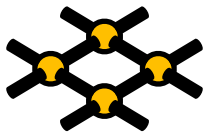
Vláknové programování
část VI

Lukáš Hejmánek, Petr Holub
{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192
2015–04–14

Vytváření vláken a procesů v Linuxu

- Vlákno vzniká systémovým voláním **`clone(2)`**
- Proces vzniká systémovým voláním **`fork(2)`** (případně **`vfork`**)
  - Nejčastější použití **`fork(2)`** je pro spuštění nového programu
  - Po **`fork(2)`** dojde k rozštěpení rodiče, duplikaci adresního prostoru, atd.
  - Následně je pomocí **`execl(3)`** zrušen obsah paměti a puštěn nový program
  - Jednodušší volání **`system(3)`**, nelze ale použít vždy
- Procesy se obvykle vytváří přímo voláním **`fork(2)`**, vlákna pomocí knihovny pthreads.

- Vytvoření procesu

```
1  #include <unistd.h>
2
3  void
4  run(char *name)
5  {
6          pid_t child;
7
8          if((child=fork())==0) {
9                  /* child */
10                 execlp(name, NULL);
11                 return;
12         }
13         if(child < 0) {
14                 perror("fork error");
15         }
16         /* parent */
17         return;
18 }
```

Vytváření vláken pomocí Pthreads

- Rukojeť vlákna **`pthread_t`**, používá se pro pro takřka všechna volání týkající se vytváření a manipulace s vlákny.
- **`int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void*), void* arg);`**

- Vytvoření vlákna v C

```
1  #include <pthread.h>
2
3  void *
4  runner(void *foo)
5  {
6          return NULL;
7  }
8
9  int
10 main(void)
11 {
12         pthread_t t;
13
14         pthread_create(&t, NULL, runner, NULL);
15         return 0;
16 }
```

- Nefunkční příklad pro C++

```
1  #include <pthread.h>
2
3  // not void*
4  void
5  runner(void *foo)
6  {
7          return;
8  }
9
10 int
11 main(void)
12 {
13         pthread_t t;
14
15         pthread_create(&t, NULL, runner, NULL);
16         return 0;
17 }
```

Ukončování vláken

- Možnosti ukončení vlákna samotným vláknem:
  - Návrat z hlavní funkce startu vlákna (třetí argument funkce **`pthread_create`**).
  - Explicitní zavolání funkce **`pthread_exit(void *value_ptr)`**.
- Možnosti ukončení vlákna jiným vláknem:
  - „Zabití“ vlákna **`pthread_kill(pthread_t thread, int sig)`**.
  - Zasláním signálu *cancel* **`pthread_cancel(pthread_t thread)`**
  - Nedoporučovaná možnost, není jisté, kde přesně se vlákno ukončí.
- Ukončení vlákna ukončením celého procesu
  - Zavoláním **`exit(3)`**
  - Posláním signálu **`SIGKILL`**, **`SIGTERM`**, ...

- Co s návratovou hodnotou ukončeného vlákna?
- Pro zjištění návratové hodnoty
  `int pthread_join(pthread_t thread, void **value)`.

```c
#include <pthread.h>
#include <signal.h>
#include <unistd.h>

void *
runner(void *foo)
{
        sleep(10);
        pthread_exit(NULL);
}

int
main(void)
{
        pthread_t t;

        pthread_create(&t, NULL, runner, NULL);

        pthread_kill(t, SIGKILL);
        return 0;
}
```

Open MP příkazy

- Přehled syntaxe
- Parallel
- Loop
- Sections
- Task (Open MP 3.0+)

Parallel

- Blok kódu prováděn několika vlákny
- Syntaxe:

```
1 #pragma omp parallel
2 {
3         /* parallel section */
4 }
```

Parallel – příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5          printf("Hello world from threads:\n");
6  #pragma omp parallel
7          {
8                  int tid = omp_get_thread_num();
9                  printf("<%d>\n", tid);
10         }
11         printf("I am sequential now\n");
12         return 0;
13 }
```

- Výstup:

```
Hello world from threads:
<1>
<0>
I am sequential now
```

Loop

- Iterace smyčky budou prováděny paralelně
- Na konci smyčky je implicitní barriéra, není-li řečeno jinak (**nowait**)
- Syntaxe:

```
1 #pragma omp for nowait
2 {
3         /* for loop */
4 }
```

Section(s)

- Neiterativní spolupráce
- Rozdělení bloků programu mezi vlákna
- Syntaxe:

```
#pragma omp sections
{
#pragma omp section
        /* first section */
#pragma omp section
        /* next section */
}
```

Section(s) příklad

```
1  #include <omp.h>
2  #define N 1000
3  int main () {
4          int i;
5          double a[N], b[N], c[N], d[N];
6          /* Some initializations */
7          for (i=0; i < N; i++) {
8                  a[i] = i * 1.5;
9                  b[i] = i + 22.35;
10         }
11 #pragma omp parallel shared(a,b,c,d) private(i)
12         {
13 #pragma omp sections
14                 {
15 #pragma omp section
16                         for (i=0; i < N; i++)
17                                 c[i] = a[i] + b[i];
18 #pragma omp section
19                         for (i=0; i < N; i++)
20                                 d[i] = a[i] * b[i];
21                 } /* end of sections */
22         } /* end of parallel section */
23         return 0;
24 }
```

Task – Open MP 3.0+

- Koncepce spuštění bloku kódu na „pozadí“
- Některé kusy kódu jdou špatně paralelizovat, např.:

```
1 while(my_pointer) {
2                 (void) do_independent_work (my_pointer);
3                   my_pointer = my_pointer->next ;
4 } // End of while loop
```

- **do_indipendent_work** by mohlo běžet v pozadí
- Pro starší OpenMP – napřed spočítat počet iterací, pak převést *while* na *for*
- Koncepce tasku:
  - Smyčka běží v jediném vlákně (kvůli procházení seznamu)
  - **do_independent_work** se pustí do pozadí
- Syntaxe:
  #pragma omp task

Task – příklad

```
1  my_pointer = listhead;
2  #pragma omp parallel
3  {
4          #pragma omp single nowait
5          {
6                  while(my_pointer) {
7                          #pragma omp task firstprivate(my_pointer)
8                          {
9                                  (void) do_independent_work (my_pointer);
10                         }
11                         my_pointer = my_pointer->next ;
12                 }
13         } // End of single - no implied barrier (nowait)
14 } // End of parallel region - implied barrier
```

Task

- Čekání na potomky (vytvořené tasky)
  `#pragma omp taskwait`
- *Task* má nepatrně vyšší režii než *for*

Task – příklad

```
1  my_pointer = listhead;
2  #pragma omp parallel
3  {
4          #pragma omp single nowait
5          {
6                  while(my_pointer) {
7                          #pragma omp task firstprivate(my_pointer)
8                          {
9                                  (void) do_independent_work (my_pointer);
10                         }
11                         my_pointer = my_pointer->next ;
12                 }
13         } // End of single - no implied barrier (nowait)
14         #pragma omp taskwait
15 } // End of parallel region - implied barrier
```

Start vlákna v C++11

- Třída **`thread`**
- **`#includle <thread>`**
- Metody
  - **`join`** – odpovídá **`pthread_join`**
  - **`detach`** – osamostatní vlákno, obdoba **`PTHREAD_CREATE_DETACHED`**

Start vlákna v C++11

```
1  #include <iostream>
2  #include <thread>
3
4  void foo1()
5  {
6          std::cout << "Foo1\n";
7  }
8
9  void foo2(int x)
10 {
11         std::cout << "Foo2\n";
12 }
13
14 int main()
15 {
16         std::thread first(foo1);
17         std::thread second(foo2,0);
18
19         second.detach();
20
21         std::cout << "main, foo and bar now execute concurrently...\n";
22
23         first.join();
24
25         std::cout << "foo and bar completed.\n";
26
27         return 0;
28 }
```

Základy synchronizace

Volatilní typy

- Nekonečná smyčka

```
1 int x=0;
2
3 void foo()
4 {
5         while(x==0);
6
7         x = 10;
8         //continue
9 }
```

```
1 foo:
2         movl    x(%rip), %eax
3         testl   %eax, %eax
4         je      .L2
5         movl    $10, x(%rip)
6         ret
7 .L2:
8 .L4:
9         jmp     .L4
```

Volatilní typy

- Funkční verze

```
1  volatile int x=0;
2
3  void foo()
4  {
5          while(x==0);
6
7          x = 10;
8          //continue
9  }
```

```
1  foo:
2  .L2:
3          movl    x(%rip), %eax
4          testl   %eax, %eax
5          je      .L2
6          movl    $10, x(%rip)
7          ret
```

Volatilní typy

- Volatilní proměnná: **`volatile int x;`** nebo **`int volatile x;`**
- Nevolatilní ukazatel na volatilní proměnnou: **`volatile int *x;`**
- Volatilní ukazatel na nevolatilní proměnnou: **`int *volatile x;`**
- Volatilní ukazatel na volatilní proměnnou: **`volatile int *volatile x;`**

Kritické sekce

- Co je to kritická sekce?
  - Nereentrantní část kódu
- Ne vždy je na první pohled zřejmé, co je a není reentrantní.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int x=0;

void *
foo(void *arg)
{
        int i;
        while(x == 0);
        for(i = 0; i < 1000000; i++) {
                x++;
        }
        printf("%d\n", x);
        return NULL;
}

int
main(void)
{
        pthread_t t1, t2, t3;

        pthread_create(&t1, NULL, foo, NULL);
        pthread_create(&t2, NULL, foo, NULL);
        pthread_create(&t3, NULL, foo, NULL);
        x=1;
        sleep(2);
        return 0;
}
```

- Příklad výstupu programu:
  - 1136215
  - 1355167
  - 1997368
- Očekávaný výstup:
  - xxxxxxx
  - yyyyyyy
  - 3000001
- Uvedené špatné chování se nazývá *race condition* (soupeření v běhu).

Řešení kritických sekcí

- Nejlépe změnou kódu na reentrantní verzi.
  - Ne vždy je to možné.
- Pomocí synchronizace = zamezení současného běhu kritické sekce
  - Snížení výkonu – přicházíme o výhodu paralelního běhu aplikace
- Synchronizační nástroje:
  - Mutexy (zámky)
  - Semafory
  - Podmíněné proměnné

Zámky

- Vzájemné vyloučení vláken
- Well-known algoritmy (ze „staré školy“)
  - Petersonův algoritmus
  - Dekkerův algoritmus
  - Lamportův algoritmus „pekařství“

Petersonův algoritmus

```
1  flag[0]   = 0;
2  flag[1]   = 0;
3  turn;
4
5  P0: flag[0] = 1;                          P1: flag[1] = 1;
6      turn = 1;                                 turn = 0;
7      while (flag[1] == 1 &&                    while (flag[0] == 1 &&
8             turn == 1)                                turn == 0)
9      {                                         {
10            // busy wait                        // busy wait
11     }                                         }
12     // critical section                       // critical section
13        ...                                    ...
14     // end of critical section                // end of critical section
15     flag[0] = 0;                              flag[1] = 0;
```

- Proč nefunguje:
  http://bartoszmilewski.wordpress.com/2008/11/05/who-ordered-memory-fences-on-an-x86/

Změny pořadí zápisů a čteni

- Proč algoritmy ze „staré školy“?
- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:

```
1 init: x=0, ready=0
2 Thread 1          Thread 2
3 x = 1             if ready == 1
4 ready = 1            R = x
```

Změny pořadí zápisů a čteni

- Proč algoritmy ze „staré školy“?
- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:

```
1 init: x=0, ready=0
2 Thread 1          Thread 2
3 x = 1             if ready == 1
4 ready = 1            R = x
```

  - **R=1 && x=0**

Změny pořadí zápisů a čteni

- Proč algoritmy ze „staré školy“?
- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:
  - ```
    1 init: x=0, ready=0
    2 Thread 1          Thread 2
    3 x = 1             if ready == 1
    4 ready = 1            R = x
    ```
  - **R=1 && x=0**
  - ```
    1 init: x=0, y=0;
    2 Thread 0          Thread 1
    3 mov [x], 1      mov [y], 1
    4 mov r1, [y]     mov r2, [x]
    ```

Změny pořadí zápisů a čteni

- Proč algoritmy ze „staré školy“?
- Současné procesory mohou měnit pořadí zápisů a čtení
- Čtení může prohozeno se starším zápisem

```
1 int a,b;
2
3  a = 5;
4  if(b) { }
```

- Důsledek:
  - ```
    1 init: x=0, ready=0
    2 Thread 1          Thread 2
    3 x = 1             if ready == 1
    4 ready = 1            R = x
    ```
  - **R=1 && x=0**
  - ```
    1 init: x=0, y=0;
    2 Thread 0          Thread 1
    3 mov [x], 1      mov [y], 1
    4 mov r1, [y]     mov r2, [x]
    ```
  - **r1=0 && r2=0**

Speciální instrukce CPU

- Paměťové bariéry
  - rmb(), wmb()
  - **`__sync_synchronize()`** – plná paměťová bariéra

Speciální instrukce CPU

- Atomické operace
  - Bit test (testandset())
  - Load lock/Store Conditional (LL/SC)
  - Compare and Swap (CAS) (x86 – **`cmpxchg`**)
    - **`__sync_bool_compare_and_swap()`**
    - **`__sync_value_compare_and_swap()`**
  - Atomická aritmetika – specialita x86, x86_64
    - Speciální instrukce **`lock`** formou prefixu
    - **`atomic_inc() { "lock xaddl %0, %1"}`**
      `__sync_fetch_and_add(val, 1)`
    - **`atomic_dec() { "lock xsubl %0, %1"}`**
      `__sync_fetch_and_sub(val, 1)`
    - **`xchg(int a, int b) { "xchgl %0, %1"}`**

Zámek

- Naivní algoritmus zámku

```
1  volatile int val=0;
2
3  void lock() {
4          while(atomic_inc(val)!=0) {
5                  //sleep
6          }
7  }
8
9  void unlock() {
10         val = 0;
11         // wake up
12 }
```

Zámek s podporou kernelu

- Podpora kernelu o „volání“ **`my_sleep_while()`**
  - Pozastaví proces právě tehdy když je podmínka splněna
- „volání“ **`my_wake()`**
  - Vzbudí pozastavený proces(y)

```
1  volatile int val=0;
2
3  void lock() {
4          int c=
5          while((c=atomic_inc(val))!=0) {
6                  my_sleep_while(val==(c+1));
7          }
8  }
9
10 void unlock() {
11         val = 0;
12         my_wake();
13 }
```

Mutexy

- Mechanismus zámků v knihovně Pthreads
- Datový typ **`pthread_mutex_t`**.
- Inicializace **`pthread_mutex_init`**
  (Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Zamykání/odemykání
  - **`pthread_mutex_lock`**
  - **`pthread_mutex_unlock`**
- Zrušení zámku **`pthread_mutex_destroy`**.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int x=0;

pthread_mutex_t x_lock;

void *
foo(void *arg)
{
        int i;
        while(x == 0);
        for(i = 0; i < 1000000; i++) {
                pthread_mutex_lock(&x_lock);
                x++;
                pthread_mutex_unlock(&x_lock);
        }
        printf("%d\n", x);
        return NULL;
}
```

```
1  int
2  main(void)
3  {
4          pthread_t t1, t2, t3;
5
6          pthread_mutex_init(&x_lock, NULL);
7          pthread_create(&t1, NULL, foo, NULL);
8          pthread_create(&t2, NULL, foo, NULL);
9          pthread_create(&t3, NULL, foo, NULL);
10         x=1;
11         sleep(2);
12         pthread_mutex_destroy(&x_lock);
13         return 0;
14 }
```

- Výstup změněného programu:
  - 2424575
  - 2552907
  - 3000001
- Což je očekávaný výsledek

Zámky „bolí“

- Mějme tři varianty předchozího příkladu:

```
1 void foo(void *arg) {
2         int i;
3         while(x == 0);
4         for(i = 0; i < 100000000; i++)
5                 x++;
6 }
```

```
1 void foo(void *arg) {
2         int i;
3         while(x == 0);
4         for(i = 0; i < 100000000; i++)
5                 asm("lock incl %0" : : "m" (x));
6 }
```

```
1 void foo(void *arg) {
2         int i;
3         while(x == 0);
4         for(i = 0; i < 100000000; i++) {
5                 pthread_mutex_lock(&x_lock);
6                 x++;
7                 pthread_mutex_unlock(&x_lock);
8         }
9 }
```

- Délky trvání jednotlivých variant (real time, 3 vlákna)
  - Bez zámku (nekorektní verze)
    1.052sec
  - „Fast lock“ pomocí assembleru
    5.716sec
  - pthread mutex
    66.414sec

Big kernel lock vs. Spin locking

- Vlákna a procesy mohou mít velké množství zámků.
- Koncepce *Big kernel lock*
  - Pro všechny kritické sekce máme jeden společný zámek
  - Název pochází z koncepce Linux kernelu verze 2.0
  - Jednoduchá implementace
  - Může dojít k velkému omezení paralelismu
- Koncepce *Spin locking*
  - Pro každou kritickou sekci zvláštní zámek
  - Název pochází z koncepce Linux kernelu verze 2.4 a dalších
  - Složitější implementace
  - Omezení paralelismu je většinou minimální
  - Velké nebezpečí vzniku skrytých race conditions.
    - Některé kritické sekce mohou spolu dohromady tvořit další kritickou sekci.

Spin locks

- Klasické zámky (mutexy) používají systémové volání **`futex()`**.
  - Podpora jádra pro NPTL implementaci POSIX threads.
  - Mutexy používají systémová volání ⇒ nutnost přepnutí kontextu.
- Zámky typu spin jsou implementovány kompletně v user space.
  - Nemusí se přepínat kontext.
  - Za cenu busy loopu při pokusu zamknout zámek (Vlákno se cyklicky dotazuje, zda je možno zámek zamknout – spinning).
  - Jsou situace, kdy přepnutí kontextu trvá déle než busy loop pro zamčení.
- Kdy je vhodné použít spin locks?
  - Při velmi krátké kritické sekci (typicky zvýšení/snížení proměnné).
  - Nedojde-li k přepnutí kontextu jinou cestou (máme-li více vláken než procesorů, spin lock neurychlí běh).
- Ne všechny implementace POSIX threads poskytují spin locks!

Spin locks

```
1  void spin_lock(volatile int *lock)
2  {
3          __sync_synchronize();
4          while(! __sync_bool_compare_and_swap(lock, 0, 1));
5  }
6
7  void spin_unlock(volatile int *lock)
8  {
9          *lock = 0;
10         __sync_synchronize();
11 }
```

Spin locks

- Datový typ **`pthread_spin_t`**.
- Inicializace **`pthread_spin_init`**
  (Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Zamykání/odemykání
  - **`pthread_spin_lock`**
  - **`pthread_spin_unlock`**
- Zrušení zámku **`pthread_spin_destroy`**.

Příklad

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4
5  int x=0;
6
7  pthread_spinlock_t x_lock;
8
9  void *
10 foo(void *arg)
11 {
12         int i;
13         while(x == 0);
14         for(i = 0; i < 100000000; i++) {
15                 pthread_spin_lock(&x_lock);
16                 x++;
17                 pthread_spin_unlock(&x_lock);
18         }
19         printf("%d\n", x);
20         return NULL;
21 }
```

Příklad

```
22 int
23 main(void)
24 {
25         pthread_t t1, t2;
26
27         pthread_spin_init(&x_lock, 0);
28         pthread_create(&t1, NULL, foo, NULL);
29         pthread_create(&t2, NULL, foo, NULL);
30         x=1;
31         pthread_join(t1, NULL);
32         pthread_join(t2, NULL);
33         pthread_spin_destroy(&x_lock);
34         return 0;
35 }
```

Doba běhu příkladu

- Test na 2 procesorovém systému.
- V případě 2 vláken:
  - Za použití mutexů: 29 sec
  - Za použití spinů: 11 sec
- V případě 3 vláken:
  - Za použití mutexů: 28 sec
  - Za použití spinů: 29 sec

Synchronizace v OpenMP

- Kritickým sekcím se nevyhneme ani v OpenMP
  - Závislosti v běhu programu (některé sekce musí být hotové dřív jak jiné)
  - Některé kusy nemohou být prováděny paralelně
- Synchronizační primitiva
  - Critical, Atomic
  - Barrier
  - Single
  - Ordered, Flush

Critical

- Critical
  - Specifikuje sekci v programu, kterou může vykonávat nejvýše jedno vlákno (je jedno které)
  - Všechna vlákna postupně sekcí projdou
  - V podstatě označuje kritickou sekci
  - Syntaxe:
    `#pragma omp ciritcal [jméno]`
  - **`jméno`** je globální identifikátor, kritické sekce stejného jména jsou považovány za identické, tj. žádné bloky stejného jména nepoběží paralelně

Atomic

- Specifikuje sekci v programu, kterou může vykonávat nejvýše jedno vlákno (je jedno které)
- Lehká forma synchronizace, synchronizované jsou pouze čtení a zápisy
- Využívá **`lock`** instrukce na x86/x86_64 architektuře
- Syntaxe:
  `#pragma omp atomic`

```
1 #pragma omp atomic
2     a[indx[i]] += b[i];
```

- Výraz musí být „atomizovatelný“ jinak je ohlášena chyba
- Typicky: **`x++`**, **`x += 2`**
- Jde přeložit: **`*a += *a + 1`** ale nefunguje korektně!

Single, Master

- Podobně jako Critical, Single specifikuje sekci, kterou může provádět pouze jedno vlákno
- Narozdíl od Critical, je tato sekce provedena pouze jednou
- Vhodné pro thread-unsafe sekce, např. I/O
- Syntaxe:
  `#pragma omp single`
- Master je stejné jako Single, sekci provede vždy „master“ vlákno
- Syntaxe:
  `#pragma omp master`

Příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main()
4  {
5     int n = 9, i, a, b[n];
6     for (i=0; i<n; i++)
7         b[i] = -1;
8  #pragma omp parallel shared(a,b) private(i)
9  {
10    #pragma omp single
11    {
12       a = 10;
13    }
14    #pragma omp barrier
15    #pragma omp for
16    for (i=0; i<n; i++)
17        b[i] = a;
18
19 } /*-- End of parallel region --*/
20    printf("After the parallel region:\n");
21    for (i=0; i<n; i++)
22        printf("b[%d] = %d\n",i,b[i]);
23    return(0);
24 }
```

Příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main()
4  {
5     int i, n = 25, sumLocal;
6     int sum = 0, a[n];
7     int ref =(n-1)*n/2;
8     for (i=0; i<n; i++)
9         a[i] = i;
10    #pragma omp parallel shared(n,a,sum) private(sumLocal)
11    {
12       sumLocal = 0;
13       #pragma omp for
14         for (i=0; i<n; i++)
15           sumLocal += a[i];
16       #pragma omp critical (update_sum)
17       {
18         sum += sumLocal;
19         printf("sumLocal = %d sum = %d\n",sumLocal,sum);
20       }
21    } /*-- End of parallel region --*/
22    printf("Value of sum after parallel region: %d\n",sum);
23    printf("Check results: sum = %d (should be %d)\n",sum,ref);
24    return(0);
25 }
```

Reduction

- Redukuje seznam proměnných do jedné za použití konkrétního operátoru
- Syntaxe:
  `#pragma omp reduction (op : list)`
- **`list`** je seznam proměnných a **`op`** je jeden z následujících
  - $+$, $-$, $*$, &, $\widehat{\ }$, |, &&, ||

Reduction – příklad

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main()
4  {
5     int i, n = 25;
6     int sum = 0, a[n];
7     int ref = (n-1)*n/2;
8     for (i=0; i<n; i++)
9         a[i] = i;
10    printf("Value of sum prior to parallel region: %d\n",sum);
11    #pragma omp parallel for default(none) shared(n,a) reduction(+:sum)
12         for (i=0; i<n; i++)
13           sum += a[i];
14    /*-- End of parallel reduction --*/
15
16    printf("Value of sum after parallel region: %d\n",sum);
17    printf("Check results: sum = %d (should be %d)\n",sum,ref);
18
19    return(0);
20 }
```

Zamykání v C++11

- Třída **`mutex`**
- **`#include <mutex>`**
- Metody
  - **`lock`** – odpovídá **`pthread_mutex_lock`**
  - **`unlock`** – odpovídá **`pthread_mutex_unlock`**

Zamykání v C++11

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  std::mutex mtx;
6
7  void print_block (int n, char c) {
8          mtx.lock();
9          for (int i=0; i<n; ++i) { std::cout << c; }
10         std::cout << '\n';
11         mtx.unlock();
12 }
13
14 int main ()
15 {
16         std::thread th1 (print_block,50,'*');
17         std::thread th2 (print_block,50,'$');
18
19         th1.join();
20         th2.join();
21
22         return 0;
23 }
```

Zamykání v C++11

- Šablona **`std::unique_lock`**
- **`#include <mutex>`**
- Šablona garantuje odemknutí zámku při destrukci objektu
- Použití: **`std::unique_lock<std::mutex> lck (mtx);`**
  - Zámek **`mtx`** je zamknut.

Atomické typy

- Šablona **`atomic`**
- **`#include <atomic>`**
- V C++ šablona akceputuje libovolný typ (včetně objektů a-la 1MB)
- Operátory
  - Přiřazení **=**
  - **++**
  - **--**
  - Přiřazení s operátorem např. **+=**

Atomické typy

- Metody
  - **`is_lock_free`** – vrací true, pokud lze použít lock free mechanismus
  - **`exchange`** – atomický swap
  - **`load`** – atomické čtení objektu
  - **`store`** – atomický zápis objektu
    - Pro **`load`** a **`store`** lze specifikovat uspořádání operací
    - **`memory_order_relaxed`** – žádné bariéry
    - **`memory_order_consume`** – synchronizace proti závislým datům z **`memory_order_seq_cst`**
    - **`memory_order_acquire`** – synchronizace proti všem datům z **`memory_order_seq_cst`**
    - **`memory_order_seq_cst`** – úplná bariéra

Atomické typy

```
1  #include <iostream>
2  #include <thread>
3  #include <atomic>
4
5  std::atomic<int> x;
6
7  void foo()
8  {
9          for(int i = 0; i < 1000000; i++) {
10                 // x=x+1 <-- does NOT work!
11                 x+=1;
12         }
13         std::cout << x << "\n";
14 }
15
16 int main()
17 {
18         x.store(0, std::memory_order_relaxed);
19         std::thread first(foo);
20         std::thread second(foo);
21         first.join();
22         second.join();
23         std::cout << "foo completed with " << x << "\n";
24         return 0;
25 }
```

Semafory

- Mutexy řeší vzájemné vyloučení v kritických sekcích.
- Semafory řeší synchronizaci úloh typu producent/konzument.
- Producent/konzument úloha:
  - Producent vyrábí
  - Konzument(i) spotřebovává
  - Problém synchronizace - konzument může spotřebovat nejvýše tolik, co producent vytvořil
  - Příklad:
    - Producent přidává objekty do fronty
    - Konzument odebírá objekty z fronty
    - Synchronizace: konzument může odebrat pouze, je-li fronta neprázdná, jinak má čekat.
    - Není vhodné čekat pomocí tzv. busy loop, tj. neustále zjišťovat stav front, vlákno zbytečně spotřebovává procesor.

Synchronizace s použitím busy loop

```
1  producer:
2
3          while( ) {
4                  pridej prvek do fronty;
5          }
6
7  consumer:
8          while( ) {
9                  /* busy loop */
10                 while(fronta prazdna) nedelej nic;
11
12                 odeber prvek z fronty
13         }
```

Semafor

- Semafor je synchronizovaný čítač.
- Producent čítač zvyšuje
- Konzument čítač snižuje
- Čítač nelze snížit k záporné hodnotě
- Pokus o snížení k záporné hodnotě zablokuje vlákno, dokud není čítač zvýšen.

- Rukojeť semaforu **`sem_t`**.
- Inicializace semaforu **`sem_init()`**
  (Inicializaci je vhodné provádět před vytvořením vláken).
- Zvýšení hodnoty semaforu **`sem_post()`**.
- Snížení hodnoty semaforu a případně čekání na zvýšení jeho hodnoty **`sem_wait()`**.
- Zrušení semaforu **`sem_destroy()`**.

```c
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

sem_t sem;

int quit=0;

void *
producer(void *arg)
{
        int i=0;
        while(!quit) {
                /* produce same data */
                printf("Sending data %d\n",i++);
                sem_post(&sem);
        }
}
```

```
1  void *
2  consumer(void *arg)
3  {
4          int i=0;
5          while(!quit) {
6                  /* wait for data */
7                  sem_wait(&sem);
8                  printf("Data ready %d\n",i++);
9                  /* consume data */
10         }
11 }
12
13 int
14 main(void)
15 {
16         pthread_t p, c;
17
18         sem_init(&sem, 0, 0);
19         pthread_create(&c, NULL, consumer, NULL);
20         pthread_create(&p, NULL, producer, NULL);
21
22         sleep(1);
23         quit = 1;
24         pthread_join(c, NULL);
25         pthread_join(p, NULL);
26         sem_destroy(&sem);
27 }
```

Ukázka části výstupu programu

```
1  Sending data 0
2  Sending data 1
3  Sending data 2
4  Sending data 3
5  Sending data 4
6  Sending data 5
7  Sending data 6
8  Sending data 7
9  Data ready 0
10 Data ready 1
11 Data ready 2
12 Data ready 3
13 Data ready 4
14 Data ready 5
15 Data ready 6
16 Data ready 7
17 Sending data 8
18 Sending data 9
19 Sending data 10
```

Podmíněné proměnné

- Synchronizace násobení matic
  - $A \times B \times C$
    1. Workery vynásobí $A \times B$
    2. Musí počkat
    3. Pokračují v násobení maticí C

Podmíněné proměnné

- Synchronizace pomocí podmínek:
  - A: Čekej na splnění podmínky
  - B: Oznam splnění podmínky

Podmíněné proměnné

- Základní rukojeť podmínky **`pthread_cond_t`**.
- Inicializace podmínky **`pthread_cond_init()`**.
- Čekání na podmínku **`pthread_cond_wait()`**.
- Signalizace splnění podmínky
  - **`pthread_cond_signal()`** – probudí alespoň jednoho čekatele.
  - **`pthread_cond_broadcast()`** – probudí všechny čekatele.
- Zrušení podmínky **`pthread_cond_destroy()`**.

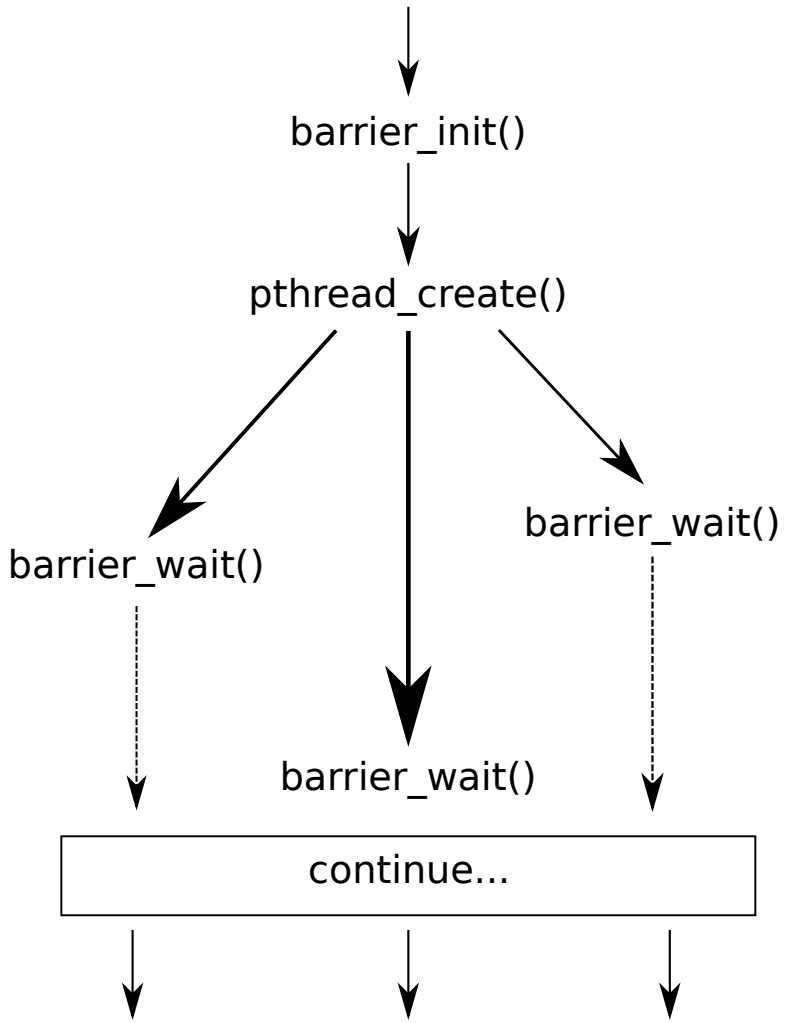
```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_cond_t condition;
pthread_mutex_t cond_lock;

void *
worker(void *arg)
{
        pthread_mutex_lock(&cond_lock);
        printf("Waiting for condition\n");
        pthread_cond_wait(&condition, &cond_lock);
        printf("Condition true\n");
        pthread_mutex_unlock(&cond_lock);
        return NULL;
}
```

```
18 int
19 main(void)
20 {
21         pthread_t p;
22
23         pthread_mutex_init(&cond_lock, NULL);
24         pthread_cond_init(&condition, NULL);
25
26         pthread_create(&p, NULL, worker, NULL);
27
28         sleep(1);
29         printf("Signaling condition\n");
30         pthread_mutex_lock(&cond_lock);
31         pthread_cond_signal(&condition);
32         pthread_mutex_unlock(&cond_lock);
33         printf("Condition done\n");
34
35         pthread_join(p, NULL);
36         pthread_cond_destroy(&condition);
37         pthread_mutex_destroy(&cond_lock);
38         return 0;
39 }
```

Bariéry

- Bariéry jsou v podstatě místa setkání.
- Bariéra je místo, kde se vlákna setkají.
- Bariéra zablokuje vlákno do doby než k bariéře dorazí všechna vlákna.
- Příklad:
  - Vláknové násobení matic: $M \times N \times O \times P$
  - Každé vlákno násobí a sčítá příslušný sloupec a řádek.
  - Po vynásobení $M \times N$ se vlákna setkají u *bariéry*.
  - Vynásobí předchozí výsledek $\times O$, opět se setkají u bariéry.
  - Dokončí výpočet vynásobením výsledku $\times P$.
- Ne všechny implementace POSIX threads poskytují bariéry!

Bariéry


barrier_init()
pthread_create()
barrier_wait()
barrier_wait()
barrier_wait()
continue...

Bariéry

- Datový typ **`pthread_barrier_t`**.
- Inicializace **`pthread_barrier_init()`**
  (Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Při inicializaci specifikujeme, pro kolik vláken bude bariéra sloužit.
- Zastavení na bariéře **`pthread_barrier_wait()`**.
- Zrušení bariéry **`pthread_barrier_destroy`**.

Příklad bariéry

```
1  #include <pthread.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  pthread_barrier_t barrier;
6
7  void *
8  foo(void *arg) {
9          int slp = (int)arg;
10         printf("Working..\n");
11         sleep(slp);
12         printf("Waiting on barrier\n");
13         pthread_barrier_wait(&barrier);
14         printf("Synchronized\n");
15         return NULL;
16 }
```

Příklad bariéry

```
17 int
18 main(void)
19 {
20         pthread_t t1, t2;
21
22         pthread_barrier_init(&barrier, NULL, 2);
23         pthread_create(&t1, NULL, foo, (void*)2);
24         pthread_create(&t2, NULL, foo, (void*)4);
25
26         pthread_join(t1, NULL);
27         pthread_join(t2, NULL);
28         pthread_barrier_destroy(&barrier);
29         return 0;
30 }
```

Read Write zámky

- Read Write zámky dovolují násobné čtení ale jediný zápis.
- Příklad:
  - Několik vláken čte nějakou strukturu (velmi často!).
  - Jedno vlákno ji může měnit (velmi zřídka!).
  - Pozorování:
    - Je zbytečné strukturu zamykat mezi čtecími vlákny
      Nemohou ji měnit a netvoří tedy kritickou sekci.
    - Je nutné strukturu zamknout, mění-li ji zapisovací vlákno
      V této chvíli nesmí strukturu ani nikdo číst (není změněna atomicky).

Read Write zámky

- Nastupují Read Write zámky.
- Pravidla:
  - Není-li zámek zamčen v režimu *Write*, může být libovolněkrát zamčen v režimu *Read*.
  - Je-li zámek zamčen v režimu *Write*, nelze jej už zamknout v žádném režimu.
  - Je-li zámek zamčen v režimu *Read*, nelze jej zamknout v režimu *Write*.
- Opět ne všechny implementace POSIX threads implementují RW zámky (korektně)!

RW zámky

- Datový typ **`pthread_rwlock_t`**.
- Inicializace **`pthread_rwlock_init()`**
  (Inicializaci je vhodné provádět ještě před vytvořením vlákna).
- Zamknutí v režimu *Read* **`pthread_rwlock_rdlock()`**.
- Zamknutí v režimu *Write* **`pthread_rwlock_wrlock()`**.
- Opakované zamčení jednoho zámku stejným vláknem skončí chybou **`EDEADLK`**.
  Není možné povýšit *Read* zámek na *Write* zámek a naopak.
- Odemknutí v libovolném režimu **`pthread_rwlock_unlock()`**
  Pthreads nerozlišují odemknutí dle režimů, některé implementace vláken párují rdlock s příslušným rdunlock, stejně tak pro wrlock.
- Zrušení rw zámku **`pthread_rwlock_destroy`**.

Příklad

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4
5  struct x_t {
6          int a;
7          int b;
8          pthread_rwlock_t lock;
9  };
10
11 struct x_t x;
12
13 int quit = 0;
14
15 pthread_barrier_t start;
```

Příklad

```
16 void *
17 reader(void *arg)
18 {
19         int n = (int)arg;
20         pthread_barrier_wait(&start);
21
22         while(!quit) {
23                 pthread_rwlock_rdlock(&x.lock);
24                 if((x.a + x.b)%n == 0)
25                         printf(".");
26                 else
27                         printf("+");
28                 pthread_rwlock_unlock(&x.lock);
29                 fflush(stdout);
30                 sleep(1);
31
32         }
33         return NULL;
34 }
```

Příklad

```
35
36 void *
37 writer(void *arg)
38 {
39         int i;
40         pthread_barrier_wait(&start);
41         for(i=0; i < 10; i++) {
42                 pthread_rwlock_wrlock(&x.lock);
43                 x.a = i;
44                 x.b = (i % 2)+1;
45                 pthread_rwlock_unlock(&x.lock);
46                 sleep(5);
47         }
48         quit = 1;
49         return NULL;
50 }
```

Příklad

```
52
53 int
54 main(void)
55 {
56         pthread_t t1, t2, t3;
57
58         x.a = 1;
59         x.b = 2;
60         pthread_rwlock_init(&x.lock, 0);
61         pthread_barrier_init(&start, NULL, 3);
62         pthread_create(&t1, NULL, reader, (void*)2);
63         pthread_create(&t2, NULL, reader, (void*)3);
64         pthread_create(&t3, NULL, writer, NULL);
65         pthread_join(t1, NULL);
66         pthread_join(t2, NULL);
67         pthread_join(t3, NULL);
68         pthread_rwlock_destroy(&x.lock);
69         pthread_barrier_destroy(&start);
70         return 0;
71 }
```

Problémy RW zámků

- Nebezpečí stárnutí zámků.
- Pokud je zamčená část kódu vykonávána déle než nezamčená, nemusí se nikdy podařit získat některý ze zámků.
- V předchozím příkladě nesmí být **`sleep()`** v zamčené části kódu!

```
1         for(i=0; i < 10; i++) {
2                 pthread_rwlock_wrlock(&x.lock);
3                 x.a = i;
4                 x.b = (i % 2)+1;
5                 pthread_rwlock_unlock(&x.lock);
6                 sleep(5);
7         }
```

Barrier v Open MP

- Klasická bariéra, synchronizuje všechna vlákna na bariéře
- Syntaxe:
  `#pragma omp barrier`
- Posloupnost paralelních sekci a bariér musí být stejná pro všechna vlákna
- Příkazy **`single`** a **`master`** nemají implicitní bariéru na vstupu a výstupu!

Podmínky v C++11

- Třída **`condition_variable`**
- **`#include <condition_variable>`**
- Metody
  - **`wait`** – odpovídá **`pthread_condition_wait`**
  - **`notify_all`** – odpovídá **`pthread_condition_broadcast`**
  - **`notify_one`** – odpovídá **`pthread_condition_signal`**

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4  #include <condition_variable>
5
6  std::mutex mtx;
7  std::condition_variable cv;
8  bool ready = false;
9
10 void print_id (int id)
11 {
12         std::unique_lock<std::mutex> lck(mtx);
13         while (!ready) cv.wait(lck);
14         std::cout << "thread␣" << id << '\n';
15 }
16
17 void go()
18 {
19         std::unique_lock<std::mutex> lck(mtx);
20         ready = true;
21         cv.notify_all();
22 }
```

```
1  int main ()
2  {
3          std::thread threads[10];
4          for (int i=0; i<10; ++i)
5                  threads[i] = std::thread(print_id,i);
6
7          std::cout << "10␣threads␣ready␣to␣race...\n";
8          go();
9
10         for (auto& th : threads) th.join();
11
12         return 0;
13 }
```