PV281: Programování v Rustu

Obsah

Lifetime

Je konstrukce překladače, která říká zda je borrow validní. Dříve byl nutností, dneska už není třeba moc často používat. Kód by měl jít napsat většinou i bez specifikace lifetimu.

Lifetime

```
fn main() {
    let i = 3; // Lifetime for `i` starts. ──────────────────┐
    //                                                       │
    { //                                                     │
        let borrow1 = &i; // `borrow1` lifetime starts. ──┐  │
        //                                                │  │
        println!("borrow1: {}", borrow1); //              │  │
    } // `borrow1 ends. ──────────────────────────────────┘  │
    //                                                       │
    //                                                       │
    { //                                                     │
        let borrow2 = &i; // `borrow2` lifetime starts. ──┐  │
        //                                                │  │
        println!("borrow2: {}", borrow2); //              │  │
    } // `borrow2` ends. ──────────────────────────────────┘  │
    //                                                       │
}   // Lifetime ends. ──────────────────────────────────────┘
```

Explicitní anotace lifetimu

```
// `print_refs` bere dvě reference na  `i32`, které mají
// lifetime `'a` a `'b`. Oba musí žít minimálně stejné
// dlouho jako funkce `print_refs`.
fn print_refs<'a, 'b>(x: &'a i32, y: &'b mut i32) {
    println!("x is {} and y is {}", x, y);
}
```

Lifetime s generikou

```rust
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetime s generikou

Pokud v předchozím příkladu nepoužijeme lifetime, tak příklad nejde přeložit. Překladač netuší jestli bude návratová hodnota má lifetime x nebo y.

Coercion

```rust
// Here, Rust infers a lifetime that is as short as possible.
// The two references are then coerced to that lifetime.
fn multiply<'a>(first: &'a i32, second: &'a i32) -> i32 {
    first * second
}

// `<'a: 'b, 'b>` reads as lifetime `'a` is at least as long as `'b`.
// Here, we take in an `&'a i32` and return a `&'b i32` as a result of coercion.
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}

fn main() {
    let first = 2; // Longer lifetime

    {
        let second = 3; // Shorter lifetime

        println!("The product is {}", multiply(&first, &second));
        println!("{} is the first", choose_first(&first, &second));
    };
}
```

Další příklad generiky

```
// Here a reference to `T` is taken where `T` implements
// `Debug` and all *references* in `T` outlive `'a`. In
// addition, `'a` must outlive the function.
fn print_ref<'a, T>(t: &'a T) where
    T: Debug + 'a {
    println!("`print_ref`: t is {:?}", t);
}
```

Lifetime u struktur

```rust
// A type `Borrowed` which houses a reference to an
// `i32`. The reference to `i32` must outlive `Borrowed`.
// Pokud máme ve struktuře referenci, tak musíme definova vždy.
// Vždycky čepičku pane Bohumile.
#[derive(Debug)]
struct Borrowed<'a>(&'a i32);

// An enum which is either an `i32` or a reference to one.
#[derive(Debug)]
enum Either<'a> {
    Num(i32),
    Ref(&'a i32),
}

fn main() {
    let x = 18;
    let y = 15;

    let single = Borrowed(&x);
    let reference = Either::Ref(&x);
    let number    = Either::Num(y);

    println!("x is borrowed in {:?}", single);
    println!("x is borrowed in {:?}", reference);
    println!("y is *not* borrowed in {:?}", number);
}
```

Lifetime traitu

```rust
// A struct with annotation of lifetimes.
#[derive(Debug)]
struct Borrowed<'a> {
    x: &'a i32,
}

// Annotate lifetimes to impl.
impl<'a> Default for Borrowed<'a> {
    fn default() -> Self {
        Self {
            x: &10,
        }
    }
}

fn main() {
    let b: Borrowed = Default::default();
    println!("b is {:?}", b);
}
```

Elision

Pro běžné příklady určuje lifecycle sám překladač. Dělá to podle následujících pravidel:

1. pravidlo pro životnost vstupních parametrů
   Každý vstupní parametr dostává vlastní lifetime.
2. pravidlo pro životnost výstupních parametrů
   Pokud má funkce jeden vstupní parametr, tak všechny výstupy mají stejný lifetime.
3. pravidlo pro metody s parametrem self
   Pokud má metoda vstupní parametr referenci na self, všechny výstupní parametry mají stejný lifetime.

'static

Snažte se mu vyhnout. Dává životnost po celý běh programu. Hodí se například pro chyvé hlášky.

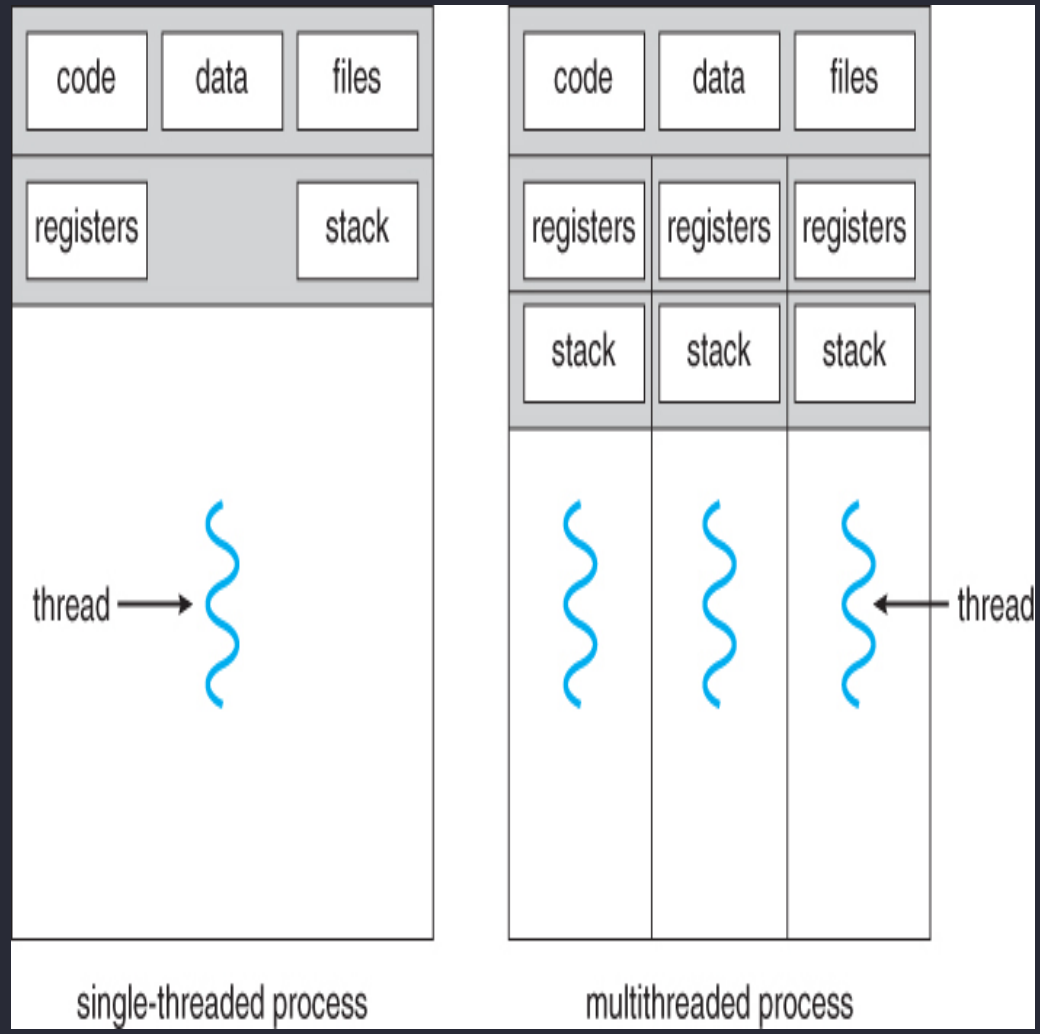
Paralelismus

Proces

Každý proces má vlastní paměťový prostor. Tj. vlastní stack a vlastní heap.
Přepínaní kontextu je drahé.
Komunikace mezi procesy je pomalejší (sdílená paměť, msg queue, socktety...).
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 - díky sdílene haldě.
Vlákna jsou mámě náročná na zdroje systému.

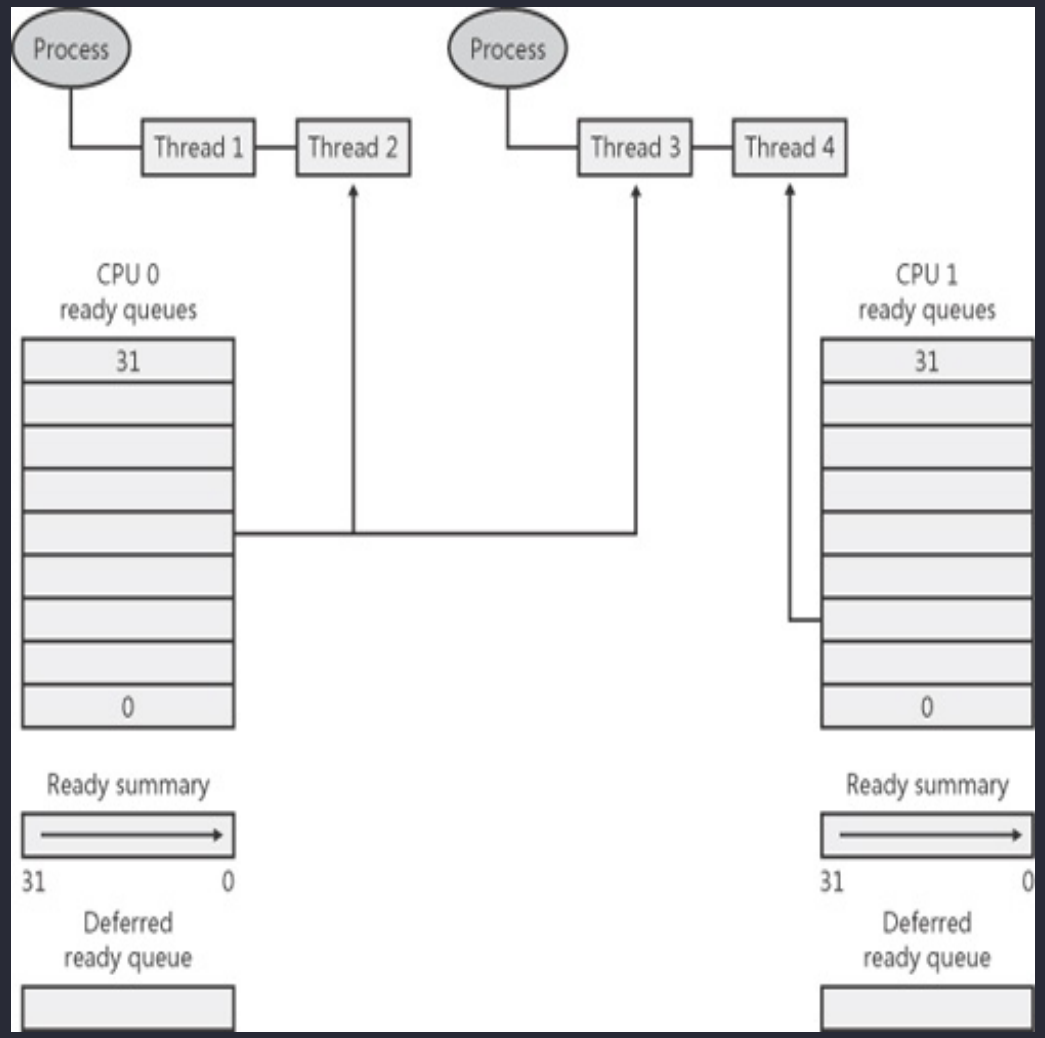


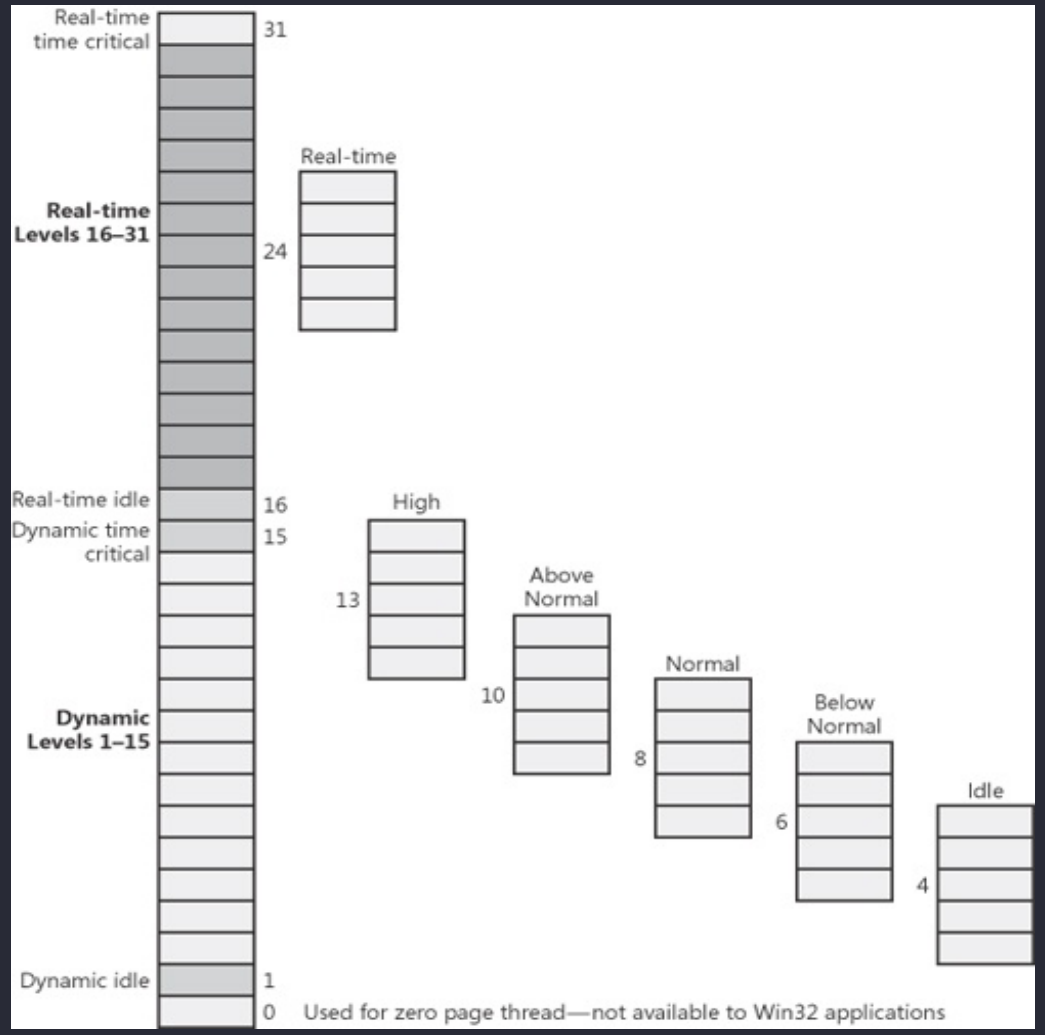
https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threa

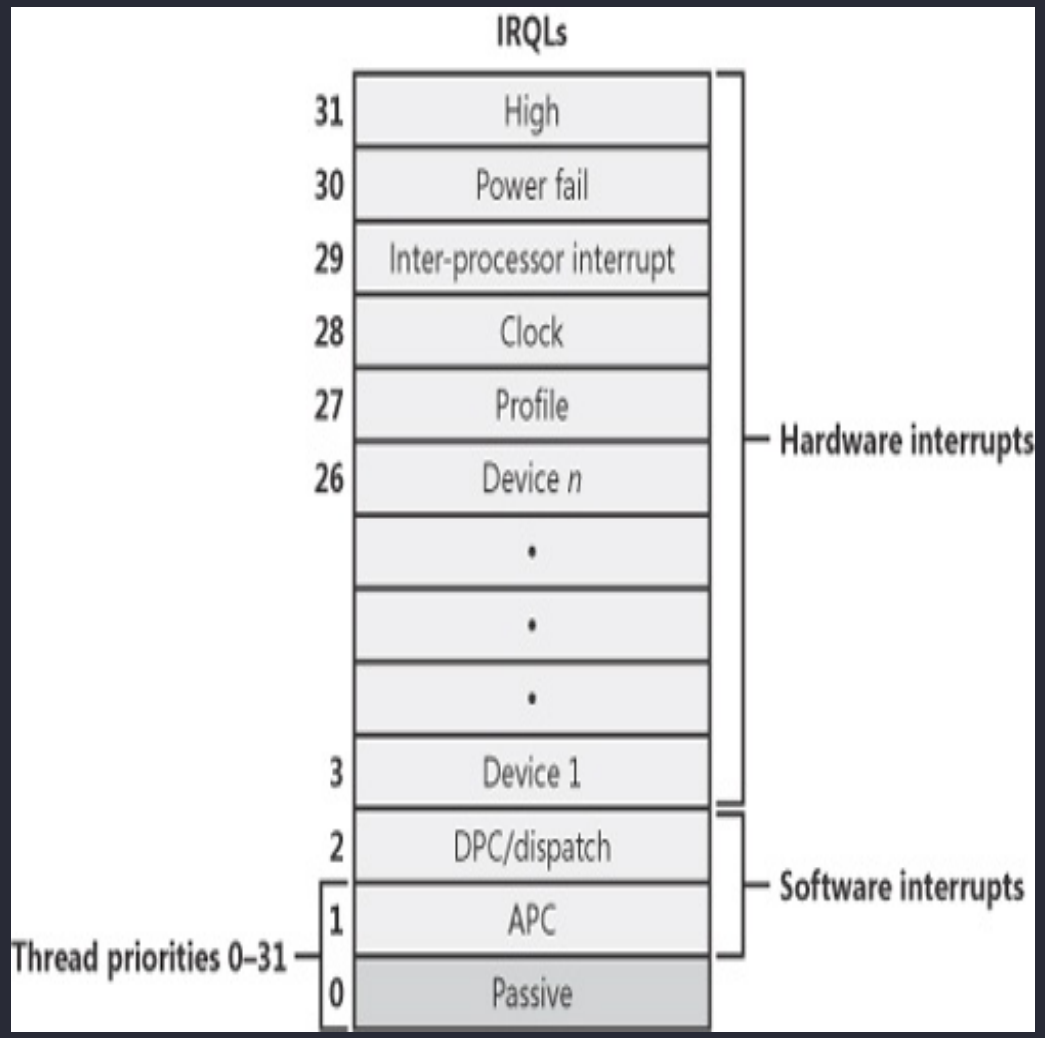
Plánování procesů ve Windows

Plánování ve Windows

1. Thread má prioritu 0-31 (nejvyšší)
2. Vlákno má přidělené časové rámce. Časové rámce jsou dávány v rámci round-robin
3. Rámec tvá na klientkých Win 2 hodinové cykly. Na serverových 12. Jeden cyklus je na většině x64 systému asi 15ms.
4. Pokud není ve vyšší prioritě kdo by běžel, prioritu dostane nižší level
5. Pokud běží nižší proces a najednou je k dispozici s vyšší prioritou, tak mu 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

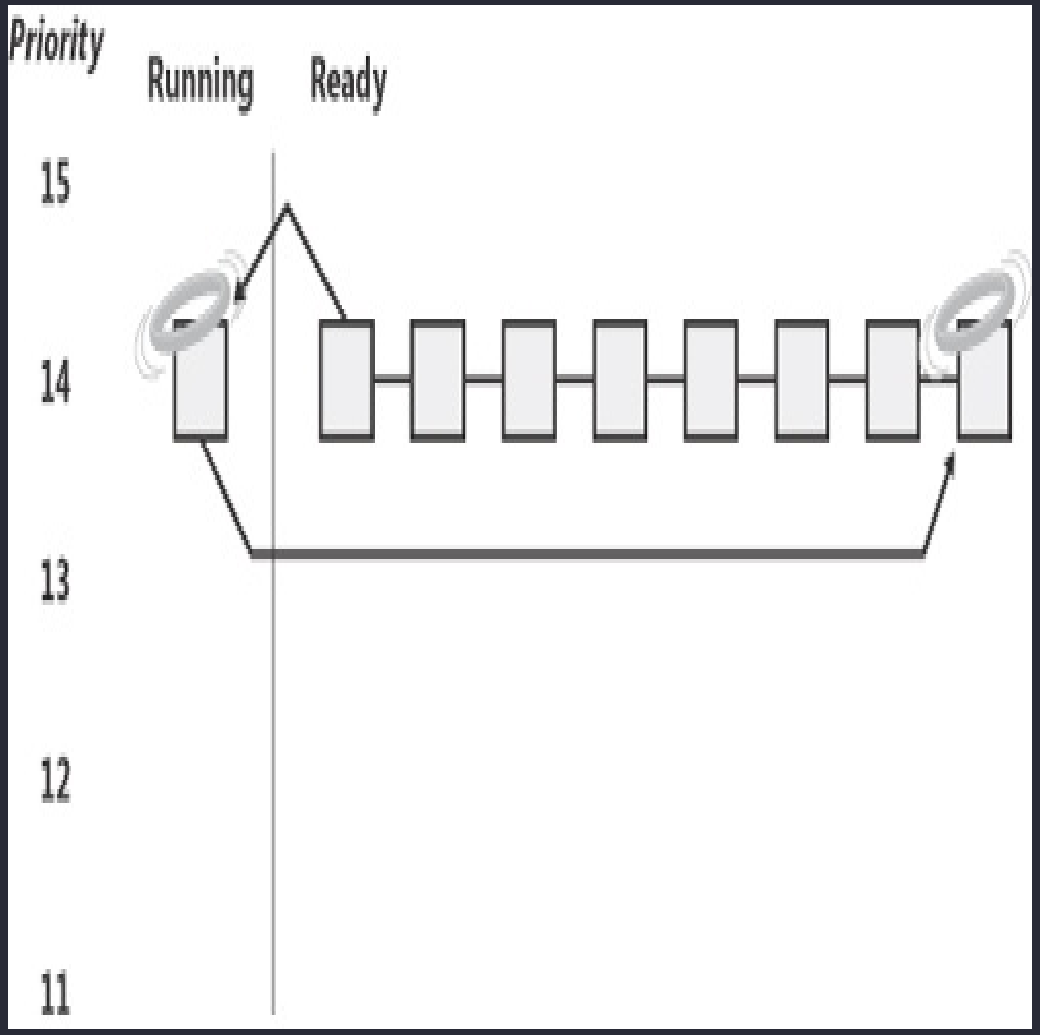

IRQLs
31
High
30
Power fail
29
Inter-processor interrupt
28
Clock
27
Profile
26
Device n
3
Device 1
2
DPC/dispatch
1
APC
0
Passive
Thread priorities 0–31
Hardware interrupts
Software interrupts

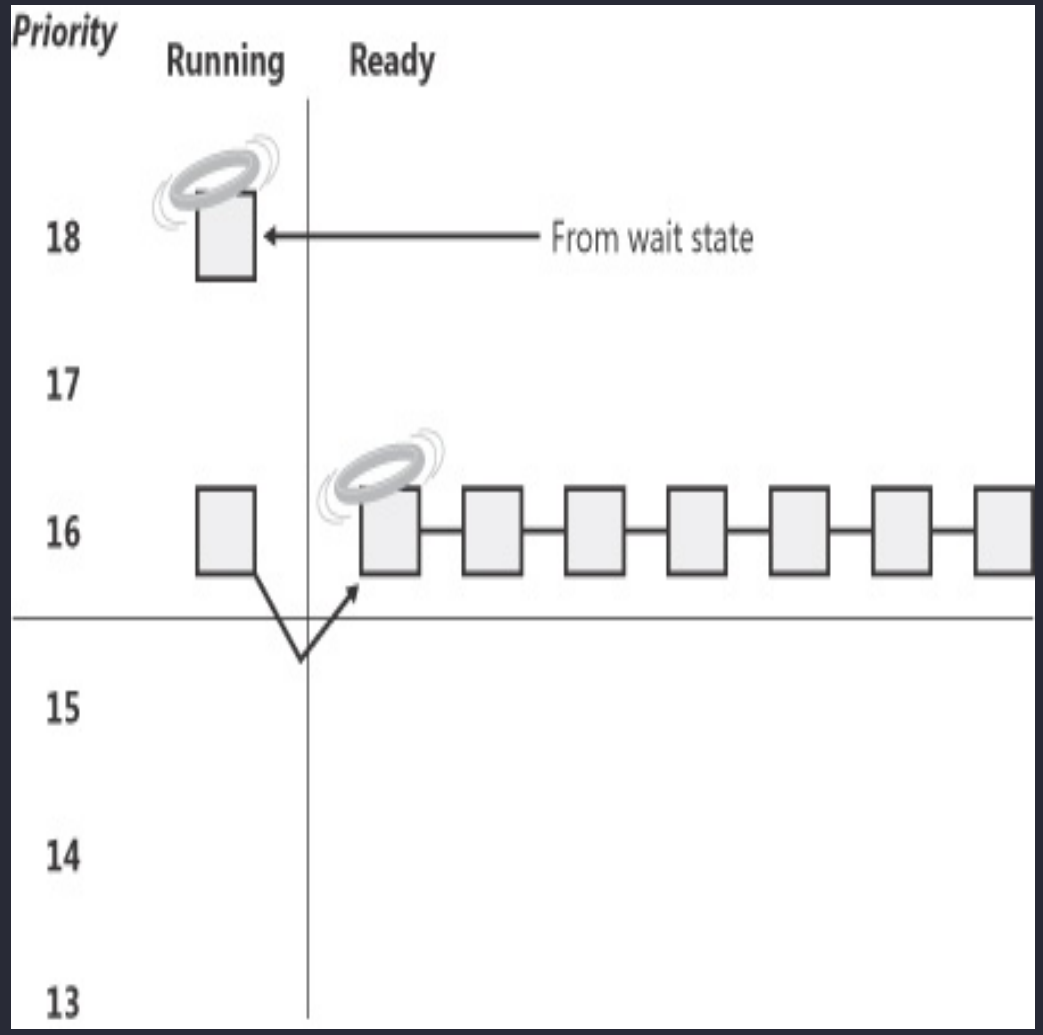
Přepínání vláken

Při přepínání se napřed uloží kontext threadu, který končí.
Umístí se nakonec 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

dostupný thread s vyšší prioritou
vypršel časový úsek pro běh
thread musí na něco čekat


Priority
Running
Ready
15
14
13
12
11


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

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

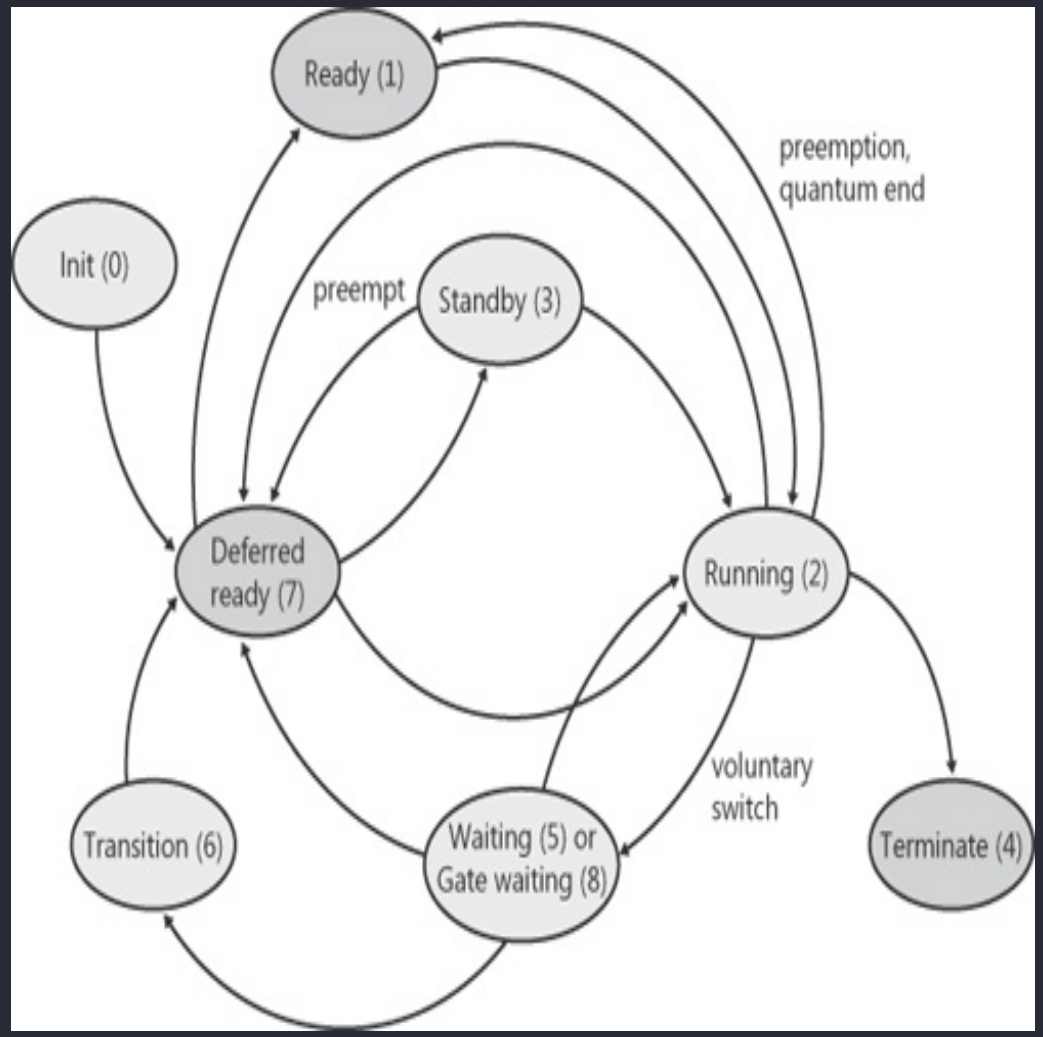
Standby - vlánko přípravené na běh na konkrétním porsoru. Jakmile to bude možné, tak dojde k přepnutí kontextu. Na jádro je jen jedno standby vlákno. Může být přeskočeno prempcí nebo pokud bude vlákno s vyšší prioritou spustitelné.

Running Once - stav při přepnutí kontextu

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

Gate Waiting - čeká na gate dispatcher objektu.

Transition - Vlánko by mohlo být ready, ale systém odstránkoval


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

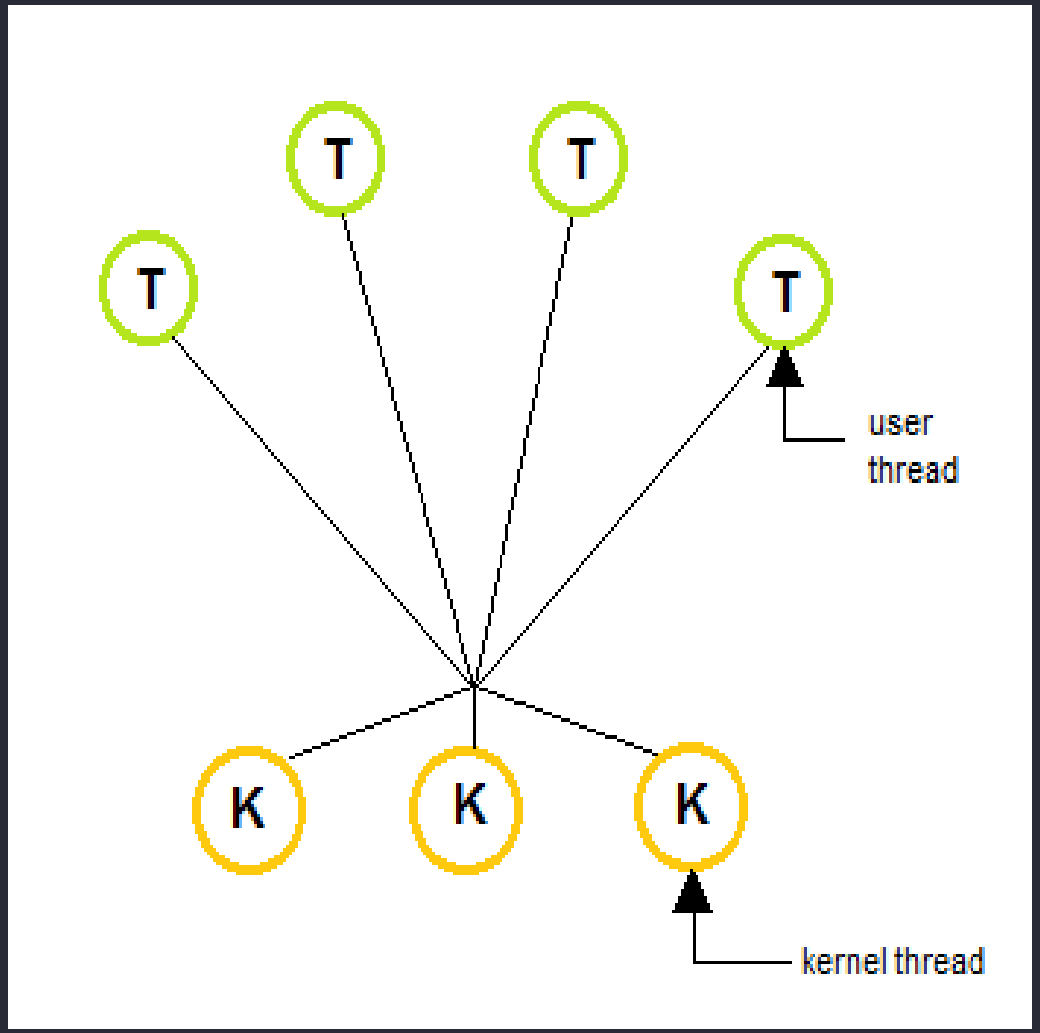
Možnosti:
NUMA
SMP - je to jedno, kde to bude běžet

Green Thread

Abysme zvýšili čas, který program pracu 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
T
K
K
K
user
thread
kernel thread

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));
    }
}
```

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();
}
```

Přenos dat pomocí kanálů

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

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

Mutex

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

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

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

    println!("m = {:?}", m);
}
```

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();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

RwLock<T>

Umožňuje n čtenářů a jednoho zapisujícího. Mutex to neřeší.

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

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);
}
```

Bariéra

Zasynchronizuje vlákna tak, aby všehna začala zároveň

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

let mut handles = Vec::with_capacity(10);
let barrier = Arc::new(Barrier::new(10));
for _ in 0..10 {
    let c = Arc::clone(&barrier);
    // The same messages will be printed together.
    // You will NOT see any interleaving.
    handles.push(thread::spawn(move|| {
        println!("before wait");
        c.wait();
        println!("after wait");
    }));
}
// Wait for other threads to finish.
for handle in handles {
    handle.join().unwrap();
}
```

Úvod do asynchronního programování

Asynchronní programování

Koncept pro psaní konkurentních programů využívající malého množství OS vláken například pro I/O operace.

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

Cargo.toml

```
[dependencies]
futures = "0.3"
```

async, .await

```
use futures::executor::block_on;

async fn hello_world() {
    println!("hello, world!");
}

fn main() {
    let future = hello_world(); // Nothing is printed
    block_on(future); // `future` is run and "hello, world!" is printed
}
```

async, .await

```rust
// `foo()` returns a type that implements `Future<Output = u8>`.
// `foo().await` will result in a value of type `u8`.
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    // This `async` block results in a type that implements
    // `Future<Output = u8>`.
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

Dotazy?

Děkuji za pozornost