PB173 Perl

11 Asynchrónne programovanie

Roman Lacko xlacko1@fi.muni.cz

2023-07-17

Obsah

Vlákna

Vlákna

Modul ‹threads› je oficiálne **neodporúčaný**.

Proces v POSIX systéme obsahuje jedno alebo viacero vláken.
Tieto zdieľajú adresný priestor a zdroje.

Väčšina programovacích jazykov umožňuje mapovať systémové vlákna na nejaký mechanizmus (napríklad PThread).

Interpretované jazyky však majú s vláknami typicky problémy, hlavne kvôli integrite dát a uvoľňovaniu zdrojov.

Perl implementuje vlákna ako samostatné inštancie interpretu.

Vlákna: Vytvorenie

```
use threads;

my $thread = threads->create(\&fun, @arguments);
my $thread = threads->create(sub (@) { ... }, @arguments);
my $thread = async { BLOCK };
```

Vlákna: Obslužná funkcia

Volanie `threads->create` určuje kontext, v ktorom sa funkcia vyhodnotí.

```
my $thread = threads->create(...);       # Scalar
my ($thread) = threads->create(...);     # List

my $thread = threads->create({context => 'void'}, ...); # Explicit
```

Vlákna: Ukončenie

Vlákna je možné ukončiť vrátením hodnoty z obslužnej funkcie, `threads->exit(VALUE)` alebo vyhodením výnimky.

```
my $thread = threads->create('worker');

sub worker(@args) {
        return -1 if !@args;            # Implicit exit
        threads->exit(3) if @args == 2; # Explicit exit
        die "Fatal error";              # Exception
}
```

Vlákna: Ukončenie

Pozor na `exit()`
`exit()` ukončí celý program aj pri `use threads`.
To preto, aby sa to správalo konzistentne s POSIX Threads.

Toto správanie je možné vypnúť explicitne pragmou `threads`:

```
use threads exit => 'threads_only';
exit(5);                                # threads->exit(5)
```

Vlákna: Spájanie a odpájanie

```
my @results = $thread->join;    # Incompatible with detach
$thread->detach;                # Incompatible with join
```

Ak vlákno skončí s výnimkou, po `join` je možné chybu zistiť volaním `$thread->error`.

Stavy vlákna je možné zistiť predikátmi:

```
$thread->is_running
$thread->is_joinable
$thread->is_detached
```

Vlákna: Objekt vlákna

Niektoré operácie môže vlákno vykonať samo na sebe alebo iných vláknach pomocou *thread objects*.

```
threads->self           # Returns the thread object
threads->object($tid)   # Returns the thread object of other thread

threads->detach         # threads->self->detach
threads->tid            # threads->self->tid
```

Vlákna: Objekt vlákna

Objekty vlákna je možné medzi sebou porovnať metódou `equal`

```
$thread1->equal($thread2)
```

Funguje však aj porovnávanie operátormi `==` a `!=`.

Ak sa k `use threads` pridá `stringify`, bude vedieť aj konverziu na reťazec

```
use threads qw{stringify};

my $thread = threads->create('worker', @args);
say "Thread $thread started";           # Thread 25 started
```

Vlákna: Zoznamy

Pomocou metódy `list` je možné získať objekty bežiacich vláken.

```
threads->list
threads->list(threads::running)
threads->list(threads::joinable)
```

Vlákna: Synchronizácia

Vlákna môžu medzi sebou posielať signály metódou `kill`.

Vláknové signály v Perli nie sú IPC signály!
Je tak nimi možné chytiť napríklad aj `SIGKILL`.

```
my $thread = threads->create('worker', @args);
local $SIG{ALRM} = sub ($) { $thread->kill('SIGKILL'); };
alarm 10;
$thread->join;

sub worker(@args) {
        local SIG{KILL} = sub ($) { threads->exit(1); };
        ...
}
```

Vlákna: Zdieľanie premenných

Ak nemusíte, použite radšej `Thread::Queue` (ďalej).

Vo východzom stave sa medzi „vláknami“ žiadne hodnoty nezdieľajú.
Zdieľané hodnoty sa musia explicitne označiť *atribútom* `:shared`.

```
use threads;
use threads::shared;

my $var :shared;
shared(@array);
```

Zdieľané premenné môžu obsahovať len skaláry
alebo referencie na zdieľané hodnoty.

`bless` pod `threads::shared` propaguje požehnanie do iných vláken.

Vlákna: Zdieľanie premenných

```
shared_clone REF
```

- Vytvorí hlbokú zdieľanú kópiu hodnoty.

```
is_shared VARIABLE
```

- Otestuje, že premenná obsahuje zdieľanú hodnotu.

Vlákna: Synchronizácia

Modul `threads::shared` exportuje aj primitíva na synchronizáciu vláken.

```
lock VAR
```

Vytvorí zámok na **zdieľanej** premennej až do *konca rozsahu*.
Ak sa o zámok pokúsi viac vláken, pridelí sa jednému a zvyšok sa uspí.

```
cond_wait CONDVAR [, LOCKVAR]
cond_timedwait CONDVAR, TIMEOUT [, LOCKVAR]
cond_signal CONDVAR
cond_broadcast CONDVAR
```

Mechanizmy pre podmienkové premenné.

‹Thread::Semaphore›

Štandardný modul Perlu určený na riadenie prístupu k zdrojom.

```
my $s = Thread::Semaphore->new;

$s->down;
        # Guarded section
$s->up;
```

‹Thread::Semaphore›

Konštruktor môže brať počiatočnú hodnotu semafora inú než 1.
V takom prípade je možné použiť aj metódy:

```
$s->down(NUMBER);
$s->up(NUMBER);
$s->down_force(NUMBER);
```

Metóda ‹down› má navyše varianty ‹_nb› a ‹_timed›.

‹Thread::Semaphore› nemá RAII variant, ‹up› musí zavolať programátor.

‹Thread::Queue›

Štandardný modul Perlu na podávanie úloh vláknam.
Užitočné pre model producentov a konzumentov.

```
my $queue = Thread::Queue->new;

my $producer = threads->create(sub () {
        $queue->enqueue($_) for 1 .. 50;
        $queue->end;
});

my $consumer = threads->create(sub () {
        process($_) while $_ = $queue->dequeue;
});
```

Metóda ‹dequeue› má aj varianty ‹_nb› a ‹_timed›.

```
$queue->limit(N)
```

Obmedzí kapacitu radu; v takom prípade bude ‹enqueue› potenciálne blokujúca.

```
$queue->peek([INDEX])
```

Vráti prvok čakajúci v rade, ale neodstráni ho. **Pozor na súbeh!**

```
$queue->insert(INDEX, LIST)
$queue->extract(INDEX, [COUNT])
```

Manipulácia s radom na zadanom indexe.

Asynchrónne programovanie

Asynchrónne programovanie

„Asynchrónne programovanie“ môže mať v rôznych kontextoch jemne odlišný popis.

Asynchrónne programovanie je sada techník, ktorá umožňujú programu reagovať na očakávané alebo neočakávané notifikácie bez blokovania.

Ako príklad sme zatiaľ videli **signály**.

Asynchrónne programovanie

Asynchrónne programy môžu často naplánovať asynchrónne udalosti sami, napr. spustením asynchrónnej IO operácie.

Túto schopnosť je možné dosiahnuť typicky rôznymi mechanizmami na nižšej úrovni:

- Signály
- Vlákna (súbežnosť a paralelizmus)
- Korutiny (kooperatívne vlákna)
- Udalosťami riadené programovanie

Každý z týchto mechanizmov je možné použiť aj bez toho, aby sme mali za cieľ striktne asynchrónny program!

Korutiny

Kooperatívna súbežnosť je mechanizmus, pri ktorom je program rozdelený na samostatné časti („vlákna“), ktoré sa musia vzdať behu, aby mohli bežať iné časti.

Týmto častiam sa v niektorých jazykoch hovorí *korutiny*.

‹Coro› — The only real threads in Perl
— Manual page Coro

Rámcopráca na vytváranie korutín.

 Toto nie je štandardný modul.

```
use Coro;

async {
        say "2";
        cede;
        say "4";
};

say "1";
cede;
say "3";
cede;

# Says 1 2 3 4
```

‹Coro›: Vytvorenie

```
async CODE;
```

Vytvorí novú korutinu a pripraví ju k behu. Spustí sa, keď ju vyberie plánovač.

Zhruba ekvivalentné konštrukcii

```
sub async(CODE) {
    my $coro = Coro->new(CODE);
    $coro->ready;
    return $coro;
}
```

‹Coro›: Kooperácia

Kooperácia vyžaduje, aby sa korutiny niekedy vzdali behu:

- Použitím blokujúcej ‹Coro› operácie (napr. sieťová komunikácia).
- Explicitne pomocou ‹Coro::schedule› alebo ‹Coro::cede›.

‹Coro› si drží zoznam pripravených korutín, z ktorých prvú **vyberie** a spustí. Korutina sa musí pred ‹Coro::schedule› vložiť do zoznamu pripravených korutín, alebo musí použiť ‹Coro::cede›.

```
$Coro::current->ready;
Coro::schedule;
```

```
Coro::cede              # 'cede' is exported by default
Coro::cede_not_self     # cede to any other thread
```

‹Coro›: Kooperácia

Korutina sa môže vzdať aj v prospech inej konkrétnej korutiny:

```
$Coro::current->schedule_to($other_coro)
$Coro::current->cede_to($other_coro)
```

‹Coro› vždy obsahuje referencie na dve korutiny:

```
$Coro::main
$Coro::current
```

‹Coro›: Ukončenie

Korutina môže skončiť

- ukončením obslužnej funkcie,
- zavolaním ‹`Coro::terminate`›,
- zrušením z inej korutiny pomocou ‹`$coro->cancel`› alebo ‹`Coro::killall`›
  (deje sa automaticky, ak zanikne posledná referencia na korutinu).

‹Coro›: Návratový kód

```
$coro->join
```

Počká na skončenie korutiny a vráti jej návratovú hodnotu.

Na rozdiel od vláken môže `$coro->join` volať viac iných korutín naraz, po skončení dostanú všetky rovnakú návratovú hodnotu.

Udalosťami riadené programovanie

Procedurálne programovanie

- Program má „lineárny” priebeh, v ktorom sa na rôznych miestach rozhoduje, ktorým smerom bude pokračovať (`if`, `while`, ...).
- Model, ktorý sme používali doteraz.

Udalosťami riadené programovanie

- Model vhodný na aplikácie, ktoré často interagujú s používateľom alebo inými aplikáciami.
- Abstrakcia nad predchádzajúcim modelom — v programe rozoznávame „udalosti“ a reakcie na nich.

Udalosťami riadené programovanie

Udalosť (event) Všeobecný pojem pre vstup udalosťami riadeného programu (URP). Napr. klik na grafický prvok, dáta v rúre, paket, signál, ...

Obsluha udalosti (event handler) Kód, ktorý sa spustí, keď nastane nejaká udalosť. Rôzne udalosti môžu spustiť rôzne obsluhy, prípadne žiadne.

Cyklus udalostí (event loop) Jadro URP, ktoré čaká na udalosti a spúšťa reakcie.

Udalosťami riadené programovanie

Hoci cyklus udalostí nemusí byť nutne skutočný cyklus,
obvyklý vzor URP vyzerá napríklad takto:

```
my $event_loop = SomeEventLoopEngine->new(…);

while (my $event = $event_loop->wait) {
    if ($even->type eq 'signal') {
        handle_signal_event($event);
    } elsif ($event->type eq 'io') {
        handle_io_event($event);
    } …
}
```

‹select(3)›, ‹poll(3)›

Systémové volania POSIX na riadenie vstupno-výstupných udalostí.

- Je možné čítať deskriptor (rúru, soket, …).
- Je možné zapisovať do deskriptora (voľný buffer).
- Na deskriptore nastala chyba (odpojený druhý koniec rúry).

Perl moduly ‹`IO::Select`› a ‹`IO::Poll`›.

Perl má aj **veľmi** nízkoúrovňové volanie ‹`select`› so štyroma argumentami. Preferujte však modul, ktorý je oveľa pohodlnejší.

Pri práci s ‹IO::Select› a ‹IO::Poll› môže nastať niekoľko problémov.

Ako vieme, koľko bajtov môžeme prečítať alebo zapísať?

Deskriptor je vo východzom stave *blokujúci*, tj. ‹read(3)› a ‹write(3)› sa môžu zablokovať, ak nie je dosť dát resp. je plná vyrovnávacia pamäť.

Pre *neblokujúci* deskriptor tieto operácie skončia s chybou.

- V C ‹open(3)› alebo ‹fcntl(3)› s príznakom ‹O_NONBLOCK›.
- V Perli je každý deskriptor ‹IO::Handle› s metódou ‹blocking()›:

```
STDIN->blocking(0);     # fcntl(0, F_SETFL, O_NONBLOCK);

if (sysread(STDIN, $buffer, 4096) == 0
        && ($! == EAGAIN || $! == EWOULDBLOCK)) {
    # No data available yet.
}
```

Ako dáta prečítať?

Perl funkcie ‹readline›, ‹read› atď. používajú vyrovnávaciu pamäť, na neblokujúce deskriptory nie sú úplne pripravené.

‹sysread› a ‹syswrite› sú nízkoúrovňové funkcie (tiež dostupné aj ako metódy ‹IO::Handle›), ktoré obchádzajú vyrovnávacie pamäte.

```
my ($chunk, $bytes);

1 while ($bytes = sysread $handle, $chunk, 4096, length $chunk) > 0;

return "End of File"
    if $bytes == 0;
return "No more data for now"
    if $! == EAGAIN || $! == EWOULDBLOCK;
die "Error on handle: $!";
```

‹IO::Select›

```
use IO::Select;

my $s = IO::Select->new;

$s->add(\*STDIN);
$s->add($pipe);

while ($s->handles) {
    foreach my $handle ($s->can_read) {
        # Process input from $handle.
    }
}
```

```
use IO::Poll qw{POLLIN POLLOUT POLLHUP};

my $poll = IO::Poll->new;
$poll->mask($handle => POLLIN);          # Reading.
$poll->mask($pipe => POLLOUT);           # Writing.

while ($poll->handles > 0) {
    die "poll: $!" if !$poll->poll;
    foreach my $handle ($poll->handles) {
        my $ev = $poll->events($handle);
        # $ev & POLLIN  ≈ OK to read
        # $ev & POLLOUT ≈ OK to write
    }
}
```

Jednotné rozhranie pre rôzne mechanizmy URP.
IO, Signály, Časovače, Podprocesy, ...

Navyše je ľahko rozšíriteľný pomocou modulov i o ďalšie mechanizmy:

- Sieťová komunikácia (TCP, HTTP, IRC, FTP, Discord, ...)
- Grafické rozhrania (Qt, Tickit, i3, Sway, ...)
- Systémové rozhrania (DBus)
- Databázy (DBI)

Spracovanie udalostí sa nastavuje pomocou „monitorov“ (*watchers*).

- Monitor môže popisovať reakcie na udalosti.
- *Neobsahuje* cyklus udalostí!
- Zničením monitora skončí aj spracovanie danej udalosti (RAII).

```perl
my $w_io;  # Separate declaration is necessary if we want to access
           # ‹$w_io› in handler. ‹my $w_io = AnyEvent->…› will not work.

$w_io = AnyEvent->io(fh => \*STDIN, poll => 'r', cb => sub {
    if (!handle_input(STDIN)) {
        # No more data will arrive.
        $w_io = undef;
    }
});
```

Cyklus udalostí skrýva *podmienková premenná*.

- ‹$cv->recv› spustí cyklus udalostí, kým sa nesplní podmienková premenná.
- ‹$cv->send› splní podmienkovú premennú.

Alternatívne ‹$cv->begin› a ‹$cv->end› pre „transakcie“ alebo počítanie zdrojov.

```
my $cv = AnyEvent->condvar;

my $w = AnyEvent->signal(signal => 'USR1', cb => sub {
    $cv->send;
});

$cv->recv;
```

‹AnyEvent::Handle›

Rozšírenie pre ‹AnyEvent› na neblokujúce čítanie vstupov.

```
my $w = AnyEvent::Handle->new(fh => \*STDIN,
    on_read => sub ($self) { … },
    on_eof => sub ($self) { … },
    on_error => sub ($self, $fatal, $message) { … },
);
```

Spracovanie zložitejšieho formátu pomocou *read queue*:

```
# Start with a line.
$w->push_read(line => sub ($, $line, $eol) { … });
# Then, 8 bytes must follow.
$w->push_read(chunk => 8, sub ($, $chunk) { … });
# Next, read everything up to ‹END›.
$w->push_read(regex => qr/END/, sub ($, $data) { … });
```

‹AnyEvent::Socket›

Rozšírenie na prácu s TCP a UNIX soketmi.

```
use AnyEvent::Socket;

my $w = tcp_server($S_HOST, $S_PORT, sub ($socket, $c_host, $c_port) {
    $log->info("Client $c_host:$c_port connected");
    ...
    push @watchers, AnyEvent::Handle->new(fh => $socket, …);
});
```

Podobne klient funkciou ‹tcp_connect›.