KAPITOLA 4

Synchronizace a sdílení dat

V této kapitole najdete:

- Soupeření o data
- Synchronizační primitiva
- Uváznutí typu deadlock a livelock
- Komunikace mezi vlákny a procesy
- Soukromá data vlákna
- Shrnutí

K tomu, aby vícevláknové aplikace dělaly něco užitečného, je obvykle nezbytné, aby vlákna sdílela nějaký druh společného stavu. Potřebná úroveň sdílení závisí na konkrétní úloze. Jedním extrémem může být jediné číslo, jež označuje úlohu, která se má provést. Například vláknu na webovém serveru může být oznámeno pouze číslo portu, na který má odpovídat. Druhým extrémem může být fond vláken neustále předávající informace mezi vlákny pro signalizaci, které úlohy jsou dokončené a která práce ještě stále čeká na dokončení. Mimo sdílení pro koordinaci činnosti existuje sdílení společných dat. Všechna vlákna tak mohou kupříkladu aktualizovat databázi nebo mohou být odpovědná za aktualizaci čítačů signalizujících množství dokončené práce.

Tato kapitola se věnuje nejrůznějším metodám pro sdílení dat mezi vlákny a nákladům spojeným s těmito přístupy. Začíná výkladem o *soupeření o data* (data races), což jsou situace, kdy více vláken aktualizuje stejná data nebezpečným způsobem. Jeden ze způsobů, jak zabránit soupeření o data, spočívá ve využití správné synchronizace mezi vlákny. Tato kapitola nabízí přehled obvyklých přístupů ke sdílení dat podporovaných většinou operačních systémů. Výklad se v maximální možné míře soustředí na abstraktní metody synchronizace a koordinace. Následující kapitoly pak nabídnou podrobnosti specifické pro konkrétní implementaci v prostředích POSIX a Windows.

Soupeření o data

Soupeření o data patří mezi nejčastější chyby vyskytující se při paralelním programování. K soupeření o data dochází tehdy, když několik vláken používá stejný datový prvek a jedno či více z těchto vláken jej začne aktualizovat. Nejlépe to lze vidět na příkladu. Představte si, že máte kód uvedený ve výpise 4.1, kde se ukazatel na celočíselnou proměnnou předává funkci, která její hodnotu zvýší o 4.

Výpis 4.1: Aktualizace hodnoty na zadané adrese

```
void update(int * a)
{
  *a = *a + 4;
}
```

Zpětný překlad tohoto kódu na architektuře SPARC by vypadal jako kód ve výpisu 4.2.

Výpis 4.2: Zpětný překlad kódu na platformě SPARC pro inkrementaci hodnoty uchovávané v paměti

```
ld      [%o0], %o1    // Load *a
add     %o1, 4, %o1   // Add 4
st      %o1, [%o0]    // Store *a
```

Řekněme, že se tento kód nachází ve vícevláknové aplikaci a že se dvě vlákna současně snaží inkrementovat stejnou proměnnou. Tabulka 4.1 ukazuje výsledný proud instrukcí.

Tabulka 4.1: Dvě vlákna aktualizující stejnou proměnnou

Hodnota proměnné a = 10	
Vlákno 1	Vlákno 2
`ld  [%o0], %o1   // Nahrej %o1 = 10`	`ld  [%o0], %o1   // Nahrej %o1 = 10`
`add %01, 4, %o1  // Přičti %o1 = 14`	`add %01, 4, %o1  // Přičti %o1 = 14`
`st  %o1, [%o0]   // Ulož %o1`	`st  %o1, [%o0]   // Ulož %o1`
Hodnota proměnné a = 14	

V tomto příkladu každé vlákno přičítá k proměnné číslo 4, jenže vzhledem k tomu, že to dělají v naprosto stejném okamžiku, uloží se nakonec do proměnné hodnota 14. Pokud by obě vlákna provedla kód v jiném okamžiku, měla by proměnná nakonec hodnotu 18.

Toto je situace, kdy obě vlákna běží souběžně. Jedná se o běžný typ soupeření o data a nejspíše jeden z nejjednodušších pro představu.

Další taková situace může nastat, když jedno vlákno běží, ale druhé vlákno bylo přepnutím kontextu odejmuto z procesoru. Představte si, že první vlákno si nahrálo hodnotu proměnné a nyní bylo přepnutím kontextu odejmuto z procesoru. Když se pak opět spustí, hodnota proměnné bude jiná a finální uložení obnoveného vlákna způsobí, že se její hodnota zase vrátí na starou hodnotu.

Uvažte situaci, kdy jedno vlákno uchovává hodnotu proměnné v registru a přijde druhé vlákno a změní tuto proměnnou v paměti, zatímco první vlákno provádí svůj kód. Hodnota uchovávaná v registru je tedy nyní nesynchronizovaná s hodnotou uchovávanou v paměti.

Závěr je ten, že k soupeření o data dochází v každém případě, kdy se nějaká proměnná nahrála a jiné vlákno uloží do téže proměnné novou hodnotu. Jedno z vláken tak nyní pracuje se „starými" daty.

Soupeření o data se těžko hledá. Vezměme předchozí ukázkový kód s inkrementováním proměnné. Může být umístěný v kontextu větší, složitější rutiny. Pouhou prohlídkou kódu tak může být obtížné identifikovat posloupnost problémových instrukcí. Posloupnost instrukcí způsobujících soupeření o data má pouze tři prvky a může být umístěná uvnitř ucelené oblasti kódu čítající stovky instrukcí.

Tento problém není jen obtížně zpozorovatelný při prohlídce, ale dochází k němu pouze tehdy, když obě vlákna shodou okolností provádějí tutéž malou oblast kódu. Takže i kdyby soupeření o data bylo ihned patrné a mohlo by se potenciálně vyskytnout pokaždé, je docela možné, že aplikace se soupeřením o data poběží delší dobu, než se chyby objeví. V tomto příkladu by detekce soupeření o data byla skutečně obtížná, pokud byste ovšem nevypisovali každou hodnotu proměnné a skutečně neviděli, že přijímá dvakrát tutéž hodnotu.

Potenciál pro soupeření o data je částí toho, co činí paralelní programování tak obtížné. Zanesení soupeření o data do kódu je běžnou chybou, jejíž existenci lze jen těžko odhalit prohlídkou kódu. Naštěstí existují nástroje, které v tom dokážou pomoci.

Nástroje pro detekci soupeření o data

Kód ve výpisu 4.3 obsahuje soupeření o data. Používá vlákna rozhraní POSIX, s nimiž se seznámíte v kapitole 5. Kód vytváří dvě vlákna, z nichž obě provádějí rutinu `func()`. Hlavní vlákno pak čeká, až obě podřízená vlákna dokončí svou činnost.

Výpis 4.3: Kód obsahující soupeření o data

```
#include <pthread.h>

int counter = 0;

void * func(void * params)
{
  counter++;
}

void main()
{
  pthread_t thread1, thread2;
  pthread_create(&thread1, 0, func, 0);
  pthread_create(&thread2, 0, func, 0);
  pthread_join(thread1, 0);
  pthread_join(thread2, 0);
}
```

Obě vlákna se pokusí inkrementovat proměnnou `counter`. Kód můžeme zkompilovat pomocí kompilátoru GNU gcc a pak použít nástroj Helgrind, který je součástí sady Valgrind[4], pro identifikování soupeření o data. Valgrind je nástroj, který umožňuje diagnostikování aplikace a prozkoumání jejího chování za běhu. Nástroj Helgrind používá tuto instrumentaci pro shromáždění údajů o soupeření o data. Výpis 4.4 ukazuje výstup z nástroje Helgrind.

Výpis 4.4: Použití nástroje Helgrind pro detekování soupeření o data

```
$ gcc -g race.c -lpthread
$ valgrind --tool=helgrind ./a.out
...
==4742==
==4742== Possible data race during write of size 4
         at 0x804a020 by thread #3
==4742==   at 0x8048482: func (race.c:7)
==4742==   by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742==   by 0x40414FE: start_thread
           (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742==   by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
```

[4] *http://valgrind.org/*

```
==4742== This conflicts with a previous write of size 4 by thread #2
==4742==    at 0x8048482: func (race.c:7)
==4742==    by 0x402A89B: mythread_wrapper (hg_intercepts.c:194)
==4742==    by 0x40414FE: start_thread
            (in /lib/tls/i686/cmov/libpthread-2.9.so)
==4742==    by 0x413849D: clone (in /lib/tls/i686/cmov/libc-2.9.so)
```

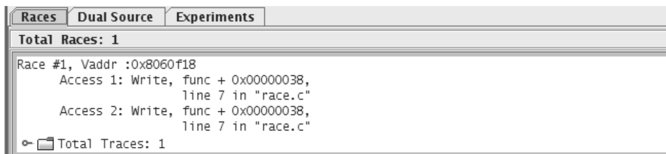
Z výstupu z nástroje Helgrind je patrné, že mezi dvěma vlákny, která provádějí řádek 7 v souboru `race.c`, existuje potenciální soupeření o data. To je očekávaný výsledek, je však třeba zdůraznit, že nástroje nacházejí i falešná zjištění. Programátor může napsat kód, kde různá vlákna přistupují k téže proměnné, přičemž může vědět, že existuje vynucené pořadí, které vlastní soupeření o data zastaví. Nicméně nástroje nemusejí být schopné detekovat vynucené pořadí a nahlásí potenciální soupeření o data.

Dalším nástrojem, který dokáže detekovat potenciální soupeření o data, je Thread Analyzer z aplikace Oracle Solaris Studio. Tento nástroj vyžaduje diagnostické sestavení aplikace, sběr dat provádí nástroj `collect` a grafické rozhraní spouští příkaz `tha`. Potřebné kroky ukazuje výpis 4.5.

Výpis 4.5: Detekování soupeření o data pomocí nástroje Sun Studio Thread Analyzer

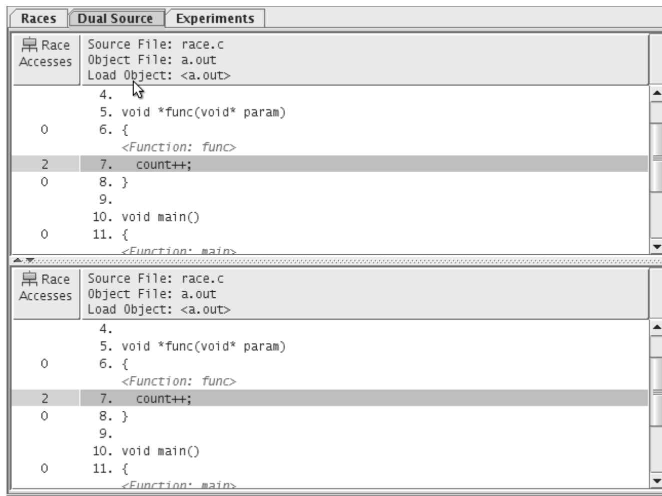
```
$ cc -g -xinstrument=datarace race.c
$ collect -r on ./a.out
Recording experiment tha.1.er ...
$ tha tha.1.er&
```

Počáteční obrazovka nástroje zobrazuje seznam soupeření o data (viz obrázek 4.1).



Obrázek 4.1: Seznam soupeření o data detekovaný nástrojem Solaris Studio Thread Analyzer

Jakmile uživatel identifikuje soupeření o data, které jej zajímá, může si prohlédnout zdrojový kód pro dvě umístění v kódu, kde k problému dochází. V příkladu zachyceném na obrázku 4.2 provádějí obě vlákna stejný řádek kódu.



Obrázek 4.2: Zdrojový kód s informacemi o soupeření o data zobrazenými v nástroji Solaris Studio Thread Analyzer

Jak se soupeření o data vyhnout

Přestože může být identifikování soupeření o data obtížné, lze se jim vyhnout velice jednoduše. Stačí zajistit, aby mohlo v daném okamžiku aktualizovat proměnnou pouze jedno vlákno. To lze nejjednodušeji provést umístěním synchronizačního zámku kolem všech přístupů k dané proměnné a zajištěním, aby vlákno muselo před použitím proměnné získat zámek. Výpis 4.6 ukazuje upravenou verzi výše uvedeného kódu. Tato verze používá zámek mutexu, popsaný podrobněji v následující části, k ochraně přístupu k proměnné `counter`. Přestože kód je takto správný, nemusí nutně dávat nejlepší výkon, jak ostatně uvidíte v pozdějších kapitolách.

Výpis 4.6: Kód upravený pro zabránění soupeření o data

```
void * func(void * params)
{
  pthread_mutex_lock(&mutex);
  counter++;
  pthread_mutex_unlock(&mutex);
}
```

Synchronizační primitiva

Synchronizace se používá pro koordinaci činnosti více vláken. Existují různé situace, kdy je to zapotřebí. Může jít o zajištění, aby ke sdíleným prostředkům nepřistupovalo více vláken současně nebo aby se před započetím nové práce dokončila veškerá práce na těchto prostředcích.

Většina operačních systémů nabízí bohatou sadu synchronizačních primitiv. Obvykle je nejvhodnější použít právě je spíše než se snažit napsat vlastní metody pro synchronizaci. Jsou pro to hned dva

důvody. Synchronizační primitiva poskytovaná operačním systémem obvykle rozpoznají nástroje poskytované tímto operačním systémem. Díky tomu pak tyto nástroje budou schopné odvádět lepší práci při detekování soupeření o data nebo při správném hlášení nákladů na synchronizaci. Operační systém často nabízí podporu pro sdílení primitiv mezi vlákny či procesy. Bez podpory operačního systému může být efektivní realizace takového sdílení velice obtížná. Nicméně nejzávažnější důvod tkví v tom, že kód poskytovaný operačním systémem nebude pravděpodobně obsahovat chyby. Vytváření vlastních synchronizačních primitiv se věnuje kapitola 8.

Mutexy a kritické oblasti

Nejjednodušší formou synchronizace je zámek se vzájemně výlučným přístupem neboli *mutex* (mutually exclusive). V daném okamžiku může pouze jedno vlákno získat zámek mutexu, takže je lze umístit kolem datové struktury pro zajištění, aby ji v daný okamžik mohlo modifikovat pouze jedno vlákno. Výpis 4.7 ukazuje, jak lze pomocí zámku mutexu chránit přístup k proměnné.

Výpis 4.7: Umístění zámků mutexu kolem přístupů k proměnné

```
int counter;

mutex_lock mutex;

void Increment()
{
  acquire(&mutex);
  counter++;
  release(&mutex);
}

void Decrement()
{
  acquire(&mutex);
  counter--;
  release(&mutex);
}
```

V tomto příkladu rutiny `Increment()` a `Decrement()` inkrementují, resp. dekrementují, proměnnou `counter`. Vlákno musí k modifikaci proměnné nejdříve získat zámek mutexu, což může v určitý okamžik provést pouze jedno vlákno. Všechna ostatní vlákna, jež chtějí získat stejný zámek, pak musejí čekat, dokud jej neuvolní vlákno, které jej drží. Obě rutiny používají stejný mutex, z čehož plyne, že v daný okamžik může inkrementovat či dekremenovat proměnnou `counter` pouze jediné vlákno.

Pokud se v jeden okamžik pokusí získat stejný mutex více vláken, uspěje pouze jedno vlákno a ostatní musejí čekat. Tato situace se označuje jako *dobývaný mutex* (contended mutex).

Oblasti kódu mezi získáním a uvolněním zámku mutex se říká *kritická sekce* (critical section) nebo *kritická oblast* (critical region). Kód v takovéto oblasti se bude v jednom okamžiku provádět pouze jedním vláknem.

Pro příklad kritické sekce si můžete představit, že operační systém nemá implementaci funkce malloc(), která by byla *bezpečná pro přístup z více vláken* (*thread-safe*). Jeden ze způsobů, jak to opravit, spočívá v umístění volání funkce malloc() do kritické sekce, což lze provést tak, že se jako ve výpisu 4.8 obklopí zámkem mutexu.

Výpis 4.8: Umístění zámku mutexu kolem určité oblasti kódu

```
void * threadSafeMalloc(size_t size)
{
  acquire(&mallocMutex);
  void * memory = malloc(size);
  release(&mallocMutex);
  return memory;
}
```

Pokud by se všechna volání funkce malloc() nahradila voláním funkce threadSafeMalloc(), pak by se v původním kódu funkce malloc() mohlo v jednom okamžiku nacházet pouze jedno vlákno, díky čemuž by se volání funkce malloc() stala bezpečná pro přístup z více vláken.

Vlákna *blokují*, pokud se pokusí získat zámek mutexu, který již drží jiné vlákno. Blokování znamená, že se vlákna uspí, a to buď okamžitě, nebo po několika málo neúspěšných pokusech o získání mutexu.

Problém s tímto přístupem tkví v tom, že dokáže program *serializovat*. Pokud několik vláken současně zavolá funkci threadSafeMalloc(), pak bude v daný okamžik postupovat jen jedno z nich. Ve vícevláknovém programu se tak bude provádět jen jediné vlákno, což znemožní programu využít výhody více jader.

Zámky spinlock

Zámky spinlock jsou v podstatě zámky mutexu. Rozdíl mezi zámkem mutexu a zámkem spinlock je ten, že vlákno čekající na získání zámku spinlock se bude neustále snažit získat zámek, aniž by došlo k jeho uspání. Na druhou stranu zámek mutexu může vést k uspání, pokud jej není možné získat. Výhodou zámků spinlock je to, že zámek získají ihned při jeho uvolnění, kdežto zámek mutexu musí operační systém před získáním zámku vzbudit. Nevýhoda pak spočívá v tom, že zámek spinlock rotuje na virtuální jednotce CPU, čímž si tento prostředek přivlastňuje. Na druhou stranu zámek mutexu způsobí spánek, čímž se virtuální jednotka CPU uvolní pro další vlákno.

Zámky mutexu se často implementují jako hybrid zámků spinlock a tradičních zámků mutexu. Vlákno, které se pokouší získat mutex, před blokováním krátkou chvíli rotuje. To vede k výhodě v oblasti výkonu. Vzhledem k tomu, že většina zámků mutexu se drží jen po velmi krátký čas, je docela pravděpodobné, že se zámek pro čekající vlákno uvolní. Díky rotování po krátký čas je pravděpodobnější, že čekající vlákno získá zámek mutexu okamžitě, jakmile se uvolní. Nicméně pokračování v rotaci po delší dobu spotřebovává hardwarové prostředky, které by se daly lépe použít pro běh ostatních softwarových vláken.

Semafory

Semafory jsou čítače, které lze buď zvyšovat, nebo snižovat. Lze je použít v situacích, kdy existuje konečný limit prostředků a je zapotřebí mechanizmus pro zajištění tohoto limitu. Příkladem může

být blok paměti, který má fixní velikost. Při každém přidání prvku do tohoto bloku se sníží počet dostupných pozic. Při každém odebrání prvku se počet dostupných pozic zvýší.

Semafory lze rovněž použít pro napodobení mutexů. Je-li v semaforu pouze jeden prvek, pak může být buď získaný, nebo dostupný, což přesně odpovídá tomu, kdy je mutex zamčený nebo odemčený.

Semafory také signalizují nebo probouzejí vlákna, která na nich čekají s cílem využít dostupné prostředky. Lze je tedy využít pro signalizaci mezi vlákny. Například vlákno může nastavit semafor, jakmile dokončí nějakou inicializaci. Ostatní vlákna mohou čekat na semaforu a obdržet signál k zahájení práce, jakmile je tato inicializace dokončená.

Podle implementace se metodě, která získává semafor, může říkat `wait`, `down` nebo `acquire`, přičemž metoda pro uvolnění semaforu může mít název `post`, `up`, `signal` nebo `release`. Když semafor již žádné další dostupné prostředky nemá, budou vlákna požadující prostředky blokovat, dokud prostředky nebudou zase dostupné.

Zámky čtenáři-zapisovač

Soupeření o data jsou problémem pouze při modifikaci sdílených dat. Více vláken provádějících čtení sdílených dat nepředstavuje žádný problém. Data určená pouze pro čtení tudíž nepotřebují ochranu nějakým druhem zámku.

Někdy je ovšem nutné data, která se obvykle jen čtou, aktualizovat. *Zámek čtenáři-zapisovač* (readers-writer lock), nebo též *zámek s více čtenáři* (multiple-reader lock), umožňuje, aby více vláken četlo sdílená data, dokáže však vlákna čtenářů zamknout a umožnit jedinému vláknu získat zapisovací zámek pro modifikaci těchto dat.

Zapisovač nemůže získat zapisovací zámek, dokud všichni čtenáři neuvolní své čtenářské zámky. Z tohoto důvodu se tyto zámky obvykle kloní na stranu zapisovačů. Jakmile se jeden zařadí do fronty, zámek ihned znemožní vstoupit dalším čtenářům. Tato akce způsobí, že počet čtenářů držících zámek slábne a nakonec umožní zapisovači získat výlučný přístup k zámku.

Úryvek kódu ve výpisu 4.9 ukazuje, jak je možné zámek čtenáři-zapisovač použít. Většina vláken bude volat rutinu `readData()` pro získání hodnoty z určité dvojice buněk. Jakmile vlákno získá čtenářský zámek, může před jeho uvolněním číst hodnotu buněk.

Pro modifikaci dat potřebuje vlákno získat zapisovací zámek. Tím se vláknům provádějícím čtení znemožní získání čtenářského zámku. Nakonec musejí všechna čtenářská vlákna svůj zámek uvolnit a až v tomto okamžiku získá zapisovací vlákno zámek a může aktualizovat data.

Výpis 4.9: Použití zámku čtenáři-zapisovač

```
int readData(int cell1, int cell2)
{
  acquireReaderLock(&lock);
  int result = data[cell] + data[cell2];
  releaseReaderLock(&lock);
  return result;
}
```

```
void writeData(int cell1, int cell2, int value)
{
  acquireWriterLock(&lock);
  data[cell1] += value;
  data[cell2] -= value;
  releaseWriterLock(&lock);
}
```

Bariéry

Existují situace, kdy musí celá skupina vláken dokončit svoji práci dříve, než se kterékoli z nich pustí do další práce. V takovýchto situacích je užitečné mít bariéru, kde budou vlákna čekat, dokud nebudou všechna přítomná.

Běžné podmínky pro použití bariéry nastávají tehdy, když existuje závislost mezi různými částmi kódu. Představte si, že řada vláken počítá hodnoty uložené v matici. Proměnnou `total` je nutné vypočítat pomocí hodnot uložených v matici. Pomocí bariéry lze zajistit, aby před výpočtem proměnné `total` dokončila všechna vlákna svůj výpočet matice. Výpis 4.10 ukazuje situaci používající bariéru pro oddělení výpočtu proměnné od jejího použití.

Výpis 4.10: Použití bariéry pro uspořádání výpočtů

```
Compute_values_held_in_matrix();
Barrier();
total = Calculate_value_from_matrix();
```

Proměnnou `total` je možné vypočítat pouze tehdy, když všechna vlákna dorazí k bariéře. Tím se zabrání situaci, kdy jedno z vláken stále provádí své výpočty, zatímco ostatní vlákna začínají používat výsledky výpočtů. Všimněte si, že po vypočtení hodnoty pro proměnnou `total` by mohla být zapotřebí další bariéra, pokud by se tato hodnota používala v dalších výpočtech. Tuto situaci s více bariérami ukazuje výpis 4.11.

Výpis 4.11: Použití několika bariér

```
Compute_values_held_in_matrix();
Barrier();
total = Calculate_value_from_matrix();
Barrier();
Perform_next_calculation(total);
```

Atomické operace a kód bez zámků

Použití synchronizačních primitiv může přinést vysoké režijní náklady. To platí zejména tehdy, jsou-li implementovány jako volání do operačního systému, a ne jako volání do podpůrné knihovny. Tato zátěž snižuje výkon paralelní aplikace a může omezit škálovatelnost. V některých případech mohou *atomické operace* nebo *kód bez zámků* vytvořit funkčně ekvivalentní kód bez takového množství zátěže.

Atomická operace je taková, která se buď úspěšně dokončí, nebo selže. Není tedy možné, aby skončila se „špatným" výsledkem nebo aby umožnila ostatním vláknům v systému sledovat nějakou pře-

chodnou hodnotu. Příkladem je atomická inkrementace, která by znamenala, že volající vlákno by nahradilo proměnnou, která aktuálně uchovává hodnotu N, hodnotou N+1. Může to znít triviálně, mějte však na paměti, že operace inkrementace proměnné může zahrnovat několik kroků, jak ostatně ukazuje výpis 4.12.

Výpis 4.12: Kroky zahrnuté v inkrementaci proměnné

```
LOAD   [%o0], %o1   // Nahraj počáteční hodnotu
ADD    %o1, 1, %o1  // Inkrementuj hodnotu
STORE  %o1, [%o0]   // Ulož novou hodnotu zpět do paměti
```

Během provádění těchto tří kroků by mohlo zasáhnout jiné vlákno a nahradit hodnotu proměnné uchovávanou v paměti novou hodnotou, čímž by vzniklo soupeření o data.

Atomická operace inkrementace by jinému vláknu neumožnila upravit stejnou proměnnou a způsobit tak zápis chybné hodnoty do paměti. Atomické provedení operace inkrementace je logicky ekvivalentní *implicitnímu* získání mutexu před inkrementací a jeho uvolnění poté. Rozdíl je v tom, že při použití mutexu se pro ochranu dané proměnné spoléhá na to, že ostatní vlákna použijí stejný mutex. Vlákno, které by stejný mutex nepoužilo, by mohlo způsobit, že se do paměti zapíše nesprávná hodnota. U atomických operací nemůže jiné vlákno způsobit zápis nesprávné hodnoty do paměti.

Většina operačních systémů nebo kompilátorů poskytuje podporu pro řadu atomických operací. Nicméně hardware obvykle nabízí podporu pouze pro omezenější skupinu operací. Pro jsou komplexnější atomické operace obvykle vytvořené z jednoduchých atomických instrukcí podporovaných hardwarem.

Atomické operace se často používají pro tvorbu *kódu bez zámků*. Použití synchronizačního zařízení, jako je zámek mutexu, řeší problém správnosti. V jednom okamžiku může k chráněné paměťové lokaci přistupovat pouze jedno vlákno. Nejedná se nicméně o přístup s nejlepším škálováním, neboť vlákna, která čekají na přístup, jsou blokována (uspána). Implementace bez zámků by se kvůli ochraně přístupu neopírala o zámek mutexu, ale použila by posloupnost operací, které by danou operaci provedly bez nutnosti získat explicitní zámek. To může vést k vyššímu výkonu než při řízení přístupu pomocí zámku. „Bez zámků“ neznamená, že by ostatní vlákna nemusela čekat. Jde pouze o to, že nemusejí čekat na zámku. Implementace *bez čekání* by vláknům umožňovala souběžný postup vpřed.

Většinu složitějších atomických operací tvoří ve skutečnosti implementace bez zámků. Používají nízkoúrovňovou hardwarem poskytovanou atomickou operaci a nabalují na ni kód pro zajištění, aby byla požadovaná operace na vyšší úrovni ve skutečnosti atomická.

Příkladem nízkoúrovňové atomické operace je operace *porovnání a prohození* (CAS – Compare And Swap), která atomickým způsobem prohodí hodnotu uchovávanou v registru s hodnotou uchovávanou v paměti, pokud hodnota v paměti odpovídá očekávané hodnotě. Jako příklad použití hardwarem poskytované atomické operace pro vytvoření atomické operace na vyšší úrovni lze uvést instrukci CAS, která může sloužit jako základ pro atomickou operaci inkrementace. K tomu stačí instrukci CAS provádět v cyklu, přičemž v každé iteraci by se pokusila prohodit hodnotu v paměti s inkrementovanou hodnotou. Cyklus by skončil v okamžiku, kdy by instrukce CAS úspěšně provedla operaci inkrementace.

Pomocí kódu bez zámků lze dosáhnout složitějších operací. Více se tomuto tématu věnuje kapitola 8.

Uváznutí typu deadlock a livelock

Dosud jsme si ukázali některé základní způsoby sdílení přístupu k prostředkům mezi vlákny. Nyní musíme probrat situace, kdy strategie vyvíjí nedobře.

První je *uváznutí typu deadlock*, kdy dvě či více vláken nemůže pokračovat dál, protože prostředky, které potřebují, drží jiná vlákna. Nejlépe si to ukážeme na příkladu. Představte si, že dvě vlákna potřebují pro dokončení nějaké úlohy získat zámky mutexu A a B. Pokud vlákno 1 již získalo zámek A a vlákno již získalo zámek B, pak vlákno 1 nemůže pokračovat, protože čeká na zámek B, a vlákno 2 nemůže pokračovat, protože čeká na zámek A. Obě vlákna se tak ocitnou v uváznutí typu deadlock. Tuto situaci ukazuje výpis 4.13.

Výpis 4.13: Dvě vlákna v uváznutí typu deadlock

```
// Vlákno 1                        // Vlákno 2
void update1()                     void update2()
{                                  {
  acquire(A);                        acquire(B);
  acquire(B); <<< Zde čeká           acquire(A); <<< Zde čeká
                  vlákno 1                           vlákno 2
  variable1++;                       variable1++;
  release(B);                        release(B);
  release(A);                        release(A);
}                                  }
```

Nejlepším způsobem, jak se vyhnout uváznutím typu deadlock, je zajistit, aby vlákna vždy získávala zámky ve stejném pořadí. Pokud by tedy vlákno 2 získalo zámky v pořadí A a B, zastavilo by se při čekání na zámek A, aniž by nejdříve získalo zámek B. Díky tomu by vlákno 1 mohlo získat zámek B a posléze uvolnit oba zámky, což by vláknu 2 umožnilo postupovat vpřed.

Uváznutí typu livelock uvězní vlákna v nekonečném cyklu uvolňování a získávání zámků. Uváznutí typu livelock mohou být způsobena kódem pro vytažení z uváznutí typu deadlock. Ve výpisu 4.14 se programátor snažil implementovat mechanizmus, který zabrání uváznutí typu deadlock. Pokud vlákno nemůže získat druhý zámek, který potřebuje, tak uvolní zámek, který již drží.

Každá z obou rutin `update1()` a `update2()` má vnější cyklus. Rutina `update1()` získá zámek A a poté se pokusí získat zámek B, zatímco rutina `update2()` provede totéž, ale v obráceném pořadí. Jedná se o klasickou příležitost k uváznutí typu deadlock, kterému se vývojář snažil zabránit napsáním kódu, jež způsobí, že se v případě nemožnosti získat druhý zámek uvolní držené zámky. Rutina `canAquire()` v tomto příkladu se okamžitě vrátí s tím, že buď získala zámek, nebo se získání zámku nezdařilo.

Výpis 4.14: Dvě vlákna v uváznutí typu livelock

```
// Vlákno 1                        // Vlákno 2
void update1()                     void update2()
{                                  {
  int done = 0;                      int done = 0;
  while (!done)                      while (!done)
  {                                  {
```

```
    acquire(A);
    if (canAcquire(B))
    {
      variable1++;
      release(B);
      release(A);
      done = 1;
    }
    else
    {
      release(A);
    }
  }
}
```

```
    acquire(B);
    if (canAcquire(A))
    {
      variable2++;
      release(A);
      release(B);
      done = 1;
    }
    else
    {
      release(B);
    }
  }
}
```

Pokud dvě vlákna narazí na tento kód ve stejný okamžik, budou chycená v uváznutí typu livelock při neustálém získávání a uvolňování mutexů, je však velmi nepravděpodobné, že by kterékoliv z nich postoupilo vpřed. Každé vlákno získá zámek a pak se pokusí získat druhý zámek, který potřebuje. Pokud se získání druhého zámku nezdaří, tak před opětovným pokusem o získání obou zámků uvolní držený zámek. Vlákno opustí cyklus, jakmile zvládne úspěšně získat oba zámky, což se nakonec může stát, avšak do té doby nemůže aplikace postupovat dále.

Komunikace mezi vlákny a procesy

Všechny paralelní aplikace vyžadují nějaký druh komunikace buď mezi vlákny, nebo mezi procesy. Obvykle je zde implicitní či explicitní akce spočívající v tom, že jedno vlákno odešle data jinému vláknu. Například jedno vlákno může signalizovat ostatním, že mají připravenou práci. Již jsme viděli příklad této signalizaci, kdy semafor může oznámit čekajícím vláknům, že inicializace je hotová. Vlákno signalizující semafor neví, zda existují nějaká další vlákna čekající na signál. Další možností je umístění zprávy do fronty, kde ji přijme vlákno, jehož úkolem je obsluha fronty.

Tyto mechanizmy obvykle vyžadují podporu operačního systému pro zprostředkování odeslání zpráv mezi vlákny a procesy. Programátoři mohou vymyslet své vlastní implementace, může však být efektivnější spolehnout se na operační systém, který vlákno uspí, dokud není splněna nějaká podmínka nebo dokud nedorazí nějaká zpráva.

Následující části nastiňují nejrůznější mechanizmy umožňující procesům a vláknům vzájemné předávání zpráv či sdílení dat.

Paměť, sdílená paměť a soubory mapované do paměti

Nejjednodušším způsobem pro komunikaci několika vláken je paměť. Pokud dvě vlákna mohou přistupovat ke stejnému místu v paměti, je cena takového přístupu o něco vyšší než prodleva paměti systému. Přístupy do paměti je i tak nutné řídit, aby bylo zajištěné, že v jednom okamžiku zapisuje do jednoho místa v paměti pouze jedno vlákno. Vícevláknová aplikace bude ve výchozím stavu sdílet paměť mezi vlákny, takže může jít o přístup s velmi nízkými náklady. Jediné věcí, které vlákna

nesdílejí, jsou proměnné na zásobníku každého vlákna (místní proměnné) a lokální proměnné vlákna, kterým se budeme věnovat později.

Sdílení paměti mezi několika procesy je komplikovanější. Všechny procesy mají standardně nezávislé adresové prostory, je proto nezbytné předem nakonfigurovat oblasti paměti, které bude možné sdílet mezi více různých procesů.

Pro zřízení sdílené paměti mezi dvěma procesy musí jeden proces pomocí volání do knihovny vytvořit oblast sdílené paměti. Při tomto volání se pro tuto sdílenou paměť použije jedinečný deskriptor. Obvykle se jedná o název souboru v systému souborů. Výsledkem volání je identifikátor popisovače, který lze nyní použít pro mapování oblasti sdílené paměti do adresového prostoru aplikace. Toto mapování vrátí ukazatel na nově namapovanou paměť. Vrácený ukazatel se úplně podobá ukazateli, který vrací funkce `malloc()`, a lze jej použít pro přístup do paměti uvnitř sdílené oblasti.

Když každý proces skončí, odpojí se od sdílené paměti, takže poslední končící proces ji může odstranit. Výpis 4.15 ukazuje přibližný proces vytvoření a odstranění oblasti sdílené paměti.

Výpis 4.15: Vytvoření a odstranění segmentu sdílené paměti

```
ID = Otevři Sdílenou Paměť(Deskriptor);
Paměť = Namapuj Sdílenou Paměť(ID);
...
Paměť[100]++;
...
Uzavři Sdílenou Paměť(ID);
Odstraň Sdílenou Paměť(Deskriptor);
```

Výpis 4.16 ukazuje proces připojení ke stávajícímu segmentu sdílené paměti. V tomto případě je sdílená oblast paměti již vytvořená, takže pomocí stejného deskriptoru, který se použil při jejím vytvoření, se lze připojit ke stávající oblasti sdílené paměti. Proces tak získá identifikátor, který může použít pro namapování této oblasti do procesu.

Výpis 4.16: Připojení stávajícího segmentu sdílené paměti

```
ID = Otevři Sdílenou Paměť(Deskriptor);
Paměť = Namapuj Sdílenou Paměť(ID);
...
Uzavři Sdílenou Paměť(ID);
```

Segment sdílené paměti může zůstat v systému, dokud není odstraněn. Proto je důležité naplánovat, který proces má odpovědnost za jeho vytvoření a odstranění.

Podmínková proměnná

Podmínkové proměnné (condition variables) oznamují připravenost mezi vlákny tak, že umožňují probuzení vlákna v okamžiku splnění určité podmínky. Bez podmínkových proměnných by čekající vlákno muselo pro kontrolu, zda je daná podmínka již splněná, použít nějakou formu dotazování.

Podmínkové proměnné pracují společně s mutexem. Mutex je zde pro zajištění, aby v jednom okamžiku mohlo k proměnné přistupovat pouze vlákno. Pomocí podmínkových proměnných lze napří-

klad implementovat model producent-konzument. Řekněme, že aplikace má jedno produkující vlákno a jedno konzumující vlákno. Producent přidává data do fronty a konzument je z fronty odebírá. Nejsou-li ve frontě žádná data, tak konzument musí spát, dokud neobdrží signál, že byl do fronty umístěn nějaký prvek dat. Výpis 4.17 ukazuje pseudokód pro produkující vlákno přidávající prvek do fronty.

Výpis 4.17: Produkující vlákno přidávající prvek do fronty

```
Získej Mutex();
Přidej Prvek do Fronty();
Pokud (Ve Frontě Je Pouze Jeden Prvek)
{
  Signalizuj Splnění Podmínek();
}
Uvolni Mutex();
```

Produkující vlákno musí signalizovat čekajícímu konzumujícímu vláknu pouze tehdy, když fronta byla prázdná a nyní do ní byl přidán nový prvek. Pokud ve frontě již bylo více prvků, musí být konzumující vlákno zaměstnáno jejich zpracováváním, a nemůže tudíž spát. Pokud ve frontě nebyly žádné prvky, pak je možné, že konzumující vlákno spí, a je tedy nutné jej vzbudit. Výpis 4.18 ukazuje pseudokód pro konzumující vlákno.

Výpis 4.18: Kód pro konzumující vlákno odebírající prvky z fronty

```
Získej Mutex();
Opakuj
  Prvek = 0;
  Pokud (Ve Frontě Není Žádný Prvek())
  {
    Počkej na Podmínkovou Proměnnou();
  }
  Pokud (Ve Frontě Je Prvek())
  {
    Prvek = Odeber z Fronty();
  }
Dokud (Prvek != 0);
Uvolni Mutex();
```

Konzumující vlákno bude čekat na podmínkovou proměnnou, je-li fronta prázdná. Když mu produkující vlákno pošle signál, aby jej probudilo, tak nejdříve zkontroluje, zda je něco ve frontě. Je docela možné, že konzumující vlákno bylo vzbuzené jen proto, aby zjistilo, že fronta je prázdná. Je důležité si uvědomit, že vzbuzení vlákna neznamená, že podmínka platí i nyní, což je důvod, proč je kód příkladu v opakujícím se cyklu. Je-li ve frontě nějaký prvek, může jej konzumující vlákno zpracovat. V opačném případě se vrátí ke spánku.

Interakce s mutexem je velice zajímavá. Produkující vlákno musí získat mutex ještě před přidáním prvku do fronty. Po přidání prvku do fronty jej musí uvolnit, avšak při signalizaci jej ještě stále drží. Konzumující vlákno nelze vzbudit, dokud se mutex neuvolní. Produkující vlákno mutex uvolní,

jakmile je signalizace hotová. Uvolnění mutexu je nezbytné k tomu, aby mohlo konzumující vlákno postupovat vpřed.

Konzumující vlákno získá mutex. Bude jej potřebovat pro bezpečnou modifikaci fronty. Pokud ve frontě nejsou žádné prvky, bude konzumující vlákno čekat na přidání prvku. Volání na čekání na podmínkovou proměnnou způsobí uvolnění mutexu a konzumující vlákno bude čekat na signalizaci. Jakmile se konzumující vlákno vzbudí, bude držet mutex, který uvolní buď po odebrání prvku z fronty, nebo v případě prázdné fronty při dalším voláním na čekání na podmínkovou proměnnou.

Produkující vlákno může používat dva typy probouzejících volání. Může buď probudit jediné vlákno, nebo může provést vyslání ke všem čekajícím vláknům. Který typ použít, závisí na kontextu. Je-li ke zpracování připraveno více prvků dat, pak dává smysl vzbudit vysíláním více vláknem. Na druhou stranu pokud produkující vlákno přidalo do fronty pouze jediný prvek, je daleko vhodnější vzbudit pouze jediné vlákno. Při probouzení všech vláken může nějakou chvíli trvat, než se všechna probudí, provedou svůj kód a vrátí se k čekání, což na systém klade zbytečnou zátěž. Všimněte si, že vzhledem k tomu, že každé vlákno musí při probuzení vlastnit mutex, je proces probouzení všech čekajících vláken sériový. V jednom okamžiku se tak může probudit pouze jedno vlákno.

Dále si všimněte, že při vyslání probouzejícího volání všem vláknům již mohou být některá z nich probuzená, není-li pro ně žádná práce. To je důvod, proč je nezbytné umístit čekání na podmínkovou proměnnou do cyklu.

Další problém, na který je dobré si dát pozor v souvislosti s podmínkovými proměnnými, je *ztracené probuzení* (lost wake-up). K tomu dochází, když se signál na probuzení čekajícího vlákna odešle ještě dříve, než je vlákno připravené jej přijmout. Výpis 4.19 ukazuje jinou verzi kódu konzumujícího vlákna. Tato verze může vykazovat problém ztraceného probuzení.

Výpis 4.19: Kód konzumujícího vlákna s potenciální problémem ztraceného probuzení

```
Opakuj
  Prvek = 0;
  Pokud (Ve Frontě Není Žádný Prvek())
  {
    Získej Mutex();
    Počkej na Podmínkovou Proměnnou();
    Uvolni Mutex();
  }
  Získej Mutex();
  Pokud (Ve Frontě Je Prvek())
  {
    Prvek = Odeber z Fronty();
  }
  Uvolni Mutex();
Dokud (Prvek != 0);
```

Problém tohoto kódu je v první podmínce. Pokud ve frontě nejsou žádné prvky, pak dojde k získání zámku mutexu a vlákno čeká na podmínkovou proměnnou. Nicméně produkující vlákno by mohlo umístit prvek do fronty a vyslat signál konzumujícímu vláknu, když konzumující vlákno provádělo

příkaz první podmínky a získávalo mutex. Když k tomu dojde, konzumující vlákno čeká na podmínkovou proměnnou nekonečně dlouho, protože produkující vlákno (viz výpis 4.17) vyšle signál pouze tehdy, když umístí první prvek do fronty.

Signály a události

Signály jsou unixový mechanizmus, kdy jeden proces může vyslat signál druhému, který má obslužnou rutinu provádějící po přijetí zprávy nějakou úlohu. Mnoho funkcí Unixu je implementováno pomocí signálů. Zastavení spuštěné aplikace stiskem ^C způsobí, že se do tohoto procesu odešle signál SIGKILL.

Systém Windows má podobný mechanizmus pro *události*. Obsluha stisků kláves a pohybu myši se provádí prostřednictvím mechanizmus událostí. Stisk jednoho z tlačítek myši způsobí, že se cílovému oknu odešle událost klepnutí.

Signály a události jsou ve skutečnosti optimalizované pro zasílání omezeného množství nebo žádných dat společně se signálem a jako takové pravděpodobně nejsou ve srovnání s ostatními možnostmi nejlepším mechanizmem pro komunikaci.

Výpis 4.20 ukazuje, jak se obsluha signálů obvykle instaluje a jak jí lze poslat signál. Jakmile se obsluha signálů nainstaluje, odeslání signálu danému vláknu způsobí, že se obsluha signálů spustí.

Výpis 4.20: Instalace a použití obsluhy signálů

```
void signalHandler(void *signal)
{
  ...
}

int main()
{
  installHandler(SIGNAL, signalHandler);
  sendSignal(SIGNAL);
}
```

Fronty zpráv

Fronta zpráv je struktura, kterou lze sdílet mezi více procesy. Zprávy je možné umístit do fronty, odkud se odebírají ve stejném pořadí, ve kterém se přidávaly. Konstrukce fronty zpráv vypadá spíše jako konstrukce segmentu sdílené paměti. První potřebnou věcí je deskriptor, kterým je obvykle umístění souboru v systému souborů. Tento deskriptor se pak může použít pro vytvoření fronty zpráv nebo pro připojení ke stávající frontě zpráv. Po nakonfigurování fronty do ní mohou procesy umisťovat zprávy nebo z ní zprávy odebírat. Po dokončení práce s frontou je třeba ji odstranit.

Výpis 4.21 ukazuje kód pro vytváření a umisťování zpráv do fronty. Tento kód je rovněž odpovědný za odstranění fronty po ukončení jejího používání.

Výpis 4.21: Vytváření a umisťování zpráv do fronty

```
ID = Otevři Frontu Zpráv(Deskriptor);
Dej Zprávu do Fronty(ID, Zpráva);
...
Zavři Frontu Zpráv(ID);
Odstraň Frontu Zpráv(Deskriptor);
```

Výpis 4.22 ukazuje proces přijímání zpráv z fronty. Pomocí deskriptoru pro stávající frontu zpráv mohou dva procesy komunikovat zasíláním a přijímáním zpráv prostřednictvím fronty.

Výpis 4.22: Otevření fronty a přijímání zpráv

```
ID = Otevři Frontu Zpráv(Deskriptor);
Zpráva = Odeber Zprávu z Fronty(ID);
...
Zavři Frontu Zpráv(ID);
```

Pojmenované roury

Unix používá pro předávání dat z jednoho procesu do druhého *roury* (pipes). Například výstup z příkazu `ls`, který vypisuje všechny soubory v daném adresáři, by se dal zařadit do roury k příkazu `wc`, který ve vstupu spočítá počet řádků, slov a znaků. Kombinací těchto dvou příkazů by tak byl součet počtu souborů v daném adresáři.

Pojmenované roury poskytují podobný mechanizmus, který lze řídit z programu. Pojmenované roury jsou objekty podobné souborům, jimž se přidělí určité jméno, které lze sdílet mezi procesy. Jakýkoliv proces pak může do takovéto do roury zapisovat nebo z ní číst. Není zde žádná představa „zprávy“. Data se považují za pouhý proud bajtů. Metoda pro použití pojmenované roury se velmi podobá metodě pro použití souboru. Roura se otevře, data se do ní zapíší nebo se z ní přečtou a potom se roura zavře.

Výpis 4.23 ukazuje kroky nezbytné pro zřízení a zápis do roury, než se roura uzavře a odstraní. Jeden proces musí provést samotné vytvoření roury, a jakmile ji vytvoří, lze ji otevřít a používat pro čtení či zápis. Jakmile proces svoji činnost dokončil, rouru může uzavřít a jeden z procesů, který ji používá, by měl být také odpovědný za její odstranění.

Výpis 4.23: Zřízení a zápis do roury

```
Vytvoř Rouru(Deskriptor);
ID = Otevři Rouru(Deskriptor);
Zapiš do Roury(ID, Zpráva, sizeof(Zpráva));
...
Uzavři Rouru(ID);
Odstraň Rouru(Deskriptor);
```

Výpis 4.24 ukazuje kroky nezbytné k otevření stávající roury a čtení zpráv z této roury. Procesy používající stejný deskriptor mohou pro účely komunikace otevřít a používat stejnou rouru.

Výpis 4.24: Otevření stávající roury pro příjem zpráv

```
ID = Otevři Rouru(Deskriptor);
Čti z Roury(ID, blok paměti, sizeof(blok paměti));
...
Uzavři Rouru(ID);
```

Komunikace prostřednictvím vrstev sítě

Vrstvy sítě (network stack) je docela komplexní sada vrstev, které sahají od síťové karty po vrstvu, která poskytuje komunikaci přes síťové pakety používanou aplikacemi, jako jsou webové prohlížeče. Kompletní pokrytí tohoto tématu je mimo rámec této knihy. Nicméně síťové rozhraní je dostupné na většině platforem, a jako takové je tedy možným kandidátem na komunikaci. Výhoda komunikace prostřednictvím síťového rozhraní spočívá v tom, že aplikace mohou komunikovat mezi procesy na jediném systému nebo procesy na jiných systémech propojených sítí. Jediná nezbytná změna by se týkala adresy, kam by se posílaly pakety s daty. Přestože komunikace napříč sítí může vykazovat docela vysokou prodlevu, použití síťového rozhraní pro komunikaci mezi procesy na tomtéž stroji bude mít většinou nižší náklady, ne však tak nízké jako u jiných metod komunikace.

Komunikace napříč sítí obvykle zahrnuje model klient-server. Pro zřízení serveru je nejdříve zapotřebí otevřít soket a ten pak před zahájením čekání na příchozí spojení svázat s adresou na místním hostiteli. Když spojení dorazí, je možné z něj číst data nebo je do něj zapisovat, dokud se spojení neuzavře. Jakmile se spojení uzavře, je možné zavřít i soket. Výpis 4.25 demonstruje, jak lze zřídit serverové vlákno síťového připojení typu klient-server.

Výpis 4.25: Zřízení soketu pro čekání na připojení

```
ID = Otevři Soket(Deskriptor);
Svaž Soket(ID, Adresa);
Naslouchej(ID);
Připojení = Počkej na Připojení(ID);
Čti(Připojení, blok paměti, sizeof(blok paměti));
...
Uzavři(Připojení);
Uzavři Soket(ID);
```

Výpis 4.26 ukazuje kroky nezbytné pro zřízení klientského soketu pro připojení k serveru. Připojení ke vzdálenému serveru rovněž vyžaduje zřízení soketu. Jakmile je soket otevřený, je možné jej použít pro připojení k serveru. Po dokončení komunikace lze soket uzavřít.

Výpis 4.26: Zřízení soketu pro připojení ke vzdálenému serveru

```
ID = Otevři Soket(Deskriptor);
Připoj(ID, Adresa);
Zapiš(ID, blok paměti, sizeof(blok paměti));
...
Uzavři(ID);
```

Další přístupy ke sdílení dat mezi vlákny

Existuje několik dalších přístupů pro sdílení dat. Data lze kupříkladu zapsat do souboru, který bude později čten jiným procesem. To může být vhodné v situaci, kdy je nutné data uložit trvale nebo kdy se data budou používat v určité pozdější době. I tak představuje zapisování na disk operaci s dlouhou prodlevou, což není nejlepší mechanizmus, je-li účelem pouhá komunikace.

Existují také přístupy pro sdílení dat mezi procesy specifické pro operační systémy. *Dveře* (doors) systému Solaris umožňují jednomu procesu předávat prvek dat jinému procesu a nechat si vrátit zpracovaný výsledek. Dveře jsou optimalizované na dobu odezvy, a proto mohou být levnější než použití dvou různých zpráv.

Soukromá data vlákna

Jednovláknová aplikace může pomocí globálních dat uchovávat stav programu. Například jednovláknový textový procesor může mít globální proměnné, jež uchovávají název upravovaného dokumentu nebo aktuální číslo řádku.

Ve vícevláknové aplikaci je někdy nutné, aby každé vlákno uchovávalo nějaký stav. Tento stav je vzhledem k tomuto vláknu soukromý, může však k němu přistupovat veškerý kód, který toto vlákno provádí. Vraťme se nyní k příkladu s textovým procesorem. Pokud otevře několik dokumentů a každý dokument obsluhuje jedno vlákno, pak každé vlákno musí mít samostatnou proměnnou pro uchovávání názvu dokument a aktuálního čísla řádku. Tato data by byla soukromá pro každé vlákno, takže žádné jiné vlákno by je nemohlo číst. Aplikace může mít i nadále nějaký globální stav (třeba zaznamenává detaily o osobě, která ji používá), přičemž všechna vlákna by měla přístup k těmto stejným informacím.

Existuje několik přístupů, které může vlákno použít pro uložení soukromých dat. Nejzjevnější způsob, jak to učinit, by spočíval v alokování pole pro uložení soukromých dat pro všechna vlákna, přičemž jako index by se používal identifikátor jednotlivých vláken. To je relativně přímočarý přístup, který může být v řadě případů dostatečný. Výpis 4.27 ukazuje, jak lze pomocí pole ukládat data, která jsou soukromá, tedy lokální vzhledem k danému vláknu. K poli `MyData[]` se přistupuje přes ID aktuálně prováděného vlákna. Díky tomu může každé vlákno uchovávat data na jedinečném místě v poli.

Výpis 4.27: Použití pole pro uložení lokálních dat vlákna

```
int MyData[20];

void ThreadedCode(int parameter)
{
  MyData[GetMyThreadID()] = parameter;
  ...
}
```

Dalším přístupem by bylo ukládání lokálních dat vlákna na zásobník. Každé vlákno má zásobník, který je pro toto vlákno soukromý. Z toho vyplývá, že vlákno může alokovat data na zásobník a nechat je jako soukromá pro dané vlákno. Nedoporučuje se předávat ukazatele na tato data ostatním vláknům, neboť data umístěná v zásobníku jsou platná pouze po dobu, kdy je vlákno živé a kdy blok

zásobníku obsahující tato data existuje. Výpis 4.28 ukazuje příklad použití zásobníku pro uchovávání lokálních dat vlákna.

Výpis 4.28: Uchovávání lokálních dat vlákna na zásobníku

```
void ThreadedCode(int parameter)
{
  int MyData = parameter;
  ...
}
```

Další možností, jak alokovat data soukromá pro dané vlákno, je použít *lokální úložiště vlákna* (thread-local storage). Jak již jeho název napovídá, proměnné alokované v lokálním úložišti vlákna jsou pro toto vlákno soukromé. Většina kompilátorů podporuje klíčové slovo `__thread`. Například kód uvedený ve výpisu 4.29 by deklaroval celočíselnou proměnnou s názvem `count`, která je lokální pro každé vlákno.

Výpis 4.29: Použití specifikátoru __thread pro označení lokálních dat vlákna

```
__thread int count;

void ThreadedCode(int parameter)
{
  count = parameter;
  ...
}
```

Pokaždé, když se vlákno odkáže na proměnnou `count`, přistoupí k hodnotě kopie, která je lokální vůči tomuto vláknu.

Další možnost spočívá v použití podpůrných funkcí pro alokování a dealokování lokálních proměnných. Hrubý nástin tohoto přístupu nabízí výpis 4.30. Nejdříve je nutné vytvořit identifikátor pro jedinečnou identifikaci lokální proměnné vlákna. Pomocí tohoto identifikátoru pak může vlákno číst data z této proměnné nebo je do ní zapisovat. Když vlákno dokončí práci s proměnnou, je nutné identifikátor odstranit.

Výpis 4.30: Použití rozhraní API pro správu lokálních dat vlákna

```
ID = Vytvoř ID();
Nastav Lokální Data Vlákna(ID, Hodnota);
Hodnota = Vezmi Lokální Data Vlákna(ID);
Odstraň ID(ID);
```

Shrnutí

Tato kapitola se věnovala různým přístupům, které lze použít pro synchronizaci vláken a pro sdílení dat mezi nimi. V pozdějších kapitolách budeme probírat implementační detaily těchto přístupů. Nyní byste měli mít přehled o atomických operacích a algoritmech bez zámků a měli byste znát nejrůznější primitiva pro synchronizaci a sdílení dat nabízená většinou operačních systémů.