Konference Czech Warpstock 2002

Pořádají
Server www.os2.cz a uživatelé OS/2

Lehký úvod do multithreadingu OS/2

Jaroslav Kačer

Článek konference Czech Warpstock č. CZW-009
Červenec 2002

Článek konference Czech Warpstock č. CZW-009
Červenec 2002

Lehký úvod do multithreadingu OS/2

Jaroslav Kačer

Résumé

Tento článek se zabývá problematikou multithreadingu v operačním systému OS/2. Představuje principy multithreadingu použité v tomto systému a funkce API pro práci s vlákny a semafory. Jejich použití je dále demonstrováno na řešeném příkladu problému večeřících filozofů, implementovaném v jazyce C. Implementace pomocí funkcí OS/2 API je následně srovnána s implementací pomocí vláken dle standardu POSIX a s implementací pomocí vláken programovacího jazyka Java.

Informace o autorovi

E-Mail: `jkacer@kiv.zcu.cz`
WWW: `home.zcu.cz/~jkacer`
Jaroslav Kačer (* 1978) je od roku 2001 doktorandem na Katedře informatiky a výpočetní techniky Západočeské univerzity v Plzni. Momentálně se věnuje převážně paralelnímu programování v jazyce Java a diskrétním procesově orientovaným simulacím. Podílí se na výuce předmětů *Programování v jazyce C* a *Objektově orientované programování*. Systém OS/2 používá od roku 1997.

1 Paralelismus a OS/2

Jednou z možností zvyšování výkonu počítačových systémů je zavedení paralelního zpracování dat. Ovšem paralelismum je častokrát vhodné zavést i tam, kde k žádnému zpracovávání masivního množství dat nedochází, jak si ukážeme v kapitole 1.2. V této kapitole budou uvedeny základní pojmy z oblasti paralelismu a základní myšlenky implementace paralelismu do operačního systému OS/2.

1.1 Paralelní zpracování

Paralelní výpočet je takový výpočet, který je *dynamicky strukturován na složky* (procesy, vlákna, ...), přičemž tyto složky spolupracují na dosažení celkového cíle výpočtu. Naproti tomu *paralelní program* je *staticky strukturován na programy* těchto složek a případné deklarace *sdílených (společných, globálních) dat*. Program je tedy statický popis výpočetního postupu, který se nemění a nemá stav. Naproti tomu proces (vlákno, ...) je aktivita probíhající podle nějakého programu, jejíž stav je určen aktuálním místem v příslušném programu a aktuální hodnotou dat procesu.

1.1.1 Data v paralelním zpracování

Data, se kterými proces pracuje, mohou být buď *lokální* anebo *globální*. Lokální data patří jen jednomu procesu a z ostatních procesů nejsou přístupná. Globální data jsou přístupná z více procesů a mohou jimi být čtena nebo měněna.

Globální proměnné lze rozdělit podle přístupu k nim do následujících kategorií:

- **nezávislé** – Jedná se o proměnné buď jenom pro čtení (RO) anebo o takové proměnné, do kterých zapisuje právě jeden proces (a nejsou paralelně čtena jiným procesem). Může sem patřit i pole, do kterého zapisuje více procesů, ovšem do jednoho prvku pole může zapsat právě jeden proces. Přístup k těmto proměnným nevyžaduje žádnou zvláštní péči.
- **závislé** – Práce s těmito proměnnými není tak bezproblémová jako v předchozí kategorii. Přistupujeme-li k nim, musíme využít nějaké

formy *synchronizace* procesů. (Synchronizaci procesů – přesněji synchronizaci OS/2 vláken – se budeme věnovat v kapitolách 2.2 a 2.3.) Podle svých vlastností se proměnné této kategorie dělí na tři podkategorie:

1. **uzamykané** – Jedná se o proměnné, které jsou čteny i měněny více procesy současně. Každý přístup k nim je třeba synchronizovat.
2. **redukční** – Téměř stejné jako předchozí kategorie, lze ovšem výrazně ušetřit čas nutný pro synchronizaci procesů tím, že většinu času se místo s danou globální proměnnou pracuje s její lokální náhradou a teprve při skončení výpočtu se globální proměnná aktualizuje hodnotou její lokální náhrady.
3. **uspořádané** – Při práci s těmito proměnnými záleží na pořadí provádění kódu v jednotlivých procesech. Takovéto úlohy nelze paralelizovat.

K synchronizaci procesů za účelem ochrany dat se nejčastěji využívají *semafory* a *zámky*. Hlavní rozdíl mezi semafory a zámky tkví ve *způsobu čekání* na dostupnost kódu chráněného mezi 'otevírací a uzavírací závorkou' semaforu nebo zámku. Semafor má frontu, kam řadí uspané procesy, které se zatím neuspěšně pokoušejí o vstup do chráněného kódu. Když proces právě provádějící kód chráněný semaforem provede jeho odemčení, je vzbuzen první proces zařazený do fronty semaforu a může pokračovat ve vykonávání chráněného kódu. Naproti tomu čekání na vstup do kódu chráněného zámkem je aktivní, procesy nejsou uspány.

Dalším prostředkem pro ochranu dat v paralelním prostředím jsou tzv. *monitory*. Jsou to objekty, jejichž vnitřní data jsou chráněna tím, že je možno k nim přistupovat pouze přes tzv. *monitorové procedury*. Ty jsou synchronizované (buď semaforem anebo zámkem) a nedovolí tak více procesům provádět je současně. Monitory využívá například programovací jazyk Java (pomocí klíčového slova `synchronized`) anebo programovací jazyk ADA (pomocí tasků a klíčových slov `entry` a `accept`).

1.1.2 Architektury používané při paralelním zpracování

Základním kritériem pro rozdělení výpočetních architektur používaných pro paralelní výpočty je to, jaký typ paměti používají procesy pro vzájemnou komunikaci. Rozlišují se dva druhy architektur:

- **paralelní počítače (multiprocesory) se sdílenou pamětí** – Procesy mohou sdílet paměť (RAM) jednoho počítače. U *symetrických* multiprocesorů může běžet kterýkoliv proces na kterémkoliv procesoru, procesory jsou univerzální. Naproti tomu u *asymetrických* multiprocesorů je každý procesor specializován na určitý úkol.

- **paralelní počítače (multiprocesory) s distribuovanou pamětí** – Jedná se o více (většinou velké množství) strojů spojených dohromady pomocí komunikační sítě. Procesy spolu většinou komunikují *zasíláním zpráv* (tzv. message passing).

1.1.3 Realizace paralelních procesů

Typické způsoby realizace paralelních procesů jsou:

1. **samostatné úlohy operačního systému** – Typicky se používají u systémů s distribuovanou pamětí. Úlohy spolu nějak spolupracují, např. pomocí nástrojů typu PVM[1] nebo MPI[2]. Jedná se o tzv. *hrubý paralelismus*, v angličtině označovaný jako *coarse grain*.

2. **vlákna** (threads, lightweight processes) – Používají se u systémů se sdílenou pamětí. Vlákna běží v rámci jedné úlohy, tj. sdílí její kontext. Jedná se o tzv. *jemný paralelismus*, v angličtině označovaný jako *fine grain*.

3. **korutiny** – Nejedná se o opravdový paralelismus, spíše pseudo-paralelismus. Jedná se o procedury, které si vzájemně předávají řízení v jednoprocesorovém systému. Jindy než v přesně definované momenty k přepnutí nedochází. Jeden proces korutiny běží podle jedné specifikace procedury, stírá se tedy rozdíl mezi procesem a programem, popsaný v kapitole 1.1. V dnešní době se korutiny využívají např. při simulacích systémů pracujících v diskrétním čase, např. v programovacím jazyce Simula a nástrojích na něj navazujících (C-Sim, J-Sim, ...).

Každý paralelní proces má:

- **program**, podle kterého je prováděn;

- lokální **data**;

[1] Parallel Virtual Machine
[2] Message Passing Interface

- **stav** – hodnoty lokálních dat, hodnoty registrů procesoru, zásobník, místo v programu, kde se proces nachází.

Existují tyto základní druhy řízení paralelních procesů:

1. **časová kvanta** (Time Sharing) – Každému procesu je přiděleno určité časové kvantum, po jeho vypršení je procesu odebrán procesor a ten je dán jinému procesu.
2. **preemtivní multitasking** (Preemptive Multitasking) – Každý proces má přidělenu prioritu, běží proces s největší prioritou. Pokud se objeví proces s vyšší prioritou, než má právě běžící proces, je tento přerušen.
3. **řízení korutin** – Přidělování procesoru je řízeno korutinami samými, tj. právě prováděná korutina rozhoduje, jaká korutina poběží dále a kdy dojde k přepnutí.

1.2 Task management v OS/2

Dále se budeme zabývat pouze paralelismem operačního systému OS/2, tj. multiprocesorem se sdílenou pamětí, jehož základním stavebním kamenem je vlákno (thread). Pro přidělování procesoru se používá preemtivní multitasking, ovšem s mnoha vylepšeními.

Pod pojmem task management budeme chápat správu všech možných jednotek paralelismu přítomných v systému, což jsou:

- **relace** (sessions);
- **procesy** (processes);
- **vlákna** (threads).

Začneme od těch 'nejjemnějších' a postupně budeme postupovat k těm 'drsnějším'.

1.2.1 Vlákna

V OS/2 je vlákno jediná jednotka paralelismu, které může být přidělen čas procesoru. Jinak řečeno, plánovač vůbec nepracuje s procesy

ani s relacemi, pouze s vlákny. Vlákno nemůže v systému existovat samo o sobě, vždy musí patřit nějakému procesu. Existuje systémový limit 4096 vláken, který nemůže být překročen. Naopak, v souboru `CONFIG.SYS` je obvykle specifikována konstanta `THREADS`, která je nastavena na mnohem nižší hodnotu. U OS/2 Warp je to implicitně 1024 vláken. Část z tohoto množství používá samotný operační systém, ale na uživatelské programy by měl zbytek postačit více než dostatečně.

Každé vlákno by mělo mít nějaký přesně definovaný úkol, např. provedení určitého množství V/V operací, provedení náročnějšího výpočtu atd. V programování aplikací pro Presentation Manager se většinou používá systém jednoho řídicího vlákna a několika pomocných vláken. Hlavní vlákno zajišťuje cyklický příjem zpráv od systému a jejich interpretaci v tzv. *window proceduře* hlavního okna aplikace. Dojde-li ze strany uživatele k vybrání časově náročnější operace[3], je vhodné vytvořit a spustit nové vlákno, které bude mít tuto operaci na starost. Hlavní vlákno tím pádem může pokračovat v příjmu a interpretaci událostí a uživatel pak nebude zaskočen tím, že aplikace nereaguje. Použitím vláken tak dojde ke zdánlivému zvýšení výkonu[4] počítače, což je příklad vhodného nasazení paralelismu i mimo oblast skutečně paralelního počítání, jak to bylo řečeno hned zkraje článku. Aplikace bude reagovat svižně i při provádění časově náročnějšího úkolu, systém totiž bude obě vlákna dostatečně rychle přepínat. V případě nepoužití vláken by byla další událost zaslaná aplikaci systémem zpracována až po skončení vybrané operace.

Pokud bychom ale chtěli na jednoprocesorovém systému paralelizovat například náročný výpočet tím, že ho rozdělíme do dvou vláken, ničím si samozřejmě nepomůžeme. Výsledný čas se spíše ještě prodlouží o režii vzniklou dalším vláknem. **Obecně nemá cenu na (matematický) výpočet nasadit více vláken, než máme k dispozici procesorů**.

Každé vlákno je v rámci systému jednoznačně označeno – má svůj *Thread ID* (TID). Dále má určitou *prioritu*. Čím vyšší priorita je, tím více času vlákno dostává. Všechna vlákna patřící jednomu procesu mají stejný *virtuální adresní prostor*, tudíž všechny globální proměnné programu a jeho zdroje (např. otevřené soubory) jsou přístupné z kteréhokoliv vlákna. Naopak, každé vlákno má svůj vlastní *zásobník* a *registry procesoru*.

Proces obsahuje vždy minimálně jedno vlákno, které je při jeho spuštění vytvořeno systémem. Nově vytvořené vlákno dědí prioritu vlákna, které ho

[3]dost dlouhé na to, aby zdánlivá nečinnost programu znervóznila uživatele, tj. asi dvě sekundy a více

[4]Předpokládáme jednoprocesorový systém, kde vlákna neběží skutečně paralelně.

vytvořilo. Vytvoření vlákna je samozřejmě pro systém mnohem méně náročné než vytvoření nového procesu.

1.2.2 Procesy

Proces si lze představit jako kontejner vláken a systémových zdrojů[5], které jsou vlákny využívány. Každý proces má svůj virtuální adresový prostor, přístupný všem vláknům ale nepřístupný ostatním procesům. Jediným způsobem, jak zařídit přístup ke stejnému kusu paměti z více procesů současně, je zřízení sdílené paměti. Paměť procesu může být systémem po stránkách odkládána na disk a později nahrána zpět do fyzické paměti. Každý proces v systému je jednoznačně označen – má svůj *Process ID* (PID). Při startu procesu systém vždy vytvoří vlákno, ve kterém se spustí kód funkce `main()`[6]. Proces může vytvořit a spustit další proces, ovšem pouze stejného typu (PM, textový v okně, textový celoobrazovkový, ...).

K vytvoření dceřinného procesu se používá API funkce `DosExecPgm()`. Dceřinný proces má oproti ostatním tu výhodu, že dědí některé zdroje vlastněné svým rodičem. Nedědí například paměť. Funkce má následující hlavičku:

```
APIRET DosExecPgm( PCHAR pchProcName,
                   LONG lProcName,
                   ULONG ulExecFlag,
                   PSZ pszArg,
                   PSZ pszEnv,
                   PRESULTCODES prcRes,
                   PSZ pszName );
```

Funkce vrací 0, pokud spuštění proběhlo úspěšně. Význam parametrů je vysvětlen v následujícím seznamu:

- `pchProcName` je buffer, kam bude uložena informace, pokud spuštění selže.
- `lProcName` je délka bufferu `pchProcName` v bajtech.
- `ulExecFlag` určuje mód, v kterém bude dceřinný proces spuštěn. Nejznámější jsou `EXEC_ASYNC` (oba procesy poběží souběžně) a `EXEC_SYNC`

[5] semaforů, handlů souborů, paměti, ...

[6] Použijeme-li jazyk C.

(rodičovský proces bude čekat na dokončení dceřinného procesu), ale existují ještě další: `EXEC_ASYNCRESULT`, `EXEC_TRACE`, `EXEC_BACKGROUND`, `EXEC_LOAD` a `EXEC_ASYNCRESULTDB`.

- `pszArg` určuje řetězec, který proces dostane coby parametr `argv` funkce `main()`, tj. jméno programu a parametry příkazové řádky. Každý údaj je ukončen nulovým znakem ('\0'), na konci řetězce je ještě jeden navíc coby ukončení řetězce dle standardu jazyka C.

- `pszEnv` udává proměnné prostředí, které proces dostane. Řetězec má formát stejný s předchozím argumentem. Pokud je zadáno `NULL`, proces zdědí současné prostředí.

- `prcRes` je ukazatel na strukturu `RESULTCODES`, kam je uložen ukončovací kód (termination code) a návratový kód (return code) procesu.

- `pszName` je plně kvalifikované jméno programu pro spuštění. Pokud je program v aktuálním adresáři anebo v adresáři, kam ukazuje systémová proměnná `PATH`, není třeba uvádět plně kvalifikované jméno.

Běžící proces je možno zabít vyvoláním API funkce `DosKillProcess()`. Funkce má následující hlavičku:

```
APIRET DosKillProcess( ULONG ulAction,
                       PID pid );
```

Funkce vrací 0, pokud spuštění proběhlo úspěšně. Význam parametrů je vysvětlen v následujícím seznamu:

- `ulAction` udává, zda se má zabít pouze jeden process (`DKP_PROCESS`) anebo proces i s jeho dceřinnými procesy (`DKP_PROCESSTREE`). Ve druhém případě musí proces zabíjet pouze sám sebe anebo proces, který vytvořil pomocí `DosExecPgm()` s `ulExecFlag` nastaveným na `EXEC_ASYNCRESULT`.

- `pid` udává PID zabíjeného procesu.

Zabíjeným procesům je systémem zaslána výjimka `KILLPROCESS`, což implicitně způsobí zapsání obsahu všech souborů a uzavření všech zdrojů. Procesy ovšem mohou nastavit vlastní obsluhu výjimky pomocí API

funkce `DosSetExceptionHandler()` a upravit tak své chování při násilném ukončení. Pak ovšem musejí samy zavolat `DosExit()`, aby se proces skutečně ukončil.

Na ukončení asynchronně spuštěného dceřinného procesu je možno počkat vyvoláním API funkce `DosWaitChild()`. Více se v tomto článku věnovaném vláknům nebudeme procesům věnovat.

1.2.3 Relace

Relace je zdánlivě podobná procesu, ovšem vlastní přístup ke klávesnici, myši a grafickému výstupu. Relace může obsahovat jeden nebo více procesů. Seznam běžících relací se v systému OS/2 vyvolá stiskem kláves `Ctrl-Esc`. Klávesnice, myš a obrazovka jsou vlastněny pouze relací v popředí, což lze změnit pouze přepnutím relace.

Nová relace může být spuštěna vyvoláním API funkce `DosStartSession()`. Již existující relaci lze ukončit vyvoláním `DosStopSession()`. Přepnutí do jiné relace lze provést vyvoláním `DosSelectSession()` a nastavení parametrů relace pomocí `DosSetSession()`. `DosStartSession()` je nutné použít namísto `DosExecPgm()` (viz kapitola 1.2.2) tehdy, je-li spouštěný program jiného typu než program, který ho spuští. Dostupné typy relací jsou uvedeny v následujícím seznamu:

- Presentation Manager – konstanta `SSF_TYPE_PM`
- okno OS/2 – konstanta `SSF_TYPE_WINDOWABLEVIO`
- celá obrazovka OS/2 – konstanta `SSF_TYPE_FULLSCREEN`
- okno DOS – konstanta `SSF_TYPE_WINDOWEDVDM`
- celá obrazovka DOS – konstanta `SSF_TYPE_VDM`
- stejný typ jako volající program – konstanta `SSF_TYPE_DEFAULT`

Spuštíme-li program určený pro Windows 3.1, lze vybrat ještě z dalších šesti typů relace, kterým odpovídají konstanty `PROG_31_*`. Konstanty se používají pro vyplnění části struktury `STARTDATA`, která se dává funkci `DosStartSession()` jako její první parametr. Alternativně lze typ spouštěné relace určit funkcí `DosQueryAppType()`. Relacemi se v tomto článku již nebudeme zabývat neboť s problematikou multithreadingu téměř nesouvisejí.

1.3 Plánovač OS/2

Jak již bylo řečeno, plánovač OS/2 pracuje pouze s vlákny, nikoliv s procesy nebo relacemi. Vlákna jsou rozčleněna do **čtyř hlavních úrovní priorit**: priority *časově kritických* vláken (`PRTYC_TIMECRITICAL`), priority *serverových* vláken (`PRTYC_FOREGROUNDSERVER`), priority *normálních* vláken (`PRTYC_REGULAR`) a priority vláken, na *kterých nezáleží* (`PRTYC_IDLETIME`). **Každá z těchto úrovní má 32 podúrovní.** Vlákna s nižší prioritou jsou plánovačem spuštěna až tehdy, pokud žádné vlákno s vyšší prioritou nemá potřebu běžet. Vláknu je odňat procesor například tehdy, když zavolá nějakou blokující funkci, např. `DosSleep()`, když se pokusí vstoupit do oblasti chráněné semaforem a semafor je již zavřený, když se pokusí vybrat událost ke zpracování z prázdné fronty, když mu vyprší časové kvantum, ...

Dle mých zjištění, prováděných s pomocí programu infoPM, trvá časové kvantum pod OS/2 verze 4.5 32 milisekund. Minimální i maximální hodnota časového kvanta je totiž nastavena právě na tuto hodnotu.

Plánovač může jednotlivým vláknům jejich priority upravovat. Vlákna od něj dostávají tzv. zesílení priority (boost), které může být tří typů:

- **zesílení na popředí** (Foregroud Boost) – Toto zesílení je přiděleno vláknu procesu právě běžícího v popředí, které zpracovává události od systému. To je obvykle hlavní (první) vlákno procesu. Tím se pomáhá lepšímu odebírání událostí z fronty. Toto zesílení mění úroveň priority – tzv. full boost. Pokud obsluhuje vlákno Presentation Manageru zprávu, dostává také toto zesílení.

- **zesílení V/V** (I/O Boost) – Toto zesílení je přiděleno vláknu, které právě dokončilo V/V operaci. Nemění se úroveň priority, pouze podúroveň se nastaví na 31 (nejvyšší hodnotu).

- **zesílení po hladovění** (Starvation Boost) – Toto zesílení je přiděleno vláknu, které je na normální úrovni priority a nemohlo běžet nejméně po dobu 3 sekund. Tuto hodnotu lze upravit nastavením `MAXWAIT` v souboru `CONFIG.SYS`.

2 Užitečné funkce OS/2 API

V této kapitole budou představeny API funkce pro práci s vlákny a semafory. U každé funkce bude uvedena její hlavička a bude vysvětlen význam jejích parametrů. Dále budou uvedeny možné návratové hodnoty a někdy i popis chování, pokud nebude zřejmý.

Chceme-li dále představené funkce používat, nesmíme zapomenout vložit na začátek našeho zdrojového kódu následující řádky:

```
#define INCL_DOSERRORS
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES

#include <os2.h>
```

První tři řádky říkají kompilátoru, že budeme používat API funkce pro práci s procesy a semafory a chceme mít přístup k chybovým konstantám. Poslední řádek vloží standardní hlavičkový soubor `os2.h`, který dále vloží hlavičkový soubor `bse.h`[7], a ten konečně vloží `bsedos.h` a `bseerr.h`. Obsah dvou posledních souborů je preprocesorem upraven v závislosti na (ne)existenci konstant `INCL_*` a přebytečné části nejsou do výsledného zdrojového textu vloženy.

2.1 Práce s vlákny

V této části uvedeme funkce pro vytvoření a zabití vlákna, změnu jeho priority, vstup a výstup do/z kritické sekce a pozastavení a znovuspuštění vlákna.

Při použití těchto funkcí nezapomeňte překladači oznámit, že vytváříte vícevláknový program. Překladač pak použije jiné verze některých knihoven, které jsou napsány s ohledem na použití v paralelním prostředí. U překladače `gcc.exe` z balíku EMX je to volba `-Zmt`, u překladače Open Watcom (`wcc386.exe`) je to volba `-bm`, kterou lze nastavit i z integrovaného vývojového prostředí po vybrání *Options → C Compiler Switches → 7. Code-Gen Strategy Switches*.

[7] S nemocí šílených krav nemá tento soubor nic společného. Zkratka `bse` je od slova *base* – základ.

2.1.1 Vytvoření vlákna

Pro vytvoření a případně i okamžité spuštění vlákna se používá funkce `DosCreateThread()`:

```
APIRET APIENTRY DosCreateThread( PTID ptidThreadID,
                                 PFNTHREAD pfnThreadFunction,
                                 ULONG ulParameter,
                                 ULONG ulThreadState,
                                 ULONG ulStackSize );
```

Význam parametrů je následující:

- `ptidThreadID` je ukazatel na `TID`, kam se uloží *Thread ID* nově vzniklého vlákna. Pozor, předávat přímo proměnnou typu `PTID` lze leda tehdy, když ukazuje na alokovanou paměť typu `TID`. Nejlepší je ale předávat adresu staticky alokované proměnné `TID`, případně přetypovanou na `PTID`.
- `pfnThreadFunction` je adresa funkce - programu vlákna. Tato funkce musí mít přesně definovanou hlavičku:

  ```
  VOID APIENTRY fnThreadFunction( ULONG ulThreadArgs );
  ```

- `ulParameter` je skutečný parametr, předávaný vláknu. Obvykle se jedná o ukazatel na nějakou uživatelskou strukturu, přetypovaný na `ULONG`. Vlákno pak musí tento prarametr přetypovat zpět na očekávaný typ.
- `ulThreadState` udává, zda bude vlákno ihned spuštěno (`0L`) anebo bude pasivně čekat na pozdější spuštění (`1L`) funkcí `DosResumeThread()`, viz kapitola 2.1.4.
- `ulStackSize` specifikuje velikost zásobníku vlákna v bajtech. Paměť pro zásobník alokuje sám systém.

Funkce vrací nulu v případě úspěchu, jinak vrací nenulové číslo. Přesné chybové konstanty se mi bohužel nepodařilo zjistit.

Některé kompilátory mohou mít problémy s použitím `DosCreateThread()`, protože si nejsou vědomy přítomnosti více vláken a jejich knihovní funkce pak nepracují vždy korektně, pokud jsou volány z jiného než hlavního vlákna. Lze tudíž nalézt i alternativní funkci `_beginthread()` pro vytvoření vlákna:

```
int _beginthread( void *(*start_address)(void *),
                  void *stack_bottom,
                  unsigned stack_size,
                  void *arglist );
```

Význam parametrů je následující:

- `start_address` je ukazatel na funkci - program vlákna. Funkce musí vracet ukazatel na `void` a mít jeden parametr – ukazatel na `void`.
- `stack_bottom` udává počátek paměti alokované pro zásobník vlákna. Pokud je `NULL`, paměť alokuje sám systém.
- `stack_size` udává velikost zásobníku.
- `arglist` je vlastní parametr, který bude předaný vláknu. Opět je to většinou ukazatel na uživatelskou strukturu přetypovaný na `(void *)`, který musí vlákno přetypovat zpět na očekávaný typ.

Funkce vrací v případě úspěchu handle na nově vytvořené vlákno, jinak vrací -1.

2.1.2 Ukončení vlákna

Vlákno se ukončí samo, pokud dojde na konec svého programu (funkce specifikované při volání `DosCreateThread()`) anebo ho opustí příkazem `return`.

K ukončení je taktéž možné použít funkci `DosExit()`, která umí ukončit jak vlákno, tak i celý proces:

```
VOID APIENTRY DosExit( ULONG ulAction,
                       ULONG ulResult );
```

Význam parametrů je následující:

- `ulAction` udává, zda ukončit pouze aktuální vlákno (`EXIT_THREAD`) nebo celý proces (`EXIT_PROCESS`).
- `ulResult` je návratový kód programu, který je případně vrácen funkcí `DosWaitChild()`, kterou může jiný proces použít k čekání na ukončení volajícího procesu.

Návratová hodnota není žádná a ani by neměla smysl, protože program se z volání funkce nikdy nevrátí.

Pro ukončení vláken vytvořených pomocí `_beginthread()` je nutno použít funkci `_endthread()`, která nic nevrací a nemá žádné parametry:

```
void _endthread( void );
```

2.1.3 Nastavení priority vlákna

Pro nastavení priority vlákna, procesu nebo celé skupiny procesů (procesu a jeho potomků a jejich potomků atd.) lze použít funkci `DosSetPriority()`:

```
APIRET APIENTRY DosSetPriority( ULONG ulScope,
                                ULONG ulClass,
                                LONG  ulDelta,
                                ULONG ulPorTid );
```

Význam parametrů je následující:

- `ulScope` udává, na co se změna priority vztahuje. Lze zadat `PRTYS_PROCESS` pro změnu priority procesu, `PRTYS_PROCESSTREE` pro změnu priority všech dceřinných procesů (rekurzivně) nebo `PRTYS_THREAD` pro změnu priority vlákna.
- `ulClass` je nová úroveň priority, viz kapitola 1.3. Může nabývat hodnot `PRTYC_NOCHANGE`, `PRTYC_IDLETIME`, `PRTYC_REGULAR`, `PRTYC_TIMECRITICAL` a `PRTYC_FOREGROUNDSERVER`.
- `ulDelta` je změna podúrovně priority, nikoliv absolutní hodnota podúrovně. Lze využít konstant `PRTYD_MINIMUM` (-31) a `PRTYD_MAXIMUM` (+31).
- `ulPorTid` je *Thread ID* vlákna, jemuž měníme prioritu.

Návratové kódy se mi nepodařilo zjistit, ale dle zaběhlých zvyklostí se vrací v případě úspěchu nula, jinak nenulové číslo.

2.1.4 Pozastavení a obnovení běhu vlákna

Vlákno lze pozastavit voláním funkce `DosSuspendThread()` a obnovit jeho běh voláním `DosResumeThread()`. Obě dvě funkce mají jako parametr *Thread ID* vlákna, s kterým se manipuluje. Pozor na to, že pozastavené vlákno musí pomocí `DosResumeThread()` rozeběhnout jiné vlákno. Vlákno nemůže rozeběhnout samo sebe, protože je blokováno a nemá tak šanci funkci zavolat. `DosResumeThread()` použijeme i tehdy, když chceme spustit vlákno vytvořené pomocí `DosCreateThread()` s `ulThreadState == 1L`.

```
APIRET APIENTRY DosResumeThread( TID tid );
APIRET APIENTRY DosSuspendThread( TID tid );
```

Návratové kódy se mi nepodařilo zjistit, ale dle zaběhlých zvyklostí se vrací v případě úspěchu nula, jinak nenulové číslo.

2.1.5 Zabití vlákna

K násilnému ukončení vlákna z jiného vlákna slouží funkce `DosKillThread()`, mající za parametr *Thread ID* vlákna, které má být ukončeno. Obvykle je ale lepší použít 'méně surové' metody ukončení, například nastavit vláknu příznak ukončení (sdílenou proměnnou) a nechat vlákno, aby se samo ukončilo a případně provedlo 'úklidové práce'.

```
APIRET APIENTRY DosKillThread( TID tid );
```

Návratové kódy se mi nepodařilo zjistit, ale dle zaběhlých zvyklostí se vrací v případě úspěchu nula, jinak nenulové číslo.

2.1.6 Čekání na ukončení vlákna

Jedno vlákno se může dočasně pozastavit až do doby, než jiné vlákno skončí. K tomuto účelu slouží funkce `DosWaitThread()`, která mimo to umožňuje bez jakéhokoliv čekání pouze otestovat, jestli nějaké jiné vlákno běží nebo ne.

```
APIRET APIENTRY DosWaitThread( PTID ptidThread,
                               ULONG ulOption );
```

Coby `ulOptions` lze zadat buď `DCWW_WAIT` anebo `DCWW_NOWAIT`. V prvním případě vlákno čeká na dokončení jiného vlákna (pokud to existuje), ve druhém případě nečeká a hned se z volání funkce vrací. Tak lze pomocí návratových kódů (které se mi nepodařilo zjistit) otestovat existenci nebo neexistenci vlákna.

2.1.7 Kritická sekce

Chceme-li si být jisti, že vlákno nebude během nějaké důležité práce (např. testování a zápisu do sdílené proměnné) přerušeno, můžeme použít *kritickou sekci* a část kódu do ní uzavřít. Existují různé výklady, co přesně se děje (tedy spíše neděje), když je nějaké vlákno uvnitř kritické sekci. Pramen [Art] uvádí, že nedojde k přepnutí na ostatní vlákna stejného procesu. Pramen [Games] uvádí, že nedojde k přepnutí v rámci celého systému. Tím by mohl špatně napsaný proces nekonečně blokovat všechny ostatní (i systémové) procesy, včetně těch životně důležitých.

Kritické sekce se doporučuje používat co nejméně, na ochranu dat před současným přístupem z více vláken je lepší využívat semafory (viz kapitola 2.2), které blokují pouze vlákno, které o vstup do chráněné části kódu neúspěšně žádá, ostatní vlákna ponechávají v běhuschopném stavu. Když už je kritická sekce použita, **měla by být co nejkratší**.

Je velmi nebezpečné uvnitř kritické sekce používat semafory jakéhokoliv druhu, tj. volat funkce `DosWaitEventSem()` (viz kapitola 2.3.6) nebo `DosRequestMutexSem()` (viz kapitola 2.2.4) atd. Důvodem je to, že pokud se zde vlákno zastaví, nedojde kvůli přítomnosti v kritické sekci k přepnutí na jiné vlákno procesu, které by mohlo semafor nastavit, respektive uvolnit. Tím pádem dojde k *uvíznutí* (deadlocku).

Vstup do kritické sekce poskytuje funkce `DosEnterCritSec()`, opuštění kritické sekce funkce `DosExitCritSec()`:

```
APIRET APIENTRY DosEnterCritSec( VOID );
APIRET APIENTRY DosExitCritSec( VOID );
```

Návratové kódy se mi nepodařilo zjistit, ale dle zaběhlých zvyklostí se vrací v případě úspěchu nula, jinak nenulové číslo.

Kritické sekce lze rekurzivně vnořovat. Pokud vlákno skončí uvnitř kritické sekce (např. příkazem `return` nebo zapomene kritickou sekci opustit), kritická sekce se automaticky zruší.

2.1.8 Uspání vlákna na dobu určitou

Vlákno lze uspat na určitý čas, zadaný v milisekundách funkci `DosSleep()`:

```
APIRET APIENTRY DosSleep( ULONG msec );
```

Čas, na který bude vlákno skutečně uspáno, se nemusí přesně shodovat se zadaným údajem. Pokud se jako čas zadá nula, vlákno bude uspáno až do konce svého časového kvanta, tj. vlastně dobrovolně přepustí místo na procesoru ostatním vláknům.

Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo `ERROR_TS_WAKEUP` (322).

2.2 Práce s mutex semafory

Slovo *mutex* je zkrácený výraz pro slova *mutual exclusion*, česky *vzájemné vyloučení*. Mutexy se použijí všude tam, kde je nutno provést nějaké operace na datech sdílených více vlákny tak, aby nejvýše jedno vlákno mohlo s daty v daný čas manipulovat a další vlákna k nim měla přístup až tehdy, když první vlákno svoji práci dokončí.

Představme si dvě vlákna, každé z nich provádějící následující kód:

```
int i;

čti(i);
i = i+1;
zapiš(i);
```

Bude-li na začátku sdílená proměnná `prom == 0`, po skončení běhu obou vláken by se `prom` měla rovnat 2. Tak to ale nemusí v paralelním prostředí vždy být. Poté, co vlákno č. 1 provede čtení, může být přepnuto na vlákno č. 2, které také provede čtení. Obě dvě vlákna tedy přečetla číslo 0. Poté v nějakém libovolném pořadí vlákna inkrementují svoji lokální proměnnou `i` (na hodnotu 1) a zapíšou ji do sdílené proměnné. V té bude po skončení běhu vláken hodnota 1, což je evidentně špatně.

Uzamčení výše zmíněného kódu problém vyřeší:

```
int i;
```

```
mutex m;

získej_mutex(m);
čti(i);
i = i+1;
zapiš(i);
uvolni_mutex(m);
```

V části chráněné mutexem může sice pořád dojít k přepnutí z vlákna č. 1 na č. 2, vlákno číslo 2 se ale zastaví na volání `získej_mutex(m)` a nebude smět pokračovat, dokud vlákno č. 1 mutex neuvolní. Vlákno č. 2 tak vždy přečte již aktualizovanou hodnotu sdílené proměnné, správně ji inkrementuje a zapíše zpět.

2.2.1 Vytvoření a otevření mutexu

Nový mutex se vytvoří voláním API funkce `DosCreateMutexSem()`. Mutex může být buď pojmenovaný nebo nepojmenovaný. Nepojmenovaný mutex může být veřejný či soukromý, zatímco všechny pojmenované mutexy jsou automaticky veřejné. Veřejný mutex je přístupný i z jiných procesů, soukromý nikoliv. Všechny mutexy jsou přístupné ze všech vláken procesu, ve kterém byly vytvořeny.

```
APIRET APIENTRY DosCreateMutexSem( PSZ pszName,
                                   PHMTX phmtx,
                                   ULONG flAttr,
                                   BOOL32 fState );
```

Význam parametrů je následující:

- `pszName` je řetězec ukončený nulou, udávající jméno mutexu. Musí začínat prefixem `\SEM32\`, musí odpovídat konvencím souborového systému a nesmí být delší než 255 znaků. Pro nepojmenované mutexy se zadá `NULL`.
- `phmtx` je ukazatel na handle mutexu, který bude voláním navrácen. Nejlepší je použít adresu staticky alokovaného handlu, nepoužívejte proměnnou typu `PHMTX`, která by neukazovala na předem alokovanou paměť!

- `flAttr` udává, zda mutex bude sdílený (`DC_SEM_SHARED`) nebo soukromý (0). Pokud `pszName` není `NULL`, ignoruje se.
- `fState` udává, zda mutex bude už po vytvoření uzamčený (vlastněný tímto vláknem – `TRUE`) anebo odemčený (`FALSE`).

Funkce vrací `NO_ERROR` (0) při úspěšném provedení nebo jeden z následujících chybových kódů: `ERROR_NOT_ENOUGH_MEMORY` (8), `ERROR_INVALID_PARAMETER` (87), `ERROR_INVALID_NAME` (123), `ERROR_DUPLICATE_NAME` (285) a `ERROR_TOO_MANY_HANDLES` (290).

Namísto vytvoření nového mutexu je možno pomocí funkce `DosOpenMutexSem()` získat přístup k již existujícímu veřejnému mutexu, který byl vytvořen jiným procesem.

```
APIRET APIENTRY DosOpenMutexSem( PSZ pszName,
                                 PHMTX phmtx );
```

Význam parametrů je následující:

- `pszName` je jméno semaforu, viz výše. Pokud je `NULL`, bude použita hodnota `phmtx`.
- `phmtx` je určen pro handle mutexu. Pokud je mutex pojmenovaný, musí být při volání funkce `NULL` a po návratu bude ukazovat na handle otevřeného mutexu. Při otevírání nepojmenovaného mutexu musí ukazovat na jeho handle před voláním.

Funkce vrací `NO_ERROR` (0) při úspěšném provedení nebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6), `ERROR_NOT_ENOUGH_MEMORY` (8), `ERROR_INVALID_PARAMETER` (87), `ERROR_SEM_OWNER_DIED` (105), `ERROR_INVALID_NAME` (123), `ERROR_SEM_NOT_FOUND` (187), `ERROR_TOO_MANY_OPENS` (291).

2.2.2 Zavření mutexu

Pokud již nechceme mutex v programu využívat, zavřeme ho vyvoláním API funkce `DosCloseMutexSem()`. Pokud již není (sdílený) semafor využíván jiným procesem, je odstraněn ze systému. Soukromý semafor je voláním `DosCloseMutexSem()` vždy odstraněn, protože na něj existuje pouze jeden odkaz.

```
APIRET APIENTRY DosCloseMutexSem( HMTX hmtx );
```

Parametr `hmtx` udává handle mutexu (nikoliv ukazatel na handle), který se má uzavřít. Funkce vrací `NO_ERROR` (0) při úspěšném provedení nebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6) a `ERROR_SEM_BUSY` (301).

2.2.3 Zjištění stavu mutexu

Pokud chceme zjistit, zda je mutex přivlastněný nějakému vláknu (tj. zda ho nějaké vlákno získalo a doposud ho neuvolnilo), použijeme API funkci `DosQueryMutexSem()`. Tato funkce navrátí *Process ID* a *Thread ID* vlastníka mutexu a počet vláken, která si na mutex dělají nárok (včetně jeho současného vlastníka), tj. počet vláken, která na tento mutex zavolala `DosRequestMutexSem()` a ještě ho neuvolnila pomocí `DosReleaseMutexSem()`. Před použitím této funkce je nezbytné mutex otevřít buď pomocí `DosCreateMutexSem()` anebo `DosOpenMutexSem()`.

```
APIRET APIENTRY DosQueryMutexSem( HMTX hmtx,
                                  PID *ppid,
                                  TID *ptid,
                                  PULONG pulCount );
```

Význam parametrů je následující:

- `hmtx` je handle mutexu, na který se dotazujeme.
- `ppid` je ukazatel na proměnnou typu `PID`, do které bude uložen *Process ID* vlastníka mutexu.
- `ptid` je ukazatel na proměnnou typu `TID`, do které bude uložen *Thread ID* vlastníka mutexu.
- `pulCount` je ukazatel na proměnnou typu `ULONG`, do které bude uložen počet vláken, která si nárokují tento mutex. Pokud je mutex otevřený (nepatří žádnému vláknu), je zde vrácena nula.

Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6), `ERROR_INVALID_PARAMETER` (87) a `ERROR_SEM_OWNER_DIED` (105).

2.2.4 Získání mutexu

Pro vstup vlákna do oblasti chráněné mutexem (tj. pro získání mutexu do vlastnictví) slouží API funkce `DosRequestMutexSem()`. Jeden mutex může v jeden čas vlastnit nejvýše jedno vlákno. Lze specifikovat maximální čas, který může vlákno na získání mutexu čekat. Pokud ani po vypršení tohoto času vlákno mutex nezíská (protože jiné vlákno ho do té doby neuvolní), funkce navrátí chybový kód. Pokud je specifikován nekonečně dlouhý čas, vlákno čeká tak dlouho, dokud se mutex nestane volným a ono ho získá.

Pokud chce více vláken získat stejný mutex, rozhodují jejich priority. Vlákno s vyšší prioritou má při získání přednost před vláknem s nižší prioritou. Pokud jsou priority vláken stejné, dostanou vlákna vlastnictví mutexu dle principu *FIFO – First In, First Out.*

```
APIRET APIENTRY DosRequestMutexSem( HMTX hmtx,
                                    ULONG ulTimeout );
```

Význam parametrů je následující:

- `hmtx` je handle semaforu, který se vlákno pokouší získat.
- `ulTimeout` je maximální čas v milisekundách, po který je vlákno ochotno na vlastnictví mutexu čekat. Lze použít konstanty `SEM_IMMEDIATE_RETURN` pro okamžitý návrat (ať se akce povede nebo ne) nebo `SEM_INDEFINITE_WAIT` pro čekání bez limitu.

Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6), `ERROR_INTERRUPT` (95), `ERROR_TOO_MANY_SEM_REQUESTS` (103), `ERROR_SEM_OWNER_DIED` (105) a `ERROR_TIMEOUT` (640).

Poznámka: V aplikacích pro Presenation Manager se místo funkce `DosRequestMutexSem()` používá funkce `WinRequestMutexSem()`, která zabraňuje zablokování zpracování událostí z fronty událostí Presentation Manageru.

2.2.5 Uvolnění mutexu

Uvolnění mutexu je opačná operace k jeho získání. Uvolněním mutexu vlákno opouští oblast kódu, který je mutexem chráněn, a umožňuje

ostatním vláknům, aby mutex pomocí `DosRequestMutexSem()` získala. Uvolnění mutexu se provádí voláním API funkce `DosReleaseMutexSem()`. Pokud nějaké vlákno čeká v blokovaném stavu na získání mutexu (uvnitř `DosRequestMutexSem()`), je automaticky systémem převedeno do běhuschopného stavu a mutex je mu přidělen. Pokud je čekajících vláken víc, pouze jedno může získat vlastnictví a pokračovat, viz 2.2.4.

```
APIRET APIENTRY DosReleaseMutexSem( HMTX hmtx );
```

Parametr `hmtx` udává handle uvolňovaného mutexu. Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6) a `ERROR_NOT_OWNER` (288).

2.3 Práce s event semafory

Na rozdíl od mutex semaforů event semafory **neobklopují část kódu, jsou bodové**. *Event* znamená v angličtině *událost*, což už předznamenává jejich použití. Event semafory slouží **k pozastavení vlákna a jeho opětovnému spuštění** poté, co mu jiné vlákno pošle přes event semafor zprávu, typicky proto, že **nastala nějaká událost**, díky které může pozastavené vlákno pokračovat.

Každý event semafor se může nacházet ve dvou stavech: *nastaveném*[8] (posted) a *nenastaveném* neboli *vynulovaném* (reset). Pokud je semafor nastavený, vlákno na něm nečeká. Pokud je nenastavený (vynulovaný), čeká na něm vlákno do té doby, než ho nějaké jiné vlákno nastaví. Jedná se tedy o celkem přesnou obdobu železničních semaforů, kde čekací vlákno je vlak a nastavovací vlákno je nějaký železničář ve stanici. Nastavení eventu (posting) lze provádět i automaticky v periodických časových intervalech, pokud se využijí systémové časovače.

2.3.1 Vytvoření a otevření eventu

Nový event se vytvoří voláním API funkce `DosCreateEventSem()`. Stejně jako mutex může být event buď pojmenovaný nebo nepojmenovaný. Nepojmenovaný event může být veřejný či soukromý, zatímco všechny pojmenované eventy jsou automaticky veřejné. Veřejný event je přístupný i z jiných

[8] Oficiální český překlad mi není znám a lepší slova jsem pro anglické ekvivalenty nenalezl.

procesů, soukromý nikoliv. Všechny eventy jsou přístupné ze všech vláken procesu, ve kterém byly vytvořeny.

```
APIRET APIENTRY DosCreateEventSem( PSZ pszName,
                                   PHEV phev,
                                   ULONG flAttr,
                                   BOOL32 fState );
```

Význam parametrů je následující:

- `pszName` je řetězec ukončený nulou, udávající jméno eventu. Musí začínat prefixem `\SEM32\`, musí odpovídat konvencím souborového systému a nesmí být delší než 255 znaků. Pro nepojmenované eventy se zadá `NULL`.
- `phev` je ukazatel na handle eventu, který bude voláním navrácen. Nejlepší je použít adresu staticky alokovaného handlu, nepoužívejte proměnnou typu `PHEV`, která by neukazovala na předem alokovanou paměť!
- `flAttr` udává, zda event bude sdílený (`DC_SEM_SHARED`) nebo soukromý (0). Pokud `pszName` není `NULL`, ignoruje se.
- `fState` udává, zda event bude už po vytvoření nastavený (nebude blokovat – `TRUE`) anebo nenastavený (bude blokovat – `FALSE`).

Funkce vrací `NO_ERROR` (0) při úspěšném provedení nebo jeden z následujících chybových kódů: `ERROR_NOT_ENOUGH_MEMORY` (8), `ERROR_INVALID_PARAMETER` (87), `ERROR_INVALID_NAME` (123), `ERROR_DUPLICATE_NAME` (285) a `ERROR_TOO_MANY_HANDLES` (290).

Namísto vytvoření nového eventu je možno pomocí funkce `DosOpenEventSem()` získat přístup k již existujícímu veřejnému eventu, který byl vytvořen jiným procesem.

```
APIRET APIENTRY DosOpenEventSem( PSZ pszName,
                                 PHEV phev );
```

Význam parametrů je následující:

- `pszName` je jméno semaforu, viz výše. Pokud je `NULL`, bude použita hodnota `phev`.

- `phev` je určen pro handle eventu. Pokud je event pojmenovaný, musí být při volání funkce `NULL` a po návratu bude ukazovat na handle otevřeného eventu. Při otevírání nepojmenovaného eventu musí ukazovat na jeho handle před voláním.

Funkce vrací `NO_ERROR` (0) při úspěšném provedení nebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6), `ERROR_NOT_ENOUGH_MEMORY` (8), `ERROR_INVALID_PARAMETER` (87), `ERROR_INVALID_NAME` (123), `ERROR_SEM_NOT_FOUND` (187), `ERROR_TOO_MANY_OPENS` (291).

2.3.2 Zavření eventu

Pokud již nechceme event v programu využívat, musíme ho stejně jako v případě mutex semaforů zavřít. Event se zavře vyvoláním API funkce `DosCloseEventSem()`. Pokud již není (sdílený) semafor využíván jiným procesem, je odstraněn ze systému. Soukromý semafor je voláním `DosCloseEventSem()` vždy odstraněn, protože na něj existuje pouze jeden odkaz.

```
APIRET APIENTRY DosCloseEventSem( HEV hev );
```

Parametr `hev` udává handle eventu (nikoliv ukazatel na handle), který se má uzavřít. Funkce vrací `NO_ERROR` (0) při úspěšném provedení nebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6) a `ERROR_SEM_BUSY` (301).

2.3.3 Zjištění stavu eventu

Chceme-li zjistit stav eventu, tj. zda je nastavený nebo nenastavený, popřípadě kolikrát je nastavený, použijeme API funkci `DosQueryEventSem()`. Funkce je o poznání chudší než její protějšek `DosQueryMutexSem()`, což je dáno tím, že event semafor nemá žádného vlastníka. Nastavit ho může v jakémkoliv pořadí libovolný počet vláken. Funkce zjistí, kolikrát byl event nastaven od doby posledního resetu (vyčištění nastavení). Před použitím této funkce je nezbytné mutex otevřít buď pomocí `DosCreateEventSem()` anebo `DosOpenEventSem()`.

```
APIRET APIENTRY DosQueryEventSem( HEV hev,
                                  PULONG pulPostCt );
```

Význam parametrů je následující:

- `hev` je handle eventu, na který se dotazujeme.
- `pulPostCt` je ukazatel na proměnnou typu `ULONG`, do které bude uložen počet nastavení (viz kapitola 2.3.5) provedených od posledního resetu.

Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6) a `ERROR_INVALID_PARAMETER` (87).

2.3.4 Vynulování eventu

K vynulování nastaveného semaforu se použije API funkce `DosResetMutexSem()`. To způsobí, že všechny následující pokusy o čekání na nastavení eventu (viz kapitola 2.3.6) skutečně volající vlákno pozastaví, a to až do prvního nastavení eventu (2.3.5). Pokud se pokusíme vynulovat už vynulovaný semafor, obdržíme chybu, se semaforem se nic nestane.

```
APIRET APIENTRY DosResetEventSem( HEV hev,
                                  PULONG pulPostCt );
```

Význam parametrů je následující:

- `hev` je handle eventu, který se má vynulovat.
- `pulPostCt` je ukazatel na `ULONG`, do kterého bude uložen počet nastavení eventu od posledního vynulování. `pulPostCt` nesmí být `NULL`, musí to být platný ukazatel na `ULONG`!

Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6) a `ERROR_ALREADY_RESET` (300).

2.3.5 Nastavení eventu

K nastavení eventu a tím i k rozeběhnutí případně čekajících vláken se používí API funkce `DosPostEventSem()`. Každý event má interní čítač, který se tímto voláním zvýší o jednu. Obvykle se používá jen do hodnoty jedna,

protože čekající vlákno event okamžitě po probuzení vynuluje. Pokud je event nastaven více než jednou mezi dvěma následujícími nulováními, je vrácena chyba, ale i přesto je čítač zvětšen. Maximální hodnota čítače je 65535.

```
APIRET APIENTRY DosPostEventSem( HEV hev );
```

Parametr `hev` určuje handle eventu, který se má nastavit. Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6), `ERROR_TOO_MANY_POSTS` (298) a `ERROR_ALREADY_POSTED` (299).

2.3.6 Čekání na nastavení eventu

Event semafory se typicky použijí pro uspání vlákna, dokud nenastane nějaká událost a event semafor není nastaven (posted). K tomuto uspání, čekání a vzbuzení[9] slouží API funkce `DosWaitEventSem()`. Pokud je event již nastaven, vlákno se z funkce hned navrátí, v opačném případě tam zůstane blokováno, dokud event nějaké jiné vlákno nenastaví pomocí `DosPostEventSem()`. Volitelně lze uvést i časový limit, který nemá být při čekání překročen.

```
APIRET APIENTRY DosWaitEventSem( HEV hev,
                                 ULONG ulTimeout );
```

Význam parametrů je následující:

- `hev` je handle eventu, na jehož nastavení bude vlákno čekat.
- `ulTimeout` udává časový limit – počet milisekund, který nebude při čekání překročen. Lze využít konstant `SEM_IMMEDIATE_RETURN` pro okamžitý návrat a `SEM_INDEFINITE_WAIT` pro neomezené čekání. Typicky se používá právě druhá možnost.

Funkce vrací `NO_ERROR` (0) při úspěšném provedení anebo jeden z následujících chybových kódů: `ERROR_INVALID_HANDLE` (6), `ERROR_NOT_ENOUGH_MEMORY` (8), `ERROR_INTERRUPT` (95) a `ERROR_TIMEOUT` (640).

I tato funkce má svého náhradníka pro aplikace Presentation Manageru. Je jím funkce `WinWaitEventSem()`, která zabrání blokování vlákna zpracovávajícího události ze systémové fronty událostí.

[9] Tři v jednom, no neberte to!

2.4 Mux-Wait semafory

Mux-wait semafory bohužel nebudou v tomto článku popsány[10], protože se mi o nich nepodařilo sehnat dostatek informací. V podstatě lze ale říci, že jeden mux-wait semafor je množina semaforů předešlých dvou typů, do které lze jednotlivé semafory přidávat a též je z ní lze ubírat.

Při práci s mux-wait semafory se používají tyto API funkce:

- `DosCreateMuxWaitSem()`
- `DosAddMuxWaitSem()`
- `DosOpenMuxWaitSem()`
- `DosCloseMuxWaitSem()`
- `DosDeleteMuxWaitSem()`
- `DosQueryMuxWaitSem()`
- `DosWaitMuxWaitSem()`
- `WinWaitMuxWaitSem()`

[10] :-(((

3 Problém večeřících filozofů

Abychom předvedli využití výše popsaných funcí v praxi, ukážeme si v této kapitole, jak pomocí nich naprogramovat jednoduchý vícevláknový program – řešení problému večeřících[11] filozofů, v angličtině označovaných jako Dining Philosophers. Pro srovnání také uvedeme, jak tento problém vyřešit pomocí vláken standardu POSIX a v jazyce Java.

3.1 Popis problému

Problém představil na začátku sedmdesátých let minulého století světoznámý počítačový vědec E. Dijkstra. Okolo kulatého stolu sedí n filozofů, uprostřed stolu je umístěna mísa se špagetami. Filozofové tráví veškerý čas pojídáním špaget a přemýšlením. Chvilku jí špagety, poté přemýšlejí, poté zasí jí špagety atd. K jídlu potřebuje každý filozof dvě vidličky. Vidliček je na stole stejný počet jako filozofů (n), mezi dvěma filozofy je umístěna právě jedna vidlička. Situace je znázorněna na obrázku 1. Pro přehlednost je každý filozof očíslován, stejně tak vidličky.

Chce-li filozof jíst, musí tedy mít obě vidličky, jednu zleva a jednu zprava. To se mu povede pouze tehdy, když jsou obě volné, tj. jeho sousedé je nemají v ruce. V opačném případě musí filozof čekat, až bude jedna nebo obě vidličky dostupné a poté může začíst jíst.

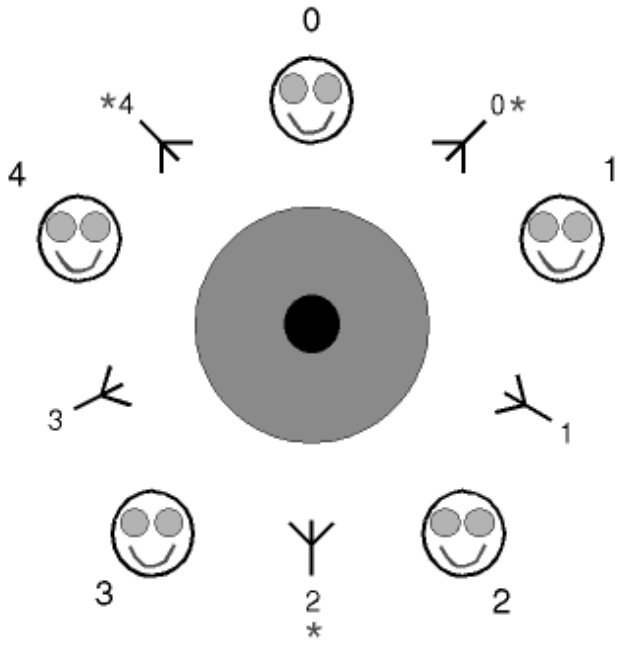
3.1.1 Uváznutí a jeho odstranění

Program života jednoho filozofa můžeme tedy naprogramovat následovně: chvíli bude přemýšlet, pak se pokusí jednu vidličku – například levou – a nakonec druhou vidličku – například pravou. Nebude-li nějaká vidlička volná, musí filozof počkat na její uvolnění, což lze lehce zařídit, například použitím mutex semaforu. Když však necháme takový program běžet dostatečně dlouhou dobu, zjistíme, že se v určitém momentě zcela zablokuje a nepokračuje dál. Co se to děje?

Uvažujme následující posloupnost akcí:

1. Všichni filozofové přemýšlejí.
2. Filozof č. 0 uchopí svoji první (levou) vidličku, která je volná.

[11] Nebo obědvajících, jak chcete...



Obrázek 1: Večeřící filozofové

3. Dojde k přepnutí na vlákno filozofa č. 1.
4. Filozof č. 1 uchopí svoji první (levou) vidličku, která je volná.
5. Dojde k přepnutí na vlákno filozofa č. 2.
6. Filozof č. 2 uchopí svoji první (levou) vidličku, která je volná.
7. Dojde k přepnutí na vlákno filozofa č. 3.
8. Filozof č. 3 uchopí svoji první (levou) vidličku, která je volná.
9. Dojde k přepnutí na vlákno filozofa č. 4.
10. Filozof č. 4 uchopí svoji první (levou) vidličku, která je volná.

Nyní mají všichni levou vidličkou, ale nikdo nemůže získat pravou. Pravá vidlička je totiž zároveň levou vidličkou souseda po pravé straně. Ten ji ale neuvolní, protože ještě nedojedl, tedy spíše ještě ani nezačal jíst,

protože nemůže získat svoji pravou vidličku, která je zároveň levou vidličkou jeho pravého souseda. Takto lze pokračovat, až se okolo stolu dojde opět k prvnímu filozofovi. Takto vzniklá situace se nazývá *uvíznutí*, anglicky *deadlock*. V tomto případě se lze do stejné situace dostat i jinou cestou než výše popsanou, to však na problému nic nemění.

V obecném případě může deadlock vzniknout tehdy, jestliže několik paralelních procesů sdílí několik zdrojů a jeden zdroj může být v jednom čase přidělen maximálně jednomu procesu. Dále musí být splněno, že proces na daný prostředek čeká až do té doby, kdy mu jej systém přidělí, a poté ho může uvolnit pouze on sám, systém mu ho nemůže odebrat. Deadlock vznikne v tom momentě, když:

- proces 1 má ve vlastnictví zdroj 1 a žádá zdroj 2 a zároveň;
- proces 2 má ve vlastnictví zdroj 2 a žádá zdroj 3 a zároveň;
- proces 3 má ve vlastnictví zdroj 3 a žádá zdroj 4 a zároveň;

  atd. až do
- proces $n-1$ má ve vlastnictví zdroj $n-1$ a žádá zdroj n a zároveň;
- proces n má ve vlastnictví zdroj n a žádá zdroj 1.

Tím jsme se dostali do situace, kdy žádný proces nemůže dostat žádaný zdroj, protože všechny zdroje jsou blokované. Zároveň nemůže být žádný zdroj uvolněn (odblokován), protože to může udělat pouze proces, který ho vlastní, a ten je blokovaný čekáním na přidělení jiného zdroje.

Deadlocku je možné se zbavit tím, že ho *detekujeme* a poté zrušíme jeden vybraný proces, čímž dojde k uvolnění jednoho zdroje. Ten bude přidělen jinému procesu, který úspěšně dokončí požadovanou činnost (za předpokladu, že nepotřebuje další zdroj, který je přidělen ještě jinému procesu) a poté oba zdroje uvolní, čímž umožní dalšímu procesu alokovat potřebné zdroje a provést požadovanou činnost a uvolnit zdroje atd. Deadlock lze detekovat např. použitím tzv. *alokačního grafu* nebo použitím *Bankéřova algoritmu*.

Místo metod detekčních lze použít metody *ochranné*, které **vznik deadlocku vůbec nepřipustí**. Jednou z několika (cca 4) ochranných metod je *zavedení priorit zdrojů*. Každý zdroj bude mít přidělenu svoji prioritu. Jestliže bude proces mít přidělen zdroj s prioritou k, nebude moci žádat o zdroj s prioritou menší než je k. Jinými slovy, musí žádat o přidělení zdrojů v pořadí jejich priorit (vzestupně).

Pokud ve výše uvedeném příkladu přiřadíme každému zdroji prioritu rovnou jeho číslu, odstraníme deadlock zrušením posledního bodu. Proces n totiž bude žádat nejdříve o zdroj 1, až poté o zdroj n. Zdroj 1 nedostane, protože ho má přidělen proces 1, bude na jeho přidělení čekat. Tím pádem bude dostupný zdroj n, který dostane proces $n-1$, který provede požadovaný úkol a uvolní zdroje $n-1$ a n. Tak se může rozeběhnou proces $n-2$, který nakonec uvolní zdroje $n-2$ a $n-1$ atd. Nakonec proces 1 dostane zdroj 2, provede požadovaný úkol a uvolní zdroje 1 a 2. Proces n získá nyní již volné zdroje 1 a n. A je vyhráno! Všechny procesy dostaly všechny potřebné zdroje, provedly všechny úkoly a všechny zdroje jsou nyní volné.

Stejné řešení (ale zde jedno z mnoha) je použito i v tomto ukázkovém příkladě večeřících filozofů. Každá vidlička má svoji prioritu. Vidlička s vyšší prioritou (menším číslem) musí být filozofem žádána dříve než druhá vidlička. Jelikož zde každý filozof žádá vždy dvě stejné vidličky a žádné jiné, lze provést zjednodušení a zavést pouze dvě úrovně priorit namísto n. I tak dojde k přetrhnutí začarovaného kruhu a deadlock nikdy nenastane. V obrázku na straně 28 jsou vidličky s vyšší prioritou označeny hvězdičkou. Filozof č. 0 si může vybrat, zda první šáhne po levé či pravé vidličce, jeho chování není nebezpečné.

3.2 Řešení pomocí OS/2 API

Program je založen práci s vlákny a semafory, které jsme popsali v kapitole 2. Zde popíšeme pouze zajímavé konstrukce, které se týkají interakce vláken a sdílení dat. Ostatní části zdrojového kódu (vytváření vláken a semaforů, jejich rušení, zpracování parametrů příkazové řádky, ...) jsou poměrně nezajímavé. Kompletní zdrojový kód se nachází v archivu `DiningPhilosophers.ZIP` (nebo `.RAR`) v aresáři `Native OS2\Sources`. Program byl překládán kompilátorem Open Watcom.

3.2.1 Reprezentace vidliček

Vidličky jsou chápány jako sdílené zdroje, protože ke každé přistupují paralelně vždy dvě vlákna. Z tohoto důvodu je každá vidlička chráněna vlastním mutex semaforem, který je deklarován přímo v definici typu vidličky spolu s odkazem na aktuálního vlastníka vidličky:

```
typedef struct _FORK
        {
```

```
        INT     iOwner;
        HMTX    hmtxSemaphore;
       }
       FORK;

typedef FORK FORKS[NO_OF_PHILOSOPHERS];
```

3.2.2 Reprezentace filozofů

Každý filozof má své pořadové číslo (viz obrázek) a stav, ve kterém se právě nachází. Možné stavy jsou: myslící (`STATE_THINKING`), čekající na vidličky (`STATE_WAITING`), jedící (`STATE_EATING`), uvolňující vidličky (`STATE_RELEASING`) a mrtvý (`STATE_DEAD`). Stav mrtvý je zde zaveden uměle pouze proto, že každý filozof proběhne svůj životní cyklus pouze pětkrát (lze přepsat konstantu `NO_OF_TURNS`) a pak skončí. Program totiž končí až po ukončení všech filozofů, respektive jejich vláken. Filozof ví, kterou vidličku má po levé a pravé ruce – má indexy `iLeftFork` a `iRightFork` do pole vidliček. To ovšem nestačí, musí též vědět, která z nich má vyšší prioritu. Proto si též uchovává indexy `iFirstFork` a `iSecondFork`. U filozofů se sudým pořadovým číslem platí, že `iFirstFork == iLeftFork` (nejdříve sahají pro levou vidličku) a `iSecondFork == iRightFork`. U filozofů s lichým pořadovým číslem je tomu naopak.

```
typedef struct _PHILOSOPHER
       {
        INT     iID;
        INT     iState;

        INT     iLeftFork;
        INT     iRightFork;

        INT     iFirstFork;
        INT     iSecondFork;
       }
       PHILOSOPHER;

typedef PHILOSOPHER PHILOSOPHERS[NO_OF_PHILOSOPHERS];
```

3.2.3 Program vláken

Program vlákna filozofů je uložen ve funkci `fnPhilosopherThread()`, jejíž hlavička přesně odpovídá vzoru uvedenému v kapitole 2.1.1. Parametr `ulThreadArgs` je používán pro přenos parametrů každému vláknu filozofa – pomocí dvou přetypování (tam a zpět) je vláknu dodán parametr typu `PHILOSOPHER_DATA`, který se pro každé vlákno liší.

```
typedef struct _PHILOSOPHER_DATA
        {
         FORKS          *pForks;
         PHILOSOPHERS   *pPhilosophers;
         UINT           uiMyNumber;
        }
        PHILOSOPHER_DATA;
```

`pForks` a `pPhilosophers` jsou ukazatele na staticky alokovaná pole vidliček a filozofů a pro všechny filozofy jsou stejné. Naopak `uiMyNumber`, udávající pořadové číslo filozofa, je vždy unikátní.

Program vlákna obsahuje `for` cyklus, ve kterém vlákno nejdříve spí po náhodně generovaný počet milisekund (simulace spaní) a poté se pokouší získat obě vidličky, k čemuž používá funkci `DosRequestMutexSem()` volanou zvlášť nad semaforem každé vidličky. Každé volání `DosRequestMutexSem()` může vlákno případně dočasně uspat. Poté, co jsou oba semafory vláknem přivlastněny, se vlákno opět uspí na náhodně dlouhý časový interval (simulace pojídání špaget) a poté oba semafory uvolní.

3.2.4 Změny stavu

Před každou akcí (spaní, jídlo, čekání na vidličky, uvolňování vidliček) se do proměnné `iState` uloží aktuální stav vlákna (jedna z konstant `STATE_*`) a poté se celý stůl vytiskne na obrazovku pomocí `print_forks()`.

```
me->iState = STATE_THINKING;
print_forks(*(ppdMyData->pForks), *(ppdMyData->pPhilosophers));
```

3.2.5 Uspání vláken

Spaní se provádí příkazem `DosSleep()`, který má za parametr náhodné číslo z intervalu 0 až `uiTimeToThink`, respektive `uiTimeToEat`. Tyto proměnné

jsou naplněny hodnotami zadanými na příkazové řádce a udávají maximální počet milisekund, po který může filozof přemýšlet a jíst.

```
DosSleep(rand() % uiTimeToThink);
```

3.2.6 Čekání na vidličky

V případě chyby při čekání na přivlastnění semaforu vidličky (která by zde ale neměla nikdy nastat) je vypsána chybová zpráva a vlákno je ukončeno.

```
arReturn1 = DosRequestMutexSem(
 ((*(ppdMyData->pForks))[me->iFirstFork]).hmtxSemaphore,
 SEM_INDEFINITE_WAIT);
if (arReturn1 != 0)
{
 fprintf(stderr,
  "\nCannot get a mutex semaphore, return code = %d\n",
  arReturn1);
 fprintf(stderr, "Thread terminated.");
 me->iState = STATE_DEAD;
 return;
}
```

Po každém vzetí vidličky do ruky filozof vidličce aktualizuje vlastníka tím, že do `iOwner` dosadí svoje ID, a celý stůl (filozofové obklopeni vidličkami) je vypsán na obrazovku.

```
((*(ppdMyData->pForks))[me->iFirstFork]).iOwner = me->iID;
print_forks(*(ppdMyData->pForks), *(ppdMyData->pPhilosophers));
```

3.2.7 Uvolnění vidliček

Položení vidličky na stůl se provede vynulováním jejího vlastníka a uvolněním jejího semaforu, takže ji může získat vedle sedící filozof. Také po každém položení vidličky jsou vidličky a filozofové vypsáni na obrazovku.

```
((*(ppdMyData->pForks))[me->iSecondFork]).iOwner = NOBODY;
arReturn2 = DosReleaseMutexSem(
```

```
 ((*(ppdMyData->pForks))[me->iSecondFork]).hmtxSemaphore);
if (arReturn2 != 0)
{
 fprintf(stderr,
  "\nCannot release a mutex semaphore, return code = %d\n",
  arReturn2);
 fprintf(stderr, "Thread terminated.");
 me->iState = STATE_DEAD;
 return;
}
print_forks(*(ppdMyData->pForks), *(ppdMyData->pPhilosophers));
```

3.2.8 Synchronizace s hlavním vláknem

Jelikož hlavní vlákno programu čeká na ukončení všech vláken voláním funkce `DosWaitEventSem()` nad sdíleným event semaforem `hevBlockMainThreadSemaphore`, je třeba, aby poslední vlákno těsně před svým ukončením hlavní vlákno vzbudilo pomocí `DosPostEventSem()`. Vlákno pozná, že je poslední, když hodnota sdílené proměnné `iNumberOfAlivePhilosophers` klesne po odečtení jedné na nulu. Odečtení provádí každé vlákno před ukončením, pouze to poslední ale budí hlavní vlákno.

```
iNumberOfAlivePhilosophers--;
if (iNumberOfAlivePhilosophers == 0)
{
 arReturn2 = DosPostEventSem(hevBlockMainThreadSemaphore);
 if (arReturn2 != 0)
 {
  fprintf(stderr,
   "\nCannot post an event semaphore, return code = %d\n",
   arReturn2);
  fprintf(stderr, "Deadlock possible.");
 }
}
```

Samotná proměnná `iNumberOfAlivePhilosophers` je také chráněna mutex semaforem, protože je zde teoretické nebezpečí zápisu dvěma vlákny současně.

Hlavní vlákno čeká tímto příkazem:

```
arReturn = DosWaitEventSem(
 hevBlockMainThreadSemaphore, SEM_INDEFINITE_WAIT);
if (arReturn != 0)
{
 fprintf(stderr,
  "\nCannot wait for an event semaphore, return code = %d\n",
  arReturn);
 fprintf(stderr, "Exiting.");
 return (2);
} /* if arReturn */
```

3.3 Řešení pomocí vláken standardu POSIX

Řešení pomocí vláken dle standardu POSIX je téměř identické s předchozím řešením. Proto zde budou uvedeny pouze rozdíly, které spočívají v jiných jménech funkcí pro práci s vlákny a semafory. Datové struktury zůstávají stejné, pouze bylo upuštěno od maďarské konvence v pojmenování identifikátorů.

Knihovna posixových vláken je dostupná na serveru Hobbes (`http://hobbes.nmsu.edu`) v podobě knihoven pro balík EMX, jehož částí je i kompilátor `gcc`. Proto byl tento program překládán právě `gcc`. Kompletní zdrojový kód se nachází v archivu `DiningPhilosophers.ZIP` (nebo `.RAR`) v adresáři `POSIX Threads\Sources`.

3.3.1 Vytvoření vláken

Nové vlákno se vytvoří a nastartuje voláním funkce `pthread_create()`. Jako parametry se mu dosadí:

- ukazatel na proměnnou typu `pthread_t`, kde budou navráceny údaje o vytvořeném vláknu;
- požadované atributy vlákna typu `pthread_attr_t`, pokud si vystačíme s předvolenými atributy, dosadíme `NULL`;
- funkce, ve které je definován program vlákna, její hlavička musí odpovídat vzoru

  ```
  void *vlakno(void *data)
  ```

- skutečné parametry předané vláknu, přetypované na ukazatel na `void`.

V případě úspěchu vrací funkce nulu.

Příklad použití:

```
retval = pthread_create(&thread, NULL,
 philosopher_thread, (void *) &philosopher_data);
if (retval != 0)
{
 fprintf(stderr, "Cannot create a thread, rc = %d\n", retval);
 error = 1;
} /* if */
```

3.3.2 Práce s mutex semafory

Mutexy jsou proměnné typu `pthread_mutex_t`. Inicializují se voláním funkce `pthread_mutex_init()`, která má dva parametry: ukazatel na mutex a atributy mutexu. Opět platí, že dosazením `NULL` se použijí implicitní atributy. Mutex se uvolní voláním funkce `pthread_mutex_destroy()`, která má jediný parametr: ukazatel na uvolňovaný mutex.

Pro vstup a výstup do/z kritického úseku kódu se používají funkce `pthread_mutex_lock()` a `pthread_mutex_unlock()`. Oba dva mají pouze jeden parametr: ukazatel na mutex. To znamená, že vstup do chráněného kódu zdaleka není tak propracovaný jako u API funkcí OS/2, podporováno je pouze neomezené čekání.

Příklad použití:

```
retval2 = pthread_mutex_lock(
 &((*(my_data->pointer_forks))[me->second_fork]).semaphore);
if (retval2 != 0)
{
 fprintf(stderr,
  "\nCannot get a mutex semaphore, return code = %d\n",
  retval2);
 fprintf(stderr, "Thread terminated.");
 me->state = STATE_DEAD;
 return (NULL);
}
```

3.3.3 Práce s událostmi

Místo event semaforů se u posixových vláken používají tzv. *podmínkové proměnné* (conditional variables), proměnné typu `pthread_cond_t`. Před použitím je nutno je inicializovat voláním `pthread_cond_init()` a po skončení výpočtu je uvolnit voláním `pthread_cond_destroy()`. Parametrem je jako vždy ukazatel na podmínkovou proměnnou, v prvním případě navíc atributy podmínkové proměnné, namísto kterých lze dosadit `NULL`.

Pro zastavení vlákna až do splnění určité podmínky (zaslání signálu) se používá funkce `pthread_cond_wait()`. Pro vzbuzení čekajícího vlákna jiným vláknem se použije funkce `pthread_cond_signal()`. Pozor: Volání obou funkcí musí být zaobaleno v části kódu chráněné nějakým mutexem. Ukazatel na tento mutex je také druhým parametrem funkce `pthread_cond_wait()`. Obě dvě funkce mají samozřejmě za parametr ukazatel na podmínkovou proměnnou.

Příklad čekání na splnění podmínky:

```
if ((retval = pthread_mutex_lock(&thread_count_mutex)) != 0)
{ ... }

retval = pthread_cond_wait(
 &block_main_thread_condition, &thread_count_mutex);
if (retval != 0)
{
 fprintf(stderr,
  "\nCannot wait for an event semaphore, return code = %d\n",
  retval);
 fprintf(stderr, "Exiting.");
 return (2);
} /* if retval */

if ((retval = pthread_mutex_unlock(&thread_count_mutex)) != 0)
{ ... }
```

Příklad signalizace splnění podmínky:

```
retval1 = pthread_mutex_lock(&thread_count_mutex);
if (retval1 != 0)
{ ... }
```

```
number_of_alive_philosophers--;
if (number_of_alive_philosophers == 0)
{
 retval2 = pthread_cond_signal(&block_main_thread_condition);
 if (retval2 != 0)
 {
  fprintf(stderr,
   "\nCannot post an event semaphore, return code = %d\n",
   retval2);
  fprintf(stderr, "Deadlock possible.");
 }
}

retval1 = pthread_mutex_unlock(&thread_count_mutex);
if (retval1 != 0)
{ ... }
```

3.4 Řešení v jazyce Java

Řešení problému večeřících filozofů je v jazyce Java ze všech tří způsobů nejkratší, nejelegantnější a patrně i nejsnáze pochopitelné. Java je objektově orientovaný jazyk, proto jak vidlička tak i filozof mají svoji vlastní třídu, umístěnou ve zvláštním souboru. V Javě se nepracuje přímo se semafory, využívá se zde tzv. monitorů, což jsou objekty, jejichž metody mají synchronizovaný kód, viz kapitola 1.1.1 a [Lea]. Kompletní zdrojový kód se nachází v archivu `DiningPhilosophers.ZIP` (nebo `.RAR`) v adresáři `Java\Sources`.

3.4.1 Realizace vidličky

Třída `Fork` je umístěna v souboru `Fork.java`. Obsahuje dvě monitorové metody: `acquireFork()` pro získání vidličky a `releaseFork()` pro její uvolnění. Každá vidlička si též pamatuje svého vlastníka – má atribut `owner`, typovaný na třídu `Philosopher`. Tento vlastník se aktualizuje, kdykoliv nějaký filozof vidličku získá.

Metoda `acquireFork()`:

```
public synchronized void acquireFork(Philosopher philosopher)
{
```

```
 while (owner != null)
 {
  try { wait(); } // try
  catch (InterruptedException e)
  { ... } // catch
 } // while
 owner = philosopher;
} // acquireFork
```

Pokud se nějaké vlákno pokouší získat vidličku, ale ta už patří někomu jinému, je volající vlákno uspáno. Probudí se až tehdy, kdy mu dosavadní majitel vidličky vzkáže, že vidličku už odevzdal a ta je nyní volná. Po probuzení (opuštění `wait()`) se zapíše nový majitel vidličky do atributu `owner`.

Metoda `releaseFork()`:

```
public synchronized void releaseFork(Philosopher philosopher)
{
 if (owner != philosopher)
  return;
 else
 {
  owner = null;
  notify();
 }
} // releaseFork
```

Chce-li vlákno vidličku odevzdat, pouze vynuluje vlastníka a následně vzbudí pomocí `notify()` jiné vlákno, které čeká v metodě `acquireFork()` na přidělení vidličky v pozastaveném stavu (v metodě `wait()`). Pokud nikdo takový nečeká, nic se nestane. Parametr `philosopher` se používá pouze pro kontrolu, aby vidličku nemohl uvolnit někdo, komu nepatří.

3.4.2 Realizace filozofa

Třída `Philosopher` je potomek třídy `Thread`, má tedy schopnost běžet paralelně vůči ostatním vláknům. Stejně jako v předchozích dvou případech si každý filozof pamatuje svoji levou a pravou vidličku, stejně tak první a druhou vidličku. Zde to ale nejsou indexy do nějakého pole vidliček, jsou to přímo reference na instance třídy `Fork`. Filozof tak může s vidličkami pracovat přímo, nemusí k nim přistupovat oklikou přes pole.

Za zmínku stojí pouze metoda `run()`, ve které filozof opakovaně přemýšlí (metoda `think()`), získává vidličky (`firstFork.acquireFork(this)` a `secondFork.acquireFork(this)`), jí (metoda `eat()`) a vidličky uvolňuje (`secondFork.releaseFork(this)` a `firstFork.releaseFork(this)`). Filozof se nestará o žádné semafory, žádné čekání ani uspávání a probouzení. Vše je elegantně vyřešeno ve třídě `Fork` použitím monitorových metod a voláním `wait()` a `notify()`.

```
public void run()
{
 for (int i = 0; i < NO_OF_TURNS; i++)
 {
  ...
  think();
  ...
  firstFork.acquireFork(this);
  ...
  secondFork.acquireFork(this);
  ...
  eat();
  ...
  secondFork.releaseFork(this);
  ...
  firstFork.releaseFork(this);
 } // for i
} // run
```

3.4.3 Čekání hlavního vlákna

Stejně jako v předchozích dvou případech čeká hlavní vlákno aplikace na dokončení všech ostatních vláken. K tomu jsou opět využity monitorové metody. Hlavní třída `DiningPhilosophers` má metodu `waitForZeroPhilosophers()`, do které vstoupí hlavní vlákno poté, co nastartuje všechny filozofy. Tam se uspí a čeká, dokud ho poslední filozof nevzbudí voláním `decrementCountOfPhilosophers()`.

```
private synchronized void waitForZeroPhilosophers()
{
 while (noOfLivingPhilosophers > 0)
 {
```

```
  try { wait(); } // try
  catch (InterruptedException e)
  { ... } // catch
 } // while
} // waitForZeroPhilosophers
```

Metoda `decrementCountOfPhilosophers()` je volána každým vláknem, které právě končí. Vždy o jednu sníží čítač běžících vláken a pokud je tento čítač roven nule a žádná další vlákna tak už neexistují, vzbudí čekající hlavní vlákno signálem vyslaným pomocí `notify()`. To se probudí uprostřed `wait()`, opustí `wait()`, opustí `waitForZeroPhilosophers()`, opustí `doIt()`, opustí `main()` a program může skončit.

```
public synchronized void decrementCountOfPhilosophers()
{
 noOfLivingPhilosophers--;
 if (noOfLivingPhilosophers == 0)
  notify();
} // decrementCountOfPhilosophers
```

Reference

[PAP] *Ježek K., Matějovic P., Racek S.*: **Paralelní architektury a programy**.
Vydavatelství ZČU, duben 1997, ISBN 80-7082-322-4

[Art] *Panov K., Solomon L. Jr., Panov A.*: **The Art of OS/2 Warp Programming**, kapitola 3.
John Wiley & Sons, 1995, ISBN 0-471-08633-9

[Games] *Duffy M.T.*: **Gearing up for Games**, Part 3.
EDM/2 Volume 3 Issue 8, září 1995.
`http://www.edm2.com`

[PPR] *Racek S.*: **Přednášky z předmětu Paralelní programování**.
Západočeská univerzita, 2000

[ZOS] *Rychlík J.*: **Přednášky z předmětu Základy operačních systémů**.
Západočeská univerzita, 1998

[Watcom] **Watcom C Library Reference**
`clib.inf`

[API] *Tabi T.*: **The OS/2 API Project**
`http://www.edm2.com/os2api`

[Lea] *Lea D.*: **Concurrent Programming in Java, Second Edition**
Addison-Wesley, listopad 2000, ISBN 0-201-31009-0