

Procesy a vlákna

Eva Chludová

Bakalářská práce
2005



Univerzita Tomáše Bati ve Zlíně
Fakulta technologická

Ráda bych zde poděkovala vedoucímu mé bakalářské práce Ing. Martinu Syslovi Ph.D. za odborné vedení, připomínky a čas věnovaný mé práci.

Prohlašuji, že jsem na celé diplomové práci pracovala samostatně a použitou literaturu jsem citovala.

Ve Zlíně, 30. 08. 2005

.....

jméno diplomanta

ABSTRAKT

Abstrakt česky

Práce popisuje základní stavy procesů, přechody mezi jejich jednotlivými stavy a také se zabývá různými strategiemi plánování. Dále pojednává o vzájemné komunikaci mezi procesy za pomoci sdílené paměti anebo zasíláním zpráv. V bakalářské práci jsou vysvětleny základní pojmy a problémy týkající se správy procesů, jsou zde také nastíněna některá řešení, jak se jich vyvarovat. Práce se podrobněji zabývá problematikou stavů procesu, jejich plánováním a komunikací mezi procesy a vlákny.

Praktická část se zabývá několika algoritmy, které s pomocí funkce zasílání zpráv nebo semaforů řeší možné situace uvážnutí, souběhu nebo cyklického čekání.

Abstrakt ve světovém jazyce

This work discusses basic states of processes, passing between their states and it deals also many scheduling strategies. Furthermore it deals mutual communication between processes by means of shared memory or sending messages. In this bachelor work are interpreted basic concepts and problems relating the managing of the processes, there are also outlined some of the solutions, how to avoid them. This work considers the problematic of processes status, their planning and communication between processes and threads, in more details.

The practical part deals with the algorithms which solve some possible situations of deadlock, race condition or circular wait using the function of sending messages.

OBSAH

ABSTRAKT	3
OBSAH	4
ÚVOD.....	6
I. TEORETICKÁ ČÁST	7
1 ZÁKLADNÍ POJMY	8
2 PROCES A STAVOVÝ MODEL PROCESŮ	10
2.1 STAVY PROCESU A JEHO ŽIVOTNÍ CYKLUS	10
2.2 PŘECHODY MEZI STAVY PROCESU.....	12
2.3 PŘEPNUTÍ PROCESU	12
2.4 PROCES (PROCESS, TASK)	13
2.4.1 Tabulka procesů	13
2.5 NULOVÝ PROCES	16
2.6 PROCES JAKO ABSTRAKCE	17
2.7 DĚLENÍ OS PODLE PROCESŮ	18
3 PLÁNOVAČE A PLÁNOVÁNÍ PROCESŮ.....	19
3.1 PLÁNOVAČE (SCHEDULE)	19
3.2 PLÁNOVÁNÍ PROCESŮ	19
3.3 PLÁNOVÁNÍ PROCESORU.....	21
3.4 STRATEGIE PLÁNOVÁNÍ.....	21
3.4.1 Druhy strategií.....	21
3.5 ZMĚNA KONTEXTU – CONTEXT SWITCH.....	23
4 SPOLUPRÁCE MEZI PROCESY	24
4.1 ZASÍLÁNÍ ZPRÁV.....	24
4.2 SDÍLENÁ PAMĚŤ	25
4.2.1 Souběh – race condition	25
4.2.2 Kritická sekce.....	25
5 UVÁZNUTÍ – DEADLOCK.....	28
5.1 PODMÍNKY UVÁZNUTÍ	28
5.2 ŘEŠENÍ OTÁZKY UVÁZNUTÍ	28
5.3 PŘEDCHÁZENÍ UVÁZNUTÍ	29
5.4 VYHÝBÁNÍ SE UVÁZNUTÍ.....	30
5.5 DETEKCE UVÁZNUTÍ A ZOTAVENÍ.....	31

6	VLÁKNA (SLEDY, THREADS)	32
6.1	ROZDĚLENÍ VLÁKEN	32
6.2	MULTITASKING A MULTITHREADING	33
6.2.1	Multitasking	33
6.2.2	Multithreading	34
6.3	REŽIMY MULTITASKINGU	35
6.3.1	Multitasking v DOSu	35
6.3.2	Kooperativní	35
6.3.3	Preemptivní	37
6.4	ŘEŠENÍ POMOCÍ MULTITHREADINGU	37
6.4.1	Pravidlo 1/10 sekundy	38
6.5	VÍCEVLÁKNOVÁ ARCHITEKTURA	38
6.6	SOUBOJE VLÁKEN	39
6.7	VÝHODY WINDOWS	39
6.8	PERSPEKTIVY MULTITHREADINGU	40
II.	PRAKTICKÁ ČÁST	41
	BANKÉŘŮV ALGORITMUS	42
	VÝROBCE – KONZUMENT	45
	PROBLÉM VEČEŘÍCÍCH FILOZOFŮ	46
	ČTENÁŘI – SPISOVATELÉ	48
	PROBLÉM SPÍCÍHO HOLIČE	49
	ZÁVĚR	50
	SEZNAM POUŽITÉ LITERATURY	51
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	52
	SEZNAM OBRÁZKŮ	53
	SEZNAM PŘÍLOH	54
	PŘÍLOHA P I: BANKÉŘŮV ALGORITMUS	55
	PŘÍLOHA P II: VÝROBCE KONZUMENT	56
	PŘÍLOHA P III: PROBLÉM VEČEŘÍCÍCH FILOZOFŮ	59
	PŘÍLOHA P IV ČTENÁŘI – SPISOVATELÉ	61
	PŘÍLOHA P V PROBLÉM SPÍCÍHO HOLIČE	62
	PŘÍLOHA P VI: CD-ROM	63

ÚVOD

Procesy a vlákna, jsou neoddělitelnou součástí operačních systémů dnešní doby. Pro běžného uživatele jsou to objekty, které téměř nepozoruje, i když prakticky tvoří základní kameny operačního systému a programů běžících v jeho prostředí. Podíváme-li se na příklad na aplikace, kde samozřejmý požadavek dostatečné iterace uživatele s grafickým rozhraním, rozpoznáme často přehlíženou existenci tohoto problému. Přitom se nemusí ani jednat o složitý serverový systém využívající vícevláknovou architekturu.

Jak je zmíněno v předchozím odstavci jedná se o základní kameny a to převážně z programového hlediska. Vlákno jako základní prvek využívající čas procesoru, který je mu od operačního systému přidělen. A nad ním nadřazený proces vlastní toto vlákno (nebo více vláken) a působící jako dozorce a správce.

Probereme-li aplikace z tohoto hlediska, musíme si všimnout, že procesy pracují souběžně v čase. V technickém jazyce je tento problém nazýván souběžné zpracování úloh. Procesy pracují zdánlivě souběžně. Na jednoprocessorovém systému se velmi rychle střídají na procesoru - nazýváme to pseudoparalelismus. Pro skutečný paralelismus potřebujeme více procesorů. Tato část teoretické informatiky zaštiťuje mnoho problémů souvisejících s procesy, například komunikaci mezi procesy a vlákny, nebo přístup k prostředkům.

Často kritickým bodem je přístup k prostředkům. Mezi procesy musí být rozdělen jak čas procesoru tak systémová paměť. Přidělování musí brát ohled na prioritu procesů, na dobu případného zablokování a další faktory.

I. TEORETICKÁ ČÁST

1 ZÁKLADNÍ POJMY

Program

Zápis algoritmu v programovacím jazyce (například strojový kód) je brán jako program. Má tyto vlastnosti: je statický a neměnný, pokud ovšem nebereme v potaz nové vývojové verze programu.

Proces (process, task)

Je tvořen neměnným kódem, konstantami a proměnnými daty, jako jsou stav procesoru, data na zásobníku, globální proměnné, halda, soubory a další. Je to běžící program.

Rozdíl mezi pojmy program a proces

Program jako takový znamená v analogii „recept“, jakýsi předpis nebo algoritmus vyjádřený v různých podobách. Naproti tomu proces je aktivitou, v analogii se dá přirovnat k „vykonávání receptu - vaření“. Má kód programu, vstup i výstup a také stav („čekající“, „běžící“, ...)

Vlákno (Thread)

Je to jednotka procesu, která je uvnitř procesu a existuje jen s procesem a využívá jeho prostředky. Ve většině systémech se ukončením procesu ukončují všechna vlákna. Vlákno je spravováno procesem.

Multitasking

Umožňuje současný běh několika programů.

CPU (Central Processing Unit)

Procesor, srdce a mozek počítače

Systémové volání (system calls)

Systémové volání služeb operačního systému a také rozhraní mezi procesem a operačním systémem

Správa paměti

Obsahuje popis paměti, to je ukazatel, velikost a přístupová práva, je v ní úsek paměti s kódem a instrukcemi programu, data a zásobník, ve kterém jsou návratové adresy, parametry funkcí a procedur a lokální proměnné.

Správa souborů

Zde je uloženo nastavení aktuálního pracovního adresáře, otevřené soubory a způsob jejich otevření pro zápis a čtení a dále pozice v otevřeném souboru. Správce souborů přiděluje prostředky na základě rozhodnutí, otevírá a zavírá soubory (programy i data).

Správa procesů (procesoru)

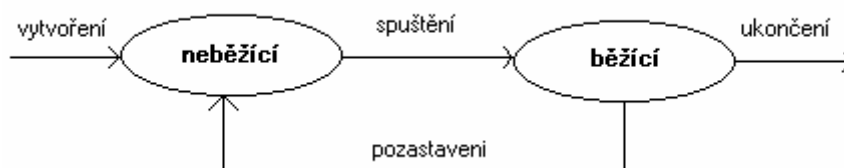
Obsahuje identifikátor procesu. Tím může být řetězec nebo číslo, dále pak stavový vektor, ve kterém je většinou jen koncepčně uložen obsah registrů, ukazatel na další instrukci (obsah čítače instrukcí), najdeme v něm i stav CPU a ukazatel zásobníku. Ve správě procesů se nachází stav procesu, jestli je „běžící“, „připraven“ nebo „blokován“, priorita, která je důležitá pro plánování procesů, odkazy na rodiče a potomky, čas spuštění procesu a čas CPU spotřebovaný procesem. Správce procesu sleduje stav procesu a po určitém čase požaduje navrácení procesoru.

2 PROCES A STAVOVÝ MODEL PROCESŮ

2.1 Stavy procesu a jeho životní cyklus

1. běžící (running) – proces má právě pro sebe čas procesoru, skutečně ho využívá
2. připraven (ready, runnable) - je dočasně pozastaven, aby mohl běžet jiný proces
3. čekající (blocked, waiting) – není schopen pokračovat v běhu procesu, dokud nenastane vnější událost, někdy se označuje názvem blokován

Životní cyklus samotného procesu v operačním systému je dán jeho přechody mezi jednotlivými stavy procesu. Stavový model je pak grafickým znázorněním těchto přechodů.

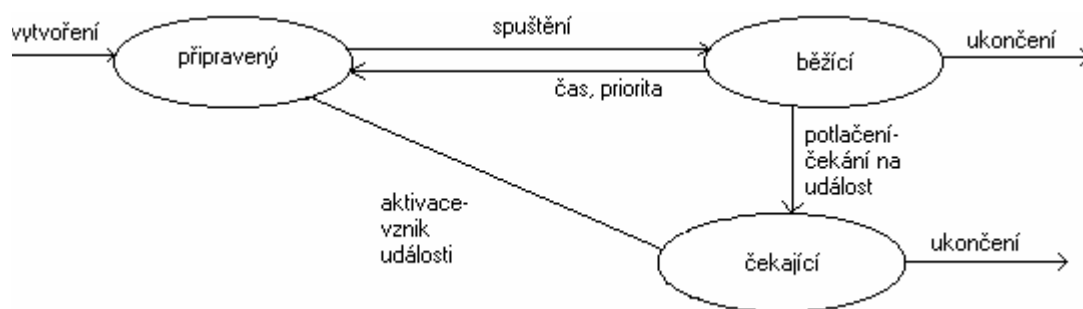


Obr. 1 Základní stavový model procesů

Formální stavový model znázorňuje neběžící procesy ve formě fronty. Tím je dáno přesnější vyjádření stavového modelu třemi stavy procesu.

- připraven(čeká na přidělení času procesoru)
- běžící
- čekající (v případě, že čeká na I/O operaci)

Pro to, aby mohl být proces v nějaké jeho stavu, je nutné, aby byl nový (vytvořen) a po jeho dokončení musí být ukončen a dát vědět, že uvolnil paměť (energetický závislou). Když vznikne proces, může mít potomka a potomky další generace. Přičemž potomek využívá vlastní zdroje systému, které předtím vlastnil jeho rodič.



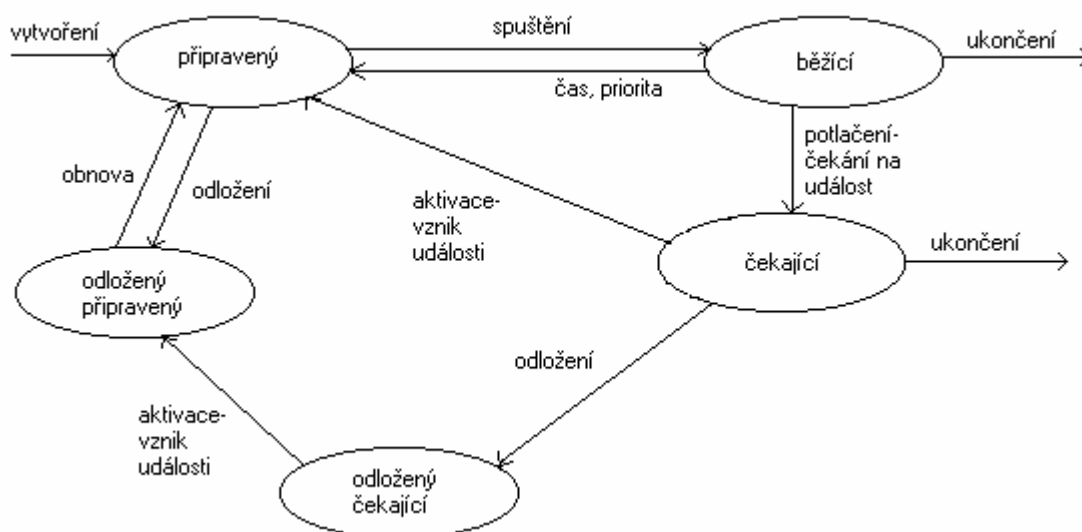
Obr. 2 3-stavový model

Ukončení procesu může nastat čtyřmi způsoby:

- proběhnutím procesu, když je proces ukončen (program proběhl), požádá o vlastní ukončení
- překročením přiděleného času
- když potomek žádá víc, než mu může proces rodič poskytnout
- ukončením nadřazeného procesu

Vytvoření, zrušení, potlačení a obnovení procesu má na starost operační systém. Také poskytuje prostředky pro jejich synchronizaci a komunikaci mezi nimi. Proces může být vytvořen příkazem uživatele anebo žádostí operačního systému, popřípadě i procesem „rodičem“. Pokud je proces vytvořen jednou z těchto možných variant, označujeme jej za „neběžící“. Do stavu „běžící“ se dostane až je vybrán plánovačem.

Každý proces, aby mohl být procesem se musí jednou dostat do operační paměti, alespoň částečně. Problémem může být omezená kapacita paměti. Proto známe ještě další stavový model, který umožňuje odkládání procesů, tzv. swapping. Když se v operační paměti nachází příliš mnoho procesů, snižuje se tím její výkonnost. Z tohoto důvodu se stavový model procesů rozšířil o dva stavy, a to „odložený čekající“ a „odložený připravený“. Tím se provádění některých procesů odloží mimo adresní prostor operační paměti, třeba na disk. Proces se dostane do stavu „odložený čekající“ v případě, že je potřeba uvolnit operační paměť pro proces, který právě „běží“ a potřebuje více prostoru. Nebo i v případech, kdy je čekajících procesů ve frontě moc nebo kdy si to přeje rodič procesu, který potřebuje synchronizovat své potomky. Dále proces přechází do stavu „odložený připravený“ a to ve chvíli, kdy nastane očekávaná událost. Do stavu „připravený“ se proces dostane, až se ve frontě „čekajících“ procesů uvolní místo.



Obr. 3 Stavový model procesů

2.2 Přejídy mezi stavy procesu

- Proces se dostane do stavu „běžící“ poté, co je vybrán plánovačem na základě určité strategie. Plánovač je část operačního systému, konkrétně část jádra, která vybírá procesy.
- Proces se dostane do stavu „čekající“ pokud čeká na nějakou externí událost. Důvodem zablokování procesu může být třeba, že proces chce tisknout, ale tiskárna není v té chvíli připravena. V momentě, kdy nastane očekávaná událost, tiskárna se stala dostupnou a připravenou, proces se dostává do stavu „připraven“.
- Ze stavu „běžící“ do stavu „připravený“ se proces dostane, pokud mu vyprší čas přidělený procesorem anebo pokud se objeví proces s vyšší prioritou.

2.3 Přepnutí procesu

Probíhá tímto způsobem. Operační systém nastaví časovač, který vyvolává přerušení v pravidelných časových intervalech. Toto přerušení může přijít i od vstupních a výstupních zařízení. Adresa obslužného podprogramu, který dané přerušení zpracuje, je na předem definovaném místě.

Příchod vyvolaného přerušení CPU způsobí, že čítač instrukcí (IP) se uloží do zásobníku. Načte se adresa obslužného podprogramu přerušení a CPU se přepne do režimu jádra (trvale běžící program). Samotný obslužný podprogram přerušení pak uloží obsah registrů do zásobníku a nastaví nový, takzvaný „prozatímní“ zásobník. Na řadu přijde úloha plánovače, který nastaví proces jako „připraven“, na základě priority nebo jiných kritérií vybere nový proces (viz. kap. 3.4.1.). Přepnutím kontextu se pak nastaví mapa paměti nového procesu, nastaví se zásobník a načte obsah registrů. Návratem z přerušení se načte do čítače instrukcí adresa zásobníku a přepne se do uživatelského registru.

2.4 Proces (process, task)

Proces je druh programu. K jeho provádění v systému (tzv. běh procesu) je zapotřebí určitého času procesoru, dále pak vnitřní paměť, vstupní a výstupní zařízení systému, soubory, atd. Proces musí být schopen pozastavit se. S tím souvisí pojem multitasking, který bude popsán v kapitole 6.2.1. K tomu musí být proces schopen komunikovat. Spouští se v operačním systému a je tvořen zpravidla neměnným kódem, konstantami a daty, které se měnit mohou (př. globální proměnné, stav procesoru, soubory, data na zásobníku, halda, ...) a jsou uloženy v adresním prostoru.

S procesem jsou sdruženy registry jako čítač instrukcí (IP, SP, univerzální registry CPU atd.) a další informace potřebné k běhu programu, což jsou stavové informace, které nutně potřebujeme v momentě, kdy se spouští pozastavený proces.

V multiprogramových systémech jsou stavové informace ukládány do tabulky procesů.

2.4.1 Tabulka procesů

Tabulka procesů v sobě ukládá všechny potřebné i méně důležité informace o každém procesu. Tato tabulka je udržována operačním systémem a každý proces v ní má položku, takzvaný Process Control Block (PCB). V této položce jsou uloženy všechny informace, které jsou potřeba uchovat v případě, že proces je přepnut ze stavu „běžící“ do „připraven“ nebo „blokován“. Tyto informace jsou nutné k tomu, aby bylo možné proces znova rozběhnout. Process Block Control slouží také k přepínání procesů. Nejprve se nastaví ukazatel na proces (číslo procesu, pod kterým je spuštěn). Nastaví se stavový vektor procesu (tzn. stav procesu uloženého v registru). Čítač instrukcí – obsah další načtené instrukce. Stav CPU – PSW (jestli jede v uživatelském režimu), ukazatel na zásobník,

popis paměti (ukazatel na začátek a velikost, popis, kde začínají data a kde instrukce). Správa vstupu a výstupu (aktuální nastavení adresáře, pozice v souboru). V tabulce se nachází priorita procesu, ukládací oblast pro jeho kontext. Součástí tabulky procesů navíc bývají data, které operační systém přímo nevyužívá, ale slouží pro případnou statistiku a ladicí účely – jméno procesu, informace o způsobu, jakým byl vytvořen a době jeho vytvoření.

Když počítač očekává vstupně výstupní operaci, vyvolá se přerušení a proces se přepne do stavu „čekající“, jakmile nastane očekávaná operace, proces je převeden do stavu „připraven“ a je zařazen do fronty procesů čekajících na CPU. Ze stavu „běžící“ pak proces může být i „ukončen“.

Položka PCB většinou obsahuje pole, které souvisí se správou procesů, správou paměti a správou souborů, konkrétně se ale liší podle systému. Hlavní paměť sestává z pole adresovatelných slov nebo i bytů, které jsou samostatné. Je energeticky závislá, to znamená, že po sebemenším výpadku napájení ztratíme data v ní uložená. Takzvanou virtuální paměť řídí operační systém, který také přiděluje a uvolňuje paměť podle potřeby a rozhoduje, kterému procesu přidělí na žádost prostor paměti po jejím uvolnění. PCB obsahuje:

- Informace pro správu procesu
 - CPU registry (včetně programového čítače, stavového slova programu PSW, příznaky, stack pointer)
 - stav procesu
 - ukazatel na další proces ve frontě
 - ukazatel na seznam procesů-potomků
 - ukazatel na rodičovský proces
 - skupina procesu
 - skutečné číslo uživatele (real user id)
 - skutečné číslo skupiny (real group id)
 - čas do příštího alarmu
 - ukazatel do fronty zpráv

- příznaky nevyřízených signálů
 - číslo procesu s různé příznaky
- Informace pro správu procesoru
 - priorita procesu
 - časové kvantum
 - využití předešlých časových kvant
- Informace pro správu paměti
 - ukazatel na segment programu
 - ukazatele na další segmenty paměti
 - tabulky stránek
 - informace pro ochranu paměti
 - efektivní číslo uživatele (effective user id)
 - efektivní číslo skupiny (effective group id)
 - kód ukončení úlohy
 - stav signálu
- Informace pro správu souboru
 - aktuální adresář
 - kořenový adresář
 - maska práv souborů
 - popisovače souborů
 - efektivní číslo uživatele (effective user id)
 - efektivní číslo skupiny (effective group id)

- Účtovací informace
 - čas startu procesu
 - spotřebovaný čas procesoru
 - čas procesoru spotřebovaný procesy-potomky
 - počet přečtených a zapsaných diskových bloků
 - počet vytisknutých stránek na tiskárně

S procesy jsou v systému spojeny určité služby:

- vytvoření nového procesu (fork v UNIXu, CreateProcess ve Win32)
- skončení procesu (exit v UNIXu, ExitProcess ve Win32)
- čekání na dokončení potomka (wait v UNIXu, WaitForSingleObject ve Win32)
- alokace a uvolnění paměti procesu
- komunikace mezi procesy (interprocess communication, IPC)
- ve víceuživatelských systémech mají všichni uživatelé identifikátor uživatele (UID). Potom každý proces běží s UID uživatele, který ho vytvořil nebo spustil

Členy skupin mohou být uživatelé, pak proces běží i s číslem skupiny (GID – group identification). Pokud běží proces ve Windows, je tam charakterizován jako spustitelný program (kód, data), virtuální paměť nebo také systémové zdroje (semaforey a komunikační porty), tam má i číslo procesu (PID – process identification). Podproces má k dispozici maximálně tolik paměti, kolik je umožněno procesu. Proces má schopnost vytvářet vlastní podprocesy (potomky). Například pokud vyvoláme jedno okno ve druhém. Ve většině operačních systémech nemůže podproces vydržet déle než jeho rodičovský proces. V Linuxu po ukončení rodičovského procesu přebírá podproces hlavní proces init (proces init je rodičem všech procesů).

2.5 Nulový proces

Je vytvořen operačním systémem při jeho inicializaci. Je to naprosto normální proces s velmi nízkou prioritou. Nižší, než je priorita všech uživatelských procesů. Tato nízká priorita zajišťuje, že mu budou přiděleny prostředky procesoru pouze v případě, kdy všechny ostatní procesy na něco čekají, třeba na stisk klávesy, a žádný z nich běžet

nemůže. Tedy takové procesy, které byly ze stavu „běžící“ převedeny do stavu „čekající“. Obsah nulového procesu bývá velmi triviální. Často to bývá jen pozastavení procesoru ve věčném cyklu. Tento proces má i své omezení, a to že nesmí nikdy přejít do jiného stavu, než „připraven“ nebo „běžící“.

2.6 Proces jako abstrakce

Důležité je představit si, že všechny spuštěné programy na počítači, někdy i včetně samotného operačního systému, jsou organizovaně uspořádány jako množina sekvenčních procesů. Tím můžeme proces považovat za běžící program včetně obsahu čítače instrukcí, registrů a proměnných, které běží ve vlastní paměti.

Z hlediska koncepce má každý proces vlastní virtuální CPU, ve skutečnosti ale přepínání mezi procesy zajišťuje reálný procesor. Tento způsob nazýváme multiprogramování, ale pro pochopení problematiky se využívá představy množiny procesů, které běží (pseudo)paralelně. Z toho plyne, že **skutečný paralelismus** je, když každý proces má svůj procesor. Naproti tomu **pseudoparalelismus** znamená, že procesor se virtuálně dělí. Při pseudoparalelním běhu máme v jednom okamžiku aktivní jen jeden proces, ostatní „neběžící“ procesy se řadí do fronty. Po určité době je aktivního běh procesu pozastaven operačním systémem, který místo něj spustí běh jiného procesu. Důvodem může být, že vypršel čas vyhrazený pro proces nebo jej přikázal ukončit sám uživatel. Potlačit ho mohla i vyšší priorita jiného procesu, v tomto případě se ukončení nazývá násilné. Během určité délky doby, kdy je proces aktivní, tedy jsou mu přiděleny prostředky procesoru, provede proces část své činnosti. Některé procesy ji mohou i dokončit. Záleží na náročnosti procesu. Rychlost běhu procesu není konstantní a většinou ani není zopakovatelná.

Příkladem může být učitel, který opravuje písemné práce. Má zadání s vypracovaným řešením (kód programu) a také písemky od studentů (vstupní data). Když přijde student na konzultaci s příkladem, založí si učitel písemné práce, které už opravil (uloží aktuální stav procesu) a vysvětlí žákovi nepochopenou látku. Poté se vrátí k opravování („pozastavený“ proces se stává „běžící“).

2.7 Dělení OS podle procesů

Operační systémy dělíme podle počtu paralelně pracujících úloh (sdílených procesů), podle počtu procesorů a podle počtu uživatelů. Jejich kombinacemi potom získáme konkrétní operační systémy. Například jednouživatelský, víceúlohový operační systém je Windows, který pracuje na více procesorech. Jeden uživatel se přepíná mezi více okny (úlohami). Může editovat soubor a mezitím si nechat stahovat data z internetu. Serverová verze Windows je víceuživatelská. Operační systém Linux pracuje na více procesech, je víceuživatelský a víceúlohový. Systém MS DOS podporuje skutečný paralelismus, zpracovává více procesů.

3 PLÁNOVAČE A PLÁNOVÁNÍ PROCESŮ

3.1 Plánovače (schedule)

Úkolem plánovače, který je programem, je určovat procesy, které přejdou ze stavu „připravený“ na „běžící“. Rozhodnutí vybrat proces mohou pouze ve čtyřech případech, a to když proces přechází ze stavů „běžící“ do stavu „čekající“, „běžící“ do „připravený“, „čekající“ do „připravený“, anebo když proces končí. Známe tři druhy plánovačů a podle nich dělíme plánování procesů (process scheduling) z časového hlediska na:

- **Krátkodobé** (short-term), nazýváno také CPU scheduling (plánování procesoru) – vybírají procesy, kterým bude přidělen čas procesoru, na základě priority, podle pořadí, logiky, délky,... U všech víceúlohových systémů
- **Střednědobé** (medium-term)– vybírají procesy, které budou zařazeny mezi odsunuté a pak „připravený“, a to v případě nedostatku vnitřní paměti. Jsou taktické (swap out, roll out)
- **Dlouhodobé** (long-term), někdy nazývané job scheduling (plánování prací, úloh) – vybírají procesy, které budou považovány za „připravené“, jsou strategické. Zejména při dávkovém zpracování. Účelem je aby byl počítač co nejvíce vytížen a nevznikaly zbytečné prostoje procesoru.

3.2 Plánování procesů

Existují dva způsoby plánování. **Preemptivní plánování** znamená plánování bez předbíhání, kdy proces končí anebo přechází ze stavu „běžící“ do stavu „čekající“. Procesu jsou odebírány prostředky procesoru samotným procesorem a to uplynutí určitého časového úseku. Na tomto principu pracuje operační systém Unix. **Nepreemptivní (kooperativní) plánování** je plánování s předbíháním. Jsou to případy, kdy proces „dobrovolně“ přechází ze stavu „běžící“ do stavu „připravený“ (aniž by musel být „čekající“), anebo když přechází ze stavu „čekající“ do stavu „připravený“ (tato varianta je u víceúlohových systémech označována za systém reálného času). Pokud si ale proces nehlídá časovač a nevzdá se procesoru, nemůže být přerušen a tím zablokuje celý operační systém. Na tomto principu pracuje starší verze Windows, např. verze 3.11.

Při plánování má svou důležitou roli i supervizor. Jeho úkolem je vědět o aktuálním stavu všech systémových prostředků, kterými jsou procesor, operační paměť, vnější paměti a vstupní a výstupní zařízení, a řídit jejich činnost podle toho, zda jsou obsazena, volná, připojena, odpojena nebo nehlásí poruchu.

Když se plánují procesy musíme uchovávat informace, které potřebujeme pro spuštění daného procesu na procesoru, také informace, které jsou potřeba ke správě paměti a informace potřebné pro správu vstupu a výstupu.

Tyto informace se vkládají do front pro plánování procesů. Fronta úloh je množina všech procesů v systému. Těchto front může být více:

- fronta připravených procesů, což je množina procesů sídlících v hlavní paměti a připravených k běhu,
- fronta procesů čekajících I/O zařízení,
- fronta odložených procesů. To je množina procesů čekajících na přidělení místa v hlavní paměti,
- fronta na semafor realizovaná množinou procesů čekajících na synchronizační událost.

Při plánování procesů můžeme narazit na problematiku odkládání procesů, tzv. swapping. Každý proces, se musí dostat do operační paměti. Pokud hrozí překročení kapacity, použije se virtuální paměť, která rozšíří fyzický adresní prostor na logický a proces je částečně přesunut.

LAP – logický adresní prostor, ve kterém běží procesy. Vidí ho uživatel. Programy a data jsou z něj do FAP přenášeny podle potřeby. Má omezenou velikost množinou adres a ta je určena počtem bitů vnitřní adresní sběrnice. Většinou bývá jednodimenzionální – adresa paměťového místa je tvořena jednou složkou.

FAP – fyzický adresní prostor. Úzce spolupracuje s logickým adresním prostorem. Je vymezen počtem bitů vnější adresní sběrnice a určuje ho velikost operační paměti.

Zobrazení LAP do FAP se řeší hardwarově (Dynamic Address Translation – DAT, Memory Management Unit – MMU).

3.3 Plánování procesoru

Pokud má systém pouze jeden procesor, není možné, aby obsloužil několik procesů naráz. Může poskytnout své prostředky pouze jednomu z nich. Současný běh programů pak musí simulovat operační systém samotný. A to plánováním přidělováním jednoho procesoru procesům. Plánování procesoru se využívá u víceúlohových systémů a to v těchto situacích:

- při dokončení procesu
- při přechodu procesů ze stavu čekající nebo běžící na připravený nebo při přechodu ze stavu běžící do blokování (čeká na vstupní nebo výstupní operaci, na semafor nebo na jinou událost)

Podle situace pak rozlišujeme druh plánování (preemptivní nebo koordinované).

3.4 Strategie plánování

Jedním z bodů při plánování je posouzení spravedlnosti. Každý proces dostane spravedlivý díl času procesu. Dalším bodem je určitě i efektivita. Procesor nepřiděluje zbytečně čas procesům, které ho nepotřebují tolik. Ke strategii plánování procesu patří i čas odezvy, což je doba, kdy zareaguje myš. Cílem je tuto dobu minimalizovat. Dalším strategickým bodem je doba obrátky, kdy se plánovač snaží minimalizovat dobu zpracování procesu a dobu přepínání mezi procesy. Plánovače posuzují i průchodnost, to znamená, že se maximalizuje množství úloh, které projdou za jednotku času.

3.4.1 Druhy strategií

FCFS (first come, first serve) - procesy jsou zpracovávány v pořadí, ve kterém vznikly a zařadily se do fronty čekající na procesor, důležitá je doba vzniku procesu. S touto strategií se setkáme při preemptivním i kooperativním plánování procesoru. Může nastat takzvaný „konvojový efekt“ a to v případě, kdy se ve frontě naskládá dlouhý proces (proces, který potřebuje ke zpracování více času procesoru) a za ním procesy krátké, které musí dlouho čekat na procesor.

Příklad:

Proces P1: doba trvání 28s

Proces P2: doba trvání 4s

Proces P3: doba trvání 3s

Pokud vzniknou procesy v pořadí P1, P2, P3, pak první proces nebude čekat vůbec, druhý 28s a třetí 32s. Z toho vyplývá průměrná doba čekání na procesor $(0+28+32)/3s$, tedy 20s. Pokud ale procesy vzniknou v pořadí P3, P2, P1, pak první proces nebude čekat vůbec, druhý 3s a třetí 7s. Průměrná doba čekání v tomto případě bude pouze 3,3s.

SJF (shortest job first) – nejdříve se v procesoru zpracuje nejkratší úloha. Tento algoritmus je optimální. Dané množině procesů zajišťuje minimální průměrnou dobu čekání. Známe jeho dvě varianty a to preemptivní s předbíráním a nepreemptivní bez předbírání. V prvním případě, v momentě, kdy se ve frontě připravených procesů objeví proces s délkou dávky CPU kratší než má proces, který je právě na procesoru, běžící proces se pozastaví a je předběhnut vybraným procesem z fronty. U nepreemptivní varianty nemůže být proces na procesoru předběhnut, vzdává se procesoru až když je dokončen.

Prioritní – každý proces má přidělenou nějakou prioritu, která při této strategii rozhoduje o tom, kterému procesu bude přidělen procesor. Čím větší číslo priority tím dříve je procesu přiděleno CPU. Tato strategie podporuje jak preemptivní plánování s předbíráním, tak nepreemptivní bez předbírání. Hrozí nebezpečí, že procesy s nejnižší prioritou nebudou nikdy zpracovány, zestárnou. Zamezit tomuto riziku se může tak, že se priorita bude postupem času zvyšovat, proces bude „zrát“. Platí, že čím menší je prioritní číslo, tím vyšší má proces prioritu.

Další strategie je založena na proměnné délce časového kvanta a většinou se využívá v kombinaci s jinou (FCFS). Funguje na principu, že procesy, které jsou delší dostávají delší časové intervaly.

SXFS (shortest execution - first served) – jako první je zpracován proces, jehož provedení zabere nejméně času

LCFS (least completed – first served) – jako první se obslouží proces, který zatím běžel nejkratší dobu

EDFS (earliest - due-time first served) – jako první se obsluhuje proces, kterému do okamžiku ukončení zbývá nejmenší časový úsek

RR (round-robin) – procesy jsou obsluhovány v cyklech o stejných časových intervalech, zpravidla při preemptivním plánování. Tzv. časová kvanta se pohybují v intervalu od desítek až po stovky ms. Po uplynutí této doby je proces předběhnut nejstarším procesem ve frontě. Označíme-li n počet procesů ve frontě připravených a q časové kvantum, pak můžeme počítat, že proces získá od procesoru dobu $1/n$. Najednou nejvýše po q časových kvantech. Zároveň je zaručeno, že žádný proces nečeká na proces déle než $(n-1) \cdot q$ časových jednotek. Na velikosti časového kvanta závisí i výkonnostní hodnocení. Pokud je q velké, pak je plánování velmi podobné strategii FCFS. V opačném případě, tedy je-li q malé, algoritmus může být neefektivní. Proto je potřeba, aby bylo q dostatečně velké s ohledem na režii přepínání kontextu.

HSFS (highest static priority first served) – jako první je obsloužen proces, který má nejvyšší statickou prioritu

3.5 Změna kontextu – context switch

Pokud na počítači běží program, je potřeba ho dočasně přerušit, ale takovým způsobem, aby mohl být po nějakém čase znova obnoven a pracoval od místa kde skončil, je nutno použít jistou „zálohu“. Kontext je soubor dat, ve kterém je archivován a definován stav procesu a registrů. Pokud máme operační systém s preemptivním plánováním procesoru, může nastat situace, kdy je proces přerušen mezi libovolnými dvěma strojovými instrukcemi v programu a řízení se pak předá jinému procesu. V takovém případě je ale potřeba pamatovat i na nutnou změnu dalších registrů (ukazatel na tabulku stránek, atd.). Při přepínání dalšího procesu se uloží do kontextu potřebná data k jeho opětovnému obnovení. Kontext najdeme buď v zásobníku procesu nebo v oblasti dat v adresním prostoru procesu.

Při vytváření procesu je nutné uměle vytvořit zásobník, pro případ, kdyby došlo k přerušení procesu před jeho první instrukcí. To se provede tak, že na zásobní nového procesu se uloží adresa jeho první instrukce a adresa tohoto zásobníku se zapíše do kontextu.

4 SPOLUPRÁCE MEZI PROCESY

Máme dva způsoby, kterými mohou procesy mezi sebou spolupracovat. A to buď pomocí sdílené paměti, kterou je snadnější naprogramovat anebo zasíláním zpráv, což je flexibilnější spolupráce. Takto spolu mohou komunikovat i ty procesy, které jsou ve stavu „běžící“.

4.1 Zasílání zpráv

Probíhá tak, že se předává ukazatel na zprávu ve sdílené paměti, umístěním zprávy do vyrovnávací paměti nebo do fronty zpráv. Buď ve sdílené paměti nebo v paměti jádra systému. Ten pak zajišťuje kopírování zpráv do paměťového prostoru příjemce zprávy. Tato zpráva může být předávána i prostřednictvím sítě. V různých systémech se liší komunikační spojení pro zasílání zpráv ve způsobu vytváření spojení, v počtu procesů využívajících spojení, v počtu spojení mezi dvěma procesy, kapacitou spojení. Kapacita spojení označuje počet možných nezpracovaných zpráv, které může spojení obsahovat. Kapacita může být dvojitá, a to neomezená, kdy odesílatel nemusí nikdy čekat anebo nulová. V tom případě musí odesílatel čekat na příjemce (tzv. rendezvous) anebo se zpráva, na kterou žádný příjemce nečeká, ztratí. Aby k takovým případům nedocházelo, zabezpečují se synchronizačními prostředky. Tato kapacita je omezená. Odesílatel musí čekat, pokud je linka zaplněná. Dále se komunikační spojení pro zasílání zpráv liší i ve velikosti zprávy, která může a nemusí být proměnná, v jednosměrnosti nebo obousměrnosti spojení, v symetrické nebo nesymetrické komunikaci, v asynchronní nebo synchronní komunikaci, v zasílání kopie nebo reference, v automatickém nebo ručním použití vyrovnávací paměti nebo přímé a nepřímé komunikaci. Nepřímá je vedena přes poštovní schránku.

Při zasílání zpráv musíme zajistit ošetření chyb v následujících situacích:

- jeden z partnerských procesů skončil
- zpráva se ztratila
- zdvojení zprávy
- zkomolení zprávy

Programovými prostředky pro předávání zpráv jsou SEND a RECEIVE. SEND zašle zprávu a přitom se nemůže stát, že by se někdy zablokoval. Pokud na zprávu čeká příjemce

operací RECEIVE, je mu zpráva doručena a příjemce odblokován. RECEIVE zprávu přijímá. Pokud však žádná zpráva není dostupná, je přijímací proces zablokován.

Je nutné provést adresaci, tj. určit odesilatele a příjemce. Jako řešení se nabízí identifikace procesu na daném počítači. Jako další můžeme zavést schránky (mailboxy). Adresace se pak provádí pomocí identifikace schránky.

4.2 Sdílená paměť

Existuje paměť, kam mají přístup všechny procesy a komunikují zde spolu. Mohou ale nastat jisté problémy, například souběh. Tuto chybu mohou způsobit data, která jsou sdílena několika procesy (tzv. kritická sekce).

4.2.1 Souběh – race condition

Máme-li procesy, které se samostatně chovají korektně, přesto se může stát, že při přístupu dvou a více procesů ke sdíleným datům, může dojít k chybě. Případ souběhu nastává v situaci, kdy jsou data měněna jedním procesem v době, kdy s ním provádí operace druhý a předpokládá, že budou provedeny jako nedělitelný celek. Datům, která jsou sdílena několika procesy a hrozí riziko souběhu, říkáme kritická oblast. Nejmenší část programu, ve které se pracuje s daty v kritické oblasti, říkáme kritická sekce. Musí být provedena jako celek. Zejména u složitějších datových struktur jsou data v určitém stadiu dočasně nekonzistentní, a když v tom samém okamžiku dojde k přepnutí kontextu na proces, který s danými daty pracuje, může dojít k souběhu. Příkladem mohou být dva procesy. Každý z nich má za úkol vytvořit soubor, ovšem se stejným jménem. Oba zjistí, že daný soubor neexistuje. Jeden proces je však zablokován druhým, který daný soubor vytvoří, zapíše do něj nějaká data a skončí. Původně zablokovaný proces pak soubor vytvoří znova a tím přemaže data, která do něj uložil předchozí proces. Souběhu lze zabránit, pokud operační systém má tu vlastnost, že jednou vytvořený soubor již nepovolí znova vytvořit a nahlásí chybu.

4.2.2 Kritická sekce

Kritická sekce pracuje s daty v kritické oblasti. Pracuje jako celek a v celku, to znamená, že není možné ji přerušit. Existují určité podmínky pro kritickou sekci, které se musejí dodržet při přístupu procesů ke kritické sekci. V jednom okamžiku tady smí být pouze

jediný proces. Tato podmínka se nazývá výhradní přístup. Dále si procesy, které o vstup usilují, sami rozhodují, kdy do kritické sekce vstoupí. Ale rozhodnutí o vstupu určitého procesu do kritické sekce nemůže být odkládáno do nekonečna. Jinak by to vedlo ke striktní alternaci. Ta nastane, když se dva procesy při průchodu kritickou sekcí pravidelně střídají. Třetí podmínkou je omezené čekání, které nastává v případě, že jeden proces usiluje o vstup do kritické sekce. Procesy, které se v této sekci střídají mu nesmí bránit ke vstupu. Řeší se to omezením počtu vstupů.

Pokud vznikne situace, kdy se v kritické sekci nachází proces a o přístup do ní usiluje další anebo o vstup usiluje více procesů naráz, je nutné tyto procesy dočasně pozdržet. Pozdržení může být realizované smyčkou, které se říká aktivní čekání (busy waiting), spotřebovává však zbytečně čas procesoru. Proto je lepší čekající proces zablokovat a zaktivovat až se kritická sekce uvolní.

S touto problematikou souvisí zajištění výhradního přístupu, protože žádné dva procesy se nesmějí nacházet zároveň v kritické sekci. Dalšími podmínkami pro efektivnost a korektnost jsou takové, že nemohou být udělány žádné předpoklady o rychlostech nebo počtu CPU, žádný proces, který běží mimo svou kritickou sekci nesmí blokovat jiný proces a žádný proces se nesmí zdržovat ve své kritické sekci nekonečně dlouho. Nabízí se několik možností jak vyřešit výhradní přístup procesu do kritické sekce. A to:

- Zákazem přerušení. Tato metoda není doporučována. Pokud zakážeme přerušení, znemožníme přepnutí kontextu. Pak se ale může stát, že proces, který přerušení zakázal zhavaroval nebo se dostal do nekonečného cyklu a tím se zacyklí celý systém. Většinou má však zákaz přerušení na starost jádro operačního systému.
- Nebo se výhradní přístup zajišťuje instrukcí typu TSL (Test and Set Lock) tak, že TSL nepřiděluje prostředek žádnému jinému procesu, dokud není kritická sekce odemčena. Máme proměnnou LOCK (zámek) typu boolean, která se nastaví na hodnotu true – zámek je uzamčen a vrací se hodnota této proměnné. Po výstupu procesu z kritické sekce se hodnota proměnné LOCK nastaví na false – zámek je odemčen a tím je umožněn přístup dalšímu procesu. Instrukce TSL sama o sobě ještě nesplňuje podmínku omezeného čekání a neodstraňuje aktivní čekání procesu, avšak je základem pro algoritmy, které tento nedostatek nemají. Počítače, které instrukce TSL nemají používají zákaz přerušení anebo

používají instrukci, která prohodí obsah registru s obsahem proměnné v paměti.

- Výhradní přístup se zajišťuje i semaforey. Zde je dvoustavová proměnná typu boolean nahrazena čítačem (proměnná typu integer). Provádějí se zde operace DOWN a UP. V některých publikacích najdeme také označení P (testovací) a V (zvětšovací). Z dánského Proberen a Verhogen..

U nezáporného čítače - operace DOWN otestuje hodnotu a pokud je tato větší jak nula, sníží se hodnota semaforu o jeden a operace skončí. Jestliže je hodnota rovna nule, operace DOWN se zablokuje daný proces nesmí vstoupit a je zařazen do fronty procesů čekajících na semaforu. Operace UP je volána po ukončení zdroje procesem a zjišťuje zda je fronta na semaforu prázdná či není. Pokud není, vybere a odblokuje například nejdéle čekající proces. Pokud je fronta prázdná, zvětší číslo o jedna a to je signálem, že proces může vstoupit. Existují i další podoperace jako třeba c – počet čekajících procesů, mutex – ten zase určuje jestli proces může čekat na semafor.

U libovolného čítače se může u operace DOWN vyskytnout i záporná hodnota, která značí počet blokováných procesů. Jinak pracuje stejně jako předchozí čítač.

Před vstupem procesu do kritické sekce se na semaforu vyvolá operace DOWN, po výstupu UP. Semaforey také nesplňují podmínku omezeného čekání a mají také nedostatek v aktivním čekání procesu, které neodstraňují.

Synchronizační prostředky odstraňující aktivní čekání – když proces nemůže vstoupit do kritické sekce (je obsazena jiným procesem), je mu odstraněno aktivní čekání odebráním procesoru a zablokováním běhu procesu. Abychom věděli, který proces po uvolnění kritické sekce odblokovat, přiřadíme každému semaforu nebo instrukci TSL frontu čekajících procesů.

5 UVÁZNUTÍ – DEADLOCK

Uvážnutí nastává v případě, že dva nebo více prostředků čekají na událost, ke které by mělo dojít, až by další proces pokračoval. Stává se tak v případech výlučného přístupu, postupného přidělování prostředků, když přidělené prostředky nelze procesu násilně odebrat nebo když jsou proces ochotny cyklicky čekat. Vyřešení situace by mohlo nastat pouze „couváním“. Příkladem může být, když ve víceúlohovém systému je jednomu procesu přidělena tiskárna a druhému magnetická páska. Potom první proces požádá o přidělení pásky a druhý tiskárny a ani jeden nemůže dál pokračovat, nastane uvážnutí.

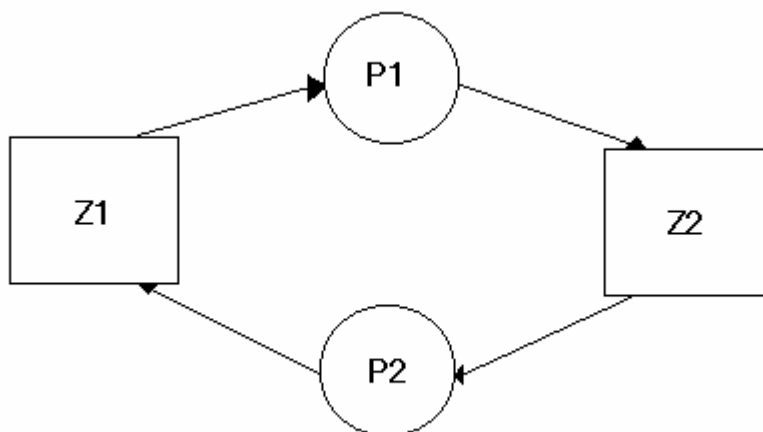
5.1 Podmínky uvážnutí

Aby došlo k uvážnutí, musí být splněny všechny čtyři následující podmínky. Tou první je výlučný přístup (mutual exclusion). Jednomu procesu jsou přidělovány nesdílitelné prostředky pro výhradní použití. Druhou podmínkou je postupné přidělování prostředků (hold and wait). Jednomu procesu nepřidělíme všechny prostředky potřebné k jeho dokončení najednou, ale postupně. S tím, že když není zrovna některý prostředek k dispozici, proces čeká. Třetí podmínkou je přidělování prostředků bez preempce (bez předbírání). To znamená, že jsou-li už procesu prostředky procesoru přiděleny, nelze mu je násilím odebrat. Čtvrtou podmínkou je cyklické čekání.

5.2 Řešení otázky uvážnutí

Řešením je ignorování uvážnutí. Nutný je zásah uživatele, který některý z procesů násilně ukončí (u operačního systému UNIX). Dalším východiskem je prevence, předcházení uvážnutí a to tak, že se zabrání alespoň z jedné podmínek uvážnutí. Uvážnutí se můžeme i vyhýbat (bankéřův algoritmus – vždy si nechá dost prostředků, aby uspokojil alespoň jeden čekající proces, musí znát maximální požadavky procesů, viz praktická část). Uvážnutí můžeme také detekovat a zotavit se z něj.

Jako jedno z možných východisek uvážnutí se nabízí i spooling, kdy má více procesů přístup k jednomu zařízení.



Obr. 4 Ukázka uvážnutí v grafu

P – proces, Z – zdroj. Z_1 je blokován v procesu 1 a P_1 čeká na Z_2 . Pokud zdroj Z_1 je blokován procesem P_1 a zdroj Z_2 blokován procesem P_2 , zároveň proces P_1 čeká na zdroj Z_2 a proces P_2 čeká na zdroj Z_1 , dojde u k uvážnutí. Pokud by proces P_2 nečekal na zdroj Z_1 , měl by splněny podmínky k ukončení. Šipka od zdroje k procesu znamená, že zdroj je procesu přidělen. Šipka od procesu ke zdroji značí, že proces požaduje zdroj.

5.3 Předcházení uvážnutí

Tato strategie, předcházení uvážnutí, má za úkol zamezit alespoň jedné z podmínek uvážnutí.

- **Výlučný přístup**

Read only soubory nemohou způsobit uvážnutí, protože jsou bez omezení sdílené. Využívá virtualizace prostředků. Například u tiskáren a snímačů děrných štítků nebo pásek. Procesy nepoužívají zařízení přímo, nýbrž zapisují výstupy do diskového souboru, který operační systém vytiskne později. U vstupních zařízení systém načte požadovaná data do souborů, ke kterým mají přístup jednotlivé procesy a čtou z nich. Tato metoda je již zmiňovaný spooling. Bohužel tato metoda se nedá aplikovat na všechny existující prostředky.

- **Postupné přidělování prostředků**

Pokud přidělujeme prostředky jednorázově, znamená to, že si každý proces vyžádá hned na začátku veškeré prostředky, které pro sebe potřebuje, s tím, že si

v průběhu již žádné další nevyžádá. V případě, kdy jeden z procesů potřebných prostředků není dostupný, není mu přidělen žádný a proces musí čekat až se uvolní. Alternativní strategií je přidělovat prostředky tomu procesu, který žádné nemá. Proces pak může během doby svého běhu několikrát vrátit všechny prostředky a pak žádat o další. Má to však i své nevýhody. Například nejsou plně využity všechny prostředky a prakticky jsou zbytečně blokovány u procesu, který je třeba v danou chvíli zatím ani nepotřebuje a nepracuje s nimi. Dále hrozí nebezpečí stárnutí procesů (starvation). Proces může čekat neomezenou dobu, a to v případě, kdy nemusí být uspokojeny nároky procesu na potřebné a často používané prostředky.

- **Přidělování prostředků bez preempce**

Tuto strategii můžeme používat pouze u prostředků jakými jsou procesor a operační paměť, protože jen ony dokážou uchovat a obnovit stav prostředků. Pokud je toto uchování a obnovení snadné, může operační systém tyto prostředky procesoru odebrat a přidělit je jinému procesu, který je zrovna potřebuje.

- **Cyklické čekání**

Každý typ prostředku má své číslo. Pokud jsou již procesu přiděleny nějaké prostředky, může takový proces žádat o prostředky, které mají číslo větší než největší z čísel procesem už držených prostředků. Pokud mají prostředky stejné číslo, musí o ně proces žádat najednou. Tento algoritmus zabraňuje cyklickému čekání.

Shrnutí: Některé z těchto strategií pro předcházení uváznutí používáme pouze pro sdílitelné prostředky a prostředky, u kterých můžeme obnovit a uschovat jejich stav. Ostatní algoritmy mohou způsobit, že některé prostředky mají zbytečné prostoje a nejsou využívány a dále i stárnutí procesů. Toto se jeví jako nevýhoda a proto nejsou tyto strategie používány v mnoha operačních systémech.

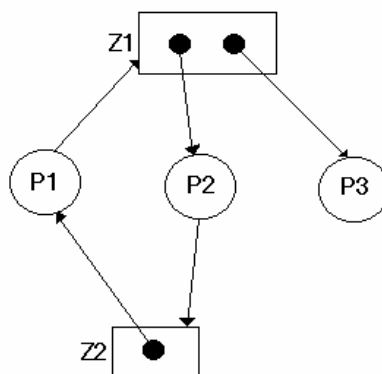
5.4 Vyhýbání se uváznutí

Předcházení uváznutí nemusí pokrýt veškeré prostředky používané v daném operačním systému a může být i omezující. Nejsme-li schopni zamezit prvním třem podmínkám vzniku uváznutí a není možné předejít nebo zabránit cyklickému čekání, používáme právě

algoritmus vyhýbání se uváznutí. Když si proces zažádá o přidělení prostředku, operační systém v prvním kroku zkontroluje, jestli existuje alespoň jeden proces, který by mohl být po přidělení tohoto prostředku dokončen. Po jeho dokončení musí mít systém zajištěno dokončení dalšího procesu, až po ukončení všech procesů. Pokud není tato podmínka být splněna, systém prostředek nepřidělí a žádost procesu zamítne. Nejznámějším algoritmem je bankéřův algoritmus, který bude podrobněji popsán v praktické části této práce.

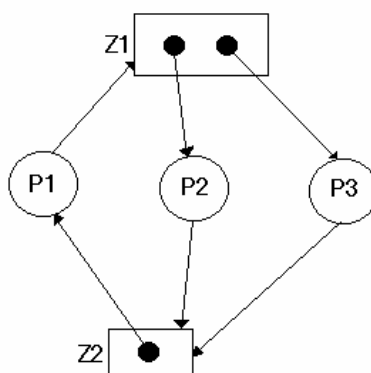
5.5 Detekce uváznutí a zotavení

Operační cyklus musí udržovat graf přidělení prostředků. K uváznutí dojde, pokud je v grafu cyklus.



Obr. 5 Zdánlivý cyklus, nejedná se o uváznutí

Proces P3 po ukončení uvolní prostředek zdroje Z1, který může být poskytnut procesu P1 a k uváznutí nedojde.



Obr. 6 Uváznutí

6 VLÁKNA (SLEDY, THREADS)

To, že mají procesy celý víceúlohový systém pro sebe je pouhou iluzí. Prakticky je systém od sebe odizolovává. Někdy je ale potřeba, aby spolu procesy komunikovaly. Proto používají prostředky operačního systému. Klasické procesy mají totiž oddělené adresní prostory. Vlákno je elementární proces, někdy nazýván odlehčený proces (lightweight-process). Některé systémy umožňují, aby se jeden „těžký“ proces skládal z více vláken řízení – threadů. Thready jednoho procesu sdílejí adresní prostor paměti a komunikují spolu pomocí sdílené paměti. Pokud systém nepodporuje multithreading, znamená to, že každý proces je tvořen právě jedním threadem. Výhodou threadu je nižší režie při přepínání mezi thready a jednodušší spolupráce mezi thready, než mezi dvěma procesy. Každý thread má potom samostatný zásobník a udržuje se pro něj stav procesoru, a to včetně programového čítače.

6.1 Rozdělení vláken

Na uživatelské úrovni ULT (User Level Threads)– přepojování mezi vlákny neprovádí jádro, závisí na aplikaci, což může být někdy výhodou, protože je to rychlejší. Operační systém nemanipuluje s vlákny, pouze s celými procesy. Když vznesе požadavek na operační systém vlákno, operační systém zastaví celý proces. Vlákna jsou spravována na uživatelské úrovni (operační systém Solaris). Když pracuje v privilegovaném režimu, tak běží pouze jádro operačního systému.

Na úrovni jádra KLT (Kernel Level Threads) – informace o vláknech určuje jádro, přepínání také provádí jádro. Nevýhodou je, že procesor ne vždy ví, co je pro aplikaci lepší a pro nás to znamená ztrátu času.

Když pak aplikace vyvolá službu, parametry se uloží na určené místo (registry, zásobník), vyvolá se speciální instrukce, která pak vyvolá obslužnou proceduru v jádře a zároveň se přepne do privilegovaného režimu. Operační systém zjistí, která služba je vyvolána, převezme parametry, provede danou službu a vrátí se zpátky do aplikace. Zároveň přepne procesor zpět do uživatelského režimu.

6.2 Multitasking a multithreading

6.2.1 Multitasking

Multitasking znamená, že se v jednom okamžiku provádí více procesů (běží více programů). Slouží k usnadnění programování. V zásadě tak pouze působí dojem, že běžící programy se provádí zároveň. Operační systém používá hardwarové hodiny a každému běžícímu procesu přiděluje časová kvanta. Pokud jsou tato dostatečně malá a zároveň počítač není příliš zatížen programy, které se snaží pracovat, vzniká pro uživatele dojem, že jsou programy zpracovávány současně. Multitasking není žádnou novinkou. Dříve byl samozřejmostí u velkých sálových počítačů. K sálovému počítači se připojily stovky terminálů a každý uživatel u terminálu musel mít zajištěn pocit výhradního přístupu. Sálové počítače byly navíc schopny spustit úlohu na pozadí a tím zaručil uživateli možnost pracovat na jiném úkolu, zatímco proces na pozadí byl tiše zpracováván. Než vznikl kooperativní multitasking, existovalo pouze vzájemné volání (MS DOS), poté přepínání programů (task switching). To se mohlo provádět neomezeně (bylo možné přepínat se mezi spuštěnými „všedními“ programy) anebo u operačních systémů s omezeným přepínáním se mohly přepínat pouze určité programy (starší verze Mac OS). Existoval jeden hlavní program a pak další speciální, které byly vytvořeny speciálně k přepínání. Příkladem může být server, který obsluhuje více klientů současně. Funguje na principu „odštěpování“ potomků procesu, které obsluhují klienta, zatímco proces rodič čeká na další žádost jiného klienta.

V dnešní době je multitasking na osobních počítačích samozřejmostí. Zavedení tohoto systému bylo, ale poměrně náročnou záležitostí. Jak z hlediska programového, kdy vývoj vícevláknových systémů nebyl na dostupných implementačních prostředích možný, tak z hlediska hardwarového, kdy prostředky nesplňovaly požadavky na takové operace. Tento problém začal být velmi patrný při nástupu Windows GUI (Grafic User Interface) prostředí. Možnost multitaskingu poskytovaly i určité verze systémů Windows, jako například 16bitové verze (kooperativní multitasking), ale pouze v omezené podobě. Ve skutečné podobě najdeme multitasking (preemptivní) u 32bitové verze Windows. U této verze se využívá i multithreading.

Výhody multitaskingu spočívají v tom, že umožňují kdykoliv přejít k jinému programu, potřebujeme-li ho na chvíli použít, aniž bychom museli přerušit rozdělanou práci. Dále

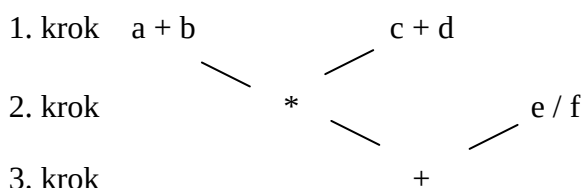
v multitaskovém prostředí mohou programy lépe spolupracovat, a to tak, že dva paralelně běžící programy mohou přímo navzájem ovlivňovat svou činnost a nemusí si předávat údaje prostřednictvím souborů. Další výhodou je lepší využití výpočetní kapacity systému. Například při práci s textovým editorem systém většinu času čeká na stisk klávesy, toto čekání může klidně využít práci na jiném výpočtu. Multitasking má ale samozřejmě i své nevýhody, jako například pokud je systém špatně navržen, může současný běh většího množství programů degradovat funkci toho programu, se kterým přímo komunikuje uživatel. Tím se zvýší doba odezvy na uživatelské příkazy a běh programu se výrazně zpomalí z důvodu režie pro přepnutí procesu.

Multitaskový operační systém je složitější a dražší než systém jednoúlohový. Často také vyžaduje větší konfiguraci – rychlejší operační paměť, větší kapacitu disků a výkonnější procesor. Pokud není multitaskový systém spojen s kvalitním systémem zabezpečení, zvyšuje se tak riziko ztráty dat. Pokud se zhroutí jednoúlohový systém, ztratíme pouze výsledky práce programu, který zhroucení způsobil. Když ale některý program zapříčiní spadnutí multitaskového operačního systému, jsou ztraceny výsledky práce všech procesů, které v té době běžely. Proto musí být multitaskový systém velmi pečlivě navržen.

6.2.2 Multithreading

Multithreading umožňuje, aby se jeden proces skládal z více vláken řízení – threadů, které sdílejí adresní prostor paměti a komunikují spolu v rámci sdílené paměti. Platí pro thready jednoho procesu. Rozdělení programu na vlákna pak působí, jakoby vlákna pracovala současně, přitom je to pouze zdánlivé. „Vícevláknové“ zpracování procesu znamená, že program může vykonávat více věcí zároveň. Příkladem je prohlížeč, když ukazuje a zároveň vyhledává z internetu. Využívá se hlavně u souborových serverů, grafických programů (Photoshop) nebo překreslování obrazovky.

Zpracování paralelismu: $(a + b) * (c + d) + (e / f)$ všechny operace v závorkách se mohou počítat paralelně, k ostatním potřebujeme výsledky závorek



Multithreading se také někdy vysvětluje jako schopnost programu zavést multitasking sám v sobě. Programu využívají multithreading hlavně proto, aby systém mohl zpracovávat dlouhodobější úlohy na pozadí a uživatel přitom nemusel přerušit práci s programem. Dříve si mohl pracovník odskočit na svačinu nebo na kávu, v dnešní době je ale z hlediska produktivity práce více žádoucí, abychom i při zatížení jednoho programu mohli pokračovat v práci s počítačem.

6.3 Režimy multitaskingu

6.3.1 Multitasking v DOSu

Kvůli nedostatku paměti nebyl schopen multitaskingu mikroprocesor Intel Pentium 8088, použitý v prvních počítačích. Při spouštění a ukončování programů jsou operačním systémem přesouvány různé bloky paměti. Jsou tak vytvářeny volné oblasti v paměti. U procesoru 8088 tak nebylo možné provedení, aby byl proces transparentní. Systém DOS pak byl navržen jako malý a oddělený od aplikací, takže neumožnil více než nahrání aplikace do paměti a přístup k souborovému systému. Toto omezení bylo vyřešeno za pomoci rezidentních programů (TSR). Rezidentním programem může být například obsluha tisku. Tyto programy byly závislé na přerušení od hardwarových hodin a prováděly programy na pozadí. Byly zaznamenány také pokusy o vybudování mechanismu přepínání úloh nebo multitasking nad DOSem, ovšem bez většího komerčního úspěchu.

6.3.2 Kooperativní

Kooperativní multitasking je historicky mladší než dokonalejší preemptivní multitasking. Jeho princip přímo vychází z mechanismu přepínání programů. Přepínání programů probíhá následovně: proces, který právě běží, musí pravidelně volat systémovou službu, kterou dá najevo, že může být přerušen. Dokud si uživatel nevyžádá přepnutí na jiný proces, nedělá tato služba nic. Pokud si ale uživatel vyžádá přepnutí procesů, zajistí tato služba při nejbližším vyvolání přepnutí kontextu (viz. kap. 3.5). Po jejím ukončení běží nový proces a dosud aktivní proces čeká, až bude znovu aktivován. U kooperativního multitaskingu operační systém nerozhoduje o tom, kdy bude proces přerušen, ale rozhoduje si to proces samotný. Výhoda kooperativního multitaskingu spočívá v tom, že je lépe využíván čas procesoru a nevznikají zbytečné časové prostoje. Bohužel kvůli zpomalení

procesu na popředí je tento multitasking nevhodný třeba pro správu sítě nebo jiné paralelní úlohy. Pokud se totiž objeví chyba v aktivním procesu, může dojít k zacyklení.

V roce 1985 byly uvedeny na trh Microsoft Windows 1.0. Obcházely omezení DOSu. Přestože Windows pracovaly v reálném režimu, ale byly schopny přesouvat paměťové bloky, což je základním předpokladem pro multitasking.

Multitasking jako takový dává mnohem větší smysl v grafickém prostředí při přepínání mezi okny než v řádkově orientovaném jednouuživatelském systému. Například pokud spustíme proces z příkazové řádky v systému UNIX na pozadí, musí být výstup tohoto programu směřován do souboru, aby se nemíchal s prací uživatele. Přepínání se mezi jednotlivými okny aplikací, které máme na jedné obrazovce je snadné a navíc je stejně jednoduché přenášet jednotlivá data mezi programy. Například obrázek vytvořený v grafickém programu můžeme importovat do textu textového editoru. Datové přenosy byly ve Windows prováděny nejprve pomocí schránky, později technologií DDE (Dynamic Data Exchange), v současnosti se využívá technologie OLE (Object Linking and Embedding).

Kooperativní, někdy se říká také nepreemptivní, multitasking podporovaly 16bitové verze Windows. Byl umožněn díky jeho architektuře založené na zprávách, kdy program nečinně čeká v paměti, až dostane zprávu. Tou může být nějaký vstup od uživatele, například stisk klávesy či pohyb myši. Po zpracování zprávy program předává řízení zpět do Windows. Přepínání se mezi úlohami se tedy neprovádí na základě přidělování časových kvant, ale až poté, co program zpracoval přijatou zprávu. Hrozí riziko zablokování celého systému a to v případě, kdy by aplikace zpracování zprávy trvalo programu příliš dlouho. U těchto systémů existovalo několik funkcí pro řešení omezení nepreemptivního multitaskingu. Známým je kurzor ve tvaru přesýpacích hodin. Toto ovšem není způsobem řešení, ale spíše vizuálním prostředkem pro uživatele, jako upozornění, že program je zaneprázdněn dlouhodobějším úkolem a systém nemůže po určitou dobu dále pracovat. Možným způsobem, částečným řešením se staly časovače. Díky nim aplikace dostávala zprávy v určitých periodických časových intervalech. Tyto časovače byly vhodné pro animace a různé časoměrné aplikace. Jako další způsob řešení se nabízela funkce PeekMessage. V normálním případě získává aplikace zprávy z fronty zpráv funkcí GetMessage. Pokud ale ta žádnou zprávu nenalezne, nevrátí se a čeká do doby, než se ve frontě nějaká zpráva objeví. Na rozdíl od ní funkce PeekMessage nečeká a vrací se, i když žádnou zprávu ve

frontě zpráv nenajde. Pro program je to pak výhoda, že může mezitím provádět jinou, třeba i déletrvající operaci a při tom občas zavolat funkci PeekMessage. Pokud se vrátí prázdná, může být úloha dále nerušeně zpracovávána.

6.3.3 Preemptivní

U kooperativního multitaskového operačního systému je základní nevýhodou kooperativnost, tedy že s operačním systémem musí spolupracovat každý proces. Pokud chceme tuto nevýhodu odstranit, potřebujeme takový operační systém, který dokáže přerušit každý proces a umožní důležitějším procesům, jako je například proces na popředí, získat prostředky procesoru přednostně, v okamžik, kdy dají na vědomí, že je potřebují. Takovému systému říkáme preemptivní – v češtině někdy překládán jako „předbíhací“. U preemptivního multitaskingu jsou procesům přidělovány krátké časové úseky. U prvních verzí systému Windows multitasking jako takový ještě nepředstavoval klasický preemptivní multitasking s přidělováním časových kvant, jaký známe z víceuživatelských operačních systémů. Tuto formu multitaskingu využívaly systémy Windows při spouštění programů systému DOS a pro to, aby mohly dynamické knihovny přijímat přerušování hardwarových hodin pro práci s multimédií. Preemptivní multitasking ukládá do kontextu pouze stav procesoru a s ostatními prvky systému povolí pracovat vždy jen jedinému procesu. Tento princip vyhrazení prostředků pro jeden jediný proces má však i jednu nevýhodu. Přináší možnost zablokování. Moderní operační systémy často zakazují užívání těchto prostředků všem procesům, kromě speciálně vyhrazených systémových, které pak ostatním procesům nabízí možnost nepřímo využít daných zařízení. Systém server – klient.

6.4 Řešení pomocí multithreadingu

První prostředí, které přineslo programátorům ve Windows multithreading, byl Presentation Manager. Při multithreadingu se aplikace může rozdělit na několik samostatných částí, takzvaných „prováděných vláken“. Ta běží současně. Ukázalo se, že vlákna jsou užitečná pro více funkcí. Z pohledu aplikace je vlákno představováno funkcí, která je schopna volat další funkce. Na začátku pracuje program s hlavním (primárním) vláknem, které v tradičním programu C reprezentuje funkce main. Při běhu programu mohou být vytvářena další vlákna, a to speciálním systémovým voláním. Přepínání řízení

mezi vlákny je pak prováděno operačním systémem preemptivně, podobně jako přepínání mezi procesy. V Presentation Manageru, každé vlákno mohlo a nemuselo vytvořit frontu zpráv. Pokud chtělo vlákno vytvářet okna, muselo ji vytvořit. Pokud ale pouze zajišťovalo nějaké datové operace nebo grafický výstup, fronta nemusela být vytvořena. Vlákna bez fronty nezpracovávala zprávy, tudíž nemohla způsobit kolaps systému nebo ho zablokovat. Jedinou nevýhodou bylo, že vlákno bez fronty nemělo způsob, kterým by mohlo poslat zprávu do vlákna s frontou, ani nemohlo zavolat si funkci, která by tuto zprávu poslala. Programátoři to řešili způsobem, že rozdělili program na jedno vlákno, které postupně vytvořilo všechna okna a zpracovávalo zprávy na další okna bez fronty. Tato vlákna se starala o déletrvající úlohy na pozadí. V té době vzniklo pravidlo 1/10 sekundy.

6.4.1 Pravidlo 1/10 sekundy

Vláknem, které zpracovává zprávy, by nemělo její zpracování trvat déle jak 1/10 sekundy. Jakákoliv další operace, která by trvala déle, než tento časový úsek, bylo převedena do dalšího vlákna. Kdyby se tímto pravidlem řídili všichni, nikdy by se nestalo, že by byl systém zablokovan programem déle jak na jednu desetinu sekundy.

6.5 Vícevláknová architektura

S vícevláknovou architekturou pracujeme nejčastěji tímto doporučeným způsobem. Primární vlákno vytvoří všechna okna, která potřebujeme pro program, a to včetně všech procedur těchto oken. Primární vlákno také zpracovává všechny zprávy těchto oken. Všechna ostatní okna jsou pak jen spuštěna na pozadí, kde zpracovávají úlohy. Nijak nekomunikují s uživatelem, pouze s primárním vláknem. Existuje také možnost, že primární vlákno obsluhuje uživatelské vstupy a další zprávy a vlákna vytváří až podle potřeby. Tato vlákna mají na starost činnosti, které nemají souvislosti s uživatelem. Vlákna v jedné aplikaci představují součást stejného procesu. Tím pádem sdílejí všechny prostředky procesu, kterými může být paměť nebo otevřené soubory a sdílejí také jeho statické proměnné. Díky tomu, že každé vlákno má svůj vlastní zásobník, každá automatická proměnná je v rámci vlákna jedinečná. Při přepínání vláken si kontext procesoru pro znovuoobnovení vlákna ukládá vlákno samotné.

6.6 Souboje vláken

Jedním z nejobtížnějších úkolů programátora ve Windows je správně navrhnout, naprogramovat a odladit vícevláknovou strukturu, která je poměrně složitá. Mezi vlákny může náhodně, nepravidelně a nepozorovatelně docházet k neplánovaným interakcím mezi vlákny. Je to způsobeno systémem preemptivního plánování, které umožňuje v kterémkoliv okamžiku vlákno přerušit a předat řízení jinému vláknu. Běžnou chybou ve vícevláknovém programu jsou „závodní podmínky“, kdy se předpokládá, že jedno vlákno dokončí úlohu dříve než tato data bude potřebovat jiné vlákno. Aby byla usnadněna koordinace jednotlivých vláken při jejich činnosti, používá operační systém různé algoritmy pro synchronizaci vláken. Pokud chceme pozastavit činnost vlákna v nějakém místě až do doby, kdy jiné vlákno neoznámí, že je možno pokračovat v činnosti, používáme synchronizační metodu semaforů. Semaforům jsou podobné kritické sekce. Jsou to sekce kódu, které mohou být vykonávány vždy jen jedním jediným vláknem. S problematikou semaforů ale souvisí i další možný problém, který je obvyklý pro vícevláknové programy. A to tehdy, pokud si dvě vlákna navzájem pozastaví běh a toto zablokování může být odstraněno pouze pokračováním běhu vlákna.

6.7 Výhody Windows

32bitové verze Windows, včetně Windows 98 a NT, používají „deserializovanou“ frontu zpráv. Pokud je určitá zpráva zpracovávána programem příliš dlouho, objeví se kurzor přesýpacích hodin pouze nad oknem této aplikace. Pokud ale najedeme nad jiné okno, kurzor se přemění zpět na šipku a můžeme přepnout okna. S programem, který pracuje nemůže uživatel pracovat, protože činnost programu nedovoluje přijetí dalších zpráv. Toho by ale aplikace měla být schopna, proto používáme sekundární vlákna. Ve Windows NT a Windows 98 neexistuje žádný rozdíl mezi vlákny s frontou a bez ní, na rozdíl od Presentation Manageru. Každé vlákno získává svou frontu zpráv v okamžiku jeho vytvoření. Dále Windows umožňuje díky funkcím, které má, jednomu vláknem zrušit jiné vlákno v tomtéž procesu. První verze systémů možnost „zabití“ vlákna neumožňovaly. Další výhodou systémů Windows je lokální úložný prostor vlákna (TLS – thread local storage). Protože jsou statické proměnné, lokální i globální, uloženy v paměťové oblasti dat procesu, jsou sdíleny všemi vlákny. Každé vlákno má vlastní zásobník, do kterého se ukládají lokální proměnné, které jsou pro každé vlákno jedinečnými.

6.8 Perspektivy Multithreadingu

Není vhodné používat více vláken v aplikaci, která to nepotřebuje. Existují programy, které nemají jak využít funkci více vláken. Při častém použití funkce PeekMessage, kterou se obchází přesýpací hodiny, je pro aplikaci vhodnější použít vícevláknovou strukturu. Při spouštění aplikace musí uživatel počítat s delší časovou prodlevou, neznamená to hned, že by bylo potřeba pro nahrávání souboru použít samostatné vlákno. Jen by se nám zvýšila režie na program.

II. PRAKTICKÁ ČÁST

BANKÉŘŮV ALGORITMUS

Při řešení problému uvážnutí se setkáváme s bankéřovým algoritmem. Nejvýhodnější strategií přidělování prostředků procesoru, která by procesy neomezovala tak dlouho dokud by nehrozilo nebezpečí zablokování, ale až ve chvíli, kdy se objeví skutečné nebezpečí zablokování by strategie prostředek nepřidělila a nechala by si ho v zásobě, dokud se situace nezlepší a přidělením prostředku by nebezpečí nehrozilo. Právě touto strategií je bankéřův algoritmus. Jeho název vznikl na základě jedné slovní úlohy. Ta spočívala v převádění problému správce prostředků do reálného světa.

Princip:

Máme bankéře (správce procesů), který má na starosti a k dispozici jisté částky peněz v různých měnách, kterými v našem případě počty jednotlivých prostředků. Bankéř půjčuje peníze zákazníkům (úlohám) a ti mu je opět vrací, ale až po dokončení svých obchodů. Bankéř ale nemůže půjčovat peníze dle libosti. Jeho úkolem je půjčovat peníze takovým způsobem, aby nikdy nemohlo dojít k situaci, že by banka byla navždy insolventní, tedy že by nebyla schopna splnit požadavky svých zákazníků, kteří by pak nemohli vracet vypůjčené peníze, protože pro nedostatek dalších peněz by nebyli schopni dokončit své záměry. Aby mohl bankéř zajistit tento požadavek, potřebuje předem znát maximální možné požadavky každého svého zákazníka. Pak je schopen při každé schůzce uvažovat tak, že peníze půjčí zákazníkovi jen tehdy, když má zaručeno, že existuje alespoň jedno pořadí dalších požadavků, při kterém všichni zákazníci ukončí své záměry a vrátí peníze. Pořadí může bankéř ovlivnit lehce, a to tak, že řekne žádajícímu, aby přišel později.

Zjednodušeně řečeno, bankéř může rozdělit pevný kapitál abstraktních peněžních jednotek mezi pevný počet zákazníků. Každý zákazník musí udat předem svou maximální potřebu peněžních jednotek. Bankéř vyhoví zákazníkovi, pokud potřeba zákazníka nepřevyšuje kapitál bankéře. Během finančních transakcí zákazník může vracet nebo si půjčovat peněžní jednotky. Bankéř také musí zaručit konečnou čekací dobu zákazníka na přidělení peněžních jednotek. Běžná půjčka zákazníka nesmí nikdy převýšit jeho maximální potřebu. Zároveň zákazník zaručuje, že dokončí své transakce a splatí půjčku v konečném čase.

Možné varianty úvah bankéře:

- Musí si nejprve rozmyslet, jestli je bezpečné půjčit peníze. Bude tedy předpokládat, že je půjčil a zkusí, jestli mu zbytek peněz dokáže zajistit splnění budoucích požadavků.
- Musí projít všechny své zákazníky a zjistit, jestli je mezi nimi alespoň jeden, který mu bude schopen vrátit peníze. Takový, jehož budoucí požadavky nepřesáhnou množství peněz, které bankéři zbývají. Pokud by se stalo, že žádný takový zákazník neexistuje, peníze určitě nepůjčí.
- Pokud někdo takový existuje, představí si, že mu zákazník peníze vrátil a poté obdobným způsobem zjistí, jestli mu bude moci peníze vrátit některý z dalších zákazníků. Pokud by tomu tak nebylo, peníze samozřejmě také nepůjčí.
- Úvahu z minulé myšlenky opakuje bankéř tak dlouho, dokud nezjistí, že peníze půjčit nemůže nebo dokud nevyhoví všem zákazníkům. V druhém případě je vše jak má být a peníze může doopravdy půjčit.

Z bankérova algoritmu vyplývá i důkaz jeho správnosti. Pokud existuje alespoň jedno pořadí požadavků, ve kterém nedojde k zablokování – to je takový stav, kdy všichni zákazníci nakonec budou schopni splnit své záměry – může operační systém toto pořadí v nejhorším případě vynutit selektivním splňováním požadavků. Nejhorším případem máme na mysli, kdy žádný zákazník nevrátí ani část peněz před dokončením svých záměrů.

Existují i matematické důkazy o správnosti:

- Po každé výpůjčce, nebo-li přidělení prostředků, existuje alespoň jeden zákazník, který může své záměry ukončit. Pokud by se stalo, že by nikdo takový neexistoval, bankéř by za žádných okolností peníze nepůjčil
- Po navrácení peněz a odchodu zákazníka, tj. ukončení úlohy, opět existuje alespoň jeden zákazník, která může své záměry ukončit. Buď totiž odešel jiný zákazník (své záměry může jistě ukončit, mohl už předtím a teď má bankéř ještě více volných prostředků) nebo odešel právě tento zákazník. Takovou situaci ale bankéř už simuloval a zjistil, že i takovém případě existuje někdo další, kdo své záměry může dokončit. Jinak by peníze nebyly zapůjčeny.

Tato úvaha může být dovedena až do uzavření banky po vrácení všech peněz a odbavení všech zákazníků. Tedy do konce práce operačního systému bez zablokování.

Samotný algoritmus má stav jistý, a to když bankéř umožní všem zákazníkům dokončení všech jejich transakcí v konečném čase, dále stav nejistý, pokud bankéř tuto skutečnost nemůže zaručit.

Příklad: Máme bankéře, který má k dispozici 12 peněžních jednotek a zákazníky, které si zažádají o půjčku. Zákazníci P, Q a R si zažádají o půjčky a to P o dvě jednotky z pěti potřebných k jeho uspokojení, Q si zažádá o 4 jednotky z 8 a R o 2 ze 7. Bankéři tím pádem zbývají 4 jednotky. Pak přijde zákazník S, který si zažádá o 3 z 5 potřebných. Bankéř mu je může půjčit, ale nejdříve poskytne 3 jednotky zákazníkovi P, aby ho plně uspokojil a on mu vrátil všech pět jednotek, teprve pak může zaručit podmínku stavu jistého a půjčit jednotky zákazníkovi S.

Algoritmus pro ověřování stavu jistého a nejistého je v příloze P I. Program používá tři pomocné funkce. A to „simulovat_prideleni“. Tato funkce vytvoří novou kopii seznamu jednotek, které má bankéř momentálně k dispozici a odebere z tohoto seznamu jednotku, o kterou ho žádá proces, jehož žádost je právě zkoumána. Další funkce „lze_dokoncit“ má parametr a to číslo procesu. Porovná momentální obsah lokálního seznamu prostředků s maximálními budoucími požadavky zadaného procesu. Vrací nulu, pokud jsou požadavky procesu nesplnitelné. Funkce „simulovat_dokonceni“ má také jeden parametr, číslo procesu. Přidá k lokálnímu seznamu prostředků všechny prostředky, které jsou zadanému procesu přiděleny.

VÝROBCE – KONZUMENT

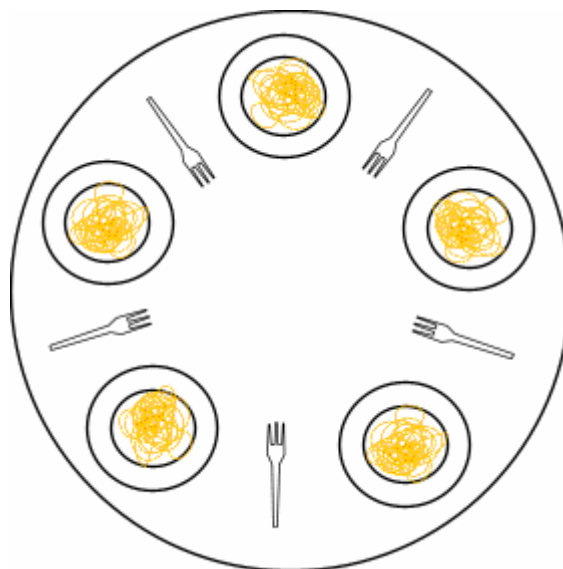
Dva procesy sdílejí společně zásobník o dané velikosti. Jeden z nich (výrobce), dává do zásobníku jednotky. Druhý (konzument) si je z ní vybírá. Tento problém lze zevšeobecnit i na n výrobců a m konzumentů, ale předpoklad jednoho výrobce a jednoho konzumenta zjednoduší řešení. Problém nastává v případě, kdy výrobce chce dát novou jednotku do zásobníku, který je ovšem plný. Jako řešení se nabízí, aby šel výrobce „spát“, tedy přestal vyrábět jednotky a nechal se probudit v momentě, kdy si konzument odebere ze zásobníku jednu nebo více jednotek. Podobně, když konzument si chce vzít jednotku z prázdného zásobníku. Jde „spát“ do té doby než výrobce nevyrobí jednotku, nedá ji do zásobníku a neprobudí jej. Tohle přiblížení se zdá dostatečné, ale vede k souběhu. Aby bylo možné sledovat počet jednotek v zásobníku, potřebujeme proměnnou počet. Jestliže maximální možný počet jednotek v zásobníku je N , funkce producenta nejprve bude testovat proměnnou počet, jestli se nerovná N . V případě rovnosti producent přestane vyrábět jednotky. „Uspí se“. V případě nerovnosti přidá jednotku do zásobníku a proměnnou počet zvýší o jedna. Funkce „konzument“ bude podobná. Nejprve si otestuje proměnnou počet jestli se rovná nule. Pokud ano, odebere si jednotku a sníží proměnnou počet o jedna. V opačném případě se „uspí“. Každý z procesů (výrobce i konzument) také testuje, jestli ho druhý proces neprobudil.

Může nastat situace, kdy přístup k proměnné počet není kontrolovatelný. Zásobník je prázdný a konzument se právě přesvědčil, že v něm nic není. V tom samém okamžiku se plánovač rozhodne dočasně pozastavit proces „konzument“ a spustí výrobce. Ten vloží do zásobníku jednotku a zvýší hodnotu proměnné počet o jedna. Otestuje stav proměnné a zjistí, že zásobník byl předtím prázdný a proto zavolá funkci na probuzení konzumenta. Naneštěstí konzument se ještě neuspál a tak signál na probuzení se ztratí. Když se pak proces konzument stane „běžící“, testuje hodnotu počtu jednotek v zásobníku, ale tu, kterou si již načetl předtím, než konzument vložil do zásobníku jednotku, takže se uspí. Dříve či později pak výrobce naplní zásobník a uspí se. Oba tak usnou navždy.

Podstata problému spočívá v tom, že pokud se vyšle signál na probuzení procesu, který ještě nespí, ztratí se a dojde tak k uváznutí. Tomu se dá zabránit tak, že přidáme proměnnou budící bit. Ten se nastaví v případě, kdy je zaslán impuls k buzení procesu, který ještě neusnul. Když se proces snaží usnout a přitom má nastavený budící bit, musí u této proměnné nejprve změnit hodnotu a pak usne.

PROBLÉM VEČEŘÍCÍCH FILOZOFŮ

V roce 1965 Dijkstra vyřešil požadavek synchronizace a tuto problematiku popsal a vysvětlil na příkladu večeřících filozofů. Problém je postaven následujícím způsobem. Pět filozofů sedí kolem kulatého stolu. Každý z nich má před sebou talíř špaget, které jsou tak hladké, že se dají sníst pouze vidličkou. Mezi každými dvěma talíři leží jedna vidlička.



Obr. 7 Večeře filozofů

Život filozofů se skládá ze střídání se myšlení a jezení. Když dostane filozof hlad, snaží se v jednom okamžiku získat levou i pravou vidličku, je jedno v jakém pořadí. Když se podaří získat vidličky, nají se, odloží je a pokračuje v přemýšlení. Klíčová otázka spočívá v tom, jestli je možné napsat program pro každého filozofa, aby si plnil svoje povinnosti a přitom se nikdy nezadrhnul. V programu (viz příloha P III) je procedura `take_fork` (vezmi vidličku), která čeká na to, až se uvolní vidlička a pak ji uchopí. Tohle řešení je ale špatné. Pokud by si všech pět filozofů vzalo současně levou vidličku, nikdo nebude schopen vzít si pravou vidličku a dojde k uváznutí. Program se rozšíří o schopnost, že pokud uchopí filozof levou vidličku a zjistí, že pravá není dostupná, levou odloží zpátky na stůl, počká nějakou dobu a pak celý proces zopakuje. Tento návrh řešení má ale také své nedostatky. Při troše smůly, všichni filozofové mohou ve stejném okamžiku vzít svou levou vidličku, zjistí, že pravá není dostupná, vidličku odloží a za chvíli to zkusí zase. Vezmou levou vidličku všichni současně a pak ji odloží. A takto to bude pokračovat donekonečna. Takové situace, ve kterých program běží po neurčitou dobu a přitom neudělá žádný pokrok se nazývá „vyhladovění“. Nabízí se řešení, kdy by každý filozof čekal různě dlouhou dobu

na to, než znova zkusí vzít vidličku. Toto opatření se využívá i u sítě Ethernet. Pokud dva počítače chtějí vyslat ve stejném okamžiku paket, čekají náhodně dlouhou chvíli a teprve pak jej odešlou. Aby nedošlo v programu k uvážnutí ani k „vyhladovění“, zavolá se funkce „myšlení“ pomocí binárního semaforu. Předtím, než si filozof vezme vidličku, sníží hodnotu na semaforu o jedna. Z teoretického hlediska je toto řešení adekvátní, prakticky to má ale „vadu na výkonu“. V jednom okamžiku totiž může jíst pouze jeden filozof. S pěti vidličkami by se ale mělo zaručit, aby v jednom okamžiku jedli alespoň dva filozofové. V příloze P III je popsán algoritmus, ve kterém nehrozí uvážnutí a je umožněn maximální paralelismus pro libovolný počet filozofů. Využívá pole, stav, ve kterém se uchovává informace o tom, jestli filozof jí, myslí nebo hladoví(snaží se vzít vidličku). Filozof se může přimět pouze do stavu „jezení“, pokud ale žádný z jeho sousedů zrovna nejí špagety. Sousedi filozofa jsou v programu nadefinováni pomocí maker LEFT a RIGHT (pravý a levý). Program využívá pole semaforů. Jedno pro filozofy, aby se hladoví filozofové zablokovali, pokud není k dispozici žádná vidlička.

ČTENÁŘI – SPISOVATELÉ

Tento příklad řeší otázku souběhu. Konkrétně přístupu k databázím. U těch je potřeba zaručit, aby do něj mohli uživatelé zapisovat a přitom ostatní z něj číst. V programu (viz příloha P IV) první čtenář, který dostane přístup k databázi nastaví hodnotu semaforu na DOWN. Následující čtenáři pouze navyšují čítač. Když čtenáři odejdou, poslední nastaví na semaforu hodnotu UP a dovolí tak spisovateli přístup k databázi, pokud ovšem nějaký čeká ve frontě. Pokud alespoň jeden čtenář čte databázi, není problém, aby k němu přibyli i další. Když přijde spisovatel a nemá výhradní přístup a v databázi jsou stále ještě čtenáři, je spisovatel pozastaven. Hrozí nebezpečí, že pokud chodí každé dvě vteřiny čtenář a setrvává v databázi 5 vteřin, spisovatel se nikdy nedostane dovnitř. Abychom zamezili takové situaci, tak ve frontě stojí čekající spisovatel a přijde čtenář, není ihned vpuštěn dovnitř, ale zařadí se za spisovatele. Tím pádem spisovatel čeká pouze na čtenáře, kteří přišli k databázi před ním, ale už ne na ty, kteří přijdou po něm. Toto řešení má ale také nevýhodu. Vede k nízkému paralelnímu zpracování procesů a tím pádem i k nižšímu výkonu. Nabízí se možnost zvýšit prioritu spisovatelům.

PROBLÉM SPÍCÍHO HOLIČE

V holičství pracuje jeden holič. Je v něm jedno křeslo pro holiče a židličky pro čekající zákazníky. Pokud není v holičství žádný zákazník, holič si sedne do křesla a usne. Příchozí zákazník musí holiče probudit. Když holič stříhá a přijde další zákazník, sedne si na židli pro čekající zákazníky anebo v případě, že jsou již všechny obsazeny odejde z obchodu. Hrozí nebezpečí, kdy se holič a zákazník mohou dostat do situace souběhu. Řešení v programu v příloze P V využívá 3 semaforů: zákazníci – počítá počet čekajících zákazníků (bez zákazníka, který sedí v křesle, ten nečeká), holiči (počítá počet holičů, může nabývat hodnot 0 a 1), kteří zahálí (čekají na zákazníka) a mutex, který se používá pro vzájemné odepření vstupu do kritické sekce. Je zapotřebí proměnné „čekající“, která počítá počet zákazníků ve frontě čekajících. V podstatě je to obdoba zákazníka. Této proměnné je zapotřebí, když potřebujeme zjistit aktuální stav semaforu. Zákazník, který vchází do dveří, zvýší počet čekajících zákazníků. Pokud je počet menší jak počet židlí, zůstane. V opačném případě opustí holičství. Když se ráno holič dostaví do práce, spustí proceduru holič, která ho zablokuje, protože semafor zákazníci je ještě na nule. S příchodem zákazníka se spustí procedura zákazník, která získá od semaforu mutex přístup do kritické sekce. Pokud vzápětí přijde další zákazník, nebude moci nic udělat, dokud neopustí předchozí zákazník kritickou sekci. Zákazník si zkontroluje počet židlí a pokud je všude obsazeno, odejde bez ostříhání. Pokud je ale židle volná, zvýší o jedna proměnnou čekající. Potom nastaví na semafor zákazníci hodnotu UP. Tedy vzbudí holiče. V jednom okamžiku jsou tedy vzhůru holič i zákazník. Když pak zákazník uvolní semafor mutex, chopí se ho holič a začne stříhat. Když je stříhání dokončeno, zákazník opustí proceduru a také obchod. Oproti předchozím příkladům zde není žádná smyčka, protože každý zákazník může být ostříhán pouze jednou. Pokud je přítomen zákazník, holič stříhá, jinak jde spát.

ZÁVĚR

Předmětem této bakalářské práce bylo zpracovat problematiku týkající se procesů a vláken, vypracovat učební texty pro studenty, prezentaci pro přednášky, soubor zkouškových otázek a html verzi praktické části. Hlavním tématem práce je shrnutí poznatků o procesech a vláknech v rámci operačních systémů. V první, teoretické části bakalářské práce jsou rozebrány hlavní problémy související s vývojem aplikací, využívající principů současných návrhů aplikací. Na první kapitolu, popisující základní pojmy této oblasti teoretické informatiky, navazuje úvod do procesní architektury informačních systémů. Je vysvětlen pojem proces, práce s ním a jeho účelové využití. Třetí kapitola je věnována plánovačům a strategiím při plánování procesu. Další kapitola pojednává o spolupráci mezi procesy a vysvětluje problémy kritické sekce a souběhu. Pátá kapitola se zmiňuje o uváznutí a možnostech, kterými se mu dá předejít. Poslední kapitola je zaměřena na vlákna samotná, multitasking, multithreading a vícevláknovou architekturu systémů.

V druhé části je bakalářská práce doplněna o praktické algoritmy ukazující nejznámější standardizované postupy řešení problémů souběhu, zacyklení a uváznutí. Je zde uveden nejznámější bankéřův algoritmus. Jeho strategie přidělování prostředků spočívá v tom, že procesy neomezuje tak dlouho, dokud opravdu nehrozí nebezpečí. Příklad výrobce konzument je zde vyřešen jak za pomoci semaforů, tak za pomoci zasílání zpráv. Příklad večeřících filozofů řeší synchronizaci procesů do takové míry, aby byl zaručen i maximální výkon zpracování procesů. Příklad spícího holiče je také řešen pomocí semaforů, aby nedošlo k uváznutí. Algoritmus čtenáři spisovatelé řeší souběžný přístup k databázovým objektům za předpokladu použití souběžných transakcí.

.

SEZNAM POUŽITÉ LITERATURY

- [1] DEITEL, Harvey M. *Operating Systems*. Third edition. Pearson Prentice Hall. c2004. ISBN 0-13-124696-8. Part 7, Security, s.869-919
- [2] TANENBAUM A. S. *Modern operating systems*. Second edition. China Machine Press. English reprint edition c2002. ISBN 7-111-09156-6.
- [3] ČADA, O. *Operační systémy*. Grada, 1993. ISBN 80-85623-44-7
- [4] KLIMEŠ, C. *Operační systémy 1*. Ostravská univerzita v Ostravě, Katedra informatiky a počítačů
- [5] KRUGLINSKI, D., SHEPHERD G., WINGO S. *Programujeme v Microsoft Visual C++*. Computer Press Praha, 2000. ISBN 80-7226-362-5
- [6] PETZOLD, CH. *Programování ve Windows* Computer Press Praha, 1999 ISBN 80-7226-206-8
- [7] VYCHODIL, V. *Linux: příručka českého uživatele*, Computer Press 2003 Brno. ISBN 80-7226-333-1
- [8] NUTT, G.J. *Operating Systems: A Modern Perspective*, Addison-Wesley, 2000. ISBN 0-201-61251-8
- [9] ANDREWS, G.R. *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison – Wesley, 2000, ISBN 0-201-56317-7
- [10] *Linux – dokumentační projekt*, Computer Press, 2003, ISBN 80-7226-761-2
- [11] KOLÁŘ, Petr. *Operační systémy*[online]. [cit. 2005-08-26]. Dostupný z WWW: <<http://www.kai.vslib.cz/~kolar/os/>>
- [12] SOLOMON, David. A.: *Windows NT pro administrátory a vývojáře*, Computer Press 1999, ISBN 80-7226-147-9
- [13] MICROSOFT CORP. *Microsoft Windows XP Professional Resource Kit*, Computer Press Praha, 2002. ISBN 80-7226-608-X
- [14] SOBELL, M.G. *Linux Praktický průvodce*, Computer Press Praha 1999. ISBN 80-7226-190-8
- [15] KOROREVA O. *Registr Microsoft Windows XP*, Computer Press Praha 2002. ISBN 80-7226-783-3
- [16] GNU/GPL *Linux Dokumentační projekt*, Computer Press 1998. ISBN 80-7226-114-2

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

GID	Identifikace skupiny, do které vlastník patří (Unix)
UID	Identifikace vlastníka (Unix)
FAP	Fyzický adresní prostor
LAP	Logický adresní prostor
OS	Operační systém
CPU	Procesor

SEZNAM OBRÁZKŮ

Obr. 1 Základní stavový model procesů	10
Obr. 2 3-stavový model	11
Obr. 3 Stavový model procesů.....	12
Obr. 4 Ukázka uvážnutí v grafu.....	29
Obr. 5 Zdánlivý cyklus, nejedná se o uvážnutí.....	31
Obr. 6 Uvážnutí.....	31
Obr. 7 Večeře filozofů	46

SEZNAM PŘÍLOH

- P I BANKÉŘŮV ALGORITMUS
- P II VÝROBCE – KONZUMENT
- P III PROBLÉM VEČEŘÍCÍCH FILOZOFŮ
- P IV ČTENÁŘI - SPISOVATELÉ
- P V PROBLÉM SPÍCÍHO HOLIČE
- P VI CD ROM

PŘÍLOHA P I: BANKÉŘŮV ALGORITMUS

```
int lze_pridelit(int prostredek)
{
    enum {OK,LOCK} tab [pocet_procesu];

    int i, locked;

    simulovat_prideleni(prostredek);

    for(i=0;i<pocet_procesu;i++)
        tab[i]=LOCK;

    Dokola:

        locked=0;

        for(i=0;i<pocet_procesu;i++)
            if(tab[i]==LOCK)
                if(lze_dokoncit(i)) {
                    simulovat_dokonceni(i);

                    goto Dokola;

                }else locked++;

        return(!locked);
}
```

PŘÍLOHA P II: VÝROBCE KONZUMENT

```
#define N 100                                /*počet míst v zásobníku*/

int count = 0;                                /*počet jednotek v zásobníku*/

void producer (void)
{
    int item;

    while (TRUE) {                            /*opakovací cyklus*/
        item = produce_item();                /*výroba jednotky*/
        if (count == N) sleep();              /*podmínka–usni, když je zásobník
plný*/
        insert_item(item);                    /* vloží jednotku do zásobníku*/
        count = count + 1;                    /*zvýší počet jednotek v zásobníku o1*/
        if (count == 1) wakeup(consumer);    /*zjišťuje, zda byl zásobník prázdný*/
    }
}

void consumer (void)
{
    int item;

    while (TRUE) {                            /*nekonečný cyklus*/
        if (count == 0) sleep();              /*podmínka, je-li je zásobník prázdný,usní*/
        item = remove_item();                 /*vezme jednotku ze zásobníku*/
        count = count – 1;                    /*sníží počet jednotek v zásobníku o1*/
        if (count == N-1) wakeup(producer);  /*zjišťuje, zda byl zásobník plný*/
        consume_item(item);                   /*vypiš počet jednotek*/
    }
}
```

VÝROBCE KONZUMENT – ŘEŠENÍ SE SEMAFORY

```
#define N 100                                /*počet jednotek v zásobníku*/
typedef int semaphore;                       /*nedefinování*/
semaphore mutex = 1;                         /*kontroluje přístup do kritické sekce*/
semaphore empty = N;                         /*počítá prázdné jednotky v zásobníku*/
semaphore full = 0;                          /*počítá počet zaplněných jednotek v zásobníku*/
void producer (void)
{
    int item;
    while (TRUE) {
        item=produce_item();/*výroba jednotky*/
        down(&empty);      /*snižuje počet prázdných jednotek*/
        down(&mutex);      /*vstup do kritické sekce*/
        insert_item(item); /*výroba jednotky*/
        up(&mutex);        /*opuštění kritické sekce*/
        up(&full);         /*navýšení počtu jednotek v zásobníku*/
    }
}

void consumer(void)
{
    int item;
    while(TRUE) {          /*nekonečná smyčka*/
        down(&full);        /*sníží počet jeedotek v zásobníku*/
        down(&mutex);      /*vstup do kritické sekce*/
        item=remove_item{};/*vezme jednotku ze zásobníku*/
        up(&mutex);        /*opuštění kritické sekce*/
        up(&empty);        /*zvýší počet prázdných jednotek v zás.*/*
        consume_item(item);/*provede něco s jednotkou*/
    }
}
```

VÝROBCE KONZUMENT – ŘEŠENÍ POMOCÍ ZASÍLÁNÍ ZPRÁV

```
#define N 100                                /*počet jednotek v zásobníku*/

void producer(void)
{
    int item;

    message m;                                /*zásobník zpráv*/

    while (TRUE) {

        item=produce_item();/*výroba jednotky*/

        receive(consumer,&m);    /*čeká na návrat zprávy s hodnotou nula*/

        build_message(&m,item);  /*vytvoří zprávu k odeslání*/

        send(consumer,&m);        /*pošle jednotku konzumentovi*/

    }
}

void consumer(void)
{
    int item, i;

    message m;

    for(i=0;i<N;i++)send(producer,&m); /*posílá počet prázdných jednotek*/

    while(TRUE) {

        receive(producer,&m);    /*dostává zprávu obsahující jednotku*/

        item=extract_item(&m);   /*vyjme jednotku ze zprávy*/

        send(producer,&m);        /*posílá zpátky prázdnou(nulovou)odpověď*/

        consume item(item);      /*provede něco s jednotkou*/

    }
}
```

PŘÍLOHA P III: PROBLÉM VEČEŘÍCÍCH FILOZOFŮ

```
#define N          5                /*počet filozofů*/

#define LEFT      (i+N-1)%N        /*počet levých sousedů*/

#define RIGHT     (i+1)%N          /*počet pravých sousedů*/

#define THINKING  0                /*filozof přemýšlí*/

#define HUNGRY    1                /*filozof se pokouší vzít vidličku*/

#define EATING    2                /*filozof jí*/

typedef int semaphore;             /*semafor jako typ integer*/

int state[N];                     /*pole, které sleduje stav všech filozofů*/

semaphore mutex=1;                /*vzájemné odepření vstupu do kritické
sekce*/

semaphore s[N];                  /*jeden semafor pro filozofy*/

void philosopher(int i)           /*i-číslo filozofa, od 0 do N-1*/
{
    while(TRUE) {                 /*nekonečná smyčka*/
        think();                  /*filozof přemýšlí*/
        take_forks(i);            /*vzít dvě vidličky nebo se zablokovat*/
        eat();                    /*konzumace špaget*/
        put_forks(i);             /*vrácení dvou vidliček na stůl*/
    }
}
```

```

void take_forks(int i)                                /*i-číslo filozofa, od 0 do N-1*/
{
    down(&mutex);                                     /*vstup do kritické sekce*/
    state[i]=HUNGRY;                                  /*zaznamená stav, že je filozof hladový*/
    test(i);                                           /*zkouší, jestli může vzít obě vidličky*/
    up(&mutex);                                        /*odchod z kritické sekce*/
    down(&s[i]);                                       /*zablokování se pokud není možno vzít vidličky*/
}

void put_forks(i)
{
    down(&mutex);                                     /*vstup do kritické sekce*/
    state[i]=THINKING;                                /*filozof dojedl*/
    test(LEFT);                                       /*podívej se, jestli levý soused může jíst*/
    test(RIGHT);                                     /*podívej se, jestli pravý soused může jíst*/
    up(&mutex);                                       /*výstup z kritické sekce*/
}

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

```

PŘÍLOHA P IV ČTENÁŘI – SPISOVATELÉ

```
typedef int semaphore;

semaphore db=1; /*kontroluje přístup do ´rc´*/

semaphore mutex=1; /*kontroluje přístup k databázi*/

int rc=0; /*počet procesů v databázi nebo čekajících na ni*/

void reader(void)
{
    while(TRUE) { /*nekonečný cyklus*/
        down(&mutex); /*získá exklusivní přístup k ´rc´*/
        rc=rc+1; /*jeden čtenář přišel*/
        if(rc==1)down(&db); /*pokud tohle je první čtenář..*/
        up(&mutex); /*dej výhradní přístup k ´rc´*/
        read_data_base(); /*přístup k datům*/
        down(&mutex); /*získá exklusivní přístup k ´rc´*/
        rc=rc-1; /*jeden čtenář odešel*/
        if(rc==0)up(&db); /*pokud je to poslední čtenář...*/
        up(&mutex); /*uvolní výhradní přístup k ´rc´*/
        use_data_read(); /*použije data*/
    }
}

void writer(void)
{
    while(TRUE) { /*nekonečný cyklus*/
        think_up_data(); /*vytvoří data*/
        down(&db); /*získá výhradní přístup*/
        write_data_base(); /*provede aktualizaci dat*/
        up(&db); /*uvolní výhradní přístup*/
    }
}
```

PŘÍLOHA P V PROBLÉM SPÍCÍHO HOLIČE

[illegible]

PŘÍLOHA P VI: CD-ROM

Součástí bakalářské práce bylo vypracování zjednodušené HTML verze vhodné pro publikování na internetu, vytvoření prezentace pro přednášky předmětu Operační systémy a návrh souboru zkouškových otázek. Soubory jsou dostupné na přiloženém CD-ROM.