PV281: Programování v Rustu

Obsah

Parallelismus

Concurency vs parallelism

Běžně se setkáme s oběma výrazy.

Rozdíl se dobře vysvětluje českým překladem na **současnost** a **souběžnost**.

Proces

Každý proces má vlastní paměťový prostor,
tj. vlastní *stack* a vlastní *heap*.

Přepínaní kontextu je drahé.

Komunikace mezi procesy je pomalejší
(sdílená paměť, message queue, sockety, ...).

Celkově na zdroje má větší náročnost.

Vlákna

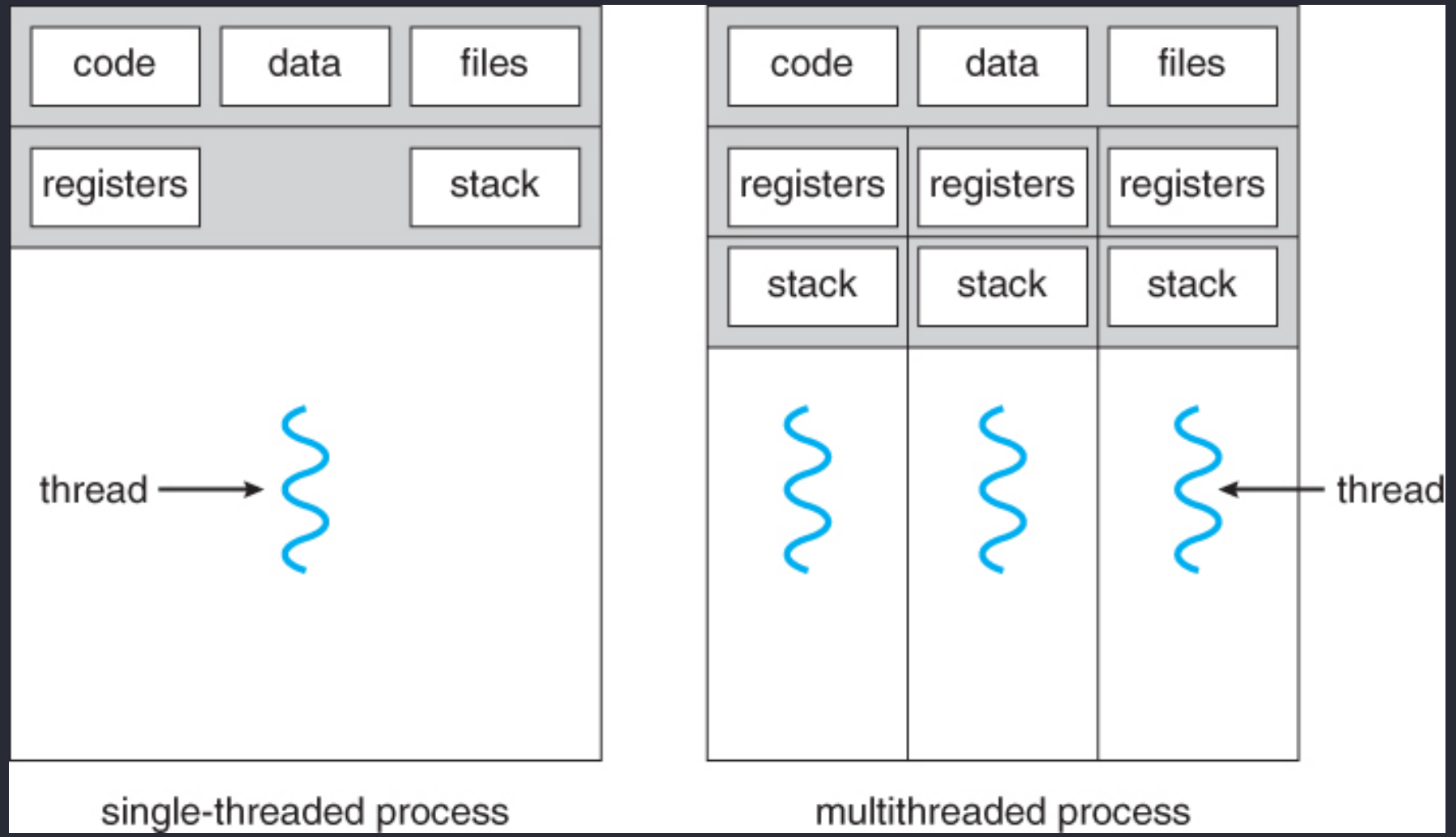
Vlákna sdílí pamět, konkrétně *heap*.

Přepínání kontextu je drahé, ale levnější než u procesů.

Komunikace mezi vlákny je rychlá právě díky sdílené haldě.

Vlákna jsou méně náročná na zdroje systému.

Vlákna


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Plánování procesů ve Windows

Plánování ve Windows

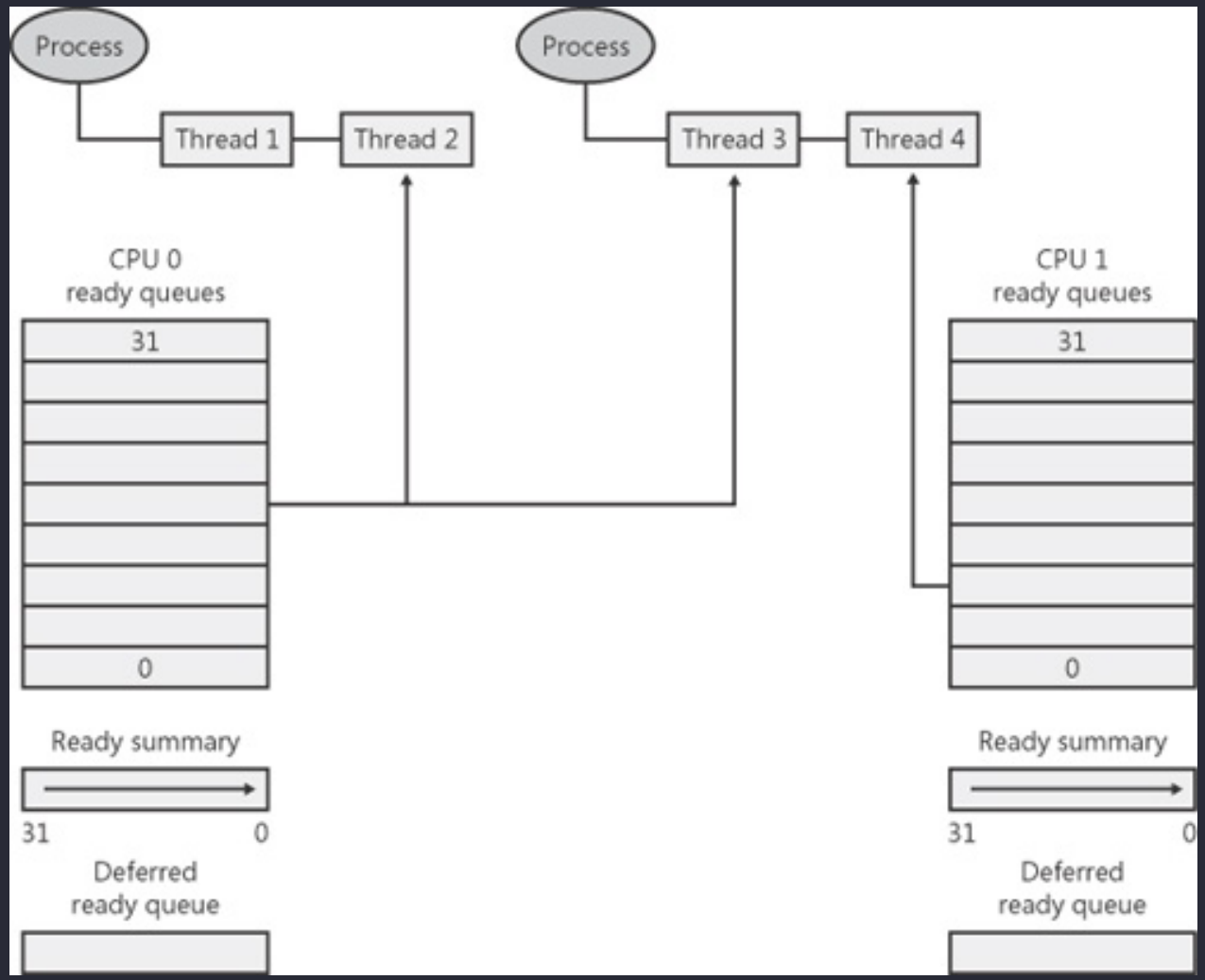
Thread má **prioritu** v rozsahu 0–31 (31 je nejvyšší)

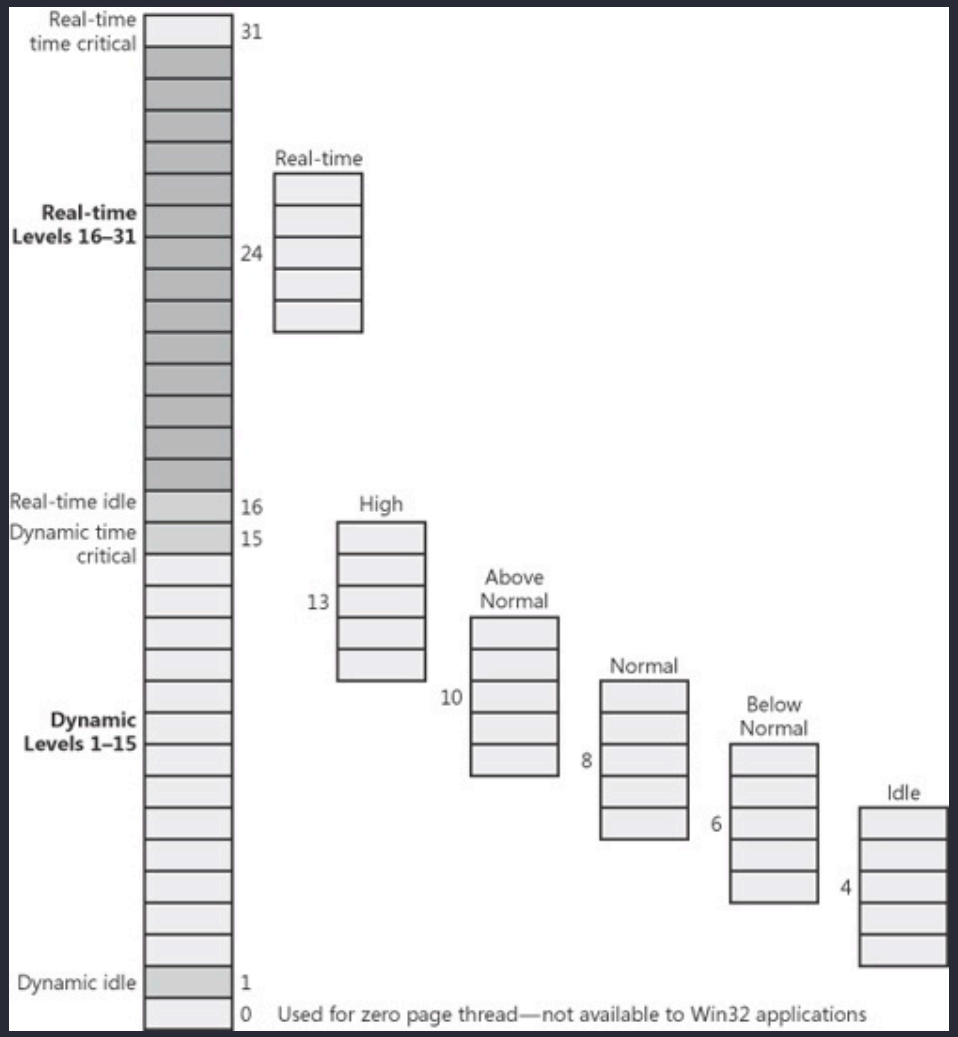
Vlákno má přidělené časové rámce. Časové rámce jsou poskytovány pomocí **round-robin** algoritmu.

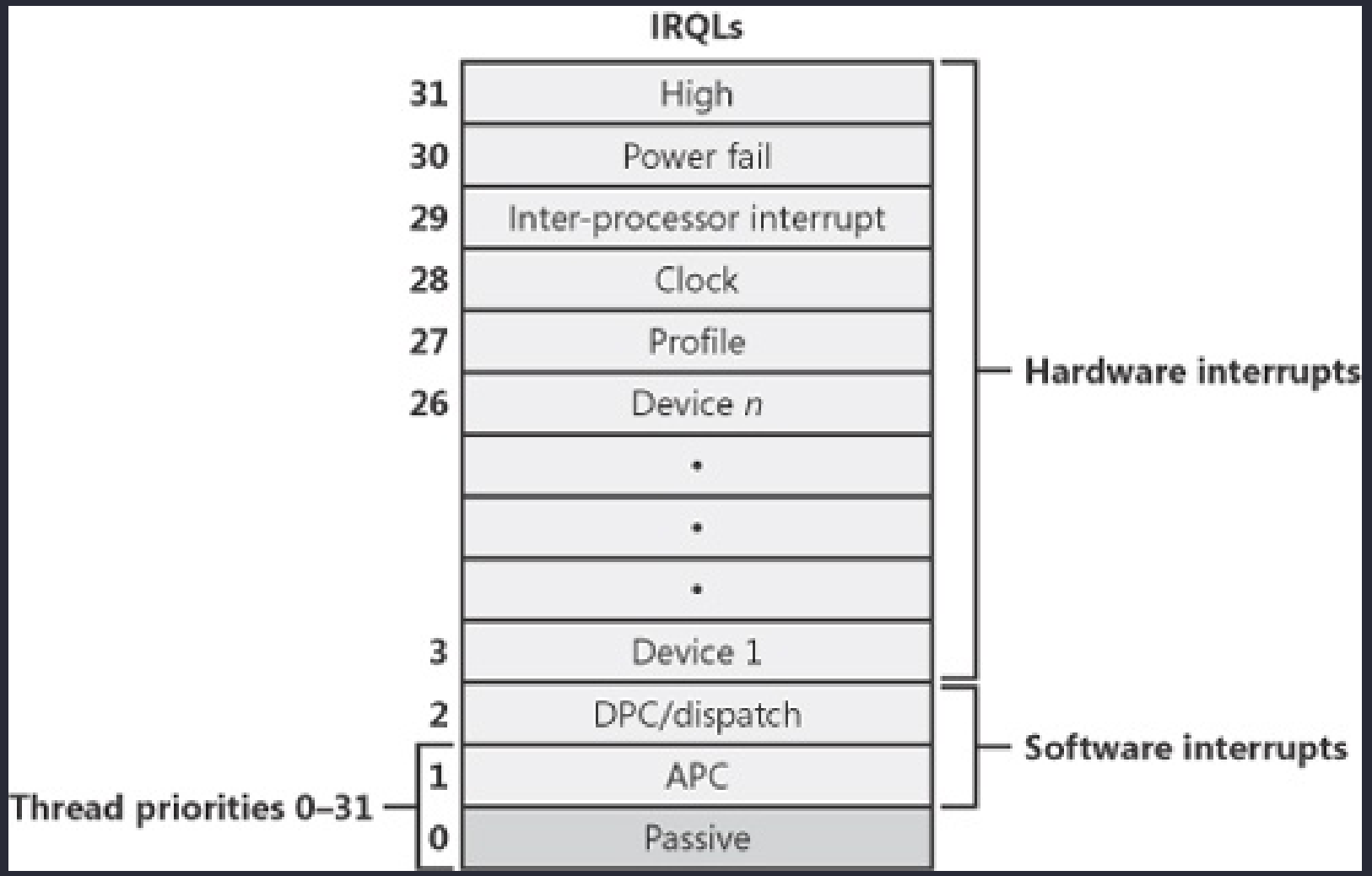
Rámec trvá na klientských Win **2** hodinové cykly, na serverových **12**. Jeden cyklus je na většině x64 systémů asi **15 ms**.

Pokud není žádné vlákno ve vyšší prioritě připraveno běžet, na řadu se dostane priorita nižší.

Pokud běží vlákno s nižší prioritou a najednou je k dispozici s vyšší prioritou, tak nižšímu systém sebere čas.


Process
Process
Thread 1
Thread 2
Thread 3
Thread 4
CPU 0
ready queues
31
0
CPU 1
ready queues
31
0
Ready summary
31
0
Deferred
ready queue
Ready summary
31
0
Deferred
ready queue


Real-time time critical
31
Real-time
Real-time Levels 16–31
24
Real-time idle
16
High
Dynamic time critical
15
13
Above Normal
10
Normal
Dynamic Levels 1–15
8
Below Normal
6
Idle
4
Dynamic idle
1
0
Used for zero page thread—not available to Win32 applications


IRQLs
31
High
30
Power fail
29
Inter-processor interrupt
28
Clock
27
Profile
26
Device n
3
Device 1
2
DPC/dispatch
1
APC
0
Passive
Hardware interrupts
Software interrupts
Thread priorities 0–31

Přepínání vláken

Při přepínání se napřed uloží kontext vlákna, které končí.

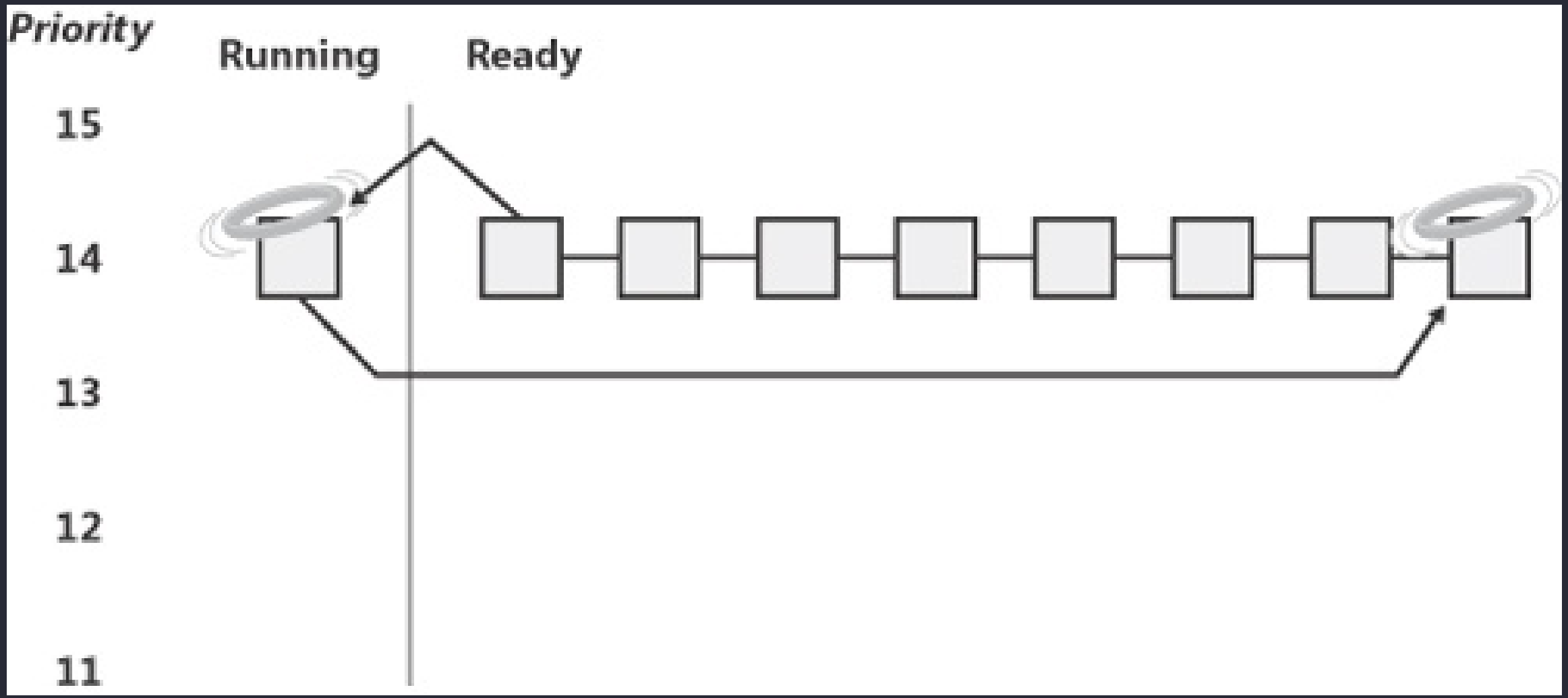
Vlákno se umístí na konec fronty dané priority.

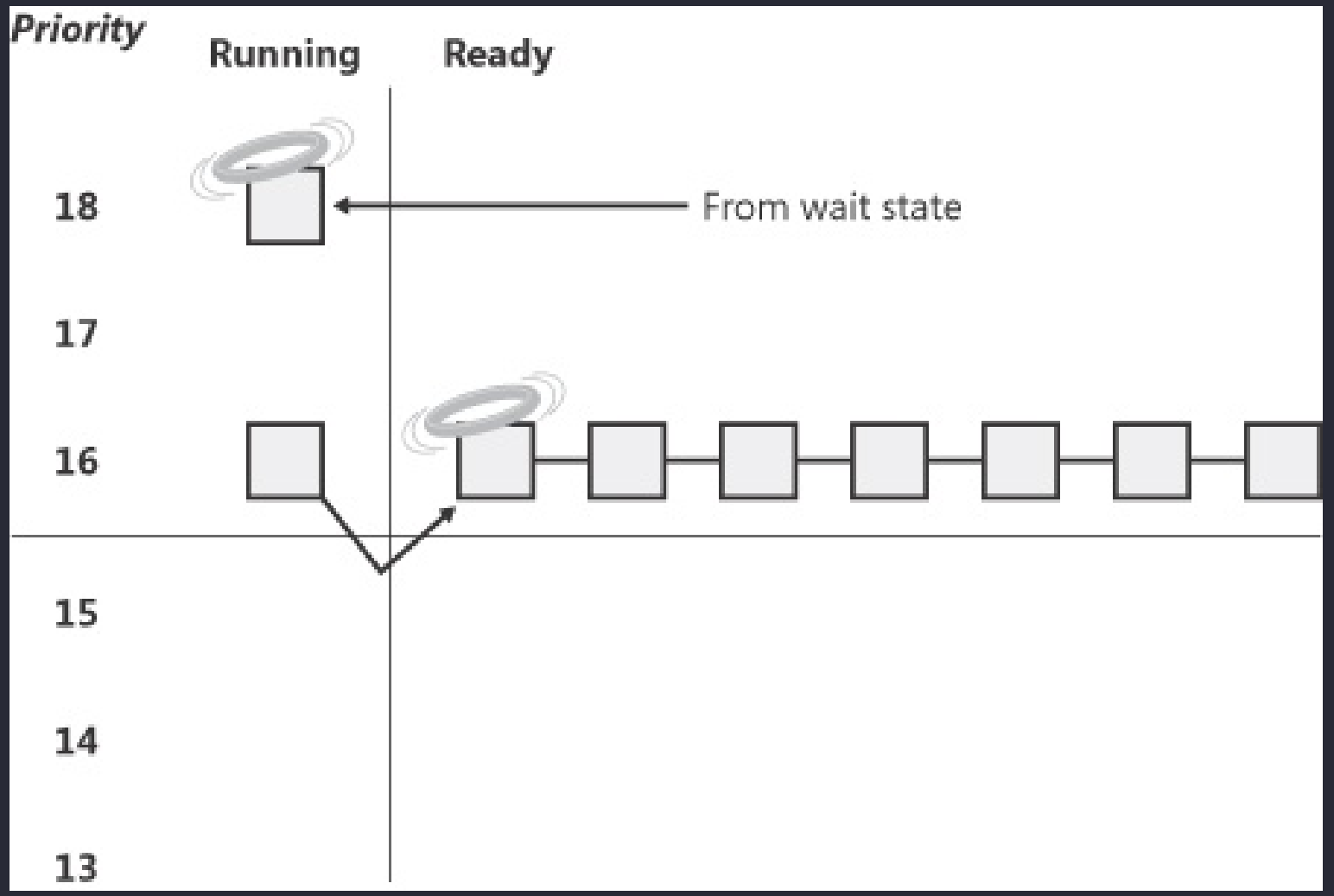
Najde se thread s nejvyšší prioritou, který může běžet.

Ten se vytáhne z fronty, načte se jeho kontext a začne se vykonávat.

Důvody přepnutí

- Je dostupné vlákno s vyšší prioritou.
- Vypršel časový úsek pro běh.
- Vlákno musí na něco čekat a vzdá se svého času.


Priority
Running
Ready
15
14
13
12
11


Priority
Running
Ready
18
17
16
15
14
13
From wait state

Stavy vlákna

`Ready` - plánovač pro běh bere v potaz pouze tato vlákna

`Deferred ready` - vlákna naplánovaná na konkrétním procesoru, ale ještě nezaplánovaná. Existuje kvůli minimalizaci locku na plánovací databázi.

`Standby` - vlákno přípravené na běh na konkrétním procesoru; jakmile to bude možné, tak dojde k přepnutí kontextu. Na jádro je jen jedno standby vlákno. Může být přeskočeno preempcí nebo pokud bude vlákno s vyšší prioritou spustitelné.

`Running Once` - stav při přepnutí kontextu

Stavy vlákna

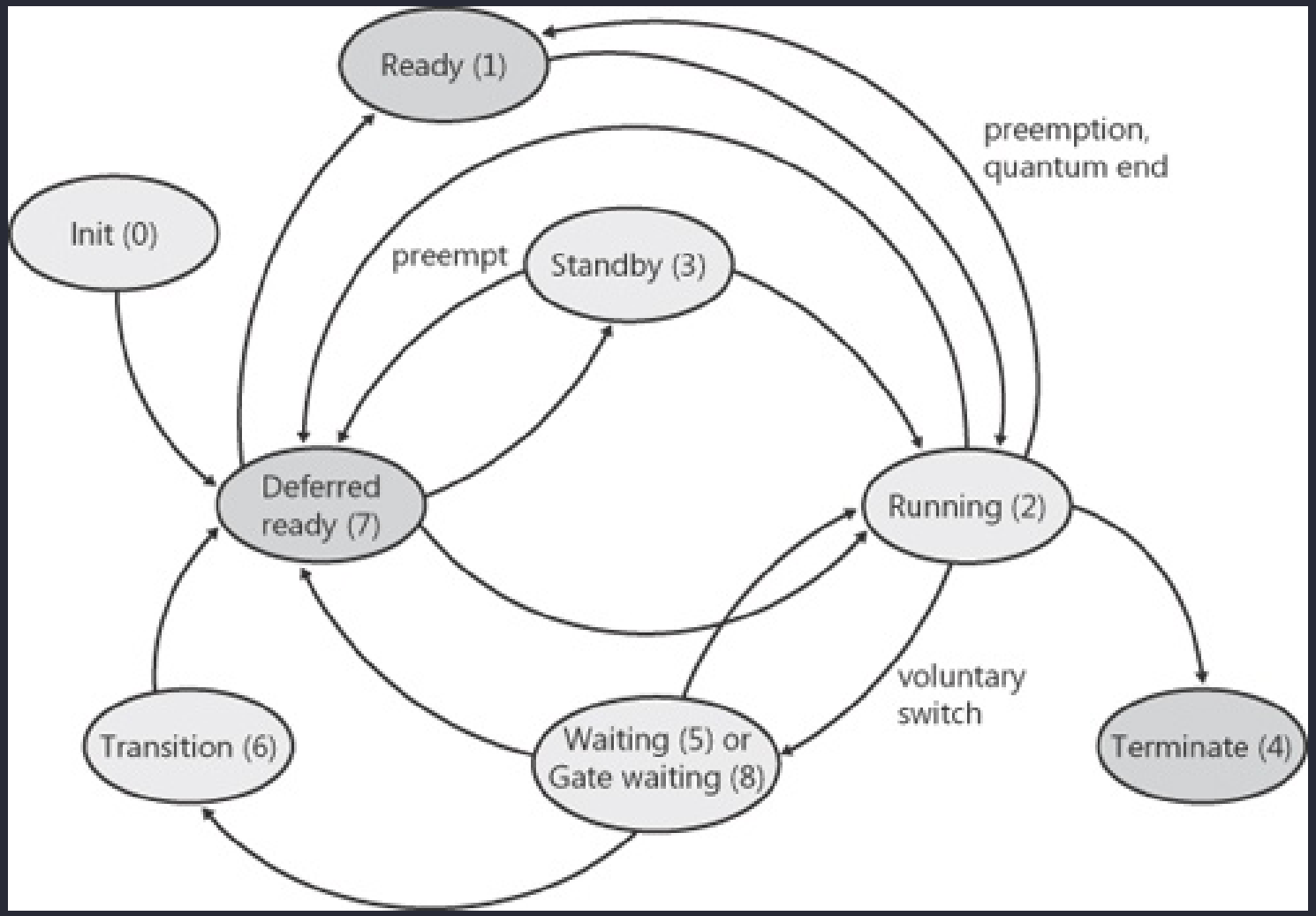
`Waiting` - vlákno na něco čeká – samo pomocí synchronizačního prostředku, na úrovni systému, na I/O operaci,...

`Gate Waiting` - čeká na gate dispatcher objektu

`Transition` - vlákno by mohlo být ready, ale systém odstránkoval zásobník z paměti, tak se čeká, než ho zase dá zpět do paměti

`Terminated` - vlákno skončilo a může být dealokováno

`Initialized` - interní stav po čerstvém vytvoření vlákna


Ready (1)
preemption,
quantum end
Init (0)
preempt
Standby (3)
Deferred
ready (7)
Running (2)
voluntary
switch
Transition (6)
Waiting (5) or
Gate waiting (8)
Terminate (4)

NUMA

Typ víceprocesorového systému s neuniformním přístupem do paměti. Tj. každému procesoru zabere různou dobu se dostat k nějakým částem paměti.

Jádra a pamět jsou seskupena pod uzly. Jsou zájemně propojené sběrnicí s kohenertní cache. Neuniformní jsou proto, že přístup k lokální paměti je mnohem rychlejší.

Nastavení affinity

Nastavení je na úrovní procesu nebo vlákna

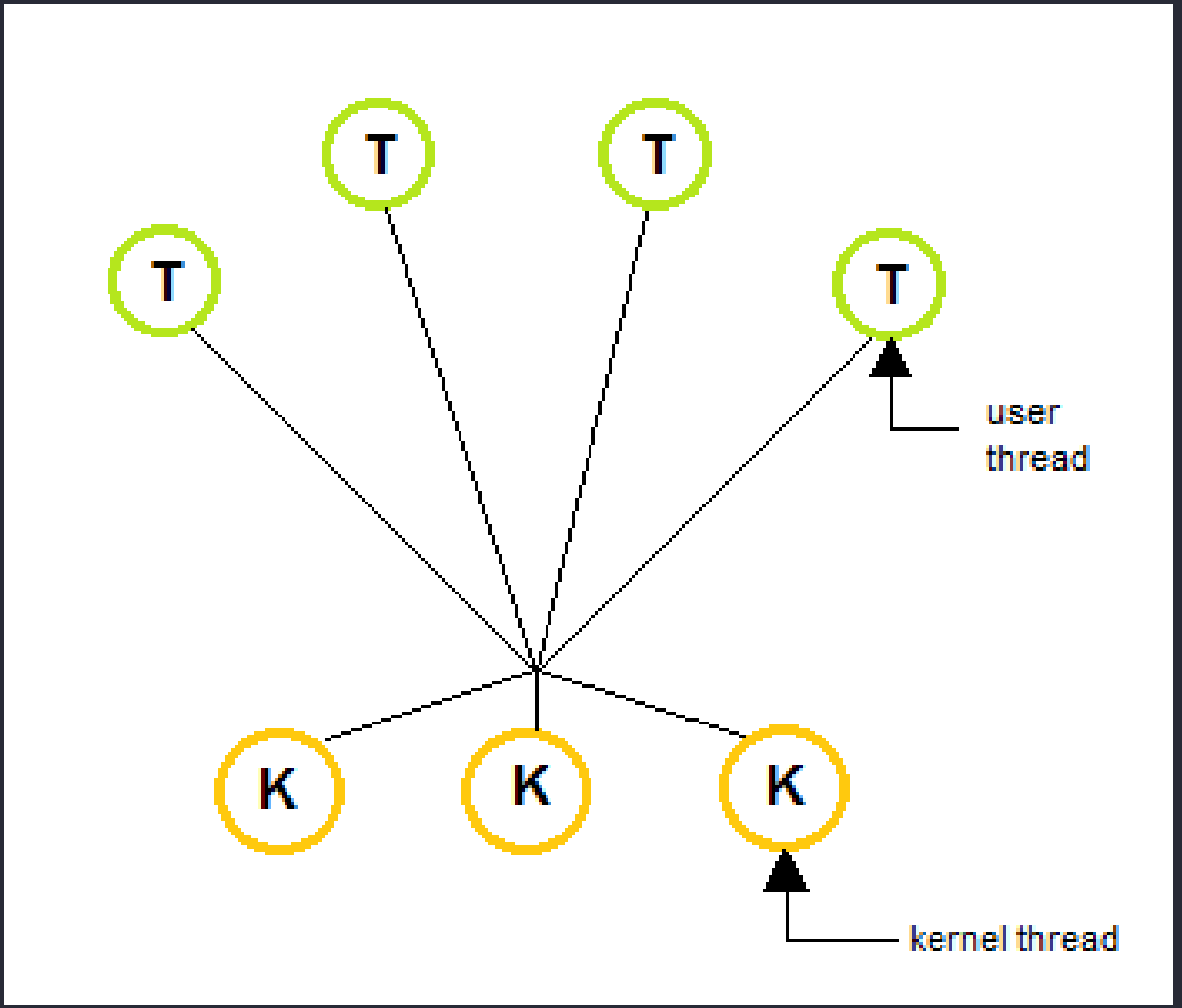
Možnosti:
NUMA
SMP - je to jedno, kde to bude běžet

Green Thread

Abysme zvýšili čas, který program pracuje a snížili prostředky potřebné při přepínání, tak můžeme využít green thready.

V programu využíváme vlastní vlákna a plánování nad systémovým. Na jednom systémovém vlákně můžeme přepínat několik vlastních.

Existují modely one-to-one, one-to-many a many-to-many.


T
T
T
T
user
thread
K
K
K
kernel thread

Práce s thready v Rustu

Práce s thready v Rustu

```rust
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!(
                "hi number {} from the spawned thread!",
                i
            );
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!(
            "hi number {} from the main thread!",
            i
        );
        thread::sleep(Duration::from_millis(1));
    }

    // před ukončením programu bychom měli
    // počkat na dokončení práce všech vláken
}
```

```
$ cargo run

hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
```

Práce s thready v Rustu

```rust
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!(
                "hi number {} from the spawned thread!",
                i
            );
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!(
            "hi number {} from the main thread!",
            i
        );
        thread::sleep(Duration::from_millis(1));
    }

    handle.join().unwrap(); // <- zde je rozdíl
}
```

```
$ cargo run

hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 3 from the main thread!
hi number 4 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

Běžně používané přístupy k paralelismu

Fork-Join

```rust
use std::{thread, io};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
    // Divide the work into several chunks.
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenames, NTHREADS);

    // Fork: Spawn a thread to handle each chunk.
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            thread::spawn(move || process_files(worklist))
        );
    }

    // Join: Wait for all threads to finish.
    for handle in thread_handles {
        handle.join().unwrap()?; // Note that panic from inside the thread propagates upward!
    }

    Ok(())
}
```

Fork-join

- jednoduchý na implementaci
- nevytváří bottleneck
- výkonnostní matematika je jednoduchá
- je jednoduché se bavit o korektnosti programu

Alternativní implementace přes `rayon`

```
use rayon::prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &GigabyteMap) -> io::Result<()> {
    filenames.par_iter()
        .map(|filename| process_file(filename, glossary))
        .reduce_with(|r1, r2| {
            if r1.is_err() { r1 } else { r2 }
        })
        .unwrap_or(Ok(()))
}
```

Přenos dat pomocí kanálů – odesílání

Kanál `mpsc` – několik producentů a jeden konzument.

```
use std::{fs, thread};
use std::sync::mpsc;

// ...

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
    for filename in documents {
        let text = fs::read_to_string(filename)?;

        if sender.send(text).is_err() {
            break;
        }
    }
    Ok(())
});

// ...
```

Přenos dat pomocí kanálů – příjem

```
while let Ok(text) = receiver.recv() {
    do_something_with(text);
}
```

Pipeline

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf) -> io::Result<()> {
    // Launch all five stages of the pipeline.
    let (texts,   h1) = start_file_reader_thread(documents);
    let (pints,   h2) = start_file_indexing_thread(texts);
    let (gallons, h3) = start_in_memory_merge_thread(pints);
    let (files,   h4) = start_index_writer_thread(gallons, &output_dir);
    let result        = merge_index_files(files, &output_dir);

    // Wait for threads to finish, holding on to any errors that they encounter.
    let r1 = h1.join().unwrap();
    h2.join().unwrap();
    h3.join().unwrap();
    let r4 = h4.join().unwrap();

    // Return the first error encountered, if any. Here, h2 and h3 can't fail as those threads are pure in-memory data processing.
    r1?;
    r4?;
    result
}
```

Implementace 1. bloku pipe

```
fn start_file_reader_thread(documents: Vec<PathBuf>)
    -> (mpsc::Receiver<String>, thread::JoinHandle<io::Result<()>>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        // ...
    });

    (receiver, handle)
}
```

Implementace 2. bloku pipe

```
fn start_file_indexing_thread(texts: mpsc::Receiver<String>)
    -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
{
    let (sender, receiver) = mpsc::channel();

    let handle = thread::spawn(move || {
        for (doc_id, text) in texts.into_iter().enumerate() { // Všimněte si, že `mpsc::Receiver` je iterovatelný.
            let index = InMemoryIndex::from_single_document(doc_id, text);

            if sender.send(index).is_err() {
                break;
            }
        }
    });

    (receiver, handle)
}
```

Piping iterátoru na channel

```
documents.into_iter()
    .map(read_whole_file)
    .errors_to(error_sender)      // filter out error results
    .off_thread()                 // spawn a thread for the above work
    .map(make_single_file_index)
    .off_thread()                 // spawn another thread for stage 2
    // ...
```

Implementace off_thread

```
use std::sync::mpsc;

pub trait OffThreadExt: Iterator {
    /// Transform this iterator into an off-thread iterator: the
    /// `next()` calls happen on a separate worker thread, so the
    /// iterator and the body of your loop run concurrently.
    fn off_thread(self) -> mpsc::IntoIter<Self::Item>;
}
```

Implementace off_thread

```rust
use std::thread;

impl<T> OffThreadExt for T
    where T: Iterator + Send + 'static,
          T::Item: Send + 'static
{
    fn off_thread(self) -> mpsc::IntoIter<Self::Item> {
        // Create a channel to transfer items from the worker thread.
        let (sender, receiver) = mpsc::sync_channel(1024);

        // Move this iterator to a new worker thread and run it there.
        thread::spawn(move || {
            for item in self {
                if sender.send(item).is_err() {
                    break;
                }
            }
        });

        // Return an iterator that pulls values from the channel.
        receiver.into_iter()
    }
}
```

Poznámky k pipeline

Pipeline nemá linární zvýšení výkonu.

U pipeline může lehce vzniknout *bottleneck*.

Optimalizací může být synchronní kanál `let (sender, receiver) = mpsc::sync_channel(1000);`.

Synchronizační primitiva

Arc<T>

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();

            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    assert_eq!(*counter.lock().unwrap(), 10);
}
```

Mutex

```
use std::sync::Mutex;

fn main() {
    let m = Mutex::new(5);

    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }

    assert_eq!(m.lock().unwrap(), 6);
}
```

Více konzumentů s využitím mutextu

```rust
pub mod shared_channel {
    use std::sync::{Arc, Mutex};
    use std::sync::mpsc::{channel, Sender, Receiver};

    /// A thread-safe wrapper around a `Receiver`.
    #[derive(Clone)]
    pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);

    impl<T> Iterator for SharedReceiver<T> {
        type Item = T;

        /// Get the next item from the wrapped receiver.
        fn next(&mut self) -> Option<T> {
            let guard = self.0.lock().unwrap();
            guard.recv().ok()
        }
    }

    /// Create a new channel whose receiver can be shared across threads.
    /// This returns a sender and a receiver, just like the stdlib's `channel()`, and sometimes works as a drop-in replacement.
    pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
        let (sender, receiver) = channel();
        (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
    }
}
```

RwLock<T>

Umožňuje **n** čtenářů a **jednoho** zapisujícího. Mutex toto neřeší.

```rust
use std::sync::RwLock;

fn main() {
    let lock = RwLock::new(5);

    { // Many reader locks can be held at once.
        let r1 = lock.read().unwrap();
        let r2 = lock.read().unwrap();
        assert_eq!(*r1, 5);
        assert_eq!(*r2, 5);
    } // Read locks are dropped at this point.

    { // Only one write lock may be held, however.
        let mut w = lock.write().unwrap();
        *w += 1;
        assert_eq!(*w, 6);

        // Uncommenting this would wait forever as `w` would never unlock:
        // let r = lock.read().unwrap();
    }

    let r = lock.read().unwrap();
    assert_eq!(*r, 6);
}
```

Bariéra

Synchronizuje vlákna tak, aby všechna začala zároveň.

```rust
use std::sync::{Arc, Barrier};
use std::thread;

fn main() {
    let mut handles = Vec::with_capacity(10);
    let barrier = Arc::new(Barrier::new(10));

    for _ in 0..10 {
        let c = Arc::clone(&barrier);

        handles.push(thread::spawn(move || {
            println!("before wait");
            c.wait();                    // <- zde je bariéra
            println!("after wait");
        }));
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

```
$ cargo run

before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
before wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
after wait
```

Alternativní implementace primitiv

Synchronizační prostředky v `std` nemusí být nejrychlejší.
Běžně používaná crate poskytující rychlejší implementaci je např.
`parking_lot`.

Další synchronizační prostředky

Pokud budete hledat pokročilejší synchronizační prostředky,
tak je najdete např. v crate `crossbeam`.

Scoped thread

```rust
let greeting = String::from("Hello world!");

thread::scope(|s| {
    s.spawn(|_| {
        println!("thread #1 says: {}", greeting); // Sdílíme proměnné jako `greeting`
    });

    s.spawn(|_| {
        println!("thread #2 says: {}", greeting);
        // Pozor, pokud bychom tu chtěli něco mutovat!
    });

    // Díky scope nemusíme dělat ručně `join` vláken.
});
```

Úvod do asynchronního programování

Asynchronní programování

Koncept pro psaní konkurentních programů.
Umožňuje využívat neblokující operace na jednom vlákně
například pro I/O.

V Rustu platí:
Je jedno- i vícevláknový.
Async víceméně nic nestojí.
Nemá výchozí runtime.

Asynchronní přístup vs vlákna

Můžeme si říct, že přece podobného efektu (*neblokovaná aplikace*) dosáhneme pomocí vláken.

Vlákna jsou řízená OS, jejich přepínání je relativně drahé.
Vlákna mohou zabírat i stovky KB paměti
(což se v případě vlákna pro každého klienta prodraží).

Ukázka synchronního kódu

```rust
use std::net;

fn cheapo_request(host: &str, port: u16, path: &str) -> std::io::Result<String> {
    let mut socket = net::TcpStream::connect((host, port))?;

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
    socket.write_all(request.as_bytes())?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response)?;

    Ok(response)
}
```

Převod na asynchronní

```rust
use async_std::io::prelude::*;
use async_std::net;

// Funkce musí být označená jako `async`.
async fn cheapo_request(host: &str, port: u16, path: &str) -> std::io::Result<String> {
    let mut socket = net::TcpStream::connect((host, port)).await?; // <- `await` zajišťuje neblokující volání

    let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
    socket.write_all(request.as_bytes()).await?;
    socket.shutdown(net::Shutdown::Write)?;

    let mut response = String::new();
    socket.read_to_string(&mut response).await?;

    Ok(response)
}
```

Future

```
trait Future {
    type Output;
    fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
}
enum Poll<T> {
    Ready(T),
    Pending,
}
```

Trait je obsažen v `std`, funkcionalitu ale poskytují crates, např:

```
[dependencies]
async_std = "1.12"
```

Princip poolingu

Task se začíná vykonávat prvním poolingem (volání `await`).

Pokud se vrací `Poll::Pending`, pokračuje se dalším taskem.

Pokud všechny tasky vrací `Poll::Pending`, *executor* se uspí.

Pokud je některé operace doběhla, *waker* probere *executor*a.

Executor ví, že operace doběhla, a předá data tam, kde jsou potřeba.

Spojení se synchronním kódem – `block_on`

```
use async_std::task::block_on;

fn main() -> std::io::Result<()> {
    let response = block_on(cheapo_request("example.com", 80, "/"))?;

    println!("{}", response);

    Ok(())
}
```

Vytvoření asynchronních tasků na jednom vlákně

```rust
use async_std::task::spawn_local;

pub async fn many_requests(requests: Vec<(String, u16, String)>) -> Vec<std::io::Result<String>> {
    let mut handles = vec![];
    for (host, port, path) in requests {
        // `spawn_local` analogické k vytvoření vlákna
        handles.push(spawn_local(cheapo_request(&host, port, &path)));
    }

    let mut results = vec![];
    for handle in handles {
        results.push(handle.await);
    }

    results
}
```

Asynchronní blok

```
fn main() {
    let serve_one = async {
        use async_std::net;

        // Listen for connections, and accept one.
        let listener = net::TcpListener::bind("localhost:8087").await?;
        let (mut socket, _addr) = listener.accept().await?;

        // Talk to client on `socket`.
        // ...
    };
}
```

Funkce z asynchronního bloku

Výstupním typem musí být `impl Future<Output = T>`:

```
use std::io;
use std::future::Future;

fn cheapo_request<'a>(host: &'a str, port: u16, path: &'a str) -> impl Future<Output = io::Result<String>> + 'a
{
    async move {
        // ... function body
    }
}
```

Vytvoření tasku na threadpoolu

```
use async_std::task;

let mut handles = vec![];
for (host, port, path) in requests {
    handles.push(task::spawn(async move {
        cheapo_request(&host, port, &path).await
    }));
}
```

Async IO

```
use async_std::fs::File;
use async_std::prelude::*;

let mut f = File::open("foo.txt").await?;
let mut buffer = [0; 10];

// read up to 10 bytes
let n = f.read(&mut buffer).await?;

println!("The bytes: {:?}", &buffer[..n]);
```

Asynch BufReader

```rust
use async_std::fs::File;
use async_std::io::BufReader;
use async_std::prelude::*;

let f = File::open("foo.txt").await?;
let mut reader = BufReader::new(f);
let mut buffer = String::new();

// read a line into buffer
reader.read_line(&mut buffer).await?;

println!("{}", buffer);
```

Stdin a stdout

```
use async_std::io;

let mut input = String::new();

io::stdin().read_line(&mut input).await?;

println!("You typed: {}", input.trim());

io::stdout().write(&[42]).await?;
```

Async main

```
#[async_std::main]
async fn main() -> std::io::Result<()> {
    Ok(())
}
```

Async v traitu

Aktuálně bohužel není možné použít `async` v traitu.
Je třeba použít makro z crate `async-trait`.

```rust
use async_trait::async_trait;

#[async_trait]
trait Advertisement {
    async fn run(&self);
}

struct Modal;

#[async_trait]
impl Advertisement for Modal {
    async fn run(&self) {
        self.render_fullscreen().await;
        for _ in 0..4u16 {
            remind_user_to_join_mailing_list().await;
        }
        self.hide_for_now().await;
    }
}
```

Perftesting

Bench atribut

Aktuálně jde o **unstable** feature, takže je potřeba **nightly compiler**.

```
#![feature(test)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

Spuštění testu

```
cargo bench
```

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/user/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

Optimalizace a benchmarking

Optimalizer vypustí část kódu, kterou pokládá za zbytečnou.

```
#![feature(test)]

extern crate test;
use test::Bencher;

#[bench]
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        (0..1000).fold(0, |old, new| old ^ new); // <- Řešením je vrátit hodnotu, tj. odstranit `;`.
    });
}
```

Blackbox

```
#![feature(test)]

extern crate test;

b.iter(|| {
    let n = test::black_box(1000);

    (0..n).fold(0, |a, b| a ^ b)
})
```

Crate `criterion`

```
[dev-dependencies]
criterion = "0.3"

[[bench]]
name = "my_benchmark"
harness = false
```

Crate `criterion`

Soubor `/benches/bench_name.rs`:

```
use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 1,
        1 => 1,
        n => fibonacci(n-1) + fibonacci(n-2),
    }
}

fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("fib 20", |b| b.iter(|| fibonacci(black_box(20))));
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

Dotazy?

Děkuji za pozornost