PV281: Programování v Rustu

Obsah

Tokio

Připomenutí asynchronního programování

Při běžné blokující I/O operaci systém preemptivně sebere vláknu čas, protože čeká na dokončení operace.

V asynchronním programování jsou operace, které není možné ihned dokončit, přesunuty do pozadí a mezitím se vykonává jiný kód.

Takovým blokům kódu se říká ***tasky***. Ty jsou běžně reprezentovány pomocí *green threadů*.

Tokio

Nejpoužívanější *asynchronní runtime* pro Rust.

Umožnuje asynchronní operace nad I/O a zjednodušuje síťové programování. (TCP, UDP, Unix sockety, dále timers, synchronizaci, různé plánovače, aj.)

Pro připomenutí: *std* poskytuje pouze rozhraní pro asynchronní programování, ale neposkytuje implementované funkce. Proto je potřeba zvolit některou z komunitních implementací.

Výhodou Tokia je výkon, spolehlivost, odzkoušenost a flexibilita.

K čemu nepoužívat Tokio

Paralelní výpočty

Tokio je určené pro scénáře, kdy jednotlivé úlohy čekají na I/O. Pokud potřebujete paralelizovat výpočty, můžete využít `rayon` nebo sami pracovat s thready. Rayon a Tokio můžete mixovat dohromady.

K čemu nepoužívat Tokio

Single requests

Pokud potřebujete poslat jeden požadavek a nemusíte jich paralelizovat několik současně, je otázka, jestli se vyplatí práce navíc s využítím Tokia a není lepší použít blokující volání. Nebude mezi nimi výkonostně rozdíl.

Tokio závislost

```
[package]
name = "my-crate"
version = "0.1.0"

[dependencies]
tokio = { version = "1", features = ["full"] }
```

Použití Tokia

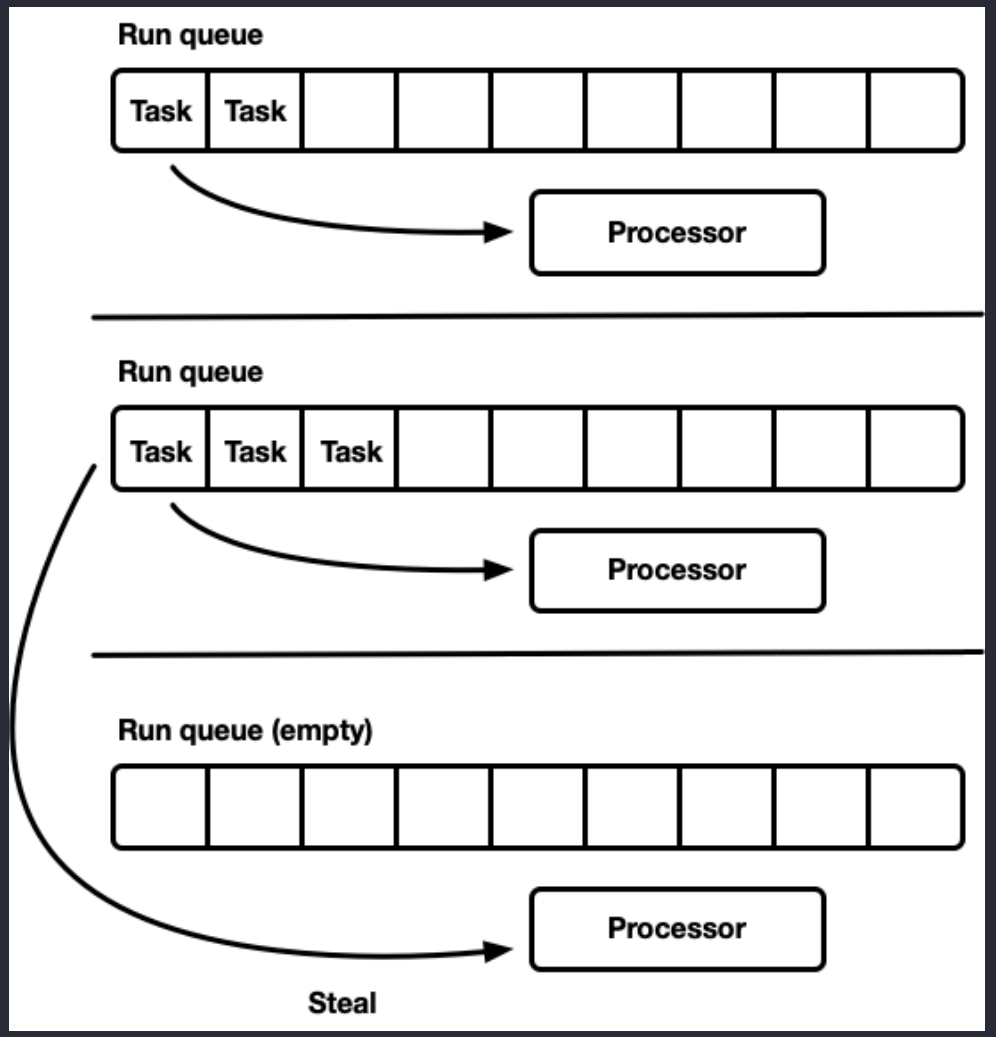
```rust
async fn say_world() {
    println!("world");
}

#[tokio::main]
async fn main() {
    // `say_world()` se ihned nespouští, async funkce jsou "lazy".
    let op = say_world();

    println!("hello");

    // `.await` na `op` provede `say_world()` a počká na výsledek.
    op.await;
}
```

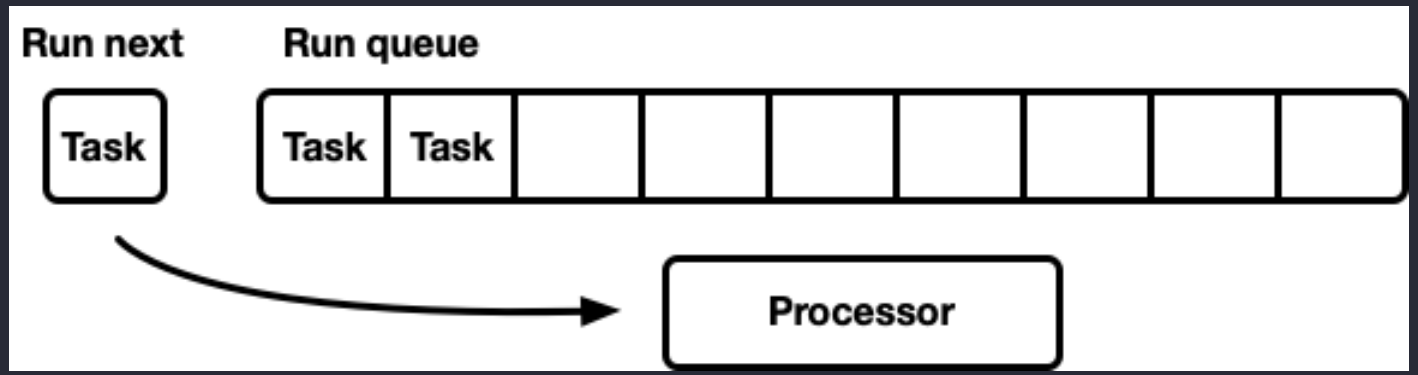
Makro `#[tokio::main]`

```rust
#[tokio::main]
async fn main() {
    println!("hello");
}
```

se makrem překonvertuje na

```rust
fn main() {
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .unwrap()
        .block_on(async {
            println!("Hello world");
        })
}
```


Run queue
Task
Task
Processor
Run queue
Task
Task
Task
Processor
Run queue (empty)
Processor
Steal


Run next
Run queue
Task
Task
Task
Processor

Práce se soubory

Práce se soubory je paralelní, ale na úrovni OS není skutečně asynchronní.

Tokio spustí souborové operace jako blokující v samostatných vláknech.

Poznámka: neblokující operace najdete v crate `tokio_uring`.

Čtení ze souboru

```rust
use tokio::io::{self, AsyncReadExt};
use tokio::fs::File; // <- Note that we aren't using File from std.

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt").await?;
    let mut buffer = Vec::new();

    // It's usually a bad idea to read the whole file at once, but this is just an example.
    f.read_to_end(&mut buffer).await?;

    Ok(())
}
```

Zápis do souboru

```rust
use tokio::io::{self, AsyncWriteExt};
use tokio::fs::File;

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut buffer = File::create("foo.txt").await?;

    buffer.write_all(b"some bytes").await?;

    Ok(())
}
```

Použití TCP socketu

```rust
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut listener = TcpListener::bind("127.0.0.1:6142").await.unwrap();

    loop {
        let (mut socket, _) = listener.accept().await?;

        tokio::spawn(async move {
            let mut buf = vec![0; 1024];

            loop {
                match socket.read(&mut buf).await {
                    Ok(0) => return, // Return value of `Ok(0)` signifies that the remote has closed
                    Ok(n) => {
                        if socket.write_all(&buf[..n]).await.is_err() {
                            return;
                        }
                    }
                    Err(_) => return,
}}});}} // <- This has to be done so the code fits on the slide.
```

crate `tokio-uring`

Využívá rozhraní io-uring v Linuxu
(nemá zatím podporu pro Windows)

Pozor: ne všechny Linux kernely jsou podporovány
(viz Requirements)

Rozhraní io-uring

Rozhraní asynchronních I/O operací, které **minimalizuje počet systémových volání**.

Rozhraní využívá dva ring buffery.
Jeden buffer slouží k předávání příkazů.
Druhý buffer k oznámení výsledku.

Buffery jsou sdílené mezi kernelem a user spacem.

Ukázka použití tokio-uring

```rust
use tokio_uring::fs::File;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    tokio_uring::start(async {
        let file = File::open("hello.txt").await?;

        let buf = vec![0; 4096];

        let (res, buf) = file.read_at(buf, 0).await;
        let n = res?;

        println!("{:?}", &buf[..n]);

        Ok(())
    })
}
```

Asynchronní mutex

```
use tokio::sync::Mutex; // <- Note that we aren't using Mutex from std.

async fn increment_and_do_stuff(mutex: &Mutex<i32>) {
    let mut lock = mutex.lock().await;
    *lock += 1;

    do_something_async().await;
} // <- Mutex lock goes out of scope here.
```

Problémy asynchronní synchronizace

Synchronizace je drahá. Proto se snažíme kód psát tak, aby byl nezávislý a nebylo třeba ho synchronizovat.

Pokud potřebujeme rychlejší implementace mutexu (nebo například podporu Windows XP), tak existuje crate parking_lot.

Message passing channels

`mpsc`: multi-producer, single-consumer

`oneshot`: single-producer, single consumer

`broadcast`: multi-producer, multi-consumer

`watch`: single-producer, multi-consumer

Ukázkové implementace

Použití MPSC

```rust
use tokio::sync::mpsc;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(32); // Create a new channel with a capacity of at most 32.

    let tx2 = tx.clone();

    tokio::spawn(async move {
        tx.send("sending from first handle").await;
    });

    tokio::spawn(async move {
        tx2.send("sending from second handle").await;
    });

    while let Some(message) = rx.recv().await {
        println!("GOT = {}", message);
    }
}
```

Použití broadcast

```rust
use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    let (tx, mut rx1) = broadcast::channel(16);
    let mut rx2 = tx.subscribe();

    tokio::spawn(async move {
        assert_eq!(rx1.recv().await.unwrap(), 10);
        assert_eq!(rx1.recv().await.unwrap(), 20);
    });

    tokio::spawn(async move {
        assert_eq!(rx2.recv().await.unwrap(), 10);
        assert_eq!(rx2.recv().await.unwrap(), 20);
    });

    tx.send(10).unwrap();
    tx.send(20).unwrap();
}
```

Lag na úrovni receiveru

Pokud dostaneme `RecvError::Lagged`, došlo ke ztrátě dat.

```rust
use tokio::sync::broadcast;

#[tokio::main]
async fn main() {
    let (tx, mut rx) = broadcast::channel(2);

    tx.send(10).unwrap();
    tx.send(20).unwrap();
    tx.send(30).unwrap();

    // The receiver lagged behind
    assert!(rx.recv().await.is_err());

    // At this point, we can abort or continue with lost messages

    assert_eq!(20, rx.recv().await.unwrap());
    assert_eq!(30, rx.recv().await.unwrap());
}
```

Stream

Asynchronní varianta k iterátorům. Bohužel zatím nejdou použít ve `for cyklu` a musíme použít `while let`.

Místo metody `into_iter()` používáme její asynchronní obdobu `into_stream()`.

Po použití potřebujeme crate `tokio-stream`. Trait pro streamy zatím není standardizovaný v std.

Zpracování streamu

```
use tokio_stream::StreamExt;

#[tokio::main]
async fn main() {
    let mut stream = tokio_stream::iter(&[1, 2, 3]);

    while let Some(v) = stream.next().await {
        println!("GOT = {:?}", v);
    }
}
```

Příklad: komunikace s Redis serverem

```rust
use mini_redis::{client, Result};
use tokio_stream::StreamExt;

async fn publish() -> Result<()> {
    let mut client = client::connect("127.0.0.1:6379").await?;

    client.publish("numbers", "1".into()).await?;
    client.publish("numbers", "two".into()).await?;
    client.publish("numbers", "3".into()).await?;

    Ok(())
}

async fn subscribe() -> Result<()> {
    let client = client::connect("127.0.0.1:6379").await?;
    let subscriber = client.subscribe(vec!["numbers".to_string()]).await?;
    let messages = subscriber.into_stream();

    pin!(messages);
    // ^ A value is pinned when it can no longer be moved in memory.
    // Callers of the value can be confident the pointer stays valid.

    while let Some(msg) = messages.next().await {
        println!("got = {:?}", msg);
    }

    Ok(())
}
```

```rust
#[tokio::main]
async fn main() -> Result<()> {
    tokio::spawn(async { publish().await });

    subscribe().await?;

    println!("DONE");

    Ok(())
}
```

```toml
# ...

[dependencies]
tokio = { version = "1", features = ["full"] }
tokio-stream = "0.1"
mini-redis = "0.4"
```

Serializace

Serializace

Převedení struktury nebo jiné reprezentace na textovou/binární/jinou formu.
V některých jazycích se jí říká *marshaling*.

Je to naprosto běžná úloha, kterou dneska potřebujeme ve všech programech. Například jde o převod dat do **JSON**u, který používáme ke komunikaci.

Obrácený proces je *deserializace*. Příkladem může být načtení **JSON** konfigurace ze souboru do struktury.

Serde

V tuto chvíli nejpoužívánější knihovna pro (de)serializaci.
Nemusí být vždy nejrychlejší, ale je odzkoušená a dobře dokumentovaná.

Hlavní část dostupná v crate `serde`.
Jednotlivé formáty dostupné v samostatných crates, např. `serde_json`, `serde_yaml`, ...

Primárně se spoléhá na *atributová makra*.

Serde závislosti

```
[package]
name = "my-crate"
version = "0.1.0"

[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

Serializace a deserializace

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    let serialized = serde_json::to_string(&point).unwrap();
    println!("serialized = '{}'", serialized);

    let deserialized: Point = serde_json::from_str(&serialized).unwrap();
    println!("deserialized = '{:?}'", deserialized);
}
```

```
serialized = '{"x":1,"y":2}'
deserialized = 'Point { x: 1, y: 2 }'
```

Různé konvence pojmenování

```rust
#[derive(Serialize)]
#[serde(rename_all = "camelCase")]
struct Person {
    first_name: String,
    last_name: String,
}
```

```
{
  "firstName": "Joe",
  "lastName": "Doe"
}
```

Reprezentace enum jako čísla

Použijeme crate `serde_repr`:

```rust
use serde_repr::*;

#[derive(Serialize_repr, Deserialize_repr, PartialEq, Debug)]
#[repr(u8)]
enum SmallPrime {
    Two = 2,
    Three = 3,
    Five = 5,
    Seven = 7,
}

fn main() -> serde_json::Result<()> {
    let j = serde_json::to_string(&SmallPrime::Seven)?;
    assert_eq!(j, "7");

    let p: SmallPrime = serde_json::from_str("2")?;
    assert_eq!(p, SmallPrime::Two);

    Ok(())
}
```

Výchozí hodnoty při deserializaci

```rust
#[derive(Deserialize, Debug)]
struct Request {
    // Use the result of a function as the default if "resource" is not included in the input.
    #[serde(default = "default_resource")]
    resource: String,

    // Use the type's implementation of `std::default::Default` if "timeout" is not included in the input.
    #[serde(default)]
    timeout: Timeout,

    // Use a method from the type as the default if "priority" is not included in the input.
    // It may also be a trait method.
    #[serde(default = "Priority::lowest")]
    priority: Priority,
}
```

Structure flattening

```rust
#[derive(Serialize, Deserialize)]
struct Pagination {
    limit: u64,
    offset: u64,
    total: u64,
}

#[derive(Serialize, Deserialize)]
struct Users {
    users: Vec<User>,

    #[serde(flatten)] // <- Flatten the contents of this field into the struct it is defined in.
    pagination: Pagination,
}
```

Přeskočení položky při serializaci

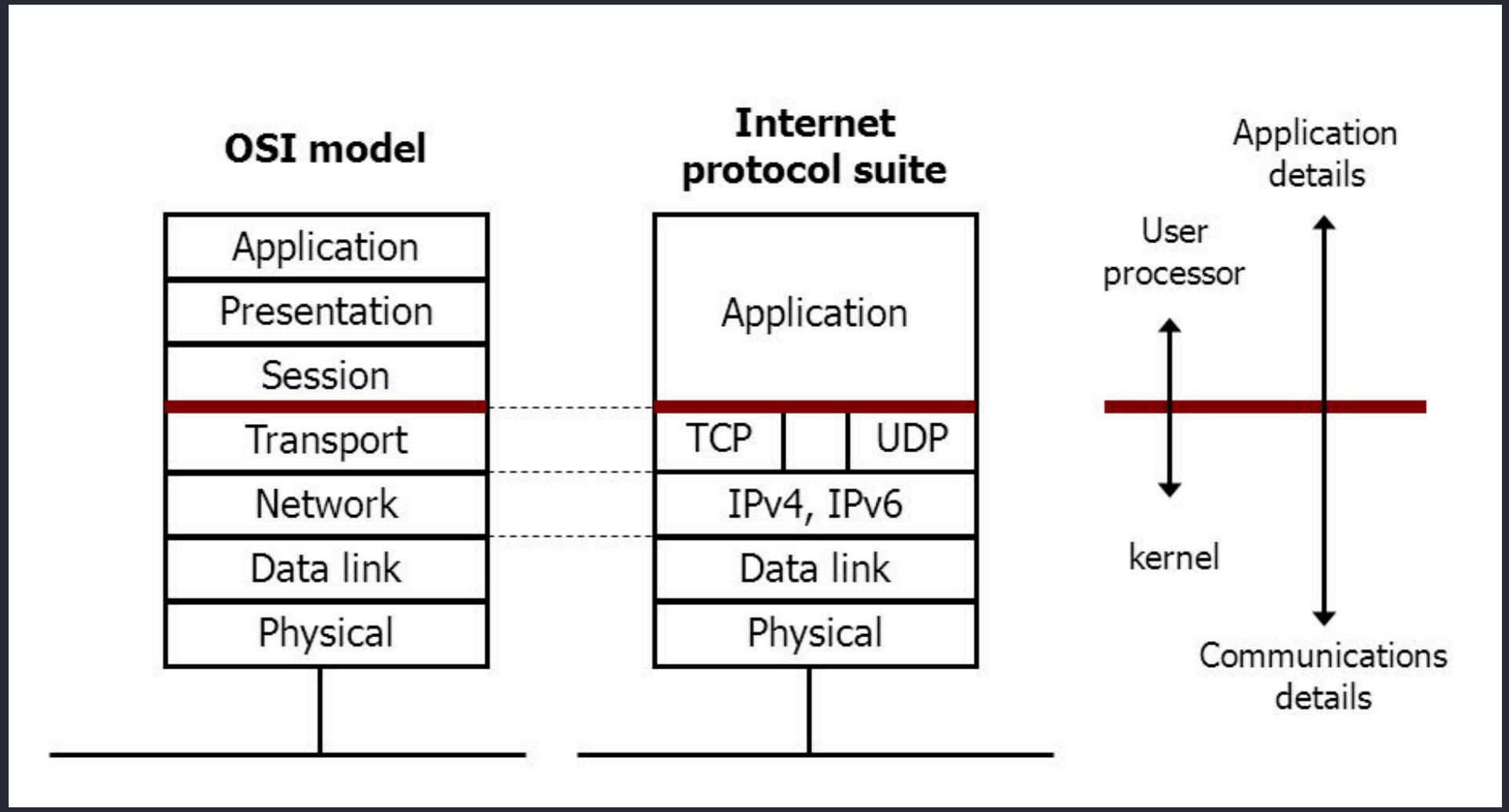
```rust
use serde::Serialize;
use std::collections::BTreeMap as Map;

#[derive(Serialize)]
struct Resource {
    // Always serialized.
    name: String,

    // Never serialized. Note that it will try to deserialize, use `skip_deserializing` or `default` then.
    #[serde(skip_serializing)]
    hash: String,

    // Use a method to decide whether the field should be serialized.
    #[serde(skip_serializing_if = "Map::is_empty")]
    metadata: Map<String, String>,
}
```

Vlastní serializace/deserializace

```
pub trait Serialize {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer;
}

pub trait Deserialize<'de>: Sized {
    fn deserialize<D>(deserializer: D) -> Result<Self, D::Error>
    where
        D: Deserializer<'de>;
}
```

Síťové programování

OSI a TCP/IP


OSI model
Application
Presentation
Session
Transport
Network
Data link
Physical
Internet protocol suite
Application
TCP
UDP
IPv4, IPv6
Data link
Physical
Application details
User processor
kernel
Communications details

Adresování

Počítač

Počítač adresujeme pomocí *IP adresy*, která nám pro socketovou komunikaci stačí.

Pro veřejné služby chceme místo konkrétní IP použít *doménové jméno*, pro což potřebujeme `A záznam` na `DNS`.

Pro synonyma využijeme `CNAME`. Často je potřeba vytvořit i `PTR záznam`, např. kvůli mailu.

Aplikace

Aplikace má přidělené číslo portu.

Důležité IP adresy

Local loopback: `127.0.0.1`

Privátní adresy

A: `10.0.0.0` — `10.255.255.255`
B: `172.16.0.0` — `172.31.255.255`
C: `192.168.0.0` — `192.168.255.255`

Fallback adresy

`169.254.0.0` — `169.254.255.255`

Porty

`0` — `1023`: *well-known* porty, bindnout je může pouze root

`1024` — `49151`: *registrované porty*, některé z nich jsou vázané na konkrétní služby

`49152` — `65535`: *dynamické* nebo také *privátní porty*

Síťové programování v Tokiu

UDP komunikace

Server i klient pracují s `UdpSocket`.
Získání socketu a provázání s portem: `bind`.

Komunikace *one to many* (strana serveru): `recv_from`, `send_to`.

Komunikace *one to one* (strana klienta): `recv`, `send`.

UDP server

```rust
use std::io;
use tokio::net::UdpSocket;

#[tokio::main]
async fn main() -> io::Result<()> {
    let sock = UdpSocket::bind("0.0.0.0:8080").await?;
    let mut buf = [0; 1024];

    loop {
        let (len, addr) = sock.recv_from(&mut buf).await?;
        println!("{:?} bytes received from {:?}", len, addr);

        let len = sock.send_to(&buf[..len], addr).await?;
        println!("{:?} bytes sent", len);
    }
}
```

UDP klient

```rust
use tokio::net::UdpSocket;
use std::io;

#[tokio::main]
async fn main() -> io::Result<()> {
    let sock = UdpSocket::bind("0.0.0.0:0").await?;

    let remote_addr = "127.0.0.1:8080";
    sock.connect(remote_addr).await?;
    let mut buf = [0; 1024];

    loop {
        let len = sock.recv(&mut buf).await?;
        println!("{:?} bytes received from {:?}", len, remote_addr);

        let len = sock.send(&buf[..len]).await?;
        println!("{:?} bytes sent", len);
    }
}
```

TCP komunikace

Strana serveru: `TcpListener`.
Pomocí `bind` naváže číslo portu.
Pomocí `accept` přijme připojení.

Strana klienta: `TcpSocket`.

Komunikace pomocí `TcpStream`.

TcpListener – server

```rust
use std::io;
use tokio::net::TcpListener;

async fn process_socket<T>(socket: T) {
    // Do work with socket here.
}

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (socket, _) = listener.accept().await?;
        process_socket(socket).await;
    }
}
```

TCPSocket – klient

```rust
use std::io;
use tokio::net::TcpSocket;

#[tokio::main]
async fn main() -> io::Result<()> {
    let addr = "127.0.0.1:8080".parse().unwrap();

    let socket = TcpSocket::new_v4()?;
    let stream = socket.connect(addr).await?;

    Ok(())
}
```

TcpStream

```rust
use tokio::net::TcpStream;
use tokio::io::AsyncWriteExt;
use std::error::Error;

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    // Connect to a peer
    let mut stream = TcpStream::connect("127.0.0.1:8080").await?;

    // Write some data.
    stream.write_all(b"hello world!").await?;

    Ok(())
}
```

Read a write

```rust
use std::io::prelude::*;
use std::net::TcpStream;

fn main() -> std::io::Result<()> {
    let mut stream = TcpStream::connect("127.0.0.1:34254")?;

    stream.write(&[1])?;
    stream.read(&mut [0; 128])?;

    Ok(())
} // The stream is closed here as it's dropped.
```

Čtení ze streamu

```rust
use tokio::{io::Interest, net::TcpStream};
use std::{error::Error, io};

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;

    loop {
        let ready = stream.ready(Interest::READABLE | Interest::WRITABLE).await?;

        if ready.is_readable() {
            let mut data = vec![0; 1024];

            // Try to read data, this may still fail with `WouldBlock` if the readiness event is a false positive.
            match stream.try_read(&mut data) {
                Ok(n) => println!("read {} bytes", n),
                Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => { // <- Note the match guard and reference binding.
                    continue;
                }
                Err(e) => return Err(e.into()),

}}}} // <- This has to be done so the code fits on the slide.
```

Psaní do streamu

```rust
use tokio::{io::Interest, net::TcpStream};
use std::{error::Error, io};

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;

    loop {
        let ready = stream.ready(Interest::READABLE | Interest::WRITABLE).await?;

        if ready.is_writable() {
            // Try to write data, this may still fail with `WouldBlock` if the readiness event is a false positive.
            match stream.try_write(b"hello world") {
                Ok(n) => println!("write {} bytes", n),
                Err(ref e) if e.kind() == io::ErrorKind::WouldBlock => {
                    continue
                }
                Err(e) => return Err(e.into()),

}}}} // <- This has to be done so the code fits on the slide.
```

Knihovny pro práci s chybami

crate `anyhow`

Poskytuje jednoduchou práci s chybami.
Funguje se všemi chybami implementující trait `std:error:Error`.

```rust
use anyhow::Result;

fn get_cluster_info() -> Result<ClusterMap> { // <- Všiměte si jednoho parametru místo dvou.
    let config = std::fs::read_to_string("cluster.json")?;
    let map: ClusterMap = serde_json::from_str(&config)?;
    Ok(map)
}
```

Pro odlišení od `std::result::Result` můžeme použít bez importu:

```rust
fn get_cluster_info() -> anyhow::Result<ClusterMap> { /* ... */ }
```

crate `thiserror`

Poskytuje atributové makro pro vytváření vlastních chyb.
Vygeneruje kód za překladu, díky tomu kód zůstává přehledný.

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum FormatError {
    #[error("Invalid header (expected {expected:?}, got {found:?})")]
    InvalidHeader {
        expected: String,
        found: String,
    },
    #[error("Missing attribute: {0}")]
    MissingAttribute(String),
}
```

Dotazy?

Děkuji za pozornost