Katedra informatiky, FEI VŠB-TUO, Petr Olivka.

Tento text je neautorizovaný a nerecenzovaný překlad doporučené literatury: „Andrew S. Tanenbaum, Operating Systems: Design and Implementation“, a je určen jen pro studijní účely.

2 Procesy

Teď je čas popsat si podrobněji operační systém. Obecný pohled, částečně MINIX, návrh a konstrukci. Hlavním bodem koncepce systému je nejčastěji *proces* - volněji řečeno, bežící program. Všechno ostatní se jen odvíjí z tohoto konceptu a je důležité, aby návrháři systému věděli co to je proces, co nejdříve je to možné.

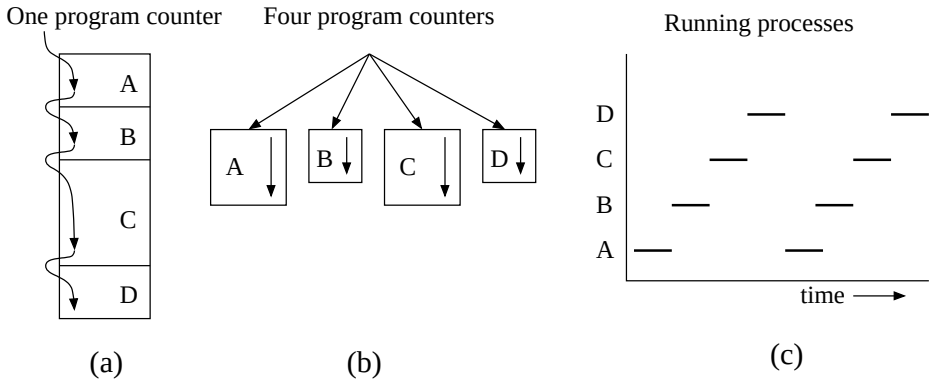
2.1 Úvod do procesů

Všechny moderní počítače musí provádět více věcí současně. Zatím, co běží program uživatele, počítač může číst data z disku, vypisovat text na obrazovku a tisknout na tiskárnu. V multiprogramovém systému se CPU také přepíná mezi programy po desítkách nebo stovkách milisekund. Jinak řečeno, v jakémkoliv čase CPU provádí právě jen jeden program a během jedné sekundy může zpracovávat více programů, což vyvolává v uživateli dojem paralelismu. Někdy se mluví o **pseudoparalelismu** jako rychlém a neustálém přepínání CPU mezi programy. Na rozdíl od skutečného paralelismu hardware v **multiprocesorových** systémech (s více procesory a sdílenou pamětí). Souběžná starost člověka o více paralelních aktivit je pro něj velmi náročná. Proto návrháři operačních systémů během let vyvinuli sekvenční model, který umožňuje snadné zvládnutí paralelismu. Tento model a jeho uživatelé jsou tématem této kapitoly.

2.1.1 Model procesu

V tomto modelu jsou všechny spustitelné programy, včetně operačního systému, organizovány jako řada **sekvenčních procesů**, zkráceně jen **procesů**. Proces je běžící program společně s hodnotami programového čítače, registrů a proměnných. Obecněji má každý proces svůj vlastní virtuální CPU. Reálně ovšem CPU neustále přepíná mezi procesy, ale pro pochopení systému je daleko snazší přemýšlet o skupině procesů bežících pseudoparalelně, než pečlivě sledovat, jak se CPU přepíná z programu do programu. Tento systém rychlého a neustálého přepínání se nazývá **multiprogramový** a byl probrán dříve.

Na obrázku 1(a) vidíme multiprogramový počítač se čtyřmi programy v paměti. Na obrázku 1(b) jsou čtyři procesy, každý s vlastním řízením běhu, tedy čítačem instrukcí, a každý z nich pracuje nezávisle na ostatních. Obrázek 1(c) ukazuje pohled v krátkém časovém úsek, jak jsou tyto procesy



Obrázek 1: (a) Multiprogramování čtyř procesů (b) Model 4 nezávislých sekvenčních procesů (c) Vždy je aktivní jen jeden program

vykonávány a je vidět, že v každém okamžiku je zpracováván právě jen jeden program.

Při neustálém přepínáním CPU mezi procesy, nebude postup přepínání stejnoměrný a pravděpodobně ani nebude opakovatelný, pokud stejné procesy spustíme znovu. Proto procesy nesmí být programovány s pevnou vazbou na čas. Mějme například V/V proces, který spustí páskovou mechaniku na obnovení zálohovaných souborů, počká 10000 krát v čekací smyčce, než se páska roztočí a pak přijde první příkaz pro čtení dat z prvního záznamu. Pokud CPU rozhodne, že během této čekací smyčky přepne na jiný proces, může proces pro čtení z pásky po návratu do čekací smyčky promeškat první záznam na pásce pro čtení. Pokud má proces kritické nároky na požadavky v reálném čase, musí přijít určitá událost během specifikovaného časového intervalu v řádu milisekund a speciálním měřením musí být zajištěno, že se tak stalo. Obvykle ale naštěstí většina procesů není dotčena principem multiprogramové činnosti CPU, nebo rychlostí ostatních procesů.

Rozdíl mezi procesem a programem je malý, ale zásadní. Můžeme si pomoci analogií, abychom to lépe vysvětlili. Mějme počítačového vědce, kterého přepadla chuť upéct své dceři narozeninový dort. Má recept na narozeninový dort a dobře vybavenou kuchyň se vším potřebným: moukou, cukrem, vajíčky, vanilkovým aroma atd. V této analogii je recept programem, mohli bychom říct i algoritmem, počítačový vědec je procesor a ingredience jsou vstupními daty. Proces je aktivita sestávající z čtení receptu na pečení, výběrem ingrediencí a pečením dortu.

Nyní si představme, že syn počítačového vědce přijde s pláčem, že jej píchla včela. Počítačový vědec zaznamená kde skončil v čtení receptu, vezme příručku první pomoci a bude provádět to, co je v ní uvedeno. Zde vidíme přepnutí procesoru z jednoho procesu (pečení) na proces s vyšší prioritou (první pomoc), přičemž tyto procesy mají jiný program. Když bude včelí

žihadlo vytaženo, počítačový expert se vrátí ke svému dortu a bude pokračovat od místa, kde přestal.

Klíčovou myšlenkou je, že proces je aktivita libovolného druhu. Ta má svůj program, vstup, výstup a stav. Jeden procesor může být sdílen mezi více procesy s určitým plánovacím mechanismem, rozhodujícím, kdy zastavit práci na jednom procesu a přejít na jiný.

Hierarchie procesů

Operační systém založený na koncepci procesů musí poskytnout nějakou metodu na vytvoření všech potřebných procesů. Ve velmi jednoduchém systému, nebo v systému navrženém pro spuštění jen jednoho procesu, je možné mít všechny potřebné procesy spuštěné ihned po startu systému. Nicméně, ve většině systémů je potřeba mít cestu, jak spouštět a ukončovat procesy průběžně během práce. V MINIXu se proces vytváří systémovým voláním *fork*, které vytvoří identickou kopii volajícího procesu. Potomek také může také volat *fork* a je tak možné vytvořit celý strom procesů. V operačním systému jsou systémová volání na vytvoření procesu, natažení programu do paměti a spuštění. Vše, co od systémových volání potřebujeme, je možnost procesu, vytvořit další proces. Nutno dodat, že každý proces má jednoho rodiče a žádného, jednoho, dva nebo více potomků.

Jako jednoduchý příklad, jak se strom procesů používá v praxi, je inicializace MINIXu. Speciální proces nazývaný *init* je obsažen ve startovacím obraze (boot image). Když se spustí, přečte soubor říkající, kolik terminálů má mít. Pak rozdvojením (fork) vytvoří pro každý terminál nový proces. Tyto procesy čekají, až se někdo přihlásí. Když je přihlášení úspěšné, přihlašovací proces spustí shell a přijímá příkazy. Tyto příkazy mohou spustit více procesů, atd. Všechny procesy v celém systému tedy tvoří jeden strom s procesem **init** jako kořenem.

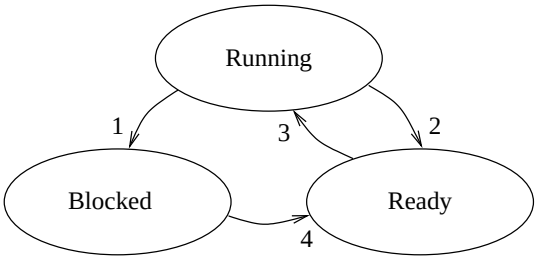
Stavy procesu

Ačkoli je proces nezávislou jednotkou, s vlastním čítačem instrukcí a vnitřními stavy, procesy potřebují komunikovat s ostatními procesy. Jeden proces může generovat výstup jako vstup do dalšího programu. Například příkaz:

`cat file1 file2 file3 | grep tree`,

kde první proces spustí *cat* a spojí všechny soubory do jednoho na výstup. Druhý proces spustí *grep* a vybere všechny řádky obsahující slovo „tree“. V závislosti na relativní rychlosti obou procesů se může stát, že proces *grep* je připraven, ale nemá žádná vstupní data. Musí být tedy **zablokován**, dokud nebudou vstupní data k dispozici.

Pokud dojde k zablokování procesu, tak je to logicky proto, že nemohl pokračovat. Je také možné, že proces je připraven a schopen běžet, ale rozhodnutím operačního operačního systému je zastaven a CPU je přiděleno jinému procesu. Tyto dva stavy jsou zásadně odlišné. V prvním případě je



Obrázek 2: Stavy procesu a přechody mezi nimi

důvod zastavení dán problémem. Ve druhém případě je to dáno technicky, či organizačně, operačním systémem. Na obrázku 2 vidíme tři stavy, ve kterých se může proces nacházet:

- Running - bežící - právě využívá CPU,
- Ready - připravený - dočasně pozastavený během jiného programu,
- Blocked - zablokovaný - nemůže běžet, dokud nepřijde nějaká externí událost.

Logicky jsou první dva stavy podobné. V obou případech je proces ochoten běžet, ale ve druhém případě je dočasně bez CPU. Třetí stav je odlišný od prvních dvou, proces nemůže běžet, i když CPU nebude mít co na práci.

Mezi třemi stavy jsou čtyři možné přechody. Přechod 1 nastane, když proces zjistí, že nemůže pokračovat. V některých systémech musí program zavolat systémové volání BLOCK, aby se do tohoto stavu dostal. V jiných systémech, včetně MINIXu, když proces čte ze souboru nebo speciálního souboru a nejsou dostupná data, dojde k zablokování automaticky.

Přechod 2 a 3 je vyvolán plánovačem procesů, jakožto součásti operačního systému, aniž by o tom procesy věděly. Přechod 2 nastane, když plánovač rozhodne, že proces již běžel dostatečně dlouho a je čas předat CPU jinému procesu. Přechod 3 nastane, pokud je plánovačem proces vybrán jako ten, komu bude předán CPU. Samotné plánování, tedy rozhodování, kdy který proces poběží a jak dlouho, je velice důležité a podíváme se na něj později. Mnoho algoritmů se snaží vyřešit rovnováhu mezi soutěžením požadavků a efektivitou v systému, stejně jako férovost vůči jednotlivým procesům.

Přechod 4 nastane, když přijde očekávaná externí událost, na kterou proces čekal. Pokud právě neběží žádný proces, přejde proces přechodem 3 ihned do stavu běžícího. Jinak bude ve stavu *připravený* čekat po krátkou dobu na přidělení CPU.

Použitím tohoto modelu bude mnohem snazší přemýšlet na tím, co se uvnitř systému děje. V některých procesech běží programy, kde vstupují pří-

kazy od uživatele. Jiné procesy, jako součást operačního systému, zajišťují požadavky na služby a obsluhu detailního ovládání disků nebo pásek. Když přijde přerušení od disku, systém zastaví aktuální běžící proces a spustí diskový proces, který byl zablokován čekáním na toto přerušení. Tedy místo úvah o přerušení, můžeme uvažovat o diskových procesech, terminálových procesech, uživatelských procesech a dalších, které jsou zablokovány čekáním, až se něco stane. Když je blok z disku načten, nebo stisknuta klávesa, proces čekající na odblokování se stává znovu spustitelným.

Tento pohled přináší model, kdy na nejnižší úrovní je plánovač s mnoha různorodými procesy nad ním. Správa všech přerušení a podrobnosti o právě běžících a zastavených procesech je skryta v plánovači, který je v podstatě velmi malý. Zbytek operačního systému je jednoduše strukturován do formy procesů. Plánovač tedy neznamená jen plánovaní procesů, ale také obsluhu přerušení a veškerou mezi-procesní komunikaci. Nicméně, je to první přiblížení, ukazující základní strukturu.

2.1.2 Implementace procesů

Pro implementaci modelu založeného na procesech musí mít operační systém tabulku, nebo pole struktur, které říkáme **tabulka procesů** (process table), s jedním záznamem pro každý proces. Tento záznam obsahuje informace o alokaci paměti, status otevřených souborů, účtovací a plánovací informace a vše o procesu, co musí být uloženo, když se proces přepíná ze stavu *běžící* do stavu *připravený*, tak aby mohl být později znovu spuštěn, jako by nikdy nebyl zastaven.

V MINIXu je správa procesů, paměti a souborů, implementována každá zvlášť v samostatném modulu uvnitř systému tak, že tabulka procesů je rozdělena na bloky a každý modul se stará o svoje položky v tabulce. V tabulce 1 je uvedena většina nejdůležitějších položek. Pouze položky prvního sloupce se týkají této kapitoly. Další sloupce jsou uvedeny proto, aby byla jasnější představa, co je potřeba jinde v systému.

Nyní, když už máme představu o tabulce procesů, je možné si podrobněji vysvětlit, jak se realizuje iluze více sekvenčních programů na počítači s jedním CPU a více V/V zařízeními. Co následuje, je technický popis, jak pracuje plánovač v MINIXu, ale ve většině moderních operačních systémů se chová v podstatě stejně. Spojení s V/V zařízeními je uloženo na začátku paměti - **vektor přerušení**. Obsahuje adresy podprogramů obsluhy jednotlivých zařízení. Předpokládejme, že uživatelský proces běží a přijde žádost o přerušení. Čítač instrukcí, status procesu a několik registrů je uloženo na aktuální zásobník ovladačem přerušení. To je vše, co udělá harware. Odtud výše už vše patří programům.

Obsluha služeb přerušení začne uložením všech registrů do záznamu v tabulce procesů aktuálnímu procesu. Aktuální číslo procesu a ukazatel na něj je uložen v globální proměnné, aby mohl být rychle vyhledán. Pak jsou

Správa procesů	**Správa paměti**	**Správa souborů**
registry	ukazatel kódového segm.	UMASK
instrukční ukazatel	ukazatel datového segm.	root dir
status	ukazatel na BSS segment	pracovní adresář
vrchol zásobníku	exist status	deskriptory souborů
stav procesu	signal status	efektivní UID
čas spuštění	proces id	efektivní GID
čas použití CPU	rodičovský proces	par. systémových volání
CPU čas potomků	skupina procesu	různé příznakové bity
čas příštího alarmu	reálné UID	
ukazatel fronty zpráv	efektivní UID	
maska blok. signálů	reálný GID	
proces ID	efektivní GID	
různé příznakové bity	bitmapa pro signály	
	různé příznakové bity	

Tabulka 1: Některé položky záznamu tabulky procesů

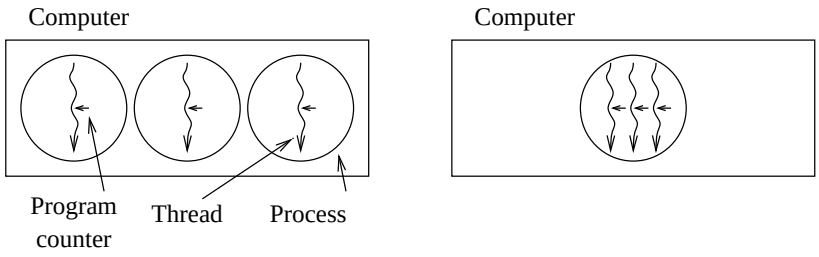
ze zásobníku vytaženy data uložené přerušením a ukazatel na zásobník je nastaven na dočasný zásobník. Operace, jako uložení registrů a nastavení ukazatele zásobníku, nemůže být provedena v jazyce C a provádí se malým podprogramem v assembleru. Když podprogram skončí, volá podprogram v C a ten dokončí požadovanou práci.

Mezi-procesní komunikace v MINIXu je pomocí zpráv, další krok je tedy vytvoření zprávy, aby tato mohla být poslána diskovému procesu, který je zablokován čekáním na tuto zprávu. Zpráva informuje, že došlo k přerušení, aby se odlišila od zprávy zaslané uživatelem pro čtení bloku a podobně. Stav diskového procesu se nyní změní ze stavu *blokovaný* na stav *připravený* a zavolá se plánovač. V MINIXu mají různé procesy různou prioritu, aby byla zajištěna lepší obsluha V/V zařízení, než uživatelských programů.

Pokud má nyní diskový proces nejvyšší prioritu, bude plánovačem vybrán pro spuštění. Pokud je přerušený proces stejně důležitý, nebo i důležitější, pak bude plánovačem opět vybrán pro spuštění a diskový proces musí chvíli počkat.

Podprogram v C volaný z podprogramu v assembleru vrátí návratový kód přerušení a kód v assembleru naplní registry a mapu paměti pro právě aktuální proces a spustí jej. Obsluhu přerušení a plánování si shrneme v následujících bodech. Není důležité, že se detaily mezi operačními systémy vzájemně mírně liší:

1. Hardware uloží na zásobník čítač instrukcí atd.
2. Hardware vezme nový čítač instrukcí z tabulky přerušení.
3. Podprogram v assembleru uloží registry.



Obrázek 3: (a) Tři procesy s jedním vláknem (b) Jeden proces se třemi vlákny

4. Dále nastaví nový zásobník.
5. Spustí obsluhu přerušení v jazyce C.
6. Plánovač označí čekající proces jako připravený.
7. Plánovač rozhodne, který proces bude následně spuštěn.
8. Obsluha v C se vrátí zpět do assembleru.
9. Podprogram v assembleru znovu spustí nový aktuální proces.

Vlákna

V tradičních procesech, jaké jsme doposud probírali, je jedno řízené vlákno a jeden programový čítač v každém procesu. Nicméně, v moderních operačních systémech je podpora pro vícenásobná řídící vlákna uvnitř procesu. Řídící vlákna nazýváme právě jako **vlákna**, řídčeji **lehkými procesy**

Na obrázku 3(a) jsou tři tradiční procesy. Každý proces má svůj adresní prostor a jedno řídící vlákno. Naproti tomu na obrázku 3(b) jeden proces se třemi řídícími vlákny. Ačkoliv oba příklady mají tři vlákna, v prvním případě každé z nich pracuje v jiném adresním prostoru, zatím co ve druhém případě všechna tři vlákna sdílí jeden adresní prostor.

Jako příklad, kde můžeme použít více vláken, si můžeme vzít proces souborového serveru. Obdrží příkaz ke čtení a k zápisu do souboru a odešle zpět požadovaná data nebo data přijme. Pro zlepšení výkonnosti má server vyrovnávací paměť aktuálně používaných souborů v paměti a čte, pokud možno, z této vyrovnávací paměti.

Tato situace nás dovede k situaci na obrázku 3(b). Když přijde požadavek, je předán vláknu k vyřízení. Pokud se toto vlákno částečně zablokuje čekáním na přenos z disku, jiné vlákno je stále schopno pracovat a tak server

je schopen přijmout nový požadavek i v situaci, kdy se čeká na V/V operace. Model na obrázku 3(a) není vyhovující, protože je nutné, aby všechna vlákna přistupovala do stejné vyrovnávací paměti a tato vlákna nesdílejí společný adresní prostor a tedy nemohou sdílet stejnou vyrovnávací paměť.

Jiný příklad, kde jsou vhodná vlákna, je prohlížeč Mozilla nebo Netscape. Mnoho stránek obsahuje množství malých obrázků. Pro každý obrázek musí prohlížeč vytvořit separátní spojení na server a vyžádat si obrázek. Velké množství času se promarní čekáním na navázání spojení a na jeho ukončení. S použitím více vláken v prohlížeči může být požadováno několik obrázků současně a v mnoha případech se velmi zvýší výkonnost, protože u malých obrázků je limitujícím faktorem navazování spojení, nikoliv rychlost sítě.

Pokud je v jednom adresním prostoru více vláken, pak některé položky z tabulky 1 nejsou pro každý proces, ale pro každé vlákno, v separátní tabulce vláken s jedním záznamem pro každé vlákno. Mezi položkami pro vlákna jsou registry, status a programový čítač. Všechny tyto položky jsou nutné proto, že každé vlákno, stejně jako proces, může být pozastaveno a znovu spuštěno a může být i zablokováno.

Některé operační systémy neznají vlákna. Jinými slovy, jsou realizovány zcela v uživatelském prostoru. Pokud se má vlákno zablokovat, tak si vybere a spustí svého následníka, než se samo zastaví. Několik knihoven s vlákny na uživatelské úrovni se používá běžně, včetně POSIX **P-thread** a Mach **C-thread** knihovny.

V ostatních systémech je systém schopen mít více vláken pro každý proces a když se vlákno zablokuje, operační systém vybere pro spuštění další ze stejného nebo jiného procesu. Aby bylo možno použít plánovač, musí mít operační systém seznam všech vláken, analogicky jako u seznamu procesů.

Ačkoliv tyto dvě alternativy vypadají stejně, liší se zásadně ve výkonnosti. Přepnutí vlákna v adresním prostoru uživatele je výrazně rychlejší, než přepnutí voláním jádra. Tento argument jasně hovoří ve prospěch vláken v uživatelském prostoru. Na druhé straně, pokud dojde k zablokování vlákna v uživatelském prostoru, vlákna jsou v jednom bloku a dojde k zablokování celého procesu, protože operační systém není schopen tato vlákna registrovat. Tento argument zase hovoří pro vlákna v jádře systému. Ze všeho plyne, že se používají vlákna obojího typu a byla představena i hybridní řešení.

Bez ohledu na to, jestli jsou vlákna spravována v kernelu nebo v uživatelském prostoru, přináší řadu problémů, které se musí řešit a která mění významně programovací model. Pro začátek předpokládejme vliv na systémové volání *fork*. Když má rodič více vláken, má je mít potomek také? Pokud ne, proces nemusí fungovat správně, protože některé jeho části mohou být nezbytné.

Když potomek získá všechna vlákna, jako jeho rodič, co se stane, když je vlákno zablokované na čtení, např. z klávesnice? Mají teď být dvě vlákna zablokovaná na klávesnici? Když uživatel napíše řádek, mají tento řádek dostat obě vlákna? Nebo jen rodič? Jenom potomek? Stejně tak to platí i

pro otevřená spojení po síti.

Jiná skupina problémů vzniká z podstaty, že vlákna sdílejí mnoho datových struktur. Co se stane, když vlákno zavře soubor a další ho stále používá? Předpokládejme, že jeden proces zjistil, že je málo paměti a začal alokovat více paměti. Uprostřed alokace dojde k přepnutí vlákna a nové vlákno také detekuje málo paměti a začne paměť alokovat. Bude tato alokace provedena jednou, nebo dvakrát? Zpočátku všechny systémy, které neznaly vlákna, měly problémy s knihovnami, protože ty nebyly reentrantní a havarovaly, když se objevilo druhé volání funkce, když první ještě neskončilo.

Jiný problém souvisí s oznamováním chyb. V UNIXech, po systémovém volání je návratový status volání uložen do globální proměnné *errno*. Co se stane, když jedno vlákno zavolá libovolnou funkci a než si přečte návratový kód, tak jiné vlákno zavolá jinou funkci a přepíše errno?

A jiný příklad. Signály. Některé signály jsou logicky svázány s vláknem, zatím co jiné nikoliv. Například signál ALARM. Je vcelku jasné, že signál přijde vláknu, které si signál vyžádalo. Pokud systém rozumí vláknům, může zajistit, že signál dostane odpovídající vlákno. Pokud jádro vlákna nezná, někdo v knihovně vláken musí zajistit manipulaci se signálem. A další komplikace nastane pro vlákna v uživatelském prostoru, když proces může mít v každém okamžiku jen jeden alarm a několik vláken současně si alarm vyžádalo.

Další signály, jako přerušení od klávesnice, není na vláknech závislé. Kdo má signál zachytit? Jedno vyhrazené vlákno? Nové vlákno? Všechna řešení mají problém. A dále, co se stane, když jedno vlákno změní obsluhu signálu bez toho, že by o tom dalo vědět ostatním?

Jeden z posledních problémů je zásobník. Ve většině systémů, když dojde k přetečení zásobníku, jádro automaticky alokuje větší prostor. Když má proces více vláken, musí mít i více zásobníků. Když jádro nezná vlákna, nemůže automaticky zvětšit velikost zásobníku, pokud dojde k jeho přetečení. Dokonce ani nemůže poznat, že tato paměťová chyba souvisí se zásobníkem.

Tyto problémy nejsou určitě nepřekonatelné, ale ukazují, že vložení vláken do stávajícího systému se bez velkých změn určitě neobejde. A musí být přepsány knihovny. A všechny tyto věci musí být provedeny při zachování zpětné kompatibility s existujícími programy pro limitní případ programů s jedním vláknem.

2.2 Mezi-procesní komunikace

Procesy potřebují často komunikovat s dalšími procesy.

Například v rouře shellu se musí výstup prvního procesu předávat do druhého procesu a tak dále, až do konce příkazového řádku. Je zde tedy potřeba pro komunikaci mezi procesy, přednostně dobře strukturovaným způsobem, bez použití přerušení. V následující kapitole se podíváme na některé

vlastnosti, vztahující se k mezi-procesní komunikaci (interprocess communication) - IPC.

Velmi krátce, zde jsou tři základní problémy. První byl zmíněn výše: jak může proces poslat informace jinému? Druhý problém je, aby si dva, nebo více procesů, vzájemně nelezlo do cesty, pokud provádějí libovolné kritické aktivity (např. pokud dva procesy chtějí alokovat poslední 1MB paměti). A třetí problém předpokládá správné pořadí při vzájemných závislostech: pokud proces A produkuje data a proces B je tiskne, musí B počkat dokud A nevyprodukuje nějaká data, než začne tisknout. Probereme si všechny tři problémy a začneme hned v následující kapitole.

2.2.1 Souběh (Race Conditions)

V operačních systémech mohou společně pracující procesy sdílet některé běžné paměti, kde všichni mohou číst i zapisovat. Sdílená paměť může být hlavní paměť, nebo sdílený soubor; umístění paměti nemění povahu komunikace nebo problém, který tím vzniká. Jak vypadá mezi-procesní komunikace v praxi, si můžeme ukázat na jednoduchém, ale běžném příkladě – na tiskovém serveru. Když proces chce tisknout soubor, vloží jméno souboru do speciálního adresáře pro tisk (spooler directory). Jiný proces, tisková služba (printer daemon) pravidelně kontroluje, zda je k dispozici soubor pro tisk, a pokud je, tak jej vytiskne a pak smaže příslušné jméno z adresáře.

Představme si náš tiskový adresář s velkým počtem pozic, číslovaných 0, 1, 2, ..., každá schopná nést jméno souboru. Také si představme, že jsou zde dvě sdílené proměnné: *out*, která ukazuje na následující soubor pro tisk, a *in*, která ukazuje na následující volné místo v adresáři. Tyto dvě proměnné mohou být v souboru dostupném všem procesům. V určité situaci je pozice 0 až 3 volná a pozice 4 až 6 obsazená. V podstatě současně se proces A a B rozhodne vložit do fronty pro tisk soubor. Situace je na obrázku **??**.

Podle Murphyho zákona se stane následující: proces A přečte *in* a uloží hodnotu 7 v lokální proměnné *nextslot*. V tomto okamžiku přijde přerušení od časovače a operační systém rozhodne, že proces A již běžel dostatečné dlouho a přepne se na proces B. Proces B také přečte *in* a také dostane 7 a tak uloží jméno svého souboru na pozici 7 a aktualizuje *in* na 8. Tím skončí a začne vykonávat jinou činnost.

Nicméně, proces A se znovu spustí v místě, kde naposledy skončil. Podívá se na svou proměnnou *nextslot*, najde zde 7 a zapíše jméno svého souboru na pozici 7, čímž přepíše právě uloženou hodnotu uloženou procesem B. Pak vypočítá hodnotu *nextslot* + 1, což je 8 a nastaví *in* na 8. Tím je tiskový adresář konzistentní a tisková služba neshledá nikde žádnou chybu, až na to, že proces B přijde o svůj výstup. Situace, kdy dva nebo více procesů čtou a modifikují sdílená data a finální výsledek je závislý na tom, kdo přesně kdy běžel, se nazývá **souběh** (race condition). Hledání chyb v programech, kde je souběh, není vůbec snadné. Výsledky většiny testů jsou správné, ale

jen ve výjimečných případech, kdy se stane něco neočekávaného, nastane nevysvětlitelný výsledek.

2.2.2 Kritická oblast (Critical Section)

Jak se vyhnout souběhu? Cesta k prevenci této i jiných situací, jako sdílená paměť, sdílené soubory a sdílené prakticky cokoliv, je zakázat, aby více než právě jeden proces nemohl číst a modifikovat sdílená data. Jinými slovy, potřebujeme **vzájemné vyloučení** (mutual exclusion) - určitou cestu, jak mít jistotu, aby během používání sdílené proměnné byly ostatní procesy vyloučeny z provádění stejné činnosti. Výše popsaný problém je proto, že proces B začal používat jednu sdílenou proměnnou dříve, než proces A s ní ukončil manipulaci. Výběr odpovídající jednoduché operace pro dosažení vzájemného vyloučení, je hlavním konstrukčním rysem v mnoha systémech a téma, které budeme probírat v následujících kapitolách.

Problém vyhnutí se souběhu může být formulován i abstraktně. Určitý čas je proces zaměstnán svým počítáním a dalšími věcmi, které nevedou k souběhu. Nicméně, občas může proces přistoupit do sdílené paměti nebo do souboru, nebo může udělat jinou kritickou aktivitu, která může vést k souběhu. Tato část programu, kde se přistupuje ke sdílené paměti, se nazývá **kritickou oblastí nebo sekcí** (critical section or region). Pokud budeme schopni připravit věci tak, aby procesy nebyly nikdy ve stejných kritických sekcích ve stejnou dobu, můžeme se vyhnout souběhu.

Ačkoliv se tímto požadavkem vyhýbáme souběhu, není to dostatečné, pokud máme paralelní procesy spolupracující a efektivně používající sdílená data. Máme čtyři podmínky, abychom měli dobré řešení:

1. Žádné dva procesy nesmí být současně uvnitř stejné kritické sekce.
2. Na řešení nesmí mít vliv počet a rychlost CPU.
3. Žádný proces mimo kritickou sekci nesmí blokovat jiný proces.
4. Žádný proces nesmí zůstat čekat nekonečně dlouho na kritickou sekci.

2.2.3 Vzájemné vyloučení s aktivním čekáním (busy waiting)

V této sekci si probereme několik návrhů pro dosažení vzájemného vyloučení tak, že zatím co jeden proces pracuje na aktualizaci sdílené paměti v kritické sekci, žádný další proces nesmí vstoupit do této kritické sekce a způsobit problémy.

Zákaz přerušení

Nejjednodušší řešení je zákaz všech přerušení procesy, které právě vstoupily do své kritické sekce a jejich znovu-povolení před opuštěním kritické sekce. Když jsou zakázána všechna přerušení, nemůže nastat přerušení od

časovače. CPU je přepnuto z procesu do procesu na základě hodin nebo jiného přerušení, no a se zakázanými přerušeními nebude CPU přepnuto pro jiný proces. Tedy jakmile proces zakáže přerušení, aktualizuje sdílenou paměť bez nebezpečí, že by do toho zasahoval jiný proces.

Tato cesta ale není příliš atraktivní, protože není rozumné, aby uživatelský proces zakázal přerušení. Představme si, že jeden to tak udělá a nikdy je znovu nepovolí. To může být konec práce systému. Kromě toho, pokud je systém dvou a víceprocesorový, zákaz přerušení se dotkne jen toho procesoru, který instrukci zákazu provede. Další procesor bude pokračovat v běhu a může přistoupit do sdílené paměti.

Na druhé straně je často vyhovující pro kernel samotný zakázat přerušení na několik instrukcí, zatím co se aktualizují proměnné v seznamu. Pokud dojde k přerušení při aktualizaci seznamu připravených procesů, dojde k nekonzistenci a tedy k možnému souběhu. Závěr je: zákaz přerušení je užitečná metoda uvnitř samotného operačního systému, ale není použitelná jako obecná metoda vzájemného vyloučení pro uživatelské procesy.

Zamykací proměnné (Lock Variables)

Zkusíme druhý pokus, podívejme se na programové řešení. Mějme jednu sdílenou zamykací proměnnou inicializovanou na 0. Když proces požaduje vstup do své kritické sekce, musí nejprve otestovat zámek. Když je zámek 0, proces ji nastaví na 1 a vstoupí do kritické sekce. Pokud je zámek již 1, proces počká, až bude 0. Hodnota 0 tedy znamená, že v kritická sekci není žádný proces, a 1 znamená, že nějaký proces je v kritické sekci.

Naneštěstí, tato myšlenka obsahuje přesně ten samý zásadní problém, o kterém jsme mluvili u adresáře s tisky. Předpokládejme, že proces přečte zámek a má 0. Než může nastavit zámek na 1, je plánovačem přepnut jiný proces, ten se spustí a také nastaví zámek na 1 a dva procesy se ocitnou v jedné kritické sekci současně.

Pokud přemýšlíte nad možností druhého testování zámku před jeho modifikací, věřte, že nám to nepomůže. Souběh nastane o chvíli později, hned po druhém otestování zámku.

Přesné střídání (Strict Alternation)

Třetí přístup k řešení vzájemného vyloučení je v příkladu 1. Tento fragment je stejně jako budou další části programů v dalším textu, napsán v jazyce C. To proto, že operační systémy jsou obvykle v jazyce C napsány a těžko budou psány v Pascalu nebo Module.

V uvedeném kódu je celočíselná proměnná *turn* inicializovaná na 0 a na sledování její přepínající se hodnoty je založen vstup do kritické sekce a vyhodnocení nebo aktualizace sdílené paměti. Na začátku první proces otestuje *turn*, detekuje 0 a vstoupí do kritické sekce. Druhý proces také najde hodnotu 0 a proto zůstane čekající ve smyčce a testující *turn*, zda už není 1. Nepřetržité testování proměnné dokud není nastavena na určitou

```
while ( true )                          while ( true )
{                                       {
  while ( turn != 0 ); // wait
while ( turn != 1 ); // wait
  enter_critical_region();
enter_critical_region();
  turn = 1;                               turn = 0;
  noncritical_region();
noncritical_region();
}                                       }
```

Příklad 1: Přesné střídání

hodnotu, nazýváme **aktivním čekáním** (busy waiting). Obvykle bychom se mu měli vyhnout, protože tak plýtváme procesorovým časem. Jen když máme nějaký zdůvodnitelný předpoklad, že čekání bude krátké, použijeme aktivní čekání.

Když první proces opustí kritickou sekci, nastaví *turn* na 1 a povolí tak vstup druhému procesu vstoupit do kritické sekce. Předpokládejme, že druhý proces projde kritickou sekcí velmi rychle a oba procesy jsou v nekritické sekci a *turn* je 0. Nyní první proces vykoná celý cyklus rychle a vrátí se do nekritické sekce s tím, že *turn* je 1. Teď první proces dokončí nekritickou sekci a vrátí se na začátek cyklu. Bohužel nemůže vstoupit do kritické sekce, protože *turn* je 1 a druhý proces je zaměstnán vykonávání své nekritické sekce. Jinak řečeno, použití přepínání není dobrá myšlenka, pokud je jeden proces výrazně rychlejší, než druhý.

Tato situace porušuje třetí podmínku stanovenou dříve: první proces je blokován procesem v nekritické sekci. Pokud se vrátíme k tiskovému adresáři probíranému dříve, pokud si spojíme kritickou sekci se čtením a zápisem do adresáře, první proces nebude smět tisknout další soubor, protože druhý proces právě dělá něco jiného.

Fakticky toto řešení vyžaduje, aby se dva procesy přesně střídaly ve vstupu do své kritické sekce, například vkládání souboru do tiskového adresáře. Ani jeden však nebude smět vložit dva za sebou. I když tento algoritmus vylučuje souběh, není vážným kandidátem na řešení souběhu, protože porušuje třetí podmínku.

Petersonovo řešení (Peterson's solution)

Kombinaci myšlenky střídání s myšlenkou zamykací a varovné proměnné vymyslel Nizozemský matematik T. Dekker jako první programové řešení problému vzájemného vyloučení, které nevyžaduje přesné střídání.

```
#define N 2                                    // počet procesů

int turn;                                      // kdo přepnul
int interested[ N ];
// všechny počáteční hodnoty FALSE

void enter_region( int process )               // proces 0 nebo 1
{
  int other = 1 - process;
// číslo druhého procesu
  interested[ process ] = TRUE;
// požadavek má tento proces
  turn = process;                              // přepínač
  while ( turn == process && interested[ other ] == TRUE );
                                               // podmínka vstupu nebo čekání
}

void leave_region( process )                   // proces 0 nebo 1
{
  interested[ process ] == FALSE;
// indikace opuštění krit. sekce
}
```

Příklad 2: Petersonův algoritmus

V roce 1981 G. L. Peterson objevil daleko jednodušší cestu k dosažení vzájemného vyloučení a tak vyřadil Dekkerovo řešení jako zastaralé. Algoritmus je v příkladu 2. Tento algoritmus sestává ze dvou podprogramů napsaných v jazyce C, což znamená, že by měly být dodány prototypy funkcí pro všechny použité a definované funkce. Z důvodu úspory místa budeme prototypy vynechávat.

Před použitím sdílené proměnné každý proces volá *enter_region* se svým číslem procesu 0 nebo 1, jako parametrem. Toto volání způsobí podle potřeby čekání, dokud není vstup bezpečný. Jakmile skončí používání sdílené paměti, zavolá proces *leave_region*, aby to dal najevo a povolí tak vstup jinému procesu, pokud to ten potřebuje.

Podívejme se, jak toto řešení funguje. Na začátku není žádný proces v kritické sekci. Nyní první proces zavolá *enter_region*. Ten dá najevo svůj zájem nastavením své položky v poli a nastaví *turn* na 0. Pokud druhý proces nepožadoval vstup do kritické sekce, podprogram *enter_region* podprogram

okamžitě skončí. Pokud nyní druhý proces zavolá *enter_region*, zůstane zde zastaven, dokud *interested[0]* nebude *FALSE*, což se stane jedině když první proces zavolá *leave_region*, aby ukončil kritickou sekci.

Nyní předpokládejme, že oba procesy zavolají *enter_region* téměř současně. Oba uloží své číslo procesu do proměnné *turn*. Kdokoliv uloží hodnotu jako poslední, je to ten, kdo se počítá. První je ztracena. Předpokládejme, že druhý proces zapsal jako poslední a *turn* je tedy 1. Když oba procesy vstoupí do čekací smyčky, první proces okamžitě podmínku přeskočí a vstoupí do kritické sekce. Druhý proces zůstane čekající ve smyčce a nevstoupí do své kritické sekce.

TSL instrukce (TSL Instruction)

Podívejme se na návrh, který vyžaduje malou pomoc hardware. Mnoho počítačů, zejména těch, kde se při návrhu uvažovalo o více procesorech, má speciální instrukci **testuj a nastav zámek** (Test And Set Lock) - TSL, která pracuje následovně. Přečte hodnotu proměnné do registru a do této proměnné uloží nenulovou hodnotu. Operace čtení a uložení do registru je garantována jako neviditelná - žádný další procesor nemůže přistupovat do paměti, dokud instrukce neskončí. Procesor vykonávající TSL instrukci uzamkne paměťovou sběrnici, aby zakázal dalším procesorům přistupovat do paměti, dokud není hotov.

Pro použití TSL instrukce použijeme sdílenou proměnnou *lock*, pro sladění přístupu do sdílené paměti. Když je *lock* 0, kterýkoliv proces může změnit hodnotu na 1 použitím TSL instrukce a pak smí přistupovat ke sdílené paměti. Když je hotov, proces nastaví *lock* zpět na 0 použitím instrukce MOVE.

Jak může být instrukce implementována pro prevenci vstupu dvou procesů do kritické sekce? Řešení je v příkladu 3. Podprogram *enter_region* obsahuje 4 instrukce ve fiktivním assembleru, ale dostatečně typické. Instrukce TSL přečte hodnotu *lock* do registru a nastaví *lock* na 1. Původní hodnota se porovná s nulou. Když je hodnota nenulová, zámek již byl nastaven a podprogram se vrací zpět na začátek. Dříve nebo později bude 0 a podprogram se ukončí s nastaveným zámkem. Odstranění zámku je pak snadné. Podprogram uloží 0 do *lock*, není třeba žádné speciální instrukce.

Máme jedno přímé řešení problému kritické sekce. Proces před vstupem do kritické sekce zavolá podprogram *enter_region*, který případný aktivním čekáním počká na uvolnění zámku, pak provede uzamčení a skončí. Na konci kritické sekce proces zavolá podprogram *leave_region*, který uvolní zámek. Jako ve všech řešeních pro kritickou sekci, musí i zde program volat podprogramy *enter_region* a *leave_region* v korektním počtu, aby metoda byla úspěšná. Pokud proces nedodržuje pravidla, vzájemné vyloučení selže.

```
enter_region:
      TSL register, lock          ; zkopíruje lock do registru
                                  ; a nastaví lock na 1
      CMP register, 0             ; byl lock nulový?
      JNE enter_region            ; pokud nebyl nulový, testujeme znovu
      RET                         ; návrat z podprogramu

leave_region:
      MOV lock, 0                 ; uloží 0 do lock
      RET                         ; ukončení podprogramu
```

Příklad 3: Nastavení zámku pomocí instrukce TSL

2.2.4 Uspání a probuzení (Sleep and Wakeup)

Obě řešení, Petersenovo i s použitím TSL jsou správná, ale jejich nedostatkem je aktivní čekání. Podstatou, co tato řešení dělají, je toto: pokud proces potřebuje vstoupit do kritické sekce, zkontroluje, zda je vstup povolen. Pokud ne, zůstane proces ve smyčce čekat na povolení.

Nejen že proces plýtvá procesorovým časem, může také mít neočekávaný efekt. Představme si počítač se dvěma procesy, H s vysokou prioritou a L s nízkou prioritou. Plánovací pravidla říkají, že proces může běžet, pokud je připraven. V určité chvíli je proces L v kritické sekci a H se stane připraveným k běhu. H začne aktivně čekat, ale L nebude nikdy plánován, dokud H běží. L tedy nedostane šanci opustit kritickou sekci a H se zacyklí navždy. Tato situace se občas nazývá jako **problém převrácené priority** (priority inversion problem).

Podívejme se na základní mezi-procesní možnosti pro blokování, místo plýtvání procesorovým časem, když není povolen vstup do kritické sekce. Jedna z nejjednodušších je dvojice SLEEP a WAKEUP. SLEEP je systémové volání, které uspí volající proces, tj. pozastaví dokud jej jiný proces neprobudí. WAKEUP má jediný parametr a sice proces k probuzení. Alternativně má SLEEP a WAKEUP jeden parametr - adresu paměti pro uspořádání (spojení) odpovídajících požadavků na uspání a probuzení.

Problém výrobce–spotřebitel (The Producer-Consumer Problem)

Jako příklad použití těchto jednoduchých principů si ukážeme problém **výrobce–spotřebitel**, známý také jako problém **omezeného bufferu** (bounded buffer). Dva procesy sdílí buffer pevné velikosti. Jeden z nich, výrobce, vkládá informace do bufferu a druhý, spotřebitel, je z bufferu vybírá. Problém se dá zobecnit pro m výrobců a n spotřebitelů. Z důvodu jednoduchosti

zůstaneme u jednoho výrobce i spotřebitele.

Problém vzniká v okamžiku, kdy výrobce chce vložit novou položku do bufferu, ale ten je už plný. Řešením pro výrobce je pozastavit činnost a čekat na probuzení, až spotřebitel odebere jednu nebo více položek. Podobně, když spotřebitel chce odebrat položku z bufferu a vidí, že buffer je prázdný, tak pozastaví činnost dokud výrobce do bufferu nějakou položku nevloží a neprobudí ho.

Tento postup vypadá velmi jednoduše, ale vede ke stejnému problému souběhu, jako tomu bylo u souborů pro tisk. Pro sledování počtu položek v bufferu budeme potřebovat proměnnou *count*. Maximální počet položek z bufferu bude N a kód výrobce se hned na začátku podívá, jestli je počet položek v buffer N. Pokud ano, tak se uspí. Pokud ne, výrobce vyrobí novou položku a vloží ji do bufferu a zvýší *count*.

Kód spotřebitele je podobný. Nejprve testuje, zda je *count* nulový. Pokud ano, usíná, pokud ne, odebírá položku z bufferu a snižuje *count*. Oba procesy také sledují, zda by druhý proces měl být pozastavený, a pokud ne, tak jej probudí. Kód pro výrobce a spotřebitele je v příkladu 4.

Pro vyjádření systémových volání SLEEP a WAKEUP používáme volání funkcí v jazyce C. Funkce nejsou standardem jazyka C, ale pravděpodobně budou k dispozici v systému, který má tato systémová volání. Podprogramy *enter_item* a *remove_item* zajišťují vložení položky do bufferu a její vyjmutí.

A nyní zpět k souběhu. Ten může nastat, protože přístup k proměnné *count* není hlídaný. Může nastat následující situace. Buffer je prázdný a spotřebitel právě přečetl *count*, aby zjistil, zda je nulový. V tomto okamžiku se plánovač rozhodne dočasně pozastavit spotřebitele a spustit výrobce. Ten vloží položku do bufferu, zvýší *count* a zjistí, že je právě roven jedné. Vyhodnotí, že *count* byl nulový a že spotřebitel musí být uspaný a zavolá *wakeup*, aby ho probudil. Naneštěstí spotřebitel ještě logicky uspán nebyl a signál je ztracen. Když plánovač znovu spustí spotřebitele, tak ten otestuje dříve přečtenou hodnotu *count*, zjistí že je nulová a uspí se. Dříve nebo později výrobce zaplní buffer a také se uspí. Oba nyní spí navždy.

Podstata problému je zde v tom, že je příkaz k probuzení zaslán procesu, který ještě nespí. Kdyby se neztratil, vše by fungovalo. Rychlé řešení uděláme přidáním **bitu probuzení** (wakeup waiting bit) do programu. Když bude poslán signál pro probuzení neuspanému procesu, tento bit se nastaví. Později, když se proces pokusí uspat sám sebe, pokud bude nastaven bit probuzení, tak jej vynuluje a neuspí se. Bit probuzení je jakási úložna či pokladnička pro signály probuzení.

Zatím, co v této chvíli nám bit probuzení zabezpečí řešení v tomto příkladu, je jednoduché vymyslet příklad se třemi, nebo i více procesy, kde bude jeden bit nedostatečný. Budeme muset udělat jinou úpravu, kde budeme muset přidat druhý bit, nebo možná 8 nebo 32 bitů, ale princip problému zde stále zůstává.

```
#define N 100
// maximální velikost bufferu
int count = 0;
// počet položek v bufferu

void producer()
{
  while( TRUE )
// nekonečný cyklus
  {
    produce_item();
// výroba nové položky
    if ( count == N )
// pokud je buffer plný, usínáme
        sleep();
    enter_item();
// vložení položky do bufferu
    count++;
// zvýšení počtu položek
    if ( count == 1 )
// byl buffer prázdný?
        wakeup( consumer );          //
probudíme spotřebitele
  }
}

void consumer()
{
  while ( TRUE )
// nekonečný cyklus
  {
    if ( count == 0 )
// pokud je buffer prázdný, usínáme
        sleep();
    remove_item();
// odebrání položky z bufferu
    count--;
// snížení počtu položek
    if ( count == N - 1 )
// byl buffer plný?
        wakeup( producer );          //
probudíme výrobce
    consume_item();                  // užij si to!
  }
}
```


Příklad 4: Implementace výrobce a spotřebitele

2.2.5 Semafory (Semaphores)

Taková byla situace v roce 1965, kdy Dijkstra navrhl použití celočíselné proměnné pro sčítání počtu probuzení pro pozdější použití. V jeho návrhu byl představen nový typ proměnné, které říkáme **semafor**. Semafor může mít hodnotu 0 indikující, že nebylo uloženo žádné probuzení, nebo kladné číslo, indikující počet nezpracovaných probuzení.

Dijkstrův návrh obsahoval dvě operace *UP* a *DOWN*, jako zobecnění *SLEEP* a *WAKEUP*. Operace semaforu *DOWN* kontroluje, jestli je hodnota větší než 0. Pokud je, sníží hodnotu a pokračuje. Pokud je hodnota 0, proces se uspí a operace *DOWN* se pro tento okamžik neukončí. Kontrola proměnné, její modifikace a případné uspání se vždy uskuteční jako jediná neviditelná **nedělitelná akce**. Je zaručeno, že jednou započatá operace se semaforem nedovolí žádnému jinému procesu k semaforu přistupovat, dokud se operace neukončí. Tato nedělitelnost je absolutně nezbytná pro řešení problémů synchronizace a souběhu.

Operace *UP* zvýší hodnotu v semaforu. Pokud jeden nebo více procesů je uspáno semaforem, bez možnosti ukončit dřívější volání, jeden z nich je vybrán systémem, aby dokončil *DOWN*. Tedy po operaci *UP* na semaforu s čekajícími procesy bude hodnota semaforu stále 0, ale bude o jeden uspaný proces méně. Operace zvýšení hodnoty semaforu a probuzení jednoho procesu je také neviditelná. Žádný proces se nikdy nezastaví voláním *UP*, stejně jako se nikdy žádný proces neuspí voláním *WAKEUP* v předchozím modelu.

Řešení problému výrobce a spotřebitele použitím semaforů

Semafory řeší problém ztraceného signálu, jak je uvedeno v příkladu 5. Řešení je založeno na nedělitelnosti (neviditelnosti) operací semaforu. Obvyklá cesta implementace *UP* a *DOWN* je jako systémové voláním, kdy operační systém krátce zakáže všechna přerušení po dobu testování a modifikace proměnné a případného uspání procesu. Všechny tyto akce představují jen několik instrukcí jejichž ochrana je zajištěna zákazem přerušení. Pokud je použito více procesorů, jsou všechny semafory chráněny zamykací proměnnou a použitím TSL je zajištěno, že jen jediný procesor může modifikovat semafor. Zamyslete se, jestli rozumíte rozdílu použití TSL pro ochranu semaforu před použitím více procesory v jednu chvíli a aktivním čekáním výrobce a spotřebitele při čekání s prázdným nebo plným bufferem. Operace se semaforem potřebují jen zlomky milisekund, zatím co práce výrobce a spotřebitele trvá libovolně dlouho.

Toto řešení využívá tři semafory: jeden pojmenovaný *full* pro počítání obsazených položek, další *empty* pro počítání počtu volných pozic a poslední *mutex*, který zajišťuje, aby výrobce a spotřebitel nepřistupovali k bufferu současně. *Full* začíná od nuly, *empty* má počáteční hodnotu stejnou, jako maximální počet položek bufferu a *mutex* začíná hodnotou 1. Semafor, který je inicializován na 1 a je použit dvěma, nebo více procesy, aby zajistil, že do

```
#define N 100
semaphore mutex = 1;
semaphore empty = N
semaphore full = 0

void producer()
{
  int item;

  while ( TRUE )
  {
    produce_item( &item );
    down( &empty );
    down( &mutex );
    enter_item( item );
    up( &mutex );
    up( &full );
  }
}

void consumer()
{
  int item;

  while ( TRUE )
  {
    down( &full );
    down( &mutex );
    remove_item( &item );
    up( &mutex );
    up( &empty );
    consume_item( item );
  }
}
```

Příklad 5: Implementace výrobce a spotřebitele se semafory

kritické sekce vstoupí vždy jen jeden z nich, se nazývá **binární semafor**. Pokud každý proces zavolá *DOWN* před vstupem do kritické sekce a po jejím ukončení *UP*, je zaručeno vzájemné vyloučení.

Teď, když máme dobrý základ meziprocesních operací, podívejme se zpět na obsluhu přerušení, viz. 2.1.2. V systémech se semafory je přirozenou cestou k ukrytí přerušení právě semafor, inicializovaný na 0 a spojený s každým V/V zařízením. Hned po startu V/V zařízení se obsluhující proces zastaví na volání *DOWN* přiřazeného semaforu a tím se okamžitě zastaví. Uvolnění zajistí odpovídající proces. V tomto modelu se krok 6 realizuje jako volání *UP* semaforu příslušného zařízení a tak v kroku 7 bude smět rozhodnout plánovač o spuštění správce zařízení. Jistě, několik procesů je v této chvíli připraveno ke spuštění a plánovač může vybrat jako další jiný důležitější proces. Na plánování procesů se podíváme později v této kapitole.

V předchozím řešení problému výrobce–spotřebitel jsme použili semafory pro dva odlišné účely. Ten rozdíl je dost důležitý, proto se o něm zmíníme. Semafor *mutex* je použit k vzájemnému vyloučení. Je navržen, aby garantoval, že jen jediný proces bude v určitém čase manipulovat s obsahem bufferu a souvisejícími proměnnými. Vzájemné vyloučení je vyžadováno, aby zabránilo chaosu.

Další použití semaforu je pro **synchronizaci**. Semafory *full* a *empty* jsou nezbytné k zajištění určité sekvence událostí. V tomto případě zajišťují, aby se zastavil výrobce, pokud je buffer plný a spotřebitel, pokud je buffer prázdný. Toto použití se liší od vzájemného vyloučení.

Ačkoliv už máme semafory déle než čtvrt století, stále se věnuje výzkum jejich používání.

Monitory

Vypadá meziprocesní komunikace s použitím semaforů jednoduše? Tak to rychle pusťte z hlavy! Podívejme se blíže na pořadí funkcí *DOWN* před vložením a odebráním položky z bufferu v příkladu 5. Představte si, že pořadí dvou volání *DOWN* bude opačné a *mutex* je dekrementován před *empty*, místo aby to bylo naopak. Pokud je buffer plný, výrobce by se měl zastavit a *mutex* je 0. Následně, když spotřebitel znovu přistoupí do bufferu, pokusí se provést *DOWN mutexu*, ten je právě 0, a zablokuje se. Oba procesy zůstanou trvale zablokovány a nic dalšího už nikdy neudělají. Tato nešťastná situace se nazývá **zablokování** (deadlock). Podrobněji probereme zablokování až v další kapitole.

Tento problém je uveden proto, aby ukázal, jak je třeba být opatrný při používání semaforů. Jedna drobná chyba a vše se zastaví natrvalo. Je to podobné programování v assembleru, jen horší, protože chyba je souběh, zablokování, nebo jiná forma nepředvídatelného či nezinscenovatelného chování.

Pro snadnější psaní bezchybných programů navrhli Hoare (1974) a Brinch Hansen (1975) vyšší úroveň synchronizačního prostředku, nazývanou **mo-**

```
monitor ProduceConsume : public Monitor
{
  Condition my_cond;

  void producer()
  {
    ...
  }

  void consumer()
  {
    ...
  }
}
```

Příklad 6: Použití monitoru

nitor. Jejich návrhy byly mírně odlišné, jak si dále popíšeme. Monitor je skupina podprogramů, proměnných a struktur, které jsou zabaleny do speciálního typu modulu, či balíku (dnes asi nejčastěji třídy). Proces může volat podprogramy z monitoru kdykoliv potřebuje, ale nikdy nemůže přímo přistupovat k vnitřním datovým strukturám z podprogramů, deklarovaných vně monitoru. Fiktivní ukázka je v příkladu 6

Monitor má důležitou vlastnost, díky které je vhodný pro vzájemné vyloučení: v jakékoliv situaci smí být jen jeden proces aktivní v monitoru. Monitor je prvek programovacího jazyka a proto jej překladač rozpozná a zajistí volání jeho podprogramů odlišným způsobem, než pro ostatní podprogramy. Typicky, když proces zavolá podprogram z monitoru, první instrukce podprogramu zkontrolují, zda již není v monitoru aktivní jiný proces. Pokud ano, proces se pozastaví, dokud jiný proces monitor neopustí. Pokud není v monitoru jiný proces, volající proces smí podprogram provést.

Je úkolem překladače implementovat vzájemné vyloučení, ale obecnou cestou je binární semafor. Protože překladač, a ne programátor, zajišťuje konstrukci vzájemného vyloučení, je daleko méně pravděpodobné, že něco bude chybně. Ve všech případech se nemusí programátor, píšící program, zabývat tím, jak zajistí překladač vzájemné vyloučení. Je dostačující vědět, že vložením všech kritických sekcí do monitoru bude zajištěno, aby dva, nebo více procesů, nikdy nevykonávalo kritickou sekci současně.

Ačkoliv monitor dává možnost řešení vzájemného vyloučení, jak jsme vysvětlili výše, není to všechno. Stále potřebujeme způsob, jak proces pozastavit, pokud je to zapotřebí. V problému výrobce–spotřebitel je snadné

vložit všechny testy na prázdný a plný buffer do funkcí monitoru, ale jak teď pozastavit výrobce, když se buffer naplní?

Řešení spočívá v *podmíněné proměnné* se dvěma operacemi *WAIT* a *SIGNAL*. Když některý podprogram monitoru zjistí, že nemůže pokračovat, zavolá *WAIT* s konkrétní podmíněnou proměnnou. Tato akce pozastaví volající proces. Tím také povolí vstup jinému procesu, který na vstup do monitoru dosud čekal.

Jiný proces, v našem případě spotřebitel, probudí svého spícího partnera pomocí *SIGNAL* s podmíněnou proměnnou, na kterou partner čekal. Abychom nemohli mít současně dva procesy aktivní v monitoru, musíme si říct co se stane voláním *SIGNAL*. Hoare navrhl spustit probuzený proces a pozastavení druhého. Brinch Hansen navrhl šikovnější řešení a požadoval, aby volání *SIGNAL* bylo na konci podprogramu a po jeho provedení *musí* proces okamžitě opustit monitor. Jinými slovy musí být *SIGNAL* posledním příkazem podprogramu. Použijeme návrh Brinch Hansena, protože jde o jednodušší koncepci a také se snadněji implementuje. Pokud zavoláme *SIGNAL* pro podmíněnou proměnnou, na kterou čeká více procesů, pouze jeden je spuštěn dle rozhodnutí plánovače procesů.

Podmíněná proměnná není čítač. Nestřádá tedy signály jako semafor. Pokud je tedy podmíněné proměnné zaslán signál a nikdo na ni nečeká, signál se ztrácí. Volání *WAIT* musí tedy předcházet *SIGNAL*. Toto pravidlo značně zjednodušuje implementaci. V praxi to pak nepřináší problémy, protože je snadné evidovat stav všech procesů s podmíněnými proměnnými, je-li to zapotřebí. A proces, který má volat *SIGNAL* může snadno zjistit podle stavu proměnných, že volání není zapotřebí.

Fiktivní model výrobce–spotřebitel s použitím monitoru je v příkladu 7. Pozor, ve skutečnosti překladač jazyka C/C++ nezná monitor!

Možná vás napadlo, že operace *WAIT* a *SIGNAL* jsou podobné *SLEEP* a *WAKEUP*, kde jsme si dříve popsali závažný problém souběhu. Jsou opravdu podobné, ale s jedním zásadním rozdílem: *SLEEP* a *WAKEUP* selhaly v okamžiku, kdy jeden proces dostal signál pro probuzení dříve než se stačil uspat. S monitorem se to stát nemůže. Automatické vzájemné vyloučení zajišťuje, aby výrobce uvnitř monitoru, pokud zjistí, že buffer je plný, měl možnost ukončit operaci *WAIT* bez obavy z možnosti, že dojde k přepnutí procesu plánovačem dříve, než je *WAIT* kompletní. Spotřebitel také nikdy nebude za žádných okolností v monitoru, dokud není *WAIT* ukončen a výrobce není pozastaven a označen jako zablokovaný.

Automatickým vzájemným vyloučením kritické sekce monitorem je paralelní programování podstatně snadnější a odolnější proti chybám, než se semafory. Přesto mají určité nevýhody. Jak už jsme řekli dříve, monitor je prvek programovacího jazyka. Překladač jej musí rozpoznat a připravit vzájemné vyloučení. Jazyk C, Pascal a většina dalších jazyků nezná monitor a nelze očekávat od jejich překladačů vynucení jakéhokoliv vzájemného vyloučení. Fakticky, jak by měl překladač poznat, který podprogram je v

```
monitor MyBuffer : public Monitor
{
  Condition full, empty;
  int count;

  void enter()
  {
    if ( count == N ) wait( full );
    enter_item();
    count++;
    if ( count == 1 ) signal( empty );
  }

  void remove()
  {
    if ( count == 0 ) wait( empty );
    remove_item;
    count--;
    if ( count == N - 1 ) signal( full );
  }
}

MyBuffer buffer( N );

void producer()
{
  while ( true )
  {
    produce_item();
    buffer.enter();
  }
}

void consumer()
{
  while ( true )
  {
    buffer.remove();
    consume_item();
  }
}
```

Příklad 7: Návrh řešení problému výrobce–spotřebitel s monitorem

monitoru a který ne? Proto jsme řekli, že příklad 7 je pouze *fiktivně* v jazyce C!

Tyto jazyky nemají ani semafory, ale jejich přidání je snadné: stačí přidat systémová volání *UP* a *DOWN*, např. i jednoduchým podprogramem v assembleru. Překladač nemusí vědět nic o existenci semaforů. Ale operační systém ano. Pokud tedy máte minimálně operační systém se semafory, můžete napsat uživatelský program v C nebo C++ (nebo dokonce i v BASICu, pokud jste masochisté), semafory využívající. Pro monitory potřebujete programovací jazyk a překladač, kde jsou implementovány.

S monitory a také se semafory souvisí jiný problém. Jsou navrženy pro jeden nebo více procesorů, které mají přístup do jedné sdílené paměti. Umístěním semaforu do sdílené paměti a jeho ochranou pomocí TSL instrukce zabráníme souběhu. Pokud přejdeme na distribuovaný operační systém s více procesory, každý se svou privátní pamětí, spojené pomocí LAN, stanou se tyto principy nepoužitelné.

Závěr je takový: semafory představují velmi nízkou úroveň a monitor není použitelný, kromě několika programovacích jazyků. Kromě toho ani jeden z obou mechanismů nezajišťuje výměnu informací mezi procesy. Potřebujeme tedy něco dalšího.

2.2.6 Předávání zpráv (Message Passing)

Něco dalšího je tedy **předávání zpráv**. Tato metoda IPC používá dvě operace: *SEND* a *RECEIVE*, které jsou stejně jako semafory, ale ne jako monitory, implementovány jako systémová volání, což je lepší, než prvek programovacího jazyka. Můžete tedy do knihovny přidat dva podprogramy, jako např.

```
send( destination, &message )
receive( source, &message )
```

Uvedené volání zašle zprávu do daného cíle a později někdo zprávu z daného zdroje vyzvedne. Když není žádná zpráva k dispozici, příjemce se zastaví, dokud nějaká nepřijde. Alternativně může ihned skončit s chybovým kódem.

Navrhované rysy systému předávání zpráv

Předávání zpráv přináší řadu zajímavých problémů a navrhovaných rysů, které s sebou semafory a monitory nepřinesly, zejména pokud jsou komunikující procesy na jiných počítačích propojených v síti. Například zpráva se může cestou po síti ztratit. Abychom zabránili ztrátě zprávy, odesílatel a příjemce si potvrdí, že jakmile je zpráva přijata, příjemce odesílá zpět speciální **potvrzovací** (acknowledgement) zprávu. Pokud odesílatel neobdrží potvrzovací zprávu v určitém časovém intervalu, tak zprávu pošle znovu.

Podívejme se, co se stane, pokud je samotná zpráva doručena korektně, ale potvrzovací zpráva se ztratí. Odesílatel přepošle zprávu znovu a příjemce

ji obdrží podruhé. Je samozřejmé, že příjemce může odlišit novou zprávu od již dříve přeposlané. Obvykle se tento problém řeší vložením souvislé řady čísel do každé jedinečné zprávy. Pokud příjemce obdrží zprávu nesoucí stejné číslo jako předchozí, tak ví, že jde o zprávu opakovanou a může ji ignorovat (ale musí ji potvrdit).

Pro implementaci posílání zpráv v rámci jednoho operačního systému je ovšem systém potvrzování zbytečný.

Systém zpráv se také musí vyrovnat s otázkou identifikace procesů, protože volání funkcí *SEND* a *RECEIVE* je nejednoznačné. Důležitým rysem je taky **autentizace**: jak může klient poznat, že posílá zprávu souborovému serveru a ne podvodníkovi?

A na druhém konci spektra jsou důležité vlastnosti, pokud je odesílatel a příjemce na jednom počítači. Jedna z nich je propustnost. Kopírování zprávy z procesu do procesu je vždy pomalejší, než operace se semaforem. Bylo uděláno mnoho práce, aby bylo předávání zpráv efektivní.

Problém výrobce–spotřebitel s předáváním zpráv

Podívejme se na řešení problému výrobce–spotřebitel pomocí předávání zpráv bez sdílené paměti. Řešení je v příkladu 8. Předpokládáme, že všechny zprávy mají stejnou velikost a odeslané nevyzvednuté zprávy se automaticky ukládají do bufferu operačním systémem. V tomto řešení je použito maximálně N zpráv, analogicky N pozicím v bufferu ve sdílené paměti. Spotřebitel začíná posláním N prázdných zpráv výrobci. Jakmile má výrobce data pro spotřebitele, vezme si prázdnou zprávu a pošle zpátky jednu naplněnou. Takto je celkový počet zpráv v systému konstantní a může být uložen v předem známém množství paměti.

Pokud výrobce pracuje rychleji než spotřebitel, skončí se všemi naplněnými zprávami a bude čekat na spotřebitele (výrobce bude zablokován čekáním na prázdnou zprávu od spotřebitele). Pokud je rychlejší spotřebitel, nastane opačná situace. Všechny prázdné zprávy budou čekat na výrobce, aby je naplnil (spotřebitel zůstane zablokován čekáním na zprávu s daty).

Předávání zpráv nabízí široké možnosti řešení. Pro začátečníky by určitě bylo dobré se podívat, jak se zprávy adresují. Jedna možnost je přiřadit každému procesu unikátní adresu a mít zprávy adresované procesům. Další možností je vytvořit novou datovou strukturu, nazývanou **schránka** (mailbox). Schránka je místo pro bufferování určitého počtu zpráv, typicky uvedeného při vytvoření schránky. Když se používá schránka, je typickým parametrem podprogramů *SEND* a *RECEIVE* adresa schránky, ne proces. Když se proces pokusí poslat zprávu do plné schránky, je pozastaven, dokud není zpráva ze schránky vyjmuta, čímž uvolní místo.

V problému výrobce–spotřebitel potřebují obě strany vytvořit schránky dostatečně velké pro N zpráv. Výrobce by měl posílat zprávy s daty do schránky spotřebitele a ten by naopak měl posílat prázdné zprávy do schránky výrobce. Při používání schránek je princip bufferování jasný: v cílové schránce

```
#define N 100
// počet položek ve frontě

void producer()
{
  int item;
  message m;                                  // fronta zpráv

  while( 1 )
  {
    produce_item( &item );
// vygenerování dat
    receive( consumer, &m );
// příjem prázdné zprávy
    build_message( &m, item );
// sestavení zprávy k odeslání
    send( consumer, &m );
// odeslání zprávy spotřebiteli
  }
}

void consumer()
{
  int item;
  message m;

  for ( int i = 0; i < N; i++ )
// odeslání N prázdných zpráv
    send( producer, &m );

  while ( 1 )
  {
    receive( producer, &m );
// převzetí zpráv s daty
    extract_item( &m, &item );
// oddělení dat ze zprávy
    send( producer, &m );
// odeslání prázdné zprávy zpět
    consume_item( item );
// a teď si to užijeme...
  }
}
```

Příklad 8: Návrh řešení problému výrobce–spotřebitel s posíláním zpráv

se hromadí odeslané zprávy pro cílový proces, pokud nebyly přijaty.

Druhý extrém může být schránka bez bufferu. Pokud se použije tento princip, zablokuje se *SEND*, pokud je použit dříve než *RECEIVE* a zůstává zablokován až do jejího zavolání příjemcem, kdy je zpráva přímo zkopírována od odesílatele k příjemci bez dočasného bufferu. Podobně, když se volá *RECEIVE* jako první, zůstane zablokován až do volání *SEND*. Tuto strategii nazýváme jako **schůzka** nebo **setkání** (rendezvous). Je snadnější na implementaci, než bufferování, ale méně přizpůsobivá, protože odesílatele a příjemce nutí k zastavení.

IPC mezi uživatelskými procesy v UNIXu je pomocí rour, což jsou efektivní schránky. Jediný reálný rozdíl mezi schránkami a rourami je v tom, že roury neevidují hranice zpráv. Jinými slovy, pokud proces pošle 10 zpráv délky 100 bytů, jiný proces přečte z roury 1000 bytů, zatím co ze schránky přijme opět 10 zpráv. Ve skutečném systému posílání zpráv každý *READ* přečte jen jednu zprávu. Jistě, pokud procesy akceptují posílání zpráv konstantní velikosti, nebo ukončené speciálním znakem, nebude žádný problém.

2.3 Klasické IPC problémy

Literatura o operačních systémech je plná zajímavých problémů, které jsou široce rozebírány a analyzovány. V následujícím textu rozebereme tři nejznámější problémy.

2.3.1 Večeřící filozofové

V roce 1965 Dijkstra předložil a vyřešil synchronizační problém, který nazval **Večeřící filozofové** (Dining Philosophers Problem). Od té doby všichni vymýšlejí nové synchronizační nástroje, u kterými cítí povinnost jimi demonstrovat, jak jsou nové nástroje elegantní při řešení problému večeřících filozofů.

Problém můžeme představit jednoduše. Pět filozofů sedí kolem kulatého stolu, každý má před sebou talíř se špagetami. Aby se daly špaget jíst, potřebuje každý filozof dvě vidličky. Mezi každou dvojicí talířů leží jedna vidlička (tedy kolik filozofů, tolik talířů i vidliček).

Život filozofa spočívá ve střídání dvou period, jedení a přemýšlení (toto je samozřejmě jen účelová abstrakce pro náš příklad). Když má filozof hlad, pokusí se vzít ze stolu vidličku do levé i pravé ruky současně. Když úspěšně získá dvě vidličky, může chvíli jíst, pak položí obě vidličky zpět na stůl a přemýšlí. Klíčová otázka je: můžeme napsat program pro každého filozofa tak, aby dělal co jsme si vysvětlili a přitom nikdy neuvízl? Neuvažujte prosím o možnosti, že by filozofové šli raději do čínské restaurace, kde nemají špagety.

Příklad 9 nabízí snadné řešení. Podprogram *take_fork* čeká, dokud se neuvolní vybraná vidlička a pak si ji vezme. K naší smůle, snadné řešení je

```
#define N 5                                    // počet filozofů

void philosopher( int n )                      // n-tý filozof
{
  while ( 1 )
  {
    think();                                   // přemýšlení...
    take_fork( n );
// vezmi levou vidličku
    take_fork( ( n + 1 ) % N );
// vezmi pravou vidličku
    eat();
// hmm, to je dobrota...
    put_fork( n );
// polož levou vidličku
    put_fork( ( n + 1 ) % N );
// polož pravou vidličku
  }
}
```

Příklad 9: Nevyřešený problém večeřících filozofů

špatné. Předpokládejme, že všech pět filozofů vezme současně levou vidličku. Žádný už nemůže vzít pravou vidličku a dojde k zablokování.

Můžeme opravit program tak, že po získání levé vidličky se podíváme, jestli je pravá volná. Pokud ne, tak položí levou vidličku zpět na stůl, chvíli počká a pak zkusí vše zopakovat. Toto řešení opět selhává, i když z jiného důvodu. Při troše smůly začnou všichni filozofové současně a když zjistí, že druhá vidlička není k dispozici, položí první zpět na stůl, chvíli počkají, vezmou současně levou vidličku a tak pořád do nekonečna. Taková situace, kdy všechny programy pracují do nekonečna, ale nejsou schopny cokoliv vyprodukovat, se nazývá **hladovění**.

A co kdyby filozofové čekali náhodný čas, místo stejného, po selhání získání druhé vidličky? Šance na zablokování po dlouhou dobu bude velmi malá. Tento postřeh je pravdivý, ale v některých aplikacích potřebujeme řešení, které pracuje vždy a nemůže selhat zaviněním série náhodných čísel.

Jedno vylepšení v příkladu 9 bez zablokování a hladovění je takové, že za volání funkce *think* vložíme binární semafor. Než filozof vezme ze stolu vidličky, může být zastaven voláním *DOWN* na *mutexu*. Jakmile vrátí vidličky, měl by zavolat *UP mutexu*. Z teoretického pohledu je toto řešení dobré. Z praktického hlediska je zde významná chyba výkonnosti: v kterémkoliv okamžiku může jíst jen jeden filozof. S pěti vidličkami bychom měli povolit jíst dvěma filozofům současně.

```
#define N         2                    // počet filozofů
#define LEFT      ( n - 1 ) % N        // levý soused
#define RIGHT     ( n + 1 ) % N        // pravý soused
#define THINKING  0                    // filozof přemýšlí
#define HUNGRY    1                    // .. čeká na vidličky
#define EATING    2                    // .. jí

int state[ N ];                        // stavy jednotlivých filozofů
semaphore mutex = 1;                   // mutex pro kritickou sekci
semaphore f[ N ];                      // pro každého filozofa jeden mutex

void philosopher( int n )
{
  while ( 1 )
  {
    think();                           // filozof přemýšlí
    take_forks( n );                   // bere dvě vidličky, nebo zablokován
    eat();                             // hmm, mňam, mňam..
    put_forks( n );                    // položí vidličky na stůl
  }
}

void take_forks( int n )
{
  down( &mutex );                      // vstup do kritické sekce
  state[ n ] = HUNGRY;                 // filozof zjistil, že je hladový
  test( n );                           // pokus o získání dvou vidliček
  up( &mutex );                        // konec kritické sekce
  down( &f[ n ] );                     // zablokování, pokud nejsou vidličky
}

void put_forks( n )
{
  down( &mutex );                      // vstup do kritické sekce
  state[ n ] = THINKING;               // filozof dojedl ...
  test( LEFT );                        // může jíst levý soused?
  test( RIGHT );                       // může jíst pravý soused?
  up( &mutex );                        // výstup z kritické sekce
}
```

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

Příklad 10: Řešení problému večeřících filozofů

Řešení v příkladu 10 je správné a také povoluje maximální paralelismus pro libovolný počet filozofů. Používá se pole *state* pro sledování situace, v jaké je filozof: jedení, přemýšlení a čekání na vidličky. Filozof může začít jíst pouze v okamžiku, kdy žádný jeho soused nejí. Sousedi filozofa jsou definováni jako makro *LEFT* a *RIGHTS*.

Program také používá pole semaforů, vždy jeden pro každého filozofa, tak aby se hladovějící filozof mohl zablokovat, při požadavku na vidličky. Podprogram *philosopher* představuje hlavní cyklus programu a může být přímo v podprogramu *main*. Pro každého filozofa bude existovat jeden samostatný proces. Ostatní podprogramy jsou obecné - *take_forks*, *put_forks* a *test* a netvoří samostatné procesy.

2.3.2 Problém čtenářů a spisovatelů (Readers and Writers Problem)

Příklad večeřících filozofů je užitečný pro modelování procesů, které spolu soutěží o výhradní přístup k omezenému počtu zdrojů, jako jsou V/V zařízení. Jiný známý problém je „čtenáři a spisovatelé“, který modeluje zápis do databáze. Představme si například rezervační systém aerolinek, kde spolu soutěží mnoho procesů o čtení a zápis do souboru. Můžeme připustit, aby více procesů současně v jeden okamžik ze souboru četlo, ale pokud jeden proces zapisuje do databáze, žádný jiný proces nesmí do databáze přistupovat, dokonce ani pro čtení. Otázka je, jak naprogramovat čtenáře a spisovatele. Jedno řešení je v příkladu 11.

V tomto řešení první čtenář se získáním přístupu do databáze zablokuje pomocí *DOWN* semaforu *db*. Ostatní čtenáři už jen inkrementují čítač *numr*. Jakmile čtenář končí, sníží čítač a poslední odemkne semafor *db* pomocí *UP*, čímž povolí případným zablokovaným spisovatelům zapisovat.

Uvedené řešení obsahuje jeden základní problém, který musíme zmínit.

Předpokládejme, že v době, kdy čtenář přistupuje do databáze, může do databáze začít přistupovat další čtenář. Mít dva čtenáře není problém, proto je čtenář přijat. A může být souběžně přijat třetí čtenář, pokud o to požádá.

Nyní předpokládejme, že přijde spisovatel. Ten nemůže být přijat, protože požaduje výhradní přístup do databáze a je tedy zablokován. Později jej probudí některý z čtenářů. Dokud je ale aktivní alespoň jeden čtenář, jsou stále přijímáni další čtenáři. V souvislosti s touto strategií, pokud bude trvat podpora čtenářů, získají čtenáři přístup vždy, jakmile o to požádají. Spisovatel zůstane zablokován, dokud neodejde poslední čtenář. Pokud přichází čtenáři každé 2 vteřiny, a řekněme, že pracují 5 vteřin, pak se spisovatel nikdy neprobudí.

Abychom předešli této situaci, musí být program napsán mírně odlišně: když přijde čtenář a je zde čekající spisovatel, je čtenář zablokován, místo aby byl okamžitě přijat. Takto čeká spisovatel jen na právě aktivní čtenáře a nemusí čekat na čtenáře, kteří přijdou až po něm. Nevýhodou řešení je dosažení mírně nižší součinnosti, a tedy nižšího výkonu. Courtois a kol. představil řešení, které dává přednost zapisovatelům.

2.3.3 Spící holič (Sleeping Barber Problem)

Dalším klasickým IPC problémem je holičství. V holičství je jeden holič, jedno holičské křeslo a n židlí pro čekající zákazníky, pokud nějací jsou. Pokud nejsou zákazníci, sedí na křesle holič a spí. Když přijde zákazník, musí probudit spícího holiče. Pokud přijde další zákazník v době, kdy holič stříhá zákazníka, tak se posadí, je-li volná židle, jinak ihned odchází, jsouli židle obsazeny. Hlavní problém je naprogramovat tento případ tak, aby nedošlo k souběhu.

Naše řešení použije 3 semafory: *customers* pro počítaní čekajících zákazníků, kromě zákazníka na křesle, *barbers* počítá volné holiče čekající na zákazníky a nakonec *mutex* pro vzájemné vyloučení. Proměnná *waiting* počítá zákazníky, je v podstatě kopií *customers*. Hlavní důvod proč mít proměnnou *waiting* je ten, že ne vždy umíme přečíst aktuální hodnotu semaforu. V tomto řešení se při vstupu zákazníka do holičství počítají čekající zákazníci. Pokud je jejich počet menší, než počet židlí, tak si sedají, jinak odchází.

Naše řešení je v příkladu 12. Když se ráno objeví holič v práci, provede podprogram *barber*, který jej přinutí zastavit (a usnout) na semaforu *customers*, dokud někdo nepřijde.

Když přijde zákazník, zavolá podprogram *customer*, začínající uzamčením semaforu *mutex* před vstupem do kritické sekce. Pokud hned za ním přijde další zákazník, tak bude muset počkat, až se uvolní *mutex*. Zákazník pak zkontroluje, zda je počet čekajících menší, než počet židlí. Pokud ne, uvolní *mutex* a odchází bez ostříhání.

Pokud je volná židle, inkrementuje proměnnou *waiting* a provede *UP* semaforu *customers*, aby probudil holiče. V tomto okamžiku jsou probuzeni

```
semaphore mutex = 1;                    // ochrana numr
semaphore db = 1;
// přístup do databáze
int numr = 0;                           // počet čtenářů

void reader()
{
  while ( 1 )
  {
    down( &mutex );
// kritická sekce pro numr
    numr++;                             // další čtenář
    if ( numr == 1 ) down( &db );
// zablokování výhradního přístupu
    up( &mutex );
// konec kritické sekce
    read_data();
// čtení dat z databáze
    down( &mutex );
// kritická sekce pro numr
    numr--;
// další čtenář odchází
    if ( numr == 0 ) up( &db );
// poslední čtenář odchází
    up( &mutex );
// konce kritické sekce
    use_data();                         // práce s daty
  }
}

void writer()
{
  while ( 1 )
  {
    do_something();                     // příprava dat
    down( &db );
// žádost o výhradní přístup
    write_data();
// zápis dat do databáze
    up( &db );
// konec výhradního přístupu
  }
}
```


Příklad 11: Řešení problému čtenářů a spisovatelů

oba, zákazník i holič. Jakmile zákazník uvolní *mutex*, přivlastní si jej holič, chvíli hospodaří a pak začne stříhat.

Po ukončení stříhání zákazník ukončí podprogram a odchází z holičství. Na rozdíl od dřívějšího příkladu, není zde smyčka pro zákazníky, protože každý se stříhá jen jednou. Nicméně opakováním u holiče se pokusíme vzít dalšího zákazníka. Pokud nějaký je, tak stříháme. Pokud ne, usínáme.

Jen tak mimochodem, je důležité poukázat, že ač čtenáři a spisovatelé i spící holič nevyvolávají žádné datové přesuny, stále patří do oblasti IPC, protože obsahují synchronizaci mezi více procesy.

2.4 Plánování procesů

V příkladech předchozích kapitol jsme často měli situaci, kdy jeden nebo více procesů, bylo připraveno ke spuštění (viz stavy procesu 2). Když je připraveno více procesů ke spuštění, operační systém musí rozhodnout, který spustí jako první. Ta část operačního systému, která toto rozhodování provádí, se nazývá **plánovač** (scheduler) a použitý algoritmus rozhodování je **plánovací algoritmus** (scheduling algorithm).

Dříve, v dávkových systémech se vstupem z děrných štítků a magnetických pásek, byl plánovací algoritmus velmi jednoduchý: jen spustit další dávku z pásky. V systému sdíleného času je plánovací algoritmus daleko složitější. Obvykle zde na obsloužení čeká více uživatelů a může zde být i více pásek s dávkami. A to nemluvíme o personálním počítači, kde může mít uživatel více procesů spuštěných současně, nehledě na procesy na pozadí, jako je příjem a odesílání elektronické pošty.

Než se podíváme na konkrétní plánovací algoritmy, měli bychom se zamyslet, čeho se vlastně plánovač snaží dosáhnout. Konec konců, plánovač zajišťuje principy a nevykonává mechanismus. Při sestavování dobrého plánovacího mechanismu nás okamžitě napadnou různá kritéria. Některé možnosti jsou:

1. férovost - dát všem procesů rovnocenné šance sdílení CPU,
2. efektivnost - udržet procesor trvale vytížený na 100%,
3. doba odezvy - minimalizovat odezvu pro interaktivní uživatele,
4. doba běhu - minimalizovat čas, po který musí uživatel čekat na vykonání svého zadání (dávky),
5. průchodnost - maximalizovat počet procesů, které budou vykonány během časové jednotky, např. za hodinu.

Jen krátké zamyšlení a je jasné, že některé z těchto cílů si vzájemně odporují. Pro minimalizaci doby odezvy interaktivním uživatelům, by neměl plánovač spouštět dávky jindy, než mezi třetí a šestou hodinou raní, kdy jsou

```
#define CHAIRS 5
// počet židlí pro zákazníky

semaphore customers = 0;
// počet čekajících zákazníků
semaphore barbers = 0;
// počet čekajících holičů
semaphore mutex = 1;
// vzájemné vyloučení
int waiting = 0;
// počet čekajících zákazníků

void barber()
{
  while ( 1 )
  {
    down( &customers );
// holič usíná, pokud nejsou zákazníci
    down( &mutex );
// výhradní přístup k waiting
    waiting--;
// sníží se počet čekajících zákazníků
    up( &barbers );
// jeden holič je připraven stříhat
    up( &mutex );
// konec kritické sekce
    cut_hair();                    // stříhání
  }
}

void customer()
{
  down( &mutex );
// výhradní přístup k waiting
  if ( waiting < CHAIRS )          // jsou vodné židle?
  {
    waiting++;
// zvýší se počet čekajících
    up( &customers );              // probouzíme holiče
    up( &mutex );
// konec kritické sekce
    down( &barbers );
// zákazník usíná, pokud není volný holič
    get_haircut();                 // stříhání
  }
  else
    up( &mutex );
// zákazník odchází a nečeká
}
```


Příklad 12: Řešení problému spícího holiče

uživatelé v posteli. Uživatelé zadávající své dávky, ale nebudou z takového kritéria kdoví jak nadšení, protože tento algoritmus narušuje kritérium 4. Dá se snadno ukázat, že algoritmus preferující určitou třídu úloh, škodí jiným úlohám. Množství procesorového času je přece jenom časově omezené. Přidělení většího množství jednomu uživateli musí jasně ubrat jinému. To je jak v životě.

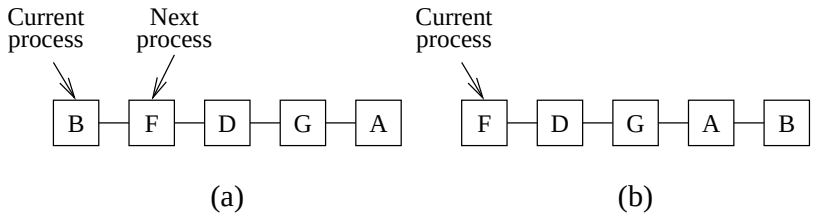
Komplikace, se kterými se musí plánovač vypořádat, jsou neočekávané a navíc každý proces je unikátní. Některý čeká dlouhou dobu na V/V operace, zatím co jiný by chtěl používat CPU několik hodin, pokud by k tomu dostal šanci. Když plánovač spustí určitý proces, nikdy neví, jak dlouho bude trvat, než se proces zablokuje, např. na V/V operaci, semaforu, nebo z jiného důvodu. Abychom měli jistotu, že proces neběží příliš dlouho, mají téměř všechny počítače vestavěny elektronické hodiny, které vyvolávají pravidelné přerušení. Frekvence 50 nebo 60Hz je vcelku běžná, ale na většině počítačů může operační systém nastavit frekvenci, jakou potřebuje. Při každém přerušení časovače se spustí operační systém a rozhodne, zda aktuální běžící proces smí pokračovat, nebo zda již měl v této chvíli dostatek procesorového času a může být pozastaven, aby mohl předat CPU jinému procesu.

Strategie, kdy je proces schopný běhu dočasně pozastaven, se nazývá **preemptivní plánování** (preemptive scheduling) (pojem *preemptivní* obvykle nepřekládáme, ale mohli bychom říct, že jde o plánování v předstihu, nebo s předstihem) a je opakem **spusť a dokonči** (run to completion) metody v dávkových systémech. Spuštění s dokončením se také nazývá **nepreemptivní plánování**. Jak uvidíme v této kapitole, proces může být pozastaven kdykoliv bez varování, aby mohl běžet proces jiný. To vede k souběhu a vynutilo si to existenci semaforů, monitorů, zpráv a jiných důmyslných metod pro jeho prevenci. Na druhé straně, princip povolující procesu běh jak dlouho to sám potřebuje, může znamenat, že určitý proces počítající π s přesností na milión míst odstaví ostatní procesy na neurčito.

Ač je tedy nepreemptivní plánování velmi jednoduché a snadno implementovatelné, obvykle není vyhovující pro obecné systémy s více vzájemně si konkurujícími uživateli. Na druhé straně, pro jednoúčelové a vyhrazené systémy, jako např. databázové servery, to může být velmi výhodné, kdy hlavní proces vytváří své potomky pro zpracování požadavků a nechá je běžet, dokud se neukončí nebo nezablokují. Liší se tak od obecného systému tím, že procesy databázového serveru jsou pod kontrolou jediného hlavního procesu, který ví, co potomci vykonávají a jak dlouho jim to bude trvat.

2.4.1 Plánování Round Robin

Podívejme se teď na jeden konkrétní plánovací algoritmus. Jeden z nejstarších, nejjednodušších, nejspravedlivějších a nejčastěji používaných algoritmů je **round robin**. Každý proces má přiřazen časový interval, nazývaný **kvantum** (quantum), po který smí běžet. Pokud je na konci svého kvanta pro-



Obrázek 4: Plánování round robin. (a) Seznam spustitelných procesů. (b) Seznam procesů když proces B vyčerpal své kvantum.

ces stále běžící, je operačním systémem preemptivně přepnut CPU na jiný proces. Když se proces zablokuje nebo ukončí, před vyčerpáním svého časového kvanta, operační systém samozřejmě přepne CPU pro jiný proces. Round robin se snadno implementuje. Vše co plánovač musí evidovat, je seznam spustitelných procesů, jako na obrázku 4. Když proces využije celé své kvantum, je zařazen na konec seznamu.

Jediným zajímavým parametrem u round robina je časové kvantum. Přepnutí z jednoho procesu na druhý vyžaduje určité množství času pro režii – uložení a obnovení registrů, mapy paměti, aktualizace některých tabulek, seznamů a pod. Mějme například pro **přepnutí procesu** nebo **přepnutí kontextu** (process or context switch), jak se obvykle tato činnost nazývá, 5 ms. Pro časové kvantum 20 ms. S těmito parametry po 20 ms užitečné práce ztratí CPU 5 ms s přepnutím procesu. Na režii se tak ztratí 20% procesorového času.

Pro zlepšení efektivity využití CPU můžeme nastavit kvantum např. na 500 ms. Nyní bude ztráta na režii méně než 1%. Ale zamyslete se, co se stane v systému sdíleného času, když deset uživatelů stiskne současně Enter. Deset procesů se zařadí na konec seznamu spustitelných procesů. Pokud byl CPU volný, spustí se první z nich, druhý se spustí asi o půl sekundy později atd. A poslední proces má smůlu, bude muset čekat 5 sekund, než dostane šanci, za předpokladu, že všechny předchozí využijí své časové kvantum. Většina z deseti uživatelů rychle pochopí, že odezva 5 sekund je příšerná. Stejný problém může nastat na personální počítači s podporou multiprogramování.

Závěr můžeme formulovat následovně: nastavení příliš krátkého časového kvanta vede k příliš častému přepínání procesů a snižuje tak efektivitu práce CPU, ale její neúměrné prodlužování způsobí špatnou odezvu na krátké interaktivní požadavky.

V dnešní době je kvantum obvykle 10 ms a čas potřebný pro přepnutí procesu výrazně menší než 1 ms, řádově jen jednotky či desítky μs.

2.4.2 Prioritní plánování (Priority Scheduling)

Plánování round robin má jeden předpoklad, všechny procesy jsou stejně důležité. Často ale lidé spravující multiuživatelský počítač mají o problému jinou představu. Například na univerzitě mohou být lidé seřazeni od děkana, přes profesory, docenty, asistenty, sekretářky, správce, až po studenty. Tato organizační struktura použitá pro účty uživatelů vede k **prioritnímu plánování**. Základní myšlenka je následující: každý proces má přiřazenu prioritu a spuštěn je spustitelný proces s nejvyšší prioritou.

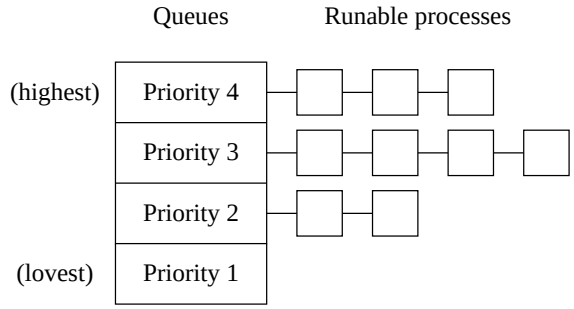
I na personální počítači s jedním uživatelem může být více procesů, kdy jsou některé důležitější, než ostatní. Například služba pro odesílání elektronické pošty na pozadí, může mít přiřazenu nízkou prioritu, na rozdíl od procesu zobrazujícího video v reálném čase.

Abychom předešli situaci, kdy proces s vysokou prioritou může běžet trvale, plánovač snižuje prioritu aktivního procesu každou časovou periodu. Až se tímto sníží priorita pod hodnotu dalšího procesu v řadě, dojde k přepnutí. Alternativně může mít každý proces přiřazeno kvantum času, jak dlouho může běžet v jednom kuse bez přerušení. Když je toto kvantum vyčerpáno, dostane příležitost následující proces s nejvyšší prioritou.

Priority mohou být procesům přiřazeny staticky i dynamicky. Na počítači v armádě může být priorita začínat na 100, procesy spuštěné plukovníkem budou mít 90, majoři 80, kapitáni 70, poručíci 60, atd. Alternativně na komerčním počítači může proces s nejvyšší prioritou představovat cenu 100$ za hodinu, střední priorita 75$ a nejnižší 50$ za hodinu. Unixové systémy mají příkaz *nice*, který uživateli povoluje dobrovolně snižovat prioritu svých procesů, aby byli ohleduplnější k ostatním. Nikdo to však nevyužívá.

Priorita se také může přiřazovat dynamicky, aby bylo dosaženo určitého cíle nebo chování. Například některé procesy jsou silně vázány na V/V operace a stráví mnoho času čekáním, až budou vyřízeny jejich V/V požadavky. Kdykoliv takový proces požaduje CPU, může mu být CPU přidělen prakticky okamžitě, protože za okamžik bude opět žádat a následně čekat na V/V operaci. Po dobu čekání může jiný proces paralelně užívat CPU pro výpočet. Nechat čekat procesy vázané na V/V operace znamená, že proces zabírá v paměti místo po příliš dlouhou dobu. Jednoduchá metoda pro dobrou obsluhu procesů vázaných na V/V je nastavení priority jako $1/f$, kde f je poslední použité časové kvantum. Proces, který použije ze svého kvanta jen 2 ms ze 100 možných, dostane prioritu 50, proces který použije 50 ms, bude mít prioritu 2 a proces, který využil celé své časové kvantum, bude mít prioritu 1.

Je vhodné slučovat procesy do skupin se stejnou prioritou a používat prioritní plánovaní mezi skupinami, zatím co ve skupině se používá plánování round robin. Na obrázku 5 je systém se čtyřmi prioritními skupinami. Plánovací algoritmus je následující: dokud jsou spustitelné procesy ve třídě 4, dostane každý své časové kvantum podle pravidel plánování round robin a



Obrázek 5: Proritní plánování se čtyřmi prioritními třídami

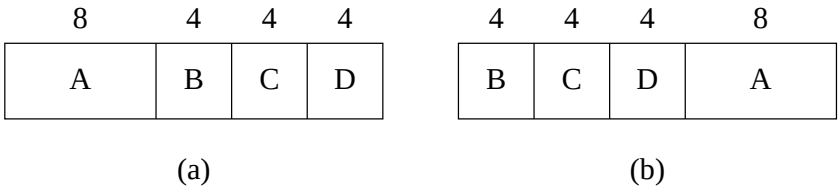
nikdy se nezajímá o skupinu s nižší prioritou. Když bude priorita 4 prázdná, začne se uplatňovat round robin ve třídě 3. Když bude třída 4 i 3 prázdná, přejde se na třídu 2, atd. Pokud nejsou priority nastaveny odpovídajícím způsobem, třídy s nízkou prioritou klidně „umřou hlady“.

2.4.3 Více front

Jeden z prvních prioritních plánovačů byl CTSS. Problém byl v rychlosti, systém byl velmi pomalý, protože mohl mít v paměti jen jeden proces. Každé přepnutí znamenalo odložení aktuálního procesu na disk a načtení dalšího z disku. Návrháři CTSS ihned realizovali pro větší efektivitu, aby procesy silně využívající CPU, pracovaly raději s větším množstvím časového kvanta, než aby se přidělovalo menší kvantum častěji. Na druhé straně, dát všem procesům větší kvantum by znamenalo horší odezvu, jak jsme již dříve zmínili. Řešeni spočívá se stanovení prioritních tříd. Procesy s nejvyšší prioritou mohou běžet jedno časové kvantum. Procesy v další třídě priority mohou běžet dvě kvanta. V další třídě 4 kvanta, atd. Kdykoliv proces využije celé časové kvantum, které mu bylo v dané třídě přiděleno, je přesunut o jednu třídu priority níže.

Mějme příklad procesu, který potřebuje procesor po dobu 100 kvant. Na počátku dostane jedno kvantum a je odložen. Při dalším spuštění už dostane 2 kvanta před odložením. V následujících bězích dostane 4, 8, 16, 32 a 64 kvant. Z posledního použije jen 37 ze 64 možných a ukončí se. Jenom 7 odložení bylo potřeba během celého běhu programu, na místo 100 odložení, kdyby se použil obyčejný algoritmus round robin. Dále by se mohl proces propadat stále hlouběji a hlouběji v jednotlivých prioritních frontách a bude spouštěn čím dál méně častěji, aby šetřil čas CPU pro interaktivní úlohy.

Následující princip byl přijat proto, aby se procesy běžící dlouhou dobu mohly opět stát interaktivními, a nebyly navždy poslední. Kdykoliv je u terminálu daného procesu stisknut Enter, byl proces přesunut do nejvyšší prioritní skupiny, protože se předpokládá, že se stává interaktivní. Jednoho



Obrázek 6: Příklad plánování nejkratší dávky jako první.

krásného dne si však jeden uživatel se svým procesem silně využívajícím CPU všimne, že když na terminálu jednou za pár sekund stiskne Enter, má to pozitivní vliv na odezvu programu. A tak to řekne svým přátelům... A ponaučení: snadno se to řekne, hůř udělá.

Používá se mnoho dalších algoritmů pro přiřazování do prioritních tříd. Například významný systém XDS 940, vyvinutý v Berkeley, měl 4 prioritní třídy nazvané terminál, V/V, krátké a dlouhé kvantum. Když je probuzen proces čekající na terminál, jde hned do nejvyšší prioritní třídy terminál. Když se probudí proces čekající na V/V, jde do druhé třídy. Pokud proces využívá své časové kvantum, je zařazen na počátku do třídy třetí. Nicméně, pokud proces využívá své časové kvantum po delší dobu a nevyužívá terminál a V/V, je přesunut do nejnižší třídy. Mnoho dalších systémů používá podobné principy, aby upřednostnily interaktivní uživatele a ostatní procesy přesunuly na pozadí.

2.4.4 Nejkratší dávka jako první (Shortes Job First)

Většina již zmíněných algoritmů je navržena pro interaktivní systémy. Podívejme se na jeden, určený speciálně pro dávkové systémy, kde je doba běhu předem známá. Například v pojišťovně lze odhadnout docela přesně, jak dlouho poběží dávka s 1000 požadavků, protože podobný úkol se dělá každý den. Když máme několik dávek stejné důležitosti čekajících ve frontě na spuštění, může plánovač použít **nejkratší dávku jako první**. Podívejme se na obrázek 6. Máme zde 4 procesy *A, B, C* a *D* s časem běhu 8, 4, 4 a 4 minuty. Spuštěním v daném pořadí bude průchod systémem pro první proces *A* 8 minut, pro *B* 12 minut, pro *C* 16 minut a konečně pro *D* 20 minut. Průměrná doba je tedy 14 minut.

Nyní předpokládejme spouštění procesů tak, že nejkratší půjde jako první, jako na obrázku 6(b). Průchod systémem nyní bude 4, 8, 12 a 20 minut, průměr bude 11 minut. Nejkratší dávka jako první je nesporně optimálnější. Předpokládejme příklad 4 dávek s dobou běhu *a, b, c* a *d*. První dávka skončí po čase *a*. Druhá po čase $a+b$, atd. Průměrná doba běhu tedy bude $(4a+3b+2c+d)/4$. Je jasné, že *a* má největší vliv na průměrnou dobu na rozdíl od dalších časů, a proto musí být nejkratší čas jako první, pak *b*, *c* a nakonec *d*, který svou délkou ovlivní průměr jen jedenkrát. Stejný postup

platí pro libovolný počet dávek.

Protože nejkratší dávka jako první zajišťuje nejkratší průměrnou dobu odezvy, může být dobré použít tuto metodu pro interaktivní procesy. S malým rozšířením by to šlo. Interaktivní procesy obvykle pracují podle následujícího vzoru: čekat na příkaz, vykonat příkaz, čekat na příkaz, vykonat příkaz, atd. Pokud budeme dbát na oddělení všech příkazů jako samostatných dávek, tak můžeme minimalizovat celkovou dobu odezvy spuštěním nejkratší dávky jako první. Jediný problém zůstává, jak seřadit dávky podle času trvání.

Jedna metoda vychází z předpokladu, založeném na předchozím chování a spuštění předpokládaného nejkratšího procesu jako prvního. Předpokládejme, že odhadovaný čas trvání procesu pro určitý terminál je T_0. V dalším měření to bude T_1. Můžeme tak upřesnit náš odhad váženým součtem těchto dvou čísel, který je $aT_0 + (1 - a)T_1$. Zde výběr a rozhoduje, zda odhad má rychle zapomínat dřívější hodnoty běhu, nebo zda si je má pamatovat po dlouhou dobu. Při $a = 1/2$ dostaneme následující odhad:

$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2$

Po třetím spuštění je váha T_0 nového odhadu snížena na 1/8. Technika odhadu následující hodnoty v řadě váženým průměrem aktuálního měření a předchozího odhadu se obvykle nazývá **stárnutím** (aging). Je použitelná v mnoha situacích, kdy se musí předpovídat na základě předchozích hodnot. Stárnutí se snadno implementuje, zejména když $a = 1/2$. Stačí přidat aktuální měření k předchozímu odhadu a dělit 2.

Je nutno zmínit, že spouštění nejkratší dávky jako první je optimální jen v okamžiku, když jsou všechny dávky k dispozici současně. Jako špatný příklad mějme dávky A až E s dobou běhu 2, 4, 1, 1 a 1. Čas vzniku požadavku je 0, 0, 3, 3, 3.

Na začátku může být výběr jen z A a B, protože další dávky ještě nejsou k dispozici. Použitím nejkratší dávky jako první dostaneme pořadí *A, B, C, D, E* s průměrným čase 4.6. Nicméně spuštění v pořadí *B, C, D, E, A* má průměr 4.4.

2.4.5 Zaručené plánování (Guaranteed Scheduling)

Zcela odlišný přístup plánování je založen na reálném slibu o výkonu pro uživatele a pak zajištění příslušného chování. Jeden realistický slib, který se pak dá v praxi dodržovat, je tento: pokud máme n přihlášených a pracujících uživatelů, mohou dostat $1/n$ procesorového času. Podobně na jednouživatelském systému s n spuštěnými rovnocennými procesy, může každý dostat opět $1/n$ z času CPU.

Abychom mohli slib dodržet, musíme sledovat, kolik procesorového času dostal proces od svého spuštění. Pak můžeme spočítat čas procesoru, na který má proces nárok, konkrétně čas od spuštění vydělený n. A protože čas spotřebovaný procesem je známý, je snadné dále spočítat poměr spo-

třebovaného procesorového času a času, na který má proces nárok. Poměr 0.5 znamená, že proces zatím spotřeboval jen polovinu času, na který měl nárok, zatím co poměr 2.0 znamená, že proces již spotřeboval dvojnásobek předpokládaného času. Plánovač pak spouští procesy s nízkým poměrem, dokud se poměr nezvýší nad úroveň dalších procesů.

2.4.6 Plánování losováním (Lottery Scheduling)

Zatím co slib uživateli, jak dlouho bude pracovat, je hezká myšlenka, špatně se implementuje. Nicméně můžeme použít další algoritmus s podobným předvídatelným výsledkem, který se snadněji implementuje. Nazýváme ho **plánování losováním**.

Hlavní myšlenka loterie spočívá v nutnosti mít losy pro různé zdroje v systému, jako třeba pro CPU. Kdykoliv musí plánovač rozhodnout, vybere náhodně los a proces, který je jeho majitelem, získá požadovaný zdroj. Použitím pro plánování procesorového času, může systém losovat 50x za sekundu, kdy každý vítěz dostane 20 ms procesorového času jako výhru.

Podle zákona: „Všechny procesy jsou si rovny, některé však rovnější“ můžeme některým procesům přidělit nějaký ten los navíc, abychom zvýšili jejich šanci na výhru.

Pokud bylo vydáno 100 losů a jeden proces jich má 20, pak má 20% šanci na výhru v každé loterii. Během dlouhé doby běhu programu dostane proces 20% procesorového času. Ve srovnání s prioritním plánováním, kde se těžko chápe, co je to priorita 40, tady je to jasné. Proces vlastnící $n\%$ losů dostane stejné procento $n\%$ požadovaných zdrojů.

Plánování losováním má několik zajímavých nastavení. Například, když se spustí nový proces s určitým množstvím přidělených losů, má šanci na výhru ihned při nejbližším losování, úměrnou počtu losů. Jinými slovy, plánování losování má dobrou odezvu.

Spolupracující procesy mohou mezi sebou měnit losy, pokud si to přejí. Například když klient pošle zprávu serveru a zablokuje se, může mu předat i své losy, aby zvýšil šanci serveru na výhru v nejbližší loterii. Až server skončí, vrátí losy klientovi, aby se ten mohl znovu spustit. Když nejsou klienti, server nepotřebuje žádné losy.

Plánování losováním může být použito i pro případy, které je těžké řešit jinými metodami. Jedním příkladem může být video server, který poskytuje proud dat klientům, ale každému jiný počet snímků za sekundu. Představme si, že procesy požadují 10, 20 a 25 snímků za sekundu. Pokud jednotlivým procesům přidělíme příslušný počet losů, rozdělíme procesorový čas automaticky v příslušném poměru.

2.4.7 Plánování Real–Time

Real–time systém je takový, kde čas hraje hlavní roli. Typicky jedno nebo více externích zařízení generuje pro počítač podněty a ten na ně musí odpovídajícím způsobem reagovat v určitém časovém intervalu. Například počítač v přehrávači CD dostává bity, které odcházejí z mechaniky a musí je zkonvertovat ve velmi krátkém čase na hudbu. Pokud bude přepočet trvat příliš dlouho, bude hudba znít trhaně. Jiným RT systémem je monitorování pacientů na jednotce intenzívní péče v nemocnici, autopilot v letadle nebo kontrola bezpečnosti jaderného reaktoru. Ve všech těchto případech je pozdní získání správné odpovědi stejně špatné, jako nemít odpověď žádnou.

RT systémy se obecně dělí na **silné RT** (hard RT) systémy, kde je absolutní mez, která se musí dodržet, nebo na **slabé RT** (soft RT) systémy, kde občasné překročení limitu není na závadu. Ve všech případech je RT chování dosaženo rozdělením programu na mnoho procesů se znalostí a předvídatelností jejich chování. Tyto procesy mají obvykle jen krátký život a ukončí svou činnost během sekundy. Když se objeví externí událost, je úkolem plánovače naplánovat procesy tak, aby byl dodržen časový limit.

Události v RT systému, na které je nutno reagovat, se mohou dělit na periodické (pravidelné) a aperiodické (náhodné). Systém musí odpovídat na více proudů synchronních událostí. V závislosti na čase potřebném pro jejich provedení, nemusí být schopen je všechny obsloužit. Například pro m periodických událostí a událost i nastane s periodou P_i a vyžaduje C_i sekund CPU pro vyřízení všech událostí, můžeme zátěž vyjádřit následovně:

$$\sum_{i=0}^{m} \frac{C_i}{P_i} \leq 1$$

RT systém, který splňuje tuto nerovnost, se nazývá **plánovatelný**.

Jako příklad mějme slabý RT systém se třemi periodickými událostmi s periodou 100, 200 a 500 ms. Pokud tyto události požadují 50, 30 a 100 ms času, je systém plánovatelný, protože $0.5 + 0.15 + 0.2 < 1$. Pokud přidáme čtvrtou událost s periodou 1 sekunda, systém bude plánovatelný, pokud obsluha této události nepřesáhne 150 ms pro každou jednotlivou událost. Implicitně v tomto výpočtu předpokládáme, že přepnutí kontextu je velmi malé a můžeme jej zanedbat.

Plánování RT systému může být dynamické nebo statické. Tvůrce rozhoduje o plánování v reálném čase, nebo může rozhodnout při spuštění systému. Jen v krátkosti si představme několik dynamických RT plánovacích mechanismů. Klasickým algoritmem je **poměrný monotonní algoritmus** (rate monotonic algorithm). Přiřazuje každému procesu prioritu úměrnou periodě příslušné události. Například proces spouštěný každých 20 ms má prioritu 50 a proces spouštěný každých 100 ms má prioritu 10. Za běhu spustí plánovač vždy připravený proces s vyšší prioritou a vynutí si přepnutí, pokud je zapotřebí.

Jiný používaný systém je plánovací algoritmus **nejbližší limit jako první** (earliets deadline firts). Kdykoliv se objeví událost, proces se vloží do seznamu připravených procesů. Seznam je udržován v závislosti na časovém limitu, což je pro periodické události čas jejich dalšího výskytu. Plánovač spustí první proces v seznamu, tedy ten, který má nejbližší časový limit.

Třetí algoritmus spočítá nejprve pro všechny procesy, kolik času jim zbývá a nazve jej **nedbalostí** (laxity). Pokud proces vyžaduje 200 ms a musí být ukončen během 250 ms, je nedbalost 50 ms. Algoritmus, nazývaný **nejmenší nedbalost**, vybírá proces s nejmenším množstvím zbývajícího času.

Zatím co teoreticky je možné použít obecný operační systém v RT systému s použitím jednoho z uvedených algoritmů, v praxi je režie přepínání kontextu obecného operačního systému příliš vysoká a RT výkonnost se dá dosáhnou jedině pro aplikace s malým časovým omezením. Většina RT aplikací používá speciální RT operační systémy, která mají určité důležité vlastnosti. Typicky jsou malé, s malým zpožděním přerušení, rychlé přepínání kontextu, krátké intervaly, kdy je zakázáno přerušení, a schopnost spravovat více časovačů v rozsahu mili- a mikrosekund.

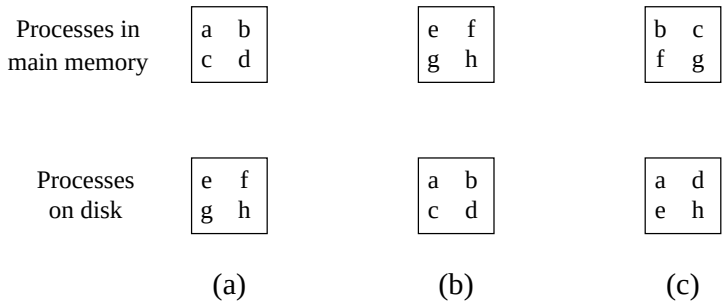
2.4.8 Dvouúrovňové plánování

Až doposud jsme více méně předpokládali, že všechny procesy jsou v hlavní paměti. Pokud ale paměti není dostatek, některé spustitelné procesy jsou uloženy na disku, a to buď celé, nebo i po částech. Tato situace má hlavní důsledek pro plánování, protože čas potřebný pro přepnutí procesu uloženého na disku je výrazně vyšší, než čas pro přepnutí procesů v paměti.

Praktickým řešením manipulace s odloženými aplikacemi je použití dvouúrovňového plánování. Určitá podmnožina spustitelných procesů je na začátku natažena do hlavní paměti, jako na obrázku 7(a). Plánovač pak sám sebe omezí, aby na chvíli používal jen tuto podmnožinu. Pravidelně se vyvolá plánovač vyšší priority a ten odstraní proces, který už byl v paměti příliš dlouho a nahradí ho procesem, který byl příliš dlouho na disku. Jakmile se provede změna, jako na obrázku 7(b), plánovač s nižší prioritou se opět omezí na množinu procesů, které jsou právě v paměti. Plánovač s nižší prioritou se tedy soustředí na výběr mezi spustitelnými procesy v paměti, zatím co plánovač s vyšší prioritou se věnuje přehazování procesů mezi pamětí a diskem.

Mezi kritéria, která by měl plánovač s vyšší prioritou používat při rozhodování, patří tato:

1. Jak dlouhý čas uplynul od odložení nebo natažení procesu?
2. Kolik procesorového času právě proces použil?
3. Jak je proces velký? Malé procesy nás nezajímají.



Obrázek 7: Dvouúrovňové plánování musí sřídat procesy mezi hlavní pamětí a diskem. Situace ve třech časových okamžicích je reprezentována situací (a), (b) a (c).

4. Jaká je priorita procesu?

I zde můžeme použít round robin, prioritní, nebo i jiný plánovač, nebo řadu jiných metod. Oba plánovače mohou, ale nemusí, použít stejný algoritmus.

2.4.9 Princip versus mechanismus (Policy versus Mechanism)

Až dosud jsme tiše předpokládali, že každý proces v systému patří jinému uživateli a proto soutěží o CPU. I když je to často pravda, někdy se stane, že jeden proces má více potomků, pracujících pod jeho řízením. Například řízení databázového systému může mít mnoho potomků. Každý potomek může zpracovávat vlastní požadavek, nebo každý může vykonávat svou specifickou funkci. Je zcela možné, že hlavní proces má přesnou představu, který z potomků je důležitější, a který méně. Nicméně, žádný z uvedených plánovačů dosud uvedených, nepřijímá žádný vstup z uživatelských procesů pro rozhodování při plánování. A výsledek? Plánovač stěží udělá správné rozhodnutí.

Řešení problému je v oddělení **plánovacího mechanismu** od **plánovacího principu**. Co to přináší, je možnost určité parametrizace, kterou musí zadat uživatelské procesy. Představme si znovu databázový příklad. Mějme jádro systému s prioritním plánováním, které nabízí možnost použít systémové volání pro změnu a nastavení priority pro své potomky. Takto může rodičovský proces přesněji ovlivňovat plánování svých potomků, i když sám plánování neřídí. Tady je mechanismus v jádře systému, ale strategii si určují procesy samy.