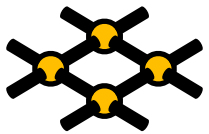
Vláknové programování
část I

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



Laboratoř pokročilých síťových technologií

PV192
2011–02–24

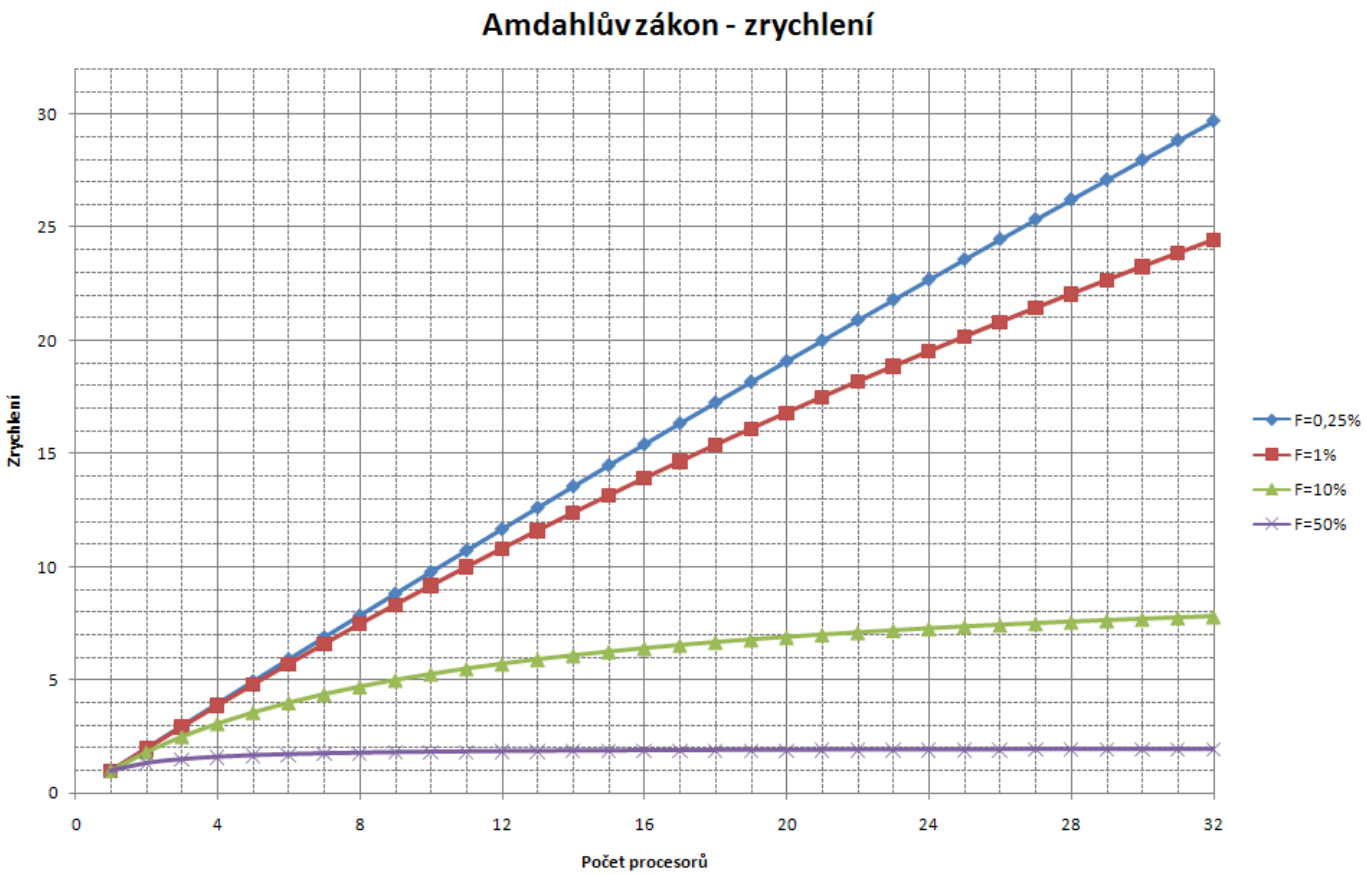
"For the past thirty years, computer performance has been driven by Moore's Law; from now in, it will be drivern by Amdahl's Law. Writing code that effectively exploits multiple processors can be very challenging. . ."

–*Doron Rajwan*, Research Scientist, Intel Corp.

Amdahlův zákon

$$\text{zrychlení} \leq \frac{1}{F + \frac{1-F}{N}}$$

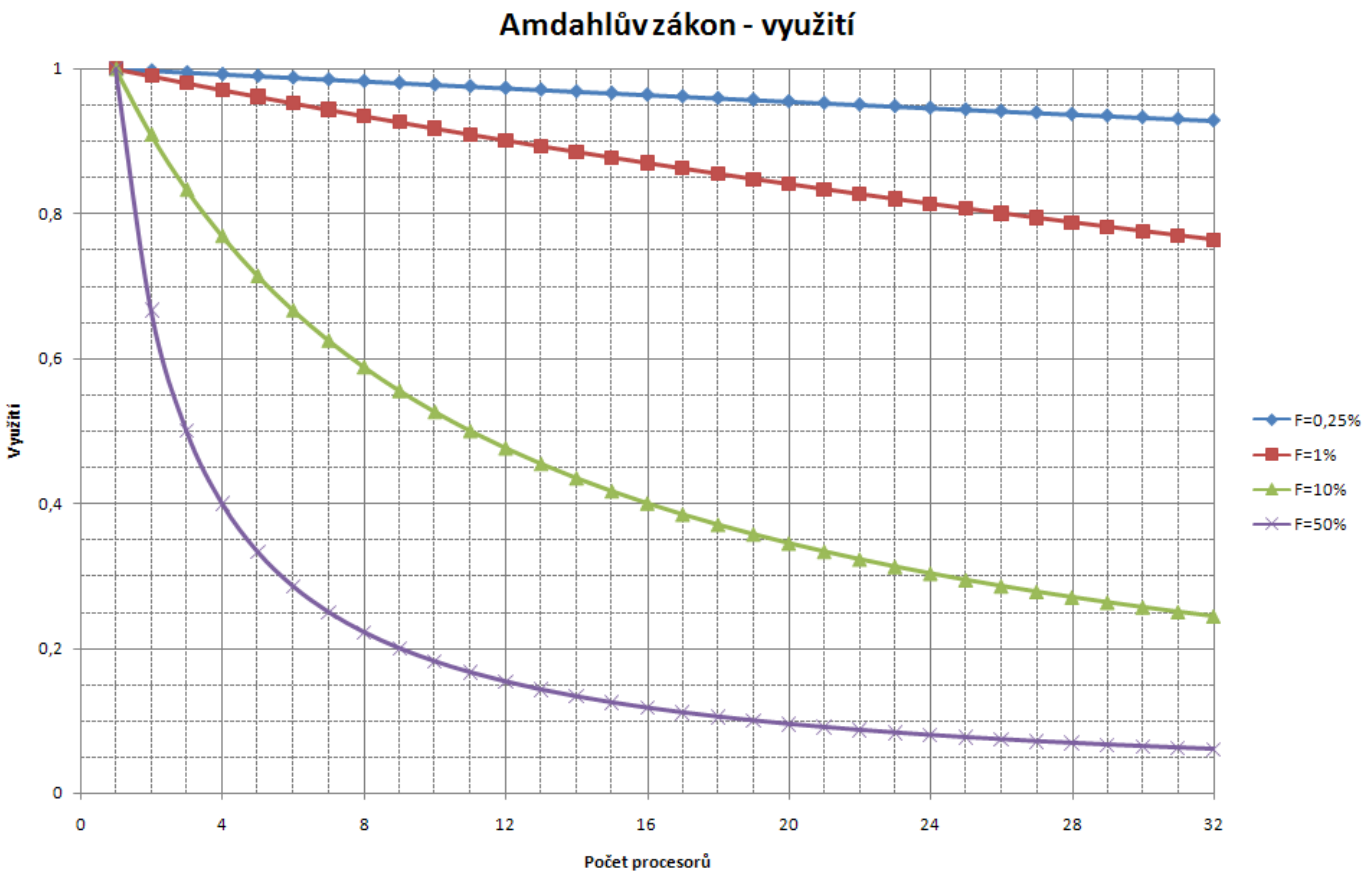
kde F je podíl sériově vykonávané práce a N je počet procesorů



Amdahlův zákon

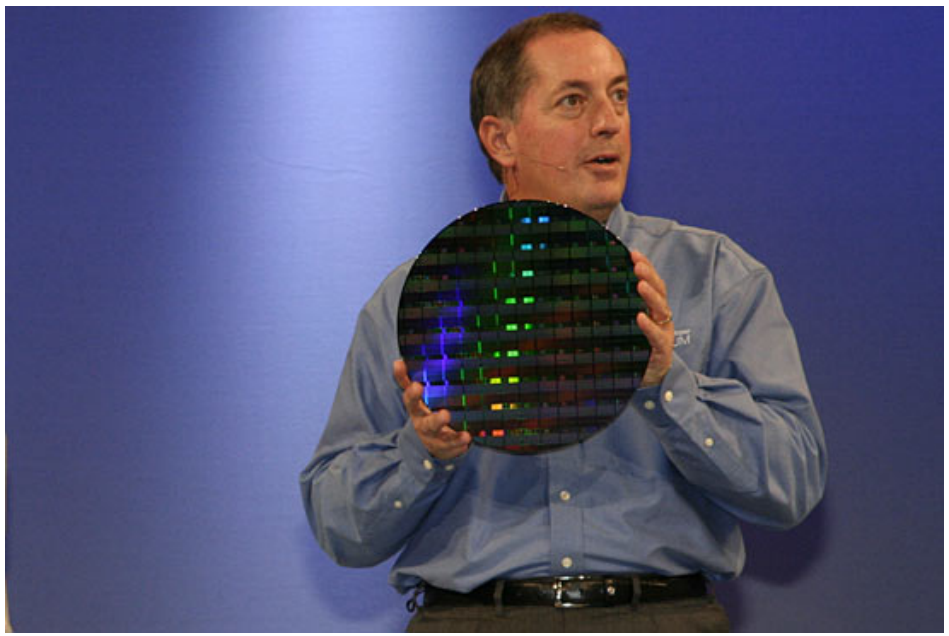
$$\text{zrychlení} \leq \frac{1}{F + \frac{1-F}{N}}$$

kde F je podíl sériově vykonávané práce a N je počet procesorů



80 jader na čipu

Intel na IDF 2006 předvedl prototyp procesoru s 80 jádry.
Výkon ≈ 1 TFLOPS



Zdroje:

http://www.news.com/Intel-shows-off-80-core-processor/2100-1006_3-6158181.html

Úvod do programování ve vláknech

- Programování v C s využitím Pthreads
- Programování v Javě
- Výlet do jiných jazyků: Ada
- Demonstrováno na praktických příkladech

Úvod do programování ve vláknech

- Programovaní v C s využitím Pthreads
  1. Procesy, vlákna, přepínání kontextu, knihovna pthreads, vznik a ukončení vláken, základy ladění aplikací
  2. Základy synchronizace: zámky, semafory, podmíněné proměnné, RCU struktury
  3. Pokročilé synchronizace: bariéry, rw zámky, pojmenované semafory, futexy
  4. Afinita, Atributy vláken, režimy startu vlákna, priority, ukončování vláken, thread-specific data
  5. OpenMP
  6. Práce s pamětí
  7. GUI, OpenGL, Futures a TPE v C++

Úvod do programování ve vláknech

- Programování v Javě
  1. Vlákna v jazyce Java, vytváření a ukončování. Viditelnost operací. Monitory a synchronizace. Signalizace a pozastavení.
  2. Paralelní datové struktury. Atomické typy. Ladění paralelních programů: uváznutí a jeho diagnostika, hladovění. Testování paralelních programů.
  3. Explicitní zamykání – RW zámky, vlastní typy zámků. Executory, thread pools, futures.
  4. Paměťový model Javy. Paralelismus a GUI.

Úvod do programování ve vláknech

- Výlet do jiných jazyků: Ada
  1. Úvod k jazyku Ada: základní rysy jazyka, syntaxe, datové typy.
  2. Podpora paralelismu v jazyce Ada: úlohy, chráněné objekty, monitory, podpora systémů v reálném čase.

Literatura

- Andrews, Gregory R. *Foundations of multithreaded, parallel, and distributed programming.* Addison-Wesley 2000.
- Ben-Ari M., *Principles of Concurrent and Distributed Programming.* 2nd Ed. Addison-Wesley, 2006
- Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. *Java Concurrency in Practice.* Addison Wesley Professional, 2006
- Butenhof D. R., *Programming with POSIX(R) Threads.* Addison-Wesley Professional, 1997
- Burns A., Wellings A. *Concurrency in Ada.* 2nd Ed. Cambridge University Press, 1998
  nebo
  Burns A., Wellings A. *Concurrent and Real-Time Programming in Ada.* Cambridge University Press, 2007

Přehled přednášky

Procesy

- Proces
  - Instance programu, která je sekvenčně prováděna.
  - Je to entita pro alokace zdrojů (procesor, paměť, atd)
  - Procesy tvoří stromovou hierarchii–vztah rodič potomek

> Sex is not really common among processes—each process has just one parent.

Procesy

- Typy procesů
  - Levný proces (Light Weight Process–LWP)
    - Levné procesy mezi sebou sdílí adresní prostor
    - Minimum privátních zdrojů
  - Drahý proces (Heavy Weight Process–HWP)
    - Drahé procesy jsou mezi sebou zcela izolované
    - Prakticky všechny zdroje jsou privátní

Procesy

- Popisovač procesu
  - Obsahuje informace o
    - Signálech
    - Přidělené paměti
    - Otevřených souborech
    - Aktuálním adresáři
    - HW kontext (obsah registrů, zásobník, ...) – TSS
    - Terminálu
    - Prioritě
    - Stav
    - ...

Procesy

- Vytvoření procesu
  - Proces vzniká rozštěpením rodiče
  - Po startu je potomek stejný jako rodič
    - Stejný obsah paměti (Copy on Write)
    - Vykonává stejný kód

Procesy

- Běh procesu
  - Vykonávání kódu programu
  - Dva režimy běhu
    - User space–kód samotného programu
    - Kernel space–kód jádra
  - Stav procesu
    - Running
    - Interruptible
    - Uninterruptible
      (Nezpracovává signály)
    - Stopped
    - Traced
  - Konkurence vs. paralelismus
    - Konkurence–vykonávání stejného nebo různého kódu více procesy, nemusí probíhat ve stejný čas
    - Paralelismus–konkurence probíhající ve stejný čas

Procesy

- Přepínání kontextu
  - Zásadní mechanismus multitaskingu
  - Mechanismus uložení a obnovení stavu CPU
  - Rozlišujeme přepnutí kontextu
    - Registrové (obsluha přerušení)
    - Vláknové (přepnutí na jiné vláko téhož procesu)
    - Procesové (přepnutí na jiný proces)

Procesy

- Kroky při přepínání kontextu
  - Uložení stavu CPU, obvykle do TSS
    - Všechny běžné registry, deskriptory segmentu, příznaky
    - Stav a registry FPU
  - Obnova adresního prostoru
  - Načtení nového stavu CPU

Procesy

- Softwarové vs. hardwarové přepínání kontextu
  - Kontext lze uložit a obnovit v softwaru (kopírování stavu)
  - Některé procesory podporují přepnutí kontextu v HW (architektura x86 od Intel 80386 a dál)
  - Linux od verze jádra 2.4 používá softwarové přepnutí kontextu
  - Softwarové i hardwarové přepnutí kontextu je velmi drahá operace!

Procesy

- Komunikace mezi procesy
  - Soubory
  - Signály (signal(7))
  - Sockets (socket(2))
  - Fronty zpráv (mq_overview(7))
  - Trubky (pipes), pojmenované vs. nepojmenované (pipe(2))
  - Semafory (sem_overview(7))
  - Sdílená paměť, paměťově mapované soubory (shm_overview(7), mmap(2))
  - Message passing (MPI knihovny)

Vlákna

- Proč vlákna?
  - Paralelismus
  - Výkon
  - Odezva
  - Komunikace

Vlákna

- Vlákno
  - Podmnožina procesu
  - Vlákno nemá vlastní adresní prostor
  - Typicky sdílí stav ostatními vlákny daného procesu
  - Shared-memory model:
    - Komunikace mezi vlákny je možná stejně jako u procesů a navíc i přes jejich sdílenou paměť
    - Oproti procesu jsou běžně sdílené globální a statické proměnné

Vlákna

- Vlákna stejného procesu sdílí
  - Kód programu
  - „Většinu“ dat
    - Novější koncepce vláken podporuje nesdílenou paměť–thread local storage (TLS)
  - Otevřené soubory (file descriptors)
  - Signály a obsluhu signálů
  - Současný pracovní adresář
  - Identifikaci uživatele a skupiny
  - Process ID (PID)
  - Pojmenované semafory, fronty zpráv a další nástroje IPC

Vlákna

- Každé vlákno má unikátní
  - Thread ID (identifikace vlákna)
  - Obsah registrů procesoru, ukazatel vrcholu zásobníku
  - Zásobník pro lokální proměnné a návratové adresy
  - Masku signálů
  - Prioritu
  - Hodnotu proměnné **`errno`** (dle POSIX.1c)

Vlákna

- Implementace vláken v operačním systému je různá
  - proces a vlákno není rozlišeno
    (vlákno je tedy procesem – Linux bez NPTL)
  - proces a vlákno jsou rozlišeny
    (vlákno se liší od procesu – Windows, Linux s NPTL)
  - vlákna v uživatelském prostoru
    (vlákna si řídí sám proces – Java Green Threads (obsolete), Erlang)

Vlákna

- Mapování vláken na plánovací entity v jádře
  - mapování vláken 1:1
    - současné produkční implementace
    - Linux, Windows, FreeBSD s libthr
  - mapování vláken N:M
    - \+ teoreticky nejefektivnější
    - − příliš složité, problémy s invertováním priorit, atd.
    - FreeBSD s Kernel Scheduler Entitites, experimenty i v Linuxu
  - mapování vláken N:1
    - zastaralý přístup, user-space threading
    - FreeBSD s libc_r

Vlákna

- Některé systémy mají „levné“ přepínání vláken a „drahé“ přepínání procesů (Windows NT, OS/2).
- Některé systémy příliš nerozlišují v přepnutí vlákna nebo procesu.
- Proč uvažujeme vlákna místo procesů?
  - Rychlejší přepnutí běžících vláken než běžících procesů.
  - Snadné sdílení paměti a dalších zdrojů mezi běžícími vlákny (někdy ovšem nevýhoda).

Knihovna Pthreads

- POSIX Threads (Pthreads) je POSIX standard pro vlákna
- Vlákna v systémech na bázi jádra Linux
  - Linux threads – neúplná implementace POSIX Threads
    - Vlákno bylo obsluhováno stejně jako proces, mělo i vlastní PID (process ID).
    - Není nutná speciální podpora jádra, problémy s výkonem, pokud se vlákno samo nevzdá procesoru (yield()).
  - Nahrazena NPTL – Native POSIX threads library
    - Výrazně vyšší výkonnost
    - Vlákno je samo o sobě jednotkou plánování, tj. procesový plánovač plánuje i vlákna obvykle úplně stejně.
    - NPTL potřebuje speciální podporu jádra pro synchronizaci.
- Pthreads knihovna má implementaci pro řadu systémů: Linux, *BSD, Windows, MacOS, ...

Knihovna Pthreads

- 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.

Knihovna 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;
}
```

Reentrantní funkce

- Vícenásobný běh programu má určitá úskalí.
- Kód funkcí musí počítat s tím, že může být prováděn několikrát najednou.
- Problematické jsou globální proměnné a statické lokální proměnné.

```c
char buffer_2[200];

char *
foo1(char * a)
{
        static char buffer_1[200];

        snprintf(buffer_1, 200, "Text: %s\n", a);

        return buffer_1;
}

char *
foo2(char *a)
{
        snprintf(buffer_2, 200, "Text: %s\n", a);

        return buffer_2;
}
```

- *Reentrantní funkce* – funkce schopná násobného běhu.
- Příklad funkce **`foo(char *)`** – implementace alokuje dynamický kus paměti.
- Makro **`__REENTRANT`** – je-li definováno, říkáme překladači a hlavičkovým souborům, že funkce mohou být vykonávány násobně.
- Knihovní funkce:
  - Thread safe – lze používat z více vláken
  - Not thread safe – nelze používat z více vláken
  - Některé funkce nemohou být thread safe z podstaty věci – **`strtok(3)`**
  - POSIX.1c rozšiřuje množinu funkcí o thread safe varianty, např. **`strtok_r(3)`**

Kompilace

- Dvě možnosti kompilace:
  - **`gcc -o foo foo.c -lpthread -D__REENTRANT`**
  - **`gcc -o foo foo.c -pthread -D__REENTRANT`**
- Nezapomínáme na to, že záleží na pořadí knihoven a objektových souborů na příkazové řádce.

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)
```