Problém synchronizace při programování vícevláknových aplikací 1

Programování vícevláknových (nebo obecněji paralelních) aplikací přináší celou řadu úskalí, která se při běžném programování nevyskytují. Velmi obvyklé je, že chyby se mohou objevovat nedeterministicky nebo se mohou projevit za velmi specifických okolností a jejich odhalení proto bývá komplikované.

1 V čem je problém a jak se mu vyhnout

Typický problém, se kterým se při programování vícevláknových aplikací můžeme setkat, jsou tzv. chyby souběhu (*race condition*), kdy více vláken pracuje souběžně se sdílenými prostředky. Těmi jsou typicky proměnné (příp. objekty v paměti), otevřené soubory, síťová a databázová spojení apod. Problém nastává, pokud jedno nebo více vláken může měnit stav sdílených prostředků,[1] např. když mění hodnotu globální proměnné nebo sdíleného objektu, zápisuje do souboru apod.

Nejefektivnějším způsobem, jak předejít chybám souběhu, je vyhnout se jim zcela. Víme-li, že potenciálním zdrojem chyb jsou změny sdílených prostředků, můžeme:

(a) *Vyhnout se sdílení*, tj. pokud je to z povahy problému možné, vytvoříme kopii dat, se kterými pracuje jen dané vlákno. To sice přináší režii, ale vyhneme se režii, kterou by měla koordinace přístupu k těmto datům.[2]

(b) *Vyhnout se změně stavu*, tj. máme-li data (nebo prostředky), která jsou od určitého bodu sdílena mezi dvěma a více vlákny, neměli bychom je měnit.

Jinými slovy je dobré, pokud můžeme vláknu předat na vstupu všechna data, která bude pro svou činnost potřebovat, tak aby vlákno nemuselo nijak interagovat s ostatními vlákny. U takto navrženého programu nejenže předejdeme některým typům chyb, ale program bude snazší na pochopení, což je žádaná vlastnost.

Jsou však situace, kdy se změnám sdíleného stavu nemůžeme vyhnout. V takovém případě je nutná *vzájemná synchronizace* vláken při přístupu ke sdíleným prostředkům. Tato synchronizace musí být zajištěna kdykoliv nějaké vlákno *mění sdílené prostředky* nebo *je čte*. Protože práce se sdílenými prostředky je místem, kde dochází nejčastěji k chybám souběhu, je vhodné, práci se sdílenými prostředky soustředit do malého množství funkcí (nebo metod), u kterých můžeme snadno ověřit, že máme korektně synchronizovaný přístup ke sdíleným prostředkům, a tak garantovat správný běh programu.

[1] V angličtině se to označuje jako *shared mutable state*.

[2] Toto řešení může na první pohled vypadat nepřirozeně, protože vyžaduje paměť navíc a programátoři mají (nebo by měli mít) tendenci zbytečně neplýtvat prostředky.

2 Synchronizace

2.1 Testovací prostředí

Problematiku synchronizace si ukážeme na tzv. problému producent-konzument. Kdy máme jedno nebo více vláken, která data vytváří (producent), a současně máme jedno nebo více vláken, která data zpracovávají (konzument).[3] Data mezi producenty a konzumenty se předávají přes sdílenou paměť, k čemuž se nejčastěji používá datová struktura fronta (FIFO). Pokud je fronta zaplněna, producent musí počkat, dokud se ve frontě neuvolní místo, než může předat data. Analogicky, pokud je fronta prázdná, čeká konzument na vložení dat do fronty.

V našem demonstračním příkladu použijeme klasickou implementaci fronty, do které je možné vkládat ukazatele, případně čísla přetypovaná na ukazatele, viz soubory `queue.c` a `queue.h`. Dále vytvoříme dvě vlákna producentů, která budou ukládat do fronty posloupnost celých čísel od zadané hodnoty. Kód tohoto vlákna vypadá následovně.

```
1  lock_t queue_lock = LOCK_INITIAL_VALUE;
2
3  void *producer(void *arg)
4  {
5          struct producer_request *req = arg;
6          struct queue *queue = req->queue;
7          int start = req->start;
8          for (int i = start; i < start + TEST_SIZE; i++) {
9                  int result;
10                 do {
11                         lock(&queue_lock);
12                         result = queue_put(queue, (void *) i);
13                         unlock(&queue_lock);
14                 } while (result == Q_FULL);
15         }
16         return NULL;
17 }
```

Vlákno převezme odkaz na objekt s frontou (řádek 6) a hodnotu, od které má ukládat čísla do fronty (řádek 7). Samotné ukládání probíhá v cyklu na řádcích 8 až 15. Vložení hodnoty je na řádku 12. Všimněme si, že tato operace je jednak obalena cyklem `while`, který zajišťuje opakované vložení, pokud je fronta plná. Dále si povšimněme funkcí `lock` a `unlock`. Funkce `lock` by měla zajistit, že v daný okamžik s frontou nebude pracovat žádné jiné vlákno. Jinými slovy, pokud s frontou již nějaké vlákno pracuje, musí vlákno volající `lock` počkat, dokud jiné vlákno nezavolá `unlock`. A opačně, pokud operace `lock` proběhne, jiná vlákna musí počkat, dokud nedojde k zavolání `unlock`.

Dále vytvoříme jedno vlákno konzumenta, které přečte všechny hodnoty z fronty a vypíše je. Kód tohoto vlákna vypadá následovně.

[3] Tato architektura je praktická, pokud potřebujeme zpracovat větší množství dat a rozložit zátěž mezi více procesorových jader.

```
1  void *consumer(void *arg)
2  {
3          struct queue *queue = arg;
4          for (int i = 0; i < 2 * TEST_SIZE; i++) {
5                  int result;
6                  do {
7                          void *value;
8                          lock(&queue_lock);
9                          result = queue_get(queue, &value);
10                         unlock(&queue_lock);
11                         if (result == Q_OK) {
12                                 printf("%i\n", (int) value);
13                                 fflush(stdout);
14                         }
15                 } while (result == Q_EMPTY);
16         }
17         return NULL;
18 }
```

Vlákno konzumenta odebírá hodnoty z fronty (řádek 9), a pokud je odebrána nějaká hodnota (řádek 11 až 14), je vypsána na standardní výstup. Protože předpokládáme, že se při testování budou objevovat chyby, po každém vypsání hodnoty pro jistotu vyprázdníme buffer (řádek 13), abychom měli vypsáno opravdu vše a nezůstalo nám něco v bufferu.

2.2 Bez synchronizace

Nejdříve vyzkoušíme, co se stane, pokud synchronizace nebude aktivní, tj. obě funkce pro synchronizaci budou vypadat následovně:[4]

```
void lock(lock_t *lock)
{
}
void unlock(lock_t *lock)
{
}
```

Protože s hodnotou zámku nijak nepracujeme, datový typ `lock_t`, můžeme zvolit libovolně, např. použít typ `int`.

```
typedef int lock_t;
```

Úkol 1: Program přeložte a spusťte.

[4] V přiložených zdrojových kódech jsou uvedeny různé varianty řešení zamykání. Pomocí podmíněného překladu je možné vybrat některou z implementací. Variantu, kdy je zamykání deaktivováno, je možné zvolit pomocí `#define USE_NO_LOCKS`.

Je pravděpodobné, že program neskončí.[5] Pokud by program běžel správně, měl by vypsat 2× `TEST_SIZE` hodnot. Počet vypsaných hodnot můžeme zjistit připojením nástroje `nl`[6] na výstup testovacího programu, tj. spustíme `./queue-test | nl`.

Vidíme, že počet řádků je menší než 2× `TEST_SIZE`. Při opakovaném spuštění bychom měli vidět, že počet vypsaných řádků se mezi spuštěními liší.

Úkol 2: Promyslete a zdůvodněte, proč program neskončil.

2.3 Špatně vyřešená synchronizace

Jako jednoduché řešení problému se nabízí vytvořit si zámek, např. proměnnou typu `int`, kde hodnota 0 bude signalizovat, že zámek je odemčený, a hodnota `1` bude signalizovat, že zámek je zamčený. Pokud je zámek zamčený, mělo by vlákno volající funkci `lock` čekat.

Takto pojaté zamykání by mohlo být implementované následovně.[7]

```
typedef int lock_t;
#define LOCK_INITIAL_VALUE (0)

void lock(lock_t *lock)
{
        while (*lock) { }
        *lock = 1;
}
void unlock(lock_t *lock)
{
        *lock = 0;
}
```

Pokud program vyzkoušíme, měl by se chovat podobně špatně jako v předchozím případě, tj. vypíše na výstup čísla, ale ne všechna.

Úkol 3: S pomocí nástroje `objdump`[8] se podívejte, jak byl kód funkce `lock` přeložen, a projděte si jednotlivé kroky (instrukce).

Úkol 4: V souboru `Makefile` změnte úroveň optimalizací, které překladač provádí, tj. zaměnte hodnotu `-O0` za `-O2` v proměnné `CFLAGS`. Odstraňte předchozí přeložené soubory pomocí `make clean` a program znovu přeložte a spusťte.

V této situaci se může stát,[9] že bude program rozbitý novým způsobem. Může se stát, že program nevypíše žádnou hodnotu. Podíváme-li se na to, jak byla funkce přeložena, můžeme v ní vidět přibližně následující.

```
100:   8b 07                    mov    eax,DWORD PTR [rdi]
```

[5] A je nutné jej ukončit, např. pomocí `Ctl+C`.
[6] Tento nástroj čte řádky ze svého vstupu a vypisuje je očíslované.
[7] Tento typ zamykání v ukázkovém kódu aktivujete pomocí `#define USE_NAIVE_SPINLOCK`
[8] Viz cvičení v letním semestru č. 2.
[9] Ale není to pravidlo.

```
102:   66 0f 1f 44 00 00       nop    WORD PTR [rax+rax*1+0x0]
108:   85 c0                   test   eax,eax
10a:   75 fc                   jne    108 <lock+0x8>
10c:   c7 07 01 00 00 00       mov    DWORD PTR [rdi],0x1
112:   c3                      ret
```

Na adrese 100 přečteme hodnotu zámku, který je předán pomocí prvního argumentu (viz registr `rdi`), instrukci na adrese 102 můžeme ignorovat, protože instrukce `nop`[10] nic neprovádí. Instrukce na adrese 108 otestuje, zde je hodnota zámku 0, nebo ne.[11] Pokud hodnota není nulová, pokračuje se instrukcí na adrese 108. V opačném případě je změněna hodnota zámku a je proveden návrat z funkce `lock`.

Tento výsledek je dán tím, že překladač předpokládal, že hodnota argumentu `lock` nebude změněna a vytvořil nekonečnou smyčku. Z kódu to tak vyplývá a překladač nemá informaci o tom, že kód dané funkce poběží ve více vláknech současně. Práce s vlákny je řešena na úrovni OS nikoliv překladače.

Řešením je označit proměnnou, která představuje zámek, jako `volatile`. Tím, můžeme signalizovat překladači, že daná proměnná může být měněna z více vláken a neměl by s ní provádět některé optimalizace.

Úkol č. 5: Změňte datový typ zámku z `int` na `volatile int`, program přeložte, spusťte a podívejte se, jak byla funkce `lock` přeložena. Pokud vše proběhlo správně, přehnaná optimalizace by měla být eliminována a program by měl být rozbitý původním způsobem, tj. vypisovat hodnoty, ale ne všechny.

2.4 Korektně vyřešená synchronizace

Důvod, proč předchozí řešení nefungovalo správně, je ten, že mezi otestováním, jestli daný zámek je `0`, a jeho nastavením na `1` byla prodleva, kdy mohlo dojít k přepnutí mezi vlákny, a tedy dvě vlákna současně mohla uzamknout jeden zámek a pracovat tak s frontou současně.

Řešením tohoto problému je použití atomických operací, tj. operací, které jsou (z pohledu vnějšího pozorovatele) vždy provedeny v jednom nedělitelném kroce. Jednou z takových operací je atomické prohození hodnoty v paměti s obsahem registru. Na procesorech x86/AMD64 se k tomu používá instrukce `xchg`.[12] Pokud chceme takovou instrukci v programu použít, musíme buď přejít do assembleru, nebo použít některé z nestandardních rozšíření,[13] které obvykle překladače nabízí. My použijeme druhou variantu, jak ukazuje následující kód.

```
1  typedef volatile int lock_t;
2  #define LOCK_INITIAL_VALUE (0)
3
4  void lock(lock_t *lock)
5  {
6          int key = 1;
7          while (__atomic_exchange_n(lock, key, __ATOMIC_SEQ_CST)) { }
```

[10] No operation.

[11] Instrukce `test` je podobná instrukci `and` s tím rozdílem, že výsledek operace se nikam neukládá a jen jsou nasteveny příznaky v příznakovém registru. Jedná se o podobný vztah jako u operací `cmp` a `sub`.

[12] K dispozici jsou i další atomické instrukce, např. `cmpxchg`.

[13] `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`

```
8  }
9  void unlock(lock_t *lock)
10 {
11         __atomic_store_n(lock, 0, __ATOMIC_SEQ_CST);
12 }
```

V cyklu na řádku 7 funkce `__atomic_exchange_n` atomicky prohodí hodnotu `key` a `lock` a tato funkce vrátí původní hodnotu proměnné `lock`.[14] Z toho plyne, že pokud hodnota `lock` byla `1`, je do této proměnné opět nastaveno `1`, je vrácena hodnota `1` a pomocí prázdného cyklu opakovaně testujeme stav proměnné `lock`. Pokud je hodnota `lock` rovna `0`, je do ní atomicky uložena hodnota `key`, tj. hodnota `1`, funkce vrátí `0` a dojde k ukončení smyčky.

Pro odemčení zámku (řádek 11) použijeme opět atomickou operaci, protože u operace přiřazení nemusí být obecně garantováno její atomické provedení.

Úkol č. 6: Program přeložte,[15] vyzkoušejte a podívejte se, jak byly přeloženy funkce `lock` a `unlock`.

V tento okamžik bychom měli mít korektní řešení, které ale není úplně efektivní, protože všechna čekající vlákna spotřebovávají procesorový čas.

[14] Třetí argument `__ATOMIC_SEQ_CST` udává, jak silné garance synchronizace vyžadujeme, v našem případě vyžadujeme úplné uspořádání operací.

[15] S použitou volbou `#define USE_TRUE_SPINLOCK`.