ověřovat vzájemnou polohu ukazatele na začátek a na konec fronty; tím bychom se však připravili o jedno místo v bufferu, protože využijeme-li buffer zcela, jsou oba ukazatele stejné - stejně jako když je buffer úplně prázdný.

Operační systémy

Ovladač přerušení výstupního zařízení vypadá velmi podobně; je jen trochu složitější, protože se musí postarat o případné ukončení přenosu:

```
/* out_iohandler.c */

#include <conf.h>
#include <driver.h>

#define DEV XYZ /* číslo zařízení */

void interrupt out iohandler()
{
    struct dev1 *dta=&ddevs[devtab[DEV].dvminor];

    if (scount(dta->oem)==OBUFLEN) /* buffer je celý volný */
        dta->stop_odev();
    else {
        dta->wrt_odev(dta->obuf[dta->otail++]);
        if (dta->otail==OBUFLEN)
            dta->otail=0;
        signál(dta->oem);
    }
}

/* end of file */
```

Rutina nejprve nastaví ukazatel 'dta' na strukturu odpovídající požadovanému zařízení. Pak zjistí, je-li v bufferu vůbec nějaký znak; jestliže tam není, nejedná se o chybu, ale prostě o dokončení přenosu - ovladač tedy pouze deaktivuje výstupní zařízení funkcí 'stop_odev'.

Je-li v bufferu nějaký znak, ovladač jej odešle a po případném upravení ukazatele do cyklického bufferu oznámí horní polovině získání jednoho volného místa v bufferu 'signal'em.

622 Servery

Nevýhodou popsaného mechanismu ovladače je to, že proces musí (v rámci některé ze služeb horní poloviny) čekat, než bude jeho požadavek splněn. Mezitím může samozřejmě pracovat jiný proces; často by se nám však lépe hodil jiný mechanismus:

- Proces by si prostřednictvím služeb horní poloviny ovladače vyžádal vstup nebo výstup a pak by okamžitě pokračoval v práci - nesměl by ovšem měnit údaje, které byly určeny pro výstup nebo snažit se číst údaje ze vstupu.
- Teprve ve chvíli, kdy proces opravdu načtená data potřebuje (nebo kdy je nutné změnit data zapisovaná), by dal ovladači na vědomí, že nyní již musí počkat. Ovladač by zjistil, zda dosud data nejsou načtena (zapsána) a pouze v takovém případě by proces pozastavil.

Tohoto mechanismu lze dosáhnout dvěma způsoby. Buďto můžeme přereprogramovat horní polovinu ovladače tak, že dosud popsané služby nebudou čekat na ukončení přenosu a namísto toho přibude služba **Iowait**, která čekání zajistí (tj. k návratu z ní nedojde dříve, než bude přenos ukončen a než dolní polovina ovladače nastaví patřičný semafor). Druhou možností je vytvořit samostatný proces, který bude sám využívat ovladač zařízení a ostatní procesy mu budou své požadavky předávat prostřednictvím zpráv⁶⁵; takový proces pak může čekat na ukončení přenosu, zatímco jeho 'klienti' - tj. procesy, které požadavky vyvolaly - bez problémů běží dále.

Takový proces, vyhrazený pro obsluhu některého zařízení, však již známe a říkáme mu server (v dalších kapitolách se k serveru dostaneme ještě několikrát na základě několika dalších požadavků - uvidíme, že servery skutečně řeší řadu problémů).

⁶⁵Aparát zpráv pak samozřejmě musí spravovat fronty zpráv a ne nejvýše jednu čekající zprávu pro každý proces, jako tomu bylo v triviálním aparátu zpráv systému XINU, jehož implementaci jsme poznali v odstavci 5.4.3.

6.3 Postavení ovladačů v operačním systému

Některé ovladače jsou samozřejmě pevnou součástí operačního systému. Ve většině případů však tomu musí být jinak - operační systém musí být schopen pracovat s řadou nejrůznějších zařízení; není dost dobře možné, aby obsahoval všechny potenciálně potřebné ovladače. Musí být také možné zapojit do operačního systému i zařízení nové, vytvořené později než operační systém sám.

Snad všechny operační systémy proto nabízejí prostředky, umožňující při startu systému zavést potřebné ovladače. Jednotlivá zařízení v takovém případě bývají obvykle identifikována jménem, které je součástí každého ovladače.

Určitou výjimkou je operační systém EPOC, který je určen pro práci na kapesních počítačích, které se prakticky nikdy nevypínají, takže *zavádění* ovladačů při startu systému by nemělo valný smysl. EPOC proto *nabízí* služby pro zavádění a odstraňování ovladačů automaticky za běhu na základě toho, zda některý z procesů služby toho ovladače potřebuje nebo ne (tentot mechanismus je velmi podobný mechanismu *zavádění sdílených knihoven*, se kterým se seznámíme v odstavci 9.2.4).

6.4 Obrana proti deadlocku

Nejúčinnější obranou je samozřejmě virtualizace, kterou jsme poznali v odstavci 6.1.1. Ne vždy se však vyplatí zařízení virtualizovat; seznámíme se proto s několika algoritmy, které zablokování bud' zamezí, nebo jej alespoň detekují.

Algoritmy popíšeme z hlediska úloh, protože pro jejich praktickou implementaci je zapotřebí mít alespoň triviálního správce úloh k dispozici. Procesy totiž na sebe mohou navazovat tak, že vyhrazení prostředků platí pro více procesů za sebou (jeden vyhradí, druhý uvolní...), takže může dojít k zablokování i na úrovni úlohy.

Budeme-li však chtít implementovat některou z těchto technik ve velmi jednoduchém operačním systému, který s úlohami nepracuje (takovým systémem je například XINU), stačí zaměnit slovo 'úloha' v následujících odstavcích slovem 'proces'. Připomeňme ovšem ještě jednou, že v takovém případě bude správce buď bezbranný proti zablokování zaviněnému tím, že jeden proces si zařízení vyhradí a až druhý (vyvolaný třeba tím prvním) jej uvolní, nebo bude muset takové situaci 'z moci úřední' zabránit (čímž se samozřejmě o něco sníží flexibilita systému).

6.4.1 Úplné vyhrazení prostředků

Nejjednodušší strategií je nepochybně úplné vyhrazení prostředků. Správce úloh v tomto případě musí mít k dispozici informace o všech prostředcích, které bude úloha vůbec kdy potřebovat. Po dobu běhu úlohy pak tyto prostředky nepřidělí nikomu jinému (můžeme také říci, že je pro úlohu vyhradí po celou dobu jejího běhu).

Je zřejmé, že při použití této strategie k zablokování dojít nemůže z toho prostého důvodu, že žádná úloha nikdy na přidělení nějakého zařízení nečeká. Úloha může čekat pouze na své spuštění, nejsou-li dosud k dispozici všechna zařízení, která bude požadovat; to však není nebezpečné, protože v té chvíli ještě úloha žádné zařízení vyhrazeno nemá.

Zároveň však vidíme i zásadní nedostatek této metody - obrovské snížení průchodnosti systému. Každá úloha po celou dobu svého běhu 'okupuje' všechna zařízení, která může kdy potřebovat; jestliže např. úloha deset hodin počítá a nakonec vytiskne jediné číslo jako výsledek, bude tiskárna - po celých deset hodin úlohou blokovaná - zřejmě využita velmi špatně.

Mimochedem, je vhodné si uvědomit, že pro realizaci této metody potřebujeme rozšířit službu 'wait' na jakési 'skupinové wait', které dokáže pracovat s množinou semaforů. Pokud bychom totiž pro vyhrazení prostředků na začátku práce úlohy použili normální operaci 'wait' (např. v programovém cyklu), mohlo by dojít k zablokování stejně, jako kdyby si proces prostředky vyhrazoval sám - dostatečně jasně tuto možnost ilustruje následující - v této

Operační systémy

formě pochopitelně poněkud umělý - příklad (povšimněte si podobnosti s příkladem na straně 124):

```
/* správce */
wait(A);
wait(B);
/* úloha */
job_1()

        jakási_činnosti();

}/* správce */
signal(B);
signal(A);

/* správce */
wait(B);
wait(A);
/* úloha */
job_2()

        jakási_činnost();

}/* správce */
signal(A);
signal(B);
```

6.4.2 Hierarchické přidělování prostředků

Na první pohled vidíme, že strategie úplného vyhrazení prostředků je až příliš 'defenzivní', že zabraňuje nebezpečí zablokování daleko dříve, než by k němu vůbec mohlo dojít. Nebylo by tedy možné nějak omezit dobu, po kterou budou mít procesy vyhrazeny jednotlivé prostředky, aniž by hrozilo zablokování?

Jednu z cest, jak toho docílit, může napovědět i minulý příklad. Na první pohled v něm vidíme, že nebezpečí by bylo odstraněno, kdyby se v obou případech prostředky vyhrazovaly ve stejném pořadí. Můžeme tedyjít o krůček dále: uspořádejme všechny prostředky do jisté hierarchie a dovolme úlohám alokovat prostředky kdykoli chtejí, ale pod jedinou podmínkou - musí prostředky požadovat ve vzrůstajícím pořadí podle dané hierarchie a uvolňovat naopak v pořadí klesajícím.

V tomto případě také nehrozí zablokování. Ačkoli to není na první pohled tak jasné, jako v minulém případě, potvrdí nám to jednoduchý 'matematický' důkaz:

- Představme si, že v kterémkoli okamžiku pozastavíme běh systému a podíváme se, jaké prostředky mají jednotlivé úlohy vyhrazeny. Porovnáme umístění těchto prostředků v hierarchii a vybereme úlohu, která měla vyhrazen 'nejvyšší' prostředek ze všech.

- Vzhledem k požadavku na postupné alokování prostředků víme, že úloha, kterou jsme si vybrali v minulém kroku, bud' již nebude potřebovat žádný další prostředek, nebo bude potřebovat prostředek na vyšší hierarchické úrovni.
- Všechny prostředky na vyšší úrovni však jsou volné - připomeňme, že v prvním kroku jsme vybrali nejvyšší z přidělených prostředků. Vybraná úloha tedy nemůže být zablokována a může bez problémů pokračovat.

V kterémkoli okamžiku tedy existuje v systému alespoň jedna úloha, která nemůže být zablokována. Po uvolnění a dalším alokování některých zařízení se samozřejmě může situace změnit; pouze však v tom smyslu, že 'nezablokovatelná' bude jiná úloha - úvahy, uvedené jako 'důkaz' platí skutečně v kterémkoli okamžiku práce operačního systému⁶⁶. I v případě, že některé úlohy přibudou nebo ubudou, musí vždy existovat nejméně jedna, která nemůže být zablokována.

Poznamenejme, že v praxi se často hierarchická strategie kombinuje se strategií úplného vyhrazení prostředků v tom smyslu, že na jedné úrovni hierarchie nebývá jediný prostředek, ale celá skupina prostředků; prostředky z této skupiny pak samozřejmě musí úlohy požadovat v jediném kroku.

Hierarchická strategie je daleko výhodnější než strategie úplného vyhrazení prostředků; přesto však je ještě pořád příliš 'defenzívní' - i ona zajišťuje systém proti zablokování daleko dříve, než se objeví skutečné nebezpečí. Cenou je - stejně jako v minulém případě - snížená průchodnost systému. Do jisté míry je možné situaci zlepšit vhodným navržením hierarchie prostředků - čím lépe bude odpovídat 'přirozenému' postupu alokace prostředků ve většině úloh, tím bude průchodnost systému lepší. Ani v tom nejlepším případě však není obtížné najít příklad úloh, které budou hierarchickou strategií zbytečně brzděny.

⁶⁶Zcela doslova by to nebyla pravda - v úvaze předpokládáme, že alespoň jeden prostředek je právě nějaké úloze přidělen. V případě, že tomu tak není, však samozřejmě zablokování nehrozí.

6.4.3 Bankéřův algoritmus

Nejvhodnější by zřejmě bylo nalézt takovou strategii přidělování prostředků, která by procesy neomezovala tak dlouho, dokud by nebezpečí zablokování nehrzoilo; teprve ve chvíli, kdy se objeví skutečně akutní nebezpečí zablokování by strategie prostředek nepřidělila a nechala by si jej V zásobě', dokud se situace nezlepší a přidělení prostředku nebude bez nebezpečí. Taková strategie skutečně existuje; říkáme jí **bankéřův algoritmus**.

Název strategie vznikl na základě 'slovní úlohy', která převádí problém správce prostředků do trochu reálnějšího světa:

Mějme bankéře (správce prostředků), který má k dispozici určité částky peněz v různých měnách (počty jednotlivých prostředků). Bankéř půjčuje peníze svým zákazníkům (úlohám) a ti mu je po dokončení svých záměrů opět vracejí. Úkolem bankéře je přitom půjčovat peníze takovým způsobem, aby nemohlo dojít k situaci, kdy by banka byla nenávratně insolventní - tj. nemohla by plnit požadavky svých zákazníků, a ti by nemohli vracet vypůjčené peníze, protože (pro nedostatek dalších peněz) by nemohli dokončit své záměry.

Pro splnění tohoto požadavku potřebuje bankéř předem znát maximami možné požadavky každého svého zákazníka. Pak může při každé půjčce uvažovat tak, že peníze půjčí jen tehdy, když má zaručeno, že existuje alespoň jedno pořadí dalších požadavků (pořadí může bankéř ovlivnit snadno - řekne žádajícímu aby přišel později), při kterém všichni zákazníci ukončí své záměry a vrátí peníze.

Je-li bankéř inteligentní, mohl by uvažovat například takto:

- *Musím si nejprve rozmyslet, je-li bezpečné půjčit tyto peníze. Budu tedy předpokládat, že jsem je půjčil, a zkusím, stačí-li mi zbytek ke splnění budoucích požadavků.*
- *Musím projít všechny své zákazníky a zjistit, zdaje mezi nimi aspoň jeden, který mi bude moci vrátit peníze - tedy takový, jehož budoucí požadavky již nepřesáhnou množství peněz, které mi ještě zbývá. Pokud by snad žádný takový zákazník neexistoval, jistě peníze nepůjčím.*

- Pokud někdo takový existuje, představím si, že mi již peníze vrátil. Pak obdobným způsobem zjistím, bude-li mi moci peníze vrátit některý z dalších zákazníků. Kdyby tomu tak nebylo, peníze samozřejmě také nepůjčím.
- Úvahu v minulém odstavci budu opakovat tak dlouho, dokud nezjistím, že peníze půjčit nemohu nebo dokud 'se nezbavím' všech zákazníků. V druhém případě je vše v pořádku a peníze mohu doopravdy půjčit.

Z algoritmu samotného v tomto případě vyplývá i důkaz jeho správnosti: jestliže existuje alespoň jedno pořadí požadavků, ve kterém nedojde k zablokování (tj. ve kterém všichni zákazníci nakonec budou schopni splnit své záměry), může operační systém toto pořadí v nejhorším případě (tj. nevrátí-li žádný zákazník ani část peněz před dokončením svých záměrů) vynutit selektivním splňováním požadavků.

Pro ty, kdo mají raději 'matematické' důkazy (jako byl důkaz korektnosti hierarchického přidělování prostředků) stačí uvést, že:

- Po každé výpůjčce (přidělení prostředků) existuje alespoň jeden zákazník, který může své záměry ukončit (je to ten první, kterého jsme našli v druhém bodě popisu algoritmu). Pokud by totiž nikdo takový neexistoval, bankéř by peníze nepůjčil.
- Po navrácení peněz a odchodu zákazníka (ukončení úlohy) opět existuje alespoň jeden zákazník, který může své záměry ukončit. Buď totiž odešel jiný zákazník, než ten 'první' z minulého bodu (a ten tedy své záměry může jistě ukončit - mohl již předtím, a nyní má bankéř ještě více volných prostředků). Nebo odešel právě tento zákazník; takovou situaci však bankéř ve třetím bodě algoritmu simuloval a zjistil, že i v takovém případě existuje někdo další, kdo své záměry může dokončit (jinak by peníze nebyly zapůjčeny).

Tuto úvahu můžeme opět indukcí dovést až do uzavření banky po vrácení všech peněz a odbavení všech zákazníků (do konce práce operačního systému bez zablokování).

Převeďme tedy bankéřův algoritmus do prostředí správce prostředků. Pro ilustraci použijeme mírně upravený úsek programu v jazyce C, který měl být původně použit v PC-XINU (nakonec to nebylo zapotřebí, protože PC-XINU neobsahuje žádný vyhrazený prostředek). Jedná se o implementaci funkce 'lze_pridelit', která zkoumá, je-li možné splnit požadavek některého z procesů na přidělení prostředku; jestliže to možné je, vrací nenulovou hodnotu, jinak vrací nulu.

Program používá tři pomocné funkce, jejichž význam je následující:

- Funkce 'simulovat_prideleni' vytvoří novou kopii seznamu prostředků, které má správce momentálně k dispozici, a odebere z tohoto seznamu prostředek, o který žádá proces, jehož žádost je právě zkoumána.

Kopie seznamu prostředků je lokální; při ukončení funkce 'lze_pridelit' tedy automaticky zanikne.

- Funkce 'lze_dokonciť', jejímž parametrem je číslo procesu, porovná momentální obsah lokálního seznamu prostředků s maximálními budoucími požadavky zadaného procesu. Vrací nulu, jsou-li požadavky procesu nesplnitelné (tj. příliš velké).
- Funkce 'simulovat_dokonceni', jejímž parametrem je také číslo procesu, prostě přidá k lokálnímu seznamu prostředků všechny prostředky, které má zadaný proces přiděleny.

Algoritmus pak je poměrně jednoduchý:

```
int lze_pridelit(int prostredek)

enum {OK, LOCK} tab[POCET_PROCESU];
int i,locked;

simulovat_prideleni(prostredek);
for (i=0;i<POCET_PROCESU;i++)
    tab[i]=LOCK;
Dokola:
locked=0;
for (i=0;i<POCET_PROCESU;i++)
    if (tab[i]==LOCK)
        if (lze_dokoncit(i)) {
```

```

        simulovat_dokonceni(i)
        goto Dokola;
    } else locked++;
return(!locked);
}

```

6.4.4 Detekce deadlocku

Je vhodné si uvědomit, že k zablokování by za určitých okolností mohlo dojít i v operačních systémech, které vůbec neobsahují žádné vyhrazené prostředky. Příčinou je to, že sdílené prostředky mají omezenou velikost. I míra virtualizace je omezena (obvykle kapacitou vnějších paměťových jednotek). Je-li pak přidělování virtuálních nebo sdílených prostředků řešeno 'klasickou' cestou, kdy je žádající proces, jehož požadavek nelze splnit, pozastaven semaforem a čeká na chvíli, kdy bude požadavek splnitelný, nebezpečí zablokování se stane realitou.

Představme si např. operační systém s virtualizaci tiskárny, na němž poběží dva procesy se společným programem, který bude vypadat takto:

```

proces_1()
{
    static DEVICE_HANDLE pole[100000];
    int pocet;

    for (pocet=0;pocet<100000;pocet++)
        pole[pocet]=chci_alokovat_zarizeni(TISKARNA);
    for (pocet=0;pocet<100000;pocet++)
        zapis_text(pole[pocet],"To jsem tě vypek!");
    while (--pocet>=0)
        uvolnuji_zarizeni(pole[pocet]);
}

```

Je celkem zřejmé, že dříve nebo později budou oba procesy čekat na uvolnění některé z dosud alokovaných virtuálních tiskáren, protože místo pro sto tisíc virtuálních tiskáren se těžko na některém reálném systému najde. K uvolnění však nikdy nedojde, protože proces by své tiskárny uvolnil teprve při ukončení práce.

Tento problém je samozřejmě poněkud umělý. Můžeme mu čelit několika způsoby:

- I pro přidělování sdílených a virtuálních zařízení použijeme některý z algoritmů pro zabránění zablokování, které jsme popsali v minulých odstavcích.
- Zakážeme procesům (respektive úlohám) požadovat více než určitý pevně daný malý počet virtuálních nebo sdílených zařízení jednoho typu.
- Pro přidělování virtuálních a sdílených zařízení nepoužíváme klasický 'čekací' mechanismus, ale mechanismus běžně používaný pro přidělování paměti⁶⁷: není-li možné požadavek uspokojit, není proces pozastaven semaforem, ale namísto toho mu operační systém vrátí informaci o tom, že se požadavek uspokojit nepodařilo (obvykle pomoci nějakého chybového kódu). Je již věcí programu samého, jak se s takovou situací vypořádá.
- Operační systém detekuje zablokování a informuje o něm operátora; ten pak může úlohy, které jej zavinily, násilně ukončit⁶⁸ - je totiž téměř jisté, že zablokování při alokaci virtuálních nebo sdílených zařízení bylo zaviněno chybou programu.

Asi nejvýhodnější je využívání posledních dvou technik. Pro realizaci druhé z nich je však zapotřebí, aby operační systém zablokování nějak detekoval - není dost dobré únosné, aby operátor zablokování poznal jen z toho, že se 'nějak dlouho nic neděje'.

Princip detekce zablokování je založen na hledání uzavřené cesty v orientovaném grafu. Po překladu do češtiny to znamená, že správce zařízení zjišťuje, jestli náhodou v pořadí 'Proces A čeká na zařízení X, které má proces

⁶⁷Která ostatně není ničím jiným než sdíleným zařízením.

⁶⁸Je ovšem nutné si uvědomit, že násilné ukončení úlohy může přinést problémy, jestliže úloha již provedla nějaké nevratné změny - příkladem mohou být třeba aktualizace údajů v databázi.

B; proces B čeká na zařízení Y, které má proces C; proces C čeká ... nemůžeme narazit znovu na proces A.

Algoritmus vhodný pro implementaci v rámci správce prostředků je založen na udržování dvou tabulek: tabulky přidělení, ve které je pro každý prostředek zapsáno, kterému procesu patří, a tabulky čekání, ve které je pro každý proces uvedeno, na který prostředek čeká. Předpokládáme-li, že prostředky i procesy jsou v systému identifikovány celými čísly, mohli bychom obě tabulky deklarovat např. takto:

```
#define EMPTY -1

int a[POCET_ZARIZENI]; /* tabulka přidělení */
int w[POCET_PROCESU]; /* tabulka čekání */
```

Potom bude i-tá položka v tabulce 'a' obsahovat číslo procesu, jemuž je přidělen prostředek s číslem 'i' nebo konstantu 'EMPTY', je-li prostředek 'i' dosud volný. V i-té položce tabulky V pak nalezneme číslo prostředku, na jehož uvolnění proces 'i' čeká nebo hodnotu 'EMPTY', jestliže proces 'i' na žádný prostředek nečeká.

Algoritmus přidělování prostředku s detekcí zablokování by pak mohl vypadat například takto (globální proměnná 'currpid' obsahuje číslo aktivního procesu, který žádá o přidělení prostředku):

```
int pridel_prostredek(int p)

    int vlastnik;
    int ceka_na;

    if (a[p]==EMPTY) {
        Pridel_prostredek:
            pridel_prostredek(p); /* volání 'wait' nebo 'lock' */
            a[p]=currpid;
            return(1); /* prostředek přidělen */

        w[currpid]=p;
        vlastnik=a[p];
        while (vlastnik!=currpid) {
            if ((ceka_na=w[vlastnik])==EMPTY)
                goto Pridel_prostredek; /* aktivní p. bude pozastaven */
```

Operační systémy

```
    vlastnik=a[ceka_na] ;  
  
    __DEADLOCK__();  
    return(0); /* prostředek nepřidělen */
```

Funkce je poměrně jednoduchá. Nejprve ověří, je-li požadovaný prostředek volný; v kladném případě jej prostě přidělí. V záporném pak postupně pomocí hodnot v polích 'a' a W prochází řetěz čekajících procesů; narazí-li přitom opět na proces, který požaduje přidělení prostředku (podmínka v příkazu 'while'), ohlásí zablokování a prostředek nepřidělí. Narazí-li namísto toho na proces, který již na nic nečeká, prostředek také 'přidělí'; funkce 'wait' nebo 'lock' však zajistí, že v tomto případě funkce 'pridel_prostredek' z hlediska volajícího procesu skončí až po uvolnění prostředku.

Poznamenejme, že nabízí-li správce prostředků služby vyhrazení několika prostředků najednou (takové služby jsou zapotřebí pro úplné vyhrazení prostředků a v praxi často i pro hierarchické vyhrazování, jak jsme viděli v odstavcích 6.4.1 a 6.4.2), je nutné algoritmus nepatrně zkomplikovat. Jediný proces pak totiž může čekat na více prostředků, které jsou momentálně přiděleny různým jiným procesům. Tabulka V proto musí pro každý proces obsahovat seznam všech prostředků, na které proces čeká; algoritmus pak musí projít všechny 'větve' vzájemného čekání.

6.5 Ovládání konkrétních zařízení

V následujících odstavcích se seznámíme s některými specifickými vlastnostmi některých běžně používaných zařízení a ukážeme si důsledky těchto vlastností jak na konstrukci ovladačů zařízení, tak i pro správce prostředků.

6.5.1 Sdílená data

Jak již víme, na sdílená data se můžeme dívat jako na speciální případ vyhrazeného prostředku⁶⁹, a tedy pro ně platí vše, co bylo o vyhrazených prostředcích řečeno v předcházejících odstavcích (s jedinou výjimkou - sdílená data samozřejmě není možné virtualizovat).

Výhoda sdílených dat oproti jiným prostředkům však spočívá v tom, že je procesy vyhrazují zpravidla na velmi krátkou dobu. V jednoduchém operačním systému pracujícím na počítači s jediným procesorem tedy obvykle postačí po dobu přístupu ke sdíleným datům zakázat přerušení (takže po tuto dobu nemůže v žádném případě dojít ke změně kontextu) a vše bude v naprostém pořádku. Tak je tomu např. v operačním systému XINU nebo - chceme-li příklad v praxi užívaného systému - v operačním systému EPOC.

Komplikovanější operační systém však někdy tuto alternativu zvolit nemůže, zvláště v případě, kdy je určen i pro řízení multiprocesorových počítačů. Pak je nutné implementovat klasické kritické sekce s využitím semaforů (viz odstavec 5.4.1.3). Potom pochopitelně existuje nebezpečí zablokování (v případě, že dva procesy potřebují vstoupit do dvou kritických sekcí v různém pořadí); toto nebezpečí je naštěstí vzhledem ke krátkému času, který procesy v kritických sekčích tráví, velmi nízké.

Přesto existují operační systémy, které preventivně zamezují i takovému zablokování - např. operační systém OS/MVT využívá v tomto případě strategii hierarchického přidělování (viz odstavec 6.4.2). Ta je pro daný účel poměrně vhodná, protože je velmi jednoduché ji implementovat a její hlavní nevýhoda - zbytečně dlouhodobé blokování prostředků - se u krátkodobě užívaných kritických sekcí příliš neprojeví.

⁶⁹Používání pojmu 'sdílený' a 'vyhrazený' je v tomto případě poněkud matoucí, při bližším pohledu však dává dobrý smysl: přístup ke sdíleným údajům - to jest k takovým údajům, které používá více procesů - musí být dočasně vyhrazen jedinému procesu, aby mu v údajích ostatní procesy nemohly nadělat nepořádek, dokud své úpravy nedokončí.

6.5.2 Systémový časovač

Systémový časovač generuje přerušení v pravidelných intervalech. Operační systém tohoto přerušení obvykle využívá pro splnění dvou úkolů: sdílení času při preemptivním multitaskingu a 'probuzení' procesů, které se pomocí služby 'sleep' vzdaly procesoru na určitou předem danou dobu.

Zajištění preempce je poměrně jednoduché. Obslužná rutina přerušení sleduje, kolik času ještě zbývá právě běžícími procesy; jakmile zjistí, že jeho čas vypršel, zajistí prostě přeplánování pomocí služby 'resched'. Připomeneme-li si implementaci služby 'resched' (popsané v odstavci 5.3.5.1), je zřejmé, že to stačí - služba zajistí, že všechny procesy se stejnou prioritou se budou v běhu pravidelně střídat.

Pro probouzení 'spících' procesů musí obslužná rutina přerušení zpracovávat delta list - snižuje prostě 'deltu' jeho první položky. Jakmile 'delta' dojde k nule, znamená to, že je zapotřebí první proces z delta listu převést do ready fronty (stejně jako všechny případné další procesy s nulovou 'deltou').

Podrobný popis obslužné rutiny přerušení časovače čtenáři vlastně dlužíme ještě z odstavce 5.3.7, kde jsme se zabývali službami správce času - ty však samozřejmě s obslužnou rutinou přerušení časovače velmi úzce spolupracují. Uvedeme proto zdrojový kód odpovídající obslužné rutině přerušení časovače v XINU⁷⁰. Nejprve však musíme vyřešit jeden problém, na který bychom právě kvůli preempci mohli nepříjemně narazit - totiž narušení komunikace s rychlými zařízeními.

⁷⁰Obslužná rutina v originálním XINU byla kvůli efektivita napsána v assembleru. Pro zvýšení přehlednosti jsme ji převedli do jazyka C; přitom je ale samozřejmě nutné mít na paměti, že obslužná rutina přerušení časovače je snad vůbec nejčastěji prováděným úsekem kódu v celém operačním systému, a její vysoká efektivita je tedy více než žádoucí. Jinými slovy, pro praktickou implementaci operačního systému by bylo vhodné tuto rutinu naprogramovat opět v assembleru.

6.5.2.1 Pozastavení hodin

I v případě, že obslužnou rutinu přerušení časovače naprogramujeme velmi efektivně v assembléru, existuje přece jen jedna situace, kdy bude trvat nezanedbatelnou dobu - dojde-li totiž v jejím rámci k přeplánování. Pro všechny běžné akce, které počítač provádí, je i tato doba dostatečně krátká, aby nenarušila žádnou funkci; je však jedna výjimka - rychlý synchronní přenos dat po blocích.

Představme si, že máme velmi rychlou sériovou linku, po které se z hlediska programu přenášejí data ne po jednotlivých bytech (jak tomu bylo v případě ovladače popsaného v odstavci 6.2.1), ale po celých blocích⁷¹ - což je samozřejmě rychlejší. Takovým zařízením může být např. jednoduchý síťový hardware, se kterým se seznámíme v kapitole 8.

Pokud by uprostřed přenosu bloku po takovémto zařízení došlo k přerušení časovače, které vyvolá přeplánování, mohli bychom přijít o nějaká data; jednotlivé byty v odesílaném nebo přijímaném bloku totiž mohou jít za sebou tak rychle, že za dobu, kterou by zabraňla obslužná rutina přerušení časovače a přeplánování, jich mělo přijít nebo odejít několik.

Na první pohled se zdá, že bychom tomuto nebezpečí dokázali zabránit dvěma způsoby:

- Dokonalejší technické vybavení, které přenáší blok jako celek - ne tedy po jednotlivých bytech jdoucích velmi rychle za sebou - by nás zbavilo popsaných problémů. Takové technické vybavení však nemusíme mít vždy k dispozici.
- Nemohli bychom ale také prostě po dobu přenosu bloku zakázat přerušení, aby nás časovač 'neobtěžoval', podobně jako to děláme v kritických sekcích? Ukazuje se, že to není možné, z velmi jednoduchého důvodu:

⁷¹Z hlediska ovladače zařízení se samozřejmě data budou přenášet stále byte po bytu; rozdíl bude ležet hlavně v horní polovině, která bude preferovat služby 'read' a 'write' namísto služeb 'getc' a 'putc'.

sám ovladač zařízení je řízen přerušením; po zákazu přerušení by tedy prostě přestal pracovat.

Bylo by samozřejmě možné přeprogramovat ovladač zařízení tak, že namísto obslužné rutiny přerušení by byl realizován jako čekací smyčka, která např. při čtení neustále zjišťuje, je-li na zařízení k dispozici znak, a jakmile tomu tak je, znak odebere (zápis by vypadal analogicky). Pak by bylo skutečně možné po dobu přenosu bloku zakázat přerušení a vše by fungovalo; i tento přístup však má své nevýhody:

* Ovladačům, které v čekací smyčce sledují stav zařízení, je lepší se vyhnout. Mezi jejich nevýhody patří např. to, že i v době, kdy na zařízení není žádný znak připraven, se se zařízením komunikuje a sběrnice není volná pro případné koprosesory.

* Doba přenosu bloku je přece jen příliš dlouhá na to, abyhom mohli s klidným svědomím zakázat přerušení. Uděláme-li to, přijde mezičím o několik přerušení časovače; systémové 'hodiny', které probouzejí 'spící' procesy, se pak začnou zpožďovat, což může být zvláště nepříjemné v systémech pro práci v reálném čase.

Operační systém XINU proto obsahuje jiný mechanismus, kterému říkáme pozastavení hodin (anglické 'deferred clock' je o něco přesnější). Jeho princip spočívá v tom, že na potřebnou dobu se nezakáže přerušení, ale zpracování jeho obslužné rutiny; operační systém si však pamatuje, jak dlouho to trvalo a po obnovení normálního stavu ihned dožene vše, co by býval měl provést, dokud byly hodiny pozastaveny (ve skutečnosti vlastně nejsou pozastaveny hodiny jako takové, ale pouze jejich obsluha).

Služby, které zajišťují pozastavení a opětovné spuštění hodin, jsou v XINU implementovány následujícím způsobem. Služba 'strtclk' potřebuje speciální funkce správce front, o kterých jsme dosud nehovořili - jedná se o služby, které

umožňují procházet delta list a zjišťovat a měnit 'delty' procesů, které jsou v něm uloženy, aniž bychom procesy z delta listu odebírali⁷².

Pro zjištění prvního procesu můžeme použít standardní službu 'firstid'. Pro další práci pak budeme potřebovat službu 'nextid', jejímž parametrem je fronta i číslo již zpracovaného procesu, a služba vrací číslo dalšího procesu ve frontě nebo zápornou hodnotu, není-li již ve frontě žádný proces. Další služba 'deltaid' vrátí adresu 'delty' zadaného procesu v zadáné frontě.

```
/* ssclock.c - stopclk, startclk */

void stopclk(void)
{
    defclk++;
}

void strtclk(void)
{
    char x;
    int difftime;

    sdisable(x);
    if (defclk==0 || --defclk>0) {
        restore(x);
        return;
    }
    preempt-=difftime=clkdiff;
    clkdiff=0;
    if (slnempty) {
        int pid=firstid(clockqueue);
        int *delta;

        do {
            if (*delta=deltaid(clockqueue,pid))<difftime) {
                difftime-=*delta;
                *delta=0;
            } else break;
            pid=nextpid(clockqueue,pid);
        } while (pid>0);
        if (pid>0) /* neprošli jsme celou frontu */
    }
}
```

⁷²Ve skutečném XINU takové služby vůbec neexistují a funkce 'strtclk' pracuje přímo s frontou; to je sice programátorským pohodlné, přineslo by to však problémy ve chvíli, kdy bychom se pokusili změnit interní mechanismy správce front.

```
*delta-=difftime;
wakeup();

if (preempt<=0)
    resched();
restore(x);
}

/* end of file */
```

Globální proměnná 'defclk' slouží podobně jako obecné semafory: ukazuje, kolikrát byla volána služba 'stopclk' bez ukončení odpovídající službou 'strtclk'. Kdykoli je tedy hodnota proměnné 'defclk' větší než nula, jsou hodiny pozastaveny. Ze stejného důvodu, jako je tomu u semaforů, musíme také test a nastavení proměnné 'defclk' na začátku služby 'strtclk' uzavřít do kritické sekce (chráněné zákazem přerušení).

Implementace služby 'stopclk' je triviální a nepotřebuje žádné další vysvětlování. Služba 'strtclk' je o dost komplikovanější; mechanismus její funkce však není složitý:

Služba nejprve ověří, jsou-li vůbec hodiny pozastaveny (test 'defclk == 0'), a pokud ano, sníží počet pozastavení dekrementováním proměnné 'defclk'. Je-li pak výsledná hodnota proměnné nulová, je zapotřebí hodiny opět rozběhnout.

V takovém případě služba zjistí počet odložených zpracování obslužné rutiny časovače, který je uložen v globální proměnné 'clkdiff', a tuto proměnnou využije (aby se do ní při příštím pozastavení hodin opět ukládaly korektní hodnoty). Zároveň služba odpovídajícím způsobem sníží hodnotu globální proměnné 'preempt'. S touto proměnnou jsme se již setkali při popisu služby 'resched' v odstavci 5.3.5.1; operační systém v ní udržuje informaci o tom, kolik času zbývá aktivnímu procesu do přeplánování v rámci sdílení času. Musíme ji tedy samozřejmě snížit právě o tolik, kolikrát obslužná rutina časovače 'nebyla k dispozici'.

Následující úsek kóduje složitý pouze na první pohled. Funkce použije globální proměnnou 'slnempty', aby zjistila, je-li delta list neprázdný (připomeňme si, že tuto proměnnou korektně nastavily služby 'sleep' a 'wakeup', popsané v odstavci 5.3.7). Čekají-li nějaké procesy na 'probuzení', postupuje služba následujícím způsobem:

- Všem procesům, jejichž zbývající čas je nižší než počet 'odložených' zpracování obslužné rutiny přerušení časovače, nastaví 'deltu' na nulu. Navíc je při každém kroku nutné snížit obsah proměnné 'difftime', protože vynulováním 'delty' vlastně 'jakoby' patřičný počet odložených přerušení obsloužíme.
- Jakmile narazíme na proces s příliš velkou 'deltou', přestaneme frontu procházet (to zajistí příkaz 'else break'). Příliš velká 'delta' totiž znamená, že takový proces bude čekat ještě déle.

'Deltu' takového procesu však ještě musíme snížit o zbývající počet odložených přerušení, který máme v proměnné 'difftime'.
- Nakonec stačí zavolat službu 'wakeup', která automaticky zajistí provedení všech procesů s nulovou 'deltou' do ready fronty a korektně nastaví globální proměnné.

Na samém konci se služba 'strtclk' postará i o sdílení času - jestliže aktivnímu procesu jeho čas uložený v globální proměnné 'preempt' vypršel, zajistí služba přeplánování. Uvědomme si, že je to zcela korektní i v případě, že došlo k přeplánování v rámci služby 'wakeup' (a další přeplánování se tedy stává nepotřebným) - proměnná 'preempt' prostě obsahuje v každém případě zbývající čas aktivního procesu, takže příkaz

```
if (preempt<=0)
    resched();
```

si můžeme dovolit zavolat naprosto kdykoliv.

6.5.2.2 Obslužná rutina přerušení časovače

Vlastní obslužná rutina přerušení časovače je již velmi jednoduchá - vlastně v ní pouze využíváme hotových služeb a naplněných globálních proměnných.

Ukážeme také velmi jednoduchý mechanismus dělení frekvence časovače, který je velmi výhodný v případě, že časovač vyvolává přerušení z hlediska

Operační systémy

operačního systému zbytečně často, takže obsluha každého z nich by pouze zbytečně zvyšovala režii systému. Frekvenci časovače budeme dělit v poměru 1:CLKDIV, kde 'CLKDIV' je symbolická konstanta:

```
/* clk_iohandler.c */  
  
void interrupt clk_iohandler()  
  
static cnt=1;  
  
if (--cnt==0) {  
    cnt=CLKDIV;  
    if (defclk) {  
        clkdiff++;  
        return;  
    }  
    if (slnempty && --*sltop==0)  
        wakeup();  
    if (--preempt)  
        resched();  
}  
/* end of file */
```

První příkaz 'if' slouží k dělení hodinové frekvence - jeho tělo bude provedeno pouze při každém 'CLKDIV'-tému přerušení.

Obslužná rutina pak nejprve ověří, nejsou-li hodiny pozastaveny; pokud by tomu tak bylo, inkrementuje pouze proměnnou 'clkdiff' a ihned skončí.

Pokud hodiny pozastaveny nejsou, ověří rutina, je-li delta list neprázdný a případně dekrementuje 'delta' jeho prvního procesu (na tu ukazuje globální proměnná 'sltop'). Je-li 'delta' po dekrementaci nulová, zavolá se služba 'wakeup' pro 'probuzení' všech procesů, které to vyžadují.

Nakonec rutina sníží zbývající čas aktivního procesu uložený v proměnné 'preempt', a je-li to zapotřebí, zajistí přeplánování.

6.5.2.3 Preempce a bezpečnost

Teprve nyní vlastně známe celý mechanismus preemptivního multitaskingu. Je proto vhodné si říci také několik slov o jeho bezpečnosti.

V odstavci 5.2.4 jsme - přibližně řečeno - definovali preemptivní multitasking jako takový multitasking, ve kterém může operační systém odebrat kterémukoli procesu procesor, uzná-li to za vhodné, aniž by to proces musel explicitně umožnit. Nyní, když víme, jakým způsobem to operační systém provede, bychom se měli zabývat také tím, nemůže-li proces operačnímu systému nějak zabránit, aby mu procesor odebral (což by samozřejmě při chybném nebo úmyslně destruktivním procesu mohlo vést ke zhroucení operačního systému).

Na první pohled je zřejmé, že existují dva mechanismy, kterých by proces mohl využít: buď by mohl extrémně zvýšit vlastní prioritu, nebo by se mohl pokusit zakázat přerušení; obslužná rutina časovače by pak nikdy nebyla volána a k preemci by nemohlo dojít. Rozeberme oba případy podrobněji:

- Proces vůbec 'neví', kam systém ukládá jeho prioritu. Pro změnu priority proto musí nutně volat službu operačního systému; je velmi snadné tuto službu naprogramovat tak, aby si proces nemohl přidělit prioritu příliš vysokou.

V systému bez ochrany paměti by se proces samozřejmě mohl pokusit vyhledat tabulku procesů a svou prioritu zvýšit přímo; nemáme-li však k dispozici ochranu paměti existuje tolik nejrůznějších způsobů, jak omylem nebo úmyslně zapříčinit zhroucení operačního systému, že tato možnost vlastně už nic nezhorší.

- Nebezpečnější je případ, kdy proces zakáže přerušení. Řešení je jednoduché, závisí ale na technických prostředcích: uživatelským procesům je prostě nutné zamezit zakazování přerušení.'

Moderní procesory, které rozlišují uživatelský a systémový pracovní režim, tuto problematiku řeší prostě tak, že v uživatelském režimu zakazovat přerušení nemůžeme - pokus o tuto akci vyvolá výjimku,

Operační systémy

kterou obslouží operační systém a s 'provinilým' procesem něco vhodného provede - nejspíše jej okamžitě ukončí.

Implementujeme-li však operační systém pro některý procesor, který touto možností nedisponuje, potřebujeme pro alespoň základní zabezpečení doplňkové technické vybavení. Příkladem mohou být kapesní počítače PSION - jejich základem je koncepčně dost zastaralý mikroprocesor V30, jemuž se o různých pracovních režimech ani nezdá; počítače PSION však jsou vybaveny hardwarovým čítačem, který sleduje dobu, po kterou je přerušení zakázáno, a je-li tato doba delší než asi vteřina (což je z hlediska počítače nesmírně dlouhá doba), využije se mechanismus nemaskovatelného přerušení a operační systém má šanci proces, který přerušení na tak dlouho zakázal, ukončit.

6.5.3 Obrazovka, klávesnice, myš

Každé ze tří zařízení zmíněných v nadpisu tohoto odstavce samozřejmě může mít vlastní nezávislý ovladač. Snad všechny moderní operační systémy ale obsahují server, který má přiděleny všechna tato zařízení a nabízí ostatním procesům komplexní služby jednoho nedílného 'systému pro interaktivní komunikaci s uživatelem'.

Vlastnostmi serveru, který tato zařízení obsluhuje, se na tomto místě nebudeme podrobně zabývat - tomuto tématu je věnována podstatná část druhého dílu knihy. Namísto toho se podíváme na specifické vlastnosti ovladačů, které takový server pro tato zařízení musí mít k dispozici. Nebudeme se zabývat ovladačem myši, na kterém není z našeho hlediska nic zajímavého (snad kromě toho, že rozhraní ovladače by mělo být navrženo tak, aby bylo možné myš snadno nahradit jiným obdobným zařízením, jako je např. digitizér).

6.5.3.1 Obrazovka

Většina mikropočítačů, se kterými se můžeme setkat, nemá specializovaný grafický procesor; někdy je sice práce s obrazovkou podporována speciálními obvody (Commodore Amiga, ATARI ST s blitterem), ty však jsou poměrně velmi jednoduché. Díky tomu je ovladač obrazovky netypický v tom, že nepotřebuje vůbec dolní polovinu: neexistuje vlastně zařízení, na které by bylo zapotřebí čekat. Horní polovina ovladače přímo zapíše potřebná data do videopaměti - ať již přímo nebo s využitím blitteru - a je hotovo.

Moderní operační systémy však této možnosti zjednodušení často nevyužívají, aby dosáhly větší efektivity. Musíme si totiž uvědomit, že sám přístup do videopaměti procesor zdržuje více, než kdyby pouze požadavky na výstup uložil do sdílené paměti pro dolní polovinu ovladače. K videopaměti totiž musí mít přístup jak sám mikroprocesor, tak i obvody generující videosignál. Oba obvody se tedy 'přetahuji' o přístup k paměti; generátor signálu přitom musí mít vyšší prioritu než mikroprocesor, aby byl obraz nerušený⁷³ - mikroprocesor je tedy odsouzen do postavení toho, kdo čeká, až bude zrovna chvíli volno. Dobře navržený ovladač obrazovky proto pouze přebere od procesu požadavky na výstup a pak je ve vhodných chvílích postupně zobrazuje, aniž by sám proces musel čekat. Příkladem takového systému je např. EPOC.

Horní polovina ovladače obrazovky má jinou podstatnou úlohu - musí zajistit virtualizaci obrazovky, protože za běžných okolností samozřejmě existuje řada procesů, které chtějí najednou vypisovat své údaje. V zásadě existují dva způsoby virtualizace obrazovky: buď může mít každý proces vlastní 'obrazovku' a uživatel volí, která z mnoha 'obrazovek' je právě vidět (tak tomu je např. ve starších verzích operačního systému OS/2), nebo ovladač může zajistit rozdělení obrazovky do jednotlivých oken a každý proces pak může používat vlastní okno. Tato druhá možnost, kterou zavedla firma Apple u svých počítačů

⁷³Pokud má vyšší prioritu mikroprocesor, je situace ještě horší - programy pak totiž musí samy testovat, zda je videopaměť zrovna volná, a časové ztráty jsou ještě daleko větší. Tak tomu např. je (či spíše bývalo) s počítači IBM PC vybavenými videoadaptérem CGA.

Macintosh⁷⁴, je dnes považována za nejvhodnější a operační systém, který by ji nepodporoval, je prakticky odsouzen k neúspěchu.

Některé operační systémy mohou obě techniky kombinovat - je tedy k dispozici více virtuálních obrazovek, přitom každá z nich (nebo jen některé z nich) může obsahovat více oken. Za běžných okolností to nemá valný význam; může to však být pohodlné u systémů, které se běžně provozují s levnými monitory s nízkým rozlišením a malou obrazovkou (Amiga DOS) nebo které jsou na minimální zobrazovací jednotku odkázány díky svému určení jako kapesní počítač (EPOC).

Horní úroveň ovladače by navíc měla nabízet služby velmi vysoké úrovně. Pro tvůrce operačního systému je sice velmi pohodlné nechat aplikační programátory aby počítali 'pixely'; všichni ale pak musí dost komplikovaným způsobem počítat s potenciálním připojením zobrazovací jednotky s jiným rozlišením a navíc musí systém obsahovat speciální ovladače pro jiná výstupní zařízení.

Jestliže naopak ovladač nabízí grafické služby na zcela abstraktní úrovni 'kreslení na ploše' (nebo ještě raději v třírozměrném prostoru), programují se aplikace velmi příjemně a není nejmenší problém bez explicitní podpory programu přesměrovat jeho grafický výstup na zcela libovolné výstupní zařízení. V tomto smyslu je dnes asi nejdokonalejší obrazový ovladač operačního systému NeXTStep s jeho interprety Display PostScriptu pro dvourozměrné kreslení a jazyka RenderMan pro třírozměrné scény.

⁷⁴Přesně řečeno, základní myšlenky takového systému vznikly ve vývojových pracovištích firmy XEROX a první počítač na trhu, vybavený takovým systémem, byl Apple Lisa. Macintosh však byl prvním počítačem s virtualizací obrazovky založenou na systému oken, který se na trhu prosadil.

6.5.3.2 Klávesnice

Na ovladači klávesnice za normálních okolností není nic zajímavého - je to zcela standardní a obyčejný ovladač zařízení. V naší zemi je však díky Mistru Janu Husovi úloha ovladače klávesnice výrazně posílena, protože musí umožnit vkládání nabodeníček krátkých i dlouhých (můžeme se utěšovat tím, že třeba Japonci jsou na tom ještě hůř).

Naprostá většina ovladačů klávesnice se stará o tuto problematiku kompletne, tj. nejen že obsluhuje přiřazení jednotlivých kódů klávesám, ale zároveň zajišťuje funkci tzv. mrtvých kláves (tj. kláves, které mění význam následující klávesy, jako např. čárka nebo háček). Jakkoli je to pro nás pohodlné, protože nám to umožnuje poměrně snadno upravit standardní ovladač pro podporu našeho jazyka, je vhodné si uvědomit, že se nejedná o právě nejčistší řešení. Ovladač klávesnice sám by měl zajistit pouze přiřazení významů jednotlivým klávesám; interpretace akcentů by měla být součástí vyšší vrstvy programového vybavení, spojené s platným kódováním znaků (jako tomu je např. v operačním systému NeXTStep).

6.5.4 Tiskárna

Tiskárna má dosud do jisté míry zvláštní postavení. To je dáno tím, že bez ohledu na zřejmé výhody elektronického zpracování dat i elektronické komunikace je dodnes bohužel daleko nejčastějším datovým nosičem - obyčejný papír.

Typický případ dnešní schizofrenie v užívání počítačových systémů⁷⁵ je situace, kdy pracovníci nějaké firmy připraví údaje pro odeslání partnerskému podniku. Grafy zpracují na počítači, průvodní dopis napíší na počítači, vše pak vytisknou na papír (!) a ten ofaxují. Příjemce se pak nějakou dobu snaží obsah přijatého faxu převést na svůj počítač pomocí programu OCR; díky kombinaci mizerného přenosu dat prostřednictvím oné tiché pošty, která se v naší zemi eufemisticky nazývá 'telefonní linky', a problémů plynoucích z interpretace

⁷⁵A to i v počítačově civilizovaných zemích - u nás dosud nejsme dokonce ani tak daleko.

Operační systémy

znaků s diakritickými znaménky se to samozřejmě nepodaří. Příjemce pak fax buď do počítače opisuje ručně, nebo založí do desek, se kterými se fax během týdne spolehlivě a definitivně ztratí.

Důsledkem tohoto poněkud absurdního přístupu je to, že i u nejjednodušších a prakticky jednoúčelových systémů patří tiskárna mezi vůbec nejpoužívanější zařízení. Je to snad vůbec jediný příklad zařízení, které se vyplatí 'virtualizovat' i v nemultitaskových operačních systémech.

Virtualizace je zde ovšem uvedena v uvozovkách, protože se sice jedná o stejnou techniku, ale použitou k jinému účelu a jiným způsobem. U nemultitaskových operačních systémů je úkolem Virtualizace' pouze zajistit z hlediska aplikací velmi vysokou rychlosť tisku. Ovladač tiskárny pak musí být poměrně velmi složitý, aby si v rámci operačního systému vynutil Vlastní multitasking', kdy by se střídal s právě běžící aplikací o strojový čas, aby mohl na pozadí tisknout.

6.5.5 Disk

Ovladač disku nebo diskety je pro operační systém velmi důležitý - jedná se totiž o klíčové zařízení, jehož snížená výkonnost se velmi markantně projeví na sníženém výkonu celého operačního systému. Budeme se mu proto věnovat podrobněji než ostatním zařízením a uvedeme i příklady zdrojových textů pro implementaci velmi jednoduchého ovladače, který je součástí operačního systému XINU⁷⁶.

Základní princip ovladače disku se samozřejmě nijak neliší od obecného ovladače, popsaného v odstavci 6.2.1; oproti uvedeným příkladům zde však existují podstatné rozdíly:

⁷⁶V PC-XINU ani v ST-XINU zvláštní ovladač disku nebyl použit - obě implementace využívaly ovladač disku a systém souborů operačního systému MS DOS, respektive ATARI TOS. Pro praktické využití by však bylo nutné vlastní ovladač doplnit, protože ovladače obou zmíněných systémů jsou navrženy pro práci v jednoprogramovém prostředí a pro preemptivní multitasking se nehodí.

- Disk nepředává data počítači po jednotlivých bytech, ale po tzv. sektorech, jejichž velikost se pohybuje kolem 512 bytů⁷⁷. Tomu je nutné přizpůsobit jak dolní, tak i horní polovinu ovladače - dolní polovina bude načítat (nebo zapisovat) údaje po celých sektorech, zatímco horní musí zajistit splnění požadavků, jejichž velikost není dělitelná velikostí sektoru.
- Zatímco typické zařízení, které jsme v odstavci 6.2.1 ilustrovali, předává sekvenčně jednotlivé byty, je disk zařízením s přímým přístupem - jeho ovladač tedy musí předat zařízení nejen požadavek na spuštění přenosu, ale také adresu požadovaného sektoru na disku.

Nemůžeme ale dost dobře po uživatelských programech chtít, aby se zabývaly detaily jednotlivých povrchů, stop a sektorů disku. Ovladač proto musí nabídnout pohled na disk jako na pole sektorů, indexované Čísla 0 až N-1, je-li na celém disku dohromady N sektorů. Ovladač pak musí sám interně překládat tyto indexy na odpovídající diskové adresy.

- Nesmíme také zapomínat, že disk je obvykle sdíleným zařízením - je tedy velmi snadno možné, že v době, kdy jeden proces čeká na splnění svého požadavku na diskový přenos, přijde jiný proces s vlastním požadavkem na práci s diskem. Náš příklad ovladače obdobný problém nemusel nijak řešit - další znak pro výstup uložil do bufferu, další znak pro vstup prostě z bufferu odebral. Ovladač disku však musí udržovat frontu požadavků, které postupně splňuje.

Řada operačních systémů se snaží zvyšovat efektivitu práce s diskem přeuspřádáváním jednotlivých požadavků ve frontě na základě znalosti fyzického i logického formátu disku.

⁷⁷K předávání takovýchto bloků dat se používá speciální obvod, nazývaný DMA (Direct Memory Access). Ten pracuje jako velmi jednoduchý koprocesor: je-li aktivní, střídá se s procesorem na systémové sběrnici a přenáší data přímo do nebo z operační paměti. Obvod DMA je součástí řadičů snad všech rozumných disků; výjimkami potvrzujícími pravidlo jsou počítač IQ151 a podle některých údajů také počítače IBM PC s tzv. IDE pevným diskem.

Operační systémy

Uvědomme si, že čteme-li z disku dva sektory, které jsou uloženy za sebou na jediné stopě, může řadič oba sektory přečíst v průběhu jediné otáčky disku⁷⁸; jestliže však budeme číst jeden sektor z první stopy a následující sektor ze stopy poslední, musí mechanika disku mezi čtením přemístit čtecí hlavičku, což trvá nezanedbatelnou dobu.

Operační systém proto může třídit požadavky ve frontách takovým způsobem, aby minimalizoval potřebu vystavování diskových hlaviček. Tato metoda není špatná; musíme si však uvědomit, že moderní SCSI disky bývají samy vybaveny dostatečně inteligentním řadičem (obsahujícím často i vlastní cache paměť), takže to nemusí být nutné.

Podívejme se nyní na základy implementace ovladače disku v operačním systému XINU.

6.5.5.1 Služby ovladače

Nejprve je zapotřebí rozhodnout, jak mají fungovat jednotlivé služby. Z hlediska ovladače samotného by bylo asi nejpřirozenější, aby uživatel nejprve použil službu 'seek' pro určení požadovaného sektoru, a pak data načetl službou 'read' nebo zapsal pomocí služby 'write'.

Daleko méně přirozené by to však bylo z hlediska uživatelského programu - ten by totiž samozřejmě musel službu 'seek' volat skoro před každým použitím služeb pro čtení nebo zápis. XINU proto nepatrнě modifikuje sémantiku obecných služeb, takže parametr služeb 'read' a 'write', který normálně udává délku načítaných nebo zapisovaných dat, bude nyní určovat právě číslo požadovaného sektoru na disku. Délku není zapotřebí specifikovat; stejně se vždy přenáší právě jeden sektor.

⁷⁸Ve skutečnosti se většina disků otáčí příliš rychle, takže dva sektory které následují bezprostředně za sebou, lze přečíst až během dvou otáček disku - mezi koncem prvního a začátkem druhého sektoru má řadič disku prostě příliš málo času, než aby stihl zahájit nové čtení. Na následujících úvahách to však nic nemění - pouze je třeba řadit požadavky tak, aby se sektory četly pokud možno 'ob jeden' (nebo ob dva či ob tři u ještě rychlejších disků).

Službu 'seek' pak vlastně k ničemu nepotřebujeme (v [XINU] je uvedeno její využití pro zvýšení efektivity celého systému). Z ostatních služeb jsou velmi důležité služby 'open' a 'close', které umožňují práci s lokálním systémem souborů; zmíníme se o nich podrobněji až v kapitole 7, která je systému souborů věnována celá.

6.5.5.2 Potřebná data

Deklarace potřebných datových struktur vychází z obecného ovladače, jehož příklad jsme uvedli v odstavci 6.2.1; vzhledem k předávání vždy právě jednoho sektoru je ale struktura daleko jednodušší - po odstranění položek potřebných pro systém souborů (jímž se nyní nebudeme zabývat) zde zbyde pouze fronta požadavků a rutiny pro řízení disku. Stejně je i to, že datové struktury jsou uloženy v poli indexovaném položkou 'dvminor' - ovladač samozřejmě může řídit více disků.

Patřičné deklarace by mohly vypadat například takto:

```
/* disk.h */

enum {                                /* diskové operace */
    DREAD,                            /* čtení z disku */
    DWRITE                           /* zápis na disk */
};

struct dreq {                          /* požadavek */
    int drpid;                         /* žádající proces */
    unsigned long drdba;               /* disková adresa */
    char *drbuff;                      /* buffer */
    char drop;                          /* operace: DREAD, DWRITE */
    int drerror;                        /* chybový stav, 0 bez chyby */
    struct dreq *drnext;                /* ukazatel na další požadavek */
};

struct dsblk {
    struct dreq *dreqlst;              /* fronta požadavků */
    void (*_dwrite)();                 /* zápis na disk */
    void (*_dread)();                  /* čtení z disku */
    void (*_derror)(void);             /* chyba přenosu */
};
```

Operační systémy

```
extern struct dsblk dstab[];  
/* end of file */
```

Význam jednotlivých polí je celkem zřejmý. Tabulka 'dstab' obsahuje pro každý disk jednu položku, v ní je po vypuštění dat potřebných pro systém souborů pouze seznam požadavků a tři rutiny. První dvě rutiny mají dva parametry - adresu sektoru na disku a adresu bufferu. Rutina '_dwrite' spustí zápis bufferu do sektoru, rutina '_dread' spustí čtení sektoru do bufferu. V obou případech řadič disku po ukončení operace vyvolá přerušení. Rutina '_derror' po přenosu zjistí, proběhl-li přenos bez problémů (pak vrátí nulu) nebo došlo-li k nějaké chybě.

Každý požadavek obsahuje diskovou adresu (kódovanou do položky 'drdba'), adresu bufferu v paměti, kód požadované operace a návratový kód chyby. Jednotlivé požadavky jsou uspořádány do jednostranně řetězeného seznamu pomocí ukazatelů 'drnext'. XINU s frontou požadavků pracuje tak, že první požadavek ve frontě je vždy ten, který je právě zpracováván zařízením; teprve po zpracování se požadavek z fronty vyřadí.

Poměrně důležité pro efektivitu celého systému je také navrhnut vhodný způsob komunikace s volajícím procesem. Šikovný může být například následující mechanismus:

- Je zřejmé, že proces který inicializoval čtení z disku, musí být pozastaven, dokud nebude čtení ukončeno, aby bylo jisté, že po návratu z patřičné služby budou načtená data v bufferu⁷⁹. Po splnění požadavku na čtení je tedy nutné vyřadit požadavek z fronty požadavků, přemístit odpovídající proces do ready fronty a vyvolat přeplánování. Proces pak již sám splněný požadavek odstraní z paměti.

⁷⁹ Inteligentnější ovladač by samozřejmě mohl čtení jen inicializovat a pak nechat proces dále běžet, dokud nebude data skutečně potřebovat; potom by proces zavolal třeba službu 'cntl', která by s vhodnými parametry zajistila čekání. Na uvedeném programu by se však nic zásadního nezměnilo - pouze by byl rozdělen na dvě části, první z nich by zůstala uvnitř služby 'read', druhá by se přestěhovala do služby 'cntl'.

- Naprosto jiná situace je v případě zápisu na disk. Zde by bylo zbytečné jakoli proces, který požadavek vyvolal, zdržovat. Proces po umístění požadavku na zápis do fronty běží klidně dál a ovladač po ukončení zápisu může klidně sám požadavek odstranit z fronty požadavků i z paměti.

6.5.5.3 Práce s frontou požadavků

Operační systém XINU přeuspořádával frontu tak, aby dostal sektory na stejně stopě do sousedních požadavků. V našem příkladu si situaci zjednodušíme a budeme požadavky do fronty zařazovat v jednoduchém pořadí FIFO ('first in first out', aneb kdo dřív přijde, ten dřív mele'). Fronta je ukončena hodnotou NULL v ukazateli 'drnnext'.

Práce s frontou požadavků je dostatečně složitá na to, abychom si pro ni vytvořili speciální služby, které pak vlastní rutiny ovladače budou jen využívat. Práci s frontou trochu zkomplikuje i to, že přece jen existují některé optimalizace, které musíme provést (protože jinak by byl ovladač disku opravdu zbytečně neefektivní):

- Představme si, že ve frontě čeká požadavek na zápis některého sektoru, a pak se objeví nový požadavek na čtení téhož sektoru. V tom případě si můžeme čtení z disku (a spoustu času) snadno ušetřit - stačí zkopirovat obsah bufferu, určeného pro zápis, do bufferu, kam se mají data načíst.
- Ve stejné situaci by se mohl objevit další požadavek na zápis do téhož sektoru. Pak je ale samozřejmě zbytečné zapisovat na disk první data a pak je hned přepsat dalšími - první požadavek na zápis můžeme snadno z fronty vyřadit.

Poznamenejme, že obdobným způsobem lze samozřejmě optimalizovat i jiné příkazy než čtení nebo zápis při implementaci komplexnějšího ovladače, který jimi bude disponovat.

Pro implementaci služeb budeme potřebovat také alokovat a uvolňovat paměť pro jednotlivé položky ve frontách požadavků. Pro zjednodušení zdrojových textů budeme předpokládat, že máme k dispozici služby 'new' a 'dispose', pracující podobně jako PASCALské 'new' a 'dispose'; realizace takových služeb je snadná s využitím maker a funkce 'sizeof' jazyka C⁸⁰.

Budeme také předpokládat, že máme k dispozici hotovou funkci

```
void bcpy(void *dest, void *src, unsigned len);
```

která zkopíruje 'len' bytů z adresy určené ukazatelem 'src' na adresu, určenou ukazatelem 'dest' (takové kopírování je velmi častým úkolem; vyplatí se proto napsat podobnou funkci velmi efektivně v assembleru a pak ji používat, namísto toho, abychom data kopírovali pomocí programového cyklu).

Pomocná služba 'dskopt', která ověří, je-li zapotřebí vůbec požadavek do fronty zařazovat (budeme ji volat při zařazování každého nového požadavku, narazíme-li ve frontě na jiný požadavek se stejnou diskovou adresou), by mohla vypadat například takto:

```
/* dskopt.c */
#include<disk.h>
/*
 * dskopt -- je třeba vůbec požadavek rq zařazovat, když
 *           je ve frontě požadavek q->drnnext? 1 ano, 0 ne
 */
int dskopt(struct dreq *q, struct dreq *rq)
{
    struct dreq *p=q->drnnext;
    if (p->drop==DWRITE && rq->drop==DWRITE) {
```

⁸⁰Služby, které budou využívat funkci (nebo makro) 'new' spoléhají na to, že paměť bude zaručeně alokována. 'New' proto musí obsahovat aparát, který v případě nedostatku paměti zajistí, že proces počká, až se paměť uvolní; takový aparát již však dávno známe - stačí semafor 'waitovaný' ve službě 'new' a 'signalovaný' ve službě 'dispose'. Poznamenejme také, že může být vhodné větší či menší blok paměti pro uspokojování těchto služeb vyhradit při startu systému, aby případné vyčerpání operační paměti neznemožnilo práci diskového ovladače.

```

/* nahradíme starý požadavek novým */
rq->drnext=p->drnext;
q->drnext=rq;
dispose(p->drbuff);
dispose(p);
return(0);
}

if (p->drop==DWRITE && rq->drop==DREAD) {
/* 'čteme' z bufferu místo z disku */
bcpy(rq->drbuff,p->drbuff,DBUFSIZ);
return(0);
}

if (p->drop==DREAD && rq->drop==DWRITE) {
/* zapíšeme do bufferu starého požadavku čímž je splněn */
unsigned long dskadr=rq->drdba;

for (;p!=NULL;p=p->drnext)
if (p->drdba==dskadr) {
    bcpy(p->drbuff,rq->drbuff,DBUFSIZ);
    p->drerror=0;
    ready(p->drpid,RESCHNO);
}
rq->drnext=p;
q->drnext=rq;
resched();
return(0);

return(1);
}
/* end of file */

```

Implementace neskrývá žádná tajemství - postupně testujeme tři možné situace vhodné pro optimalizaci.

Nejprve ověříme, zda starý i nový požadavek nechtějí na disk zapisovat. Pokud by tomu tak bylo, starý požadavek z fronty odstraníme a nový vložíme na jeho místo. (Je to o něco pohodlnější, než nepatrнě korektnější přístup, při kterém bychom starý požadavek odstranili a nový zařadili na konec. Takto napsaná služba 'dskopt' bude navíc fungovat zcela korektně i v případě, že budeme při zařazování požadavků do fronty používat optimalizaci 'podle sousedních

Operační systémy

sektorů'.) Starý požadavek odstraníme i z paměti, protože se jedná o požadavek na zápis, který již na původní proces není nijak vázán.

Druhou možností je to, že starý požadavek zapisuje na disk, zatímco nový chce tentýž blok číst. Pak pouze zkopiujeme data z bufferu starého požadavku do bufferu nového požadavku a nový vůbec nezařazujeme - je automaticky splněn. Z paměti jej však odstranit nemůžeme, protože se jedná o požadavek na čtení, který je spojen s původním procesem (ten jej zanedlouho odstraní sám).

Poslední a nejsložitější optimalizace se týká možnosti, že ve frontě je jeden nebo více požadavků na čtení z disku a nový požadavek chce uvedený sektor přepsat. Pak všechny požadavky splníme tím, že do jejich bufferu zkopiujeme data z bufferu nového požadavku⁸¹, vyřadíme je z fronty a aktivujeme odpovídající procesy. Musíme také vynulovat položky 'drerror', podle nichž procesy poznají, že 'přenos' proběhl bez chyb.

Nyní se konečně můžeme podívat na algoritmus vlastního zařazení požadavku do fronty. Je poměrně velmi jednoduchý: funkce 'dskenq' se pokusí zařadit požadavek do fronty a vrátí nenulovou hodnotu, jestliže se jí to podařilo, a nulu v případě, že požadavek byl 'automaticky' (optimalizační rutinou) splněn.

Funkce 'dskenq' používá jednu speciální službu, o které jsme se zatím nezmínili. Jedná se o službu 'run_disk'; ta spustí vlastní přenos z disku nebo na disk. Služba je napsána tak, že nesmí být volána, jestliže je fronta požadavků prázdná:

```
/* run_disk.c */
#include <disk.h>

void run_disk(struct dsblk *dsk)
{
    struct dreq *rq=dsk->dreqlst;
    switch (dr->drop) {
```

⁸¹ Je to vůbec korektní? Požadavky na čtení přišly přece dříve než požadavek na zápis; neměly by tedy vrátit původní, ještě nepřepsaná, data z disku? Zkuste si uvědomit je-li tento postup opravdu správný a proč (nebo proč ne) dříve, než dočtete na konec tohoto odstavce!

```

    case DREAD:
        dsk->_dread(dr->drlba,dr->buff);
        break;
    case DWRITE:
        dsk->_dwrite(dr->drlba,dr->buff);
        break;
}
}

/* end of file */

```

Implementace služby je zřejmá. Připomeňme, že po ukončení přenosu řadič vygeneruje přerušení a umožní tak dolní polovině ovladače přejít na další požadavek ve frontě.

Uvědomme si také rozdíl mezi touto službou a službou 'run_odev', kterou jsme použili při popisu obecného ovladače v odstavci 6.2.1. Zatímco služba 'run_odev' pouze povolila přerušení a o celý přenos se starala dolní polovina ovladače, zde horní polovina přímo inicializuje přenos⁸².

Zdrojový kód pomocné služby 'dskenq' by mohl vypadat takto:

```

/* dskenq.c */

#include <disk.h>

/*
 * dskenq -- do fronty v bloku dsk zařadíme požadavek rq
 */
int dskenq(struct dsblk *dsk,struct dreq *rq)
{
    struct dreq *q;
    if ((q=dsk->dreqlst)==NULL) {
        dsk->dreqlst=rq;
        rq->drnext=NULL;
    }
}

```

⁸²Použití toho či onoho mechanismu samozřejmě nezáleží na libovůli programátora, ale na tom, jak pracuje ovládané zařízení - proto jsme pro každý z obou příkladů volili jinou variantu. V praxi se snadno může stát, že budeme muset přeprogramovat ovladač obecného zařízení tak, aby horní polovina přímo inicializovala přenos nebo naopak ovladač disku tak, aby se celý přenos konal v režii dolní poloviny ovladače.

```
run_disk(dsk);
return(1);

}

for (;q->drnext !=NULL;q=q->drnext)
    if (q->drnext->drdba==rq->drdba && dskopt(q,rq)==0)
        return(0);
    q->drnext=rq;
    rq->drnext=NULL;
    return(1);
}

/* end of file */
```

Služba 'dskenq' pracuje také celkem samozřejmým způsobem. Nejprve zjistí, není-li náhodou fronta požadavků prázdná; pokud by tomu tak bylo, uloží do ní požadavek a spustí disk (který obratem prostřednictvím přerušení spustí dolní polovinu ovladače, která zajistí splnění požadavku - vypadá to komplikovaně, presto se jedná o poměrně velmi efektivní implementaci ovladače).

Jestliže fronta prázdná nebyla, služba ji celou projde a u každé její položky ověří, není-li možné optimalizovat a provést požadovanou akci bez skutečného přístupu na disk. Pokud se to podaří, služba samozřejmě ihned skončí.

Nakonec se nový požadavek uloží na konec fronty a jsme hotovi. V tomto případě není zapotřebí spouštět disk - neprázdná fronta ukazuje, že disk právě zpracovává nějaký požadavek (první ve frontě, ten, který jsme přeskočili použitím 'q->drnext'); až disk zpracování požadavku ukončí, vyvolá přerušení sám od sebe.

A jak to bylo s otázkou, kterou jsme si položili v poznámce 81 na straně 184? Situace není složitá a můžeme ji snadno rozdělit do dvou případů:

- V případě, že požadavky přišly od různých vzájemně nesynchronizovaných procesů, jedná se pravděpodobně o nějaký nedostatek v návrhu těchto procesů, protože jejich programátor prostě nemohl předem vědět, v jakém pořadí se oba požadavky - ten na zápis a ten na čtení - sejdou. Můžeme proto požadavky čtení zcela stejně správně splnit dodáním 'starých' nebo 'nových' dat; protože 'nová' data dokážeme dodat mnohem rychleji, není o čem uvažovat.

Pokud by však požadavky přišly od jediného procesu (nebo od dvou procesů synchronizovaných), musel by být požadavek na čtení nutně vydán až po požadavku na zápis - proces je totiž při vydání požadavku na čtení pozastaven, dokud není požadavek splněn; nemůže proto vydat další požadavek na zápis. Ve chvíli, kdy ho již vydat může, je požadavek na čtení z fronty dávno vyřazen. Oba požadavky se tedy ve frontě mohou sejít pouze při pořadí zápis-čtení.

To však neodpovídá popsané situaci, kdy požadavky na čtení již ve frontě jsou a požadavek na zápis je do ní teprve zařazován. Proto jsme také v tomto odstavci důsledně používali podmiňovacího způsobu - k této situaci prostě nemůže dojít. Jinými slovy, je-li zařazován požadavek na zápis sektoru do fronty, ve které již jsou nějaké požadavky na čtení téhož sektoru, musí být tyto požadavky generovány nezávislými procesy - a takovou situací jsme se zabývali již v minulém bodě.

6.5.5.4 Horní polovina ovladače

Horní polovina ovladače se skládá z funkcí 'dsk_read' a 'dsk_write' (které budou samozřejmě uloženy v odpovídajících položkách v tabulce zařízení, takže uživatelské programy je budou moci volat prostřednictvím logického systému zařízení, popsaného v odstavci 6.2.1.1 a v odstavci za ním následujícím).

Použijeme zde pomocnou službu 'disk_address', jejímž parametrem je číslo sektoru na disku. Služba převede toto číslo na fyzickou adresu diskového sektoru a tu vrátí. Implementace této služby je za prvé triviální a za druhé velmi úzce závislá na konkrétním disku; proto její zdrojový text nebudeme uvádět.

Nejprve si ukážeme implementaci služby 'dsk_read':

```
/* dsk_read.c */  
  
#include <disk.h>  
#include <conf.h>
```

```

/*
 * +----- požadavek na čtení z disku -----
 */

int dsk_read(struct devsw *dev, char *buff, unsigned snum)
{
    struct dreq *rq;
    int ret=0; /* bez chyby */
    char x;

    sdisable(x);
    new(rq); /* vyhradíme paměť */
    rq->drdba=disk_address(snum);
    rq->drpid=currpid;
    rq->drbuff(buff);
    rq->drop=DREAD;
    if (dskenq(&dstab[dev->dvmminor], rq)) {
        suspend(surrepid);
        ret=rq->drerror;
    }
    dispose(rq);
    restore(x);
    return(ret);
}
/* end of file */

```

Služba nejprve alokuje paměť pro nový požadavek pomocí služby 'new', a pak nastaví všechny položky v požadavku na příslušné hodnoty. Potom se pokusí požadavek zařadit do fronty pomocí služby 'dskenq'.

Jestliže požadavek byl do fronty zařazen, musí proces čekat na jeho splnění. Čekání je zajištěno nejjednodušším možným způsobem: proces se vzdá procesoru a přemístí se do stavu 'suspended'. Procesor mu bude opět přidělen teprve tehdy, až dolní polovina ovladače ukončí zpracování požadavku; pak si také bude proces moci v položce 'drerror' přečíst, proběhl-li přenos bez problémů.

Pokud požadavek do fronty zařazen nebyl, znamená to, že byl splněn ihned. V každém případě služba nakonec jen uvolní paměť, přidělenou požadavku, obnoví stav přerušení a vrátí informaci o tom, byl-li přenos bezchybný.

Služba 'dsk_write', tak, jak je v XINU navržena, je určena pro použití **výhradně** vyššími vrstvami operačního systému (jako je systém souborů) a v žádném případě přímo uživatelskými programy. Pokud bychom ji uživatelským programům nabídli, vnesli bychom do operačního systému velké riziko chyb.

Prozatím popíšeme pouze sémantiku služby, takže se čtenář může pokusit odhalit její nebezpečí sám - po přečtení dosavadních kapitol by mělo být patrné na první pohled (ve skutečnosti k jeho odhalení stačil již dosavadní popis ovladače disku a pozornější čtenář si již nebezpečí možná povšiml dříve). Pro ty, kteří jej přece jenom přehlédnou, se k této problematice vrátíme v příštím odstavci, věnovaném dolní polovině ovladače.

Implementace služby 'dsk_write' je ještě jednodušší, než tomu bylo v případě služby 'dsk_read'; služba 'dsk_write' totiž nemusí čekat na splnění požadavku a nemusí se starat ani o uvolnění paměti, do které byl požadavek uložen. Její zdrojový kód vypadá takto:

```
/* dsk_write.c */

#include <disk.h>
#include <conf.h>

/*
 * dsk_write -- požadavek na zápis na disk
 */
int dsk_write(struct devsw *dev, char *buff, unsigned snum)
{
    struct dreq *rq;
    char x;

    sdisable(x);
    new(rq); /* vyhradíme paměť */
    rq->drdba=disk_address(snum);
    rq->drpid=currpid;
    rq->drbuff(buff);
    rq->drop=DWRITE;
    dskenq(&dstab[dev->dvmminor], rq);
    restore(x);
    return(0);
}
/* end of file */
```

Služba nejprve alokuje paměť pro nový požadavek pomocí služby 'new' a pak nastaví všechny položky v požadavku na příslušné hodnoty. Potom požadavek zařadí do fronty pomocí služby 'dskenq'. To je vše; dále se o požadavek (pokud nebyl náhodou již splněn v rámci optimalizace uvnitř služby 'dskenq') postará ovladač sám.

6.5.5.5 Dolní polovina ovladače

Dolní polovina ovladače disku není opět ničím jiným, než obslužnou rutinou přerušení, které generuje řadič disku. Jediný principiální rozdíl oproti příkladu obecného ovladače zde spočívá v tom, že disk je zároveň vstupním i výstupním zařízením; dolní polovina ovladače tedy obsahuje pouze jedinou rutinu, která zajišťuje splnění služeb obou typů.

Podobně, jako tomu bylo v příkladu obecného ovladače, může rutina znát číslo 'svého' zařízení jako konstantu DISKDEV. Zdrojový text by pak mohl vypadat následovně:

```
/* dsk_iohandler.c */

#include <disk.h>
#include <conf.h>

void interrupt dsk_iohandler()

struct dsblk *dsk=&dstab[devtab[DISKDEV].dvminor];
struct dreq *rq=dsk->dreqlst;

rq->drerror=dsk->_derror();
if ((dsk->dreqlst=rq->next)!=NULL)
    run_disk(dsk);
switch (rq->drop) {
    case DREAD:
        ready(rq->drpid,RESCHYES);
        return;
    case DWRITE:
        dispose(rq->buff);
        dispose(rq);
}
/* end of file */
```

Dolní polovina ovladače je příjemně jednoduchá. Nejprve nastaví ukazatele na blok parametrů disku a na první položku ve frontě požadavků (fronta nemůže být prázdná, protože došlo k přerušení - disk tedy ukončil zpracování nějakého požadavku a ten musí být na prvním místě fronty). Obsah položky 'drerror' v požadavku se pak nastaví v závislosti na výsledku přenosu.

Potom rutina odstraní požadavek z fronty, a je-li ve frontě další požadavek, spustí odpovídající diskovou operaci pomocí služby 'run_disk'.

Nakonec se rutina věnuje korektnímu ukončení požadavku: jestliže se jednalo o požadavek na čtení, převede proces (který dosud čeká ve stavu 'suspended') do ready fronty. Pokud se jednalo o požadavek na zápis, na jehož splnění proces nečeká, uvolní rutina paměť potřebnou pro buffer i pro požadavek.

Zde jsme však právě narazili na bezpečnostní hazard, o kterém se zmiňujeme v minulém odstavci. Rutina uvolňuje buffer, který původně alokoval někdojiný - totiž proces, který volal službu 'dsk_write' z horní poloviny ovladače. Proces tedy musí buffer alokovat a naplnit daty, ale po odeslání službou 'dsk_write' musí na buffer zapomenout, stejně jako by již byl uvolněn. Pokus o uvolnění bufferu i případný zápis jiných dat do bufferu by vedly k obtížím - velmi pravděpodobně k zápisu nesmyslných údajů na disk a tedy ke ztrátě dat.

Je zřejmé, že ovladač disku se může spolehnout na vyšší vrstvy systému, že budou službu 'dsk_write' používat korektně⁸³; nemůže se však ve stejném smyslu spolehnout na uživatelské programy. Pokud by tedy měly mít uživatelské programy služby pro práci s diskem k dispozici, museli bychom pro ně vytvořit speciální 'nadstavbu' služby 'dsk_write', která by alokovala vlastní buffer a data z bufferu uživatele by do něj zkopiovala.

⁸³Takovéto spoléhání na 'slušnost' vyšších vrstev systému však má jednu nevýhodu - implementace vyšších vrstev je pak závislá na implementaci vrstev nižších, což vede k potenciálnímu nebezpečí vzniku chyb uvnitř samotného operačního systému (k chybě dojde ve chvíli, kdy na zvláštnosti služby 'dsk_write' programátor vyšší vrstvy systému prostě zapomene). Z bezpečnostního hlediska by tedy bylo vhodnější, aby operační systém v tomto smyslu nespolehl ani sám na sebe.

6.6 Ovladače a bezpečnost

Při návrhu systému ovladačů je zapotřebí věnovat zvláštní pozornost také bezpečnosti celého operačního systému. Ovladače totiž musí mít přístup k zařízením, která ovládají, na té nejnižší úrovni; to automaticky znamená, že alespoň část kódu každého ovladače musí pracovat v systémovém režimu práce procesoru, kdy je Všechno dovoleno', a mohla by tedy potenciálně zapříčinit zhroucení systému.

Neexistuje samozřejmě obrana proti chybě v kódu ovladače nebo dokonce proti úmyslně destruktivně napsanému ovladači. Ovladače zařízení jsou součástí operačního systému a celý systém je pochopitelně jen tak spolehlivý, jak je spolehlivý jeho nejslabší článek.

Kromě toho je však zapotřebí věnovat pozornost také tomu, aby prostřednictvím i bezchybně napsaného ovladače neměl možnost vyvolat omylem nebo úmyslně nějakou destruktivní činnost obyčejný uživatelský program.

6.6.1 Příčiny nebezpečí

Systém ovladačů zařízení patří mezi nejrizikovější části operačního systému z hlediska bezpečnosti vůbec. Podíváme se nejprve na příčiny tohoto stavu a v dalších odstavcích si pak ukážeme několik konkrétních rizik, která musí návrhář ovladačů mít na vědomí a jimž musí čelit.

Prvním problémem, na který naráží zabezpečení ovladačů zařízení, je požadavek na **efektivitu a rychlosť**. Programátoři pak často zrychlují jednotlivé služby velmi jednoduchým způsobem - odstraní z nich všechno, co zdržuje; v první řadě zabezpečení.

Je samozřejmé, že zabezpečený systém bude o něco pomalejší než systém který se o bezpečnost vůbec nijak nestará. Kvůli zabezpečení je zapotřebí provádět množství testů, kontrolovat vstupní parametry funkcí a podobně; to pochopitelně nějakou dobu trvá.

Velmi dobrým příkladem je právě ovladač disku popsaný v minulém odstavci. Jeho služba 'dsk_write' vnáší do systému jistý bezpečnostní hazard; přístup k disku však bude jistě rychlejší, než kdyby služba používala vlastní buffer i při volání z vyšších vrstev systému (které samy používají vlastních bufferů, takže operační systém by neustále kopíroval data z jednoho místa na druhé).

S výjimkou zcela uzavřených operačních systémů⁸⁴ se však vyplatí bezpečnosti se věnovat - jinak se ušetřené milisekundy po prvním výpadku změní ve ztracené týdny a měsíce, ne-li roky práce. Mírné riziko, uzavřené kompletně uvnitř operačního systému a nepřístupné uživatelským programům, jakým je např. zmíněná služba 'dsk_write', může být přípustné, jestliže přinese výrazné zrychlení; na každé takové riziko by však mělo být v dokumentaci systému velmi výslovné upozornění.

Větším nebezpečím pro spolehlivost systému ovladačů je jejich komplikovanost. Porovnejme např. situaci při práci ovladače se situací při přístupu k operační paměti:

- K operační paměti přistupuje jediný proces, fyzickými účastníky celé operace jsou pouze procesor a paměť sama.
- Při práci se zařízením je situace daleko složitější: samo zařízení (např. řadič diskety) komunikuje s médiem (s konkrétní disketou), přitom používá paměťové buffery a komunikuje s procesorem. Fyzickými účastníky celé operace jsou tedy zařízení, médium, procesor a operační paměť.

Systém zabezpečení musí zajistit korektní vzájemnou spolupráci všech čtyř účastníků přenosu, a každého z nich navíc musí chránit před nežádoucím přístupem.

⁸⁴Tj. takové systémy, které neřídí běh uživatelských programů. Veškeré programové vybavení je pevnou součástí operačního systému a již se nemění.

6.6.2 Nebezpečné příkazy

Jedním z potenciálních nebezpečí jsou příkazy, které by mohly vést k chybě nebo přímo k destrukci zařízení. Ovladač musí provedení takových příkazů zamezit nebo je musí modifikovat takovým způsobem, aby bylo nebezpečí odstraněno.

Programátorský folklór např. uvádí, že vhodným časováním nastavování hlaviček pevného disku je možné dosáhnout rezonance, hlavičky rozkmitat a disk zničit (autor této knihy není dostatečným odborníkem na mechaniku pevného disku, aby mohl zmíněnou hypotézu potvrdit nebo vyvrátit - z hlediska dalšího textu to ostatně není důležité).

Pokud by tomu tak totiž skutečně bylo, musí ovladač pevného disku mimo jiné kontrolovat seřazení požadavků ve frontě i z hlediska této rezonance, a pokud by k ní mohlo dojít, musí pořadí požadavků přeuspěšně bez ohledu na efektivitu (přesněji řečeno s ohledy na efektivitu, ale pouze v rámci uspořádání, která rezonanci vyvolat nemohou).

6.6.3 'Callback' rutiny

Některé ovladače využívají tzv. 'callback' rutiny. To jsou funkce připravené uživatelem; adresa takové funkce je předána ovladači a ten pak funkci zavolá při případné aktivitě zařízení.

'Callback' rutiny jsou potenciálně velmi nebezpečné, protože uživatelský kód je zde vlastně prováděn jako součást ovladače zařízení. V principu je sice možné vytvořit bezpečné konvence pro používání 'callback' rutin; daleko lepší je však kompletně se 'callback' rutinám vyhnout a namísto nich pro informování uživatelského programu o aktivitě zařízení použít mechanismus zpráv nebo semaforů.

6.6.4 Přístup k paměti

Velkým bezpečnostním hazardem je přístup k operační paměti. Uvědomme si, že ovladač - který pracuje v systémovém režimu procesoru - má přístup k celé operační paměti. Uživatelské rutiny pro vstup přitom ovladači předávají adresy bufferů, do nichž má ovladač uložit načtená data.

Pokud návrhář systému ovladačů nebude mít tento faktor na paměti a nezabezpečí, aby předávané buffery jistě patřily procesu, který službu vyvolal, mohl by kterýkoli proces velmi snadno dosáhnout zhroucení systému: stačilo by vyžádat si třeba čtení sektoru z disku a jako adresu bufferů uvést adresu někde uvnitř chráněných dat operačního systému (pokud proces nezná adresy dat operačního systému, stačí úplně generovat takové adresy náhodně - dříve nebo později se jistě 'strefí').

Jestliže je počítač vybaven virtuální pamětí, musí ovladač - který samozřejmě dostane od programu logickou adresu bufferů - adresu sám přeložit na fyzickou, protože téměř všechna zařízení pracují přímo s fyzickými adresami v operační paměti. To ale samozřejmě přináší mnoho rizik, která musí programátor mít na paměti a odstranit - ovladač musí úzce spolupracovat se správcem paměti, který samozřejmě stránku nebo stránky odpovídající bufferů nesmí přidělit nikomu jinému, dokud není přenos ukončen.

6.6.5 Změna média

Dalším rizikem z hlediska bezpečnosti je možnost výměny média. Představme si, že operační systém ukládá nějaké údaje na disketu; nejprve zapíše vlastní data a potom chce ještě zaznamenat provedené změny v adresáři diskety. Mezi zápisem dat a záznamem do adresáře je však odpovídající proces na čas pozastaven (mohl se třeba objevit jiný proces s vyšší prioritou); uživatel mezitím disketu vyndá a nahradí jinou. Výsledkem bude nekonzistentní obsah obou disket a pravděpodobná ztráta dat.

Existují v zásadě dvě cesty, jak se může ovladač tomuto riziku ubránit:

Operační systémy

První z nich používá většina rozumných počítačů a pracovních stanic při práci s disketou: disketu není možné vyjmout jinak, než příkazem programu. Operační systém tedy v každém okamžiku 'ví', jestli mohlo dojít k výměně média - protože pokud sám operační systém médium nevysunul, uživatel to udělat nemohl.

- Nutnost využít některé ze služeb systému pro vyjmutí média může být někdy nepohodlná; pak je možné ji kompenzovat tím, že zařízení vyjmutí média detekuje a operačnímu systému je ohláší.

V obou případech však musí mít operační systém možnost po opětovném zasunutí média ověřit, jedná-li se o původní médium, na kterém jsou rozpracovaná nekonzistentní data, nebo ne. V ideálním případě by mělo být každé médium vybaveno vlastním sériovým číslem, které se liší pro každá dvě média, a systém jej může snadno ověřit; tak tomu je např. u paměťových jednotek počítačů PSION, ale ne u běžných disket. Pro ty je pak nutné kontrolovat, nezměnil-li se obsah diskety; je-li takový test dostatečně komplexní, je dost spolehlivý, ale bohužel také dost pomalý.

6.6.6 Výpadek systému

Vyhrocením problémů popsaných v minulém odstavci je možnost výpadku systému (např. vinou přerušení dodávky elektrické energie nebo poruchy některé části technického vybavení). Má-li být systém opravdu bezpečný, nemělo by k výpadku v žádném případě dojít ve chvíli, kdy jsou data na jakémkoli médiu nekonzistentní.

Toto je snad jediná oblast, ve které je na tom poměrně špatně jinak špičkový operační systém - totiž UNIX. Běžné implementace UNIXu totiž udržují poměrně rozsáhlé oblasti dat, která by měla být na disku, v operační paměti, a na disk je zapisují až v případě potřeby. Výpadek systému pak samozřejmě může zanechat disky ve stavu poněkud rozháraném.

Existují v podstatě tři způsoby obrany proti těmto problémům; první, přístupný každému operačnímu systému, druhý, náročný na technické vybavení a třetí,

náročný na kapacitu vnější paměti. Jen třetí mechanismus však může zajistit skutečnou bezpečnost operačního systému:

- Nejjednodušší technikou je vytvářet ovladače tak, aby data na kterémkoli zařízení byla v nekonzistentním stavu vždy co možná nejkratší dobu. Tím se snižuje pravděpodobnost chyby, protože i když k výpadku systému dojde, máme poměrně vysokou šanci, že všechna data jsou konzistentní.
- Druhý mechanismus vyžaduje zálohovaný napájecí zdroj, který v případě výpadku proudu okamžitě vyvolá přerušení, které aktivuje 'nouzovou' rutinu operačního systému. Ta okamžitě uvede všechna data do konzistentního stavu a uloží na vnější paměti kompletní stav systému, včetně času, ve kterém k výpadku došlo. Zálohovaný zdroj musí být samozřejmě schopen dodávat energii dost dlouho na provedení všech těchto akcí. Při nejbližším startu systém detekuje k čemu došlo, a obnoví původní stav všech údajů.

Tento mechanismus dokáže zcela zabránit všem problémům, které mohou být důsledkem výpadku elektrického proudu; není však nic platné pro případ chyby samotného technického vybavení počítače.

- Skutečně bezpečný systém proto automaticky vytváří a ukládá kopie svého stavu preventivně, buď v určitých časových intervalech, nebo před významnými změnami (nebo obojí). Dojde-li potom skutečně k výpadku, je možné obnovit stav systému před poslední změnou a zopakovat příkazy, které byly po této změně provedeny; to sice znamená jistou ztrátu času, zaručeně však zanedbatelnou ve srovnání s případnou ztrátou kompletních zpracovávaných dat. Poznamenejme, že tento velmi luxusní, ale i náročný zabezpečovací systém není žádnou utopií - jsou jím vybaveny např. snad všechny moderní databázové servery.

Operační systémy

7. Systém souborů

Na systém souborů můžeme pohlížet jako na styčný bod dvou na sobě do jisté míry nezávislých tendencí. Z jedné strany si vznik systému souborů vynutila potřeba ukládat na jeden disk množství 'balíků' na sobě navzájem nezávislých údajů a tedy potřeba zavést nad diskem nějakou uživatelsky pohodlnou a strojově nezávislou logickou strukturu. Ze strany druhé se na systém souborů můžeme dívat také jako na nejvyšší fázi virtualizace disku.

Obecnou tendencí dnešních systémů je posilování logické struktury systému souborů a odbourávání posledních vazeb na konkrétní fyzickou strukturu disku. Velmi často se o soubory stará samostatný server; běžně s ním však programy nekomunikují přímo, ale prostřednictvím ovladačů logických zařízení, odpovídajících souborům. To do značné míry zvyšuje flexibilitu operačního systému - programy nemusí rozlišovat přístup k souborům a přístup ke skutečným zařízením, všechny rozdíly skrytě obsluží operační systém⁸⁵.

V rámci rozsahu této knihy není bohužel možné popisovat implementaci rozsáhlého a komplikovaného systému souborů s mechanismem přístupových práv, jaký je součástí všech moderních operačních systémů. Navíc implementace systému souborů není v principu obtížná - je to jen hodně práce, ale nenarazíme při tom na zásadní technické obtíže. Musíme jen vhodně navrhnout a ošetřit systém sdílení souborů mezi procesy; vše ostatní je poměrně mechanická práce. Probereme proto systém souborů daleko stručněji než většinu ostatních oborů, jimž se v této knize věnujeme.

⁸⁵Některé operační systémy volí právě opačný přístup, na první pohled o něco komplikovanější, ale v praxi daleko výhodnější: systém souborů je natolik flexibilní, že zahrnuje i všechna zařízení a programátor tedy s kterýmkoli zařízením může pracovat jako se souborem. Z hlediska vlastních operací nad souborem nebo zařízením je to samozřejmě naprostě lhostejné; výhoda se projeví ve chvíli, kdy bychom rádi použili pro některé zařízení rozsáhlý aparát vlastnických práv a řízeného sdílení, který bývá součástí systému souborů. Dobrým příkladem takového operačního systému je právě UNIX.

7.1 Systém souborů

Pro implementaci systému souborů musíme obvykle rozdělit disk na řadu **alokačních jednotek**, které pro nás budou základním stavebním kamenem souborů. Kromě toho musíme vyhradit část disku pro **řetězení** - informace, které určují přidělení alokačních jednotek jednotlivým souborům. Konečně třetí diskovou oblastí, kterou budeme potřebovat, je tzv. **index** - ten obsahuje o každém souboru základní informace, jako je typ souboru, délka, doba vytvoření, doba poslední modifikace, přístupová práva uživatelů a podobně. V indexu by mohlo být uloženo i jméno souboru a u jednodušších systémů tomu tak skutečně bývá; daleko výhodnější však je po vzoru UNIXu jména souborů prozatím ponechat stranou (vrátíme se k nim v příštím odstavci).

Logická struktura systému souborů je pak poměrně jednoduchá. Každý soubor je jednoznačně určen položkou v indexu; ta obsahuje všechny důležité informace o souboru a také číslo první alokační jednotky, ve které jsou data souboru. Jestliže chceme mít přístup k pokračování souboru, zjistíme z informací o řetězení, ve které alokační jednotce soubor pokračuje. Tak můžeme postupovat až ke konci souboru.

Konkrétních implementací může být celá řada:

- Alokační jednotkou může být v nejjednodušším případě sektor. Na velkém disku je však v takovém případě alokačních jednotek příliš mnoho - uvědomme si, že sektor zabírá nejčastěji 512 byteů, zatímco kapacita disků se dnes pohybuje řádově v miliardách byteů. Řetězení tolika alokačních jednotek potom *zabírá* na disku 'zbytečně' mnoho místa.

Nejčastěji je proto alokační jednotka složena ze dvou nebo čtyř sektorů.

- Snad nejvíce možností implementace existuje pro řetězení alokačních jednotek. MS DOS a řada podobných operačních systémů např. používají velmi jednoduchý princip tzv. alokační tabulky. To je tabulka čísel, která má právě tolik položek, kolik je na disku alokačních jednotek. Chceme-li pak zjistit, která alokační jednotka logicky

následuje za jednotkou číslo 'n', podíváme se prostě do 'n'té položky alokační tabulky.

Bezpečnější, ale programátorskýméně pohodlná, metoda ukládá řetězicí informace přímo do alokačních jednotek. Takový disk je poměrně velmi dobře chráněn proti zničení dat - i v případě, že je poškozena část disku, lze údaje ze zbytku obnovit; zničíme-li však v MS DOSu alokační tabulku, jsou data na disku definitivně ztracena⁸⁶.

- Index bývá obvykle implementován nejjednodušším možným způsobem jako tabulka pevné velikosti, jejíž položky odpovídají jednotlivým souborům.

Na úrovni takového systému souborů není samozřejmě soubor identifikován svým jménem, ale číslem, které udává jeho pozici v indexu. Pro identifikaci souborů podle jmen slouží vyšší vrstva systému souborů, které říkáme systém adresářů.

7.2 Systém adresářů

Systém adresářů sám využívá systému souborů a umožňuje uživateli, aby k souborům přistupoval prostřednictvím logických jmen. Celé kouzlo systému adresářů spočívá v tom, že ne každý soubor musí obsahovat uživatelská data; některé soubory mohou obsahovat také seznam jmen spojených s čísly souborů. Jeden soubor (třeba ten první v indexu, který má číslo 0), má pak speciální postavení tzv. kořenového adresáře - zadá-li uživatel nějaké jméno, začne jej systém adresářů vyhledávat právě v tomto souboru.

Popsaný mechanismus nabízí uživateli daleko větší flexibilitu, než kdybychom jména souborů ukládali přímo do indexu. Bez jakýchkoli dodatečných prostředků máme k dispozici hierarchickou strukturu adresářů: jméno

⁸⁶Teoreticky by i v tomto případě bylo možné data obnovit; je to však natolik obtížné, že se v praxi obvykle vyplatí data vytvořit znova.

Operační systémy

v kořenovém adresáři může určovat další adresář, ve kterém se bude vyhledávat další jméno přesně stejným způsobem, jakým se první jméno hledalo v adresáři kořenovém. Jediný soubor může být určen řadou nejrůznějších jmen; to je velmi výhodné zvláště ve víceuživatelském prostředí. Struktura adresářů ani nemusí být přísně hierarchická - kterýkoli adresář může obsahovat u nějakého jména třeba odkaz na adresář kořenový.

7.3 Formátované soubory

Základní systém souborů nabízí programům velmi jednoduchý pohled na soubory: soubor je prostě řetězec bytů určité délky. Existuje samozřejmě řada problémů, pro které je tento přístup výhodný; daleko více úloh však bezpochyby využívá soubory formátované. Rozumný operační systém by proto měl podporovat alespoň dva nejčastěji používané formáty: textový soubor organizovaný po řádcích a databázový soubor s pevnou délkou záznamu. Kvalitní systémy by navíc měly podporovat i náročnější, ale zato daleko praktičtější databázový formát s proměnnou délkou záznamu; přidá-li se k tomu možnost automatického vytváření a údržby indexů, nemůžeme si již ani více přát.

7.4 Sdílení souborů

Důležitou součástí systému souborů je i mechanismus sdílení. Uvědomme si, že není možné nechat zcela nekoordinovaně dva procesy měnit údaje v jediném souboru. Na druhé straně, vyjma nejjednodušších systémů (mezi které patří např. XINU), si nemůžeme dovolit *zakázat* dvěma procesům současný přístup k jedinému souboru - nejmarkantnějším příkladem jsou databázové systémy, kde bývají často databáze sdíleny řadou nejrůznějších programů, které je doplňují i zpracovávají.

Obvykle operační systém umožňuje procesům zvolit způsob přístupu k souboru. Zvolí-li proces tzv. **sdílený přístup**, není mu umožněno obsah souboru měnit, zato však může k témuž souboru přistupovat libovolné množství procesů

najednou. Zbývající možností je samozřejmě **výhradní přístup**, při kterém má proces nad souborem plnou kontrolu, ovšem za tu cenu, že žádný jiný proces nemůže se souborem pracovat.

Moderní operační systémy, které umožňují práci s formátovanými soubory, mohou této vymožnosti využívat i pro 'chytréjší' metodu současného přístupu k souborům. Při ní každý proces otevírá soubor 'sdíleným' způsobem a teprve ve chvíli, kdy chce nějakou část souboru měnit, vyžádá si od systému dočasné vyhrazení této části souboru - nejčastěji se může jednat o jeden databázový záznam. Operační systém záznam 'zamkne' a umožní procesu jej měnit; přitom však jiné procesy mohou pohodlně zpracovávat ostatní části souboru (jiný proces může dokonce měnit jinou část souboru ve stejné chvíli, kdy první proces mění 'svůj' záznam).

Pro implementaci takovéhoto 'inteligentního' systému souborů je obvykle nejvhodnější vytvořit opět server, tj. samostatný proces. Ten má jako jediný přístup k souborům; všechny ostatní procesy mohou se soubory pracovat pouze prostřednictvím serveru (požadavky serveru obvykle předávají prostřednictvím systému zpráv). Server tak může poměrně velmi snadno koordinovat požadavky jednotlivých procesů; další výhodou je velmi snadná realizace 'asynchronních' akcí, kdy ve chvílích nečinnosti systému souborů může server provádět nějaké vlastní akce - statistiku, ukládání vyrovnávacích pamětí na disk a podobně.

7.5 Soubory a bezpečnost

Z bezpečnostního hlediska je v systému souborů nejpodstatnější mechanismus přístupových прав. Tento mechanismus určuje každému uživateli, jaké akce smí nebo nesmí provádět nad jednotlivými soubory a zabrání pokusům o provedení těch zakázaných. Např. systémové soubory tak mohou být pro uživatele (a samozřejmě tedy také uživatelské programy) zcela nepřístupné; díky tomu prakticky nepřipadá v úvahu, aby některý z uživatelů omylem nebo se zlým úmyslem jakkoli narušil operační systém. Podobně jsou chráněny i soubory jednotlivých uživatelů navzájem.

Operační systémy

Typický mechanismus přístupových práv UNIXu, který lze dnes považovat za standard, je poměrně jednoduchý:

- Každý soubor obsahuje identifikaci majitele a tří skupiny tří atributů. První skupina se vztahuje k majiteli souboru; druhá skupina atributů určuje práva jisté vyhrazené skupiny uživatelů a třetí atributy platí pro všechny ostatní.

Existence zvláštních práv pro vlastníka souboru a pro všechny ostatní je samozřejmostí. Na první pohled možná není tak zřejmé, proč by měla existovat ještě speciální přístupová práva pro zvolenou skupinu uživatelů; důvod je však jednoduchý: často na jediném projektu pracuje více lidí. Ti pak musí mít k souborům projektu volnější přístup než všichni ostatní uživatelé systému.

- Majitel má jedno význačné privilegium: může měnit přístupová práva i identifikaci majitele.
- Přístupová práva v každé kategorii uživatelů jsou určena trojicí 'povolení': číst, psát a provést jako program. Význam všech tří atributů je asi zřejmý; uvědomme si ale význam samostatného atributu pro čtení souboru. Znemožníme-li někomu číst obsah souboru, znamená to, že soubor nemůže prohlížet, číst nebo kopírovat; může jej však využít jinak - obsahuje-li např. soubor spustitelný program a má-li uživatel povolení tento soubor spustit, může jej volně používat. Nemůže však studovat jeho obsah a nemůže se tedy např. na základě jeho implementačních detailů pokusit proniknout do systému.

Existuje samozřejmě řada dalších mechanismů přístupových práv; přístupová práva UNIXu se však ukazují jako dostatečně silná pro všechny běžné situace a zároveň dostatečně jednoduchá pro implementaci. Uvedeme pro srovnání příklady jednoduššího a komplikovanějšího mechanismu přístupových práv:

- Síťový subsystém operačního systému 7 pro počítače Apple Macintosh obsahuje mechanismus pro zabezpečení přístupových práv uživatelů. Tento mechanismus však má řadu nepřijemných omezení - na každém

počítači jej lze aplikovat pouze na deset objektů, není možné umístit do jediné složky⁸⁷ dva soubory s různými právy a podobně.

- Příkladem extrémně komplikovaného systému přístupových práv je naopak operační systém Multics. Každý jeho soubor obsahuje libovolně dlouhý seznam uživatelů a jejich přístupových práv; můžeme tedy přiřadit v krajním případě každému uživateli specifická přístupová práva ke každému souboru. Z hlediska správce systému je takový mechanismus samozřejmě nejvýhodnější; jeho záporem však je složitá implementace, zpomalení přístupu k souborům a v neposlední řadě i poměrně značné místo na disku, které spotřebuje jen samotná informace o přístupových právech.

Zmiňme se nakonec o jednom problému, který takovéto rozdělení přístupových práv přináší. Čas od času je totiž zapotřebí, aby mohl i běžný uživatel používat programy, které pracují se systémovými soubory - často třeba chceme, aby mohl uživatel sám určovat heslo, pomocí kterého autorizuje svůj přístup k systému⁸⁸. Hesla však pravděpodobně budou uložena v souboru přístupném pouze operačnímu systému - jak to tedy vyřešit?

UNIX tento problém řeší tak, že zavádí možnost tzv. **propůjčení identity**. Uživatel s dostatečnou prioritou může vytvořit potřebný program a uložit jej do souboru který vlastní; tento soubor pak označí speciálním atributem a zpřístupní jej pouze pro spuštění i ostatním uživatelům. Operační systém při spouštění takového programu interpretuje speciální atribut a programu po dobu běhu přidělí práva vlastníka souboru s programem, a ne uživatele, který

⁸⁷'Složka' (folder) je to samé, co zná uživatel počítače třídy IBM PC pod názvem 'adresář' (directory) - totiž objekt na disku, který může obsahovat další objekty, složky nebo soubory. Pojem 'adresář' je technicky přesnější z hlediska skutečného formátu, jakým jsou data uložena na disk; pro běžného uživatele, který není ani nechce být odborníkem na počítače, je samozřejmě pojmenování 'složka' daleko vhodnější. Z uživatelského hlediska totiž složka obsahuje skutečně další složky a soubory (a ne nějaké 'adres').

⁸⁸Bezpečnější je ovšem uživatelům hesla, vytvořená nejlépe vhodným náhodným generátorem, přidělovat - zabráníme tak užívání snadno uhádnutelných hesel, k němuž jinak většina uživatelů snadnosklouzne.

jej skutečně spustil. Tento mechanismus je účinný, a při zachování určité ostražitosti ze strany systémových programátorů⁸⁹ je i bezpečný.

⁸⁹Je např. samozřejmé, že kdyby měl uživatel možnost takový soubor nejen používat, ale také měnit, mohl by velmi snadno proniknout do systému - jakýkoli kód, který by do souboru uložil, by byl proveden s dostatečnými právy např. pro skutečné poškození systému.

8. Sítě

Počítačové sítě jsou samostatným oborem, kterému by mohla jen z hlediska operačních systémů být věnována dlouhá řada knížek. Uvedeme si proto jen základní informace a ukážeme si možnost implementace nejnižších vrstev programového vybavení pro velmi jednoduchou lokální síť. I tak tato kapitola nebude zdaleka patřit mezi nejkratší.

8.1 Co je to lokální síť

Po technické stránce tvorí lokální síť skupina počítačů propojených dohromady. To se samozřejmě dá vztahnout i na globální síť WAN ('Wide Area Network'); za lokální síť proto budeme považovat pouze síť s omezenou vzdáleností mezi dvěma nejvzdálenějšími počítači v jedné síti. U lokálních sítí tato vzdálenost nepřevyší stovky až tisíce metrů⁹⁰.

Součástí počítačové sítě je samozřejmě především technické vybavení, které umožňuje vlastní propojení počítačů. Nejjednodušší síť může pracovat s obyčejným sériovým rozhraním; výhodou takové sítě je relativní jednoduchost, nevýhodou velmi nízká přenosová rychlosť (sériové rozhraní je velmi významným prvkem v sítích globálních, kterým se budeme věnovat zanedlouho). Naprostá většina dnešních lokálních sítí disponuje speciálním technickým vybavením, které průchodnost sítě zvyšuje o několik řádů.

Snad ještě důležitější než kvalita technického vybavení je pro síť kvalita programového vybavení, které uživatelům sítě nabízí potřebné služby. Vyšší vrstvy tohoto vybavení určují kvalitu a různorodost služeb sítě (snad všechny sítě zajišťují sdílení souborů, časté je sdílení dalších prostředků - tiskáren, modemů nebo faxů; kvalitnější síťové vybavení však může sloužit i pro

⁹⁰Poznamenejme, že s technickým pokrokem se tato vzdálenost samozřejmě zvětšuje; 'oficiální' definice lokální sítě se proto tomuto údaji vyhýbá a za lokální síť prohlašuje takovou počítačovou síť, která může být celá pod kontrolou jediného správce. Taková definice je nepochybně přesnější; 'prostorová' definice však dá čtenáři lepší představu lokální sítě.

Operační systémy

komunikaci mezi programy nebo třeba pro distribuované zpracování úloh). Ještě daleko významnější je však úloha nižších vrstev síťového programového vybavení - právě ty totiž určují spolehlivost a průchodnost sítě.

Návrh lokální sítě je totiž problematický v tom smyslu, že se v něm musíme vypořádat se skutečným paralelismem na mnoha úrovích: práce procesů na jednom počítači je plně paralelní vůči práci zařízení, které zajišťuje síťový přenos; nezávisle na obou pracuje zařízení zabezpečující síťový přenos na jiném počítači v síti i jeho procesy. Všechny tyto prvky musíme synchronizovat tak, aby jejich spolupráce probíhala bez chyb, a přitom aby systémy nestrávily více času vzájemnou komunikací než zpracováváním programů.

V příkladu nižších vrstev síťového rozhraní si také ukážeme, jak na první pohled nepatrna chyba může zapříčinit úplnou neprůchodnost celé počítačové sítě.

8.1.1 Technické vybavení lokální sítě

Technické propojení počítačů může být realizováno řadou možných způsobů:

- Propojení počítačů v síti prostřednictvím sériového rozhraní je poměrně velmi levné a jednoduché. Jeho nevýhodou však je relativně nízká rychlosť přenosu dat. Počítače třídy IBM PC proto prakticky do sítě pomocí sériových rozhraní propojit nelze; u počítačů Apple Macintosh, jejichž sériové rozhraní je daleko rychlejší, se taková síť běžně používá. Ani zde ovšem sériové rozhraní nestačí pro ty případy, kdy se po síti přenáší velké množství dat.

Jako zajímavost dodejme, že lokální síť využívající sériového rozhraní mají i počítače PSION; jejich sériové rozhraní se však přenosovou rychlostí řádu Mb/s řadí spíše mezi běžná síťová rozhraní. Bohužel, jako síťový server slouží obvykle počítač se sériovým rozhraním RS-232 nebo RS-422; počítač PSION tedy musí být vybaven převodníkem ze svého rychlého rozhraní na tento pomalý standard a komunikace se zpomaluje až k neúnosnosti.

- Propojení počítačů prostřednictvím speciálního síťového rozhraní má právě opačné vlastnosti. Je poměrně nákladné, protože rychlé síťové rozhraní je samozřejmě dražší než obyčejné rozhraní sériové, a totéž obvykle platí i o propojovacích kabelech; zato však umožňuje přenos velkého množství dat. Speciální síťová rozhraní bývají velice rychlá (s přenosovou rychlostí zpravidla přes milión bitů za sekundu). Dnes jsou prakticky rozšířeny tři typy síťových rozhraní:
 - Rozhraní Ethernet má a pravděpodobně i nadále bude mít vedoucí postavení. Bylo vyvinuto firmou XEROX v roce 1975 a dnes je podporováno tzv. konsorciem DIX (Digital Equipment, Intel a Xerox). Typická přenosová rychlosť pro síť Ethernet je 10 Mb/s. Architektura sítě je taková, že průchodnost sítě je mimořádně vysoká při nízkém zatížení, ale se zvětšujícím se zatížením se výrazně snižuje. Rozhraní Ethernet je tedy ideální pro sítě, které budou v průměru relativně málo vytíženy - což je případ naprosté většiny běžných lokálních sítí.
 - Rozhraní Token Ring firmy IBM používá jinou architekturu, která zajišťuje poměrně stálou průchodnost sítě bez ohledu na její vytížení. Přenosová rychlosť sítě Token Ring je však pouze 4 Mb/s (objevuje se modernější varianta sítě Token Ring s přenosovou rychlosťí 16 Mb/s; ta však je samozřejmě ještě daleko dražší než rozhodně nelevná 'normální' síť Token Ring).
 - Pravděpodobně nejméně rozšířená je síť Arcnet s architekturou podobnou síti IBM Token Ring a s přenosovou rychlosťí 2.5 Mb/s.

- Ve výjimečných případech mohou být počítače v síti propojeny prostřednictvím nějakého jiného zařízení, než je sériové nebo síťové rozhraní. Jedná se buďto spíše o kuriozity - v této souvislosti můžeme jmenovat počítače ATARI ST, pro které existuje lokální síť využívající jejich rozhraní MIDI⁹¹ - nebo o moderní zařízení, která se pravděpodobně teprve budou prosazovat. Do této druhé skupiny můžeme zařadit bezdrátové sítě, kdy je každý počítač vybaven vysílačem a přijímačem; takové sítě jsou nesmírně praktické především

⁹¹Rozhraní MIDI bylo vyvinuto speciálně pro komunikaci počítačů se zařízeními pro elektronickou hudbu (syntezátory apod.).

u přenosných počítačů, které díky nim nejsou vázány na nějaké kabely nebo zásuvky (podobnou síť vyvíjí například firma PSION pro své technologické počítače HC - řízené samozřejmě operačním systémem EPOC - ve spolupráci s firmou Motorola).

Z technického hlediska je u lokální sítě důležité nejen síťové rozhraní, ale také vlastní propojovací médium. Existují zde desítky a desítky možností, od velmi jednoduché a levné kroucené dvojlinky (která se dnes prosazuje zvláště u levnějších sítí Ethernet) až po velmi drahé, ale zato naprosto spolehlivé optické kabely. Sítě propojené prostřednictvím sériového rozhraní používají samozřejmě běžné sériové kabely.

8.1.2 Programové vybavení lokální sítě

Programovým vybavením, které musí být součástí operačního systému, jenž chce nosit hrdé přízvisko 'síťový', se budeme zabývat podrobněji ve zbytku této kapitoly (vyjma odstavce o globálních sítích WAN). Na tomto místě proto pouze stručně uvedeme obecnou strukturu síťového programového vybavení a vysvětlíme si jeho činnost.

8.1.2.1 Opět vrstvená struktura

Podobně, jako sám operační systém nebo jako grafický subsystém, je ovladač sítě rozložen do několika relativně samostatných vrstev. Důvod je stejný, jako tomu bylo v případě celého operačního systému - síťový ovladač je příliš komplikovaný, než aby bylo možné jej naprogramovat a odladit jako jeden nedílný programový celek. Jednotlivé vrstvy však mohou být vytvořeny postupně a díky tomu, že nižší vrstvy nikdy nevyužívají služeb vrstev vyšších, mohou být postupně také odladěny.

Rozdelení síťového programového vybavení do vrstev samozřejmě závisí na programátorovi a návrháři systému; existují však vztáta rozdělení, která velmi dobře vyhovují všem požadavkům na lokální síť. Je samozřejmě velmi dobrým cvičením pokusit se navrhnout vlastní strukturu a určit její výhody a nevýhody vůči jiným možnostem; pokud však nebudeš při návrhu vedeni opravdu velmi

šťastnou můzou (nebo velmi speciálními požadavky na funkci sítě), bude náš definitivní a nejlepší návrh nejspíše velmi podobný rozložení podle ISO:

- Nejnáze je **fyzická vrstva** - to je technické vybavení a jeho mechanické i elektrické parametry; tedy to, čím jsme se zabývali v minulém odstavci. Všechny ostatní vrstvy již budou realizovány programovým vybavením.
- **Linková vrstva** se stará o protokol předávání jednoho datového balíku mezi dvěma stanicemi lokální počítačové sítě, které jsou přímo propojeny pomocí technického vybavení.
- Navázání a opětné zrušení logického propojení mezi dvěma stanicemi je úkolem další, tzv. **síťové vrstvy**. Logické propojení přidává k linkové vrstvě dvě nové služby: zabezpečení spolehlivosti v případě výpadku technického vybavení a rozlišení 'vlastních' a 'cizích' dat. Síťová vrstva již tedy při příjmu pozná bloky, které patří jinému počítači (a musí být odeslány dále) a bloky vlastní (které musí předat vyšším vrstvám).
- **Relační vrstva** leží nad síťovou vrstvou a zajišťuje správné adresování datových balíků. Relační vrstva tedy musí znát topografii sítě a musí vědět, komu vlastně je zapotřebí data odeslat. V případě složitějších sítí složených z několika samostatných částí je opět relační vrstva garantem správného předávání dat ve spojovacích uzlech.

Zároveň na relační vrstvě spočívá zabezpečení přenosu při 'konceptních' chybách v síti. Nejedná se o náhodné chyby na Unce (ty odstraní již síťová vrstva); hlavním problémem je zde to, že kterákoli stanice může být odpojena od sítě - ať již v důsledku požadavku jejího uživatele nebo v důsledku výpadku stanice - aniž by to omezilo provoz ostatních stanic v síti.

- Nad relační vrstvou leží **transportní vrstva**, která umožňuje předávání zpráv mezi kterýmkoli dvěma stanicemi lokální sítě. Transportní vrstva se tedy již nestará o konkrétní topografii sítě - zajišťuje prostě spojení jedné stanice s druhou; o předání dat po správné cestě (tedy o tzv. routing) se stará vrstva relační.

Operační systémy

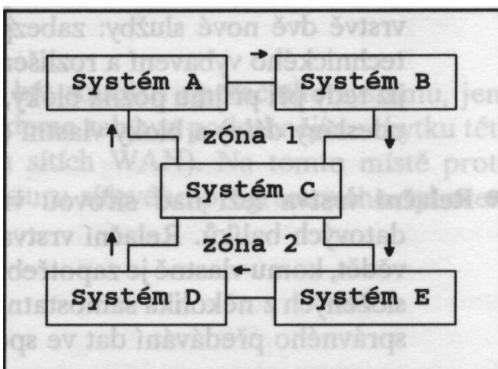
Hlavním úkolem transportní vrstvy je rozložit zprávy do datových balíků vhodných pro síťovou komunikaci (říkáme jim pakety) a předat je společně se správnou cílovou adresou nižším vrstvám.

- Prezentační vrstva již je k dispozici přímo aplikacím a systém převodu formátů a kódů dat při přenosu po síti.
- Poslední a nejvyšší vrstvou je aplikační vrstva, která definuje nástroje uživatele pro práci s lokální sítí.

8.1.2.2 Cesta dat sítí

Činnost jednotlivých vrstev si ukážeme na příkladu komunikace dvou procesů prostřednictvím sítě složené ze dvou samostatných částí - takovým částem někdy říkáme 'zóny'.

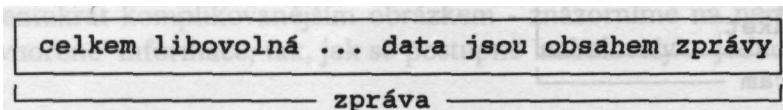
Topologii sítě vidíme na obr. 17. V zóně 1 jsou umístěny počítače A, B a C; v zóně 2 jsou počítače C, D a E. Počítač C je v obou zónách a zajišťuje předávání dat mezi nimi. Směr toku dat v síti ukazují šipky.



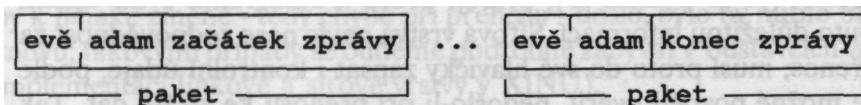
obr. 17: síť se dvěma zónami

Předpokládejme nyní, že proces 'adam' na počítači A chce poslat nějakou zprávu procesu 'eva' na počítači E (jména procesů volíme samozřejmě zcela náhodně - takto se nám však bude dobře pamatovat, kdo je odkud). Proces 'adam' tedy využije nejvyšších dvou vrstev (aplikáční a prezentační) k formulaci a odeslání zprávy. Činnost těchto dvou vrstev není nijak zvlášť zajímavá; obsahují pouze služby pro pohodlnější práci v síti⁹². Budeme se proto podrobněji zabývat až transportní vrstvou; zprávu prozatím můžeme znázornit třeba takto:

⁹²Můžeme zde také nalézt třeba služby pro kódování dat pro zvýšení bezpečnosti a řadu dalších, které z našeho hlediska nejsou příliš zajímavé.



Transportní vrstva počítače A dostane zprávu ve znázorněné formě a navíc se dozví "poslat 'evě'". Transportní vrstva především zprávu 'roztrhá' na úseky, vhodné pro transport po síti. Před každý z úseků navíc přidá hlavičku obsahující informace podstatné právě pro transportní vrstvu - může to být např. identifikace vysílajícího a přijímajícího procesu. Datovému úseku i s hlavičkou říkáme paket: zpráva rozdělená na pakety tedy vypadá nějak takto:

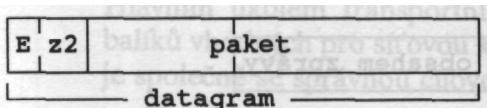


Transportní vrstva pak ještě musí zjistit, na kterém počítači vlastně 'eva' běží. Způsob, jak to zjistit, závisí na konkrétní realizaci sítě a síťových služeb - transportní vrstva může tuto informaci získat ze systémových tabulek nebo třeba od vyšších vrstev programového vybavení. Cílový počítač může specifikovat také volající proces explicitně ('poslat zprávu evě na E v zóně 2'). Transportní vrstva pak pakety i s cílovou adresou jeden po druhém předá relační vrstvě.

Z hlediska **relační vrstvy počítače A** se již budeme zabývat jen jednotlivými pakety - relační vrstva neví a nepotřebuje vědět, že pakety dohromady tvořily (a u příjemce zase budou tvořit) nějakou zprávu.

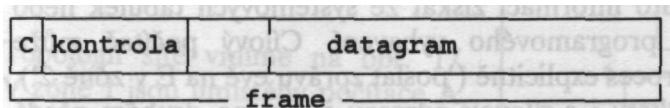
Rutiny relační vrstvy tedy dostanou paket a spolu s ním také informaci: "poslat na počítač E v zóně 2". Relační vrstva k paketu připojí svou vlastní hlavičku obsahující informace, které jsou pro ni důležité - jmenovitě tedy adresu cílového počítače. Z paketu tak vytvoří tzv. datagram - ten vypadá takto (uvnitř paketu je neúplnými čarami znázorněno vnitřní dělení na data a hlavičku transportní vrstvy):

Operační systémy



Úkolem relační vrstvy je postarat se o routing. Relační vrstva proto prozkoumá topografii sítě a zjistí, že zóna 2 není přímo přístupná; je však dosažitelná prostřednictvím počítače C. Předá tedy datagram síťové vrstvě spolu s požadavkem "poslat počítači C".

Činnost síťové vrstvy počítače A si již asi čtenář dokáže snadno představit. Společně s datagramem dostane síťová vrstva i adresu počítače, jemuž má datagram odeslat (je to počítač C); připojí proto k datagramu svou vlastní hlavičku s touto důležitou informací. Síťová vrstva navíc nese zodpovědnost za bezchybný přenos; musí proto do své hlavičky zapsat i kontrolní údaje, podle kterých bude možné později ověřit, nedošlo-li při přenosu ke ztrátě dat. Tak vznikne tzv. frame, který si můžeme znázornit takto (uvnitř datagramu je neúplnými čarami znázorněno vnitřní dělení na data a hlavičku relační vrstvy):

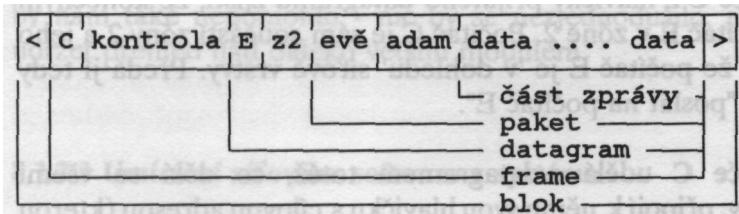


Síťová vrstva žádnou další adresu vyhledávat nemusí, protože cílová adresa, na kterou je zapotřebí frame odeslat, je zřejmá - nejbližší další počítač, se kterým je počítač A přímo propojen pomocí technického vybavení sítě: počítač B. Předá tedy 'bez komentáře' frame linkové vrstvě.

Linková vrstva počítače A frame odešle po síti počítači B. Aby mohl příjemce správně detekovat začátek a konec přenášených dat, musí linková vrstva před frame i za frame doplnit speciální kódy pro začátek a konec přenášeného bloku - označíme je '<' a '>'. Linková vrstva by také mohla frame nějak upravit; mohla by jeho obsah např. komprimovat pro urychlení přenosu⁹³. My budeme předpokládat, že frame zůstal beze změny a ukážeme si jeho formát

⁹³Uvidíme, že např. linková vrstva XINU mění obsah frame tak, aby v žádném případě nemohl kolidovat se speciálními znaky pro ukončení a zahájení bloku.

tentokrát komplikovanějším obrázkem - znázorníme na něm totiž i všechny 'vnořené' informace, tak, jak se postupně 'nabalovaly' v jednotlivých vrstvách:



Síťový hardware tedy zobrazený blok přenese beze změny na počítač B (pokud by k nějaké změně - tedy chybě při přenosu - došlo, bylo by nutno blok odeslat znova; takovými případy se budeme podrobně zabývat v odstavcích věnovaných implementaci linkové a síťové vrstvy v XINU).

Linková vrstva počítače B blok přijme a odstraní z něj úvodní a ukončovací kód. Výsledný frame - přesně stejný jako frame vytvořený na počítači A - předá síťové vrstvě.

Síťová vrstva počítače B nejprve zkонтroluje, odpovídají-li údaje uvnitř datagramu (který je z hlediska síťové služby datovou částí frame) kontrolním informacím uloženým v poli 'kontrola'. Pokud by tomu tak nebylo, vyžádala by si síťová vrstva od počítače A nové odeslání bloku; předpokládejme však, že frame byl přijat bez chyb.

Pak se síťová vrstva 'podívá' na adresu, kam má frame dojít, a zjistí, že se jedná o počítač C. Protože to je 'cizí' adresa, odešle síťová vrstva počítače B frame bez jakékoli změny ihned dále. To znamená, že jej předá linkové vrstvě.

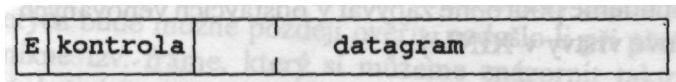
Linková vrstva počítače B frame opět 'obalí' kódy '<' a '>' a blok, který přesně odpovídá poslednímu obrázku, odešle počítači C prostřednictvím síťového hardware. Blok přijme **linková vrstva počítače C**; odstraní z něj úvodní a ukončovací kód a výsledný frame - dosud přesně stejný jako frame vytvořený na počítači A - předá síťové vrstvě.

Síťová vrstva počítače C samozřejmě nejprve ověří, odpovídají-li přijatá data kontrolním informacím, a pak se 'podívá' na adresu. Tentokrát adresu 'pozná'

jako svou vlastní; 'vybalí' proto z frame datagram - přesně stejný, jako když byl vytvořen na počítači A - a předá jej relační vrstvě.

Relační vrstva počítače C z hlavičky přijatého datagramu zjistí, že konečným příjemcem má být počítač E v zóně 2. Počítač C je sám součástí zóny 2 a jeho relační vrstva tedy ví, že počítač E je 'v dohledu' síťové vrstvy. Předá jí tedy datagram s informací "poslat na počítač E".

Síťová vrstva počítače C udělá s datagramem totéž, co dělá se všemi odesílanými datagramy: připojí k němu svou hlavičku s cílovou adresou (kterou dostala společně s datagramem) a s kontrolními informacemi. Výsledný frame vypadá takto:



Tento frame pak **linková vrstva počítače C** odešle jako blok prostřednictvím sítě **linkové vrstvě počítače E**; ta frame předá **síťové vrstvě počítače E**, která pozná v hlavičce frame vlastní adresu a proto z něj vybalí datagram a předá jej relační vrstvě.

Relační vrstva počítače E z hlavičky přijatého datagramu zjistí, že datagram se konečně dostal ke svému **cíli**. Vybere z něj proto paket a ten předá transportní vrstvě.

Transportní vrstva počítače E přijaté pakety skladuje tak dlouho, dokud z nich nevytvoří kompletní zprávu - přesně stejnou, jakou dostala transportní vrstva na počítači A k odeslání. Zprávu potom transportní vrstva předá procesu 'eva'⁹⁴, jehož jméno zjistila z hlaviček paketů; zároveň mu může sdělit i informaci, že zpráva byla odeslána procesem 'adam'.

Vypadá to složitě a také to složité je. Jedná se však asi o jediný způsob, jak zajistit transparentní a spolehlivou komunikaci mezi počítači prostřednictvím lokální sítě. Žádnou z vrstev nemůžeme vynechat, protože bez jejích služeb by

⁹⁴Prostřednictvím služeb dvou nejvyšších vrstev sítě, které mohou zprávu podle potřeby překódovat.

přenos nefungoval; snad jen ve velmi triviálním případě sítě, která nikdy nebude mít více zón a která nebude rezistentní vůči odstranění některé ze stanic, bychom se mohli obejít bez relační vrstvy. Spojení více vrstev do jedné by nám také nepomohlo - nic by se nezjednodušilo, pouze bychom museli udržet přehled nad daleko větším modulem.

8.1.2.3 Umístění síťového software v systému

Poměrně zásadní otázkou je také umístění síťového programového vybavení v rámci operačního systému. Připomeňme si strukturu operačního systému ilustrovanou na obr. 1 na straně 29 - zde není síť vůbec uvedena (protože v PC-XINU nebyla implementována); máme však možnost ji umístit téměř mezi kterékoli dvě vrstvy. Obecně platí, že čím výše síťový software umístíme, tím bude návrh systému jednodušší a ladění systému snazší (protože poměrně velmi komplikovaný síťový software budeme zařazovat již do solidně odladěného a stabilního jádra). Naopak ale služby sítě mohou být tím rozsáhlejší, čím bude síťový ovladač v operačním systému níž.

Pokud bychom např. uložili síťový ovladač jako vůbec nejnižší vrstvu operačního systému, budeme moci procesům nabídnout i tak luxusní služby, jako je sdílení operační paměti mezi jednotlivými počítači v síti (protože služby síťového systému budou k dispozici správci paměti). To by bylo velmi příjemné např. v kombinaci s mechanismem sdílených knihoven (se kterým se seznámíme v odstavci 9.2.4).

Uložíme-li síťový software pod správce procesů (tak tomu bývá v moderních síťových operačních systémech), není žádný problém v implementaci distribuovaného zpracování procesů, zcela transparentního pro programátora i pro uživatele.

Asi nejvyšší místo, kam má smysl síťové programové vybavení ukládat, je těsně pod správu souborů - pak je možné prostřednictvím sítě alespoň sdílet soubory (což je nejčastější případ využití lokálních sítí). Ještě vyšší umístění by dovolilo využití sítě jen speciálním programům - mohli bychom tedy po síti např. soubory kopírovat, transparentní přístup z kterékoli aplikace by ale nebyl možný (tak tomu bylo před mnoha lety v původním UNIXu).

8.2 Globální sítě

Globální sítě WAN jsou vlastně všechny počítačové sítě, které neodpovídají definici lokální sítě z odstavce 8.1. Jedná se obvykle o velmi rozsáhlé sítě, které obepínají celý svět a umožňují bezproblémovou komunikaci např. českému studentovi s jeho kolegou na univerzitě M.I.T.

Zatímco topologie lokálních sítí bývá poměrně jednoduchá - jak by také nebyla, když součástí lokální sítě bývá jen málokdy více než pár desítek počítačů - mají globální sítě obvykle poměrně komplikovanou strukturu. Navíc je obvyklé, že jednotlivé globální sítě jsou propojeny; uživatelé jedné z nich tedy mohou (obvykle za cenu trochu menšího pohodlí) komunikovat s uživateli řady sítí dalších.

Globální sítě se asi nejvíce využívají pro elektronickou poštu - možnost konzultovat své problémy s řadou odborníků po celém světě (nebo si jen elektronicky 'pokecat') je velmi příjemná, a ten, kdo ji měl možnost poznat, by se jí jen velmi nerad vzdával. Další velmi častou službou globálních sítí je přenos souborů. Některé globální sítě umožňují v omezené míře i distribuované zpracování problémů; nejčastěji pouze v nejjednodušší možné formě, kdy můžeme prostřednictvím sítě na vzdáleném počítači spustit vlastní program a po jeho skončení si přebrat výsledky.

Technickým vybavením užívaným pro propojování počítačů v globálních sítích je obyčejná telefonní linka. Uzly profesionálně vedených sítí (Internet, BITNET, USENET, ...) bývají propojeny pevnou linkou, která zajišťuje velmi dobrou odezvu. Amatérské sítě, mezi které patří například u nás poměrně rozšířená síť FIDO, naproti tomu bývají obvykle 'propojeny' pomocí běžné komutované linky; uzel takové sítě - jímž bývá osobní počítač vybavený modemem - pak jen musí 'čas od času' (obvykle jednou za den) zavolat svým nejbližším partnerům. Odezva sítě je tak velmi pomalá - zprávy se po ní šíří rychlostí 'jedna vrstva za den'; na jiné využití než šíření zpráv prakticky nelze ani pomyslet (přenos obecných souborů je obvykle možný, v praxi však bývá použitelný jen mezi sousedními počítači). Všechny nevýhody takovýchto sítí jsou však více než zaplaceny relativně velmi levným provozem.

Programové vybavení globální sítě obvykle není tak náročné na přesnou synchronizaci jako programové vybavení sítě lokální díky typicky nižším přenosovým rychlostem. Navíc - opět na rozdíl od sítě lokální - nebývá většinou komunikace s globální sítí transparentní vůči ostatní práci počítače: dnes považujeme za samozřejmé, že se soubory všech počítačů na lokální síti můžeme pracovat stejně snadno, jako se soubory na lokálním disku. Nikdo se však nepozastaví nad tím, že soubory z globální sítě si musíme nejprve speciálním programem vyžádat, uložit na vlastní disk a teprve potom s nimi můžeme pracovat.

Programové vybavení potřebné pro využívání globálních sítí je do jisté míry problematické začlenit do operačního systému, protože nemusí být předem jasné, do které globální sítě se bude chtít uživatel připojit. V profesionální oblasti je naštěstí bezkonkurenčním standardem Internet s protokolem TCP/IP; operační systémy určené pro profesionální využití (typicky operační systémy pro sálové počítače a minipočítače) ji proto obvykle podporují. Operační systém NeXTStep dokonce doplňuje standardní mechanismus elektronické pošty v Internetu o možnost posílání plně formátovaných zpráv obsahujících libovolné doplňky - obrazová data, samplované zvuky nebo třeba obecné soubory. Operační systémy vhodné spíše na hraní naproti tomu globální sítě a elektronickou poštu nepodporují vůbec nebo přicházejí s vlastním systémem, který nelze do stávajících sítí začlenit buď vůbec, nebo jen velmi obtížně (jako např. Windows NT).

8.3 Implementace lokální sítě

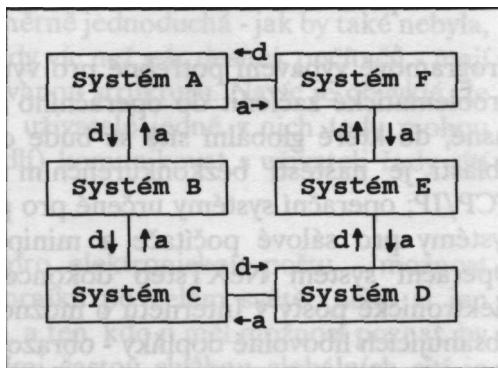
V následujících dvou odstavcích se podíváme na návrh lokální sítě, která je součástí operačního systému XINU. Jedná se o velmi jednoduchou síť, která pro komunikaci využívá běžné sériové linky a obsahuje řadu zjednodušení i na úrovni programového vybavení; to však není nikterak na úkor obecné platnosti principů, na kterých je síť postavena. Moderní systémy lokálních sítí, které jako technické vybavení využívají Ethernet nebo Token Ring, nabízejí samozřejmě řadu dalších služeb včetně automatického kódování paketů a propojují desítky nebo stovky počítačů v relativně komplikované hierarchii. Přesto se od sítě, kterou budeme popisovat, neliší v principu, ale pouze ve složitosti.

Operační systémy

Vzhledem k omezenému rozsahu této knihy se budeme zabývat pouze dvěma nejnižšími vrstvami sítě - totíž ovladačem linky (linkovou vrstvou, viz přehled vrstev v odstavci 8.1.2), a síťovou vrstvou, která se stará o spolehlivost spojení. Tyto dvě vrstvy však patří zároveň mezi nejdůležitější - právě na kvalitě jejich návrhu a implementace závisí především efektivita a spolehlivost celé sítě. Vyšší vrstvy pak již není takový problém vytvořit; je na nich samozřejmě hodně práce, nečekají nás však při ní 'podrazy', kterým se musíme vyhýbat při synchronizaci na úrovni obou nejnižších vrstev.

Počítač s operačním systémem XINU, který se má stát součástí sítě, musí být vybaven dvěma sériovými linkami⁹⁵. Prostřednictvím každé z nich je propojen s nejbližším dalším uzlem; 'poslední' uzel je propojen opět s prvním - topologie celé sítě je tedy kruhová, jak vidíme na obr. 18. Všechny sériové linky jsou samozřejmě obousměrné; nejsou však používány oběma směry pro tok dat. Data jsou v síti předávána pouze jedním směrem (který naznačují šipky s označením potvrzení o korektním přijetí jednotlivých paketů (tomu odpovídají šipky s označením 'a')). Tato potvrzení, generovaná síťovou vrstvou, jsou velmi důležitá pro spolehlivost sítě; dopodrobna se s nimi seznámíme v odstavci 8.3.2.

Dříve, než se budeme věnovat vlastní implementaci obou nejnižších vrstev síťového programového vybavení, je zapotřebí zmínit se o několika základních vlastnostech návrhu:



obr. 18: topologie sítě XINU

⁹⁵Počítač, který má linek více, může sloužit pro vzájemné propojení dvou sítí do většího funkčního celku (nebo, chcete-li, pro propojení dvou samostatných částí jedné větší sítě). V dalších odstavcích si ukážeme i případ takového využití a jeho obsluhy.

- Pro alokaci paměti potřebné pro odesílaná a přijatá data budeme používat služby 'new' a 'dispose' se stejnou sémantikou, jako tomu bylo u ovladače disku - 'new' tedy vždy paměť alokuje; není-li paměti dostatek, je volající proces pozastaven, dokud paměť nebude k dispozici.

Tentokrát je však nutné implementovat obě služby tak, aby alokovaly paměť ze samostatného bloku, nepřistupněho běžným službám alokace paměti⁹⁶. Síťový přenos je totiž často rychlejší než zpracování přijatých dat; pokud by příjemce nebyl omezen, mohl by alokovat paměť pro další a další přijaté bloky tak dlouho, až by vyčerpal všechnu volnou paměť v systému a došlo by k deadlocku.

- Buffer, který se používá pro předávání dat, je vždy bufferem pro frame. Vyšší vrstvy tedy v tomto bufferu vyplňují pouze část, hlavičky nechávají volné pro pozdější doplnění nižšími vrstvami.

Důvod je jednoduchý - kdyby měla každá vrstva kopírovat data do svého vlastního bufferu, byl by ovladač sítě příliš neefektivní. Takto prochází jediný buffer všemi vrstvami; ty si předávají pouze ukazatel na jeho začátek.

- Linková vrstva sítě není ničím jiným, než ovladačem zařízení; rozhraní mezi ní a síťovou vrstvou je tedy standardním rozhraním ovladačů zařízení. Rozhraní mezi ostatními vrstvami však musíme teprve definovat.

Velmi dobře se pro komunikaci mezi jednotlivými vrstvami síťového software hodí aparát zpráv; proto jej také použijeme - síťová vrstva bude využívat zpráv pro komunikaci s relační vrstvou (jejíž implementaci již z prostorových důvodů neuvedeme - jak jsme se zmínili na začátku kapitoly, neměl by pro zájemce být problém ji vytvořit).

⁹⁶Není to samozřejmě zapotřebí, máme-li k dispozici virtuální paměť.

Operační systémy

Systém zpráv, popsaný v odstavci 5.4.3, je však příliš jednoduchý a jeho nedostatky (např. nemožnost odeslat více zpráv jedinému procesu, který by si je pak mohl postupně odebírat) jej pro využití v tomto případě diskvalifikují. Použijeme proto luxusnější systém zpráv, aniž bychom explicitně uváděli jeho implementaci - pozorný čtenář by se jistě toho úkolu dokázal zhodit sám.

V novém systému zpráv nebude příjemcem zprávy proces, ale nový objekt; budeme mu říkat **port**. Port je identifikován svým číslem; každý proces, který zná číslo portu, na něj může odeslat zprávu (která obsahuje adresu bufferu). Podobně každý proces, který zná číslo portu, z něj může zprávu přečíst - je-li nějaká k dispozici. Pokud žádná zpráva k dispozici není, je čtoucí proces pozastaven až do chvíle, kdy někdo zprávu na port zapíše. Pokud po nějakou dobu nikdo zprávy z portu neče, dokáže port spravovat frontu několika zpráv; je-li fronta již plná a některý proces se pokusí odeslat na port další zprávu, je pozastaven až do doby, kdy někdo nějaké zprávy z portu odebere.

Pro práci s porty budeme používat služby

```
void psend(int port, void *msg);  
void *preceive(int port);
```

První z nich odešle zprávu 'msg' na port 'port', druhá naopak z portu 'port' čte první zprávu.

- Standardní aparát zpráv popsaný v odstavci 5.4.3 použijeme také; poslouží dolní polovině ovladače linky pro předávání jednotlivých přijatých bytů horní polovině - tím si ušetříme ukládání bytů do bufferu, které v tomto případě nepotřebujeme.

Bude se nám však hodit mírně modifikovaná varianta služby 'send', kterou nazveme 'sendf'. Jediným rozdílem mezi nimi bude to, že služba 'sendf' pošle procesu zprávu bez ohledu na to, zda proces již nějakou zprávu má - případná stará zpráva tedy bude zahozena. Implementace služby 'sendf' je samozřejmě triviální; stačí vypustit test '| phasmag! = 0' ze začátku služby 'send' (její zdrojový kód najeznete na straně 123).

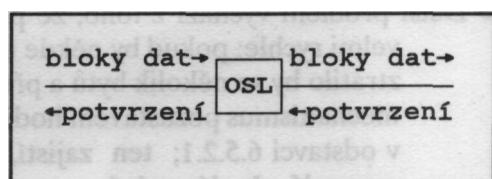
S3.1 Linková vrstva

Software pro linkovou vrstvu síťového vybavení není nicím jiným, než ovladačem sériového rozhraní. Oproti obecnému ovladači, se kterým jsme se seznámili v odstavci 6.2.1, je zde však řada rozdílů:

- Ovladač síťové linky nemusí být schopen obsloužit více procesů, protože k němu přistupují pouze rutiny vyšších vrstev síťového vybavení. Ovladač proto nevyužívá semafory pro synchronizaci procesů - jednoduše předpokládá, že zatímco obsluhuje požadavek jednoho procesu (který je na tu dobu pozastaven), jiný proces jeho služby volat nebude⁹⁷.
- Zatímco obecný ovladač předával přijatá data po jednotlivých bytech, bude ovladač síťové linky předávat po lince celé bloky. Z toho důvodu také nepotřebuje zvláštní buffer na ukládání přijatých nebo ještě neodeslaných bytů; namísto toho využívá přímo bufferu, ve kterém je uložen frame.

Zdálo by se tedy, že ovladač síťové linky bude daleko jednodušší než obecný ovladač. To však je pravda jen pro vlastní přenos; konstrukci celého ovladače komplikuje několik dalších faktorů:

- Síťový ovladač má dvě linky, jednu pro vstup a druhou pro výstup dat. Opačný směr obou linek využívá síťová vrstva pro předávání potvrzení o tom, že frame přišel a je v pořádku nebo naopak - frame nepřišel vůbec, nebo přišel a je poškozen.



obr. 19: tok dat v síti

skutečné implementaci by se vyplatilo tuto možnost velmi jednoduše ošetřit testem, zda právě probíhá komunikace - jestliže ano, služba ovladače by pouze vrátila chybovou hodnotu (nebo upozornila operátora, že některý z procesů se snaží využít ovladač síťové linky přímo, což není povolené).

Operační systémy

Tuto situaci ilustruje obr. 19. 'OSL' je ovladač síťové linky; obě linky vedou jedním směrem datové bloky a druhým potvrzení. Ačkoli potvrzení by mohla být odeslána ve formě speciálních datových bloků (a u složitějších sítí tomu tak skutečně bývá), je daleko jednodušší využít pro potvrzení jediný byte.

Z toho však vyplývá, že ovladač síťové linky musí být schopen pracovat ve dvou režimech - v blokovém a v bytovém. Po jedné lince - na obrázku ji vidíme vlevo - ovladač přijímá data v blokovém režimu a odesílá potvrzení jako byty. U druhé linky je tomu právě opačně - data jsou odesílána po blocích, ale příjem dat pracuje v bytovém režimu.

Pro blokový režim použijeme standardní služby systému ovladačů 'read' a 'write'. Pro odesílání bytů v bytovém režimu můžeme pohodlně použít službu 'putc' a pro jejich příjem službu 'getc'. Zatímco však služba 'putc' může být naprogramována poměrně jednoduše - prostě odešleme znak - se službou 'getc' je situace složitější: nemáme k dispozici ani buffer, ani vstupní semafor, musíme si proto pomocí jinak. Ideální je pro tento účel mechanismus zpráv, se kterým jsme se seznámili v odstavci 5.4.3: služba 'getc' bude čekat na příjem zprávy; tu jí pošle dolní polovina ovladače po příjetí znaku, obsahem zprávy bude právě přijatý znak.

Další problém vychází z toho, že přenos síťového bloku probíhá poměrně velmi rychle; pokud by někde uprostřed přenosu došlo k přeplánování, ztratilo by se několik bytů a přenos by byl chybný. Musíme proto využít mechanismus pozastavení hodin, se kterým jsme se podrobně seznámili v odstavci 6.5.2.1; ten zajistí, že časovač po dobu přenosu bloku nevyvolá přeplánování.

Pozastavení hodin však pro dokonalé vyřešení problému nestačí. Přičinou přeplánování totiž nemusí být pouze časovač; může k němu dojít také z řadyjiných důvodů z iniciativy aktivního procesu. Abychom přeplánování po dobu přenosu s jistotou zabránili, musíme kromě pozastavení hodin také dočasně přidělit nulovému procesu velmi vysokou prioritu, takže po celou dobu poběží právě on.

Je zřejmé, že tak o něco snížíme výkon celého systému, abychom zvýšili průchodnost sítě (tím, že většina bloků bude přenesena bez chyby a nebude nutné je odesílat znovu). Toto řešení lze považovat za výhodné v tom smyslu, že za velmi výrazné zvýšení průchodnosti sítě platíme poměrně malým snížením výkonu lokálního počítače; pokud by však v konkrétní instalaci měl mít lokální výkon před sítí absolutní prioritu, přepínání na nulový proces bychom nedělali⁹⁸.

Dodejme, že existuje jediná cesta, jak zajistit maximální průchodnost sítě a zároveň maximální výkon systému: musíme mít k dispozici speciální síťové technické vybavení, které přenáší bloky nezávisle na činnosti procesoru, takže jej přeplánování nemůže 'vyvést z míry'.

- V životě mají problémy sklon k řetězení - vyřešíme jeden a objeví se další, jako důsledek našeho řešení. Nejinak tomu je i při tvorbě programových systémů.

To, že příjemce na začátku příjmu bloku přepne na nulový proces, přece jen nějakou chvíliku trvá - je nutné nulovému procesu zvýšit prioritu a přeplánovat, ve většině případů součástí přeplánování i přepnutí kontextu. Pokud bychom pro přenos bloků používali opravdu rychlé sériové rozhraní, mohlo by mezičíme 'utéct' několik prvních bytů z bloku. Odesírající proto musí po odeslání speciálního znaku pro začátek bloku chvíli počkat, a pak teprve může začít odesílat frame.

Odesírající však je implementován jako obslužná rutina přerušení; v takovém případě není 'chvíli počkat' nic jednoduchého:

* Nemůžeme využít standardní mechanismus služby 'sleep', protože obslužná rutina přerušení může běžet v rámci nulového procesu a ten službu 'sleep' využívat nesmí.

⁹⁸Hodiny však musíme pozastavit v každém případě, pokud přenos bloku trvá déle než jeden interval sdílení času (což bude téměř jistě platit, používáme-li pro síť sériové rozhraní). Pokud bychom totiž v takovém případě hodiny nepozastavili, nebyl by korektně přenesen zádný blok.

Operační systémy

* Nepomůžeme si však ani čekací smyčkou - při běhu obslužné rutiny přerušení jsou další přerušení zakázána; došlo by tak k absurdní situaci, kdy by čekání odesílací rutiny síťového ovladače (mimo jiné) znemožnilo funkci jeho přijímací části, která je pochopitelně na přerušení závislá.

Pro vyřešení tohoto problému musíme využít technické vybavení⁹⁹. Dolní polovina ovladače odešle speciální znak pro začátek bloku a pak ukončí práci; nejprve však inicializuje technické zařízení, které zajistí, že po určitém čase bude aktivováno přerušení, které obslužnou rutinu znova aktivuje. Potom teprve obslužná rutina odešle vlastní blok.

- Posledním zásadním problémem je rozumné ošetření situace, kdy přijímací rutina sama detekuje nějakou chybu při přenosu. Nejjednodušší by samozřejmě bylo takovou chybu prostě ohlásit síťové vrstvě, která by ji obsloužila stejným způsobem, jako chybu zjištěnou při kontrole obsahu frame. Bylo by to jednoduché; zároveň však velmi neefektivní - zvláště v případě, že by se chyba objevila na samém začátku přenášeného bloku.

Přijímací rutina se proto pokusí vyžádat si od vysílajícího okamžité nové odeslání bloku. Jestliže tento požadavek vysílací rutina stačí zachytit dříve, než odešle celý blok a skončí, odešle prostě blok znovu. Jinak zůstane odstranění chyby skutečně až na síťové vrstvě; to již však nebude znamenat takové zpomalení, protože chyba se jistě objevila až ke konci bloku (jinak by požadavek na nové odeslání přišel včas).

- Spíše z technických důvodů musíme vyřešit ještě jeden detail. Přijímací musí být schopen spolehlivě rozpoznat speciální kódy pro začátek a konec bloku; jsou-li tyto kódy odesílány jako běžné byty (a při použití sériového rozhraní nemáme jinou možnost), musíme zajistit, aby se stejně kódy nemohly vyskytnout uvnitř bloku.

⁹⁹Za cenu němalých komplikací programového vybavení bychom tento problém samozřejmě mohli řešit i zavedením speciálního procesu, který by se staral o časování výstupu.

Řešení je celkem jednoduché: zavedeme navíc tzv. escape znak, který mění význam následujícího znaku. Místo bytu, který by měl stejnou hodnotu jako některý ze speciálních kódů, odešleme dvojici escape znak + změněný byte; přijímající část tuto dvojici opět přeloží zpátky na původní byte.

Návrh ovladače síťové linky je velmi citlivý na jakékoli nedomyšlenosti a chyby - i nepatrný nedostatek může být snadno příčinou významného snížení průchodnosti sítě nebo dokonce jejího úplného zablokování. Může např. dojít ke klasickému zablokování, kdy několik systémů čeká vzájemně na data a/nebo potvrzení, a ani jeden z nich se nedočká - to je síťová varianta známého deadlocku, kterým jsme se již zabývali. Jinou nebezpečnou situací je stav, kdy se systémy snaží sesynchronizovat, ale nikdy se jim to nepodaří - mohlo by např. dojít ke ztrátě potvrzení bloku; vysílající pak bude blok neustále posílat znova, zatímco příjemce jej nebude akceptovat, protože již čeká na blok následující.

Proto dříve, než uvedeme skutečné zdrojové texty, věnujeme jeden odstavec formálnímu návrhu ovladače.

83.1.1 Funkce ovladače

Pro návrh ovladače je asi nejvhodnější použít stavový diagram. Jedná se vlastně opět o modularitu, tentokrát přenesenou až dovnitř do jediné funkce: navrhovaná rutina se může nacházet v několika stavech, každý odpovídá jiné situaci; potřebné chování rutiny pak zkoumáme pro každý stav zvlášť. Tím si celý problém rozdělíme na několik problémů menších a jednodušších, a snáze se vyhneme případné chybě nebo tomu, že bychom na něco zapomněli. Další výhodou stavového diagramu je to, že se velmi snadno přenáší do skutečného programu - stačí využít statickou proměnnou obsahující identifikaci momentálního stavu, a přechody mezi jednotlivými stavami pak realizovat změnou obsahu této proměnné.

Vstupní část linkového ovladače může mít následující stavy:

Operační systémy

- Stav **IINIT** je základním stavem, ve kterém se ovladač nachází, když 'se nic neděje'. Mohou v něm být přijímány jednotlivé znaky v neblokovém režimu.
- Do stavu **IREADY**, ve kterém je ovladač připraven ke čtení bloku, se ze stavu **IINIT** dostane při volání vstupní služby 'read' vyšší vrstvou síťového programového vybavení.
- Po přijetí speciálního kódu pro začátek bloku se ovladač dostane ze stavu '**IREADY**' do stavu **IREAD**, ve kterém postupně čte jednotlivé byty bloku a ukládá je do paměti. Přijme-li ovladač znak 'escape', neukládá nic, ale přejde do stavu **IESCAPED**; ten se od stavu '**IREAD**' liší pouze tím, že přijmutý byte navíc překóduje, uloží a vrátí se do stavu '**IREAD**'.
- Příchod speciálního znaku pro ukončení bloku převede ovladač do stavu **IDONE**. Ten nemá žádný zvláštní význam; ovladač v něm pouze čeká, než jej ukončení služby 'read' opět převede do stavu **IINIT**.

Po případném přerušení přenosu bloku by musel ovladač ve stavu '**IREADY**' čekat na začátek znova posílaného bloku. Na první pohled by to 'fungovalo' automaticky (prostým čekáním na speciální znak začátku bloku); z technických důvodů spojených z odesíláním požadavku na opakování bloku je výhodnější přidat ještě jeden stav, **IWAIT**, ve kterém bude ovladač čekat namísto stavu '**IREADY**'.

Jednotlivé stavy ovladače a přípustné přechody mezi nimi jsou vyznačeny na obr. 20. Přerušované čáry označují přechody, které jsou 'v režii' horní poloviny, plnými čarami jsou vyznačeny stavové přechody zajišťované dolní polovinou ovladače. Šipky ukazují možné stavové přechody - ovladač tedy např. může na základě vstupních dat přecházet mezi stavy '**IREAD**' a '**IESCAPED**', nemůže však v žádném případě přejít ze stavu '**IREAD**' do stavu '**IREADY**' (jinak než prostřednictvím stavu '**IWAIT**'). Analyzujme nyní podrobně chování ovladače v jednotlivých stavech; obrázek nám přitom pomůže v orientaci. Je také vhodné si uvědomit, že vlastně analyzujeme chování dolní poloviny ovladače vstupní části linky - horní polovina jen zajistí čárkování přechody, výstupnímu ovladači se budeme věnovat zanedlouho. Dolní polovina ovladače je aktivována v případě, že technické vybavení přijalo po lince nějaký znak - budeme tedy

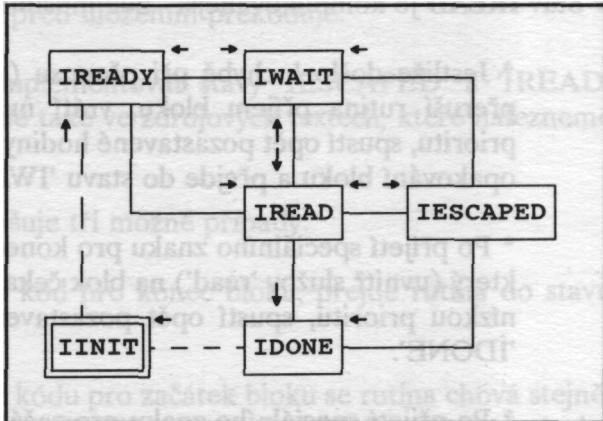
jednotlivé stavy analyzovat především z hlediska vyhodnocení přijatého znaku.

- Ve stavu **IINIT** může být ovladač aktivován jedině v případě, že přišel nějaký znak v neblokovém režimu, tj. odeslaný službou 'putc' (tento ovladač tedy datové bloky pouze vysílá, přijímá jen jednotlivé znaky). Ovladač musí rozlišit dvě možnosti:

- * Jedná-li se o požadavek linkové vrstvy druhého počítače na nové odeslání bloku (ten je identifikován speciální hodnotou znaku), předá se tento požadavek výstupnímu ovladači.
- * Jinak se jedná o vzájemnou 'potvrzovací' komunikaci síťových vrstev; znak proto předáme službě 'getc' pomocí mechanismu zpráv.

V obou případech ovladač zůstane ve stavu '**IINIT**'.

- Ve stavu **IREADY** ovladač čeká na začátek přenosu bloku. Může dojít k těmto situacím:
 - * Objeví se speciální znak pro začátek bloku. Ovladač pozastaví hodiny, zvýší prioritu nulového procesu, zajistí případné přeplánování, připraví se na příjem bloku a přejde do stavu '**IREAD**'.
 - * Objeví se jakýkoliv jiný znak. To zřejmě znamená, že vysílající 'vypadl ze synchronizace' a právě odesílá nějaký blok. Ovladač odešle požadavek na nové odeslání bloku a přejde do stavu '**IWAIT**'.



obr. 20: stavy vstupní rutiny ovladače

- Stav **IREAD** je komplikovanější - zde musíme rozlišit více různých případů:
 - * Jestliže došlo k chybě při přenosu (to ohlásí technické vybavení), přeruší rutina příjem bloku, vrátí nulovému procesu jeho nízkou prioritu, spustí opět pozastavené hodiny, odešle po lince požadavek na opakování bloku a přejde do stavu 'IWATF'.
 - * Po přijetí speciálního znaku pro konec bloku rutina aktivuje proces, který (uvnitř služby 'read') na blok čekal, vrátí nulovému procesu jeho nízkou prioritu, spustí opět pozastavené hodiny a přejde do stavu 'IDONE'.
 - * Po přijetí speciálního znaku pro začátek bloku prostě rutina zahodí dosud načtené znaky a začne znova přijímat blok od začátku. Stav se v tomto případě nemění (je možné jej samozřejmě nastavit opět na 'IREAD' pro větší shodu se stavem 'IESCAPED' - viz níže).
 - * Po přijetí speciálního znaku 'escape' rutina přejde do stavu 'IESCAPED', nic více a nic méně.
 - * Přišel-li jakýkoli jiný znak, uloží jej rutina do přijímaného bloku. Aby nedošlo k přetečení paměti vyhrazené pro blok, musí rutina zkontolovat, není-li již buffer zaplněn; pokud tomu tak je, ukončí se přenos stejně jako po přijetí znaku pro konec bloku¹⁰⁰.
- Stav **IESCAPED** se do značné míry podobá stavu 'IREAD' - v případě chyby při přenosu, přijetí speciálního znaku pro konec bloku nebo speciálního znaku pro začátek bloku se chová stejně (po zachycení začátku bloku navíc pouze změní stav na 'IREAD'). Jediný rozdíl spočívá v tom, že:
 - * Přijetí znaku 'escape' je považováno za chybu při přenosu a rutina se tedy chová stejně, jako když dojde ke skutečné chybě;

¹⁰⁰To je výhodnější, než kdyby rutina čekala na to přijde-li ještě znak pro konec bloku a v záporném případě hlásila chybu. Je-li totiž blok dobré, bylo by to zbytečné; je-li blok špatně, síťová vrstva to stejně pozná a na případné nové odeslání bloku na úrovni linkových vrstev již je stejně pozdě.

- * Jakýkoli jiný znak se před uložením překóduje.

V praxi je výhodné implementovat stav 'IESCAPED' a 'IREAD' společně; tak tomu bude také ve zdrojových textech, které nalezneme v dalších odstavcích.

- Ve stavu **IWAIT** rutina rozlišuje tři možné případy:
 - * Objeví-li se speciální kód pro konec bloku, přejde rutina do stavu 'IREADY'.
 - * Po přijetí speciálního kódu pro začátek bloku se rutina chová stejně, jako by byla ve stavu 'IREADY': pozastaví hodiny, zvýší prioritu nulového procesu, zajistí případné přeplánování, připraví se na příjem bloku a přejde do stavu 'IREAD'.
 - * Jakýkoli jiný znak se ignoruje beze změny stavu.
 - Nejjednodušší je chování rutiny v posledním stavu **IDONE**. V něm rutina jakýkoli znak ignoruje a jen čeká, až bude horní polovinou ovladače převedena do stavu 'IINIT'.
- Nyní bychom již bez větších problémů dokázali výstupní část ovladače naprogramovat. Dříve, než se do toho pustíme, se ale vyplatí provést stejně podrobnou analýzu **výstupní části** linkového ovladače; ta by mohla disponovat těmito stavami:
- Stav **OINIT** je základním stavem, ve kterém se ovladač nachází, když 'se nic neděje'. V tomto stavu rutina zajistí vysílání jednotlivých znaků v neblokovém režimu, je-li to zapotřebí.
 - Do stavu **OREADY** se ze stavu 'OINIT' ovladač dostane při volání výstupní služby 'write' vyšší vrstvou síťového programového vybavení. Ovladač zajistí odeslání speciálního znaku pro začátek bloku a přejde do stavu '**OWRITE**'.

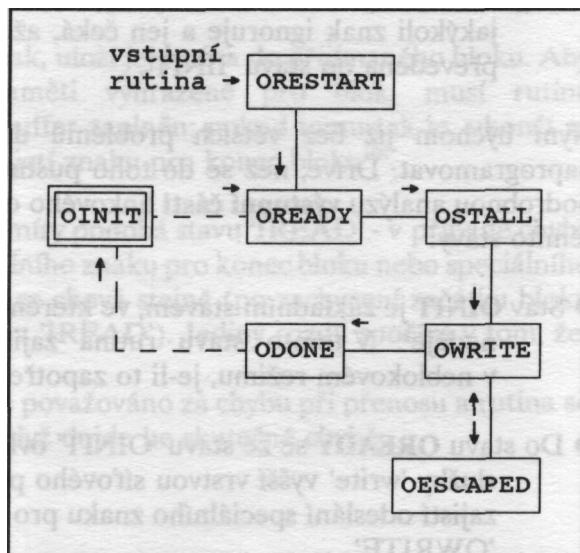
Operační systémy

- Ve stavu OWRITE ovladač odešle byte po bytu celý blok. Je-li zapotřebí odeslat byte, který odpovídá některému ze speciálních znaků, ovladač odešle nejprve znak 'escape' a pak překódovaný byte. Formálně tuto situaci vyřešíme pomocným stavem **OESCAPED**; při implementaci pak uvidíme, že se vyplatí oba tyto stavy spojit dohromady, stejně jako u vstupní rutiny. Po odeslání celého bloku ovladač odešle speciální znak pro ukončení bloku a přejde do stavu 'ODONE'.
- Ve stavu ODONE již ovladač pouze čeká, než jej ukončení služby 'write' opět převede do stavu 'OINIT'.

Z technických důvodů musíme k tomuto 'ideálnímu návrhu' přidat ještě dva další stavy. Stav **OSTALL** je vsunut mezi stavy 'OREADY' a 'OWRITE' a ve spolupráci s technickým vybavením zajišťuje pauzu mezi odesláním hlavičky bloku a vlastních dat bloku, o jejíž potřebě jsme se zmiňovali v odstavci 8.3.1. Do stavu **ORESTART** je ovladač přemístěn z iniciativy vstupní rutiny po přijetí požadavku na nové odeslání bloku; ovladač v tomto stavu 'zapomene', co již odeslal, a přejde do stavu 'OREADY'.

Jednotlivé stavy ovladače a přípustné přechody mezi nimi jsou vyznačeny na obr. 21. Přerušované čáry opět označují přechody, které jsou V režii horní poloviny, plnými čarami jsou vyznačeny stavové přechody zajišťované dolní polovinou ovladače. Šipky ukazují možné stavové přechody. Do stavu 'ORESTART' se ovladač může dostat odkudkoli, vyjma stavu 'OINIT' z iniciativy vstupní rutiny.

Analyzujme nyní podrobně chování ovladače v jednotlivých stavech. Tentokrát se



obr. 21: stavy výstupní rutiny ovladače

věnujeme analýze chování dolní poloviny ovladače výstupní části linky - horní polovina jen zajistí čárkované přechody, vstupní ovladač již známe. Dolní polovina ovladače je aktivována v případě, že technické vybavení může odeslat nějaký znak - tj. v případě, že buď byl právě ukončen přenos znaku minulého nebo po povolení přerušení.

Výstupní rutina je jednodušší než vstupní; navíc si její činnost již čtenář jistě dokáže představit. Popíšeme ji tedy daleko stručněji:

- Ve stavu **OINIT** může být ovladač aktivován ve dvou případech:
 - * Je zapotřebí odeslat znak v neblokovém režimu (tentot ovladač tedy datové bloky přijímá). Ovladač znak odešle.
 - * Technické vybavení hlásí ukončení přenosu a již není co odesílat. Ovladač deaktivuje zařízení, takže to již další přerušení nebude generovat (dokud jej někdo opět neaktivuje).

V obou případech ovladač zůstane ve stavu '**OINIT**'.

- Ve stavu **OREADY** ovladač prostě odešle speciální znak pro začátek bloku a přejde do stavu '**OSTALL**'.
- Stav **OSTALL** spolupracuje s technickým vybavením; jeho obsah tedy silně závisí na konkrétním počítači. V originálním XINU, psaném pro mikropočítač LSI 11, se ve stavu '**OSTALL**' zařízení pro linkový přenos nejprve deaktivuje a hned nato opět aktivuje. To u zmíněného systému zajistí vygenerování dalšího přerušení technickým vybavením. Ovladač ve stavu '**OSTALL**' jen počítá tato 'zbytečná' přerušení; je-li jich dostatek (trvá-li tedy pauza již dost dlouho), přejde do stavu '**OWRITE**'.¹⁰¹

¹⁰¹ Je vhodné si uvědomit, že tento způsob čekání je korektní pouze v případě, že má přerušení výstupní části linkového zařízení velmi nízkou prioritu - speciálně nižší, než přerušení vstupní části linkového zařízení. Tak tomu u mikropočítače LSI 11 skutečně bylo.

Operační systémy

- Ani ve stavu **OWRITE** nenarazíme na žádné problémy. Rutina prostě odešle další znak bloku; je-li to zapotřebí, zakóduje jej ve spolupráci se stavem '**OESCAPED**'.

V praxi je opět výhodné implementovat stavy '**OESCAPED**' a '**OWRITE**' společně; tak tomu bude také ve zdrojových textech, které nalezneme v dalších odstavcích.

- Jedině ve stavu **ODONE** je výstupní rutina nepatrně složitější než vstupní. Musí v něm totiž nejprve aktivovat proces, který čekal na odeslání bloku, a pak deaktivovat linkové zařízení. Pak již rutina opět jen čeká, až bude horní polovinou ovladače převedena do stavu '**OINIT**'.
- Poslední stav **ORESTART** pouze znova inicializuje odeslání bloku a ihned přejde do stavu '**OREADY**'. Protože stav '**ORESTART**' sám nic neodesílá, může do stavu '**OREADY**' přejít skutečně ihned, tedy ne až po přijetí dalšího přerušení, jako tomu je v ostatních případech stavových přechodů.

Nyní se konečně dáme do psaní programů:

8.3.1.2 Společné deklarace

Deklarace potřebných datových struktur opět vychází z obecného ovladače, jehož příklad jsme uvedli v odstavci 6.2.1; navíc zde jsou pouze potřebné konstanty pro jednotlivé stavy a pro speciální kódy. Stejně je i to, že datové struktury jsou uloženy v poli indexovaném položkou '**dvminor**' - ovladač samozřejmě může řídit více sítí.

Patřičné deklarace by mohly vypadat například takto:

```
/* net_link.h */  
  
enum { /* stavy vstupní rutiny */  
    IINIT, IREADY, IREAD, IESCAPED, IWAIT, IDONE  
};  
  
enum { /* stavy výstupní rutiny */
```

```

OINIT, OREADY, OSTALL, OWRITE, OESCAPED, ODONE, ORESTART
};

#define NLESC      '\xa8'    /* escape znak */
#define NLRESTART  '\xa9'    /* požadavek na nové odeslání */
#define NLSTART    '\xaa'    /* začátek bloku */
#define NLEOB      '\xab'    /* konec bloku */

#define NLSPEC     '\xa8'    /* bity určující spec. znaky */
#define NLSMASK    '\xfc'    /* maska pro -- " -- */

#define NL_DOESC   '\x7f'    /* maska pro zakódování po escape */
#define NL_UNESC   '\x80'    /* 'odkódování' po escape */

#define NSTALL     3        /* počet průchodů stavem OSTALL */

struct nblk {
    char istate;           /* stav vstupního ovladače */
    int  ictount;          /* počet načtených znaků */
    int  ipid;              /* id procesu, čekajícího na vstup */
    char *inext;            /* první volná pozice v bufferu */
    char *istart;           /* buffer */
    int  imax;               /* max. délka čteného bloku */

    char ostate;           /* stav výstupního ovladače */
    int  ocount;             /* počet odeslaných znaků */
    int  opid;                /* proces, čekající na výstup */
    char *onext;             /* první neodeslaný znak v bufferu */
    char *ostart;            /* buffer */
    int  olen;                  /* celkový počet znaků */
    int  ostall;                 /* čítač čekání ve stavu OSTALL */

    int cpid;                /* proces čekající na znak. vstup */
    char cchar;                /* znak pro znakový výstup */
    char cvalid;               /* !=0 je-li cchar validní */

    void (*run_odev)(void); /* aktivace výstupního zařízení */
    void (*stop_odev)(void);/* deaktivace výstupního zařízení */
    void (*wrt_odev)(char);/* zápis znaku na výst. zařízení */
    char (*rd_idev)(void);/* čtení znaku ze vstup, zařízení */
    int  (*err_idev)(void);/* zjištění chyby vstup, zařízení */
};

extern struct nblk nldevs[];

/* end of file */

```

Operační systémy

Obsah všech položek by měl být celkem samozřejmý: skupina položek 'i...' určuje stav vstupní rutiny ovladače, vstupní buffer, proces čekající na vstup a několik pomocných údajů. Podobná skupina položek, odpovídající výstupní části ovladače, začíná písmenem 'o'. Tři následující položky slouží pro neblokový vstup a výstup: případný přijatý znak je předán procesu 'cpid' (který na něj čeká uvnitř služby 'getc'); je-li hodnota 'cvalid' nenulová, je zapotřebí odeslat znak 'cchar'.

Funkce 'run_odev' povolí výstupnímu zařízení, aby přerušením hlásilo, že 'nemá co na práci'. Zařízení nemá k dispozici žádná data, a proto vygeneruje přerušení, které samozřejmě bude obsluženo dolní polovinou ovladače - ta pak data skutečně odešle.

Funkce 'stop_odev' naopak výstupnímu zařízení generování přerušení zakáže. Výstupní zařízení v takovém případě samozřejmě dokončí případný přenos dat, pokud nějaký provádí, ale jeho ukončení již nebude hlásit přerušením. Pak zůstane zařízení v klidu, dokud nebude opět zavolána služba 'run_odev'.

Funkce 'wrt_odev' výstupnímu zařízení předá znak k odeslání. Zařízení znak odešle (což může trvat delší dobu), a potom generuje přerušení (má-li to povoleno předchozím voláním funkce 'run_odev').

Funkce 'rd_idev' naproti tomu přenos dat prostřednictvím vstupního zařízení nevyvolá; pouze z tohoto zařízení přečte znak, který již byl zařízením přijat. Vstupní zařízení nemá nikdy zakázané přerušení; kdykoli načte nějaký znak, generuje přerušení a jeho obslužná rutina pak použije funkci 'rd_idev' ke zjištění načteného znaku. Podobně funkce 'err_idev' zjistí, došlo-li při posledním přenosu k chybě (a v takovém případě vrátí nenulovou hodnotu).

Implementace funkcí 'run_odev', 'stop_odev', 'wrt_odev', 'rd_idev' a 'err_idev' samozřejmě závisí podstatným způsobem na konkrétním zařízení; nemá proto smysl, abychom k nim uváděli zdrojové texty. Budeme prostě předpokládat, že v odpovídajících položkách již jsou umístěny ukazatele na správné funkce.

Tabulka 'nldev' pak obsahuje pro každé síťové zařízení jednu položku. Každý počítač přitom obsluhuje nejméně dvě síťová zařízení.

8.3.1.3 Horní polovina

Díky jednoduchosti rozhraní mezi síťovým ovladačem a vyššími vrstvami programového vybavení jsou rutiny horní poloviny ovladače příjemně jednoduché - není zapotřebí se starat ani o synchronizaci mezi procesy, ani o buffery.

Jedná se o služby 'nl_read' a 'nl_write' pro blokový vstup a výstup a o služby 'nl_getc' a 'nl_putc' pro znakový výstup. Připomeňme, že znakový výstup není nikdy 'promíchán' s blokovým výstupem, a totéž platí pro vstup - zařízení, které odesílá bloky, čte znaky, a (jiné) zařízení, které čte bloky, znaky odesílá.

Služba 'nl_read' by mohla být implementována takto:

```
/* nl_read.c */
#include <net_link.h>
#include <conf.h>

/*
 * nl_read -- požadavek na čtení bloku
 */
int nl_read(struct devsw *devptr,char *buff,unsigned max)

    struct nlblk *nl=&nldevs[devptr->dvminor];
    int nread;
    char x;

    sdisable(x);
    nl->istate=IREADY;
    nl->inext=nl->istart(buff);
    nl->imax=max;
    nl->icount=0;
    nl->ipid=currpid;
    suspend(currpid);
    nread=nl->icount;
    nl->istate=INIT;
    restore(x);
    return(nread);
}

/* end of file */
```

Operační systémy

Implementace služby je zcela jednoduchá: nejprve se převede ovladač do stavu 'IREADY', pak se vyplní všechny potřebné položky v tabulce 'nlblk' a nakonec proces sám sebe suspenduje. Dolní polovina síťového ovladače po přijetí bloku proces opět převede do ready fronty; ten pak zjistí kolik znaků se skutečně načetlo ('nread=nl->icount'), převede ovladač zpět do stavu 'IINIT' a vrátí řízení tomu, kdo volal službu 'read'.

Poznamenejme, že pro praktickou implementaci by bylo vhodné rutinu zabezpečit tak, že by nejprve ověřila, je-li momentální stav ovladače 'IINIT' a v záporném případě (tj. při volání služby 'read' v době, kdy jiný proces tuto službu provádí) by upozornila operátora na chybu systému. Totéž samozřejmě platí i pro všechny další služby.

Služba 'nl_write' je složitější jen o potřebnou aktivaci výstupního zařízení:

```
/* nl_write.c */

#include <net_link.h>
#include <conf.h>

/*
 * nl_write -- požadavek na odeslání bloku po síti
 */
int nl_write(struct devsw *devptr,char *buff,unsigned len)
{
    struct nlblk *nl=&nlidevs[devptr->dvmminor];
    char x;

    sdisable(x);
    nl->ostate=OREADY;
    nl->onext=nl->ostart=	buff;
    nl->olen=nl->ocount=len;
    nl->opid=currpid;
    nl->run_odev();
    suspend(currpid);
    nl->ostate=OINIT;
    restore(x);
    return(0);
}

/* end of file */
```

Funkci služby 'nl_write' je již jistě zbytečné podrobně popisovat. Ukážeme si namísto toho stejně jednoduchou implementaci služby 'nl_putc', používané pro odeslání jednoho bytu po síti:

```
/* nl_putc.c */

#include <net_link.h>
#include <conf.h>

/*
 * nl_putc -- požadavek na odeslání bytu
 */
int nl_putc(struct devsw *devptr, char c)

    struct nlblk *nl=&nldevs[devptr->dvmminor];
    char x;

    sdisable(x);
    nl->cchar=c;
    nl->cvalid=1;
    nl->run_odev();
    restore(x);
    return(0);
}

/* end of file */
```

Služba 'nl_putc' na odeslání nečeká - prostě vyplní patřičné položky, aktivuje výstupní zařízení a skončí. Dolní polovina ovladače již znak odešle sama.

Implementace služby 'nl_getc' je trochu zajímavější díky využití mechanismu zpráv:

```
/* nl_getc.c */

#include <net_link.h>
#include <conf.h>

/*
 * nl_getc -- čtení jednoho bytu ze sítě
 */
int nl_getc(struct devsw *devptr)
```

Operační systémy

```
struct nlblk *nl=&nldevs [devptr->dvmminor] ;  
char x;  
  
sdisable(x);  
recvclr();  
nl->cpid=currid;  
restore(x);  
return(0);  
}  
  
/* end of file */
```

Služba nejprve smaže případnou zprávu službou 'recvclr', pak vyplní potřebné tabulky, aby byla zpráva odeslána patřičnému procesu, a pak - skončí. Proces si na zprávu musí počkat sám voláním služby 'receive'.

Tento netradiční mechanismus byl zvolen proto, že proces musí nejprve inicializovat 'čtení potvrzení', tedy provést službu 'getc', potom odeslat blok a až nakonec čekat na potvrzení. Pořadí služeb tedy bude

```
getc(); /* zatím nic neče! */  
write();  
receive();
```

Pokud bychom se pokusili přidat službu 'receive' na konec funkce 'nl_getc' a použít pak (na první pohled logičtější) pořadí

```
write();  
getc();
```

se zlou bychom se potázali - odesílání bloků a potvrzení probíhá do jisté míry asynchronně, takže zprávu bychom mohli ve skutečnosti dostat 'mezi' ukončením služby 'write' a začátkem služby 'getc'. Ta by pak sama zprávu zahodila službou 'recvclr'.

8.3.1.4 Dolní polovina

Dolní polovinou síťového ovladače jsou samozřejmě obslužné rutiny přerušení vstupního a výstupního zařízení. Jako jednoduché rozhraní zjednodušilo implementaci horní poloviny ovladače, tak poměrně komplikovaná funkce zkomplikuje rutiny jeho dolní poloviny; díky 'stavovému' návrhu však jsou rutiny i tak velmi dobře srozumitelné.

Nejprve uvedeme implementaci o něco jednodušší obslužné rutiny výstupního zařízení. Jako vždy obsahuje symbolická konstanta NLDEV číslo odpovídajícího zařízení:

```
/* nl_ohandler.c */

#include <net link.h>
#include <conf.h>

void interrupt nl_ohandler()
{
    struct nlblk *nl=&nltab[devtab[NLDEV].dvminor];

    switch (nl->ostate) {
        case OINIT:
            if (nl->cvalid) {
                nl->cvalid=0;
                nl->wrt_odev(nl->cchar);
            } else nl->stop_odev();
            return;
        case ORESTART:
            nl->ostate=OREADY;
            nl->onext=nl->ostart;
            nl->ocount->nl->olen;
            /* fall thru */
        case OREADY:
            nl->wrt_odev(NLSTART);
            nl->ostate=OSTALL;
            nl->ostall=NSTALL;
            return;
        case OSTALL:
            nl->stop_odev();
            nl->run_odev();
            if (nl->ostall--==0)
                nl->ostate=OWRITE;
            return;
        case OWRITE:
    }
```

```
case OESCAPED: {
    char ch;

    if (nl->ocount---==0) {
        nl->wrt_odev(NLEOB);
        nl->ostate=ODONE;
        return;
    }
    if (((ch=*nl->onext++) &NLSMASK)==NLSPEC)
        if (nl->ostate==OWRITE) {
            nl->onext--;
            nl->ocount++;
            nl->ostate=OESCAPED;
            nl->wrt_odev(NLESC);
        } else {
            nl->ostate=OWRITE;
            nl->wrt_odev(ch&NL_DOESC);
        }
        else nl->wrt_odev(ch);
    }
    return;
}
case ODONE:
if (nl->ocount<0) {
    ready(nl->opid,RESCHYES);
    nl->ocount=0;
}
nl->stop_odev();

/* end of file */
```

Rutinu není zapotřebí podrobně popisovat; pokud si čtenář není zcela jist významem některé služby v některém stavu, naleze ne jistě vyčerpávající odpověď v podrobné analýze stavů výstupní částí ovladače na straně 231.

Dolní polovina vstupního ovladače je o něco komplikovanější; podrobný rozbor na straně 227 však jistě vyřeší všechny případné nejasnosti:

```
/* nl_ihandler.c */

#include <net_link.h>
#include<conf.h>

void interrupt nl_ihandler()
```

```

struct nlblk *nl=&nltab[devtab[NLDEV].dvminor];
char ch=nl->rd_idev();

switch (nl->istate) {
    case UNIT:
        if (ch==NLRESTART && nl->ostate!=OINIT)
            nl->ostate=ORESTART;
        else if (nl->cpid)
            sendf(nl->cpid,ch);
        return;
    case IREADY:
        if (ch==NLSTART) {
SetRead:
            nl->istate=IREAD;
            stopclk();
            _nullprio(0,32767);
        } else {
            nl->cchar=NLRESTART;
            nl->cvalid=1;
            nl->run_odev();
            nl->istate=IWAIT;
        }
        return;
    case IREAD:
    case IESCAPED:
        if (nl->err_idev!=0) {
Restart:
            nl->cchar=NLRESTART;
            nl->cvalid=1;
            nl->run_odev();
            nl->istate=IWAIT;
            nl->icount=0;
            nl->inext=nl->istart;
            _nullprio(0,0);
            strtclk();
            return;
        }
switch (ch) {
    case NLEOB:
Done:
        nl->istate=IDONE;
        ready(nl->ipid,RESCHNO);
        _nullprio(0,0);
        strtclk();
        resched();
        return;
    case NLSTART:
        nl->icount=0;
}

```

```
    nl->inext=nl->istart;
    return;
case NLESC:
    if (nl->istate==IESCAPED)
        goto Restart;
    nl->istate=IESCAPED;
    return;
default:
    if (nl->istate==IESCAPED) {
        nl->istate=IREAD;
        ch|=NL_UNESC;

        *nl->inext++=ch;
        if (++nl->icount==nl->imax)
            goto Done;
    }
    return;
case IWAIT:
    if (ch==NLSTART)
        goto SetRead;
    if (ch==NLEOB)
        nl->istate=IREADY;
    return;
case IDONE:
    return;
}

/* end of file */
```

V implementaci této obslužné rutiny může kromě věcí již známých stát za pozornost snad jen využití služby '_nullprio'. Tato služba změní prioritu nulového procesu; je-li to zapotřebí, přemístí jej na začátek ready fronty a je-li to zapotřebí, vyvolá přeplánování. Kvůli efektivitě by bylo nejvhodnější službu realizovat jako makro; vytvoření takového makra (které není nikterak složité) ponecháme pro procvičení na čtenáři.

8.3.2 Síťová vrstva

Hlavním úkolem síťové vrstvy je zabezpečení spolehlivosti přenosu po síti. Je zapotřebí navrhnout rutiny síťové vrstvy takovým způsobem, aby se vyšší vrstvy síťového vybavení již nemusely starat o to, zda se nějaký blok dat neztratil nebo zda jednotlivé bloky přicházejí ve správném pořadí.

Jaksi v 'nadplánu' síťová vrstva zajišťuje rozdělování přijatých frame na ty, které je zapotřebí odeslat dále po síti (a ty rovnou předá linkové vrstvě k odeslání), a na ty, jejichž cílovou adresou na úrovni zóny je tento počítač¹⁰² - ty předá vyšším vrstvám síťového vybavení. Tuto úlohy by stejně snadno mohla zajišťovat některá z vyšších vrstev; je však výhodné ji přidělit vrstvě síťové, protože 'cizí' frame je velmi rychle 'odbaven' a odesán dále, aniž by proto musel procházet mnoha vrstvami programového vybavení.

8.3.2.1 Návrh protokolu

Asi nejdůležitější částí návrhu (nejen) síťové vrstvy je návrh protokolu, podle kterého spolu budou síťové vrstvy dvou počítačů komunikovat. Součástí protokolu je způsob ohlašování, že při přenosu došlo k chybě, mechanismus vzájemné synchronizace např. po výpadku jednoho z počítačů a podobně. Protokol je samozřejmě do jisté míry ovlivněn také ostatními vrstvami (a naopak) - připomeneme-li si implementaci linkové vrstvy, vidíme, že aktivita bude na straně počítače, který vysílá data, zatímco příjemce bude na blok pasivně čekat (mohlo by tomu být naopak - vysílající by mohl ukládat bloky připravené k odeslání do fronty, ze které by vždy na přímou žádost příjemce jeden odeslal).

Vzhledem k tomu, že hlavním úkolem síťové vrstvy je zabezpečení spolehlivosti, budeme se nejprve věnovat jemu. Síťová vrstva musí mít k dispozici prostředky, které před odesláním bloku doplní frame o kontrolní údaje, a po příjmu porovnáním těchto kontrolních údajů s obsahem ověří,

¹⁰²Definitivním cílem přenášených dat samozřejmě může být úplně jiný počítač v úplně jiné zóně.

Operační systémy

nebyl-li blok 'po cestě' poškozen. V nejjednodušším případě může být kontrolním údajem pouhá velikost frame, porovnaná u příjemce se skutečnou velikostí přijatého bloku (tak tomu např. je v originálním XINU). Pro zvýšení spolehlivosti přenosu však můžeme kontrolní informace rozšířit např. o tzv. checksum nebo kód CRC. My zde použijeme mechanismus obecné funkce, která generuje nebo ověřuje kontrolní kódy; implementace této funkce může záviset na konkrétním použití. V extrémním případě velmi spolehlivého technického vybavení by funkce mohla být snadno zcela vypuštěna - implementovali bychom ji jako makro, které nevytváří nic a při kontrole pouze ohláší 'v pořádku'. Funkce by mohla být deklarována např. takto:

```
/* crc.h - makecrc, iscrcok */

#define CRC_LEN 2      /* délka kontrolní informace v bytech */

void makecrc(char *buff,unsigned len);
int iscrcok(char *buff,unsigned len);

/* end of file */
```

Funkce 'makecrc' uloží na začátek bufferu kontrolní kódy zabírající 'CRC_LEN' byteů (předpokládáme, že tam je pro ně vyhrazené místo). Funkce 'testcrc' naopak tyto kódy ověřuje; jsou-li v pořádku (a je-li správná i délka bufferu), vrátí nenulovou hodnotu.

S využitím funkcí 'makecrc' a 'iscrcok' tedy síťová vrstva dokáže zjistit, zda je přijatý frame v pořádku nebo ne. V případě, že je v něm chyba, musí mít síťová vrstva možnost tento fakt oznámit odesírajícímu; pro větší spolehlivost přenosu navrhнемe protokol tak, že síťová vrstva příjemce bude potvrzovat příjem každého frame. Způsob potvrzení se přitom bude samozřejmě lišit v závislosti na tom, byl-li frame přijat bez chyby nebo ne - v prvním případě bude potvrzení znamenat 'mám frame, pošli další', ve druhém 'frame poškozen, pošli ho znova'. Prvnímu případu budeme říkat **pozitivní potvrzení** (positive acknowledgement), druhému **negativní potvrzení** (negative acknowledgement).

Mechanismus potvrzování sám o sobě však ještě nezajistí spolehlivý přenos. Představme si velmi jednoduchou situaci, kdy je blok dat přijat bez chyby, ale odpovídající pozitivní potvrzení se ztratí (dojde tedy k chybě při přenosu potvrzení, ne dat). Systém v takovém případě skončí v deadlocku: vysílající

bude na věky věků marně čekat na potvrzení odeslaného bloku, zatímco příjemce bude stejně dlouho očekávat další blok.

Protokol je proto nutné doplnit o časování (timeout). Jestliže vysílající 'nějak příliš dlouho' nedostává potvrzení, může se rozhodnout o vlastní iniciativě odeslat blok znova¹⁰³. Tento mechanismus je velmi důležitý a jestliže nemáme stoprocentně spolehlivé technické vybavení (a takové dosud nikdo nevyrobil a těžko kdy vyrobí), nemůžeme bez něj dosáhnout bezpečného a spolehlivého síťového přenosu.

Bohužel, ani časování ještě nestačí k dosažení potřebné spolehlivosti. Problém spočívá v tom, že oba propojené počítače pracují zcela paralelně, a proces na jednom z nich proto nemůže nikdy vědět, co právě teď dělá proces na druhém. V našem konkrétním případě to znamená, že ve chvíli kdy síťová vrstva přijímajícího systému odešle potvrzení, mohla by síťová vrstva příjemce omylem potvrzení spojit s jiným blokem. Proto je zapotřebí jednotlivé datové bloky číslovat; součástí potvrzení pak může v každém případě být číslo bloku, který síťová vrstva očekává: číslo chybně přijatého při negativním potvrzení nebo příští číslo při potvrzení pozitivním.

Existují dva důvody, proč není možné číslovat všechny bloky sekvenčně:

- Čísla by v průběhu práce systému příliš narůstala. Nezapomínejme, že potvrzení je kódováno v jediném bytu; aby se do tohoto bytu vešlo i číslo bloku, nesmí zabrat více než několik bitů.
- Po delším výpadku některé ze stanic musíme zajistit resynchronizaci, při které se číslování opět 'sejde'. Není proto ani zapotřebí udržovat dlouhou sekvenci čísel bloků, protože po takovéto resynchronizaci by stejně byla porušena.

¹⁰³Samozřejmě by bylo stejně dobré možné vybavit časováním příjemce, který by mohl po nějaké době znova odeslat potvrzení; v našem návrhu protokolu je však vysílající aktivním a příjemce pasivním účastníkem, proto ponecháme odpovědnost za případné nové odeslání dat na vysílajícím.

Bloky proto budeme číslovat modulo nějaké vhodné malé číslo - běžně se čísla bloků kódují do dvou- nebo tříbitových hodnot.

Číslování bloků a potvrzení však přináší nebezpečí další možnosti zablokování systému - totiž ztráty synchronizace. Představme si, že se při přenosu ztratí třeba pozitivní potvrzení bloku číslo 2. Vysílající je vybaven časováním; proto po nějaké době dojde ke správnému názoru, že se potvrzení někde ztratilo a odešle blok číslo 2 znovu¹⁰⁴. Příjemce však již blok číslo dvě zpracoval; nehodlá proto akceptovat nic jiného, než blok číslo 3. Taková situace by opět mohla trvat na věky věků nebo alespoň do vypnutí počítačové sítě.

Je proto zapotřebí protokol doplnit ještě o možnost **resynchronizace**. Jestliže příjemce dostává 'nějak příliš dlouho' bloky se špatným pořadovým číslem¹⁰⁵, odešle vysílajícímu speciální potvrzení vyžadující resynchronizaci. Vysílající na základě tohoto potvrzení odešle blok znova, stejně jako při negativním potvrzení, přidělí mu však číslo 0 (a další bloky čísluje dál od jedničky). Příjemce samozřejmě při odesílání požadavku na resynchronizaci inicializoval i své číslování a blok 0 očekává.

Analyzujeme-li podrobně tento mechanismus vidíme že k zablokování nemůže dojít a že každý frame se dostane ke svému příjemci, a to ve správném pořadí. Při ztrátě pozitivního potvrzení však může dojít k tomu, že některý frame bude zduplikován. Vyšší vrstvy síťového vybavení se tedy ještě musí postarat o vyřazení případných duplicitních dat (což je samozřejmě velmi snadné); s touto výjimkou se mohou na síťovou vrstvu plně spolehnout.

¹⁰⁴Nemůže odeslat rovnou blok číslo tři, protože ztracené potvrzení mohlo být klidně negativní.

¹⁰⁵Samozřejmě, že resynchronizace je potřebná již při přijetí jediného bloku se špatným sekvenčním číslem; nemůžeme však vědět, nejdřív-li se o pouhou chybu na lince, která zrovna zasáhla číslo bloku. Pokud bude číslo bloku zabezpečeno proti chybě, bude na místě vyvolat resynchronizaci již napoprvé.

8.3.2.2 Struktura síťové vrstvy

Na první pohled vidíme, že síťová vrstva by se měla skládat ze dvou procesů. První z nich bude číst data ze sítě, ověřovat jejich správnost a odesílat je dál nebo předávat je vyšším vrstvám síťového software. Druhý proces - zcela nezávislý na prvním - bude naopak přebírat datagramy od vyšších vrstev a jako frame je bude odesílat dále.

Pokusíme-li se právě popsaným způsobem síťovou vrstvu skutečně naprogramovat, narazíme na problém s realizací časování (timeout) pro odesílající proces. Ten totiž čeká na potvrzení uvnitř služby 'getc'; podíváme-li se na její realizaci v linkové vrstvě (v odstavci 8.3.1.3), uvidíme, že vlastně čeká na přijetí zprávy. Mechanismus zpráv, který máme v XINU k dispozici, neumožňuje přerušit čekání po uplynutí nějakého času.

Mohli bychom samozřejmě doplnit mechanismus zpráv o novou službu 'receive_with_timeout', která by zajistila právě to, co potřebujeme; obvykle však není vhodné zasahovat při implementaci některé z vrstev operačního systému do nižších a již odladěných vrstev. Využijeme proto jiného mechanismu:

Vytvoříme třetí proces, který se bude starat o časování sám. Dříve, než zavolá vysílající proces službu 'getc' (obsahující volání služby 'receive'), aktivuje časovač proces. Ten využije služby 'sleep' k tomu, aby nějakou dobu počkal; jestliže odesílající proces dostane potvrzení, opět časovač deaktivuje a nic se neděje. Pokud se však časovač 'dočká' vypršení doby, po kterou měl čekat, poše prostě volajícímu procesu zprávu 'TIMEOUT'. Volající proces zprávu dostane samozřejmě jako návratovou hodnotu služby 'getc'; stačí však zvolit kód zprávy 'TIMEOUT' odlišný od všech možných potvrzení a vysílající bude moci snadno zjistit, v jaké situaci se nachází.

Zbývá nám ještě rozhodnout, jakým způsobem budou procesy síťové vrstvy komunikovat s vyššími vrstvami síťového programového vybavení. K tomu se právě velmi dobře hodí mechanismus portů, jak jsme si jej popsali v odstavci 8.3. Použijeme dvou portů: na první z nich budou vyšší vrstvy ukládat adresy bufferu s daty, která chtějí odeslat po síti; vysílající proces bude postupně datagramy odebírat a odesílat. Na druhý port bude naopak přijímající

Operační systémy

proces ukládat adresy bufferů, do kterých uložil načtená data; buffery tam budou k dispozici vyšším vrstvám. První port bude zároveň sloužit uvnitř síťové vrstvy pro vzájemnou komunikaci přijímajícího a vysílajícího procesu: příjemce na něj bude ukládat adresy bufferů obsahujících frame, které je zapotřebí odeslat ihned dále; vysílající je odtamtud bude odebírat a odesílat.

Tento návrh má jednu nevýhodu: fronta zpráv na portu je fronta typu FIFO, aneb 'kdo dřív přijde, ten dřív mele'. Frame proto budou odesílány ve stejném pořadí, ve kterém jsou ukládány na 'výstupní' port, bez ohledu na to, zda byly generovány vyššími vrstvami síťového vybavení nebo zda se jedná o cizí frame, přijaté po síti a odesílané dále. Pro síť, na které bychom předpokládali větší zatížení, by bylo vhodnější tento mechanismus modifikovat, aby měly cizí frame přednost (tj. "především odbavím to co už v síti je, a pak tam teprve budu přidávat něco dalšího") - jinak by mohlo případně dojít k zahlcení sítě.

Nyní se konečně můžeme pustit do programování:

8.3.2.3 Společné deklarace

Všechny tři procesy síťové vrstvy samozřejmě potřebují mít k dispozici řadu sdílených proměnných a musí využívat společných deklarací. Některé deklarace však budou potřebovat i vyšší vrstvy síťového programového vybavení - připomeňme, že buffery pro data na všech úrovních jsou ve skutečnosti buffery pro frame. Všechny potřebné deklarace proto shrneme do společného hlavičkového souboru 'net_net.h'¹⁰⁶:

```
/* net_net.h */

#define PACK      '\x10'      /* pozitivní potvrzení */
#define NACK      '\x20'      /* negativní potvrzení */
#define SNACK     '\x30'      /* požadavek na resynchronizaci */
#define TIMEOUT   0xff0       /* kód pro timeout */

#define SEQN      '\x0f'      /* číslo bloku v potvrzení */
```

¹⁰⁶Pro zvýšení modularity by v komplexnějším systému bylo pravděpodobně výhodnější ponechat v souboru 'net_net.h' skutečně pouze ty deklarace, které jsou zapotřebí pro všechny vrstvy síťového software, a specifické deklarace pro síťovou vrstvu soustředit např. do souboru 'net_frame.h'.

```

#define BCAST      0xff      /* rozeslání bloku všem počítačům */
#define ACKTMO     4          /* timeout při ztrátě potvrzení */
#define SFAIL       2          /* počet chyb sekvence pro SNACK */
#define SMODULO    7          /* modulo pro sekvenční čísla */

#define FMINLEN   CRC_LEN+5  /* minimální velikost frame */
#define FDATALEN  128         /* max. velikost datagramu */
#define FLEN       sizeof(struct frame)/* max. velikost frame */

struct frame {
    char ctrl[CRC_LEN];      /* kontrolní údaje */
    char from,to;            /* odesilatel a adresát frame */
    char seq;                /* sekvenční číslo */
    int len;                 /* délka datagramu (<=FDATALEN) */
    char data[FDATALEN];     /* data frame, tj. datagram */
};

struct frnet { /* globální data síťové vrstvy */
    int iport;              /* port pro načtené bloky */
    int oport;              /* port pro odesílaná data */
    int idev;               /* číslo vstup. síťového zařízení */
    int odev;               /* číslo výst. síťového zařízení */
    int timer;              /* čítač pro timeout */
    int tpid;               /* časovač proces */
    int tdpid;              /* proces, který čeká na timeout */
    char stid;              /* síťové číslo tohoto počítače */
    char iseq;               /* čítač sekvence pro vstup */
    char sfails;             /* počet chyb sekvence při vstupu */
    char oseq;               /* čítač sekvence pro výstup */
};

extern struct frnet frnets[];

/* end of file */

```

Význam všech polí je jasný z komentářů; zvláštní vysvětlení si snad zaslouží jen symbolická konstanta 'BCAST': někdy může být vhodné určitý paket rozeslat všem počítačům na celé síti (může se to hodit třeba pro rozesílání administrativních údajů jako je změna konfigurace sítě). Budeme tedy chtít, aby síťová vrstva odpovídající frame předala vyšším vrstvám programového vybavení a zároveň jej odeslala dál jako cizí frame. To samozřejmě není nic těžkého - musíme jen síťové vrstvě tento požadavek nějak oznámit. K tomu právě slouží hodnota 'BCAST', kterou ve zmíněném případě použijeme na místě síťové adresy cílového počítače.

'Globální' proměnné pro síťovou vrstvu jsou uloženy v tabulce 'frnets' proto, aby mohl jeden počítač obsluhovat více zón sítě. V tomto smyslu budeme také nadále používat pojem 'číslo zóny': jako interní číslo zóny v rámci jednoho počítače, čili jako jeho index do pole 'frnets'.

8.3.2.4 Rozhraní pro vyšší vrstvy

Nejprve uvedeme služby, které síťová vrstva nabízí vyšším vrstvám síťového vybavení. Jsou jen dvě a jsou poměrně jednoduché: služba 'freceive', jejímž parametrem je číslo zóny, vrátí adresu bufferu, ve kterém je uložen přijatý datagram¹⁰⁷. Tato služba je skutečně mimořádně jednoduchá:

```
/* freceive.c - freceive */

#include <conf.h>
#include <crc.h>
#include <net_net.h>

/*
 * freceive -- čtení frame ze sítě
 */
char *freceive(int net)

    return(preceive(frnets[net].iport));

/* end of file */
```

Vysvětlovat funkci služby 'freceive' je asi opravdu zbytečné. Podíváme se namísto toho na implementaci služby 'fsend', která naopak zadáný datagram po síti odešle. Služba má tři parametry: číslo sítě, identifikační číslo příjemce v rámci zóny a adresu bufferu, který obsahuje odesílaný datagram (i zde platí poznámka 107). Její implementace je jen nepatrně komplikovanější:

¹⁰⁷Připomeňme, že datagram je uložen v bufferu deklarovaném v souboru 'net_net.h' jako 'struct frame'. Relační vrstva, která bude služby 'freceive' a 'fsend' používat, tedy musí přeskočit začátek bufferu v délce 'FMINLEN'; teprve od bytu číslo 'FMINLEN' datagram začíná. Délku datagramu přitom relační vrstva zjistí z (po 'freceive') nebo uloží do (před 'fsend') položky 'len' ve struktuře 'struct frame'.

```
/* fsend.c - fsend */

#include <conf.h>
#include <crc.h>
#include <net_net.h>

/*
 *  fsend -- odeslání frame po síti
 */
void fsend(int net, char wkid, char *buff)
{
    struct frnet *fn=&frnets[net];
    struct frame *fm=buf;
    fm->to=wkid;
    fm->from=fn->stid;
    psend(fn->oport, buff);
}

/* end of file */

```

Služba nejprve uloží do hlavičky frame síťové adresy zdrojového a cílového počítače. Cílovou adresu službě předá relační vrstva, zdrojovou zjistí z položky 'stid' ve svých datech, kde je uloženo číslo lokálního počítače. Pak služba umístí frame na patřičný port.

8.3.2.5 Vstupní proces

Vstupní proces používá pro odesílání potvrzení pomocné služby 'sendack', 'sendnack' a 'sendsack'. První z nich odešle pozitivní potvrzení, druhá negativní potvrzení a třetí podle počtu sekvenčních chyb odešle buď negativní potvrzení nebo požadavek na resynchronizaci. Podívejme se na jejich zdrojové texty:

```
/* finp_serv.c - sendack, sendnack, sendsack */

#include <conf.h>
#include <crc.h>
#include <net_net.h>

/*
 *  sendack -- odešle pozitivní potvrzení se sekvenčním číslem

```

Operační systémy

```
/*
 */
void sendack(struct frnet *fn)
{
    fn->sfails=0;
    if (++fn->iseq>SMODULO)
        fn->iseq=0;
    putc(fn->idev,PACK|fn->iseq);

/*
 *  sendnack -- odešle negativní potvrzení se sekvenčním číslem
 */
void sendnack(struct frnet *fn)

    putc(fn->idev,NACK|fn->iseq);

/*
 *  sensack -- zpracuje situaci po chybě sekvence
 */
void sensdack(struct frnet *fn)
{
    if (++fn->sfails==SFAIL) {
        fn->iseq=0;
        fn->sfails=0;
        putc(fn->idev,SNACK);
    } else
        putc(fn->idev,NACK);

/* end of file */
```

S využitím těchto pomocných služeb již není obtížné připravit kód, který bude zpracováván vstupním procesem:

```
/* finpproc.c - vstupní proces síťové vrstvy */

#include <conf.h>
#include <crc.h>
#include <net_net.h>

/*
 *  finpproc -- proces
```

```

*/
void finpproc(int net)
{
    struct frnet *fn=&frnets[net];
    struct frame *fm;

    new(fm);
    fn->iseq=0;
    fn->sfails=0;
    for (;;) { /* forever */
        int len;

        if ((len=read(fn->idev,fm,FLEN))<FMINLEN || 
            liscrcok(fm,len))
            sendnack(fn);
        else if (fm->seq!=fn->iseq)
            sendsack(fm);
        else {
            if (fm->to==BCAST) {
                if (fm->from==fn->stid)
                    psend(fm->iport, fm);
                else {
                    struct frame *f;

                    new(f);
                    bcopy(f, fm, len);
                    psend(fm->iport, f);
                    sendIt:
                    psend(fm->oport, fm);
                }
            } else if (fm->to==fn->stid)
                psend(fm->iport, fm);
            else if (fm->from==fn->stid) {
                syslog("nemám spojení s %d ?!?", fm->to);
                goto SkipAlloc;
            } else goto SendIt;
            new(fm);
        SkipAlloc:
            sendack(fm);
        }
    }
}

/* end of file */

```

Proces není komplikovaný; jen na první pohled může jeho porozumění ztížit trochu častější použití příkazu 'goto'.

Operační systémy

Nejprve proces inicializuje lokální proměnné, alokuje blok paměti pro načtený frame a pak přejde do nekonečného cyklu. To je běžný případ systémových procesů - celou dobu tráví v nekonečném cyklu, ve kterém zpracovávají potřebná data. Většinu doby samozřejmě takový proces nepracuje, ale v rámci některé ze systémových služeb čeká mimo ready frontu (zde je 'čekac' službou služba 'read', která - jak jsme viděli v popisu linkové vrstvy - přemístí proces do stavu 'suspended')¹⁰⁸.

První, co proces v cyklu udělá, je čtení bloku dat prostřednictvím ovladače síťového zařízení (jinými slovy prostřednictvím linkové vrstvy). Po přijetí bloku proces nejprve ověří, může-li se vůbec jednat o frame, tj. je-li blok alespoň tak dlouhý, aby se do něj vešla hlavička frame. Pak si vyžádá ověření dat frame službou 'iscrcok'. Jestliže blok není v pořádku, odešle negativní potvrzení a čte další blok.

Byl-li blok v pořádku, srovná proces jeho sekvenční číslo s očekávaným sekvenčním číslem. Jestliže se obě čísla liší, odešle proces prostřednictvím služby 'sendsack' buď negativní potvrzení nebo požadavek na synchronizaci a opět Čte další blok.

V případě, že byl blok v pořádku a měl správné sekvenční číslo, přejde proces na zpracování frame. Podívejme se postupně na větvení programu:

- První příkaz 'if' zjistí, není-li příjemcem 'BCAST' - nejdenná-li se tedy o zprávu, rozesílanou po celé síti. Je-li tomu tak, předá frame v každém případě relační vrstvě; pokud byl frame vytvořen na jiném počítači, odešle jej také dále.

Smysl tohoto mechanismu je jasný: jeden počítač zprávu typu 'BCAST' vygeneruje. Zpráva projde všemi ostatními počítači v kruhové síti XINU; každý počítač si ji ponechá a zároveň ji odešle dále. Když se zpráva dostane opět na počítač, který ji odeslal původně, nebude se již

¹⁰⁸Pro takovéto procesy se běžně používá pojem 'démon'.

samozřejmě posílat znovu dokola; vyšší vrstva programového vybavení ji však dostane jako potvrzení, že zpráva prošla skutečně celou sítí¹⁰⁹.

- Druhý příkaz 'if' ověří, není-li lokální počítač příjemcem frame. Je-li tomu tak, předá frame relační vrstvě.
- Třetí příkaz 'if' srovná odesílatele frame s číslem lokálního počítače. Pokud se shodují, znamená to, že příjemce z nějakého důvodu frame neodebral. Proces tuto situaci oznámí uživateli a frame 'zahodí' - tj. nechá do téže paměti číst další blok (proto 'goto SkipAlloc').

Setkáváme se zde se speciální službou 'syslog'. Ta nedělá nic jiného, než že vypíše své argumenty 'systémového záznamu'. Tím může být třeba speciální soubor nebo zvolené okno na obrazovce, nebo tiskárna připojená k počítači - to záleží na konkrétní implementaci systému. Operační systém službu 'syslog' používá pro hlášení všech důležitých nebo podivných situací; hledá-li pak administrátor systému řešení nějakého problému, může mu být systémový záznam velmi významným pomocníkem. Systémový záznam (spojený s intenzivním používáním služby 'syslog') je také prakticky nutnou podmínkou pro odladění operačního systému.

- Jestliže frame 'nepoznal' žádný z příkazů 'if', jedná se o cizí frame a proces jej prostě odešle dále ('goto SendIt').

Paměť, ve které je přijatý blok uložen, není ještě možné uvolnit (s výjimkou případu 'třetího if-u') - data v ní čekají buď na odeslání na další počítač, nebo na zpracování relační vrstvou. Proces proto musí alokovat další blok paměti službou 'new'.

Potom proces již jen odešle pozitivní potvrzení a čte další blok ze sítě.

¹⁰⁹ Přesněji řečeno celou zónou. Je-li síť složena z několika zón, musí se o rozeslání zprávy všem zónám pochopitelně postarat relační vrstva.

8.3.2.6 Výstupní proces

Výstupní proces používá podobně jako vstupní také svou pomocnou službu - její název je '_frsend'. Tentokrát je však tato pomocná služba daleko složitější než samotný proces - ten totiž pouze odebírá jednotlivé frame z výstupního portu a předává je právě službě '_frsend' k odeslání.

Služba '_frsend' tedy musí zajistit nejen odeslání frame - což je jednoduché, stačí předat frame linkové vrstvě - ale i zpracování potvrzení a případnou resynchronizaci. Podívejme se na její zdrojový text:

```
/* frsend.c - _frsend */

#include <conf.h>
#include <crc.h>
#include <net/net.h>

/*
 * _frsend -- odeslání frame, zpracování potvrzení
 */
void frsend(struct frnet *fn, struct frame *fm)
{
    char seq,nextseq;
    int len;
    int msg;

    fm->seq=seq=fn->oseq;
    nextseq=seq==SMODULO?0:seq+1;
    makecrc(fm,len=fm->len+FMINLEN);
    for (;;) {
        getc(fn->oodev);110
        write(fn->oodev,fm,len);
        fn->timer=ACKTMO;
        fn->tqid=currpid;
        resume(fn->tqid);
        msg=receive();
        fn->timer=-1;
        switch (msg&~SEQN) {
            case PACK:
                if (msg&SEQN==nextseq) {
```

¹¹⁰Služba 'getc' ještě nic neče, jen 'připraví půdu' pro skutečné přečtení potvrzení službou 'receive'. Připomeňme si implementaci služby 'getc' na straně 239.

```

        fn->oseq=nextseg;
        return;
    }
    break;
case SNACK:
    fm->seg=fn->oseg=0;
    nextseg=1;
    sleep(1);
}
sleep(1);

/* end of file */

```

Služba nejprve připraví kontrolní údaje pomocí funkce 'makecrc', pak zjistí právě platné sekvenční číslo bloku pro výstup a vstoupí do cyklu 'for'. Tentokrát se nejedná o věčný cyklus, jako v démonech - cyklus bude ukončen po přijetí pozitivního potvrzení se správným sekvenčním číslem. Než nepředbíhejme, jak se praví často v jiných dobrodružných knížkách, a vraťme se na *začátek* cyklu.

Funkce tam nejprve inicializuje příjem potvrzení službou 'getc' linkové vrstvy, a potom teprve skutečně odešle frame službou 'write'. Pak nastaví čítač časovacího procesu na hodnotu 'ACKTMO', časovačí proces aktivuje a čeká na potvrzení pomocí služby 'receive'. Ta může vrátit některou z následujících hodnot:

- 'PACK' s číslem příštího požadovaného bloku, jestliže druhý počítač odeslaný blok bez problémů přijal;
- 'NACK', jestliže druhý počítač blok přijal chybně;
- 'SNACK', jestliže druhý počítač požaduje resynchronizaci;
- 'TIMEOUT', jestliže od druhého počítače nepřišlo vůbec žádné potvrzení tak dlouho, že mezitím časovačí proces vyčerpal čítač 'timer'.

Operační systémy

Ihněd po návratu ze služby 'receive' proto funkce '_frsend' nejprve deaktivuje časovačí proces uložením záporné hodnoty do čítače, a pak rozliší potřebné alternativy pomocí příkazu 'switch':

- Varianty 'NACK' a 'TIMEOUT' není zapotřebí vůbec detekovat - cyklus automaticky zajistí nové odeslání bloku, což je přesně to, co v takovémto případě chceme dělat. Abychom dali příjemci nějaký čas na zotavení (a abychom nechali lokálnímu počítači také trochu času pro práci, jestliže druhý počítač odmítá vůbec komunikovat), zavoláme službu 'sleep'.
- Po přijetí potvrzení 'PACK' ještě zkонтrolujeme, je-li v pořádku sekvenční číslo bloku. Jestliže ano, byl blok skutečně bez problémů přijat a služba '_frsend' může skončit. Jestliže ne, odešleme blok znovu.
- Konečně po přijetí požadavku 'SNACK' nejprve synchronizujeme sekvenční číslo na nulu, pak ponecháme příjemci (který samozřejmě také musí synchronizovat) ještě více času než obvykle voláním služby 'sleep', a blok odešleme znovu.

Máme-li hotovou službu '_frsend', je výstupní proces již tak jednoduchý, že to ve světle dlouhého a podrobného rozboru, který jsme síťové vrstvě věnovali, až není hezké - čtenář může mít pocit, že jsme dělali mnoho povyku pro nic:

```
/* foutproc.c - výstupní proces síťové vrstvy */

#include <conf.h>
#include <crc.h>
#include<net_net.h>

/*
 *  foutproc -- proces
 */
void foutproc(int net)
{
    struct frnet *fn=&frnets[net];
    struct frame *fm;

    fn->oseq=0;
    for (;;) { /* forever */
        _frsend(fn,fm=preceive(fn->oport));
    }
}
```

```

    dispose(fm);
}

/* end of file */

```

Program nepotřebuje podrobnější popis. Vyplatí se pouze povšimnout si, že zde službou 'dispose' uvolňujeme paměť, kterou alokovaly vyšší vrstvy síťového vybavení (ty zase na oplátku uvolňují paměťové bloky, alokované službami 'new' ve vstupním procesu).

8.3.2.7 Časovači proces

Časovači proces je velmi jednoduchý. Jeho jediným úkolem je počítat čas na základě čítače 'timer', a jestliže dojde k nule, poslat procesu 'tdpid' zprávu 'TIMEOUT'. Z technických důvodů je proces navržen tak, aby bylo možné jej 'zastavit' uložením záporné hodnoty do čítače; to je proto, že pokud by náhodou proces zrovna čekal ve stavu 'sleeping', nelze jej přímo suspendovat. Tako vypadá zdrojový kód:

```

/* ftimeproc.c - časovači proces síťové vrstvy */

#include <conf.h>
#include <crc.h>
#include <net/net.h>

/*
 * ftimeproc -- proces
 */
void ftimeproc(int net)
{
    struct frnet *fn=&frnets[net];

    for (;;) { /* forever */
        for (fn->timer++;--fn->timer>0;)
            sleep(10);
        if (fn->timer==0) send(fn->tdpid, TIMEOUT);
        suspend(currpid);
    }
}
/* end of file */

```

Operační systémy

Při inicializaci (kterou jako obvykle neuvádíme) využijeme toho, že všechny procesy jsou automaticky službou 'create' vytvořeny ve stavu 'suspended'. Procesy pro vstup a pro výstup při inicializaci ihned převedeme službou 'resume' do ready fronty; časovačí proces však ponecháme suspendovaný - je navržen tak, že bude stejně dobře pracovat po aktivaci na samém začátku, jako po aktivaci uvnitř jeho služby 'suspend'.

8.4 Sítě a bezpečnost

Zabezpečení klade na počítačové sítě poměrně vysoké nároky. Vzhledem k dosud poměrně výrazně odlišné struktuře lokálních a globálních sítí se v obou případech setkáváme se zásadně odlišnými problémy; podíváme se proto z bezpečnostního hlediska na každý typ sítě zvlášť.

8.4.1 Lokální síť

Lokální síť musí být zabezpečeny především proti výpadku systému. Jestliže jeden počítač přestane pracovat nebo je od sítě odpojen, nesmí to znemožnit komunikaci ostatních počítačů po síti.

(Jednoduchá síť XINU, jejíž implementací jsme se zabývali, je samozřejmě rezistentní vůči odpojení některého počítače pouze za předpokladu, že bude propojení sítě 'ručně' opět doplněno na celý kruh.)

Kromě toho může být někdy vhodné kódovat pakety, aby nebylo možné na jedné stanici zjistit, jaká data si posílají procesy na jiných stanicích. To samozřejmě není žádný technický problém - stačí data v rámci některé z vyšších vrstev síťového programového vybavení zakódovat na základě uživatelského klíče; rutiny příjemce pak pakety opět dekódují.

8.4.2 Globální sítě

Hlavní úlohou návrháře zabezpečení globálních sítí je naproti tomu zajištění sítě proti průniku zvenčí (tzv. hackingu). Standardně užívaným mechanismem je systém hesel; pro skutečné zabezpečení je však zapotřebí jej doplnit kódováním a pravidelnými změnami hesel. Vhodné je také využít nějaký mechanismus pro generování různých hesel pro každé připojení (např. na základě čítače připojení nebo data a času).

Operační systémy

9. Systém služeb

V kapitole 2 jsme si vybrali definici, podle které je operační systém správcem prostředků. Operační systém však má ještě jednu velmi důležitou úlohu: musí zajistit maximální množství často požadovaných služeb tak, aby každý programátor nemusel vytvářet pokaždé znovu a znovu stejný kód. Jak se postupně vyvíjely operační systémy, objevovalo se samozřejmě čím dál tím více takových služeb; i když nejdůležitější částí každého moderního operačního systému zůstává správa prostředků, bývají co do objemu služby daleko a daleko rozsáhlejší.

Kromě vlastního množství služeb je samozřejmě nesmírně důležitá i jejich univerzálnost a možnost jejich mírného přizpůsobení konkrétním požadavkům právě vyvíjeného programu tak, aby programátor byl nucen vytvářet skutečně jen a pouze nový kód. Zde se znova a daleko výrazněji objevuje výhoda objektově orientovaných operačních systémů jako je NeXTStep nebo EPOC - dědičnost objektových programátorských prostředků dává službám těchto systémů daleko větší flexibilitu, než tomu je u obdobných služeb systémů ostatních.

9.1 Jaké služby potřebujeme

Od samého začátku samozřejmě všechny operační systémy nabízely programátorům alespoň základní služby pro přidělování a uvolňování operační paměti (nemáme-li k dispozici virtuální adresový prostor, ani by to bez nich nešlo) a poměrně komfortní služby pro práci se soubory (jak jsme ostatně viděli v kapitole 7). Prostřednictvím ovladačů zařízení, připojených k systému, měli programátoři navíc k dispozici i služby pro práci s těmito zařízeními; málokdy se však jednalo o služby dostatečně vysoké úrovně. Multitaskové operační systémy pak samozřejmě musí nabízet služby správce procesů.

To vše je sice hezké, ale dnes to ani zdaleka nestačí. Operační systém musí nabízet poměrně velmi luxusní služby alespoň v následujících oblastech:

Operační systémy

- Grafické operace, práce s okny a graficky orientovaná komunikace s uživatelem. Této problematice se budeme velmi podrobně věnovat ve druhém dílu knihy; zde ji již proto nebudeme dále rozebírat - uvědomme si pouze, že sem patří také např. volba stylu písma, volba barvy, editace textu v rozsahu jediného řádku i na úrovni kompletního editoru a podobně. Do této oblasti patří mimo jiné také tisk (v rozumném operačním systému je tisk textu pouze speciálním případem tisku grafiky); i o něm se v druhém dílu knihy zmíníme.
- Práce s textovými řetězci a s bloky dat v operační paměti. Nejrůznější přesunování textů nebo skupin bytů, vyhledávání v nich a jejich kombinace patří snad mezi nejčastější základní operace, které kterýkoli program provádí. Zpracování textu pak je základní součástí prakticky všech programů, které vůbec nějak komunikují s uživatelem.

Tato problematika velmi úzce souvisí s graficky orientovanou komunikací s uživatelem - je totiž nutné si uvědomit, že práce s textem nekončí u knihoven jazyka C a standardních služeb pro kopírování či připojení textového řetězce; služby moderního operačního systému musí být schopné pracovat s formátovaným textem obsahujícím údaje o použitém fontu, velikosti a typu písma, o zarovnání jednotlivých odstavců, o použité barvě ...

- Není-li operační systém určen výhradně pro procesory vybavené jednotkou pro výpočty v pohyblivé řádové čárce (nebo alespoň odpovídajícím koprocesorem), je zapotřebí, aby obsahoval rozsáhlou skupinu služeb pro práci s neceločíselnými hodnotami.
- Téměř pro každé zařízení (jako příklad jmenujme zvukový vstup a výstup nebo systémové hodiny) by měl operační systém nabízet skupinu služeb, které samy zpracují nejběžnější požadavky kladené na zařízení. S našimi příklady by tedy systém jistě neměl ponechávat na každém programátorovi, aby se sám staral o generování tónů požadované výšky, intenzity, délky a barvy nebo o přehrávání samplovaného zvukového záznamu; analogicky pro zvukový vstup by měl systém zajistit dekódování a případné uložení samplovaného záznamu.

- Nesmírně důležitou oblastí - zvláště z našeho hlediska - je podpora práce v cizím jazyce. Operační systém by měl nabízet řadu prostředků umožňujících jak práci s dokumenty psanými v jiném jazyce, než pro který byl systém původně navržen, tak i vlastní komunikaci s uživatelem v 'jeho' jazyce. Nejmodernější operační systémy (např. NeXTStep) pak mohou s každým ze svých uživatelů komunikovat jinou řečí podle jeho osobní volby.
- Jednou z nejčastějších problematik řešených na počítačích je ukládání a opětovné vyhledávání údajů - tedy databáze. Dokonce i řada 'nedatabázových' úloh se dá vyřešit daleko pohodlnějším a efektivnějším způsobem, je-li možné jako základ řešení použít již hotový, fungující a rychlý databázový systém. Programátoři by proto v moderním operačním systému měli mít k dispozici i poměrně velmi rozsáhlý balík 'databázových' služeb.
- Velmi významnou oblastí je i komunikace mezi programy na vyšší úrovni. Jednotlivé procesy samozřejmě mohou snadno komunikovat s využitím aparátu zpráv nebo semaforů, které jim nabízí správce procesů; je však zapotřebí umožnit aplikacím, aby si mohly bez extrémního úsilí a explicitní dohody jejich programátorů navzájem předávat data ke zpracování, informace o stavu těchto dat a o jejich případné modifikaci a podobně.

Uvedený seznam pochopitelně není vyčerpávající; shrnuje však v zásadě všechny oblasti, které pokrýval rozumný operační systém v době vzniku této knihy. Není nepravděpodobné, že s rozvojem výpočetní techniky se budou 'povinné'¹¹¹ služby rozšiřovat a že během času budou zahrnovat např. univerzální prázdný expertní systém nebo automatické překlady z jednoho jazyka do druhého. V každém případě se již dnes pomalu mezi 'povinné' služby operačního systému řadí třeba korektor pravopisu; nejmodernější systémy jdou ještě mnohem dále - např. součástí NeXTstepu, přístupnou kterékoli aplikaci pracující s textem, je kompletní výkladový slovník s mnoha desítkami tisíc pojmu.

¹¹¹Ve smyslu konkurenceschopnosti operačního systému.

9.2 Implementace služeb

V klasických operačních systémech existovaly v zásadě dvě možnosti implementace služeb. Buď se mohlo jednat o 'plnohodnotné' systémové služby, které operační systém zajišťoval na vyžádání, nebo mohly být služby uloženy na knihovnách, odkud se při vytváření aplikáčních programů připojily k jejich kódu.

Dnešní operační systémy obvykle disponují dvěma dalšími prostředky pro implementaci služeb. Jedná se o servery a o sdílené knihovny. Nejmodernější objektově orientované operační systémy pak mají k dispozici navíc také aparát komunikace objektů.

V praxi operační systém pro zajištění služeb obvykle kombinuje všechny popsané metody. Podívejme se proto postupně na jejich princip, výhody a nevýhody:

9.2.1 Služby systému

Služby systému jsou obvykle vyvolávány pomocí speciální instrukce procesoru, která přepne pracovní režim z uživatelského do systémového a předá řízení předem zvolené rutině operačního systému. Ta na základě vstupních parametrů zjistí, kterou službu program požadoval a předá řízení rutině, která služby zpracuje. Nejčastěji se používají instrukce pro vyvolání programového přerušení; řada procesorů má navíc některé speciální instrukce, od velmi jednoduchých a užitečných (jako je instrukce SVC systémů IBM360/370 nebo skupina instrukcí 'LineA' mikroprocesorů řady Motorola 680x0) až po velmi komplikované a v praxi jen obtížně použitelné (jako je systém bran '/gates' / mikroprocesorů řady Intel 80x86)¹¹².

¹¹²Případný zájemce o mechanismus popsaných instrukcích naleznete technické podrobnosti v [OS] (SVC), [68030] (skupina instrukcí 'LineA') a v [386] (aparát bran).

Parametry se při vyvolání systémových služeb nejčastěji předávají v registrech. To je výhoda z hlediska rychlosti zpracování služby; je to však nepohodlné v případě, kdy potřebujeme předávat větší množství formátovaných parametrů a musíme je nějak umístit do několika 'bezformátových' registrů. Systémové služby proto v praxi bývají velmi často 'zabaleny' do velmi jednoduchých služeb (umístěných na klasických knihovnách), které pouze překódují parametry z 'pohodlného' tvaru uloženého nejčastěji na zásobníku do registrů a pak zavolají odpovídající systémovou službu.

Další a nejvýznamnější nevýhodou systémových služeb je právě to, že jsou přímo součástí operačního systému. Připomeňme si obrovský rozsah služeb z odstavce 9.1; takový operační systém by se sám nevešel do paměti (pokud bychom neměli k dispozici virtuální paměť). Navíc by v obrovském operačním systému - i přesto, že by byl samozřejmě složen a řady modulů - přece jen narůstalo riziko chyb.

Hlavní výhodou systémových služeb je jejich 'právo dělat cokoli'. Jsou-li služby součástí systému, je také jejich kód zpracováván v systémovém režimu procesoru; mohou tedy přímo ovládat jednotlivá zařízení nebo kanály, mohou maskovat jednotlivá přerušení, mohou přeprogramovávat jednotku řízení paměti a podobně. Systémové služby proto musí v každém případě pokrýt všechny úlohy, pro které je provádění takovýchto potenciálně nebezpečných akcí nezbytné; ostatní úkoly - zpravidla daleko vyšší úrovně - pak mohou být realizovány jinak.

Při určitém zjednodušení¹¹³ může být dobrým příkladem třeba práce s diskem: služby systému by mohly zajistit pouze přečtení a zápis požadovaného bloku dat z disku a na disk (protože pro splnění tohoto úkolu je samozřejmě nutné přímo komunikovat s řadičem disku); všechny ostatní úkoly by mohly být realizovány mimo vlastní systém.

¹¹³V praxi musí být v uživatelském režimu nepřístupný i samotný systém souborů se svým aparátem přístupových práv, aby bylo možné zajistit ochranu souborů jednoho uživatele před náhodným nebo úmyslným poškozením programem jiného uživatele.

9.2.2 Klasické knihovny

Princip klasických knihoven je asi většině čtenářů známý, pro jistotu jej však stručně shrneme. Program se obvykle vytváří ve dvou krocích:

- Nejprve se pomocí překladačů vytvoří jednotlivé moduly. V modulech samozřejmě je množství volání nejrůznějších služeb; na každém takovém místě je instrukce volání podprogramu bez cílové adresy, namísto této adresy je v modulu uloženo jméno požadované služby¹¹⁴. Modul může nějakou službu také sám nabízet; pak je jeho součástí informace o jménu služby a o relativní adrese odpovídajícího podprogramu uvnitř modulu.
- Druhým krokem je spojení všech modulů dohromady. Spojuvací program přitom má k dispozici nejen moduly, které vytvořil programátor aplikace, ale i množství modulů, které jsou uloženy právě v systémových knihovnách. Program pak vybere všechny moduly z knihoven, které obsahují služby volané z 'aplikáčních' modulů, a ze všech modulů dohromady vytvoří hotový program. Všechny instrukce volání podprogramů přitom doplní správnými adresami.

Ve výsledném programu je tedy uložen kompletní kód služeb z knihoven, stejně, jako by jej programátor zapsal při tvorbě programu.

To je samozřejmě současně hlavní nevýhodou klasických knihoven. Máme-li totiž potom sto programů používajících služby z klasických knihoven na disku, máme na disku sto jedna kopii téhož kódu (sto v programech, sto první kopie je uvnitř knihovny, která je na disku obvykle uložena také). Zavedeme-li v multitaskovém prostředí třicet takovýchto programů do paměti, budeme v ní mít opět třicet kopií téhož kódu.

Žádnou významnější výhodu klasické knihovny nemají (aparát sdílených knihoven - není-li pro něj využit systém virtualizace adres - může znamenat určité zvýšení režie; není to však obvyklé). Asi hlavním důvodem, proč klasické

¹¹⁴To je samozřejmě pouze princip; skutečný formát modulů je daleko komplikovanější.

knihovny dosud ani v moderních operačních systémech nevymizely je to, že jsme na jejich používání zvyklí.

V dnešních operačních systémech klasické knihovny obvykle obsahují pouze kratičké pomocné funkce, které převedou normální volání podprogramu na vyvolání systémové služby nebo na odeslání zprávy serveru.

Klasické knihovny mohou sloužit i pro dosažení kompatibility se staršími systémy nebo pro usnadnění práce programátorům, kteří jsou zvyklí na klasické prostředky a nechtějí přecházet na objektově orientované systémy - např. součástí NeXTstepu jsou standardní knihovny UNIXu, součástí daleko menšího EPOCu jsou alespoň knihovny ANSI C. V obou případech se jedná pouze o 'zabalení' komunikace mezi objekty do normálních podprogramů.

923 Servery

Se servery (speciálními procesy, nabízejícími ostatním určité služby) jsme se již několikrát setkali. Stačí si pouze uvědomit, že server nemusí být určen pouze pro obsluhu nějakého fyzického zařízení, ale může také naprostě stejným způsobem nabízet služby, ovládající nějaké 'zařízení' logické - např. systém souborů nebo databází.

Výhody a nevýhody serverů se do jisté míry podobají výhodám a nevýhodám služeb samotného operačního systému. Hlavní rozdíl spočívá v tom, že servery obvykle nepracují v systémovém režimu procesoru (a musí tedy samy využívat systémových služeb pro přímé ovládání zařízení). Zůstává společná nevýhoda zabrané paměti (není-li k dispozici virtuální paměť), i nevýhoda snadnějšího zavedení chyb do velkého programu, jakým takový server obvykle bývá. Vzhledem k tomu, že pro každý úkol může být určen samostatný server, však tato nevýhoda není ani zdaleka tak markantní, jako tomu bylo v případě systémových služeb.

Servery jsou relativně 'samostatné'; kód ze sdílených knihoven je naproti tomu součástí programu, který jej využívá. Servery proto častěji slouží tam, kde je zapotřebí provádět nejen akce na přímou výzvu programu, ale i akce asynchronní, 'o vlastní vůli' - dobrým příkladem může být třeba správa souborů

(s asynchronní obsluhou cache pamětí) nebo inteligentní ovladač obrazovky (který požadovaný výstup dokresluje také asynchronně - viz odstavec 6.5.3.1).

9.2.4 Sdílené knihovny

Snad každého při čtení odstavce 9.2.2 napadne, že udržování tolika kopii jediného kusu kódu je nešikovné a že by to snad mělo jít vyřešit nějak lépe. Samozřejmě to vyřešit jde; řešením jsou právě sdílené knihovny. Jejich funkce je podobná jako funkce klasických knihoven, právě jen s jediným podstatným rozdílem: není zapotřebí, aby na disku nebo v paměti byly více než jednou (máme-li k dispozici systém virtuální paměti, stačí jediná kopie pro disk i pro paměť dohromady).

Celá 'finta' spočívá v tom, že spojovací program svou práci 'nedodělá' a ponechá to na zavaděči, který ukládá programy při spuštění do operační paměti. Celý mechanismus pak vypadá přibližně takto:

- Nejprve se pomocí překladačů vytvoří jednotlivé moduly, naprostě stejně jako tomu bylo v případě knihoven klasických.
- Druhým krokem je opět spojení všech modulů dohromady. Spojovací program však tentokrát vyřeší pouze vzájemné odkazy mezi moduly, které vytvořil programátor aplikace¹¹⁵; sdílených knihoven si nevšímá a odkazy na služby, které jsou na sdílených knihovnách uloženy, ponechá beze změny.
- Teprve při zavádění programu do paměti zavaděč zjistí, které ze sdílených knihoven jsou zapotřebí. Pak ověří, nejsou-li již tyto knihovny v paměti (jako důsledek zavedení některého z minulých programů); jestliže tomu tak není, do paměti je zavede.

Potom teprve zavaděč uloží do paměti i kód spouštěného programu; přitom jeho instrukce pro volání podprogramů ze sdílených knihoven

¹¹⁵A samozřejmě k nim připojí případné moduly z klasických knihoven, jsou-li v systému využity.

doplňí správnými adresami, odvozenými od umístění sdílené knihovny v paměti.

Z využití sdílených knihoven vyplývá navíc jedna podstatná výhoda, která přináší do systémů se sdílenými knihovnami určitý rys systémů objektových: jestliže totiž nahradíme starou verzi operačního systému verzí novější, budou všechny programy - i ty, které byly dohotoveny v době, kdy se nikomu ještě o nové verzi systému nesnilo - automaticky při zavádění spojovány s novými knihovnami a budou tedy automaticky využívat jejich výhod (odstraněných chyb, doplněných nových funkcí a podobně)¹¹⁶.

Pro jejich výhody jsou sdílené knihovny součástí všech modernějších operačních systémů (jako je EPOC, NeXTStep a dokonce i třeba MS Windows), snad s jedinou výjimkou operačního systému počítačů Apple Macintosh (zde však existují tzv. packages, které mohou sdílené knihovny do jisté míry nahradit).

9.2.5 Objekty

Objekt je z hlediska pravého objektového systému jakási 'černá skříňka'. Objektu můžeme poslat nějakou zprávu a můžeme si počkat na odpověď. Objekt může také na základě zprávy začít zpracovávat nějakou časově náročnější akci a její ukončení může naopak zprávou ohlásit nám ('my' - tedy hlavní program - jsme samozřejmě také objektem).

Implementace některého z objektově orientovaných programovacích jazyků existují dnes snad již pro každý operační systém. To však je do jisté míry 'podvod' - z hlediska programátora totiž překladač převede plnoprávné objekty na naprostě obvyklý kód využívající běžné volání podprogramů.

Naprosto jiná je situace v pravých objektových operačních systémech, jako je např. EPOC nebo NeXTStep. Tam objekt zůstane objektem - tedy nezávislou 'černou skříňkou' - i po překladu a operační systém sám nabízí pohodlné a

¹¹⁶Tento rys se nazývá **pozdní** nebo také **dodatečná vazba (late binding)** a patří mezi typické vlastnosti pravých objektových systémů.

Operační systémy

efektivní prostředky pro komunikaci mezi nimi. Pak je ovšem snadné využít právě těchto prostředků pro volání služeb; odpovídající kód je ukryt v sadě hotových objektů, dodávaných s operačním systémem.

Toto řešení je zhruba na půli cesty mezi sdílenými knihovnami a servery - objekt není (respektive nemusí být) samostatným procesem jako server; je však daleko samostatnější než podprogram, uložený na sdílené knihovně. Z hlediska samotného volání služeb zde žádné výrazné rozdíly nejsou¹¹⁷: objekty se podobají spíše knihovnám, sám aparát komunikace s nimi je podobnější komunikaci se servery.

9.3 Obsah služeb

Zatímco obsah některých skupin služeb (jmenníme např. služby pro neceločíselnou aritmetiku) je celkem jasný, k některým dalším oblastem je vhodné uvést ještě podrobnější rozbor. Podíváme se proto postupně na textové služby, na podporu práce v národním prostředí, na databázový systém a na komunikaci na vyšší úrovni.

9.3.1 Textové služby

Textové služby můžeme velmi snadno rozdělit do dvou skupin. V první z nich jsou jednoduché služby pro nejrůznější přesuny dat v operační paměti; na nich není dohromady nic zajímavého (snad jen může být vhodné si uvědomit, že se nejedná jen o pohodlí programátora - operační systém může často přesuny větších bloků dat zajistit rychleji než aplikační program, protože má k dispozici speciální vybavení - může např. využít blitter, může selektivně vypínat cache paměť a podobně).

¹¹⁷Hlavní výhodou objektů proti ostatním metodám je dědičnost, která programátorům umožňuje velmi pohodlným a intuitivním způsobem mírně modifikovat činnost služeb, které využívají.

Druhá skupina je komplikovanější: jedná se o zpracování formátovaných textů zahrnujících různé fonty, velikosti písma, barvy a podobně. Je nutné, aby sám operační systém práci s formátovaným textem podporoval, a to ze dvou důvodů:

- O prvním z nich jsme se již zmínili. Práce s formátovaným textem je příliš častým případem, než aby se vyplatilo jí programovat pokaždé znovu.
- Druhým a hlavním důvodem je komunikace programů, kterou se budeme zabývat podrobněji v odstavci 9.3.4. Jde o to, aby bylo možné bez problémů přenášet formátovaný text mezi dvěma aplikacemi, jejichž tvůrci se na ničem nedomluvili.

Služby pro práci s formátovaným textem (i některé služby pro práci s textem neformátovaným) navíc velmi úzce souvisejí s podporou národního prostředí, které věnujeme příští odstavec. Součástí mnoha služeb je totiž také interpretace jednotlivých znaků (např. při převodu malých písmen na velká nebo při dělení odstavce na řádky). Operační systém musí v takovém případě korektně bez extrémního úsilí programátora konkrétní aplikace zajistit správnou interpretaci národních znaků.

93.2 Národní prostředí

I podpora národního prostředí se dá snadno a nenásilně rozdělit na dvě skupiny služeb. První skupina umožňuje uživateli počítače zpracovávat dokumenty v různých jazycích; druhá skupina služeb pak je určena k tomu, aby uživatel mohl s počítačem komunikovat ve své rodné řeči.

Obě skupiny jsou velmi komplexní; mezi hlavní součásti prvej skupiny patří:

- Možnost zobrazovat speciální znaky, potřebné pro požadovaný jazyk, na obrazovce počítače. Český uživatel dobře ví, že to není zcela triviální ani s našimi diakritickými znaménky; skutečné problémy však nastávají s východními abecedami, jejichž znaky jsou velmi komplikované a bývá jich značný počet (např. čínské a japonské abecedy).

Situace je ještě o to komplikovanější, že některé východní jazyky mají pro jediné písmeno různé 'obrázky' v závislosti na kontextu.

- Neméně důležitá je i možnost přenést texty v požadovaném jazyce na všechna potřebná výstupní zařízení - může se jednat o tiskárnu, fax, osvítovou jednotku nebo třeba stroj na výrobu kreditních karet¹¹⁸.

Zde je velkou výhodou grafický subsystém se službami vysoké úrovni, který umožňuje využití jediného mechanismu zobrazování pro kterékoli výstupní zařízení (jaký je součástí NeXTstepu nebo v menší míře operačního systému počítačů Macintosh). Pak stačí vyřešit zobrazování národních znaků jen jednou a není nutné se jím zabývat pro každé nové výstupní zařízení vždy znova a znova.

- Nemalým problémem je i vhodné ovládání klávesnice. Řada jazyků má příliš mnoho znaků, než aby bylo možné je rozumným způsobem poskládat na běžnou počítačovou klávesnici. Je proto nutné zavádět tzv. mrtvé klávesy, různé pracovní režimy a další speciální prostředky pro vstup znaků.

Řada firem, nabízejících 'české prostředí pro počítače IBM PC', se domnívá, že popsanými třemi body seznam potřebných služeb končí. Pravý opak je pravdou; rozumná podpora národních prostředí musí zajišťovat také:

- Klasifikaci a převody znaků. Mnoho programátorů v jazyce C např. používá často standardní službu 'isupper', která ověří, je-li zadaný znak velkým písmenem; málokdo si však přitom uvědomí, že pokud tato služba nepozná také velká písmena 'Á', 'Č', 'Ď', ..., není její implementace korektní.

Totéž platí pro převody malých a velkých písmen. Standardní systémové služby musí převádět korektně všechny znaky, včetně národních, a musí být automaticky přístupné všem programátorům

¹¹⁸Mimořádem, spořitelní kreditní karty a výpisy kont bez diakritických znamének jsou dobrým dokladem toho, že tento problém není dosud ani zdaleka všechna vyřešen.

prostřednictvím standardních funkcí (jakými jsou např. funkce 'toupper' nebo 'tolower' v jazyce C).

- Řadu stále složitějších a složitějších problémů vyvolává tak základní úkol, jakým je setřídění několika slov nebo vět. Operační systém samozřejmě musí třídění zajišťovat sám v závislosti na aktivním jazyce; při implementaci třídicích rutin je však zapotřebí:
 - * Nejprve zajistit, aby vůbec všechna písmena měla správné pořadí - pro češtinu tedy např. musí být 'á' před 'b'. ačkoli má ve všech běžně užívaných kódováních vyšší kód.
 - * Pak narazíme na písmena, složená z dvojice znaků, která se však z hlediska třídění chovají skutečně jako jediné písmeno (například v češtině máme 'ch').
 - * Po vyřešení obou problémů zjistíme, že třídění v některých jazycích - mezi které čeština patří - dosud není korektní. Správné třídění v nich totiž z jakýchsi prapodivných a nepochopitelných důvodů není lexikografické, ale daleko složitější - např. správné pořadí českých slov 'Děkan', 'Dentista' a 'Děvín' je to, ve kterém jsme je uvedli, ačkoli první a třetí slovo začínají stejně.
 - * Vyřešíme-li přece jen všechny problémy a implementujeme korektní třídění, musíme vyřešit také problém, kde jej použít a kde ne. Má být např. seznam souborů setříděn lexikograficky nebo podle aktivního jazyka?
- Součástí podpory národního prostředí je i 'překlad' automaticky generovaných údajů (jako jsou data, jména dní a měsíců apod.). Často je překlad velmi obtížný, protože v některých jazycích - mezi nimiž samozřejmě čeština patří na čestné místo - se slova ohýbají, takže jedno slovo může v různých kontextech vypadat jinak.
- Komplexní textové služby na národním prostřední závisejí také velmi podstatným způsobem - uvědomme si například, že nemalé množství jazyků standardně píše zprava doleva a některé píší dokonce ve

sloupcích¹¹⁹. Služby operačního systému by samozřejmě měly podporovat i toto.

- Nikoli nepodstatnými službami systému je i podpora pro dělení slov na konci řádků a korektor (spellchecker).

Systém podpory práce v nejrůznějších národních prostředích je dnes pravděpodobně nejlépe vyřešen v operačním systému počítačů Apple Macintosh, jejichž systém tzv. scriptů pokrývá všechny popsané problémy, vyjma dvou posledních (systém 7 dokonce do jisté míry řeší i ohýbání slov, o kterém jsme se zmínili ve třetím bodě od konce).

Druhou skupinou služeb, které musí operační systém zajistit, je podpora komunikace s uživatelem ve zvoleném jazyce (uvědomme si, v čem je rozdíl: počítač Apple Macintosh s anglickým operačním systémem umožní zpracovávat texty třeba ve svahilštině - máme-li k dispozici patřičný script - ale s uživatelem přitom bude komunikovat výhradně anglicky).

Zásadním problémem je pochopitelně vlastní překlad textů. Přitom však je nutné si uvědomit, že je často zapotřebí 'přeložit' i obrázky, které mohou mít výrazně odlišný význam v jiné kulturní oblasti. Černobílé zobrazená pěticípá hvězda např. v českém uživateli vyvolá zcela jiné asociace a představy než v Američanovi; jiným příkladem je třeba číslo 69, které je u nás prakticky bez emočního náboje, zatímco v USA je považováno za téměř vulgární.

To jsou však všechno problémy, které musí řešit a vyřešit ten, kdo bude připravovat operační systém a jeho obslužné programy pro práci v některém konkrétním jazyce. Sám operační systém má jiný úkol - musí obsahovat podporu pro práci vícejazyčných aplikací.

Vícejazyčná aplikace je program, který obsahuje komunikační prvky pro několik různých jazyků. Teprve ve chvíli spuštění se ve spolupráci s operačním systémem automaticky zvolí jeden z těchto jazyků - ten, který uživateli nejlépe

¹¹⁹Doufejme, že nikdo nebude vytvářet operační systém schopný zpracovávat indiánské obrázkové písmo, kterým se standardně píše do spirály.

vyhovuje - a kdykoli pak program použije některý z komunikačních prvků, zajistí systémové služby automaticky použití prvku z tohoto jazyka.

Tvorba vícejazyčných aplikací je samozřejmě v zásadě možná pod libovolným operačním systémem, který vůbec aplikaci umožní zjistit, jaký jazyk je právě 'platný'. Skutečnou podporu, kdy by operační systém převzal většinu úkolů spojených s vícejazyčnými aplikacemi od programátora na sebe, však nalezneme zatím asi pouze v NeXTstepu a v daleko menší míře také v EPOCu.

9.3.3 Databáze

Přístup k datovým souborům na úrovni binárního nebo textového souboru se sekvenčním nebo přímým přístupem, jakým disponuje naprostá většina dnešních operačních systémů, je samozřejmě věc nutná a potřebná, ale zdaleka ne postačující.

Moderní operační systém musí zajišťovat také služby vyšší úrovně pro práci s formátovanými soubory, obsahujícími databáze s proměnnou délkou záznamu, se sekvenční nebo s indexsekvenční organizací. Důvody pro tento požadavek jsou dva:

- Tento typ souborů může s výhodou využít naprostá většina aplikačních programů. Je proto zbytečné, aby každý programátor vytvářel vlastní databázový systém - zbytečně by tak utrácel síly, které může v kvalitnějším operačním systému věnovat na řešení konkrétního problému.
- I v případě, že existuje řada nejrůznějších databázových systémů, které pracují pod daným operačním systémem a nabízejí programátorům své služby, narazíme na jeden zásadní problém: databáze vytvořené pod různými systémy budou vzájemně nekompatibilní a čas, který jejich programátoři ušetří na vlastní tvorbě databázového systému opět ztratí při programování nejrůznějších převodníků z jednoho systému do druhého.

U operačních systémů určených pro výkonnější pracovní stanice samozřejmě z analogických důvodů nestačí pouhá podpora indexekvenčních souborů a bylo by vhodné, aby operační systém obsahoval plnohodnotný databázový server, schopný obsluhovat rozsáhlé a komplikované databáze a schopný nabízet své služby po síti (a - alespoň pokud se neobjeví dokonalejší systém - podporující jazyk SQL).

Takový server je často velmi komplikovaný a drahý program a jeho zařazení do systému by neúměrně zvýšilo cenu celého operačního systému pro tu skupinu uživatelů, kteří jeho služby nepotřebují. Dosud nejmodernější operační systém NeXTStep proto tento problém řeší tak, že obsahuje luxusní služby pro zcela standardizované využití kteréhokoli z komerčně přístupných databázových serverů; jestliže si tedy uživatel server - a to jakýkoli - pořídí, stanou se služby serveru stejně snadno přístupné všem aplikacím a programátorům, jako by byl server přímo součástí operačního systému.

9.3.4 Komunikace programů

Komunikace programů na této úrovni je něco zcela jiného než komunikace procesů, se kterou jsme se seznámili v odstavci 5.4. Hlavním úkolem komunikace procesů bylo zajistit jejich synchronizaci. Tentokrát se o synchronizaci nezajímáme (přesněji řečeno, musí být zajištěna v nižších vrstvách tak, aby fungovala a my abychom se o ni nemuseli starat); naším úkolem je zajistit spolupráci několika programů nad společnými daty.

Pro nebohé uživatele operačních systémů typu MS DOS, kde žádné podobné služby nejsou k dispozici, uvedeme nejprve několik příkladů:

- Čteme například nějakou zajímavou zprávu v elektronické poště a rádi abychom z ní některé úryvky přebrali do článku, který máme rozepsaný v nějakém textovém editoru. Tuto úlohu lze samozřejmě vyřešit uložením zprávy do souboru, načtením tohoto souboru do textového editoru a vybráním požadovaných úseků; to však je velmi nepohodlné.

Pokud abychom četli tutéž zprávu např. na Macintoshi, můžeme každý zajímavý úsek přímo v elektronické poště označit, jedním příkazem

přenést do systémové komunikační oblasti (tzv. schránky), pak aktivovat editor a dalším příkazem text ze schránky vložit na libovolné místo v rozepsaném článku.

- V další zprávě můžeme narazit na jméno, které je nám povědomé ale nemůžeme si jej vybavit. Zkusíme jej tedy vyhledat v databázi svých známých, ale ouha - pro vyhledávání musíme jméno opsat ručně (s rizikem případného překlepu).

Na Macintoshi jméno opět jediným příkazem uložíme do schránky, a v databázi pohodlně obsahem schránky odpovíme na otázku 'co se má hledat' (pod operačním systémem NeXTStep je to ještě jednodušší - stačí přímo vyvolat příkaz 'Hledat v osobní databázi').

- Vraťme se k prvnímu příkladu. Tentokrát však čteme formátovaný text s různými fonty, velikostmi a typy písma, a rádi bychom jej přenesli do několika dalších dokumentů. Některé z nich však jsou formátované - tam bychom rádi formátování zachovali - a jiné obsahují jen ASCII text; tam chceme přenést alespoň text bez formátu.

Narazíme-li na tento problém na počítači PSION s operačním systémem EPOC, nebudeme dlouze hledat řešení: požadovaný text pouze označíme a v každém programu, do kterého jej chceme přenést, vyvoláme službu 'Bring'. Program si pak (prostřednictvím služby operačního systému) sám vyžádá text v takovém formátu, jaký dokáže zpracovávat.

Je nutné si uvědomit, že komunikaci skutečně musí zajišťovat sám operační systém. Bylo by totiž sice poměrně jednoduché kterýkoli z dále popsaných mechanismů implementovat bez podpory systému; problém je ale v tom, že by fungoval pouze mezi několika málo programy. Mají-li být systémy komunikace uživateli počítač skutečnou pomůckou, musí korektně pracovat s naprostou většinou aplikací.

Ukažme si některé systémy komunikace mezi programy, které jsou dnes k dispozici u většiny operačních systémů, ajako ukázku budoucího trendu i dva 'objektové' komunikační modely NeXTstepu.

Operační systémy

9.3.4.1 Schránka

Mechanismus schránky, jehož princip jsme popsali hned v prvním příkladu, je vlastně stařičký - byl zahrnut již v operačním systému prvních počítačů Apple Macintosh a dnes jej podporuje téměř každý.

Rozumně implementovaná schránka by navíc měla být schopna obsahovat zároveň data v několika formátech (u počítačů Macintosh tomu tak samozřejmě je). To je proto, že ve chvíli, kdy ukládáme data do schránky, není jasné, kdo je ze schránky bude odebírat. Ukládáme-li tedy např. obrázek v kreslicím programu, měl by program do schránky uložit obrázek dvakrát: jednou ve vektorovém formátu, zachovávajícím všechny informace o struktuře obrázku a umožňujícím jej dále upravovat, a jednou jako bitovou mapu ('snímek obrazovky'). Bude-li později chtít údaje ze schránky převzít program, který nedokáže komplikovanější vektorový formát zpracovávat, bude mít k dispozici alespoň bitovou mapu.

Z toho ovšem plyne i určitá nevýhoda schránky: poměrně snadno se může stát, že některý program data ze schránky nebude schopen převzít, ačkoli v principu by je zpracovat mohl. Jestliže třeba náš kreslicí program do schránky uloží obrázek ve vektorovém formátu s barevnou informací a zároveň jako bitovou mapu, nebudeme moci obsah schránky použít v černobílém vektorovém kreslicím programu.

Tento problém by se dal řešit (nebo spíše minimalizovat) tak, že by každý program do schránky ukládal data ve všech možných formátech. To však přináší jinou nevýhodu: taková schránka by zabrala neúměrně mnoho paměti, a přenos dat do ní by trval velmi dlouho.

Bylo by zřejmě daleko výhodnější, kdyby se oba programy domluvily na tom, v jakém formátu se vlastně mají data předávat. Takový systém opravdu existuje a podíváme se na něj v následujícím odstavci.

9.3.4.2 Inteligentní schránka

Schránka operačních systémů EPOC a NeXTStep je implementována trochu jiným způsobem¹²⁰, i když z pohledu uživatele funguje naprosto stejně. Program, ze kterého chceme data přenést, ve skutečnosti do schránky žádná data neukládá (a odpovídající příkaz díky tomu také trvá mnohem kratší dobu). Namísto toho však do schránky uloží informaci o všech formátech, ve kterých je schopen data exportovat.

Teprve ve chvíli, kdy se pokusíme přebrat obsah schránky v 'cílovém' programu, dojde k vlastnímu přenosu. 'Cílový' program nejprve zjistí, zda dokáže zpracovat data v některém z formátů zapsaných ve schránce. Jestliže ano, vyžádá si prostřednictvím operačního systému od 'zdrojového' programu data právě v tom formátu, na kterém se oba shodli.

Tento mechanismus přináší dvě výhody:

- Data se přenášejí pouze v jediném formátu, ne ve všech potenciálně potřebných; tak se šetří operační paměť i čas.
- Díky tomu si programy mohou dovolit nabízet daleko více datových formátů; je tedy mnohem větší pravděpodobnost, že se na některém z nich dva programy 'shodnou'.

Inteligentní schránka má nicméně i jednu nevýhodu, kterou pozorný čtenář již jistě odhalil sám: ve chvíli přenosu musí oba programy běžet. Klasická schránka naproti tomu může obsahovat data uložená některým programem ještě dávno poté, co byl program ukončen. Proto může inteligentní schránka snadno fungovat u preemptivních operačních systémů jako je NeXTStep nebo EPOC, kde současný běh řady procesů nepřináší žádné problémy; proto by mohla být jen s obtížemi implementována na Macintoshi, jehož kooperativní multitasking je uživateli na obtíž už když na pozadí běží jeden jediný proces.

¹²⁰Oba systémy ale mají i 'klasickou' schránku podle odstavce 9.3.4.1, která se použije tam, kde to je výhodnější.

9.3.4.3 Spojení dat

Schránka - ať již klasická nebo inteligentní - je velmi pohodlná, dokud přenášíme hotové údaje z jednoho místa na druhé. Jakmile však pracujeme paralelně na dvou věcech (např. na textu a na obrázku k němu), které budou tvořit společný výsledek, je neustálé přenášení nových verzí únavné. Navíc se nám může stát, že nejnovější verzi přenést prostě zapomeneme a celý výsledek bude chybný.

Moderní operační systémy pro takový případ zavádějí tzv. **spojení dat**¹²¹. Jeho mechanismus je podobný jako mechanismus inteligentní schránky; spojení mezi oběma programy však není přerušeno ani po přenosu dat. 'Zdrojový' program pak po každé změně předaných dat o změně informuje 'cílový' program, takže ten může zcela automaticky změnu reflektovat.

V praxi to funguje tak, že na začátku práce provedeme spojení dat, a pak již zcela libovolně pracujeme na obou částech projektu (nebo na všech, je-li jich více než dvě). Operační systém se přitom zcela transparentně postará o to, aby všechna 'spojená' data byla na správném místě vždy v té nejnovější verzi.

9.3.4.4 'Drag and drop'

Systém 'drag and drop' (asi nejvýstižnější překlad 'táhni a vhod' nezní příliš dobře, přidržíme se proto anglického termínu) operačního systému NeXTStep je vlastně velmi příjemným grafickým uživatelským rozhraním nad (trochu rozšířeným) mechanismem inteligentní schránky.

Jedná se o to, že uživatel může pomocí myši 'uchopit' téměř libovolný objekt v kterémkoli okně a 'přetáhnout' jej jinam - třeba do okna úplně jiného programu. Tam pak objekt 'pustí'. Operační systém se přitom postará o to, aby se data, která objekt reprezentoval, přenesla do patřičného programu a na správné místo.

¹²¹"Paste & Link" v NeXTstepu, 'publisher / subscriber' v operačním systému 7 pro Apple Macintosh.

Čtěme-li tedy např. v systému elektronické pošty na NeXTu zprávu a líbí se nám v ní nějaký obrázek (elektronická pošta systému NeXTStep může přenášet plně formátované dokumenty obsahující obrázky i ledacos jiného), nemusíme se starat o schránku: obrázek prostě myší 'přetáhneme' do článku, který máme rozepsaný v textovém editoru, a je hotovo.

93.4.5 Datové služby

Mechanismus datových služeb je alespoň zatím také implementován pouze v NeXTstepu¹²². Jedná se o logické rozšíření mechanismu inteligentní schránky na poskytování libovolných služeb.

Při využívání datových služeb spolu kooperují dva programy. Jeden z nich nabízí data pomocí již známého mechanismu inteligentní schránky; tomuto programu budeme říkat klient. Druhý program - budeme mu říkat datový server - nabízí služby; každá služba může pracovat nad daty určitého typu (nebo určitých typů) a je identifikována jménem.

Jestliže si uživatel, který pracuje s klientem, vyžádá datové služby, spojí si operační systém automaticky typy dat, které klient nabízí, a datové služby, které nabízejí všechny datové servery. Uživateli pak nabídne seznam jmen všech služeb, které mohou zpracovávat některý z nabízených typů dat.

Jestliže uživatel zvolí některou ze služeb, aktivuje operační systém patřičný datový server a pomocí mechanismu inteligentní schránky mu předá klientova data, takže server je může v rámci služby téměř libovolným způsobem zpracovat.

Libovolný program se může snadno stát datovým serverem; existuje proto obrovské množství datových služeb - od vyhledání slova ve výkladovém slovníku nebo ve slovníku synonym přes zařazení textu do osobního zápisníku až po

¹²²Implementaci tohoto mechanismu stejně jako systému 'drag and drop' totiž výrazně usnadňuje objektové prostředí. Jiný operační systém než NeXTStep by proto implementaci obdobných služeb umožnil jen za cenu neúměrného programátorského úsilí.

Operační systémy

automatický překlad obrázku (např. přijatého faxu) na text datovým serverem, který má schopnost OCR.

Díl druhý

Grafické nadstavby a rozhraní

"Jediný obrázek často řekne víc než několik stran textu."

Dávno známá pravda

"What you see is what you get" - "(na obrazovce) vidíš přesně to, co dostaneš (povytiskení)"

Snad nejčastěji slibovaná a právě tak často nesplněná služba

Operační systémy

10. Nástup grafiky

V osmdesátých letech proběhla na trhu stolních počítačů skutečná revoluce. Její bezprostřední příčinou byl rozvoj programů umožňujících připravovat na takových systémech jednodušší a později již jakékoli publikace - noviny, letáky, časopisy, knihy. Takovým systémům se začalo říkat DTP (DeskTop Publishing). Bezpodmínečnou nutností pro práci takových systémů totiž byla možnost grafického výstupu na obrazovku počítače.

Do té doby - až na čestnou výjimku systémů Apple Macintosh - převládaly počítače orientované na neformátovanou textovou komunikaci s uživatelem, stejně jako tomu bylo u terminálů sálových počítačů před mnoha lety. Uživatelé počítačů s operačním systémem MS DOS nebo implementací 'čistého' UNIXu bez některé z rozšířených grafických nadstaveb ostatně důsledky zmíněného stavu pocitují dodnes.

Zanedlouho poté, co si uživatelé zvykli na pohodlí grafických rozhraní DTP programů, začali obdobně intuitivní a pohodlné rozhraní požadovat i v ostatních aplikacích i ve vlastním operačním systému. To přinutilo snad všechny tvůrce operačních systémů (ano, dokonce i firmu Microsoft) zahrnout do svých projektů také grafický subsystém. Jeho jednotlivým částem a jeho potřebným vlastnostem se budeme v tomto dílu věnovat.

Na grafický subsystém se snadno můžeme dívat jako na nadstavbu operačního systému (v tom smyslu, v jakém jsme si operační systém popsali v prvním dílu). S výjimkou ovladačů klávesnice, obrazovky a myši totiž operační systém vůbec žádným způsobem s grafickým systémem nesouvisí; grafický systém naopak sám používá služeb operačního systému a často není z jeho hlediska ničím jiným než jedním z mnoha procesů. V některých případech je tato struktura zcela zřetelná (např. ATARI TOS), existují i samostatné grafické systémy zpracované tak, aby bylo možné je relativně snadno přizpůsobit kterémukoli operačnímu systému (např. systém GEM firmy Digital Research).

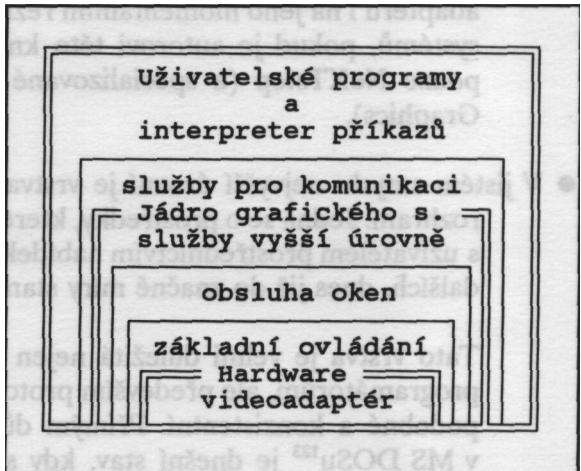
Grafickým systémům se budeme věnovat daleko méně podrobně, než jak jsme se v prvním dílu knihy zabývali jádrem operačního systému. Důvodů pro to je více:

Operační systémy

- Zatímco jádro operačního systému se obvykle skládá z relativně malých, ale 'chytrých' služeb, bývá implementace grafického systému složena ze sice také 'chytrých', ale bohužel velmi rozsáhlých modulů. Dokumentovat popisovanou problematiku téměř životaschopnými zdrojovými programy proto v tomto případě není dost dobře možné.
- Hlavním tématem této knihy je přece jen operační systém jako takový. Jeho nadstavbám - mezi které grafický subsystém nepochybně patří - se věnujeme spíše pro úplnost; pokud bychom je měli popisovat stejně podrobně jako jádro, byl by celkový rozsah této knihy neúnosný.

11. Struktura grafického systému

Jen u skutečně nejjednodušších systémů může být celý grafický subsystém jediným nedílným celkem. Jinak je zapotřebí jej rozdělit do několika vrstev, z nichž každá zajišťuje vlastní skupinu úkolů a slouží vrstvě vyšší (vzpomeňme si na obecnou vrstvenou strukturu operačního systému, popsanou v odstavci 3.4 - na obr. 22 vidíme podobný pohled, avšak jednotlivé vrstvy tentokrát odpovídají samostatným částem grafického subsystému).



obr. 22: vrstvený grafický systém

- Na nejnižší úrovni musí být jednoduchý systém základních grafických služeb umožňujících vlastní zápis základních grafických objektů na obrazovku. Je-li počítač osazen kvalitním grafickým procesorem, nemusí být tento systém vůbec zapotřebí.
- V další vrstvě musí stát systém zajišťující práci s obrazovkovými okny a/nebo s virtuálními obrazovkami. V závislosti na konkrétním návrhu grafického systému mohou být tyto dvě nejnižší vrstvy navzájem 'prohozeny'.

Na vrstvě obsluhující okna každopádně leží zodpovědnost za korektní spolupráci s interaktivními vstupními zařízeními (jako je klávesnice nebo myš) i se samotnými procesy - ty totiž někdy potřebují vědět, v jakém stavu jsou právě jejich okna.

- Na další úrovni je velmi vhodné implementovat vrstvu vyšších služeb grafického systému, o které jsme se zmínili již v odstavci 6.5.3.1. Tato

Operační systémy

vrstva umožní programátorům aplikací pracovat skutečně s grafickými objekty (jako je čára, čtverec, kruh, plocha nebo třeba koule), a ne s nějakými obrazovými body, jejichž počet i barva závisí na grafickém adaptéru i na jeho momentálním režimu práce. Z dnešních operačních systémů, pokud je autorovi této knihy známo, tuto vrstvu obsahuje pouze NeXTStep (a specializované animační systémy firmy Silicon Graphics).

- V jistém smyslu nejvyšší úrovní je vrstva služeb uživatelského grafického rozhraní. Jedná se o prostředky, které programům usnadní komunikaci s uživatelem prostřednictvím nabídek (menu), dialogových oken a řady dalších, dnes již do značné míry standardizovaných, prvků.

Tato vrstva je velmi důležitá nejen pro usnadnění práce aplikačním programátorům, ale především proto, aby bylo ovládání všech aplikací podobné a konzistentní. Přímým důsledkem neexistence této vrstvy v MS DOSu¹²³ je dnešní stav, kdy se každý program ovládá jinak a nebohý uživatel, který musí střídavě pracovat třeba s nějakým programem firmy Borland, s nějakým programem od pana Nortona, s WordPerfectem a - nejhorší nakonec - třeba s editorem T602, z toho přinejmenším zešediví, uchrání-li se horších důsledků.

- Ačkoli jsme minulou vrstvu nazvali nejvyšší, zmíníme se ještě o jedné - je jí grafický interpret příkazů uživatele, který v grafických systémech stojí na místě 'shellu' systémů orientovaných textově.

Na kvalitě a ergonomii interpretu příkazů totiž do značné míry záleží, bude-li se uživatelům se systémem pracovat pohodlně a dosáhnou-li snadno vysoké efektivity práce (jako je tomu u grafických systémů Apple Macintosh nebo NeXTStep), nebo bude-li tomu právě naopak (jako tomu je u grafického systému MS Windows, minimálně do verze 3.0).

V následujících kapitolách se postupně podíváme na zajímavější vrstvy grafického subsystému, ukážeme si jejich implementaci v některých

¹²³Kde ovšem nenalezneme ani vrstvy nižší.

Struktura grafického systému

konkrétních operačních systémů (nejčastěji budeme využívat za příklad Apple Macintosh, EPOC a NeXTStep) a pokusíme se zjistit, v čem jsou jejich nedostatky i jejich výhody.

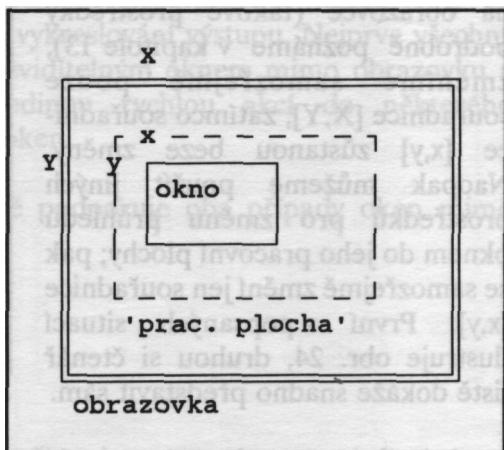
Operační systémy

12. Okna

Okna jsou dnes nejrozšířenějším a nejúspěšnějším mechanismem pro virtualizaci obrazovky. Existuje řada mechanismů; podíváme se nejprve na obecný základ, a pak se zmíníme i o některých konkrétních implementacích.

12.1 Co je to okno

Okno je vlastně samostatné grafické výstupní zařízení. Z hlediska programu je okno obvykle pravoúhelníkem určitých rozměrů (s některými dalšími atributy), do kterého lze zapisovat libovolné grafické informace. Uživatel pak - víceméně nezávisle na programu, který s oknem pracuje - určí postavení okna na obrazovce, jeho rozměry a viditelnost; operační systém sám se postará o správné zobrazení obsahu okna (určeného programem) na obrazovce.



obr. 23: okno na obrazovce

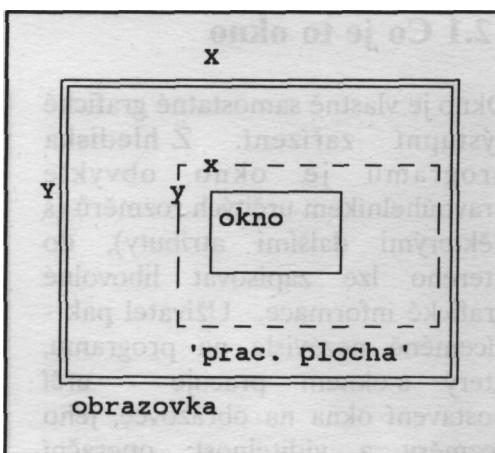
Nejčastěji program pracuje s plochou okna, která je vidět na obrazovce (nebo která by byla vidět, kdyby okno nebylo částečně zakryto jinými objekty - o tom se budeme bavit později). Má-li tedy okno sloužit jako 'průhled' na nějakou větší pracovní plochu - jak tomu také v naprosté většině případů bývá - musí se o to postarat programátor sám: v rámci programu je známa souřadnice okna uvnitř této pracovní plochy; na jejím základě program sám pozná, co (tedy kterou část plochy) má do okna kreslit.

Tuto situaci nám ozřejmí několik obrázků. Základem je obr. 23, který ukazuje celou situaci: dvojitý rámeček odpovídá celé obrazovce, plnou čarou je

Operační systémy

znázorněno okno a přerušovaná čára ukazuje pracovní plochu, jejíž část vidíme 'skrz' okno (není samozřejmě nikde psáno, že tato pracovní plocha musí být menší než obrazovka, jako tomu je na našem obrázku - pracovní plocha 'za oknem' může být samozřejmě zcela libovolně velká). Vzájemné postavení obrazovky a pracovní plochy je přitom určeno dvojicí souřadnic - první z nich, označená v obrázku jako $[X,Y]$, určuje polohu okna na obrazovce. Druhá souřadnice, označená jako $[x,y]$, naproti tomu udává relativní polohu okna vůči jeho pracovní ploše.

Jestliže nyní použijeme některý z prostředků pro změnu polohy okna na obrazovce (takové prostředky podrobně poznáme v kapitole 13), změníme samozřejmě pouze souřadnice $[X,Y]$, zatímco souřadnice $[x,y]$ zůstanou beze změny. Naopak můžeme použít jiných prostředků pro změnu průhledu oknem do jeho pracovní plochy; pak se samozřejmě změníjen souřadnice $[x,y]$. První z popsánych situací ilustruje obr. 24, druhou si čtenář jistě dokáže snadno představit sám.



obr. 24: totéž okno po přemístění

Popisovali jsme samozřejmě zcela obecný případ - existuje řada situací, kdy pracovní plocha okna přesně odpovídá oknu samotnému a na žádné straně jej nepřesahuje; souřadnice $[x,y]$ pak jsou ovšem neměnné a mají hodnotu $[0,0]$.

Na konec tohoto odstavce poznamenejme, že nejmodernější systémy (EPOC, NeXTStep) umožňují programátorovi pracovat přímo s pracovní plochou okna; zajištění přemísťování okna po této ploše a zobrazení správného průhledu je již pouze věcí operačního systému a programátor se o ně nemusí starat.

12.2 Okna mimo obrazovku?

Dosud jsme tiše předpokládali, že okna jsou součástí obrazovky. To však nemusí nutně být pravda; můžeme narazit na dva případy, které se tomuto pravidlu budou vymykat:

- Okno může být přemístěno (změnou souřadnic [X,Y] z minulého odstavce) tak, že jeho část nebo dokonce celé bude mimo obrazovku.
- Program si může vyžádat okno, které je principiálně mimo obrazovku - to se může hodit např. v případě, kdy programátor nechce, aby na obrazovce bylo vidět postupné vykreslování výstupu. Nejprve všechny potřebné akce provede nad neviditelným oknem mimo obrazovku a jeho obsah pak přenese jedinou rychlou akcí do některého z 'normálních' zobrazovaných oken.

Rozumný grafický systém samozřejmě podporuje oba případy oken mimo obrazovku transparentním způsobem.

12.3 Hierarchie oken

Moderní grafické systémy rozšiřují možnosti programátora zavedením tzv. **hierarchie oken**. To znamená, že kromě 'horizontálních' vztahů mezi jednotlivými okny, určených jejich vzájemnou polohou na obrazovce, existují i 'vertikální' vztahy, kdy jedno okno může být součástí jiného. To umožňuje poměrně pohodlným způsobem implementovat např. automatické přemístování okna vůči jeho pracovní ploše, o kterém jsme se zmínili již v odstavci 12.1, i další potřebné služby. Popišme si jako příklad např. hierarchii oken v operačním systému EPOC:

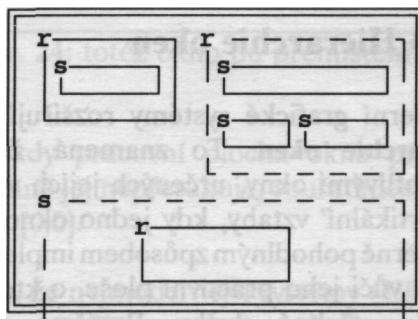
Okna jsou uložena ve stromové hierarchické struktuře, ve které má každé okno svého předchůdce (tzv. rodiče) a libovolný počet svých následníků (tzv. synů). Operační systém pak automaticky zajišťuje následující mechanismy:

Operační systémy

- Souřadnice 'syna' na obrazovce jsou relativní vůči souřadnicím jeho 'rodiče'. Díky tomu je velmi snadné vytvářet skupiny oken, které lze pohodlně přemísťovat jako jediný celek - stačí, jsou-li všechna tato okna syny jediného rodiče a přemísťujeme-li právě tohoto rodiče. Relativní souřadnice zajistí, že všichni 'synové' se přemísťují zároveň s ním.
- Má-li jak rodičovské, tak synovské okno nějaký vlastní obsah, překrývá obsah 'syna' na obrazovce obsah 'rodiče'. Samozřejmost tohoto požadavku snad nepotřebuje další vysvětlení.
- Obsah synovského okna není nikdy zobrazován mimo souřadnice rodiče. To na první pohled vypadá 'podezřele', dává to však velmi dobrý smysl - právě tato služba totiž umožňuje pohodlně zajistit automatické zobrazování 'průhledu' do pracovaní plochy: 'rodič' je vlastní okno, zatímco 'syn' reprezentuje právě pracovní plochu. Na obrazovce samozřejmě vidíme jen tu část pracovní plochy, která je uvnitř souřadnic 'rodiče'; pouhou změnou souřadnic syna (které nejsou ničím jiným než souřadnicemi [-x,-y] z odstavce 12.1) zajistíme změnu průhledu.

Na obr. 25 si můžeme prohlédnout nejčastější případy využití hierarchie oken. Rodičovské okno je v levém horním rohu vždy označeno písmenem 'r', synovské písmenem 's'; okno znázorněné plnou čarou je přímo viditelné na obrazovce, okno znázorněné přerušovanou z různých důvodů viditelné není.

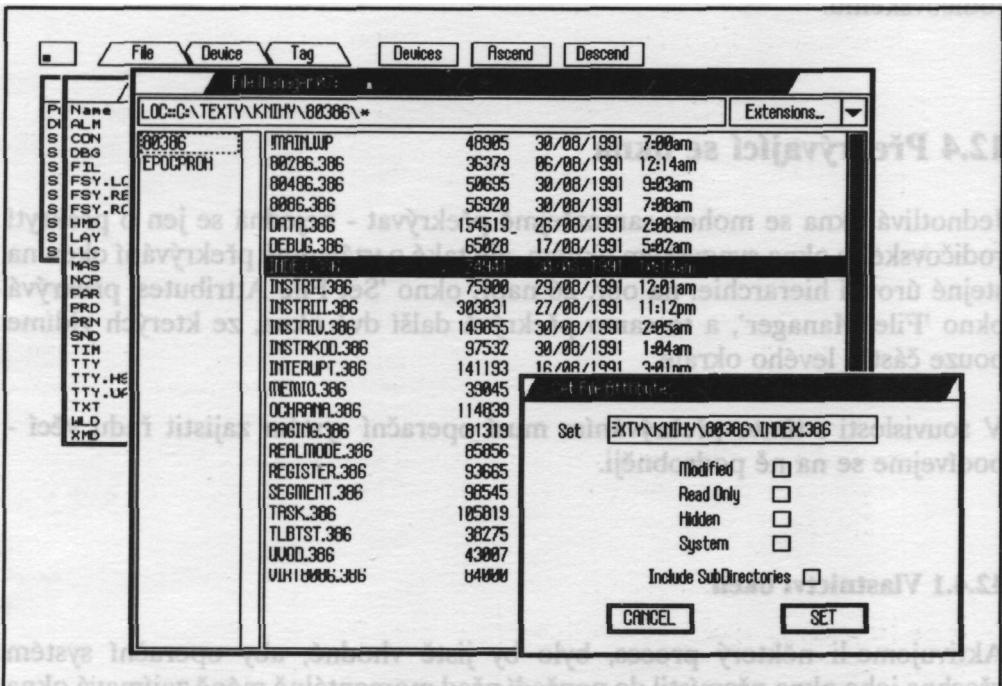
První příklad v levém horním rohu obrazovky je nepochybně nejčastější - rodičovské okno obsahuje rámeček, název a další pomocné prvky, zatímco v synovském okně je uložen vlastní obsah okna. Z hlediska uživatele počítače se obě okna jeví jako okno jediné, obsahující řadu ovládacích prvků rámujících vlastní obsah - několik takových oken EPOCu



obr. 25: hierarchie oken

Okna

ukazuje obr. 26: každé okno je orámováno, má záhlaví s názvem okna a v pravé části okna 'File Manager' vidíme i pruh posuvníku.



obr. 26: okna EPOCu na velké obrazovce

Vraťme se však k dalším příkladům na obr. 25. Druhý příklad v pravém horním rohu obrazovky ukazuje 'neviditelné' rodičovské okno (neviditelné proto, že mu programátor prostě nepřiřadil žádný vlastní obsah), jehož jediným účelem je spojit několik synovských oken do jediného celku, který se bude snadno pohybovat po obrazovce. Rodičovské okno by samozřejmě i v tomto případě mohlo obsahovat orámování, společné všem synovským oknům - takovým příkladem je okno 'File Manager' na obr. 26, které obsahuje synovských oken několik: v jednom z nich je jméno adresáře ("LOC::C:\TEXTY\KNIHY\80386*"), v dalším jména adresářů ('80386' a 'EPOCPROH') a v dalším jména souborů.

Operační systémy

Poslední příklad z obr. 25 právě ukazuje synovské okno, které je větší než okno rodičovské; na obrazovce proto vidíme pouze část synovského okna¹²⁴; viditelnou část můžeme snadno změnit posunutím synovského okna vůči rodičovskému.

12.4 Překrývající se okna

Jednotlivá okna se mohou samozřejmě překrývat - nejedná se jen o překrytí rodičovského okna synovským oknem, ale také o vzájemné překrývání oken na stejně úrovni hierarchie: na obr. 26 např. okno 'Set File Attributes' překrývá okno 'File Manager', a to samo překrývá další dvě okna, ze kterých vidíme pouze část u levého okraje.

V souvislosti s tímto překrýváním musí operační systém zajistit řadu věcí - podívejme se na ně podrobněji.

12.4.1 Vlastnictví oken

Aktivujeme-li některý proces, bylo by jistě vhodné, aby operační systém všechna jeho okna přemístil do popředí před momentálně méně zajímavá okna ostatních procesů. Operační systém proto musí jako součást každého okna udržovat také informaci o tom, kterému procesu okno patří.

Je vhodné si uvědomit, že se nejedná o jediný důvod pro udržování informací o vlastníkovi: je např. běžné, že klepneme-li myší do okna některého z neaktivních procesů, operační systém proces ihned aktivuje. Často také operační systém umožňuje skrýt všechna okna zvoleného procesu nebo všechna okna všech neaktivních procesů (to dělá EPOC na malé obrazovce automaticky).

¹²⁴Volba přerušovaných a plných čar tedy v tomto případě vlastně není zcela správná: nevidíme rodičovské okno, ale synovské; to však vidíme pouze v omezených souřadnicích okna rodičovského.

12.4.2 Správné překreslování

Jednotlivé procesy samozřejmě nevědí nic o tom, je-li některé z jejich oken momentálně překryto oknem jiného procesu (nebo dokonce jiným oknem téhož procesu nebo ne). Pokojně proto do okna kreslí; je věcí operačního systému zajistit, aby se grafika objevila na správném místě a ve správnou dobu. Nepřipadá tedy v úvahu situace, kdy program kreslí do zcela zakrytého okna, a jeho kresba se přesto objeví na 'povrchu' některého z oken, které původní okno zakrývají.

Nejjednodušší grafické systémy (např. GEM) nabízejí programátorům prostředky pro korektní kreslení do okna, nejčastěji ve formě služby, která omezí výstupní prostor zadaný obdélníkem (tzv. 'clip'). Grafický výstup mimo zadaný obdélník se prostě nekreslí.

Tento mechanismus má dvě nevýhody, velmi úzce vzájemně související. V případě, že programátor službu 'clip' použije špatně, může k výše popsané situaci dojít. Z toho vyplývá, že programátor se musí velmi aktivně starat o správné 'clipování', a to je právě druhá nevýhoda - zbývá mu totiž samozřejmě méně času a pozornosti na řešení vlastního problému.

Většina moderních grafických systémů proto pracuje rozdílným způsobem - služba 'clip' je sice stále k dispozici, je však pouze jakousi 'nadstavbou'; i v případě, že ji programátor nevyužije, bude kreslení automaticky omezeno jen na viditelnou plochu okna, do kterého program právě kreslí¹²⁵. Programátor může 'clipováním' tuto plochu nanejvýše omezit, ale v žádném případě rozšířit.

¹²⁵V případě EPOCu a jiných systémů s hierarchií oken je, jak již víme, dalším omezením plocha všech oken rodičovských.

12.4.3 Překreslení okna

Jestliže byla část okna zakryta a uživatel přeuspořádal obrazovku tak, že se stala viditelnou nebo jestliže jsme zviditelnili dosud ukryté okno, je nutné překreslit jeho obsah.

Existuje několik metod, kterými může operační systém překreslení zajistit; jednodušší grafické systémy umožňují využití jen jedné či dvou z nich, komplexnější systémy umožňují programátorovi zvolit momentálně nejvýhodnější metodu. Probereme postupně všechny možnosti a zmíníme se i o operačních systémech, které je podporují.

- Nejjednodušší, ale obvykle neakceptovatelná, metoda prostě **zakáže zakrytí části okna** - pak samozřejmě není zapotřebí okno překreslovat. U moderních multitaskových systémů samozřejmě tato metoda vůbec nepřipadá v úvahu - jejím důsledkem je mimo jiné totiž to, že dokud je takové okno zobrazeno, nemůžeme aktivovat jiný proces. S touto metodou se proto setkáme jen u dialogových oken starších systémů GEM¹²⁶ nebo Apple Macintosh.
- Poměrně nenáročnou metodou je i **volání procesu, jemuž okno patří**, aby si potřebnou část okna laskavě překreslil sám. V řadě případů je to jediná možná metoda, a proto ji podporují snad všechny grafické systémy. Její jedinou nevýhodou je to, že ztěžuje práci programátorů a přidává jim další úkoly, o které se musí starat - samozřejmě na úkor vlastního řešeného problému. Moderní systémy proto tam, kde to dává smysl, nabízejí pro tuto metodu i různé alternativy.
- Jednou ze zmíněných alternativ je **zálohování obsahu okna** - operační systém samozřejmě může všechny grafické operace provádět nad pomocným oknem mimo obrazovku a na obrazovku pak přenášet pouze ty, které odpovídají viditelným částem okna. Je-li pak zapotřebí některou část okna překreslit, stačí pouze přenést odpovídající úsek z pomocného

¹²⁶K velké a oprávněné nelibosti všech uživatelů je bohužel podle dostupných informací tento nešvar i součástí verze GEMu, která pracuje s ATARI MultiTOSem.

'mimoobrazovkového' okna. Tuto metodu podporuje volitelně většina grafických systémů (X Windows, EPOC, NeXTStep).

Nevýhodou je určité zpomalení grafických operací a především značná spotřeba paměti - např. pro zálohování okna zabírajícího celou obrazovku NeXTstepu bychom na barevném systému potřebovali řádově megabyty paměti. Praktické využití této velmi pohodlné metody proto zůstává omezeno na poměrně velmi malá okna¹²⁷.

- Proces také může určit pouze to, že okno má být při překreslování smazáno nebo naopak vyplněno zadanou barvou. Toho je možné velmi pohodlně využít v případě, že má okno obsahovat pouze rámeček pro synovské okno (nebo pro několik synovských oken).
- Zdokonalenou verzí minulé metody je případ, kdy proces operačnímu systému předá hotový obrázek, jímž má být okno automaticky při každém překreslení pokryto. To je samozřejmě velmi výhodné u různých informačních oken, jejichž obsah se mění jen málokdy nebo vůbec.
- Spíše pro zajímavost uvedeme další zdokonalení, které je implementováno v EPOCu: proces může operačnímu systému předat ne jeden, ale libovolnou sekvenci obrázků, které pak operační systém v určených intervalech kreslí jako podklad okna - získáme tak okno, které bude automaticky překreslováno a jeho podklad bude navíc animován.
- Poslední možnost se podobá zálohování; tentokrát si však operační systém nepamatuje skutečný obsah okna, ale grafické operace, které byly nad oknem provedeny; je-li zapotřebí obsah okna obnovit, prostě všechny grafické operace zopakuje.

Při implementaci této metody však narazíme na řadu technických i principiálních problémů (překreslení dlouho používaného okna

¹²⁷Velice často se s ním proto setkáváme v EPOCu na malé obrazovce počítačů PSION Series 3.

Operační systémy

s dlouhou řadou příkazů by např. zabralo neúměrně dlouhou dobu); obvykle proto tato metoda nebývá k dispozici.

13. Uživatelské rozhraní

Grafické uživatelské rozhraní využívá grafických služeb nižších úrovní a nabízí základní operace pro komunikaci s uživatelem. Patří sem tedy skupina grafických prvků, jejichž prostřednictvím operační systém uživateli ukazuje momentální stav jednotlivých subsystémů a nabízí přípustné povely, i mechanismus spolupráce se vstupními zařízeními - jako je myš, klávesnice a v některých případech i další zařízení (např. mikrofon u pracovních stanic NeXT).

Ačkoli je variabilita mezi různými systémy grafického uživatelského rozhraní poměrně vysoká, využívají všechny několika shodných základních prvků - jmenujme například okno, nabídku (menu) nebo ikony. Praxe již dostatečně prokázala užitečnost těchto základních stavebních kamenů; s velice slušnou pravděpodobností můžeme předpokládat, že ani v dohledné budoucnosti nebudou grafická uživatelská rozhraní doplněna o zcela nové prvky. Spíše bude docházet k dalšímu zdokonalování prvků stávajících - zatímco např. ikony původních systémů Apple Macintosh byly určeny černobílým obrázkem 32x32 obrazových bodů, využívá dnešní NeXTStep pro ikony libovolných obrázků ve formátu TIFF s rozměry o polovinu většími.

Jednotlivé systémy grafického uživatelského rozhraní se navzájem odlišují jedním nebo několika ze tří možných základních způsobů:

- Vzhled základních prvků jednoho systému se obvykle více či méně liší od vzhledu základních prvků systémů ostatních. Již jsme se zmínili např. o rozdílu mezi ikonami Macintoshe¹²⁸ a NeXTstepu; jiným rozdílem může být třeba to, že základní nabídka příkazů je u Macintoshe umístěna horizontálně v nejvyšším řádku obrazovky, zatímco operační systém NeXTStep zobrazuje základní nabídka vertikálně jako sloupec příkazů na místě, které si zvolí uživatel (nejčastěji to bývá v levém horním rohu obrazovky). EPOC, který se na kapesních počítačích PSION snaží šetřit každým bitem obrazovky, naproti tomu základní nabídka nezobrazuje vůbec, dokud uživatel nestiskne klávesu 'Menu'.

¹²⁸Jehož grafické uživatelské rozhraní dnes již podporuje i ikony barevné.

- Funkce základních prvků grafického rozhraní je určena daleko pevněji než jejich vzhled - je např. zvykem, že ikona reprezentuje nějaký objekt, a skutečně se jen velmi zřídkakdy setkáme s případem, že by ikona byla použita v jiném kontextu. Přesto existuje i zde řada rozdílů:

Chceme-li vybrat kupříkladu příkaz z nabídky na počítači Apple Macintosh, musíme na něj dojet myší a stisknout tlačítko; nabídka se pak rozevře, my z ní vybereme další příkaz a pak teprve tlačítko uvolníme. Tím se příkaz aktivuje. Naproti tomu u grafického rozhraní GEM stačí dojet myší na příkaz v základní nabídce. Nabídka se ihned otevře a můžeme z ní vybírat další příkaz; tlačítko stiskneme teprve ve chvíli, kdy chceme příkaz aktivovat. Systém nabídek NeXTstepu umožňuje využívat obou popsaných režimů.

Dialogová okna EPOCu nebo NeXTstepu uživateli nijak nebrání v práci s ostatními programy nebo s ostatními dokumenty v též programu. Dialogové okno Macintoshe nebo GEMu naproti tomu znemožní jakoukolijinou práci se systémem, dokud dialog neukončíme.

- Často se odlišuje i způsob vzájemného propojení a spolupráce základních prvků. V grafickém uživatelském rozhraní pracovních stanic Sun si můžeme například v kterémkoli okamžiku vyžádat stisknutím pravé klávesy myši zobrazení nabídky příkazů, které lze aplikovat na objekt, na který právě kurzor myši ukazuje. Při práci s počítačem Apple Macintosh naproti tomu musíme nejprve požadovaný objekt zvýraznit, a pak můžeme vybírat odpovídající příkazy z hlavní nabídky (která se ve chvíli zvýraznění objektu automaticky pozmění tak, aby obsahovala příkazy týkající se objektu). Většina aplikací EPOCu ponechá i nabídku beze změny a teprve při pokusu o provedení příkazu se dozvíme, že na aktivní objekt příkaz použít nelze.

Nyní postupně probereme základní prvky grafických uživatelských rozhraní. U každého z nich popíšeme, čím je typický a se kterými jeho vlastnostmi se setkáme prakticky v každém prostředí, zmíníme se i o konkrétních zajímavých implementacích v některém konkrétním grafickém systému. Pro lepší srozumitelnost doplníme text řadou ilustračních obrázků, přebraných z grafických uživatelských rozhraní Macintoshe a EPOCu; zatímco Macintosh bude sloužit jako ukázka velmi luxusního uživatelského rozhraní, je EPOC

zajímavým příkladem velmi úspěšného řešení problémů vyplývajících z extrémně malé obrazovky. (Obrázky z NeXTstepu až na jedinou výjimku nepřebíráme z technických důvodů - NeXTStep využívá barevný antialiasing, takže jeho zobrazení působí dojmem daleko vyššího rozlišení, než jaké je ve skutečnosti k dispozici. Po převodu do černobílé grafiky, kterou je možné v této knize tisknout, nastane bohužel efekt právě opačný - výsledný efekt nestojí za mnoho.)

13.1 Okna

Snad nejdůležitějším objektem grafického uživatelského rozhraní vůbec je okno. Systémem oken jsme se poměrně podrobně zabývali v kapitole 12; na tomto místě proto nebudeme detailně rozebírat mechanismus oken, ale zaměříme se spíše na jejich vzhled a na jejich interakci s uživatelem.

Okno je téměř vždy pravoúhelník (na čemž nic nemění fakt, že některé systémy pro vylepšení estetického dojmu zobrazují okna se zakulacenými rohy), který obsahuje obrazovou (nebo textovou) reprezentaci nějakých dat a který umožňuje uživateli s těmito daty do jisté míry manipulovat jako s celkem.

V typických grafických systémech ovládaných myší¹²⁹ bývá okno složeno z následujících prvků (ne každé okno ovšem musí obsahovat všechny najednou):

Jméno nebo **titulek** okna:

Jméno okna je obvykle velmi stručnou informací o tom, co okno vlastně obsahuje. V titulku okna obsahujícího data načtená z nějakého souboru tak bývá nejčastěji jméno tohoto souboru a podobně.

¹²⁹EPOC pro počítače Senes 3, které myš nemají, by většinu popisovaných ovládacích prvků nevyužil; jeho okna je proto také nemají. Okna používaná ve verzi EPOCu pro (nepříliš rozšířené) počítače PSION MC, které zařízení nahrazující myš mají, jsou samozřejmě popsanými ovládacími prvky vybavena.

Operační systémy



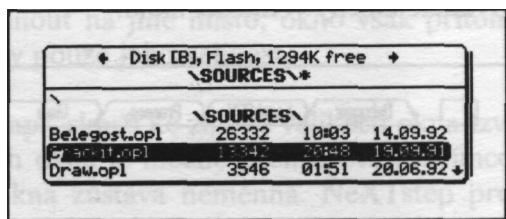
obr. 27: okno v NeXTstepu

Okna i s titulky si můžeme prohlédnout na snímku obrazovky EPOCu ve verzi pro počítače IBM PC, který je na obr. 26 na straně 299.

Podtitulek:

Některé systémy grafického uživatelského rozhraní umožňují zobrazit pod jménem okna ještě jeden řádek textu. Ten může buď určovat podrobněji obsah okna (je-li např. titulek okna 'Účty', může podtitulek obsahovat text 'grafické vyjádření příjmů v roce 1992'), nebo shrnovat zajímavé informace o okně jako celku. Okna zobrazující soubory tak často mají v podtitulku uveden počet zobrazených souborů a jejich celkovou velikost.

Podtitulek můžeme samozřejmě vytvořit v libovolném okně jako součást jeho pracovní plochy. Existují však grafické systémy, které zobrazování podtitulku podporují přímo; z hlediska uživatele programu je to naprostě lhostejné, programátorovi to může uspořít trochu práce. Mezi takové systémy patří např. univerzální grafická nadstavba GEM.



obr. 28: titulek a podtitulek (EPOC)

Ovladač přemístění okna:

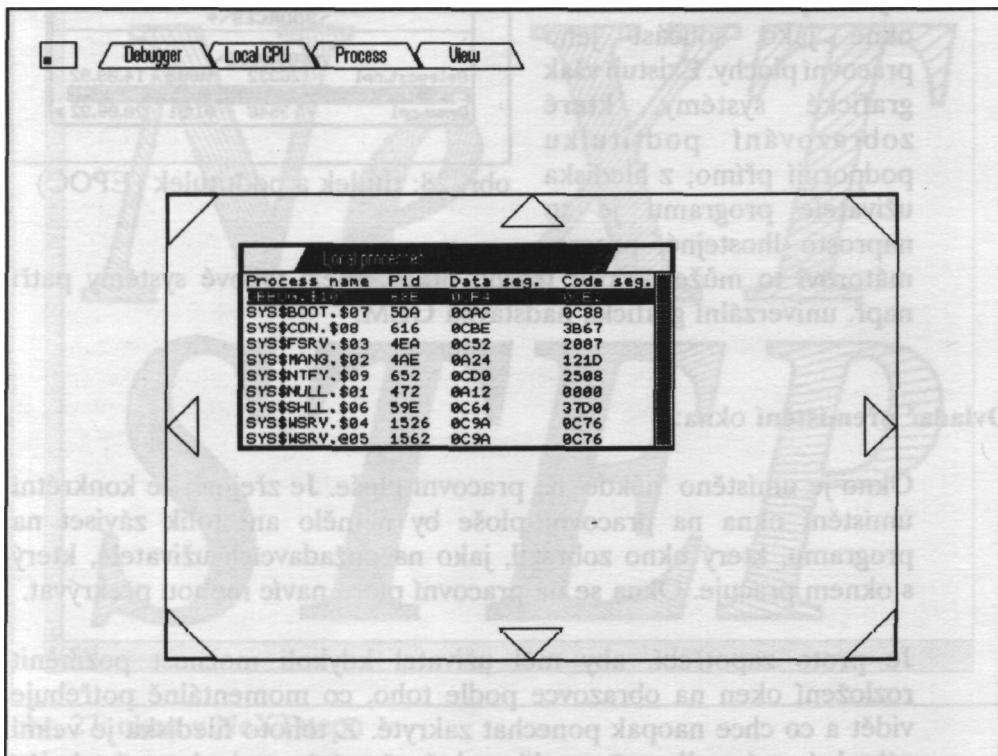
Okno je umístěno 'někde' na pracovní ploše. Je zřejmé, že konkrétní umístění okna na pracovní ploše by nemělo ani tolik záviset na programu, který okno zobrazil, jako na požadavcích uživatele, který s oknem pracuje. Okna se na pracovní ploše navíc mohou překrývat.

Je proto zapotřebí, aby měl uživatel kdykoli možnost pozměnit rozložení oken na obrazovce podle toho, co momentálně potřebuje vidět a co chce naopak ponechat zakryté. Z tohoto hlediska je velmi výhodné, máme-li možnost libovolně přemíštěvat i okna obsahující dialogy a výstražná hlášení, jako je tomu např. v NeXTstepu.

Snad všechna grafická uživatelská rozhraní pro přemístění okna využívají titulku: jestliže ukážeme myší na titulek některého okna, stiskneme tlačítko a táhneme myší, přesunuje se obvykle zároveň celé okno a zůstane na místě, na kterém tlačítko myši uvolníme. Běžné operační systémy přemísťují po obrazovce pouze obrys okna a obsah doplní na cílové místo až po dokončení akce; operační systémy, které mají velmi kvalitní a výkonný systém primitivních grafických operací, si však mohou dovolit přesunovat skutečně celá okna včetně obsahu - to je samozřejmě pro uživatele daleko pohodlnější.

Operační systémy

Ovladač změny velikosti okna:



obr. 29: změna velikosti okna v EPOCu

Obdobná situace jako s polohou okna na pracovní ploše je i s jeho velikostí. Jestliže není velikost okna pevně určena typem zobrazovaných dat (tak tomu bývá u tzv. dialogových oken, kterými se budeme zabývat později), je opět vhodnější ponechat nastavení velikosti okna na uživateli systému. Důvody jsou stejné, jako tomu bylo v minulém případě - uživatel ví daleko lépe, kolik údajů chce v okně vidět a jak velkou část pracovní plochy chce oknem zakrýt, než program, který okno nabízí.

Obvykle proto existuje i standardní metoda pro změnu velikosti okna. Nejčastěji se jedná o zvýrazněné okraje a/nebo hrany okna, které je

možné 'uchopit' myší a přetáhnout na jiné místo; okno však přitom zůstává na místě - měníme tedy pouze jeho velikost.

U počítačů Apple Macintosh např. slouží ke změně velikosti okna tzv. pohyblivý roh - pravý dolní roh okna je možné přemístit, zatímco pozice levého horního rohu okna zůstává neměnná. NeXTStep pro změny velikosti vyhrazuje celý dolní okraj okna - pravým i levým dolním rohem můžeme pomocí myši libovolně pohybovat; 'uchopíme'-li okno za střed dolního okraje, můžeme měnit jeho výšku, aniž by se měnila šířka.

Trochu jiný mechanismus pro změnu velikosti okna nabízí EPOC. Po aktivaci odpovídajícího ovládacího prvku (jímž je v tomto případě levý okraj titulku okna) se objeví obrys okna, doplněný šipkami ve všech osmi základních směrech. Pomocí těchto šipek můžeme obrys libovolně zvětšit nebo zmenšit; můžeme jej snadno i přenést na jiné místo. Po dokončení celé akce zobrazí EPOC okno přesně na místě a ve velikosti zvoleného obrysu. Příklad vidíme na ilustračním obrázku.

Nastavení standardní velikosti okna:

Jakkoli je obecně lepší ponechat nastavení okna na uživateli, jsou poměrně časté i případy, kdy je program schopen nastavit velikost okna nějakým velmi 'smysluplným' způsobem. Okno tak například může být 'roztaženo' na co možná největší velikost - ta ovšem záleží na obrazovce, na které se okno právě nalézá, jinou možností je zvětšení či zmenšení okna tak, aby právě zobrazilo celý svůj obsah a nezabíralo přitom zbytečně mnoho místa.

Může proto existovat standardní ovládací prvek, který 'se zeptá' programu, který okno vytvořil, na ideální velikost okna a pak okno na tuto velikost nastaví. Další aktivace téhož ovladače obvykle nastaví opět původní velikost okna. Tak tomu je např. u počítače Apple Macintosh.

Operační systémy

Častěji tomu bývá tak, že program - ačkoli by potenciálně mohl pracovat stejně jako u Macintoshe - okno pouze roztahne na celou obrazovku. Tak tomu je např. v EPOCu nebo v GEMu.

NeXTStep kupodivu obdobným ovladačem nedisponuje (a podle osobního názoru autora této knihy to patří mezi několik málo jeho nejnepříjemnějších nedostatků).

Závěr okna:

Jestliže již s oknem nechceme pracovat, ale přitom ještě nechceme ani ukončit program, který okno původně vytvořil, musíme jej zavřít. Ve většině případů proto bývá součástí okna i speciální ovladač, jehož aktivací okno skutečně zavřeme. Naopak okna, která nelze zavřít (protože obsahují nějakou důležitou informaci, zobrazovanou po celou dobu práce programu), nebo okna, která se zavírají jiným způsobem (dialogová okna, zavíraná automaticky po výběru některého z voličů), ovladač pro *zavírání* samozřejmě nemají.

Zajímavým a velmi vtipně vymyšleným způsobem funguje tento ovladač u operačního systému NeXTStep. Případný fakt, že obsah okna byl změněn, ale není dosud uložen, je zde totiž indikován právě drobnou změnou vzhledu ovladače. To je velice šikovné, protože uživatele tato informace zajímá především právě ve chvíli *zavírání* okna - právě tehdy si ale změny vzhledu ovladače jistě povšimne.

Ikonizace okna:

Zvláště u multitaskových operačních systémů se snadno stane, že máme najednou na obrazovce příliš mnoho oken a ve vzniklému zmatku se již špatně vyznáme. Zavírat nebo ukrývat okna se nám ale nechce - rádi bychom je měli k dispozici 'na jediné klepnutí' myší.

Některé systémy grafického rozhraní řeší tento problém tzv. ikonizací. Ikonizované okno z obrazovky nezmizí; pouze se zmenší na malíčkovou ikonu (o ikonách budeme podrobněji hovořit níže), kterou může

uživatel snadno umístit někam do rohu obrazovky, kde nebude překážet (velmi často na podobné místo uloží ikonu sám operační systém). Jakmile však opět potřebujeme s oknem pracovat, stačí vybrat ikonu myší a okno je opět celé k dispozici.

Ikonizaci oken podporuje např. grafické uživatelské rozhraní operačního systému NeXTStep nebo EPOC. Ovladač pro ikonizaci oken v EPOCu není součástí okna; namísto toho jej vidíme v levém horním rohu obrazovky (např. na obr. 29), kde může být využit pro ikonizaci aktivního okna.

Pracovní oblast okna:

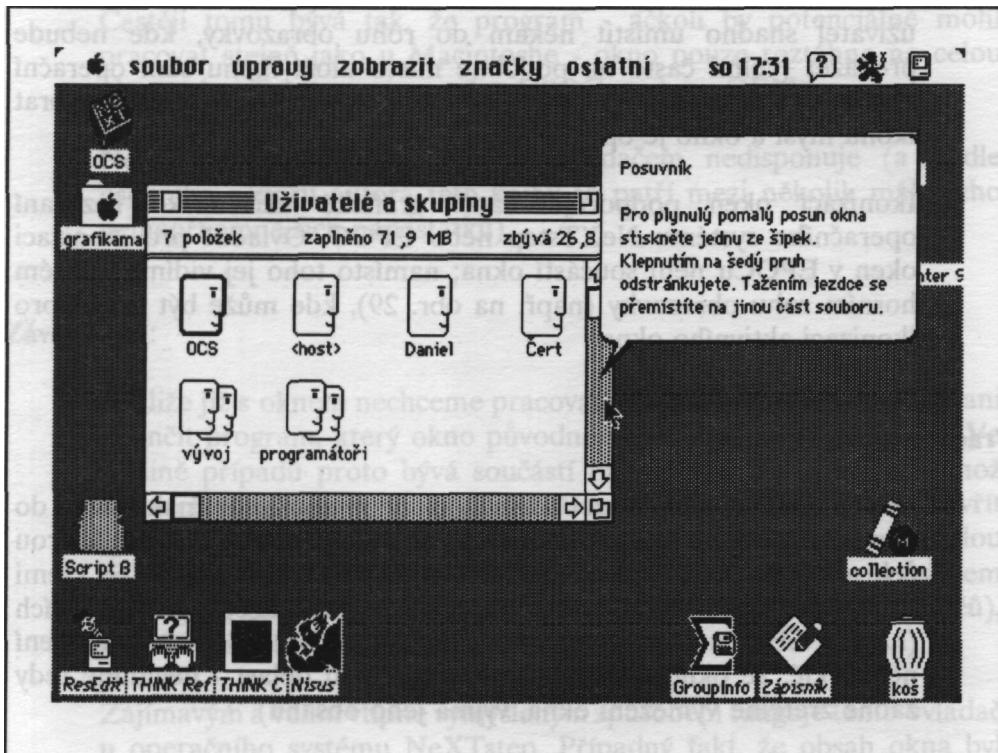
Hlavní součástí okna bývá obdélník, který obsahuje data nakreslená do okna programem. Tento obdélník je obvykle menší než oblast, kterou celé okno zabírá, protože na této ploše musí koexistovat se všemi ostatními součástmi okna popsanými v předcházejících odstavcích (např. v EPOCu na malé obrazovce počítačů PSION Series 3 však není neobvyklé, že okno obsahuje pouze pracovní oblast - nevidíme tedy žádné zřetelné vymezení okna, vyjma jeho obsahu).

Pracovní oblast okna může být interpretována dvěma způsoby: buď jako omezený obdélník - tak tomu bývá v dialogových oknech, o kterých budeme hovořit zanedlouho - nebo jako obdélníkový průhled na daleko větší plochu, na které jsou zobrazena všechna požadovaná data (touto alternativou jsme se podrobně zabývali v odstavci 12.1).

Posuvníky a šipky:

Jestliže však je okno pouze průhledem do nějaké rozsáhlejší plochy, musí existovat obecný a konzistentní způsob, jak oknem po této ploše posunovat (někdo může dát přednost alternativní představě pevného okna a posunování plochy za ním) tak, abychom si mohli postupně prohlédnout celou plochu. Pro tento úkol se zpravidla používají posuvníky ('scroll bars'), šipky a jezdec posuvníku ('thumb' nebo 'slider').

Operační systémy



obr. 30: vertikální posuvník na Macintoshi

Horizontální posuvníky posunují obsah okna doleva nebo doprava, vertikální posuvníky posunují obsah okna nahoru nebo dolů. Posuvník je obvykle reprezentován dlouhým a úzkým obdélníkem, který reprezentuje celou pracovní plochu za oknem, a jezdcem, který je umístěn na posuvníku a jeho pozice ukazuje pozici okna na pracovní ploše. Kvalitní grafické systémy zároveň informují uživatele relativní velikostí jezdce vůči posuvníku o relativní velikosti okna vůči ploše.

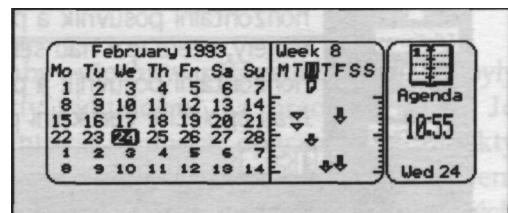
Součástí posuvníku obvykle bývá několik ovladačů, které umožňují posun plochy jedním či druhým směrem o celé okno a o malý úsek. Tyto ovladače bývají nejčastěji značeny šípkami; u grafického systému Apple Macintosh je nalezneme na protilehlých okrajích posuvníku, v NeXTstepu jsou naopak umístěny obě vedle sebe u 'dolního' okraje posuvníku (což je - alespoň podle osobního vkusu autora této knihy -

daleko pohodlnější: chceme-li posunovat okno po řádcích 'sem a tam', nemusíme téměř měnit polohu myši).

Panely:

Někdy může být vhodné, aby okno obsahovalo několik 'synovských' oken. Tato se dělí o pracovní prostor 'rodičovského' okna a každé z nich může obsahovat vlastní data, do značné míry nezávislou¹³⁰ na ostatních oknech' (z odstavce 12.3 již víme, že některé grafické systémy podporují tento mechanismus prostřednictvím hierarchického systému oken, jinde se o vše musí postarat programátor).

V rámci uživatelského rozhraní často 'synovská' okna nazýváme panely. Panely mají nejčastěji pouze pracovní plochu, někdy mívají vlastní posuvník. Jen zcela výjimečně mívají panely vlastní titulek; ostatní ovládací prvky mívá pouze nadřízené okno.



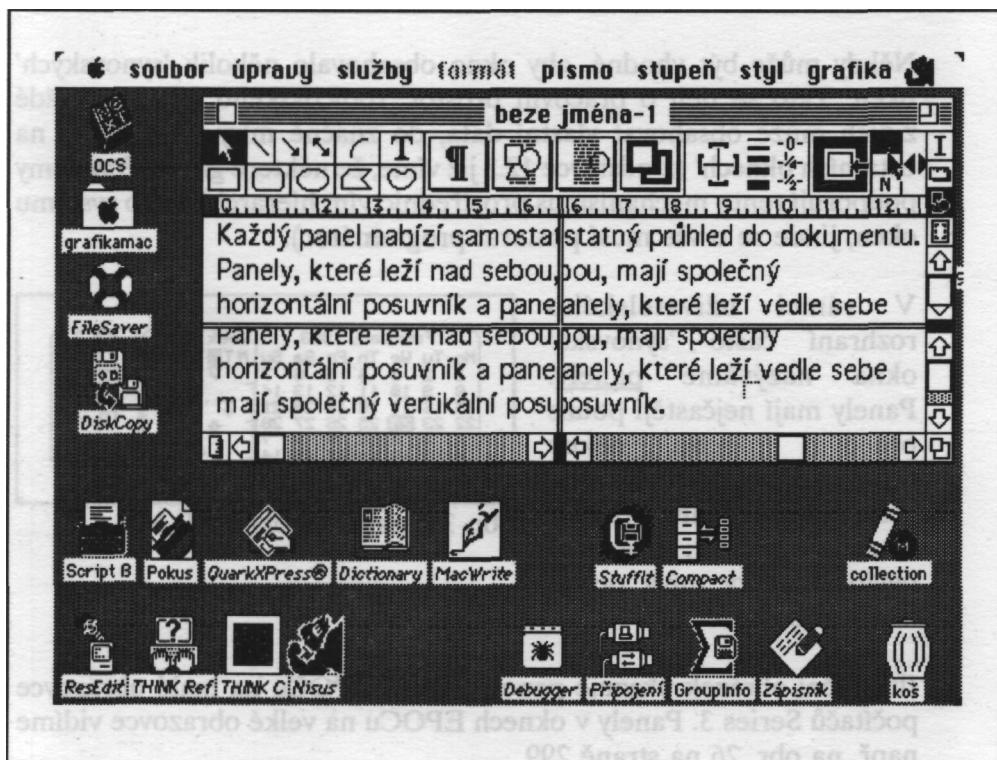
obr. 31: panely v okně EPOCu

Ilustrační obrázek ukazuje panely v okně EPOCu na malé obrazovce počítačů Series 3. Panely v oknech EPOCu na velké obrazovce vidíme např. na obr. 26 na straně 299.

¹³⁰Pokud by data byla naprostě nezávislá, nemělo by pravděpodobně smysl je ukládat do společného okna.

13.2 Ikony

Ikona je malý obrázek zabírající zpravidla plochu jednoho až pěti čtverečních centimetrů. Velmi často bývá doprovázena stručným titulkem.



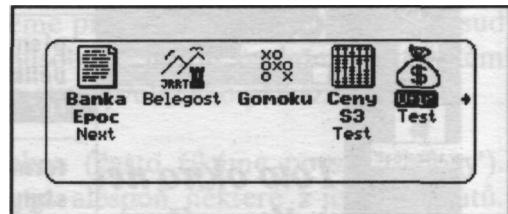
obr. 32: panely v Nisu a mnoho ikon

Ikony se používají na mnoha různých místech a pro mnoho různých úkolů; nejčastěji však reprezentují objekty nebo příkazy. Hlavním účelem ikon je umožnit rychlou identifikaci objektu na základě jeho grafické reprezentace, a ne pouze na základě jeho jména.

Většina grafických systémů disky, složky a soubory zobrazuje jako ikony doprovázené stručným titulkem obsahujícím jméno souboru (nebo disku nebo složky). Sama ikona přitom napovídá, o co se jedná - ikona disku má tvar

diskety nebo pouzdra s pevným diskem, ikona složky složku skutečně zobrazuje, ikona souboru může připomínat list papíru nebo knihu.

EPOC naproti tomu používá ikony pro reprezentaci jednotlivých programů; každá ikona je přitom doprovázena seznamem dokumentů, které jsou pro ten který program na discích počítače k dispozici.



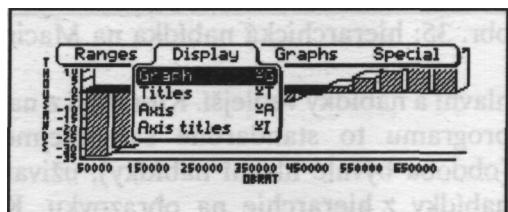
obr. 33: ikony EPOCu

133 Nabídky

Není samozřejmě možné zajistit, aby všechny přípustné příkazy byly proveditelné nějakou akcí nad objekty viditelnými na pracovní ploše. Je například velmi jednoduché reprezentovat pomocí operací nad objekty kopírování či mazání souborů; obtížné by však bylo zadávat tímto způsobem příkazy potřebné pro formátování disku nebo změnu atributů jednotlivých souborů. Podobné příkazy jsou proto obvykle soustředěny do **nabídek** (menu).

Většina grafických uživatelských rozhraní disponuje nabídkami složenými ze dvou úrovní: hlavní nabídka neobsahuje přímo příkazy, ale spíše názvy jednotlivých skupin příkazů. Teprve výběrem názvu některé skupiny se otevře tzv. vedlejší nabídka obsahující příkazy z vybrané skupiny. Přesně tímto

způsobem pracují nabídky grafického systému GEM a nabídky starších verzí operačního systému počítačů Apple Macintosh.



obr. 34: běžná nabídka EPOCu

Většina moderních GUI dnes disponuje tzv. hierarchickými nabídkami, kdy vedlejší nabídky mohou obsahovat nejen příkazy, ale i názvy dalších skupin příkazů; vybereme-li takový název, otevře se vedlejší nabídka na další úrovni, a můžeme vybírat příkazy z ní. Nejlepší systémy nabídek již nerozlišují nabídku

Operační systémy



obr. 35: hierarchická nabídka na Macintoshi

hlavní a nabídky vedlejší. Kterákoli z nabídek může být zobrazena; při spuštění programu to standardně samozřejmě bývá nejvyšší nabídka v hierarchii (obdoba bývalé hlavní nabídky), uživatel si však sám může 'vytáhnout' nižší nabídky z hierarchie na obrazovku. Kterákoli nabídka přitom samozřejmě může obsahovat příkazy i názvy skupin příkazů (které vyvolají otevření další nabídky). Tímto způsobem se chovají například nabídky operačního systému NeXTStep.

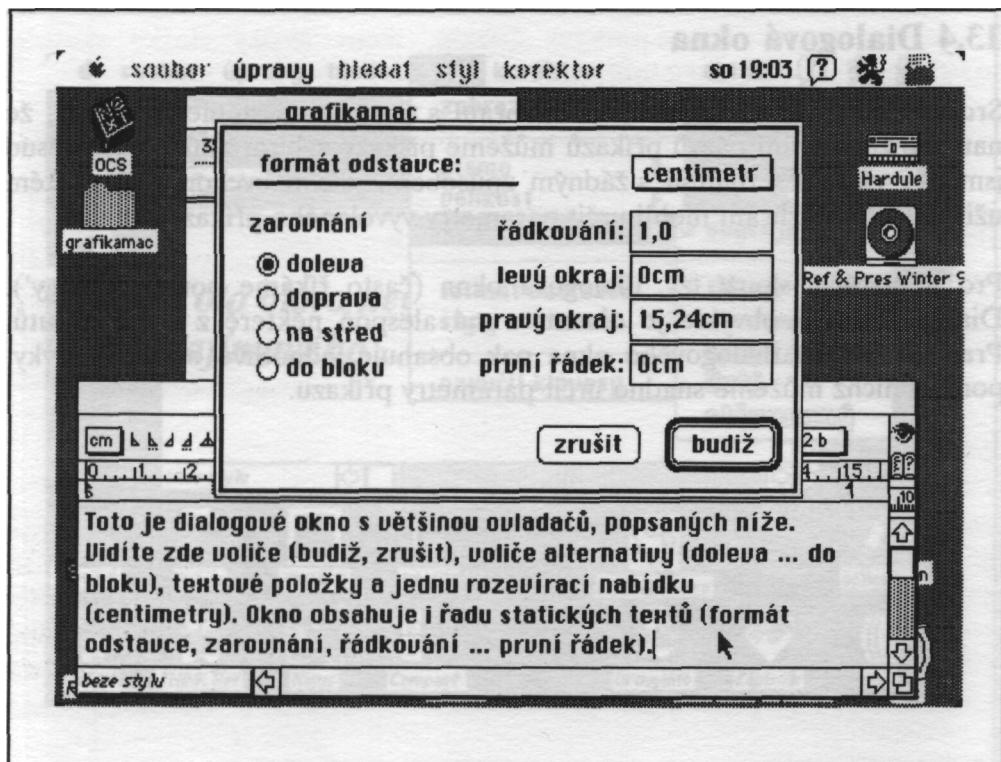
Kromě takovéto 'základní' nabídky mohou programy obsahovat další nabídky vázané na některé konkrétní objekty. Na okraji okna obsahujícího editovaný text může být například umístěn nadpis nabídky typu písma; otevřeme-li pak tuto nabídku a vybereme-li z ní nějaké písmo, zobrazí se text v okně tímto typem písma. Někdy také těmto nabídkám říkáme **rozevírací nabídky**.

13.4 Dialogová okna

Srovnáme-li grafické uživatelské rozhraní s textovým systémem, vidíme, že namísto zapisování názvů příkazů můžeme příkazy vybírat z nabídek. Dosud jsme se však neseznámili s žádným způsobem, jakým bychom v grafickém uživatelském rozhraní mohli určit parametry vyvolaného příkazu.

Pro tento účel slouží tzv. dialogová okna (často říkáme pouze 'dialogy'). Dialogové okno obvykle je oknem a má alespoň některé z jeho atributů. Pracovní plocha dialogového okna pak obsahuje jednotlivé ovládací prvky, pomocí nichž můžeme snadno určit parametry příkazu.

Operační systémy



obr. 36: dialog na Macintoshi

Pro zadávání parametrů obsahují dialogová okna nejčastěji následující ovládací prvky (poznamenejme, že konkrétní program samozřejmě může vždy vytvořit dialogové okno s vlastními nestandardními ovladači):

Volič (button):

Jedná se obvykle o obdélníček nebo ovál, který obsahuje stručný název nějaké akce (v NeXTstepu často také ikonu). Vybereme-li volič, akce se ihned provede.

Typickým voličem je ovládací prvek 'OK', který u většiny systémů znamená 'parametry nastaveny, a jedem'. Dialogová okna mívají často také volič označený 'cancel'; po jeho zvolení se dialogové okno zavře a žádný příkaz vyvolán není.

Volič alternativy (radio button):

Tento ovládací prvek slouží pro výběr jedné z několika předem daných alternativ. Vyskytuje se proto vždy ve skupinách; v každé skupině je jeden z voličů zřetelně označen jako aktivní.

Ze skupiny voličů alternativy můžeme vybrat právě jeden; volič alternativy, který byl vybrán předtím, se automaticky deaktivuje. Nastavení voliče alternativy obvykle nemívá za následek přímou akci, a slouží pouze jako parametr pro příkaz aktivovaný voličem 'OK'.

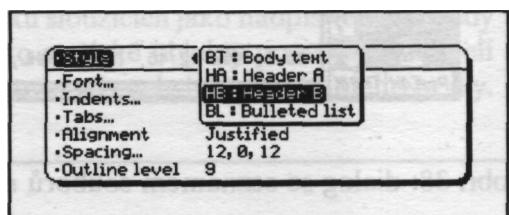
Přepínač (check item):

Přepínače se používají pro určení jednoduché volby typu ano/ne. Jednotlivé grafické systémy používají různé způsoby rozlišení zapnutého ('ano') a vypnutého ('ne') přepínače; nejčastěji bývá u přepínače malý čtvereček, který je u zapnutého přepínače 'zaškrtnut' křížkem nebo jiným znakem, zatímco u vypnutého přepínače je prázdný.

Rozevírací nabídka (popup menu):

Pro případy, kdy chceme umožnit uživateli volbu jedné z alternativ, ale nemáme v dialogovém okně dostatek volného místa pro skupinu voličů alternativy, nebo kdy není předem jasné, kolik a jakých alternativ vlastně budeme nabízet, slouží rozevírací nabídka.

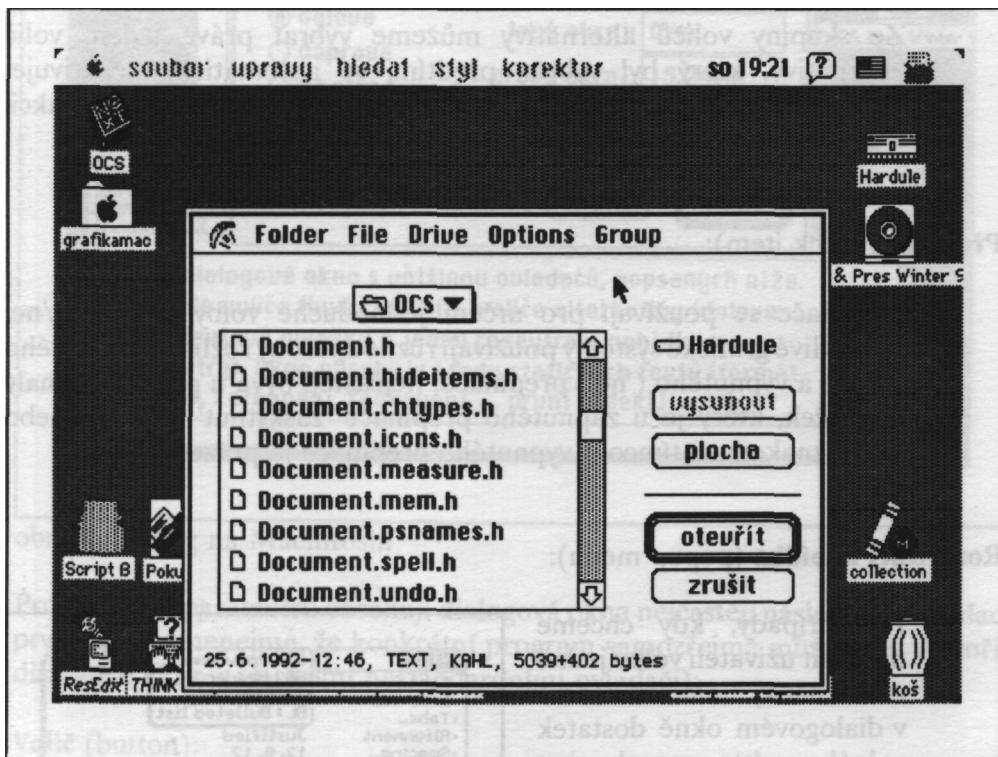
Jedná se vlastně o nadpis běžné nabídky, tak jak ji známe z odstavce 13.3. Při volbě tohoto ovladače se rozevře nabídka, ze které



obr. 37: rozevírací nabídka v EPOCu

můžeme vybrat požadovanou alternativu. Ta pak bývá zobrazena vedle rozevírací nabídky nebo přímo uvnitř jejího nadpisu jako jeho nové jméno.

Seznam (list):



obr. 38: dialog se seznamem souborů na Macintoshi

Existují případy, kdy potřebujeme uživateli umožnit výběr ze skupiny alternativ, které nejsou předem známy (ale určíme je až za běhu programu). Alternativ je přitom příliš mnoho, než aby bylo pohodlné použít rozevírací nabídku. Typickým příkladem je otevřání souboru, kdy musí dialogové okno nabídnout uživateli všechny přípustné soubory, aby z nich mohl vybrat ten, který se má otevřít.

Grafická uživatelská rozhraní proto umožňují umístit do dialogového okna seznam všech volitelných alternativ; je-li jich více, než dokáže seznam naráz zobrazit, objeví se na okraji seznamu posuvník který můžeme standardním způsobem používat pro procházení celého seznamu.

Text (edit item):

Při používání všech dosud popsaných řídících prvků pro zadávání parametrů je poměrně málodky zapotřebí umožnit jako některý z parametrů vkládat libovolný text. Pro ty případy, kdy to však zapotřebí je (např. zadávání jména uživatele či hesla pro přístup k síti), nabízejí grafická rozhraní i speciální ovladač pro zadávání textu.

Obvykle se jedná o malý 'editorek', nejčastěji jednořádkový, ve výjimečných případech několikařádkový. Ve většině případů se s tímto 'editorkem' pracuje přesně stejným způsobem jako s textem v libovolném textovém editoru¹³¹.

Statické objekty:

Pro lepší přehlednost obsahuje dialogové okno obvykle množství statických objektů - textů a obrázků sloužících jako nadpisy či návodů k ostatním objektům v okně. Tyto statické objekty pomáhají uživateli lépe se v dialogovém okně orientovat a bezchybně využívat jeho prvky.

¹³¹Součástí rozumného grafického uživatelského rozhraní je samozřejmě také to, že práce s textem je postavena na stejných principech, ať již píšeme knihu pomocí výkonného textového editoru nebo ať třeba pouze měníme jméno některého souboru.

13.5 Výstražné dialogy

Speciálním případem dialogového okna bývají tzv. výstražné dialogy. Nejčastěji se jedná o velmi jednoduchá dialogová okna, která obsahují jeden či dva obrázky, několik řádek textu a jeden, dva či tři voliče.

Výstražné dialogy bývají používány v případech, kdy je zapotřebí uživateli informovat o nějaké situaci (viz také indikátory popsané v příštím odstavci), nebo kdy je zapotřebí vyžádat si od něj potvrzení nějaké akce či volbu mezi několika alternativami.

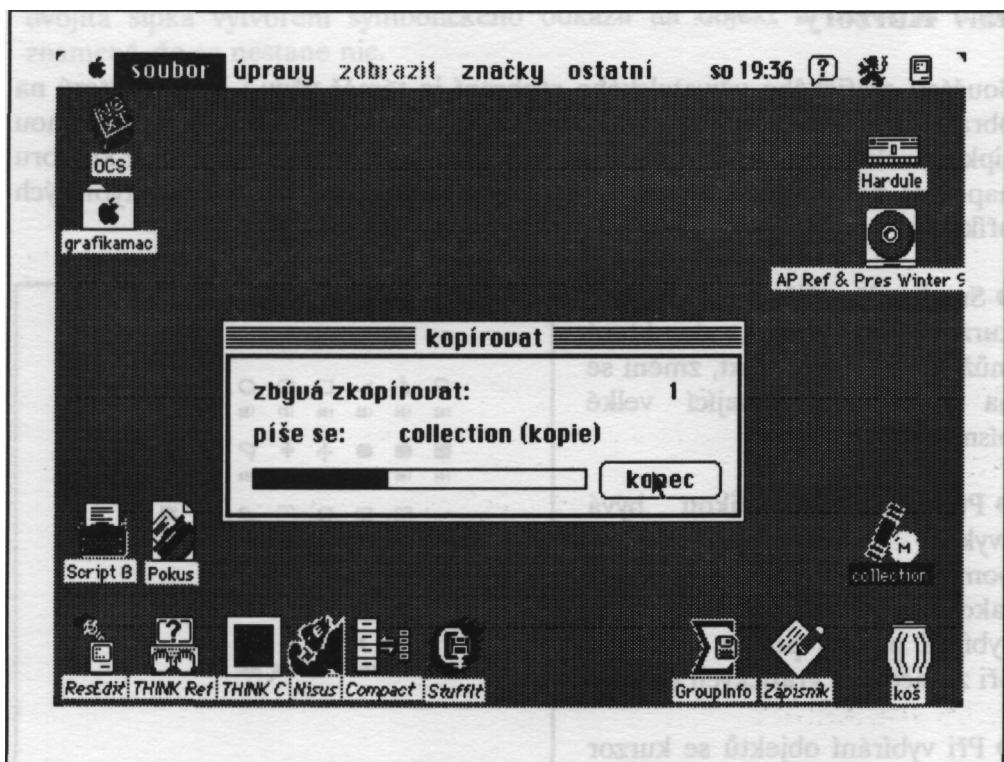
Výstražné dialogy mají specifické postavení mezi dialogy v tom smyslu, že obvykle vyžadují speciální pozornost; grafické systémy je proto často podporují zvláštními službami. Na Macintoshi je např. možné si vyžádat automatické výstražné 'pípnutí' při zobrazení takovéhoto dialogu. NeXTStep umožňuje totéž a navíc zobrazuje výstražné dialogy před všemi ostatními okny, takže výstražný dialog nemůže být náhodou zakryt jiným oknem nebo běžným dialogem.

13.6 Indikátory

V minulém odstavci jsme se seznámili s výstražnými dialogy zobrazovanými v případě, kdy systém hlásí uživateli nějakou situaci a požaduje od něj reakci ve smyslu 'beru to na vědomí'.

Existuje však řada situací, ve kterých je vhodné uživatele informovat, nelze však očekávat žádnou reakci (přesněji řečeno, nemůžeme otravovat uživatele tím, že bychom od něj zbytečnou reakci požadovali). Typickým příkladem může být kopírování souborů: uživatel jistě přivítá, bude-li v průběhu kopírování informován o tom, kolik se toho už zkopiovalo a který soubor je právě na řadě.

Pro zobrazení informací tohoto typu se často používají tzv. indikátory. Grafická uživatelská rozhraní poměrně málokdy zajišťují automaticky služby indikátorů a ponechávají jejich vytváření a obsluhu na programátorovi. Snad jediný operační systém, který přímo podporuje tvorbu jednoduchých textových



obr. 39: indikátor postupu kopírování na Macintoshi

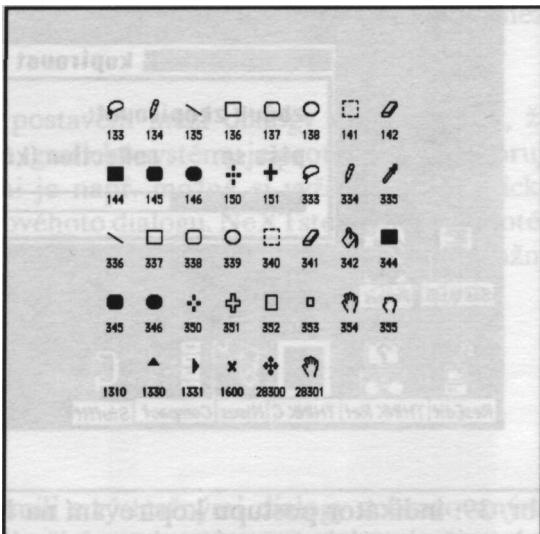
indikátorů, je EPOC.

Indikátor může být v jednoduchém případě tvořen prostou textovou informací ('Moment, prosím ... pracuji'); v praxi je vhodnější, jestliže indikátor udává nějakou vhodnou grafickou formou postup prováděné operace. Takové indikátory bývají nejčastěji obdélníkové nebo kruhové.

13.7 Kurzory

Součástí grafického uživatelského rozhraní je téměř vždy i kurzor, který na obrazovce sleduje pohyby myši. Nejčastěji bývá reprezentován stylizovanou šipkou; velmi často však je uživatelské rozhraní vytvořeno tak, že tvar kurzoru napovídá typ operace, kterou budeme provádět. Uvedeme několik typických příkladů:

- Snad nejčastější je to, že je-li kurzor nad oblastí, do které můžeme zapisovat text, změní se na znak připomínající velké písmeno I.
- Při práci s grafikou bývá zvykem reprezentovat kurzor pomocí malého křížku; pomocí takového kurzoru se pohodlněji vybírají přesné pozice potřebné při zpracování grafických dat.
- Při vybírání objektů se kurzor často změní na stylizovaný obrázek ruky s ukazujícím prstem.
- V době, kdy program pracuje a není momentálně schopen reagovat na podněty uživatele, se kurzor obvykle změní na nějaký obrázek, který tuto situaci indikuje (hodinky u počítačů Apple Macintosh, rotující terč v NeXTstepu, někdy také šálek kávy naznačující 'zajděte si zatím na kafe' nebo včelka naznačující, jak moc je program pilný). V tomto smyslu slouží kurzor jako jednoduchý indikátor toho, že program pracuje.
- Při přemísťování objektu pomocí myši může kurzor měnit podobu podle toho, k jaké akci by došlo, kdybychom objekt 'pustili' - např. v NeXTstepu dvojitý čtvereček znázorní kopírování, 'rozsvícená' šipka přemístění souboru,



obr. 40: kurzory programu ResEdit

dvojitá šipka vytvoření symbolického odkazu na objekt a obyčejná šipka znamená, že se nestane nic.

Operační systémy

Díl třetí

Aplikační programy

"Buy one (OS), get many (applications) free."

Nové reklamní heslo pro moderní operační systémy

Operační systémy

14. Pohled uživatele

Z uživatelského pohledu se při troše zjednodušení dá říci, že operační systém se skládá ze dvou základních prvků: z toho, co nám ukazuje - tím jsme se zabývali v minulém dílu - a z toho, co nám nabízí.

Služby, které operační systém nabízí programátorům, jsme velmi podrobně probírali v prvém dílu knihy. Nyní se zaměříme na služby, které operační systém nabízí uživateli.

14.1 Interpret příkazů

Součástí každého operačního systému je tzv. **interpret příkazů**. Již jsme se o něm zmínili v souvislosti s grafickým uživatelským rozhraním - jedná se o speciální program, jehož úkolem je přímo komunikovat s uživatelem, přebírat od něj příkazy a s využitím služeb operačního systému je plnit. Interpret příkazů bývá obvykle zcela standardní aplikací; někdy má mírně výsadní postavení - často např. není možné interpret příkazů ukončit (proč také).

Existují v zásadě dva základní typy příkazových interpretů: rádkový a grafický. Historicky daleko starší rádkový interpret je dědictvím po terminálech sálových počítačů a pracuje velmi jednoduchým způsobem: uživatel zapíše řádek, má přitom k dispozici základní editační příkazy. Teprve po odeslání řádku stisknutím klávesy 'Enter' je obsah řádku interpretován jako příkaz. Typickým příkladem operačního systému, vybaveného rádkovým interpretem příkazů, je UNIX nebo MS DOS (jemuž 'přehlednost' jeho příkazů v anglicky mluvících zemích vysloužila přezdívku 'MeSsy DOS').

Takový typ komunikace maximálně vyhovuje velmi zkušenému uživateli nebo programátorovi, jemuž obvykle umožňuje vytvářet příkazové dávky (tj. skupiny příkazů, které budou zpracovány automaticky v zadaném pořadí) a dává mu daleko rychlejší přístup k většině služeb. Začátečník je naproti tomu s rádkovým interpretorem příkazů ztracen, protože na rozdíl od zkušeného uživatele nezná desítky nejrůznějších příkazů a jejich parametrů z hlavy a musí

Operační systémy

neustále listovat v manuálech. Elektronická nápověda, kterou UNIX disponuje odjakživa (příkaz 'man') a MS DOS od verze 5.0, sice trochu pomůže, ani zdaleka však není samospasitelná.

Začátečníkům a méně zkušeným uživatelům proto dnes vychází naprostá většina operačních systémů po vzoru počítačů Macintosh vstříc druhou alternativou příkazového interpretu, kterou je grafické uživatelské rozhraní. V něm se řada příkazů volí prostřednictvím 'odpovídajících' akcí vyvolaných pomocí myši - soubor se např. smaže odtažením jeho ikony nad ikonu koše na odpadky, zkopíruje na jiný disk přemístěním jeho ikony nad ikonu požadovaného disku a podobně. Ostatní akce, které by se tímto způsobem vyjadřovaly obtížně, jsou k dispozici ve formě nabídek (menu), kde se uživatel může jednotlivými příkazy doslova přebírat. Parametry příkazů se pak zadávají pomocí jakýchsi formulářů, které uživatel vyplní; těmto formulářům říkáme dialogy nebo dialogová okna. Starší grafické příkazové interpretory (např. GEM nebo starší verze systému počítačů Macintosh) spoléhaly na pouhou přehlednost a intuitivní ovládání; modernější systémy se snaží vzdorovat i těm nejméně nápaditým uživatelům navíc i mohutnými systémy nápověd.

Je zřejmé, že grafické uživatelské rozhraní je pro začátečníka a méně zkušeného uživatele manou z nebe. Často jím nepohrdne ani zkušený uživatel nebo programátor; ten však brzy pozná, že existuje řada problémů, pro jejichž řešení je prostředí jednoduchého řádkového interpretu k nezaplacení:

- Již jsme se zmínili o možnosti vytvářet příkazové dávky. To, že grafická uživatelská rozhraní podobnou možností obvykle nedisponují, je jejich velmi podstatnou nevýhodou; speciální programy, které umožňují vytváření záznamů akcí uživatele (např. MacroMaker pro Apple Macintosh) jsou jen slabou náplastí.
- Pro toho, kdo si dokáže zapamatovat jména a parametry několika desítek příkazů, jsou příkazy prostřednictvím řádkového interpretu přístupné. Často rychleji a pohodlněji než prostřednictvím grafického rozhraní. Chceme-li např. v UNIXu přemístit soubor 'text' do adresáře 'texty' v domovském adresáři, napišeme pravděpodobně 'mv text ~/texty' za kratší dobu, než jakou nám zabere tažení souboru z místa na místo pomocí myši - nemluvě o tom, že často musíme nejprve vyhledat a otevřít příslušná okna. Chceme-li vytvořit pro změnu třeba v MS DOSu

nový adresář, napíšeme daleko rychleji 'md data', než kdybychom hledali příkaz 'Nový adresář' třeba v nabídkách GEMu.

Tvůrci grafických uživatelských rozhraní jsou si tohoto handicapu vědomi a snaží se jej dohnat pomocí tzv. klávesových zkratek (hot keys, keyboard shortcuts). Princip klávesových zkratek je velmi jednoduchý: většinu často používaných příkazů můžeme vyvolat pouhým stisknutím vhodné kombinace kláves, aniž bychom museli hledat *příkaz* v nabídkách.

Klávesové zkratky jsou samozřejmě výborné (grafické uživatelské rozhraní bez nich hraničí s nepoužitelností), ale mají také své nevýhody. Snad hlavní z nich spočívá v tom, že se přece jen hůře pamatuje než anglická slova, která často tvoří příkazy řádkových interpretů.

- Řádkový interpret umožňuje velmi jednoduché vkládání příkazů, které grafické rozhraní není prakticky schopné nabídnout. UNIXový příkaz 'rm obr[123].ti*.Z' například smaže všechny soubory se jménem 'obr1', 'obr2' nebo 'obr3', které mají příponu začínající písmeny 'ti' (bude se tedy asi jednat o nějaké obrázky s příponami 'tiff' nebo 'tif') a jsou zkomprimovány programem COMPRESS (který připojuje vlastní příponu 'Z').
- Prostřednictvím řádkového interpretu můžeme kdykoli spustit kterýkoli program, který je na počítači vůbec k dispozici. Grafické uživatelské rozhraní nám naproti tomu umožní spustit pouze program, jehož ikonu momentálně vidíme na obrazovce.

Rozumná grafická uživatelská rozhraní si pomáhají řadou technik, z nichž asi nejrozšířenější je spojování programů a jejich datových souborů. Operační systém dokáže zjistit, který program má zpracovávat určený datový soubor; uživateli pak stačí tento soubor 'otevřít' a operační systém vyhledá a spustí potřebný program sám. Přesto však je spouštění programů - zvláště těch méně často používaných - obecně pohodlnější v řádkovém interpretu.

Operační systémy

Většina moderních operačních systémů proto oba přístupy kombinuje tak, že uživatel má k dispozici grafické uživatelské rozhraní; jakmile však potřebuje provádět některé akce, které se v grafickém rozhraní realizují obtížně, může vyvolat jediným příkazem řádkový interpret. Snad jedinou - tentokrát nepříliš čestnou - výjimkou mezi významnějšími systémy je zde operační systém počítačů Apple Macintosh, který řádkové rozhraní nenabízí vůbec nikomu a nikdy¹³².

14.2 Příkazy

Řádkové interpretory často rozlišovaly tzv. vnitřní a vnější příkazy. Vnitřní příkazy byly přitom skutečně interpretovány, zatímco vnější příkazy byly naprostě běžnými programy. Jejich volání však mělo stejnou syntaxi se zápisem příkazů vnitřních, takže uživatel je za běžných okolností vůbec nemusel rozlišovat.

V operačním systému MS DOS např. existují příkazy COPY a XCOPY. Příkaz XCOPY je daleko 'chytrější', jejich základní chování je však naprostě stejné: napíšeme-li příkaz

```
copy a:data.txt b:
```

nebo příkaz

```
xcopy a:data.txt b:
```

zkopíruje se v obou případech soubor 'data.txt' z disku 'a' na disk 'b'.

Vnější příkazy v grafických rozhraních dosud dobře nelze použít; většina služeb, které tyto příkazy původně zajišťovaly, proto musí být integrována v grafickém příkazovém interpretu. Zbývající vnější příkazy - zpravidla to bývají

¹³²Abychom byli poctiví, musíme přiznat, že programátoři - tedy ti, kdo řádkový interpret potřebují daleko nejčastěji - jej mají integrován ve velmi kvalitní verzi ve vývojovém systému MPW. Bohužel, systém MPW je velmi nepříjemně pomalý i na nejvýkonnějších modelech Apple Macintosh, což tuto možnost poněkud diskvalifikuje.

administrativní prostředky pro správce systému - jsou buď ponechány jako přístupné pouze prostřednictvím řádkového interpretu (tak tomu podle některých údajů má být např. ve Windows NT, kde je prý lokální síť TCP/IP nutno konfigurovat z řádkového prostředí á la MS DOS), nebo se postupně mění v plnohodnotné aplikační programy, které se již netváří jako součást příkazového interpretu (tak tomu je např. v NeXTstepu).

143 Aplikace

Součástí většiny dnešních operačních systémů je i několik programů, které lze jen těžko považovat za nutné příkazy operačního systému. Tyto programy spíše zajišťují, aby byl počítač jakž takž použitelný i bez nákupu dalších komerčních programů.

- Snad každý operační systém je kupříkladu vybaven jednoduchým textovým editorem - původně bylo důvodem pravděpodobně to, že většina operačních systémů obsahuje konfigurační údaje uložené v textových souborech; správce systému proto potřebuje textový editor jako základní prostředek pro svou činnost. Dnes řada systémů nabízí pro konfiguraci samostatné aplikace s pohodlným grafickým uživatelským rozhraním; editor však součástí systému zůstává jako jakýsi 'nadbytečný luxus', k němuž se často přidávají i další aplikační programy.
- Jak jsme se zmínili, bývají součástí systému i programy sloužící systémovému administrátorovi. Jejich služby zahrnují konfiguraci sítě, správu uživatelských kont a přístupových práv jednotlivých uživatelů k nejrůznějším zařízením, správu sdílení souborů a řadu dalších prostředků.
- Mnoho operačních systémů obsahuje i více či méně luxusní terminálový program, umožňující navázat spojení s jiným počítačem prostřednictvím jednoduché sériové linky (nebo modemu a telefonu).

Jak uvidíme v dalších odstavcích, může být standardní součástí operačních systémů skutečně ledacos. Setkáme se i s méně praktickými doplňky - operační

Operační systémy

systém MS Windows např. obsahuje několik jednoduchých her; součástí operačního systému počítačů Macintosh je snad nejnezajímavější možná hra - totiž Lloydova patnáctka. NeXTStep zase obsahuje v elektronické formě kompletní sebrané spisy Williama Shakespeara.

14.4 Vývojové prostředí

Připomeňme odstavec 9.3.3, ve kterém jsme uváděli důvody, proč by měl operační systém standardně obsahovat podporu alespoň jednoduchých databází - usnadnění práce programátorů a kompatibilitu aplikací, které takový systém budou využívat.

Z obdobných důvodů je vhodné, aby nedílnou součástí operačního systému bylo i dobře navržené vývojové prostředí pro tvorbu aplikací. Nejenže to usnadní život programátorům, ale navíc to umožní snadné sdílení knihoven (nebo objektových balíků) mezi jednotlivými programátory a tedy další vzrůst produktivity.

Podaří-li se navíc připravit vývojové prostředí, které je snadno použitelné, intuitivní a lehce zvládnutelné, přináší to další výhodu - zkušenější uživatelé, kteří nejsou profesionálními programátory, si mohou sami vytvářet velmi jednoduché jednoúčelové programy.

Naprostá většina operačních systémů pro mikropočítače řeší pouze problematiku, o které jsme se zmínili na druhém místě - obsahují více (ATARI TOS nebo EPOC) či méně (MS DOS) použitelný interpret jazyka BASIC, poměrně snadno přístupného i neprogramátorům. Určitou výjimkou je klasický UNIX, který je určen spíše zkušenějším uživatelům a proto v něm standardně nic podobného BASICu nenajdeme, zato je zde však velmi luxusní vývojové prostředí s jazykem C. Příkladem operačního systému, jehož tvůrci počítají pouze s nezkušenými uživateli, a proto standardně nenabízejí 'ani ten BASIC', je operační systém počítačů Apple Macintosh. Dosud jediným příkladem operačního systému vybaveného standardně velmi luxusním vývojovým rozhraním, přístupným i méně zkušeným uživatelům a přitom více než dostatečně silným i pro profesionální programátory, je NeXTStep.

Dodatky

V dodacích knihy konečně vyjdeme vstříc encykopedistům, které příliš nezajímaly obecné informace o operačních systémech jako takových a těšili se na přehled mnoha a mnoha běžných operačních systémů.

Pro ty, kdo nemají zkušenosti s jazykem C, používaným v příkladech, je zde také stručný úvod do tohoto jazyka, který by měl stačit k pochopení programů. Nemůže samozřejmě stačit k psaní programů vlastních - na to jsou vhodnější knihy, uvedené v seznamu literatury.

Několik málo speciálních služeb programovacího jazyka Turbo C, se kterými se čtenář setká v příkladech, by mohlo ztížit čtení knihy tomu, kdo Turbo C nezná - ať již proto, že není programátorem, nebo proto, že pracuje na systému, pro který implementace tohoto překladače není k dispozici. Jeden z dodatků proto obsahuje i základní popis těchto rozšíření.

Jak poslední dodatek je zařazen i seznam literatury, jejímž studiem si čtenář může doplnit základní informace o operačních systémech, které snad načerpal z této knihy.

Operační systémy

A. Přehled operačních systémů

V tomto dodatku naleznete velmi stručnou charakteristiku všech operačních systémů, o kterých se autorovi této knihy podařilo získat alespoň o něco více údajů než pouhé jméno.

Seznam samozřejmě není a nemůže být vyčerpávající - není v něm např. poměrně rozšířený a kvalitní operační systém VMS, používaný na minipočítacích. Autor však doufá, že podobných mezer v seznamu najdete co nejméně.

Operační systémy jsou seřazeny podle abecedy. Na konci kapitoly je pak uvedena přehledná tabulka, která shrnuje nejvýznačnější vlastnosti všech uvedených systémů.

- **Amiga DOS** byl až do vzniku EPOCu pravděpodobně vůbec nejkvalitnějším operačním systémem, dosažitelným na jednoduchých domácích počítacích (na které se 'nevezde' UNIX).

Amiga DOS disponuje preemptivním multitaskingem; jeho užitečnost je však výrazně snížena tím, že počítače Commodore Amiga nedisponují ochranou paměti a operační systém se ani nesnaží využít bezpečnostních prvků mikroprocesorů Motorola. Přesto lze Amiga DOS označit za velmi dobře navržený operační systém pro domácí počítače, které se většinou nepoužívají k 'vážné' práci, ale spíše na hraní - tam pak menší spolehlivost nevadí.

Amiga DOS má i vlastní grafické uživatelské rozhraní, tzv. WorkBench. Zkušení uživatelé mají k dispozici i interpret řádkových příkazů (CLI).

- **ATARI DOS** je 'dolní polovinou' operačního systému počítačů ATARI ST, tzv. ATARI TOSu. Samostatně se o něm zmiňujeme proto, že existuje nemalé procento programů, které 'horní polovinu' (tj. grafické uživatelské rozhraní GEM) ignorují a pracují pod samotným DOSem.

Ačkoli autor této knihy má počítače ATARI v oblibě, nemůže s dobrým svědomím o ATARI DOSu napsat nic jiného, než že se jedná o velmi triviální operační systém vhodný spíše pro osmibitový počítač; můžeme jej dobře přirovnat k MS DOSu. O multitaskingu nebo jiných moderních prvcích se nedá ani snít¹³³.

ATARI DOS však podporuje částečnou ochranu paměti - počítače ATARI totiž chrání před uživatelskými programy nejzranitelnější části paměti vyhrazené pro operační systém a nedovolují uživatelským programům ani přímý přístup k vstupním a výstupním zařízením. ATARI DOS tuto ochranu podporuje s využitím bezpečnostních prvků mikroprocesorů Motorola.

Do nejnižší bezpečnostní kategorie D (viz tabulka na konci kapitoly) řadí ATARI DOS to, že nabízí uživatelským programům možnost 'přepnout' do systémového režimu a v něm pak pracovat bez omezení. Nebýt toho, řadili bychom jej do třídy C1, podobně jako EPOC.

Zatímco ATARI DOS by se spíše hodil do osmibitového světa, **CP/M** je operačním systémem pro osmibitové systémy, a v této třídě je bezpochyby tím nejlepším, co je vůbec k dispozici. Uvádíme jej zde spíše z historických důvodů a také proto, že se stal vzorem operačních systémů ATARI TOS a MS DOS (k jejich škodě - ukázalo se, že do šestnáctibitového prostředí je lepší přenášet mechanismy používané běžně u minipočítačů a střediskových počítačů).

CP/M od firmy Digital Research jako první zavedl rozdelení operačního systému na vrstvu strojově závislých ovladačů (tzv. BIOS - Basic Input/Ouput System) a zbývající vrstvy operačního systému. Tím umožnil na svou dobu nebývalou portabilitu - CP/M mohl snadno pracovat na jakémkoli počítači s mikroprocesorem Intel 8080 nebo Zilog Z80.

¹³³Firma ATARI připravuje novou verzi TOSu, která by snad měla podporovat preemptivní multitasking a úplnou ochranu paměti. Tato verze - tzv. MultiTOS - by se tak mohla stát konečně po letech důstojným soupeřem nejbližšího konkurenta ATARI, totiž Amiga DOSu. Bohužel, v době psaní této knihy ještě nebyly o MultiTOSu k dispozici podrobné informace.

Dodejme, že firma Digital Research vytvořila varianty operačního systému CP/M i pro mikroprocesory Intel 80x86 a Motorola 680x0. První mohla být základem alespoň únosně kvalitního operačního systému pro počítače třídy IBM PC, nebyť bezvýhradné orientace firmy IBM na firmu Microsoft (která se - alespoň podle informací o Windows NT, které má autor této knihy k dispozici - nenaučila psát operační systémy dodnes). Varianta pro mikroprocesory Motorola se pak stala základem ATARI TOSu.

DOS EC je operačním systémem pro sálové počítače a tomu také odpovídají jeho vlastnosti: samozřejmě preemptivní multitaskový operační systém s podporou virtuální paměti a s uživatelským rozhraním zaměřeným na dávkové zpracování.

EPOC můžeme označit jako 'nejkvalitnější operační systém pod 8MB operační paměti a pod 100MB pevného disku'. EPOC v této knize často používáme jako příklad; nebudeme jej proto na tomto místě již podrobně popisovat a uvedeme pouze stručné shrnutí:

EPOC je objektově orientovaný operační systém s preemptivním multitaskingem. Počítače PSION, pro které je EPOC vytvořen¹³⁴, disponují ochranou paměti a EPOC jí samozřejmě využívá pro ochranu systémových dat i pro vzájemnou ochranu jednotlivých procesů. Správce paměti EPOCu zajišťuje dokonale využití paměti pomocí transparentního setřásání bloků; pro lepší využití paměti EPOC zároveň nabízí sdílené knihovny, mechanismus dynamicky zaveditelných a odstranitelných ovladačů a sdílení kódu. Neexistenci bezpečnostních prvků u mikroprocesorů Intel¹³⁵ supluje technické vybavení počítačů PSION; EPOC proto můžeme zařadit do kategorie C1.

¹³⁴EPOC je v zásadě portabilní na libovolný počítač s mikroprocesorem řady Intel 80x86. Existuje např. port EPOCu pro počítače třídy IBM PC; tam samozřejmě EPOC nemá k dispozici hardwarovou podporu pro zabezpečení které nabízí.

¹³⁵Modernější modely - 80286, 80386 a 80486 - již samozřejmě mají i zabezpečovací aparát.

Operační systémy

EPOC má vlastní grafické uživatelské rozhraní, které je velmi pohodlné a intuitivní. Interpret řádkových příkazů není bohužel běžně k dispozici; součástí standardního programového vybavení pro komunikaci s počítači třídy IBM PC však je 'remote' interpret řádkových příkazů (interpret tedy běží na IBM PC, ale řídí PSION).

- **GEM** firmy Digital Research je jen o málo více než grafickou nadstavbou použitelnou téměř pro jakýkoli operační systém. Pokud je autorovi této knihy známo, existují implementace pro počítače ATARI (kde se GEM stal standardem), pro počítače IBM PC a pro počítače s operačním systémem UNIX.

Před lety, kdy byl GEM nový, byl prakticky jedinou - a poměrně velmi solidní - konkurencí grafického uživatelského rozhraní počítačů Apple Macintosh; firma Apple si toho nicméně povšimla také a zahájila sérii soudních sporů s firmou Digital Research, ve kterých prosazovala svá práva na jediného vlastníka operačního systému, který obsahuje ikonu koše na odpadky. Namísto dalšího rozvoje byl proto GEM spíše oklesťován; díky tomu již dnes prakticky nemá šanci na prosazení v konkurenzi špičkových grafických rozhraní jako je NeXTStep nebo X Windows.

GEM sám o sobě obsahuje prostředky pro zajištění omezeného kooperativního multitaskingu; je-li tedy implementován nad operačním systémem, který sám multitasking nepodporuje (jako MS DOS nebo ATARI DOS), přinese mu alespoň tuto výhodu.

- **MACH** není kompletní operační systém, ale tzv. microkernel. Jedná se o minimalizované obecné jádro operačního systému, které zajišťuje velmi efektivním způsobem všechny potřebné služby (jako je např. virtualizace paměti a adres, plné zabezpečení nebo preemptivní multitasking s podporou multithreadingu). MACH obsahuje i velmi kvalitní podporu lokálních sítí a distribuovaného zpracování úloh na více procesorech. MACH je pravděpodobně nejlépe navrženým a nejfektivnějším jádrem operačního systému, jaké je dnes k dispozici. Samozřejmě je MACH plně portabilní na jakékoli technické vybavení, které splňuje alespoň nejzákladnější požadavky pro jeho implementaci (tj. jednotku řízení paměti apod.).

MACH kromě toho obsahuje prostředky pro snadné a efektivní 'naroubování' konkrétních služeb konkrétního operačního systému na své vlastní jádro. Na jediném počítači tak může běžet velmi efektivně MACH a nad ním mohou zároveň díky jednoduchému rozhraní pracovat 'slupky' operačních systémů UNIX V, OS/2, NeXTstepu a třeba ještě Windows NT a Solarisu. Díky tomu, že MACH je plně objektový, mají jeho 'uživatele' (tj. vlastní operační systémy) k dispozici značnou flexibilitu - každý z výše vyjmenovaných operačních systémů např. může využívat vlastní mechanismus stránkování apod.

MACH byl vytvořen na Carnegie Mellon University, kde je dále rozvíjen. Dnes na dalším rozvoji MACHu spolupracuje i firma NeXT, která jej zvolila jako jádro operačního systému NeXTStep.

- **Macintosh System 7** jsme již podobně jako EPOC celkem podrobně popsali v předcházejících kapitolách.

System 7 by se dal označit za ukázku toho, jak dlouho lze důsledným zamazáváním děr vydržet s doškovou střechou v průtrži mračen. Operační systém počítače Macintosh byl původně navržen jako jednoprogramový systém s omezeným přepínáním programů. Pravděpodobně za největší nedostatek Systému 7 bychom mohli označit to, že nevyužívá absolutně žádného zabezpečení. Základní koncepce se nezměnila dodnes; nabízáním dalších služeb se však systém stal poměrně velmi luxusním systémem s plným kooperativním multitaskingem. I přes to všechno je dnes operační systém počítačů Macintosh více než důstojným konkurentem o mnoho let mladšího systému MS Windows.

Někde se však důsledky tohoto vývoje musely projevit. U počítačů Macintosh je to vývoj aplikací. Programátoři mají k dispozici obrovskou paletu velmi luxusních služeb; ty jsou však svázány takovým množstvím komplikovaných konvencí a omezení, že v nejhorších případech nakonec programátor musí vytvořit skoro stejně množství kódu, jako kdyby odpovídající úlohy řešil sám.

Operační systémy

Operační systém počítačů Apple Macintosh je od začátku vybaven velmi luxusním grafickým uživatelským rozhraním, které se stále rozvíjí; možnosti Systému 7 již jsou natolik luxusní, že s výjimkou NeXTstepu dnes prakticky nemají konkurenci. Na druhé straně zkušeným uživatelům počítače Macintosh často velmi chybí interpret řádkových příkazů.

- **MINIX** je jednoduchý školní operační systém, dostupný ve zdrojových textech. Autor této knihy neměl možnost jej přímo studovat; na základě informací z druhé ruky se však zdá, že je hůře navržen než XINU, které jsme používali pro výklad zde.
- **MS DOS (PC DOS)** snad nemusíme žádnému čtenáři představovat. Platí o něm přibližně totéž, co jsme napsali o ATARI DOSu: daleko lépe by se vyjímal na osmibitových počítačích, kam ostatně svou koncepcí také patří¹³⁶.
- **MS Windows** je pokusem firmy Microsoft alespoň částečně dohnat nedostatky MS DOSu; pokusem, jehož úspěšnost byla již komerčně potvrzena. Podle osobního názoru autora této knihy však rozšíření Windows nepotvrzuje jejich kvalitu, ale pouze nekvalitu MS DOSu - uživatelé prostě volí menší zlo: buď vyloženě mizerný MS DOS, nebo pouze špatné MS Windows.

MS Windows lze velmi dobře přirovnat k Systému 7 počítačů Apple Macintosh. Systém 7 má daleko lepší grafické uživatelské rozhraní a je na srovnatelně výkonného počítače o poznání rychlejší; Windows to

¹³⁶A pro které byl také navržen. První počítače IBM PC byly postaveny na mikroprocesoru Intel 8088; ačkoli firma Intel tento mikroprocesor tvrdoslově označuje jako 'šestnáctibitový', jedná se o poměrně kvalitní osmibitový mikroprocesor a o nic jiného. Srovnejme jej např. s mikroprocesorem Zilog Z80: oba mohou zpracovávat osmibitová nebo šestnáctibitová data, oba mají osmibitovou sběrnici. Intel má i s čítačem instrukcí devět šestnáctibitových registrů, Zilog deset; navíc má Zilog ještě dva osmibitové registry. Oba mají instrukce pro automatické procházení paměti. Žádný z nich nemá žádné zabezpečení. Intel má navíc pouze o něco málo lepší instrukční sadu (obsahující např. dělení a násobení) a možnost adresovat 1MB paměti (Zilog může stejněho cíle - a pro programátory stejně nepohodlným způsobem - dosáhnout za pomocí vnějších obvodů).

vyrovňávají konzistentnější vnitřní strukturu a přece jen o něco lepším prostředím pro programátory.

- **Multics** je špičkový operační systém vyvinutý společným úsilím odborníků univerzity M.I.T., Bell Telephone Laboratories a General Electric Company. Multics byl vytvořen pro konkrétní počítač se speciálním technickým vybavením, vyráběný firmou General Electric; později převzala výrobu tohoto systému firma Honeywell.

Multics je velmi komplikovaný operační systém využívající řadu velmi efektivních, ale zároveň velmi složitých mechanismů. Virtuální paměť systému Multics je např. realizována na základě hierarchie různě rychlých (a samozřejmě i různě velkých a drahých) paměťových prvků. Stránky operační paměti jsou tedy odkládány na velmi rychlý, ale drahý a poměrně malý 'magnetický buben'. Tepřve pro doplnění kapacity bubnu se používá velkokapacitní, ale relativně pomalý disk.

Variantu algoritmu LRU pro výběr stránek, které mají být odstraněny z paměti, je v Multicsu také více než komplikovaná: systém pro každý proces udržuje seznam 200 posledních výpadků stránek a na základě tohoto seznamu a dalších informací o stránkách volí potřebnou strategii. Pro další zvýšení efektivity Multics umožňuje procesům, aby jej informovaly o 'náhlých změnách výpočtu' (jako je třeba přechod k další fázi překladu), kdy je pravděpodobné, že se bude měnit pravděpodobnost využití jednotlivých stránek.

Správcem úloh ani správcem procesů se již raději zabývat nebudeme - nejsou o nic jednodušší (ani méně efektivní) než správce paměti.

Multics obsahuje také velmi luxusní zabezpečení. Jako příklad můžeme uvést systém přístupových práv k souborům: Multics u každého souboru udržuje seznam uživatelů a jejich přístupových práv; každému uživateli tedy můžeme přidělit zcela individuální přístupová práva k libovolnému souboru.

Z našeho hlediska je Multics asi nejjazímací tím, že jeho složitost přivedla skupinu programátorů v Bell Laboratories k nápadu vytvořit

jiný operační systém, velmi efektivní a bezpečný, ale přitom daleko jednodušší. Aby vyjádřili kontrast mezi jednoduchostí vznikajícího systému a složitostí Multicsu, nazvali nový systém Unics. Z toho později vzniklo fonetickým přepisem jméno UNIX.

- **NeXTStep** jsme používali velmi často jako příklad; nebudeme se jím proto zabývat příliš podrobně.

NeXTStep je moderní objektový operační systém. Jeho jádrem je MACH, o kterém jsme se již zmínili; ten dává NeXTstepu velmi efektivní preemptivní multitasking s možností multithreadingu, systém virtuální paměti i s virtualizací adresového prostoru, transparentní podporu lokálních sítí a řadu dalších potřebných služeb.

Sám NeXTStep doplňuje MACH o vyšší vrstvy programového vybavení, obsahující především rozsáhlé skupiny objektů; ty poskytují programátorům ještě daleko bohatší paletu služeb, než jakou nabízí EPOC nebo Macintosh.

Součástí NeXTstepu je i grafický subsystém, který podporuje libovolné grafické výstupní zařízení. Pro dvourozměrnou grafiku je k dispozici špičkový standard Adobe Display PostScript Level 2 (doplňený o některé služby specifické pro NeXTStep - jako příklad můžeme jmenovat průhlednost objektů); pro tvorbu a zobrazování třírozměrných scén ve fotorealistickém provedení je k dispozici technologie Pixar RenderMan.

NeXTStep je navíc vybaven množstvím standardních aplikací, které usnadňují systémovou administrativu (konfigurace sítě apod.), běžnou práci s počítačem (textový editor, korektor pravopisu a thesaurus, indexační a rešeršní systém) nebo komunikaci s ostatními uživateli počítačů prostřednictvím lokální sítě i globálních sítí WAN (elektronická pošta). Standardně je součástí NeXTstepu i velmi luxusní objektové vývojové prostředí.

Grafické uživatelské rozhraní NeXTstep je bezpochyby nejkvalitnějším grafickým rozhraním, jaké měl autor této knihy možnost testovat, a pravděpodobně vůbec nejlepším uživatelským rozhraním, jaké je na

dnešních systémech vůbec k dispozici. Přesto je k dispozici i interpret řádkových příkazů - jedná se o standardní konzoli UNIXu nabízející všechny standardní 'shelly' (C, Bourne). Už tak značná flexibilita tohoto interpretu¹³⁷ je ještě zvýšena možností spolupráce s grafickým uživatelským rozhraním NeXTstepu.

- **OS/2** je poměrně velmi kvalitně navržený operační systém pro počítače třídy IBM PC. OS/2 disponuje preemptivním multitaskingem a řadou dalších moderních prvků. Odhlédneme-li od implementací UNIXu na počítače třídy IBM PC, je v době psaní této knihy OS/2 pravděpodobně nejekvalitnějším operačním systémem pro tyto počítače¹³⁸.

Starší verze OS/2 však byly postiženy množstvím chyb a prakticky nepoužitelným uživatelským rozhraním. Spouštění hotového programového vybavení pro MS DOS - kterého se jeho uživatelé celkem pochopitelně nechtěli vzdát - bylo také obtížné. Pravděpodobně právě z těchto důvodů se OS/2 příliš neujal, ačkoli je bez jakýchkoli pochyb nejméně o třídě lepší než velmi rozšířené MS Windows.

- **OS 360/370** je opět operačním systémem pro sálové počítače; můžeme proto zopakovat specifikaci, kterou jsme uvedli pro DOS EC: preemptivní multitaskový operační systém s podporou virtuální paměti a s uživatelským rozhraním zaměřeným na dávkové zpracování.
- **Solaris** bychom mohli snad nejlépe přirovnat k NeXTstepu v podání firmy Sun - jedná se opět o portabilní komplexní operační systém založený na systému UNIX. Solaris však má oproti NeXTstepu řadu významných nevýhod:

¹³⁷Autor této knihy dosud nepoznal žádný interpret řádkových příkazů, jehož možnosti by byly alespoň srovnatelné s Bourne shellem, o ještě luxusnějším C shellu ani nemluvě.

¹³⁸V době vydání této knihy to již ale nebude pravda - firma NeXT totiž zhruba na tu dobu ohlásila prodej verze NeXTstepu pro počítače třídy IBM PC.

Operační systémy

- * Jádro Solarisu nenabízí ani zdaleka takovou flexibilitu a takovou efektivnost jako MACH;
- * Solaris není objektový;
- * Grafický subsystém Solarisu je založen na systému X Windows; ten je pro práci na dvourozměrné grafice daleko slabší než Display PostScript, a třírozměrnou grafiku nenabízí vůbec;
- * Uživatelské rozhraní, které je v Solarisu k dispozici, není ani zdaleka tak pohodlné a intuitivní jako prostředí NeXTstepu;
- * Programátoři v Solarisu nenaleznou nic, co by se dalo srovnat s vývojovým prostředím NeXTstepu.

TOS je kompletní operační systém počítačů ATARI. Jedná se o kombinaci ATARI DOSu a GEMu; o obou jsme se již zmínili. GEM doplní ATARI DOS o poměrně kvalitní grafické uživatelské rozhraní (firma ATARI 'svůj' GEM dále rozvíjí sama a uživatelské rozhraní jeho posledních verzí není o mnoho horší než uživatelské rozhraní Macintoshe) a o omezený kooperativní multitasking.

Kombinace ATARI DOS+GEM je velmi kvalitní ve srovnání např. s MS DOSem; dokud však nebude k dispozici MultiTOS, nemůže dobré konkurovat ani Amiga DOSu, o moderních multitaskových systémech ani nemluvě.

UNIX (XENIX, VENIX, AIX, A/UX, HP/UX ...) je operačním systémem, který vytvořili odborníci z Bell Laboratories, když si chtěli odpočinout od složitosti Multicsu. Původní UNIX byl skutečně dost jednoduchý; postupně se však dále rozvíjel, takže dnes patří naopak mezi nejkomplikovanější operační systémy pro malé počítače. Základní koncepce UNIXu však byla navržena tak dobře, že i dnešní nejmodernější operační systémy (jako je Solaris nebo NeXTStep) jsou na UNIXu - byť zcela nově implementovaném - stále založeny.

'UNIX' je ochranná známka Bell Laboratories; proto se množství nejrůznějších implementací UNIXu vyznačuje také nejrůznějšími jmény, které nejčastěji končí písmeny '-IX'.

Za zvláštní pozornost by snad mohla stát jedna implementace, totiž A/UX. Jedná se o implementaci UNIXu pro počítače Apple Macintosh, která disponuje standardním uživatelským rozhraním počítačů Macintosh a možností spouštět 'Macintoshovské' aplikační programy. A/UX kupodivu není příliš rozšířen; autor této knihy o něm nemá k dispozici dostatek informací a může se proto jen dohadovat, že příčinou by mohla být buď nestabilita tohoto systému, nebo jeho nepřiměřeně vysoká cena.

- **Windows NT** je pokusem firmy Microsoft vytvořit konečně pořádný operační systém¹³⁹; jsou-li informace, které měl autor této knihy k dispozici správné, pokusem opět nepříliš podařeným. Sepišme několik informací ze studie firmy Locus Computing (CI No 2058, výběr ze studie byl získán prostřednictvím sítě USENET):

* Jak je již dobrým zvykem firmy Microsoft, nedělá si systém Windows NT velké starosti s dodržováním standardů (mezi nedodrženými studie uvádí např. Posix 1003.2, Posix 1002.4, Posix.2, Posix.4, XPG3, OSI nebo X25). Windows NT nepodporují ani grafický standard X Windows.

* Ačkoli je systém Windows NT uváděn jako portabilní, nedodržuje většinu požadavků Open Systems na portabilní systémy.

¹³⁹ Zajímavé je, že tvůrcem NT je pan David Cutler, bývalý vedoucí vývojového týmu firmy DEC. U této firmy pracoval na návrhu nového operačního systému pro počítače DEC, který měl nahradit dosavadní systém VMS; firma však nový systém neakceptovala a práci na něm zastavila. Podle (bohužel velmi kusých) informací, které má autor této knihy k dispozici o systému VMS a které o něm hovoří v superlativedech, není divu. David Cutler tedy nabídl koncepce operačního systému firmě Microsoft; tam se konečně dočkal uznání a podpory a jeho systém se stal novým programem firmy Microsoft pod názvem Windows NT.

Operační systémy

- * Systém Windows NT obsahuje podporu lokální sítě, ale nestandardní; stejně je tomu i elektronickou poštou. Neexistuje podpora systému NFS pro sdílení souborů prostřednictvím lokální sítě. Windows NT dokonce nepodporují ani Novell NetWare. Sdílení prostředků (jako jsou např. tiskárny) je možné opět pouze mezi systémy Windows NT (a MS Windows). Homogenní síť složená pouze z počítačů s Windows NT tedy bude pracovat (snad) bez problémů; ten, kdo chce do sítě připojit i jiné systémy, by se však měl raději NT vyhnout.
- * Na rozdíl od roky starého UNIXu nedokáže systém Windows NT obsloužit více uživatelů zároveň.
- * Windows NT neobsahují žádnou podporu pro vícejazyčné aplikace.
- * Uživatelské rozhraní neumožňuje aplikacím využít žádným způsobem službu 'drag and drop'; ta tedy zůstává 'výhradním vlastnictvím' příkazového interpretu, podobně jako tomu je u Systému 7 pro Macintosh. Mechanismus spojení datových souborů s odpovídajícími aplikacemi sice NT nabízí, uživatel jej však nemůže uzpůsobit podle svých potřeb. Ani systém nápovědy (help) nepatrí mezi nejdokonalejší - neexistuje možnost jej doplnit multimediálními informacemi na jedné straně, a nelze jej ani využívat na základě čistě znakového rozhraní (což by potřebovali uživatelé terminálů) na straně druhé.
- * Programátoři nenaleznou žádné vývojové prostředí, které by přesahovalo pouhou kombinaci překladače a ladícího programu. Navíc Windows NT sice umožní běh programů psaných pro MS DOS nebo MS Windows, žádný z nich však nebude moci využívat ani jedinou z nových služeb NT, pokud nebude od základu kompletně přepracován.
- **XINU** je podobně jako MINIX jednoduchým školním operačním systémem vybaveným preemptivním multitaskingem. Čtenář této knihy se s nemalou částí zdrojových textů operačního systému XINU již seznámil; kompletní zdrojové texty lze nalézt v [XINU].

Následující **tabulka** stručně shrnuje nejvýznačnější vlastnosti popsaných operačních systémů. Většina položek tabulky je asi zřejmá; zvláštní pozornost proto věnujeme pouze sloupcům 'ochrana', 'bez.' a 'CLI'.

• Přehled operačních systémů

'Ochrana' je samozřejmě myšlená ochrana paměti. 'CLI' je řádkový interpret příkazů (command line interpreter). Konečně bezpečnost je určena bezpečnostní třídou podle standardů USA; v tabulce se setkáme s těmito třídami:

- D . . . zabezpečení veškeré žádné
- C1 . . bezpečnost závislá na operátorovi - uživatel tedy takový systém dokáže 'zbořit' snadno, ale jestliže si dá pozor, aby to neudělal, nemělo by k tomu dojít.
- C2 . . systém je zabezpečen i před běžnými uživateli; ti mají přidělena omezená práva, která jim neumožňují 'páchat' nebezpečné akce.
- C2+ taková kategorie neexistuje; má pouze naznačit, že operační systém Multics je pravděpodobně ve vyšší třídě než C2 (ale přesné informace se autorovi této knihy nepodařilo získat).

název	multitasking	správa paměti	ochrana	síť	bezp.	GUI	CLI
Amiga DOS	preemptivní, multithreading	bloková	-	-	D	ano	ano
ATARI DOS	-	bloková	zčásti	-	D	-	-
CP/M	-	bloková	-	-	D	-	ano
DOS EC	preemptivní, multithreading	virtuální paměť i adresový prostor	ano	WAN	?		ano
EPOC	preemptivní, multithreading	bloková s transparentním setřásáním	ano	LAN	C1	ano	remote
GEM	omezený kooperativní	bloková	-	-	D	ano	-

Operační systémy

název	multitasking	správa paměti	ochrana	sít'	bezp.	GUI	CLI
MACH	preemptivní, multithreading	virtuální paměť i adresový prostor	ano	LAN			
Macintosh System 7	kooperativní	bloková s kooperativním setřásáním, virtuální		LAN	D	ano	
MINIX	preemptivní	7	9	7	7	-	ano
MSWindows	kooperativní	virtuální?	-	-	D	ano	MS DOS
MS DOS	-	bloková	-	-	D	-	ano
Multics	preemptivní	virtuální	ano	7	C2+	-	ano
NeXTStep	preemptivní, multithreading	virtuální paměť i adresový prostor	ano	LAN WAN	C2	ano	UNIX
OS/2	preemptivní, multithreading	bloková?	ano	-	C1?	ano	MS DOS
OS 360/370	preemptivní, multithreading	virtuální paměť i adresový prostor	ano	WAN	?		ano
Solaris	preemptivní	virtuální?	ano	LAN	C2?	ano	UNIX
ATARI TOS	omezený kooperativní	bloková	zčásti	-	D	GEM	-

- Přehled operačních systémů

název	multitasking	správa paměti	ochrana	síť	bezp.	GUI	CLI
UNIX	preemptivní	virtuální	ano	?	C2	-	ano
WindowsNT	preemptivní	virtuální?	ano?	nestd. LAN	C2?	ano	MS DOS
XINU	preemptivní	bloková	ne	nestd. LAN	D	-	-

Operační systémy

B. Jazyk C

Vzhledem k tomu, že příklady v knížce jsou uváděny v jazyce C vhodném pro tvorbu operačních systémů, zatímco 'programátorská veřejnost' je u nás zvyklá spíše na daleko abstraktnější PASCAL, uvedeme v tomto dodatku velmi stručné srovnání C a PASCALu pro ty, kdo se nechtějí této problematice věnovat podrobněji (ostatní snadno seženou některou z řady učebnic jazyka C - viz např. seznam literatury v dodatku D).

Informace v tomto dodatku nejsou samozřejmě ani zdaleka dostatečné pro toho, kdo by chtěl psát v jazyce C vlastní programy; usnadní však čtení příkladů uvedených v této knize čtenářům, kteří znají pouze jazyk PASCAL (a do jisté míry i těm, kdo neznají ani PASCAL, ačkoli lze předpokládat, že pro ně budou některé pasáže obtížněji pochopitelné).

Poznamenejme, že pro lepší čitelnost jsou klíčová slova v této kapitole tištěna tučnými písmeny. Jedná se o výjimečný případ, jehož účelem je zjednodušit trochu život začátečníkům; nikde jinde tohoto netradičního způsobu nepoužíváme.

B.1 Syntaktické odlišnosti PASCALu a C

Tabulka obsahuje přehled zápisu některých často používaných prostředků jazyka, které si v PASCALu a v C víceméně odpovídají a přitom mají různou syntaxi.

PASCAL	C	příklad v PASCALu	příklad v C
{ }	/ * * /	{ komentář }	/* komentář */
:=	=	sum:=sum+b;	sum=sum+b;
^^	*	i:=intptr^;	i=*intptr;
<>	!=	a<>b	a!=b

Operační systémy

PASCAL	C	příklad v PASCALu	příklad v C
and ¹	&&	(a<x) and (x<b)	a < x & & x < b
or ¹		(x<a) or (b<x)	x < a b < x
not ¹	!	not (x<a)	!(x<a)
and ²	&	hi_byte and \$f0	hi_byte & 0xf0
or ²		lo_byte or \$f0	lo_byte 0xf0
not ²	~	not \$80	~0x80
nil	NULL	ptr:= nil ;	ptr=NULL;
., ^ ., *, ->		clen.jmeno	clen.jmeno
		cptr^.jmeno	cptr->jmeno
		cptr^	*cptr
begin	{	begin	{
end	}	h:=a; a:=b; b:=h end ;	h=a; a=b; b=h; }

¹ Logické operátory ² Bitové operátory

B.2 Datové typy

Datové typy v C v podstatě odpovídají PASCALským; je zde však jeden podstatný rozdíl: PASCALské typy jsou logické, znak v PASCALu je opravdu znakem a tomu odpovídají i operace nad ním. Naproti tomu v C jsou typy jen různými pohledy na uložení dat v paměti. Znak v C je tedy především byte, celé číslo slovo (16 nebo 32 bitů) a podobně. Proto také v C lze např. přiřadit celočíselné proměnné znakovou konstantu (třeba 'A'), která reprezentuje byte, tj. (osmibitové) celé číslo (v kódu ASCII 41h).

Typ **Boolean** v C není, místo něj lze použít libovolný celočíselný typ s konvencí 0 = **false**. Podmíněný příkaz (nebo příkazy typu **while** nebo **repeat**) se tedy od

PASCALu liší tím, že na místě podmínky může stát libovolný celočíselný výraz; je-li nenulový, je chápán jako 'pravdivý' (true).

V C existují odvozené typy *ukazatel*, *pole*, *struktura* a *union* (struktura odpovídá PASCALskému záznamu bez variantní části, union odpovídá záznamu, který má jen variantní část). Na rozdíl od PASCALu v C existuje *ukazatel na funkci*, s nímž lze tvořit složené datové typy (pole ukazatelů na funkci, funkce vracející ukazatel na funkci a podobně).

Jazyk C má speciální typ **void**, který lze použít třemi způsoby:

- deklarujeme-li funkci, která nevrací žádnou hodnotu¹⁴⁰, deklarujeme ji jako by vracela typ **void**;
- deklarujeme-li funkci bez parametrů, uvedeme klíčové slovo **void** na místě seznamu parametrů funkce;
- deklarujeme-li 'ukazatel na **void**', deklarujeme beztypový ukazatel na cokoli.

B.3 Deklarace

Deklarace v jazyce C má ve většině případů obecný tvar '<typ> <deklarátor>'. <typ> je název libovolného typu; <deklarátor> pak syntakticky až na několik výjimek odpovídá výrazu. Potom platí, že deklarovenému identifikátoru je přiřazen takový typ, aby výraz <deklarátor> byl skutečně typu <typ>. Deklarujeme-li tedy proměnnou ptri jako ukazatel na integer, píšeme:

```
int *ptri;
```

Deklarujeme-li pole dvanácti ukazatelů na int arrptri, píšeme:

```
int *(arrptri[12]); /* závorka je zde nadbytečná */
```

¹⁴⁰Problematiku funkcí, které vracejí nebo nevracejí hodnoty, probereme podrobněji v odstavci B.6.

Operační systémy

Deklarujeme-li naopak ukazatel na pole dvanácti integerů ptriarr, píšeme:

```
int (*ptriarr)[12]; /* závorka je zde nutná */
```

Struktura a union se deklarují podobným způsobem, jako PASCALský záznam:

```
struct Complex {  
    float re, im;  
}
```

je tedy struktura, obsahující dvě čísla v plovoucí řádové čárce (vhodná pro práci s komplexními čísly). Podobně

```
struct TwoByte {  
    char c1, c2;  
}
```

je struktura, obsahující dva znaky a

```
union BreakInt {  
    struct TwoByte b;  
    int i;  
}
```

pak dává přístup ke dvěma bytům paměti buď zvlášť (je-li x proměnná typu **union** BreakInt, pak např. x.b.c1 je její první byte), nebo najednou jako k celému číslu (x.i)¹⁴¹.

Deklaraci proměnné může předcházet klíčové slovo **volatile**, které znamená, že tato proměnná může být v multitaskovém prostředí měněna nezávisle na programu (a proto na ní nelze provádět některé optimalizace), klíčové slovo **extern**, které znamená, že proměnná je deklarovaná jinde (obvykle v jiném modulu) nebo klíčové slovo **static**, které zajíšťuje, že jméno proměnné není z modulu publikováno. Pro lokální proměnné lze použít i klíčové slovo **register**,

¹⁴¹Je však zapotřebí si uvědomit, že takováto konstrukce je principiálně nepřenosná, protože bude jinak pracovat na počítači jehož procesor ukládá významnější byty vícebytových hodnot na vyšší adresy (jak to dělají např. mikroprocesory Intel) a jinak na počítači, který ukládá významnější byty na nižší adresy (jak to dělají např. mikroprocesory Motorola).

které upozorňuje překladač, že proměnná je intenzivně využívána a je-li to možné, měla by být realizována jako rychlý registr procesoru.

B.4 Operátory

Pro konstrukci *numerických výrazů* jsou v C o něco silnější prostředky než v PASCALu. Kromě běžných aritmetických operátorů má C operátory bitového NOT (~), AND (&), OR (), XOR (^,nonekvivalence) a bitových posunů (>> a <<, druhým operandem je počet bitů). Přiřazovací 'příkaz' v C má hodnotu (tu, která se přiřadila levé straně) a lze jej tedy použít ve výrazech. C má (podobně jako ALGOL 60) podmíněný výraz - hodnotou výrazu se syntaxí '<podmínka>?<výraz1>:<výraz2>' je <výraz1>, je-li podmínka pravdivá, jinak <výraz2>.

Protože aritmetický výraz v C je zároveň logickým a naopak (nula je false a nenulová hodnota true), lze k aritmetickým operátorům řadit i operátory pro konjunkci (&&), disjunkci (||) a negaci (!). Hodnotou nepravdivého výrazu s logickými operátory je nula, hodnota pravdivého je nenulová (obvykle 1 nebo -1). Vyhodnocování termů spojených logickými operátory se ukončí ve chvíli, kdy je zřejmá pravdivost nebo nepravdivost celého výrazu; termy se tedy nemusí vždy vyhodnotit všechny (jako je tomu v PASCALu).

Pro zjednodušení práce s poli obsahuje jazyk C operátory autoinkrementace a autodekrementace. Výraz 'i++' má stejnou hodnotu, jako výraz 'i'; navíc však zvýší hodnotu proměnné i o 1. Naopak výraz '+j' má hodnotu proměnné j po přičtení jedné. Analogicky lze používat výrazy pro autodekrementaci tvaru '<proměnná>--' a '--<proměnná>'.

Většinu binárních operátorů lze v C spojit s operátorem přiřazení, je-li prvním operandem cílová proměnná. Místo 'i=i+1' tak lze psát 'i+=1'; obecně zápis

<proměnná> <oper>= <výraz>

je ekvivalentní zápisu

Operační systémy

<proměnná> = <proměnná> <oper> <výraz>

Flexibilitu výrazů v C ještě zvyšuje vzájemná *převeditelnost typů*. Ukazatel je adresa, a tedy celé číslo; proto je na celé číslo plně převeditelný¹⁴². Podobně znak lze chápat jako celé číslo s rozsahem hodnot 0 až 255 (nebo -128 až 127, je-li pomocí modifikátoru **signed** deklarován jako znaménkový). Kromě toho lze proměnnou kteréhokoli typu explicitně převést na libovolný jiný typ: zápis '<jméno typu>'<výraz>' je výrazem typu <jméno typu>, který má hodnotu <výrazu> (po případných konverzích).

B.5 Příkazy

Funkci příkazových závorek (v PASCALu **begin** a **end**) v jazyce C zastávají složené závorky { a }. Kterýkoli složený příkaz v C se může stát blokem, tj. na jeho začátku mohou stát deklarace proměnných v něm lokálních (to umožňoval např. ALGOL 60).

V popisu každého příkazu je uveden jednoduchý příklad.

B.5.1 Příkaz if

Příkaz má stejnou sémantiku jako PASCALské **if**; příklady:

```
if (i>120) i=120;          /* zarovnej i na horní mez */
if (c>=' ') zpracuj_znak(c); /* odlišení kontrolních znaků */
else zpracuj_ctrlchar(c);
```

Na místo podmínky může stát (jako kdekoli v C) celočíselný výraz, před klíčovým slovem **else** se píše středník (v jazyce C středník ukončuje příkaz, na rozdíl od PASCALu, kde je středník oddělovačem příkazů).

¹⁴²Na dvaatřicetibitových systémech můžeme ukazatel pohodlně převádět na číslo typu **int**. Překládáme-li však program pro šestnáctibitový počítač, musíme obvykle pro ukazatel použít číslo typu **long**.

B.5.2 Příkaz for

Příkaz cyklu s řídící proměnnou (for) v C dovoluje zadat libovolnou podmínu ukončení cyklu a libovolnou iteraci řídící proměnné. Jeho tvar ilustrujeme příkladem:

```
sum=0;
for (i=0; i<ARRSIZE; i++)          /* sečti prvky pole a */
    sum+=a[i];
```

kde první výraz ('i=0') nastavuje výchozí podmínky cyklu, druhý výraz je podmínkou ukončení: cyklus probíhá, dokud platí 'i<ARRSIZE'. Konečně výraz 'i+' je proveden po každém průchodu cyklem a zajišťuje iteraci. Tělem cyklu samozřejmě může být i složený příkaz.

B.5.3 Příkazy while, do

```
while (a[i]!='x') i++;           /* hledej 'x' v poli */
do
    c=getch();                  /* čekej na vstup 'y/n' */
    /* čte znak */
while (c!='y' && c!='n');
```

odpovídají PASCALskému **while** a **repeat** s jedinými dvěma rozdíly: na místě podmínky může stát (jako kdekoli jinde v C) libovolný celočíselný výraz; cyklus **do - while** končí při nesplnění podmínky (**repeat** naopak při splnění).

B.5.4 Příkaz switch

PASCALskému **case** v C odpovídá příkaz **switch**, který však jen předá řízení na vybrané návěští (jako příkaz **goto**) a od něj provádí program dál (na rozdíl od PASCALského **case**, které provede jen vybraný příkaz). Návěští příkazu **switch** v jazyce C je označeno klíčovým slovem **case**, pro akci při nesplnění

Operační systémy

žádné z podmínek lze použít návěští **default**. Chceme-li ukončit celý příkaz **switch** před dosažením jeho konce, musíme explicitně použít příkaz **break**:

```
switch (i) {  
    case 1 : printf("i je jedna");break;  
    case 2 : printf("i je dvě");break;  
    case 3 : printf("i je tři a ne ");  
    default: printf("jiné číslo !");
```

Je-li tedy $i=3$, vypíše se "i je tři a ne jiné číslo !", protože po předání řízení na návěští **case 3** se program provádí běžným sekvenčním způsobem dál.

B.5.5 Příkaz goto

V C je i často diskutovaný příkaz **goto**; návěští mohou být alfanumerická a není třeba je deklarovat. C navíc obsahuje příkazy **break** a **continue** pro opuštění cyklu před splněním ukončovací podmínky a pro opuštění funkce před dosažením jejího konce (nejčastější případy použití **goto** v PASCALu).

B.5.6 Příkazy break a continue

Příkaz **break** způsobí okamžité ukončení nejvnitrnějšího cyklu nebo příkazu **switch**. Příkaz **continue** způsobí okamžité ukončení těla nejvnitrnějšího cyklu (platí-li však stále iterační podmínka, cyklus není ukončen).

B.5.7 Příkaz return

Příkaz **return** ukončí funkci. Je-li uveden s parametrem, zajistí, že funkce vrací hodnotu parametru. Dosažení konce funkce je ekvivalentní příkazu **return** bez parametru.

B.6 Funkce

Mechanismus volání funkcí v C se od PASCALu liší ve dvou bodech: C umožňuje deklarovat funkce s *proměnným počtem i typy parametrů* a nerozlišuje funkce a procedury (je zvykem používat v obou případech termín 'funkce'). Každá procedura (=funkce) v C tedy vrací hodnotu; u každé funkce (=procedury) v C lze vrácenou hodnotu ignorovat, tj. zavolat funkci jako příkaz¹⁴³. Píšeme-li funkci, která v žádném případě vracet hodnotu nemá, deklarujeme jí jako by vracela 'hodnotu typu **void**'. Překladač pak hlídá, nepoužíváme-li návratovou hodnotu omylem ve výrazech.

Parametry funkcí jsou v C předávány zásadně hodnotou, parametrem nesmí být funkce; je-li parametrem pole, předává se automaticky ukazatel na jeho první složku. Parametrem však může být libovolný ukazatel - lze si tedy explicitně předepsat volání 'čehokoliv' referencí (C má operátor, který zjišťuje adresu objektu (= ukazatel na něj)). Parametrem může být struktura, je však obecně vhodnější předávat ukazately na struktury.

B.6.1 Hlavičky funkcí

ANSI C umožňuje v deklaraci funkce zadat v seznamu parametrů i jejich typy; překladač pak při volání kontroluje shodu počtu parametrů a provádí případné konverze. Funkci, která má dva argumenty typu **long** a vrací číslo typu **float**, tedy deklarujeme takto:

```
float proc(long i,long j)
{ .... }
```

Potřebné konverze při případném volání funkce s argumenty typu **int** nebo **char** a v podobných případech se provedou automaticky.

¹⁴³Jedná se o obecnou vlastnost jazyka C: libovolný výraz se může stát příkazem, ukončíme-li jej středníkem. Např. '1 + 1;' je tedy také korektní příkaz (nedělá ovšem nic a správný překladač jej ignoruje). Často se ale setkáme s takto tvořenými příkazy na základě přiřazovacího výrazu ('i = 10;') nebo operátorů inkrementace a dekrementace ('i++ ; j--;').

B.7 Preprocesor

Překladač jazyka C standardně obsahuje poměrně výkonný textový preprocesore s možností *podmíněného překladu* a definování *makroinstrukcí s parametry*. Makropresor je silný a často užívaný prostředek. Definice makra bez parametrů:

```
#define <jméno> <alias>
```

V dalším textu programu je pak každé <jméno> (mimo komentáře a řetězce) nahrazeno <alias>em. Je-li <jméno> následováno seznamem formálních parametrů, jsou při každém rozvoji makra lexikálně nahrazeny skutečnými parametry. Jako příklad makro

```
#define max(a,b) ((a)>(b)?(a):(b))
```

vrátí hodnotu většího ze dvou čísel.

Podmíněný překlad nabízí (mimo jiné) direktivy #ifdef, #else a #endif, které umožňují podmínit překlad definovaností (tj. existencí) symbolu:

```
#ifdef EGA_PRESENT
#define MAX_LNE 42           /* číslo poslední řádky obr.*/
#else
#define MAX_LNE 24
#endif
```

Hodnota konstanty MAX_LNE je určena v závislosti na tom, je-li při překladu definován symbol EGA_PRESENT.

B.8 Inicializace

Všechny globální proměnné a jednoduché lokální proměnné je možno inicializovat tak, že se za deklarátor zapíše konstrukce '=výraz' projednoduché a '= {výraz1, ... }' pro složené proměnné:

```
int i=5;
```

```
int a[5]={1,2,3,4,5};
```

Lokální proměnnou je možno inicializovat libovolným výrazem. Jsou-li např. i a j globální proměnné, lze v proceduře deklarovat:

```
proc()
{
    int k=2*i+j;
    ....
}
```

Globální proměnné je samozřejmě nutné inicializovat konstantními výrazy.

Operační systémy

C. Strojově závislá rozšíření Turbo C

Pro implementaci operačního systému XINU na počítače třídy IBM PC bylo využito toho, že překladač Turbo C nabízí některé speciální služby usnadňující spolupráci s mikroprocesorem Intel, kterým jsou počítače IBM PC osazeny. Díky tomu bylo možné vytvořit jádro operačního systému kompletně ve vyším jazyce, bez použití jediné instrukce assembleru¹⁴⁴.

Abychom usnadnili pochopení těchto příkladů i čtenářům, kteří neznají překladače Turbo C, uvádíme velmi stručný popis použitých rozšíření.

C.1 Funkce typu interrupt

Funkce v jazyce Turbo C může mít specifikátor **interrupt**, který zajistí, že funkce se přeloží jako *handler přerušení*: před začátkem práce uloží na zásobník všechny registry (a po ukončení je opět obnoví) a končí instrukcí IRET. Implementace XINU pro počítače třídy IBM PC využívá funkce typu **interrupt** např. k provedení výměny kontextu.

Registry se na zásobník ukládají v pořadí AX, BX, CX, DX, SI, DI, ES, DS a BP. Nad registrem AX je ještě uložen standardní záznam, ukládaný na zásobník mechanismem přerušení mikroprocesoru: (dlouhá) návratová adresa a registr příznaků.

¹⁴⁴Tím jsme získali na přehlednosti; samozřejmě, že úseky kódu psané pomocí speciálních rozšíření Turbo C jsou principiálně nepřenositelné, stejně jako kdyby byly psány v assembleru.

Operační systémy

C.2 Práce na strojové úrovni

V operačním systému potřebujeme mít přímý přístup k registrům procesoru a k operační paměti. Turbo C pro tyto účely nabízí následující prostředky:

C.2.1 Přístup k registrům

Překladač Turbo C dává uživateli přímý přístup k registrům prostřednictvím *pseudoproměnných*. Pseudoproměnné **_AX**, **_BX**, **_CX**, **_DX**, **_SI**, **_DI**, **_BP**, **_SP**, **_FLAGS**, **_CS**, **_SS**, **_DS** a **_ES** odpovídají registrům AX, BX, ... ,ES. Mohou být kdykoli čteny a při dodržení konvencí Turbo C (např. neměnit uvnitř funkce obsah registru BP, pomocí nějž se přistupuje k lokálním proměnným) i měněny.

C.2.2 Přístup k paměti

Turbo C nabízí pro přístup do paměti funkce **poke** a **peek**; navíc je možno použít standardní přístup s využitím ukazatelů:

```
c=*(char *)0; /* čti do c obsah adresy 0 */
```

C.2.3 Strojový kód

V některých případech potřebujeme přece jen použít nějakou instrukci strojového kódu - typicky se jedná o instrukce ovládající přerušení. Instrukce pro zákaz a pro povolení přerušení je možné v Turbo C zapsat pomocí předdefinovaných maker

```
disable(); /* zákaz přerušení */
enable(); /* povolení přerušení */
```

Pro potřebu operačního systému XINU tato makra nestačí; tam je totiž zapotřebí při zákazu přerušení uložit jeho původní stav do pomocné proměnné,

a namísto povolení přerušení pak pouze obnovit původní stav (důvody pro toto řešení jsou podrobně rozebrány v odstavci 6.2.1.4).

Turbo C naštěstí nabízí dostatečně silné prostředky i pro realizaci tohoto úkolu - stačí, vytvoríme-li potřebná makra následujícím způsobem:

```
#define INT_FLAG 0x200 /* poloha bitu IF v registru FLAGS */

#define sdisable(x) { /* stav->x, disable */ \
    x=_FLAGS; \
    disable(); \
}

#define restore(x) /* obnov podle x, je zakázáno */ \
if (x&INT_FLAG) \
    enable()
```

První makro pouze uloží momentální nastavení přepínačů do určené proměnné a pak zakáže přerušení. Druhé makro - využívající toho, že je voláno pouze při zakázaném přerušení - pak jen ověří, je-li třeba přerušení povolit nebo ne, a v kladném případě jej povolí.

D. Literatura

Mnoho dalších zajímavých informací o různých operačních systémech a o související problematice čtenář nalezne v následující literatuře. Zvláště kvalitní a 'doporučenihodné' knihy jsou označeny hvězdičkou:

- [386] . . . O. Čada: **80386, 80387, příručka programátora**, PLUS, Praha 1991
- [68030] O. Čada: **Mikroprocesor Motorola 68030**, Grada, Praha 1992
- [CAKE]..... Ed Post: **Opravdoví programátoři neužívají PASCAL**, podklad pro panelovou diskusi, 1983
- [DOS] . . . O. Čada: **MS-DOS 5.0, příručka uživatele**, PLUS, Praha 1991
- [HOR] . . . J. Hořejš, J. Brodský, J. Staudek: **Struktura počítačů a jejich programového vybavení**, SNTL, Praha 1981
- [JSEP] . . . V. Navrátil, J. Sokol, V. Žák: **Operační systémy JSEP**, SNTL, Praha 1984
- *[K&R]..... B. W. Kerningham, D. M. Ritchie: **The C Programming Language**, Prentice-Hall 1978
- [OCC] . . . O. Čada: **Učebnice jazyka Turbo C**, PLUS, Praha 1990
- [MAC]_____ O. Čada: **Apple Macintosh**, Grada, Praha 1993
- *[OS] . . . Stuart E. Madnick, John J. Donovan: **Operační systémy**, SNTL, Praha 1983
- [ST] . . . Petr Jandík: **ATARI ST/TT, obsluha a programové vybavení**, Grada, Praha 1992
- [UNIX] Jan Brodský, Luděk Skočovský: **Operační systém Unix a jazyk C**, SNTL, Praha 1989
- [UXPC] . . . Jan Brodský, Luděk Skočovský: **Unix na osobních počítačích typu IBM PC - srovnávací studie**, sborník MOP '88, MFF UK Praha 1988
- *[XINU] . . . Douglas Comer: **Operating System Design - The XINU Approach**, Bell Laboratories, London 1984

Množství firemní literatury firem Apple, IBM, Microsoft, PSION a NeXT.

Operační systémy

Rejstřík

- Adresáře 201
- Algoritmus
- bankéřův 156
 - detekce zablokování 159
 - LRU 57-59, 345
- Apple Macintosh 18, 42-45, 47, 59, 74, 75, 204, 302, 306, 317, 343, 344, 349, 352
- B**
- Bezpečnost
- globální síť 263
 - lokální síť 262
 - multitaskingu 71, 171
 - ochrana paměti 33, 36, 45, 47, 50-52, 66-68, 171, 339, 340, 341, 351
 - ovladače zařízení 192
 - soubory 203
 - třídy zabezpečení 351
- C**
- C 355
- D**
- Deadlock 82, 124, 125, 129, 131, 152-157, 159-163, 221, 227, 246
- Delta list 87, 88, 91, 107-110, 164, 167-170
- Dialogy 319
- E**
- EPOC 18, 28, 42, 47-49, 71, 152, 283, 297, 303, 306, 341, 342, 351
- F**
- Fragmentace paměti 37-40, 46, 47, 50, 61
- G**
- GEM 74, 289, 301, 302, 306, 309, 317, 339, 342, 348, 351, 352
- Grafická nadstavba
- GEM 74, 289, 301, 302, 306, 309, 317, 339, 342, 348, 351, 352
 - MS Windows 344, 352

Operační systémy

- X Windows 303, 342, 348, 349
 - Ikony 316
 - Interpret příkazů 331
 - Kontext 75
 - Kritické sekce 110-113, 117, 118, 119-121, 124, 128, 129, 144, 145, 147, 163, 165, 168
 - M
 - Macintosh 18, 42-45, 47, 59, 74, 75, 204, 302, 306, 317, 343, 344, 349, 352
 - Menu 317
 - MS DOS 17, 34, 75, 200, 201, 340, 344, 352
 - MS Windows 344, 352
 - Multitasking
 - definice 69
 - kontext 75
 - kooperativní 45, 71, 73, 78, 79, 80, 81, 83, 283, 342, 343, 348, 351, 352
 - kritické sekce 110-113, 117, 118-121, 124, 128, 129, 144, 145, 147, 163, 165, 168
 - nulový proces 91, 147, 224
 - preemptivní 71, 81-83, 130, 143, 164, 171, 283, 339, 341, 342, 346, 347, 350, 351, 352, 353
 - princip funkce 73
 - priority 106
 - proces 76
 - round robin 95
 - sdílení času 83, 89, 94, 95, 106, 164, 168
 - semafory 111-119, 121, 124, 128, 129, 135, 142, 143, 144, 145, 148, 149, 151, 153, 159, 160, 163, 194
 - stavy procesu 89
 - výhody a nevýhody 69
 - zabezpečení 71
 - zablokování 82, 124, 125, 129, 131, 152-157, 159, 160, 161-163, 221, 227, 246
 - zprávy 121-123, 151, 194, 203, 221, 222, 224, 229, 239, 240, 249, 250, 267, 271
- N
- Nabídky 317
 - Národní prostředí 275
 - NeXTStep 18, 27, 28, 174, 219, 280, 283-285, 303, 306, 318, 326, 342, 343, 346, 347, 352
 - Nulový proces 91, 147, 224

- Objektové programování 26-28, 30, 265, 268, 271, 273, 274, 281, 336, 341, 343, 346
- Ochrana paměti 33, 36, 45, 47, 50, 51, 52, 66-68, 171, 339, 340, 341, 351
- Okna 295, 307
- Operační systém
 - Amiga DOS 339
 - ATARI TOS 339
 - CP/M 340
 - definice 24
 - DOS EC 341
 - EPOC 18, 28, 42, 47-49, 71, 152, 283, 297, 303, 306, 341, 342, 351
 - MACH 342
 - Macintosh 18, 42-45, 47, 59, 74, 75, 204, 302, 306, 317, 343, 344, 349, 352
 - MINIX 344
 - MS DOS 17, 34, 75, 200, 201, 340, 344, 352
 - Multics 345
 - NeXTStep 18, 27, 28, 174, 219, 280, 283-285, 303, 306, 318, 326, 342, 343, 346, 347, 352
 - OS 360/370 347
 - OS/2 347
 - Solaris 347
 - TOS+GEM 348
- UNIX 18, 27, 196, 204, 205, 348, 349, 352, 353
- Windows NT 27, 219, 335, 341, 349, 350, 353
- XINU 17, 27, 29, 34, 83, 84, 87-89, 93, 102, 104, 106, 111, 115, 122, 123, 136, 138, 164, 166, 176, 178, 180, 181, 189, 219, 350, 353, 367, 368
- Ovladače zařízení
 - časovač 164
 - disk 176
 - dolní polovina 148
 - horní polovina 140
 - logický systém 136
 - popis 132
 - sítě 223
 - služby 133
 - zabezpečení 192
- Paměť
 - alok. strategie 38
 - fragmentace 37-40, 46, 47, 50, 61
 - ochrana 33, 36, 45, 47, 50, 51, 52, 66-68, 171, 339, 340, 341, 351
 - přiděl. po blocích 34, 37
 - setřásání 39-50, 341, 351, 352
 - správce 33-48, 50-52, 54, 55, 56, 61-63, 85, 101, 102, 105, 128, 195, 341, 345

Operační systémy

virtuální 50, 51, 54, 56, 58, 59, 60-64, 195, 265, 269, 271, 272, 341, 345-347, 351-353

Přístupová práva 203

Proces 76

S

Schránka 282

datové služby 285

drag-and-drop 284

inteligentní 283

spojení dat 284

Sdílené knihovny 63, 67, 132, 152, 217, 268, 271-274, 336, 341

Sdílení času 83, 89, 94, 95, 106, 164, 168

Semafory 111-119, 121, 124, 128, 129, 135, 142-145, 148, 149, 151, 153, 159, 160, 163, 194

binární 112

obecné 114

Sítě 207

globální 218

linková vrstva 223

lokální 207

síťová vrstva 245

WAN 218

zabezpečení 262

Soubory 200

formátované 202, 279

sdílení 202

zabezpečení 203

Správce

času 107, 164

front 86

paměti 33-48, 50-52, 54-56, 61-63, 85, 101, 102, 105,

128, 195, 341, 345

procesů 52, 63, 85, 86, 89, 91, 92, 93, 96, 97, 101, 106, 107, 109, 125, 217

prostředků 156

sítě 67

úloh 125, 152, 153

zařízení 128

Správce paměti 33-48, 50-52,

54-56, 61-63, 85, 101,

102, 105, 128, 195, 341,

345

Stavy procesu 89

U

UNIX 18, 27, 196, 204, 205,

348, 349, 352, 353

Virtualizace

adres 63

paměti 58

zařízení 129

Virtuální paměť 50, 51, 54, 56,

58, 59-64, 195, 265, 269,

271, 272, 341, 345-347,

351-353

W

Windows 344, 352
Windows NT 27, 219, 335, 341,
349, 350, 353

X Windows 303, 342, 348, 349
XINU 17, 27, 29, 34, 83, 84,
87, 88, 89, 93, 102, 104,
106, 111, 115, 122, 123,
136, 138, 164, 166, 176,
178, 180, 181, 189, 219,
350, 353, 367, 368

Zablokování 82, 124, 125, 129,
131, 152-157, 159-163,
221, 227, 246

Zařízení
sdílené 131
společné 131
vyhrazené 128

Zprávy 121-123, 151, 194, 203,
221, 222, 224, 229, 239,
240, 249, 250, 267, 271

Ondřej Čada

Operační systémy

Šéfredaktor Jiří Škácha

Odpovědný redaktor Jaroslav Foršt

Odborná recenze Tomáš Hůrka

Grafická úprava obálky Karel Kárász

Foto Vladimír Svoboda

Počet stran 384

Vydala Grada, a.s., v Praze roku 1994