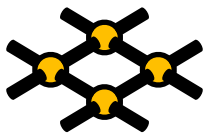
Vláknové programování
část VII

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



Laboratoř pokročilých síťových technologií

PV192
2014–04–08

Přehled přednášky

Ukončování

Ukončování

- Dvě varianty ukončení:
  - Samotným vláknem
    - **`pthread_exit()`**.
    - Návrat z hlavní funkce vlákna.
  - Jiným vláknem
    - **`pthread_kill()`**
    - **`pthread_cancel()`**

pthread_cancel()

- **pthread_cancel()** pošle danému vláknu notifikaci, aby se ukončilo.
- Vlákna mohou mít nastaveny dva různé typy kancelace:
  - **PTHREAD_CANCEL_DEFERRED** – vlákno je ukončeno pouze v tzv. kancelačních bodech (default).
  - **PTHREAD_CANCEL_ASYNCHRONOUS** – vlákno je ukončeno okamžitě.
- Dále vlákna mohou kancelaci odmítnout **PTHREAD_CANCEL_DISABLE**, opětovně přijmout kancelaci jde pomocí **PTHREAD_CANCEL_ENABLE**.
- Typy kancelace nastavíme pomocí **pthread_setcanceltype()**.
- Přijmout/odmítnout kancelaci lze pomocí **pthread_setcancelstate()**.

Kancelační body

- Kancelační bod je volání funkce, ve které může být vlákno ukončeno, je-li typu **`PTHREAD_CANCEL_DEFERRED`**.
- Základní kancelační body jsou:
  - **`pthread_testcancel()`** – pouze zjistí, zda nebylo signalizováno *cancel*
  - **`pthread_setcancelstate()`** – pokud měníme stav z **`PTHREAD_CANCEL_DISABLE`** na **`PTHREAD_CANCEL_ENABLE`**, je volání kancelačním bodem.
- Další kancelační body:
  - **`pthread_cond_wait()`**, **`pthread_cond_timedwait()`**, **`pthread_join()`**, **`sem_wait()`** (pouze z knihovny pthreads, pokud je poskytnuta knihovnou libc, není to kancelační bod!).
  - Většina funkcí **`libc`** (zejména I/O funkce), je vhodné konzultovat dokumentaci.

Příklad na kancelaci

```
1  #include <pthread.h>
2
3  void *
4  foo(void *arg)
5  {
6          int old;
7          pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &old);
8          while(1) {
9                  pthread_testcancel();
10         }
11         return NULL;
12 }
13
14 int
15 main()
16 {
17         pthread_t t;
18
19         pthread_create(&t, NULL, foo, NULL);
20
21         pthread_cancel(t);
22
23         return 0;
24 }
```

Cleanup Push/pop

- Co dělat v případě, že vlákno, kterému posíláme cancel, zrovna drží nějaký zámek?
- **`pthread_testcancel()`** rovnou vlákno ukončí, nelze použít pro test a případně zámek odemknout.
- Push/pop
  - Vlákno má zásobník funkcí, které se mají provést v případě kancelace.
  - **`pthread_cleanup_push()`** přidá specifikovanou funkci na vrchol zásobníku.
  - **`pthread_cleanup_pop()`** odebere funkci z vrcholu zásobníku (lze říct, zda funkci rovnou provést).
  - Některé implementace pthreads hlídají párování push/pop pomocí maker a ke každému push v každé funkci musí být odpovídající pop!

Příklad na cleanup

```c
#include <pthread.h>

pthread_mutex_t lock;

void *
foo(void *arg)
{
        int old;
        pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &old);
        pthread_cleanup_push(pthread_mutex_unlock, &lock);
        pthread_mutex_lock(&lock);
        while(1) {
                pthread_testcancel();
        }
        pthread_cleanup_pop(1); /*execute unlock*/
        return NULL;
}

int
main()
{
        pthread_t t;

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

        pthread_cancel(t);
        return 0;
}
```

Atributy funkcí pthread knihovny

Start vlákna

- **`pthread_create()`** funkci můžeme předávat atributy pro nově vytvářené vlákno.
- Atributy ovlivňují tři základní oblasti:
  - Osamostatnění vlákna
  - Nastavování priorit plánovače
  - Nastavení zásobníku
- Datový typ atributu **`pthread_attr_t`**.
- Inicializace **`pthread_attr_init()`**.
- Zrušení **`pthread_attr_destroy()`**.

Start vlákna – osamostatnění

- Osamostatněné vlákno uvolní všechny své zdroje jakmile skončí.
- Neosamostatněné vlákno je uvolní až při zavolání **`pthread_join()`**.
- Implicitně je každé vlákno neosamostatněné.
- **`pthread_attr_setdetachstate()`** nastaví vlákno osamostatněné (**`PTHREAD_CREATE_DETACHED`**) nebo neosamostatněné (**`PTHREAD_CREATE_JOINABLE`**).

Příklad osamostatnění

```
1  #include <pthread.h>
2
3  void *
4  foo(void * arg)
5  {
6          return NULL;
7  }
8
9  int
10 main(void)
11 {
12         pthread_t t;
13         pthread_attr_t attr;
14
15         pthread_attr_init(&attr);
16         pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
17
18         pthread_create(&t, &attr, foo, NULL);
19         pthread_attr_destroy(&attr);
20         return 0;
21 }
```

Nastavení zásobníku

- Implicitní velikost zásobníku pro vlákno je v Linuxu 8 MB.
- Chceme-li vytvořit 1000 vláken, potřebovali bychom 8 GB paměti jen pro zásobníky vláken.
- Pthread knihovna umožňuje změnit velikost zásobníku pro vlákno.
- **`pthread_attr_setstacksize()`**.
- Je nutné nastavit velikost zásobníku tak, aby se na něj vešly lokání proměnné všech funkcí, které se po sobě mohou zavolat. V opačném případě obdržíme signál **`SIGSEGV`** při vstupu do funkce, jejíž proměnné se na zásobník už nevlezou. Chyba vypadá na první pohled dost záhadně!

Příklad nastavení zásobníku

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  void*
5  foo(void* arg)
6  {
7          return NULL;
8  }
9
10 int
11 main(void)
12 {
13         pthread_attr_t attr;
14         pthread_t t;
15
16         pthread_attr_init(&attr);
17
18         pthread_attr_setstacksize(&attr, 65536);
19
20         pthread_create(&t, &attr, foo, NULL);
21
22         pthread_join(t, NULL);
23         pthread_attr_destroy(&attr);
24
25         return 0;
26 }
```

Afinita

Architektura systému

- Vývoj
  - Single procesor
  - SMP systémy (symetrický multiprocesing – více rovnocenných procesorů)
    - Obvykle společná paměť
    - Všechny procesory mají do paměti „stejně daleko“
  - NUMA systémy (více procesorů, nejsou rovnocenné)
    - Procesory mají lokální paměť
    - Přístup do ne-lokální paměti přes ostatní CPU
  - SMT systémy (symetrický multithreading – procesory mají více jader)

Architektura systému

- Procesory mají lokální cache
- SMT systémy mívají lokální cache pro jádro a společnou cache pro více jader

Nastavení afinity

- Vlákno může být obecně plánováno na libovolný procesor
- Nemusí být vždy vhodné
- Migrace mezi procesory bývá poměrně drahá
- Pokud jde o výkon aplikace, můžeme chtít zabránit migraci procesů/vláken
- Úskalí ve statickém přiřazení procesů/vláken na procesor
- Afinita – nastavení množiny procesorů, na kterých má proces/vlákno běžet

Nastavení afinity

- Nastavení afinity procesů
  - **`sched_setaffinity();`**
  - **`sched_getaffinity();`**
  - Nelze použít pro vlákna
- Nastavení afinity vláknům
  - **`pthread_setaffinity_np();`**
  - **`pthread_getaffinity_np();`**
- Svázání procesů/vláken a CPU je realizováno pomocí CPU SET

CPU SET

- Množina typu **cpu_set_t** do níž přidáváme/odebíráme CPU
- CPU číslujeme od 0
- **cpu_set_t *CPU_ALLOC();**
- **void CPU_ZERO_S();**
- **void CPU_SET_S();**
- **void CPU_CLR_S();**
- **int CPU_ISSET_S();**
- **void CPU_COUNT_S();**
- Logické operace mezi dvěma cpu_set_t: AND, OR, XOR, EQUAL

Příklad

```
1  #define _GNU_SOURCE
2  /* Musi byt jako prvni pred vsemi ostatnimi include */
3  #include <pthread.h>
4  #include <sched.h>
5  #define NUM_CPU 8
6
7  int
8  main()
9  {
10         cpu_set_t * set;
11
12         set = CPU_ALLOC(NUM_CPU);
13
14         CPU_ZERO_S(NUM_CPU, set);
15
16         CPU_SET(0, set);
17
18         pthread_setaffinity_np(pthread_self(), NUM_CPU, set);
19
20         CPU_FREE(set);
21
22         return 0;
23 }
```

Afinita před spuštěním aplikace

- Nastavení afinity u hotové aplikace
- **`numactl(8), taskset(1)`**
- Příklad

```
numactl -cpubind=0 aplikace
taskset 0x1 aplikace
```

  - Pustí aplikaci výhradně na CPU 0

Thread specific data

Thread-Specific Data

- Řada nástrojů pro paralelní běhy vláken umožňuje vytvořit privátní datovou oblast vlákna – TLS (Thread local storage).
- TLS je využito například knihovnou OpenGL (i když poněkud nešťastně) pro uchování kontextu.
- TLS je poskytnuto Javou, některými C/C++ variantami (GNU C, Intel C/C++, Visual C++, a další), C#, Python, Dephi.

TLS v Pthreads

- Princip použití:
  - Vytvoření klíče (s volitelným destruktorem).
  - Svázání klíče s nějakými daty.
  - Vyhledání dat podle klíče.
- Klíč je globální pro všechna vlákna daného procesu.
- Vazba dat na klíč je pro každé vlákno separátní.

TLS v Pthreads

- Datový typ klíče **`pthread_key_t`**.
- Vytvoření klíče **`pthread_key_create()`**.
  - Při vytváření klíče je možné specifikovat destruktor, který se zavolá v případě ukončení vlákna.
- Svázání dat a klíče **`pthread_setspecific()`**.
- Vyhledání dat dle klíče **`pthread_getspecific()`**.
- Zrušení klíče **`pthread_key_delete()`**.

Příklad na TLS

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  pthread_key_t key;
6
7  void
8  msg(char *m)
9  {
10         char *buff = pthread_getspecific(key);
11         sprintf(buff, "%s\n", m);
12         printf(buff);
13 }
14
15 void *
16 runner(void *arg)
17 {
18         char *array;
19         int i;
20
21         array = malloc(20);
22         pthread_setspecific(key, array);
23         for(i = 0; i < 10; i++) {
24                 msg(arg);
25         }
26         return NULL;
27 }
```

Příklad na TLS

```
27
28 int
29 main(void)
30 {
31         pthread_t t1, t2;
32
33         pthread_key_create(&key, free);
34
35         pthread_create(&t1, NULL, runner, "Hello");
36         pthread_create(&t2, NULL, runner, "Hello world");
37
38         pthread_join(t1, NULL);
39         pthread_join(t2, NULL);
40
41         pthread_key_delete(key);
42         return 0;
43 }
```

Jednodušší použití

- Použití pomocí klíčů je trochu těžkopádné
- GCC nabízí (neportabilní) direktivu **`__thread`**
- Použití:
  - **`__thread proměnná`**
  - **`__thread int x`**
  - Má zde význam slovo **`volatile`**?

Příklad

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  __thread int x=0;
5
6
7  void *
8  worker(void *arg) {
9          for(;x<1000000;x++) {
10                 asm volatile(""::"m" (x));
11         }
12         printf("X val: %d, addr %p\n", x, &x);
13 }
14
15 int main()
16 {
17         pthread_t t[2];
18
19         pthread_create(&t[0], NULL, worker, NULL);
20         pthread_create(&t[1], NULL, worker, NULL);
21         pthread_join(t[0], NULL);
22         pthread_join(t[1], NULL);
23         printf("X val: %d, addr %p\n", x, &x);
24 }
```

- Příklad výstupu:
  X val: 1000000, addr 0x7ff3966d470c
  X val: 1000000, addr 0x7ff395ed370c
  X val: 0, addr 0x7ff396e706fc

Open MP

Klauzule if, private, shared

- Základní formát
  `#pragma omp jméno-příkazu [klauzule] nový_řádek`
- **`if (výraz)`**
  - omp příkaz bude proveden paralelně, pokud je **`výraz`** vyhodnocen jako pravdivý, jinak je blok proveden sekvenčně
- **`private(list)`**
  - Úložiště objektu není asociováno s původní lokací
  - Všechny reference jsou k lokálnímu objektu
  - Nemá definovanou hodnotu při vstupu a výstupu
- **`shared(list)`**
  - Data jsou přístupná ze všech vláken v týmu
  - Všechna data jsou pro vlákna na stejných lokacích
  - Přístup k datům není synchronizován!

Pokročilé klauzule

- firstprivate, lastprivate
- default
- nowait

firstprivate, lastprivate

- **`firstprivate(seznam)`**
  - Proměnné v **`seznamu`** jsou inicializovány na hodnotu, kterou měl objekt před zahájením paralelní sekce
- **`lastprivate(seznam)`**
  - Vlákno vykonávající poslední iteraci smyčky nebo poslední sekci zapíše obsah proměnných v **`seznamu`** do původního objektu

```
1   int n, C, B, A = 10;
2  #pragma omp parallel
3  {
4    #pragma omp for private(i) firstprivate(A) lastprivate(B)...
5    for (i=0; i<n; i++)
6    {
7        /*-- A undefined, unless declared first private */
8        B = A + i;
9    }
10   /*-- B undefined, unless declared lastprivate */
11   C = B;
12 }
```

default

- **`default(none | shared)`**
- **`none`**
  - Žádné výchozí nastavení
  - Všechny proměnné je potřeba explicitně určit jako **`private`** nebo **`shared`**
- **`shared`**
  - Všechny proměnné jsou implicitně **`shared`**
  - Výchozí stav, pokud není přítomna **`default`** klauzule
- Fortran podporuje navíc: **`default(private | firstprivate)`**

nowait

- Za účelem minimalizace synchronizace, některé Open MP příkazy podporují volitelnou **`nowait`** klauzuli
- Pokud je přítomna, vlákna nejsou synchronizována (nečekají) na konci paralelního bloku

Základy ladění aplikací

GDB

- Kompilace s podporou GDB: **`gcc -g -o a.out foo.c -pthread -D__REENTRANT`**
- Spuštění pro ladění: **`gdb a.out`**
- Breakpoint – místo, kde je zastaven běh programu
  - **`br main`** – breakpoint na funkci main, tj. gdb zastaví na začátku programu
  - **`br foo.c:10`** – breakpoint v souboru foo.c na řádku 10
  - **`del 1`** – smaže první breakpoint
  - **`del`** – smaže všechny breakpointy
- Běh programu: **`(gdb) run [parametry]`** – spustí nahraný program **`a.out`** s případnými parametry
  - Běh se zastaví a) na breakpointu, b) skončením programu, c) přerušením ctrl-c nebo jiným signálem
- **`help příkaz`** zobrazí nápovědu k zadnému příkazu

- Pohybování se mezi funkcemi (po zastavení běhu)
  - **`list`** – vypíše okolí místa zastavení
  - **`up`** – posune se do volající funkce (v sekvenci volání směrem k prvotní funkci **`main()`**)
  - **`down`** – posune se do volané funkce (směrem od prvotní funkce **`main()`**)
  - **`where`** – vypíše sekvenci volání od **`main()`** po aktuální funkci, kde byl přerušen běh
- Sledování proměnných
  - **`print [promenná]`** vypíše jednorázově obsah proměnné, lze používat přetypování, dereference, např. **`print (struct *foo)x->next`**.
  - Je-li použita volba **`-O2`** při překladu, nemusí některé proměnné existovat – optimalizace je nahradily. Pro přístupné proměnné je nutné použít **`-O0`**.
  - **`display [promenná]`** bude vypisovat obsah proměnné po každém příkazu pro gdb.

- Příkazy pro trasování programu
  - **`n [enter]`** krok dál, je-li funkce, provede se a krok skončí za ní
  - **`s [enter]`** krok dál, je-li funkce, vstoupí se do ní
  - Příkazy **`n`** a **`s`** není nutné stále psát, **`[enter]`** automaticky opakuje poslední příkaz

Ladění aplikací

- Ladící výpisy
- Debugger

Ladící výpisy

- Pozor na mixování výpisů jednotlivých vláken do sebe.
- Jeden print je obvykle atomický.
- **`getpid()`** vrací pro vlákna stejnou hodnotu.
- **`pthread_self()`** vrací identifikaci vlákna (**`pthread_t`** – lze vypsat jako integer).
- Ladící výpisy způsobují určitou synchronizaci!

Debugger

- Použití *gdb*:
  - **`info threads`** – vypíše základní informace o běžících vláknech.
  - **`thread ID`** – přepnutí se na konkrétní vlákno.
- Debugger způsobuje velkou synchronizaci!

Ladění aplikací

```
(gdb) info threads
  2 Thread 0x40da4950 (LWP 12809)  0x00007f9f852c7b99 in
pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
* 1 Thread 0x7f9f856de6e0 (LWP 12806)  0x00007f9f84ff8b81 in nanosleep ()
   from /lib/libc.so.6
(gdb) thread 2
[Switching to thread 2 (Thread 0x40da4950 (LWP 12809))]#0  0x00007f9f852c7b99
in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
(gdb) where
#0  0x00007f9f852c7b99 in pthread_cond_wait@@GLIBC_2.3.2 ()
   from /lib/libpthread.so.0
#1  0x0000000000400907 in worker (arg=0x0) at conditions.c:13
#2  0x00007f9f852c33f7 in start_thread () from /lib/libpthread.so.0
#3  0x00007f9f85032b2d in clone () from /lib/libc.so.6
#4  0x0000000000000000 in ?? ()
(gdb)
```

Ladění aplikací

- **`valgrind`** – ladící nástroj
- **`helgrind`** – režim **`valgrindu`**
- Použití: **`valgrind --tool=helgrind aplikace`**
- Detekuje
  - Chybné použití knihovny pthreads
  - Nekonzistentní použití zámků
  - Některé nezamknuté přístupy ke sdíleným datům (data races)

Ladění aplikací

```
1  ==20556== Possible data race during read of size 4 at 0x601040 by thread #3
2  ==20556==    at 0x400630: foo (critsec1.c:10)
3  ==20556==  This conflicts with a previous write of size 4 by thread #1
4  ==20556==    at 0x4006D9: main (critsec1.c:24)
5  ==20556==
6  ==20556== Possible data race during write of size 4 at 0x601040 by thread #3
7  ==20556==    at 0x40064C: foo (critsec1.c:12)
8  ==20556==  This conflicts with a previous write of size 4 by thread #1
9  ==20556==    at 0x4006D9: main (critsec1.c:24)
10 ==20556==
11 ==20556== Possible data race during read of size 4 at 0x601040 by thread #2
12 ==20556==    at 0x400643: foo (critsec1.c:12)
13 ==20556==  This conflicts with a previous write of size 4 by thread #4
14 ==20556==    at 0x40064C: foo (critsec1.c:12)
```

Ladění aplikací

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4  #include <math.h>
5
6  void * test_prime(void *arg) {
7          long i,j, prime = *(long*)arg;
8          int sieve[prime];
9          for(i = 0; i < prime; i++) sieve[i] = 1;
10         for(i = 2; i < prime; i++) {
11                 if(sieve[i]) {
12                         printf("%ld, ", i);
13                         for(j=i*2; j < prime; j+=i) sieve[j] = 0;
14                 }
15         }
16         printf("\n"); return NULL;
17 }
18
19 int main(int argc, char *argv[]){
20         if(argv[1] == NULL) {
21                 printf("usage: test prime\n"); return 1;
22         } else {
23                 long test = atoi(argv[1]);
24                 test_prime(&test);
25         }
26         return 0;
27 }
```

Ladění aplikací

```
1  pthread_mutex_t lock_x, lock_y, lock_z;
2  volatile int x = 20, y = 10, z = 0, quit = 0;
3
4  void * foo(void *arg) {
5          while(!quit) {
6                  if(*(int*)arg == 1) { // z = x + y
7                          pthread_mutex_lock(&lock_x);
8                          pthread_mutex_lock(&lock_y);
9                          pthread_mutex_lock(&lock_z);
10                         z = x + y;
11                         pthread_mutex_unlock(&lock_z);
12                         pthread_mutex_unlock(&lock_y);
13                         pthread_mutex_unlock(&lock_x);
14                 }
15                 if(*(int*)arg == 2) { // z = y + x
16                         pthread_mutex_lock(&lock_y);
17                         pthread_mutex_lock(&lock_x);
18                         pthread_mutex_lock(&lock_z);
19                         z = y + x;
20                         pthread_mutex_unlock(&lock_z);
21                         pthread_mutex_unlock(&lock_x);
22                         pthread_mutex_unlock(&lock_y);
23                 }
24         }
25         return NULL;
26 }
```

Ladění aplikací

```
1  int
2  main() {
3          pthread_t t1, t2;
4          int asoc1=1, asoc2=2;
5
6          pthread_create(&t1, NULL, foo, &asoc1);
7          pthread_create(&t2, NULL, foo, &asoc2);
8
9          sleep(10);
10         quit = 1;
11
12         pthread_join(t1, NULL);
13         pthread_join(t2, NULL);
14         return 0;
15 }
```

Open GL

OpenGL

- Průmyslový standard specifikující multiplatformní rozhraní pro tvorbu aplikací počítačové grafiky
- Existuje v řadě verzí od 1.0 po poslední 4.0
- Aplikace využívající OpenGL je schopna ±běžet na různém HW
  - OpenGL podporuje různá rozšíření, která spolu se změnami verzí zhoršují poratabilitu
  - Aplikace musí umět detekovat co daná platforma nabízí
- Rasterizaci objektů provádí obvykle HW, existují ale i SW razterizátory (Mesa)

OpenGL pipeline

- OpenGL funkce jsou asynchronní
- OpenGL je stavová
- Návrh scény provádíme:

```
1 glBegin( GL_POLYGON );        /* Begin issuing a polygon */
2 glColor3f( 0, 1, 0 );         /* Set the current color to green */
3 glVertex3f( -1, -1, 0 );      /* Issue a vertex */
4 glVertex3f( -1, 1, 0 );       /* Issue a vertex */
5 glVertex3f( 1, 1, 0 );        /* Issue a vertex */
6 glVertex3f( 1, -1, 0 );       /* Issue a vertex */
7 glEnd();                      /* Finish issuing the polygon */
```

- Problematické při použití vláken

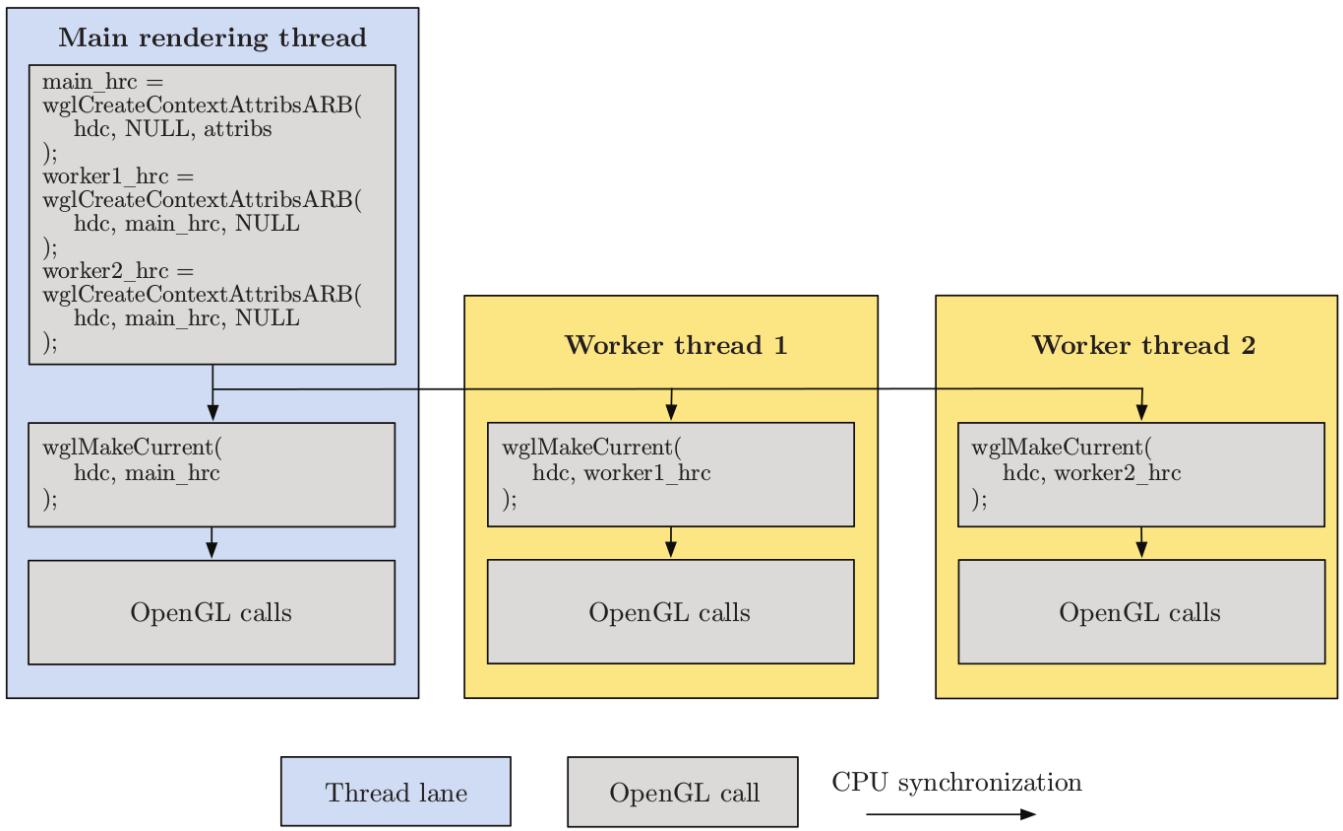
OpenGL kontext

- Kontex OpenGL není zachycen OpenGL specifikací, je tedy implementačně závislý
- Kontex uchovává stav a další informace pro rasterizér
- Existují rozšíření OpenGL pro manipulaci s kontexty
  - WGL – **`wglCreateContext`**, **`wglMakeCurrent`**, **`wglDeleteContext`**
  - GLX – **`glXCreateNewContext`**, **`glXMakeCurrent`**, **`glXDestroyContext`**
  - Pozor na použití glXDestroyContext – OpenGL je asynchronní!
- Pouze jeden kontext může být aktivní v rámci 1 vlákna
- Kontext je často uložen v TLS

OpenGL kontext a vlákna

- S implicitním kontextem
  - OpenGL na MacOS dovoluje přistup k GL funkcím z více vláken
  - Windows ohlásí resource busy
  - Linux (s Nvidia drivery) končí segmentation fault
  - Obecně je u vláken s implicitním kontextem problém, že kontext má právě master vlákno

OpenGL kontext pro Windows



OpenGL kontext pro Linux

```
1  void *
2  foo(void *ctx)
3  {
4          GLXContext mainCtx = (GLXContext)ctx;
5
6          GLXContext myCtx = glXCreateContext(dpy, vi, mainCtx, GL_TRUE);
7
8          glXMakeCurrent (dpy, win, ctx);
9
10         glClearColor (sl/6.0, sl/6.0, 1, 1);
11         glClear (GL_COLOR_BUFFER_BIT);
12         glXSwapBuffers (dpy, win);
13         glFlush();
14         XFlush(dpy);
15 }
```

OpenGL kontext pro Linux

```
1  int
2  main()
3  {
4          [...]
5          GLXContext ctx = glXCreateContext(dpy, vi, 0, GL_TRUE);
6
7          glXMakeCurrent (dpy, win, ctx);
8
9          pthread_create(&t1, NULL, foo, ctx);
10         pthread_create(&t2, NULL, foo, ctx);
11         pthread_create(&t3, NULL, foo, ctx);
12
13         pthread_join(t1, NULL);
14         pthread_join(t2, NULL);
15         pthread_join(t3, NULL);
16
17         ctx = glXGetCurrentContext();
18         glXDestroyContext(dpy, ctx);
19 }
```

OpenGL kontext a vlákna

- Alternativní změna kontextu na jiné než master vlákno
  - Při použití **libSDL** triviálně tak, že zavoláme **SDL_init()** z ne-master vlákne, o zbytek se postará **libSDL**
  - Implicitní kontext vytváří GLX rozšíření Xserveru samo o sobě
  - Jediná cesta pro GLX je reload GLX z vlákna, které má kreslit
  - Návod lze najít např. ve zdrojových kódech **libSDL** **src/video/x11/SDL_x11gl.c**